МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ   
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ**

**ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА»**

**(БГТУ им. В.Г. Шухова)**

Кафедра программного обеспечения вычислительной техники и автоматизированных систем

Лабораторная работа №6

по дисциплине: Компьютерная графика

тема: «Знакомство с библиотекой OpenGL»

Выполнил: ст. группы ВТ-32

Воскобойников И. С.

Проверил: Гибкин Ю. С.

Белгород 2020 г.

Цель работы: изучение функций OpenGL для построения трёхмерных объектов в среде Qt Creator.

Порядок выполнения работы

1. Изучить функции OpenGL для задания формата вершин, цветов, векторов нормалей; выполнения матричных преобразований.
2. Выбрать и согласовать с преподавателем предметную область для дальнейшего выполнения лабораторных работ. Примеры предметных областей:
   1. Сцена для игры (настольная игра, лего, кубики или др.);
   2. Трёхмерные модели реальных объектов различной величины;
   3. Простейшая трёхмерная компьютерная игра (симулятор гонок, стратегия или другой жанр);
   4. Моделирование физического процесса (поверхность воды, распространение волн, столкновение упругих шаров);
   5. Визуализация научных вычислений (графики в трёхмерном пространстве, тепловые карты);
   6. Геометрические задачи в OpenGL, например:
      1. Сортировка прозрачных объектов по глубине,
      2. Нахождение пересечения селектирующего луча с объектами на сцене,
      3. Отсечение невидимых объектов с использованием пирамиды видимости;
   7. Изучение физического движка Havok Physics.
3. Разработать алгоритм и написать программу для создания трёхмерной сцены, содержащей графические объекты, в соответствии с выбранной предметной областью.
4. Разрешается написание одной сложной программы в группе из нескольких человек (не более трёх) по согласованию с преподавателем. Программа в данном случае должна быть разбита на модули, каждый из которых разрабатывает отдельный программист. Например, один программист создаёт эффекты (освещение, текстурирование), второй занимается видовыми преобразованиями и проектированием, третий – создаёт модели.

Требования к программе

1. Графические объекты должны быть изображены на экране в виде набора закрашенных полигонов. В памяти объекты необходимо хранить в виде массива многоугольников. На сцене должно быть не менее 10 различных объектов.
2. Предоставить пользователю возможность перемещать, поворачивать, масштабировать объекты сцены с использованием клавиш и мыши, а также изменять положение камеры (наблюдателя).

**Camera.h**

#ifndef CAMERA\_H

#define CAMERA\_H

#include <QQuaternion>

#include <QMatrix4x4>

class **Camera**{

QQuaternion rotate;

QQuaternion rotate\_X;

QQuaternion rotate\_Y;

QVector3D trans;

QMatrix4x4 viewMatrix;

public:

**Camera**();

void **rotateX**(const QQuaternion& r);

void **rotateY**(const QQuaternion& r);

void **translate**(const QVector3D& vec);

void **updateViewMatrix**();

const QMatrix4x4 &**getviewMatrix**() const;

void **setDefault**();

};

#endif // CAMERA\_H

**Camera.c**

#include "camera.h"

Camera::**Camera**(){

setDefault();

}

void Camera::**rotateX**(const QQuaternion &r){

rotate\_X = r \* rotate\_X;

rotate = rotate\_Y \* rotate\_X;

updateViewMatrix();

}

void Camera::**rotateY**(const QQuaternion &r){

rotate\_Y = r \* rotate\_Y;

rotate = rotate\_X \* rotate\_Y;

updateViewMatrix();

}

void Camera::**translate**(const QVector3D &vec){

trans += vec;

updateViewMatrix();

}

void Camera::**updateViewMatrix**(){

viewMatrix.setToIdentity();

viewMatrix.translate(trans);

viewMatrix.rotate(rotate);

}

const QMatrix4x4 &Camera::**getviewMatrix**() const{

return viewMatrix;

}

void Camera::**setDefault**(){

rotate = QQuaternion::fromAxisAndAngle(QVector3D(1.0, 0.0, 0.0), -55);

rotate\_X = QQuaternion::fromAxisAndAngle(QVector3D(1.0, 0.0, 0.0), -55);

rotate\_Y = QQuaternion::fromAxisAndAngle(QVector3D(0.0, 0.0, 0.0), 0);

trans = QVector3D(0.0, 0.0, -200.0);

updateViewMatrix();

