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

ОТЧЕТ

по лабораторной работе №6
по дисциплине «Компьютерная графика»
Тема: Реализация трехмерного объекта
с использованием библиотеки OpenGL

Студент гр. 0304 _____ Максименко Е.М.

Преподаватель _____ Герасимова Т.В.

Санкт-Петербург
2023

Цель работы.

- Изучение способов построения трехмерных объектов в OpenGL.

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

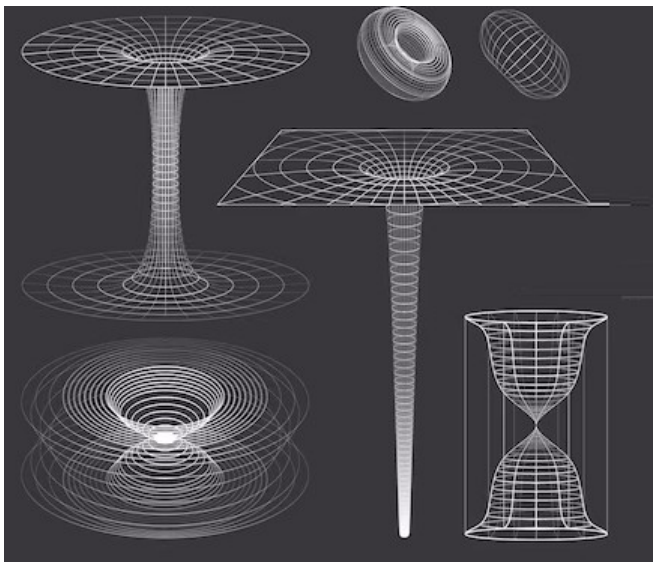
Задание.

Вариант 222:

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.

Работа была выполнена с использованием языка программирования C++ и фреймворка Qt 6. Каркасом программы послужила программа из работы 1.

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*.

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. Реализацию основных методов классов GLShaderProgram и GLVertexObject см. в листинге 1.

Листинг 1. Реализация основных методов классов GLShaderProgram и GLVertexObject.

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

bool GLShaderProgram::init()
{
    /* check if initialized before */
    if (isInitialized())
    {
        return initialized_;
    }

    initialized_ = true;
    /* compile all given shaders */
    for (const auto& [type, path]: shaders_.asKeyValueRange())
    {
        initialized_ = initialized_ && addShaderFromSourceFile(type,
path);
    }
    /* link and bind shader program */
    initialized_ = initialized_ && link() && bind();

    return initialized_;
}

bool GLVertexObject::init()
{
    /* check if initialized before */
    if (initialized_)
    {
        return initialized_;
    }

    /* create vertex buffers and vertex array */
    vertexArray_.create();
    vertexBuffer_.create();
    elementBuffer_.create();

    initialized_ = true;
    return initialized_;
}

template <typename VertexDataType>
void GLVertexObject::loadVertices(
    const QVector<VertexDataType>& vertices
)
{
    loadVertices<VertexDataType, std::nullptr_t>(vertices, {});
}

template <typename VertexDataType, typename IndexDataType>
void loadVertices(
    const QVector<VertexDataType>& vertices,
    const QVector<IndexDataType>& indices
)
{
    /* release previous buffers data */
```

```
            vertexBuffer_.release();
            elementBuffer_.release();

            /* load data to VBO */
            vertexBuffer_.bind();
            vertexBuffer_.allocate(vertices.constData(), vertices.size() *
sizeof(VertexDataType));

            /* load data to EBO (if needed) */
            if (indices.size())
            {
                elementBuffer_.bind();
                elementBuffer_.allocate(indices.constData(), indices.size() *
sizeof(IndexDataType));
            }
        }

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

Основная работа с объектом выполняется в классе GLScene. В классе создается два экземпляра класса GLShaderProgram и два экземпляра класса GLVertexObject. Это связано с тем, что 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).

Отрисовка фигуры происходит поэтапно: сперва отрисовываются кольца, которые задают форму фигуры, после этого отрисовываются вертикальные линии. После отрисовки фигуры отрисовываются оси координат.

