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

ОТЧЕТ

по курсовой работе

по дисциплине «Компьютерная графика»

Тема: «Реализация сцены с визуализацией 3D-сцены»

Студент гр. 7381 _____ Алясова А.Н.

Преподаватель _____ Герасимова Т.В.

Санкт-Петербург

2020

Цель работы.

Реализовать сцену с визуализацией 3D-сцены.

Задачи.

1. Подбор материала по теме для обзора (1-2 страницы), материал должен быть творчески переработан, дополнен примерами вашей реализации. Обязательны ссылки на литературу.

2. Создать описание генерации вашей модели (не создавать в средствах типа Blender, 3D MAX).

3. Разработка демонстрационной сцены.

4. Курсовая должна быть распечатана.

Для выполнения задания необходимо создать сцену (фотореалистичность желательна). Оценка, выставленная за задание, зависит от исполнения сцены, и использованных в ней средств.

Возможности облететь сцену и изменить положение источников света.

Ход работы.

Соберу зонтик из следующих 3D примитивов:

Сфера с небольшой мелкостью разбиения, соответствующей форме зонтика и частично отсеченная плоскостью;

Палочка будет представлять длинный и тонкий цилиндр;

Ручка - торус, но отмасштабированный по одной из осей координат и также отсеченный плоскостью пополам.

За генерацию сферы отвечает разработанный класс `Sphere`, который генерирует точки сферы, с помощью параметрического уравнения сферы с заданным разбиением, вычисляет нормали точек и координаты текстур, также вычисляется массив индексов проходов по вершинам. При отрисовке проходится лишь одна треть вершин, таким образом рисуется не вся сфера, а ее часть.

На рис. 1 пример отрисовки части сферы в каркасном режиме.

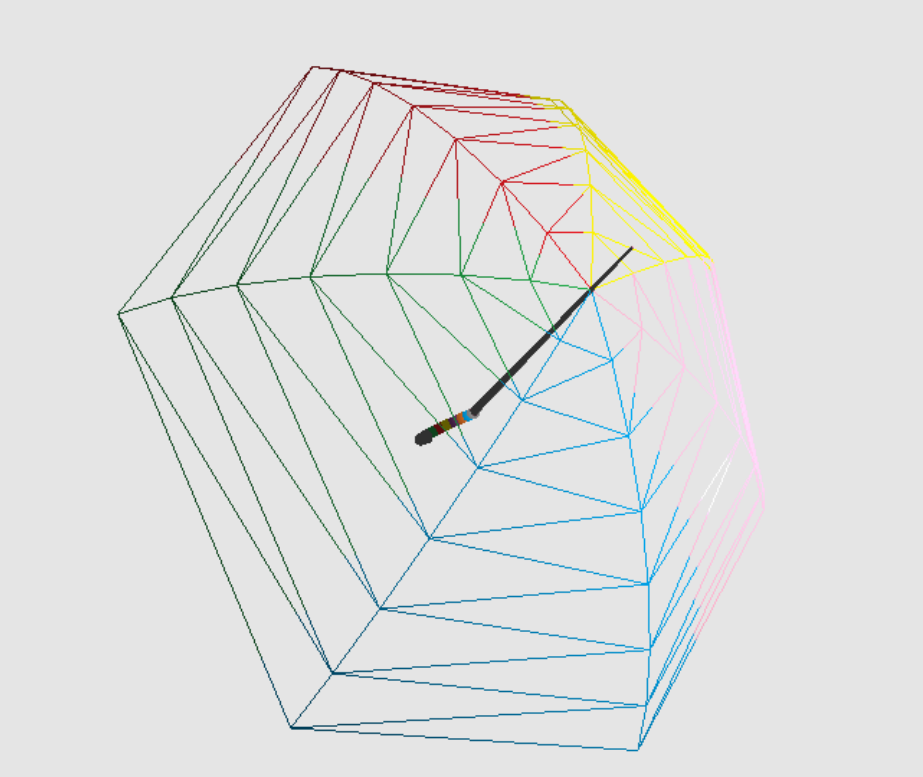

Рисунок 1 - Частично отрисованная сфера в каркасном режиме

Теперь палочка. Как было сказано выше, палочка будет представлять из себя простой длинный и тонкий цилиндр. За генерацию точек цилиндра с заданной мелкостью разбиения, генерацию нормалей, координат текстур и индексацию отвечает класс `Cylindr`. На рис. 2, отрисованная в каркасном режиме длинная палочка.