}

**import\_obj.h**

#ifndef IMPORT\_OBJ\_H

#define IMPORT\_OBJ\_H

#include <QString>

#include <QFile>

#include <QDebug>

#include <QVector3D>

#include <QVector2D>

struct **VertexData**{

**VertexData**(){}

**VertexData**(QVector3D p, QVector2D t, QVector3D n):

position(p), texCoord(t), normal(n){}

**VertexData**(QVector3D p, QVector3D n):

position(p), normal(n){}

**VertexData**(QVector3D p, QVector2D t):

position(p), texCoord(t){}

**VertexData**(QVector3D p):

position(p){}

QVector3D position;

QVector2D texCoord;

QVector3D normal;

};

void **load\_obj**(const QString &f\_name, QVector<VertexData> &v\_data, QVector<uint> &indexes);

#endif // IMPORT\_OBJ\_H

**import\_obj.c**

#include "import\_obj.h"

//загружает .obj модель файла

void **load\_obj**(const QString &f\_name, QVector<VertexData> &v\_data, QVector<uint> &indexes){

v\_data.clear();

QFile obj\_f(f\_name);

if (!obj\_f.exists()){

qDebug() << "Файл " << f\_name << " не найден!";

return;

}

obj\_f.*open*(QIODevice::ReadOnly);

QTextStream input(&obj\_f);

QString str;

QStringList list, list\_f;

QVector<QVector3D> vert;

QVector<QVector2D> tex\_coord;

QVector<QVector3D> normals;

while(!input.atEnd()){

str = input.readLine();

list = str.split(" ");

if (list[0] == "v"){

vert.append(QVector3D(list[1].toFloat(), list[2].toFloat(), list[3].toFloat()));

continue;

}

if (list[0] == "vt"){

tex\_coord.append(QVector2D(list[1].toFloat(), list[2].toFloat()));

continue;

}

if (list[0] == "vn"){

normals.append(QVector3D(list[1].toFloat(), list[2].toFloat(), list[3].toFloat()));

continue;

}

if (list[0] == "f"){

for (int i = 1; i <= 3; i++){

list\_f = list[i].split("/");

if (list\_f[1].size() == 0 && list\_f[2].size() == 0)

v\_data.append(VertexData(vert[list\_f[0].toLong()-1]));

if (list\_f[1].size() == 0 && list\_f[2].size() != 0)

v\_data.append(VertexData(vert[list\_f[0].toLong()-1], normals[list\_f[2].toLong()-1]));

if (list\_f[1].size() != 0 && list\_f[2].size() == 0)

v\_data.append(VertexData(vert[list\_f[0].toLong()-1], tex\_coord[list\_f[1].toLong()-1]));

if (list\_f[1].size() != 0 && list\_f[2].size() != 0)

v\_data.append(VertexData(vert[list\_f[0].toLong()-1], tex\_coord[list\_f[1].toLong()-1], normals[list\_f[2].toLong()-1]));

indexes.append((uint)indexes.size());

}

}

}

obj\_f.*close*();

}

**Main.c**

#include "mainwindow.h"

#include <QApplication>

int **main**(int argc, char \*argv[])

{

QApplication a(*argc*, argv);

MainWindow w;

w.showFullScreen();

return a.exec();

}

**simpleobject.h**

#ifndef MODEL3D\_H

#define MODEL3D\_H

#define TEAM\_WHITE 1

#define TEAM\_BLACK 0

#include <import\_obj.h>

#include <QMatrix4x4>

#include <QOpenGLBuffer>

#include <QOpenGLFunctions>

#include <QOpenGLShaderProgram>

#include <QOpenGLTexture>

#include <QtMath>

class **SimpleObject**{

protected:

QMatrix4x4 modelMatrix;

QOpenGLBuffer arrayBuffer;

QOpenGLBuffer indexBuffer;

QOpenGLTexture \*texture;

QVector4D color;

QVector3D min, max, center;

QVector3D trans;

float k, r\_x, r\_y, r\_z;

bool use\_texture = false;

public:

