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

ОТЧЕТ

по лабораторной работе №7
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 разработанной вами трехмерной сцены с добавлением возможности использования различных видов источников света, используя предложенные функции OpenGL (матрицы видового преобразования, проецирование, модель освещения, типы источников света, свойства материалов).

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

Задание.

Разработать программу, реализующую представление разработанной вами трехмерной сцены с добавлением возможности формирования различного типа проекций теней, используя предложенные функции OpenGL.

Общие сведения.

В OpenGL используются как основные три системы координат: левосторонняя, правосторонняя и оконная.

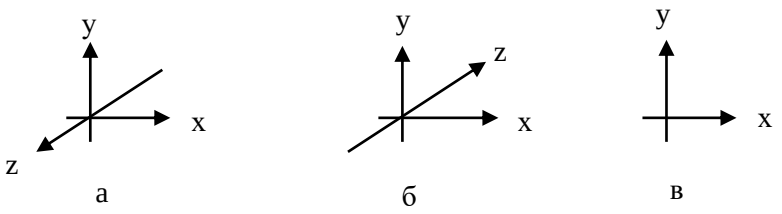

Рис..1. Системы координат в OpenGL: а) правосторонняя система; б) левосторонняя система; в) оконная система

Первые две системы являются трехмерными и отличаются друг от друга направлением оси z: в правосторонней она направлена на наблюдателя, в левосторонней – в глубину экрана. Ось x направлена вправо относительно наблюдателя, ось y – вверх.

Левосторонняя система используется для задания значений параметрам команды gluPerspective(), glOrtho(), которые будут рассмотрены в пункте 0. Правосторонняя система координат используется во всех остальных случаях. Отображение трехмерной информации происходит в двумерную оконную систему координат.

Строго говоря, OpenGL позволяет путем манипуляций с матрицами моделировать как правую, так и левую систему координат. Но на данном этапе лучше пойти простым путем и запомнить: основной системой координат OpenGL является правосторонняя система.

1. Работа с матрицами

Для задания различных преобразований объектов сцены в OpenGL используются операции над матрицами, при этом различают три типа матриц: модельно-видовая, матрица проекций и матрица текстуры. Все они имеют размер 4x4. Видовая матрица определяет преобразования объекта в мировых координатах, такие как параллельный перенос, изменение масштаба и поворот. Матрица проекций определяет, как будут проецироваться трехмерные объекты

на плоскость экрана (в оконные координаты), а матрица текстуры определяет наложение текстуры на объект.

Умножение координат на матрицы происходит в момент вызова соответствующей команды OpenGL, определяющей координату (как правило, это команда glVertex*)

Для того чтобы выбрать, какую матрицу надо изменить, используется команда: void glMatrixMode (GLenum mode), вызов которой со значением параметра mode равным GL_MODELVIEW, GL_PROJECTION или GL_TEXTURE включает режим работы с модельно-видовой матрицей, матрицей проекций, или матрицей текстуры соответственно. Для вызова команд, задающих матрицы того или иного типа, необходимо сначала установить соответствующий режим.

Для определения элементов матрицы текущего типа вызывается команда void glLoadMatrix[f d] (GLtype *m), где m указывает на массив из 16 элементов типа float или double в соответствии с названием команды, при этом сначала в нем должен быть записан первый столбец матрицы, затем второй, третий и четвертый. Еще раз обратим внимание: в массиве m матрица записана по столбцам.

Команда void glLoadIdentity (void) заменяет текущую матрицу на единичную.

Часто бывает необходимо сохранить содержимое текущей матрицы для дальнейшего использования, для чего применяются команды void glPushMatrix (void) и void glPopMatrix (void). Они записывают и восстанавливают текущую матрицу из стека, причем для каждого типа матриц стек свой. Для модельно-видовых матриц его глубина равна как минимум 32, для остальных – как минимум 2.

Для умножения текущей матрицы на другую матрицу используется команда void glMultMatrix[f d] (GLtype *m), где параметр m должен задавать матрицу размером 4x4. Если обозначить текущую матрицу за M, передаваемую матрицу за T, то в результате выполнения команды glMultMatrix текущей

становится матрица M * T. Однако обычно для изменения матрицы того или иного типа удобно использовать специальные команды, которые по значениям своих параметров создают нужную матрицу и умножают ее на текущую.

В целом, для отображения трехмерных объектов сцены в окно приложения используется последовательность, показанная на рисунке 2.

