Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное 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 (РК)

КАФЕДРА Системы автоматизированного проектирования (РК6)

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОМУ ПРОЕКТУ

НА ТЕМУ:

«Разработка графического программного обеспечения для визуализации трехмерных объектов»

Студент РК6-83Б _____ **Губанов Д.А.**
(Подпись, дата) И.О. Фамилия

Руководитель _____ **Витюков Ф.А.**
(Подпись, дата) И.О. Фамилия

2025 г.

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

УТВЕРЖДАЮ
Заведующий кафедрой РК6
А.П. Карпенко

«_____» _____ 2025 г.

З А Д А Н И Е
на выполнение курсового проекта

по дисциплине _____Модели и методы анализа проектных решений_____

Студент группы __РК6-83Б___

_____Губанов Даниил Александрович_____
(Фамилия, имя, отчество)

Тема курсового проекта Разработка графического программного обеспечения для визуализации_____
трехмерных объектов _____

Направленность КП (учебная, исследовательская, практическая, производственная, др.) _учебная___
Источник тематики (кафедра, предприятие, НИР) _кафедра _____

График выполнения НИР: 25% к 5 нед., 50% к 11 нед., 75% к 14 нед., 100% к 16 нед.

Задание. Разработать графическое ПО, включающее в себя:_____
1) Отображения трехмерных объектов с возможностью кастомизации_____
2) Система рефлексии пользовательских типов данных_____
3) Сборщик мусора для контроля над пользовательскими данными_____

Оформление курсового проекта:

Расчетно-пояснительная записка на 41 листах формата А4.
Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.):
3 графических листа_____
_____
_____
_____

Дата выдачи задания «6» октября 2024 г.

Руководитель курсовой работы _____ **Витюков Ф.А.**
(Подпись, дата) И.О. Фамилия

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

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

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью данной работы является создание эффективного и гибкого инструмента для визуализации трехмерных объектов, который может быть использован в различных областях

В современном мире информационные технологии играют ключевую роль в различных областях науки и промышленности. Одним из важных направлений является разработка программного обеспечения для визуализации трехмерных объектов, которое находит применение в архитектуре, инженерии, медицине и многих других сферах. Графическое программное обеспечение позволяет наглядно представлять сложные объемные структуры, облегчая анализ и интерпретацию данных.

Реализация графического программного обеспечения для визуализации трехмерных объектов будет осуществляться с использованием библиотеки OpenGL 4.6 и языка программирования C++. OpenGL (Open Graphics Library) представляет собой кроссплатформенный интерфейс для работы с графикой, который широко используется для создания интерактивных и реалистичных визуализаций.

Язык программирования C++ выбран за его производительность и возможности низкоуровневого управления ресурсами, что особенно важно при разработке графических приложений.

Целью данной работы является создание эффективного и гибкого инструмента для визуализации трехмерных объектов, который может быть использован в различных областях.

В ходе выполнения работы будут решены следующие задачи:

1) Изучение основ работы с библиотекой OpenGL и языком программирования C++.
2) Проектирование архитектуры графического приложения.
3) Реализация основных алгоритмов для визуализации трехмерных объектов.
4) Интеграция разработанных компонентов и различных библиотек для

работы с библиотекой OpenGL.

5) Оптимизация производительности и улучшение пользовательского интерфейса.

Результатом данной работы станет готовое графическое программное обеспечение, способное визуализировать трехмерные объекты с использованием современных технологий и подходов. Данное приложение может служить основой для дальнейших исследований и разработок в области компьютерной графики и визуализации данных.

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 разделе 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 высокодетализированных 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.

2. Прототипирование

Для разработки графического программного обеспечения за пример взят игровой движок Unreal Engine 4, в который входит:

1) Работа и интеграция с различными операционными системами, такими как Windows и Linux. (Для поддержки кроссплатформенности взята библиотека GLFW, которая будет описана в других главах)
2) Отображение трехмерных объектов на любой операционной системе. (Для поддержки используется библиотека OpenGL)
3) Удобный и практичный интерфейс для работы с трехмерными объектами на сцене. (Аналогом используется библиотека ImGUI)
4) Загрузка сцены и различных трехмерных объектов (Аналогом используется библиотека Assimp)
5) Система сборщика мусора (Garbage collector)
6) Система рефлексии и сериализации данных (Для реализации используется библиотека clang-c)

На рисунке 1 представлен прототип интерфейса, который будет представлен в данной работе. Пример работы с вкладками взят с Unreal Engine, на рисунке 2 представлен интерфейс самого Unreal Engine. Интерфейс подразумевает наличие таких вкладок:

1) Viewport, в котором отображаются все визуальные объекты, с которыми работает пользователь
2) Content Browser, в котором можно выбрать сцену или объект, который будет отображен во вкладке Viewport
3) World Outliner, в котором отображается дерево объектов расположенных на сцене
4) Object Settings, в котором находятся различные параметризованные значения объекта, с которым работает пользователь
5) Logs – вкладка со всеми системными и пользовательскими дебажными данными.

Типизированные значения, которые будут отображены во вкладке Object

Settings, отображаются за счет работы системы рефлексии, которая в свою очередь получает их из пользовательского кода при помощи определенных макросов. Так же благодаря макросам помеченные переменные будут серилизованы и записаны в определенные файлы для дальнейшей работы с ними после перезапуска приложения.