**SimpleObject**();

**SimpleObject**(const QString &name, QVector4D color = QVector4D(0.2f, 0.4f, 0.6f, 1.0f));

~**SimpleObject**();

void **init**(const QString &name, QVector4D color = QVector4D(0.2f, 0.4f, 0.6f, 1.0f));

virtual void ***drawObject***(QOpenGLShaderProgram \*sh\_program, QOpenGLFunctions \*functions);

void **scale**(float k);

void **translate**(const QVector3D &vec);

void **rotateX**(float angle);

void **rotateY**(float angle);

void **rotateZ**(float angle);

void **updateModelMatrix**();

QVector3D **getMin**(){return min;}

QVector3D **getMax**(){return max;}

QVector3D **getCenterOfMass**(){return QVector3D(modelMatrix\*center);}

QVector4D **getColor**(){return color;}

void **setTexture**(QString name);

bool **getUseTexture**(){return use\_texture;}

};

class **ChessFigure**: public SimpleObject{

int pos\_x, pos\_y;

int grid\_sz, field\_sz;

bool enable;

int team;

QVector4D enableColor, defaultColor;

public:

**ChessFigure**(const QString &name, int grid\_sz, int field\_sz, int x, int y, int team);

void **move**(int x1, int y1);

bool **onThisPos**(int x, int y);

void **select**();

void **deselect**();

int **getTeam**();

};

struct **PointLightStruct**{

QVector3D pos;

QVector4D color;

float power;

float constant;

float linear;

float quadratic;

};

class **LampObject**: public SimpleObject{

bool turn\_on;

QVector4D base\_color, light\_color;

float power;

float constant, linear, quadratic; // коэффициенты затухания

public:

**LampObject**(const QString &name, QVector4D base\_color = QVector4D(0.6f, 0.5f, 0.2f, 1.0f), QVector4D light\_color = QVector4D(1.0f, 1.0f, 1.0f, 1.0f));

void **setState**(bool state);

bool **getState**(){return turn\_on;}

void **setPower**(float power){this->power = power;}

PointLightStruct **getPointLightStruct**();

void ***drawObject***(QOpenGLShaderProgram \*sh\_program, QOpenGLFunctions \*functions);

void **drawObjAsLamp**(QOpenGLShaderProgram \*sh\_program, QOpenGLFunctions \*functions);

};

struct **SpotLightStruct**{

QVector3D pos;

QVector3D direction;

QVector4D color;

float power;

float cutOff;

};

class **Spot**{

bool turn\_on;

QVector4D light\_color;

float power;

QVector3D position, direction;

float cutOff; // cos(phi)

public:

**Spot**(QVector4D light\_color = QVector4D(1.0f, 1.0f, 1.0f, 1.0f), bool turn\_on = false):

turn\_on(turn\_on), light\_color(light\_color), power(1){}

void **setState**(bool state){turn\_on = state;}

bool **getState**(){return turn\_on;}

void **setPower**(float power){this->power = power;}

void **setAngle**(float angle){cutOff = qCos(angle\*M\_PI/180);}

SpotLightStruct **getSpotLightStruct**();

void **setPositionAndDirection**(QVector3D pos, QVector3D dir\_point);

};

#endif // MODEL3D\_H

**Simpleobject.cpp**

#include "simpleobject.h"

SimpleObject::**SimpleObject**(){

}

SimpleObject::**SimpleObject**(const QString &name, QVector4D color):

arrayBuffer(), indexBuffer(QOpenGLBuffer::IndexBuffer), texture(0)

{

init(name, color);

}

SimpleObject::~**SimpleObject**(){

if (arrayBuffer.isCreated()) arrayBuffer.destroy();

if (indexBuffer.isCreated()) indexBuffer.destroy();

}

QVector3D **findMinXY**(const QVector<VertexData> &vd){

QVector3D min = vd[0].position;

for (int i = 0; i < vd.size(); i++)

if (vd[i].position.x() <= min.x() && vd[i].position.y() <= min.y())

min = vd[i].position;

return min;

}

QVector3D **findMaxXY**(const QVector<VertexData> &vd){

QVector3D max = vd[0].position;

for (int i = 0; i < vd.size(); i++)

if (vd[i].position.x() >= max.x() && vd[i].position.y() >= max.y())

max = vd[i].position;

return max;

