**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

|  |  |  |
| --- | --- | --- |
| 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.

- Изучение способов использования освещения

- Изучение моделей освещения

**Задание.**

Разработать программу, реализующую представление трехмерной сцены 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.**

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

1. Реализация камеры.

Для возможности использования камеры для перемещения по сцене был реализован класс Camera. Устройство класса Camera см. на рис. 1.

Рисунок 1. Устройство класса камеры

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.

2. Реализация представления материала.

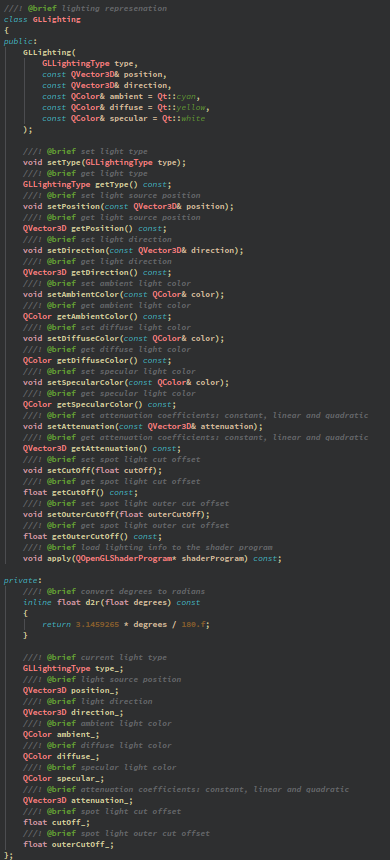
Для представления материала был реализован класс GLMaterial. Устройство класса GLMaterial см. на рис. 2.

Рисунок 2. Устройство класса 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.

3. Реализация представления освещения.

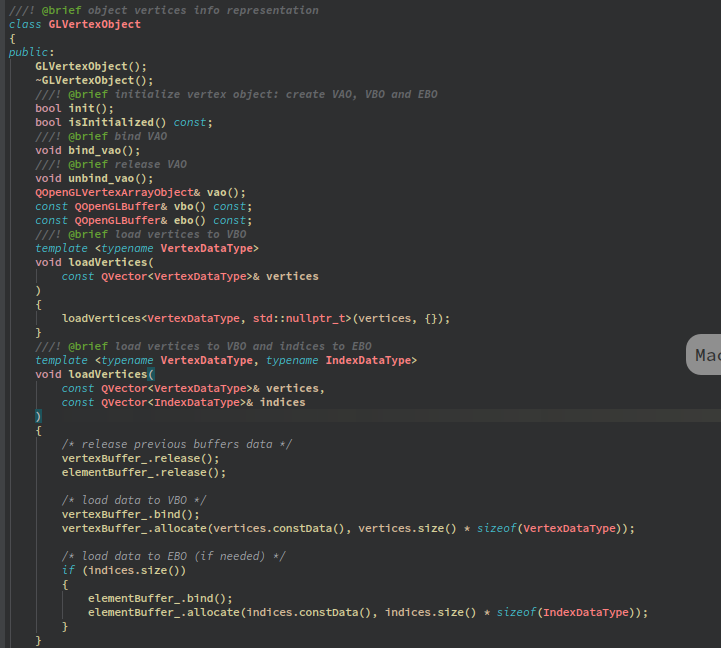
Для представления освещения был реализован класс GLLighting. Устройство класса GLLighting см. на рис. 3.

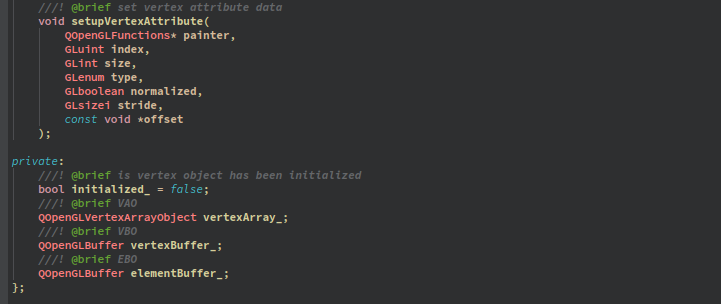
Рисунок 3. Устройство класса GLLighting

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.

4. Реализация обертки для работы с буферами OpenGL.

Для упрощения работы с буферами OpenGL (VAO, VBO, EBO) был реализован класс GLVertexObject. Устройство класса см. на рис. 4.

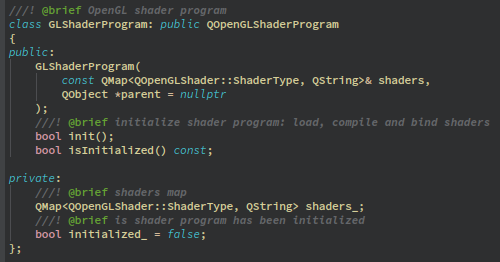


Рисунок 4. Устройство класса GLVertexObject

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.

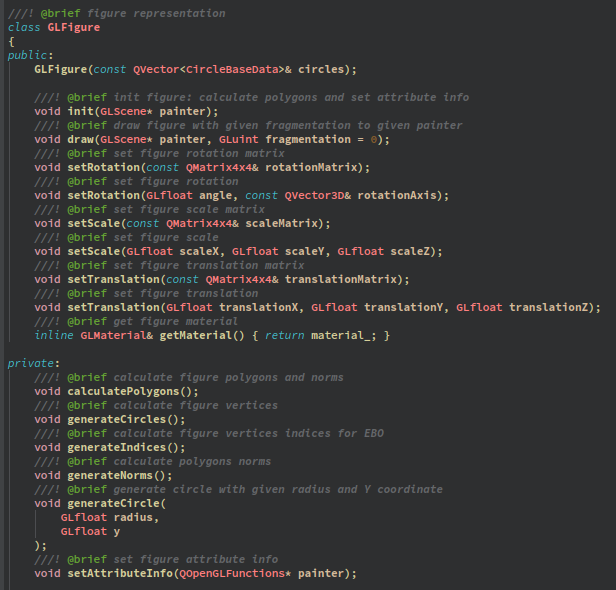
5. Реализация обертки для работы с шейдерной программой.

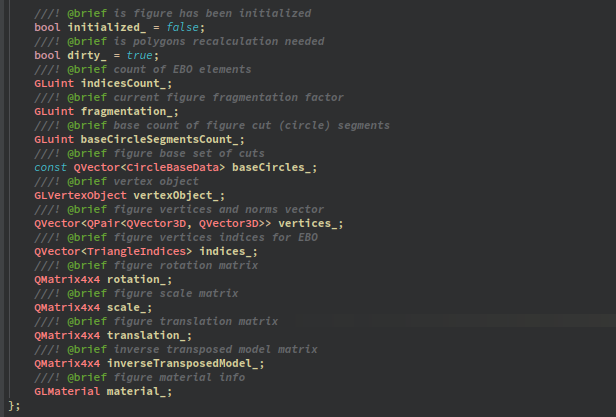
Для упрощения работы с шейдерными программами был реализован класс GLShaderProgram. Устройство класса GLShaderProgram см. на рис. 5.

Рисунок 5. Устройство класса 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.

6. Реализация представления фигуры

Для представления фигуры был реализован класс GLFigure. Устройство класса GLFigure см. на рис. 6.

Рисунок 6. Устройство класса 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.

7. Реализация сцены.

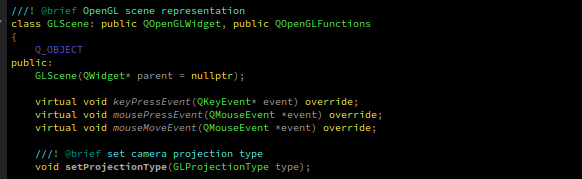
Для сцены был реализован класс GLScene. Устройство класса GLScene см. на рис. 7.

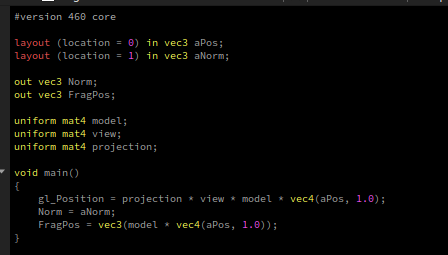
Рисунок 7. Устройство класса GLScene

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.

8. Реализация шейдеров.

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

Реализацию вершинного шейдера для фигур см. на рис. 8.

Рисунок 8. Реализация вершинного шейдера для фигуры

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).

Реализацию фрагментного шейдера для фигуры см. на рис. 9.



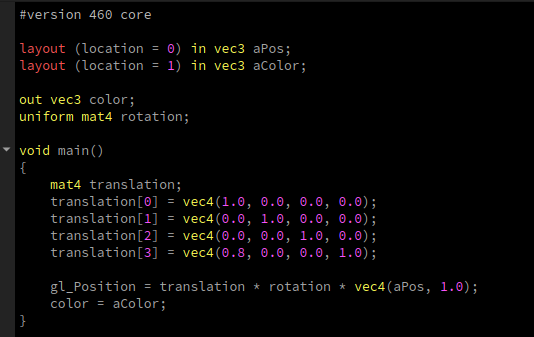
Рисунок 9. Реализация фрагментного шейдера для фигуры

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.

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

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.

Реализацию вершинного шейдера для отрисовки осей координат см. на рис. 10.