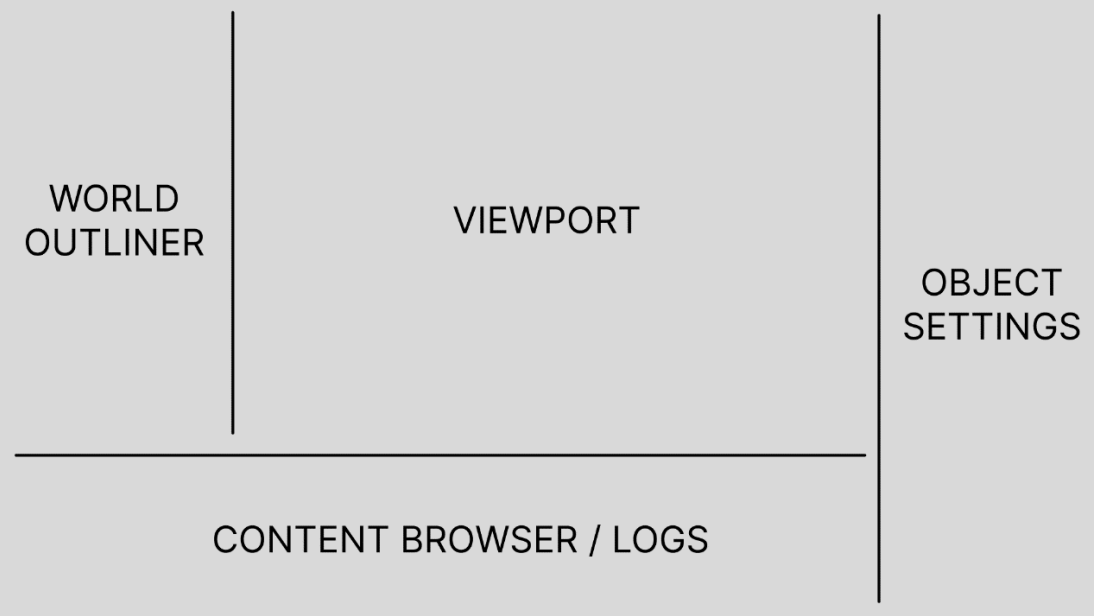

Рисунок 1. Прототип интерфейса графического программного обеспечения

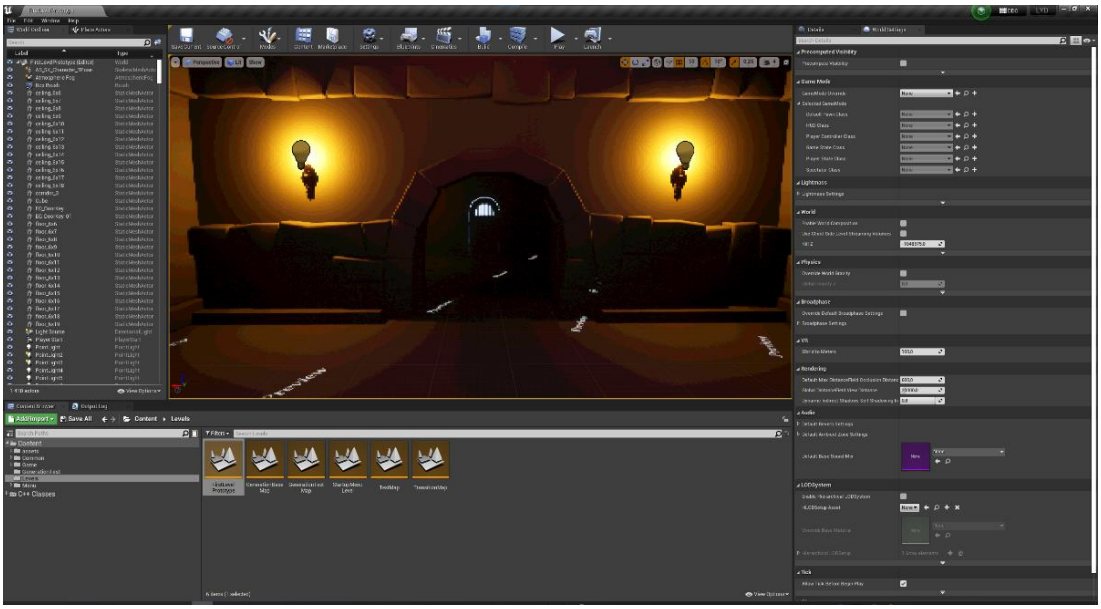

Рисунок 2. Интерфейс Unreal Engine 4

3. Создание графического окна

Прежде чем приступить к созданию сложной графической визуализации, необходимо обеспечить наличие контекста и окна приложения, в котором будет происходить отрисовка графики. Однако следует учитывать, что операции по созданию окна и определению контекста специфичны для каждой операционной системы, и библиотека OpenGL активно абстрагируется от этих процессов. Это означает, что разработчику необходимо самостоятельно реализовать создание окна, определение контекста и обработку пользовательского ввода.

Для отображения окна была использована библиотека GLFW. GLFW представляет собой библиотеку, написанную на языке программирования C, предназначенную для обеспечения OpenGL необходимыми функциональностями для отрисовки контента на экране. Данная библиотека позволяет создавать контекст, задавать параметры окна и обрабатывать пользовательский ввод, что является критически важным для разработки графических приложений.

Однако настройка OpenGL еще не завершена. Остается выполнить дополнительные шаги. Поскольку OpenGL является лишь спецификацией, реализация ложится на разработчиков видеокарт. Вследствие этого, из-за наличия множества реализаций OpenGL, реальное расположение функций OpenGL не доступно на этапе компиляции, и их приходится получать на этапе выполнения, необходимость производить получение адреса для каждой OpenGL функции делает этот процесс просто трудоемким. Однако, к счастью, существуют библиотеки, осуществляющие данную динамическую линковку, и одной из наиболее широко используемых является GLEW.

