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

ОТЧЕТ

по лабораторной работе №6

по дисциплине «Компьютерная графика»

Тема: «Реализация трехмерного объекта с использованием библиотеки OpenGL»

Студентка гр. 7381 _____ Алясова А.Н.

Студентка гр. 7381 _____ Кушкоева А.О.

Преподаватель _____ Герасимова Т.В.

Санкт-Петербург

2020

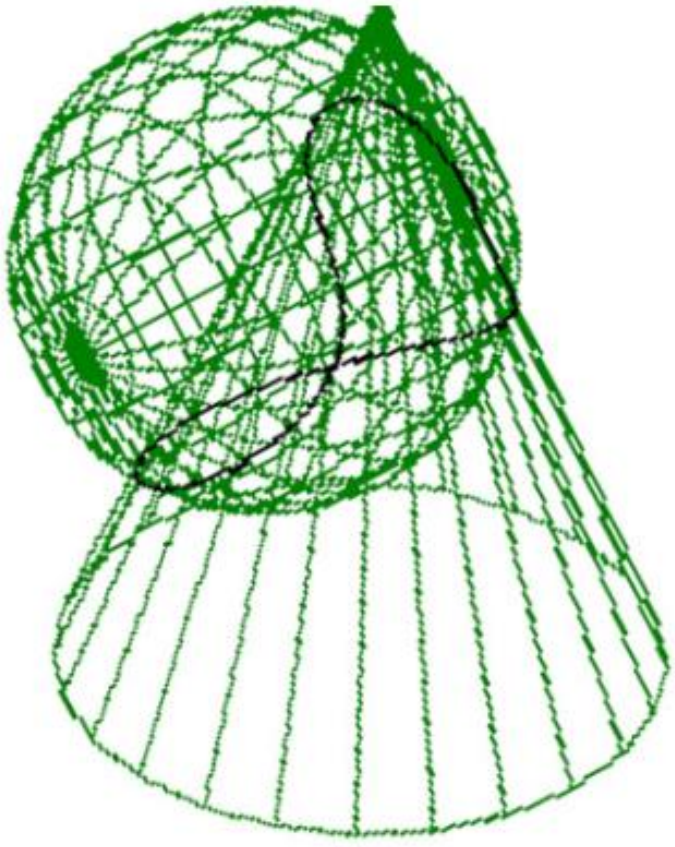
Цель работы.

Разработать программу, реализующую представление разработанного вами трехмерного рисунка, используя предложенные функции библиотеки OpenGL (матрицы видового преобразования, проецирование) и язык GLSL.

Разработанная программа должна быть пополнена возможностями остановки интерактивно различных атрибутов через вызов соответствующих элементов интерфейса пользователя, замена типа проекции, управление преобразованиями, как с помощью мыши, так и с помощью диалоговых элементов.

Написать программу, рисующую проекцию трехмерного каркасного объекта.

Задание.



Требования

1) Грани объекта рисуются с помощью доступных функций рисования отрезка в координатах окна. При этом использовать шейдеры GLSL и OpenGL

2) Вывод многогранника с удалением или прорисовкой невидимых граней;

3) Ортогональное и перспективное проецирование;

4) Перемещения, повороты и масштабирование многогранника по каждой из осей независимо от остальных.

5) Генерация многогранника с заданной мелкостью разбиения.

6) Д.б. установлено изменение свойств источника света (интенсивность).

7) При запуске программы объект сразу должно быть хорошо виден.

8) Пользователь имеет возможность вращать фигуру (2 степени свободы) и изменять параметры фигуры.

9) Возможно изменять положение наблюдателя.

10) Нарисовать оси системы координат.

11) Все варианты требований могут быть выбраны интерактивно.

Общие сведения.

Основная задача, которую необходимо решить при выводе трехмерной графической информации, заключается в том, что объекты, описанные в мировых координатах, необходимо изобразить на плоской области вывода экрана, т.е. требуется преобразовать координаты точки из мировых координат (x,y,z) в оконные координаты (X,Y) ее центральной проекции. Это отображение выполняют в несколько этапов.

Первый этап – *видовое преобразование* – преобразование мировых координат в видовые (видовая матрица);

второй этап – *перспективное преобразование* – преобразование видовых координат в усеченные (матрица проекции).

Для того, чтобы активизировать какую-либо матрицу, надо установить текущий режим матрицы, для чего служит команда:

`void glMatrixMode (GLenum mode)`

Параметр `mode` определяет, с каким набором матриц будет выполняться последовательность операций, и может принимать одно из трех значений: GL_MODELVIEW, GL_PROJECTION, GL_TEXTURE. Для определения элементов матрицы используются следующие команды: `glLoadMatrix`, `glLoadIdentity`.

Преобразование объектов выполняется при помощи следующих операций над матрицами.

***void glRotate[f d]*(GLdouble angle, GLdouble x, GLdouble y, GLdouble z)**

Эта команда рассчитывает матрицу для выполнения вращения вектора против часовой стрелки на угол, определяемый параметром angle, осуществляемого относительно точки (x,y,z). После выполнения этой команды все объекты изображаются повернутыми.

***void glTranslate[f d]*(GLdouble x, GLdouble y, GLdouble z);**

При помощи этой команды осуществляется перенос объекта на расстояние x по оси X, на расстояние y по оси У и на z по оси Z.

Проекции.

