МИНИСТЕРСТВО науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Кафедра автоматизации обработки информации (АОИ)

ПРИМЕНЕНИЕ БУФЕРА ТРАФАРЕТА В OPENGL.

Отчет по Курсовой работе
по дисциплине
«Компьютерная Графика»

Выполнила: ст. гр.
Ильюшина Марина Владимировна
«09.03.04»
29.09.2024 г.

Руководитель _____
(должность, ученая степень)
_____ _____
(подпись) (И. О. Фамилия)

Геретсрид 2024

Оглавление

Введение

Целью данной курсовой работы является изучение и применение методов работы с буфером трафарета в библиотеке OpenGL для создания интерактивной трехмерной сцены. В рамках работы необходимо реализовать трехмерную сцену, содержащую тетраэдр с вырезанными отверстиями при помощи буфера трафарета. Внутри тетраэдра располагается полупрозрачный цилиндр, описанный тором.

Данная задача позволяет продемонстрировать использование OpenGL для обработки сложных визуальных эффектов, таких как работа с трафаретом и прозрачностью, а также знакомит с основными принципами работы трехмерной графики. Для реализации проекта будет использоваться среда разработки Microsoft Visual Studio 2022 с языком программирования C++ и библиотекой OpenGL.

1 ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ

Моё Задание № 11 исходя из личного кабинета.

Тема: Применение буфера трафарета в OpenGL.

Задание: реализовать трехмерную сцену, содержащую изображение тетраэдра с вырезанными (при помощи буфера трафарета) в гранях отверстиями. Внутри тетраэдра поместить полупрозрачный цилиндр, описанный тором. Полученную сцену вращать по таймеру

2 РАЗРАБОТКА ТРЕХМЕРНОЙ СЦЕНЫ С ИСПОЛЬЗОВАНИЕМ БУФЕРА ТРАФАРЕТА В OPENGL

2.1 Среда разработки

Для реализации проекта была выбрана среда разработки Microsoft Visual Studio 2022, использующая язык программирования C++ и библиотеку OpenGL. OpenGL предоставляет набор функций для работы с трехмерной графикой и является кроссплатформенным решением, что делает его удобным для создания графических приложений. Дополнительно использовались библиотеки GLAD для загрузки OpenGL-функций и GLFW для управления окнами и обработки событий. Библиотека GLM была применена для работы с математическими операциями, такими как преобразования матриц.

2.2 Постановка задачи

Основная задача проекта — создание трехмерной сцены, содержащей тетраэдр с вырезанными отверстиями, выполненными с помощью буфера трафарета. Внутри тетраэдра располагается полупрозрачный цилиндр, описанный тором. Сцена должна непрерывно вращаться с использованием таймера, демонстрируя работу буфера трафарета и эффекты прозрачности.

3 ПОДГОТОВКА ОКРУЖЕНИЯ

На данном этапе нашей работы мы приступаем к подготовке окружения для разработки. Для этого нам предстоит скачать и установить необходимые компоненты:

- **Microsoft Visual Studio 2022**: Я выбрала эту среду разработки для работы с C++. Она поддерживает создание и отладку графических приложений, что идеально подходит для моего проекта. На этом этапе у меня уже существует данная IDE Visual Studio 2022 и мне не придётся её устанавливать.
- **OpenGL**: хотя OpenGL уже предустановлен в операционной системе Windows, для более удобной работы с графическим контекстом и окнами необходимо дополнительно скачать библиотеки **GLFW** и **GLAD.** GLFW будет использоваться для управления окнами и обработкой ввода с клавиатуры и мыши. GLAD понадобится для автоматической загрузки всех функций OpenGL. Эти библиотеки мы также будем скачивать с официальных сайтов. [1,3]
- **GLM**: для работы с математическими операциями, такими как матрицы преобразований, векторы и вращения, мы использовали библиотеку **GLM**. Для работы с математическими операциями в OpenGL нам потребуется библиотека **GLM**. Она предоставляет функции для работы с векторами, матрицами и другими математическими операциями, необходимыми для трехмерной графики. На этом этапе мы скачиваем GLM, чтобы впоследствии интегрировать её в проект. [2,4]

После скачивания всех компонентов я начинаю их установку и настройку для дальнейшей работы над проектом. Подготовка окружения — важный шаг для успешного выполнения задачи.

4 НАСТРОЙКА ПРОЕКТА В VISUAL STUDIO 2022

Итак, после того как мы скачали все нужные нам библиотеки от для работы с OpenGL мы приступаем к созданию проекта и пошаговой её установки

4.1 Создание нового проекта

Для создания нового проекта открываем IDE Visual Studio 2022 и в появившемся окне выбираем Create a new project (Рисунок 4.1)

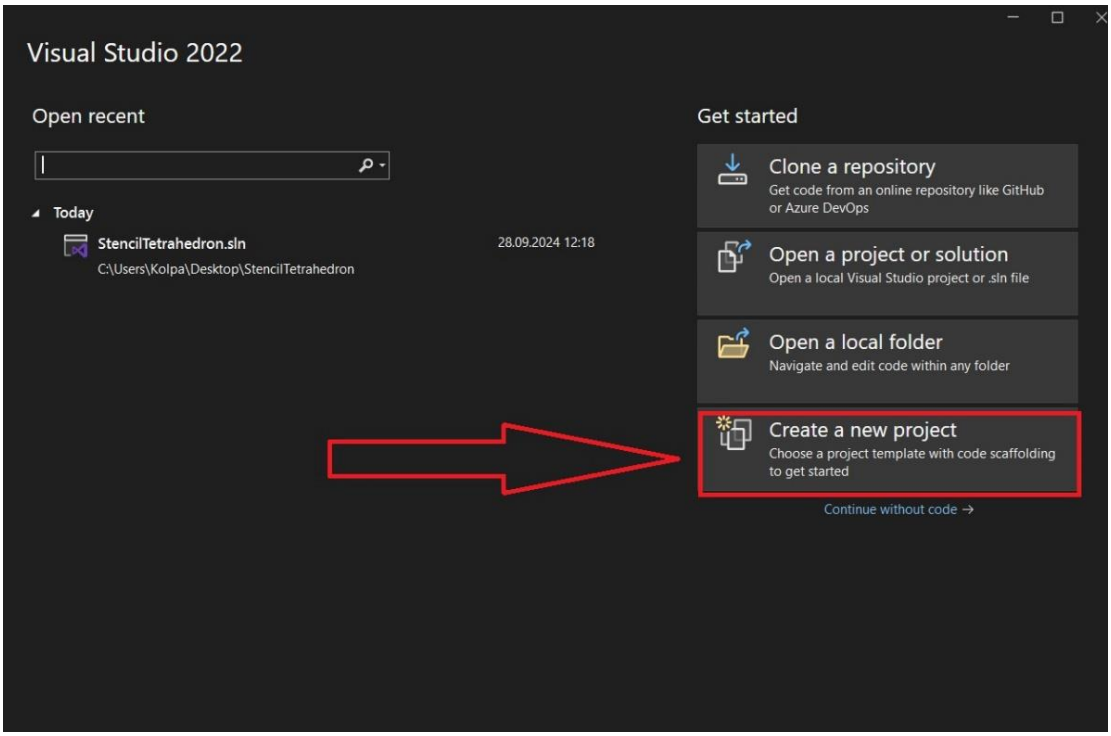
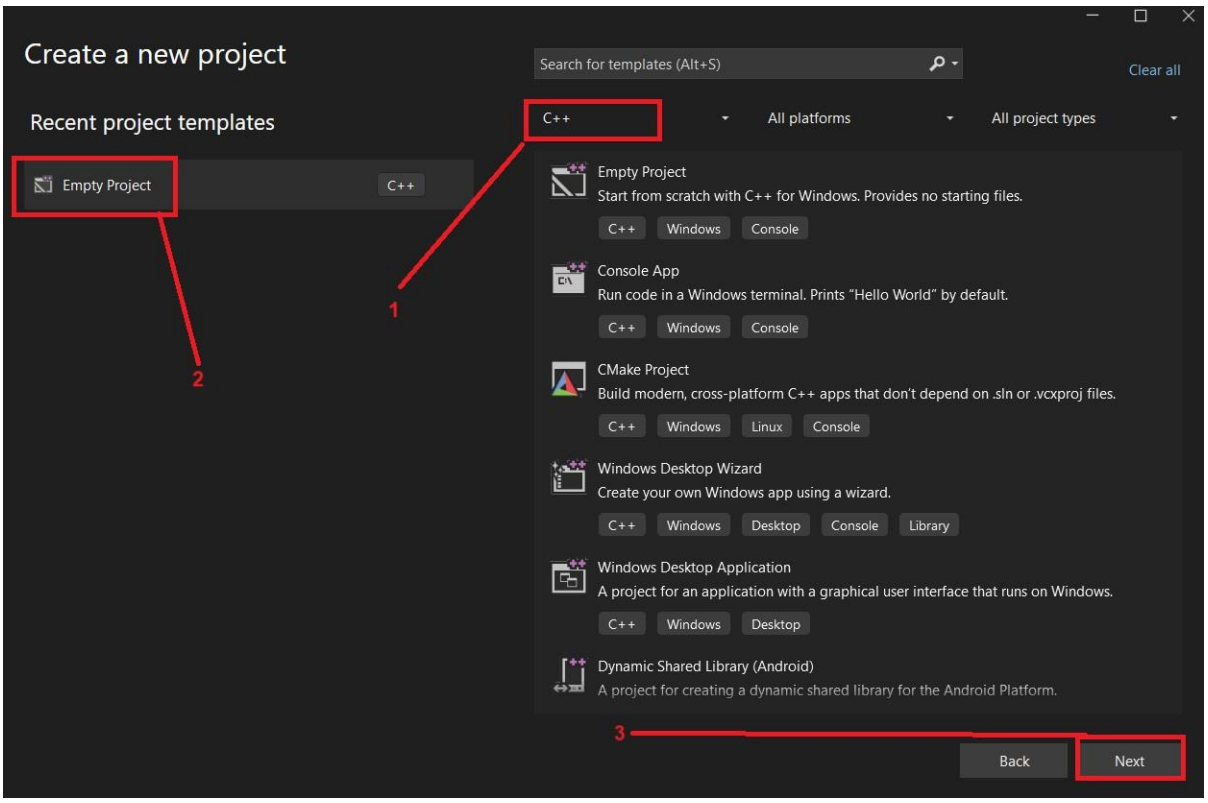


