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»

Студентка гр. 7381 _____ Кушкоева А.О.

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.

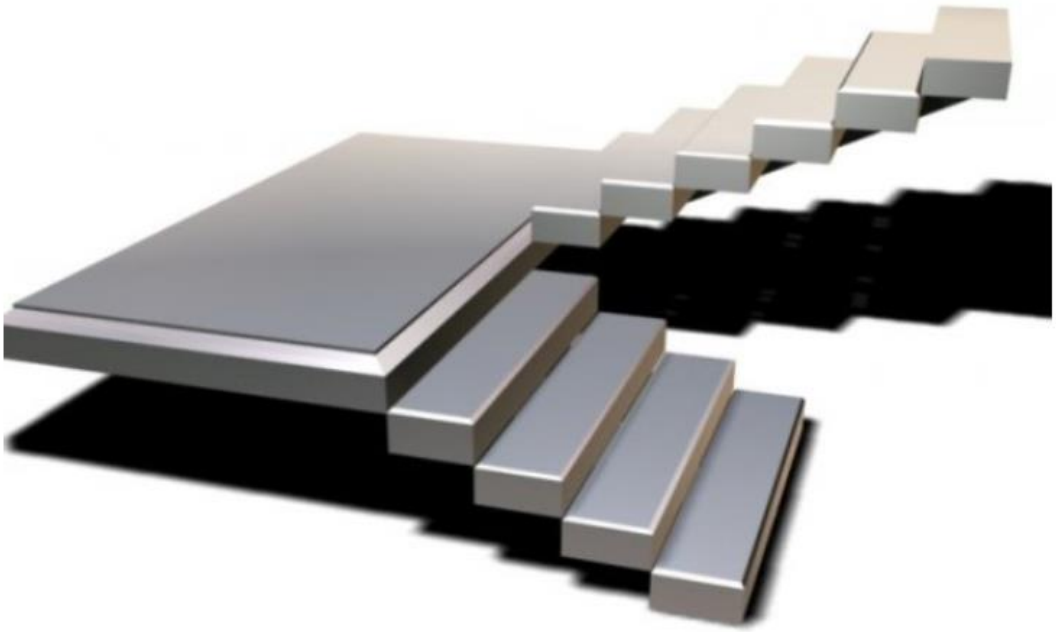
2. Создать описание генерации вашей модели (не создавать в средствах типа Blender, 3D MAX).

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.

Если внимательно проанализировать предложенную сцену, то станет ясно, что лестница состоит из одних и тех же 3d примитивов, а именно куба, к которому применены афинные преобразования. Таким образом задача сводится к моделированию куба, который будет немного изменен, чтобы соответствовать ступеньке. Затем к кубу будет применена итерирующая функция, которая удлинняет его, а затем переносит на вбок и вверх на свою ширину и высоту. В какой-то момент пространство повернется на 180 градусов и сдвинется на длинну ступеньки, и отрисовка продолжится уже для следующего этажа. Между этажами добавлю очень широкий куб.

Алгоритм работы прописан, можно приступать к реализации.

Шаг 1.

Моделирую кубик, у которого верхняя грань будет немного отмасштабирована и перенесена наверх. Затем соединю отсеченные грани полигоном, это создаст эффект среза углов у куба.

Также проинициализирую массив нормалей вершин, массив координат текстур вершин и массив индексации прохода по вершинам. Текстурирую кубик используя фрагментный шейдер и класс QOpenGLTexture, предоставленный библиотекой Qt, который представляет из себя обертку объекта текстуры OpenGL. Полученный текстурированный кубик в каркасном и залитом режимах показан на рисунках 1-2.

