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 C 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.

Для получения матрицы просмотра, проекции, текущей рассматриваемой позиции и дальнейшего использования их в шейдерах был создан класс Camera.

Конструктор Camera::Camera(): Инициализирует начальные значения для позиции камеры (cameraPos), направления взгляда (cameraFront), вектора вверх (cameraUp), а также размеры видимой области (height и width), угол обзора (fov), а также углы камеры (pitch и yaw).

Метод Camera::getView(): Создает и возвращает матрицу просмотра (view), используя текущие значения позиции камеры, направления взгляда и вектора вверх. Эта матрица позволяет преобразовать координаты сцены в систему координат камеры.

Метод Camera::getProjection(): Создает и возвращает матрицу проекции (projection), используя текущие значения размеров видимой области, угла обзора и параметры перспективы. Эта матрица отвечает за перспективное искажение и проецирование объектов на двумерный экран.

Метод Camera::getPos(): Возвращает текущую позицию камеры (cameraPos).

Метод Camera::setViewport(int w, int h): Устанавливает новые размеры видимой области.

Метод Camera::move(MoveDirection direction): Перемещает камеру в заданном направлении (Forward, Back, Left, Right, Up, Down). Это происходит путем изменения cameraPos в соответствии с текущим направлением камеры и скоростью.

Метод Camera::rotate(float deltaYaw, float deltaPitch): Поворачивает камеру на заданные углы deltaYaw и deltaPitch. Это изменяет значения yaw и pitch, а затем пересчитывает cameraFront на основе этих углов.

```
#include "camera.h"

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;
}

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

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

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

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

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;
    }
}

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();
}
```

Далее был реализован класс MyOpenGL, который является подклассом QOpenGLWidget и предназначен для отрисовки трехмерной сцены с помощью OpenGL в графическом интерфейсе Qt.

Конструктор MyOpenGL::MyOpenGL(QWidget parent) и деструктор MyOpenGL::~MyOpenGL():* В конструкторе инициализируются начальные значения для позиции света (lightPos), количество вершин (countVertices) и

набор контрольных точек (controlPoints) для кривой. В деструкторе происходит очистка буферов, если они были созданы.

Метод MyOpenGL::initializeGL(): Инициализирует OpenGL контекст и параметры. Включает тест глубины (GL_DEPTH_TEST), инициализирует шейдеры (initShaders()), и объект сцены (initObject()), а затем поворачивает камеру и смещает ее влево.

Метод MyOpenGL::resizeGL(int w, int h): Устанавливает область вывода OpenGL (glViewport) и обновляет размеры область видимости камеры.

Метод MyOpenGL::paintGL(): Очищает буферы кадра, устанавливает цвет очистки, рисует объект сцены вызывая drawObject().

Метод MyOpenGL::initShaders(): Загружает и компилирует вершинный и фрагментный шейдеры из файлов и привязывает их к программе шейдеров.

Метод MyOpenGL::initObject(): Вначале создается контейнер vertices для хранения вершин объекта. Затем происходит генерация вершин объекта на основе заданной кривой. Далее количество вершин записывается в переменную countVertices. После этого вызывается функция calculateNormals(vertices), которая вычисляет нормали для вершин объекта. Далее создается объект буфера вершин vertexBuffer, который используется для хранения вершин в видеопамяти видеокарты. Вершины из контейнера vertices копируются в буфер вершин с использованием функции allocate. Затем настраиваются указатели атрибутов вершин (позиция, цвет, нормаль) с помощью функции glVertexAttribPointer, а затем они активируются с помощью функции glEnableVertexAttribArray. Наконец, буфер вершин освобождается, чтобы предотвратить его дальнейшее изменение.

Метод MyOpenGL::drawObject(): В данном коде создаются матрицы viewMatrix, projectionMatrix и modelMatrix, которые представляют собой матрицы вида, проекции и модели соответственно. Далее получаем позицию камеры с помощью метода getPos() объекта camera. После этого выполняется связывание шейдерной программы с помощью shaderProgram.bind(). Затем устанавливаются uniform-переменные для шейдера, такие как projectionMatrix,

viewMatrix, modelMatrix, lightPos и viewPos, с помощью метода setUniformValue(). Эти uniform-переменные позволяют передавать данные из программы OpenGL в шейдерную программу для использования в рендеринге. После настройки uniform-переменных происходит связывание VAO (массива вершин) с помощью QOpenGLVertexArrayObject::Binder vaoBinder(&objectVao). Затем вызывается функция glDrawArrays(), которая выполняет рендеринг примитивов по вершинам из VAO. Наконец, после завершения рисования объектов шейдерная программа отвязывается с помощью вызова shaderProgram.release().