Несоответствие между пространственными объектами и плоским изображением устраняется путем введения проекций, которые отображают объекты на двумерной проекционной картинной плоскости. Очень важное значение имеет расстояние между наблюдателем и объектом, поскольку "эффект перспективы" обратно пропорционален этому расстоянию. Таким образом, вид проекции зависит от расстояния между наблюдателем и картинной плоскостью, в зависимости от которой различают два основных класса проекций: *параллельные* и *центральные*. В работе использовалась центральная проекция, а именно перспективная проекция. Для задания проекции использовалась команда.

gluPerspective(Gldouble angley,Gldouble aspect,Gldouble znear,Gldouble zfar).

Предполагается, что точка схода имеет координаты (0, 0, 0) в видовой системе координат. Параметр angley задает угол видимости (в градусах) в направлении оси y. В направлении оси x угол видимости задается через отношение сторон aspect, которое обычно определяется отношением сторон области вывода. Два других параметра задают расстояние от наблюдателя (точки схода) до ближней и дальней плоскости отсечения. Ориентация объекта задана при помощи команды

gluLookAp(Gldouble eyex,Gldouble eyey,Gldouble eyez,Gldouble eyex, Gldouble centerx,Gldouble centery,Gldouble centerz,Gldouble upx,Gldouble upy,Gldouble upz)

точка наблюдения задается группой параметров eye, центр сцены – center, верх сцены – up.

```
void CGLlab3View::OnSize(UINT nType, int cx, int cy)
{
	COpenGLView::OnSize(nType, cx, cy);
	// Установка параметров области вывода, проекции
	// и преобразования координат
	glViewport(0, 0, cx, cy);
	GLdouble gldAspect = (GLdouble) cx/ (GLdouble) cy;
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	gluPerspective(10.0, gldAspect, 1.0, 100.0);
	// Точка наблюдения
	gluLookAt(6, 8, h, 0, 0, 0, 0, 0, 1);
	glMatrixMode(GL_MODELVIEW);
}
```

Для получения реалистичного изображения используем освещение сцены. Число источников света может достигать восьми. Далее приводится фрагмент программы, обеспечивающий включение источника света и задание его параметров. В этом фрагмента показаны способы задания свойств материала объекта и включение необходимых тестов при отображении трехмерных объектов.

```
void CGLlab3View::OnInitialUpdate()
{
	COpenGLView::OnInitialUpdate();

	// Устанавливаем параметры источника света
	static float ambient[] = {0.0f, 0.0f, 0.0f, 1.0f};
	static float diffuse[] = {1.0, 1.0, 1.0, 1.0};
	static float specular[] = { 1.0, 1.0, 1.0, 1.0 };
	static float position[] = {100.0, 60.0, 150.0, 0.0};

	// Определяем свойства материала лицевой поверхности
	static float front_mat_shininess[] = {10.0};
	static float front_mat_specular[] = {0.5, 0.4, 0.4, 0.1};
	static float front_mat_diffuse[] = {0.0, 0.9, 0.3, 0.1};

	// Определяем свойства материала обратной поверхности
	static float back_mat_shininess[] = {3.2};
	static  float  back_mat_specular[]  =  {0.07568,  0.61424,  0.07568,
1.0};
	static float back_mat_diffuse[] = {0.633, 0.727811, 0.633, 1.0};

	static float lmodel_ambient[] = {1.0, 1.0, 0.0, 1.0};
	static float lmodel_twoside[] = {GL_TRUE};

	// Определяем цвет фона используемый по умолчанию
	glClearColor(1.0f, 0.96f, 0.866f, 1.0f);
	// Включаем различные тесты
	glDepthFunc(GL_LEQUAL);
	glEnable(GL_DEPTH_TEST);
	glEnable(GL_BLEND);
	glBlendFunc(GL_ONE, GL_SRC_COLOR);

	// Задаем источник света
	glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
	glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
	glLightfv(GL_LIGHT0, GL_POSITION, position);
	glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
	glLightModelfv(GL_LIGHT_MODEL_TWO_SIDE, lmodel_twoside);

	// Разрешаем освещение и включаем источник света
	glEnable(GL_LIGHTING);
	glEnable(GL_LIGHT0);

	// Устанавливаем параметры материалов
	glMaterialfv(GL_FRONT, GL_SHININESS, front_mat_shininess);
```

```
glMaterialfv(GL_FRONT, GL_SPECULAR, front_mat_specular);
glMaterialfv(GL_FRONT, GL_DIFFUSE, front_mat_diffuse);
glMaterialfv(GL_BACK, GL_SHININESS, back_mat_shininess);
glMaterialfv(GL_BACK, GL_SPECULAR, back_mat_specular);
glMaterialfv(GL_BACK, GL_DIFFUSE, back_mat_diffuse);
```

Ход работы.

В результате работы, было написано приложение, реализующее все требования:

1) Грани объекта рисуются с помощью доступных функций рисования отрезка в координатах окна. При этом использовать шейдеры GLSL и OpenGL.

Расчет точек конуса и сферы, а также индексов их прохода происходит в классе `Sphere_and_conus`. Координаты вершин и их нормали передаются вертексному шейдеру, в котором координаты вершины умножаются на `mvp` матрицу и передаются фрагментному шейдеру, в котором происходит расчет цвета фрагмента по алгоритму затенения. Исходный код фрагментного шейдера показан в приложении А. Работы приложения с закрашеными гранями с работой освещения показана на рис. 1.

