Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования «Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ *Робототехника и комплексная автоматизация*

КАФЕДРА *Системы автоматизированного проектирования (РК-6)*

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

НА ТЕМУ

«Разработка графического программного обеспечения для визуализации трехмерных объектов»

Студент РК6-83Б — Д.А. Губанов
(Группа) (подпись, дата) (инициалы и фамилия)

Руководитель ВКР — Ф.А. Витюков
(подпись, дата) (инициалы и фамилия)

Нормоконтролер — С.В. Грошев
(подпись, дата) (инициалы и фамилия)

Москва, 2025 г.

Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)» (МГТУ им. Н.Э. Баумана)

УТВЕРЖДАЮ
Заведующий кафедрой *РК-6*
(индекс)

_____ А.П. Карпенко
(инициалы и фамилия)

«12» февраля *2025* г.

З А Д А Н И Е
на выполнение выпускной квалификационной работы

Студент группы РК6-83Б

Губанов Даниил Александрович
(фамилия, имя, отчество)

Тема выпускной квалификационной работы

Разработка графического программного обеспечения для визуализации трехмерных объектов

При выполнении ВКР:

	Используется / Не используется	Да/Нет
1)	Литературные источники и документы, имеющие гриф секретности	Нет
2)	Литературные источники и документы, имеющие пометку «Для служебного пользования», иных пометок, запрещающих открытое опубликование	Нет
3)	Служебные материалы других организаций	Нет
4)	Результаты НИР (ОКР), выполняемой в МГТУ им. Н.Э. Баумана	Нет
5)	Материалы по незавершенным исследованиям или материалы по завершенным исследованиям, но ещё не опубликованные в открытой печати	Нет

Тема выпускной квалификационной работы утверждена распоряжением по факультету:	
Название факультета:	«Робототехника и комплексная автоматизация»
Дата и рег. номер распоряжения:	№ _____ от « » _____ *2025* г.

Часть 1. *КОНСТРУКТОРСКАЯ*

Необходимо проанализировать способы, методы и подходы для отображения трехмерных объектов. Изучить современные технологии, выявить ограничения, обосновать целесообразность реализации собственных математических и технологических решений. Следует рассмотреть существующие инструменты, проанализировать способы интеграции различных библиотек и фреймворков.

Часть 2. *ТЕХНОЛОГИЧЕСКАЯ*

Разработать архитектуру графического программного обеспечения для визуализации трехмерных объектов. Реализовать качественный и практичный пользовательский графический и программный интерфейс.

Оформление выпускной квалификационной работы:

Расчетно-пояснительная записка на 73 листах формата А4.
Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.):

1. Прототипирование
2. Вершинные данные и шейдеры
3. Текстурирование
4. Аффинные преобразования
5. Виды проекций
6. Тривиальное освещение
7. Освещение по Фонгу
8. Графический и программный интерфейс

Дата выдачи задания «10» февраля 2025 г.

В соответствии с учебным планов выпускную квалификационную работу выполнить в полном объеме в срок до 2 июня 2025 года.

Студент Д.А. Губанов
(Подпись, дата) (И.О.Фамилия)

Руководитель выпускной квалификационной работы Ф.А. Витюков
(Подпись, дата) (И.О.Фамилия)

Примечание: Задание оформляется в двух экземплярах: один выдается студенту, второй хранится на кафедре.

Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)» (МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ **РК**
КАФЕДРА **РК-6**
ГРУППА **РК6-83Б**

УТВЕРЖДАЮ
Заведующий кафедрой *РК-6*
(индекс)
_____ А.П. Карпенко
(инициалы и фамилия)

«12» февраля *2025* г.

КАЛЕНДАРНЫЙ ПЛАН
выполнения выпускной квалификационной работы

студента: *Губанов Даниил Александрович*
(Фамилия, имя, отчество)

Тема выпускной квалификационной работы: «Разработка графического программного обеспечения для визуализации трехмерных объектов»

№ п/п	Наименование этапов выпускной квалификационной работы	Сроки выполнения этапов		Отметка о выполнении	
		план	факт	Должность	ФИО, подпись
1.	Задание на выполнение работы. Формулирование проблемы, цели и	*11.02.2025*	___.___.*20*___	Руководитель ВКР	*Ф.А. Витюков*
2.	Часть 1: *КОНСТРУКТОРСКАЯ*	*26.02.2025*	___.___.*20*___	Руководитель ВКР	*Ф.А. Витюков*
3.	Утверждение окончательных формулировок решаемой проблемы, цели работы и перечня задач	*06.03.2025*	___.___.*20*___	Заведующий кафедрой	*А.П. Карпенко*
4.	Часть 2: *ТЕХНОЛОГИЧЕСКАЯ*	*28.03.2025*	___.___.*20*___	Руководитель ВКР	*Ф.А. Витюков*
5.	1-я редакция работы	*30.05.2025*	___.___.*20*___	Руководитель ВКР	*Ф.А. Витюков*
6.	Подготовка доклада и презентации	*06.06.2025*	___.___.*20*___	Руководитель ВКР	*Ф.А. Витюков*
7.	Отзыв руководителя	*10.06.2025*	___.___.*20*___	Руководитель ВКР	*Ф.А. Витюков*
8.	Нормоконтроль	*05.06.2025*	___.___.*20*___	Нормоконтролер	*С.В. Грошев*
9.	Внешняя рецензия	*09.06.2025*	___.___.*20*___	Секретарь ГЭК	*И.А. Кузьмина*
10.	Защита работы на ГЭК	*20.06.2025*	___.___.*20*___	Секретарь ГЭК	*И.А. Кузьмина*

Студент _____ *Д.А. Губанов* *Руководитель работы* _____ *Ф.А. Витюков*
(подпись, дата) (подпись, дата)

СПИСОК ОБОЗНАЧЕНИЙ И СОКРАЩЕНИЙ

В настоящей ВКР применяют следующие сокращения и обозначения:

Библиотека (Library) – Набор предварительно написанного кода для решения специфических задач, который можно повторно использовать в различных проектах.

Фреймворк (Framework) – Каркас для разработки программного обеспечения, предоставляющий структуру и базовую функциональность.

API (Application Programming Interface) - Интерфейс программирования приложений, набор методов для взаимодействия между различными программными компонентами.

OpenGL (Open Graphics Library) – Кроссплатформенный API для 2D и 3D графики.

GLFW (Graphics Library Framework) – библиотека для кроссплатформенного создания и открытия окон, создания OpenGL-контекста и управления вводом.

Кроссплатформенность (Cross-platform) – Способность программного обеспечения работать на нескольких аппаратных платформах и операционных системах.

Метапрограммирование (Metaprogramming) – Написание программ, которые генерируют или манипулируют другими программами.

Рефлексия (Reflection) – Возможность программы исследовать и модифицировать свою структуру и поведение во время выполнения.

Метаданные (Metadata) – Данные, описывающие другие данные (например, информацию о структуре классов).

Сериализация (Serialization) – Процесс преобразования объекта в последовательность байтов для хранения или передачи.

Движок (Engine) – Базовый программный компонент, обеспечивающий ключевую функциональность (например, отображение 3D-моделей).

Рендеринг (Rendering) – Процесс создания изображения по описанию сцены.

Сцена (Scene) – Виртуальное пространство, содержащее все объекты для рендеринга.

Генерация (Generation) – Процесс автоматического создания кода или контента.

Видеодрайвер (Graphics Driver) – Программное обеспечение, обеспечивающее взаимодействие между ОС и графическим оборудованием.

Пиксель (Pixel) – Наименьший элемент изображения на экране.

Шейдер (Shader) – Программа, выполняемая на GPU для обработки графических данных.

Текстура (Texture) – Растровое изображение, накладываемое на поверхность 3D-модели.

Примитив (Primitive) – Базовая геометрическая форма (треугольник, линия, точка) в 3D-графике.

GPU (Graphics Processing Unit) – Графический процессор, специализированный для обработки графических данных.

CPU (Central Processing Unit) - Центральный процессор, основной вычислительный компонент компьютера.

Буфер (Buffer) – Область памяти для временного хранения данных.

Растеризатор (Rasterizer) – Компонент графического конвейера, преобразующий векторные данные в растровое изображение.

Фрагмент (Fragment) – Обрабатываемый элемент изображения в процессе растеризации (пиксель с дополнительной информацией).

Runtime – Время выполнения программы; также может означать среду выполнения.

РЕФЕРАТ

Работа посвящена созданию гибкого и производительного программного обеспечения для визуализации 3D-объектов, которое может быть применено в архитектуре, инженерии, медицине и других областях. Современные технологии визуализации играют ключевую роль в анализе и представлении сложных данных, поэтому разработка эффективного инструмента для работы с трехмерной графикой является актуальной задачей.

В работе применяются методы программирования на C++ и технологии OpenGL 4.6, обеспечивающие высокую производительность и кроссплатформенность. Основные этапы разработки включают проектирование архитектуры приложения, реализацию алгоритмов рендеринга, интеграцию вспомогательных библиотек и оптимизацию пользовательского интерфейса.

В результате создано графическое приложение, позволяющее эффективно визуализировать 3D-объекты с поддержкой современных технологий. Разработанный инструмент может служить основой для дальнейших исследований в области компьютерной графики и интерактивной визуализации данных.

Тип работы: выпускная квалификационная работа.

Тема работы: Разработка инструмента для визуализации трехмерных объектов на основе OpenGL.

Объектом исследования выступает процесс визуализации трехмерных объектов с использованием современных графических библиотек.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Современные технологии трехмерной визуализации становятся неотъемлемой частью научных исследований и промышленных разработок. От архитектурного проектирования до медицинской диагностики, от инженерного анализа до образовательных технологий – везде требуется эффективное программное обеспечение, способное наглядно представлять сложные объемные данные. Особую актуальность приобретает создание универсальных инструментов визуализации, сочетающих высокую производительность с гибкостью применения в различных предметных областях.

Анализ существующих решений показывает, что большинство коммерческих продуктов либо узкоспециализированы, либо требуют значительных аппаратных ресурсов. В то же время открытые библиотеки компьютерной графики, такие как OpenGL (Open Graphics Library), предоставляют широкие возможности для создания собственных решений, оптимизированных под конкретные задачи. Это создает предпосылки для разработки специализированного программного обеспечения, которое могло бы заполнить нишу между тяжеловесными коммерческими пакетами и ограниченными по функционалу открытыми решениями.

Целью данной работы является создание эффективного графического программного обеспечения для визуализации трехмерных объектов, основанного на современных технологиях компьютерной графики.

Для достижения поставленной цели необходимо решить следующие задачи:

1) Провести анализ существующих решений в области 3D-визуализации;
2) Исследовать возможности библиотеки OpenGL и особенности программирования графических приложений;
3) Разработать архитектуру программного комплекса;
4) Реализовать основные алгоритмы визуализации;
5) Реализовать систему рефлексии для работы с типами данных;
6) Разработать пользовательский интерфейс;

Научная новизна исследования заключается в разработке:

1) Гибкой архитектуры графического приложения с поддержкой динамической загрузки модулей;
2) Универсальной системы рефлексии для работы с пользовательскими типами данных;
3) Оптимизированных алгоритмов визуализации сложных сцен;

Практическая значимость работы определяется широкой областью применения разработанного программного обеспечения:

1) В образовательном процессе для изучения основ компьютерной графики;
2) В научных исследованиях, требующих визуализации многомерных данных;
3) В инженерных расчетах и проектировании;
4) Как базовой платформы для разработки специализированных графических приложений;

Реализация проекта выполнена на языке C++ с использованием библиотеки OpenGL 4.6, что обеспечивает высокую производительность и кроссплатформенность решения. Для работы с пользовательским интерфейсом применяются современные фреймворки, а система рефлексии реализована с использованием метапрограммирования.

1. КОНСТРУКТОРСКАЯ

1.1. Обзор существующих решений

В области разработки графического программного обеспечения для визуализации трехмерных объектов существует множество инструментов и библиотек, которые могут быть использованы для реализации подобных проектов. В данном списке представлен обзор наиболее популярных и широко используемых решений, которые могут служить основой для разработки собственного программного обеспечения на базе OpenGL и языка C++:

1) Unity является мощным игровым движком, который предоставляет разработчикам инструменты для создания 3D-игр и приложений. Он поддерживает множество языков программирования, включая C#, но также предоставляет возможность интеграции с C++ через плагины.
2) Unreal Engine является еще одним популярным игровым движком, разработанным Epic Games. Он предоставляет разработчикам мощные инструменты для создания игр и интерактивных приложений с высококачественной графикой.
3) CryEngine является игровым движком, разработанным Crytek, который известен своими возможностями в области рендеринга реалистичных визуальных эффектов. Он предоставляет разработчикам инструменты для создания высоко детализированных 3D-миров с поддержкой передовых графических технологий.
4) Godot является открытым игровым движком, который предоставляет разработчикам мощные инструменты для создания 2D и 3D игр. Он поддерживает множество языков программирования, включая GDScript, но также предоставляет возможность интеграции с C++.

Выбор подходящего инструмента для разработки графического программного обеспечения зависит от множества факторов, включая целевую платформу, требования к производительности и опыт разработчика.

1.2.Прототипирование

Для разработки графического программного обеспечения в качестве концептуального примера был взят игровой движок Unreal Engine 4, поскольку он предоставляет удобную и функциональную среду для работы с 3D-графикой. Анализ его архитектуры позволил выделить ключевые компоненты, необходимые для реализации аналогичного инструмента:

1) Кроссплатформенность – поддержка различных операционных систем (Windows, Linux). Для этого в проекте используется библиотека GLFW (Graphics Library Framework), обеспечивающая создание окон и обработку ввода;
2) Рендеринг 3D-графики – отображение объектов в реальном времени. В качестве графического API выбрана библиотека OpenGL, предоставляющая широкие возможности для работы с аппаратным ускорением;
3) Пользовательский интерфейс – удобное управление сценой и объектами. В качестве аналога рассматривается ImGUI (Immediate Mode Graphical User Interface), позволяющий создавать интерактивные элементы с минимальными накладными расходами;
4) Импорт 3D-моделей – загрузка сцен и объектов из различных форматов. Для этого применяется библиотека Assimp (Open Asset Import Library), поддерживающая большинство популярных форматов (FBX, OBJ, GLTF и др.);
5) Рефлексия и сериализация – динамическое управление свойствами объектов и сохранение их состояния. Реализация основана на Clang API, что позволяет анализировать пользовательский код и автоматически генерировать метаданные;

Интерфейс разрабатываемого приложения (рис. 1) вдохновлен интерфейсом Unreal Engine 4 (рис. 2) и включает следующие основные вкладки:

1) Viewport – область визуализации сцены и объектов.
2) Content Browser – панель для выбора и загрузки ресурсов.

3) World Outliner – иерархическое представление объектов на сцене.
4) Object Settings – редактирование параметров выбранного объекта.
5) Logs – отладочная информация и системные сообщения.

Такой подход обеспечивает удобство работы с 3D-графикой, приближая функциональность к профессиональным инструментам.