}

QVector3D **findCenter**(const QVector<VertexData> &vd){

float x = 0, y = 0, z = 0;

for (int i = 0; i < vd.size(); i++){

x += vd[i].position.x();

y += vd[i].position.y();

z += vd[i].position.z();

}

return QVector3D(x/vd.size(), y/vd.size(), z/vd.size());

}

void SimpleObject::**init**(const QString &name, QVector4D color){

this->color = color;

if (arrayBuffer.isCreated()) arrayBuffer.destroy();

if (indexBuffer.isCreated()) indexBuffer.destroy();

QVector<VertexData> vertData;

QVector<uint> indexes;

load\_obj(name, *vertData*, *indexes*);

min = findMinXY(vertData);

max = findMaxXY(vertData);

center = findCenter(vertData);

arrayBuffer.create();

arrayBuffer.bind();

arrayBuffer.allocate(vertData.constData(), vertData.size()\*sizeof(VertexData));

arrayBuffer.release();

indexBuffer.create();

indexBuffer.bind();

indexBuffer.allocate(indexes.constData(), indexes.size()\*sizeof(uint));

indexBuffer.release();

modelMatrix.setToIdentity();

k = 1; trans = QVector3D(0,0,0);

r\_x = r\_y = r\_z = 0;

}

void SimpleObject::***drawObject***(QOpenGLShaderProgram \*sh\_program, QOpenGLFunctions \*functions){

if (!arrayBuffer.isCreated() || !indexBuffer.isCreated()) return;

sh\_program->setUniformValue("u\_modelMatrix", modelMatrix);

sh\_program->setUniformValue("u\_objColor", color);

sh\_program->setUniformValue("u\_use\_texture", use\_texture);

int offset = 0;

arrayBuffer.bind();

int vertLoc = sh\_program->attributeLocation("a\_position");

sh\_program->enableAttributeArray(vertLoc);

sh\_program->setAttributeBuffer(vertLoc, GL\_FLOAT, offset, 3, sizeof(VertexData));

offset += sizeof(QVector3D); // смещение для текстурных координат

//текстурные координаты

if (use\_texture){

texture->bind(0);

sh\_program->setUniformValue("u\_texture", 0);

int texLoc = sh\_program->attributeLocation("a\_texCoord");

sh\_program->enableAttributeArray(texLoc);

sh\_program->setAttributeBuffer(texLoc, GL\_FLOAT, offset, 2, sizeof(VertexData));

}

offset += sizeof(QVector2D);

int normalLoc = sh\_program->attributeLocation("a\_normal");

sh\_program->enableAttributeArray(normalLoc);

sh\_program->setAttributeBuffer(normalLoc, GL\_FLOAT, offset, 3, sizeof(VertexData));

indexBuffer.bind();

functions->glDrawElements(GL\_TRIANGLES, indexBuffer.size(), GL\_UNSIGNED\_INT, 0);

arrayBuffer.release();

indexBuffer.release();

}

void SimpleObject::**scale**(float k){

if (abs(k) < 0.0001f) return;

this->k \*= k;

updateModelMatrix();

}

void SimpleObject::**translate**(const QVector3D &vec){

trans += vec;

updateModelMatrix();

}

void SimpleObject::**rotateX**(float angle){

r\_x += angle;

updateModelMatrix();

}

void SimpleObject::**rotateY**(float angle){

r\_y += angle;

updateModelMatrix();

}

void SimpleObject::**rotateZ**(float angle){

r\_z += angle;

updateModelMatrix();

}

void SimpleObject::**updateModelMatrix**(){

modelMatrix.setToIdentity();

modelMatrix.rotate(r\_x, 1, 0, 0);

modelMatrix.rotate(r\_y, 0, 1, 0);

modelMatrix.rotate(r\_z, 0, 0, 1);

modelMatrix.translate(trans);

modelMatrix.scale(k, k, k);

}

void SimpleObject::**setTexture**(QString name){

texture = new QOpenGLTexture(QImage(name)); // mirrored;

use\_texture = true;

}

//-------------------------------------------------------------------------------

ChessFigure::**ChessFigure**(const QString &name, int grid\_sz, int field\_sz, int x, int y, int team):

SimpleObject(name), grid\_sz(grid\_sz), field\_sz(field\_sz), team(team)