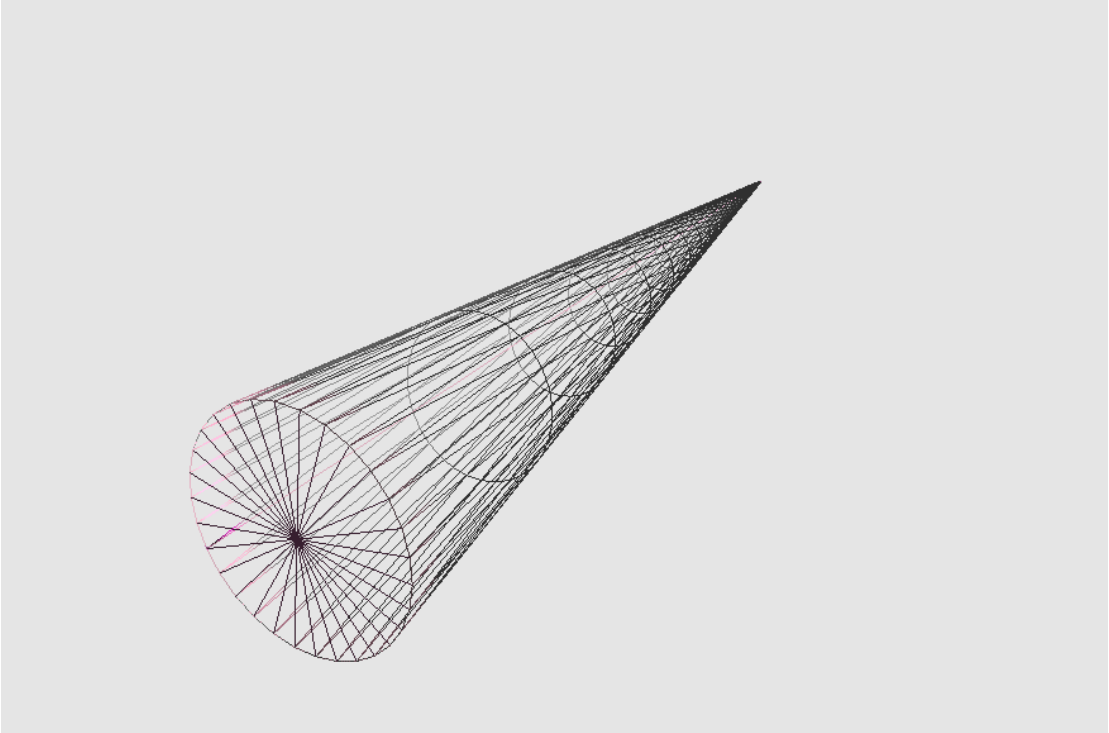

Рисунок 2 - Палочка

Ручка цилиндра будет представлять из себя торус.

За отрисовку торуса отвечает функция `drawTorus`. Точки генерируются с помощью параметрического уравнения торуса.

В вершинном шейдере, с помощью плоскости отсечения, отсекается половина торуса, затем афинными преобразованиями торус вытягивается вдоль оси X, принимая форму ручки зонтика. Получившаяся ручка показана на рис. 3.