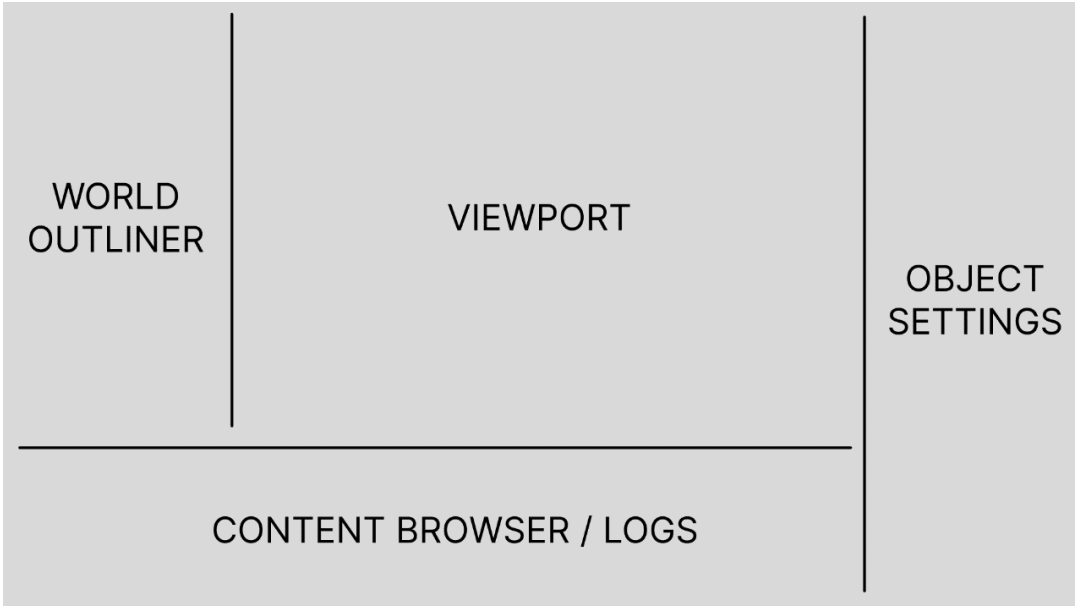

Рисунок 1 – Прототип интерфейса графического программного обеспечения

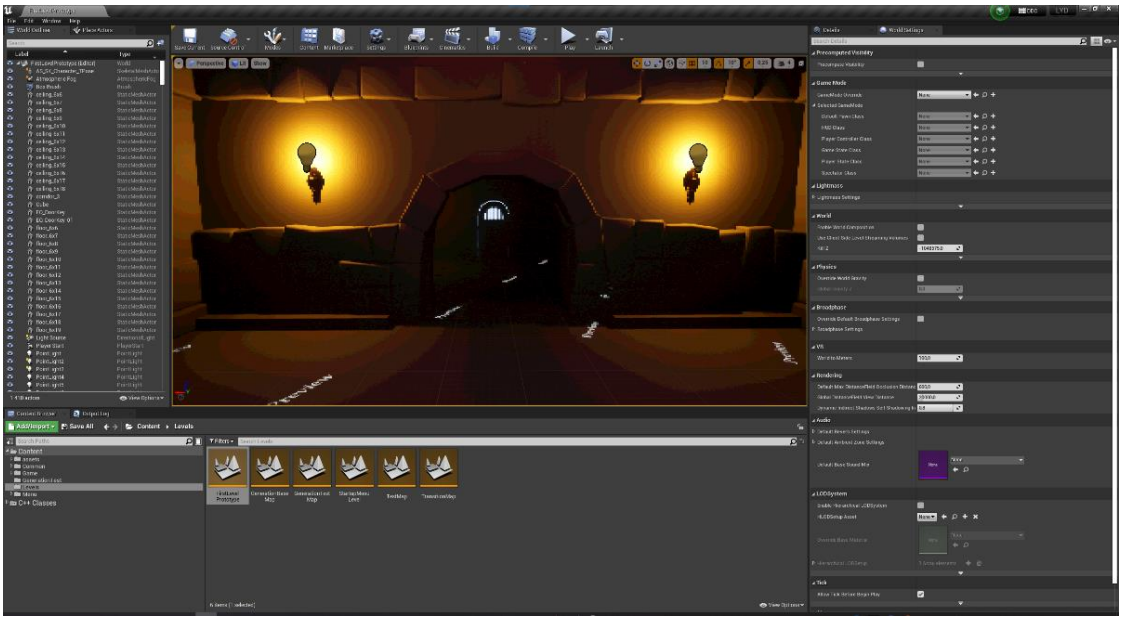

Рисунок 2 – Интерфейс Unreal Engine 4

1.3.Основы создания графического контекста

Разработка графических приложений требует тщательного подхода к созданию базовой инфраструктуры для отрисовки. OpenGL, будучи лишь спецификацией, не предоставляет механизмов для создания окон и обработки ввода, что вынуждает разработчиков использовать дополнительные библиотеки. Эта особенность обусловлена необходимостью поддержки различных операционных систем, каждая из которых имеет собственный API для работы с графикой [1].

Библиотека GLFW была выбрана в качестве решения для управления окнами и вводом благодаря своей кроссплатформенности и простоте использования. Она абстрагирует специфичные для ОС вызовы, предоставляя единый API для создания окон, обработки событий и управления контекстом OpenGL. Однако сама по себе GLFW не решает проблему динамической загрузки функций OpenGL, которая возникает из-за различий в реализации драйверов видеокарт.

Для решения этой проблемы используется библиотека GLEW, которая берет на себя задачу получения адресов функций OpenGL во время выполнения программы. Это критически важно, так как расположение этих функций неизвестно на этапе компиляции и может отличаться в зависимости от видеодрайвера и версии OpenGL.

1.4.Архитектура графического конвейера OpenGL

В OpenGL все объекты находятся в трёхмерном пространстве, однако экран и окно представляют собой двумерную матрицу пикселей. Следовательно, значительная часть задач OpenGL связана с преобразованием трёхмерных координат в двумерные для отображения на экране. Этот процесс преобразования управляется графическим конвейером OpenGL. [2]

Графический конвейер можно разделить на две основные части: первая часть отвечает за преобразование трёхмерных координат в двумерные, а вторая – за преобразование двумерных координат в цветные пиксели. В рамках данной

главы мы подробно рассмотрим графический конвейер и способы его использования для создания высококачественного графического контента.

Графический конвейер принимает набор трёхмерных координат и преобразует их в цветные двумерные пиксели на экране. Этот процесс можно разделить на несколько этапов, каждый из которых требует на вход результат работы предыдущего. Все этапы конвейера являются специализированными и могут выполняться параллельно, что позволяет современным графическим процессорам (GPU) эффективно обрабатывать данные.

Благодаря параллельной природе графического конвейера, большинство современных GPU оснащены тысячами маленьких процессоров, которые быстро обрабатывают данные, запуская множество небольших программ на каждом этапе конвейера. Эти программы называются шейдерами.

Некоторые из этих шейдеров могут быть настроены разработчиком, что позволяет создавать собственные шейдеры для замены стандартных. Это предоставляет широкие возможности для тонкой настройки различных этапов конвейера и, благодаря выполнению шейдеров на GPU, позволяет экономить процессорное время. Шейдеры пишутся на языке программирования GLSL (OpenGL Shading Language), которое используется в данной работе.

На рисунке 3 представлено схематическое представление этапов графического конвейера. Синие блоки обозначают этапы, для которых возможно создание пользовательских шейдеров.

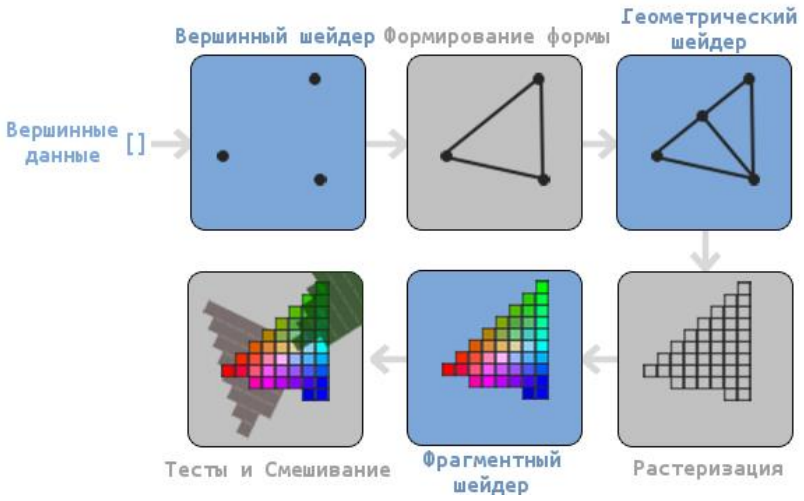

Рисунок 3 – Графический конвейер

Как можно видеть, графический конвейер включает в себя множество секций, каждая из которых отвечает за свою часть обработки вершинных данных до полного отрисовки пикселя. Кратко опишем каждую секцию конвейера в упрощённом виде, чтобы дать вам полное представление о его работе.

На вход конвейера подаётся массив трёхмерных координат, из которых формируются треугольники, называемые вершинными данными. Вершинные данные представляют собой набор вершин. Вершина – это набор данных, включающий трёхмерную координату. Эти данные представляются с использованием атрибутов вершины, которые могут содержать любую информацию, но для упрощения будем считать, что вершина состоит из трёхмерной позиции и значения цвета.

Поскольку OpenGL необходимо знать, какую фигуру составить из переданной коллекции координат и значений цвета, требуется указать, какую фигуру вы хотите сформировать из данных. Хотите ли вы отрисовать набор точек, набор треугольников или просто одну длинную линию. Такие фигуры называются примитивами и передаются OpenGL во время вызова команд отрисовки.

Первый этап конвейера – это вершинный шейдер, который принимает на вход одну вершину. Основная задача вершинного шейдера – это преобразование трёхмерных координат в другие трёхмерные координаты (об этом будет рассказано позже), и возможность изменения этого шейдера позволяет выполнять некоторые основные преобразования над значениями вершины.

Этап сборки примитивов принимает на вход все вершины из вершинного шейдера, которые формируют примитив, и собирает из них сам примитив; в нашем случае это будет треугольник.

Результат этапа сборки примитивов передаётся геометрическому шейдеру. Геометрический шейдер, в свою очередь, принимает на вход набор вершин, формирующих примитивы, и может генерировать другие фигуры путём создания новых вершин для формирования новых (или других) примитивов. Например, в нашем случае он может сгенерировать второй треугольник помимо

данной фигуры.

Результат работы геометрического шейдера передаётся на этап растеризации, где результирующие примитивы соотносятся с пикселями на экране, формируя фрагмент для фрагментного шейдера. Перед запуском фрагментного шейдера выполняется вырезка, которая отбрасывает все фрагменты, находящиеся вне поля зрения, тем самым повышая производительность.

Основная задача фрагментного шейдера заключается в вычислении конечного цвета пикселя, и именно на этом этапе чаще всего реализуются различные дополнительные эффекты OpenGL. Фрагментный шейдер обычно содержит всю необходимую информацию о трёхмерной сцене, которая используется для модификации финального цвета, включая освещение, тени, цвет источника света и другие параметры.

После определения всех соответствующих цветовых значений результат проходит этап альфа-тестирования и смешивания. Этот этап проверяет значение глубины и шаблона фрагмента и использует их для определения местоположения фрагмента относительно других объектов: находится ли он спереди или сзади. Также выполняется проверка значений прозрачности и смешивание цветов, если это необходимо. В результате, при отрисовке множественных примитивов результирующий цвет пикселя может отличаться от цвета, вычисленного фрагментным шейдером.

Графический конвейер является довольно сложным и включает множество конфигурируемых частей. Тем не менее, в основном работа происходит только с вершинными и фрагментными шейдерами. Геометрический шейдер не является обязательным и часто оставляется в стандартном виде.

В современном OpenGL необходимо задать как минимум вершинный шейдер (поскольку на видеокартах отсутствуют стандартные вершинные и фрагментные шейдеры).

Нормализованные координаты устройства NDC (Normalized Device Coordinates). После обработки вершинных координат в вершинном шейдере они

должны быть нормализованы в NDC. NDC представляет собой систему координат, где значения x, y и z находятся в диапазоне от -1.0 до 1.0. Координаты, выходящие за пределы этого диапазона, будут отброшены и не отобразятся на экране.

На рисунке 4 представлен заданный треугольник в контексте NDC. В отличие от экранных координат, в NDC положительное значение оси y направлено вверх, а координаты (0, 0) соответствуют центру области отображения, а не верхнему левому углу экрана.

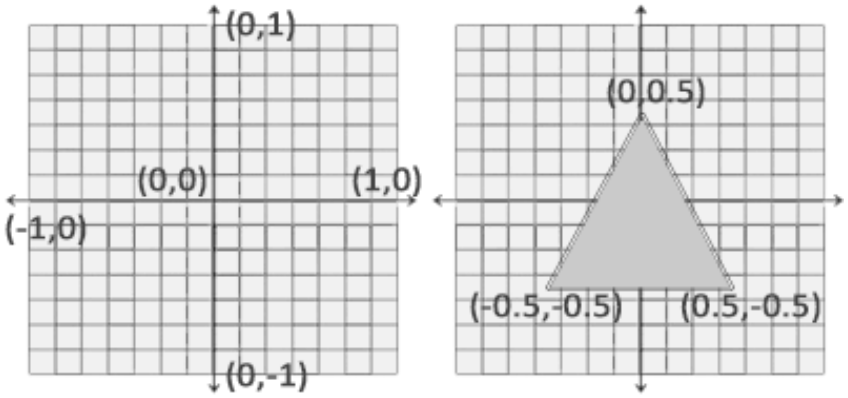

Рисунок 4 – Треугольник в контексте NDC

Полученные NDC координаты затем преобразуются в экранные координаты посредством Viewport с использованием данных, предоставленных вызовом функций OpenGL. Экранные координаты далее трансформируются во фрагменты и передаются фрагментным шейдерам для дальнейшей обработки.

1.5.Основы работы с шейдерами и вершинными данными

Шейдерная программа – это конечный объект, полученный в результате объединения нескольких шейдеров. Для использования собранных шейдеров их необходимо соединить в объект шейдерной программы, а затем активировать эту программу при отрисовке объектов. Эта программа будет использоваться при вызове команд отрисовки.

При соединении шейдеров в программу выходные значения одного шейдера сопоставляются с входными значениями другого. Ошибки могут возникнуть в процессе соединения шейдеров, если входные и выходные

значения не совпадают.

На данный момент мы передали GPU вершинные данные и указали, как их обрабатывать. Мы почти завершили процесс. OpenGL пока не знает, как представить вершинные данные в памяти и как связывать их с атрибутами вершинного шейдера.

Вершинный шейдер позволяет нам указать любые данные для каждого атрибута вершины, но это не означает, что мы должны явно указывать, какой элемент данных относится к какому атрибуту. Вместо этого мы должны сообщить OpenGL, как интерпретировать вершинные данные перед отрисовкой. Формат нашего вершинного буфера, следующий и представлен на рисунке 5:

1) Информация о позиции хранится в 32-битном (4 байта) значении с плавающей точкой.
2) Каждая позиция формируется из 3 значений.
3) Между наборами из 3 значений нет разделителей; такой буфер называется плотно упакованным.
4) Первое значение в переданных данных является началом буфера.

Рисунок 5 – Вершинный буфер

После того как мы сообщили OpenGL, как интерпретировать вершинные данные, необходимо включить атрибут, передав вершинному атрибуту позицию аргумента. После всех настроек мы инициализировали вершинные данные в буфере с использованием VBO (Vertex Buffer Object), установили вершинный и фрагментный шейдер и сообщили OpenGL, как связать вершинный шейдер и вершинные данные.

Этот процесс необходимо повторять при каждой отрисовке объекта. Хотя это может показаться не слишком сложным, представьте, что у вас более 5 вершинных атрибутов и около 100 различных объектов. Постоянная установка

этих конфигураций для каждого объекта становится трудоемкой задачей. Было бы удобно иметь способ хранения всех этих состояний, чтобы при отрисовке объектов требовалось лишь привязываться к нужному состоянию.

Объект вершинного массива VAO (Vertex Array Object) может быть привязан аналогично VBO, после чего все последующие вызовы вершинных атрибутов будут сохраняться в VAO. Основное преимущество этого метода заключается в том, что настройка атрибутов требуется лишь однократно, а в последующих случаях будет использоваться конфигурация, сохраненная в VAO, как представлено на рисунке 6. Это также упрощает процесс смены вершинных данных и конфигураций атрибутов путем простого привязывания различных VAO.

