МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего профессионального образования

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ

ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Программирование компьютерной графики с использованием OpenGL

Составитель К.А. Брынцев, А.Ю. Дёмин

Издательство

Томского политехнического университета

2023

Содержание

1. Введение

1.1. Компьютерная графика

Компьютерная графика – это область информационных технологий, связанная с созданием, обработкой и визуализацией графических изображений на компьютере. Она занимается представлением и манипуляцией графических объектов, а также их отображением на экране. В компьютерной графике используются различные алгоритмы и методы для создания и редактирования графических объектов. Это включает в себя технологии моделирования объектов и сцен, освещения, текстурирования, анимации, рендеринга и многое другое. Компьютерные графические программы работают с геометрическими моделями объектов, оперируя вершинами, гранями и текстурными координатами.

Для создания и визуализации компьютерной графики используются специальные программные пакеты и библиотеки, такие как OpenGL, DirectX, WebGL и другие. Они предоставляют программистам возможности для работы с графическими возможностями компьютера, обеспечивая высокую производительность и эффективное использование аппаратного обеспечения.

В этом методическом пособии будут использоваться библиотеки OpenGL, GLAD, GLFW. Работа с этими библиотеками будет производиться в Visual Studio Code на OS Windows 11.

OpenGL (открытая графическая библиотека) — это графический API с открытым исходным кодом (интерфейс прикладного программирования), который позволяет разработчикам создавать высокопроизводительные графические и интерактивные 3D-приложения. Он предоставляет набор функций для рендеринга 2D- и 3D-графики, а также обработки ввода и других аспектов графического программирования.

Glad (GL Loader) — популярная библиотека с открытым исходным кодом, которая упрощает процесс загрузки и использования функций OpenGL. Он генерирует необходимый код для загрузки указателей функций OpenGL во время выполнения, устраняя необходимость загрузки функций вручную.

GLFW (Graphics Library Framework) — еще одна широко используемая библиотека с открытым исходным кодом, которая предоставляет простой и кроссплатформенный API для создания окон, обработки пользовательского ввода и управления контекстами OpenGL. Он служит оболочкой для специфичных для платформы API окон и ввода, например, предоставляемых Windows, macOS и Linux. Помимо управления окнами и вводом, GLFW также поддерживает такие функции, как настройка нескольких мониторов и управление временем.

Комбинация OpenGL, Glad и GLFW предоставляет разработчикам мощный и гибкий набор инструментов для создания современных графических приложений. OpenGL определяет основные возможности рендеринга, а Glad упрощает процесс загрузки и доступа к функциям OpenGL. GLFW, с другой стороны, занимается созданием окон, управлением вводом и другими задачами, специфичными для платформы. Вместе они предлагают комплексное

решение для графического программирования, позволяющее разработчикам сосредоточиться на создании визуально привлекательных и интерактивных приложений.

1.2. Подготовка библиотек glad и GLFW

Создадим рабочую директорию. В ней создадим две другие директории, одну для графических библиотек, а другую для исходников программы.

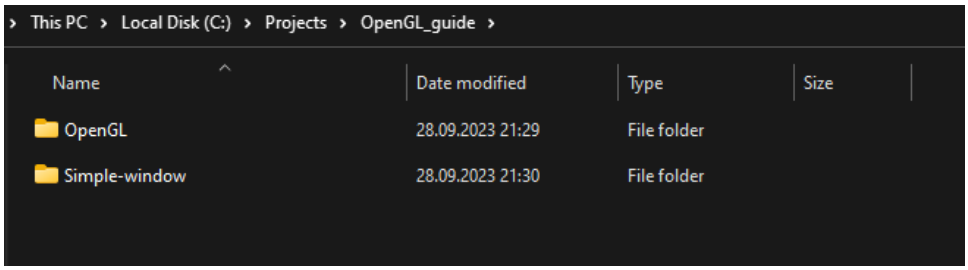

Рис. 1(рабочие директории)

Теперь необходимо скачать библиотеки для работы с OpenGL. Библиотеку glad можно скачать с официального сайта https://glad.dav1d.de. Выберите язык C++, специализацию OpenGL, профиль Compatibility и gl версии 4.6. Нажмите Generate и скачайте исходники.

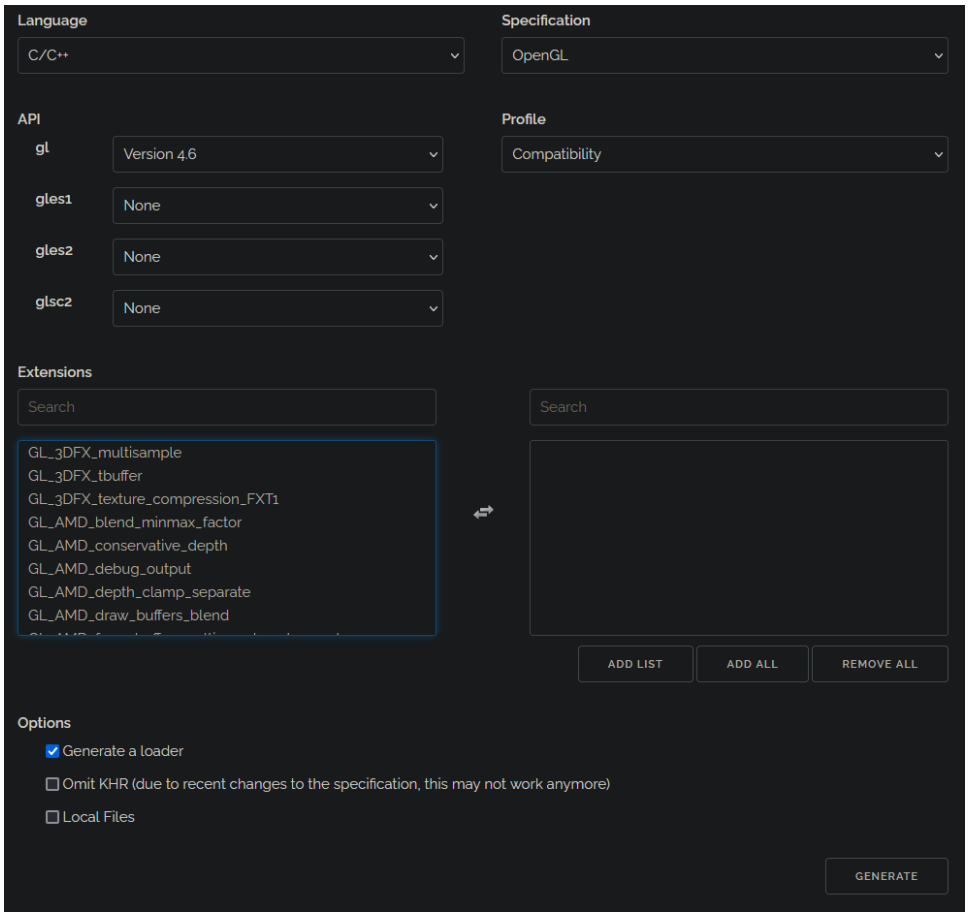

Рис. 2(генерация библиотеки glad)

Создайте директорию *glad* и поместите туда скаченные файлы.

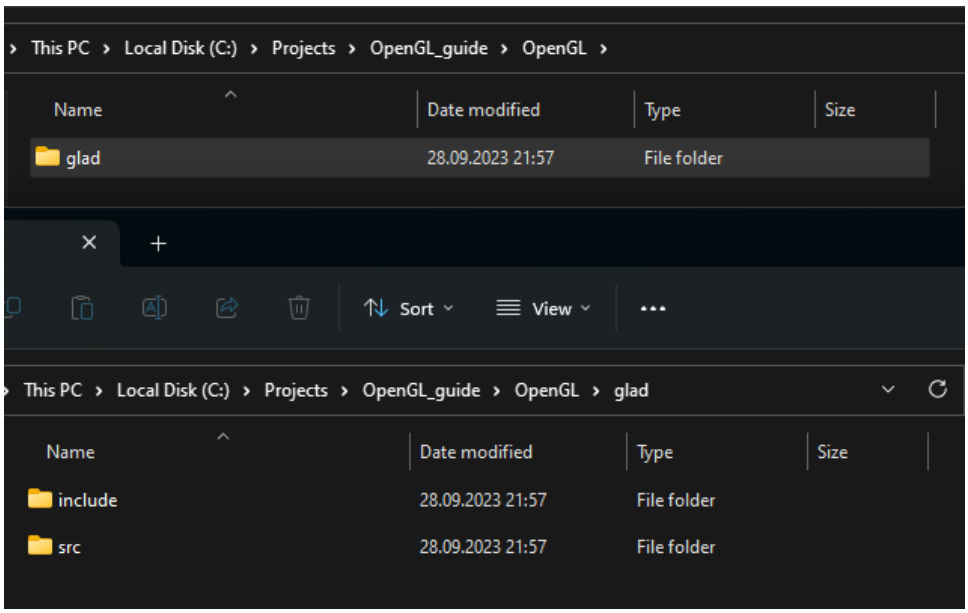

Рис. 3(установка библиотеки glad)

Далее скачаем GLFW, но не в виде исходных файлов, а в виде скомпилированной библиотеки. Сделать это можно с официального сайта https://www.glfw.org/download.html.