{

enable = false;

if (team == TEAM\_WHITE) defaultColor = QVector4D(1.0, 1.0, 1.0, 1.0);

else defaultColor = QVector4D(0.2, 0.2, 0.2, 1.0);

color = defaultColor;

enableColor = QVector4D(0.45, 0.8, 0.55, 1.0);

pos\_x = x;

pos\_y = y;

float x1 = grid\_sz\*(x-field\_sz/2) - grid\_sz/2.f;

float y1 = grid\_sz\*(y-field\_sz/2) - grid\_sz/2.f;

modelMatrix.translate(x1, y1, 0.0);

}

void ChessFigure::**move**(int x1, int y1){

float dx = (x1 - pos\_x) \* grid\_sz;

float dy = (y1 - pos\_y) \* grid\_sz;

modelMatrix.translate(dx, dy, 0.0);

pos\_x = x1;

pos\_y = y1;

}

bool ChessFigure::**onThisPos**(int x, int y){

return pos\_x == x && pos\_y == y;

}

void ChessFigure::**select**(){

enable = true;

color = enableColor;

}

void ChessFigure::**deselect**(){

enable = false;

color = defaultColor;

}

int ChessFigure::**getTeam**(){

return team;

}

//---------------------------------------------------------------------------------------

LampObject::**LampObject**(const QString &name, QVector4D base\_color, QVector4D light\_color):

SimpleObject(name, base\_color), turn\_on(false), base\_color(base\_color), light\_color(light\_color),

power(1.0f), constant(1.0f), linear(0.004f), quadratic(0.00007f){

}

void LampObject::**setState**(bool state){

turn\_on = state;

if (turn\_on)

color = light\_color;

else

color = base\_color;

}

PointLightStruct LampObject::**getPointLightStruct**(){

PointLightStruct strct;

strct.pos = getCenterOfMass();

strct.power = power;

strct.color = color;

strct.linear = linear;

strct.constant = constant;

strct.quadratic = quadratic;

return strct;

}

void LampObject::***drawObject***(QOpenGLShaderProgram \*sh\_program, QOpenGLFunctions \*functions){

if (turn\_on){

qDebug() << "Лампа включена, но вызвано отрисовывание выключеной лампы";

return;

}

SimpleObject::drawObject(sh\_program, functions);

}

void LampObject::**drawObjAsLamp**(QOpenGLShaderProgram \*sh\_program, QOpenGLFunctions \*functions){

if (!turn\_on){

qDebug() << "Лампа выключена, но вызвано отрисовывание включеной лампы";

return;

}

SimpleObject::drawObject(sh\_program, functions);

}

//-------------------------------------------------------------------------------------------

SpotLightStruct Spot::**getSpotLightStruct**(){

SpotLightStruct strct;

strct.pos = position;

strct.color = light\_color;

strct.power = power;

strct.direction = direction;

strct.cutOff = cutOff;

return strct;

}

void Spot::**setPositionAndDirection**(QVector3D pos, QVector3D dir\_point){

position = pos;

direction = QVector3D(pos - dir\_point).normalized();

}

**mainwindow.h**

#ifndef MAINWINDOW\_H

#define MAINWINDOW\_H

#include <QOpenGLWidget>

#include "import\_obj.h"

#include <QOpenGLShaderProgram>

//#include <QOpenGLTexture>

#include <QOpenGLBuffer>

#include "simpleobject.h"

#include <QMouseEvent>

#include <QOpenGLContext>

#include <QKeyEvent>

#include "camera.h"

#include <QString>

#include <QFileDialog>

#define WHITE\_MOVE TEAM\_WHITE

#define BLACK\_MOVE TEAM\_BLACK

class **MainWindow** : public QOpenGLWidget

{

Q\_OBJECT

public:

**MainWindow**(QWidget \*parent = nullptr);

~***MainWindow***() override;

protected:

void ***initializeGL***() override;

void ***resizeGL***(int w , int h) override;

void ***paintGL***() override;

void **initShaders**();

private:

QMatrix4x4 projectionMatrix;

QVector2D mouse\_pos, click\_pos;

int field\_sz, grid\_sz;

Camera camera;

QOpenGLShaderProgram sh\_program, sh\_program\_simple;

int selected\_chess\_index;

int game\_status;

int white\_count, black\_count;

int cur\_obj\_num;

QVector<SimpleObject\*> chessBoard;