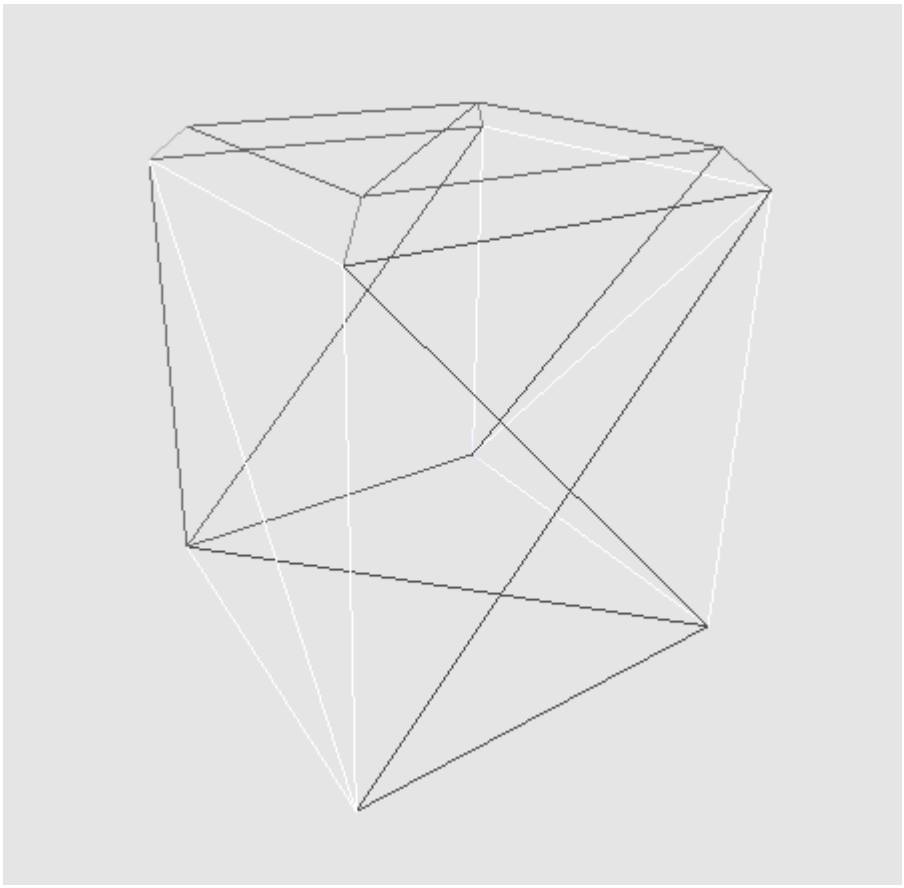


Рисунок 1 - Отрисовка кубика в каркасном режиме

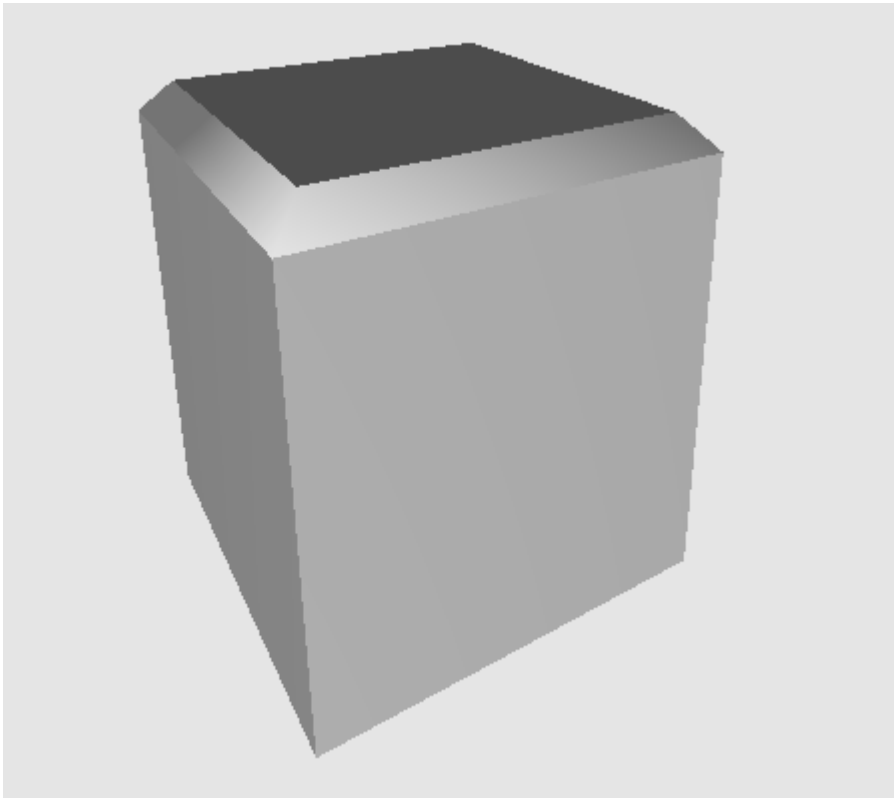


Рисунок 2 - Отрисовка залитого кубика

Шаг 2.

Применение афинных преобразований, преданию кубику формы ступеньки. Для этого к модельной матрице кубика применяется масштабирование по по длинне и высоте. Полученная ступенька показана на рисунке 3.

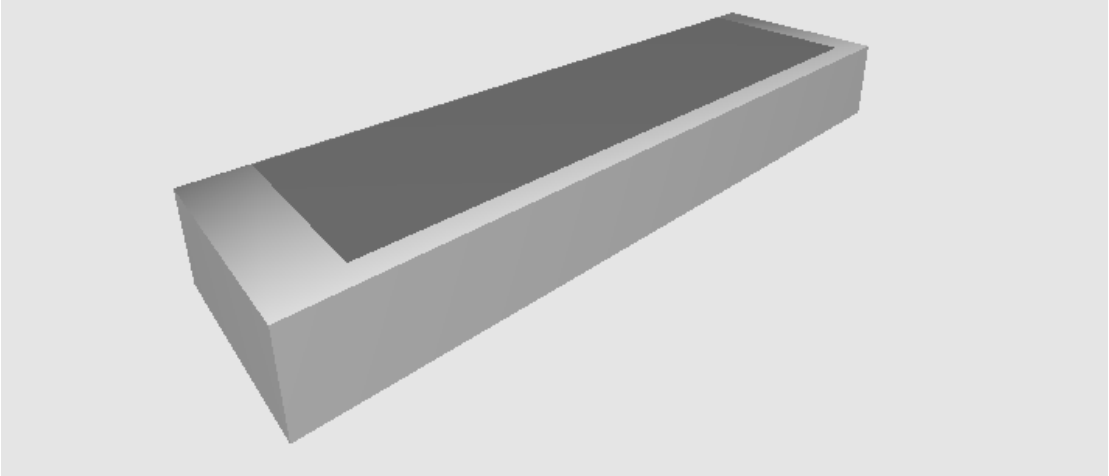


Рисунок 3 - Ступенька

Шаг 4.