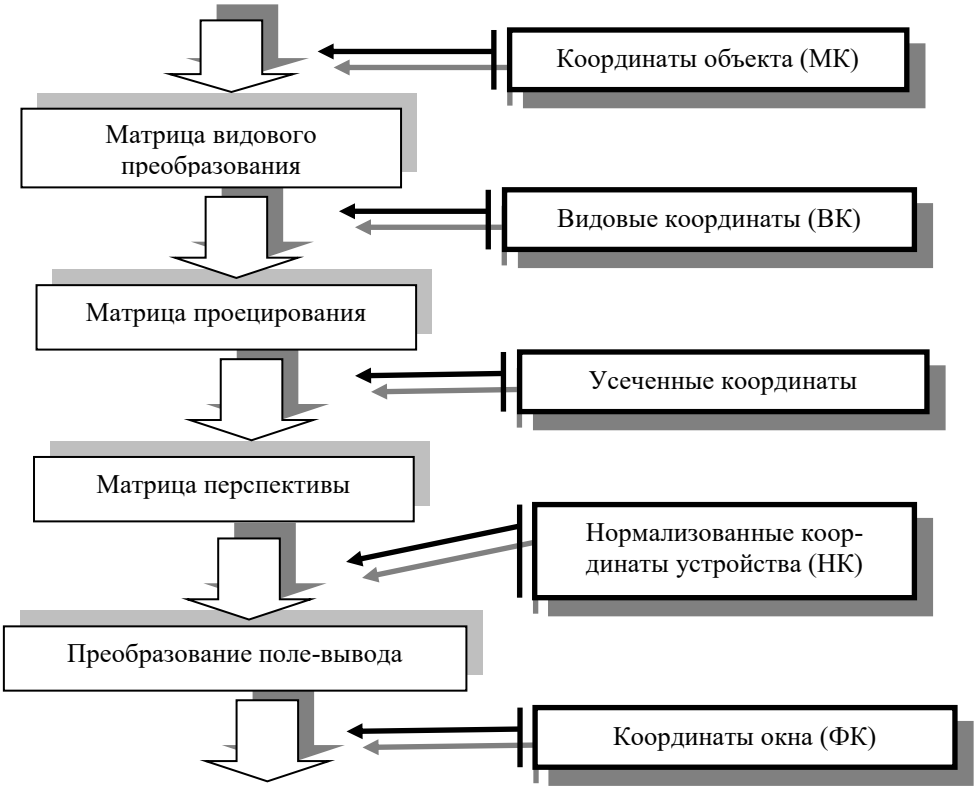

Рис.2. Преобразования координат в OpenGL

2. Модельно-видовые преобразования

К модельно-видовым преобразованиям будем относить перенос, поворот и изменение масштаба вдоль координатных осей. Для проведения этих операций достаточно умножить на соответствующую матрицу каждую вершину объекта и получить измененные координаты этой вершины:

$$(x', y', z', 1)^T = M * (x, y, z, 1)^T$$

где M – матрица модельно-видового преобразования. Перспективное преобразование и проектирование производится аналогично. Сама матрица может быть создана с помощью следующих команд:

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

void glRotate[f d] (GLtype angle, GLtype x, GLtype y, GLtype z);

void glScale[f d] (GLtype x, GLtype y, GLtype z).

glTranlsate*() производит перенос объекта, прибавляя к координатам его вершин значения своих параметров.

glRotate*() производит поворот объекта против часовой стрелки на угол angle (измеряется в градусах) вокруг вектора (x,y,z).

glScale*() производит масштабирование объекта (сжатие или растяжение) вдоль вектора (x,y,z), умножая соответствующие координаты его вершин на значения своих параметров.

Все эти преобразования изменяют текущую матрицу, а поэтому применяются к примитивам, которые определяются позже. В случае, если надо, например, повернуть один объект сцены, а другой оставить неподвижным, удобно сначала сохранить текущую видовую матрицу в стеке командой glPushMatrix(), затем вызвать glRotate() с нужными параметрами, описать примитивы, из которых состоит этот объект, а затем восстановить текущую матрицу командой glPopMatrix().