После интеграции библиотек GLFW и GLEW, следующим этапом было создание базового окна и настройка OpenGL контекста. Для этого были выполнены следующие шаги:

1) Инициализация GLFW: Библиотека GLFW была инициализирована, что позволило создать окно и настроить его параметры, как размер,

заголовок и многое другое, реализация представлена на листинге 1.

2) Создание контекста OpenGL: С использованием функций GLEW был создан контекст OpenGL, который необходим для отрисовки графики, представлено на листинге 1. Это обеспечило возможность использования функций OpenGL в проекте.

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();
```

4. Графический конвейер

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.

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 данного урока 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 OpenGL Shading Language (GLSL), которое используется в данной работе.

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

создание пользовательских шейдеров.

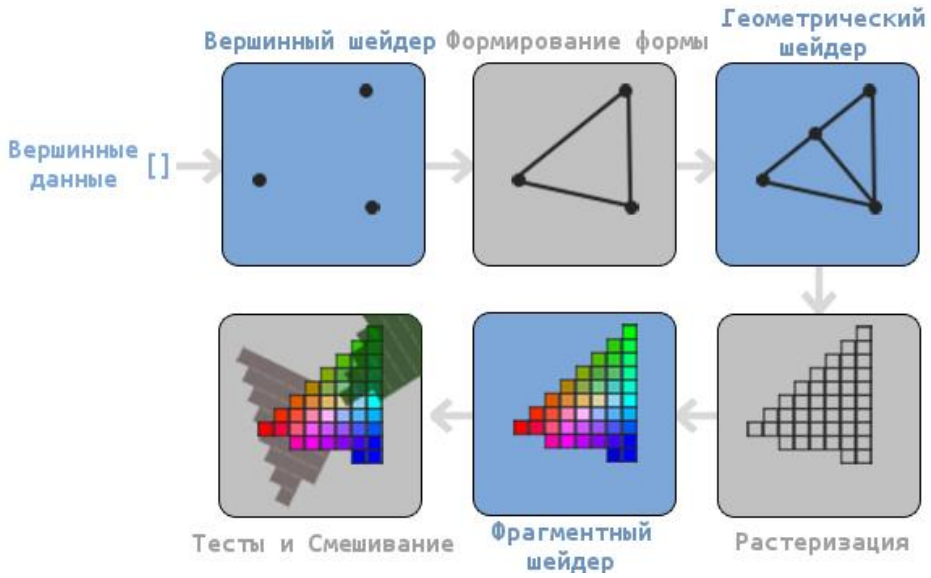

Рисунок 3. Графический конвейер

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 (Normalized Device Coordinates, NDC). 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.

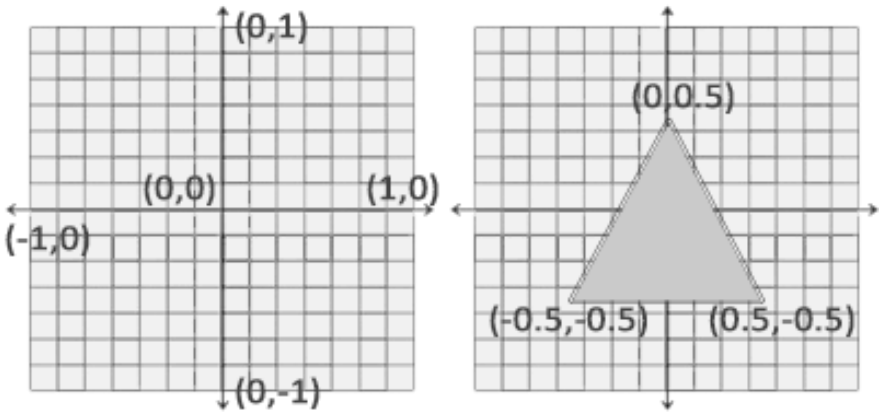

Рисунок 4. Треугольник в контексте NDC

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.

5. Работа с шейдерами

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 (Vertex Buffer Objects, 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);
```

5.1 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

```
#version 330 core

layout (location = 0) in vec3 position;
layout (location = 1) in vec3 color;
layout (location = 2) in vec2 texCoord;

out vec3 ourColor;
out vec2 TexCoord;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    gl_Position = projection * view * model * vec4(position, 1.0f);
    ourColor = color;
    TexCoord = vec2(texCoord.x, 1.0 - texCoord.y);
}
```

5.2 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

```
#version 330 core

in vec3 ourColor;
in vec2 TexCoord;

out vec4 color;

uniform sampler2D ourTexture;

void main()
{
    color = texture(ourTexture, TexCoord);
}
```

5.3 Шейдерная программа

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, 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 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.

