МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторным работам №6
по дисциплине «Компьютерная графика»
Тема: Реализация трехмерного объекта
с использованием библиотеки OpenGL.

Студентка гр. 1304 _____ Чернякова В.А.

Студентка гр. 1304 _____ Ярусова Т.В.

Преподаватель _____ Герасимова Т.В.

Санкт-Петербург
2024

Цель работы.

- Изучение способов построения трехмерных объектов в OpenGL.

- Изучение способов применения шейдеров в программах OpenGL для отображения трехмерных объектов.

Задание.

Вариант 226:

Написать программу, рисующую проекцию трехмерного каркасного объекта.

Требования

1. Грани объектов рисуются с помощью доступных функций рисования отрезка в координатах окна. При этом использовать шейдеры GLSL и OpenGL;

2. Вывод объектов с прорисовкой невидимых граней;

3. Перемещения, повороты и масштабирование объектов по каждой из осей независимо от остальных.

4. Генерация объектов с заданной мелкостью разбиения.

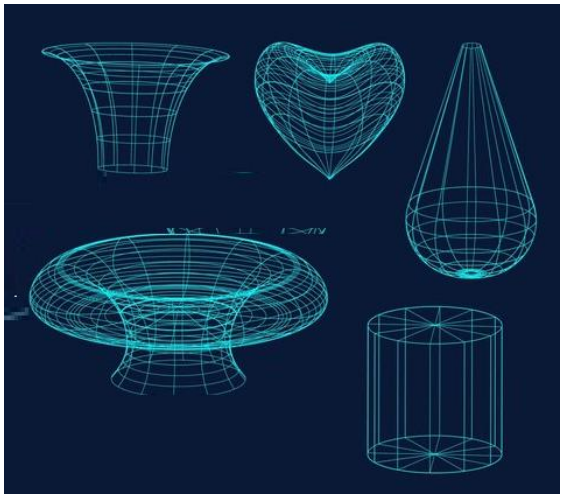
5. При запуске программы объекты сразу должны быть хорошо виден.

6. Пользователь имеет возможность вращать фигур (2 степени свободы) и изменять параметры фигур.

7. Возможно изменять положение наблюдателя.

8. Нарисовать оси системы координат.

9. Все варианты требований могут быть выбраны интерактивно.



Выполнение работы.

Программа была написана на языке программирования C++ с применением фреймворка Qt.

Для удобства работы с шейдерными программами был реализован класс GLShaderProgram. Данный класс при конструировании принимает словарь, в котором ключом является тип шейдера, а значением — путь до шейдера. Класс имеет метод bool init(), в котором происходит компиляция и линковка шейдеров. Если компиляция и линковка прошли успешно, метод вернет true, иначе — false.

```
#include "glshaderprogram.h"

GLShaderProgram::GLShaderProgram(
    const QMap<QOpenGLShader::ShaderType, QString>
    & shaders, QObject *parent):
    QOpenGLShaderProgram{parent},
    shaders_{shaders}
{}


bool GLShaderProgram::init() {
    if (isInitialized()) { return initialized_; }

    initialized_ = true;
    for (const auto& [type, path]: shaders_.asKeyValueRange()) {
        initialized_ = initialized_ && addShaderFromSourceFile(type, path);
    }
    initialized_ = initialized_ && link() && bind();

    return initialized_;
}


bool GLShaderProgram::isInitialized() const {
    return initialized_;
}
```

Также был реализован класс GLVertexObject для работы с буферами OpenGL. Класс также содержит метод bool init(), в котором создаются необходимые для работы буферы: VAO, VBO и EBO. Также класс имеет шаблонный метод void loadVertices(const QVector<VertexDataType>& vertices, const QVector<IndexDataType>& indices), где VertexDataType и IndexDataType — шаблонные параметры. Метод используется для записи в буфер VBO данных из vertices, а также, при необходимости, индексов в EBO из indices. Также класс имеет метод void setupVertexAttribute(QOpenGLFunctions* painter, GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const void *offset),

который устанавливает расположение аттрибута вершинного шейдера в элементе из VBO.

```
#include "glvertexobject.h"

#include <QOpenGLFunctions>
#include <QVector3D>

GLVertexObject::GLVertexObject():
    vertexBuffer_{QOpenGLBuffer::VertexBuffer},
    elementBuffer_{QOpenGLBuffer::IndexBuffer}
{}

GLVertexObject::~GLVertexObject() {
    elementBuffer_.destroy();
    vertexBuffer_.destroy();
    vertexArray_.destroy();
}

bool GLVertexObject::init() {
    if (initialized_) {
        return initialized_;
    }

    vertexArray_.create();
    vertexBuffer_.create();
    elementBuffer_.create();

    initialized_ = true;
    return initialized_;
}

void GLVertexObject::setupVertexAttribute(
    QOpenGLFunctions* painter,
    GLuint index,
    GLint size,
    GLenum type,
    GLboolean normalized,
    GLsizei stride,
    const void *offset
)
{
    painter->glEnableVertexAttribArray(index);
    painter->glVertexAttribPointer(index, size, type, normalized, stride, off-
set);
}

QOpenGLVertexArrayObject& GLVertexObject::vao() {
    return vertexArray_;
}

const QOpenGLBuffer& GLVertexObject::vbo() const {
    return vertexBuffer_;
}

const QOpenGLBuffer& GLVertexObject::ebo() const {
    return elementBuffer_;
}

bool GLVertexObject::isInitialized() const {
    return initialized_;
}
```

```
void GLVertexObject::bind_vao() {
    if (initialized_) {
        vertexArray_.bind();
    }
}

void GLVertexObject::unbind_vao() {
    if (initialized_) {
        vertexArray_.release();
    }
}
```

Для отрисовки используются две шейдерные программы и два буфера: для фигуры и для осей координат. Два буфера используется в связи с тем, что есть необходимость данные о вершинах фигуры хранить отдельно от данных о вершинах для осей (логическое разделение).

Фигурные шейдеры.

Вершинный шейдер. Данный шейдер принимает на вход вершину с её позицией в трехмерном пространстве (aPos), применяет к ней операции поворота (rotation), масштабирования (scale) и трансляции (translation), и затем передает преобразованную позицию в видео-память OpenGL для последующего рендеринга.