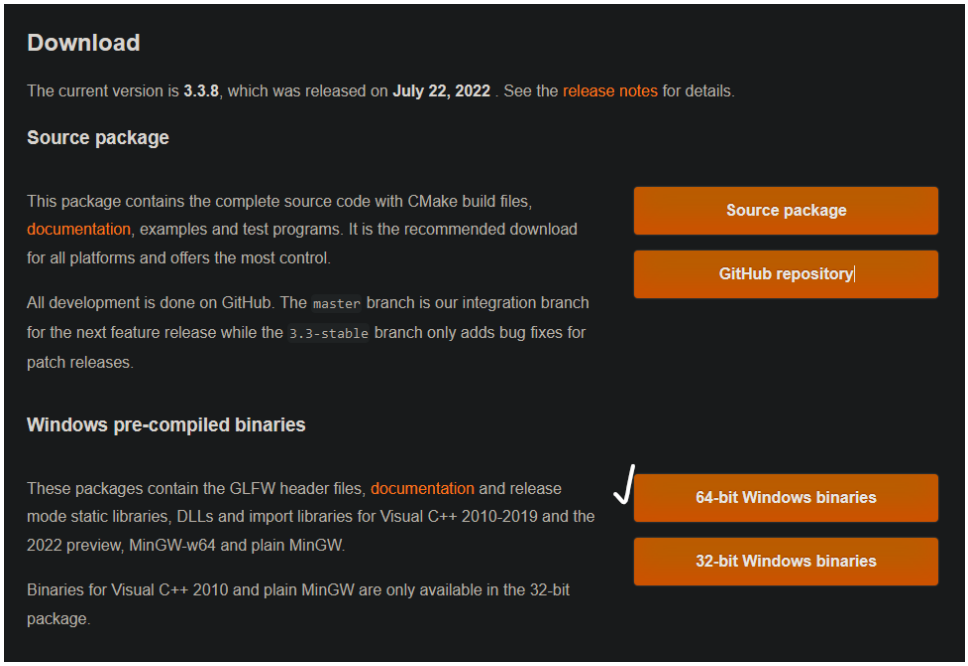

Рис.4(скачивание скомпилированной библиотеки GLFW)

Разархивируйте скаченные файлы в директорию *GLFW* рядом с директорией *glad*.

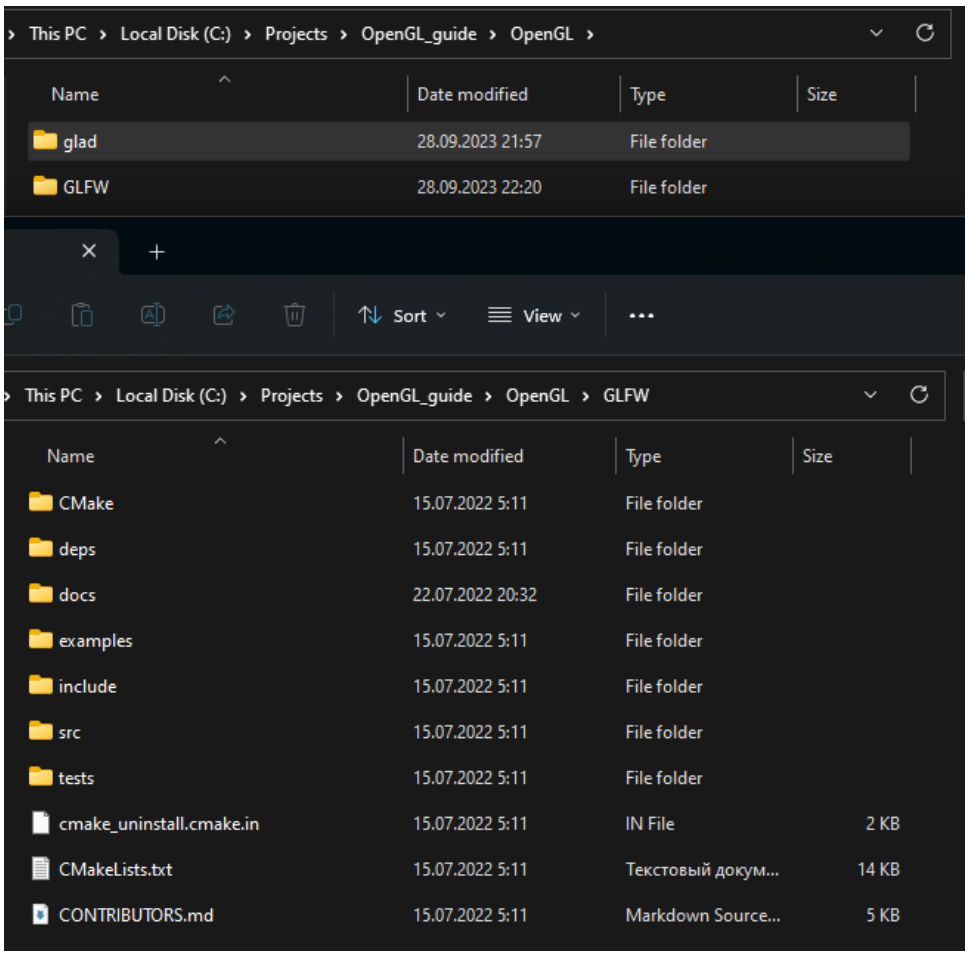

Рис.4 (установка библиотеки GLFW)

1.3. Настройка Visual Studio Code

Установите расширение для C++ и перезагрузите VS Code.

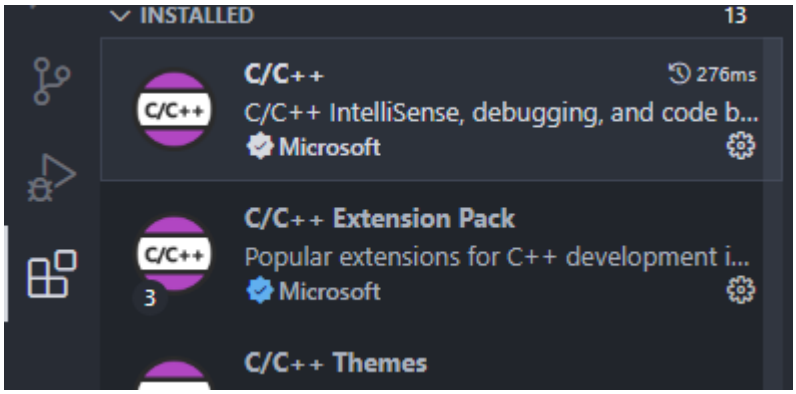

Рис.5 (расширение для C++)

Откройте ранее созданную директорию. В ней создайте 2 новых директории *src* и *include*.

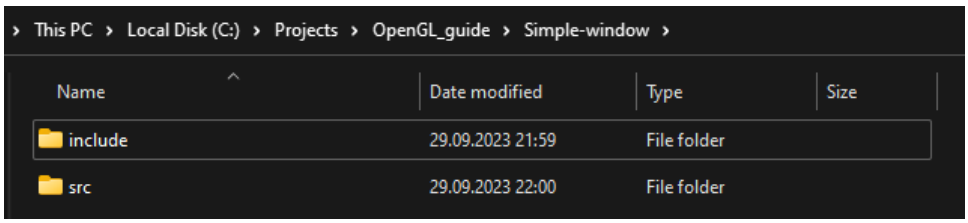

Рис.6 (базовая структура проекта)

Откройте эту директорию в Visual Studio Code.

Нажмите сочетание клавиш *Ctrl+Shift+P* и выберите *“C/C++:Select a Configuration”*. Затем выберите *“Изменить конфигурфцию (JSON)”*. В результате получиться следующий файл.

```
{
    "configurations": [
        {
            "name": "Win32",
            "includePath": [
                "${workspaceFolder}/**"
            ],
            "defines": [
                "_DEBUG",
                "UNICODE",
                "_UNICODE"
            ]
        }
    ],
    "version": 4
}
```

Измените “includePath” следующим образом:

```
"includePath": [
      "${workspaceFolder}/include",
      "${workspaceFolder}/../OpenGL/GLFW/include",
      "${workspaceFolder}/../OpenGL/glad/include"
]
```

Обратите внимание на предупреждение в правом нижнем углу. Нажмите на него и выберите, компилятор для C++ в вашей системе.

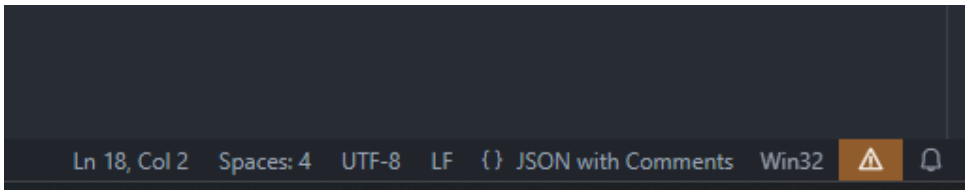

Рис.7 (предупреждение об отсутствии компилятора)

Создайте рядом с файлом *c_cpp_properties.json*, который был сгенерирован при конфигурации, файл *tasks.json.*

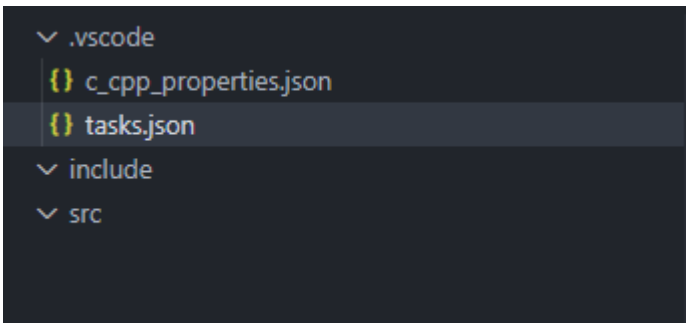

Рис.8 (Созданный файл tasks.json)

Поместите в него следующий код:

.vscode/tasks.json

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "type": "cppbuild",
            "label": "C++ Compile",
            "command": "c:\\msys64\\mingw64\\bin\\g++.exe",
            "args": [
                "-fdiagnostics-color=always",
                "${workspaceRoot}/src/*.cpp",
                "${workspaceRoot}/../OpenGL/glad/src/glad.c",
                "-I${workspaceRoot}/include",
                "--std=c++20",
                "-I${workspaceRoot}/../OpenGL/GLFW/include",
                "-L${workspaceRoot}/../OpenGL/GLFW/lib-mingw-w64",
                "-I${workspaceRoot}/../OpenGL/glad/include",
                "-static",
                "-lopengl32",
                "-lglfw3",
                "-lgdi32",
                "-lglfw3dll",
                "-o",
                "test_openGL.exe"
            ],
            "options": {
                "cwd": "${fileDirname}"
            },
            "problemMatcher": [
                "$gcc"
            ],
            "group": {
                "kind": "build",
                "isDefault": true
            }
        },
        {
            "type": "shell",
            "label": "C++ Compile & Run",
            "command": "${fileDirname}\\test_openGL.exe",
            "dependsOn": [
                "C++ Compile"
            ],
            "dependsOrder": "sequence",
```