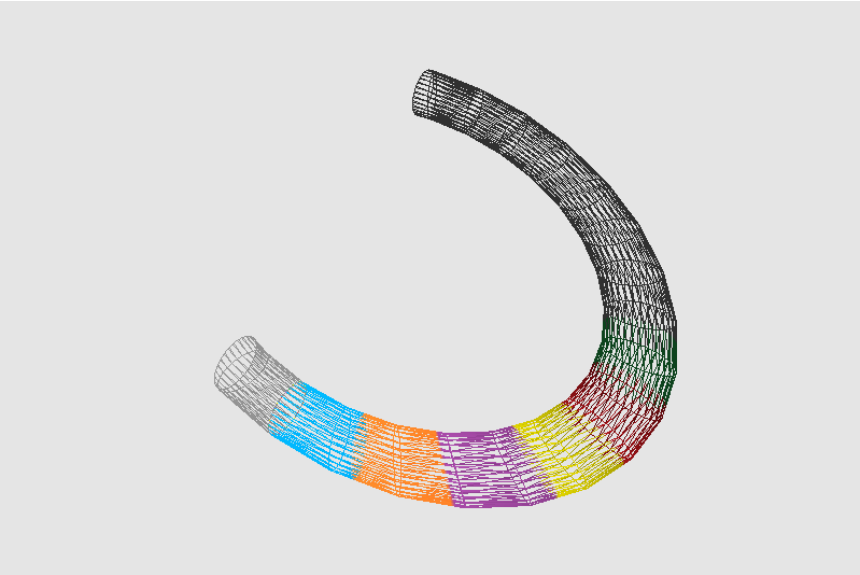

Рисунок 3 - Ручка зонтика

Теперь собираю все вместе, для этого каждую деталь зонтика нужно перевести из пространства модельных координат в пространство мировых координат.

В листинге 1 показан пример отрисовки одной детали зонтика.

```
model.setToIdentity();  //сбрасываю модельную матрицу
model.rotate(90.0f,{1.0f,0.0f,0.0f}); //поворачиваю торус
model.scale({1.0f,1.5f,1.0f});  //вытягиваю, чтобы придать форму ручки
model.translate({-0.6f,0.0f,0.0f}); //переношу на свое место
model.rotate(-45.0f,{1.0f,0.0f,0.0f}); //поворачиваю каждую деталь, что зонтик лежал диагонально
def_sh->bind();                          //бинд шейдерной программы
def_sh->setUniformValue("matrix",projection*view*model);   //передаю матрицы преобразования шейдеру
def_sh->setUniformValue("modelview",view*model);
def_sh->setUniformValue("normal_m",model.normalMatrix());
handler_tex->bind();        //бинд текстуры
def_sh->setUniformValue("texture",0); //передаю текстуру как юниформ переменную
def_sh->release();

drawTorus(def_sh);  //рисую торус
```

Листинг 1 - Отрисовка ручки зонтика

Данный алгоритм повторяется для каждой детали, отличаются лишь модельные матрицы.

Зонтик с разных сторон показан на рис. 4-5.