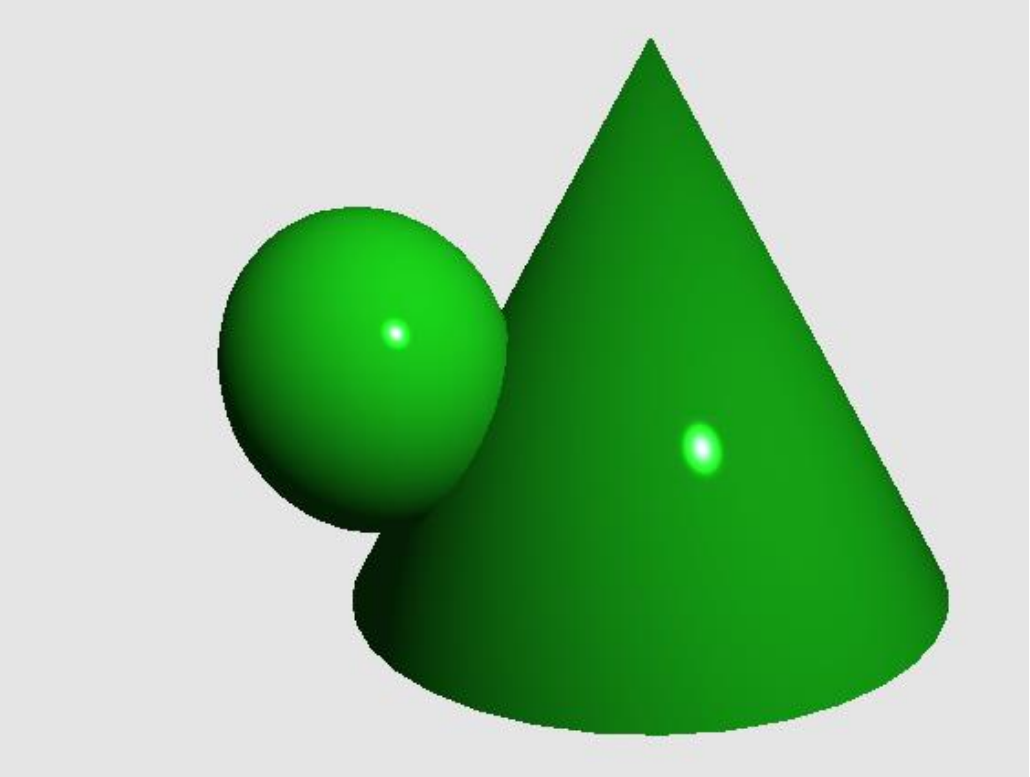


Рисунок 1 - Работа приложения

2) Вывод многогранника с удалением или прорисовкой невидимых граней.

Для прорисовки и удаления граней объект рисуется с включенным режимом `glPolygonMode(GL_FRONT_AND_BACK,GL_LINE)`, который вместо рисования граней обводит их, для того, чтобы объект был каркасным и при

этом оставалась возможность использовать тест `glCullFace(GL_FRONT)`, который отбрасывает грани, в зависимости от направления обхода.

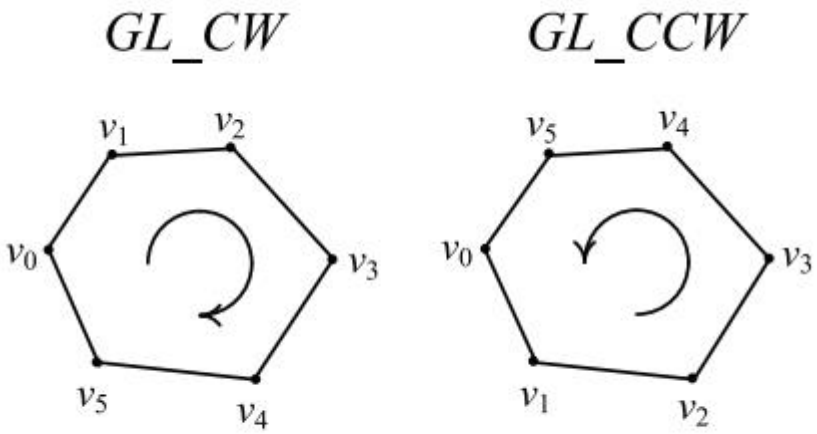


Рис. 12.6 ❖ Возможные варианты направления обхода

По направления обхода при отрисовки граней можно определить лицевая сторона или нет, если сторона лицевая, то грань рисуется, иначе отбрасывается.

На рис. 2-3 показана работа приложения в каркасном режиме с отрисовкой и без отрисовки невидимых граней.