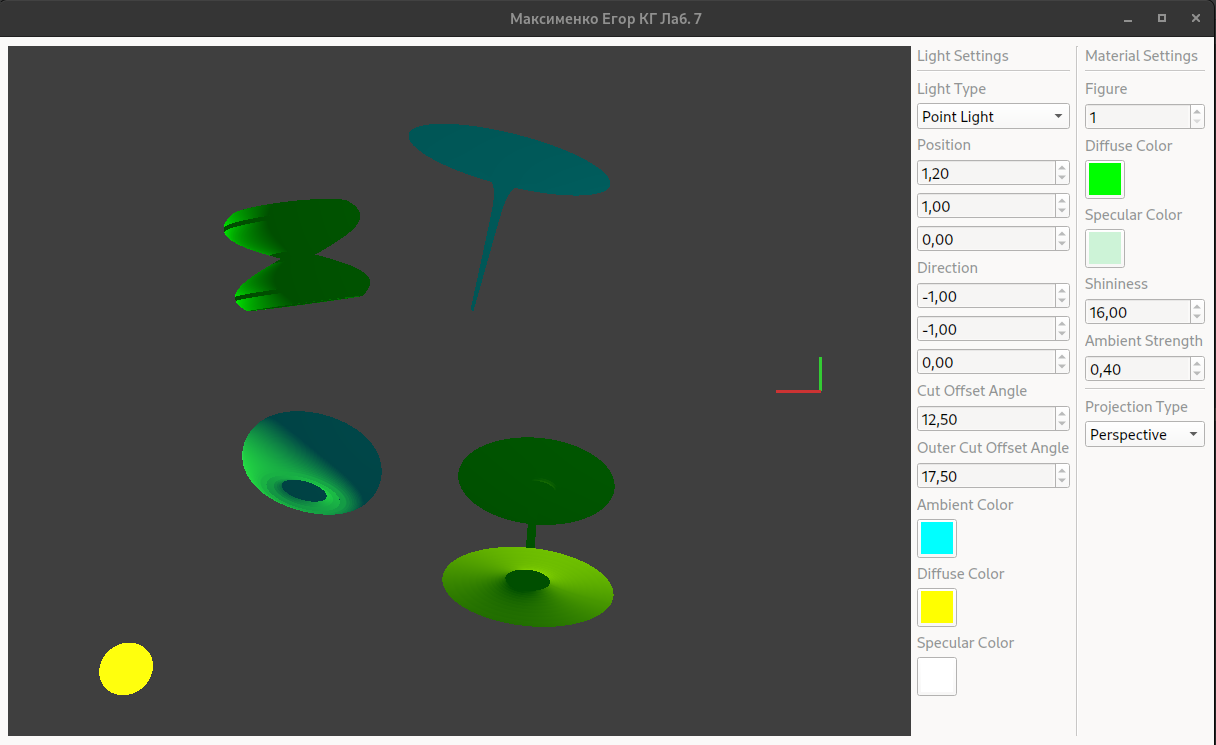
Рисунок 10. Реализация вершинного шейдера для осей координат

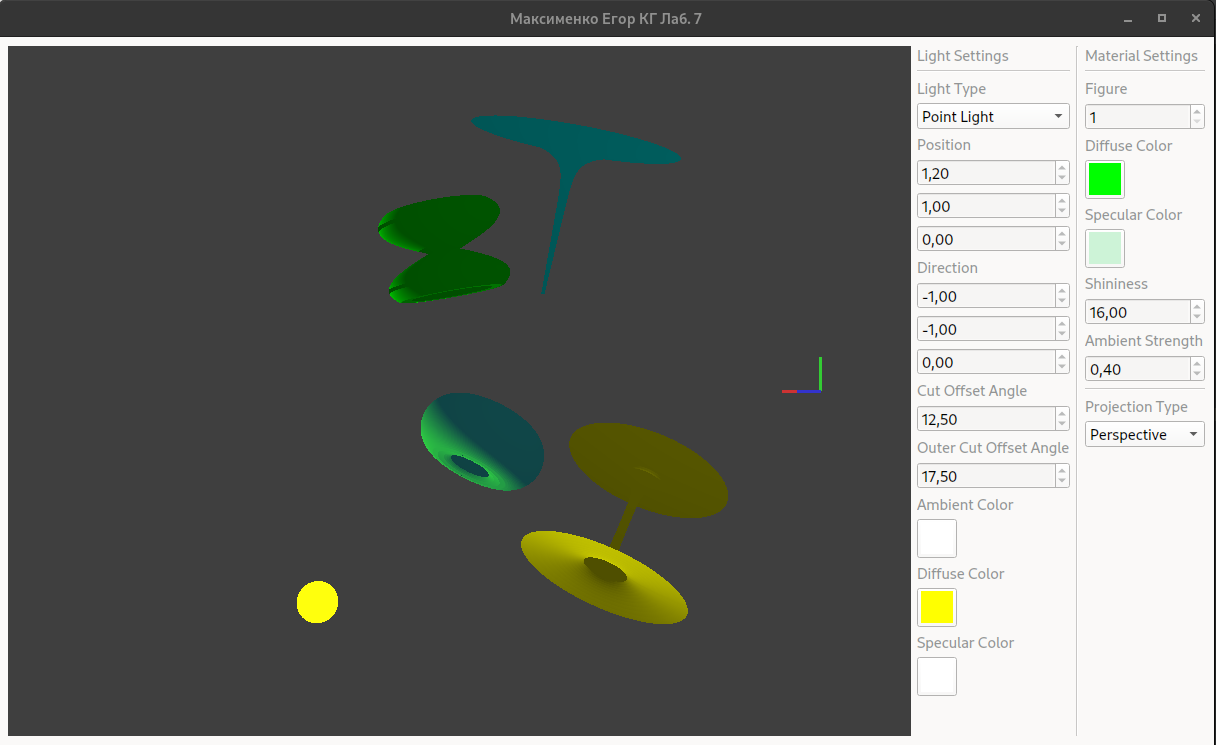
В качестве входных параметров шейдер принимает позицию вершины и цвет оси, которой принадлежит данная вершина. В качестве выходного параметра передается цвет вершины. В качестве uniform-переменной передается матрица поворота камеры. Позиция вершины изменяется с помощью матрицы сдвига: оси координат сдвигаются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

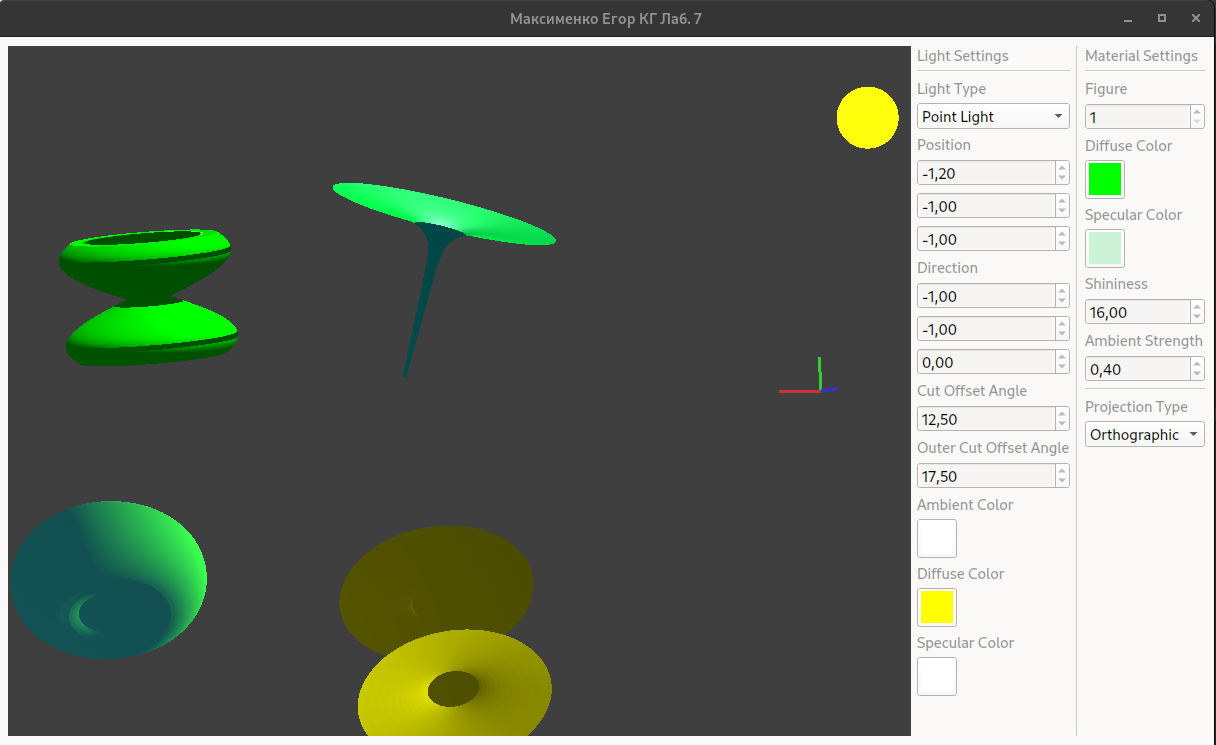
Реализация фрагментного шейдера для осей координат состоит только из установки цвета пикселя в значение переданного в шейдер цвета.