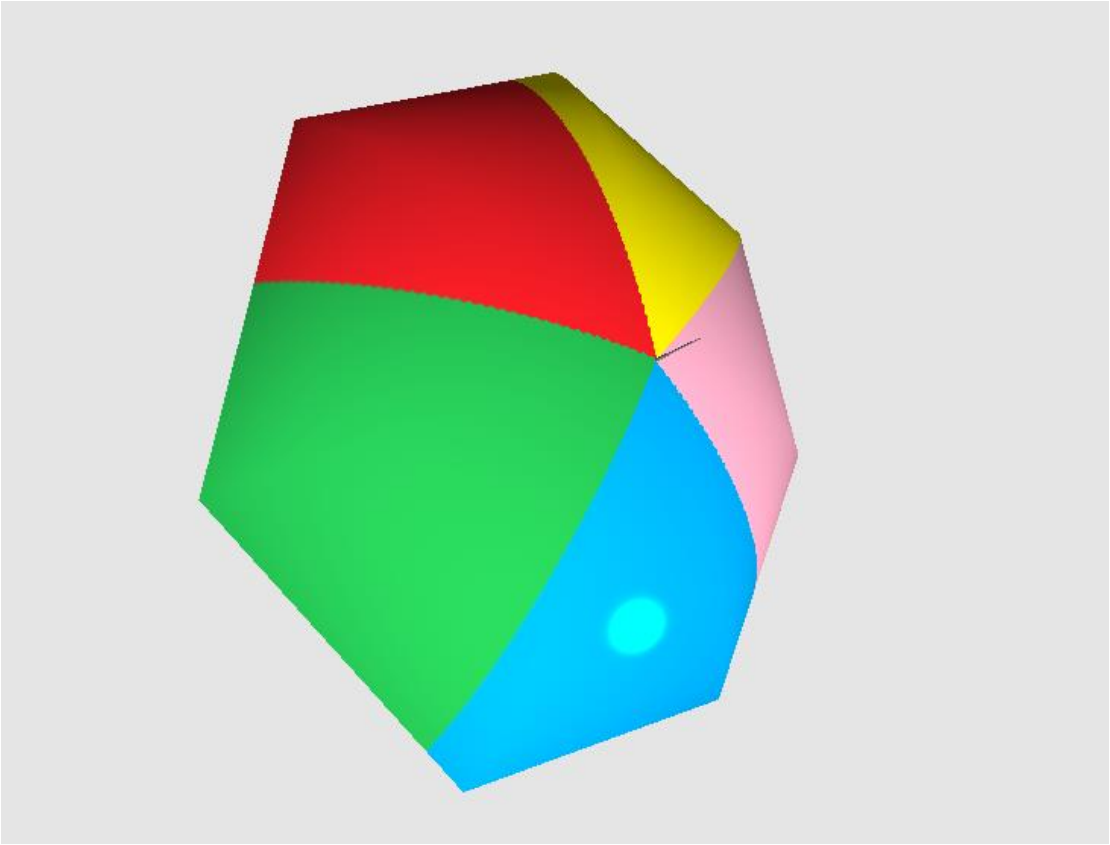

Рисунок 4 - Зонтик спереди

Рисунок 5 - Зонтик сбоку

Каждый класс, генерирующий точки для деталей, также генерирует координаты текстур, которые передаются на вход шейдеру, где цвет координаты фрагментов интерполируются из координат вершин и их цвет связывается с цветом в текстуре.

В Qt существует класс `QOpenGLTexture`, который облегчает работу с текстурами, для создания текстуры достаточно передать конструкторы изображения, и текстура уже готова для передачи шейдеру.

Алгоритм освещения, используемый во фрагментом шейдере, показан в листинге 2.

```
varying vec4 color;
in vec3 eyecoord;
in vec3 tnorm;
in vec2 v_texCoord;

struct LightInfo{
    vec4 position;
    vec3 la; //amb color
    vec3 ld;
    vec3 ls;
};

uniform LightInfo l;

struct MaterialInfo{
    vec3 ka; //amb str
    vec3 kd;
    vec3 ks;
    float Shininess;
};

uniform MaterialInfo material;
uniform sampler2D texture;

void main()
{
    vec3 n = normalize( tnorm );
    vec3 s = normalize( vec3(l.position) - eyecoord );
    vec3 v = normalize(vec3(-eyecoord));
    vec3 r = reflect( -s, n );


    vec3 ambient=l.la*material.ka;

    float sDotN=max(dot(s,tnorm),0.0f);

    vec3 diffuse=l.ld*material.kd*sDotN;

    vec3 spec=l.ls*material.ks*pow(max( dot(r,v), 0.0 ), material.Shininess);

    vec4 color_t=texture2D(texture,v_texCoord);
    gl_FragColor=color_t*vec4((ambient+diffuse+spec),1.0f);
}
```

Листинг 2 - Алгоритм расчета освещения

Класс камеры:

Класс камеры вычисляет нормализованный вектора пространства камеры: `front, right, up,` с помощью углов Эйлера. Алгоритм расчета в листинге 3.

```
void Camera::updateCamVectors()
{
    QVector3D front;
    front.setX( cosf(qDegreesToRadians(this->yaw)) * cosf(qDegreesToRadians(this->pitch)) );
    front.setY( sinf(qDegreesToRadians(this->pitch)) );
    front.setZ( sinf(qDegreesToRadians(this->yaw)) * cosf(qDegreesToRadians(this->pitch)) );
    this->front=front;
    this->front.normalize();

    this->right = QVector3D::normal(this->front, this->worldUp);  // Normalize the vectors, beca
    this->up    = QVector3D::normal(this->right, this->front);
}
```

Листинг 3 - Расчет векторов

При нажатии клавиши wsad, space, left control позиция камеры смещается вдоль вычисленных векторов.

При зажатии левой кнопки мыши и перемещении курсора, вычисляется смещение курсора в координатах x, y, на которые корректируются значения углов Эйлера. В начале каждого кадра класс возвращает матрицу вида, которая используется в расчете проекции вершин.

Таким образом реализовано свободное перемещение по сцене.

Рисунок 6 - Свободное перемещение по сцене.

Выводы.