```
            "group": {
                "kind": "build",
                "isDefault": true
            }
        }
    ]
}
```

Так мы создали две задачи. Задача *“C++ Compile”* компилирует файлы, а задача *"C++ Compile & Run"* запускает задачу *“C++ Compile”*, а затем запускает созданный файл *test_openGL.exe*, имя которого мы указали в аргументах компилятора. **Не забудьте указать корректный путь до компилятора.**

1.4. Создание простого окна

Создайте файл *main.cpp*.

Следующий код, приведенный с подробными комментариями, создает простое окно.

Весь код представлен на https://codelab.tpu.ru/kab33/opengl-tutorial

scr/main.cpp

```
#include <glad/glad.h> // glad должен включаться первым
#include <GLFW/glfw3.h>


#include <iostream>


int main()
{
    // Инициализируем GLFW
    glfwInit();


    // Устанавливаем мажорную и минорную версию GLFW
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    // Говорит GLFW использовать основной профиль
    // Таким образом мы будем иметь только современные функции
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    // Создаем окно размером 800x800 и называем его Simple Window
    GLFWwindow *window = glfwCreateWindow(800, 800, "Simple Window", NULL, NULL);
    // Проверяем, что окно было успешно создано, иначе завершаем программу с ошибкой
    if (!window)
    {
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    }


    // Устанавливаем окно как текущий контекст потока
    glfwMakeContextCurrent(window);
```

```
    // Запускаем OpenGL
    gladLoadGL();


    // Устанавливаем сцену
    // В этом случае сцена начинается с координат 0,0 и имеет размер 800x800
    glViewport(0, 0, 800, 800);


    // Основной цикл программы
    while (!glfwWindowShouldClose(window))
    {
        // Указываем цвет фона
        glClearColor(0.07f, 0.13f, 0.17f, 1.0f);
        // Очищаем задний буфер и устанавливаем цвет фона
        glClear(GL_COLOR_BUFFER_BIT);
        // Меняем задний и передний буферы местами
        glfwSwapBuffers(window);
        // Запускаем все события, необходимые для работы GLFW
        glfwPollEvents();
    }


    // Удаляем окно
    glfwDestroyWindow(window);
    // Завершаем работу GLFW
    glfwTerminate();
    return 0;
}
```

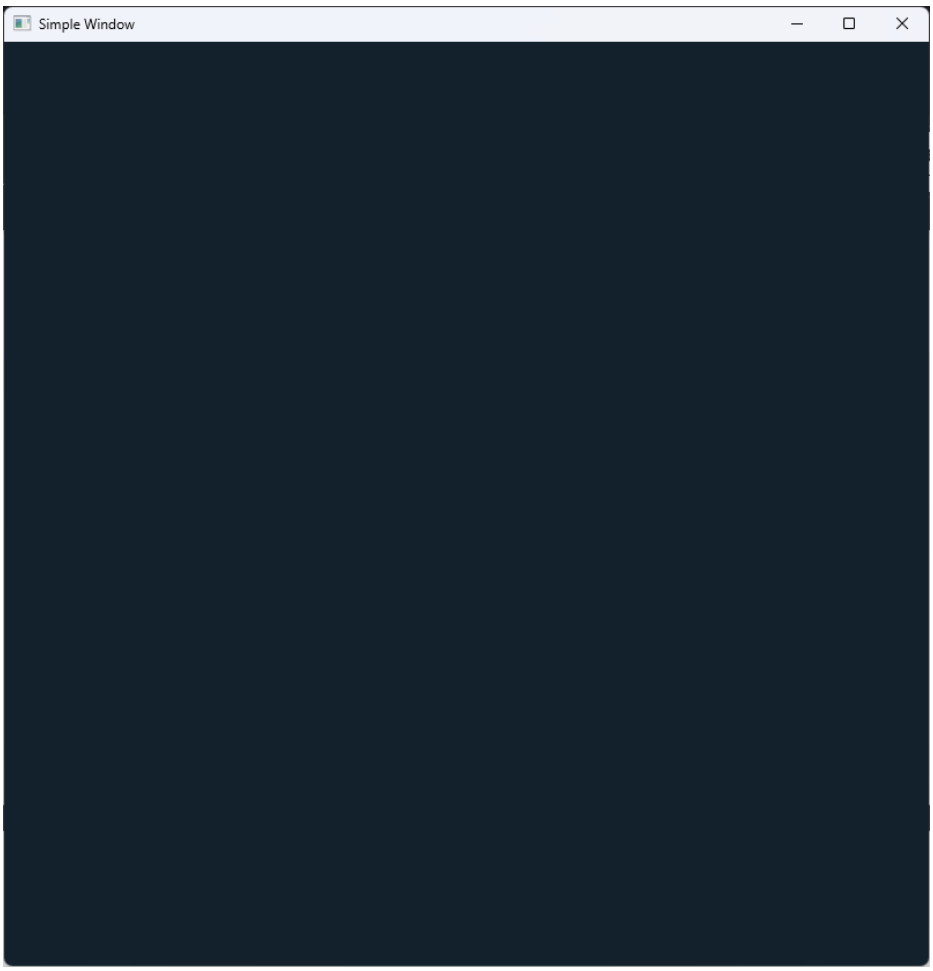

Рис. 9 (результат работы главы 1.4)

2. Основы OpenGL

2.1. Буферы

VBO, EBO и VAO являются основными компонентами OpenGL, используемыми для эффективного хранения и рендеринга геометрии.

VBO (Vertex Buffer Object) представляет собой буфер, в котором хранятся вершины (координаты, цвета, нормали и другие атрибуты) модели. Вместо передачи каждой вершины по отдельности, их данные хранятся в **памяти видеокарты**, что позволяет быстро передавать информацию на GPU. VBO можно рассматривать как массив, в котором каждый элемент представляет собой одну вершину. Это позволяет **эффективно** передавать и обрабатывать огромное количество вершин.

EBO (Element Buffer Object) используется для хранения индексов, определяющих порядок вершин в полигональной модели. Использование индексов позволяет определить повторяющиеся вершины и избежать повторной передачи одних и тех же данных. Это особенно полезно при отрисовке больших моделей, так как позволяет сократить объем передаваемой информации.

VAO (Vertex Array Object) представляет собой контейнер для хранения информации о расположении и структуре VBO и EBO. VAO содержит ссылки на VBO и EBO, а также информацию о расположении атрибутов вершин. Это позволяет легко связывать данные из VBO и EBO с атрибутами вершин при отрисовке моделей.

Вместе VBO, EBO и VAO образуют мощный инструмент для эффективного хранения и рендеринга геометрии с использованием OpenGL. Надлежащее использование этих компонентов может **значительно увеличить производительность** и уменьшить объем передаваемых данных при отрисовке сложных сцен.

Реализуем эти компоненты в виде классов.

В нашей рабочей директории создадим директорию include. В неё мы будем класть заголовочные файлы (.h).

Создадим классы VAO, VBO и EBO.

src/VBO.h

```
#ifndef VBO_CLASS_H

#define VBO_CLASS_H

#include <glad/glad.h>

class VBO

{
```

```
public:
    GLuint ID;
    VBO(GLfloat *vertices, GLsizeiptr size);
    void bind();
    void unbind();
    void remove();
};
#endif // VBO_CLASS_H
```

src/VAO.h

```
#ifndef VAO_CLASS_H
#define VAO_CLASS_H

#include <glad/glad.h>
#include <VBO.h>

class VAO
{
public:
    GLuint ID;
    VAO();
    void linkAttrib(VBO VBO, GLuint layout, GLuint numComponents,
                    GLenum type, GLsizeiptr stride, void *offset);
    void bind();
    void unbind();
    void remove();
};

#endif // VAO_CLASS_H
```

src/EBO.h

```
#ifndef EBO_CLASS_H
#define EBO_CLASS_H
```

```
#include <glad/glad.h>

class EBO
{
public:
    GLuint ID;
    EBO(GLuint *indixes, GLsizeiptr size);
    void bind();
    void unbind();
    void remove();
};
#endif // EBO_CLASS_H
```

Напишем для них реализацию.

src/VBO.cpp

```
#include "VBO.h"

VBO::VBO(GLfloat *vertices, GLsizeiptr size)
{
    glGenBuffers(1, &ID);
    glBindBuffer(GL_ARRAY_BUFFER, ID);
    // Introduce the vertices into the VBO
    glBufferData(GL_ARRAY_BUFFER, size, vertices, GL_DYNAMIC_DRAW);
}
void VBO::bind()
{
    glBindBuffer(GL_ARRAY_BUFFER, ID);
}
void VBO::unbind()
{
    glBindBuffer(GL_ARRAY_BUFFER, 0);
```

```
}
void VBO::remove()
{
    glDeleteBuffers(1, &ID);
}
```

src/VAO.cpp

```
#include "VAO.h"


VAO::VAO()
{
    glGenVertexArrays(1, &ID);
}
void VAO::linkAttrib(VBO VBO, GLuint layout, GLuint numComponents, GLenum type,
GLsizeiptr stride, void *offset)
{
    VBO.bind();
    // Скажем OpenGL как читать VBO
    glVertexAttribPointer(layout, numComponents, type, GL_FALSE, stride, offset);
    // Скажем OpenGL как использовать VBO
    glEnableVertexAttribArray(layout);
    VBO.unbind();
}
void VAO::bind()
{
    glBindVertexArray(ID);
}
void VAO::unbind()
{
    glBindVertexArray(0);
}
void VAO::remove()
{
    glDeleteVertexArrays(1, &ID);
```

```
}
```

src/EBO.cpp

```
#include "EBO.h"