Многократно повторяю процесс отрисовки кубика внутри цикла, при этом на каждой итерации изменяя модельную матрицу таким образом, что кубик переносится на высоту и ширину вбок. Процесс напоминает реализацию очень простого фрактала.

В листинге 1. Показан описанный процесс. На рисунке 4 показана получившаяся часть ступеньки.

```
for(int i=0; i<4; i++){     //рисую 4 ступеньки

    m_program->setUniformValue("matrix",projection*view*sc*model); //передаю на вход шейдера
    m_program->setUniformValue("modelview",view*sc*model);          //матрицы преобразования
    m_program->setUniformValue("normal_m",(sc*model).normalMatrix());
    m_program->setUniformValue("model",(sc*model));

    glFrontFace(GL_CW);  //направление обхода вершин
    cube_tex->bind();  //связывание текстуры
    m_program->setUniformValue("texture",0); //передаю текстуру на вход шейдеру
    glDrawElements(GL_TRIANGLE_STRIP,indices.size(),GL_UNSIGNED_SHORT,indices.data()); //рисую грани кубик
    glFrontFace(GL_CCW);
    angle_tex->bind();
    m_program->setUniformValue("texture",0);
    glDrawArrays(GL_QUADS,start_clipped,clipped.size());
                                        //модельная матрица сдвигает пространство так, чтобы следующая
    model.translate(size.x(),size.y(),0);       //ступенька нарисовалась выше и левее
}
```

Листинг 1 - Отрисовка части лестницы

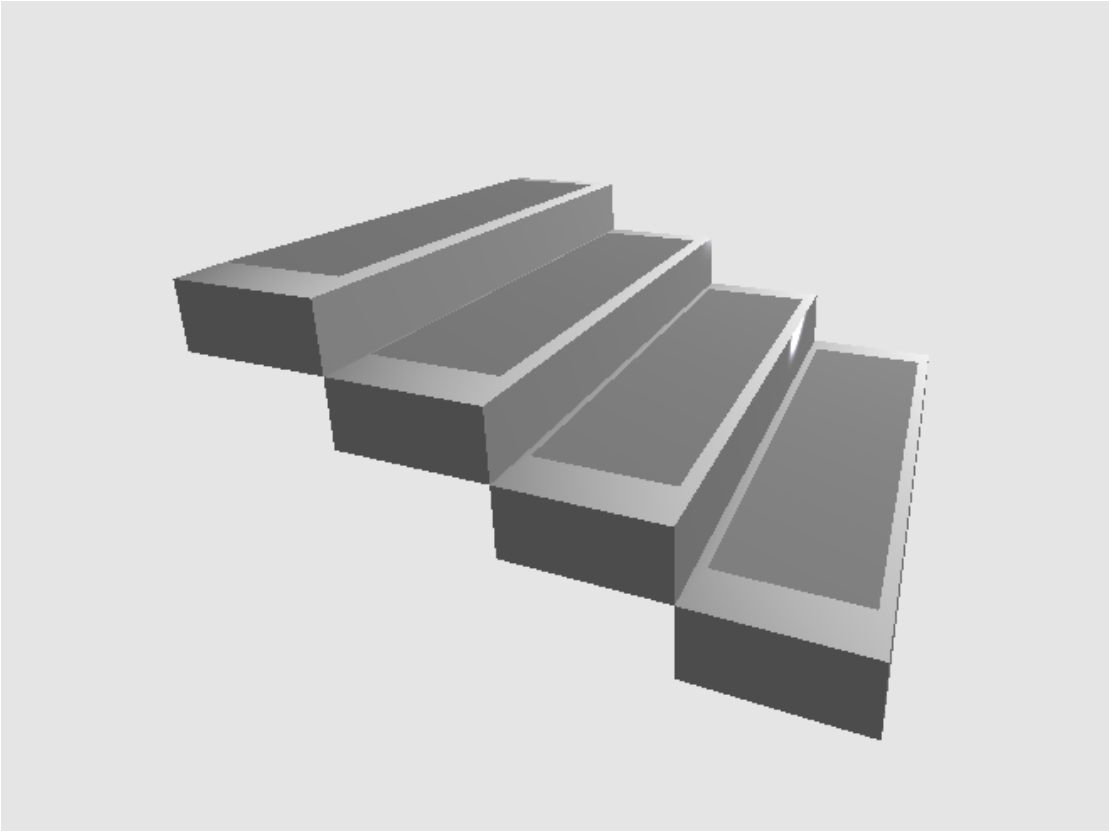


Рисунок 4 - Получившаяся часть ступеньки

Далее сдвигаю совершаю поворот на 180 градусов, сдвиг на длинну ступеньки и снова повторяю цикл, результат на рисунке 5.

