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

ОТЧЕТ

по лабораторной работе №7
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.

- Изучение способов использования освещения

- Изучение моделей освещения

Задание.

Разработать программу, реализующую представление трехмерной сцены (ваша 6 лаб работа) с добавлением возможности формирования различного типа проекций, отражений, используя предложенные функции OpenGL (модель освещения, типы источников света, свойства материалов(текстура)).

Разработанная программа должна быть пополнена возможностями остановки интерактивно различных атрибутов через вызов соответствующих элементов интерфейса пользователя:

- замена типа источника света,
- управление положением камеры,
- изменение свойств материала модели, как с помощью мыши, так и с помощью диалоговых элементов)

Выполнение работы.

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

Реализация камеры.

Для возможности использования камеры для перемещения по сцене был реализован класс Camera.

Перемещение камеры происходит по нажатию клавиш WASD, поворот производится с помощью мыши при зажатой ЛКМ. За перемещение камеры отвечает метод move, который принимает направление движения: forward = 1 — вперед, forward = -1 — назад, right = 1 — вправо, right = -1 — влево. Значение

forward или right, равное 0, означает, что движение в данном направлении не производится. За поворот камеры отвечает метод rotate, который принимает координаты мыши. По переданным координатам и предыдущим определяется направление поворота камера. Для настройки чувствительности мыши используется поле sensitivity, для настройки скорости перемещения — поле speed. Поле screenSize используется для корректного определения позиции мыши на экране.

```
#include "camera.h"

Camera::Camera(
    const QPointF& screenSize,
    float speed,
    const QVector3D& position
):
    speed_{ speed },
    position_{ position }
{
    /* init sensitivity and window size */
    resize(screenSize);
    /* init direction vectors */
    setDirectionVectors(QVector3D{ 0.f, 0.f, -1.f }, QVector3D{ 0.f, 1.f, 0.f });
    /* init view matrix */
    rebuildMatrix();
}


void Camera::setDirectionVectors(const QVector3D& front, const QVector3D& up)
{
    frontVector_ = front;
    upVector_ = up;
    rightVector_ = QVector3D::crossProduct(frontVector_, upVector_);
}


void Camera::rebuildMatrix()
{
    /* calculate view matrix */
    matrix_.setToIdentity();
    matrix_.lookAt(position_, position_ + frontVector_, upVector_);
    /* calculate rotation matrix */
    rotation_.setToIdentity();
    rotation_.rotate(QQuaternion::fromDirection(frontVector_, upVector_));
}


void Camera::move(int forward, int right)
{
    /* move camera */
    position_ += forward * speed_ * frontVector_;
    position_ += right * speed_ * rightVector_;

    rebuildMatrix();
}


void Camera::rotate(float xPos, float yPos)
{
    /* calculate new direction */
```

```
52.     float xOffset = (xPos - lastMousePosition_.x()) * sensitivity_;
53.     float yOffset = (lastMousePosition_.y() - yPos) * sensitivity_;
54.     lastMousePosition_ = { xPos, yPos };
55.
56.     yaw_    += xOffset;
57.     pitch_  += yOffset;
58.
59.     if (pitch_ > 89.f)
60.     {
61.         pitch_ =  89.f;
62.     }
63.     else if (pitch_ < -89.f)
64.     {
65.         pitch_ = -89.f;
66.     }
67.
68.     /* set new camera direction */
69.     QVector3D direction = {
70.         std::cos(d2r(yaw_)) * std::cos(d2r(pitch_)),
71.         std::sin(d2r(pitch_)),
72.         std::sin(d2r(yaw_)) * std::cos(d2r(pitch_))
73.     };
74.     setDirectionVectors(direction.normalized(), upVector_);
75.
76.     rebuildMatrix();
77. }
78.
79.
80. void Camera::rotate(const QPointF& position)
81. {
82.     rotate(position.x(), position.y());
83. }
84.
85.
86. void Camera::mousePress(const QPointF& position)
87. {
88.     lastMousePosition_ = { position.x(), position.y() };
89. }
90.
91.
92. void Camera::resize(const QPointF& size)
93. {
94.     /* update screen size and sensitivity */
95.     screenSize_ = size;
96.     sensitivity_ = 380.f / size.x();
97. }
```

Реализация представления материала

Для представления материала был реализован класс GLMaterial.

Представление материала имеет такие свойства, как цвет рассеивания (diffuse_) материала, цвет отражения (specular_) материала, степень блеска материла (shininess_) и силу ambient-освещения (ambientStrength_) материала. В классе присутствуют методы для задания и получения параметров материала, а также метод применения материала. При применении в шейдер с помощью uniform-переменных передаются свойства материала, которые используются для просчета освещения.

```cpp
#include "glmaterial.h"

#include <qopenglshaderprogram>
#include <iostream>


GLMaterial::GLMaterial():
    GLMaterial {
        QColor{ 237, 69, 57 },
        QColor{ 237, 69, 57 },
    }
{}


GLMaterial::GLMaterial(
    const QColor& diffuse,
    const QColor& specular,
    float shininess,
    float ambientStrength
):
    diffuse_{ diffuse },
    specular_{ specular },
    shininess_{ shininess },
    ambientStrength_{ ambientStrength }
{}


void GLMaterial::setDiffuseColor(const QColor& color)
{
    diffuse_ = color;
}


QColor GLMaterial::getDiffuseColor() const
{
    return diffuse_;
}


void GLMaterial::setSpecularColor(const QColor& color)
{
    specular_ = color;
}


QColor GLMaterial::getSpecularColor() const
{
    return specular_;
}


void GLMaterial::setShininess(float shininess)
{
    shininess_ = shininess;
}


float GLMaterial::getShininess() const
{
    return shininess_;
}


void GLMaterial::setAmbientStrength(float strength)
{
    ambientStrength_ = strength;
}

```

```
69.
70. float GLMaterial::getAmbientStrength() const
71. {
72.     return ambientStrength_;
73. }
74.
75.
76. void GLMaterial::apply(QOpenGLShaderProgram* shaderProgram) const
77. {
78.     if (!shaderProgram)
79.     {
80.         std::cerr << "[error] can't apply material settings: shader program is nullptr" <<
    std::endl;
81.         return;
82.     }
83.     shaderProgram->setUniformValue("material.diffuse", QVector3D{
84.         diffuse_.redF(),
85.         diffuse_.greenF(),
86.         diffuse_.blueF()
87.     });
88.     shaderProgram->setUniformValue("material.specular", QVector3D{
89.         specular_.redF(),
90.         specular_.greenF(),
91.         specular_.blueF()
92.     });
93.     shaderProgram->setUniformValue("material.shininess", shininess_);
94.     shaderProgram->setUniformValue("material.ambient_strength", ambientStrength_);
95. }
```

Реализация представления освещения

Для представления освещения был реализован класс GLLighting.