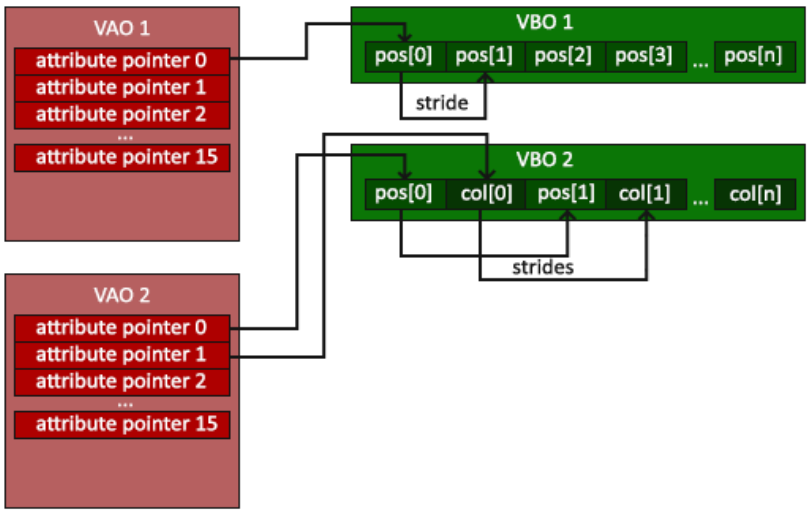

Рисунок 6 – Вершинный массив VAO

1.6.Основы работы с текстурами

Текстура представляет собой двумерное изображение (также существуют одномерные и трехмерные текстуры), используемое для добавления деталей объектам. По сути, текстура — это фрагмент бумаги с изображением, например, кирпича, который наклеивается на поверхность объекта, создавая иллюзию, что сам объект выполнен из этого материала.

Кроме того, текстуры могут содержать большие объемы данных,

передаваемых в шейдеры. На рисунке ниже показана текстура кирпичной стены, наложенная на треугольник.

Для привязки текстуры к треугольнику необходимо указать для каждой вершины треугольника, какая часть текстуры соответствует этой вершине. Каждая вершина должна иметь ассоциированные текстурные координаты, указывающие на соответствующую часть текстуры.

Текстурные координаты располагаются в диапазоне от 0 до 1 по осям x и y (в данном случае используются двумерные текстуры). Процесс получения цвета текстуры с использованием текстурных координат называется отбором (sampling). Начальной точкой текстурных координат является нижний левый угол текстуры (0, 0), а конечной – верхний правый угол (1, 1). На рисунке 7 демонстрируется наложение текстурных координат на треугольник.

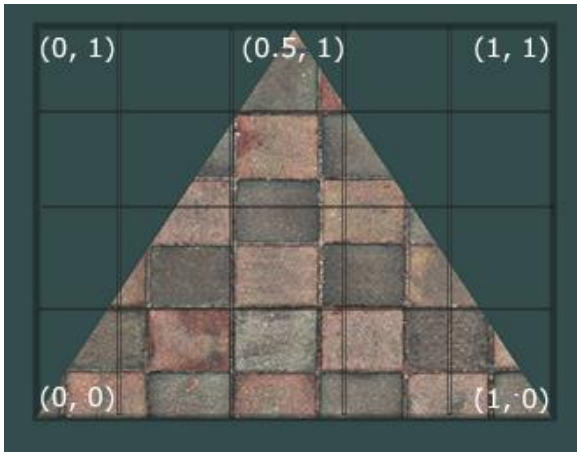

Рисунок 7 – Расположение текстурных координат на треугольнике

Обычно текстурные координаты находятся в диапазоне от (0,0) до (1,1). Однако, что происходит, если текстурные координаты выходят за пределы этого диапазона, по умолчанию OpenGL повторяет изображение (игнорируется целая часть числа с плавающей точкой), но существуют и другие опции:

1) Стандартное поведение для текстур, при котором текстура повторяется.
2) Аналогично стандартному поведению, но с отражением текстуры.
3) Привязка координат к диапазону от 0 до 1. Координаты, выходящие за пределы диапазона, привязываются к границе текстуры.

4) Координаты, выходящие за пределы диапазона, используют установленный пользователем цвет границы.

Каждая из этих опций демонстрирует различное поведение при использовании текстурных координат, выходящих за пределы диапазона. На рисунке 8 наглядно иллюстрирует различия.

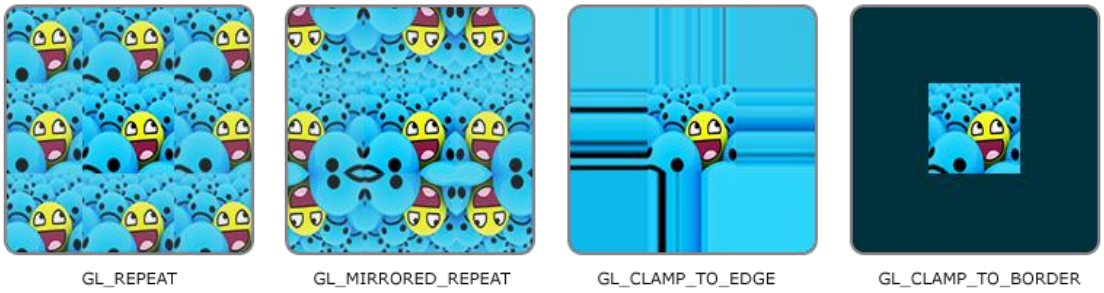

Рисунок 8 – Изображения квадрата при различных опциях текстуры

Возьмем большое помещение, в котором находятся тысячи объектов, каждый из которых имеет привязанную текстуру. Некоторые объекты располагаются ближе к наблюдателю, другие – дальше, и каждому объекту назначена текстура высокого разрешения. Когда объект находится на значительном расстоянии от наблюдателя, необходимо обработать лишь небольшое количество фрагментов. Однако OpenGL сталкивается с трудностями при определении корректного цвета для каждого фрагмента текстуры высокого разрешения, особенно когда приходится учитывать множество пикселей. Это приводит к возникновению артефактов на мелких объектах и избыточному использованию памяти, связанному с применением текстур высокого разрешения на незначительных объектах.

Для решения данной проблемы OpenGL применяет технологию, известную как мипмапы (mipmaps). Мипмапы представляют собой набор текстур, каждая последующая из которых вдвое меньше предыдущей. Основная идея мипмапов довольно проста: после достижения определенного расстояния от наблюдателя OpenGL переключается на использование другой мипмап текстуры, которая обеспечивает более качественное отображение на текущем расстоянии. Чем дальше объект находится от наблюдателя, тем меньшее

разрешение текстуры используется, поскольку пользователю сложнее заметить разницу между уровнями разрешения. Кроме того, мипмапы способствуют повышению производительности, что является дополнительным преимуществом, пример представлен на рисунке 9.

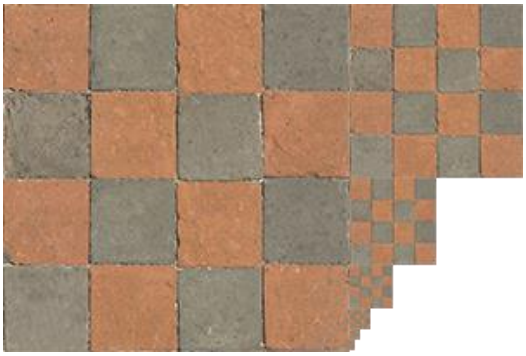

Рисунок 9 – Расположение текстурных координат на треугольнике

Создание набора мипмап текстур для каждого изображения довольно муторно, но OpenGL умеет генерировать их после создания текстуры, что упрощает работу с ними.

Нам надо сообщить OpenGL, как сэмплировать текстуру, поэтому мы обновим вершинные данные, добавив в них текстурные координаты. После добавления дополнительных атрибутов нам снова придется оповестить OpenGL о нашем новом формате, как представлено на рисунке 10.

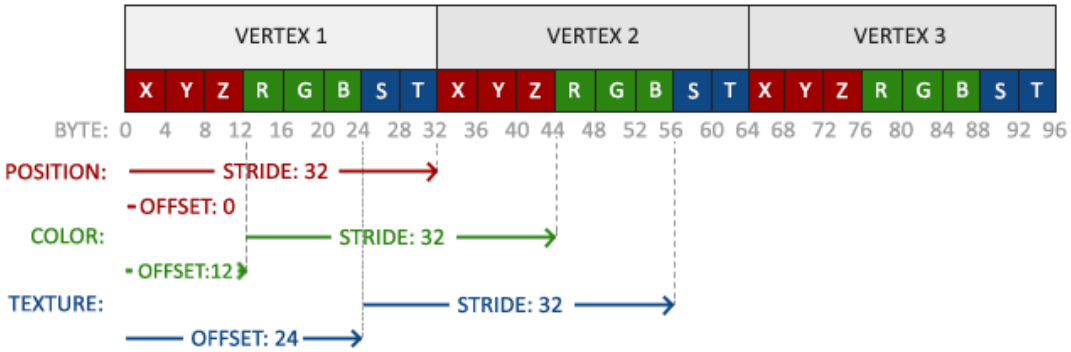

Рисунок 10 – Вершинный буфер с координатами в пространстве, цветом и координатами текстуры

1.7.Геометрические преобразования в трёхмерном пространстве

1.7.1. Аффинные преобразования

Аффинные преобразования составляют математический фундамент современной компьютерной графики, обеспечивая мощный и гибкий инструментарий для манипуляции объектами в трехмерном пространстве. Эти преобразования, сохраняя ключевые геометрические свойства, такие как параллельность прямых и отношение длин вдоль них, позволяют точно контролировать положение, ориентацию и масштаб любых объектов в виртуальной сцене. Их значение простирается от базовых операций вроде простого перемещения камеры до сложных процедур скелетной анимации персонажей и физического моделирования деформаций твердых тел. [3]

В системе однородных координат, которая является стандартом в компьютерной графике, аффинные преобразования находят свое выражение через матрицы размерности 4×4. Особенность такого представления заключается в фиксированной последней строке [0 0 0 1], что сохраняет важные свойства преобразований при их композиции. Перенос (трансляция) реализуется добавлением вектора смещения к координатам точки, что в матричной форме принимает вид единичной матрицы с дополнительным столбцом трансляционных компонент. Это простое, но мощное преобразование лежит в основе всех операций позиционирования объектов в сцене.

Основные виды аффинных преобразований:

1) Перенос (Трансляция). Преобразование, изменяющее положение объекта в пространстве.

$$\mathbf{T} = \begin{pmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} & \boldsymbol{t_x} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \boldsymbol{t_y} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \boldsymbol{t_z} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{pmatrix}$$

2) Масштабирование. Преобразование, изменяющее размеры объекта.

$$\mathbf{S} = \begin{pmatrix} \boldsymbol{s_x} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{s_y} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{s_z} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{pmatrix}$$

3) Вращение. Преобразование, изменяющее ориентацию объекта.

Основные матрицы вращения вокруг осей:

$$\boldsymbol{R_x(\theta)} = \begin{pmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{\cos\theta} & \mathbf{-\sin\theta} & \mathbf{0} \\ \mathbf{0} & \mathbf{\sin\theta} & \mathbf{\cos\theta} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{pmatrix}$$

$$\boldsymbol{R_y(\theta)} = \begin{pmatrix} \mathbf{\cos\theta} & \mathbf{0} & \mathbf{\sin\theta} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{-\sin\theta} & \mathbf{0} & \mathbf{\cos\theta} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{pmatrix}$$

$$\boldsymbol{R_z(\theta)} = \begin{pmatrix} \mathbf{\cos\theta} & \mathbf{-\sin\theta} & \mathbf{0} & \mathbf{0} \\ \mathbf{\sin\theta} & \mathbf{\cos\theta} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{pmatrix}$$

Аффинные преобразования обладают свойством композиции - последовательность преобразований может быть представлена единой матрицей, равной произведению матриц отдельных преобразований. Важно учитывать порядок умножения матриц, так как матричное умножение некоммутативно. Общая формула преобразования точки:

$$\boldsymbol{P' = M \times P = T \times R \times S \times P}$$

В сложных 3D-сценах объекты часто организуются в иерархические структуры, где каждый дочерний объект существует в локальной системе координат родительского объекта. Для корректного отображения и взаимодействия таких объектов необходимо:

1) Определять глобальные координаты дочерних объектов
2) Обрабатывать каскадные преобразования
3) Обеспечивать эффективное обновление при изменении иерархии.

Глобальная трансформация объекта вычисляется как произведение матриц преобразований всех родительских объектов:

$$\boldsymbol{M_{global} = M_{parentN} \times \ldots \times M_{parent1} \times M_{local}}$$

1.7.2. Кватернионы

Несмотря на широкое применение матричных преобразований в компьютерной графике, данный подход обладает рядом существенных ограничений, которые необходимо учитывать при разработке систем управления 3D-объектами. Рассмотрим ключевые проблемы, возникающие при использовании матричного представления вращений, наиболее известной проблемой матричного вращения является Gimbal Lock, возникающий при совпадении двух осей вращения.

Рассмотрим классический пример с последовательностью вращений ZYX:

1) При повороте вокруг оси Y на 90° (тангаж), оси X и Z совпадают в одной плоскости.
2) Последующие вращения вокруг этих осей становятся эквивалентными – система больше не может различить, вокруг какой именно оси происходит вращение.
3) Математически это проявляется как вырождение матрицы вращения, когда две компоненты матрицы становятся линейно зависимыми.

Традиционные методы пытались обойти проблему:

1) Ограничение углов (например, запрет поворотов на ± 90° по второй оси)
2) Переключение осей при приближении к опасному углу
3) Реортогонализация матриц для компенсации ошибок

Эти ограничения привели к поиску принципиально иного подхода — кватернионов.

Кватернионы – это система гиперкомплексных чисел, расширяющая понятие комплексных чисел в четырехмерное пространство. Они были открыты Уильямом Гамильтоном в 1843 году и нашли широкое применение в компьютерной графике, робототехнике, аэрокосмической навигации и физике благодаря своей эффективности в представлении вращений. Визуальное представление иллюстрируется на рисунке 11.

Кватернион записывается в виде:

$$\boldsymbol{q} = \boldsymbol{w} + \boldsymbol{x} \cdot \boldsymbol{i} + \boldsymbol{y} \cdot \boldsymbol{j} + \boldsymbol{z} \cdot \boldsymbol{k}$$

где:

1) $\boldsymbol{w}$ – скалярная (вещественная) часть,
2) $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ – векторная (мнимая) часть,
3) $\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}$ – мнимые единицы, удовлетворяющие следующим правилам умножения:

$$\mathbf{i^2 = j^2 = k^2 = ijk = -1}$$

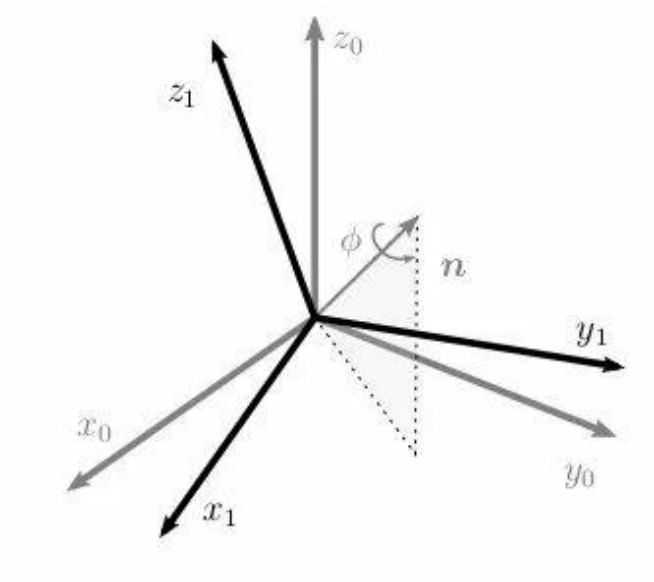

Рисунок 11 – Представление кватерниона в пространстве

Для представления вращений используются единичные кватернионы (норма $|| \boldsymbol{q} || = \mathbf{1}$). Вращение на угол $\boldsymbol{\theta}$ вокруг оси, заданной единичным вектором $\mathbf{n} = (\boldsymbol{n_x}, \boldsymbol{n_y}, \boldsymbol{n_z})$ описывается кватернионом:

$$\mathbf{q} = \boldsymbol{\cos}\left(\frac{\boldsymbol{\theta}}{\mathbf{2}}\right) + \boldsymbol{sin}\left(\frac{\boldsymbol{\theta}}{\mathbf{2}}\right)(\boldsymbol{n_x i} + \boldsymbol{n_y j} + \boldsymbol{n_z k})$$

1.7.3. Системы координат

Преобразование координат обычно происходит в несколько этапов: из нормализованных координат в экранные координаты через промежуточные координатные системы. Прежде чем вершины объекта будут преобразованы в экранные координаты, они проходят через несколько различных координатных систем. Это преобразование через промежуточные системы имеет преимущество, поскольку некоторые операции и вычисления проще выполнять

в определенных системах координат, как это вскоре станет очевидно. [4]

Вероятно, на данный момент вас может запутать, что представляют собой эти координатные системы и как они работают. Мы рассмотрим их более подробно, чтобы создать общее представление и понять функции каждой из них.