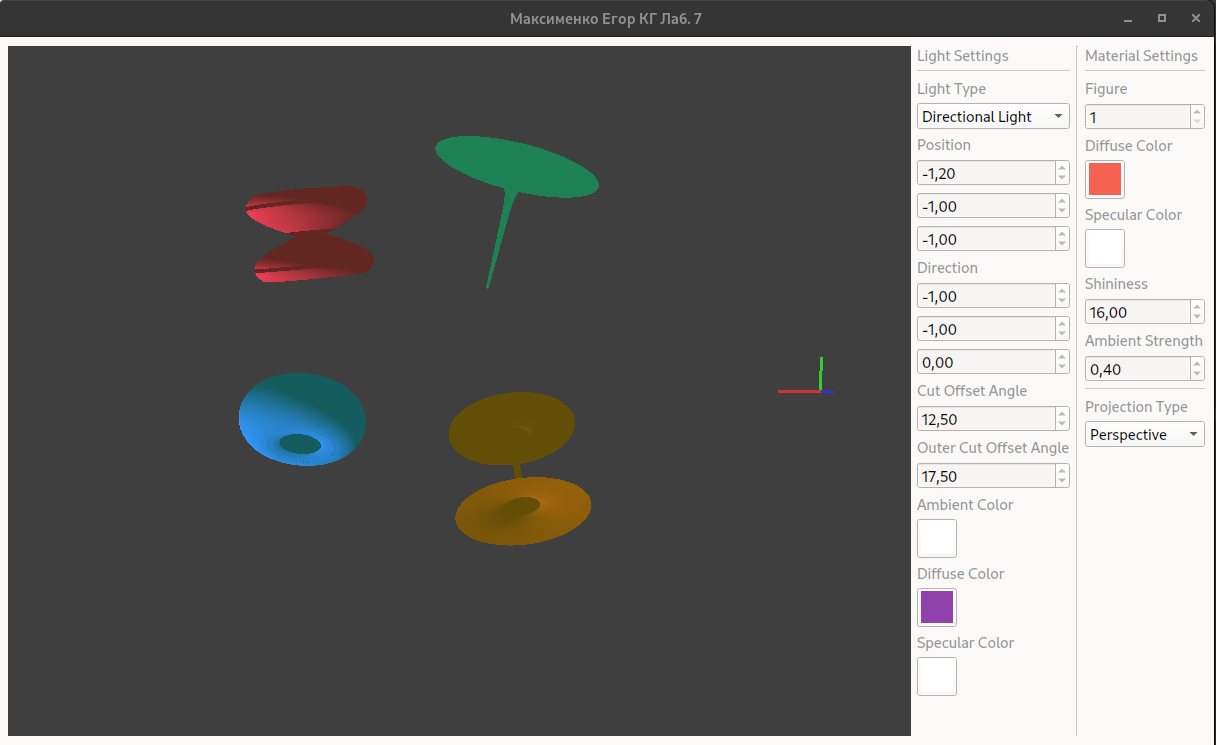
**Тестирование программы.**

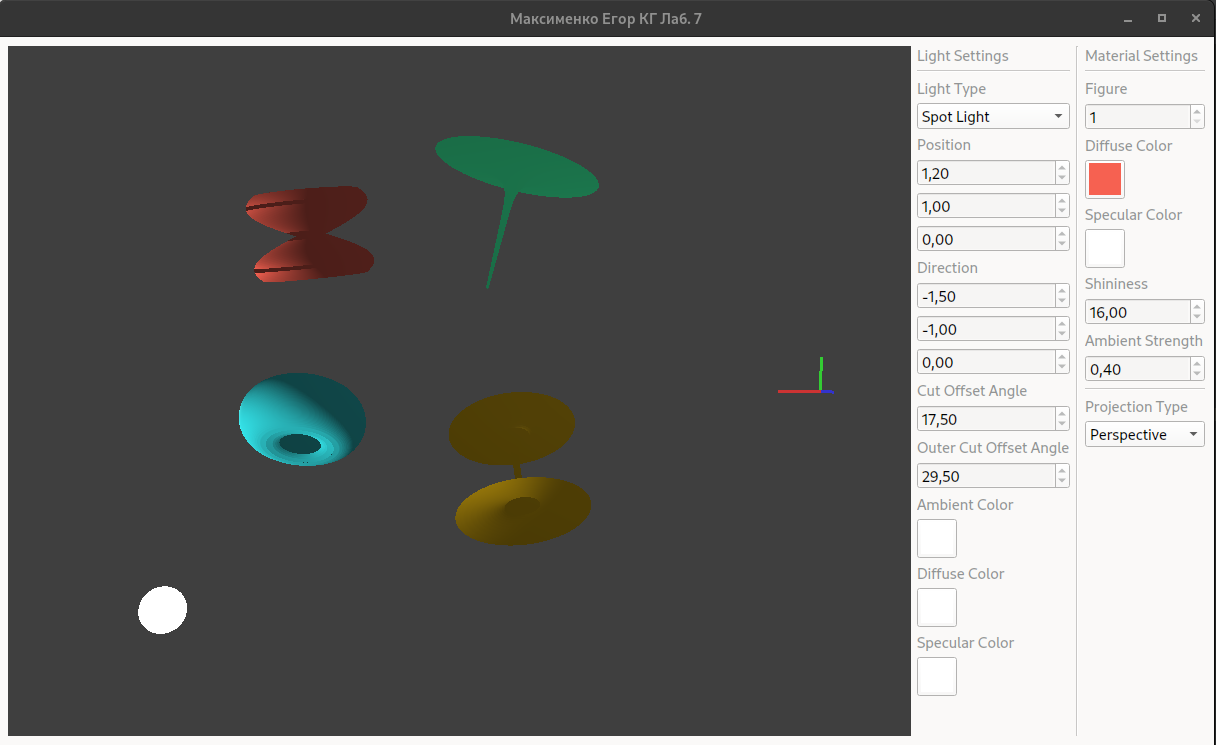
Программа была протестирована с различными параметрами сцены и освещения. Результаты тестов см. на рис. 11-16.

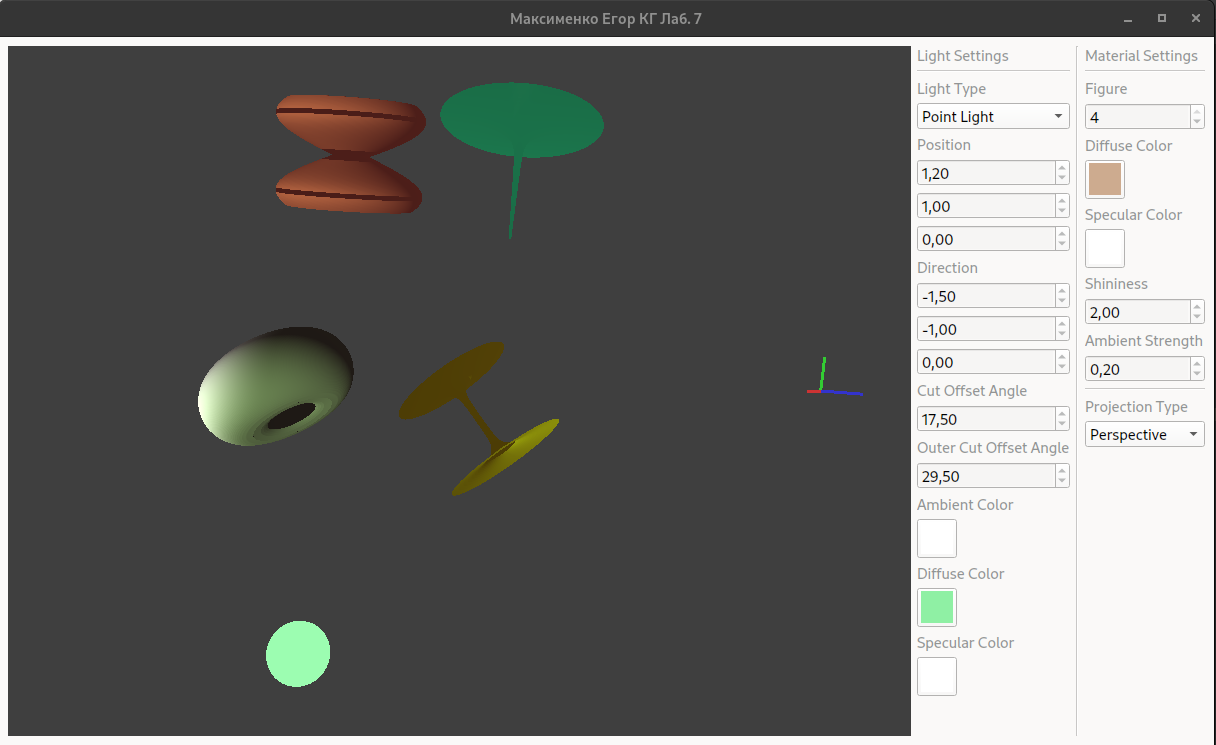
Рисунок 11. Тестирование программы. Запуск 1

Рисунок 12. Тестирование программы. Запуск 2

Рисунок 13. Тестирование программы. Запуск 3

Рисунок 14. Тестирование программы. Запуск 4

Рисунок 15. Тестирование программы. Запуск 5

Рисунок 16. Тестирование программы. Запуск 6

**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. Исходный код представлен в приложении 1.

**Приложение 1**

**Исходный код программы**

Файл: main.cpp

#include "mainwindow.h"

#include <QApplication>

int main(int argc, char \*argv[])

{

QApplication a(argc, argv);

MainWindow w;

w.show();

w.setWindowTitle("Максименко Егор КГ Лаб. 7");

return a.exec();

}

Файл: camera.h

#ifndef CAMERA\_H

#define CAMERA\_H

#include <QMatrix4x4>

#include <QVector3D>

///! @brief camera representation

class Camera

{

public:

Camera(

const QPointF& screenSize,

float speed = 0.02f,

const QVector3D& position = { 0.f, 0.f, 3.f }

);

///! @brief move camera forward and right

void move(int forward, int right);

///! @brief rotate camera to look at xPos and yPos

void rotate(float xPos, float yPos);

///! @brief rotate camera to look at position

void rotate(const QPointF& position);

///! @brief setup mouse press position

void mousePress(const QPointF& position);

///! @brief set window size

void resize(const QPointF& size);

inline QVector3D getPosition() const { return position\_; }

inline QMatrix4x4 getView() const { return matrix\_; }

inline QMatrix4x4 getRotation() const { return rotation\_; }

private:

///! @brief convert degrees to radians

inline float d2r(float degrees) const

{

return 3.1459265 \* degrees / 180.f;

}

///! @brief set front and up vectors and calculate right vector

void setDirectionVectors(const QVector3D& front, const QVector3D& up);

///! @brief recalculate view and rotation matrices

void rebuildMatrix();

///! @brief view matrix

QMatrix4x4 matrix\_;

///! @brief camera rotation matrix

QMatrix4x4 rotation\_;

///! @brief camera movement speed

float speed\_;

///! @brief camera position

QVector3D position\_;

///! @brief camera front direction vector

QVector3D frontVector\_;

///! @brief camera up direction vector

QVector3D upVector\_;

///! @brief camera right direction vector

QVector3D rightVector\_;

///! @brief mouse sensitivity for camera rotation

float sensitivity\_ = 0.01f;

///! @brief camera yaw

float yaw\_ = -90.f;

///! @brief camera pitch

float pitch\_ = 0.f;

///! @brief window size

QPointF screenSize\_;

///! @brief mouse last location after press

QPointF lastMousePosition\_;

};