Кроме изменения положения самого объекта, часто бывает необходимо изменить положение наблюдателя, что также приводит к изменению модельно-видовой матрицы. Это можно сделать с помощью команды void gluLookAt (GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz), где точка (eyex,eyey,eyez) определяет точку наблюдения, (centerx, centery, centerz) задает центр сцены, который будет проектироваться в центр области вывода, а вектор (upx,upy,upz) задает положительное направление оси y, определяя поворот камеры. Если, например, камеру не надо поворачивать, то задается значение (0,1,0), а со значением (0,-1,0) сцена будет перевернута.

Строго говоря, эта команда совершает перенос и поворот объектов сцены, но в таком виде задавать параметры бывает удобнее. Следует отметить, что вызывать команду gluLookAt() имеет смысл перед определением преобразований объектов, когда модельно-видовая матрица равна единичной.

В общем случае матричные преобразования в OpenGL нужно записывать в обратном порядке. Например, если вы хотите сначала повернуть объект, а затем передвинуть его, сначала вызовите команду glTranslate(), а только потом – glRotate(). Ну а после этого определяйте сам объект.

3. Проекции

В OpenGL существуют стандартные команды для задания ортографической (параллельной) и перспективной проекций. Первый тип проекции может быть задан командами

void glOrtho (GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far) и void gluOrtho2D (GLdouble left, GLdouble right, GLdouble bottom, GLdouble top);

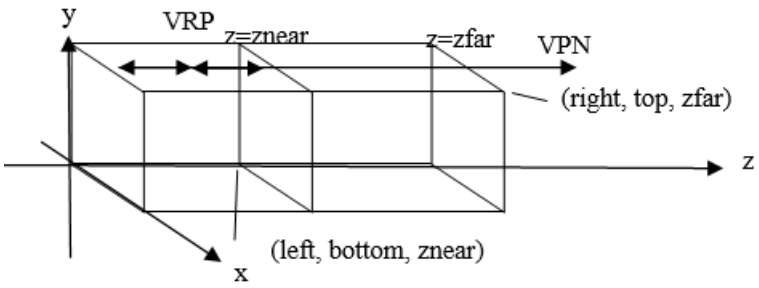

Рис. 3. Ортографическая проекция

Первая команда создает матрицу проекции в усеченный объем видимости (параллелепипед видимости) в левосторонней системе координат. Параметры команды задают точки (left, bottom, znear) и (right, top, zfar), которые отвечают левому нижнему и правому верхнему углам окна вывода. Параметры near и far задают расстояние до ближней и дальней плоскостей отсечения по удалению от точки (0,0,0) и могут быть отрицательными.

Во второй команде, в отличие от первой, значения near и far устанавливаются равными –1 и 1 соответственно. Это удобно, если OpenGL используется для рисования двумерных объектов. В этом случае положение вершин можно задавать, используя команды glVertex2*().

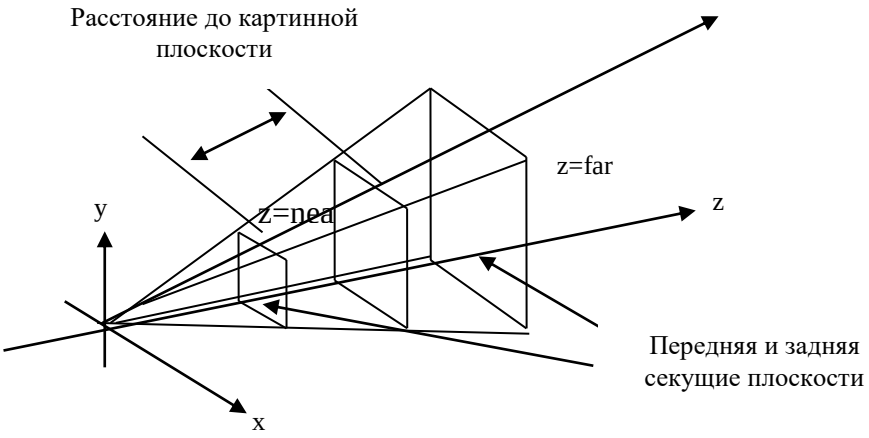

Рис. 4. Перспективная проекция

Перспективная проекция определяется командой void gluPerspective (GLdouble angley, GLdouble aspect, GLdouble znear, GLdouble zfar) , которая задает усеченный конус видимости в левосторонней системе координат. Параметр angley определяет угол видимости в градусах по оси y и должен находиться в диапазоне от 0 до 180. Угол видимости вдоль оси x задается параметром aspect, который обычно задается как отношение сторон области вывода (как правило, размеров окна) Параметры zfar и znear задают расстояние от наблюдателя до плоскостей отсечения по глубине и должны быть положительными. Чем больше отношение zfar/znear, тем хуже в буфере глубины будут различаться расположенные рядом поверхности, так как по умолчанию в него будет записываться ‘сжатая’ глубина в диапазоне от 0 до 1 (см. п. 0.).