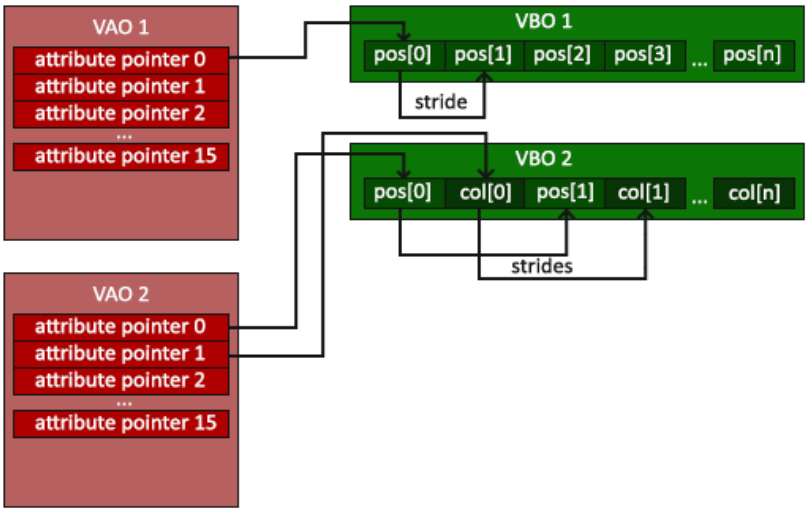

Рисунок 6. Вершинный массив VAO

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

```
GLuint VBO, VAO;

glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);
glBindVertexArray(VAO);

glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
GL_STATIC_DRAW);

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 *
sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(0);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 5 *
sizeof(GLfloat), (GLvoid*)(3 * sizeof(GLfloat)));
glEnableVertexAttribArray(2);

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

6. Отображение текстур

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.

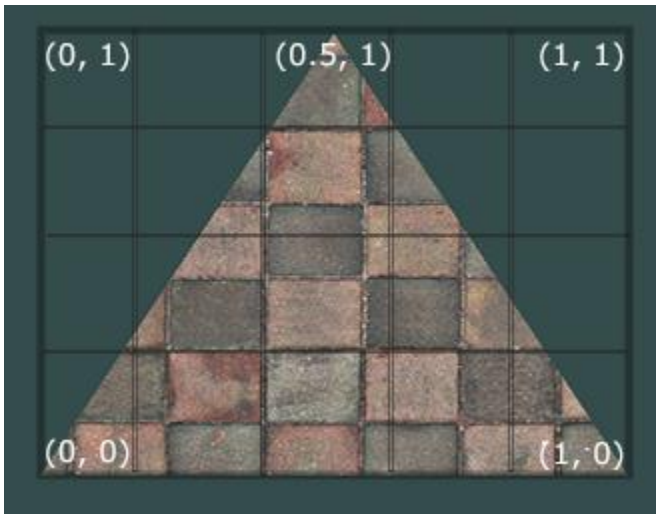

Рисунок 7. Расположение текстурных координат на треугольнике

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 9 наглядно иллюстрирует различия.

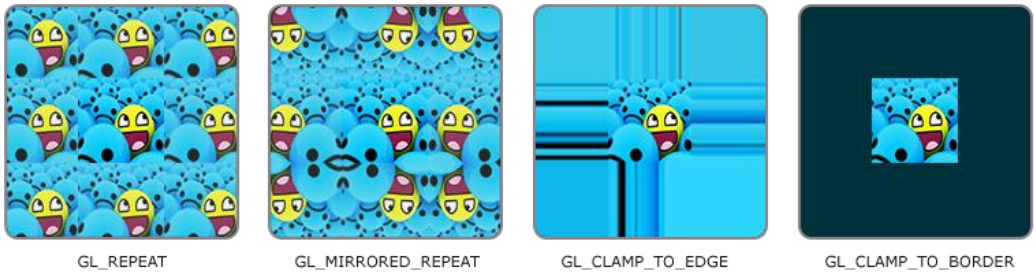

Рисунок 8. Изображения квадрата при различных опциях текстуры

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 ниже:

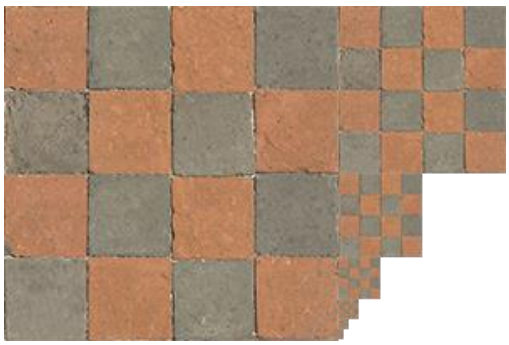

Рисунок 9. Набор мипмап текстур для отдельной текстуры

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

Прежде чем приступить к использованию текстур в нашем приложении, необходимо осуществить их загрузку. Текстурные изображения могут быть представлены в различных форматах, каждый из которых характеризуется своей уникальной структурой и организацией данных. В связи с этим возникает вопрос: каким образом передать изображение в приложение?

Одним из решений является использование удобного формата, например, .PNG, и разработка собственной системы загрузки изображений в виде большого массива байт. Хотя создание собственного загрузчика изображений не является невыполнимой задачей, это может быть довольно трудоемким процессом, особенно если планируется поддержка множества различных форматов файлов.

Альтернативным решением является применение готовых библиотек для загрузки изображений, которые поддерживают множество популярных форматов и выполняют значительную часть работы за нас. Примером такой библиотеки является SOIL (Simple OpenGL Image Library). SOIL поддерживает большинство распространенных форматов изображений и может существенно облегчить процесс загрузки текстур, реализация представлена на листинге 6.

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.

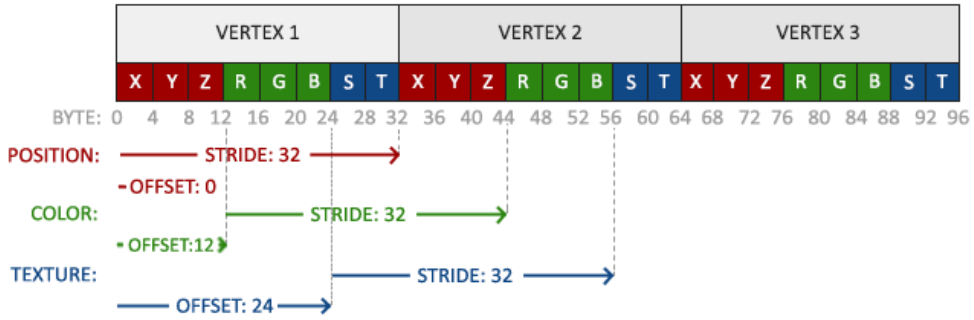

Рисунок 10. Вершинный буфер с координатами в пространстве, цветом и координатами текстуры.

Листинг 6 – Привязка и отрисовка текстуры на объекте

