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

ОТЧЕТ

по лабораторным работам №5
по дисциплине «Компьютерная графика»
Тема: Расширения OpenGL, программируемый графический конвейер. Шейдеры.

Студентка гр. 1304 _____ Чернякова В.А.

Студентка гр. 1304 _____ Ярусова Т.В.

Преподаватель _____ Герасимова Т.В.

Санкт-Петербург
2024

Цель работы.

- Изучение языка шейдеров GLSL;
- Изучение способов применения шейдеров в программах OpenGL.

Задание.

Задание 2:

Разработать визуальный эффект по заданию, реализованный средствами языка шейдеров GLSL.

Превратить кривую в поверхность на сцене и добавив в программу дополнительный визуальный эффект, реализованный средствами языка шейдеров.

Вариант Эффекта: 22

Зеркальное освещение от источника света в заданной позиции.

Теоретические положения.

Язык программирования высокоуровневых расширений называются языком затенения OpenGL (OpenGL Shading Language –GLSL), иногда именуемым языком шедеров OpenGL (OpenGL Shader Language). Этот язык очень похож на язык С но имеет встроенные типы данных и функции полезные в шедерах вершин и фрагментов.

Шейдер является фрагментом шейдерной программы, которая заменяет собой часть графического конвейера видеокарты. Тип шейдера зависит от того, какая часть конвейера будет заменена. Каждый шейдер должен выполнить свою обязательную работу, т. е. записать какие-то данные и передать их дальше по графическому конвейеру.

Шейдерная программа – это небольшая программа, состоящая из шейдеров (вершинного и фрагментного, возможны и др.) и выполняющаяся на GPU (Graphics Processing Unit), т. е. на графическом процессоре видеокарты.

Существует пять мест в графическом конвейере, куда могут быть встроены шейдеры. Соответственно шейдеры делятся на типы:

– вершинный шейдер (vertex shader);

– геометрический шейдер (geometric shader);

– фрагментный шейдер (fragment shader);

– два тесселяционных шейдера (tesselation), отвечающие за два разных этапа тесселяции (они доступны в OpenGL 4.0 и выше).

Дополнительно существуют вычислительные (compute) шейдеры, которые выполняются независимо от графического конвейера.

Выполнение работы.

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

Camera.cpp

Это исходный файл, содержащий реализацию класса Camera. Этот класс предназначен для управления функциональностью камеры в приложении для трехмерной графики.

Этот конструктор устанавливает начальные значения для позиции камеры (cameraPos), направления взгляда (cameraFront), вектора "вверх" (cameraUp), размеров экрана (height и width), угла обзора (fov), а также начальные углы поворота (pitch и yaw).

```
Camera::Camera()
{
    cameraPos = { 0.0f, 0.0f, 3.0f };
    cameraFront = { 0.0f, 0.0f, -1.0f };
    cameraUp = { 0.0f, 1.0f, 0.0f };

    height = 640;
    width = 480;

    fov = 45.0f;
    pitch = 0.0f;
    yaw = -90.0f;
}
```

Метод getView() возвращает матрицу вида (view), которая используется для преобразования мировых координат в координаты камеры. В коде вызывается метод lookAt() объекта view, который генерирует матрицу вида на основе трех векторов: позиции камеры (cameraPos), точки, на которую направлена камера (cameraPos + cameraFront), и направления "вверх"

(cameraUp). Метод lookAt() вычисляет матрицу вида таким образом, чтобы камера находилась в позиции cameraPos, направлялась на точку cameraPos + cameraFront и имела направление "верха" cameraUp. Таким образом, метод getView() создает и возвращает матрицу вида, которая представляет положение и ориентацию камеры в трехмерном пространстве.

```
QMatrix4x4 Camera::getView()
{
    QMatrix4x4 view;
    view.lookAt(cameraPos, cameraPos + cameraFront, cameraUp);
    return view;
}
```

Метод getProjection() возвращает матрицу проекции (projection), которая используется для преобразования трехмерных координат в двумерные координаты экрана. Он использует функцию perspective() для создания перспективной проекции на основе угла обзора, соотношения сторон экрана и параметров для удаления ближних и дальних плоскостей отсечения.

```
QMatrix4x4 Camera::getProjection()
{
    QMatrix4x4 projection;
    float aspect = width / (height ? (float)height : 1.0f);
    projection.perspective(fov, aspect, 0.1f, 50.0f);
    return projection;
}
```