Для преобразования координат из одного пространства в другое используются несколько матриц трансформации, среди которых наиболее важными являются матрицы Модели, Вида и Проекции. Координаты вершин начинаются в локальном пространстве как локальные координаты и последовательно преобразуются в мировые координаты, затем в координаты вида, отсечения и, наконец, в экранные координаты. Ниже представлено изображение, иллюстрирующее эту последовательность преобразований и функции каждого из них:

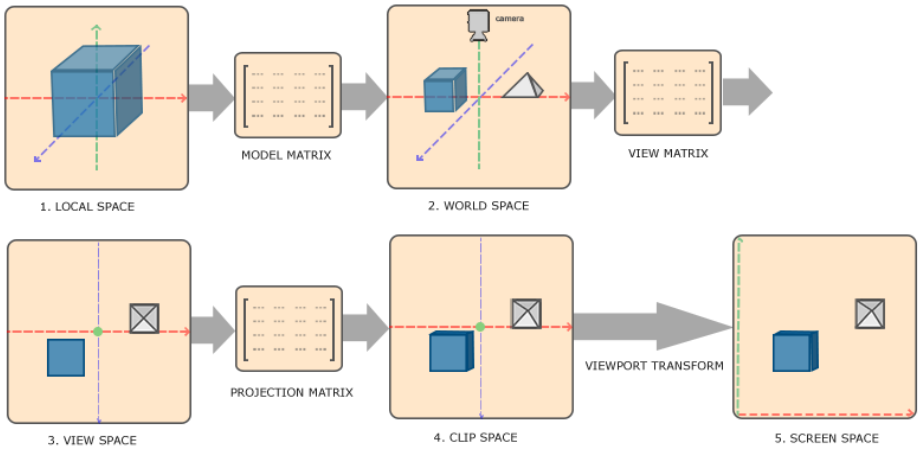

Рисунок 12 – Преобразование координат с локального до экранного

1) Локальные координаты: это координаты объекта, измеряемые относительно начальной точки, расположенной в месте расположения самого объекта.
2) Мировые координаты: представляют собой координаты в более крупной мировой системе, измеряемые относительно единой глобальной точки отсчёта, общей для всех объектов в мировом пространстве.
3) Координаты вида: преобразуют мировые координаты таким образом, что каждая вершина видима, как если бы на неё смотрели из камеры или

с точки зрения наблюдателя.

4) Координаты отсечения: после преобразования в координаты вида, они проецируются в диапазон от -1.0 до 1.0, определяя, какие вершины будут отображаться на экране.
5) Экранные координаты: в процессе преобразования, называемом трансформацией области просмотра, координаты отсечения от -1.0 до 1.0 преобразуются в экранные координаты.

После всех этих преобразований полученные координаты отправляются растеризатору для преобразования их во фрагменты. Преобразование вершин в различные координатные пространства необходимо, поскольку некоторые операции становятся более понятными или более простыми в определённых системах координат.

Например, модификацию объекта удобнее всего выполнять в локальном пространстве, а вычисление операций, учитывающих расположение других объектов, лучше производить в мировых координатах и т.д. Хотя можно было бы задать одну матрицу трансформации, которая преобразовывала бы координаты из локального пространства в пространство отсечения за один шаг, это лишило бы нас гибкости. Ниже мы подробно обсудим каждую координатную систему.

1.7.4. Пространство отсечения

После выполнения вершинных шейдеров, OpenGL ожидает, что все координаты будут находиться в определенном диапазоне, а все, что выходит за его границы, будет отсечено. Координаты, выходящие за пределы этого диапазона, отбрасываются, а оставшиеся становятся видимыми фрагментами на экране. Отсюда и происходит название “пространство отсечения”. [5, 6]

Задание всех видимых координат значениями из диапазона [-1.0, 1.0] может быть интуитивно непонятным, поэтому для работы мы определяем собственный набор координат и затем преобразуем их обратно в нормализованные координаты устройства (NDC), как того ожидает OpenGL.

Для преобразования координат из пространства вида в пространство отсечения используется так называемая матрица проекции, которая определяет диапазон координат, например, от -1000 до 1000 по каждой оси. Матрица проекции трансформирует координаты этого диапазона в нормализованные координаты устройства (-1.0, 1.0). Все координаты, выходящие за пределы заданного диапазона, не попадут в область [-1.0, 1.0] и, следовательно, будут отсечены. Например, координата (1250, 500, 750) в заданном диапазоне не будет видна, так как её X-компонента выходит за границу, и будет преобразована в значение, превышающее 1.0 в NDC, что приведет к отсечению вершины.

Важно отметить, что, если вне объема отсечения находится не весь примитив, например, треугольник, а только его часть, OpenGL перестроит этот треугольник в один или несколько треугольников, полностью находящихся в диапазоне отсечения. Этот объем просмотра, задаваемый матрицей проекции, называется усеченной пирамидой (frustum), и каждая координата, попадающая в эту пирамиду, будет видна на экране пользователя.

Процесс конвертации координат определенного диапазона в нормализованные координаты устройства NDC, которые легко отображаются в двумерные координаты пространства вида, называется проецированием, так как матрица проекции проецирует 3D координаты на простые для преобразования в 2D нормализованные координаты устройства.

После перевода координат всех вершин в пространство отсечения выполняется заключительная операция, называемая перспективным делением. В этом процессе x, y и z компоненты вектора позиции вершины делятся на гомогенную компоненту вектора w. Перспективное деление преобразует 4D координаты пространства отсечения в трехмерные нормализованные координаты устройства. Этот шаг выполняется автоматически после завершения работы каждого вершинного шейдера. Именно после этого этапа полученные координаты отображаются на координаты экрана и превращаются во фрагменты.

Матрица проекции, преобразующая координаты вида в координаты отсечения, может принимать две различные формы, каждая из которых

определяет свою усеченную пирамиду. Мы можем создать ортографическую матрицу проекции или перспективную.

Матрица ортографической проекции задает усеченную пирамиду в форме параллелограмма, которая представляет собой пространство отсечения, как представлено на рисунке 13. Все вершины, находящиеся за пределами этого объема, отсекаются. При создании матрицы ортографической проекции определяются ширина, высота и длина видимой пирамиды отсечения.

Координаты, преобразованные матрицей проекции в пространство отсечения и попадающие в ограниченный пирамидой объем, не подвергаются отсечению. Усеченная пирамида напоминает контейнер и определяет область видимых координат, заданную шириной, высотой, ближней и дальней плоскостями.

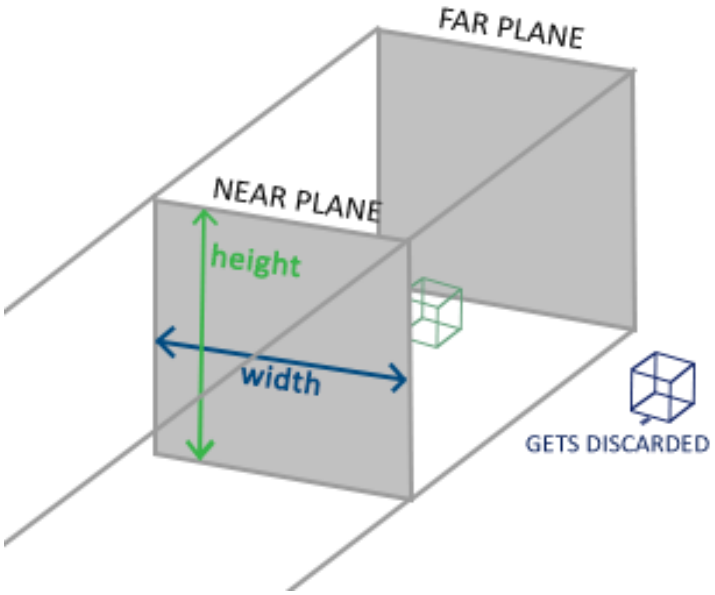

Рисунок 13 – Иллюстрация ортографической проекции в пространстве

Любая координата, расположенная перед ближней плоскостью, отсекается, аналогично поступают и с координатами, находящимися за задней плоскостью. Ортографическая усеченная пирамида непосредственно переводит координаты, попадающие в её объем, в нормализованные координаты устройства. При этом w-компоненты векторов не используются; если w-компонент равен 1.0, то перспективное деление не изменяет значений координат.

Ортографическая матрица проекции отображает координаты

непосредственно на двумерную плоскость, которой является ваш дисплей. Однако прямое проецирование дает нереалистичные результаты, поскольку не учитывает перспективу. Это исправляет матрица перспективной проекции.

Если вы когда-либо наблюдали за реальным миром, то, вероятно, замечали, что объекты, расположенные дальше, кажутся значительно меньше. Этот феномен мы называем перспективой. Перспектива особенно заметна при взгляде в конец бесконечной автомагистрали или железной дороги, как показано на следующем изображении.

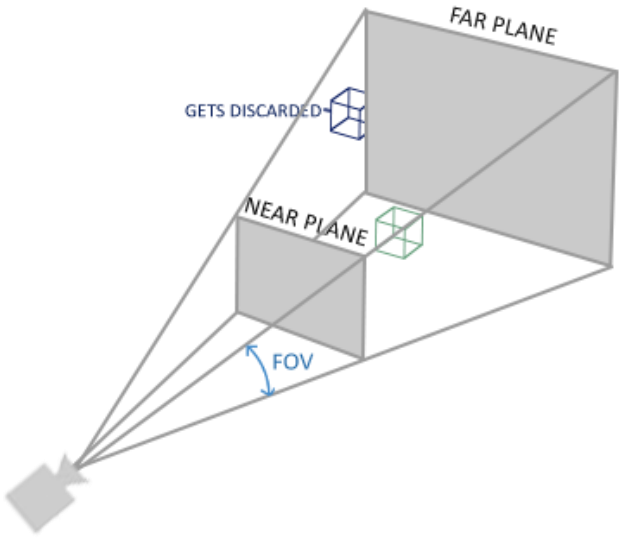

Рисунок 14 – Иллюстрация перспективн5ой проекции в пространстве

Как можно видеть, перспектива приводит к тому, что линии кажутся сходящимися тем больше, чем дальше они находятся. Именно этот эффект пытается имитировать перспективная проекция, достигаемая посредством матрицы перспективной проекции. Данная матрица отображает заданный диапазон усеченной пирамиды в пространство отсечения, при этом манипулируя w-компонентой каждой вершины таким образом, что чем дальше вершина находится от наблюдателя, тем больше становится её w-значение. После преобразования координат в пространство отсечения все они попадают в диапазон от -w до w (вершины, находящиеся вне этого диапазона, отсекаются). OpenGL требует, чтобы конечным выводом вершинного шейдера были координаты, находящиеся между значениями -1.0 и 1.0.

Таким образом, когда координаты находятся в пространстве отсечения, к ним применяется перспективное деление: каждый компонент координаты вершины делится на свою w-компоненту, что уменьшает значения координат пропорционально удалению от зрителя. Это подчеркивает важность w-компонента, так как он способствует реализации перспективной проекции. Полученные координаты находятся в нормализованном пространстве устройства.

При использовании ортогональной проекции каждая координата вершины непосредственно отображается в пространство отсечения без какого-либо мнимого перспективного деления. Хотя перспективное деление выполняется, w-компонент на результат не влияет, оставаясь равным 1, и, следовательно, не оказывает никакого эффекта.

Поскольку ортографическая проекция не учитывает перспективу, объекты, расположенные дальше, не кажутся меньше, что создает необычное визуальное впечатление. По этой причине ортографическая проекция преимущественно используется для 2D-рендеринга и в различных архитектурных или инженерных приложениях, где предпочтительно отсутствие искажений, обусловленных перспективой. В приложениях для 3D-моделирования, таких как Blender, ортографическая проекция иногда применяется во время моделирования, так как она более точно отображает измерения и пропорции каждого объекта.

Ниже приведено сравнение обоих проекционных методов. Можно заметить, что при перспективной проекции удаленные вершины кажутся значительно дальше, в то время как в ортографической проекции скорость удаления вершин остается одинаковой и не зависит от расстояния до наблюдателя.

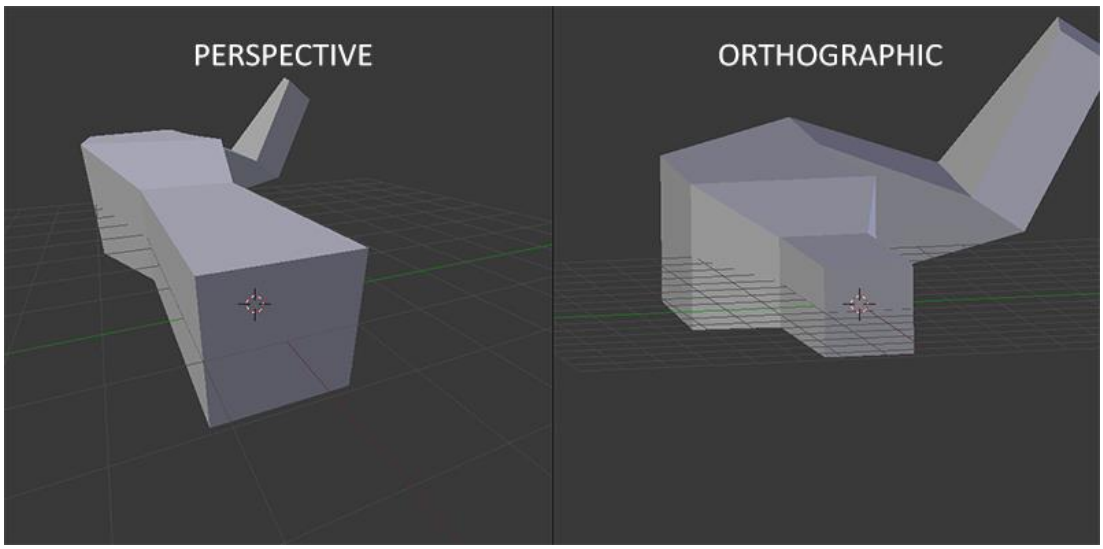

Рисунок 15 – Иллюстрация перспективной проекции в пространстве

Создадим матрицу преобразования для каждого из вышеупомянутых шагов: модели, вида и матрицы проекции. Координата вершины преобразуется в координаты пространства отсечения следующим образом:

$$\boldsymbol{V_{clip} = M_{projection} \cdot M_{view} \cdot M_{model} \cdot V_{local}}$$

Полученная координата вершины должна быть присвоена в вершинном шейдере, после чего OpenGL автоматически выполнит перспективное деление и отсечение.

1.8.Математическая модель освещения по Фонгу

Распространение света в реальном мире представляет собой чрезвычайно сложное явление, зависящее от множества факторов. В условиях ограниченных вычислительных ресурсов мы не можем позволить себе учитывать все нюансы в расчетах. Поэтому освещение в OpenGL базируется на использовании упрощенных математических моделей, приближенных к реальности. Эти модели описывают физику света на основе нашего понимания его природы и рассчитываются гораздо проще по сравнению с полным учетом всех факторов. Одной из таких моделей является модель освещения по Фонгу (Phong). Она состоит из трех основных компонентов:

1) Фоновое освещение (ambient) – описывает общее освещение, которое

равномерно распределяется по поверхности объекта.