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.

Для отрисовки фигуры задается базовый набор колец, количество которых увеличивается с увеличением степени детализации. При увеличении степени

детализации генерируются промежуточные кольца, а также увеличивается количество граней многогранников, которыми аппроксимируются кольца. Вертикальные линии соединяют вершины многогранников соседних колец.

Реализацию основных методов класса GLScene см. в листинге 2.

Листинг 2. Реализация основных методов класса GLScene.

```
void GLScene::initializeGL()
{
    /* int opengl window */
    QColor bgc(0x2E, 0x2E, 0x2E);
    initializeOpenGLFunctions();
    glClearColor(bgc.redF(), bgc.greenF(), bgc.blueF(), bgc.alphaF());
    /* create figure and axes shader programs */
    createShaderPrograms();
    /* initialize shader programs */
    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;
    }
    /* initialize vertex buffers */
    figureVertexObject_.init();
    axesVertexObject_.init();
    /* calculate and load figure and axes vertices */
    beforeUpdate();
    /* set transformations by default */
    setRotation(0.f, 0.f, 0.f);
    setScale(1.f, 1.f, 1.f);
    setTranslation(0.f, 0.f, 0.f);
    /* set wireframe mode */
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);
}

void GLScene::paintGL()
{
    /* draw figure */
    figureShaderProgram_->bind();
    figureVertexObject_.bind_vao();

    glLineWidth(1.0f);
    /* draw circles */
    GLuint circleFragments = fragmentationFactor_ * baseCircleFragmentsCount_;
    GLuint circleCount = fragmentationFactor_ * (baseCircles_.size() - 1) + 1;
    GLuint offset = 0;
    for (GLuint i = 0; i < circleCount; ++i)
    {
        glDrawArrays(GL_LINE_LOOP, offset, circleFragments);
        offset += circleFragments;
    }
    /* draw lines */
    for (GLuint i = 0; i < circleFragments; ++i)
    {
        glDrawArrays(GL_LINE_STRIP, offset, circleCount);
        offset += circleCount;
    }
    /* set figure transformations */
    {
                        int   rotationMatrixLocation   =   figureShaderProgram_-
>uniformLocation("rotation");
```

```
                    figureShaderProgram_->setUniformValue(rotationMatrixLocation,
rotationMatrix_);

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

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

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

    /* draw axes */
    axesShaderProgram_->bind();
    axesVertexObject_.bind_vao();

    glLineWidth(3.0f);
    /* set axes rotation */
    {
                           int   rotationMatrixLocation   =   axesShaderProgram_-
>uniformLocation("rotation");
                    axesShaderProgram_->setUniformValue(rotationMatrixLocation,
rotationMatrix_);
    }
    glDrawArrays(GL_LINES, 0, 6);

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

void GLScene::prepareCircles()
{
    if (!baseCircles_.size())
    {
        return;
    }
    circleVertices_.clear();

    GLuint circleFragments = fragmentationFactor_ * baseCircleFragmentsCount_;
    GLdouble radius = baseCircles_[0].radius;
    GLdouble y = baseCircles_[0].y;

    for (GLsizei i = 0; i < baseCircles_.size() - 1; ++i)
    {
        /* delta radius and Y between 2 between two adjacent circles */
        GLdouble deltaRadius = (
            baseCircles_[i + 1].radius - baseCircles_[i].radius
        ) / fragmentationFactor_;
        GLdouble deltaY = (
            baseCircles_[i + 1].y - baseCircles_[i].y
        ) / fragmentationFactor_;
        /* generate circles vertices */
        for (GLuint j = 0; j < fragmentationFactor_; ++j)
        {
            /* generate circle vertices */
            generateCircleVertices(radius, y, circleFragments);
            radius += deltaRadius;
            y += deltaY;
        }
    }
    /* generate last circle vertices */
    generateCircleVertices(radius, y, circleFragments);
}

void GLScene::prepareLines()
{
```

```
    if (!baseCircles_.size())
    {
        return;
    }
    lineVertices_.clear();

    GLdouble circleFragments = fragmentationFactor_ * baseCircleFragmentsCount_;
    GLuint circleCount = fragmentationFactor_ * (baseCircles_.size() - 1) + 1;
    /* generate lines between circles */
    for (GLuint i = 0; i < circleFragments; ++i)
    {
        for (GLuint j = 0; j < circleCount; ++j)
        {
            lineVertices_.push_back(circleVertices_[i + j * circleFragments]);
        }
    }
}

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

    /* x axis */
    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 }
    });

    /* y axis */
    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 }
    });

    /* z axis */
    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 }
    });
}

void GLScene::generateCircleVertices(
    GLdouble radius,
    GLdouble y,
    GLdouble circleFragments
)
{
    GLdouble angle = 0.f;
    GLdouble deltaAngle = 2.0f * 3.14159265f / circleFragments;
    /* generate and write single circle with radius and Y vertex data */
    for (GLuint k = 0; k < circleFragments; ++k)
    {
        circleVertices_.emplaceBack(
            radius * cos(angle),
            y,
            radius * sin(angle)
            );
        angle += deltaAngle;
    }
```

```
}
```