```
20.  GLuint VBO, VAO;
21.  Texture texture(GL_TEXTURE_2D, "Data/Textures/CRATE.BMP",
22.  ColorFormat::RGB);
23.  texture.Bind();
24.
25.  texture.setParameter(GL_TEXTURE_MAG_FILTER, GL_LINEAR);
26.  texture.setParameter(GL_TEXTURE_MIN_FILTER,
27.  GL_LINEAR_MIPMAP_LINEAR);
28.
29.  texture.setParameter(GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
30.  texture.setParameter(GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
31.
32.  GLfloat borderColor[] = { 0.5f, 0.5f, 0.5f, 1.0f };
33.  texture.setParameter(GL_TEXTURE_BORDER_COLOR, borderColor);
34.
35.  texture.UnBind();
```

7. 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.

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:

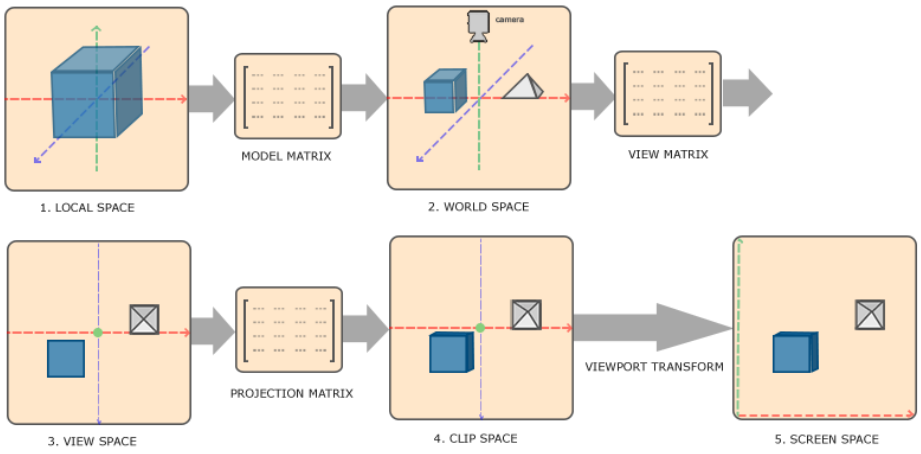

Рисунок 11. Преобразование координат с локального до экранного

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.

7.1 Локальное пространство

Локальное пространство представляет собой координатную систему, которая является локальной для объекта, то есть начинается в той же точке, что

и сам объект. Допустим, вы создали куб в программном пакете моделирования, таком как Blender. Начальная точка вашего куба, вероятно, расположена в (0,0,0), даже если куб в координатах приложения может находиться в другом месте. Возможно, что все созданные вами модели имеют начальную точку (0,0,0). Следовательно, все вершины вашей модели находятся в локальном пространстве, и их координаты являются локальными по отношению к вашему объекту.

Вершины контейнера, который мы использовали, были определены с координатами между -0.5 и 0.5, с начальной точкой отсчета в 0.0, это локальные координаты.

7.2 Мировое пространство

Если мы напрямую импортируем все наши объекты в приложение, они, вероятно, окажутся нагроможденными друг на друга около мировой точки отсчета (0,0,0), что не соответствует нашим ожиданиям. Нам необходимо определить положение каждого объекта для их размещения в более обширном пространстве. Координаты в мировом пространстве соответствуют своему названию: координаты всех ваших вершин относительно игрового мира. Это координатное пространство, в котором вы хотите видеть ваши объекты распределенными в пространстве.

Координаты объекта преобразуются из локального в мировое пространство посредством матрицы модели. Матрица модели представляет собой матрицу, которая перемещает, масштабирует и/или вращает ваш объект для его расположения в мировом пространстве в заданной позиции и ориентации, пример показа на листинге 7.

Листинг 7 – Передача матрицы мировых координат вершинному шейдеру

```
1.  GLuint modelLoc = glGetUniformLocation(shader.getProgram(),
2.  "model");
3.  glm::mat4 model(1.0f);
4.  model = glm::translate(model, glm::vec3(0.0f, 0.0f, -2.0f));
5.  glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
```

7.3 Пространство вида

Пространство Вида также известно как камера OpenGL и иногда называется пространством камеры или наблюдателя. Пространство вида является результатом преобразования мировых координат в координаты, которые выглядят так, как будто пользователь смотрит на них спереди. Таким образом, пространство вида представляет собой пространство, видимое через видоискатель камеры.

Это достигается совокупностью сдвигов и вращений сцены, при которых некоторые объекты располагаются перед камерой. Эти комбинированные преобразования обычно хранятся в матрице вида, которая трансформирует мировые координаты в пространство вида, реализация представлена на листинге 8.

Листинг 8 – Передача матрицы вида вершинному шейдеру