Представление материала имеет такие свойства, как тип освещения (type_): точечный источник, направленный свет, прожектор, - позицию источника освещения (position_) для точечного источника и прожектора, направление освещения (direction_) для направленного света и прожектора, цвет ambient-освещения (ambient_), цвет рассеянного освещения (diffuse_), цвет отражений (specular_), коэффициента уравнения затухания света (attenuation_) для точечного источника и прожектора, угол конуса отсечения (cutOff_ и outerCutOff_) для прожектора. В классе присутствуют методы задания и получения всех параметров, а также метод apply для применения настроек освещения в шейдере. При применении настроек освещения в шейдер с помощью uniform-переменных передаются настройки освещения которые используются для просчета освещения.

```
1. #include "gllighting.h"
2.
3. #include <qopenglshaderprogram>
4. #include <qopenglfunctions>
5. #include <qvector4d>
6. #include <iostream>
```

```
7.
8.
9.  GLLighting::GLLighting(
10.     GLLightingType type,
11.     const QVector3D& position,
12.     const QVector3D& direction,
13.     const QColor& ambient,
14.     const QColor& diffuse,
15.     const QColor& specular
16. ):
17.     type_{ type },
18.     position_{ position },
19.     direction_{ direction },
20.     ambient_{ ambient },
21.     diffuse_{ diffuse },
22.     specular_{ specular },
23.     attenuation_{ 1.0f, 0.09f, 0.032f },
24.     cutOff_{ 12.5f },
25.     outerCutOff_{ 17.5f }
26. {}
27.
28.
29. void GLLighting::setType(GLLightingType type)
30. {
31.     type_ = type;
32. }
33.
34.
35. GLLightingType GLLighting::getType() const
36. {
37.     return type_;
38. }
39.
40.
41. void GLLighting::setPosition(const QVector3D& position)
42. {
43.     position_ = position;
44. }
45.
46.
47. QVector3D GLLighting::getPosition() const
48. {
49.     return position_;
50. }
51.
52.
53. void GLLighting::setDirection(const QVector3D& direction)
54. {
55.     direction_ = direction;
56. }
57.
58.
59. QVector3D GLLighting::getDirection() const
60. {
61.     return direction_;
62. }
63.
64.
65. void GLLighting::setAmbientColor(const QColor& color)
66. {
67.     ambient_ = color;
68. }
69.
70.
71. QColor GLLighting::getAmbientColor() const
72. {
73.     return ambient_;
74. }
```

```
75.
76.
77. void GLLighting::setDiffuseColor(const QColor& color)
78. {
79.     diffuse_ = color;
80. }
81.
82.
83. QColor GLLighting::getDiffuseColor() const
84. {
85.     return diffuse_;
86. }
87.
88.
89. void GLLighting::setSpecularColor(const QColor& color)
90. {
91.     specular_ = color;
92. }
93.
94.
95. QColor GLLighting::getSpecularColor() const
96. {
97.     return specular_;
98. }
99.
100.
101.        void GLLighting::setAttenuation(const QVector3D& attenuation)
102.        {
103.            attenuation_ = attenuation;
104.        }
105.
106.
107.        QVector3D GLLighting::getAttenuation() const
108.        {
109.            return attenuation_;
110.        }
111.
112.
113.        void GLLighting::setCutOff(float cutOff)
114.        {
115.            cutOff_ = cutOff;
116.        }
117.
118.
119.        float GLLighting::getCutOff() const
120.        {
121.            return cutOff_;
122.        }
123.
124.
125.        void GLLighting::setOuterCutOff(float outerCutOff)
126.        {
127.            outerCutOff_ = outerCutOff;
128.        }
129.
130.
131.        float GLLighting::getOuterCutOff() const
132.        {
133.            return outerCutOff_;
134.        }
135.
136.
137.        void GLLighting::apply(QOpenGLShaderProgram* shaderProgram) const
138.        {
139.            if (!shaderProgram)
140.            {
141.                std::cerr << "[error] can't apply lighting settings: shader program is
    nullptr" << std::endl;
```

```
142.                 return;
143.             }
144.
145.             if (GLLightingType::Point == type_)
146.             {
147.                 shaderProgram->setUniformValue("light.position", position_);
148.                 shaderProgram->setUniformValue("light.direction", QVector4D{ 0.0f, 0.0f,
    0.0f, 1.f });
149.             } else if (GLLightingType::Directional == type_)
150.             {
151.                 shaderProgram->setUniformValue("light.position", QVector3D{ 0.0f, 0.0f,
    0.0f });
152.                 shaderProgram->setUniformValue("light.direction", QVector4D{
153.                     direction_.x(),
154.                     direction_.y(),
155.                     direction_.z(),
156.                     0.f
157.                 });
158.             } else if (GLLightingType::Spot == type_)
159.             {
160.                 shaderProgram->setUniformValue("light.position", position_);
161.                 shaderProgram->setUniformValue("light.direction", QVector4D{
162.                     direction_.x(),
163.                     direction_.y(),
164.                     direction_.z(),
165.                     1.f
166.                 });
167.             }
168.
169.             shaderProgram->setUniformValue("light.ambient", QVector3D{
170.                 ambient_.redF(),
171.                 ambient_.greenF(),
172.                 ambient_.blueF()
173.             });
174.             shaderProgram->setUniformValue("light.diffuse", QVector3D{
175.                 diffuse_.redF(),
176.                 diffuse_.greenF(),
177.                 diffuse_.blueF()
178.             });
179.             shaderProgram->setUniformValue("light.specular", QVector3D{
180.                 specular_.redF(),
181.                 specular_.greenF(),
182.                 specular_.blueF()
183.             });
184.
185.             shaderProgram->setUniformValue("light.attenuation_coeffs", attenuation_);
186.
187.             if (GLLightingType::Spot == type_)
188.             {
189.                 shaderProgram->setUniformValue("light.cut_off", std::cos(d2r(cutOff_)));
190.                 shaderProgram->setUniformValue("light.outer_cut_off",
    std::cos(d2r(outerCutOff_)));
191.             } else
192.             {
193.                 shaderProgram->setUniformValue("light.cut_off", 0.f);
194.                 shaderProgram->setUniformValue("light.outer_cut_off", 0.f);
195.             }
196.         }
```

Реализация обертки для работы с буферами OpenGL

Для упрощения работы с буферами OpenGL (VAO, VBO, EBO) был реализован класс GLVertexObject.

Класс GLVertexObject является оберткой над VAO, VBO и EBO, которые хранятся в нем в полях vertexArray_, vertexBuffer_ и elementBuffer_. Для получения доступа к ним в классе реализованы методы получения данных буферов. В классе присутствует метод загрузки данных в VBO и индексов в EBO loadVertices, который загружает данные о вершинах в VBO и, если были переданы, данные об индексах в EBO. Также присутсвует метод setupVertexAttribute, который указывает OpenGL расположения входных параметров в 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()
{
    /* check if initialized before */
    if (initialized_)
    {
        return initialized_;
    }

    /* create vertex buffers and vertex array */
    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
)
{
    /* set vertex attribute pointer */
    painter->glEnableVertexAttribArray(index);
```

```
50.     painter->glVertexAttribPointer(index, size, type, normalized, stride, offset);
51. }
52.
53.
54. QOpenGLVertexArrayObject& GLVertexObject::vao()
55. {
56.     return vertexArray_;
57. }
58.
59.
60. const QOpenGLBuffer& GLVertexObject::vbo() const
61. {
62.     return vertexBuffer_;
63. }
64.
65.
66. const QOpenGLBuffer& GLVertexObject::ebo() const
67. {
68.     return elementBuffer_;
69. }
70.
71.
72. bool GLVertexObject::isInitialized() const
73. {
74.     return initialized_;
75. }
76.
77.
78. void GLVertexObject::bind_vao()
79. {
80.     if (initialized_)
81.     {
82.         vertexArray_.bind();
83.     }
84. }
85.
86.
87. void GLVertexObject::unbind_vao()
88. {
89.     if (initialized_)
90.     {
91.         vertexArray_.release();
92.     }
93. }
```

Реализация обертки для работы с шейдерной программой

Для упрощения работы с шейдерными программами был реализован класс GLShaderProgram.

Класс GLShaderProgram является расширением класса QOpenGLShaderProgram. При создании объекта этого класса ему необходимо передавать словарь, в котором ключами являются типы шейдеров, значениями — пути до шейдеров. Словарь шейдеров сохраняется в поле shaders_. Метод инициализации init загружает шейдеры, компилирует их и собирает в шейдерную программу.

```
#include "glshaderprogram.h"

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

Реализация представления фигуры

Для представления фигуры был реализован класс GLFigure.