Прежде чем задавать матрицы проекций, не забудьте включить режим работы с нужной матрицей командой glMatrixMode(GL_PROJECTION) и сбросить текущую, вызвав glLoadIdentity(). Например:

```
/* ортографическая проекция */
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0, w, 0, h, -1.0, 1.0);
```

4. Область вывода

После применения матрицы проекций на вход следующего преобразования подаются так называемые усеченные (clipped) координаты. Затем находятся нормализованные координаты вершин по формуле:

$(x_n, y_n, z_n)^T = (x_c/w_c , y_c/w_c, z_c/w_c)^T$

Область вывода представляет собой прямоугольник в оконной системе координат, размеры которого задаются командой:

void glViewPort (GLint x, GLint y, GLint width, GLint height)

Значения всех параметров задаются в пикселах и определяют ширину и высоту области вывода с координатами левого нижнего угла (x,y) в оконной системе координат. Размеры оконной системы координат определяются текущими размерами окна приложения, точка (0,0) находится в левом нижнем углу окна.

Используя параметры команды glViewPort(), OpenGL вычисляет оконные координаты центра области вывода (o_x,o_y) по формулам o_x=x+width/2, o_y=y+height/2.

Пусть p_x=width, p_y=height, тогда можно найти оконные координаты каждой вер-

шины:

(xw, yw, zw)T = ((px/2) xn+ ox , (py/2) yn+ oy , [(f-n)/2] zn+(n+f)/2)T

При этом целые положительные величины n и f задают минимальную и максимальную глубину точки в окне и по умолчанию равны 0 и 1 соответственно. Глубина каждой точки записывается в специальный буфер глубины (z-буфер), который используется для удаления невидимых линий и поверхностей. Установить значения n и f можно вызовом функции

void glDepthRange (GLclampd n, GLclampd f);

Команда glViewPort() обычно используется в функции, зарегистрированной с помощью команды glutReshapeFunc(), которая вызывается, если пользователь изменяет размеры окна приложения.

Ход работы.

В результате выполнения работы, к возможностям предыдущей программы была добавлена возможность изменять материал модели с помошью интерфейса пользователя, помимо изменения положения камеры с помощью мыши, была добавлена возможность менять положение камеры с помощью кнопок (параллельно плоскостям, вернуть начальное положение как при запуске программы). Материала показаны на рисунках 1-2.

