ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

МОСКОВСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Факультет Информационных технологий
Кафедра Информатики и информационных технологий

направление подготовки
09.03.02 «Информационные системы и технологии»

КУРСОВОЙ ПРОЕКТ

Дисциплина: Технология прикладного программирования

Тема: Разработка системы трёхмерного графического рендеринга

Выполнил: студент группы 231-338

Шаура Илья Максимович
(Фамилия И.О.)

Дата, подпись _____ _____
(Дата) (Подпись)

Проверил: _____
(Фамилия И.О., степень, звание)

Дата, подпись _____ _____
(Дата) (Подпись)

Замечания:

_____

_____

Москва

2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Одними из самых распространённых приложений во всей сфере информационных технологий являются приложения для работы с компьютерной двухмерной и трёхмерной графикой. Трёхмерная графика — это раздел компьютерной графики, посвящённый методам создания изображений или видео путём моделирования объектов в трёх измерениях. 3D-моделирование — это процесс создания трёхмерной модели объекта. Задача 3D-моделирования — разработать зрительный объёмный образ желаемого объекта.

Актуальность данного проекта обусловлена популярностью использования программного обеспечения с трёхмерной графикой. Продуктами, использующими подобные технологии можно считать: игровые движки, сами игры, приложения для трёхмерного моделирования, технологии AR/VR, интерактивные приложения (например, карты), САПР и т.п.

Цель: разработать программное обеспечение с открытым исходным кодом для обработки (рендеринга) и работы с двухмерными и трёхмерными объектами.

Задачи:

- произвести исследование предметной области,
- определить архитектуру работы,
- разработать программные модули,
- произвести отладку продукта.

Объект исследования: пространство информационной разработки компьютерных моделей, игр, многообразной компьютерной графики, а также, программных интерфейсов для разработки приложений, использующих двумерную и трёхмерную компьютерную графику.

Предмет исследования: разрабатываемое приложение для работы с компьютерной графикой и его исходный код.

Проектирование программного обеспечения

1.1 Файловая структура

Структура каталогов позволяет писать модульный код, так как она создаёт чёткую систему разделения ответственностей между различными концепциями, задаваемыми нашими зависимостями. Это помогает искать в репозитории переменные, функции и компоненты. Более того, это помогает хранить в отдельных каталогах минимальный объём содержимого, что, в свою очередь, облегчает работу с ними. В качестве системы контроля версий выбран сервис Git. Сборкой проекта из исходного кода занимается утилита CMake - кроссплатформенная утилита, обладающая возможностями автоматизации сборки программного обеспечения из исходного кода.

Каждый класс, структура, поле, элемент сначала описывается в заголовке (файле с расширением .h), а затем реализовывается в файле определения (.cpp). Каждый класс, структура, поле, элемент хранится в своём окружении (namespace) с названием, идентичным названию каталога, в котором лежат сами файлы. Все файлы, каталоги и подкаталоги с исходным кодом хранятся в общем каталоге «src» (сокр. англ. Sources - «исходники»). Все текстуры, шейдеры и остальные файлы, не связанные напрямую с исходниками, хранятся в каталоге «res» (сокр. англ. Resources - «ресурсы»). Внешние библиотеки, такие, как GLFW, GLM и GLAD, хранятся в каталоге «external» (англ. «внешнее»).

В корневом каталоге находятся файлы:

- «.gitignore» - текстовый файл системы контроля версий Git, который хранит в себе фильтры игнорируемых Git-ом, чаще всего временных файлов и файлов компиляции;
- «CmakeLists.txt» - текстовый файл параметров сборки проекта.

Оставшиеся каталоги (Рисунок 1.1) представляют из себя файлы собранного проекта и файлы параметров для сред разработки (IDE) и Git-а.

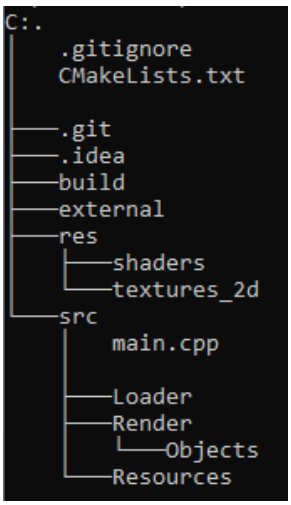

Рисунок 1.1 – Файловая структура репозитория

1.2 Определение функциональных модулей

В качестве интерфейса для взаимодействия с графическим процессором и работы с трёхмерной графикой взят OpenGL. Для него написана дополнительная библиотека GLFW для упрощённого создания окон, контекстов, получения сигналов ввода и событий. Для OpenGL так же создана библиотека GLAD, которая отвечает за взаимодействие с определениями и функциями OpenGL. В основе работы с графическим процессором видеокарты лежит линейная алгебра. Матричные трансформации и векторы – это то, на чём основана вся компьютерная графика. Для OpenGL существует дополнительная библиотека GLM, в которой описаны все необходимые математические функции и структуры из линейной алгебры.

Динамическая обработка изображения состоит из двух этапов:

- Start,
- Update.

При запуске программы, компьютер читает и обрабатывает известную ему информацию. Очень важно загрузить в этап «Start» как можно больше заранее известных статических данных, никак не изменяющихся после его отработки. В нём выполняется:

- инициализация библиотек (включая GLFW, GLM и т. д.),

- объявление структур данных,
- объявление стартовых объектов,
- объявление начальных значений обработки,
- инициализация таймера.