Каждая из трех матриц (rotation, scale, translation) является uniform-переменной, что означает, что они могут установлены извне, из кода на языке C++ приложения OpenGL.

Этот код служит для трансформации вершин, что позволяет создавать анимации, вращения объектов, масштабирование и перемещение их в трехмерном пространстве перед отображением на экране.

```
#version 460 core

layout (location = 0) in vec3 aPos;

uniform mat4 rotation;
uniform mat4 scale;
uniform mat4 translation;

void main()
{
    gl_Position = rotation * translation * scale * vec4(aPos, 1.0);
}
```

Фрагментный шейдер. Данный шейдер отвечает за окраску каждого отдельного фрагмента (точки) на изображении, которое формируется на экране.

В этом конкретном примере, каждый фрагмент будет окрашен в цвет с координатами (0.7, 0.7, 1.0, 1.0), что представляет собой оттенок синего цвета.

Значение vec4(0.7, 0.7, 1.0, 1.0) задает цвет фрагмента в формате RGBA (красный, зеленый, синий, альфа), где каждый компонент цвета находится в диапазоне от 0 до 1. Таким образом, данный цвет соответствует оттенку синего с небольшими оттенками красного и зеленого.

Затем это значение цвета присваивается переменной FragColor, которая представляет собой выходную переменную шейдера, указывая OpenGL, какой цвет следует использовать для окраски данного фрагмента.

```
#version 460 core

out vec4 FragColor;

void main()
{
    FragColor = vec4( 0.7, 0.7, 1.0, 1.0);
}
```

Осевые шейдеры.

Вершинный шейдер. Данный шейдер вершин выполняет преобразование каждой вершины перед её рендерингом. В данном случае, шейдер принимает на вход позицию вершины (aPos) и её цвет (aColor), а также матрицу поворота (rotation) как uniform-переменную.

Далее, создается матрица трансляции (translation), которая задает перемещение вершин в пространстве. Затем, позиция каждой вершины преобразуется путем умножения матриц трансляции (translation), поворота (rotation) и текущей позиции вершины (aPos).

Полученная преобразованная позиция вершины устанавливается в специальную переменную gl_Position, которая определяет позицию вершины после всех преобразований и передается в последующие этапы рендеринга.

Кроме того, цвет вершины (aColor) просто передается через переменную color для использования во фрагментном шейдере (fragment shader).

```
#version 460 core

layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;

out vec3 color;
```

```
uniform mat4 rotation;

void main()
{
    mat4 translation;
    translation[0] = vec4(1.0, 0.0, 0.0, 0.0);
    translation[1] = vec4(0.0, 1.0, 0.0, 0.0);
    translation[2] = vec4(0.0, 0.0, 1.0, 0.0);
    translation[3] = vec4(0.8, 0.0, 0.0, 1.0);

    gl_Position = translation * rotation * vec4(aPos, 1.0);
    color = aColor;
}
```

Фрагментный шейдер. Данный шейдер управляет окраской каждого отдельного фрагмента (точки) на изображении, формируемом на экране. В данном случае, он принимает цвет фрагмента (color), который был передан из вершинного шейдера (vertex shader), и устанавливает его как выходной цвет для данного фрагмента.

Функция vec4(color, 1.0) создает 4-компонентный вектор, используя значения компонент цвета (RGB) из переменной color, и устанавливает альфа-канал (прозрачность) в 1.0, что означает полную непрозрачность.

Затем это значение цвета присваивается переменной FragColor, которая представляет собой специальную переменную, определяющую цвет фрагмента. Этот цвет будет использован для окраски фрагмента на экране.

```
#version 460 core

in vec3 color;
out vec4 FragColor;

void main()
{
    FragColor = vec4( color, 1.0);
}
```

Основная работа с объектом выполняется в классе GLScene. В классе создается два экземпляра класса GLShaderProgram и два экземпляра класса GLVertexObject для отрисовки самой фигуры икоординатных осей.

Создание и загрузка шейдерных программ для отрисовки фигур и осей (в методе createShaderPrograms()).

Инициализация OpenGL контекста и установка параметров OpenGL, таких как цвет фона, режим отрисовки, активация буфера глубины и лицевая/обратная отсечка (в методе initializeGL()).

Определение функций для изменения параметров сцены, таких как тип фигуры, коэффициент фрагментации (в методах setType(), setFragmentationFactor()).

В программе присутствует возможность управлять отрисовкой фигуры: можно изменить растяжение фигуры вдоль каждой из осей независимо, изменять поворот вокруг каждой из осей, а также смещать фигуру вдоль любой из осей.

Отрисовка сцены с помощью фрагментного и вершинного шейдеров. Фигуры, такие как тор, гиперболоид и цилиндр, отрисовываются в зависимости от установленного типа фигуры (в методе paintGL()). Оси координат также отрисовываются для наглядности.