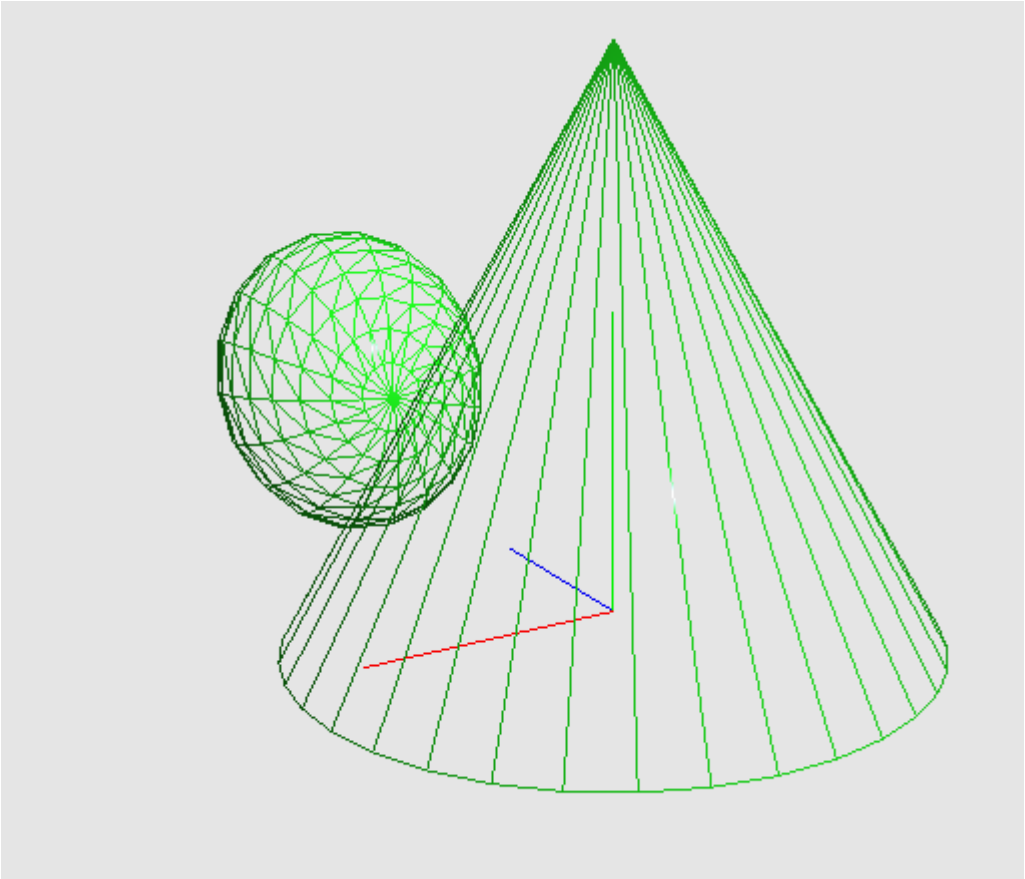


Рисунок 2 - Невидимых граней нет

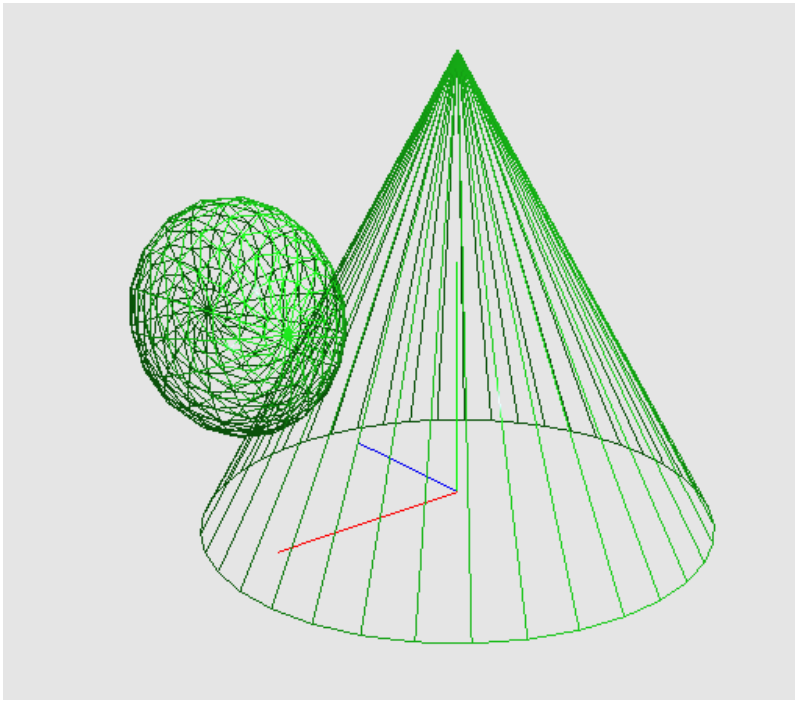


Рисунок 3 - Отрисовка невидимых граней

3) Ортогональное и перспективное проецирование.

Проецирования реализуется матричными преобразованиями, которые вычисляются внутри вертексного шейдера. Примеры работы в обоих режимах показаны на рис. 4-5.