Исходный код шейдеров для фигуры и осей см. в листингах 3-6.

Листинг 3. Исходный код вершинного шейдера фигуры.

```
#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);
}
```

Листинг 4. Исходный код фрагментного шейдера фигуры.

```
#version 460 core

out vec4 FragColor;

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

Листинг 5. Исходный код вершинного шейдера осей.

```
#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;
}
```

Листинг 6. Исходный код фрагментного шейдера осей.

```
#version 460 core

in vec3 color;
out vec4 FragColor;

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

Тестирование программы.

При тестировании программы она была запущена с различными параметрами, отражающими различные настройки. Результаты тестирования см. на рис. 1-4.

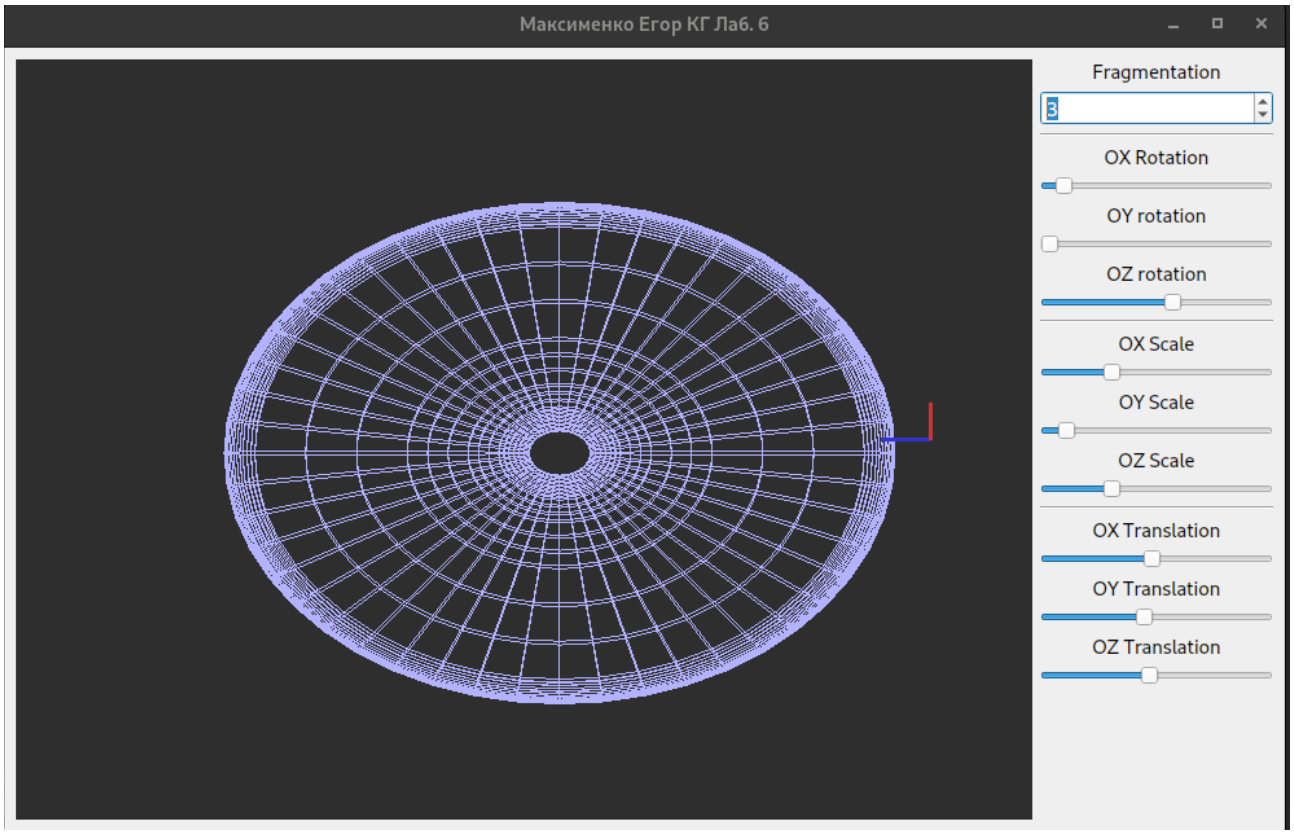


Рисунок 1.