Подготовка вершин для отрисовки осей координат (в методе prepareAxes()).

```
#include "glscene.h"

#include "hyperboloid.h"
#include "torus.h"
#include "cylinder.h"

#include <iostream>

GLScene::GLScene(QWidget* parent):QOpenGLWidget{ parent }{}

void GLScene::initializeGL() {
    QColor bgc(0x2E, 0x2E, 0x2E);
    initializeOpenGLFunctions();
    glClearColor(bgc.redF(), bgc.greenF(), bgc.blueF(), bgc.alphaF());
    createShaderPrograms();
    if (!figureShaderProgram_->init() || !axesShaderProgram_->init()) {
        std::cerr << "Unable to initialize Shader Programs" << std::endl;
        std::cerr << "Figure Shader Program log: " << figureShaderProgram_-
>log().toStdString() << std::endl;
        std::cerr << "Axes Shader Program log: " << axesShaderProgram_-
>log().toStdString() << std::endl;
        return;
    }
    figureVertexObject_.init();
    axesVertexObject_.init();
    setRotation(0.f, 0.f, 0.f);
    setScale(1.f, 1.f, 1.f);
    setTranslation(0.f, 0.f, 0.f);
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);
}

void GLScene::resizeGL(int w, int h) {
    _width = w;
    _height = h;
```

```
}

void GLScene::paintGL() {
    figureShaderProgram_->bind();
    figureVertexObject_.bind_vao();

    glLineWidth(1.0f);

    if (type_ == "Torus") {
        torus(0.2, 0.3, fragmentationFactor_ * 15, fragmentationFactor_ * 15);
    }

    if (type_ == "Hyperboloid") {
        hyperboloid(0.1, 0.1, 0.35, fragmentationFactor_ * 10, fragmentationFac-
tor_ * 10);
    }

    if (type_ == "Cylinder") {
        cylinder(0.25, 1.0, fragmentationFactor_ * 10);
    }

    prepareAxes();
    loadAxes();

    {
        int rotationMatrixLocation = figureShaderProgram_-
>uniformLocation("rotation");
        figureShaderProgram_->setUniformValue(rotationMatrixLocation, rotation-
Matrix_);

        int scaleMatrixLocation = figureShaderProgram_-
>uniformLocation("scale");
        figureShaderProgram_->setUniformValue(scaleMatrixLocation, scale-
Matrix_);

        int translationMatrixLocation = figureShaderProgram_-
>uniformLocation("translation");
        figureShaderProgram_->setUniformValue(translationMatrixLocation, trans-
lationMatrix_);
    }

    figureVertexObject_.unbind_vao();
    figureShaderProgram_->release();

    axesShaderProgram_->bind();
    axesVertexObject_.bind_vao();

    glLineWidth(3.0f);
    {
        int rotationMatrixLocation = axesShaderProgram_-
>uniformLocation("rotation");
        axesShaderProgram_->setUniformValue(rotationMatrixLocation, rotationMa-
trix_);
    }
    glDrawArrays(GL_LINES, 0, 6);

    axesVertexObject_.unbind_vao();
    axesShaderProgram_->release();
}

void GLScene::createShaderPrograms() {
    QMap<QOpenGLShader::ShaderType, QString> figureShaders;
    QMap<QOpenGLShader::ShaderType, QString> axesShaders;
```

```
    figureShaders[QOpenGLShader::Vertex] = ":/shaders/figure.vs";
    figureShaders[QOpenGLShader::Fragment] = ":/shaders/figure.fs";

    axesShaders[QOpenGLShader::Vertex] = ":/shaders/axes.vs";
    axesShaders[QOpenGLShader::Fragment] = ":/shaders/axes.fs";

    figureShaderProgram_ = new GLShaderProgram(figureShaders, this);
    axesShaderProgram_ = new GLShaderProgram(axesShaders, this);
}

void GLScene::loadAxes() {
    axesVertexObject_.bind_vao();
    axesVertexObject_.loadVertices(axesVertices_);
    axesVertexObject_.setupVertexAttribute(this, 0, 3, GL_FLOAT, GL_TRUE,
sizeof(QVector3D) * 2, nullptr);
    axesVertexObject_.setupVertexAttribute(this, 1, 3, GL_FLOAT, GL_TRUE,
sizeof(QVector3D) * 2, reinterpret_cast<void*>(sizeof(QVector3D)));
    axesVertexObject_.unbind_vao();
}

void GLScene::setType(std::string type) {
    type_ = type;
    update();
}

void GLScene::setFragmentationFactor(GLuint factor) {
    fragmentationFactor_ = factor;
    update();
}

void GLScene::setRotation(GLdouble x, GLdouble y, GLdouble z) {
    rotationMatrix_.setToIdentity();
    rotationMatrix_.rotate(x, 1.f, 0.f, 0.f);
    rotationMatrix_.rotate(y, 0.f, 1.f, 0.f);
    rotationMatrix_.rotate(z, 0.f, 0.f, 1.f);

    update();
}

void GLScene::setScale(GLdouble x, GLdouble y, GLdouble z) {
    scaleMatrix_.setToIdentity();
    scaleMatrix_.scale(x, y, z);

    update();
}

void GLScene::setTranslation(GLdouble x, GLdouble y, GLdouble z) {
    translationMatrix_.setToIdentity();
    translationMatrix_.translate(x, y, z);

    update();
}

void GLScene::prepareAxes() {
    axesVertices_.clear();

    axesVertices_.push_back({
        { 0.0, 0.0, 0.0 },
        { 0.8, 0.2, 0.2 }
    });
    axesVertices_.push_back({
        { 0.1, 0.0, 0.0 },
```

```
        { 0.8, 0.2, 0.2 }
    });

    axesVertices_.push_back({
        { 0.0, 0.0, 0.0 },
        { 0.2, 0.8, 0.2 }
    });
    axesVertices_.push_back({
        { 0.0, 0.1, 0.0 },
        { 0.2, 0.8, 0.2 }
    });

    axesVertices_.push_back({
        { 0.0, 0.0, 0.0 },
        { 0.2, 0.2, 0.8 }
    });
    axesVertices_.push_back({
        { 0.0, 0.0, 0.1 },
        { 0.2, 0.2, 0.8 }
    });
}
```

Разберем детальнее коды, отвечающией за отрисовку каркасных объектов.

Цилиндр.

Представленный ниже код реализует функцию cylinder::cylinder(), которая создает геометрию цилиндра и рисует его с помощью набора вызовов OpenGL.

Создание вершин цилиндра:

- Цилиндр состоит из двух круговых оснований и боковой поверхности.
- Для создания круговых оснований используется цикл, который проходит через каждый угол от 0 до 2π (полный оборот), делая шаги на основе количества сегментов.
- Для каждого угла вычисляются координаты вершины с помощью тригонометрических функций (cos и sin) для радиуса и угла.
- Для вершин верхнего основания устанавливается z-координата в половину высоты цилиндра, а для нижнего - в отрицательную половину.

Отрисовка круговых оснований:

- Круговые основания отрисовываются с помощью GL_LINE_LOOP, что позволяет соединить вершины в замкнутый контур.