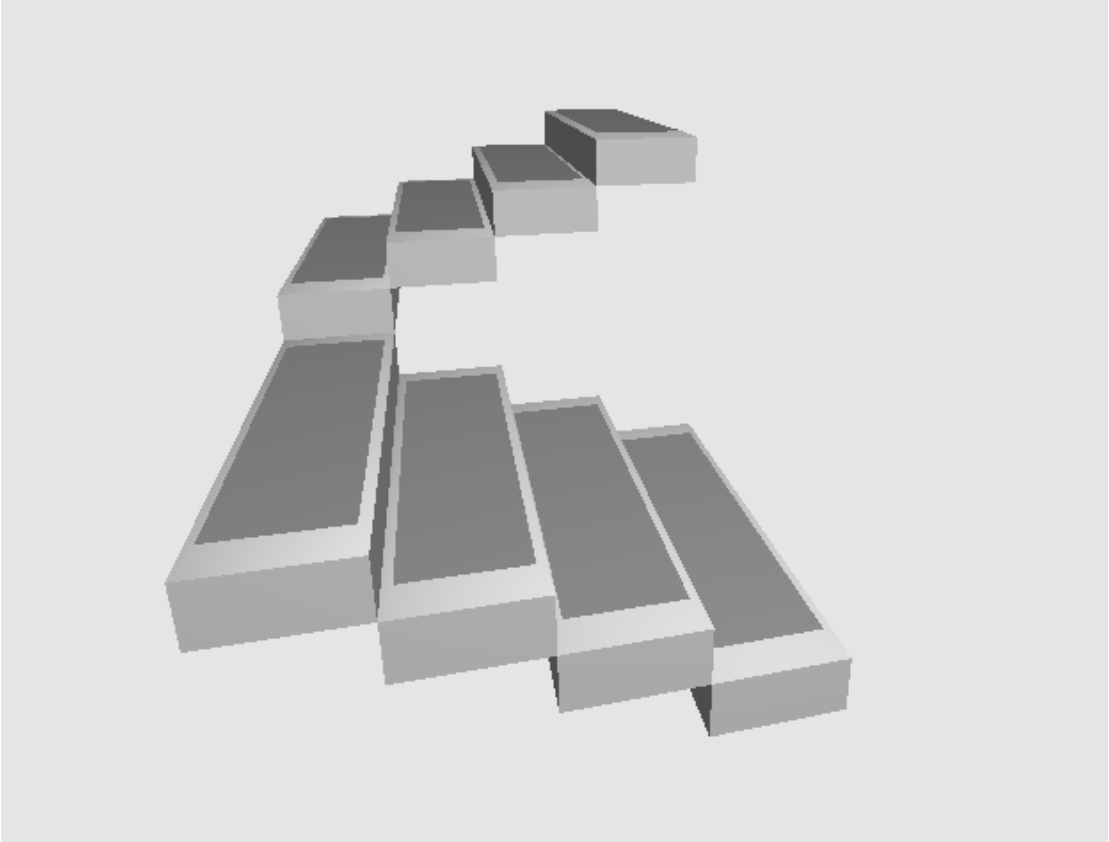


Рисунок 5 - Полученная часть лестницы

Осталось добавить этаж между лестницами, который представляет из себя тот же куб, но отмасштабированный пошире и с двойной длинной. Также добавлю пол, который представляет простой квадрат. Полученная сцена показана на рисунке 6.

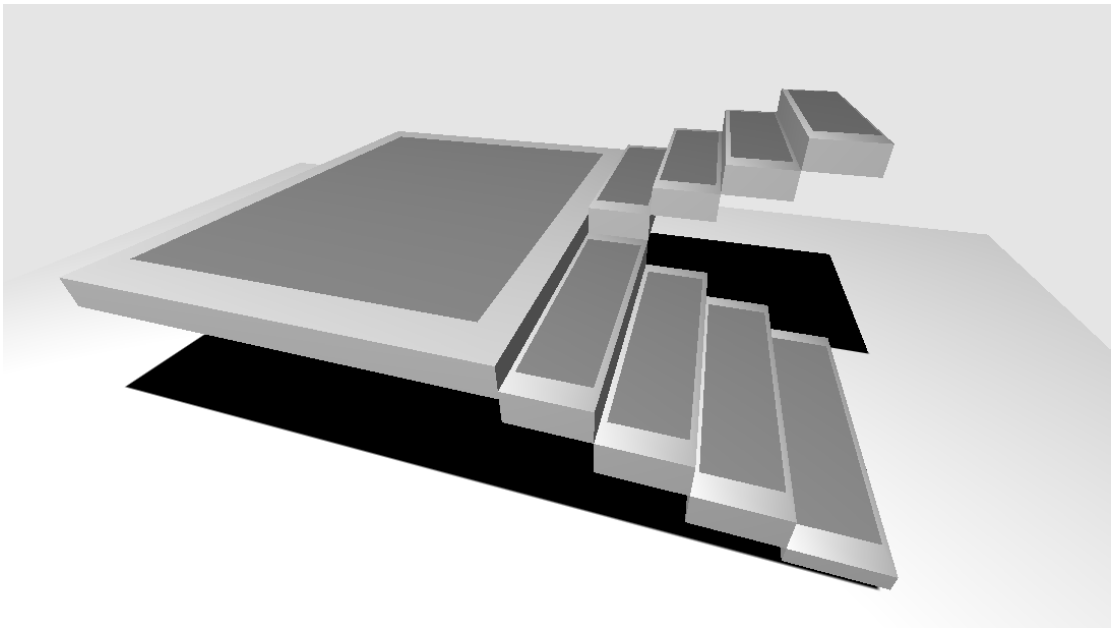


Рисунок 6 - Полученная сцена

Класс камеры:

Для реализации перемещения по сцене разработан класс камеры, который с помощью 2х углов Эйлера, а именно тангажа и рыскания, расчитывает нормальные вектора пространства камеры:

Front, Right, Up. Для того, чтобы переместить камеру достаточно просто умножить нужный вектор на скорость камеры и изменить координату камеры. Чтобы изменить направление взгляда с помощью мыши, расчитывается изменение положения курсора в координатах (x, y) и через сеттеры складываются с углами Эйлера.