#endif // CAMERA\_H

Файл: camera.cpp

#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 \*/

float xOffset = (xPos - lastMousePosition\_.x()) \* sensitivity\_;

float yOffset = (lastMousePosition\_.y() - yPos) \* sensitivity\_;

lastMousePosition\_ = { xPos, yPos };

yaw\_ += xOffset;

pitch\_ += yOffset;

if (pitch\_ > 89.f)

{

pitch\_ = 89.f;

}

else if (pitch\_ < -89.f)

{

pitch\_ = -89.f;

}

/\* set new camera direction \*/

QVector3D direction = {

std::cos(d2r(yaw\_)) \* std::cos(d2r(pitch\_)),

std::sin(d2r(pitch\_)),

std::sin(d2r(yaw\_)) \* std::cos(d2r(pitch\_))

};

setDirectionVectors(direction.normalized(), upVector\_);

rebuildMatrix();

}

void Camera::rotate(const QPointF& position)

{

rotate(position.x(), position.y());

}

void Camera::mousePress(const QPointF& position)

{

lastMousePosition\_ = { position.x(), position.y() };

}

void Camera::resize(const QPointF& size)

{

/\* update screen size and sensitivity \*/

screenSize\_ = size;

sensitivity\_ = 380.f / size.x();

}

Файл: glmaterial.h

#ifndef GLMATERIAL\_H

#define GLMATERIAL\_H

#include <QColor>

class QOpenGLShaderProgram;

///! @brief material representation

class GLMaterial

{

public:

GLMaterial();

GLMaterial(

const QColor& diffuse,

const QColor& specular,

float shininess = 16.f,

float ambientStrength = 0.4f

);

///! @brief set material diffuse light color

void setDiffuseColor(const QColor& color);

///! @brief get material diffuse light color

QColor getDiffuseColor() const;

///! @brief set material specular light color

void setSpecularColor(const QColor& color);

///! @brief get material specular light color

QColor getSpecularColor() const;

///! @brief set material shininess

void setShininess(float shininess);

///! @brief get material shininess

float getShininess() const;

///! @brief set material ambient light strength

void setAmbientStrength(float strength);

///! @brief get material ambient light strength

float getAmbientStrength() const;

///! @brief load material info to the shader program

void apply(QOpenGLShaderProgram\* shaderProgram) const;

private:

///! @brief material diffuse light color

QColor diffuse\_;

///! @brief material specular light color

QColor specular\_;

///! @brief material shininess

float shininess\_;

///! @brief material ambient light strength

float ambientStrength\_;

};

#endif // GLMATERIAL\_H

Файл: glmaterial.cpp

#include "glmaterial.h"

#include <QOpenGLShaderProgram>

#include <iostream>

GLMaterial::GLMaterial():

GLMaterial {

QColor{ 237, 69, 57 },

QColor{ 205, 243, 215 },

}

{}

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;

}

float GLMaterial::getAmbientStrength() const

{

return ambientStrength\_;

}

void GLMaterial::apply(QOpenGLShaderProgram\* shaderProgram) const

{

if (!shaderProgram)

{

std::cerr << "[error] can't apply material settings: shader program is nullptr" << std::endl;

return;

}

shaderProgram->setUniformValue("material.diffuse", QVector3D{

diffuse\_.redF(),

diffuse\_.greenF(),

diffuse\_.blueF()

});

shaderProgram->setUniformValue("material.specular", QVector3D{

specular\_.redF(),

specular\_.greenF(),

specular\_.blueF()

});

shaderProgram->setUniformValue("material.shininess", shininess\_);

shaderProgram->setUniformValue("material.ambient\_strength", ambientStrength\_);

}

Файл: gllighting.h

#ifndef GLLIGHTING\_H

#define GLLIGHTING\_H

#include <QMap>

#include <QVector3D>

#include <QMatrix4x4>

#include <QColor>

class QOpenGLShaderProgram;

class QOpenGLFunctions;

///! @brief light type

enum class GLLightingType

{

Point,

Directional,

Spot

};

///! @brief lighting represenation

class GLLighting

{

public:

GLLighting(

GLLightingType type,

const QVector3D& position,

const QVector3D& direction,

const QColor& ambient = Qt::cyan,

const QColor& diffuse = Qt::yellow,

const QColor& specular = Qt::white

);

///! @brief set light type

void setType(GLLightingType type);

///! @brief get light type

GLLightingType getType() const;

///! @brief set light source position

void setPosition(const QVector3D& position);

///! @brief get light source position

QVector3D getPosition() const;

///! @brief set light direction

void setDirection(const QVector3D& direction);

///! @brief get light direction

QVector3D getDirection() const;

///! @brief set ambient light color

void setAmbientColor(const QColor& color);

///! @brief get ambient light color

QColor getAmbientColor() const;

///! @brief set diffuse light color

void setDiffuseColor(const QColor& color);

///! @brief get diffuse light color

QColor getDiffuseColor() const;

///! @brief set specular light color

void setSpecularColor(const QColor& color);

///! @brief get specular light color

QColor getSpecularColor() const;

///! @brief set attenuation coefficients: constant, linear and quadratic

void setAttenuation(const QVector3D& attenuation);

///! @brief get attenuation coefficients: constant, linear and quadratic

QVector3D getAttenuation() const;

///! @brief set spot light cut offset

void setCutOff(float cutOff);

///! @brief get spot light cut offset

float getCutOff() const;

///! @brief set spot light outer cut offset

void setOuterCutOff(float outerCutOff);

///! @brief get spot light outer cut offset

float getOuterCutOff() const;

///! @brief load lighting info to the shader program

void apply(QOpenGLShaderProgram\* shaderProgram) const;

private:

///! @brief convert degrees to radians

inline float d2r(float degrees) const

{

return 3.1459265 \* degrees / 180.f;

}

///! @brief current light type

GLLightingType type\_;

///! @brief light source position

QVector3D position\_;

///! @brief light direction

QVector3D direction\_;

///! @brief ambient light color

QColor ambient\_;

///! @brief diffuse light color

QColor diffuse\_;

///! @brief specular light color

QColor specular\_;

///! @brief attenuation coefficients: constant, linear and quadratic

QVector3D attenuation\_;

///! @brief spot light cut offset

float cutOff\_;

///! @brief spot light outer cut offset

float outerCutOff\_;

};

#endif // GLLIGHTING\_H

Файл: gllighting.cpp

#include "gllighting.h"

#include <QOpenGLShaderProgram>

#include <QOpenGLFunctions>

#include <QVector4D>

#include <iostream>

GLLighting::GLLighting(

GLLightingType type,

const QVector3D& position,

const QVector3D& direction,

const QColor& ambient,

const QColor& diffuse,

const QColor& specular

):

type\_{ type },

position\_{ position },

direction\_{ direction },

ambient\_{ ambient },

diffuse\_{ diffuse },

specular\_{ specular },

attenuation\_{ 1.0f, 0.09f, 0.032f },

cutOff\_{ 12.5f },

outerCutOff\_{ 17.5f }

{}

void GLLighting::setType(GLLightingType type)

{

type\_ = type;

}

GLLightingType GLLighting::getType() const

{

return type\_;

}

void GLLighting::setPosition(const QVector3D& position)

{

position\_ = position;

}

QVector3D GLLighting::getPosition() const

{

return position\_;

}

void GLLighting::setDirection(const QVector3D& direction)

{

direction\_ = direction;

}

QVector3D GLLighting::getDirection() const

{

return direction\_;

}

void GLLighting::setAmbientColor(const QColor& color)

{

ambient\_ = color;

}

QColor GLLighting::getAmbientColor() const

{

return ambient\_;

}

void GLLighting::setDiffuseColor(const QColor& color)

{

diffuse\_ = color;

}

QColor GLLighting::getDiffuseColor() const

{

return diffuse\_;

}

void GLLighting::setSpecularColor(const QColor& color)

{

specular\_ = color;

}

QColor GLLighting::getSpecularColor() const

{

return specular\_;

}

void GLLighting::setAttenuation(const QVector3D& attenuation)

{

attenuation\_ = attenuation;

}

QVector3D GLLighting::getAttenuation() const

{

return attenuation\_;

}

void GLLighting::setCutOff(float cutOff)

{

cutOff\_ = cutOff;

}

float GLLighting::getCutOff() const

{

return cutOff\_;

}

void GLLighting::setOuterCutOff(float outerCutOff)

{

outerCutOff\_ = outerCutOff;

}

float GLLighting::getOuterCutOff() const

{

return outerCutOff\_;

}

void GLLighting::apply(QOpenGLShaderProgram\* shaderProgram) const

{

if (!shaderProgram)

{

std::cerr << "[error] can't apply lighting settings: shader program is nullptr" << std::endl;

return;

}

if (GLLightingType::Point == type\_)

{

shaderProgram->setUniformValue("light.position", position\_);

shaderProgram->setUniformValue("light.direction", QVector4D{ 0.0f, 0.0f, 0.0f, 1.f });

} else if (GLLightingType::Directional == type\_)

{

shaderProgram->setUniformValue("light.position", QVector3D{ 0.0f, 0.0f, 0.0f });

shaderProgram->setUniformValue("light.direction", QVector4D{

direction\_.x(),

direction\_.y(),

direction\_.z(),

0.f

});

} else if (GLLightingType::Spot == type\_)

{

shaderProgram->setUniformValue("light.position", position\_);

shaderProgram->setUniformValue("light.direction", QVector4D{

direction\_.x(),

direction\_.y(),

direction\_.z(),

1.f

});

}

shaderProgram->setUniformValue("light.ambient", QVector3D{

ambient\_.redF(),

ambient\_.greenF(),

ambient\_.blueF()

});

shaderProgram->setUniformValue("light.diffuse", QVector3D{

diffuse\_.redF(),

diffuse\_.greenF(),

diffuse\_.blueF()

});

shaderProgram->setUniformValue("light.specular", QVector3D{

specular\_.redF(),

specular\_.greenF(),

specular\_.blueF()

});

shaderProgram->setUniformValue("light.attenuation\_coeffs", attenuation\_);

if (GLLightingType::Spot == type\_)

{

shaderProgram->setUniformValue("light.cut\_off", std::cos(d2r(cutOff\_)));

shaderProgram->setUniformValue("light.outer\_cut\_off", std::cos(d2r(outerCutOff\_)));

