**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.

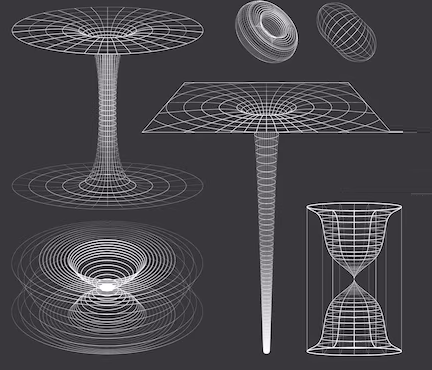
5. При запуске программы объекты сразу должны быть хорошо виден.

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,

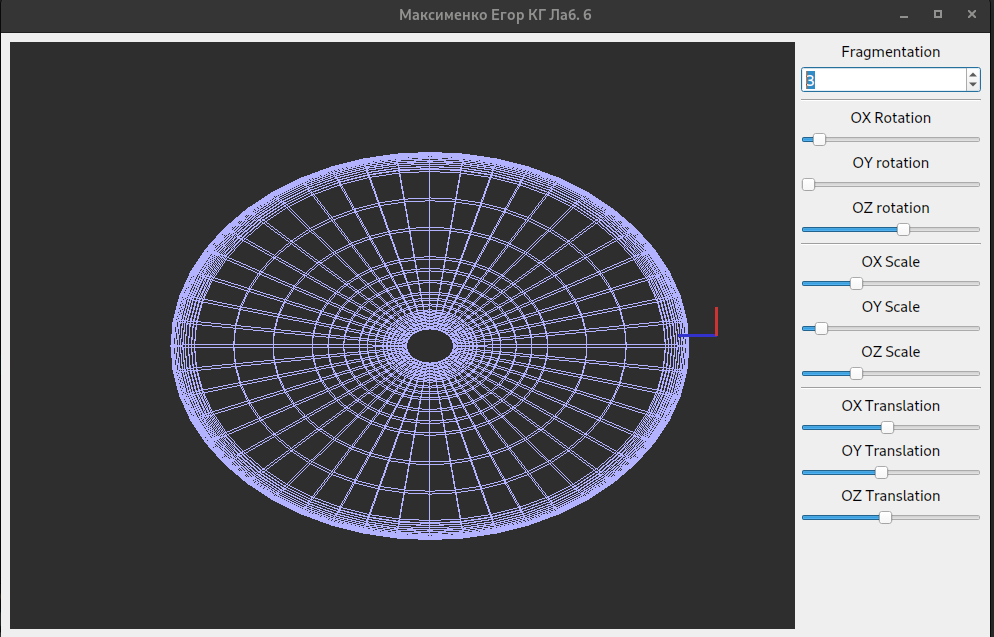
1.0

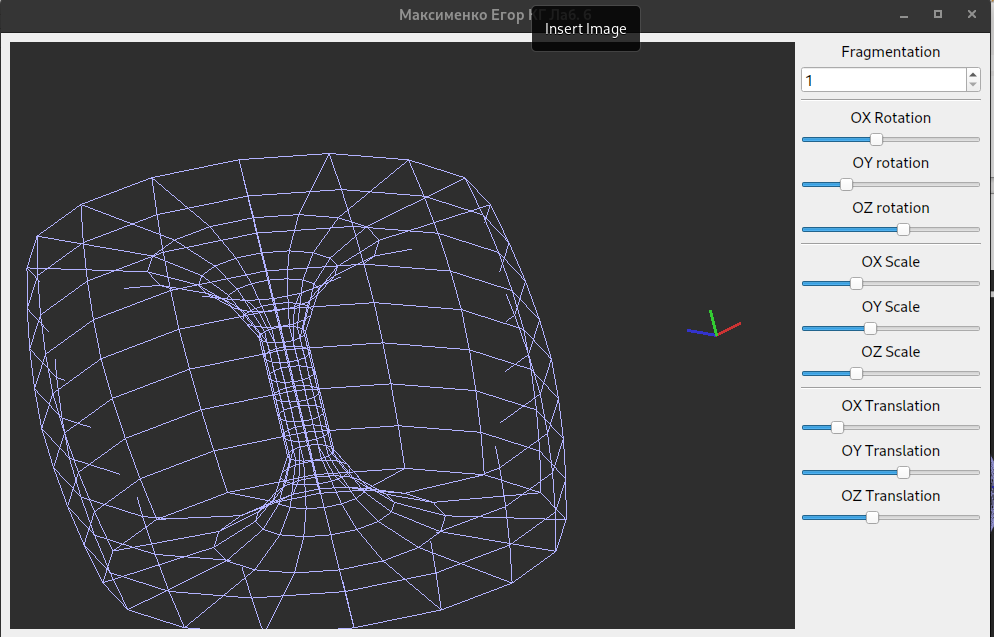
);