Отрисовка боковой поверхности:

- Боковая поверхность цилиндра отрисовывается с помощью набора вертикальных линий, соединяющих соответствующие вершины на верхнем и нижнем основаниях.
- Для этого используется цикл, который проходит через каждый угол на верхнем основании, и соединяет соответствующие вершины на верхнем и нижнем основаниях с помощью GL_LINES.
- Кроме того, соединяются вершины между собой на верхнем и нижнем основаниях, чтобы создать ребра боковой поверхности.

```
#include "cylinder.h"

cylinder::cylinder(float radius, float height, int segments) {
    QVector<QVector3D> vertices;

    float angleDelta = 2.0f * M_PI / segments;

    for (int i = 0; i < segments; ++i) {
        float angle = i * angleDelta;
        float x = radius * cos(angle);
        float y = radius * sin(angle);
        float z = height / 2.0f;
        vertices.push_back(QVector3D(x, y, z));
    }

    for (int i = 0; i < segments; ++i) {
        float angle = i * angleDelta;
        float x = radius * cos(angle);
        float y = radius * sin(angle);
        float z = -height / 2.0f;
        vertices.push_back(QVector3D(x, y, z));
    }

    glBegin(GL_LINE_LOOP);
    for (int i = 0; i < segments; ++i) {
        QVector3D v = vertices[i];
        glVertex3f(v.x(), v.y(), v.z());
    }
    glEnd();

    glBegin(GL_LINE_LOOP);
    for (int i = 0; i < segments; ++i) {
        QVector3D v = vertices[i + segments];
        glVertex3f(v.x(), v.y(), v.z());
    }
    glEnd();

    for (int i = 0; i < segments / 2; ++i) {
        int next = i + (segments / 2);
        int topIndex = i;
        int bottomIndex = i + segments;

        QVector3D vTop0 = vertices[topIndex];
        QVector3D vTop1 = vertices[next];
        QVector3D vBottom0 = vertices[bottomIndex];
        QVector3D vBottom1 = vertices[next + segments];
```

```
        glBegin(GL_LINES);
        glVertex3f(vTop0.x(), vTop0.y(), vTop0.z());
        glVertex3f(vBottom0.x(), vBottom0.y(), vBottom0.z());

        glVertex3f(vTop1.x(), vTop1.y(), vTop1.z());
        glVertex3f(vBottom1.x(), vBottom1.y(), vBottom1.z());
        glEnd();

        glBegin(GL_LINES);
        glVertex3f(vTop0.x(), vTop0.y(), vTop0.z());
        glVertex3f(vTop1.x(), vTop1.y(), vTop1.z());

        glVertex3f(vBottom0.x(), vBottom0.y(), vBottom0.z());
        glVertex3f(vBottom1.x(), vBottom1.y(), vBottom1.z());
        glEnd();
    }
}
```

Гиперболоид.

Представленный ниже код реализует функцию hyperboloid::hyperboloid(), которая создает геометрию гиперболоида и рисует его с помощью набора вызовов OpenGL.

Создание вершин гиперболоида:

- Гиперболоид состоит из сетки вершин, которая формирует его поверхность.
- Для создания вершин используется двойной цикл: внешний цикл проходит по параметру "u", который описывает вертикальные слои гиперболоида, а внутренний цикл проходит по параметру "v", который описывает окружности внутри каждого слоя.
- Для каждой пары значений "u" и "v" вычисляются координаты вершины гиперболоида с помощью формул, использующих гиперболические функции (cosh и sinh) и тригонометрические функции (cos и sin).
- Полученные координаты добавляются в вектор вершин.

Отрисовка гиперболоида:

- Гиперболоид отрисовывается с помощью набора четырехугольников, соединяющих вершины соседних углов в сетке вершин.
- Для этого используется двойной цикл, проходящий по всем парам вершин в сетке.

- Для каждой пары вершин строится четырехугольник с помощью четырех вызовов glVertex3f() для каждой вершины, и заканчивается вызовом glEnd().

```
#include "hyperboloid.h"

hyperboloid::hyperboloid(float a, float b, float c, int uSegments, int vSeg-
ments) {
    QVector<QVector3D> vertices;

    float uDelta = M_PI / uSegments;
    float vDelta = 2.0f * M_PI / vSegments;

    for (int i = 0; i <= uSegments; ++i) {
        float u = -M_PI / 2.0f + i * uDelta;
        float sinhU = sinh(u);
        float coshU = cosh(u);

        for (int j = 0; j <= vSegments; ++j) {
            float v = j * vDelta;
            float cosV = qCos(v);
            float sinV = qSin(v);

            float x = a * coshU * cosV;
            float y = b * coshU * sinV;
            float z = c * sinhU;

            vertices.push_back(QVector3D(x, y, z));
        }
    }

    for (int i = 0; i < uSegments; ++i) {
        for (int j = 0; j < vSegments; ++j) {
            int nextI = i + 1;
            int nextJ = j + 1;

            QVector3D v0 = vertices[i * (vSegments + 1) + j];
            QVector3D v1 = vertices[i * (vSegments + 1) + nextJ];
            QVector3D v2 = vertices[nextI * (vSegments + 1) + j];
            QVector3D v3 = vertices[nextI * (vSegments + 1) + nextJ];

            glBegin(GL_LINE_LOOP);
            glVertex3f(v0.x(), v0.y(), v0.z());
            glVertex3f(v1.x(), v1.y(), v1.z());
            glVertex3f(v3.x(), v3.y(), v3.z());
            glVertex3f(v2.x(), v2.y(), v2.z());
            glEnd();
        }
    }
}
```

Тороид.

Представленный ниже код реализует функцию torus::torus(), которая создает геометрию тора и рисует его с помощью набора вызовов OpenGL.

Создание вершин тора:

- Тор состоит из сетки вершин, которая формирует его поверхность.

- Для создания вершин используется двойной цикл: внешний цикл проходит по параметру "rings", который описывает кольца тора, а внутренний цикл проходит по параметру "sides", который описывает количество вершин на каждом кольце.
- Для каждой пары значений "rings" и "sides" вычисляются координаты вершины тора с помощью параметрических уравнений, которые используют тригонометрические функции (cos и sin).
- Полученные координаты добавляются в вектор вершин.