} else

{

shaderProgram->setUniformValue("light.cut\_off", 0.f);

shaderProgram->setUniformValue("light.outer\_cut\_off", 0.f);

}

}

Файл: glvertexobject.h

#ifndef GLVERTEXOBJECT\_H

#define GLVERTEXOBJECT\_H

#include <QOpenGLBuffer>

#include <QOpenGLVertexArrayObject>

class QOpenGLFunctions;

///! @brief object vertices info representation

class GLVertexObject

{

public:

GLVertexObject();

~GLVertexObject();

///! @brief initialize vertex object: create VAO, VBO and EBO

bool init();

bool isInitialized() const;

///! @brief bind VAO

void bind\_vao();

///! @brief release VAO

void unbind\_vao();

QOpenGLVertexArrayObject& vao();

const QOpenGLBuffer& vbo() const;

const QOpenGLBuffer& ebo() const;

///! @brief load vertices to VBO

template <typename VertexDataType>

void loadVertices(

const QVector<VertexDataType>& vertices

)

{

loadVertices<VertexDataType, std::nullptr\_t>(vertices, {});

}

///! @brief load vertices to VBO and indices to EBO

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));

}

}

///! @brief set vertex attribute data

void setupVertexAttribute(

QOpenGLFunctions\* painter,

GLuint index,

GLint size,

GLenum type,

GLboolean normalized,

GLsizei stride,

const void \*offset

);

private:

///! @brief is vertex object has been initialized

bool initialized\_ = false;

///! @brief VAO

QOpenGLVertexArrayObject vertexArray\_;

///! @brief VBO

QOpenGLBuffer vertexBuffer\_;

///! @brief EBO

QOpenGLBuffer elementBuffer\_;

};

#endif // GLVERTEXOBJECT\_H

Файл: glvertexobject.cpp

#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);

painter->glVertexAttribPointer(index, size, type, normalized, stride, offset);

}

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();

}

}

Файл: glshaderprogram.h

#ifndef GLSHADERPROGRAM\_H

#define GLSHADERPROGRAM\_H

#include <QOpenGLShaderProgram>

#include <QString>

#include <QMap>

///! @brief OpenGL shader program

class GLShaderProgram: public QOpenGLShaderProgram

{

public:

GLShaderProgram(

const QMap<QOpenGLShader::ShaderType, QString>& shaders,

QObject \*parent = nullptr

);

///! @brief initialize shader program: load, compile and bind shaders

bool init();

bool isInitialized() const;

private:

///! @brief shaders map

QMap<QOpenGLShader::ShaderType, QString> shaders\_;

///! @brief is shader program has been initialized

bool initialized\_ = false;

};

#endif // GLSHADERPROGRAM\_H

Файл: glshaderprogram.cpp

#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\_;

}

Файл: glfigure.h

#ifndef GLFIGURE\_H

#define GLFIGURE\_H

#include <QMatrix4x4>

#include "glvertexobject.h"

#include "glmaterial.h"

class GLScene;

///! @brief figure cut info

struct CircleBaseData

{

GLfloat radius;

GLfloat y;

};

///! @brief EBO element

struct TriangleIndices

{

GLuint first;

GLuint second;

GLuint third;

};

///! @brief figure representation

class GLFigure

{

public:

GLFigure(const QVector<CircleBaseData>& circles);

///! @brief init figure: calculate polygons and set attribute info

void init(GLScene\* painter);

///! @brief draw figure with given fragmentation to given painter

void draw(GLScene\* painter, GLuint fragmentation = 0);

///! @brief set figure rotation matrix

void setRotation(const QMatrix4x4& rotationMatrix);

///! @brief set figure rotation

void setRotation(GLfloat angle, const QVector3D& rotationAxis);

///! @brief set figure scale matrix

void setScale(const QMatrix4x4& scaleMatrix);

///! @brief set figure scale

void setScale(GLfloat scaleX, GLfloat scaleY, GLfloat scaleZ);

///! @brief set figure translation matrix

void setTranslation(const QMatrix4x4& translationMatrix);

///! @brief set figure translation

void setTranslation(GLfloat translationX, GLfloat translationY, GLfloat translationZ);

///! @brief get figure material

inline GLMaterial& getMaterial() { return material\_; }

private:

///! @brief calculate figure polygons and norms

void calculatePolygons();

///! @brief calculate figure vertices

void generateCircles();

///! @brief calculate figure vertices indices for EBO

void generateIndices();

///! @brief calculate polygons norms

void generateNorms();

///! @brief generate circle with given radius and Y coordinate

void generateCircle(

GLfloat radius,

GLfloat y

);

///! @brief set figure attribute info

void setAttributeInfo(QOpenGLFunctions\* painter);

///! @brief is figure has been initialized

bool initialized\_ = false;

///! @brief is polygons recalculation needed

bool dirty\_ = true;

///! @brief count of EBO elements

GLuint indicesCount\_;

///! @brief current figure fragmentation factor

GLuint fragmentation\_;

///! @brief base count of figure cut (circle) segments

GLuint baseCircleSegmentsCount\_;

///! @brief figure base set of cuts

const QVector<CircleBaseData> baseCircles\_;

///! @brief vertex object

GLVertexObject vertexObject\_;

///! @brief figure vertices and norms vector

QVector<QPair<QVector3D, QVector3D>> vertices\_;

///! @brief figure vertices indices for EBO

QVector<TriangleIndices> indices\_;

///! @brief figure rotation matrix

QMatrix4x4 rotation\_;

///! @brief figure scale matrix

QMatrix4x4 scale\_;

///! @brief figure translation matrix

QMatrix4x4 translation\_;

///! @brief inverse transposed model matrix

QMatrix4x4 inverseTransposedModel\_;

///! @brief figure material info

GLMaterial material\_;

};

#endif // GLFIGURE\_H

Файл: glfigure.cpp

#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)

{

rotation\_.setToIdentity();

rotation\_.rotate(angle, rotationAxis);

dirty\_ = true;

}

void GLFigure::setScale(const QMatrix4x4& scaleMatrix)

{

scale\_ = scaleMatrix;

dirty\_ = true;

}

void GLFigure::setScale(GLfloat scaleX, GLfloat scaleY, GLfloat scaleZ)

{

scale\_.setToIdentity();

scale\_.scale(scaleX, scaleY, scaleZ);

dirty\_ = true;

}

void GLFigure::setTranslation(const QMatrix4x4& translationMatrix)

{

translation\_ = translationMatrix;

dirty\_ = true;

}

void GLFigure::setTranslation(GLfloat translationX, GLfloat translationY, GLfloat translationZ)

{

translation\_.setToIdentity();

translation\_.translate(translationX, translationY, translationZ);

dirty\_ = true;

}

void GLFigure::calculatePolygons()

{

if (!initialized\_)

{

std::cerr << "[error] can't generate polygons: not initialized" << std::endl;

return;

}

if (!baseCircles\_.size())

{

std::cerr << "[error] can't generate polygons: base circles is empty" << std::endl;

return;

}

/\* recalculate inverse transposed matrix \*/

inverseTransposedModel\_ = rotation\_ \* translation\_ \* scale\_;

inverseTransposedModel\_ = inverseTransposedModel\_.inverted();

inverseTransposedModel\_ = inverseTransposedModel\_.transposed();

/\* prepare data \*/

generateCircles();

generateNorms();

generateIndices();

/\* load data to buffer \*/

vertexObject\_.bind\_vao();

vertexObject\_.loadVertices(vertices\_, indices\_);

/\* set dirty to false \*/

dirty\_ = false;

}

void GLFigure::generateCircles()

{

if (!initialized\_)

{

std::cerr << "[error] can't generate circles: not initialized" << std::endl;

return;

}

if (!baseCircles\_.size())

{

std::cerr << "[error] can't generate circles: base circles is empty" << std::endl;

return;

}

/\* remove previous data \*/

vertices\_.clear();

/\* circle parameters \*/

GLfloat radius = baseCircles\_.at(0).radius;

GLfloat y = baseCircles\_.at(0).y;

/\* generate circles \*/

for (GLuint i = 0; i < baseCircles\_.size() - 1; ++i)

{

/\* deltas between circles parameters \*/

GLfloat deltaRadius = (

baseCircles\_.at(i + 1).radius -

baseCircles\_.at(i).radius

) / fragmentation\_;

GLfloat deltaY = (

baseCircles\_.at(i + 1).y -

baseCircles\_.at(i).y

) / fragmentation\_;

/\* generate intermediate circles \*/

for (GLuint j = 0; j < fragmentation\_; ++j)

{

generateCircle(radius, y);

radius += deltaRadius;

y += deltaY;

}

}

/\* generate last circle \*/

generateCircle(radius, y);

}

void GLFigure::generateIndices()

{

if (!initialized\_)

{

std::cerr << "[error] can't generate indices: not initialized" << std::endl;

return;

}

if (!vertices\_.size())

{

std::cerr << "[error] can't generate indices: vertices is empty" << std::endl;

return;

}

/\* remove previous data \*/

indices\_.clear();

/\* circles count \*/

GLuint circleSegmentsCount = fragmentation\_ \* baseCircleSegmentsCount\_;

GLuint circlesCount = fragmentation\_ \* (baseCircles\_.size() - 1) + 1;

/\* generate indices for polygons \*/

for (GLuint i = 0; i < circlesCount - 1; ++i)

{

for (GLuint j = 0; j < circleSegmentsCount - 1; ++j)

{

/\* indices for left triangle \*/

indices\_.push\_back({

(i + 1) \* circleSegmentsCount + j + 0,

(i + 0) \* circleSegmentsCount + j + 0,

(i + 1) \* circleSegmentsCount + j + 1,

});

/\* indices for right triangle \*/

indices\_.push\_back({

(i + 1) \* circleSegmentsCount + j + 1,

(i + 0) \* circleSegmentsCount + j + 0,

(i + 0) \* circleSegmentsCount + j + 1,

});

}

/\* connect last and first vertices \*/

/\* indices for left triangle \*/

indices\_.push\_back({

(i + 1) \* circleSegmentsCount + circleSegmentsCount - 1,

(i + 0) \* circleSegmentsCount + circleSegmentsCount - 1,

(i + 1) \* circleSegmentsCount + 0,

});