Метод getPos() возвращает текущую позицию камеры.

```
QVector3D Camera::getPos()
{
    return cameraPos;
}
```

Метод setViewport() устанавливает размеры экрана для камеры.

```
void Camera::setViewport(int w, int h)
{
    width = w;
    height = h;
}
```

Метод move() перемещает камеру в указанном направлении с учетом скорости и текущего направления взгляда.

```
void Camera::move(MoveDirection direction)
{
    float cameraSpeed = 0.5f;

    QVector3D cameraRight = QVector3D::crossProduct(cameraFront, cameraUp).normalized();

    if (direction == MoveDirection::Forward) {
        cameraPos += cameraSpeed * cameraFront;
    }
    if (direction == MoveDirection::Back) {
        cameraPos -= cameraSpeed * cameraFront;
    }
    if (direction == MoveDirection::Left) {
        cameraPos -= cameraSpeed * cameraRight;
    }
    if (direction == MoveDirection::Right) {
        cameraPos += cameraSpeed * cameraRight;
    }
    if (direction == MoveDirection::Up) {
        cameraPos += cameraSpeed * cameraUp;
    }
    if (direction == MoveDirection::Down) {
        cameraPos -= cameraSpeed * cameraUp;
    }
}
```

Метод rotate() вращает камеру на указанные углы. При этом обрабатываются ограничения на углы поворота камеры, чтобы предотвратить перевороты. Затем вычисляется новое направление взгляда камеры на основе углов поворота yaw и pitch.

```
void Camera::rotate(float deltaYaw, float deltaPitch)
{
    yaw += deltaYaw;
    pitch += deltaPitch;

    if (pitch > 89.0f) {
        pitch = 89.0f;
    }
    if (pitch < -89.0f) {
        pitch = -89.0f;
    }

    QVector3D front;
    front.setX(qCos(qDegreesToRadians(yaw)) * qCos(qDegreesToRadians(pitch)));
    front.setY(qSin(qDegreesToRadians(pitch)));
    front.setZ(qSin(qDegreesToRadians(yaw)) * qCos(qDegreesToRadians(pitch)));

    cameraFront = front.normalized();
}
```

Myopengl.cpp

Этот код реализует класс MyOpenGL, который наследуется от QOpenGLWidget и предназначен для отрисовки графики с использованием OpenGL.

Конструктор класса MyOpenGL инициализирует объект класса QOpenGLWidget с указанным родительским виджетом. Затем инициализируются переменные lightPos, countVertices, и controlPoints.

```
MyOpenGL::MyOpenGL(QWidget* parent) :
    QOpenGLWidget(parent), vertexBuffer(QOpenGLBuffer::VertexBuffer
{
    lightPos = QVector3D(0.0f, 0.0f, 0.0f);
    countVertices = 0;
    controlPoints = {
        QVector2D(-0.75f, -0.45f),
        QVector2D(0.0f, -0.85f),
        QVector2D(0.75f, -0.3f)
    };
}
```

Метод initializeGL() вызывается при создании контекста OpenGL. В нем инициализируются функции OpenGL, включается тест глубины, инициализируются шейдеры, инициализируется объект, а также производится некоторые операции с камерой.

```
void MyOpenGL::initializeGL()
{
    initializeOpenGLFunctions();
    glEnable(GL_DEPTH_TEST);
    initShaders();
    initObject();
    camera.rotate(30.0f, 0.0f);
    camera.move(Camera::MoveDirection::Left);
    camera.move(Camera::MoveDirection::Left);
    camera.move(Camera::MoveDirection::Left);
}
```

Метод resizeGL() вызывается при изменении размеров окна. В нем устанавливается область просмотра (glViewport) и обновляются параметры камеры.

```
void MyOpenGL::resizeGL(int w, int h)
{
    glViewport(0, 0, w, h);
    camera.setViewport(w, h);
}
```

Метод paintGL() вызывается при отрисовке сцены. Он очищает буфер цвета и буфер глубины, а затем рисует объект сцены.

```
void MyOpenGL::paintGL()
{
    glClearColor(0, 0, 0, 0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    drawObject();
}
```