Метод MyOpenGL::curve(float t): Вычисляет точки кривой для заданного параметра t.

Метод MyOpenGL::calculateNormals(QVector<VertexData> &vertices): Вычисляет нормали для вершин объекта на основе их позиций. Вычисление нормалей для шейдеров играет важную роль в трехмерной графике и особенно при использовании моделей с освещением. Нормали представляют собой векторы, перпендикулярные к поверхности объекта, которые используются для определения направления освещения и расчета интенсивности света на каждой точке поверхности. Нормали играют ключевую роль в определении того, как свет будет отражаться от поверхности объекта. Они используются для расчета угла между направлением света и направлением нормали, что позволяет определить, насколько ярко должна быть данная область поверхности освещена.

Метод MyOpenGL::changeLightPosition(QVector3D pos): Изменяет позицию света и обновляет сцену.

```
#include "myopengl.h"

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)
    };
}
```

```
MyOpenGL::~MyOpenGL()
{
    if (objectVao.isCreated()) {
        objectVao.destroy();
    }
    if (vertexBuffer.isCreated()) {
        vertexBuffer.destroy();
    }
}

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);
}

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

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

void MyOpenGL::initShaders()
{
    if     (!shaderProgram.addShaderFromSourceFile(QOpenGLShader::Vertex,
":/vshader.vsh")

||        !shaderProgram.addShaderFromSourceFile(QOpenGLShader::Fragment,
":/fshader.fsh")
        || !shaderProgram.link()) {
        close();
    }
}

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

    for (float t = 0; t <= 1; t += 0.01) {
        QVector2D p = curve(t);
        vertices.append(VertexData(QVector3D(p.x(),     p.y(),     -0.5),
QVector3D(1.0f, 1.0f, 0.0f), QVector3D(0.0f, 0.0f, 0.0f)));
        vertices.append(VertexData(QVector3D(p.x(),     p.y(),     0.5),
QVector3D(1.0f, 1.0f, 0.0f), QVector3D(0.0f, 0.0f, 0.0f)));
```

```
    }

    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();
}

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();
}

QVector2D MyOpenGL::curve(float t)
{
    float x = pow((1 - t), 2) * controlPoints[0].x() + 2 * t * (1 - t) *
controlPoints[1].x() + pow(t, 2) * controlPoints[2].x();
    float y = pow((1 - t), 2) * controlPoints[0].y() + 2 * t * (1 - t) *
controlPoints[1].y() + pow(t, 2) * controlPoints[2].y();
    return QVector2D(x, y);
}
```

```
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();
    }
}

void MyOpenGL::changeLightPosition(QVector3D pos)
{
    lightPos = pos;
    update();
}
```

Для получения нужного светового эффекта были написаны вершинный и фрагментный шейдер.

Вершинный шейдер. Файл с расширением .vsh

Описание:

layout(location = 0) in vec3 objectPosition;: Это атрибут входных данных, представляющий позицию вершины объекта.

layout(location = 1) in vec3 objectColor;: Атрибут входных данных, представляющий цвет вершины объекта.

layout(location = 2) in vec3 normal;: Атрибут входных данных, представляющий нормаль вершины объекта.

uniform mat4 projectionMatrix;: Это uniform-переменная, содержащая матрицу проекции.

uniform mat4 viewMatrix;: Uniform-переменная, содержащая матрицу вида (видовую матрицу).

uniform mat4 modelMatrix;: Uniform-переменная, содержащая модельную матрицу (матрицу преобразования объекта).

out vec3 Color;: Это переменная, передающая цвет вершины во фрагментный шейдер.

out vec3 FragPos;: Переменная, содержащая позицию вершины в мировой системе координат.

out vec3 Normal;: Переменная, передающая нормаль вершины во фрагментный шейдер.

Действия:

Преобразует позицию вершины в мировые координаты и проецирует ее на экран с помощью матриц проекции и вида.

Передает цвет вершины, ее позицию и нормаль фрагментному шейдеру.

```
#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;
}
```

Фрагментный шейдер. Файл с расширением .fsh

Фрагментный шейдер рассчитывает освещение и цвет каждого пикселя на поверхности объекта.