EBO::EBO(GLuint *indexes, GLsizeiptr size)
{
    glGenBuffers(1, &ID);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ID);
    // Внедрение индексов
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, size, indexes, GL_DYNAMIC_DRAW);
}
void EBO::bind()
{
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ID);
}
void EBO::unbind()
{
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
}
void EBO::remove()
{
    glDeleteBuffers(1, &ID);
}
```

2.2. Шейдеры

Шейдеры в OpenGL — это программные единицы, которые выполняют различные операции над графическими данными в процессе рендеринга 3D графики. Они выполняются на графическом процессоре (GPU) и позволяют контролировать каждый пиксель и вершину, отображаемых на экране.

В OpenGL существует два основных типа шейдеров: вершинные (Vertex Shader) и фрагментные (Fragment Shader).

Вершинный шейдер — это программный код, который обрабатывает каждую вершину входной модели. Он может выполнять операции, такие как трансформации координат, освещение и применение матриц проекции и вида. Вершинные шейдеры обычно используются для задания положения, нормалей и текстурных координат вершин, а также передачи данных другим шейдерам.

Фрагментный шейдер, также известный как пиксельный шейдер, работает с отдельными пикселями на экране. Он отвечает за определение цвета пикселя на основе его координат, текстурных данных, освещения и других входных параметров. Фрагментные шейдеры могут выполнять сложные операции, такие как текстурирование, освещение, отбрасывание теней и постобработку изображений.

Шейдеры в OpenGL программируются на языке GLSL (OpenGL Shading Language). Он поддерживает множество возможностей, включая математические операции, текстурирование, освещение и другие функции, необходимые для создания реалистичной графики.

Шейдеры являются ключевым компонентом в современных графических приложениях, позволяя разработчикам контролировать каждый аспект визуализации. Они обеспечивают гибкость и эффективность при создании сложных и реалистичных сцен.

Напишем вершинный и фрагментный шейдер. В папке *src* создадим папку *shaders*, а в ней файлы *default.vert* и *default.frag.*

src/shaders/default.vert

```
#version 330 core

layout (location = 0) in vec3 aPos;

layout (location = 1) in vec3 aColor;

// Вывод цвета для фрагментного шейдера

out vec3 color;

void main()

{

    // Получаем позиции из массива вершин

    gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);

    // Получаем цвета из массива вершин

    color = aColor;

}
```

src/shaders/default.frag

```
#version 330 core
out vec4 FragColor;
// Ввод цвета из вершинного шейдера
in vec3 color;
void main()
{
    FragColor = vec4(color, 1.0f);
}
```

Теперь создадим класс, для управления шейдером.

include/shader.h

```
#ifndef SHADER_CLASS_H
#define SHADER_CLASS_H

#include <glad/glad.h>
#include <string>
#include <fstream>
#include <sstream>
#include <iostream>
#include <cerrno>

std::string get_file_content(const char *filename);
class Shader
{
public:
    GLuint ID;
    Shader(const char *vertexFile, const char *fragmentFile);
    void activate();
    void remove();
};
#endif // SHADER_CLASS_H
```

Напишем реализацию.

src/shader.cpp

```
#include "shader.h"

std::string get_file_content(const char *filename)
{
    std::ifstream in(filename, std::ios::binary);
    if (in)
    {
        std::string contents;
        in.seekg(0, std::ios::end);
        contents.resize(in.tellg());
        in.seekg(0, std::ios::beg);
        in.read(&contents[0], contents.size());
        in.close();
        return (contents);
    }
    throw(errno);
}
Shader::Shader(const char *vertexFile, const char *fragmentFile)
{
    std::string vertexCode = get_file_content(vertexFile);
    std::string fragmentCode = get_file_content(fragmentFile);

    const char *vertextSource = vertexCode.c_str();
    const char *fragmentSource = fragmentCode.c_str();

    // Создаем объект вершинного шейдера
    GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
    // Объединяем объект вершинного шейдера с его кодом
    glShaderSource(vertexShader, 1, &vertextSource, NULL);
    // Компилируем шейдер
    glCompileShader(vertexShader);
    // Создаем объект фрагментного шейдера
```

```
    GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
    // Объединяем объект фрагментного шейдера с его кодом
    glShaderSource(fragmentShader, 1, &fragmentSource, NULL);
    // Компилируем шейдер
    glCompileShader(fragmentShader);
    // Создаем объект программы шейдера
    ID = glCreateProgram();
    // Объединяем программу шейдера с объектами вершинного и фрагментного шейдера
    glAttachShader(ID, vertexShader);
    glAttachShader(ID, fragmentShader);
    // Связываем все в программе шейдера
    glLinkProgram(ID);


    // Удаляем отработавшие объекты
    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);
}
void Shader::activate()
{
    glUseProgram(ID);
}
void Shader::remove()
{
    glDeleteProgram(ID);
}
```

2.3. Создание простой фигуры

OpenGL предполагает, что все вершины, которые мы хотим увидеть, после запуска шейдера будут в нормализованных координатах устройства (NDC — normalized device coordinates). Это означает, что x, y и z координаты каждой вершины должны быть между -1.0 и 1.0. Центр координат будет находиться в середине сцены.

Массив данных о вершинах будет в виде [x1, y1, z1, r1, g1, b1, x2, y2, z2, r2, g2, b2... xn, yn, zn, rn, gn, bn], где n количество вершин. Создадим глобальную переменную перед функцией main. В ней будет храниться данные о квадрате с вершинами (-0.5, 0.5), (0.5, 0.5), (0.5, -0.5) и (-0.5, -0.5). Вершины квадрата буду раскрашены в синий, зеленый, желтый и красный.

src/main.cpp

```
GLfloat vertices[] = {
    //             COORDINATE     /         COLORS         //
    -0.5f, 0.5f, 0.0f, 0.0f, 128 / 255.0f, 255 / 255.0f,
    0.5f, 0.5f, 0.0f, 255 / 255.0f, 0 / 255.0f, 0 / 255.0f,
    0.5f, -0.5f, 0.0f, 0 / 255.0f, 153 / 255.0f, 0 / 255.0f,
    -0.5f, -0.5f, 0.0f, 255 / 255.0f, 255 / 255.0f, 51 / 255.0f};
```

Рядом с массивом вершин создадим массив индексов. Он нужен, чтобы показать OpenGL в каком порядке обходить вершины.

src/main.cpp

```
GLuint indexes[] = {
    0, 1, 2, 3};
```

Перед основным циклом создадим программу шейдер, VBO, VAO и EBO.

```
    Shader shaderProgram("src/shaders/default.vert", "src/shaders/default.frag");


    VAO VAO1;
    VAO1.bind();


    VBO VBO1(vertices, sizeof(vertices));
    EBO EBO1(indexes, sizeof(indexes));


    VAO1.linkAttrib(VBO1, 0, 3, GL_FLOAT, 6 * sizeof(float), (void *)0);
    VAO1.linkAttrib(VBO1, 1, 3, GL_FLOAT, 6 * sizeof(float), (void *)(3 *
sizeof(float)));
    VAO1.unbind();
    VBO1.unbind();
    EBO1.unbind();
```

В самом цикле активируем программу шейдер и отрисуем фигуру.

```
// Основной цикл программы
while (!glfwWindowShouldClose(window))
{
    // Указываем цвет фона
    glClearColor(0.07f, 0.13f, 0.17f, 1.0f);
    // Очищаем задний буфер и устанавливаем цвет фона
    glClear(GL_COLOR_BUFFER_BIT);


    // Активируем программу шейдер
    shaderProgram.activate();
    // Скажем OpenGL использовать VBO
    VAO1.bind();
    // Скажем OpenGL нарисовать фигуру с 4 вершинами
    glDrawElements(GL_TRIANGLE_FAN, 4, GL_UNSIGNED_INT, 0);