Второй этап динамической обработки изображения начинает работать сразу после инициализации первого и не прекращает работу до завершения программы. Он представляет из себя цикл с условием остановки, по умолчанию определённым GLFW как «!glfwWindowShouldClose(window)», что является флагом того, должно ли окно закрыться при следующем проходе цикла:

Листинг 1.1 – Цикл отрисовки

```
/* Цикл работает до закрытия окна */
   while (!glfwWindowShouldClose(window))
   {
      /* Рендеринг */
      glClear(GL_COLOR_BUFFER_BIT);
      /* Смена буферов */
      glfwSwapBuffers(window);
      /* Запрос и обработка событий */
      glfwPollEvents();
   }
```

Каждый проход цикла представляет из себя один кадр. Именно в следствие этого, количество кадров в секунду (FPS) зависит от вычислительной мощности нагружаемого компьютера. Для фиксации кадров нужен таймер, инициализирующийся на первом этапе. Таймер выполняет функцию ограничителя. Без него, видеокарта и процессор будут работать на предельном уровне даже с простой сценой. Он считает то, сколько времени прошло с последнего выполнения цикла, и, если это время меньше заданного, цикл не начнётся, пока условие не выполнится.

Изначально, ни одна из библиотек и API не предоставляет сложный непроцедурный код и его структуру и оба этапа динамической обработки изображения происходят прямо в точке входа компилятора – функции «main». Решение о структуризации, разбиении и стиле написания кода принимает сам разработчик исходя из субъективных предпочтений и специфики разрабатываемого приложения. Для данной работы выбрано решение придерживаться принципов объектно-ориентированного программирования (ООП) и принципа Don’t Repeat Yourself (DRY).

Разбиение приложения на функциональные модули и объекты обусловлено принципом Don’t Repeat Yourself: уменьшение сложности кода и снижение повторения информации. Объектно-ориентированное программирование, в свою очередь, позволяет структурировать информацию и не допускать путаницы и точно определять взаимодействие одних элементов с другими.

Код программы включет в себя функциональные модули:

- инициализации и первичной настройки подключенных библиотек;
- создания и первичной настройки окна и среды для отображения графических элементов;
- объявления необходимых для процесса обработки, вывода и изменения структуры данных и методы взаимодействия с ними;
- определения структур графических объектов для непосредственного взаимодействия со сценой;
- работы с внешними ресурсами, загрузки текстур, шейдеров.

Разделение функциональных модулей в коде проводится средствами языка C++ с помощью пространств имён (namespace).

Разработка программных модулей

2.1 Реализация графического отображения и взаимодействия с OpenGL

Для взаимодействия с OpenGL, через утилиту CMake были подключены библиотеки GLFW, GLAD и GLM (Рисунок 2.1). Для этого, в папку «external» клонируются их репозитории, они, в свою очередь, содержат свои параметры сборки CMake. В CMakeLists.txt добавляются строки подключения этих библиотек.

Листинг 2.1 - Добавление библиотек GLFW, GLM и GLAD

```
add_subdirectory(external/glfw)
target_link_libraries(${PROJECT_NAME} glfw)
add_subdirectory(external/glad)
target_link_libraries(${PROJECT_NAME} glad)
add_subdirectory(external/glm)
target_link_libraries(${PROJECT_NAME} glm)
```

Поскольку существует много разных версий драйверов OpenGL, расположение большинства его функций неизвестно во время компиляции, и их необходимо запрашивать во время выполнения. Приложение получает местоположение необходимых ему функций и сохраняет их в указателях функций для последующего использования. Для автоматизации этих ёмких и затратных процессов добавлен GLAD.

Для загрузки GLFW и GLAD’a, в пространстве имён Loader был создан класс GLLoad (Листинг 2.1), в который помещаются функции их инициализации и сопутствующие параметры, такие как целевая версия OpenGL и его профиль.

Листинг 2.2 - Инициализация GLFW

```
bool GLLoad::glfw_init() {

        if (!glfwInit()) {
            std::cerr << "glfwInit failed!";
            return false;
        }
        glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    return true;
      }
```

Для работы с окном в пространстве имён Loader создан класс Window. В конструкторе этого класса происходит создание окна посредством функции glfwCreateWindow, а также установка его размера. Функция glfwMakeContextCurrent позволяет определить настоящее окно для взаимодействия с OpenGL.

2.2 Реализация менеджера файловых ресурсов

Архитектура разрабатываемого проекта предполагает размещение всех ресурсов в соответствующей папке «res». В пространстве имён Resources создан класс Resource, отвечающий за загрузку в память различного рода объектов. Внутри определяются методы получения полного пути из относительного и установки пути с необходимыми файлами (Листинг 2.3).