В ходе выполнения курсовой работы были получены навыки построения модели, настройки материалов, наложения текстур, использования алгоритма освещения средствами последней спецификации OpenGL.

Список использованных источников

1. https://learnopengl.com

2. http://www.opengl-tutorial.org/ru/

3. Расчет точек сферы

http://www.songho.ca/opengl/gl_sphere.html

ПРИЛОЖЕНИЕ А

КЛАСС СЦЕНЫ

```
#include "scene.h"
#include <QtMath>

Scene::Scene(QWidget* parent)
    :QOpenGLWidget(parent)
{
    a=new Axes;
    objs=new Sphere_and_conus;
    hand=new Cylinder(def,30,30,0.08,0.01,5.9);

    type=GL_POLYGON;
    cam=new Camera;
    l_angle=0.0f;
    l_pos={10*cosf(l_angle/50),6,10*sinf(l_angle/50),0.0f};
    l={l_pos,{0.2f,0.2f,0.2f},{1.0f,1.0f,1.0f},{1.9f,1.9f,1.9f}};
    glnc={{1.9f,1.9f,1.9f},{0.9f,0.9f,0.9f},{5.0f,5.0f,6.0f},256.0f};
    mat={{1.1f,1.1f,1.1f},{0.5f,0.5f,0.5f},{0.3f,0.3f,0.3f},8.0f};
}

void Scene::initializeGL(){
    initializeOpenGLFunctions();
    glClearColor(0.9f,0.9f,0.9f,0.3f);
    def_sh=new QOpenGLShaderProgram;
    def_sh-
>addShaderFromSourceFile(QOpenGLShader::Vertex,":/vShader.glsl");
    def_sh-
>addShaderFromSourceFile(QOpenGLShader::Fragment,":/fShader.glsl");
    def_sh->link();
```

```
    spline_sh=new QOpenGLShaderProgram;
    spline_sh->addShaderFromSourceFile(QOpenGLShader::Vertex,":/vShader_spline.glsl");
    spline_sh->addShaderFromSourceFile(QOpenGLShader::Fragment,":/fShader_spline.glsl");
    spline_sh->link();

    initTextures();

    glEnable(GL_CLIP_DISTANCE0);

    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);
}

void Scene::resizeGL(int w, int h){
    glViewport(0,0,w,h);
}

void Scene::paintGL(){

    l_pos={10*cosf(l_angle/50),6,10*sinf(l_angle/50),0.0f};
    l={l_pos,{0.2f,0.2f,0.2f},{1.0f,1.0f,1.0f},{1.9f,1.9f,1.9f}};
    const qreal retinaScale = devicePixelRatio();
    glViewport(0, 0, width() * retinaScale, height() * retinaScale);
    glClear(GL_COLOR_BUFFER_BIT);

    cam->moveCam(&keys);
```

```
QMatrix4x4 model, view=cam->getMatrix(), projection;
Material m=glnc;

projection.perspective(70.0f, 2300.0f/1080.0f, 0.1f, 100.0f);

//matrix.rotate(100.0f * m_frame / 300, 0, 1, 0);

def_sh->bind();
def_sh->setUniformValue("clip",false);
def_sh->setUniformValue("matrix",projection*view*model);
def_sh->setUniformValue("normal_m",model.normalMatrix());
def_sh->setUniformValue("modelview",view*model);
def_sh->setUniformValue("LightInfo.position",l.light_pos);
def_sh->setUniformValue("LightInfo.la",l.la);
def_sh->setUniformValue("LightInfo.ld",l.ld);
def_sh->setUniformValue("LightInfo.ls",l.ls);
def_sh->setUniformValue("MaterialInfo.ka",m.ka);
def_sh->setUniformValue("MaterialInfo.kd",m.kd);
def_sh->setUniformValue("MaterialInfo.ks",m.ks);
def_sh->setUniformValue("MaterialInfo.Shininess",m.Shininess);
def_sh->release();
def_sh->bind();

def_sh->setUniformValue("l.position",l.light_pos);
def_sh->setUniformValue("l.la",l.la);
def_sh->setUniformValue("l.ld",l.ld);
def_sh->setUniformValue("l.ls",l.ls);
def_sh->setUniformValue("material.ka",m.ka);
def_sh->setUniformValue("material.kd",m.kd);
```

```
    def_sh->setUniformValue("material.ks",m.ks);
    def_sh->setUniformValue("material.Shininess",m.Shininess);
    def_sh->release();

    model.setToIdentity();  //сбрасываю модельную матрицу
    model.rotate(90.0f,{1.0f,0.0f,0.0f}); //поворачиваю торус
    model.scale({1.0f,1.5f,1.0f});  //вытягиваю, чтобы придать форму
ручки
    model.translate({-0.6f,0.0f,0.0f}); //переношу на свое место
    model.rotate(-45.0f,{1.0f,0.0f,0.0f}); //поворачиваю каждую
деталь, что зонтик лежал диагонально
    def_sh->bind();                          //бинд шейдерной программы
    def_sh->setUniformValue("matrix",projection*view*model);
//передаю матрицы преобразования шейдеру
    def_sh->setUniformValue("modelview",view*model);
    def_sh->setUniformValue("normal_m",model.normalMatrix());
    handler_tex->bind();        //бинд текстуры
    def_sh->setUniformValue("texture",0); //передаю текстуру как
юниформ переменную
    def_sh->release();

    drawTorus(def_sh);  //рисую торус

    model.setToIdentity();
    model.rotate(-45.0f,{1.0f,0.0f,0.0f});
    model.scale({7.0f,7.0f,5.0f});

    def_sh->bind();