Рисунок 4.1–Создание нового проекта в IDE Visual Studio 2022

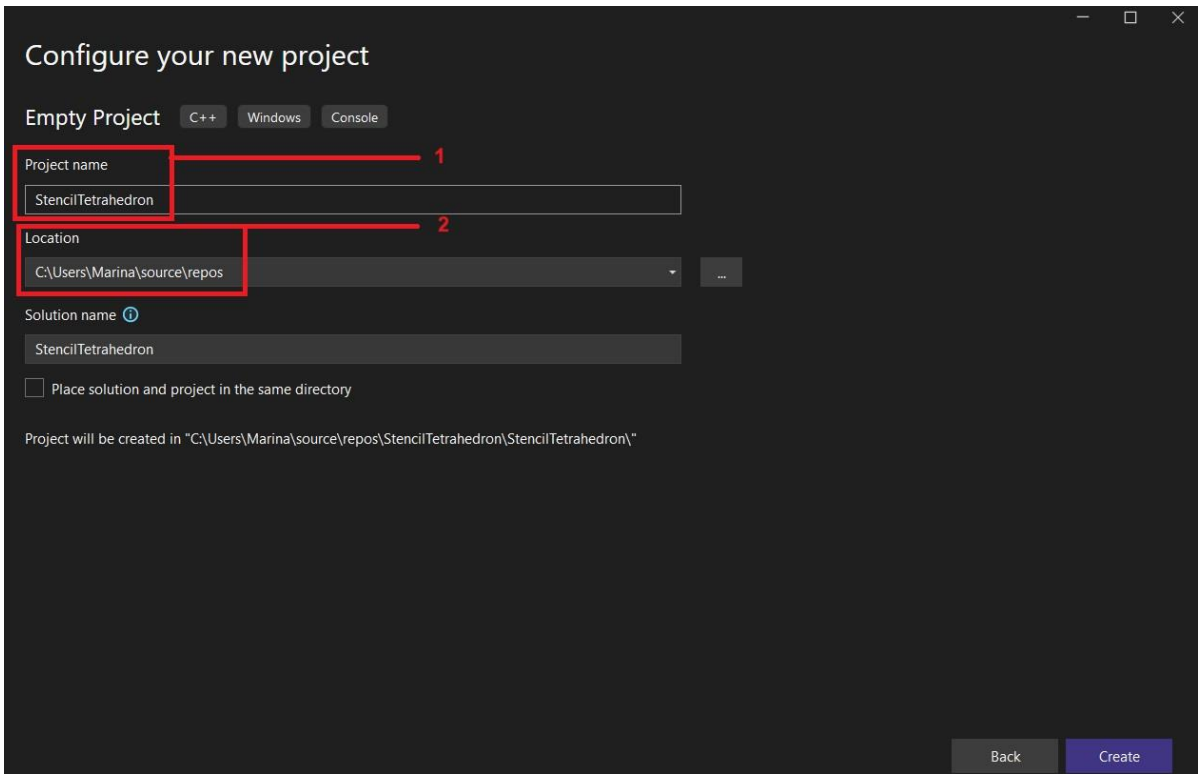
Далее мы выбираем язык программирования C++ и находим там Empty Project в нашей IDE Visual studio 2022 и нажимаем на кнопку next (Рисунок 4.2)



1-Выбор языка программирования; 2-выбор пустого проекта на C++; 3-кнопка next для прохождения на следующий этап

Рисунок 4.2 –Создание проекта и выбор языка программирования.

После нажатия кнопки «Next» открывается окно, в котором необходимо выбрать путь для сохранения и компиляции проекта, а также задать его имя (Рисунок 4.3). Таким образом, создаётся проект. После его создания добавляем новый файл main.cpp, в котором будет реализован весь код проекта.



1-название проекта; 2-расположение проекта

Рисунок 4.3 –Создание проекта

4.2 Добавление библиотеки GLFW

После загрузки библиотеки GLFW [3] с официального сайта, её необходимо установить в проект. Для этого в папке проекта создаём директорию **dependencies**, внутри которой создаём папки **include** и **lib**. В эти папки распаковываем соответствующие файлы библиотеки. (Рисунок 4.4)

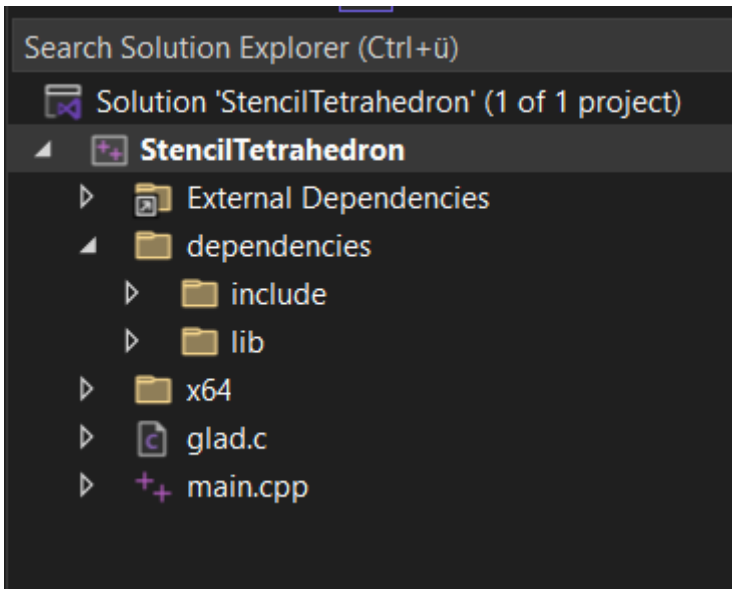


Рисунок 4.4 –Директории проекта

Необходимо скопировать содержимое папки **include** из загруженного архива в папку **include** нашего проекта. Затем скопировать файл **glfw3.lib** из папки **lib-vc2022** загруженного архива в папку **lib** нашего проекта. Далее добавляем пути к заголовочным файлам и библиотекам в проект.

Нажимаем правой кнопкой мыши на наш проект и выбираем вкладку «Properties». В появившемся окне выполняем следующие действия:

- В **C/C++ -> General -> Additional Include Directories** добавляем путь к GLFW/include который мы скопировали в наш проект. (Рисунок 4.5)
- В **Linker -> General -> Additional Library Directories** добавляем путь к GLFW/lib-vc2022 который мы скопировали в наш проект. (Рисунок 4.6)
- В **Linker -> Input -> Additional Dependencies** добавляем glfw3.lib, opengl32.lib, user32.lib, gdi32.lib, shell32.lib (Рисунок 4.7)