Отрисовка тора:

- Тор отрисовывается с помощью набора четырехугольников, соединяющих вершины соседних углов в сетке вершин.
- Для этого используется двойной цикл, проходящий по всем парам вершин в сетке.
- Для каждой пары вершин строится четырехугольник с помощью четырех вызовов glVertex3f() для каждой вершины, и заканчивается вызовом glEnd().

```
#include "torus.h"

torus::torus(float innerRadius, float outerRadius, int sides, int rings) {
    QVector<QVector3D> vertices;
    float ringDelta = 2.0f * M_PI / rings;
    float sideDelta = 2.0f * M_PI / sides;
    for (int i = 0; i < rings; ++i) {
        float theta = i * ringDelta;
        float cosTheta = qCos(theta);
        float sinTheta = qSin(theta);
        for (int j = 0; j < sides; ++j) {
            float phi = j * sideDelta;
            float cosPhi = qCos(phi);
            float sinPhi = qSin(phi);
            float x = cosTheta * (outerRadius + innerRadius * cosPhi);
            float y = sinTheta * (outerRadius + innerRadius * cosPhi);
            float z = innerRadius * sinPhi;

            vertices.push_back(QVector3D(x, y, z));
        }
    }

    for (int i = 0; i < rings; ++i) {
        for (int j = 0; j < sides; ++j) {
            int nextI = (i + 1) % rings;
            int nextJ = (j + 1) % sides;

            QVector3D v0 = vertices[i * sides + j];
            QVector3D v1 = vertices[i * sides + nextJ];
```

```
                QVector3D v2 = vertices[nextI * sides + j];
                QVector3D v3 = vertices[nextI * sides + nextJ];

                glBegin(GL_LINE_LOOP);
                glVertex3f(v0.x(), v0.y(), v0.z());
                glVertex3f(v1.x(), v1.y(), v1.z());
                glVertex3f(v3.x(), v3.y(), v3.z());
                glVertex3f(v2.x(), v2.y(), v2.z());
                glEnd();
            }
        }
    }
```

Тестирование.

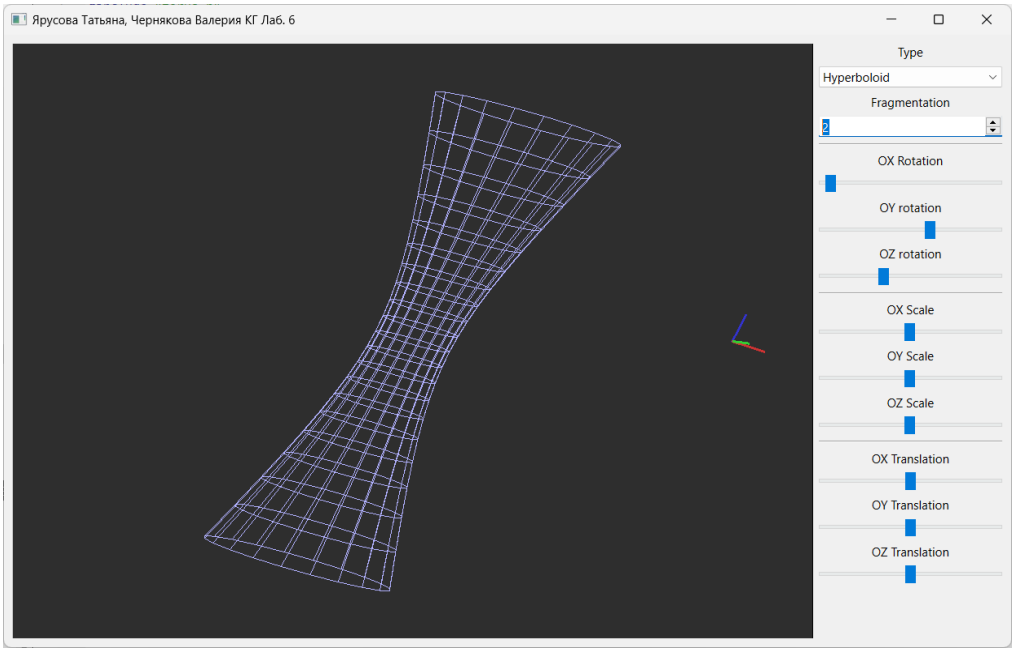


Рисунок 1.

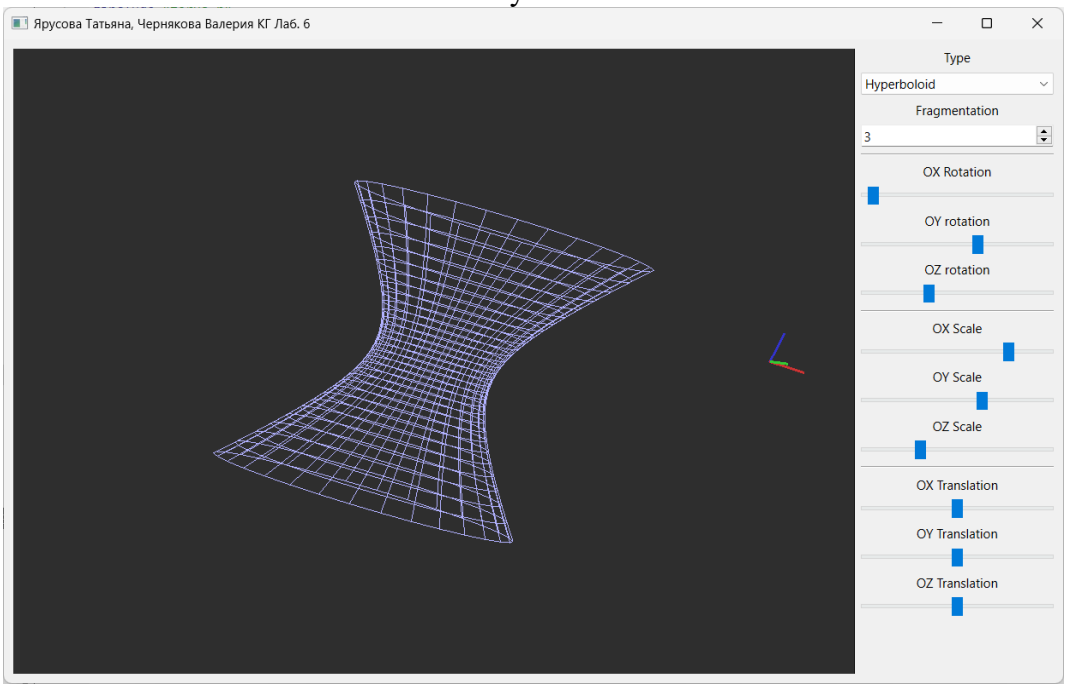


Рисунок 2.