    def_sh->setUniformValue("clip",false);
    def_sh->setUniformValue("matrix",projection*view*model);
    def_sh->setUniformValue("modelview",view*model);
```

```
    def_sh->setUniformValue("normal_m",model.normalMatrix());
    sphere_tex->bind();
    def_sh->setUniformValue("texture",0);
    def_sh->release();
    objs->drawObj(def_sh, GL_FILL);
    model.setToIdentity();
    model.translate({0.0f,2.2f,2.2f});
    model.rotate(90.0f,{1.0f,0.0f,0.0f});

    model.rotate(-45.0f,{1.0f,0.0f,0.0f});

    def_sh->bind();
    def_sh->setUniformValue("clip",false);
    def_sh->setUniformValue("matrix",projection*view*model);
    def_sh->setUniformValue("modelview",view*model);
    def_sh->setUniformValue("normal_m",model.normalMatrix());
    cyl_tex->bind();
    def_sh->setUniformValue("texture",0);
    def_sh->release();
    hand->draw(def_sh);

    if(light_flag)l_angle++;
    ++m_frame;
    update();
}

void Scene::mousePressEvent(QMouseEvent *event){
    start=QPointF(event->x(),event->y());
    if(event->button()==Qt::RightButton){
        this->setCursor(Qt::BlankCursor);
```

```
        mouse_flag=true;
    }
    if(event->button()==Qt::LeftButton){
        light_flag=true;
    }
}

void Scene::mouseReleaseEvent(QMouseEvent *event){
    if(event->button()==Qt::RightButton){
        QCursor a;
        a.setPos(QWidget::mapToGlobal({width()/2,height()/2}));
        setCursor(a);
        mouse_flag=false;
        this->unsetCursor();
    }
    if(event->button()==Qt::LeftButton){
        light_flag=false;
    }
}

void Scene::drawTorus(QOpenGLShaderProgram* m_program, double r,
double c, int rSeg, int cSeg)
{
    QVector<QVector3D> points;
    QVector<QVector2D> tex_coords;
    QVector<QVector3D> normals;

      for (int i = 0; i < rSeg; i++) {
        for (int j = 0; j <= cSeg; j++) {
          for (int k = 0; k <= 1; k++) {
```

```
            double s = (i + k) % rSeg + 0.5;
            double t = j % (cSeg + 1);

            float x = (c + r * cos(s * M_PI*2 / rSeg)) * cosf(t *
M_PI*2 / cSeg);
            float y = (c + r * cos(s * M_PI*2 / rSeg)) * sinf(t *
M_PI*2 / cSeg);
            float z = r * sin(s * M_PI*2 / rSeg);

            float u = (i + k) / (float) rSeg;
            float v = t / (float) cSeg;

            points.append({2 * x, 2 * y, 2 * z});
            tex_coords.append({u,v});
            normals.append({2 * x, 2 * y, 2 * z});
          }
        }
      }

      m_program->bind();

      m_program->setUniformValue("clip",true);
      m_program->setAttributeArray(0, points.data());
      m_program->setAttributeArray(2,normals.data());
      m_program->setAttributeArray(3,tex_coords.data());

      m_program->setAttributeValue(1,QVector3D{0.1f,0.8f,0.1f});

      m_program->enableAttributeArray(0);
      m_program->enableAttributeArray(2);
      m_program->enableAttributeArray(3);
```

```
      glFrontFace(GL_CW);
      glDrawArrays(GL_TRIANGLE_STRIP,0,points.size());
      m_program->disableAttributeArray( 0);
      m_program->disableAttributeArray(2);
      m_program->release();


}
void Scene::mouseMoveEvent(QMouseEvent *event){
    start.setX(event->x()-start.x());
    start.setY(start.y()-event->y());

    if(mouse_flag){
        this->cam->changeYawAndPitch(start.x(),start.y());
    }

    start=event->pos();
    update();
}

void Scene::keyPressEvent(QKeyEvent *event){
    keys.insert(event->key());
}

void Scene::keyReleaseEvent(QKeyEvent *event){
    if(event->isAutoRepeat()==false)keys.remove(event->key());
}

QOpenGLTexture* Scene::initTexture(const char *nof){
```

```
    QOpenGLTexture * texture = new
QOpenGLTexture(QImage(nof).mirrored());

    // Set nearest filtering mode for texture minification
    texture->setMinificationFilter(QOpenGLTexture::Nearest);

    // Set bilinear filtering mode for texture magnification
    texture->setMagnificationFilter(QOpenGLTexture::Linear);

    // Wrap texture coordinates by repeating
    // f.ex. texture coordinate (1.1, 1.2) is same as (0.1, 0.2)
    texture->setWrapMode(QOpenGLTexture::Repeat);

    return texture;
}

void Scene::initTextures(){
    sphere_tex=initTexture(":/sphere.bmp");
    handler_tex=initTexture(":/handler.bmp");
    cyl_tex=initTexture(":/cylinder.bmp");
}
```