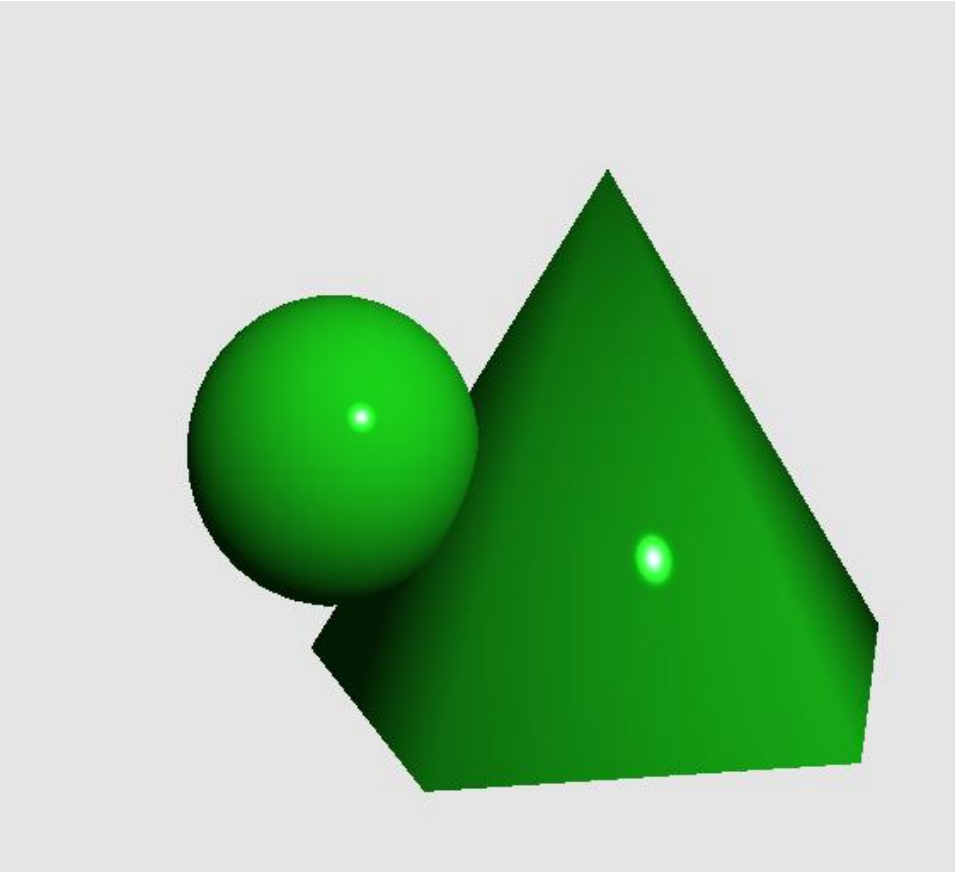

Рисунок 1 - Материал 1

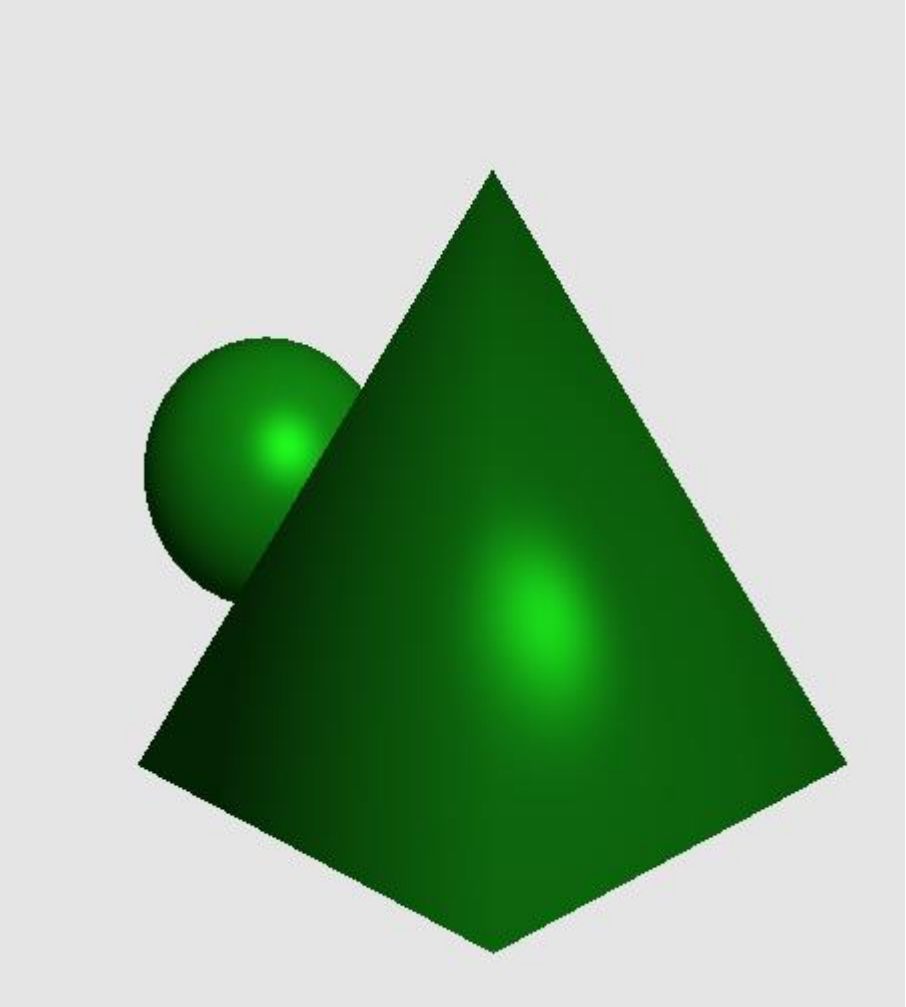

Рисунок 2 - Материал 2

Выводы.

Разработана программа, реализующую представление разработанной вами трехмерной сцены с добавлением возможности использования различных видов источников света, используя предложенные функции OpenGL.

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)
};
}
```