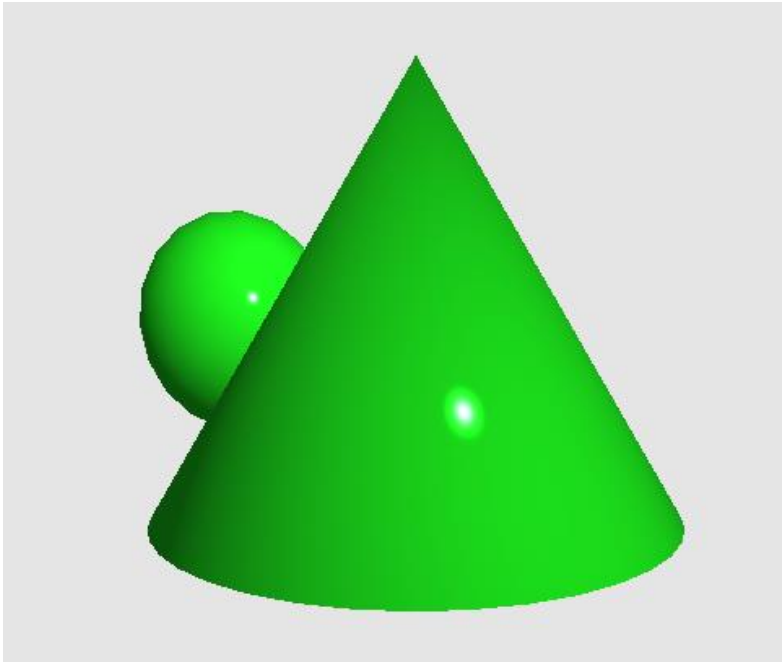


Рисунок 4 - Перспективная проекция

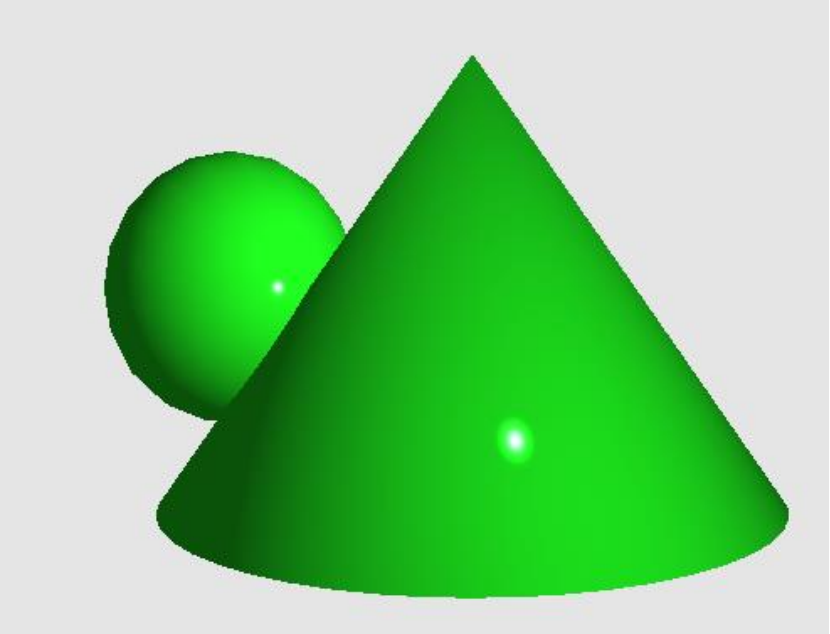


Рисунок 5 - Ортогональная проекция

4) Перемещения, повороты и масштабирование многогранника по каждой из осей независимо от остальных.

Все требования, включая афинные преобразования, выбирются интерактивно с помощью интерфейса вправой части окна.

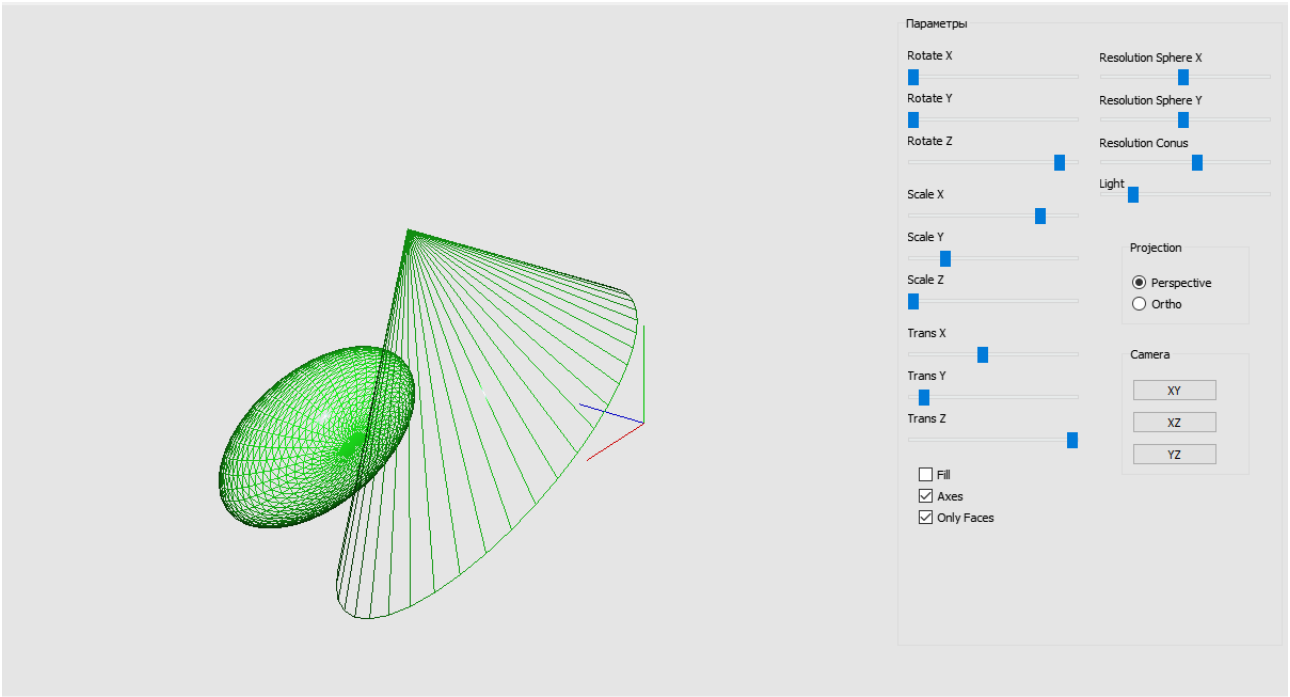


Рисунок 6 - Афинные преобразования с помощью интерфейса пользователя

5) Генерация многогранника с заданной мелкостью разбиения.

Все точки моделируются математически, поэтому можно изменить разбиение обоих объектов. Примеры разбиения конуса и сферы на рис. 7-8.

