**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.

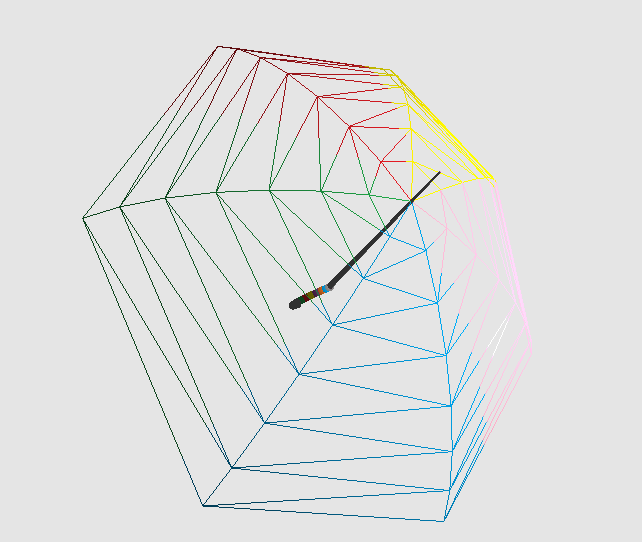


Рисунок 1 - Частично отрисованная сфера в каркасном режиме

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.

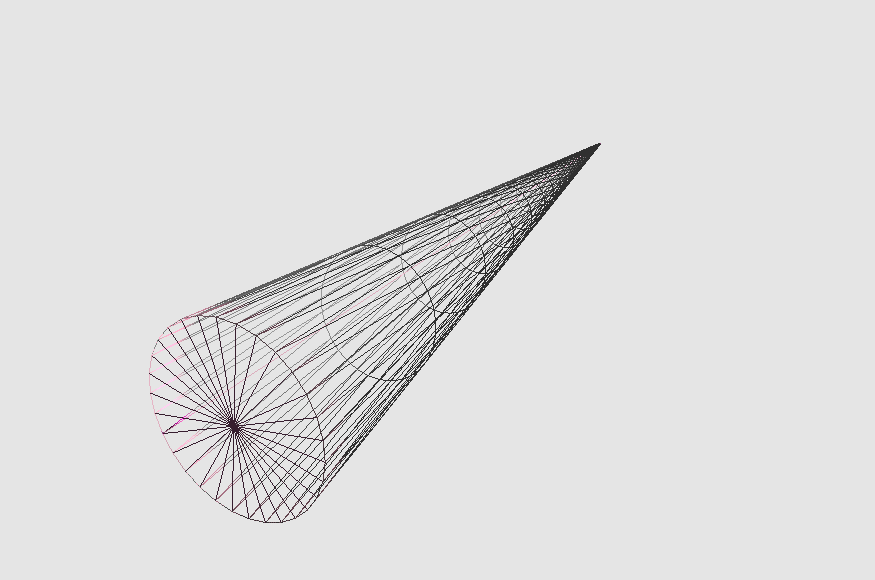


Рисунок 2 - Палочка

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 Х, принимая форму ручки зонтика. Получившаяся ручка показана на рис. 3.

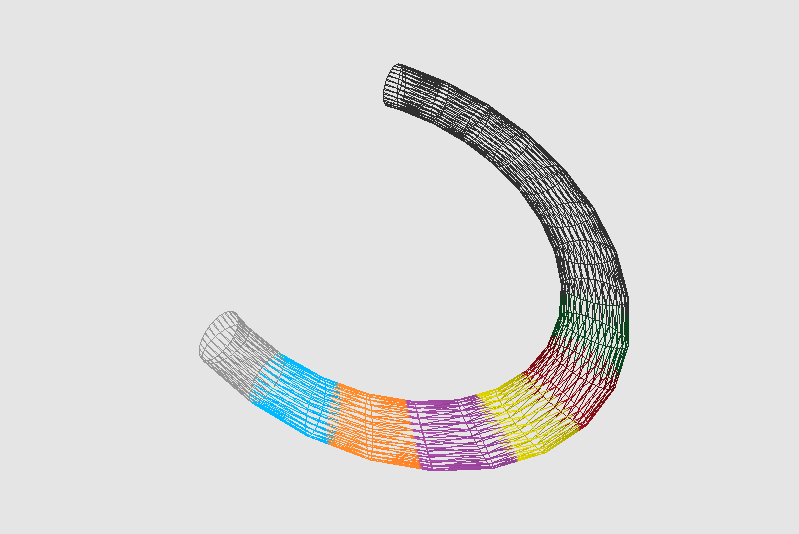
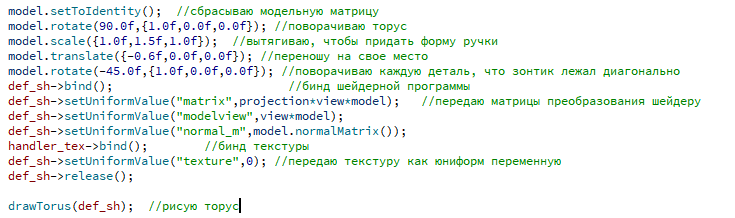