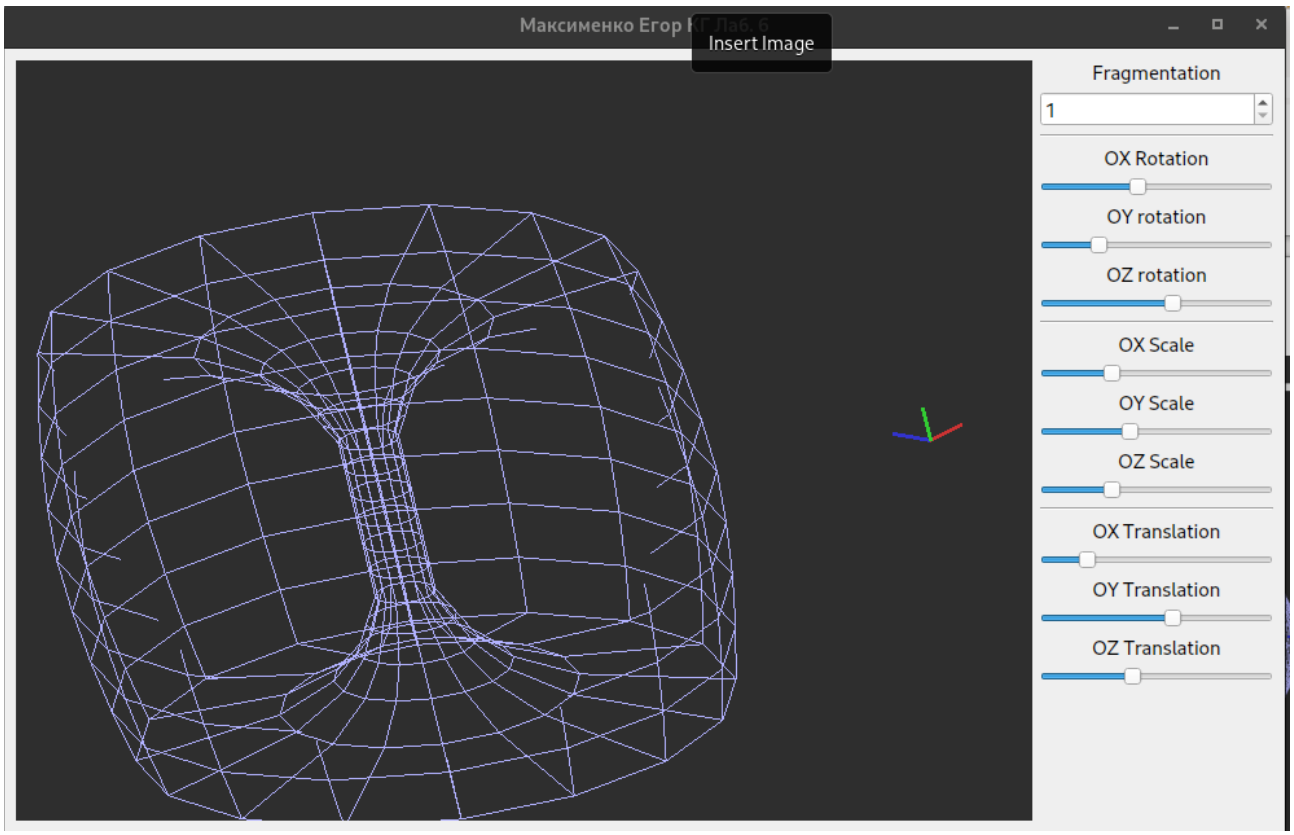


Рисунок 2.