```
1.  GLuint viewLoc = glGetUniformLocation(shader.getProgram(), "view");
2.  view = sceneCamera.getViewMatrix();
3.  glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(view));
```

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”.

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.

Ортографическая проекция

Матрица ортографической проекции задает усеченную пирамиду в форме параллелограмма, которая представляет собой пространство отсечения, как представлено на рисунке 12. 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.

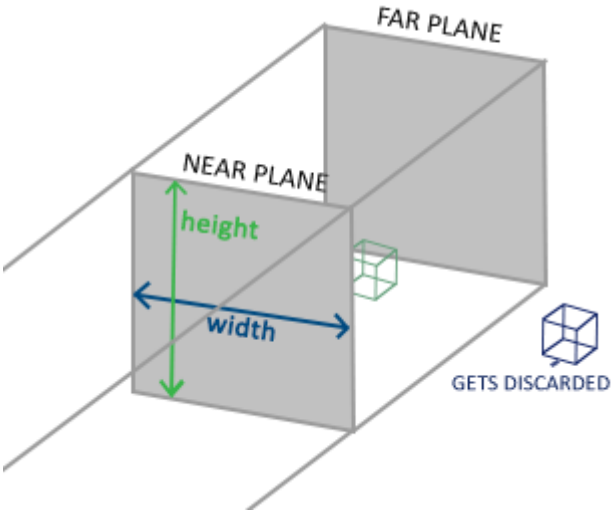

Рисунок 12. Иллюстрация ортографической проекции в пространстве

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.

Для создания матрицы ортографической проекции используется встроенная функция библиотеки GLM.

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.

Перспективная проекция

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.

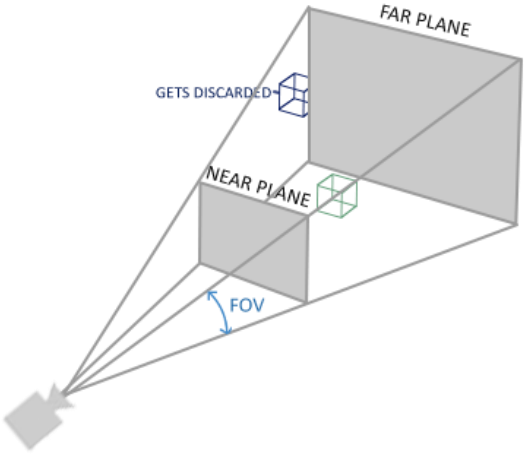

Рисунок 13. Иллюстрация перспективной проекции в пространстве

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.

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 45.0f, но для достижения стиля, подобного Doom, можно задавать и большие значения. 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 0.1f, а дальнее — 100.0f. Все вершины, расположенные между ближней и дальней плоскостью и попадающие в объем усеченной пирамиды, будут визуализированы.

Листинг 9 – Передача матрицы вида вершинному шейдеру