    // Меняем задний и передний буферы местами
    glfwSwapBuffers(window);
    // Запускаем все события, необходимые для работы GLFW
    glfwPollEvents();
}
```

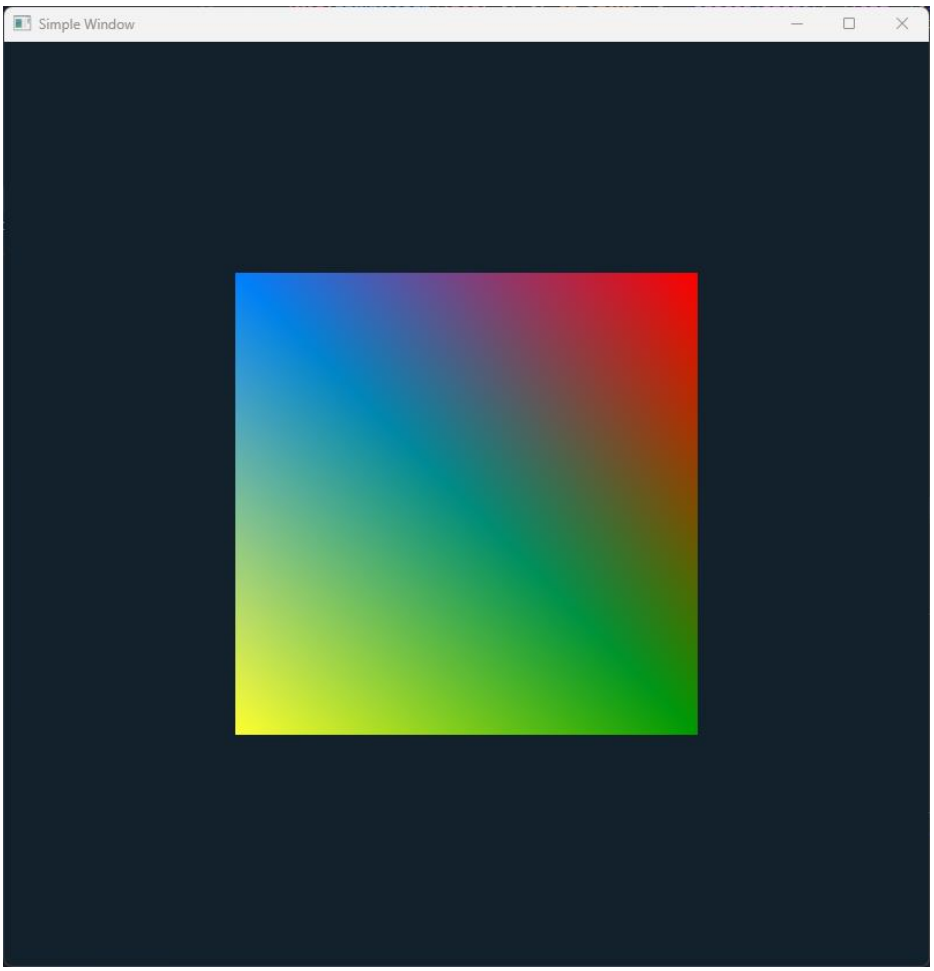

Рис.10 (результата работы главы 2.3)

2.4. Текстуры

В OpenGl текстура – это изображение или набор данных, которые применяются к геометрии 3D-объекта. Она может быть использована для придания поверхности или объекту визуальных свойств, таких как цвет, освещение или шейдеры.

В OpenGL текстуры манипулируются в виде прямоугольной сетки пикселей. Каждый пиксель содержит информацию о цвете или других свойствах. Текстуры могут быть созданы из изображений в файле или генерироваться динамически в программе.

OpenGL поддерживает различные типы текстур, такие как 2D-текстуры, кубические текстуры, объемные текстуры и массивы текстур. Каждый тип имеет свои особенности и применяется в зависимости от потребностей визуализации.

Текстуры в OpenGl могут быть использованы для создания сложных и реалистичных визуальных эффектов, таких как затенение, отражение, прозрачность и текстурирование поверхностей. Они являются важным инструментом для разработки графических приложений и игр, обеспечивая более реалистичную и привлекательную визуализацию.

Текстуры лучше оптимизированы, если они имеют размер в виде 2^n x 2^n px, где n - неотрицательное целое число. Вы можете скачать любое изображения на ваш вкус. Для примера возьмем текстуру гравия с сайта https://3dtextures.me/2022/04/27/gravel-001. Нам нужно изображение с название BaseColor.

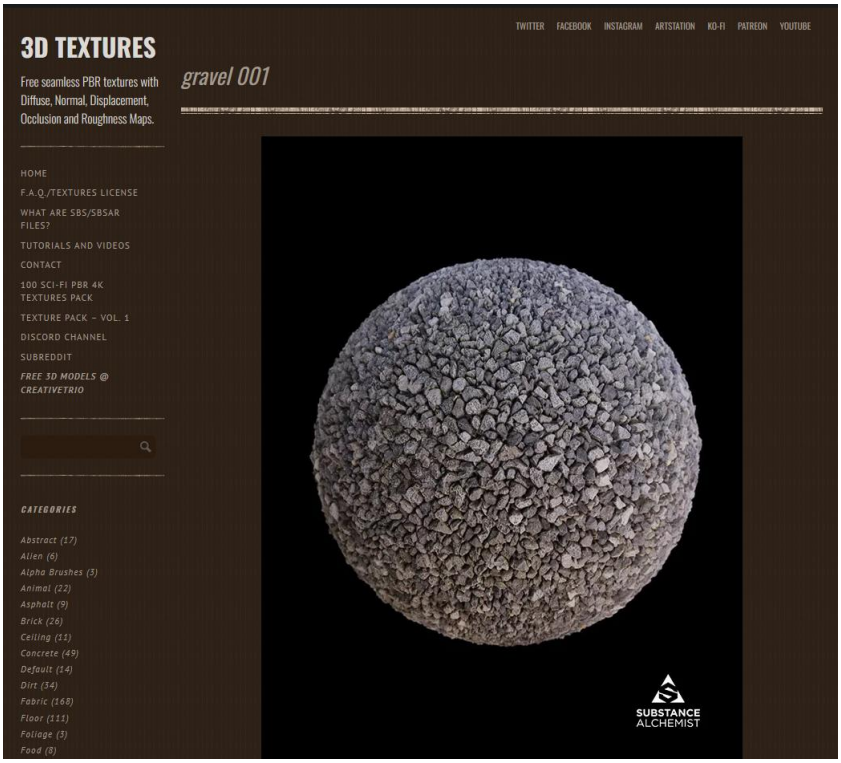

Рис. 11 (представление текстуры на сайте)

Рис. 12 (текстура гравия)

Создадим директорию *textures* в директории *src* и пометим туда скачанную текстуру.

Чтобы не забивать *main.cpp*, создадим специальный класс, который будет олицетворять текстуру, но перед этим нам необходима библиотека *stb*, чтобы с текстурами было удобнее работать. Её можно скачать в виде исходного кода из открытого репозитория и скомпилировать на своей машине. Нам нужен только один файл из этого репозитория. Вот ссылка на него: *https://github.com/nothings/stb/blob/master/stb_image.h*

Поместим этот файл в директорию *stb,* созданную рядом с директорией *OpenGL*.

Рис. 13 (структура проекта в главе 2.4)

В директории *src* создадим файл *stb_image.cpp*. И поместим туда следующий код, как сказано в документации.

src/stb_image.cpp

```
#define STB_IMAGE_IMPLEMENTATION

#include <stb_image.h>
```

В задачу сборки необходимо добавить новую директорию, как мы это делали для OpenGL.

.vscode/tasks.json

```
"-I${workspaceRoot}/../stb",
```

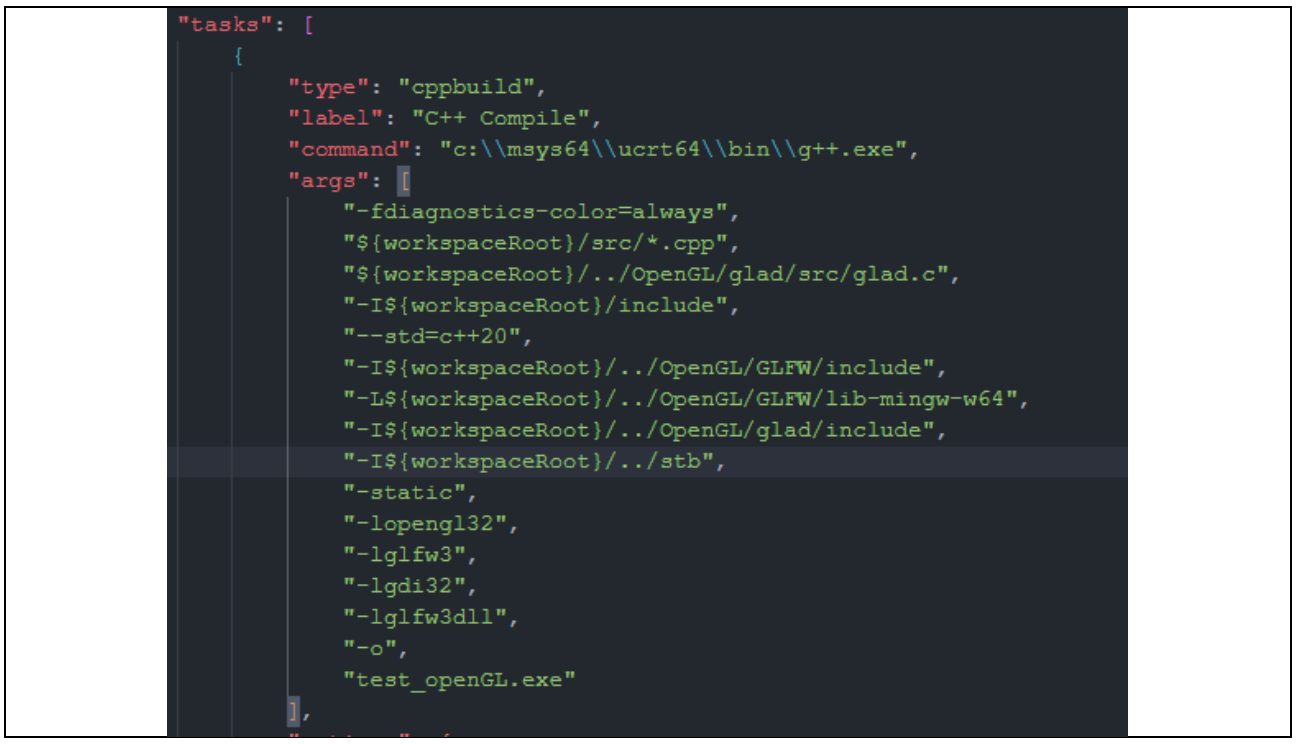

Рис. 14 (файл *tasks.json* в главе 2.4)

Чтобы редактор видел функции из заголовочного файла, добавим директорию *stb* в *c_cpp_properties.json*

```
"${workspaceFolder}/../stb"
```

```
{
    "configurations": [
        {
            "name": "c++ config",
            "includePath": [
                "${workspaceFolder}/include",
                "${workspaceFolder}/../OpenGL/GLFW/include",
                "${workspaceFolder}/../OpenGL/glad/include",
                "${workspaceFolder}/../stb"
            ],
            "compilerPath": "c:\\msys64\\mingw64\\bin\\g++.exe",
            "cStandard": "c20",
            "cppStandard": "c++20",
            "intelliSenseMode": "gcc-x86"
        }
    ],
    "version": 4
}
```

Рис. 15 (файл *c_cpp_properties.json* в главе 2.4)

Теперь создадим класс текстуры. Ниже приведен код с подробными комментариями.

inlcude/texture.h

```
#ifndef TEXTURE_CLASS_H

#define TEXTURE_CLASS_H

#include <glad/glad.h>

#include <stb/stb_image.h>


#include "shader.h"


class Texture

{

public:

    GLuint ID;

    GLenum type;

    Texture(const char *image, GLenum texType, GLenum slot, GLenum format, GLenum
pixelType);


    // Присваивает текстуре единицу текстуры
```

```
    void texUnit(Shader &shader, const char *uniform, GLuint unit);
    // Связывает текстуру
    void bind();
    // Отвязывает текстуру
    void unbind();
    // Удаляет текстуру
    void delete();
};
#endif
```

src/texture.cpp

```
#include "texture.h"

Texture::Texture(const char *image, GLenum texType, GLenum slot, GLenum format, GLenum
pixelType)
{
    // Присваиваем тип текстуры объекту texture
    type = texType;

    // Сохраняем ширину, высоту и количество цветовых каналов изображения
    int widthImg, heightImg, numColCh;
    // Переворачивает изображение так, чтобы оно отображалось правой стороной вверх
    // Это необходимо так как stb и OpenGL имеет разную систему координат для текстур
    stbi_set_flip_vertically_on_load(true);
    // Считываем изображение из файла и сохраняем его в байтах
    unsigned char *bytes = stbi_load(image, &widthImg, &heightImg, &numColCh, 0);

    // Генерируем текстурный объект OpenGL
    glGenTextures(1, &ID);
    // Присваиваем текстуру текстурному элементу
```

```
    glActiveTexture(slot);
    glBindTexture(texType, ID);

    // Настраиваем тип алгоритма, который используется для уменьшения или увеличения изображения
    glTexParameteri(texType, GL_TEXTURE_MIN_FILTER, GL_NEAREST_MIPMAP_LINEAR);
    glTexParameteri(texType, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