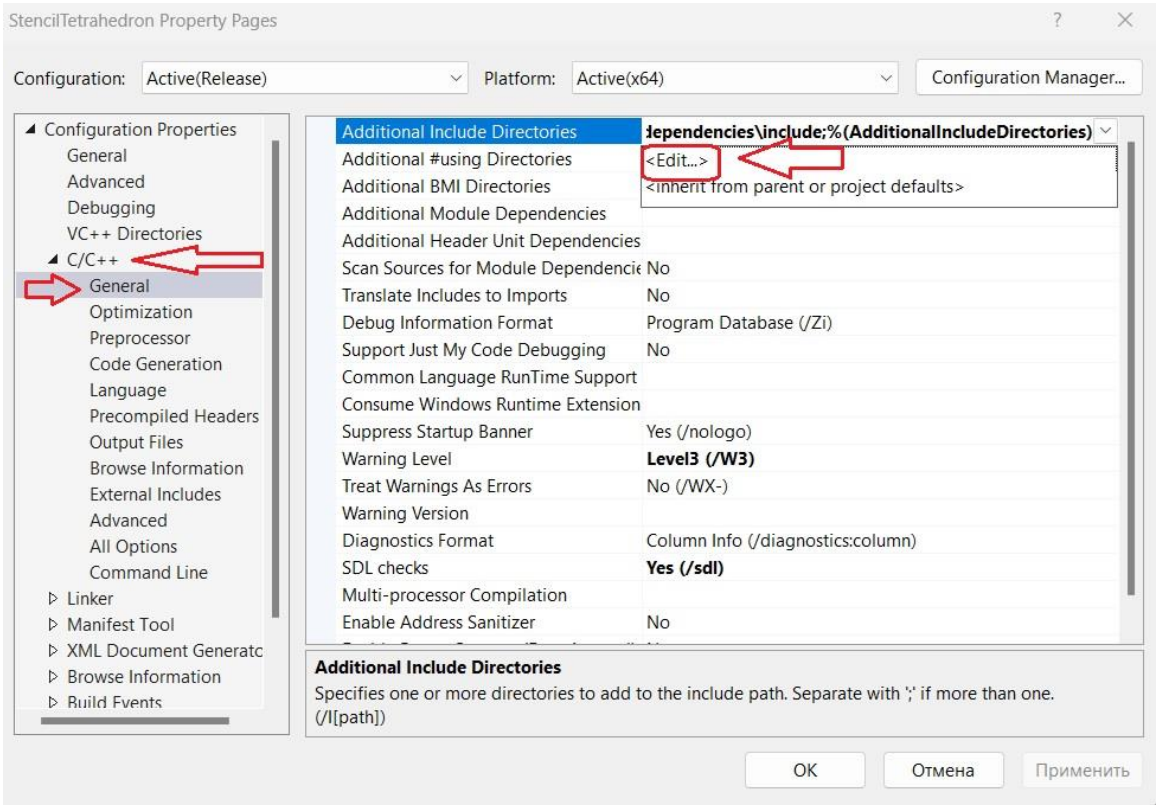


Рисунок 4.5 –Добавление в каталог библиотек GLFW

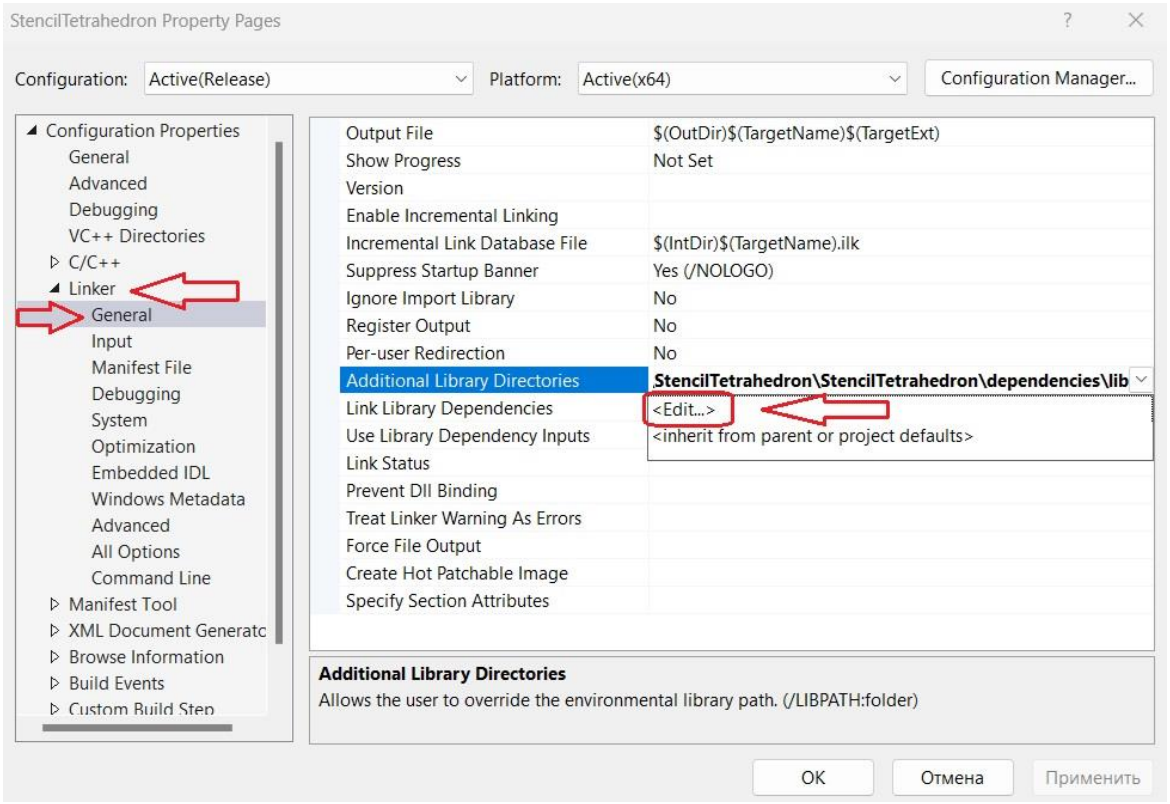


Рисунок 4.6 –Добавления справочных библиотек GLFW

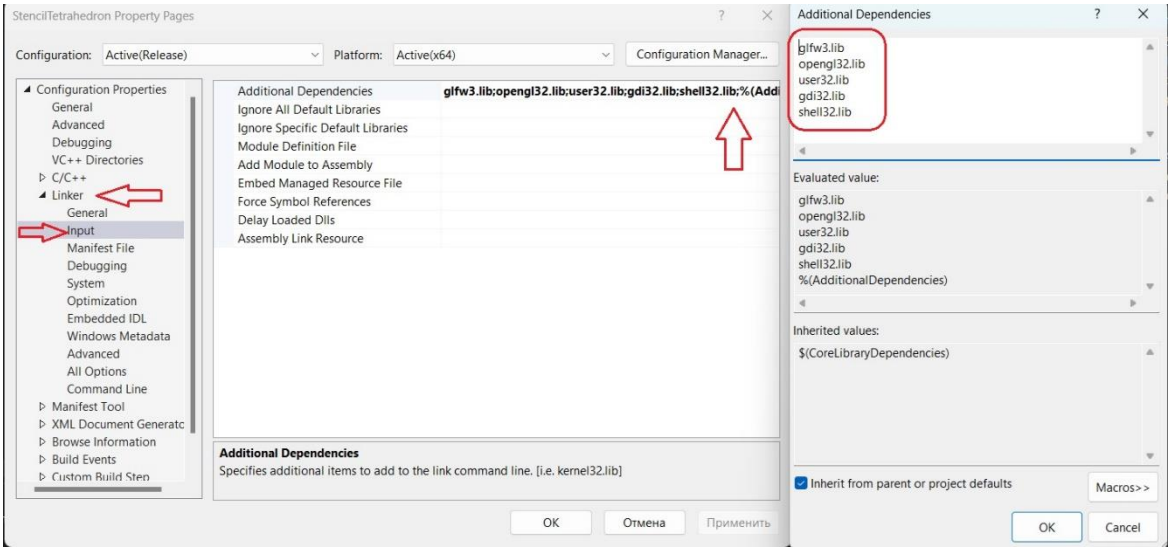
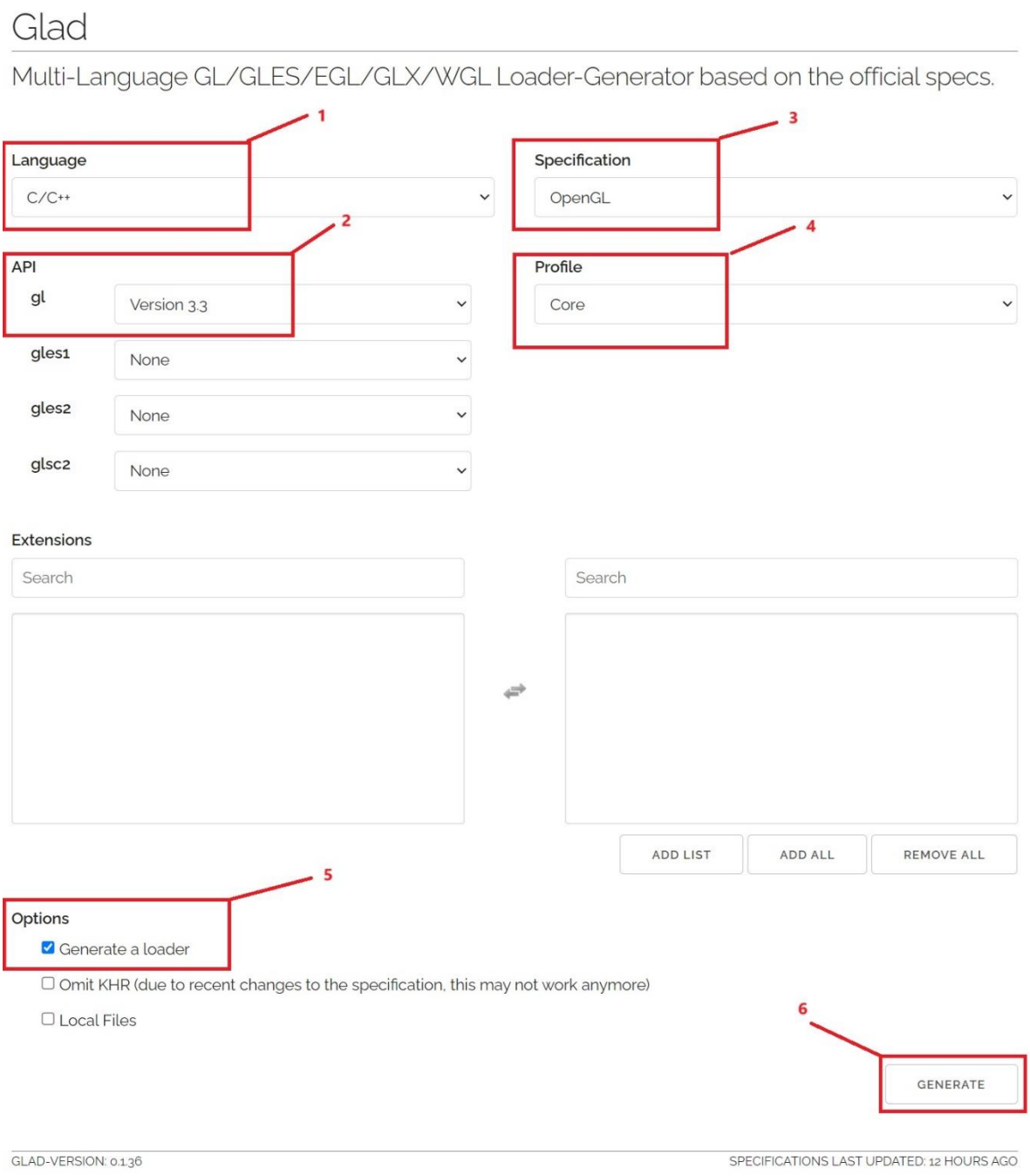


Рисунок 4.7 –Добавление дополнительных зависимостей

4.3 Добавление GLAD

Переходим на сайт [1] чтобы сгенерировать и загрузить наши файлы. Выбираем в пункте **Language:** C/C++, **Specification**: OpenGL, **API** / GL: Version 3.3, **Profile**: Core. в **Options** ставим галочку **Generate a loader** и жмем на кнопку **GENERATE**. (Рисунок 4.8)



1-выбор языка программирования; 2-Выбор версии API; 3-Выбор интерфейса; 4-выбор движка; 5-выбор опций скачивания; 6-генерация файлов.

Рисунок 4.8 –Генерация файлов GLAD

После загрузки сгенерированного архива распаковываем его и добавляем файл **glad.c** и заголовочные файлы в наш проект.