Листинг 2.3 - Методы получения пути ресурсов

```
    std::string ResourceManager::get_file_string(const std::string &relative_file_path) {
        std::ifstream f;
        f.open(m_path_ + "/" + relative_file_path, std::ios::in | std::ios::binary);
        if (!f.is_open()) {
```

```
                std::cerr << "Failed to open file: " << relative_file_path << std::endl;
                return std::string{};
        }
        std::stringstream buffer;
        buffer << f.rdbuf();
        return buffer.str();
}
void ResourceManager::set_executable_path(const std::string &executable_path) {
        m_shader_programs_.emplace("default_setup", nullptr);
        m_sprites_.emplace("default_setup", nullptr);
        m_textures_2d_.emplace("default_setup", nullptr);
        const size_t found = executable_path.find_last_of("/\\");
        m_path_ = executable_path.substr(0, found);
}
```

В пространстве имён Render создан класс Texture2D, и подкласс SubTexture2D. SubTexture2D описывает двумерное координатное представление левого нижнего и правого верхнего края подтекстуры в диапазоне от [0.f; 0.f] до [1.f; 1.f]. Texture2D в свою очередь описывает весь текстурный атлас и хранит в себе информацию об известных подтекстурах и их названиях. В случае использования цельной текстуры вместо текстурного атласа, при объявлении этой текстуры без параметров подтекстур, экземпляр этого класса будет хранить в себе единственную подтекстуру с координатами ([0.f; 0.f]; [1.f; 1.f]).

При объявлении экземпляра класса Texture2D, изображение передаётся в виде массива байт. Для преобразования изображения с расширением jpeg/.png/.bmp/.psd/.tga была выбрана библиотека «stb_image.h». Так как сама библиотека относительно небольшая и состоит из одного файла заголовка, нет смысла включать её в проект в качестве внешнего источника, поэтому она хранится в каталоге Resources рядом с классом ResourceManager.

Принцип работы менеджера ресурсов:

1. В заголовке объявляется приватный тип данных, представляющий собой контейнер, состоящий из строки (названия экземпляра) и указателя на экземпляр необходимого для хранения объекта и объявляется поле этого типа (Листинг 2.4).

Листинг 2.4 - Контейнер класса Texture2D

```
typedef std::map<const std::string, std::shared_ptr<Render::Texture2D>> textures_2d_map;
static textures_2d_map m_textures_2d_;
```

2. В заголовке объявляются методы создания и получения объекта, принимающие все необходимые аргументы для, соответственно, создания и получения экземпляра определённого класса (Листинг 2.5).

Листинг 2.5 - Методы создания и получения

```
static std::shared_ptr<Render::Texture2D>
load_texture_2d(const std::string &texture_name, const std::string &texture_path);
static std::shared_ptr<Render::Texture2D> get_texture_2d(const std::string &texture_name);
```

3. Применяются нужные манипуляции для получения необходимых аргументов определения объекта (в случае с текстурой, с помощью функций библиотеки stb_image изображение преобразуется в массив байт и передаётся в конструктор);

4. Полученный объект с указанным именем добавляется в свой контейнер;

5. При попытке получить определённый объект из контейнера по имени, по нему производится стандартный поиск и возвращается указатель на объект (Листинг 2.6).

Листинг 2.6 - Метод получения

```
std::shared_ptr<Render::Texture2D> ResourceManager::get_texture_2d(const std::string &texture_name) {
    if (texture_name == "default_setup") return nullptr;
    const auto it = m_textures_2d_.find(texture_name);
    if (it == m_textures_2d_.end()) {
```

```
            std::cerr << "Can't find the texture: " << texture_name << std::endl;
            return nullptr;
        }
        return it->second;
}
```

Похожим способом организованы все методы создания, хранения и поиска объектов, различаются только способы создания экземпляра выбранного класса.

2.3 Разработка обрабатываемых элементов программного обеспечения

Для преобразования чисел, байтов и структур в графику, используются шейдеры. Шейдеры — это небольшие программы, работающие на графическом процессоре. Эти программы запускаются для каждого конкретного участка графического конвейера. По сути, шейдеры — это не что иное, как программы, преобразующие входные данные в выходные. Шейдеры являются изолированными, им не разрешено взаимодействовать друг с другом; единственная коммуникация, которую они имеют, осуществляется через их входные и выходные данные. Шейдеры написаны на Си-подобном языке GLSL [1]. GLSL предназначен для работы с графикой и содержит полезные функции, специально предназначенные для работы с векторами и матрицами.

Существует 5 мест в графическом конвейере, куда могут быть встроены шейдеры [2]. Каждый из шейдеров ответственен за свой этап обработки вершинных данных (Рисунок 2.1). Соответственно, шейдеры делятся на типы:

- Вершинный (vertex). Код представлен на листинге 2.7.

Листинг 2.7 - Вершинный шейдер

```
#version 330
layout(location = 0) in vec3 vertex_position;
layout(location = 1) in vec2 texture_coords;
layout(location = 2) in vec2 specular_map_coords;
```

```
out vec2 tex_coords;
out vec2 spec_map_coords;
out vec3 normal;
out vec3 frag_pos;

uniform mat4 mvp;
uniform mat4 model;

void main() {
  tex_coords = texture_coords;
  spec_map_coords = specular_map_coords;
  normal = mat3(transpose(inverse(model))) * vertex_position;
  frag_pos = vec3(model * vec4(vertex_position, 1.0));
  gl_Position = mvp * vec4(vertex_position, 1.0);
}
```

- Фрагментный (fragment). Код представлен на листинге 2.8.