Метод initShaders() инициализирует шейдерную программу из файлов вершинного и фрагментного шейдеров.

```
void MyOpenGL::initShaders()
{
    if (!shaderProgram.addShaderFromSourceFile(QOpenGLShader::Vertex, ":/vshader.vsh")
        || !shaderProgram.addShaderFromSourceFile(QOpenGLShader::Fragment, ":/fshader.fsh")
        || !shaderProgram.link()) {
        close();
    }
}
```

Метод initObject() инициализирует объект для отрисовки. Создаются вершины, определяющие объект. В блоке кода заполняется вектор vertices вершинами объекта. Цикл проходит от 0 до 1 с шагом 0.01, вычисляя точки на кривой с помощью функции curve(t). Для каждой точки создаются две вершины, одна с координатой Z равной -0.5 (нижняя точка объекта), а другая с координатой Z равной 0.5 (верхняя точка объекта). Каждая вершина имеет цвет (1.0f, 1.0f, 0.0f) и нормаль (0.0f, 0.0f, 0.0f). Также создаются и связываются объект массива вершин (VAO). VAO представляет собой контейнер, который хранит настройки состояния OpenGL, такие как указатели на данные вершин и настройки атрибутов. Создается буфер вершин, связывают его с текущим контекстом OpenGL и выделяют память под данные вершин. Данные вершин передаются в буфер с использованием метода allocate(), который принимает указатель на начало данных и их размер. Настраиваются атрибуты вершин. Каждый вызов glVertexAttribPointer устанавливает указатель на данные для атрибута вершин (например, позицию, цвет, нормаль) и их формат. glEnableVertexAttribArray активирует использование атрибута в шейдерной программе.

```
void MyOpenGL::initObject()
{
    QVector<VertexData> vertices;

    for (float t = 0; t <= 1; t += 0.01) {    △Variable 't' with floating point type 'float' sh
        QVector2D p = curve(t);
        vertices.append(VertexData(QVector3D(p.x(), p.y(), -0.5), QVector3D(1.0f, 1.0f, 0.0f), QVector3D(6
        vertices.append(VertexData(QVector3D(p.x(), p.y(), 0.5), QVector3D(1.0f, 1.0f, 0.0f), QVector3D(0.
    }

    countVertices = vertices.size();
    calculateNormals(vertices);

    objectVao.create();
    QOpenGLVertexArrayObject::Binder vaoBinder(&objectVao);

    vertexBuffer.create();
    vertexBuffer.bind();
    vertexBuffer.setUsagePattern(QOpenGLBuffer::StaticDraw);
    vertexBuffer.allocate(vertices.constData(), vertices.size() * sizeof(VertexData));

    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 9 * sizeof(GLfloat), (GLvoid*) nullptr);
    glEnableVertexAttribArray(0);

    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 9 * sizeof(GLfloat), (GLvoid*)(3 * sizeof(GLfloat)));
    glEnableVertexAttribArray(1);

    glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, 9 * sizeof(GLfloat), (GLvoid*)(6 * sizeof(GLfloat)));
    glEnableVertexAttribArray(2);

    vertexBuffer.release();
}
```

Метод drawObject() отрисовывает объект сцены. Он связывает шейдерную программу, устанавливает uniform-переменные, передает данные вершин, а затем вызывает отрисовку. Далее более подробно. В строках кода создаются и инициализируются матрицы viewMatrix, projectionMatrix и modelMatrix. viewMatrix и projectionMatrix получаются из камеры с помощью методов getView() и getProjection(). modelMatrix устанавливается в единичную матрицу, что представляет собой начальное положение модели. shaderProgram.bind() - Этот вызов связывает шейдерную программу с текущим контекстом OpenGL. Все последующие операции с шейдерами будут применяться к этой программе. Следующие строки устанавливают значения uniform-переменных в шейдерной программе. Программа ожидает получить матрицы projectionMatrix, viewMatrix и modelMatrix, а также позицию источника света lightPos и позицию камеры viewPos. QOpenGLVertexArrayObject::Binder vaoBinder(&objectVao) – этот объект создает и связывает VAO с текущим контекстом OpenGL. VAO хранит настройки состояния OpenGL для отрисовки объекта. glDrawArrays(GL_TRIANGLE_STRIP, 0, countVertices) - Эта строка выполняет команду рисования. Она рисует геометрию, хранящуюся в VAO, используя определенный режим примитива (в данном случае - треугольные полосы) и количество вершин (countVertices).

```
void MyOpenGL::drawObject()
{
    QMatrix4x4 viewMatrix = camera.getView();
    QMatrix4x4 projectionMatrix = camera.getProjection();
    QMatrix4x4 modelMatrix;
    modelMatrix.setToIdentity();
    QVector3D viewPos = camera.getPos();

    shaderProgram.bind();
    shaderProgram.setUniformValue("projectionMatrix", projectionMatrix);
    shaderProgram.setUniformValue("viewMatrix", viewMatrix);
    shaderProgram.setUniformValue("modelMatrix", modelMatrix);
    shaderProgram.setUniformValue("lightPos", lightPos);
    shaderProgram.setUniformValue("viewPos", viewPos);

    QOpenGLVertexArrayObject::Binder vaoBinder(&objectVao);
    glDrawArrays(GL_TRIANGLE_STRIP, 0, countVertices);

    shaderProgram.release();
}
```