В начале каждого кадра класс возвращает матрицу вида через функцию LookAt. Реализацию класса можно найти в приложении.

Примеры сцены с разных положений камеры показаны на рисунках 7-8. Изменение положение источника света реализуется нажатием клавиши. Свет перемещается по окружности вокруг сцены.

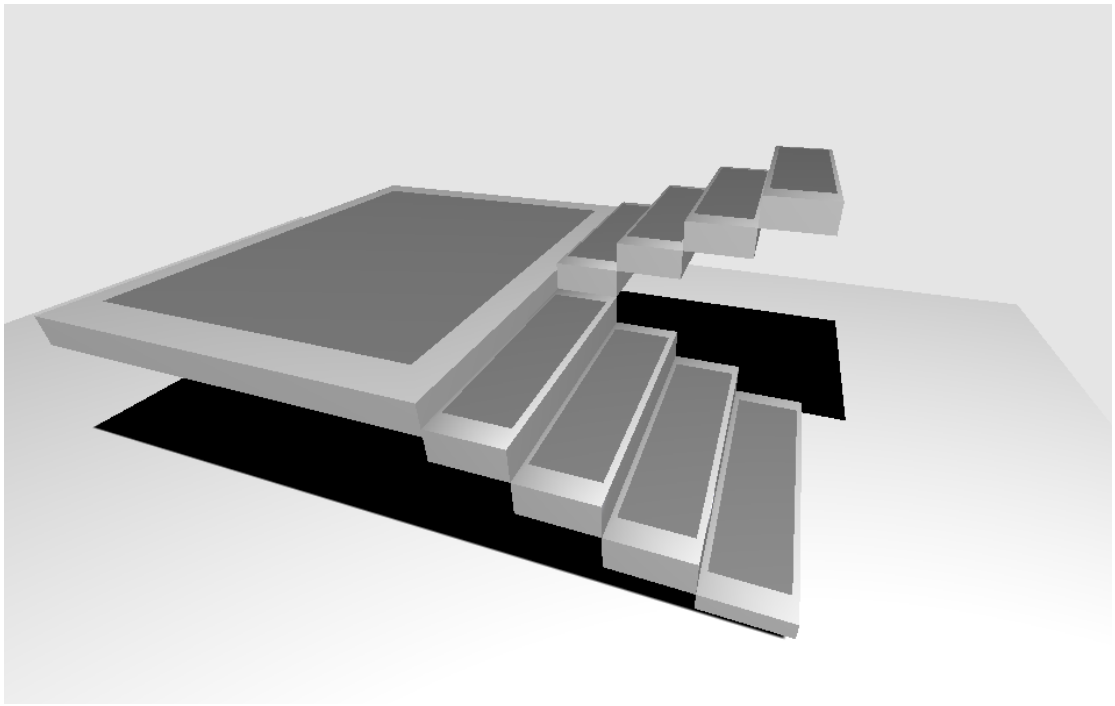


Рисунок 7 - Сцена с разных позиций

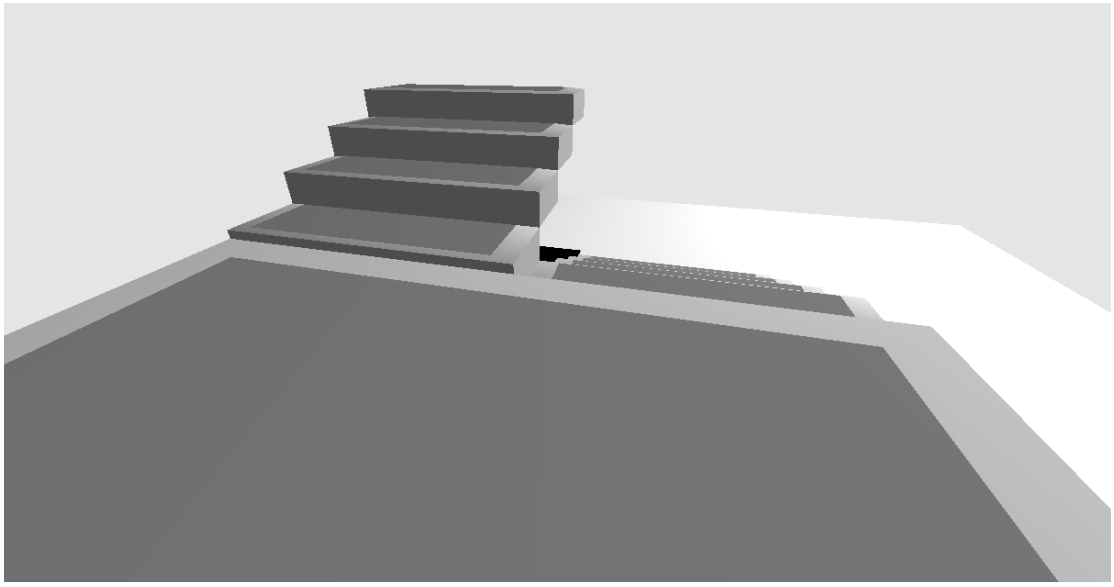


Рисунок 8 - Сцена с разных позиций

Вывод

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

Источники:

1. https://learnopengl.com
2. http://www.opengl-tutorial.org/ru/

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={5*cosf(l_angle/20),3,5*sinf(l_angle/20),0.0f};
    l={l_pos,{0.35f,0.35f,0.35f},{1.0f,1.0f,1.0f},{1.9f,1.9f,1.9f}};
    glnc={{1.7f,1.7f,1.7f},{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();



    def_sh=new QOpenGLShaderProgram;

def_sh->addShaderFromSourceFile(QOpenGLShader::Vertex,":/vShader.glsl
");

def_sh->addShaderFromSourceFile(QOpenGLShader::Fragment,":/fShader.gl
sl");
    def_sh->link();

    fbo_sh=new QOpenGLShaderProgram;
```

```
fbo_sh->addShaderFromSourceFile(QOpenGLShader::Vertex,":/vShader_fbo.glsl");

fbo_sh->addShaderFromSourceFile(QOpenGLShader::Fragment,":/fShader_fbo.glsl");
    fbo_sh->link();

    initTextures();

    QOpenGLFramebufferObjectFormat format;
    format.setAttachment(QOpenGLFramebufferObject::Depth);
    format.setSamples(16);

    glEnable(GL_MULTISAMPLE);

}

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

}

void Scene::paintGL(){


    l_pos={10*cosf(l_angle/20),6,10*sinf(l_angle/20),0.0f};
    l.light_pos=l_pos;
    cam->moveCam(&keys);


 //   glBindBuffer(GL_FRAMEBUFFER,0);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);
    glViewport(0, 0, width(), height());

    glClearColor(0.9f, 0.9f, 0.9f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    def_sh->bind();

//    glBindTexture(GL_TEXTURE_2D,fbo->texture());
```

```
    def_sh->setUniformValue("shadowMap",0);
    def_sh->release();
    drawScene(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;
    }
    if(event->button()==Qt::MidButton){
        fbo->toImage().save("screen.png");
        qDebug()<<event->button();
    }
}

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::drawScene(QOpenGLShaderProgram* m_program, bool from_light)
{
    QMatrix4x4 model, view, projection, lightmatr;
    Material m=glnc;
    if(from_light){
        projection.ortho(-10.0f, 10.0f, -10.0f, 10.0f, 1.0f, 10.0f);

view.lookAt({l.light_pos.x(),l.light_pos.y(),l.light_pos.z()},{0.0f,0
.0f,0.0f},{0.0f,1.0f,0.0f});
    }else{
        projection.perspective(70.0f, 2300.0f/1080.0f, 0.1f, 100.0f);
        view=cam->getMatrix();
        lightmatr.ortho(-10.0f, 10.0f, -10.0f, 10.0f, 1.0f, 10.0f);

lightmatr.lookAt({l.light_pos.x(),l.light_pos.y(),l.light_pos.z()},{0
.0f,0.0f,0.0f},{0.0f,1.0f,0.0f});
        m_program->setUniformValue("lightmatrix",lightmatr);

    }

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

    QVector<QVector3D> points;
    QVector<QVector3D> up_face={
        QVector3D(-1.0f,  1.0f,  1.0f)*0.85+QVector3D(0.0f,0.3f,0.0f),
// v20
        QVector3D( 1.0f,  1.0f,  1.0f)*0.85+QVector3D(0.0f,0.3f,0.0f),
// v21
        QVector3D(-1.0f,  1.0f, -1.0f)*0.85+QVector3D(0.0f,0.3f,0.0f),
// v22
        QVector3D( 1.0f,  1.0f, -1.0f)*0.85+QVector3D(0.0f,0.3f,0.0f)
    };  // v23;
    QVector<QVector2D> tex_coords;
    QVector<QVector3D> normals;
    QVector<GLushort> indices;

    QVector<QVector3D> clipped;
    QVector<QVector3D> clipped_normals;
```

```
QVector<QVector2D> clipped_tex_coords;

QVector<QVector3D> floor;
QVector<QVector3D> floor_normals;
QVector<QVector2D> floor_tex_coords;

points={

    QVector3D(-1.0f, -1.0f,  1.0f),  // v0
    QVector3D( 1.0f, -1.0f,  1.0f), // v1
    QVector3D(-1.0f,  1.0f,  1.0f),  // v2-
    QVector3D( 1.0f,  1.0f,  1.0f), // v3-

    // Vertex data for face 1
    QVector3D( 1.0f, -1.0f,  1.0f), // v4
    QVector3D( 1.0f, -1.0f, -1.0f),  // v5
    QVector3D( 1.0f,  1.0f,  1.0f),  // v6-
    QVector3D( 1.0f,  1.0f, -1.0f), // v7-

    // Vertex data for face 2
    QVector3D( 1.0f, -1.0f, -1.0f), // v8
    QVector3D(-1.0f, -1.0f, -1.0f),  // v9
    QVector3D( 1.0f,  1.0f, -1.0f), // v10-
    QVector3D(-1.0f,  1.0f, -1.0f), // v11-