Представление фигуры имеет такие свойства, как вершинный объект vertexObject_ (объект GLVertexObject) для загрузки информации о вершинах фигуры в шейдер, вектор описаний сечений фигуры baseCircles_ (каждое сечение является окружностью с определенным радиусом и определенной координатой Y), вектор вершин vertices_, который содержит информацию о позиции и нормали вершины.

```
#include "glfigure.h"

#include <qopenglshaderprogram>
#include <qvector3d>
#include <iostream>

#include "glscene.h"


GLFigure::GLFigure(const QVector<circlebasedata>& circles):
    baseCircleSegmentsCount_{16},
```

```
    baseCircles_{circles}
{
    rotation_.setToIdentity();
    scale_.setToIdentity();
    translation_.setToIdentity();
}


void GLFigure::init(GLScene* painter)
{
    if (initialized_)
    {
        std::cerr << "[warning] already initialized" << std::endl;
        return;
    }
    /* initialize buffers */
    initialized_ = vertexObject_.init();
    if (!initialized_)
    {
        std::cerr << "[error] can't initialize vertex object" << std::endl;
        return;
    }
    /* generate polygons for fragmentation = 1 */
    fragmentation_ = 1;
    calculatePolygons();
    setAttributeInfo(painter);
}


void GLFigure::draw(GLScene* painter, GLuint fragmentation)
{
    if (!initialized_)
    {
        std::cerr << "[error] can't draw: not initialized" << std::endl;
        return;
    }
    if (!painter)
    {
        std::cerr << "[error] can't draw: painter is nullptr" << std::endl;
        return;
    }
    vertexObject_.bind_vao();
    /* update poygons if fragmentation changed or transforms dirty */
    if (dirty_ || (fragmentation && fragmentation != fragmentation_))
    {
        /* update polygons */
        fragmentation_ = fragmentation;
        calculatePolygons();
        /* set attribute info */
        setAttributeInfo(painter);
    }
    /* apply material */
    material_.apply(painter->getFigureShaderProgram());
    /* draw polygons */
    glDrawElements(GL_TRIANGLES, indicesCount_, GL_UNSIGNED_INT, nullptr);
    vertexObject_.unbind_vao();
}


void GLFigure::setRotation(const QMatrix4x4& rotationMatrix)
{
    rotation_ = rotationMatrix;
    dirty_ = true;
}


void GLFigure::setRotation(GLfloat angle, const QVector3D& rotationAxis)
{
```

```
80.     rotation_.setToIdentity();
81.     rotation_.rotate(angle, rotationAxis);
82.     dirty_ = true;
83. }
84.
85.
86. void GLFigure::setScale(const QMatrix4x4& scaleMatrix)
87. {
88.     scale_ = scaleMatrix;
89.     dirty_ = true;
90. }
91.
92.
93. void GLFigure::setScale(GLfloat scaleX, GLfloat scaleY, GLfloat scaleZ)
94. {
95.     scale_.setToIdentity();
96.     scale_.scale(scaleX, scaleY, scaleZ);
97.     dirty_ = true;
98. }
99.
100.
101.        void GLFigure::setTranslation(const QMatrix4x4& translationMatrix)
102.        {
103.            translation_ = translationMatrix;
104.            dirty_ = true;
105.        }
106.
107.
108.        void GLFigure::setTranslation(GLfloat translationX, GLfloat translationY, GLfloat
    translationZ)
109.        {
110.            translation_.setToIdentity();
111.            translation_.translate(translationX, translationY, translationZ);
112.            dirty_ = true;
113.        }
114.
115.
116.        void GLFigure::calculatePolygons()
117.        {
118.            if (!initialized_)
119.            {
120.                std::cerr << "[error] can't generate polygons: not initialized" <<
    std::endl;
121.                return;
122.            }
123.            if (!baseCircles_.size())
124.            {
125.                std::cerr << "[error] can't generate polygons: base circles is empty" <<
    std::endl;
126.                return;
127.            }
128.            /* recalculate inverse transposed matrix */
129.            inverseTransposedModel_ = rotation_ * translation_ * scale_;
130.            inverseTransposedModel_ = inverseTransposedModel_.inverted();
131.            inverseTransposedModel_ = inverseTransposedModel_.transposed();
132.            /* prepare data */
133.            generateCircles();
134.            generateNorms();
135.            generateIndices();
136.            /* load data to buffer */
137.            vertexObject_.bind_vao();
138.            vertexObject_.loadVertices(vertices_, indices_);
139.            /* set dirty to false */
140.            dirty_ = false;
141.        }
142.
143.
144.        void GLFigure::generateCircles()
```

```
145.       {
146.           if (!initialized_)
147.           {
148.               std::cerr << "[error] can't generate circles: not initialized" <<
    std::endl;
149.               return;
150.           }
151.           if (!baseCircles_.size())
152.           {
153.               std::cerr << "[error] can't generate circles: base circles is empty" <<
    std::endl;
154.               return;
155.           }
156.           /* remove previous data */
157.           vertices_.clear();
158.           /* circle parameters */
159.           GLfloat radius = baseCircles_.at(0).radius;
160.           GLfloat y = baseCircles_.at(0).y;
161.           /* generate circles */
162.           for (GLuint i = 0; i < baseCircles_.size() - 1; ++i)
163.           {
164.               /* deltas between circles parameters */
165.               GLfloat deltaRadius = (
166.                   baseCircles_.at(i + 1).radius -
167.                   baseCircles_.at(i).radius
168.               ) / fragmentation_;
169.               GLfloat deltaY = (
170.                   baseCircles_.at(i + 1).y -
171.                   baseCircles_.at(i).y
172.               ) / fragmentation_;
173.               /* generate intermediate circles */
174.               for (GLuint j = 0; j < fragmentation_; ++j)
175.               {
176.                   generateCircle(radius, y);
177.                   radius += deltaRadius;
178.                   y += deltaY;
179.               }
180.           }
181.           /* generate last circle */
182.           generateCircle(radius, y);
183.       }
184.
185.
186.       void GLFigure::generateIndices()
187.       {
188.           if (!initialized_)
189.           {
190.               std::cerr << "[error] can't generate indices: not initialized" <<
    std::endl;
191.               return;
192.           }
193.           if (!vertices_.size())
194.           {
195.               std::cerr << "[error] can't generate indices: vertices is empty" <<
    std::endl;
196.               return;
197.           }
198.           /* remove previous data */
199.           indices_.clear();
200.           /* circles count */
201.           GLuint circleSegmentsCount = fragmentation_ * baseCircleSegmentsCount_;
202.           GLuint circlesCount = fragmentation_ * (baseCircles_.size() - 1) + 1;
203.           /* generate indices for polygons */
204.           for (GLuint i = 0; i < circlesCount - 1; ++i)
205.           {
206.               for (GLuint j = 0; j < circleSegmentsCount - 1; ++j)
207.               {
208.                   /* indices for left triangle */
```

```
209.                    indices_.push_back({
210.                        (i + 1) * circleSegmentsCount + j + 0,
211.                        (i + 0) * circleSegmentsCount + j + 0,
212.                        (i + 1) * circleSegmentsCount + j + 1,
213.                    });
214.                    /* indices for right triangle */
215.                    indices_.push_back({
216.                        (i + 1) * circleSegmentsCount + j + 1,
217.                        (i + 0) * circleSegmentsCount + j + 0,
218.                        (i + 0) * circleSegmentsCount + j + 1,
219.                    });
220.                }
221.                /* connect last and first vertices */
222.                /* indices for left triangle */
223.                indices_.push_back({
224.                    (i + 1) * circleSegmentsCount + circleSegmentsCount - 1,
225.                    (i + 0) * circleSegmentsCount + circleSegmentsCount - 1,
226.                    (i + 1) * circleSegmentsCount + 0,
227.                });
228.                /* indices for right triangle */
229.                indices_.push_back({
230.                    (i + 1) * circleSegmentsCount + 0,
231.                    (i + 0) * circleSegmentsCount + circleSegmentsCount - 1,
232.                    (i + 0) * circleSegmentsCount + 0,
233.                });
234.            }
235.            indicesCount_ = indices_.size() * sizeof(TriangleIndices);
236.        }
237.
238.
239.        void GLFigure::generateNorms()
240.        {
241.            if (!initialized_)
242.            {
243.                std::cerr << "[error] can't generate norms: not initialized" << std::endl;
244.                return;
245.            }
246.            if (!vertices_.size())
247.            {
248.                std::cerr << "[error] can't generate norms: vertices is empty" <<
    std::endl;
249.                return;
250.            }
251.
252.            GLuint circleSegmentsCount = fragmentation_ * baseCircleSegmentsCount_;
253.            GLuint circlesCount = fragmentation_ * (baseCircles_.size() - 1) + 1;
254.            QVector3D norm;
255.            QVector4D norm4;
256.            /* generate norms for vertices */
257.            for (GLuint i = 0; i < circlesCount - 1; ++i)
258.            {
259.                for (GLuint j = 0; j < circleSegmentsCount - 1; ++j)
260.                {
261.                    /* norms has to be rotated, translated and scaled like vertices */
262.                    norm4 = inverseTransposedModel_ * QVector4D(
263.                        QVector3D::normal(
264.                            vertices_.at((i + 0) * circleSegmentsCount + j + 0).first, // 0
    vertex position
265.                            vertices_.at((i + 0) * circleSegmentsCount + j + 1).first, //
    right vertex position
266.                            vertices_.at((i + 1) * circleSegmentsCount + j + 0).first  //
    down vertex position
267.                        ),
268.                        1.0f
269.                    );
270.                    norm = QVector3D{ norm4.x(), norm4.y(), norm4.z() };
271.                    /* set all polygon vertices norms */
272.                    vertices_[(i + 0) * circleSegmentsCount + j + 0].second = norm;
```

```
273.                    vertices_[(i + 0) * circleSegmentsCount + j + 1].second = norm;
274.                    vertices_[(i + 1) * circleSegmentsCount + j + 0].second = norm;
275.                    vertices_[(i + 1) * circleSegmentsCount + j + 1].second = norm;
276.                }
277.            }
278.        }
279.
280.
281.        void GLFigure::generateCircle(
282.            GLfloat radius,
283.            GLfloat y
284.        )
285.        {
286.            if (!initialized_)
287.            {
288.                std::cerr << "[error] can't generate circle: not initialized" << std::endl;
289.                return;
290.            }
291.            GLuint circleSegmentsCount = fragmentation_ * baseCircleSegmentsCount_;
292.            GLfloat angle = 0.f;
293.            GLfloat deltaAngle = 2.0f * 3.14159265f / circleSegmentsCount;
294.            QVector4D vertex;
295.            /* generate circle segments */
296.            for (GLuint i = 0; i < circleSegmentsCount; ++i)
297.            {
298.                /* apply all transformations */
299.                vertex = translation_ * rotation_ * scale_ * QVector4D{
300.                    radius * std::cos(angle),
301.                    y,
302.                    radius * std::sin(angle),
303.                    1.f
304.                };
305.                /* save transformed vertex */
306.                vertices_.push_back({
307.                    {
308.                        vertex.x(),
309.                        vertex.y(),
310.                        vertex.z()
311.                    },
312.                    {1.f, 1.f, 1.f} // temporary norm
313.                });
314.                angle += deltaAngle;
315.            }
316.        }
317.
318.
319.        void GLFigure::setAttributeInfo(QOpenGLFunctions* painter)
320.        {
321.            if (!initialized_)
322.            {
323.                std::cerr << "[error] can't set attribure pointer: not initialized" <<
    std::endl;
324.                return;
325.            }
326.            if (!painter)
327.            {
328.                std::cerr << "[error] can't set attribure pointer: painter is nullptr" <<
    std::endl;
329.                return;
330.            }
331.            /* set attribure info */
332.            vertexObject_.bind_vao();
333.            /* set position info */
334.            vertexObject_.setupVertexAttribute(
335.                painter, 0, 3, GL_FLOAT, GL_FALSE,
336.                2 * sizeof(QVector3D), nullptr
337.            );
338.            /* set norm info */
```

```
339.           vertexObject_.setupVertexAttribute(
340.               painter, 1, 3, GL_FLOAT, GL_TRUE,
341.               2 * sizeof(QVector3D), reinterpret_cast<void*>(sizeof(QVector3D))
342.           );
343.       }
```