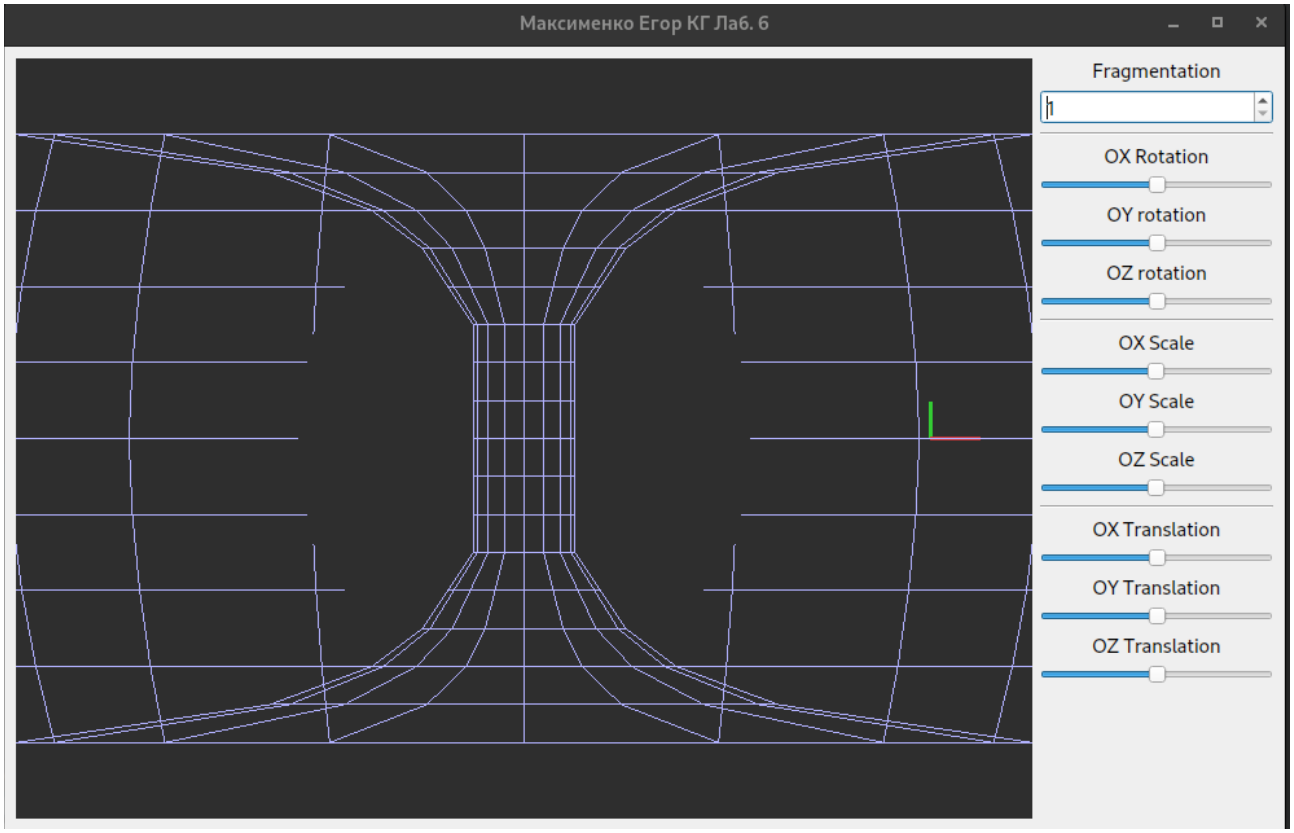


Рисунок 3.