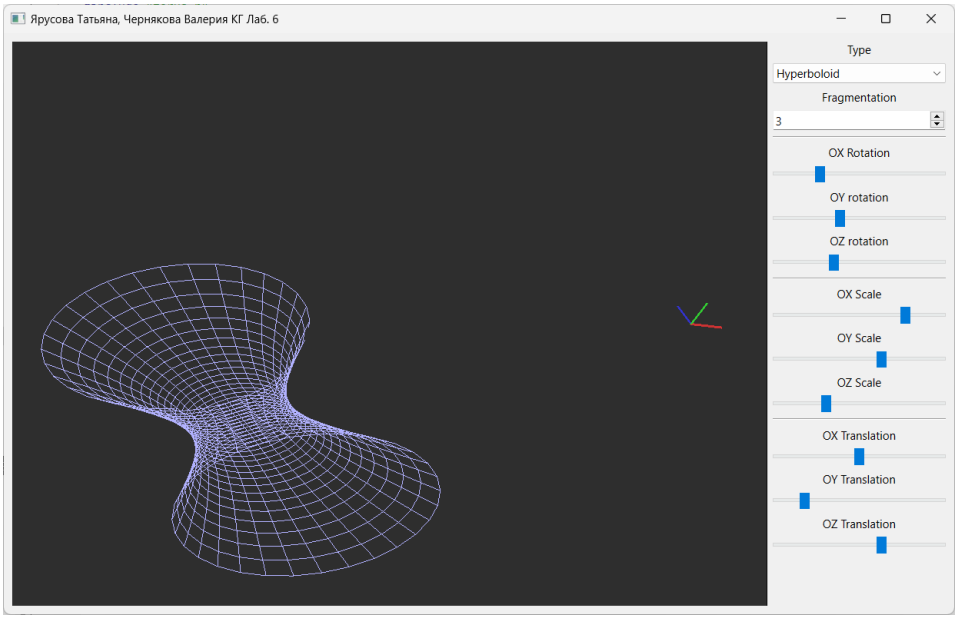


Рисунок 3.

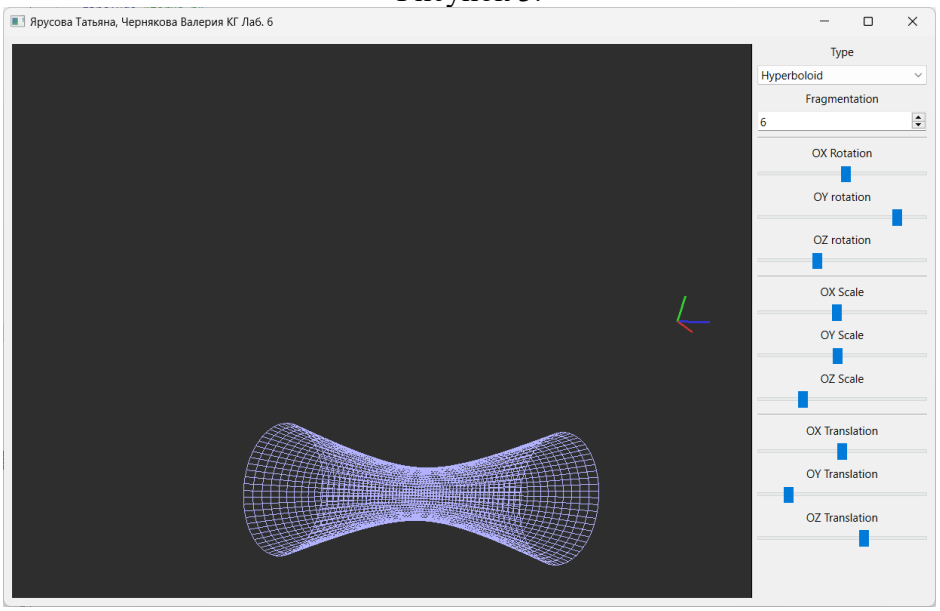


Рисунок 4.

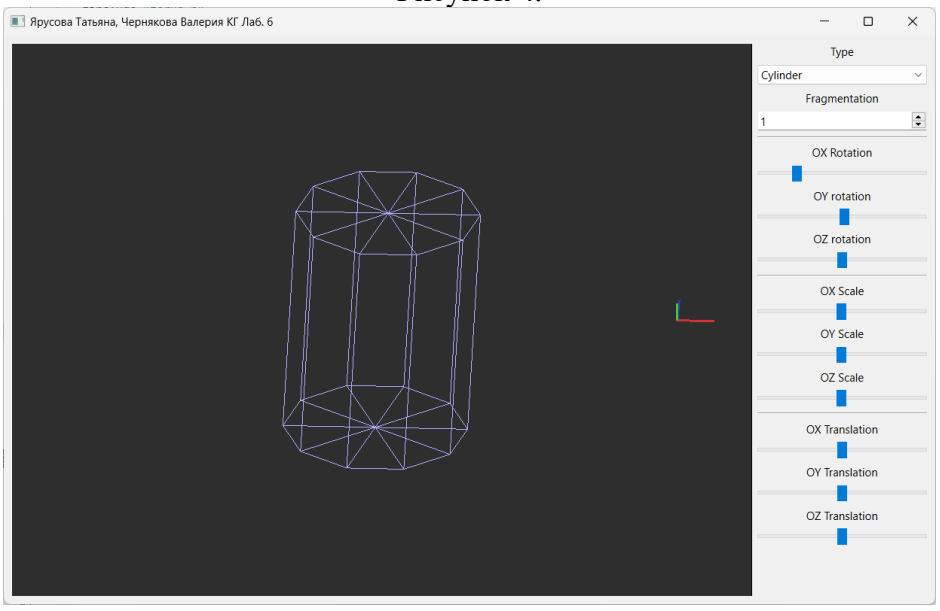


Рисунок 5.