    // Vertex data for face 3
    QVector3D(-1.0f, -1.0f, -1.0f), // v12
    QVector3D(-1.0f, -1.0f,  1.0f),  // v13
    QVector3D(-1.0f,  1.0f, -1.0f), // v14-
    QVector3D(-1.0f,  1.0f,  1.0f),  // v15-

    // Vertex data for face 4
    QVector3D(-1.0f, -1.0f, -1.0f), // v16
    QVector3D( 1.0f, -1.0f, -1.0f), // v17
    QVector3D(-1.0f, -1.0f,  1.0f), // v18
    QVector3D( 1.0f, -1.0f,  1.0f), // v19


 };

points.append(up_face);
```

```
clipped={
    up_face[0], up_face[1],points[3],points[2],
    up_face[1], up_face[3], points[7], points[3],
    up_face[3], up_face[2], points[11], points[7],
    up_face[2], up_face[0], points[2], points[11]
};



tex_coords={
    QVector2D(0.0f, 0.0f),  // v0
    QVector2D(0.33f, 0.0f), // v1
    QVector2D(0.0f, 0.5f),  // v2
    QVector2D(0.33f, 0.5f), // v3

    // Vertex data for face 1
    QVector2D( 0.0f, 0.5f), // v4
    QVector2D(0.33f, 0.5f), // v5
    QVector2D(0.0f, 1.0f),  // v6
    QVector2D(0.33f, 1.0f), // v7

    // Vertex data for face 2
    QVector2D(0.66f, 0.5f), // v8
    QVector2D(1.0f, 0.5f),  // v9
    QVector2D(0.66f, 1.0f), // v10
    QVector2D(1.0f, 1.0f),  // v11

    // Vertex data for face 3
    QVector2D(0.66f, 0.0f), // v12
    QVector2D(1.0f, 0.0f),  // v13
    QVector2D(0.66f, 0.5f), // v14
    QVector2D(1.0f, 0.5f),  // v15

    // Vertex data for face 4
    QVector2D(0.33f, 0.0f), // v16
    QVector2D(0.66f, 0.0f), // v17
    QVector2D(0.33f, 0.5f), // v18
    QVector2D(0.66f, 0.5f), // v19

    // Vertex data for face 5
```

```
        QVector2D(0.33f, 0.5f), // v20
        QVector2D(0.66f, 0.5f), // v21
        QVector2D(0.33f, 1.0f), // v22
        QVector2D(0.66f, 1.0f)


    };

    indices={
        0,  1,  2,  3,  3,      //Face 0 - triangle strip ( v0,  v1,  v2,
v3)
        4,  4,  5,  6,  7,  7, //Face 1 - triangle strip ( v4,  v5,  v6,
v7)
        8,  8,  9, 10, 11, 11, //Face 2 - triangle strip ( v8,  v9, v10,
v11)
       12, 12, 13, 14, 15, 15, //Face 3 - triangle strip (v12, v13, v14,
v15)
       16, 16, 17, 18, 19, 19, //Face 4 - triangle strip (v16, v17, v18,
v19)
       20, 20, 21, 22, 23      //Face 5 - triangle strip (v20, v21, v22,
v23)
   };

    normals={
        QVector3D(0.0f,0.0f,  1.0f),  // v0
        QVector3D(0.0f,0.0f,  1.0f),  // v0
        QVector3D(0.0f,0.0f,  1.0f),  // v0
        QVector3D(0.0f,0.0f,  1.0f),  // v0

        QVector3D( 1.0f, 0.0f,  0.0f), // v4
        QVector3D( 1.0f, 0.0f, 0.0f),  // v5
        QVector3D( 1.0f,  0.0f,  0.0f),  // v6-
        QVector3D( 1.0f,  0.0f, 0.0f), // v7-

        QVector3D( 0.0f, 0.0f, -1.0f), // v8
        QVector3D( 0.0f, 0.0f, -1.0f), // v8
        QVector3D( 0.0f, 0.0f, -1.0f), // v8
        QVector3D( 0.0f, 0.0f, -1.0f), // v8

        // Vertex data for face 3
        QVector3D(-1.0f, 0.0f, 0.0f), // v12
```

```
    QVector3D(-1.0f, 0.0f,  0.0f),  // v13
    QVector3D(-1.0f, 0.0f, 0.0f), // v14-
    QVector3D(-1.0f, 0.0f,  0.0f),  // v15-

    QVector3D(0.0, -1.0f, 0.0f), // v16
    QVector3D( 0.0f, -1.0f, 0.0f), // v17
    QVector3D(0.0f, -1.0f,  0.0f), // v18
    QVector3D( 0.0f, -1.0f,  0.0f), // v19

    QVector3D(0.0f, 1.0f, 0.0f), // v16
    QVector3D(0.0f, 1.0f, 0.0f), // v16
    QVector3D(0.0f, 1.0f, 0.0f), // v16
    QVector3D(0.0f, 1.0f, 0.0f), // v16