Рисунок 3 - Ручка зонтика

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.



Листинг 1 - Отрисовка ручки зонтика

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

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

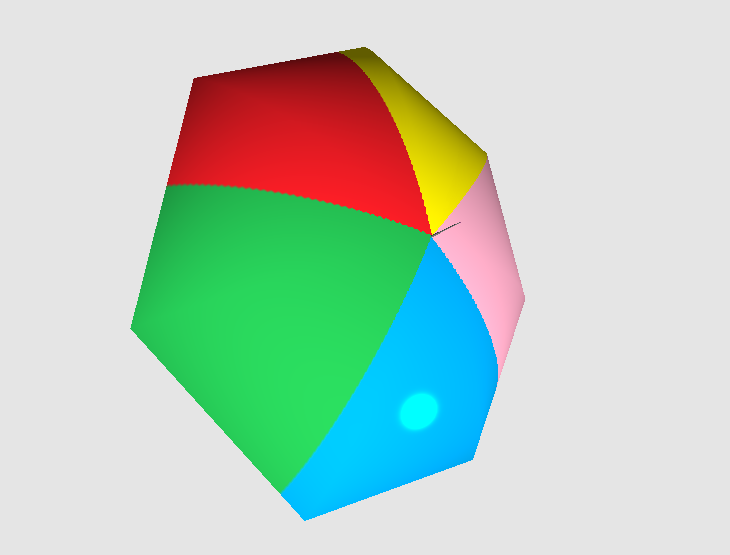


Рисунок 4 - Зонтик спереди

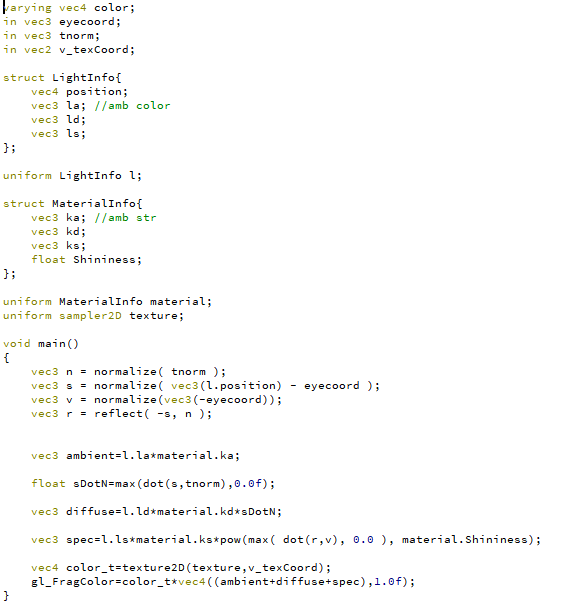


Рисунок 5 - Зонтик сбоку

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.

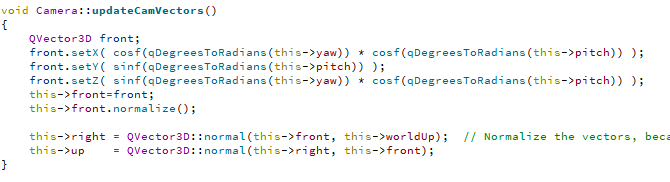
Алгоритм освещения, используемый во фрагментом шейдере, показан в листинге 2.



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.



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),center({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.data());

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)};

}