Листинг 2.8 - Фрагментный шейдер

```
#version 330

struct Material{
   sampler2D diffuse;
   sampler2D specular;
   float shininess;
};

struct Light{
   vec3 position;

   vec3 ambient;
   vec3 diffuse;
   vec3 specular;
```

```
    float constant;
    float linear;
    float quadratic;
};

uniform Material material;
uniform Light light;

in vec3 normal;
in vec3 frag_pos;
in vec2 tex_coords;
in vec2 spec_map_coords;

out vec4 frag_color;

uniform vec3 view_pos;

void main(){
    vec3 norm = normalize(normal);
    vec3 light_dir = normalize(light.position - frag_pos);
    vec3 view_dir = normalize(view_pos - frag_pos);
    vec3 reflect_dir = reflect(-light_dir, norm);
    float diff = max(dot(norm, light_dir), 0.0);
    float spec = pow(max(dot(view_dir, reflect_dir), 0.0), material.shininess);
    vec3 ambient  = light.ambient  * vec3(texture(material.diffuse, tex_coords));
    vec3 diffuse  = light.diffuse  * vec3(texture(material.diffuse, tex_coords))       * diff;
    vec3 specular = light.specular * vec3(texture(material.specular, spec_map_coords)) * spec;
    float distance = length(light.position - frag_pos);
    float attenuation = 1.0 / (light.constant + light.linear * distance + light.quadratic * (distance *
distance));
    ambient *= attenuation;
    diffuse *= attenuation;
    specular *= attenuation;
    frag_color = vec4(ambient + diffuse + specular, 1.0);
}
```

- Геометрический (geometry)
- 2 тесселяционных шейдера (tesselation), отвечающие за 2 разных этапа тесселяции

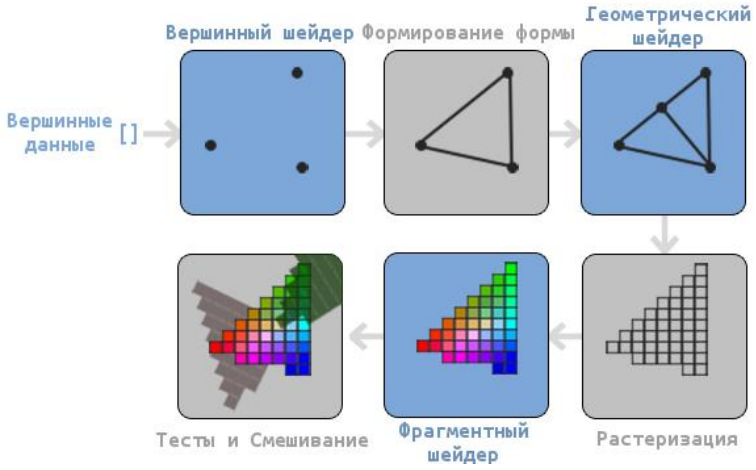

Рисунок 2.1 - Этапы обработки вершинных данных

Геометрический и тессяляционные шейдеры не являются обязательными. Современный OpenGL [3] требует наличия только вершинного и фрагментного шейдера. Хотя существует сценарий, при котором фрагментный шейдер также может отсутствовать. Он в данной работе не рассматривается.

В вершинный шейдер передаются координаты позиций вершин фигур и координаты позиций краёв текстуры. Дополнительно создано поле для передачи в него матриц преобразования. Матрицы преобразования умножаются на позиции вершин и заносятся в глобальную переменную gl_Position.

Координаты текстур передаются дальше по конвейеру, прямиком во фрагментный шейдер, выходным параметром которого является цвет, в который необходимо закрасить пиксель.

Перед использованием шейдеры должны быть слинкованы в шейдерную программу. Для этих целей создан класс ShaderProgram, конструктор которого принимает относительные пути до двух созданных шейдеров, передаёт их в

приватный метод создания шейдеров (Листинг 2.9), после чего связывает их и сохраняет id линкованной программы в приватном поле [4].

Листинг 2.9 - Создание шейдера

```
bool ShaderProgram::create_shader(const std::string &source, const GLenum shader_type, GLuint &shader_id) {
shader_id = glCreateShader(shader_type);
const char *code = source.c_str();
glShaderSource(shader_id, 1, &code, nullptr);
glCompileShader(shader_id);
GLint success;
glGetShaderiv(shader_id, GL_COMPILE_STATUS, &success);
if (!success) {
        GLchar info_log[1024];
        glGetShaderInfoLog(shader_id, 1024, nullptr, info_log);
        std::cerr << "ERROR :: SHADER COMPILE TIME ERROR\n" << info_log << std::endl;
        return false;
}
return true;
}
```

Полученная программа инкапсулируется, и доступ к её использованию происходит через публичный метод класса ShaderProgram «use», которая исполняет публичную функцию GLAD glUseProgram(program_id), указывающую какую шейдерную программу будет использовать OpenGL для обработки.

Перед работой с шейдерами, на предыдущем этапе конвейера, для того чтобы что-то отрисовать, для начала надо передать OpenGL данные о вершинах. OpenGL — это 3D библиотека и поэтому все координаты, которые сообщаются OpenGL находятся в трехмерном пространстве (x, y, z). OpenGL не преобразовывает все переданные ему 3D координаты в 2D пиксели на экране; OpenGL только обрабатывает 3D координаты в определенном

промежутке между -1.0 и 1.0 по всем 3 координатам (x, y и z). Все такие координаты называются координатами, нормализованными под устройство (или просто нормализованными).