2) Рассеянное/диффузное освещение (diffuse) – учитывает свет, рассеивающийся равномерно по всем направлениям от источника света.
3) Бликовое освещение (specular) – моделирует яркие блики, возникающие на поверхности объекта в направлении источника света.

Эти компоненты вместе позволяют создать визуально реалистичное представление освещения объектов в компьютерной графике. Ниже вы можете видеть, что они из себя представляют:

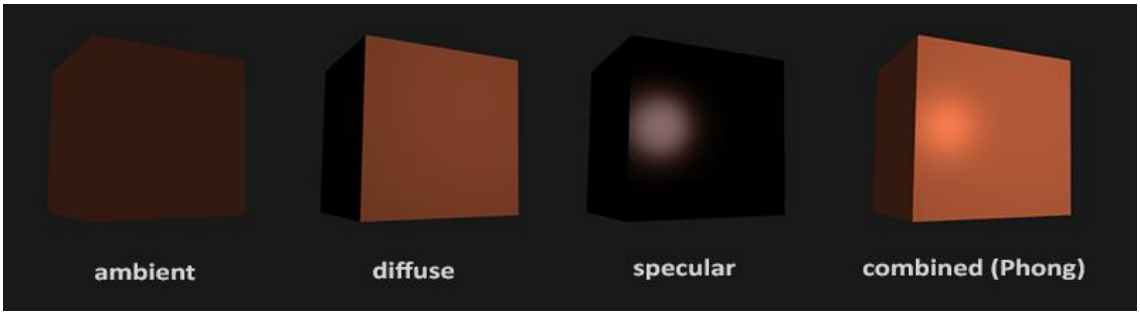

Рисунок 16 – Компоненты освещения и общая модель

Свет в большинстве случаев исходит не от одного, а от множества источников света, окружающих нас, даже если мы их не видим непосредственно. Одним из свойств света является его способность рассеиваться и отражаться в различных направлениях, достигая областей, которые не находятся в прямой видимости. Таким образом, свет может отражаться от различных поверхностей и оказывать косвенное влияние на освещение объекта. Алгоритмы, учитывающие эти свойства света, называются алгоритмами глобального освещения, однако они требуют значительных вычислительных ресурсов и/или сложны в реализации.

Поскольку мы предпочитаем избегать сложных и ресурсоемких алгоритмов, начнем с использования упрощенной модели глобального освещения — модели Фонового освещения. В предыдущем главе вы видели, как применялся неяркий постоянный цвет, который суммировался с цветом фрагмента объекта, создавая впечатление наличия рассеянного света в сцене, хотя прямого источника такого света не было.

Добавить фоновое освещение в сцену достаточно просто. Для этого необходимо взять цвет источника света, умножить его на небольшой константный коэффициент фонового освещения, затем умножить полученное значение на цвет объекта и использовать вычисленную величину в качестве цвета фрагмента.

1.8.1. Диффузное освещение

Фоновое освещение само по себе не предоставляет интересных визуальных результатов, в отличие от диффузного освещения, которое оказывает значительное влияние на внешний вид объекта. Чем более перпендикулярно направлению лучей источника света расположены фрагменты объекта, тем большую яркость им придает диффузная составляющая освещения. Для лучшего понимания диффузного освещения рассмотрим на рисунке 17.

Слева изображен луч света, исходящий от источника и направленный на определенный фрагмент объекта. Необходимо измерить угол падения луча на фрагмент. Воздействие света на цвет фрагмента максимально при перпендикулярном направлении луча к поверхности объекта. Для измерения угла между лучом света и фрагментом используется вектор нормали, перпендикулярный поверхности фрагмента (вектор нормали представлен желтой стрелкой), о чем будет рассказано далее. Угол между двумя векторами можно вычислить с помощью скалярного произведения.

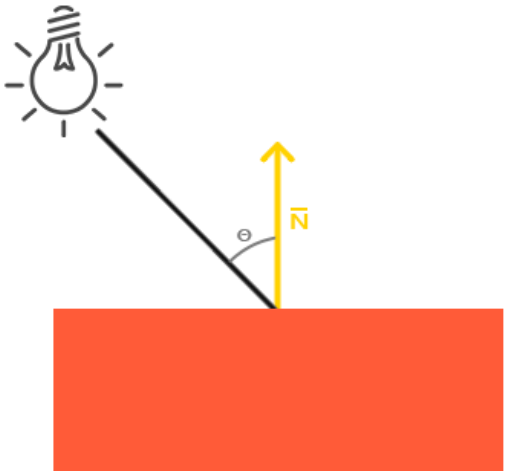

Рисунок 17 – Иллюстрация источника света на поверхность

Таким образом, величина, возвращаемая скалярным произведением, может быть использована для расчета силы влияния источника света на цвет фрагмента, что приводит к различной освещенности фрагментов в зависимости от их ориентации относительно направления световых лучей.

Вектор нормали – это (единичный) вектор, перпендикулярный поверхности, построенной на данной вершине. Поскольку вершина сама по себе не имеет поверхности (это лишь точка в пространстве), вектор нормали определяется на основе соседних вершин. Для вычисления нормалей вершин куба можно применить векторное произведение к граням, либо вручную добавить нормали к вершинам куба.

После добавления вектора нормали к каждой вершине и обновления вершинного шейдера, необходимо обновить указатели атрибутов вершин. Обратите внимание, что объект-лампа извлекает данные вершин из того же массива, однако вершинный шейдер лампы не использует вновь добавленные вектора нормалей. Обновлять шейдеры лампы и её атрибуты не требуется, но необходимо изменить настройку указателей атрибутов в связи с изменением размера массива вершин.

Последним необходимым элементом является позиция текущего фрагмента. Все расчеты освещения будут производиться в мировом пространстве координат, поэтому позиции вершин должны быть в мировых координатах. Преобразование позиции вершины в мировые координаты осуществляется умножением её атрибута позиции только на матрицу модели (без матриц вида и проекции), что может быть выполнено в вершинном шейдере.

Далее, посредством скалярного произведения векторов, вычисляется величина воздействия диффузного освещения на текущий фрагмент. Полученное значение умножается на цвет источника света, что дает компоненту диффузного освещения, которая темнеет с увеличением угла между векторами.

Если угол между векторами превышает 90 градусов, результат скалярного произведения становится отрицательным, что приводит к отрицательной составляющей диффузного света. Отрицательных значений цвета в моделях

освещения не существует, поэтому их следует избегать.

Теперь, когда у нас есть фоновый и диффузный компоненты, мы суммируем их цвета, а затем умножаем результат на цвет объекта, получая таким образом результирующий цвет выходного фрагмента, как показано на рисунке 18.

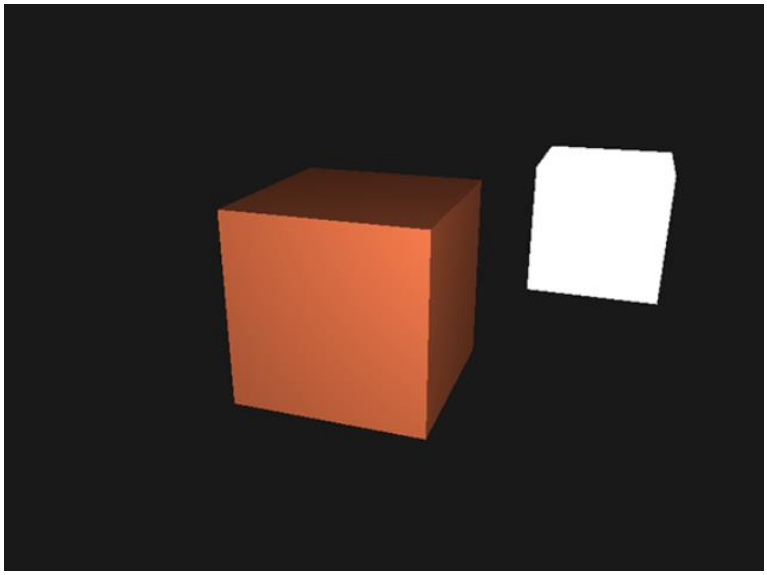

Рисунок 18 – Пример отображения фонового и диффузного освещения

1.8.2. Бликовое освещение

Освещение зеркальных бликов, аналогично рассеянному освещению, базируется на векторе направления источника света и нормали поверхности объекта. Однако в вычислениях также учитывается позиция наблюдателя, то есть направление, в котором игрок смотрит на фрагмент. Зеркальное освещение основывается на отражательных свойствах света. Если представить поверхность объекта в виде зеркала, то освещение бликов будет наибольшим в точке, где мы бы увидели отраженный от поверхности свет источника. Этот эффект проиллюстрирован на следующем изображении:

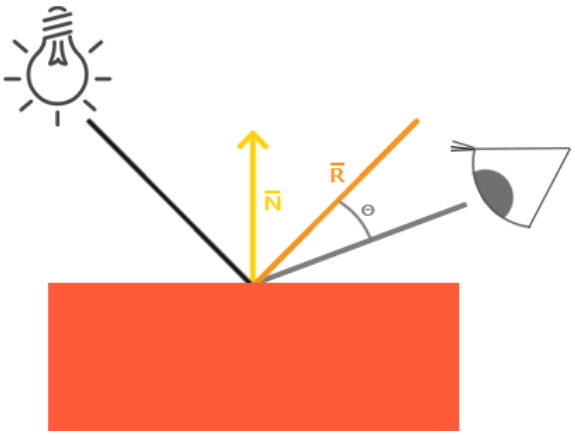

Рисунок 19 – Иллюстрация отражения света от источника к приемнику

Вектор отражения вычисляется путем отражения направления света относительно вектора нормали. Затем определяется угловое расстояние между этим вектором отражения и направлением взгляда наблюдателя; чем меньше угол между этими векторами, тем большее влияние на цвет фрагмента оказывает освещение зеркальных бликов. В результате данного эффекта, при взгляде в направлении источника света, на поверхности объекта наблюдается отраженный блик.

Вектор просмотра является дополнительной переменной, необходимой для расчета освещения зеркальных бликов. Он может быть вычислен с использованием мировых координат точки зрения наблюдателя и положения фрагмента. Затем вычисляется интенсивность блика, которая умножается на цвет освещения и добавляется к ранее рассчитанным компонентам фонового и рассеянного освещения.

Первоначально вычисляется скалярное произведение векторов отражения и направления взгляда (с отсевом отрицательных значений), после чего результат возводится в 32-ю степень. Константное значение 32 определяет силу блеска: чем больше это значение, тем сильнее свет отражается, а не рассеивается, и тем меньше размер пятна блика. Ниже представлено изображение, демонстрирующее влияние различных значений блеска на внешний вид объекта:

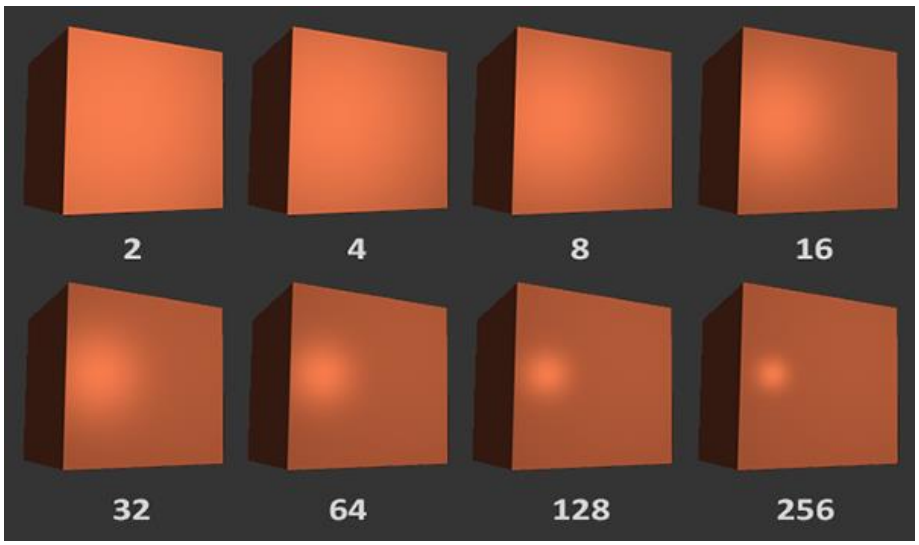

Рисунок 20 – Воздействие различных степеней блеска

Теперь все компоненты освещения модели освещения Фонга рассчитаны, результат можно увидеть на рисунке 21.

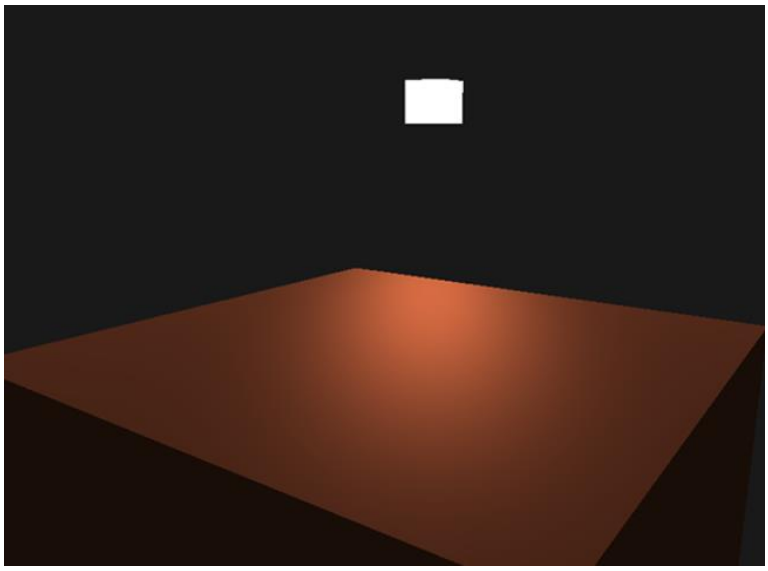

Рисунок 21 – Освещение модели по Фонгу

1.9.Архитектурный подход системы рефлексии и кодогенерации

Разработка современных программных систем, особенно в таких областях как игровые движки, CAD-приложения и инструменты разработки, требует мощных механизмов интроспекции и работы с метаданными во время выполнения программы. Система рефлексии, предоставляющая возможность исследовать структуру программы динамически, становится критически важным

компонентом архитектуры. В отличие от языков с богатой runtime-интроспекцией (таких как C# или Java), C++ не предоставляет встроенных средств для получения информации о типах данных, их членах и отношениях во время выполнения. Это ограничение существенно усложняет реализацию таких важных функций как сериализация объектов, динамическое создание экземпляров классов, интерактивное инспектирование свойств, привязка к скриптовым языкам и многие другие сценарии, требующие работы с типами как с данными.

Традиционные подходы к реализации рефлексии в C++ можно разделить на несколько категорий, каждая из которых имеет свои ограничения. Макросы и шаблонные решения, такие как различные вариации регистрации типов, требуют значительного количества рутинного кода и часто не могут охватить все аспекты сложных типов. Генераторы кода на основе внешних описаний создают дополнительный уровень абстракции и усложняют процесс разработки. Подходы, основанные на декораторах или атрибутах, хотя и более элегантны, все же требуют модификации исходного кода и часто ограничены в возможностях.

Фреймворк Clang-C, являющийся частью инфраструктуры LLVM (Low Level Virtual Machine), представляет собой мощный инструмент для решения задач статического анализа и трансформации кода. Его архитектура, основанная на детальном представлении абстрактного синтаксического дерева AST (Abstract Syntax Tree), предоставляет уникальные возможности для реализации систем рефлексии и кодогенерации. В отличие от традиционных подходов к рефлексии, которые полагаются на динамический анализ во время выполнения, подход на основе Clang-C позволяет перенести значительную часть работы на этап компиляции, что приводит к существенному повышению производительности конечного приложения.