Далее включаем путь к GLAD в **C/C++ -> General -> Additional Include Directories**. (Рисунок 4.5)

4.4 Добавление GLM библиотеку

Переходим на сайт [2] находим последнюю версию библиотек, скачиваем и распаковываем её. Затем читаем документацию из источника [4] и распаковываем соответствующий архив. После того как разархивировали данный файл мы находим папку **glm** и копируем её по в наш проект в папку **include.** И опять указываем путь к заголовочному файлу **glm** в **C/C++ -> General -> Additional Include Directories**. (Рисунок 4.5)

4.5 Настройка и проверка исходных файлов

Далее включаем все добавленные в проект файлы библиотек в **IDE Visual Studio 2022**. Находим файлы, скопированные в папку **dependencies**, и все файлы, подсвеченные красным, выделяем правой кнопкой мыши, затем выбираем «Include in Project». Это можно сделать как для каждого файла по отдельности, так и для всей папки сразу, чтобы вся папка была добавлена в проект (Рисунок 4.9).

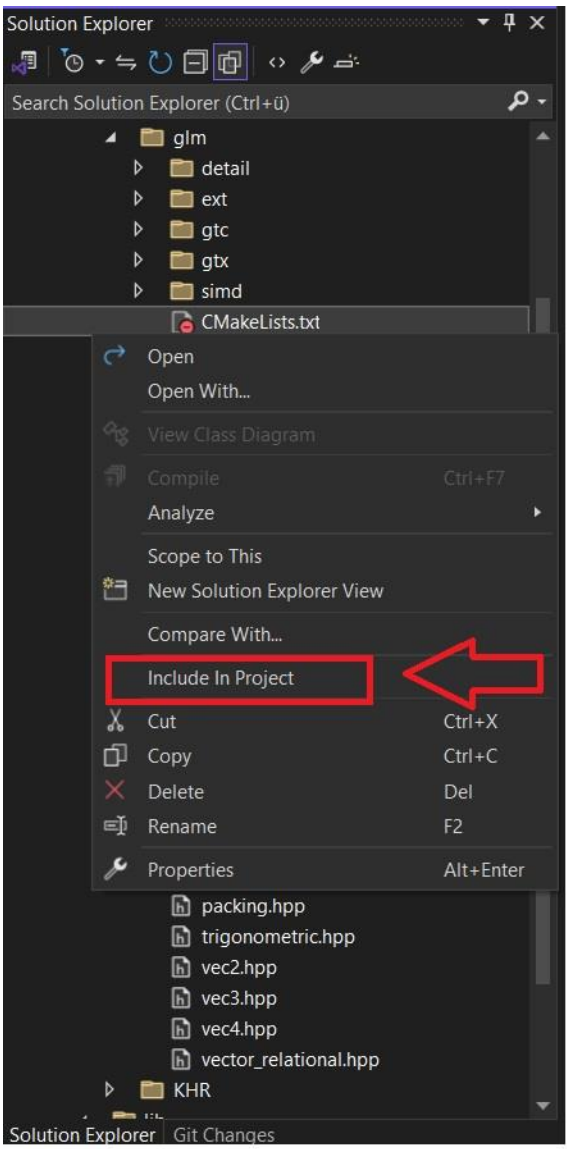


Рисунок 4.9 –Добовление файлов в проект

После выполнения этих действий приступаем к тестированию, чтобы убедиться в правильности подключения библиотек. Для этого следует ознакомиться с документацией на сайтах [3] и [4] где представлены обе библиотеки. Изучив примеры кода в документации, пишем собственный тестовый код, чтобы проверить, всё ли работает корректно.

```
#include <GLFW/glfw3.h>
#include <glm/vec3.hpp>
#include <glm/vec4.hpp>
#include <glm/mat4x4.hpp>
#include <glm/ext/matrix_transform.hpp>
#include <glm/ext/matrix_clip_space.hpp>
```

```
#include <glm/ext/scalar_constants.hpp>
#include <iostream>

glm::mat4 camera(float Translate, glm::vec2 const& Rotate)
{
    glm::mat4 Projection = glm::perspective(glm::pi<float>() * 0.25f, 4.0f / 3.0f,
0.1f, 100.f);
    glm::mat4 View = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f, -
Translate));
    View = glm::rotate(View, Rotate.y, glm::vec3(-1.0f, 0.0f, 0.0f));
    View = glm::rotate(View, Rotate.x, glm::vec3(0.0f, 1.0f, 0.0f));
    glm::mat4 Model = glm::scale(glm::mat4(1.0f), glm::vec3(0.5f));
    return Projection * View * Model;
}

int main(void)
{

    if (!glfwInit())
    {
        std::cerr << "GLFW initialization failed!" << std::endl;
        return -1;
    }

    GLFWwindow* window = glfwCreateWindow(640, 480, "GLM + GLFW
Test", NULL, NULL);
    if (!window)
    {
        std::cerr << "Window creation failed!" << std::endl;
        glfwTerminate();
        return -1;
    }

    glfwMakeContextCurrent(window);
    glm::vec2 rotation(0.5f, 1.0f);
    float translate = 2.0f;

    while (!glfwWindowShouldClose(window))
    {
        glClear(GL_COLOR_BUFFER_BIT);

        glm::mat4 MVP = camera(translate, rotation);

        std::cout << "MVP Matrix: \n";
```

```
        for (int i = 0; i < 4; ++i)
        {
            for (int j = 0; j < 4; ++j)
            {
                std::cout << MVP[i][j] << " ";
            }
            std::cout << std::endl;
        }


        glfwSwapBuffers(window);

        glfwPollEvents();
    }
    glfwTerminate();
    return 0;
}
```

После проверки видим, что наш код загружает пустое окно без ошибок. Это означает, что всё выполнено правильно (Рисунок 4.10).

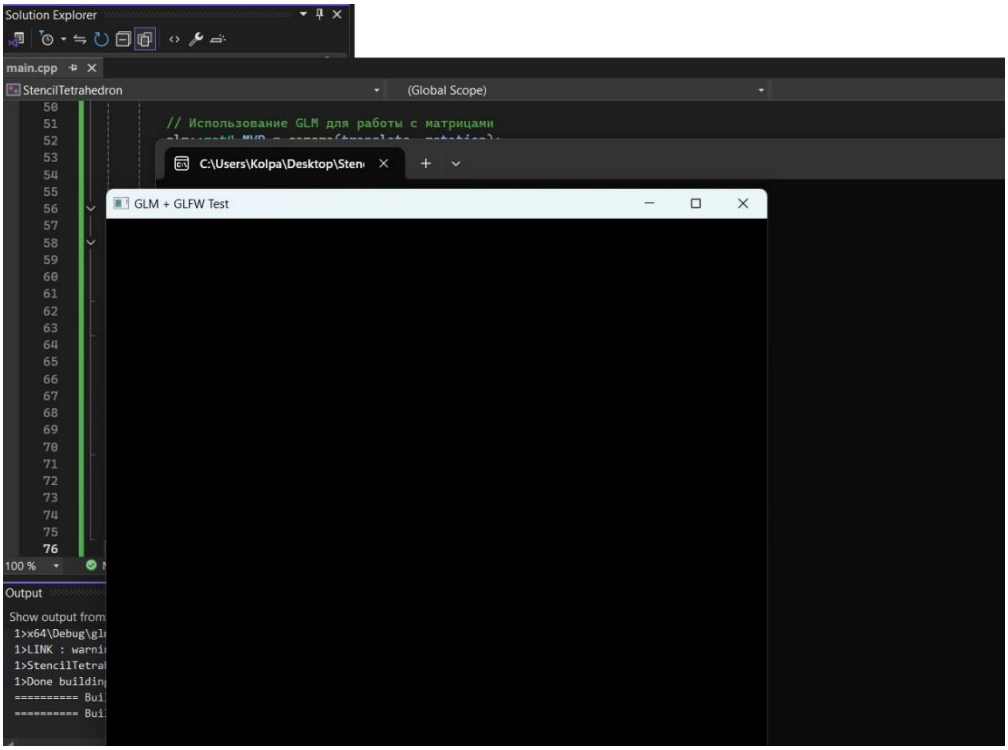


Рисунок 4.10 –Проверка работы библиотек и программы.

5 РЕАЛИЗАЦИЯ СЦЕНЫ В OPENGL

5.1 Инициализация GLFW и создание окна

Для начала работы с OpenGL мы выполнили инициализацию библиотеки GLFW, которая отвечает за создание окна и настройку контекста OpenGL. Также мы настроили параметры окна, такие как его размер (800x600 пикселей) и указали версию OpenGL, с которой будем работать (3.3). Это было выполнено с помощью вызовов функций glfwInit() и glfwCreateWindow(), после чего окно было сделано активным с помощью glfwMakeContextCurrent(). [3]

Для начала работы с OpenGL необходимо инициализировать GLFW. Мы проверяем успешность инициализации, чтобы убедиться, что среда была корректно настроена. Если инициализация завершается неудачно, программа выводит сообщение об ошибке и завершает выполнение.

```
if (!glfwInit()) {
    std::cerr << "Failed to initialize GLFW" << std::endl;
    return -1;
}
```

Затем мы настраиваем версию OpenGL и профиль контекста. В этом проекте мы используем OpenGL версии 3.3 с профилем ядра, чтобы обеспечить совместимость с современными графическими картами:

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_PROFILE,
GLFW_OPENGL_CORE_PROFILE);
```

После настройки версии OpenGL создается окно с указанными размерами — 800x600 пикселей. Мы также проверяем, удалось ли успешно создать окно:

```
GLFWwindow* window = glfwCreateWindow(800, 600, "Stencil Tetrahedron",
NULL, NULL);
if (window == NULL) {
    std::cerr << "Failed to create GLFW window" << std::endl;
    glfwTerminate();
    return -1;
}
```

5.2 Инициализация GLAD

После того как контекст OpenGL был успешно создан, мы использовали GLAD для загрузки всех функций OpenGL. Это позволяет нам напрямую вызывать необходимые функции OpenGL в коде. Инициализация была проведена с помощью вызова gladLoadGLLoader((GLADloadproc)glfwGetProcAddress). [5]