После определения вершинных данных требуется передать их в первый этап графического конвейера: в вершинный шейдер. Это делается следующим образом: выделяется память на GPU, куда сохраняются вершинные данные, для OpenGL указывается то, как он должен интерпретировать переданные ему данные. В GPU передаётся информация о количестве переданных нами данных. Затем, вершинный шейдер обработает такое количество вершин, которое ему сообщили.

Управление этой памятью происходит через, так называемые, объекты вершинного буфера (Vertex Buffer Objects (VBO)), которые могут хранить большое количество вершин в памяти GPU [5]. Преимущество использования таких объектов буфера в том, что в видеокарту можно посылать большое количество наборов данных за один раз, без необходимости отправлять по одной вершине за раз. Отправка данных с CPU на GPU довольно медленная, поэтому, за один раз, отправляется как можно большое данных. Как только данные окажутся в GPU, вершинный шейдер получит их практически мгновенно.

Разработаны классы: VertexBuffer и VertexBufferLayout. VertexBufferLayout является вспомогательным классом, необходимым для определения размеров буфера и параметра нормализации координат в этом буфере [6]. Стандартный формат вершинного буфера представлен на рисунке 2.2.

Рисунок 2.2 - Формат вершинного буфера

Он используется для отрисовки полигона, и описан с необходимыми характеристиками:

- Информация о позиции хранится в 32 битном (4 байта) значении с плавающей точкой;
- Каждая позиция формируется из 3 значений;
- Не существует никакого разделителя между наборами из 3 значений. Такой буфер называется плотно упакованным;
- Первое значение в переданных данных — это начало буфера.

Алгоритм отрисовки объекта в OpenGL с использованием Vertex Buffer будет выглядеть так:

1. Копирование массива с вершинами в буфер OpenGL;
2. Установка указателей на вершинные атрибуты;
3. Использование шейдерной программы;
4. Вызов функции отрисовки объекта.

Подобный затратный процесс должен повторяться при каждой отрисовке объекта. Для оптимизации, существует способ хранения всех этих состояний и привязки к ним - Vertex Array Object. Объект вершинного массива (VAO) может быть также привязан, как и VBO, и после этого, все последующие вызовы вершинных атрибутов будут храниться в VAO. Преимущество этого метода в том, что разработчику требуется настроить атрибуты лишь единожды, а все последующие разы будет использована конфигурация VAO. Также такой метод упрощает смену вершинных данных и конфигураций атрибутов простым привязыванием различных VAO (Рисунок 2.3). Более того, Core OpenGL требует, использования VAO для того чтобы OpenGL знал, как работать с входными вершинами. Если не указывать VAO, OpenGL может отказаться отрисовывать что-либо. В пространство имён Render добавляется класс VertexArray [7].

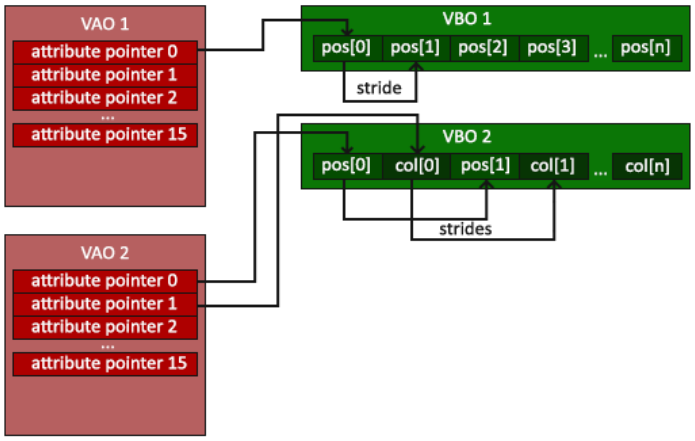

Рисунок 2.3 - Принцип работы VAO

Использование VAO происходит посредством правильного исполнения функции GLAD glBindVertexArray. Необходимо предворительно настроить/привязать требуемые VBO и указатели на атрибуты, а в конце отвязать VAO для последующего использования. Каждый раз, когда цикл вызывает объект для его отрисовки, VAO просто привязывается с требуемыми настройками перед отрисовкой объекта [8].

Алгоритм отрисовки объекта после добавления функционала VAO [9] выглядит так:

1. Привязка VAO;
2. Копирования массива вершин в буфер для OpenGL;
3. Установка указателей на вершинные атрибуты;
4. Отвязка VAO;
5. Вызов функции отрисовки объекта.

Последний этап оптимизации алгоритма заключается в решении проблемы повторяющихся вершин, ведь для отрисовки, например, квадрата из двух полигонов, необходимо указывать оба полигона, и, несмотря на то что у квадрата 4 стороны, при настоящем способе задания значений, в массиве с вершинами будут лежать 6 координат (каждый угол каждого треугольника-полигона). В пространстве имён Render создан класс IndexBuffer. IndexBuffer выполняет функции ElementBufferObject’a (EBO). EBO — это буфер, вроде

VBO, но он хранит индексы, которые OpenGL использует, чтобы решить какую вершину отрисовать. Это называется отрисовка по индексам (indexed drawing) и является решением вышеуказанной проблемы (Рисунок 2.4).