Реализация сцены

Для сцены был реализован класс GLScene.

Сцена OpenGL содержит такую информацию о сцене, как размеры виджета сцены (width_ и height_), шейдерные программы для фигур (figureShaderProgram_), для осей координат (axesShaderProgram_), для источника освещения (lightSourceShaderProgram_), список фигур на сцене (figures_), список вершин для отрисовки осей координат (axesVertices_), камера на сцене (camera_), матрица модельных преобразований (modelMatrix_), матрица проекции (projectionMatrix_), тип проекции (projection_), освещение сцены (lighting_). Для обработки пользовательского ввода посредством клавиатуры и мыши реализованы методы keyPressEvent, mousePressEvent, mouseMoveEvent. Для изменения типа проекции реализован метод setProjectionType. Для получения элементов сцены, таких как освещение и фигуры, реализованы методы getLighting, getFigures. В методе createShaderProgram происходит создание шейдерных программ, в методе generateFigures происходит создание фигур путем генерации сечений для каждой из них, в методе generateAxes — генерация вершин для осей координат. Также присутствуют методы initializeGL для инициализации окна, шейдерных программ и фигур, resizeGL для пересчета матрицы проекции, paintGL для непосредственно отрисовки сцены.

```
1.  #include "glscene.h"
2.
3.  #include <qcolor>
4.  #include <qmouseevent>
5.  #include <qkeyevent>
6.  #include <qtimer>
7.  #include <algorithm>
8.  #include <iostream>
9.  #include <iomanip>
10. #include <cmath>
11.
12.
13. GLScene::GLScene(QWidget* parent):
14.     QOpenGLWidget{ parent },
15.     lighting_{ GLLightingType::Point, { 1.2f, 1.0f, 0.0f }, { -1.0f, -1.0f, 0.0f } },
```

```
16.     camera_{ { 800.f, 600.f } }
17. {
18.     /* generate vector of 6 figures */
19.     generateFigures();
20. 
21.     width_ = width();
22.     height_ = height();
23. 
24.     modelMatrix_.setToIdentity();
25.     projectionMatrix_.setToIdentity();
26. }
27. 
28. 
29. void GLScene::initializeGL()
30. {
31.     /* init opengl window */
32.     QColor bgc(0x3F, 0x3F, 0x3F);
33.     initializeOpenGLFunctions();
34.     glClearColor(bgc.redF(), bgc.greenF(), bgc.blueF(), bgc.alphaF());
35.     /* create figures shader program */
36.     createShaderProgram();
37.     /* initialize figures shader programs */
38.     if (!figureShaderProgram_->init())
39.     {
40.         std::cerr << "[error] Unable to initialize Figure Shader Program" << std::endl;
41.         std::cerr << "Shader Program log: " << figureShaderProgram_->log().toStdString()
    << std::endl;
42.         return;
43.     }
44.     /* initialize axes shader programs */
45.     if (!axesShaderProgram_->init())
46.     {
47.         std::cerr << "[error] Unable to initialize Axes Shader Program" << std::endl;
48.         std::cerr << "Shader Program log: " << axesShaderProgram_->log().toStdString() <<
    std::endl;
49.         return;
50.     }
51.     /* initialize light source shader programs */
52.     if (!lightSourceShaderProgram_->init())
53.     {
54.         std::cerr << "[error] Unable to initialize Light Source Shader Program" <<
    std::endl;
55.         std::cerr    <<    "Shader    Program    log:    "    <<    lightSourceShaderProgram_-
    >log().toStdString() << std::endl;
56.         return;
57.     }
58.     /* initialize axes vertex object */
59.     axesVertexObject_.init();
60.     generateAxes();
61.     /* initialize all figures */
62.     for (auto figure: figures_)
63.     {
64.         figure->init(this);
65.     }
66.     /* initialize light source */
67.     lightSource_->init(this);
68. 
69. //    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
70.     glEnable(GL_DEPTH_TEST);
71. }
72. 
73. 
74. void GLScene::resizeGL(int w, int h)
75. {
76.     width_ = w;
77.     height_ = h;
78.     /* update camera window size */
79.     camera_.resize( { width_, height_ } );
```

```
80.     /* update projection matrix */
81.     setProjectionType(projection_);
82. }
83.
84.
85. void GLScene::paintGL()
86. {
87.     figureShaderProgram_->bind();
88.     glLineWidth(1.0f);
89.     /* set scene transformations */
90.     figureShaderProgram_->setUniformValue("model", modelMatrix_);
91.     figureShaderProgram_->setUniformValue("view", camera_.getView());
92.     figureShaderProgram_->setUniformValue("projection", projectionMatrix_);
93.     /* set scene lighting params */
94.     lighting_.apply(figureShaderProgram_);
95.     figureShaderProgram_->setUniformValue("cameraPos", camera_.getPosition());
96.     /* draw figures */
97.     for (auto figure: figures_)
98.     {
99.         figure->draw(this, fragmentationFactor_);
100.           }
101.           figureShaderProgram_->release();
102.
103.           /* draw axes */
104.           axesShaderProgram_->bind();
105.           axesVertexObject_.bind_vao();
106.           glLineWidth(3.0f);
107.           /* set axes rotation */
108.           {
109.               int          rotationMatrixLocation          =          axesShaderProgram_-
    >uniformLocation("rotation");
110.               axesShaderProgram_->setUniformValue(rotationMatrixLocation,            cam-
    era_.getRotation());
111.           }
112.           glDrawArrays(GL_LINES, 0, 6);
113.           axesVertexObject_.unbind_vao();
114.           axesShaderProgram_->release();
115.
116.           /* draw light source */
117.           lightSourceShaderProgram_->bind();
118.           lightSourceShaderProgram_->setUniformValue("model", modelMatrix_);
119.           lightSourceShaderProgram_->setUniformValue("view", camera_.getView());
120.           lightSourceShaderProgram_->setUniformValue("projection", projectionMatrix_);
121.           lightSourceShaderProgram_->setUniformValue("cameraPos", camera_.getPosition());
122.           if (lighting_.getType() != GLLightingType::Directional)
123.           {
124.               lightSource_->setTranslation(
125.                   -lighting_.getPosition().x(),
126.                   -lighting_.getPosition().y(),
127.                   -lighting_.getPosition().z()
128.               );
129.               lighting_.apply(lightSourceShaderProgram_);
130.               lightSource_->draw(this, fragmentationFactor_);
131.           }
132.           lightSourceShaderProgram_->release();
133.       }
134.
135.
136.       void GLScene::createShaderProgram()
137.       {
138.           QMap<qopenglshader::shadertype, qstring=""> figureShaders;
139.
140.           figureShaders[QOpenGLShader::Vertex] = ":/shaders/figure.vert";
141.           figureShaders[QOpenGLShader::Fragment] = ":/shaders/figure.frag";
142.
143.           figureShaderProgram_ = new GLShaderProgram(figureShaders, this);
144.
145.           QMap<qopenglshader::shadertype, qstring=""> axesShaders;
```

```
146.
147.            axesShaders[QOpenGLShader::Vertex] = ":/shaders/axes.vert";
148.            axesShaders[QOpenGLShader::Fragment] = ":/shaders/axes.frag";
149.
150.            axesShaderProgram_ = new GLShaderProgram(axesShaders, this);
151.
152.            QMap<qopenglshader::shadertype, qstring=""> lightSourceShaders;
153.
154.            lightSourceShaders[QOpenGLShader::Vertex] = ":/shaders/lightSource.vert";
155.            lightSourceShaders[QOpenGLShader::Fragment] = ":/shaders/lightSource.frag";
156.
157.            lightSourceShaderProgram_ = new GLShaderProgram(lightSourceShaders, this);
158.        }
159.
160.
161.        void GLScene::generateFigures()
162.        {
163.            /* generate different scene objects */
164.            GLfloat pi = 3.14159265f;
165.