calculateNormals. Этот метод предназначен для вычисления нормалей к вершинам объекта. Этот метод важен для правильной отрисовки объектов с использованием освещения. Нормали к вершинам определяют ориентацию поверхности объекта и влияют на то, как свет будет отражаться от этой поверхности. Вычисление нормалей к вершинам объекта является важной частью процесса отрисовки объектов, особенно при использовании освещения. Нормали определяют ориентацию поверхности объекта в каждой его точке и используются для расчета освещения, что позволяет создавать реалистичные эффекты освещения и теней. При применении освещения к объекту, например, при использовании моделей освещения Фонга или Ламберта, нормали используются для расчета интенсивности света, отражаемого от поверхности в каждой точке. Правильно вычисленные нормали позволяют создавать реалистичные эффекты освещения, включая отражение света и формирование теней.

```
void MyOpenGL::calculateNormals(QVector<VertexData> &vertices)
{
    for (int i = 0; i < vertices.size() - 2; i++) {
        QVector3D v1 = vertices[i + 1].position - vertices[i].position;
        QVector3D v2 = vertices[i + 2].position - vertices[i + 1].position;
        QVector3D normal = QVector3D::crossProduct(v1, v2);
        normal.normalize();

        if (normal.y() < 0) {
            normal.setY(-normal.y());
        }

        normal.normalize();

        vertices[i].normal += normal;
        vertices[i + 1].normal += normal;
    }

    for (int i = 0; i < vertices.size(); ++i) {
        vertices[i].normal.normalize();
    }
}
```

Шейдеры

Шейдеры в OpenGL представляют собой программы, которые используются для выполнения различных вычислений на графическом процессоре (GPU) во время процесса рендеринга.

Шейдеры в OpenGL пишутся на специальном языке программирования, который называется OpenGL Shading Language (GLSL). GLSL предоставляет широкий набор инструментов для создания графических эффектов, включая возможность выполнения вычислений на графическом процессоре.

Vshader.vsh

Задача вершинного шейдера - генерировать координаты пространства отсечения. Они применяются к каждой вершине объекта и используются для преобразования координат вершин из локального пространства объекта в экранное пространство, а также для передачи дополнительных данных для последующего использования в других шейдерах.

#version 330 core: Эта директива определяет версию языка GLSL. В данном случае используется версия 3.30.

layout(location = 0) in vec3 objectPosition;: Эта строка определяет атрибут вершины objectPosition, который представляет собой позицию вершины в локальном пространстве объекта. location = 0 указывает на местоположение этого атрибута во входных данных.

layout(location = 1) in vec3 objectColor;: Эта строка определяет атрибут вершины objectColor, который представляет собой цвет вершины. Он также имеет location = 1.

layout(location = 2) in vec3 normal;: Эта строка определяет атрибут вершины normal, который представляет собой нормаль вершины (вектор, перпендикулярный поверхности в данной точке). Он также имеет location = 2.

uniform mat4 projectionMatrix;, uniform mat4 viewMatrix;, uniform mat4 modelMatrix;: Эти строки определяют uniform-переменные, которые будут использоваться для передачи матриц преобразования в вершинный

шейдер. projectionMatrix представляет матрицу проекции, viewMatrix - матрицу вида (камеры), а modelMatrix - матрицу модели объекта.

out vec3 Color;, out vec3 FragPos;, out vec3 Normal;: Эти строки определяют выходные переменные (out) шейдера. Они будут передавать данные из вершинного шейдера во фрагментный шейдер для последующего использования.

vec4 objectPos = vec4(objectPosition, 1.0);: Эта строка создает четырехмерный вектор objectPos, представляющий позицию вершины в формате (x, y, z, w), где w установлен в 1.

mat4 modelViewMatrix = viewMatrix * modelMatrix;: Эта строка вычисляет матрицу модель-вида, перемножая матрицы вида и модели. Эта матрица будет использоваться для преобразования координат вершин в пространство камеры.

gl_Position = projectionMatrix * modelViewMatrix * objectPos;: Эта строка вычисляет финальную позицию вершины в экранных координатах, умножая позицию вершины на матрицы проекции и модель-вида.