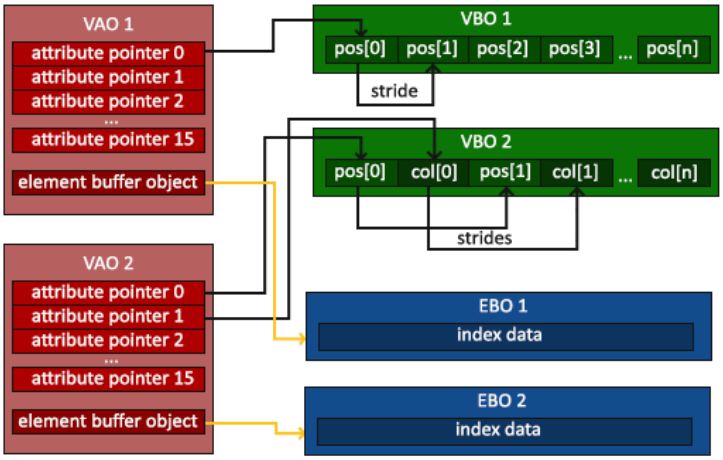

Рисунок 2.4- Схема работы EBO

Алгоритм отрисовки объекта после добавления функционала EBO выглядит так:

1. Привязка VAO;
2. Копирования массива вершин в буфер для OpenGL;
3. Копирования массива индексов в буфер для OpenGL;
4. Установка указателей на вершинные атрибуты;
5. Отвязка VAO (не EBO);
6. Вызов функции отрисовки объекта.

Данный алгоритм [10] отрисовки используется для всех созданных объектов.

2.4 Определение сцены и камеры

Сценой является совокупность объектов, материалов и некоторых настроек программы [11].

Камера – это особый объект сцены, который является векторным представлением точки в пространстве, откуда происходит обработка сцены [12].

Для описания этих понятий в пространстве имён Render созданы соответствующие классы Scene и Camera.

При попытке создать экземпляр, конструктор сцены требует передать указатель на шейдерную программу, которую будет использовать данная сцена [13].

Главный метод класса Scene render() принимает в качестве аргументов указатели на окно, в котором будет обрабатываться сцена, используемую камеру и массив объектов, обрабатываемых сценой.

Внутри метода находится цикл, который проходится по каждому переданному объекту, применяет для него используемый шейдер, выполняет матричные преобразования и вызывает функцию отрисовки объекта [14].

Разные трансформации сцена может приобрести на разных этапах её рендеринга, для этого, в пространстве меняются системы координат. Глобальное пространство состоит из 5 систем координат (Рисунок 2.5):

1. Локальные координаты — это координаты объекта относительно его локального источника; это координаты, с которых начинается объект;
2. Следующие координаты являются сборником всех локальных координат в координаты мирового пространства;
3. Затем, с помощью матрицы обзора, мировые координаты преобразуются в координаты пространства обзора таким образом, чтобы каждая координата была видна с точки зрения камеры;
4. После того, как координаты находятся в пространстве обзора, они проецируются на координаты отсечения. Координаты отсечения обрабатываются в диапазоне -1.0 и 1.0 и определяют, какие вершины окажутся на экране. Использование матрицы проекции на координатах отсечения может добавить перспективу, если используется перспективная проекция.
5. И, наконец, координаты отсечения преобразуются в координаты экрана. Функция ViewportTransform преобразует координаты из -1.0 и 1.0 в диапазон координат, соответствующий размеру окна. Затем, полученные

координаты отправляются в растеризатор, чтобы превратиться во фрагменты для фрагментного шейдера [15].

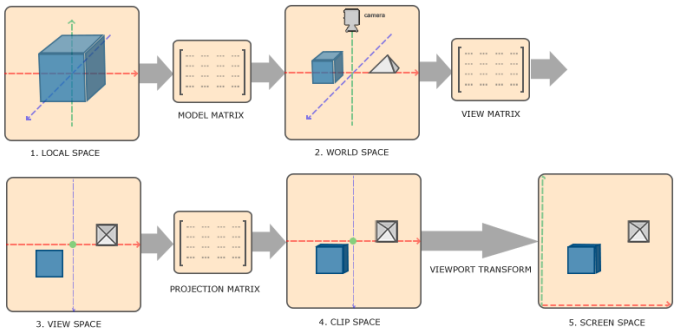

Рисунок 2.5 – Преобразование систем координат

Исходя из этого, чтобы взаимодействовать с объектом, необходимо изменять модельную матрицу, с камерой – матрицу обзора, с перспективой/приближением/кадром – проекционную матрицу [16].

Поведение камеры напрямую связано с матрицей обзора и контролируется движением мыши, прокруткой колеса мыши и нажатием клавиш W, A, S, D. Обработчики этих событий находятся в классе Camera и передаются в GLFW [17].

Листинг 2.10 - Метод обновления векторов направленности камеры

```
void Camera::updateCameraVectors() {
        glm::vec3 front;
        front.x = cos(glm::radians(Yaw)) * cos(glm::radians(Pitch));
        front.y = sin(glm::radians(Pitch));
        front.z = sin(glm::radians(Yaw)) * cos(glm::radians(Pitch));
        Front = glm::normalize(front);
        Right = glm::normalize(glm::cross(Front, WorldUp));
        Up = glm::normalize(glm::cross(Right, Front));
    }
```

Поворот камеры производится по углам Эйлера. В трёхмерном пространстве существует 3 угла Эйлера: pitch – наклон (x), yaw – отклонение (y) и roll – крен (z). Так как крутить камеру сразу вокруг трёх координатам –

нецелесообразно, при повороте мышью учитываются только повороты по осям x и y.