Глубокий анализ возможностей Clang-C показывает, что его можно использовать не только для традиционных задач статического анализа, но и для создания сложных систем метапрограммирования. В частности, механизмы посещения AST, предоставляемые Clang, позволяют извлекать полную

информацию о структуре классов, шаблонов, функций и их взаимосвязях. Эта информация может быть затем использована для генерации дополнительного кода, реализующего различные аспекты рефлексии – от простой интроспекции до сложных сценариев динамического создания и модификации объектов.

Интересным примером практического применения подобных технологий является игровой движок Unreal Engine, где система рефлексии играет ключевую роль в архитектуре. В Unreal Engine механизмы рефлексии используются практически во всех аспектах работы движка - от редактора уровней до системы сериализации и сетевого взаимодействия. Особенностью реализации является использование специального инструмента UHT (Unreal Header Tool), который выполняет предварительный анализ исходного кода и генерирует дополнительные метаданные. На рисунке 22 представлен пример из практики Unreal Engine. [7]

```
UCLASS()
1 derived blueprint class
class LYD_API ALYDAIController : public AAIController
{
    GENERATED_BODY()

public:
    ALYDAIController();

    UPROPERTY()
    AActor* Target;

protected:
    UPROPERTY(EditDefaultsOnly,Category="Components")
    ULYDAIPerceptionComponent* AIPerceptionComponent;

    UFUNCTION()
    virtual void OnPossess(APawn* InPawn) override;
};
```

Рисунок 22 – Пример класса с поддержкой рефлексии в Unreal Engine

В этом примере макросы UCLASS(), UPROPERTY() и UFUNCTION() предоставляют системе рефлексии Unreal Engine всю необходимую информацию о структуре класса. На этапе компиляции UHT анализирует эти аннотации и

генерирует дополнительный код, который позволяет, например, автоматически отображать свойства в редакторе, сериализовать состояние объектов.

Вдохновляясь этим подходом, но стремясь к большей гибкости и независимости от конкретного движка, мы разработали систему, основанную на возможностях LibClang, которая использует аналогичные, но более универсальные аннотации CLASS(), PROPERTY() и FUNCTION(), как проиллюстрировано на рисунке 23. Ключевое отличие нашей системы заключается в использовании API Clang для непосредственного анализа абстрактного синтаксического дерева (AST), что позволяет охватить более широкий спектр возможностей C++, включая шаблоны, constexpr вычисления и другие современные особенности языка. [8, 9]

```
#include "Generated/RCameraComponent.generated.h"

class CLASS() RCameraComponent : public RSceneComponent
{
    GENERATED_BODY()

public:
    void Construct() override;

protected:
    PROPERTY()
    float FOV = 60.0f;

    PROPERTY()
    float Speed = 5.0f;

};

META_REFLECT()
```

Рисунок 23 – Пример класса с поддержкой рефлексии в работе

Глубокий анализ существующих решений выявил несколько фундаментальных проблем, которые необходимо было решить в нашей реализации. Во-первых, это вопрос производительности - традиционные runtime-рефлексии часто требуют значительных накладных расходов. Во-вторых, проблема точности - многие системы не могут корректно обрабатывать сложные шаблонные конструкции. И наконец, вопрос масштабируемости - системы,

жестко привязанные к конкретной архитектуре (как в Unreal Engine), сложно адаптировать для других проектов.

Наша система решает эти проблемы через комбинацию compile-time кодогенерации и эффективной организации метаданных. В отличие от подхода Unreal Engine, где генерация кода выполняется отдельным препроцессором, мы интегрируем анализ непосредственно в процесс сборки, используя возможности libClang для посещения AST. Это позволяет нам сохранить все преимущества статической типизации C++ при добавлении возможностей динамической интроспекции.

2. ТЕХНОЛОГИЧЕСКАЯ

2.1.Архитектура программной реализации

Разрабатываемое программное обеспечение представляет собой полноценную среду для работы с 3D-графикой, включающую все необходимые компоненты профессионального инструмента. В основе реализации лежат современные подходы к разработке графических приложений.

Реализация системы основана на следующих технических решениях:

1) Кроссплатформенность;
   a) Используется GLFW для создания окон и обработки событий ввода (мышь, клавиатура);
   b) OpenGL обеспечивает единый графический конвейер вне зависимости от ОС;
2) Рендеринг и управление сценой;
   a) OpenGL 4.6 с поддержкой современных шейдеров (GLSL);
   b) Буферизация данных через Vertex Array Objects (VAO) и Vertex Buffer Objects (VBO);
   c) Реализация системы камер (FPS-режим)
3) Пользовательский интерфейс;
   a) ImGUI интегрирован в рендер-конвейер OpenGL;
   b) Поддержка drag-and-drop для импорта моделей;
   c) Динамическое обновление параметров через систему рефлексии;
4) Импорт 3D-моделей;
   a) Assimp загружает меши, материалы, текстуры;
   b) Оптимизация данных (нормализация);
5) Рефлексия и сериализация;
   a) Clang API анализирует исходный код, извлекая метаданные;
   b) Макросы вида CLASS(), PROPERTY(), FUNCTION() автоматически регистрируют классы, поля и функции соответственно;
   c) сериализация в JSON/бинарный формат для сохранения данных;

2.2.Реализация системы отображения и управления окнами

Разрабатываемое программное обеспечение представляет собой полноценную среду для работы с 3D-графикой, включающую все необходимые компоненты профессионального инструмента. В основе реализации лежат современные подходы к разработке графических приложений.

Приступая к реализации графического приложения, первым шагом становится настройка рабочего окружения. Процесс начинается с инициализации GLFW, где важно проверить результат операции, так как неудачная инициализация может быть вызвана отсутствием необходимых драйверов или проблемами совместимости. После успешной инициализации устанавливаются параметры будущего окна, включая версию OpenGL и профиль совместимости.

Создание окна через glfwCreateWindow должно сопровождаться проверкой на ошибки, так как эта операция может завершиться неудачей из-за неподдерживаемых параметров или ограничений системы. Получив указатель на созданное окно, необходимо сделать его контекст текущим, что позволит последующим вызовам OpenGL работать с правильным контекстом рендеринга.

Инициализация GLEW требует особого внимания. Установка флага glewExperimental в GL_TRUE позволяет получить доступ к современным функциям OpenGL, но может вызвать ошибки в некоторых реализациях. Проверка возвращаемого значения glewInit обязательна, так как неудачная инициализация сделает невозможным использование любых функций OpenGL.

Настройка вьюпорта через glViewport должна учитывать возможную разницу между размерами окна и фреймбуфера, особенно на системах с высоким DPI. Это гарантирует правильное отображение графики независимо от масштабирования системы.

Основной цикл рендеринга строится вокруг обработки событий и отрисовки кадра. Вызов glfwPollEvents обеспечивает реакцию на пользовательский ввод, в то время как glfwSwapBuffers отвечает за отображение отрисованного кадра. Между этими операциями происходит очистка буферов,

установка шейдеров и параметров рендеринга, а также непосредственный вызов команд отрисовки. Приведенный код в листинге 1 демонстрирует базовый сценарий.

Листинг 1 – Инициализация окна и контекста OpenGL

```
glfwInit();

glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 6);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);

GLFWwindow* window = glfwCreateWindow(WIDTH, HEIGHT, "Window",
nullptr, nullptr);
glfwMakeContextCurrent(window);
glewInit();

int width, height;
glfwGetFramebufferSize(window, &width, &height);
glViewport(0, 0, width, height);

while (!glfwWindowShouldClose(window))
{
      glfwPollEvents();

      glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
      glClear(GL_COLOR_BUFFER_BIT);
      glUseProgram(shaderProgram);
      glBindVertexArray(VAO);
      glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_INT, 0);
      glBindVertexArray(0);

      glfwSwapBuffers(window);
}

glfwTerminate();
```

Важно отметить, что приведенный пример является минимальной рабочей конфигурацией. В реальном приложении следует добавить обработку ошибок, управление ресурсами и более сложную логику рендеринга. Например, стоит рассмотреть возможность изменения размеров окна, обработку различных событий ввода и реализацию правильного завершения работы приложения, что мы сделаем в последующих главах.

2.3.Реализация и настройка рендеринга в OpenGL

После определения вершинных данных необходимо передать их на первый этап графического конвейера — вершинный шейдер. Это осуществляется следующим образом: выделяется память на GPU для хранения вершинных данных, указывается OpenGL, как интерпретировать переданные данные, и передаётся GPU количество данных. Затем вершинный шейдер обрабатывает указанное количество вершин.

Управление этой памятью осуществляется через объекты вершинного буфера (VAO, VBO), которые могут хранить большое количество вершин в памяти GPU, как представлено на листинге 2. Преимущество использования таких объектов буфера заключается в возможности отправки большого количества наборов данных на видеокарту за один раз, без необходимости передавать по одной вершине за раз. Отправка данных с центрального процессора (CPU) на GPU является относительно медленной операцией, поэтому целесообразно отправлять как можно больше данных за один раз. После того как данные оказываются в GPU, вершинный шейдер получает их практически мгновенно.

Листинг 2 – Пример создания VBO с двумя атрибутами

```
1.  glGenBuffers(1, &VBO);
2.  glBindBuffer(GL_ARRAY_BUFFER, VBO);
3.  glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
4.  GL_STATIC_DRAW);
5.
6.  glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 *
7.  sizeof(GLfloat), (GLvoid*)0);
8.      glEnableVertexAttribArray(0);
9.      glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 5 *
10. sizeof(GLfloat), (GLvoid*)(3 * sizeof(GLfloat)));
11. lEnableVertexAttribArray(2);
12.
13. glBindBuffer(GL_ARRAY_BUFFER, 0);
```

2.3.1. Вершинный шейдер

Вершинный шейдер является одним из программируемых шейдеров. Современный OpenGL требует задания вершинного и фрагментного шейдеров для выполнения отрисовки. В связи с этим мы предоставим два простых шейдера для отрисовки нашего треугольника. В следующем уроке мы рассмотрим шейдеры более детально.

Сначала необходимо написать шейдер на специализированном языке GLSL (OpenGL Shading Language), а затем собрать его для использования в приложении.

Как представлено на листинге 3, для обозначения результата работы вершинного шейдера необходимо присвоить значение предопределенной переменной gl_Position, имеющей тип vec4. После завершения работы функции main, независимо от того, что передано в gl_Position, это значение будет использовано в качестве результата работы вершинного шейдера.

Листинг 3 – Пример вершинного шейдера

```
1.  #version 330 core
2.
3.  layout (location = 0) in vec3 position;
4.  layout (location = 1) in vec3 color;
5.  layout (location = 2) in vec2 texCoord;
6.
7.  out vec3 ourColor;
8.  out vec2 TexCoord;
9.
10. uniform mat4 model;
11. uniform mat4 view;
12. uniform mat4 projection;
13.
14. void main()
15. {
16.     gl_Position = projection * view * model * vec4(position, 1.0f);
17.     ourColor = color;
18.     TexCoord = vec2(texCoord.x, 1.0 - texCoord.y);
19. }
```

2.3.2. Фрагментный шейдер

Фрагментный шейдер представляет собой второй и заключительный шейдер, необходимый для отрисовки треугольника. Фрагментный шейдер отвечает за вычисление цветов пикселей.

В компьютерной графике цвет представляется массивом из 4 значений: красного (red), зеленого (green), синего (blue) и прозрачности (alpha), такая компонентная база обозначается как RGBA. При задании цвета в OpenGL или GLSL мы указываем величину каждого компонента в диапазоне от 0.0 до 1.0. Например, если установить величину красного и зеленого компонентов в 1.0f, результатом будет смесь этих цветов — желтый. Комбинация из 3 компонентов предоставляет около 16 миллионов различных цветов.

Фрагментный шейдер требует на выходе только значения цвета, представленного в виде 4-компонентного вектора. Мы можем определить выходную переменную с использованием ключевого слова out и назвать эту переменную color. Затем мы просто присваиваем значение этой переменной vec4 с непрозрачным оранжевым цветом, как представлено на листинге 4. После сборки обоих шейдеров остается только связать их в программу, чтобы использовать их при отрисовке.

Листинг 4 – Пример фрагментного шейдера

```
1.  #version 330 core
2.
3.  in vec3 ourColor;
4.  in vec2 TexCoord;
5.
6.  out vec4 color;
7.
8.  uniform sampler2D ourTexture;
9.
10. void main()
11. {
12.     color = texture(ourTexture, TexCoord);
13. }
```

2.3.3. Шейдерная программа

Прежде чем приступить к работе с графикой, необходимо подготовить шейдерную программу. Этот процесс начинается с создания отдельных шейдеров — как правило, вершинного и фрагментного. Каждый шейдер компилируется независимо, после чего происходит их связывание в программу.

Важно понимать, что на этом этапе могут возникнуть различные проблемы. Например, если шейдер содержит синтаксические ошибки, компиляция завершится неудачно. Аналогично, если выходные переменные вершинного шейдера не соответствуют входным переменным фрагментного шейдера, линковка программы не будет выполнена. Поэтому после каждого этапа необходимо проверять статус операций с помощью glGetShaderiv и glGetProgramiv.

В соответствии с требованиями Core OpenGL, для корректной работы с входными вершинами необходимо использовать VAO. Если VAO не указан, OpenGL может отказаться выполнять отрисовку. Для использования VAO достаточно выполнить привязку VAO. Затем следует настроить и привязать необходимые VBO и указатели на атрибуты, а по завершении работы отвязать VAO для последующего использования. В дальнейшем, каждый раз при необходимости отрисовки объекта, следует просто привязать VAO с требуемыми настройками перед выполнением команды отрисовки объекта, как показано на листинге 5.

Листинг 5 – Генерация VBO и привязка к VAO

```
1.  GLuint VBO, VAO;
2.
3.  glGenVertexArrays(1, &VAO);
4.  glGenBuffers(1, &VBO);
5.  glBindVertexArray(VAO);
6.
7.  glBindBuffer(GL_ARRAY_BUFFER, VBO);
8.  glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
9.  GL_STATIC_DRAW);
10.
11. glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 *
12. sizeof(GLfloat), (GLvoid*)0);
```

```
13.   glEnableVertexAttribArray(0);
14.   glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 5 *
15.   sizeof(GLfloat), (GLvoid*)(3 * sizeof(GLfloat)));
16.   glEnableVertexAttribArray(2);
17.
18.   glBindBuffer(GL_ARRAY_BUFFER, 0);
19.   glBindVertexArray(0);
```

2.4.Загрузка, настройка и применение текстур

После определения вершинных данных необходимо передать их на первый этап графического конвейера – вершинный шейдер. Это осуществляется следующим образом: выделяется память на GPU для хранения вершинных данных, указывается OpenGL, как интерпретировать переданные данные, и передаётся GPU количество данных. Затем вершинный шейдер обрабатывает указанное количество вершин.

Прежде чем использовать текстуры в приложении, необходимо правильно их подготовить и загрузить. Современные графические приложения могут работать с различными форматами изображений, включая PNG, JPEG, BMP и другие. Хотя теоретически можно создать собственный загрузчик текстур, на практике гораздо эффективнее использовать специализированные библиотеки, такие как SOIL (Simple OpenGL Image Library), которые значительно упрощают процесс загрузки и обработки изображений.

После загрузки изображения необходимо создать текстуру в OpenGL. Этот процесс начинается с генерации текстуры с помощью функции glGenTextures(). Затем текстура привязывается к соответствующей цели (например, GL_TEXTURE_2D для двумерных текстур), после чего в нее загружаются данные изображения с помощью glTexImage2D().

После создания текстуры необходимо правильно настроить ее параметры. Ключевые параметры включают:

1) Фильтрацию (GL_TEXTURE_MIN_FILTER и GL_TEXTURE_MAG_FILTER): определяет, как текстура будет масштабироваться при уменьшении или

увеличении. Для плавного отображения рекомендуется использовать GL_LINEAR или GL_LINEAR_MIPMAP_LINEAR;

2) Режим наложения (GL_TEXTURE_WRAP_S и GL_TEXTURE_WRAP_T): определяет поведение текстуры при выходе координат за пределы диапазона [0, 1]. Доступные варианты включают GL_REPEAT (повторение), GL_MIRRORED_REPEAT (зеркальное повторение) и GL_CLAMP_TO_BORDER (фиксация на границе);

Для режима GL_CLAMP_TO_BORDER можно дополнительно задать цвет границы с помощью glTexParameterfv() и GL_TEXTURE_BORDER_COLOR.