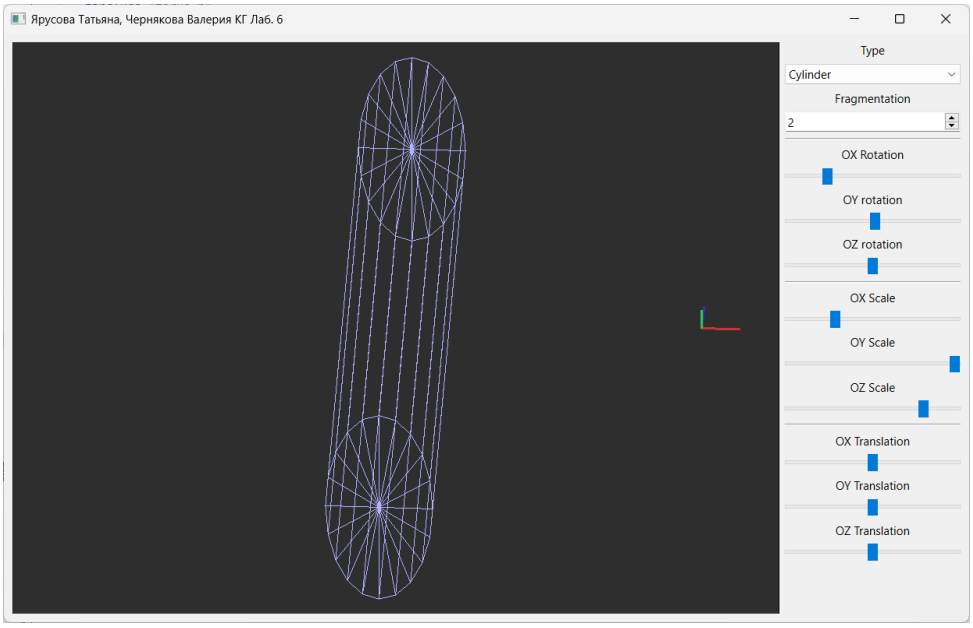


Рисунок 6.

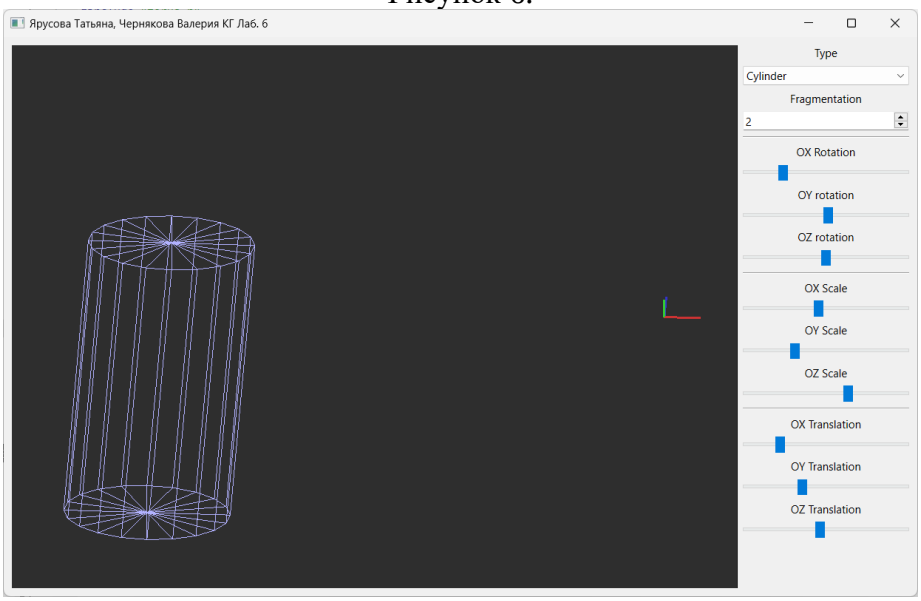


Рисунок 7.

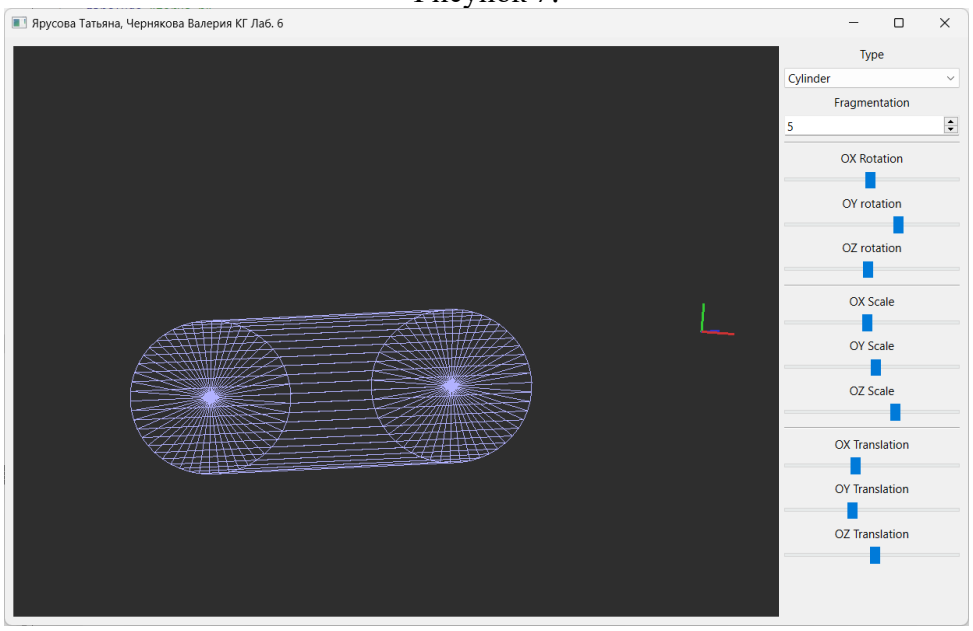


Рисунок 8.