Передвижение камерой производится клавишами W, A, S, D и осуществляется изменением позиции камеры в пространстве [18].

Прокруткой колеса мыши производится приближение и отдаление объектов и осуществляется увеличением и уменьшением угла обзора камеры, указываемом в проекционной матрице.

Сама матрица обзора получается с помощью функции GLM lookAt, в которую передаются параметры позиции камеры, вектора её направления и вектора верха сцены.

Далее, с помощью функции GLM perspective строится проекционная матрица, которая принимает в качестве аргументов: поле зрения, соотношение сторон, минимально возможное приближение и максимально возможное приближение [19].

В конце, формируется общая матрица, посредством умножения всех предыдущих матриц. Поле для этой матрицы находится в вершинном шейдере, и с помощью функций класса ShaderProgram подставляется в него.

2.5 Добавление базовых геометрических фигур

Для всех объектов, отрисовываемых на сцене, создан родительский класс NullObject с приватными полями для описания размера объекта, его поворта и позиции. Инкапсулированные поля можно изменять и считывать с помощью созданных методов получения и задания. Также, создаётся виртуальный метод draw, отвечающий за отрисовку объекта.

В качестве самого простого объекта взят двухмерный прямоугольник, состоящий из 2-х полигонов. Такой объект называется спрайтом. Для этого, в пространстве имён Objects создан класс Sprite, наследуемый от родительского класса NullObject. Объявлены приватные поля для работы VBO, VAO и EBO. Внутри переопределён метод draw, в котором выставляется текущий VAO, а используемая текстура заносится в буфер текстур, после чего, функция

glDrawElements вырисовывает заданный в конструкторе данного объекта VAO, учитывая все его элементы.

Точно так же создаётся трёхмерный куб, меняются только параметры создания VBO, координаты вершин, текстур и их индексы [20].

Сфера создаётся динамически, ведь для относительной гладкости поверхности шара пришлось бы создавать 3 (значения вершины) * 3 (вершины) * 2 (треугольника) * 50^2 (параллелей и меридиан) = 45000 значений для вершин, плюс столько же значений для текстур, плюс 67500 значений индексов. Итого 157500 значений, вписанных вручную. Расчёт значений происходит динамически на центральном процессоре. Динамический подход, в данной ситуации позволяет выставлять разную степень гладкости сферы, но при этом расходует больше ресурсов, чем предварительно просчитанная модель. Созданные фигуры были размещены на сцене. На них были наложены текстуры (Рисунок 2.6).

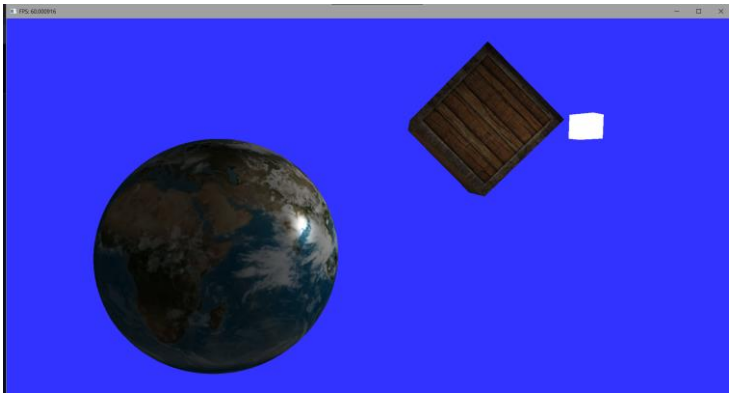

Рисунок 2.6 – Итоговая сцена

2.6 Создание функций взаимодействия с трёхмерными объектами

В классе NullObject изначально были созданы поля и методы взаимодействия с параметрами трансформации модели. При вызове любого класса, наследующегося от класса NullObject можно изменить и прочитать эти параметры. Чтобы эти изменения были зафиксированы, в методе рендера сцены для модельной матрицы производятся трансформации величины (scale), поворота (rotation) и позиции (translate), величины для которых были взяты из обрабатываемого в данный момент объекта.

Одна из функций по взаимодействию пользователя с окружением – смена графического отображения объектов на отображение исключительно сетки полигонов (mesh), из которых они состоят (Рисунок 2.7). Переключение происходит по нажатию на клавиатуре на букву «m».

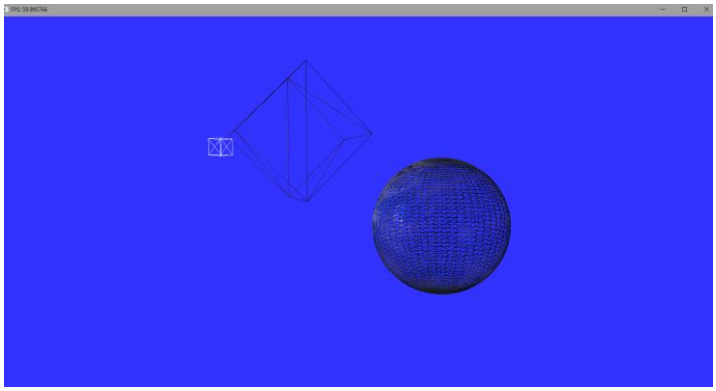

Рисунок 2.7 – Отображение сетки полигонов

Второй альтернативный режим отображения сцены (Рисунок 2.8) – изменение глубины отрисовки (depth). Позволяет использовать альтернативные расчёты глубины, а именно – отрисовку только внешних поверхностей объектов, игнорируя те, на которые рассчитываются как внутренние.