};


floor={
    {1.0f,0.0f,1.0f},
    {1.0f,0.0f,-1.0f},
    {-1.0,0.0f,-1.0f},
    {-1.0f,0.0f,1.0f}
};
floor_normals={
    {0.0f,1.0f,0.0f},
    {0.0f,1.0f,0.0f},
    {0.0f,1.0f,0.0f},
    {0.0f,1.0f,0.0f}
};

floor_tex_coords={
    {0.0f,1.0f},
    {1.0f,1.0f},
    {1.0f,0.0f},
    {0.0f,0.0f}
};
clipped_normals=clipped;
clipped_tex_coords={
    {0.0f,0.0f},{1.0f,0.0f},{1.0f,1.0f},{0.0f,1.0f},
    {0.0f,0.0f},{1.0f,0.0f},{1.0f,1.0f},{0.0f,1.0f},
```

```
    {0.0f,0.0f},{1.0f,0.0f},{1.0f,1.0f},{0.0f,1.0f},
    {0.0f,0.0f},{1.0f,0.0f},{1.0f,1.0f},{0.0f,1.0f},
};

int start_clipped=points.size();
points.append(clipped);
normals.append(clipped_normals);
tex_coords.append(clipped_tex_coords);

int start_floor=points.size();
points.append(floor);
normals.append(floor_normals);
tex_coords.append(floor_tex_coords);

m_program->bind();
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);


m_program->setUniformValue("l.position",l.light_pos);
m_program->setUniformValue("l.la",l.la);
m_program->setUniformValue("l.ld",l.ld);
m_program->setUniformValue("l.ls",l.ls);
m_program->setUniformValue("material.ka",m.ka);
m_program->setUniformValue("material.kd",m.kd);
m_program->setUniformValue("material.ks",m.ks);
m_program->setUniformValue("material.Shininess",m.Shininess);



QVector3D size(2.0f,2.0f,2.0f);
model.setToIdentity();
QMatrix4x4 sc,sc1,matr;
sc.scale({0.5f,0.2f,1.8f});
```

```
    for(int i=0; i<4; i++){



m_program->setUniformValue("matrix",projection*view*sc*model);
        m_program->setUniformValue("modelview",view*sc*model);

m_program->setUniformValue("normal_m",(sc*model).normalMatrix());
        m_program->setUniformValue("model",(sc*model));
     /*   handler_tex->bind();
        def_sh->setUniformValue("texture",0);*/



        glFrontFace(GL_CW);
        cube_tex->bind();
        m_program->setUniformValue("texture",0);

glDrawElements(GL_TRIANGLE_STRIP,indices.size(),GL_UNSIGNED_SHORT,ind
ices.data());
        glFrontFace(GL_CCW);
        angle_tex->bind();
        m_program->setUniformValue("texture",0);
        glDrawArrays(GL_QUADS,start_clipped,clipped.size());

        model.translate(size.x(),size.y(),0);
    }

    matr.translate(3*size.x()+0.5,size.y()-0.5,size.z()-0.2);
    matr.scale(3.0f,0.2f,3.6f);

    m_program->setUniformValue("matrix",projection*view*sc1*matr);
    m_program->setUniformValue("modelview",view*sc1*matr);
    m_program->setUniformValue("normal_m",(sc1*matr).normalMatrix());
     m_program->setUniformValue("model",(sc*model));
    /*   handler_tex->bind();
```

```
    def_sh->setUniformValue("texture",0);*/

    glFrontFace(GL_CW);
    cube_tex->bind();
    m_program->setUniformValue("texture",0);

glDrawElements(GL_TRIANGLE_STRIP,indices.size(),GL_UNSIGNED_SHORT,ind
ices.data());
    glFrontFace(GL_CCW);
    angle_tex->bind();
    m_program->setUniformValue("texture",0);
    glDrawArrays(GL_QUADS,start_clipped,clipped.size());

    model.translate(0.0f,0.0f,size.z());
    model.rotate(180.0f,{0.0f,1.0f,0.0f});
    model.translate(size.x(),0.0f,0.0f);
    for(int i=0; i<4; i++){




m_program->setUniformValue("matrix",projection*view*sc*model);
        m_program->setUniformValue("modelview",view*sc*model);

m_program->setUniformValue("normal_m",(sc*model).normalMatrix());
         m_program->setUniformValue("model",(sc*model));
        glFrontFace(GL_CW);
        cube_tex->bind();
        m_program->setUniformValue("texture",0);

glDrawElements(GL_TRIANGLE_STRIP,indices.size(),GL_UNSIGNED_SHORT,ind
ices.data());
        glFrontFace(GL_CCW);
        angle_tex->bind();
        m_program->setUniformValue("texture",0);
        glDrawArrays(GL_QUADS,start_clipped,clipped.size());

        model.translate(size.x(),size.y(),0);
    }

    model.setToIdentity();
```

```
    model.scale(20.0f,3.0f,4.0f);
    model.translate(0.5f,0.0f,0.0f);
    m_program->setUniformValue("matrix",projection*view*sc*model);
    m_program->setUniformValue("modelview",view*sc*model);
    m_program->setUniformValue("normal_m",(sc*model).normalMatrix());
     m_program->setUniformValue("model",(sc*model));
     glFrontFace(GL_CW);
     floor_tex->bind();
     m_program->setUniformValue("texture",0);
     glDrawArrays(GL_QUADS,start_floor,floor.size());




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

QOpenGLTexture *Scene::initTexture(QImage img)
{
    QOpenGLTexture * texture = new QOpenGLTexture(img.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(){
    angle_tex=initTexture(":/cube.bmp");
```

```
    cube_tex=initTexture(":/angle.bmp");
    floor_tex=initTexture(":/floor.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);
}
```