/\* indices for right triangle \*/

indices\_.push\_back({

(i + 1) \* circleSegmentsCount + 0,

(i + 0) \* circleSegmentsCount + circleSegmentsCount - 1,

(i + 0) \* circleSegmentsCount + 0,

});

}

indicesCount\_ = indices\_.size() \* sizeof(TriangleIndices);

}

void GLFigure::generateNorms()

{

if (!initialized\_)

{

std::cerr << "[error] can't generate norms: not initialized" << std::endl;

return;

}

if (!vertices\_.size())

{

std::cerr << "[error] can't generate norms: vertices is empty" << std::endl;

return;

}

GLuint circleSegmentsCount = fragmentation\_ \* baseCircleSegmentsCount\_;

GLuint circlesCount = fragmentation\_ \* (baseCircles\_.size() - 1) + 1;

QVector3D norm;

QVector4D norm4;

/\* generate norms for vertices \*/

for (GLuint i = 0; i < circlesCount - 1; ++i)

{

for (GLuint j = 0; j < circleSegmentsCount - 1; ++j)

{

/\* norms has to be rotated, translated and scaled like vertices \*/

norm4 = inverseTransposedModel\_ \* QVector4D(

QVector3D::normal(

vertices\_.at((i + 0) \* circleSegmentsCount + j + 0).first, // 0 vertex position

vertices\_.at((i + 0) \* circleSegmentsCount + j + 1).first, // right vertex position

vertices\_.at((i + 1) \* circleSegmentsCount + j + 0).first // down vertex position

),

1.0f

);

norm = QVector3D{ norm4.x(), norm4.y(), norm4.z() };

/\* set all polygon vertices norms \*/

vertices\_[(i + 0) \* circleSegmentsCount + j + 0].second = norm;

vertices\_[(i + 0) \* circleSegmentsCount + j + 1].second = norm;

vertices\_[(i + 1) \* circleSegmentsCount + j + 0].second = norm;

vertices\_[(i + 1) \* circleSegmentsCount + j + 1].second = norm;

}

}

}

void GLFigure::generateCircle(

GLfloat radius,

GLfloat y

)

{

if (!initialized\_)

{

std::cerr << "[error] can't generate circle: not initialized" << std::endl;

return;

}

GLuint circleSegmentsCount = fragmentation\_ \* baseCircleSegmentsCount\_;

GLfloat angle = 0.f;

GLfloat deltaAngle = 2.0f \* 3.14159265f / circleSegmentsCount;

QVector4D vertex;

/\* generate circle segments \*/

for (GLuint i = 0; i < circleSegmentsCount; ++i)

{

/\* apply all transformations \*/

vertex = translation\_ \* rotation\_ \* scale\_ \* QVector4D{

radius \* std::cos(angle),

y,

radius \* std::sin(angle),

1.f

};

/\* save transformed vertex \*/

vertices\_.push\_back({

{

vertex.x(),

vertex.y(),

vertex.z()

},

{1.f, 1.f, 1.f} // temporary norm

});

angle += deltaAngle;

}

}

void GLFigure::setAttributeInfo(QOpenGLFunctions\* painter)

{

if (!initialized\_)

{

std::cerr << "[error] can't set attribure pointer: not initialized" << std::endl;

return;

}

if (!painter)

{

std::cerr << "[error] can't set attribure pointer: painter is nullptr" << std::endl;

return;

}

/\* set attribure info \*/

vertexObject\_.bind\_vao();

/\* set position info \*/

vertexObject\_.setupVertexAttribute(

painter, 0, 3, GL\_FLOAT, GL\_FALSE,

2 \* sizeof(QVector3D), nullptr

);

/\* set norm info \*/

vertexObject\_.setupVertexAttribute(

painter, 1, 3, GL\_FLOAT, GL\_TRUE,

2 \* sizeof(QVector3D), reinterpret\_cast<void\*>(sizeof(QVector3D))

);

}

Файл: glscene.h

#ifndef GLSCENE\_H

#define GLSCENE\_H

#include <QtOpenGLWidgets/QOpenGLWidget>

#include <QOpenGLFunctions>

#include <QMatrix4x4>

#include <memory>

#include "glshaderprogram.h"

#include "glfigure.h"

#include "gllighting.h"

#include "camera.h"

///! @brief camera projection type

enum class GLProjectionType

{

Perspective,

Orthographic

};

///! @brief OpenGL scene representation

class GLScene: public QOpenGLWidget, public QOpenGLFunctions

{

Q\_OBJECT

public:

GLScene(QWidget\* parent = nullptr);

virtual void keyPressEvent(QKeyEvent\* event) override;

virtual void mousePressEvent(QMouseEvent \*event) override;

virtual void mouseMoveEvent(QMouseEvent \*event) override;

///! @brief set camera projection type

void setProjectionType(GLProjectionType type);

inline GLShaderProgram\* getFigureShaderProgram() { return figureShaderProgram\_; }

inline GLLighting& getLighting() { return lighting\_; }

inline QVector<std::shared\_ptr<GLFigure>>& getFigures() { return figures\_; }

inline std::shared\_ptr<GLFigure> getFigure(ssize\_t index) { return figures\_.at(index); }

protected:

///! @brief initialize OpenGL window

virtual void initializeGL() override;

///! @brief resize OpenGL window

virtual void resizeGL(int w, int h) override;

///! @brief draw scene

virtual void paintGL() override;

private:

///! @brief create OpenGL shader programs for figures, axes and light source

void createShaderProgram();

///! @brief generate figures base cuts

void generateFigures();

///! @brief generate axes vertices

void generateAxes();

///! @brief convert degrees to radians

inline float d2r(float degrees) const

{

return 3.1459265 \* degrees / 180.f;

}

///! @brief scene width

GLfloat width\_;

///! @brief scene height

GLfloat height\_;

///! @brief figure shader program

GLShaderProgram\* figureShaderProgram\_ = nullptr;

///! @brief scene lighting params

GLLighting lighting\_;

///! @brief axes shader program

GLShaderProgram\* axesShaderProgram\_ = nullptr;

///! @brief axes vertex object

GLVertexObject axesVertexObject\_;

///! @brief light source shader program

GLShaderProgram\* lightSourceShaderProgram\_ = nullptr;

///! @brief light source representation

std::shared\_ptr<GLFigure> lightSource\_;

///! @brief detalization factor

GLuint fragmentationFactor\_ = 5;

///! @brief camera settings

Camera camera\_;

///! @brief camera projection type

GLProjectionType projection\_ = GLProjectionType::Perspective;

///! @brief model matrix

QMatrix4x4 modelMatrix\_;

///! @brief projection matrix

QMatrix4x4 projectionMatrix\_;

///! @brief figures representations

QVector<std::shared\_ptr<GLFigure>> figures\_;

///! @brief axes vertices vector

QVector<QPair<QVector3D, QVector3D>> axesVertices\_;

};

#endif // GLSCENE\_H

Файл: glscene.cpp

#include "glscene.h"

#include <QColor>

#include <QMouseEvent>

#include <QKeyEvent>

#include <QTimer>

#include <algorithm>

#include <iostream>

#include <iomanip>

#include <cmath>

GLScene::GLScene(QWidget\* parent):

QOpenGLWidget{ parent },

lighting\_{ GLLightingType::Point, { 1.2f, 1.0f, 0.0f }, { -1.0f, -1.0f, 0.0f } },

camera\_{ { 800.f, 600.f } }

{

/\* generate vector of 6 figures \*/

generateFigures();

width\_ = width();

height\_ = height();

modelMatrix\_.setToIdentity();

projectionMatrix\_.setToIdentity();

}

void GLScene::initializeGL()

{

/\* init opengl window \*/

QColor bgc(0x3F, 0x3F, 0x3F);

initializeOpenGLFunctions();

glClearColor(bgc.redF(), bgc.greenF(), bgc.blueF(), bgc.alphaF());

/\* create figures shader program \*/

createShaderProgram();

/\* initialize figures shader programs \*/

if (!figureShaderProgram\_->init())

{

std::cerr << "[error] Unable to initialize Figure Shader Program" << std::endl;

std::cerr << "Shader Program log: " << figureShaderProgram\_->log().toStdString() << std::endl;

return;

}

/\* initialize axes shader programs \*/

if (!axesShaderProgram\_->init())

{

std::cerr << "[error] Unable to initialize Axes Shader Program" << std::endl;

std::cerr << "Shader Program log: " << axesShaderProgram\_->log().toStdString() << std::endl;

return;

}

/\* initialize light source shader programs \*/

if (!lightSourceShaderProgram\_->init())

{

std::cerr << "[error] Unable to initialize Light Source Shader Program" << std::endl;

std::cerr << "Shader Program log: " << lightSourceShaderProgram\_->log().toStdString() << std::endl;

return;

}

/\* initialize axes vertex object \*/

axesVertexObject\_.init();

generateAxes();

/\* initialize all figures \*/

for (auto figure: figures\_)

{

figure->init(this);

}

/\* initialize light source \*/

lightSource\_->init(this);

// glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_LINE);

glEnable(GL\_DEPTH\_TEST);

}

void GLScene::resizeGL(int w, int h)

{

width\_ = w;

height\_ = h;

/\* update camera window size \*/

camera\_.resize( { width\_, height\_ } );

/\* update projection matrix \*/

setProjectionType(projection\_);

}

void GLScene::paintGL()

{

figureShaderProgram\_->bind();

glLineWidth(1.0f);

/\* set scene transformations \*/

figureShaderProgram\_->setUniformValue("model", modelMatrix\_);

figureShaderProgram\_->setUniformValue("view", camera\_.getView());

figureShaderProgram\_->setUniformValue("projection", projectionMatrix\_);

/\* set scene lighting params \*/

lighting\_.apply(figureShaderProgram\_);

figureShaderProgram\_->setUniformValue("cameraPos", camera\_.getPosition());

/\* draw figures \*/

for (auto figure: figures\_)

{

figure->draw(this, fragmentationFactor\_);

}

figureShaderProgram\_->release();