    // Настраиваем способ повторения текстуры (если это вообще происходит)
    glTexParameteri(texType, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(texType, GL_TEXTURE_WRAP_T, GL_REPEAT);

    // Присваиваем изображение объекту текстуры OpenGL
    glTexImage2D(texType, 0, GL_RGBA, widthImg, heightImg, 0, format, pixelType, bytes);
    // Генерируем MipMaps
    glGenerateMipmap(texType);

    // Удаляем данные изображения в том виде, в каком они уже есть в объекте текстуры OpenGL
    stbi_image_free(bytes);

    // Отменяет привязку объекта текстуры OpenGL, чтобы его нельзя было случайно изменить
    glBindTexture(texType, 0);
}

void Texture::texUnit(Shader &shader, const char *uniform, GLuint unit)
{
    // Получаем местоположение униформы
    GLuint texUni = glGetUniformLocation(shader.ID, uniform);
```

```
    // Шейдер необходимо активировать перед изменением значения униформы
    shader.activate();
    // Устанавливаем значение униформы
    glUniform1i(texUni, unit);
}

void Texture::bind() {glBindTexture(type, ID);}

void Texture::unbind() {glBindTexture(type, 0);}

void Texture::delete() {glDeleteTextures(1, &ID);}
```

Изменим ранее написанные шейдеры.

Внимание! Любая опечатка в шейдерах может привести к ошибкам, которые компилятор не покажет.

shaders/default.vert

```
#version 330 core
// Координаты
layout (location = 0) in vec3 aPos;
// Цвет
layout (location = 1) in vec3 aColor;
// Координаты текстуры
layout (location = 2) in vec2 aTex;

// Вывод цвета для фрагментного шейдера
out vec3 color;
// Вывод координат тектсуры для фрагментного шейдера
out vec2 texCoord;

void main()
{
```

```
    // Получаем позиции из массива вершин
    gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
    // Получаем цвета из массива вершин
    color = aColor;
    // Получаем координаты текстуры
    texCoord = aTex;
}
```

shaders/default.frag

```
#version 330 core
out vec4 FragColor;

// Ввод цвета из вершинного шейдера
in vec3 color;
// Ввод координат текстуры из вершинного шейдера
in vec2 texCoord;

// Получаем единицу текспуры из основной функции
uniform sampler2D tex0;

void main()
{
    FragColor = texture(tex0, texCoord);
}
```

Обратите внимание на униформу в *default.frag.*

Униформа в OpenGL — это механизм, который позволяет передавать данные из приложения в шейдеры. Эти данные могут быть использованы для управления параметрами рендеринга, такими как положение объектов, матрицы преобразования, цвета и другие свойства. Униформы являются глобальными переменными, доступными во всех шейдерах программы и обеспечивают синхронизацию данных между приложением и шейдерами. Они обычно используются для передачи данных, которые изменяются редко или не изменяются вообще в течение одного кадра рендеринга.

Добавим следующий код перед основным циклом для применения текстуры.

src/main.cpp

```
Texture grav_tex("C:\\Projects\\tpu-opengl-
tutorial\\simple\\src\\textures\\gravel_base_color.jpg",
                    GL_TEXTURE_2D,
                    GL_TEXTURE0,
                    GL_RGB,
                    GL_UNSIGNED_BYTE);

grav_tex.texUnit(shaderProgram, "tex0", 0);
```

В самом цикле добавим активацию текстуры перед активацией VBO.

```
grav_tex.bind();
```

В массиве вершин добавим информацию о координатах текстуры.

```
GLfloat vertices[] = {
    //             COORDINATE     /         COLORS      / TEXTURE COORDINATE
    -0.5f, 0.5f, 0.0f, 0.0f, 128 / 255.0f, 255 / 255.0f, 0.0f, 0.0f,
    0.5f, 0.5f, 0.0f, 255 / 255.0f, 0 / 255.0f, 0 / 255.0f, 0.0f, 1.0f,
    0.5f, -0.5f, 0.0f, 0 / 255.0f, 153 / 255.0f, 0 / 255.0f, 1.0f, 1.0f,
    -0.5f, -0.5f, 0.0f, 255 / 255.0f, 255 / 255.0f, 51 / 255.0f, 1.0f, 0.0f};
```

Необходимо изменить параметры в привязке, так как мы добавили новую информацию.

```
VAO1.linkAttrib(VBO1, 0, 3, GL_FLOAT, 8 * sizeof(float), (void *)0);
VAO1.linkAttrib(VBO1, 1, 3, GL_FLOAT, 8 * sizeof(float), (void *)(3 *
sizeof(float)));
VAO1.linkAttrib(VBO1, 2, 2, GL_FLOAT, 8 * sizeof(float), (void *)(6 *
sizeof(float)));
```

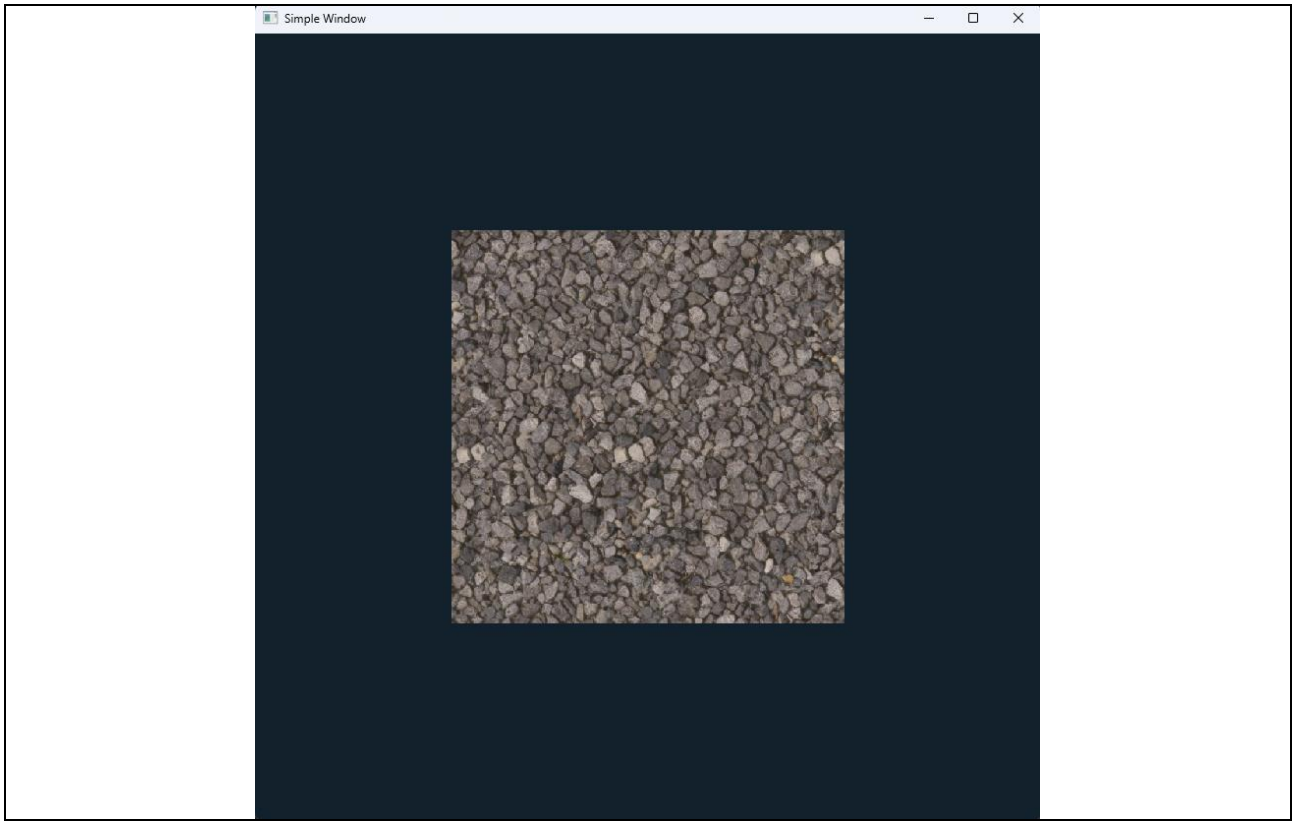

Рис. 16 (результат работы главы 2.4)

2.5. Анимация вершин

Под анимацией вершин подразумевается трансформация объекта. Вращение и перемещение всей фигуры будет рассмотрены в главе 3.

Для трансформации фигуры нам достаточно изменить вершину и снова отрисовать фигуру. Чтобы это выглядело плавно можно воспользоваться таймером.

Создадим анимацию превращения нашего квадрата в трапецию и обратно.

Напишем функция трансформации.

src/main.cpp

```
void transform()

{

    // переменная для направления анимации

    static bool isStretch = true;

    // шаг трансформации
```

```
    GLfloat step = 0.01;


    if (isStretch) {
        vertices[2 * 8] += step;
        vertices[3 * 8] -= step;


        if (vertices[2 * 8] >= 0.7f) {
            isStretch = false;
        }
    } else {
        vertices[2 * 8] -= step;
        vertices[3 * 8] += step;


        if (vertices[2 * 8] <= 0.5f) {
            isStretch = true;
        }
    }
    // обновление данных в буфере
    glBufferSubData(GL_ARRAY_BUFFER, sizeof(GLfloat) * 2 * 8, sizeof(GLfloat), new
GLfloat(vertices[2*8]));
    glBufferSubData(GL_ARRAY_BUFFER, sizeof(GLfloat) * 3 * 8, sizeof(GLfloat), new
GLfloat(vertices[3*8]));
}
```

Перед основным циклом объявим, текущее время.

```
double prevTime = glfwGetTime();
```

В самом цикле добавим код для анимации. Его необходимо поместить между активацией шейдерной программы и привязкой текстуры.