```
1.  GLuint projectionLoc = glGetUniformLocation(shader.getProgram(),
2.  "projection");
3.  projection = glm::perspective(glm::radians(FOV),
4.  (GLfloat)screenWidth/screenHeight, 0.01f, 100.0f);
5.  glUniformMatrix4fv(projectionLoc, 1, GL_FALSE,
6.  glm::value_ptr(projection));
```

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.

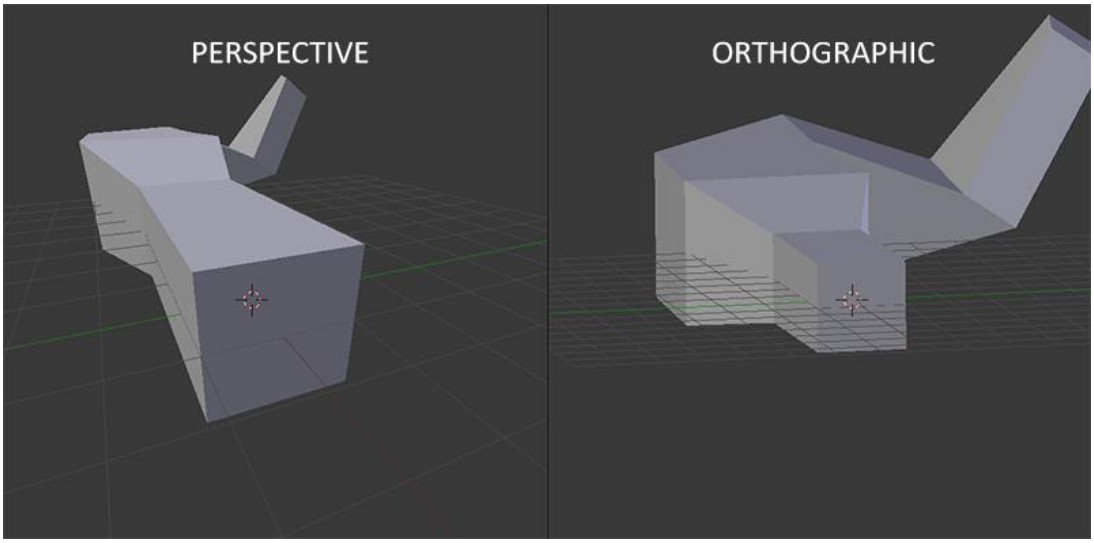

Рисунок 14. Сравнение ортографической и перспективной проекциях

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

координаты пространства отсечения следующим образом:

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

Обратите внимание, что порядок умножения матриц обратный. Полученная координата вершины должна быть присвоена в вершинном шейдере, после чего OpenGL автоматически выполнит перспективное деление и отсечение.

8. Освещение

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:

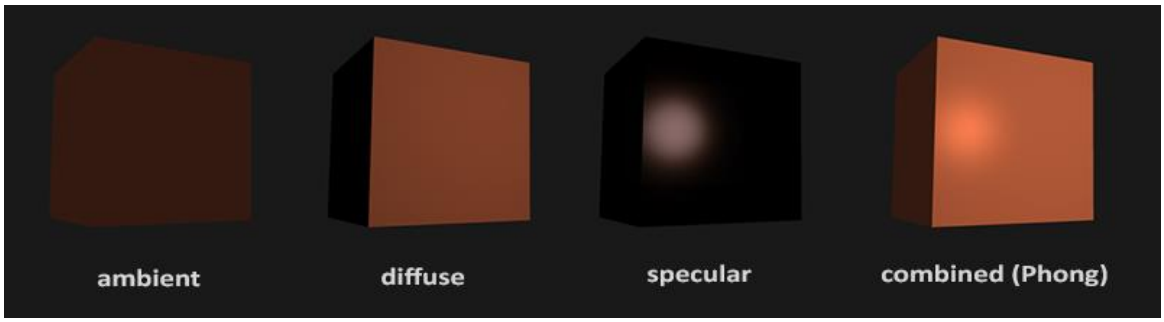

Рисунок 15. Компоненты освещения и их совокупность

8.1 Фоновое освещение

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 разделе 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.

8.2 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 следующее изображение:

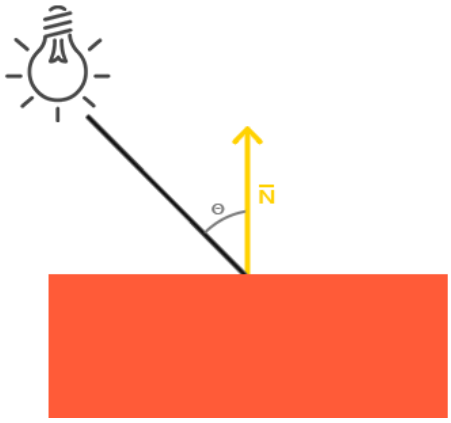

Рисунок 16. Иллюстрация источника света на поверхность

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.

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.

Теперь для каждой вершины имеется вектор нормали, но также необходимы вектора с координатами источника света и фрагмента. Позиция источника света задается одной неизменной переменной, которую мы объявим во фрагментном шейдере как uniform-переменную, как представлено на листинге 10.

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. Для предотвращения отрицательных значений диффузной компоненты используется функция max, которая возвращает наибольшее из переданных значений, гарантируя, что диффузная компонента света (и, следовательно, цвета) никогда не будет меньше 0.0. 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:

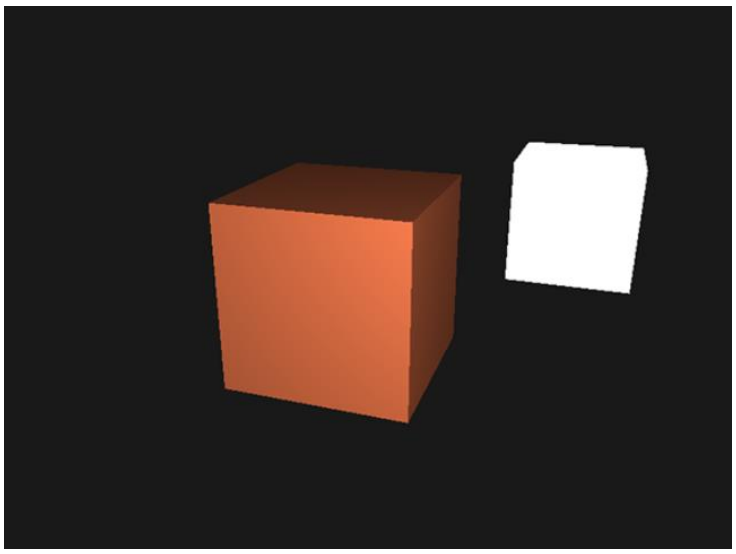

Рисунок 17. Пример отображения фонового и диффузного освещения

8.3 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:

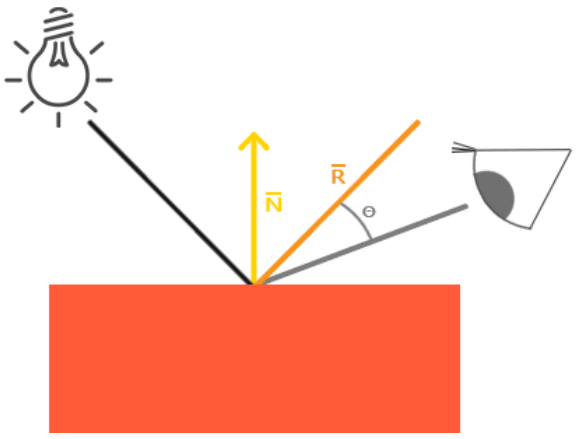

Рисунок 18. Иллюстрация отражения света от источника к приемнику

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:

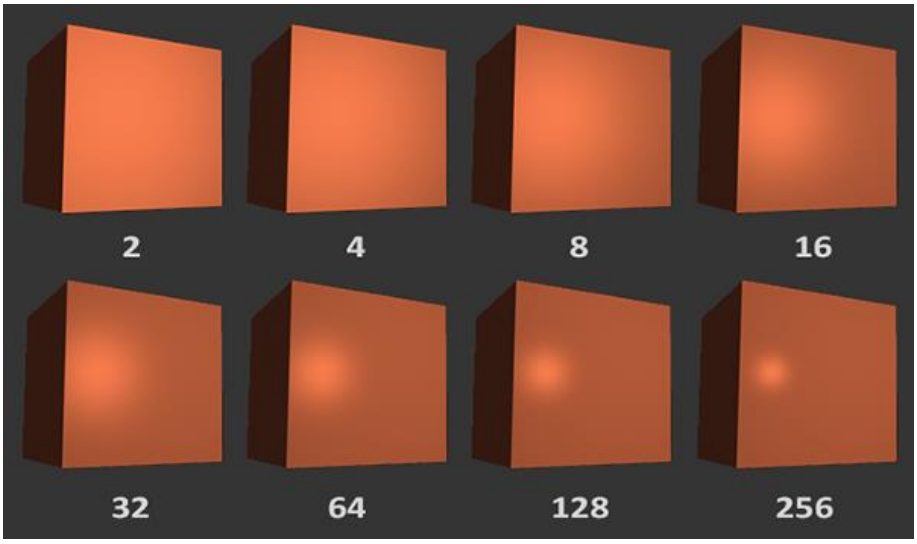

Рисунок 19. Воздействие различных степеней блеска

Теперь все компоненты освещения модели освещения Фонга рассчитаны, результат можно увидеть на рисунке 20 и саму реализацию на листинге 10.

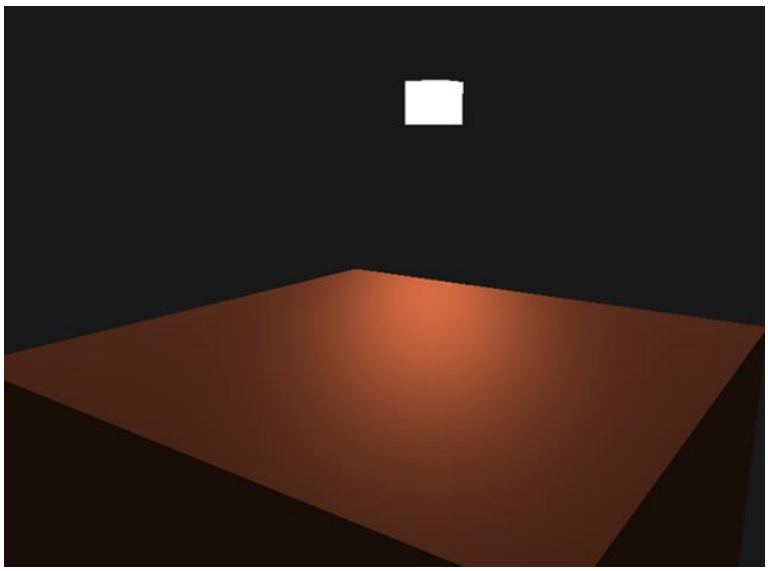

Рисунок 20. Освещение модели по Фонгу

Листинг 10 – 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));

    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32.0f);
    vec3 specular = light.specular * spec *
    vec3(texture(material.specular, TexCoord));

    float distance    = length(light.position - fragPos);
    float attenuation = 1.0 / (light.constant + light.linear *
    distance + light.quadratic * (distance * distance));

    ambient  *= attenuation;
    diffuse  *= attenuation;
    specular *= attenuation;

    return (ambient + diffuse + specular);
}
```