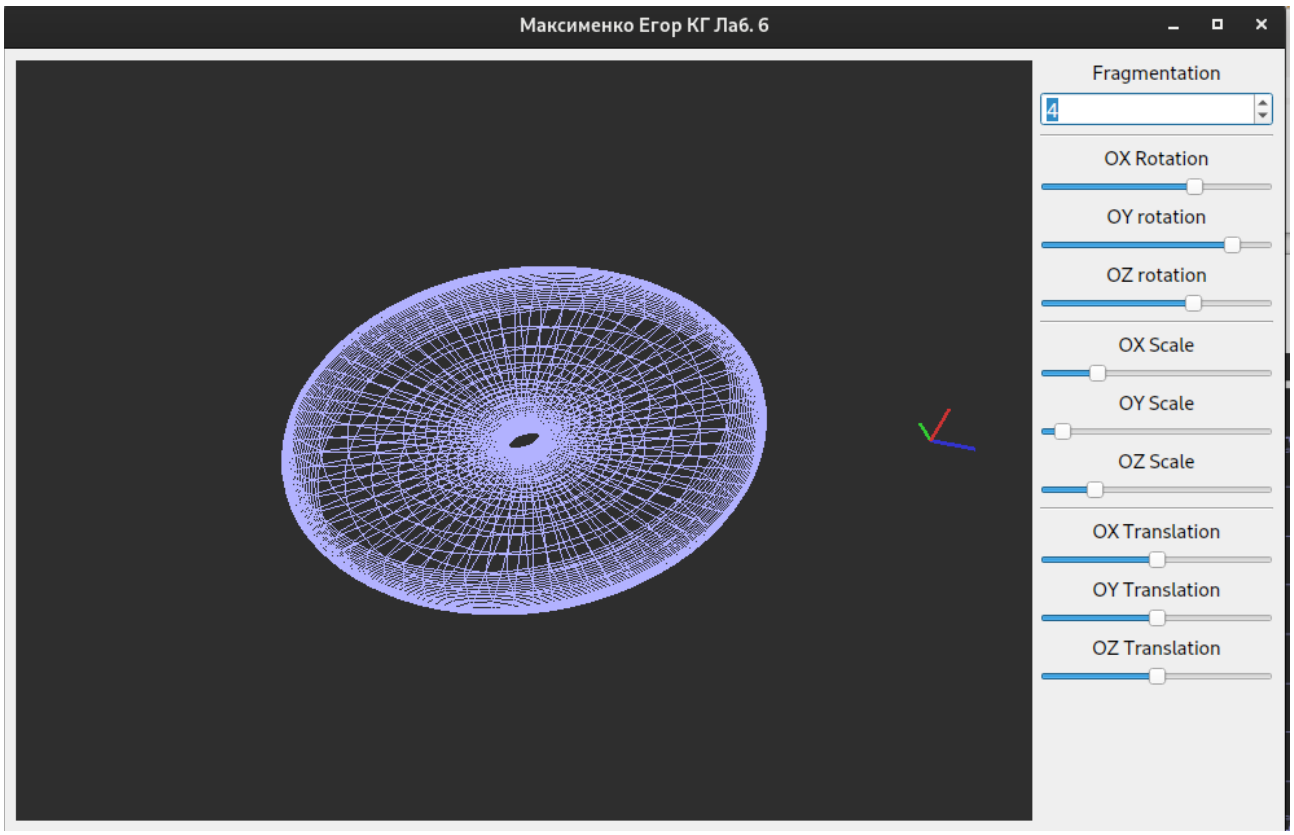


Рисунок 4.

Выводы.

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

Была разработана программы, реализующая отрисовку трехмерного объекта, который можно динамически растягивать, поворачивать и смещать вдоль осей. Также в программе можно устанавливать уровень детализации.