/\* draw axes \*/

axesShaderProgram\_->bind();

axesVertexObject\_.bind\_vao();

glLineWidth(3.0f);

/\* set axes rotation \*/

{

int rotationMatrixLocation = axesShaderProgram\_->uniformLocation("rotation");

axesShaderProgram\_->setUniformValue(rotationMatrixLocation, camera\_.getRotation());

}

glDrawArrays(GL\_LINES, 0, 6);

axesVertexObject\_.unbind\_vao();

axesShaderProgram\_->release();

/\* draw light source \*/

lightSourceShaderProgram\_->bind();

lightSourceShaderProgram\_->setUniformValue("model", modelMatrix\_);

lightSourceShaderProgram\_->setUniformValue("view", camera\_.getView());

lightSourceShaderProgram\_->setUniformValue("projection", projectionMatrix\_);

lightSourceShaderProgram\_->setUniformValue("cameraPos", camera\_.getPosition());

if (lighting\_.getType() != GLLightingType::Directional)

{

lightSource\_->setTranslation(

-lighting\_.getPosition().x(),

-lighting\_.getPosition().y(),

-lighting\_.getPosition().z()

);

lighting\_.apply(lightSourceShaderProgram\_);

lightSource\_->draw(this, fragmentationFactor\_);

}

lightSourceShaderProgram\_->release();

}

void GLScene::createShaderProgram()

{

QMap<QOpenGLShader::ShaderType, QString> figureShaders;

figureShaders[QOpenGLShader::Vertex] = ":/shaders/figure.vert";

figureShaders[QOpenGLShader::Fragment] = ":/shaders/figure.frag";

figureShaderProgram\_ = new GLShaderProgram(figureShaders, this);

QMap<QOpenGLShader::ShaderType, QString> axesShaders;

axesShaders[QOpenGLShader::Vertex] = ":/shaders/axes.vert";

axesShaders[QOpenGLShader::Fragment] = ":/shaders/axes.frag";

axesShaderProgram\_ = new GLShaderProgram(axesShaders, this);

QMap<QOpenGLShader::ShaderType, QString> lightSourceShaders;

lightSourceShaders[QOpenGLShader::Vertex] = ":/shaders/lightSource.vert";

lightSourceShaders[QOpenGLShader::Fragment] = ":/shaders/lightSource.frag";

lightSourceShaderProgram\_ = new GLShaderProgram(lightSourceShaders, this);

}

void GLScene::generateFigures()

{

/\* generate different scene objects \*/

GLfloat pi = 3.14159265f;

QVector<CircleBaseData> base1;

for (GLfloat y = 0.700f; y >= -0.701f; y -= 0.01)

{

base1.push\_back({

0.70f \* std::abs(

std::sin(y / 0.70f \* 0.40f \* pi)

), y

});

}

for (GLfloat y = -0.700f; y <= 0.701f; y += 0.01)

{

base1.push\_back({

0.75f \* std::sin(

std::abs(y) / 0.70f \* 0.70f \* pi

) + 0.25f, y

});

}

base1.push\_back({0.70f, 0.70f});

figures\_.push\_back(std::make\_shared<GLFigure>(base1));

figures\_.last()->setScale(0.25f, 0.25f, 0.25f);

figures\_.last()->setRotation(20.f, { 0.3f, 1.f, 0.5f });

figures\_.last()->setTranslation(-0.6f, 0.5f, -0.1f);

figures\_.last()->getMaterial().setDiffuseColor(Qt::green);

QVector<CircleBaseData> base2;

for (GLfloat y = 0.700f; y >= -0.701f; y -= 0.01)

{

base2.push\_back({

0.0001f \* std::exp(std::abs(y) / 0.70f \* 9.1f) + 0.05f,

y,

});

}

figures\_.push\_back(std::make\_shared<GLFigure>(base2));

figures\_.last()->setScale(0.3f, 0.3f, 0.3f);

figures\_.last()->setRotation(-40.f, { 1.f, 0.f, 0.2f });

figures\_.last()->setTranslation(0.25f, -0.5f, 0.1f);

figures\_.last()->getMaterial().setDiffuseColor(Qt::yellow);

QVector<CircleBaseData> base3;

for (GLfloat y = 0.700f; y >= -0.701f; y -= 0.01)

{

base3.push\_back({

0.0001f \* std::exp(y / 0.70f \* 9.1f) + 0.015f + 0.03f \* (y + 0.70f),

y,

});

}

figures\_.push\_back(std::make\_shared<GLFigure>(base3));

figures\_.last()->setScale(0.35f, 0.35f, 0.35f);

figures\_.last()->setRotation(-20.f, { 0.f, 1.f, 1.f });

figures\_.last()->setTranslation(0.1f, 0.5f, 0.3f);

figures\_.last()->getMaterial().setDiffuseColor(Qt::cyan);

QVector<CircleBaseData> base4;

for (GLfloat y = 0.450f; y >= -0.451f; y -= 0.01)

{

base4.push\_back({

0.5f - std::sqrt(0.4500000001f \* 0.4500000001f - y \* y),

y

});

}

for (GLfloat y = -0.450f; y <= 0.451f; y += 0.01)

{

base4.push\_back({

0.5f + std::sqrt(0.4500000001f \* 0.4500000001f - y \* y),

y

});

}

base4.push\_back({0.50f, 0.45f});

figures\_.push\_back(std::make\_shared<GLFigure>(base4));

figures\_.last()->setScale(0.3f, 0.3f, 0.3f);

figures\_.last()->setRotation(-50.f, { 1.f, 1.f, 0.f });

figures\_.last()->setTranslation(-0.6f, -0.3f, -0.4f);

figures\_.last()->getMaterial().setDiffuseColor(QColor{ 52, 229, 235 });

QVector<CircleBaseData> baseLightSource;

for (GLfloat y = 0.450f; y >= -0.451f; y -= 0.01)

{

baseLightSource.push\_back({

0.450f \* std::sin(std::acos(std::abs(y) / 0.450f)),

y

});

}

lightSource\_ = std::make\_shared<GLFigure>(baseLightSource);

lightSource\_->setScale(0.2f, 0.2f, 0.2f);

}

void GLScene::generateAxes()

{

/\* 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 }

});

/\* load axes data to axes buffer \*/

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::keyPressEvent(QKeyEvent\* event)

{

GLint forward = 0;

GLint right = 0;

if (event->key() == Qt::Key\_W)

{

forward += 1;

}

if (event->key() == Qt::Key\_S)

{

forward -= 1;

}

if (event->key() == Qt::Key\_D)

{

right += 1;

}

if (event->key() == Qt::Key\_A)

{

right -= 1;

}

camera\_.move(forward, right);

update();

}

void GLScene::mousePressEvent(QMouseEvent \*event)

{

camera\_.mousePress(event->position());

}

void GLScene::mouseMoveEvent(QMouseEvent \*event)

{

camera\_.rotate(event->position());

update();

}

void GLScene::setProjectionType(GLProjectionType type)

{

float aspectRatio = width\_ / height\_;

projection\_ = type;

projectionMatrix\_.setToIdentity();

// set projection matrix

switch (projection\_)

{

case GLProjectionType::Perspective:

projectionMatrix\_.perspective(45.0f, aspectRatio, 0.01f, 100.0f);

break;

case GLProjectionType::Orthographic:

projectionMatrix\_.ortho(

-1.0f \* aspectRatio,

1.0f \* aspectRatio,

-1.0f,

1.0f,

-8.f,

8.f

);

break;

default:

break;

}

}

Файл: mainwindow.h

#ifndef MAINWINDOW\_H

#define MAINWINDOW\_H

#include <QMainWindow>

#include <QtGui/qopengl.h>

#include <QMap>

QT\_BEGIN\_NAMESPACE

namespace Ui { class MainWindow; }

QT\_END\_NAMESPACE

class MainWindow: public QMainWindow

{

Q\_OBJECT

public:

MainWindow(QWidget\* parent = nullptr);

~MainWindow();

void keyPressEvent(QKeyEvent\* event);

void mousePressEvent(QMouseEvent \*event);

void mouseReleaseEvent(QMouseEvent \*);

void mouseMoveEvent(QMouseEvent \*event);

private slots:

void on\_lightTypeSelector\_currentTextChanged(const QString &arg1);

void on\_positionX\_valueChanged(double arg1);

void on\_positionY\_valueChanged(double arg1);

void on\_positionZ\_valueChanged(double arg1);

void on\_directionX\_valueChanged(double arg1);

void on\_directionY\_valueChanged(double arg1);

void on\_directionZ\_valueChanged(double arg1);

void on\_cutoffSelector\_valueChanged(double arg1);

void on\_outerCutoffSelector\_valueChanged(double arg1);

void on\_lightAmbientSelector\_clicked();

void on\_lightDiffuseSelector\_clicked();

void on\_lightSpecularSelector\_clicked();

void on\_shininessSelector\_valueChanged(double arg1);

void on\_ambientStrengthSelector\_valueChanged(double arg1);

void on\_figureSelector\_valueChanged(int arg1);

void on\_materialDiffuseSelector\_clicked();

void on\_materialSpecularSelector\_clicked();

void on\_projectionSelector\_currentTextChanged(const QString &arg1);

private:

void init();

void loadFigureMaterial();

bool drag\_ = false;

Ui::MainWindow \*ui;

};

#endif // MAINWINDOW\_H

Файл: mainwindow.cpp

#include "mainwindow.h"

#include "ui\_mainwindow.h"

#include <QMouseEvent>

#include <QColorDialog>

#include <QPainter>

#include <iostream>

#include "gllighting.h"

MainWindow::MainWindow(QWidget \*parent):

QMainWindow(parent),

ui(new Ui::MainWindow)

{

ui->setupUi(this);

init();

setFocusPolicy(Qt::StrongFocus);

}

MainWindow::~MainWindow()

{

delete ui;

}

void MainWindow::init()

{

ui->lightSettingsLayout->setAlignment(Qt::AlignTop);

ui->materialSettingsLayout->setAlignment(Qt::AlignTop);

ui->position->setAlignment(Qt::AlignTop);

ui->direction->setAlignment(Qt::AlignTop);

ui->lightColors->setAlignment(Qt::AlignTop);

ui->materialColors->setAlignment(Qt::AlignTop);

ui->materialSpecialSettingLayout->setAlignment(Qt::AlignTop);