166.            QVector<circlebasedata> base1;
167.            for (GLfloat y = 0.700f; y >= -0.701f; y -= 0.01)
168.            {
169.                base1.push_back({
170.                    0.70f * std::abs(
171.                        std::sin(y / 0.70f * 0.40f * pi)
172.                    ), y
173.                });
174.            }
175.            for (GLfloat y = -0.700f; y <= 0.701f; y += 0.01)
176.            {
177.                base1.push_back({
178.                    0.75f * std::sin(
179.                        std::abs(y) / 0.70f * 0.70f * pi
180.                    ) + 0.25f, y
181.                });
182.            }
183.            base1.push_back({0.70f, 0.70f});
184.            figures_.push_back(std::make_shared<glfigure>(base1));
185.            figures_.last()->setScale(0.25f, 0.25f, 0.25f);
186.            figures_.last()->setRotation(20.f, { 0.3f, 1.f, 0.5f });
187.            figures_.last()->setTranslation(-0.6f, 0.5f, -0.1f);
188.            figures_.last()->getMaterial().setDiffuseColor(Qt::green);
189.
190.            QVector<circlebasedata> base2;
191.            for (GLfloat y = 0.700f; y >= -0.701f; y -= 0.01)
192.            {
193.                base2.push_back({
194.                    0.0001f * std::exp(std::abs(y) / 0.70f * 9.1f) + 0.05f,
195.                    y,
196.                });
197.            }
198.            figures_.push_back(std::make_shared<glfigure>(base2));
199.            figures_.last()->setScale(0.3f, 0.3f, 0.3f);
200.            figures_.last()->setRotation(-40.f, { 1.f, 0.f, 0.2f });
201.            figures_.last()->setTranslation(0.25f, -0.5f, 0.1f);
202.            figures_.last()->getMaterial().setDiffuseColor(Qt::yellow);
203.
204.            QVector<circlebasedata> base3;
205.            for (GLfloat y = 0.700f; y >= -0.701f; y -= 0.01)
206.            {
207.                base3.push_back({
208.                    0.0001f * std::exp(y / 0.70f * 9.1f) + 0.015f + 0.03f * (y + 0.70f),
209.                    y,
210.                });
211.            }
212.            figures_.push_back(std::make_shared<glfigure>(base3));
213.            figures_.last()->setScale(0.35f, 0.35f, 0.35f);
```

```
214.            figures_.last()->setRotation(-20.f, { 0.f, 1.f, 1.f });
215.            figures_.last()->setTranslation(0.1f, 0.5f, 0.3f);
216.            figures_.last()->getMaterial().setDiffuseColor(Qt::cyan);
217.
218.            QVector<circlebasedata> base4;
219.            for (GLfloat y = 0.450f; y >= -0.451f; y -= 0.01)
220.            {
221.                base4.push_back({
222.                    0.5f - std::sqrt(0.4500000001f * 0.4500000001f - y * y),
223.                    y
224.                });
225.            }
226.            for (GLfloat y = -0.450f; y <= 0.451f; y += 0.01)
227.            {
228.                base4.push_back({
229.                    0.5f + std::sqrt(0.4500000001f * 0.4500000001f - y * y),
230.                    y
231.                });
232.            }
233.            base4.push_back({0.50f, 0.45f});
234.            figures_.push_back(std::make_shared<glfigure>(base4));
235.            figures_.last()->setScale(0.3f, 0.3f, 0.3f);
236.            figures_.last()->setRotation(-50.f, { 1.f, 1.f, 0.f });
237.            figures_.last()->setTranslation(-0.6f, -0.3f, -0.4f);
238.            figures_.last()->getMaterial().setDiffuseColor(QColor{ 52, 229, 235 });
239.
240.            QVector<circlebasedata> baseLightSource;
241.            for (GLfloat y = 0.450f; y >= -0.451f; y -= 0.01)
242.            {
243.                baseLightSource.push_back({
244.                    0.450f * std::sin(std::acos(std::abs(y) / 0.450f)),
245.                    y
246.                });
247.            }
248.            lightSource_ = std::make_shared<glfigure>(baseLightSource);
249.            lightSource_->setScale(0.2f, 0.2f, 0.2f);
250.        }
251.
252.
253.        void GLScene::generateAxes()
254.        {
255.            /* x axis */
256.            axesVertices_.push_back({
257.                { 0.0, 0.0, 0.0 },
258.                { 0.8, 0.2, 0.2 }
259.            });
260.            axesVertices_.push_back({
261.                { 0.1, 0.0, 0.0 },
262.                { 0.8, 0.2, 0.2 }
263.            });
264.            /* y axis */
265.            axesVertices_.push_back({
266.                { 0.0, 0.0, 0.0 },
267.                { 0.2, 0.8, 0.2 }
268.            });
269.            axesVertices_.push_back({
270.                { 0.0, 0.1, 0.0 },
271.                { 0.2, 0.8, 0.2 }
272.            });
273.            /* z axis */
274.            axesVertices_.push_back({
275.                { 0.0, 0.0, 0.0 },
276.                { 0.2, 0.2, 0.8 }
277.            });
278.            axesVertices_.push_back({
279.                { 0.0, 0.0, 0.1 },
280.                { 0.2, 0.2, 0.8 }
281.            });
```

```
282.            /* load axes data to axes buffer */
283.            axesVertexObject_.bind_vao();
284.            axesVertexObject_.loadVertices(axesVertices_);
285.            axesVertexObject_.setupVertexAttribute(
286.                this, 0, 3, GL_FLOAT, GL_TRUE,
287.                sizeof(QVector3D) * 2,
288.                nullptr
289.            );
290.            axesVertexObject_.setupVertexAttribute(
291.                this, 1, 3, GL_FLOAT, GL_TRUE,
292.                sizeof(QVector3D) * 2,
293.                reinterpret_cast<void*>(sizeof(QVector3D))
294.            );
295.            axesVertexObject_.unbind_vao();
296.        }
297.
298.
299.        void GLScene::keyPressEvent(QKeyEvent* event)
300.        {
301.            GLint forward = 0;
302.            GLint right = 0;
303.
304.            if (event->key() == Qt::Key_W)
305.            {
306.                forward += 1;
307.            }
308.            if (event->key() == Qt::Key_S)
309.            {
310.                forward -= 1;
311.            }
312.            if (event->key() == Qt::Key_D)
313.            {
314.                right += 1;
315.            }
316.            if (event->key() == Qt::Key_A)
317.            {
318.                right -= 1;
319.            }
320.
321.            camera_.move(forward, right);
322.            update();
323.        }
324.
325.
326.        void GLScene::mousePressEvent(QMouseEvent *event)
327.        {
328.            camera_.mousePress(event->position());
329.        }
330.
331.
332.        void GLScene::mouseMoveEvent(QMouseEvent *event)
333.        {
334.            camera_.rotate(event->position());
335.            update();
336.        }
337.
338.
339.        void GLScene::setProjectionType(GLProjectionType type)
340.        {
341.            float aspectRatio = width_ / height_;
342.            projection_ = type;
343.            projectionMatrix_.setToIdentity();
344.            // set projection matrix
345.            switch (projection_)
346.            {
347.                case GLProjectionType::Perspective:
348.                    projectionMatrix_.perspective(45.0f, aspectRatio, 0.01f, 100.0f);
349.                    break;
```

```
350.                 case GLProjectionType::Orthographic:
351.                     projectionMatrix_.ortho(
352.                         -1.0f * aspectRatio,
353.                         1.0f * aspectRatio,
354.                         -1.0f,
355.                         1.0f,
356.                         -8.f,
357.                         8.f
358.                     );
359.                     break;
360.                 default:
361.                     break;
362.             }
363.         }
```