QVector<ChessFigure\*> chessFigures;

QVector<SimpleObject\*> otherObjects;

QVector<LampObject\*> lamps;

Spot\* spotLight;

void ***mousePressEvent***(QMouseEvent\* ev) override;

void ***mouseMoveEvent***(QMouseEvent\* ev) override;

void ***wheelEvent***(QWheelEvent\* wheelevent) override;

void ***keyPressEvent***(QKeyEvent \*event) override;

void **loadObj**(QString name);

void **loadTexture**(QString name);

QVector3D **screenToWord**(const QVector2D &mosue\_pos) const;

QVector2D **wordToField**(const QVector3D& coord) const;

int **findFig**(const QVector2D &pos) const;

void **endMove**();

void **drawLines**();

};

#endif // MAINWINDOW\_H

**mainwindow.c**

#include "mainwindow.h"

#include <QDebug>

MainWindow::**MainWindow**(QWidget \*parent)

: QOpenGLWidget(parent)

{

selected\_chess\_index = -1;

grid\_sz = 6; field\_sz = 8;

white\_count = black\_count = 16;

game\_status = WHITE\_MOVE;

cur\_obj\_num = -1;

}

MainWindow::~***MainWindow***()

{

}

void MainWindow::***initializeGL***(){

glClearColor(0.1f, 0.1f, 0.1f, 1.0f);

//glClearColor(1.0f, 1.0f, 1.0f, 1.0f);

glCullFace(GL\_BACK);

glEnable(GL\_CULL\_FACE);

glEnable(GL\_DEPTH\_TEST);

initShaders();

lamps.append(new LampObject(":/objects/cube.obj"));

lamps.at(0)->translate(QVector3D(-100, -100, 100));

lamps.at(0)->setPower(3);

spotLight = new Spot();

spotLight->setAngle(2.4f);

}

void MainWindow::***resizeGL***(int w , int h){

float aspect = w/(float)h;

projectionMatrix.setToIdentity();

projectionMatrix.perspective(45, aspect, 10.0f, 1000.0f);

}

void MainWindow::***paintGL***(){

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

sh\_program.bind();

sh\_program.setUniformValue("u\_projectionMatrix", projectionMatrix);

sh\_program.setUniformValue("u\_viewMatrix", camera.getviewMatrix());

QVector<PointLightStruct> strct\_vec;

int count = 0;

for (int i = 0; i < lamps.size(); i++)

if (lamps.at(i)->getState()){

strct\_vec.append(lamps.at(i)->getPointLightStruct());

count++;

}

sh\_program.setUniformValue("u\_numPointLights", count);

for (int i = 0; i < count; i++){

sh\_program.setUniformValue(QString("u\_pointLights[%1].pos").arg(i).toStdString().c\_str(), strct\_vec[i].pos);

sh\_program.setUniformValue(QString("u\_pointLights[%1].color").arg(i).toStdString().c\_str(), strct\_vec[i].color);

sh\_program.setUniformValue(QString("u\_pointLights[%1].power").arg(i).toStdString().c\_str(), strct\_vec[i].power);

sh\_program.setUniformValue(QString("u\_pointLights[%1].constant").arg(i).toStdString().c\_str(), strct\_vec[i].constant);

sh\_program.setUniformValue(QString("u\_pointLights[%1].linear").arg(i).toStdString().c\_str(), strct\_vec[i].linear);

sh\_program.setUniformValue(QString("u\_pointLights[%1].quadratic").arg(i).toStdString().c\_str(), strct\_vec[i].quadratic);

}

QVector<SpotLightStruct> strct\_vec\_sp;

QVector3D vec\_1, vec\_2;

for (int i = 0; i < chessFigures.size(); i++)

if(chessFigures[i]->getTeam() == game\_status){

//vec\_1 = chessFigures[i]->getCenterOfMass(); vec\_1.setZ(50);

vec\_1 = QVector3D(0.0, 0.0, 50.0);

vec\_2 = chessFigures[i]->getCenterOfMass(); vec\_2.setZ(0);

spotLight->setPositionAndDirection(vec\_1, vec\_2);

strct\_vec\_sp.append(spotLight->getSpotLightStruct());