Color = objectColor;, FragPos = vec3(modelMatrix * objectPos);, Normal = mat3(transpose(inverse(modelMatrix))) * normal;: Эти строки присваивают значения выходным переменным шейдера. Color получает цвет вершины, FragPos получает позицию вершины в пространстве модели, а Normal получает нормаль вершины после преобразования матрицей модели.

```
#version 330 core

layout(location = 0) in vec3 objectPosition;
layout(location = 1) in vec3 objectColor;
layout(location = 2) in vec3 normal;

uniform mat4 projectionMatrix;
uniform mat4 viewMatrix;
uniform mat4 modelMatrix;

out vec3 Color;
out vec3 FragPos;
out vec3 Normal;

void main()
{
    vec4 objectPos = vec4(objectPosition, 1.0);
    mat4 modelViewMatrix = viewMatrix * modelMatrix;
    gl_Position = projectionMatrix * modelViewMatrix * objectPos;
    Color = objectColor;
    FragPos = vec3(modelMatrix * objectPos);
    Normal = mat3(transpose(inverse(modelMatrix))) * normal;
}
```

Fshader.fsh

Задача фрагментного шейдера - устанавливать цвет для текущего пикселя при растеризации. Это программы, которые выполняются на графическом процессоре (GPU) для каждого фрагмента (точки) на экране в процессе рендеринга. Они отвечают за вычисление цвета каждого фрагмента на основе различных параметров, таких как освещение, текстуры, цвета и так далее.

#version 330 core: Эта директива определяет версию языка GLSL. В данном случае используется версия 3.30.

in vec3 Color;, in vec3 FragPos;, in vec3 Normal;: Эти строки определяют входные переменные (in) шейдера. Они получают данные, переданные из вершинного шейдера, в данном случае цвет вершины (Color), позицию вершины (FragPos) и нормаль вершины (Normal).

uniform vec3 lightPos;, uniform vec3 viewPos;: Эти строки определяют uniform-переменные, которые будут использоваться для передачи позиции источника света (lightPos) и позиции камеры (viewPos) в фрагментный шейдер.

out vec4 color;: Эта строка определяет выходную переменную (out) шейдера, которая будет передавать итоговый цвет фрагмента.

vec3 lightColor = vec3(1.0f, 1.0f, 1.0f);: Эта строка определяет цвет источника света.

float ambientStrength = 0.1f;, vec3 ambient = ambientStrength * lightColor;: Эти строки вычисляют составляющую рассеянного освещения (амбиентного освещения), которая равна некоторой доле от цвета источника света.

vec3 norm = normalize(Normal);, vec3 lightDir = normalize(lightPos - FragPos);: Эти строки вычисляют нормализованные векторы направления нормали и направления к источнику света относительно текущего фрагмента.

float diff = max(dot(norm, lightDir), 0.0);, vec3 diffuse = diff * lightColor;: Эти строки вычисляют составляющую диффузного освещения, которая зависит от угла между нормалью поверхности и направлением света.

float specularStrength = 0.5f;, vec3 viewDir = normalize(viewPos - FragPos);, vec3 reflectDir = reflect(-lightDir, norm);, float spec = pow(max(dot(viewDir, reflectDir), 0.0), 64);, vec3 specular = specularStrength * spec * lightColor;: Эти строки вычисляют составляющую зеркального освещения, которая зависит от угла между направлением камеры и отраженным от поверхности светом.

vec3 result = (ambient + diffuse + specular) * Color;: Эта строка суммирует все составляющие освещения и умножает их на цвет фрагмента.