Для автоматического создания мипмапов после загрузки основной текстуры достаточно вызвать glGenerateMipmap(). Это создаст полную пирамиду текстур с различными уровнями детализации, которые OpenGL будет автоматически выбирать в зависимости от расстояния до объекта.

Чтобы использовать текстуру, необходимо добавить текстурные координаты к вершинам объекта. Обычно это делается путем расширения вершинного буфера и добавления соответствующих атрибутов. В вершинном шейдере текстурные координаты передаются во фрагментный шейдер, где используется для выборки цвета из текстуры с помощью texture(), реализация представлена на листинге 6.

Листинг 6 – Привязка и отрисовка текстуры на объекте

```
1.  GLuint VBO, VAO;
2.  Texture texture(GL_TEXTURE_2D, "Data/Textures/CRATE.BMP",
3.  ColorFormat::RGB);
4.  texture.Bind();
5.
6.  texture.setParameter(GL_TEXTURE_MAG_FILTER, GL_LINEAR);
7.  texture.setParameter(GL_TEXTURE_MIN_FILTER,
8.  GL_LINEAR_MIPMAP_LINEAR);
9.
10. texture.setParameter(GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
11. texture.setParameter(GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
12.
13. GLfloat borderColor[] = { 0.5f, 0.5f, 0.5f, 1.0f };
14. texture.setParameter(GL_TEXTURE_BORDER_COLOR, borderColor);
15.
```

```
16.   texture.UnBind();
```

2.5.Применение геометрических преобразований

Библиотека GLM (OpenGL Mathematics) предоставляет мощный и удобный инструментарий для работы с аффинными преобразованиями в современных графических приложениях. Ее главное преимущество заключается в полной совместимости с шейдерными языками OpenGL и интуитивно понятном синтаксисе, который значительно упрощает реализацию сложных математических операций.

При работе с преобразованиями в GLM важно понимать несколько ключевых принципов. Во-первых, все преобразования работают с матрицами 4×4 в однородных координатах, что соответствует стандартам современных графических API. Во-вторых, GLM предоставляет как отдельные функции для каждого типа преобразований (перенос, масштабирование, вращение), так и возможность их комбинирования. Особое внимание следует уделить порядку применения преобразований - в GLM, как и в математике, умножение матриц выполняется справа налево, что соответствует последовательному применению преобразований.

Одной из самых распространенных операций является создание матрицы модели (model matrix), которая объединяет все преобразования объекта. В GLM эту матрицу можно получить несколькими способами. Наиболее простой - последовательное применение функций translate, rotate и scale к единичной матрице. Однако для сложных случаев, особенно когда вращение задается кватернионом, лучше использовать комбинацию отдельных матриц с правильным порядком умножения, как представлено в листинге 7.

Листинг 7 – Аффинное преобразовании точки

```
1.   FMatrix FTransform::GetMatrix() const
2.   {
3.       return Position.GetTranslationMatrix() * Quaternion.GetMatrix()
     * Scale.GetScaleMatrix();
4.   }
```

Особую сложность в практической реализации представляют иерархические преобразования, когда объекты связаны отношениями родитель-потомок. В GLM такие преобразования удобно реализовывать через рекурсивное вычисление мировых матриц. Каждый объект должен хранить свою локальную матрицу преобразований и ссылку на родителя. При вычислении мировой матрицы объекта нужно учитывать цепочку всех его родителей. Для оптимизации производительности в таких случаях используют механизм "грязных" флагов, который позволяет избежать избыточных пересчетов, как представлено в листинге 8.

Листинг 8 – Расчета глобальной трансформации

```
1.  FMatrix RSceneComponent::GetWorldTransformMatrix() const
2.  {
3.      if (const std::shared_ptr<RSceneComponent> Parent =
    GetParentComponent())
4.      {
5.          return Parent->GetWorldTransformMatrix() *
    GetRelativeTransformMatrix();
6.      }
7.
8.      return GetRelativeTransformMatrix();
9.  }
```

При реализации графических приложений правильная настройка проекционных преобразований играет ключевую роль в формировании конечного изображения. Библиотека GLM создает усеченную пирамиду, которая определяет видимое пространство. Всё, что находится за пределами этого пространства и не попадает в объем усеченной пирамиды, будет отсечено. Перспективная усеченная пирамида может быть представлена как трапециевидная коробка, каждая координата внутри которой отображается в точку в пространстве отсечения. Как представлено на листинге 9, первый параметр устанавливает значение field of view (поле обзора), определяющее, насколько велика видимая область. Для реалистичного представления этот параметр обычно устанавливается равным 45 до 120. Второй параметр задает соотношение сторон, рассчитываемое путем деления ширины области просмотра на её высоту. Третий и четвертый параметры задают ближнюю и дальнюю

плоскости усеченной пирамиды. Обычно мы устанавливаем ближайшее расстояние равным 0.1, а дальнее – 100. Все вершины, расположенные между ближней и дальней плоскостью и попадающие в объем усеченной пирамиды, будут визуализированы.

Листинг 9 – Расчета матриц проекции и вида

```
1.  FMatrix RCameraComponent::GetViewMatrix() const
2.  {
3.      const FVector WorldPosition = GetWorldPosition();
4.      return glm::lookAt(WorldPosition, WorldPosition +
    GetForwardVector(), GetUpVector());
5.  }
6.
7.  FMatrix RCameraComponent::GetProjectionMatrix() const
8.  {
9.      auto Engine = REngine::GetEngine();
10.     RCheckReturn(Engine, {});
11.
12.     auto Editor = Engine->GetEditor();
13.     RCheckReturn(Editor, {});
14.
15.     auto Frame = Editor->GetFrame();
16.     RCheckReturn(Frame, {});
17.
18.     const FIntVector FrameSize = Frame->GetFrameSize();
19.     const GLfloat FrameRatio = static_cast<GLfloat>(FrameSize.x) /
    static_cast<GLfloat>(FrameSize.y);
20.
21.     return glm::perspective(glm::radians(FOV), FrameRatio, 0.1f,
    100.0f);
22. }
```

2.6.Практическое применение модели освещения по Фонгу в рендеринге

Реализация модели освещения по Фонгу потребовала разработки комплексного решения, интегрирующего математические основы освещения с современными технологиями компьютерной графики. Практическая реализация системы основана на использовании библиотеки GLM для матричных вычислений и шейдерного языка GLSL для программируемого конвейера рендеринга.

Основой системы освещения стал модульный подход, позволяющий гибко

настраивать параметры материалов и источников света. Центральным компонентом реализации выступает шейдерная программа, обрабатывающая три ключевых аспекта освещения: фоновую составляющую, диффузное отражение и зеркальные блики. Каждый из этих компонентов требует особого подхода к вычислениям и передаче параметров.

Фоновое освещение реализовано как базовый уровень, обеспечивающий минимальную видимость объектов даже в отсутствие прямых источников света. В системе предусмотрена возможность настройки глобального фонового освещения сцены и индивидуальных коэффициентов для каждого материала. Это позволяет создавать разнообразные световые сценарии - от ярко освещенных помещений до полумрака подземных пещер.

Диффузная составляющая, основанная на законе Ламберта, потребовала тщательной работы с векторами нормалей и направлениями света. Особое внимание было уделено преобразованию нормалей из локального пространства объекта в мировое пространство с использованием нормальной матрицы. Это преобразование учитывает все трансформации объекта (перемещение, вращение, масштабирование) и гарантирует корректность расчетов освещения при любых условиях. Для оптимизации производительности все векторные операции выполняются с использованием встроенных функций GLSL.

Реализация зеркальных бликов основана на вычислении вектора отражения и его сопоставлении с направлением взгляда наблюдателя. В системе предусмотрена регулировка параметра блеска (shininess), позволяющая имитировать материалы с различной степенью полировки - от матовых поверхностей до идеальных зеркал. Для достижения большей реалистичности в расчетах используется возведение в степень с ограничением минимального значения, что исключает артефакты при малых углах.

Важным аспектом реализации стала работа с несколькими источниками света различного типа. Система поддерживает направленные, точечные и прожекторные источники с индивидуальными параметрами для каждого компонента освещения. Для эффективной передачи данных в шейдеры

используются uniform-буферы, что значительно снижает нагрузку на графический конвейер при большом количестве источников света, как представлено в листинге 10.

Листинг 10 – Заполнение uniform буфера освещения

```
Shader->Use();
Shader->setUniform("pointLight.position", CameraPosition);
Shader->setUniform("pointLight.ambient", glm::vec3(0.1f, 0.1f, 0.1f));
Shader->setUniform("pointLight.diffuse", glm::vec3(0.5f, 0.5f, 0.5f));
Shader->setUniform("pointLight.specular", glm::vec3(1.0f, 1.0f, 1.0f));
Shader->setUniform("pointLight.constant", 1.0f);
Shader->setUniform("pointLight.linear", 0.22f);
Shader->setUniform("pointLight.constant", 0.20f);
```

Особое внимание было уделено интеграции системы освещения с различными типами материалов. Реализация поддерживает как простые цветовые коэффициенты, так и сложные текстурированные материалы с картами нормалей. Это позволяет создавать разнообразные визуальные эффекты - от гладких пластиковых поверхностей до шероховатых металлических конструкций с выраженным микрорельефом.

Данная реализация модели освещения по Фонгу успешно интегрирована в систему трехмерной визуализации и демонстрирует стабильную работу в различных условиях, общая модель расчетов представлено в листинге 11. Полученные результаты подтверждают корректность теоретических положений и эффективность выбранных практических решений.

Листинг 11 – Расчет освещения по Фонгу от точечного освещения

```
uniform PointLight pointLight;

vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir)
{
    vec3 lightDir = normalize(light.position - fragPos);
    vec3 ambient  = light.ambient * vec3(texture(material.diffuse, TexCoord));

    float diff = max(dot(normal, lightDir), 0.0);
    vec3 diffuse  = light.diffuse  * diff *
    vec3(texture(material.diffuse, TexCoord));
```

```
12.
13.       vec3 reflectDir = reflect(-lightDir, normal);
14.       float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32.0f);
15.       vec3 specular = light.specular * spec *
16.       vec3(texture(material.specular, TexCoord));
17.
18.       float distance    = length(light.position - fragPos);
19.       float attenuation = 1.0 / (light.constant + light.linear *
20.       distance + light.quadratic * (distance * distance));
21.
22.       ambient  *= attenuation;
23.       diffuse  *= attenuation;
24.       specular *= attenuation;
25.
26.       return (ambient + diffuse + specular);
27.   }
```

2.7.Интеграция графического интерфейса

Для реализации графического интерфейса была выбрана библиотека ImGUI, представляющая собой мощное решение для создания инструментальных панелей и редакторов. Основными преимуществами данного выбора стали:

1) Immediate-mode подход, позволяющий описывать интерфейс декларативным образом непосредственно в коде логики приложения.
2) Высокая производительность и минимальные накладные расходы.
3) Простота интеграции с современными графическими API (OpenGL, Vulkan, DirectX).
4) Гибкая система кастомизации внешнего вида.
5) Широкая экосистема расширений и плагинов

Архитектура системы интерфейса была построена по модульному принципу, где каждый функциональный компонент (окно, панель управления, виджет) представляет собой независимый модуль с четко определенным интерфейсом взаимодействия. Такой подход позволил достичь высокой степени сопровождаемости кода и упростить процесс добавления новых функциональных возможностей.

Первоначальная настройка системы интерфейса начинается с создания

контекста ImGUI и его привязки к оконной системе и графическому API. В проекте использовалась связка GLFW и OpenGL, как наиболее стабильная и кроссплатформенная комбинация, пример создания контекста представлен в листинге 12. [10, 11, 12]

Листинг 12 – Инициализации контекста ImGUI

```
ImGui::CreateContext();
ImGuiIO& io = ImGui::GetIO(); (void)io;
io.ConfigFlags |= ImGuiConfigFlags_NavEnableKeyboard;
io.ConfigFlags |= ImGuiConfigFlags_DockingEnable;

io.FontGlobalScale = 1.1f;
ImGui::GetStyle().ScaleAllSizes(1.1f);

ImGui::StyleColorsDark();
ImGui_ImplOpenGL3_NewFrame();
ImGui_ImplGlfw_NewFrame();
ImGui::NewFrame();
ImGui::EndFrame();
ImGui::Render();
ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());
```

Основной структурной единицей интерфейса являются окна, для работы с которыми был разработан базовый класс RWindow. Этот абстрактный класс определяет общий интерфейс для всех окон редактора, включая такие свойства как заголовок, видимость и состояние фокуса. Особенностью реализации стало разделение процесса отрисовки окна на три этапа: подготовка (BeginWindow), собственно отрисовка содержимого (DrawContents) и завершение (EndWindow). Такой подход обеспечивает единообразие работы со всеми элементами интерфейса и упрощает добавление новых окон. Для управления компоновкой интерфейса был создан класс EditorDockSpace, который отвечает за организацию док-станций - областей, куда могут пристыковываться окна редактора. Эта система позволяет пользователю настраивать рабочее пространство под свои нужды, создавая индивидуальные компоновки из окон. Реализация базового класса представлен в листинге 13.

Листинг 13 – Реализация базового класса RWindow

```
class RWindow : public RContainerWidget
{
```

```
18.  public:
19.      using RContainerWidget::RContainerWidget;
20.
21.      static void SetForceViewport(const ImGuiID& ViewportID);
22.      FVector2D GetWindowPosition() const;
23.      FVector2D GetWindowSize() const;
24.      void SetWindowFlags(const ImGuiWindowFlags& Flags);
25.      bool IsPointInside(const FVector2D& Point) const;
26.
27.      virtual bool IsNeedDockspace() const;
28.
29.      void Construct() override;
30.      void Initialize(const std::shared_ptr<REditor>& InEditor)
     override;
31.      void Draw() override;
32.
33.      void OnMouseDown(int Button, int Mods, const FVector2D&
     CursorPosition) override;
34.      void OnMouseUp(int Button, int Mods, const FVector2D&
     CursorPosition) override;
35.
36.  protected:
37.      void SetWindowName(const std::string& Name);
38.      std::string GetWindowName() const;
39.
40.      virtual void PushWindowStyle();
41.      virtual void PopWindowStyle();
42.
43.      virtual void InitDockspace();
44.      virtual void DrawWindowContent() const;
45.
46.  private:
47.      FWindowData WindowData;
48.      int WindowStyleNums;
49.
```

};

Графический интерфейс редактора представляет собой сложную систему взаимосвязанных панелей и инструментов, организованных в единое рабочее пространство. Центральным элементом интерфейса выступает Viewport – область визуализации 3D-сцены, где происходит непосредственное взаимодействие с объектами. Viewport не просто отображает результат рендеринга, но и предоставляет инструменты управления камерой, включая орбитальное вращение, панорамирование и масштабирование. Визуальные гизмо позволяют трансформировать объекты непосредственно в окне просмотра,

обеспечивая интуитивно понятное взаимодействие со сценой. Система выделения объектов реализована через пиккинг с визуальной обратной связью, что значительно упрощает работу со сложными сценами.

Content Browser представляет собой полнофункциональный файловый менеджер проекта, интегрированный непосредственно в редактор. Его реализация включает древовидную навигацию по директориям, миниатюрный просмотр ассетов и сложную систему фильтрации. Особое внимание уделено поддержке drag-and-drop операций, позволяющих легко импортировать ресурсы из внешних источников или перераспределять их внутри проекта. Для графических ассетов реализована система предпросмотра с возможностью быстрого просмотра текстур, моделей и материалов без необходимости открывать их в специализированных программах.