```
if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)) {
    std::cerr << "Failed to initialize GLAD" << std::endl;
    return -1;
}
```

Этот шаг позволяет программе использовать все функции OpenGL, так как GLAD автоматически загружает их с учетом версии и профиля OpenGL.

5.3 Создание и компиляция шейдеров

Далее мы создали два шейдера: вершинный и фрагментный. Для этого мы определили исходный код шейдеров внутри программы и использовали функции OpenGL для их компиляции. Шейдеры обрабатывают

геометрические данные (например, координаты вершин) и определяют, как каждый пиксель будет окрашен на экране. Мы создали шейдерную программу с помощью функций glCreateProgram(), glAttachShader() и glLinkProgram(). [5,6]

```
GLuint shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);
```

5.4 Основной цикл программы

Главный цикл программы (while (!glfwWindowShouldClose(window))) выполняет следующие задачи:

- Обработка ввода пользователя, включая возможность закрытия окна при нажатии клавиши ESC.
- Очистка буферов цвета, глубины и трафарета с помощью glClear(), чтобы каждый кадр рендерился с нуля.
- Отображение сцены и обмен буферов, чтобы рендеринг был плавным и отображался в окне в реальном времени.
- Обработка событий и обновление окна с помощью функций glfwPollEvents() и glfwSwapBuffers(). [6]

```
while (!glfwWindowShouldClose(window)) {

    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);……-→
```

5.5 Очистка ресурсов

После завершения работы программы мы освобождаем все занятые ресурсы, включая удаление шейдерной программы с помощью glDeleteProgram() и завершение работы GLFW вызовом glfwTerminate().

6 СОЗДАНИЕ ТЕТРАЭДРА С ОТВЕРСТИЯМИ ЧЕРЕЗ БУФЕР ТРАФАРЕТА

Для создания тетраэдра с отверстиями в гранях мы будем использовать буфер трафарета. Процесс включает следующие шаги:

- Рендеринг тетраэдра, обновляя буфер трафарета
- Использование буфера трафарета для вырезания отверстий
- Использование буфера трафарета для вырезания отверстий

6.1 Определение вершин тетраэдра

Сначала определим вершины тетраэдра:

```
// Вершины тетраэдра
float tetrahedronVertices[] = {
    // Вершины
     1.0f,  1.0f,  1.0f,  // Вершина A
    -1.0f, -1.0f,  1.0f,  // Вершина B
    -1.0f,  1.0f, -1.0f,  // Вершина C
     1.0f, -1.0f, -1.0f   // Вершина D
};

// Индексы граней тетраэдра
unsigned int tetrahedronIndices[] = {
    0, 1, 2, // Грань ABC
    0, 3, 1, // Грань ABD
    0, 2, 3, // Грань ACD
    1, 3, 2  // Грань BCD
};
```

6.2 Настройка VAO и VBO для тетраэдра

```
// Создание VAO и VBO для тетраэдра
GLuint tetraVAO, tetraVBO, tetraEBO;
glGenVertexArrays(1, &tetraVAO);
glGenBuffers(1, &tetraVBO);
glGenBuffers(1, &tetraEBO);
```

```
glBindVertexArray(tetraVAO);

// Вершины
glBindBuffer(GL_ARRAY_BUFFER, tetraVBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(tetrahedronVertices),
tetrahedronVertices, GL_STATIC_DRAW);

// Индексы
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, tetraEBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(tetrahedronIndices),
tetrahedronIndices, GL_STATIC_DRAW);

// Атрибуты вершин
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);

glBindVertexArray(0);
```

6.3 Отрисовка тетраэдра с использованием буфера трафарета

Я буду использовать буфер трафарета для создания отверстий в гранях тетраэдра. Для этого:

- Включаем буфер трафарета.
- Рисуем тетраэдр и обновляем буфер трафарета
- Используем буфер трафарета для ограничения области рендеринга отверстий.

```
// Включение буфера трафарета
glEnable(GL_STENCIL_TEST);

// Конфигурация теста трафарета
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);

// Настройка маски
glStencilFunc(GL_ALWAYS, 1, 0xFF);
glStencilMask(0xFF);

// Рисуем тетраэдр
```

```
glUseProgram(shaderProgram);
glm::mat4 model = glm::mat4(1.0f);
GLuint modelLoc = glGetUniformLocation(shaderProgram, "model");
GLuint viewLoc  = glGetUniformLocation(shaderProgram, "view");
GLuint projLoc  = glGetUniformLocation(shaderProgram, "projection");
GLuint colorLoc = glGetUniformLocation(shaderProgram, "color");

// Пример простого вида и проекции
glm::mat4 view = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f, -5.0f));
glm::mat4 projection = glm::perspective(glm::radians(45.0f),
    800.0f / 600.0f, 0.1f, 100.0f);

// Установка матриц
glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(view));
glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(projection));

// Установка цвета тетраэдра
glUniform4f(colorLoc, 0.0f, 1.0f, 0.0f, 1.0f);

// Рисуем тетраэдр, обновляя буфер трафарета
glStencilMask(0xFF);
glStencilFunc(GL_ALWAYS, 1, 0xFF);
glBindVertexArray(tetraVAO);
glDrawElements(GL_TRIANGLES, 12, GL_UNSIGNED_INT, 0);
glBindVertexArray(0); [5,6,7]
```

6.4 Вырезание отверстий

Предположим, что отверстия будут круглыми вырезами в гранях тетраэдра. Для этого мы будем рисовать окружности в буфере трафарета, чтобы обновить его и создать области вырезов.

```
// Отключение записи в буфер трафарета
glStencilMask(0x00);

// Установка функции трафарета для вырезания
glStencilFunc(GL_NOTEQUAL, 1, 0xFF);

// Рисуем отверстия (например, окружности)
int numSegments = 100;
float radius = 0.2f;
```

```
// Функция для рисования круга
auto drawCircle = [&](glm::vec3 center) {
    float vertices[300];
    int idx = 0;
    for(int i = 0; i <= numSegments; ++i){
        float angle = 2.0f * glm::pi<float>() * i / numSegments;
        vertices[idx++] = center.x + radius * cos(angle);
        vertices[idx++] = center.y + radius * sin(angle);
        vertices[idx++] = center.z;
    }

    GLuint circleVAO, circleVBO;
    glGenVertexArrays(1, &circleVAO);
    glGenBuffers(1, &circleVBO);

    glBindVertexArray(circleVAO);
    glBindBuffer(GL_ARRAY_BUFFER, circleVBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
GL_STATIC_DRAW);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float),
(void*)0);
    glEnableVertexAttribArray(0);

    // Рисуем круг как триангуляцию
    glDrawArrays(GL_TRIANGLE_FAN, 0, numSegments + 2);

    glDeleteBuffers(1, &circleVBO);
    glDeleteVertexArrays(1, &circleVAO);
};

// Пример: рисуем 4 отверстия на каждой вершине тетраэдра
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE); //
Отключение цветового буфера
glDepthMask(GL_FALSE); // Отключение буфера глубины

glUseProgram(shaderProgram);
glUniform4f(colorLoc, 1.0f, 0.0f, 0.0f, 1.0f); // Цвет не важен

// Пример координат для отверстий
glm::vec3 holeCenters[] = {
    glm::vec3(1.0f, 1.0f, 1.0f),
    glm::vec3(-1.0f, -1.0f, 1.0f),
    glm::vec3(-1.0f, 1.0f, -1.0f),
```

```
    glm::vec3(1.0f, -1.0f, -1.0f)
};

for(auto &center : holeCenters){
    drawCircle(center);
}

glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
glDepthMask(GL_TRUE);
```

7 РЕНДЕРИНГ ПОЛУПРОЗРАЧНОГО ЦИЛИНДРА ВНУТРИ ТЕТРАЭДРА

Полупрозрачный цилиндр можно смоделировать как тор. Для этого воспользуемся GLM для создания матриц трансформации и установим соответствующую прозрачность.

7.1 Определение геометрии тора

Для простоты, мы можем использовать готовые функции для генерации тороиды. Однако, поскольку OpenGL не предоставляет встроенных функций для этого, мы можем создать простую тороидальную сетку вручную или воспользоваться сторонними библиотеками, такими как GLUT или GLU для генерации цилиндра. Но для нашего проекта мы реализуем простую тороиду.

```
// Функция для генерации тороиды
struct Torus {
    std::vector<float> vertices;
    std::vector<unsigned int> indices;
    unsigned int numc, numt;
};

Torus generateTorus(float innerRadius, float outerRadius, unsigned int numc,
unsigned int numt){
    Torus torus;
    torus.numc = numc;
    torus.numt = numt;

    for(unsigned int i = 0; i <= numc; ++i){
        for(unsigned int j = 0; j <= numt; ++j){
            float s = (float)i / numc * 2.0f * glm::pi<float>();
            float t = (float)j / numt * 2.0f * glm::pi<float>();

            float x = (outerRadius + innerRadius * cos(t)) * cos(s);
            float y = (outerRadius + innerRadius * cos(t)) * sin(s);
            float z = innerRadius * sin(t);
```

```
            torus.vertices.push_back(x);
            torus.vertices.push_back(y);
            torus.vertices.push_back(z);
        }
    }

    for(unsigned int i = 0; i < numc; ++i){
        for(unsigned int j = 0; j < numt; ++j){
            unsigned int first = i * (numt + 1) + j;
            unsigned int second = first + numt + 1;

            torus.indices.push_back(first);
            torus.indices.push_back(second);
            torus.indices.push_back(first + 1);

            torus.indices.push_back(second);
            torus.indices.push_back(second + 1);
            torus.indices.push_back(first + 1);
        }
    }

    return torus;
}

// Генерация тороиды
Torus torus = generateTorus(0.5f, 1.0f, 30, 30);

// Создание VAO и VBO для тороиды
GLuint torusVAO, torusVBO, torusEBO;
glGenVertexArrays(1, &torusVAO);
glGenBuffers(1, &torusVBO);
glGenBuffers(1, &torusEBO);

glBindVertexArray(torusVAO);

glBindBuffer(GL_ARRAY_BUFFER, torusVBO);
glBufferData(GL_ARRAY_BUFFER, torus.vertices.size() * sizeof(float),
&torus.vertices[0], GL_STATIC_DRAW);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, torusEBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, torus.indices.size() *
sizeof(unsigned int), &torus.indices[0], GL_STATIC_DRAW);
```

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);

glBindVertexArray(0); [7]
```