```
        // анимация с шагом в 0.01 секунды
        double currTime = glfwGetTime();
        if (currTime - prevTime >= 0.1) {
            // скажем OpenGL использовать VBO
```

```
        VBO1.bind();
        // Произедем шаг трансформации
        transform();
        // обновим данные в VBO
        VAO1.linkAttrib(VBO1, 0, 3, GL_FLOAT, 8 * sizeof(float), (void *)0);
        // обновим время
        prevTime = currTime;
    }
```

```
// Основной цикл программы
while (!glfwWindowShouldClose(window))
{
    // Указываем цвет фона
    glClearColor(0.07f, 0.13f, 0.17f, 1.0f);
    // Очищаем задний буфер и устанавливаем цвет фона
    glClear(GL_COLOR_BUFFER_BIT);
    // Активируем программу шейдер
    shaderProgram.activate();

    // анимация с шагом в 0.01 секунды
    double currTime = glfwGetTime();
    if (currTime - prevTime >= 0.1) {
        // скажем OpenGL использовать VBO
        VBO1.bind();
        // Произедем шаг трансформации
        transform();
        // обновим данные в VBO
        VAO1.linkAttrib(VBO1, 0, 3, GL_FLOAT, 8 * sizeof(float), (void *)0);
        // обновим время
        prevTime = currTime;
    }

    // Скажем OpenGL использовать текстуру
    grav_tex.bind();
    // Скажем OpenGL использовать VBO
    VAO1.bind();
    // Скажем OpenGL нарисовать фигуру с 4 вершинами
```

Рис. 17 (основой цикл в главе 2.5)

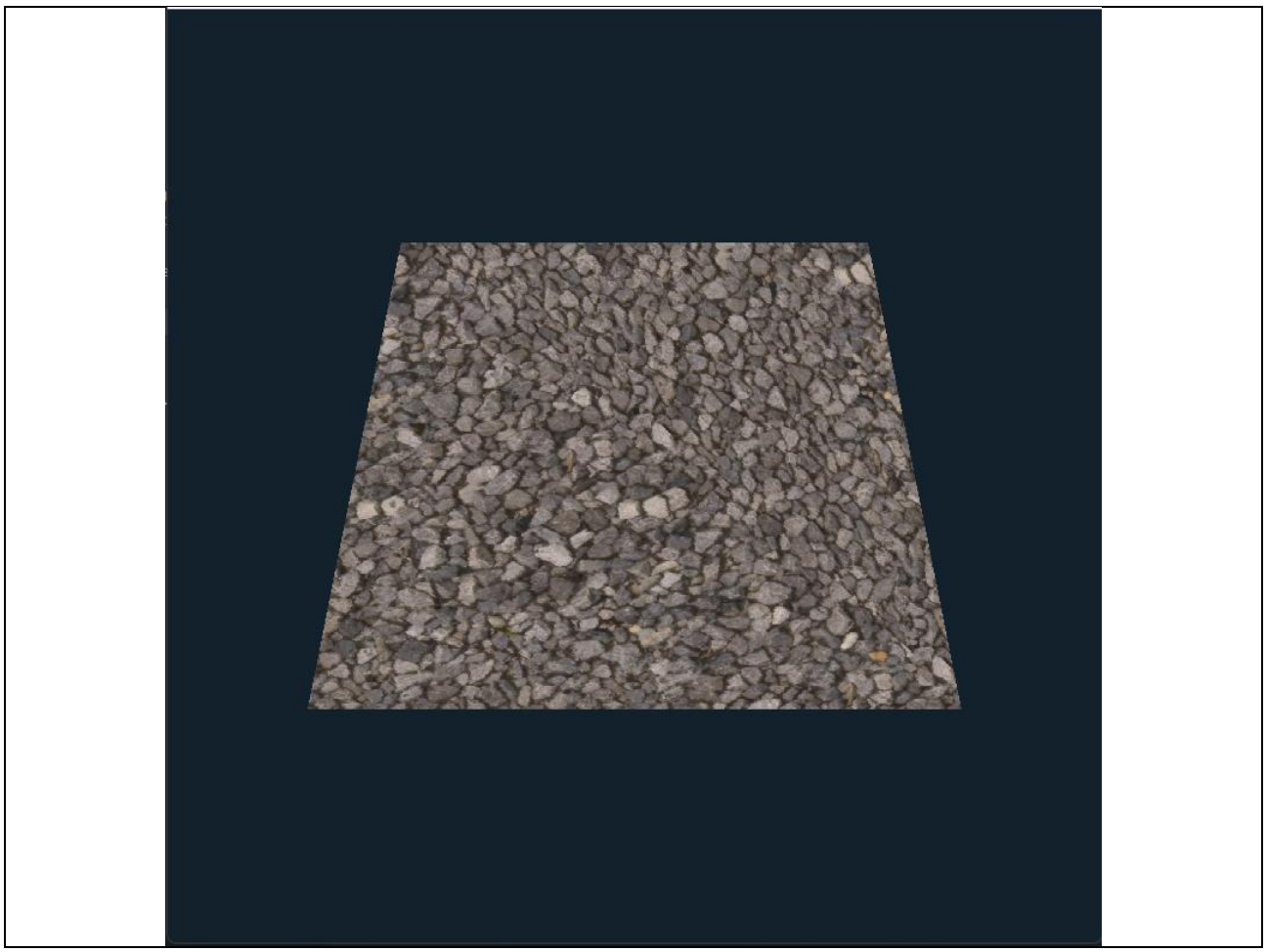

Рис. 18 (вид фигуры в результате анимации)

3. 3D в OpenGL

Эта глава и все последующие не содержат код главы 2.5 (см. дерево глав).

3D графика в OpenGL основана на использовании трехмерных координат и матриц преобразований. В OpenGL каждый объект, такой как треугольник или прямоугольник, задается своими вершинами в трехмерном пространстве. Затем с помощью матриц преобразования можно изменять положение, масштабирование и поворот объекта.

Для работы с матрицами нам понадобиться библиотека GLM (GL Mathematics). Её можно скачать из репозитория *https://github.com/g-truc/glm*. Нам нужна директория *glm*.

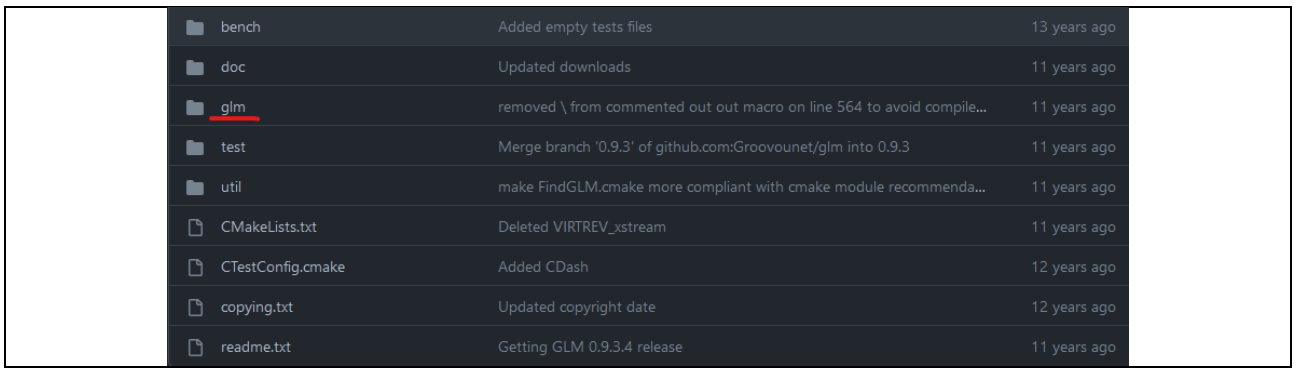

Рис.19 (требуемая директория в репозитории *glm*)

Поместите её рядом с *OpenGL* и *stb.*

Как и с предыдущими библиотеками, нам нужно добавить её в задание сборки и IntelliSense.

```
"includePath": [
    "${workspaceFolder}/include",
    "${workspaceFolder}/../OpenGL/GLFW/include",
    "${workspaceFolder}/../OpenGL/glad/include",
    "${workspaceFolder}/../stb"
    "${workspaceFolder}/../glm"
],
```

```
"args": [
    "-fdiagnostics-color=always",
    "${workspaceRoot}/src/*.cpp",
    "${workspaceRoot}/../OpenGL/glad/src/glad.c",
    "-I${workspaceRoot}/include",
    "--std=c++20",
    "-I${workspaceRoot}/../OpenGL/GLFW/include",
    "-L${workspaceRoot}/../OpenGL/GLFW/lib-mingw-w64",
    "-I${workspaceRoot}/../OpenGL/glad/include",
    "-I${workspaceRoot}/../stb",
    "-I${workspaceRoot}/../glm",
    "-static",
    "-lopengl32",
    "-lglfw3",
    "-lgdi32",
    "-lglfw3dll",
    "-o",
    "test_openGL.exe"
],
```

Рис. 20 (добавление *gml* в конфигурации)

В фрагментарном шейдере необходимо добавить матричные униформы и изменить установку позиции.

default.frag

#version 330 core

// Координаты

layout (location = 0) in vec3 aPos;

// Цвет

layout (location = 1) in vec3 aColor;

// Координаты текстуры

layout (location = 2) in vec2 aTex;

// Вывод цвета для фрагментного шейдера

```
out vec3 color;
// Вывод координат тектсуры для фрагментного шейдера
out vec2 texCoord;
uniform mat4 model;
uniform mat4 view;
uniform mat4 proj;
void main()
{
    // Получаем позиции из матриц
    gl_Position = proj * view * model * vec4(aPos, 1.0);
    // Получаем цвета из массива вершин
    color = aColor;
    // Получаем координаты тектсуры
    texCoord = aTex;
}
```

Создадим в основном цикле матрицы, преобразуем их и выведем в вершинный шейдер.

src/main.cpp

```
// Создадим матрицу модели, содержащей единицы
glm::mat4 model = glm::mat4(1.0f);
// Создадим матрицу вида, содержащей единицы
glm::mat4 view = glm::mat4(1.0f);
// Создадим матрицу проекции, содержащей единицы
glm::mat4 proj = glm::mat4(1.0f);

// Повернем фигуру на 45 градусов
model = glm::rotate(model, glm::radians(45.0f), glm::vec3(0.0f, 0.0f, 1.0f));
// Отодвинем фигуру от нас
view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));
// Скажем не обрезать фигуру,
```

```
// если она находиться от нас от 0.1 до 100.0
proj = glm::perspective(glm::radians(45.0f), (float)800 / 800, 0.1f, 100.0f);