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной работы были достигнуты все поставленные цели и задачи. В результате было создано эффективное и гибкое графическое программное обеспечение для визуализации трехмерных объектов на базе библиотеки OpenGL 4.6 и языка программирования C++.

Основные этапы работы включали изучение основ работы с библиотекой OpenGL и языком программирования C++, проектирование архитектуры графического приложения, реализацию основных алгоритмов для визуализации трехмерных объектов, интеграцию разработанных компонентов и различных библиотек, а также оптимизацию производительности и улучшение пользовательского интерфейса.

Разработанное программное обеспечение позволяет наглядно представлять сложные объемные структуры, что значительно облегчает анализ и интерпретацию данных в таких областях, как архитектура, инженерия, медицина и многих других. Благодаря использованию современных технологий и подходов, данное приложение может служить основой для дальнейших исследований и разработок в области компьютерной графики и визуализации данных.

Таким образом, результаты данной работы имеют высокую практическую значимость и могут быть успешно применены в различных сферах, требующих визуализации трехмерных объектов.

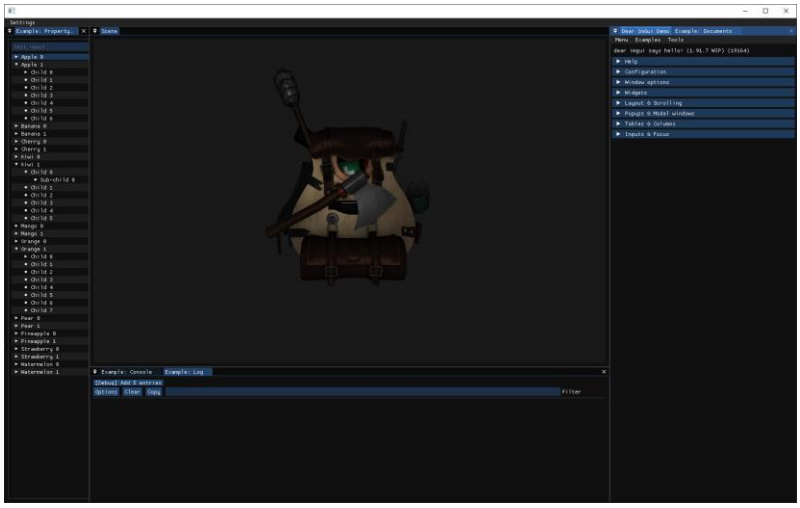

Рисунок 21. Разработанное графическое программное обеспечение

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: 30.11.2024).
2) Joey de Vries OpenGL / Joey de Vries [Электронный ресурс] // Learn OpenGL : [сайт]. — URL: https://learnopengl.com/ (дата обращения: 30.11.2024).
3) Song Ho Ahn (안성호) OpenGL / Song Ho Ahn (안성호) [Электронный ресурс] // OpenGL : [сайт]. — URL: https://www.songho.ca/opengl/ (дата обращения: 30.11.2024).