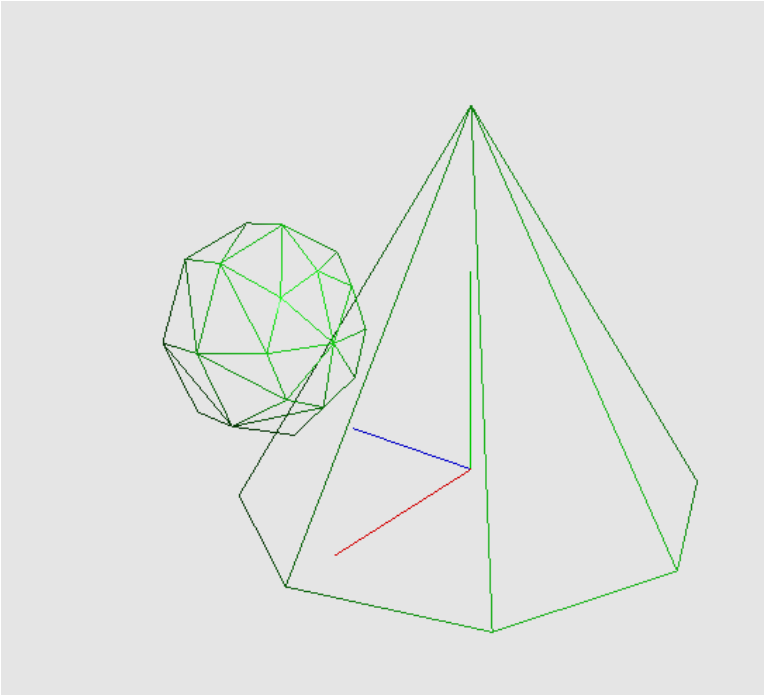


Рисунок 7 - Мало разбиений

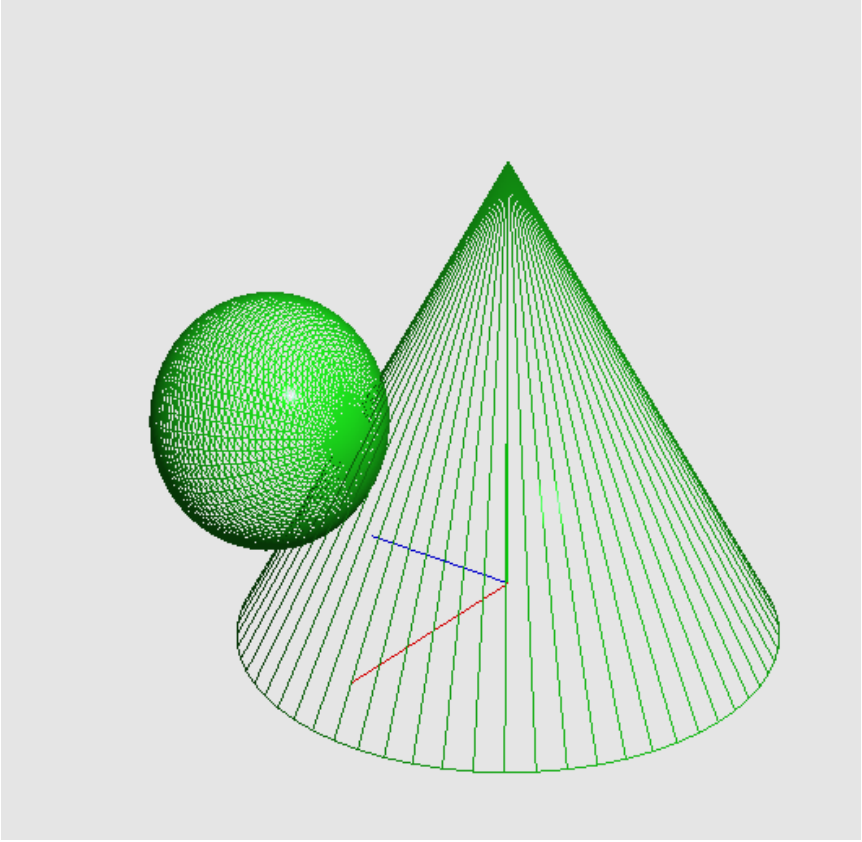


Рисунок 8 - Много разбиений

6) Д.б. установлено изменение свойств источника света (интенсивность).

За изменения интенсивности отвечает слайдер.

На рис. 9-10 работа программы с разной интенсивностью света.