7.2 Рендеринг тороиды

```
// Включение смешивания для полупрозрачности
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

// Настройка матриц модели
model = glm::mat4(1.0f);
// Пример трансформации (можно добавить вращение)
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));

// Установка цвета с прозрачностью
glUniform4f(colorLoc, 0.0f, 0.0f, 1.0f, 0.5f); // Полупрозрачный синий

// Рисуем тороиду
glBindVertexArray(torusVAO);
glDrawElements(GL_TRIANGLES, torus.indices.size(), GL_UNSIGNED_INT,
0);
glBindVertexArray(0);

glDisable(GL_BLEND);
```

8 АНИМАЦИЯ ВРАЩЕНИЯ СЦЕНЫ

Для вращения сцены по таймеру добавим переменную угла и обновляем ее в основном цикле.

```
float angle = 0.0f;

// В основном цикле перед рендерингом
while(!glfwWindowShouldClose(window)){
    // Ввод
    if(glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);

    // Обновление угла
    angle += 0.5f; // Скорость вращения
    if(angle > 360.0f)
        angle -= 360.0f;

    // Вращение модели
    glm::mat4 rotation = glm::rotate(glm::mat4(1.0f), glm::radians(angle),
glm::vec3(0.5f, 1.0f, 0.0f));
    glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(rotation));

    // Рендеринг
    glClearColor(0.1f, 0.1f, 0.1f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
GL_STENCIL_BUFFER_BIT);

    // Рендеринг тетраэдра
    glStencilMask(0xFF);
    glStencilFunc(GL_ALWAYS, 1, 0xFF);
    glUseProgram(shaderProgram);
    glUniform4f(colorLoc, 0.0f, 1.0f, 0.0f, 1.0f);
    glBindVertexArray(tetraVAO);
    glDrawElements(GL_TRIANGLES, 12, GL_UNSIGNED_INT, 0);
    glBindVertexArray(0);

    // Рендеринг отверстий
    glStencilMask(0x00);
    glStencilFunc(GL_ALWAYS, 0, 0xFF);
    glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
    glDepthMask(GL_FALSE);
```

```
    glUniform4f(colorLoc, 1.0f, 0.0f, 0.0f, 1.0f);
    for(auto &center : holeCenters){
        drawCircle(center);
    }
    glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
    glDepthMask(GL_TRUE);

    // Рендеринг тороиды
    glStencilMask(0x00);
    glStencilFunc(GL_ALWAYS, 0, 0xFF); // Неважно, но для ясности
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glUseProgram(shaderProgram);
    glUniform4f(colorLoc, 0.0f, 0.0f, 1.0f, 0.5f);
    glBindVertexArray(torusVAO);
    glDrawElements(GL_TRIANGLES, torus.indices.size(), GL_UNSIGNED_INT,
0);
    glBindVertexArray(0);
    glDisable(GL_BLEND);

    // Обмен буферов и обработка событий
    glfwSwapBuffers(window);
    glfwPollEvents();
} [6, 7]
```

9 ИСПОЛЬЗОВАНИЕ ТАЙМЕРА GLFW

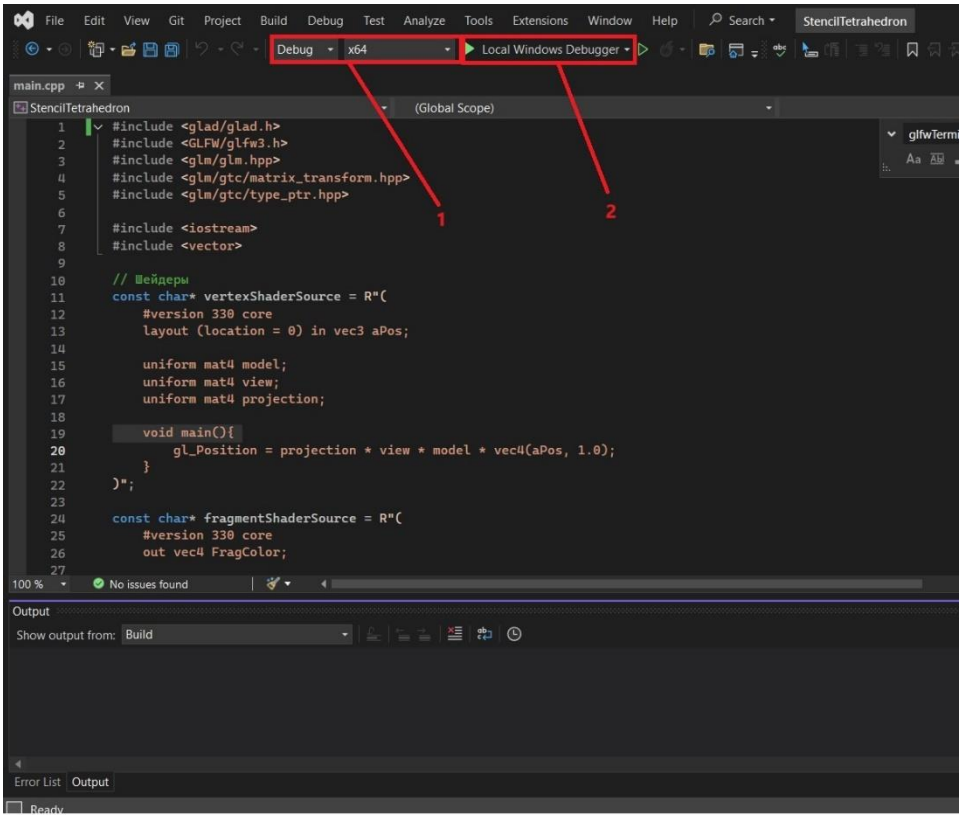
Для более точного управления временем я решила использовать функцию GLFW для отслеживания времени и обновления угла.

```
double previousTime = glfwGetTime();
while(!glfwWindowShouldClose(window)){
    double currentTime = glfwGetTime();
    double deltaTime = currentTime - previousTime;
    previousTime = currentTime;

    // Обновление угла с учетом deltaTime
    angle += 50.0f * deltaTime; // Скорость вращения в градусах в секунду
    if(angle > 360.0f)
        angle -= 360.0f;
```

10 РЕЗУЛЬТАТ РАБОТЫ ПРОГРАММЫ

После того как код завершен и среда разработки Visual Studio 2022 не выявила ошибок, можно приступить к сборке проекта. Для этого в верхнем меню IDE сначала выбираем вкладку Debug и x64, затем переходим во вкладку Build и в выпадающем меню выбираем пункт Build Solution, либо одновременно нажав клавиши Ctrl+Shift+B (Рисунки 10.1 и 10.2).



1 -Сборка решения в формате Debug и выбор архитектурной платформы;

2 -кнопка запуска отладчика

Рисунок 10.1–Сборка решения в формате Debug

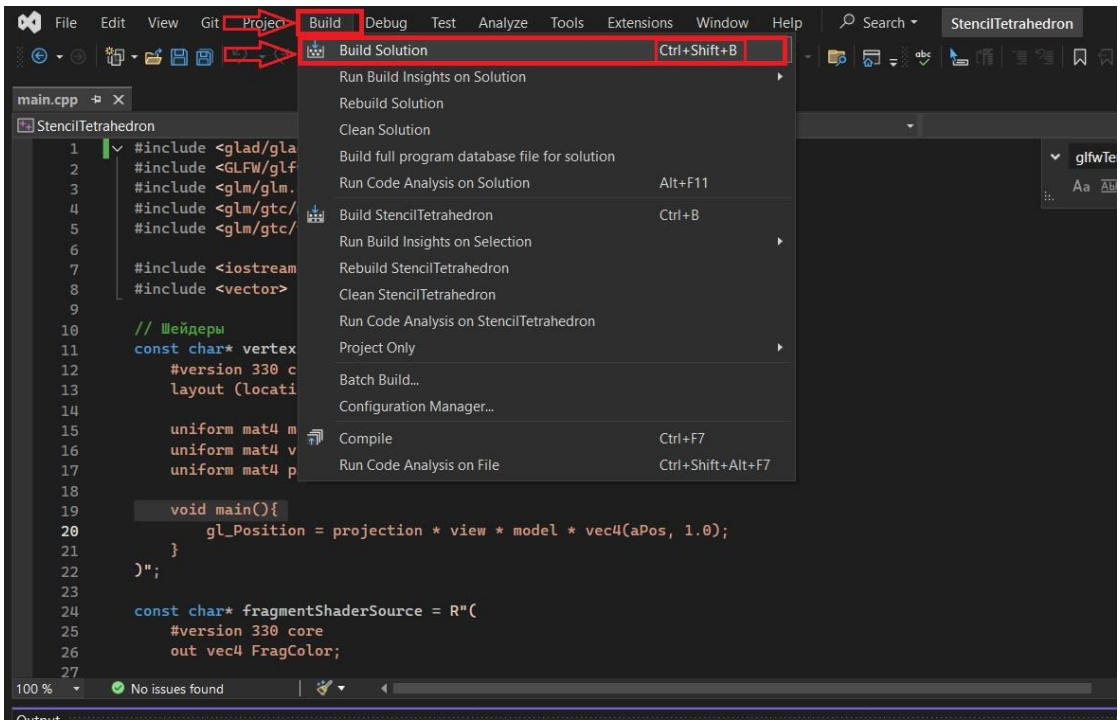


Рисунок 10.2 –Сборка проекта.