World Outliner организует объекты сцены в иерархическую структуру, отражающую родительско-дочерние отношения между ними. Этот инструмент не просто отображает список объектов, но и позволяет управлять их видимостью, блокировкой и порядком в иерархии. Реализована система поиска по имени и фильтрации по типу объекта, что особенно полезно при работе со сложными сценами, содержащими сотни элементов. Контекстное меню предоставляет быстрый доступ к часто используемым операциям: созданию примитивов, группировке объектов или применению стандартных материалов.

Панель Object Settings представляет собой динамический инспектор свойств, содержание которого меняется в зависимости от выбранного объекта. Для трансформаций реализованы комбинированные элементы управления, позволяющие как вводить точные числовые значения, так и интерактивно изменять параметры с помощью мыши. Компонентная система отображается в виде сворачиваемых секций, где каждый компонент предоставляет свои уникальные настройки. Особое внимание уделено редактору материалов, который включает визуальный пикер цвета, слайдеры для физических параметров и превью шейдеров в реальном времени.

Консоль Logs выполняет не только функцию отображения системных

сообщений, но и выступает важным инструментом отладки. Реализована система цветового кодирования сообщений по их типу (информация, предупреждение, ошибка), а также расширенный фильтр для быстрого поиска нужных записей. Для критических ошибок предусмотрена возможность перехода к месту возникновения проблемы через двойной клик. Логи сохраняют историю между сеансами работы, что позволяет анализировать проблемы, возникшие при предыдущих запусках редактора.

Все элементы интерфейса связаны сложной системой взаимодействий: выделение объекта в World Outliner автоматически активирует его свойства в Object Settings, импорт модели через Content Browser сразу делает ее доступной в сцене, а ошибки при операциях мгновенно отображаются в консоли Logs. Такая глубокая интеграция между компонентами интерфейса создает целостную рабочую среду, где все инструменты работают согласованно, значительно ускоряя процесс работы со сложными 3D-сценами, результат интеграции иллюстрируется на рисунке 24.

Интеграция ImGUI в проект потребовала решения ряда сложных задач, включая синхронизацию состояния интерфейса с системой рендеринга, обработку ввода и управление ресурсами. Результатом стала гибкая и производительная система интерфейса, которая не только удовлетворяет всем требованиям редактора, но и предоставляет возможности для дальнейшего расширения. Особое внимание было уделено оптимизации - даже при сложных сценах и большом количестве окон интерфейс сохраняет плавность работы и быстроту реакции на действия пользователя.

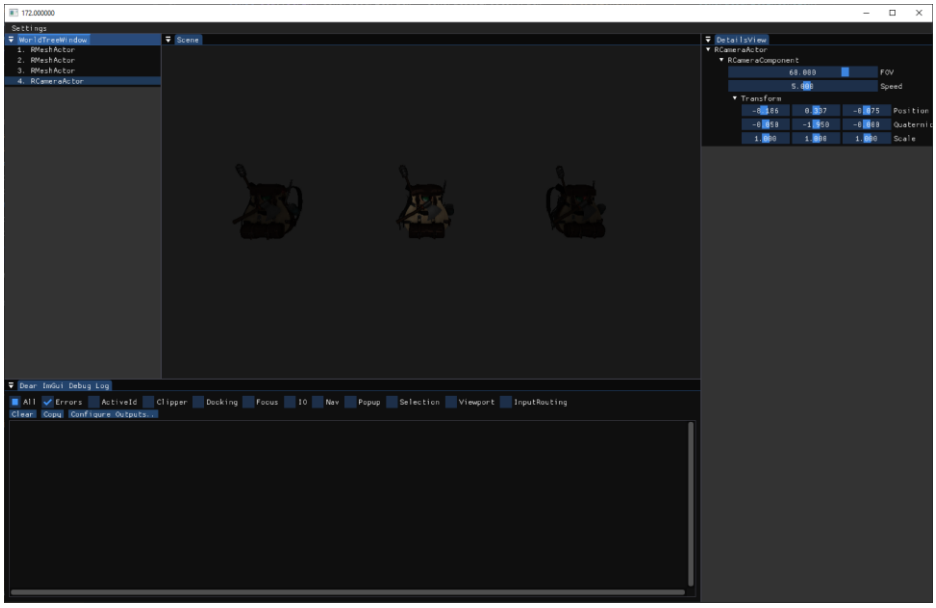

Рисунок 24 – Интерфейс разработанного графического программного обеспечения

2.8. Реализация системы рефлексии и кодогенерации на Clang

Разработанная система рефлексии построена на многоуровневой архитектуре, где каждый слой отвечает за определенный аспект функциональности. Базовым уровнем является анализатор исходного кода, использующий Clang AST для извлечения структурной информации. Над ним располагается слой трансформации, преобразующий сырые данные AST в семантическую модель рефлексии. Завершает архитектуру генератор кода, производящий оптимизированные метаданные и вспомогательные функции.

Особенностью нашей реализации является использование функции clang_visitChildren() для рекурсивного обхода AST. Этот подход был выбран благодаря его надежности и гибкости - в отличие от ручного парсинга исходного кода, использование официального API Clang гарантирует корректную обработку всех конструкций языка, включая сложные случаи шаблонов, макроподстановок и условной компиляции, как представлено в листинге 14. Функция-обработчик получает полную информацию о каждом узле дерева, включая его тип, расположение в исходном коде и семантические свойства.

Листинг 14 – Рекурсивный обход AST файла

```
1.  ReflectionGenerator* This =
    static_cast<ReflectionGenerator*>(ClientData);
2.  RCheckReturn(This, CXChildVisit_Break);
3.
4.  const CXSourceLocation Location = clang_getCursorLocation(Cursor);
5.  if(clang_Location_isFromMainFile(Location) == 0)
6.  {
7.      return CXChildVisit_Continue;
8.  }
9.
10. std::shared_ptr<ClassParser> Parser =
    std::make_shared<ClassParser>();
11. if (Parser->CanParse(Cursor))
12. {
13.     Parser->SetSolutionPath(This->SolutionPath);
14.     Parser->Parse(Cursor);
15.     This->ClassParsers.push_back(Parser);
16. }
17.
18. clang_visitChildren(Cursor, VisitRecursive, This);
19. return CXChildVisit_Continue;
```

Когда система обнаруживает объявление класса или структуры с аннотацией CLASS(), запускается многоэтапный процесс анализа и генерации метаданных. Первым шагом является извлечение базовой информации о классе – его имени, полного квалифицированного имени с учетом пространств имен, спецификаторов доступа и модификаторов (final, abstract и т.д.). Особое внимание уделяется корректной обработке шаблонных классов - система сохраняет информацию обо всех параметрах шаблона, включая их типы и значения по умолчанию.

Следующим этапом является анализ аннотации CLASS() и извлечение указанных в ней атрибутов. Аннотация может содержать произвольное количество параметров, которые влияют на поведение системы рефлексии в отношении данного класса. Например, атрибут "Serializable" указывает, что объекты этого класса должны поддерживать сериализацию, а "Editable" - что свойства класса могут изменяться через редактор.

После обработки общей информации о классе система приступает к рекурсивному анализу его содержимого, как представлено в листинге 15. Для этого используется все та же функция clang_visitChildren(), но с ограничением

области видимости текущим классом. Это позволяет эффективно обрабатывать даже очень большие классы с десятками полей и методов, не загружая всю структуру AST в память.

Листинг 15 – Рекурсивный обход AST класса

```
1.  CXChildVisitResult ClassParser::VisitChildRecursive(CXCursor
    Cursor, CXCursor Parent, CXClientData ClientData)
2.  {
3.      ClassParser* This = static_cast<ClassParser*>(ClientData);
4.      RCheckReturn(This, CXChildVisit_Break);
5.
6.      const CXCursorKind CursorKind = clang_getCursorKind(Cursor);
7.      if (CursorKind == CXCursor_CXXBaseSpecifier)
8.      {
9.          This->ParentClassName =
    clang_getCString(clang_getCursorDisplayName(Cursor));
10.     }
11.     if (CursorKind == CXCursor_FieldDecl)
12.     {
13.         const std::shared_ptr<PropertyParser> Parser =
    std::make_shared<PropertyParser>();
14.         if (Parser->CanParse(Cursor))
15.         {
16.             Parser->Parse(Cursor);
17.             This->Properties.push_back(Parser);
18.         }
19.     }
20.     else if (CursorKind == CXCursor_CXXMethod)
21.     {
22.         const std::shared_ptr<FunctionParser> Parser =
    std::make_shared<FunctionParser>();
23.         if (Parser->CanParse(Cursor))
24.         {
25.             Parser->Parse(Cursor);
26.             This->Functions.push_back(Parser);
27.         }
28.     }
29.
30.     return CXChildVisit_Continue;
31. }
```

Каждое поле класса, помеченное аннотацией PROPERTY(), подвергается тщательному анализу. Система определяет точный тип поля, включая все квалификаторы (const, volatile), указатели, ссылки и массивы. Для сложных типов, таких как шаблоны стандартной библиотеки или пользовательские шаблонные классы, генерируется полное описание с информацией о параметрах

шаблона.

Особый интерес представляет вычисление смещения поля в структуре класса. В отличие от простых случаев, где можно использовать стандартные средства языка типа offsetof, для сложных иерархий наследования и классов с виртуальными функциями требуется использование API Clang для получения точной информации о расположении полей в памяти.

Аннотация PROPERTY() может содержать множество параметров, определяющих поведение поля в системе рефлексии. Например, параметр "Range(0,100)" указывает на допустимый диапазон значений для числового поля, а "Tooltip("Description")" добавляет поясняющий текст для отображения в редакторе. Все эти параметры тщательно анализируются и сохраняются в метаданных.

Методы класса, помеченные аннотацией FUNCTION(), обрабатываются с учетом всех особенностей их сигнатуры. Система анализирует возвращаемый тип, типы всех параметров, квалификаторы метода (const, volatile), спецификаторы исключений (noexcept), а также атрибуты, указанные в аннотации.

Для каждого параметра метода извлекается не только его тип, но и имя, если оно указано в исходном коде. Это важно для генерации удобочитаемых метаданных, которые могут использоваться, например, в редакторах скриптов или инструментах документирования.

Особое внимание уделяется обработке перегруженных методов - система генерирует уникальные идентификаторы для каждой перегрузки, учитывая типы параметров. Это позволяет однозначно идентифицировать конкретную перегрузку при динамическом вызове через механизм рефлексии.

На основе собранной информации система генерирует несколько категорий метаданных, каждая из которых предназначена для решения определенных задач. Основной структурой метаданных является шаблонная специализация, содержащая полное описание класса. Эта структура включает информацию о имени класса, его атрибутах, списке полей с их типами и

атрибутами, а также списке методов с полными сигнатурами. Особый интерес представляет генерация кода для каждого проанализированного файла, пример сгенерированного файла можно рассмотреть в листинге 16.

Листинг 16 – Пример сгенерированного кода класса

```
#pragma once
#include "MetaReflection/MetaReflection.h"

#ifdef GENERATED_BODY
#undef GENERATED_BODY
#endif

#define GENERATED_BODY(...)\
public:\
     using RSceneComponent::RSceneComponent;\
     std::string GetClassName() const override { return "RCameraComponent" ; }\
     static std::string GetStaticClassName() { return "RCameraComponent" ; }\
private:\
     using ThisClass = RCameraComponent;\
     using SuperClass = RSceneComponent;\
     friend struct refl_impl::metadata::type_info__<RCameraComponent>;\
private:

#ifdef META_REFLECT
#undef META_REFLECT
#endif

#define META_REFLECT(...)\
REFL_AUTO(\
     type(RCameraComponent, bases<RSceneComponent>), \
     field(FOV, Serializable()), \
     field(Speed, Serializable())\
)
```

ЗАКЛЮЧЕНИЕ

Представленное исследование и реализация обосновывает необходимость создания современного решения для 3D-визуализации, способного объединить производительность специализированных коммерческих продуктов с гибкостью открытых технологий. Анализ текущего состояния отрасли выявил значительный потенциал для разработки оптимизированного инструментария на базе OpenGL, который мог бы эффективно решать задачи визуализации в различных предметных областях - от научных исследований до инженерного проектирования.

Основной вклад работы заключается в создании принципиально нового подхода к построению графических приложений, сочетающего модульную архитектуру с системой динамической рефлексии данных. Это позволяет достичь высокой степени адаптируемости решения под конкретные задачи визуализации при сохранении производительности, характерной для нативных C++ приложений.

Использование современных технологий, включая OpenGL 4.6 и методы метапрограммирования, обеспечивает разрабатываемому решению не только кроссплатформенность, но и возможность интеграции в различные технологические цепочки. Особое значение имеет образовательный потенциал проекта, который может служить как практическим инструментом, так и демонстрационной платформой для изучения принципов компьютерной графики.

Перспективы дальнейшего развития проекта связаны с расширением функциональных возможностей системы, оптимизацией алгоритмов рендеринга и адаптацией решения для работы с графическими API, что позволит сохранять его актуальность в условиях быстро развивающихся технологий визуализации.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Уроки по OpenGL с сайта OGLDev / [Электронный ресурс] // Уроки по OpenGL с сайта OGLDev : [сайт]. — URL: https://triplepointfive.github.io/ogltutor/ (дата обращения: 16.06.2025).
2. Joey de Vries OpenGL / Joey de Vries [Электронный ресурс] // Learn OpenGL : [сайт]. — URL: https://learnopengl.com/ (дата обращения: 04.06.2025).
3. Song Ho Ahn (안성호) OpenGL / Song Ho Ahn (안성호) [Электронный ресурс] // OpenGL : [сайт]. — URL: https://www.songho.ca/opengl/ (дата обращения: 16.06.2025).
4. Sam Buss 3D Computer Graphics: A Mathematical Introduction with (Modern) OpenGL / Sam Buss [Электронный ресурс] // 3D Computer Graphics: A Mathematical Introduction with (Modern) OpenGL : [сайт]. — URL: https://mathweb.ucsd.edu/~sbuss/MathCG2/(дата обращения: 16.06.2025).
5. Куров, В. Л. Компьютерная графика: модели и алгоритмы [Текст] / Куров, В. Л. — 1-е издание. — М.: Наука, 2023 — 752 с.
6. Khronos Group OpenGL Programming Guide / Khronos Group [Электронный ресурс] // OpenGL Programming Guide : [сайт]. — URL: https://www.khronos.org/opengl/wiki/ (дата обращения: 16.06.2025).
7. Epic Games Unreal Engine 5.2 Documentation / Epic Games [Электронный ресурс] // Unreal Engine 5.2 Documentation : [сайт]. — URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-5-2-documentation?application_version=5.2 (дата обращения: 16.06.2025).
8. The Clang Team Clang 21.0.0git documentation / The Clang Team [Электронный ресурс] // Clang 21.0.0git documentation : [сайт]. — URL: https://clang.llvm.org/docs/index.html (дата обращения: 16.06.2025).
9. The Clang Team llvm-project / The Clang Team [Электронный ресурс] // GitHub : [сайт]. — URL: https://github.com/llvm/llvm-project (дата обращения: 16.06.2025).
10. Unity Technologies Unity Documentation / Unity Technologies [Электронный

ресурс] // Unity Documentation : [сайт]. — URL: https://docs.unity3d.com/Manual/index.html (дата обращения: 16.06.2025).

11. Unity Technologies IMGUI Basics / Unity Technologies [Электронный ресурс] // Unity Documentation : [сайт]. — URL: https://docs.unity3d.com/2023.1/Documentation/Manual/gui-Basics.html (дата обращения: 16.06.2025).
12. ocornut imgui / ocornut [Электронный ресурс] // GitHub : [сайт]. —URL: https://github.com/ocornut/imgui (дата обращения: 16.06.2025).

ПРИЛОЖЕНИЕ А

В графическую часть выпускной квалификационной работы входят 8 чертежей.

1) Прототипирование
2) Вершинные данные и шейдеры
3) Текстурирование
4) Аффинные преобразования
5) Виды проекций
6) Тривиальное освещение
7) Освещение по Фонгу
8) Графический и программный интерфейс