Описание:

in vec3 Color;: Переменная, полученная из вершинного шейдера, содержащая цвет вершины.

in vec3 FragPos;: Переменная, полученная из вершинного шейдера, содержащая позицию вершины.

in vec3 Normal;: Переменная, полученная из вершинного шейдера, содержащая нормаль вершины.

uniform vec3 lightPos;: Uniform-переменная, содержащая позицию источника света.

uniform vec3 viewPos;: Uniform-переменная, содержащая позицию камеры (точку обзора).

out vec4 color;: Выходная переменная, содержащая цвет фрагмента.

Действия:

Рассчитывает освещение на поверхности объекта, основываясь на его нормалях и позиции источника света.

Вычисляет составляющие освещения: амбиентное, диффузное и зеркальное.

Смешивает составляющие освещения с цветом вершины, чтобы получить итоговый цвет фрагмента.

```
#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);
}
```

Тестирование.

Рисунок 1.

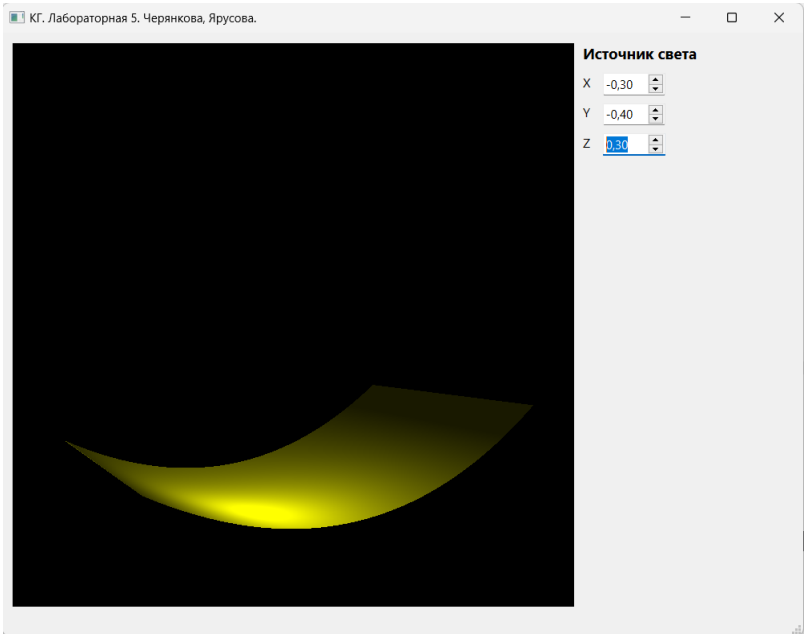

Рисунок 2.

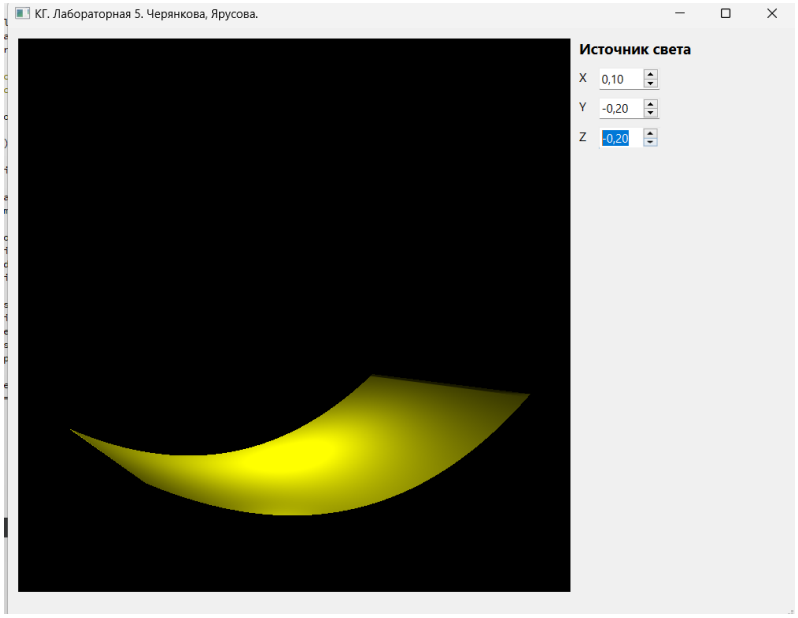

Рисунок 3.

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.