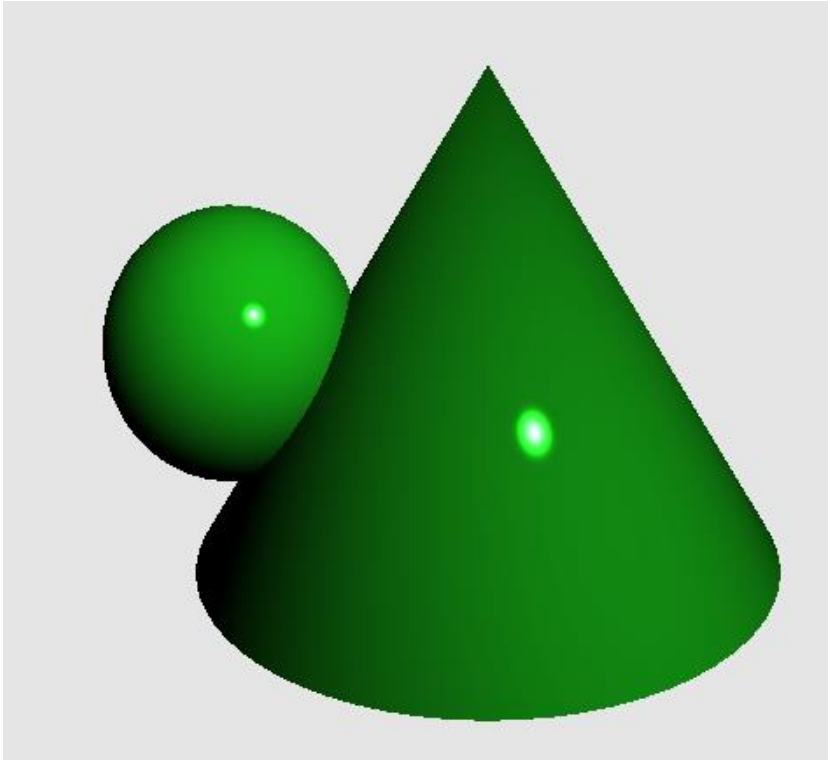


Рисунок 9 - Мало света

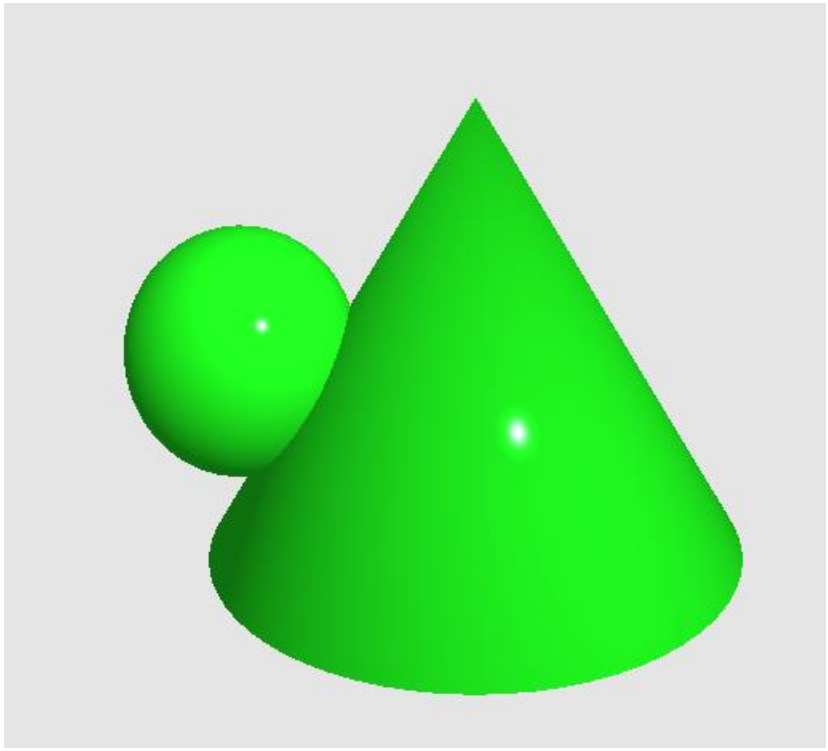


Рисунок 10 - Добавление интенсивности

7) При запуске программы объект сразу должно быть хорошо виден.

Значение камеры по умолчанию выбрано так, что объект видно хорошо.

8) Пользователь имеет возможность вращать фигуру (2 степени свободы) и изменять параметры фигуры.

9) Возможно изменять положение наблюдателя.

Зажатием левой кнопки мыши, пользователь перемещает камеру вокруг объекта. Зажатие правой кнопки мыши - вращение объекта с 2мя степенями свободы.

10) Нарисовать оси системы координат.

Как видно из примеров, оси рисуются, с помощью интерфейса их можно отключить.

Выводы.

В итоге лабораторной работы получены навыки работы с 3d объектами, написано приложение с широким интерфейсом.

ПРИЛОЖЕНИЕ А
ФРАГМЕНТНЫЙ ШЕЙДЕР

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

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;


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);