ПРИЛОЖЕНИЕ Б

КЛАСС КАМЕРЫ

```
#include "camera.h"


Camera::Camera(QVector3D pos, QVector3D worldUp): pos(pos),
worldUp(worldUp),yaw(YAW),pitch(PITCH),front({0.0f,0.0f,-1.0f}),
movementSpeed(0.1f)
{
    sens=0.1f;
    updateCamVectors();
}


QMatrix4x4 Camera::getMatrix()
{
    QMatrix4x4 a;
    a.lookAt(this->pos,this->pos+this->front,this->up);
    return a;
}


void Camera::changeYawAndPitch(float yaw, float pitch)
{
    this->yaw   += yaw*sens;
    this->pitch += pitch*sens;


    // Make sure that when pitch is out of bounds, screen doesn't get
flipped
    if (true)
    {
        if (this->pitch > 89.0f)
            this->pitch = 89.0f;
        if (this->pitch < -89.0f)
```

```
            this->pitch = -89.0f;
    }
    // Update Front, Right and Up Vectors using the updated Eular
angles
    this->updateCamVectors();
}


void Camera::moveCam(QSet<int> *keys)
{
    if(keys->contains(Qt::Key_W))
        this->pos+=this->movementSpeed*this->front;
    if(keys->contains(Qt::Key_S))
       this->pos-=this->movementSpeed*this->front;
    if(keys->contains(Qt::Key_A))
        this->pos-=this->right*this->movementSpeed;
    if(keys->contains(Qt::Key_D))
        this->pos+=this->right*this->movementSpeed;
    if(keys->contains(Qt::Key_Space))
        this->pos+=this->up*this->movementSpeed;
    if(keys->contains(Qt::Key_Control))
        this->pos-=this->up*this->movementSpeed;
}


void Camera::updateCamVectors()
{
    QVector3D front;
    front.setX( cosf(qDegreesToRadians(this->yaw)) *
cosf(qDegreesToRadians(this->pitch)) );
    front.setY( sinf(qDegreesToRadians(this->pitch)) );
    front.setZ( sinf(qDegreesToRadians(this->yaw)) *
cosf(qDegreesToRadians(this->pitch)) );
```

```
    this->front=front;
    this->front.normalize();


    this->right = QVector3D::normal(this->front, this->worldUp);  // Normalize the vectors, because their length gets closer to 0 the more you look up or down which results in slower movement.
    this->up    = QVector3D::normal(this->right, this->front);
}
```