color = vec4(result, 1.0f);: Эта строка присваивает итоговый цвет фрагмента выходной переменной color.

```
#version 330 core

in vec3 Color;
in vec3 FragPos;
in vec3 Normal;

uniform vec3 lightPos;
uniform vec3 viewPos;

out vec4 color;

void main()
{
    vec3 lightColor = vec3(1.0f, 1.0f, 1.0f);

    float ambientStrength = 0.1f;
    vec3 ambient = ambientStrength * lightColor;

    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(lightPos - FragPos);
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = diff * lightColor;

    float specularStrength = 0.5f;
    vec3 viewDir = normalize(viewPos - FragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), 64);
    vec3 specular = specularStrength * spec * lightColor;

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

Uniform переменные в шейдерах OpenGL представляют собой глобальные переменные, значения которых задаются на стороне приложения (на центральном процессоре) и затем передаются в шейдеры. Они не могут быть изменены внутри шейдера и остаются неизменными для всех выполняющихся фрагментов или вершин.

Тестирование.

Рисунок 1.

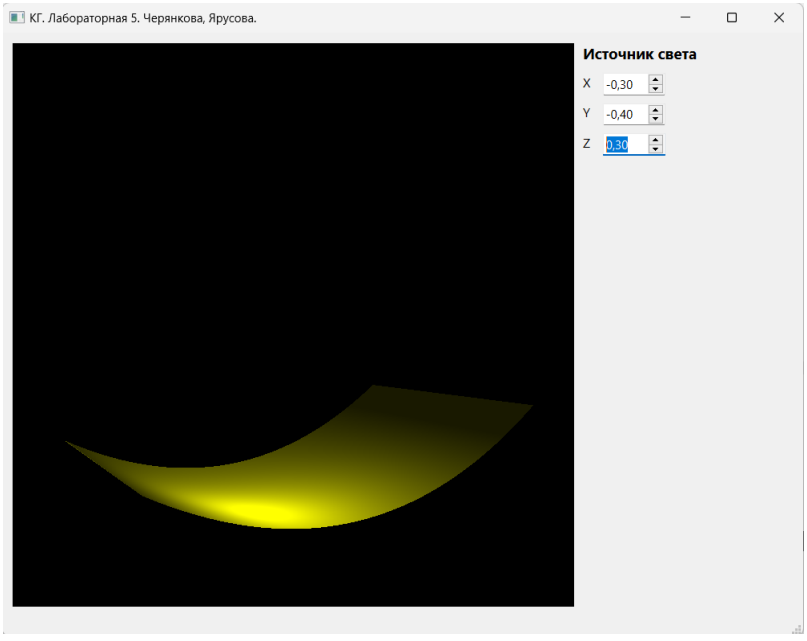

Рисунок 2.

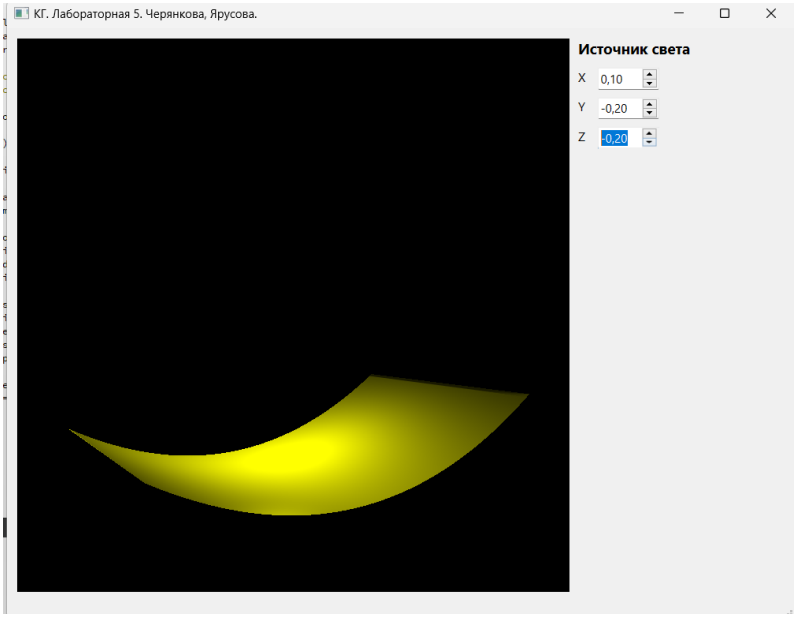

Рисунок 3.

Выводы.

В ходе лабораторной работы были изучены принципы работы с языком шейдеров в OpenGL – GLSL. Была создана шейдерная программа, применяющая зеркальное освещение от источника света.