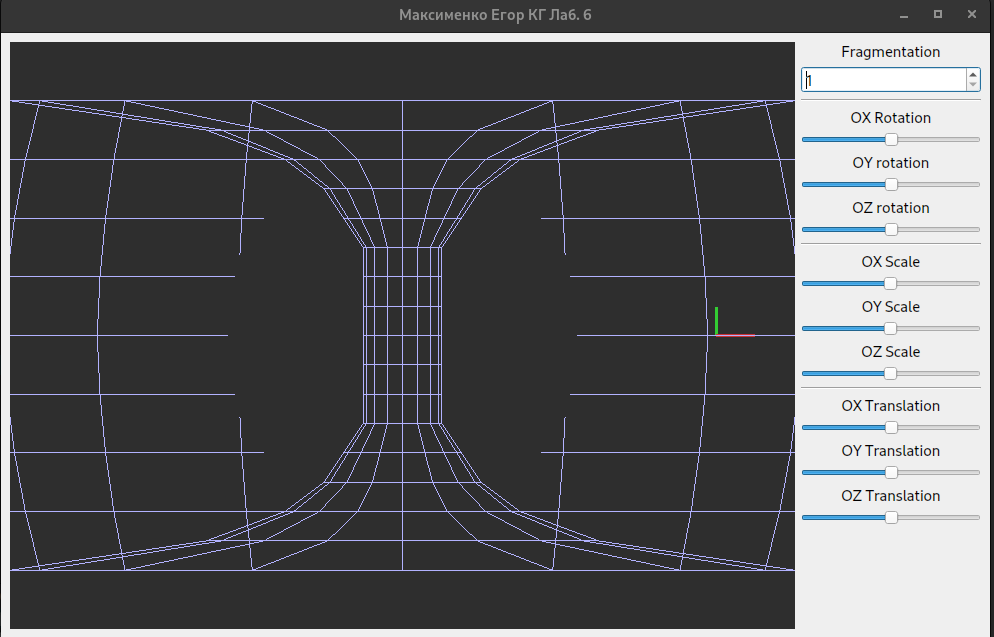
}

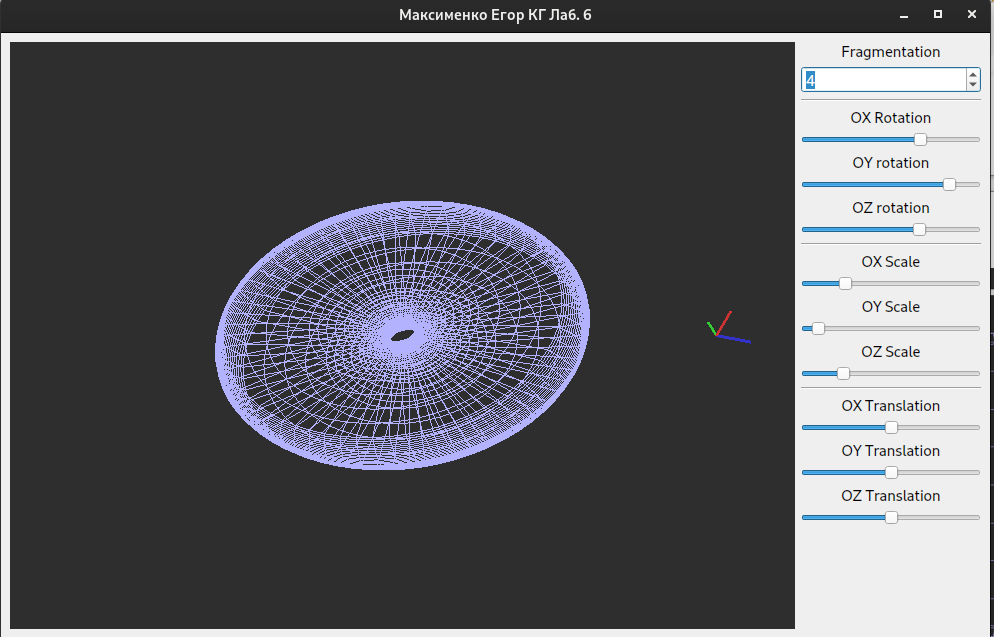
**Тестирование программы.**

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.**

Входе 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.