    gl_FragColor=color*vec4((ambient+diffuse+spec),1.0f);
}
```

ПРИЛОЖЕНИЕ Б
КЛАСС СЦЕНЫ

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

Scene::Scene(QWidget* parent)
    :QOpenGLWidget(parent)
{
    a=new Axes;
    objs=new Sphere_and_conus;

    cam_y=3.5f;
    cam_angle=-65.0f;
    cam_r=5.9f;
    persp=true;

    angle_x=0;
    angle_y=0;
    angle_z=0;

    tr_x=0;
    tr_y=0;
    tr_z=0;

    sc_x=1;
    sc_y=1;
    sc_z=1;

    face=true;
    axes=true;
    fill=false;
    mat_flag=true;

    type=GL_POLYGON;

    deep=9;


l={{5.0f,5.0f,5.0f,0.0f},{0.4f,0.4f,0.4f},{1.0f,1.0f,1.0f},{1.9f,1.9f,1.9
f}};
    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::setProject(bool f)
```

```
{
    this->persp=f;
    update();
}

void Scene::setAngleX(GLfloat angle)
{
    this->angle_x=angle;
    update();
}

void Scene::setAngleY(GLfloat angle)
{
    this->angle_y=angle;
    update();
}

void Scene::setAngleZ(GLfloat angle)
{
    this->angle_z=angle;
    update();
}

void Scene::setFace(bool f)
{
    this->face=f;
    update();
}

void Scene::setAxes(bool f)
{
    this->axes=f;
    update();
}

void Scene::setScX(GLfloat sc)
{
    this->sc_x=sc;
    update();
}

void Scene::setScY(GLfloat sc)
{
    this->sc_y=sc;
    update();
```

```
}

void Scene::setScZ(GLfloat sc)
{
    this->sc_z=sc;
    update();
}

void Scene::setTrX(GLfloat tr)
{
    this->tr_x=tr;
    update();
}

void Scene::setTrY(GLfloat tr)
{
    this->tr_y=tr;
    update();
}

void Scene::setTrZ(GLfloat tr)
{
    this->tr_z=tr;
    update();
}

void Scene::setFill(bool f)
{
    this->fill=f;
    update();
}


void Scene::setLight(float lght)
{
    l.la={lght,lght,lght};
    update();
}

void Scene::setType(GLenum type)
{
    this->type=type;
    update();
}
```

```
void Scene::setDeep(int d)
{
    this->deep=d;
    update();
}

void Scene::setMat(bool f)
{
    this->mat_flag=f;
    update();
}

void Scene::setSphereResX(int res)
{
    this->objs->setSphereResX(res);
    update();
}

void Scene::setSphereResY(int res)
{
    this->objs->setSphereResY(res);
    update();
}

void Scene::setConusRes(int res)
{
    this->objs->setConusRes(res);
    update();
}

void Scene::setCamXY()
{
    this->cam_angle=90.0f;
    this->cam_y=0.0f;
    this->cam_r=4.0f;
    update();

}

void Scene::setCamXZ()
{
    this->cam_angle=0.0f;
    this->cam_r=0.0001f;
    this->cam_y=4.0f;
```

```
void Scene::setCamYZ()
{
    this->cam_angle=0.0f;
    this->cam_y=0.0f;
    this->cam_r=4.0f;
    update();
}


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();

    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);
}

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

}

void Scene::paintGL(){

    const qreal retinaScale = devicePixelRatio();
    glViewport(0, 0, width() * retinaScale, height() * retinaScale);
    glClear(GL_COLOR_BUFFER_BIT);

    face ? glEnable(GL_CULL_FACE) : glDisable(GL_CULL_FACE);
```

```
    glCullFace(GL_FRONT);

    QMatrix4x4 matrix, projection;
    QVector3D cam={this->cam_r*cosf(qDegreesToRadians(this-
>cam_angle)),this->cam_y,this->cam_r*sinf(qDegreesToRadians(this-
>cam_angle))};
    Material m=mat_flag ? glnc : mat;

    if(persp)
        projection.perspective(70.0f, 2300.0f/1080.0f, 0.1f, 100.0f);
    else
        projection.ortho(-8,8,-4,4,-20,20);



    matrix.lookAt(cam,{0.0f,1.5f,0.0f},{0,1,0});
    //matrix.rotate(100.0f * m_frame / 300, 0, 1, 0);

    def_sh->bind();
    def_sh->setUniformValue("matrix",projection*matrix);
    def_sh->setUniformValue("normal_m",matrix.normalMatrix());
    def_sh->setUniformValue("modelview",matrix);
    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();

    if(axes) a->drawAxis(def_sh);

    matrix.setToIdentity();

    matrix.lookAt(cam,{0.0f,1.5f,0.0f},{0,1,0});

    matrix.translate(tr_x,tr_y,tr_z);
    matrix.rotate(angle_x,1,0,0);
    matrix.rotate(angle_y,0,1,0);
    matrix.rotate(angle_z,0,0,1);
    matrix.scale({sc_x,sc_y,sc_z});
```

```
    def_sh->bind();
    def_sh->setUniformValue("matrix",projection*matrix);
    def_sh->setUniformValue("normal_m",matrix.normalMatrix());
    def_sh->setUniformValue("modelview",matrix);
    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();


  // dode->drawDode(def_sh,fill ? GL_FILL : GL_LINE, type,
projection*matrix, 1);
    objs->drawObj(def_sh, fill ? GL_FILL : GL_LINE);
    ++m_frame;
}

void Scene::wheelEvent(QWheelEvent *event)
{
    this->cam_r-=event->delta()/500.0f;
    if(this->cam_r<0)this->cam_r+=event->delta()/500.0f;
    update();
}


void Scene::mousePressEvent(QMouseEvent *event){

    start=QPointF(event->x(),event->y());
    if(event->button()==Qt::LeftButton)mouse_flag=true;
    else if(event->button()==Qt::RightButton)mouse_flag=false;
}

void Scene::mouseReleaseEvent(QMouseEvent *event){

}

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

    if(mouse_flag){
```

```
        this->cam_y+=start.y()/100.0f;
        this->cam_angle+=start.x()/10.0f;
    }else{
        this->angle_x+=start.y()/10.0f;
        this->angle_y+=start.x()/10.0f;
    }


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

ПРИЛОЖЕНИЕ В
КЛАСС ОБЪЕКТА

```
#include "sphere_and_conus.h"

Sphere_and_conus::Sphere_and_conus():sectorCount(40),stackCount(40),cente
r({1.5f,1.5f,1.5f}),rad_sphere(1),res_conus(10),rad_conus(2.5)
{
    initSpherePoints();
    initConusPoints();
}

void Sphere_and_conus::drawObj(QOpenGLShaderProgram *m_program, GLenum
type)
{
    glPolygonMode(GL_FRONT_AND_BACK,type);
    drawSphere(m_program);
    drawConus(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::setConusRes(int res)
{
    res_conus=res;
    initConusPoints();
}

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});
        }
    }

    int k1, k2;
    for(uint i = 0; i < stackCount; ++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::initConusPoints()
{
    conus_points.clear();
    float step=2*M_PI/res_conus;

    conus_points.append(QVector3D(0.0f,3.5f,0.0f));
    for(float i=0.0f;i<=M_PI*2;i+=step){

conus_points.append(QVector3D(rad_conus*cosf(i),0.0f,rad_conus*sinf(i)));
    }
    conus_points.append(conus_points[1]);
    conus_normals=conus_points;
}

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->setAttributeValue("colorAttr",{0.1f,0.8f,0.1f,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->release();
}
```

```
void Sphere_and_conus::drawConus(QOpenGLShaderProgram *m_program)
{
    initializeOpenGLFunctions();

    m_program->bind();

    m_program->setAttributeArray(0, conus_points.data());
    m_program->setAttributeArray(2,conus_normals.data());
    m_program->setAttributeValue("colorAttr",{0.1f,0.8f,0.1f,0.1f});

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

    glFrontFace(GL_CW);

    glDrawArrays(GL_POLYGON ,1,conus_points.size()-1);
    glFrontFace(GL_CCW);
    glDrawArrays(GL_TRIANGLE_FAN ,0,conus_points.size());

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