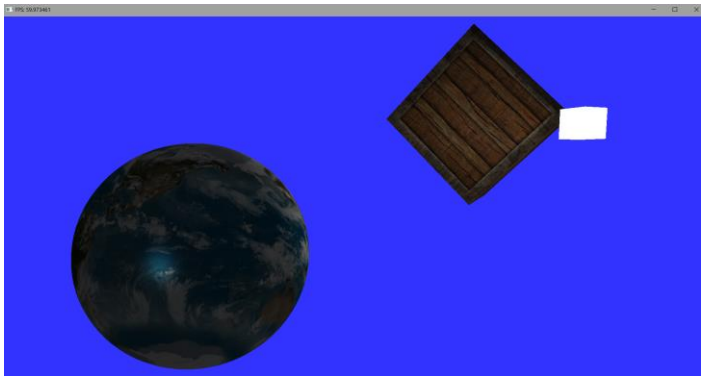

Рисунок 2.8 – Отключение отрисовки внутренней поверхности объектов

Этот режим отрисовки полезен для оптимизации отрисовки сцены. Он может не изменять внешний вид сцены, но при этом могут игнорироваться все «ненужные» расчёты.

По окончанию этой главы был рассмотрен весь функционал и полный принцип работы приложения.

ЗАКЛЮЧЕНИЕ

Курсовая работа оформлена по необходимым правилам и, помимо прочего, содержит: введение, две главы и заключение.

В первой главе проделана проектировочная работа. Было проведено проектирование архитектуры программного обеспечения.

В ходе курсовой работы производилось изучение принципов работы компьютерной графики, была проанализирована всевозможная литература, связанная с разработкой программного обеспечения для взаимодействия с компьютерной графикой при помощи API OpenGL.

Была изучена литература по программированию на языке C++.

Во второй главе были изложены принципы и алгоритмы работы программного обеспечения.

Программный код был написан, придерживаясь принципов объектно-ориентированного программирования и правила Don’t Repeat Yourself (DRY).

Все классы были разработаны для максимально безопасного взаимодействия с памятью, на каждом этапе разработки производилась отладка в целях устранить утечки памяти и ускорить работу приложения.

Сборка проекта осуществлялась посредством кроссплатформенной утилиты CMake.

Все изменения, вносимые в проект, фиксировались и хранились в системе контроля версий Git. Все исходные файлы хранятся и будут храниться в открытом доступе.

В результате, было разработано программное обеспечение с открытым исходным кодом для обработки (рендеринга) и работы с двухмерными и трёхмерными объектами в трёхмерном пространстве.

Проект планируется развивать, дорабатывать и поддерживать.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Шрайнер, Д. OpenGL Redbook / OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V, 9th Edition 2017 – 802 с.
2. Страуструп Бьёрн, Программирование. Принципы и практика с использованием C++ (2е издание) 2018 – 1328 с.
3. Джосаттис Николаи М., Грегор Дуглас, Шаблоны C++. Справочник разработчика (2е издание) 2018 – 848 с.
4. Верма Рахул Девендрович, Введение в OpenGL (Отдельное издание) 2017 – 304 с.
5. Gabor Szauer, Hands-On C++ Game Animation Programming. Learn modern animation techniques from theory to implementation with C++ and OpenGL 2020, 368 с.
6. Sergey Kosarevsky, Viktor Latypov, 3D Graphics Rendering Cookbook. A comprehensive guide to exploring rendering algorithms in modern OpenGL and Vulkan 2021 – 670 с.
7. Боресков Алексей Викторович, Программирование компьютерной графики. Современный OpenGL, 2019 – 372 с.
8. David Wolff, OpenGL 4 Shading Language Cookbook - Third Edition. Build high-quality, real-time 3D graphics with OpenGL 4.6, GLSL 4.6 and C++17, 2018 – 472 с.
9. Страуструп Б., Язык программирования C++. 4-е изд. 2022 – 1216с.
10. Лафоре Роберт, Объектно-ориентированное программирование в C++. Классика Computer Science, 2022, 928 с.
11. Чукич И., Функциональное программирование на C++, 2020, 360с.
12. Мейерс Скотт, Эффективное использование C++. 55 верных способов улучшить структуру и код ваших программ, 2017 – 300 с.
13. Пикус Ф., идиомы и паттерны проектирования в современном C++, 2020 – 452 с.

14. Пол Дж. Дейтел, Харви Дейтел, Как программировать на C++ 2021 – 1032 с.

15. Конова Елена Александровна, Поллак Галина Андреевна. Алгоритмы и программы. Язык C++. Учебное пособие. Гриф УМО вузов РФ, 2021 – 132 с.

16. Васильев А.Н., Самоучитель C++ с задачами и примерами, 2018 - 480с.

17. https://antongerdelan.net/opengl/, дата обращения – 11.03.2024

18. https://learnopengl.com/, дата обращения – 10.03.2024

19. https://www.glfw.org/docs/latest/quick.html, дата обращения – 09.03.2024

20. https://github.com/glfw/glfw/wiki, дата обращения – 11.03.2024