Реализация шейдеров

Для отрисовки различных объектов сцены были реализованы шейдеры для фигур, осей и источника освещения.

Реализация вершинного шейдера для фигур.

В качестве входных параметров данный шейдер принимает позицию вершины и ее норму. В качестве выходных параметров выступают вершина и координаты вершины без учета преобразований проекционных и видовых (фраментные координаты). Также, с помощью uniform-переменных в шейдер передаются матрицы модельных, видовых и проекционных преобразований. Координата вершины получается путем перемножения применения проекционных, видовых и матричных преобразований к позиции вершины (входной параметр).

```
1.  #version 460 core
2.
3.  layout (location = 0) in vec3 aPos;
4.  layout (location = 1) in vec3 aNorm;
5.
6.  out vec3 Norm;
7.  out vec3 FragPos;
8.
9.  uniform mat4 model;
10. uniform mat4 view;
11. uniform mat4 projection;
12.
13. void main()
14. {
15.     gl_Position = projection * view * model * vec4(aPos, 1.0);
16.     Norm = aNorm;
17.     FragPos = vec3(model * vec4(aPos, 1.0));
18. }
```

Реализация фрагментного шейдера для фигуры.

В качестве входных параметров шейдер принимает норму вершины и фрагментные координаты. В качестве выходного параметра выступает цвет пикселя. В качестве uniform-параметров шейдер принимает позицию камеры, описание материала и освещения. В шейдере просчитывается освещение согласно модели Блинна-Фонга. Также поддерживаются различные типы света: точечный свет, направленный свет и прожектор.

```
#version 460 core

struct Material
{
    vec3 diffuse;
    vec3 specular;

    float shininess;
    float ambient_strength;
};

struct Light
{
    vec3 position;
    vec4 direction;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;

    vec3 attenuation_coeffs;
    float cut_off;
    float outer_cut_off;
};

in vec3 Norm;
in vec3 FragPos;

out vec4 FragColor;

uniform vec3 cameraPos;
uniform Material material;
uniform Light light;

void main()
{
    // common
    vec3 norm = normalize(Norm);
    vec3 lightDir = light.direction.w > 0.0f ?
        normalize(light.position.xyz - FragPos) :
        normalize(-light.direction.xyz);
    vec3 viewDir = normalize(cameraPos - FragPos);
    vec3 halfwayDir = normalize(lightDir + viewDir);

    // ambient light
    vec3 ambient = material.ambient_strength * light.ambient * material.diffuse;

    // diffuse light
    float diffuseIntensity = max(dot(norm, lightDir), 0.0f);
    vec3 diffuse = diffuseIntensity * light.diffuse * material.diffuse;

    // specular light
    float spec = pow(max(dot(norm, halfwayDir), 0.0f), material.shininess);
```

```
    vec3 specular = spec * light.specular * material.specular;

    if (light.direction.w > 0.0f)
    {
        if (light.cut_off > 0.0f)
        {
            // spotlight settings
            float theta = dot(lightDir, normalize(-light.direction.xyz));
            float epsilon = (light.cut_off - light.outer_cut_off);
            float intensity = smoothstep(0.0, 1.0, (theta - light.outer_cut_off) / epsi-
lon);
            diffuse  *= intensity;
            specular *= intensity;
        }

        // attenuation settings
        float distance = length(light.position - FragPos);
        float attenuation = 1.0f / (
            light.attenuation_coeffs.x +
            light.attenuation_coeffs.y * distance +
            light.attenuation_coeffs.z * distance * distance
        );

        ambient *= attenuation;
        diffuse *= attenuation;
        specular *= attenuation;
    }

    vec3 result = ambient + diffuse + specular;
    FragColor = vec4(result, 1.0f);
}
```