После сборки нашего проекта должно появится сообщение что сборка завершена без ошибок в IDE во вкладке Output (Рисунок 10.3)

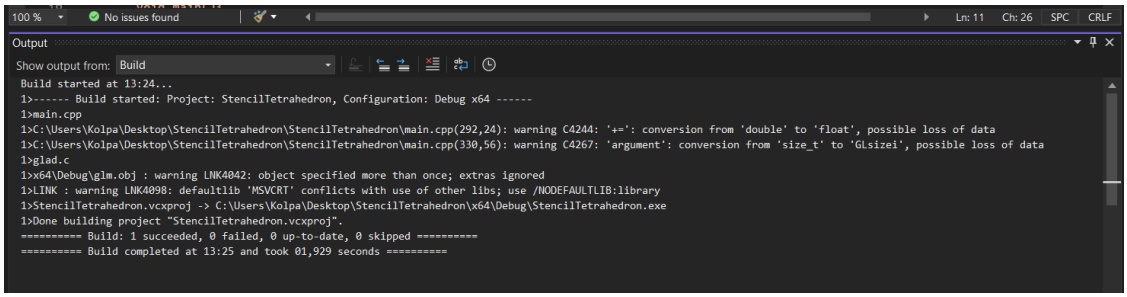


Рисунок 10.3 –сообщение что наша программа была собрана без ошибок.

Далее нажимаем на кнопку запуска отладчика в нашей IDE в меню Local Windows Debugger (Рисунок 10.1). После запуска отладчика мы видим, что программа работает корректно, без ошибок и перегрузки системы. Тетраэдр с полупрозрачным цилиндром также отображается правильно, вращаясь по таймеру (Рисунок 10.4).

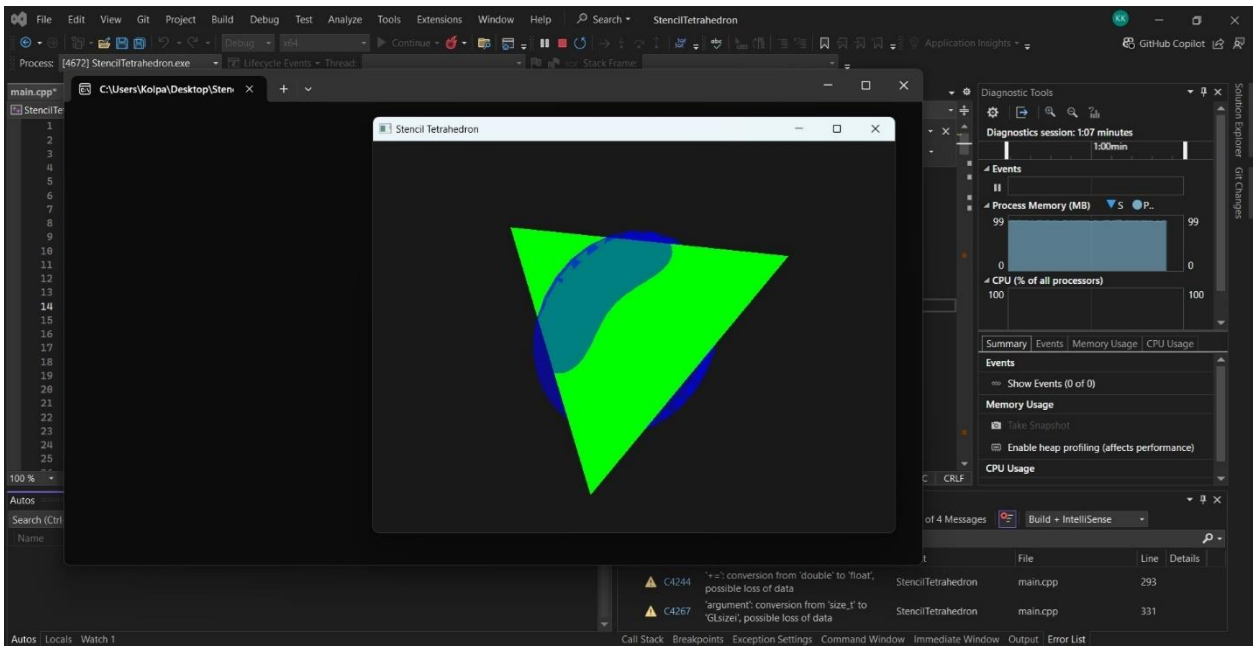


Рисунок 10.4 –Результат работы программы.

Итак, мы видим, что наша программа работает, и задание выполнено. Далее изменяем конфигурацию на Release в меню IDE (Рисунок 10.1) и снова собираем проект (Рисунок 10.2). В результате в окне Output получаем сообщение о том, что проект успешно собран без ошибок (Рисунок 10.5).

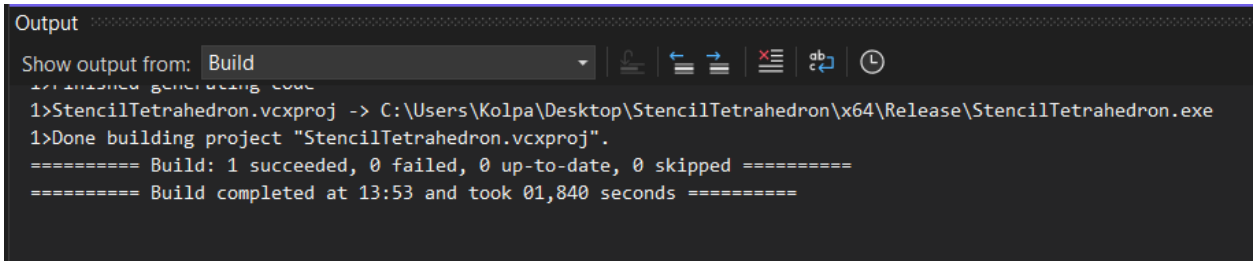


Рисунок 10.5 -Сборка Release прошла успешно

Путь к нашей программе .exe лежит по \StencilTetrahedron\x64\Release\StencilTetrahedron.exe

Выводы

В результате выполнения данной курсовой работы была создана интерактивная трехмерная сцена, использующая возможности буфера трафарета в OpenGL. Мы успешно реализовали основные задачи, поставленные перед нами:

1. **Использование буфера трафарета**: Мы научились применять буфер трафарета для создания отверстий в гранях тетраэдра. Этот подход показал свою эффективность для выполнения сложных графических задач, таких как маскирование и выборочные области рендеринга.
2. **Реализация полупрозрачного объекта**: внутри тетраэдра был размещен полупрозрачный цилиндр, описанный тором, что продемонстрировало использование прозрачности в OpenGL. Правильное использование смешивания (blending) позволило создать реалистичный эффект прозрачных объектов.
3. **Анимация сцены**: Мы добавили анимацию вращения сцены с использованием таймера, что сделало сцену динамичной и визуально более впечатляющей. Обновление угла вращения с учетом времени обеспечило плавное и реалистичное движение объектов
4. **Интеграция сторонних библиотек**: В ходе работы были успешно интегрированы и использованы сторонние библиотеки, такие как GLFW для работы с окнами, GLAD для загрузки OpenGL-функций и GLM для математических расчетов. Это позволило значительно упростить работу с графическим контекстом и математическими преобразованиями.
5. **Среда разработки**: Мы использовали Microsoft Visual Studio 2022, которая обеспечила удобные инструменты для разработки и отладки графических приложений на языке C++ с использованием OpenGL.

Данная работа дала нам опыт практического применения OpenGL для решения задач трёхмерной графики, использования буфера трафарета и

создания анимационных эффектов. Мы закрепили знания по настройке графической среды и реализации сложных визуальных эффектов, что важно для дальнейшего изучения компьютерной графики.

Список использованных Источников

1. GLAD — онлайн-генератор загрузки функций OpenGL [Электронный ресурс]. Режим доступа: **glad.dav1d.de** (дата обращения: 21.09.2024).

2. GLM — библиотека математических функций для OpenGL [Электронный ресурс]. Режим доступа: **sourceforge.net** (дата обращения: 21.09.2024).

3. GLFW — библиотека для создания окон и обработки ввода [Электронный ресурс]. Режим доступа: **glfw.org** (дата обращения: 21.09.2024).

4. GitHub — исходный код библиотеки GLM [Электронный ресурс]. Режим доступа: **github.com** (дата обращения: 21.09.2024).

5. Stack Overflow - сайт вопросов и ответов для программистов (Форум) [Электронный ресурс]. Режим доступа: **stackoverflow.com** (дата обращения: 21.09.2024).

6. Microsoft Learn OpenGL Documentation — Руководство по использованию OpenGL в Windows [Электронный ресурс]. Режим доступа: **learn.microsoft.com** (дата обращения: 22.09.2024).

7. LearnOpenGL — учебный ресурс по изучению OpenGL с примерами и руководствами [Электронный ресурс]. Режим доступа: learnopengl.com (дата обращения: 25.09.2024).