ПРИЛОЖЕНИЕ В

КЛАСС СФЕРЫ

```
#include "sphere_and_conus.h"


Sphere_and_conus::Sphere_and_conus():sectorCount(7),stackCount(122),ce
nter({0.0f,0.0f,0.0f}),rad_sphere(1){
    initSpherePoints();
}


void Sphere_and_conus::drawObj(QOpenGLShaderProgram *m_program, GLenum
type)
{
    glPolygonMode(GL_FRONT_AND_BACK,type);
    drawSphere(m_program);
}


void Sphere_and_conus::setSphereResX(int res)
{
    sectorCount=res;
    initSpherePoints();
}


void Sphere_and_conus::setSphereResY(int res)
{
    stackCount=res;
    initSpherePoints();
}



void Sphere_and_conus::initSpherePoints()
```

```
{
    sphere_points.clear();
    sphere_normals.clear();
    indices.clear();
    float x, y, z, xy;                              // vertex position

    float sectorStep = 2 * M_PI / sectorCount;
    float stackStep = M_PI / stackCount;
    float sectorAngle, stackAngle;

    for(uint i = 0; i <= stackCount; ++i)
    {
        stackAngle = M_PI / 2 - i * stackStep;        // starting from
pi/2 to -pi/2
        xy = rad_sphere * cosf(stackAngle);             // r * cos(u)
        z = rad_sphere * sinf(stackAngle);              // r * sin(u)

        // add (sectorCount+1) vertices per stack
        // the first and last vertices have same position and normal,
but different tex coords
        for(uint j = 0; j <= sectorCount; ++j)
        {
            sectorAngle = j * sectorStep;           // starting from 0
to 2pi

            // vertex position (x, y, z)
            x = xy * cosf(sectorAngle);             // r * cos(u) *
cos(v)
            y = xy * sinf(sectorAngle);             // r * cos(u) *
sin(v)
            sphere_points.append(center+QVector3D{x,y,z});
```

```
            sphere_normals.append(QVector3D{x,y,z});

            float s = (float)j / sectorCount;
            float t = (float)i / stackCount;

            tex_coords.append({s,t});
        }
    }

    int k1, k2;
    for(uint i = 0; i < stackCount/3; ++i)
    {
        k1 = i * (sectorCount + 1);
        k2 = k1 + sectorCount + 1;

        for(uint j = 0; j < sectorCount; ++j, ++k1, ++k2)
        {

            if(i != 0)
            {
                indices.push_back(k1);
                indices.push_back(k2);
                indices.push_back(k1 + 1);
            }

            // k1+1 => k2 => k2+1
            if(i != (stackCount-1))
            {
                indices.push_back(k1 + 1);
                indices.push_back(k2);
```

```
                indices.push_back(k2 + 1);
            }
        }
    }
}

void Sphere_and_conus::drawSphere(QOpenGLShaderProgram *m_program)
{
    initializeOpenGLFunctions();

    m_program->bind();

    m_program->setAttributeArray(0, sphere_points.data());
    m_program->setAttributeArray(2,sphere_normals.data());
    m_program->setAttributeArray(3,tex_coords.data());
    m_program-
>setAttributeValue("colorAttr",QVector3D{0.1f,0.8f,0.1f});

    m_program->enableAttributeArray(0);
    m_program->enableAttributeArray(2);

    glFrontFace(GL_CW);

glDrawElements(GL_TRIANGLES,indices.size(),GL_UNSIGNED_INT,indices.dat
a());

    m_program->disableAttributeArray( 0);
    m_program->disableAttributeArray(2);
    m_program->disableAttributeArray(3);
    m_program->release();
}
```

```
QVector3D Sphere_and_conus::SphereFun(float u, float v)
{
    return
{rad_sphere*sinf(u)*sinf(v),rad_sphere*sinf(u)*cosf(v),rad_sphere*cosf
(v)};
}
```