Реализация вершинного шейдера для источника освещения полностью совпадает с реализацией вершинного шейдера для фигуры.

```
#version 460 core

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

out vec3 Norm;
out vec3 FragPos;

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

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    Norm = aNorm;
    FragPos = vec3(model * vec4(aPos, 1.0));
}
```

Реализация фрагментного шейдера для источника освещения совпадает с реализацией фрагментного шейдера, за исключением того, что в нем не просчитывается интенсивность рассеянного освещения, а также не просчитываются параметры для прожектора.

```
#version 460 core

struct Light
{
    vec3 position;
    vec4 direction;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;

    vec3 attenuation_coeffs;
    float cut_off;
    float outer_cut_off;
};

in vec3 Norm;
in vec3 FragPos;

out vec4 FragColor;

uniform vec3 cameraPos;
uniform Light light;

void main()
{
    // common
    vec3 norm = normalize(Norm);
    vec3 lightDir = light.direction.w > 0.0f ?
        normalize(light.position.xyz - FragPos) :
        normalize(-light.direction.xyz);
    vec3 viewDir = normalize(cameraPos - FragPos);
    vec3 halfwayDir = normalize(lightDir + viewDir);

    // ambient light
    vec3 ambient = 0.05 * light.ambient;

    // diffuse light
    float diffuseIntensity = max(dot(norm, lightDir), 0.0f);
    vec3 diffuse = light.diffuse;

    // specular light
    float spec = pow(max(dot(norm, halfwayDir), 0.0f), 512.0f);
    vec3 specular = spec * light.specular;

    vec3 result = ambient + diffuse + specular;
    FragColor = vec4(result, 1.0f);
}
```

Реализация вершинного шейдера для отрисовки осей координат.

В качестве входных параметров шейдер принимает позицию вершины и цвет оси, которой принадлежит данная вершина. В качестве выходного параметра передается цвет вершины. В качестве uniform-переменной передается матрица поворота камеры. Позиция вершины изменяется с помощью матрицы сдвига: оси координат сдвигаются в правую часть сцены. Цвет передается в фрагментный шейдер.

```
#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;
}
```