Приложение А Листинг Программы

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

#include <iostream>
#include <vector>

// Шейдеры
const char* vertexShaderSource = R"(
    #version 330 core
    layout (location = 0) in vec3 aPos;

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

    void main(){
        gl_Position = projection * view * model * vec4(aPos, 1.0);
    }
)";

const char* fragmentShaderSource = R"(
    #version 330 core
    out vec4 FragColor;

    uniform vec4 color;

    void main(){
        FragColor = color;
    }
)";

// Функция для компиляции шейдера
GLuint compileShader(GLenum type, const char* source){
    GLuint shader = glCreateShader(type);
    glShaderSource(shader, 1, &source, NULL);
    glCompileShader(shader);
    // Проверка ошибок
    int success;
    char infoLog[512];
```

```
    glGetShaderiv(shader, GL_COMPILE_STATUS, &success);
    if(!success){
        glGetShaderInfoLog(shader, 512, NULL, infoLog);
        std::cout << "ERROR::SHADER::COMPILATION_FAILED\n" << infoLog
<< std::endl;
    }
    return shader;
}

// Функция для создания шейдерной программы
GLuint createShaderProgram(){
    GLuint vertexShader = compileShader(GL_VERTEX_SHADER,
vertexShaderSource);
    GLuint fragmentShader = compileShader(GL_FRAGMENT_SHADER,
fragmentShaderSource);

    // Создание программы
    GLuint shaderProgram = glCreateProgram();
    glAttachShader(shaderProgram, vertexShader);
    glAttachShader(shaderProgram, fragmentShader);
    glLinkProgram(shaderProgram);
    // Проверка ошибок
    int success;
    char infoLog[512];
    glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
    if(!success){
        glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);
        std::cout << "ERROR::PROGRAM::LINKING_FAILED\n" << infoLog <<
std::endl;
    }
    // Удаление шейдеров
    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);
    return shaderProgram;
}

// Обработчик изменения размера окна
void framebuffer_size_callback(GLFWwindow* window, int width, int height){
    glViewport(0, 0, width, height);
}

// Структура для тороиды
struct Torus {
    std::vector<float> vertices;
```

```
    std::vector<unsigned int> indices;
    unsigned int numc, numt;
};

// Функция генерации тороиды
Torus generateTorus(float innerRadius, float outerRadius, unsigned int numc,
unsigned int numt){
    Torus torus;
    torus.numc = numc;
    torus.numt = numt;

    for(unsigned int i = 0; i <= numc; ++i){
        for(unsigned int j = 0; j <= numt; ++j){
            float s = (float)i / numc * 2.0f * glm::pi<float>();
            float t = (float)j / numt * 2.0f * glm::pi<float>();

            float x = (outerRadius + innerRadius * cos(t)) * cos(s);
            float y = (outerRadius + innerRadius * cos(t)) * sin(s);
            float z = innerRadius * sin(t);

            torus.vertices.push_back(x);
            torus.vertices.push_back(y);
            torus.vertices.push_back(z);
        }
    }

    for(unsigned int i = 0; i < numc; ++i){
        for(unsigned int j = 0; j < numt; ++j){
            unsigned int first = i * (numt + 1) + j;
            unsigned int second = first + numt + 1;

            torus.indices.push_back(first);
            torus.indices.push_back(second);
            torus.indices.push_back(first + 1);

            torus.indices.push_back(second);
            torus.indices.push_back(second + 1);
            torus.indices.push_back(first + 1);
        }
    }

    return torus;
}
```

```
int main(){
    // Инициализация GLFW
    if(!glfwInit()){
        std::cerr << "Failed to initialize GLFW" << std::endl;
        return -1;
    }

    // Настройка GLFW
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE,
GLFW_OPENGL_CORE_PROFILE);

    // Создание окна
    GLFWwindow* window = glfwCreateWindow(800, 600, "Stencil Tetrahedron",
NULL, NULL);
    if(window == NULL){
        std::cerr << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    }
    glfwMakeContextCurrent(window);
    // Установка обработчика изменения размера
    glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);

    // Инициализация GLAD
    if(!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)){
        std::cerr << "Failed to initialize GLAD" << std::endl;
        return -1;
    }

    // Включение теста глубины и буфера трафарета
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_STENCIL_TEST);

    // Создание шейдерной программы
    GLuint shaderProgram = createShaderProgram();

    // Вершины тетраэдра
    float tetrahedronVertices[] = {
        // Вершины
        1.0f,  1.0f,  1.0f,  // Вершина A
       -1.0f, -1.0f,  1.0f,  // Вершина B
       -1.0f,  1.0f, -1.0f,  // Вершина C
```

```
        1.0f, -1.0f, -1.0f   // Вершина D
    };

    // Индексы граней тетраэдра
    unsigned int tetrahedronIndices[] = {
        0, 1, 2, // Грань ABC
        0, 3, 1, // Грань ABD
        0, 2, 3, // Грань ACD
        1, 3, 2  // Грань BCD
    };

    // Создание VAO и VBO для тетраэдра
    GLuint tetraVAO, tetraVBO, tetraEBO;
    glGenVertexArrays(1, &tetraVAO);
    glGenBuffers(1, &tetraVBO);
    glGenBuffers(1, &tetraEBO);

    glBindVertexArray(tetraVAO);

    // Вершины
    glBindBuffer(GL_ARRAY_BUFFER, tetraVBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(tetrahedronVertices),
tetrahedronVertices, GL_STATIC_DRAW);

    // Индексы
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, tetraEBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(tetrahedronIndices),
tetrahedronIndices, GL_STATIC_DRAW);

    // Атрибуты вершин
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float),
(void*)0);
    glEnableVertexAttribArray(0);

    glBindVertexArray(0);

    // Генерация тороиды
    Torus torus = generateTorus(0.3f, 0.8f, 30, 30);

    // Создание VAO и VBO для тороиды
    GLuint torusVAO, torusVBO, torusEBO;
    glGenVertexArrays(1, &torusVAO);
    glGenBuffers(1, &torusVBO);
    glGenBuffers(1, &torusEBO);
```

```
    glBindVertexArray(torusVAO);

    glBindBuffer(GL_ARRAY_BUFFER, torusVBO);
    glBufferData(GL_ARRAY_BUFFER, torus.vertices.size() * sizeof(float),
&torus.vertices[0], GL_STATIC_DRAW);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, torusEBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, torus.indices.size() *
sizeof(unsigned int), &torus.indices[0], GL_STATIC_DRAW);

    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float),
(void*)0);
    glEnableVertexAttribArray(0);

    glBindVertexArray(0);

    // Пример координат для отверстий (центры граней)
    glm::vec3 holeCenters[] = {
        glm::vec3(1.0f, 1.0f, 1.0f),
        glm::vec3(-1.0f, -1.0f, 1.0f),
        glm::vec3(-1.0f, 1.0f, -1.0f),
        glm::vec3(1.0f, -1.0f, -1.0f)
    };

    // Функция для рисования круга
    auto drawCircle = [&](glm::vec3 center) {
        int numSegments = 100;
        float radius = 0.2f;
        std::vector<float> vertices;
        for(int i = 0; i <= numSegments; ++i){
            float angle = 2.0f * glm::pi<float>() * i / numSegments;
            vertices.push_back(center.x + radius * cos(angle));
            vertices.push_back(center.y + radius * sin(angle));
            vertices.push_back(center.z);
        }

        GLuint circleVAO, circleVBO;
        glGenVertexArrays(1, &circleVAO);
        glGenBuffers(1, &circleVBO);

        glBindVertexArray(circleVAO);
        glBindBuffer(GL_ARRAY_BUFFER, circleVBO);
```

```
        glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(float),
&vertices[0], GL_STATIC_DRAW);
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float),
(void*)0);
        glEnableVertexAttribArray(0);

        glDrawArrays(GL_TRIANGLE_FAN, 0, numSegments + 2);

        glDeleteBuffers(1, &circleVBO);
        glDeleteVertexArrays(1, &circleVAO);
    };

    // Установка матриц вида и проекции
    glm::mat4 view = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f, -5.0f));
    glm::mat4 projection = glm::perspective(glm::radians(45.0f),
        800.0f / 600.0f, 0.1f, 100.0f);

    // Получение местоположений uniform-переменных
    glUseProgram(shaderProgram);
    GLuint modelLoc = glGetUniformLocation(shaderProgram, "model");
    GLuint viewLoc  = glGetUniformLocation(shaderProgram, "view");
    GLuint projLoc  = glGetUniformLocation(shaderProgram, "projection");
    GLuint colorLoc = glGetUniformLocation(shaderProgram, "color");

    glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(view));
    glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(projection));

    // Включение смешивания для полупрозрачности
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

    // Переменная для вращения
    float angle = 0.0f;
    double previousTime = glfwGetTime();

    // Основной цикл
    while(!glfwWindowShouldClose(window)){
        // Ввод
        if(glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
            glfwSetWindowShouldClose(window, true);

        // Обновление времени и угла вращения
        double currentTime = glfwGetTime();
        double deltaTime = currentTime - previousTime;
```

```
        previousTime = currentTime;
        angle += 50.0f * deltaTime; // 50 градусов в секунду
        if(angle > 360.0f)
            angle -= 360.0f;

        // Вращение модели
        glm::mat4 rotation = glm::rotate(glm::mat4(1.0f), glm::radians(angle),
glm::vec3(0.5f, 1.0f, 0.0f));
        glUseProgram(shaderProgram);
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(rotation));

        // Очистка буферов
        glClearColor(0.1f, 0.1f, 0.1f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
GL_STENCIL_BUFFER_BIT);

        // Рендеринг тетраэдра в буфер трафарета
        glStencilMask(0xFF);
        glStencilFunc(GL_ALWAYS, 1, 0xFF);
        glUniform4f(colorLoc, 0.0f, 1.0f, 0.0f, 1.0f); // Зеленый цвет
        glBindVertexArray(tetraVAO);
        glDrawElements(GL_TRIANGLES, 12, GL_UNSIGNED_INT, 0);
        glBindVertexArray(0);

        // Рендеринг отверстий
        glStencilMask(0x00);
        glStencilFunc(GL_ALWAYS, 0, 0xFF);
        glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
        glDepthMask(GL_FALSE);
        glUniform4f(colorLoc, 1.0f, 0.0f, 0.0f, 1.0f); // Цвет не важен
        for(auto &center : holeCenters){
            drawCircle(center);
        }
        glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
        glDepthMask(GL_TRUE);

        // Рендеринг тороиды (полупрозрачный цилиндр)
        glStencilMask(0x00);
        glStencilFunc(GL_ALWAYS, 0, 0xFF); // Не влияет на трафарет
        glUniform4f(colorLoc, 0.0f, 0.0f, 1.0f, 0.5f); // Полупрозрачный синий
        glBindVertexArray(torusVAO);
        glDrawElements(GL_TRIANGLES, torus.indices.size(),
GL_UNSIGNED_INT, 0);
        glBindVertexArray(0);
```

```
        // Обмен буферов и обработка событий
        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    // Очистка ресурсов
    glDeleteVertexArrays(1, &tetraVAO);
    glDeleteBuffers(1, &tetraVBO);
    glDeleteBuffers(1, &tetraEBO);
    glDeleteVertexArrays(1, &torusVAO);
    glDeleteBuffers(1, &torusVBO);
    glDeleteBuffers(1, &torusEBO);
    glDeleteProgram(shaderProgram);

    glfwTerminate();
    return 0;
}
```