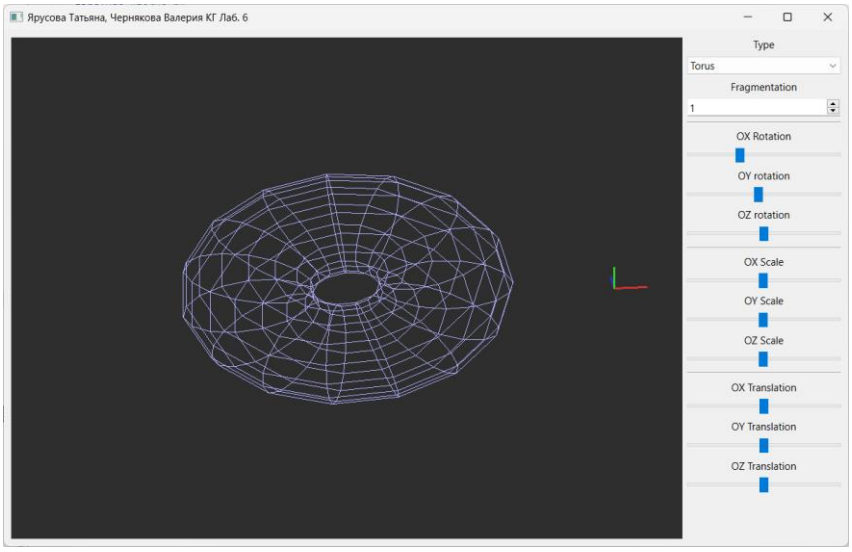


Рисунок 9.

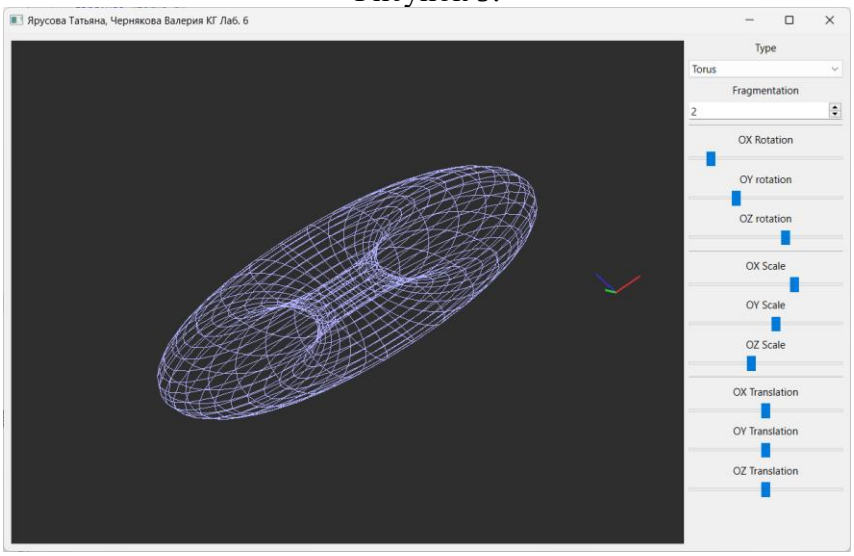


Рисунок 10.

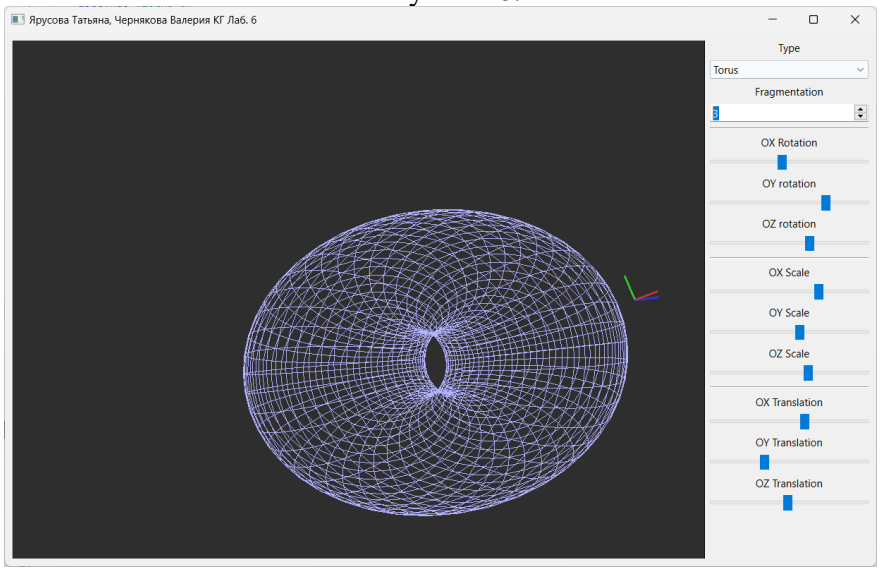


Рисунок 11.

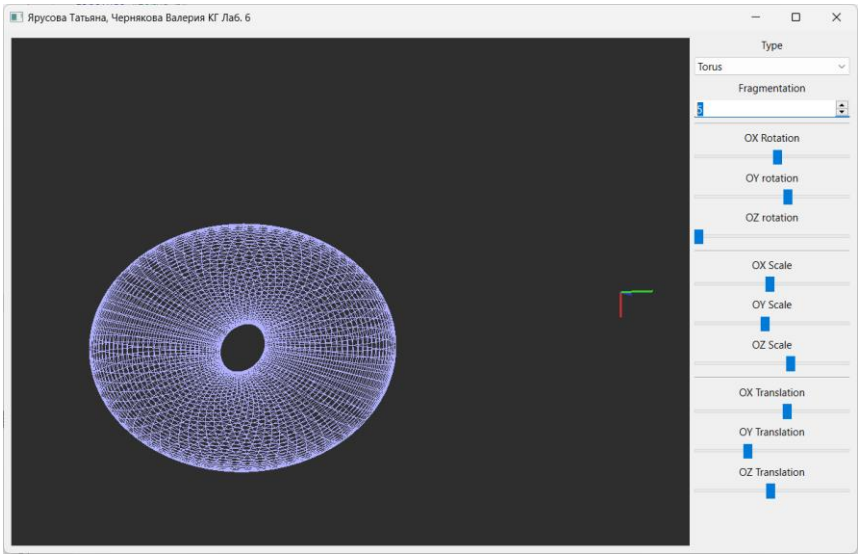


Рисунок 12.

Выводы.

В ходе лабораторной работы были рассмотрены способы построения трехмерных объектов с использованием OpenGL.

Была разработана программы, реализующая отрисовку трехмерных объектов (цилиндр, тор, гиперболоид), которые можно динамически растягивать, поворачивать и смещать вдоль осей. Также в программе можно устанавливать уровень детализации.