Реализация фрагментного шейдера для осей координат состоит только из установки цвета пикселя в значение переданного в шейдер цвета.

```
#version 460 core

in vec3 color;
out vec4 FragColor;

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

Тестирование.

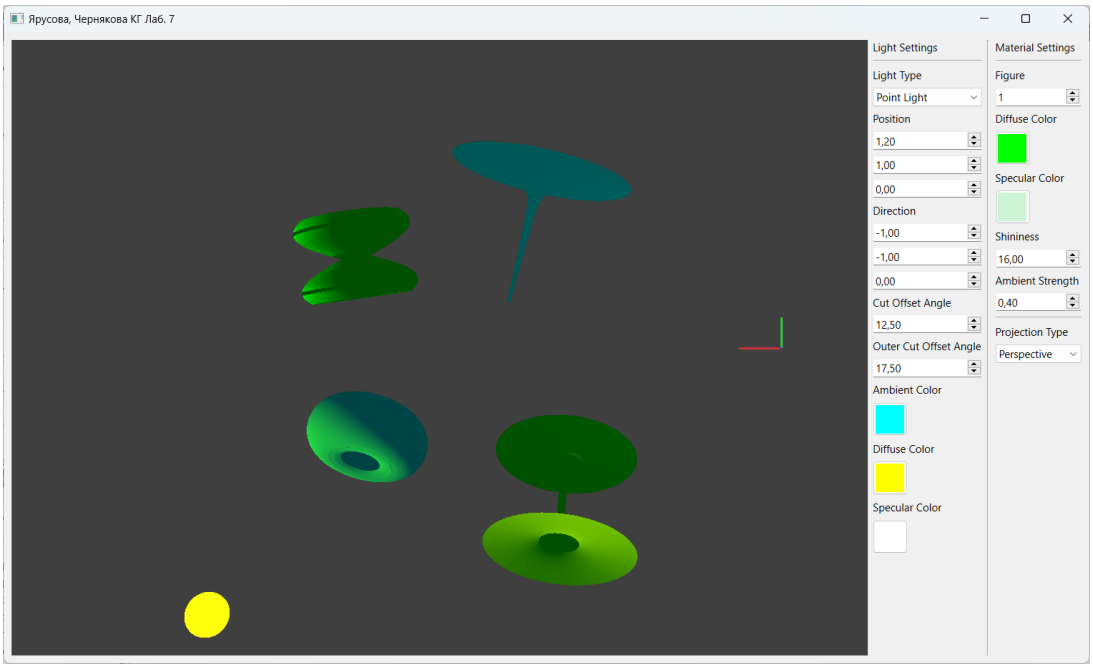

Рисунок 1.

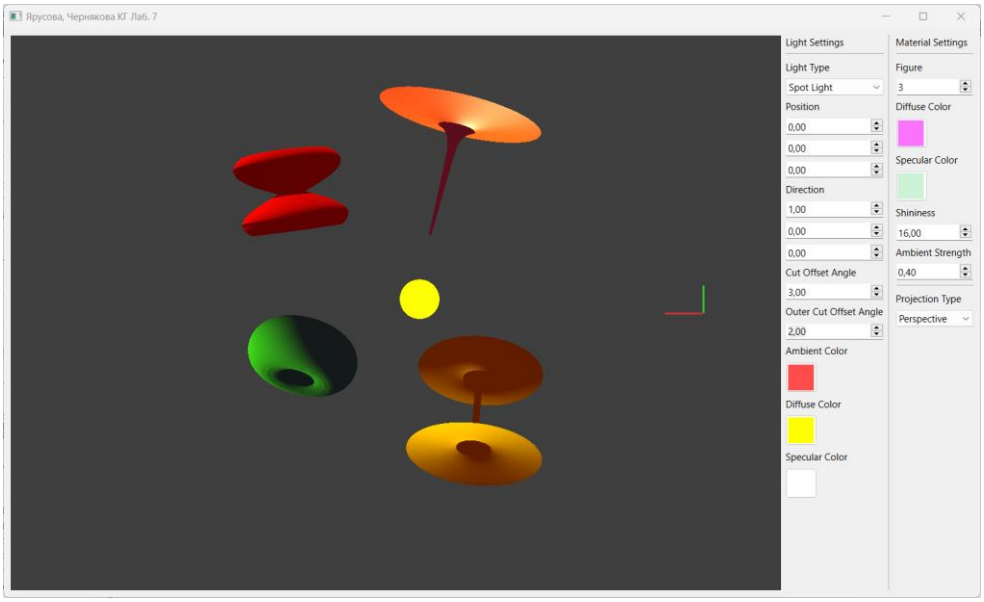

Рисунок 2.

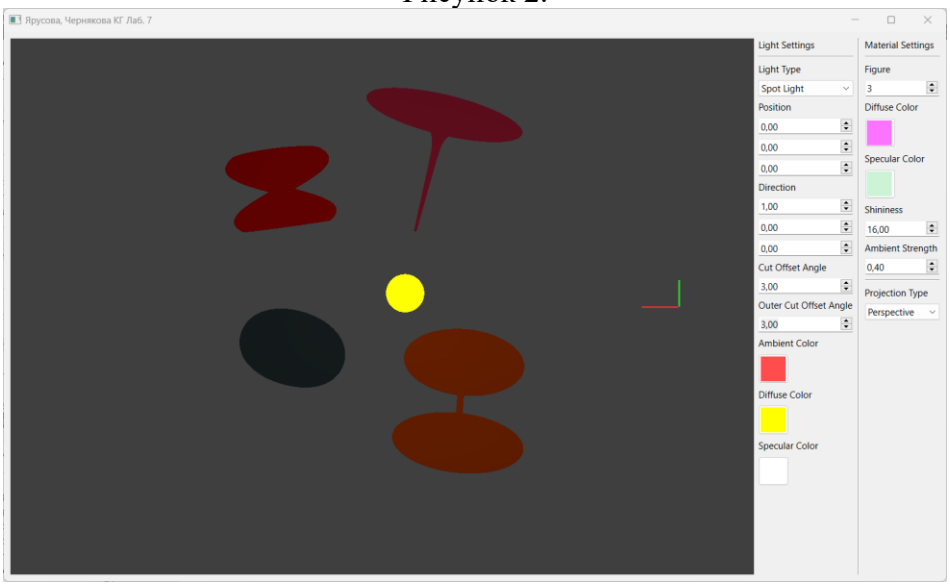

Рисунок 3.

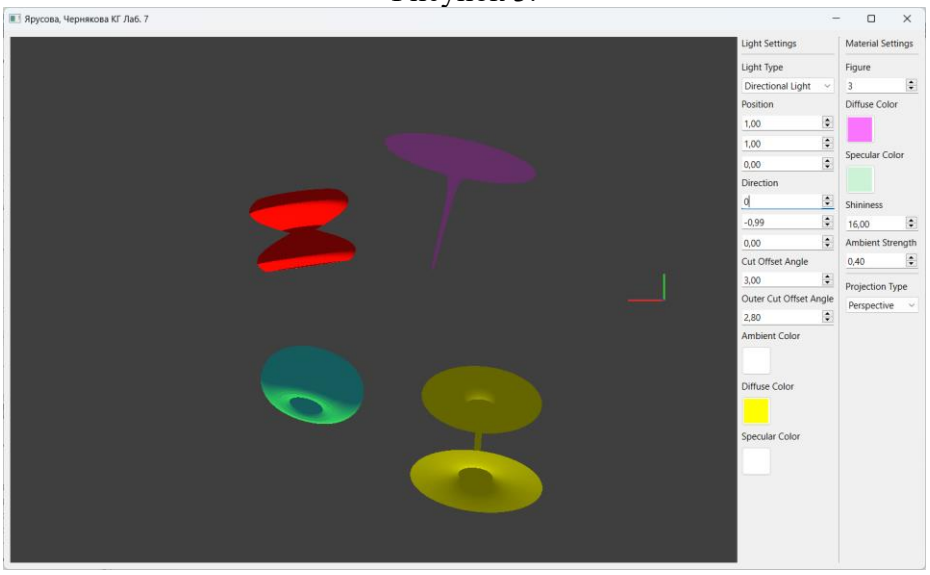

Рисунок 4.

Рисунок 5.

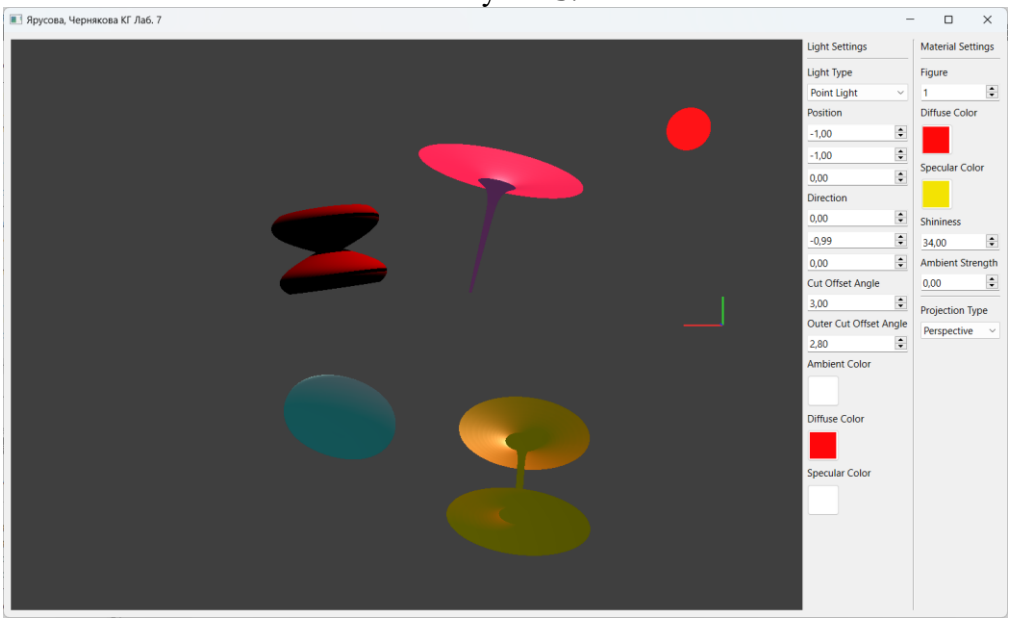

Рисунок 6.

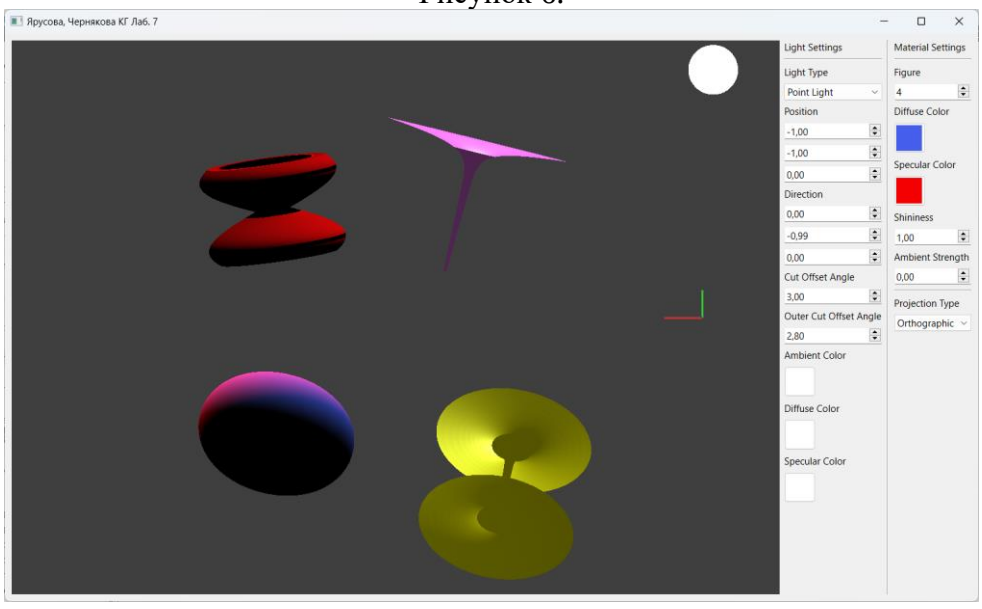

Рисунок 7.

Выводы.

В ходе работы была разработана программа, позволяющая управлять 3D сценой с несколькими фигурами. В программе присутствует возможность интерактивно управлять камерой, задавать различные настройки освещения, задавать различные настройки материала и изменять тип проекции. Программа реализована с помощью Qt6 и C++, шейдеры написаны на языке GLSL версии 4.6.