// Выводим матрицы в вершинный шейдер
int modelLoc = glGetUniformLocation(shaderProgram.ID, "model");
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
int viewLoc = glGetUniformLocation(shaderProgram.ID, "view");
glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(view));
int projLoc = glGetUniformLocation(shaderProgram.ID, "proj");
glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(proj));
```

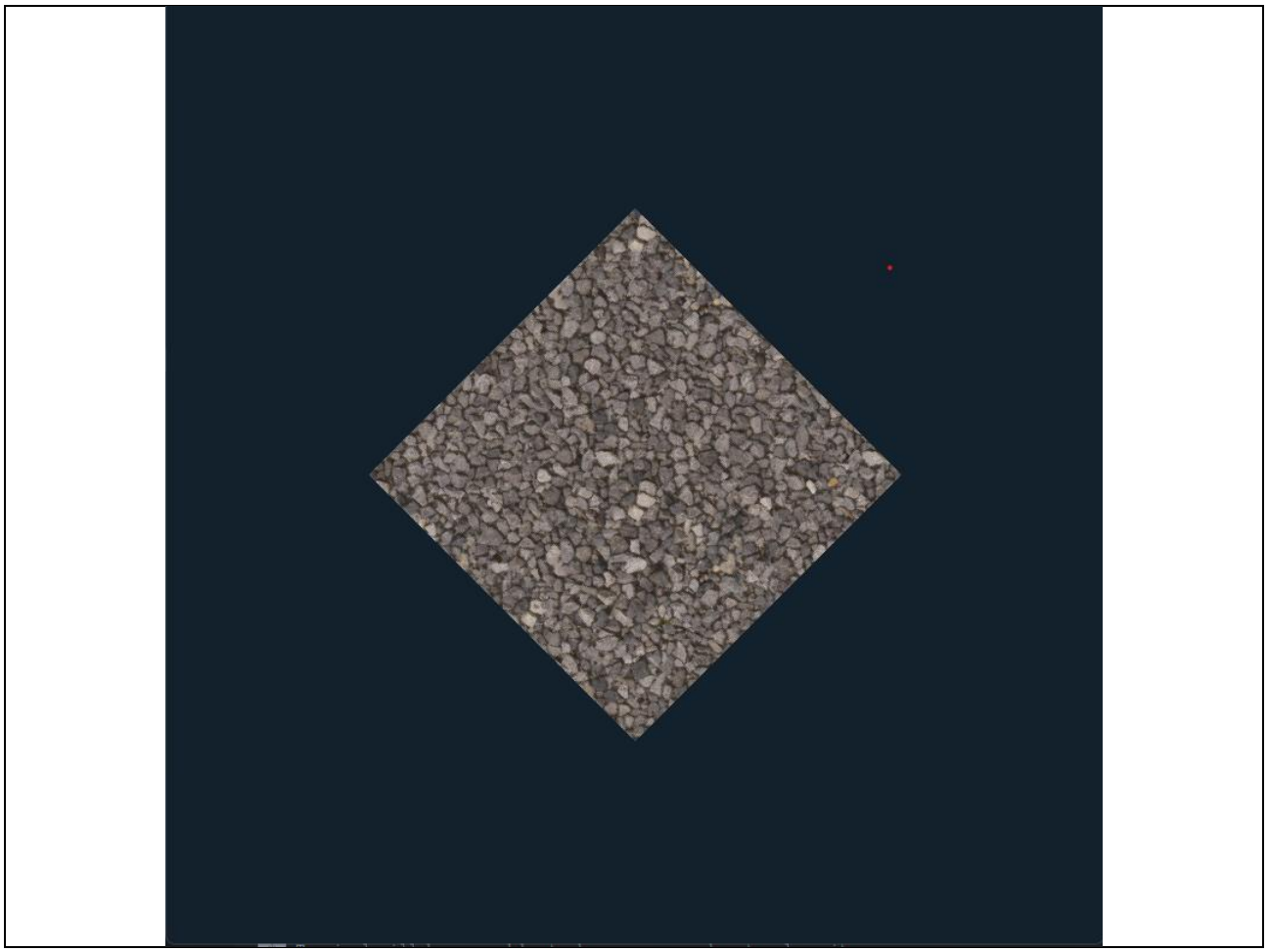

Рис. 21 (результат поворота фигуры)

Так как мы теперь можем работать с 3D, создадим подходящую для этого фигуру.

Изменим матрицу вершин и индексов для отрисовки октаэдра.

```
GLfloat vertices[] =
{ //     COORDINATES     /        COLORS      /   TexCoord  //
```

```
    -0.5f, 0.0f, -0.5f,     0.0f, 0.0f, 0.0f,   1.0f, 0.0f,
    -0.5f, 0.0f,  0.5f,     0.0f, 0.0f, 0.0f,   0.0f, 0.0f,
     0.5f, 0.0f,  0.5f,     0.0f, 0.0f, 0.0f,   1.0f, 0.0f,
     0.5f, 0.0f, -0.5f,     0.0f, 0.0f, 0.0f,   0.0f, 0.0f,
     0.0f, 0.8f,  0.0f,     0.0f, 0.6f, 0.0f,   0.5f, 1.0f,
     0.0f, -0.8f,  0.0f,     0.0f, 0.6f, 0.0f,  0.5f, 1.0f,
};
GLuint indexes[] = {
    4,0,1,
    4,1,2,
    4,2,3,
    4,3,0,
    5,0,1,
    5,1,2,
    5,2,3,
    5,3,0,};
```

Изменим параметры для функции отрисовки вершин.

```
glDrawElements(GL_TRIANGLES, sizeof(indexes) / sizeof(int), GL_UNSIGNED_INT, 0);
```

Чтобы OpenGL понимал, какие грани нужно отрисовывать впереди, добавим флаг перед основным циклом.

```
glEnable(GL_DEPTH_TEST);
```

И в очистке буфера.

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Добавим таймер для плавного поворота октаэдра в основном цикле.

```
// Таймер для анимации
double crntTime = glfwGetTime();
if (crntTime - prevTime >= 0.05)
{
     rotation += 2.0f;
     prevTime = crntTime;
}
```

И его инициализацию перед основным циклом.

```
float rotation = 0.0f;

double prevTime = glfwGetTime();
```

Не забудьте поменять число на переменную в месте, где мы указываем величину поворота для матрицы модели. Еще поменяем ось, вокруг которой будет происходить вращение. Пусть это будет ось Y.

```
model = glm::rotate(model, glm::radians(rotation), glm::vec3(0.0f, 1.0f, 0.0f));
```

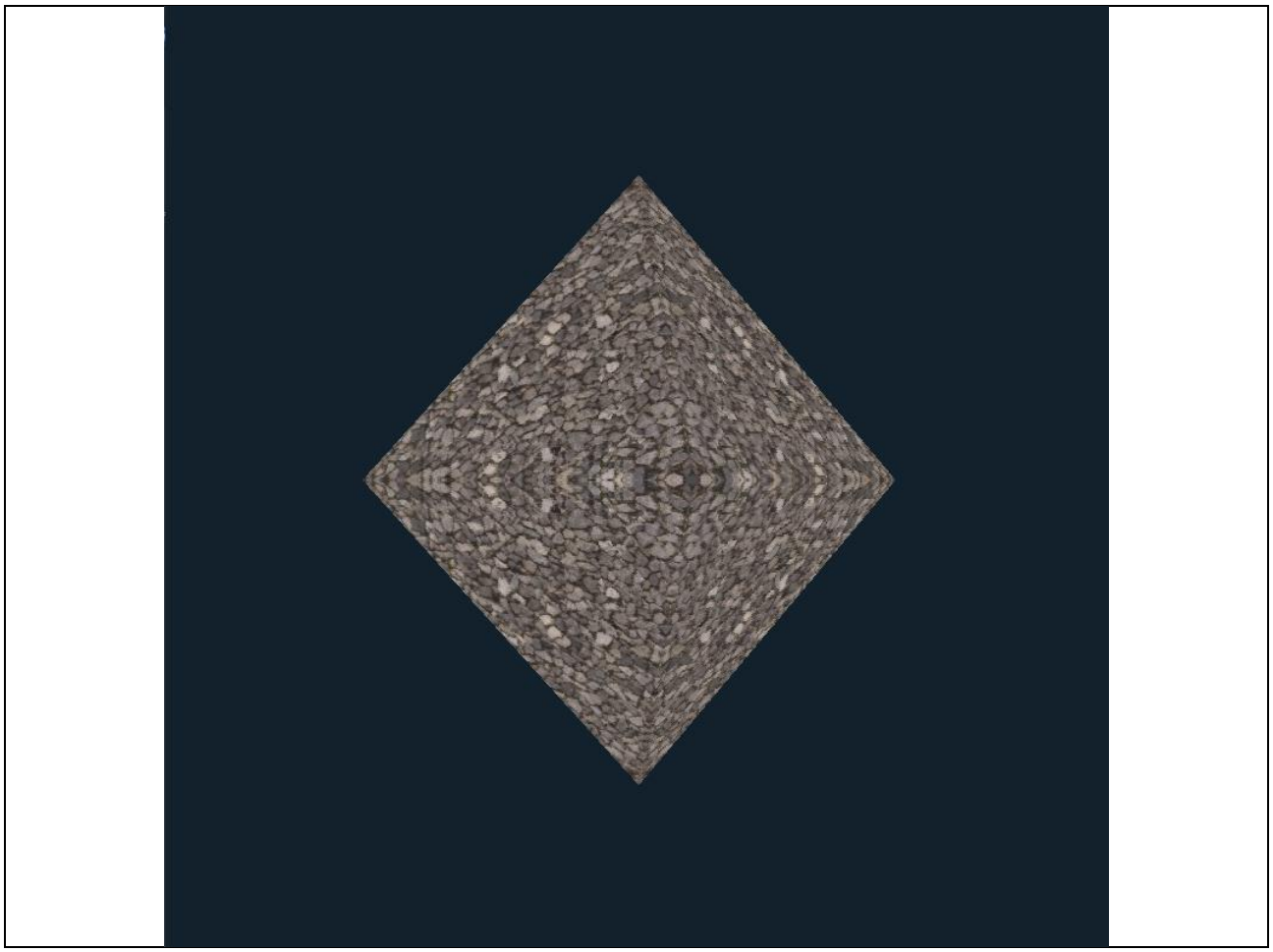

Рис. 21 (результат работы программы в главе 3)