}

sh\_program.setUniformValue("u\_numSpotLights", strct\_vec\_sp.size());

for (int i = 0; i < strct\_vec\_sp.size(); i++){

sh\_program.setUniformValue(QString("u\_spotLights[%1].pos").arg(i).toStdString().c\_str(), strct\_vec\_sp[i].pos);

sh\_program.setUniformValue(QString("u\_spotLights[%1].direction").arg(i).toStdString().c\_str(), strct\_vec\_sp[i].direction);

sh\_program.setUniformValue(QString("u\_spotLights[%1].color").arg(i).toStdString().c\_str(), strct\_vec\_sp[i].color);

sh\_program.setUniformValue(QString("u\_spotLights[%1].power").arg(i).toStdString().c\_str(), strct\_vec\_sp[i].power);

sh\_program.setUniformValue(QString("u\_spotLights[%1].cutOff").arg(i).toStdString().c\_str(), strct\_vec\_sp[i].cutOff);

}

if (cur\_obj\_num != -1)

otherObjects[cur\_obj\_num]->*drawObject*(&sh\_program, context()->functions());

for (int i = 0; i < lamps.size(); i++)

if (!lamps[i]->getState())

lamps[i]->*drawObject*(&sh\_program, context()->functions());

if (count > 0){

sh\_program\_simple.bind();

sh\_program\_simple.setUniformValue("u\_projectionMatrix", projectionMatrix);

sh\_program\_simple.setUniformValue("u\_viewMatrix", camera.getviewMatrix());

for (int i = 0; i < lamps.size(); i++){

if(lamps[i]->getState())

lamps[i]->drawObjAsLamp(&sh\_program\_simple, context()->functions());

}

}

}

void MainWindow::**initShaders**(){

if (!sh\_program.addShaderFromSourceFile(QOpenGLShader::Vertex, ":/shaders/vshader.vert")){

qDebug() << "ошибка добавления вершинного шейдера";

close();

}

if (!sh\_program.addShaderFromSourceFile(QOpenGLShader::Fragment, ":/shaders/fshader.frag")){

qDebug() << "ошибка добавления фрагментного шейдера";

close();

}

if (!sh\_program.*link*()){

qDebug() << "ошибка линковки шейдеров";

close();

}

if (!sh\_program\_simple.addShaderFromSourceFile(QOpenGLShader::Vertex, ":/shaders/vshader.vert")){

qDebug() << "ошибка добавления вершинного шейдера";

close();

}

if (!sh\_program\_simple.addShaderFromSourceFile(QOpenGLShader::Fragment, ":/shaders/fshader\_simple.frag")){

qDebug() << "ошибка добавления фрагментного шейдера";

close();

}

if (!sh\_program\_simple.*link*()){

qDebug() << "ошибка линковки шейдеров";

close();

}

}

void MainWindow::**loadObj**(QString name){

otherObjects.append(new SimpleObject(name));

cur\_obj\_num = otherObjects.size() - 1;

}

void MainWindow::**loadTexture**(QString name){

otherObjects[cur\_obj\_num]->setTexture(name);

}

//----------------------------------------------------------------------------------------------

void MainWindow::***wheelEvent***(QWheelEvent\* wheelevent){

camera.translate(QVector3D(0.0, 0.0, wheelevent->delta() / 100.f));

update();

}

void MainWindow::***keyPressEvent***(QKeyEvent \*event){

if (event->key() == Qt::Key\_Space)

camera.setDefault();

if (event->key() == Qt::Key\_Left)

lamps.at(0)->rotateZ(2);

if (event->key() == Qt::Key\_Right)

lamps.at(0)->rotateZ(-2);

if (event->key() == Qt::Key\_Up)

lamps.at(0)->translate(QVector3D(0, 0, 2));

if (event->key() == Qt::Key\_Down)

lamps.at(0)->translate(QVector3D(0, 0, -2));

if (event->key() == Qt::Key\_1)

lamps.at(0)->setState(!lamps.at(0)->getState());

/\*if (event->key() == Qt::Key\_2)

lamps.at(1)->setState(!lamps.at(1)->getState());\*/

if (event->key() == Qt::Key\_Escape)

showNormal();

if (event->key() == Qt::Key\_4) // выбор объекта

loadObj(QFileDialog::getOpenFileName());

if (event->key() == Qt::Key\_5) // выбор текстуры

loadTexture(QFileDialog::getOpenFileName());

if (event->key() == Qt::Key\_7){ // предыдущий объект

if (cur\_obj\_num == 0)

cur\_obj\_num = otherObjects.size()-1;

else cur\_obj\_num = (cur\_obj\_num-1)%otherObjects.size();