GLLighting& lighting = ui->glWindow->getLighting();

QVector3D lightingPos = lighting.getPosition();

QVector3D lightingDir = lighting.getDirection();

// light type

ui->lightTypeSelector->addItems({

"Point Light",

"Directional Light",

"Spot Light"

});

// position

ui->positionX->setValue(lightingPos.x());

ui->positionY->setValue(lightingPos.y());

ui->positionZ->setValue(lightingPos.z());

// direction

ui->directionX->setValue(lightingDir.x());

ui->directionY->setValue(lightingDir.y());

ui->directionZ->setValue(lightingDir.z());

// cutoff

ui->cutoffSelector->setValue(lighting.getCutOff());

ui->outerCutoffSelector->setValue(lighting.getOuterCutOff());

// light colors

QPixmap pixmap(32, 32);

// ambient

pixmap.fill(lighting.getAmbientColor());

ui->lightAmbientSelector->setIcon(QIcon(pixmap));

// diffuse

pixmap.fill(lighting.getDiffuseColor());

ui->lightDiffuseSelector->setIcon(QIcon(pixmap));

// specular

pixmap.fill(lighting.getSpecularColor());

ui->lightSpecularSelector->setIcon(QIcon(pixmap));

// material

ui->figureSelector->setMaximum(ui->glWindow->getFigures().size());

loadFigureMaterial();

// projection type

ui->projectionSelector->addItems({

"Perspective",

"Orthographic"

});

}

void MainWindow::keyPressEvent(QKeyEvent\* event)

{

this->ui->glWindow->keyPressEvent(event);

}

void MainWindow::mousePressEvent(QMouseEvent \*event)

{

if (event->button() & Qt::LeftButton)

{

drag\_ = true;

this->ui->glWindow->mousePressEvent(event);

}

}

void MainWindow::mouseReleaseEvent(QMouseEvent \*)

{

drag\_ = false;

}

void MainWindow::mouseMoveEvent(QMouseEvent \*event)

{

if (drag\_)

{

this->ui->glWindow->mouseMoveEvent(event);

}

}

void MainWindow::on\_lightTypeSelector\_currentTextChanged(const QString &arg1)

{

if (arg1 == "Point Light")

{

ui->glWindow->getLighting().setType(GLLightingType::Point);

} else if (arg1 == "Directional Light")

{

ui->glWindow->getLighting().setType(GLLightingType::Directional);

} else if (arg1 == "Spot Light")

{

ui->glWindow->getLighting().setType(GLLightingType::Spot);

}

ui->glWindow->update();

}

void MainWindow::on\_positionX\_valueChanged(double)

{

ui->glWindow->getLighting().setPosition({

static\_cast<float>(ui->positionX->value()),

static\_cast<float>(ui->positionY->value()),

static\_cast<float>(ui->positionZ->value())

});

ui->glWindow->update();

}

void MainWindow::on\_positionY\_valueChanged(double)

{

ui->glWindow->getLighting().setPosition({

static\_cast<float>(ui->positionX->value()),

static\_cast<float>(ui->positionY->value()),

static\_cast<float>(ui->positionZ->value())

});

ui->glWindow->update();

}

void MainWindow::on\_positionZ\_valueChanged(double)

{

ui->glWindow->getLighting().setPosition({

static\_cast<float>(ui->positionX->value()),

static\_cast<float>(ui->positionY->value()),

static\_cast<float>(ui->positionZ->value())

});

ui->glWindow->update();

}

void MainWindow::on\_directionX\_valueChanged(double)

{

ui->glWindow->getLighting().setDirection({

static\_cast<float>(ui->directionX->value()),

static\_cast<float>(ui->directionY->value()),

static\_cast<float>(ui->directionZ->value())

});

ui->glWindow->update();

}

void MainWindow::on\_directionY\_valueChanged(double)

{

ui->glWindow->getLighting().setDirection({

static\_cast<float>(ui->directionX->value()),

static\_cast<float>(ui->directionY->value()),

static\_cast<float>(ui->directionZ->value())

});

ui->glWindow->update();

}

void MainWindow::on\_directionZ\_valueChanged(double)

{

ui->glWindow->getLighting().setDirection({

static\_cast<float>(ui->directionX->value()),

static\_cast<float>(ui->directionY->value()),

static\_cast<float>(ui->directionZ->value())

});

ui->glWindow->update();

}

void MainWindow::on\_cutoffSelector\_valueChanged(double)

{

ui->glWindow->getLighting().setCutOff(static\_cast<float>(ui->cutoffSelector->value()));

ui->glWindow->update();

}

void MainWindow::on\_outerCutoffSelector\_valueChanged(double)

{

ui->glWindow->getLighting().setOuterCutOff(static\_cast<float>(ui->outerCutoffSelector->value()));

ui->glWindow->update();

}

void MainWindow::on\_lightAmbientSelector\_clicked()

{

GLLighting& lighting = ui->glWindow->getLighting();

QColor peek = QColorDialog::getColor(lighting.getAmbientColor(), nullptr, "Choose Light Ambient Color", QColorDialog::ShowAlphaChannel | QColorDialog::NoButtons);

if (peek.isValid())

{

lighting.setAmbientColor(peek);

QPixmap pixmap(32, 32);

pixmap.fill(peek);

ui->lightAmbientSelector->setIcon(QIcon(pixmap));

}

ui->glWindow->update();

}

void MainWindow::on\_lightDiffuseSelector\_clicked()

{

GLLighting& lighting = ui->glWindow->getLighting();

QColor peek = QColorDialog::getColor(lighting.getDiffuseColor(), nullptr, "Choose Light Diffuse Color", QColorDialog::ShowAlphaChannel | QColorDialog::NoButtons);

if (peek.isValid())

{

lighting.setDiffuseColor(peek);

QPixmap pixmap(32, 32);

pixmap.fill(peek);

ui->lightDiffuseSelector->setIcon(QIcon(pixmap));

}

ui->glWindow->update();

}

void MainWindow::on\_lightSpecularSelector\_clicked()

{

GLLighting& lighting = ui->glWindow->getLighting();

QColor peek = QColorDialog::getColor(lighting.getSpecularColor(), nullptr, "Choose Light Specular Color", QColorDialog::ShowAlphaChannel | QColorDialog::NoButtons);

if (peek.isValid())

{

lighting.setSpecularColor(peek);

QPixmap pixmap(32, 32);

pixmap.fill(peek);

ui->lightSpecularSelector->setIcon(QIcon(pixmap));

}

ui->glWindow->update();

}

void MainWindow::on\_shininessSelector\_valueChanged(double arg1)

{

std::shared\_ptr<GLFigure> figure = ui->glWindow->getFigure(ui->figureSelector->value() - 1);

figure->getMaterial().setShininess(static\_cast<float>(arg1));

ui->glWindow->update();

}

void MainWindow::on\_ambientStrengthSelector\_valueChanged(double arg1)

{

std::shared\_ptr<GLFigure> figure = ui->glWindow->getFigure(ui->figureSelector->value() - 1);

figure->getMaterial().setAmbientStrength(static\_cast<float>(arg1));

ui->glWindow->update();

}

void MainWindow::on\_figureSelector\_valueChanged(int)

{

loadFigureMaterial();

ui->glWindow->update();

}

void MainWindow::on\_materialDiffuseSelector\_clicked()

{

std::shared\_ptr<GLFigure> figure = ui->glWindow->getFigure(ui->figureSelector->value() - 1);

QColor peek = QColorDialog::getColor(figure->getMaterial().getDiffuseColor(), nullptr, "Choose Material Diffuse Color", QColorDialog::ShowAlphaChannel | QColorDialog::NoButtons);

if (peek.isValid())

{

figure->getMaterial().setDiffuseColor(peek);

QPixmap pixmap(32, 32);

pixmap.fill(peek);

ui->materialDiffuseSelector->setIcon(QIcon(pixmap));

}

ui->glWindow->update();

}

void MainWindow::on\_materialSpecularSelector\_clicked()

{

std::shared\_ptr<GLFigure> figure = ui->glWindow->getFigure(ui->figureSelector->value() - 1);

QColor peek = QColorDialog::getColor(figure->getMaterial().getSpecularColor(), nullptr, "Choose Material Specular Color", QColorDialog::ShowAlphaChannel | QColorDialog::NoButtons);

if (peek.isValid())

{

figure->getMaterial().setSpecularColor(peek);

QPixmap pixmap(32, 32);

pixmap.fill(peek);

ui->materialSpecularSelector->setIcon(QIcon(pixmap));

}

ui->glWindow->update();

}

void MainWindow::on\_projectionSelector\_currentTextChanged(const QString &arg1)

{

if (arg1 == "Perspective")

{

ui->glWindow->setProjectionType(GLProjectionType::Perspective);

} else if (arg1 == "Orthographic")

{

ui->glWindow->setProjectionType(GLProjectionType::Orthographic);

}

ui->glWindow->update();

}

void MainWindow::loadFigureMaterial()

{

std::shared\_ptr<GLFigure> figure = ui->glWindow->getFigure(ui->figureSelector->value() - 1);

// colors

QPixmap pixmap(32, 32);

// diffuse

pixmap.fill(figure->getMaterial().getDiffuseColor());

ui->materialDiffuseSelector->setIcon(QIcon(pixmap));

// specular

pixmap.fill(figure->getMaterial().getSpecularColor());

ui->materialSpecularSelector->setIcon(QIcon(pixmap));

// shininess

ui->shininessSelector->setValue(figure->getMaterial().getShininess());

// ambient strength

ui->ambientStrengthSelector->setValue(figure->getMaterial().getAmbientStrength());

}

Файл: figure.vert

#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));

}

Файл: figure.frag

#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) / epsilon);

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);

}

Файл: lightSource.vert

#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));

}

Файл: lightSource.frag

#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);

}

Файл: axes.vert

#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;

}

Файл: axes.frag

#version 460 core

in vec3 color;

out vec4 FragColor;

void main()

{

FragColor = vec4(

color,

1.0

);

}