}

if (event->key() == Qt::Key\_8) // следующий объект

cur\_obj\_num = (cur\_obj\_num+1)%otherObjects.size();

update();

}

void MainWindow::***mouseMoveEvent***(QMouseEvent\* ev){

if (ev->buttons() != Qt::LeftButton) return;

//if (ev->buttons() != Qt::LeftButton) return;

QVector2D diff = QVector2D(ev->localPos()) - mouse\_pos;

float angleX = diff.y() / 2.0f;

float angleY = diff.x() / 2.0f;

camera.rotateX(QQuaternion::fromAxisAndAngle(1.0f, 0.0f, 0.0f, angleX));

camera.rotateY(QQuaternion::fromAxisAndAngle(0.0f, 0.0f, 1.0f, angleY));

mouse\_pos = QVector2D(ev->localPos());

update();

}

void MainWindow::***mousePressEvent***(QMouseEvent\* ev){

//if (ev->buttons() == Qt::MidButton)

if (ev->buttons() == Qt::LeftButton)

mouse\_pos = QVector2D(ev->localPos());

/\*if (ev->buttons() == Qt::LeftButton)

click\_pos = QVector2D(ev->localPos());\*/

}

//---------------------------------------------------------------------------------------------

QVector3D MainWindow::**screenToWord**(const QVector2D &mouse\_pos) const{

QVector4D tmp(2.0f \* mouse\_pos.x() / width() - 1.0f, -2.0f \* mouse\_pos.y() / height() + 1.0f, -1.0f, 1.0f);

QVector4D iTmp((projectionMatrix.inverted() \* tmp).toVector2D(), -1.0f, 0.0f);

QVector3D direction((camera.getviewMatrix().inverted() \* iTmp).toVector3D().normalized());

QVector3D camPos((camera.getviewMatrix().inverted() \* QVector4D(0.0f, 0.0f, 0.0f, 1.0f)).toVector3D());

// Ax + By +Cz + D = 0

// Norm = (A, B, C)

// P \* N - P0 \* N = 0

// O + D\*t

// (O + D\*t)\*N - P0\*N = 0

// t = (P0\*N - O\*N)/(D\*N)

// result = O + D \* t

QVector3D N(0.0f, 0.0f, 1.0f);

float t = -QVector3D::dotProduct(camPos, N) / QVector3D::dotProduct(direction, N);

QVector3D result = camPos + direction \* t;

return result;

}

QVector2D MainWindow::**wordToField**(const QVector3D &coord) const{

int x = int(coord.x() + grid\_sz \* field\_sz/2.0)/6 + 1;

int y = int(coord.y() + grid\_sz \* field\_sz/2.0)/6 + 1;

return QVector2D(x, y);

}

int MainWindow::**findFig**(const QVector2D &pos) const{

for (int i = 0; i < chessFigures.size(); i++)

if (chessFigures[i]->onThisPos(pos.x(), pos.y()))

return i;

return -1;

}

void MainWindow::**endMove**(){

if (game\_status == WHITE\_MOVE)

game\_status = BLACK\_MOVE;

else if (game\_status == BLACK\_MOVE)

game\_status = WHITE\_MOVE;

}

void MainWindow::**drawLines**(){

QVector3D min, max;

min = chessBoard[0]->getMin();

max = chessBoard[0]->getMax();

float dl\_x = (max.x()-min.x())/8;

float dl\_y = (max.y()-min.y())/8;

glBegin(GL\_LINES);

glColor3f(0.0f, 0.0f, 0.0f);

for (int i = 1; i < 8; i++){

glVertex3f(min.x(), min.y()+i\*dl\_y, min.z()+0.1f);

glVertex3f(max.x(), min.y()+i\*dl\_y, min.z()+0.1f);

glVertex3f(min.x()+i\*dl\_x, min.y(), min.z()+0.1f);

glVertex3f(min.x()+i\*dl\_x, max.y(), min.z()+0.1f);

}

glEnd();

}

