**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

Расчетно – Графическое Задание

Дисциплина: Компьютерная графика

по теме кубик-рубика на столе

Выполнил: ст. группы ВТ-32

Воскобойников И. С.

Проверил: Гибкин Ю. С.

**Белгород 2020**

ImageLoader.h

#pragma once

#include <assert.h>

#include <fstream>

using namespace std;

//Represents an image

class Image {

public:

Image(char\* ps, int w, int h);

~Image();

char\* pixels;

int width;

int height;

};

//Reads a bitmap image from file.

Image\* loadBMP(const char\* filename);

GLuint loadTexture(Image\* image) {

GLuint\* textures = new GLuint[11];//

glGenTextures(12, textures); //make room for 12 texture//

glBindTexture(GL\_TEXTURE\_2D, \*textures);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGB, image->width, image->height,

0, GL\_RGB, GL\_UNSIGNED\_BYTE, image->pixels); // actual pixel data

return \*textures;

}

GLuint \_textureId1;

GLuint \_textureId2;

GLuint \_textureId3;

GLuint \_textureId4;

GLuint \_textureId5;

GLuint \_textureId6;

Image::Image(char\* ps, int w, int h) : pixels(ps), width(w), height(h) {}

Image::~Image() {

delete[] pixels;

}

namespace {

// Converts a four-character array to an integer, using little-endian form

int toInt(const char\* bytes) {

return (int)(

((unsigned char)bytes[3] << 24) |

((unsigned char)bytes[2] << 16) |

((unsigned char)bytes[1] << 8) |

(unsigned char)bytes[0]

);

}

//Converts a two-character array to a short, using little-endian form

short toShort(const char\* bytes) {

return (short)(((unsigned char)bytes[1] << 8) |

(unsigned char)bytes[0]);

}

//Reads the next four bytes as an integer, using little-endian form

int readInt(ifstream& input) {

char buffer[4];

input.read(buffer, 4);

return toInt(buffer);

}

//Reads the next two bytes as a short, using little-endian form

short readShort(ifstream& input) {

char buffer[2];

input.read(buffer, 2);

return toShort(buffer);

}

//Just like auto\_ptr, but for arrays

template<class T>

class auto\_array {

private:

T\* array;

mutable bool isReleased;

public:

explicit auto\_array(T\* array\_ = NULL) :

array(array\_), isReleased(false) {

}

auto\_array(const auto\_array<T>& aarray) {

array = aarray.array;

isReleased = aarray.isReleased;

aarray.isReleased = true;

}

~auto\_array() {

if (!isReleased && array != NULL) {

delete[] array;

}

}

T\* get() const {

return array;

}

T& operator\*() const {

return \*array;

}

void operator=(const auto\_array<T>& aarray) {

if (!isReleased && array != NULL) {

delete[] array;

}

array = aarray.array;

isReleased = aarray.isReleased;

aarray.isReleased = true;

}

T\* operator->() const {

return array;

}

T\* release() {

isReleased = true;

return array;

}

void reset(T\* array\_ = NULL) {

if (!isReleased && array != NULL) {

delete[] array;

}

array = array\_;

}

T\* operator+(int i) {

return array + i;

}

T& operator[](int i) {

return array[i];

}

};

}

Image\* loadBMP(const char\* filename) {

ifstream input;

input.open(filename, ifstream::binary);

assert(!input.fail() || !"Could not find file");

char buffer[2];

input.read(buffer, 2);

assert(buffer[0] == 'B' && buffer[1] == 'M' || !"Not a bitmap file");

input.ignore(8);

int dataOffset = readInt(input);

//Read the header

int headerSize = readInt(input);

int width;

int height;

switch (headerSize) {

case 40:

//V3

width = readInt(input);

height = readInt(input);

input.ignore(2);

assert(readShort(input) == 24 || !"Image is not 24 bits per pixel");

assert(readShort(input) == 0 || !"Image is compressed");

break;

case 12:

//OS/2 V1

width = readShort(input);

height = readShort(input);

input.ignore(2);

assert(readShort(input) == 24 || !"Image is not 24 bits per pixel");

break;

case 64:

//OS/2 V2

assert(!"Can't load OS/2 V2 bitmaps");

break;

case 108:

//Windows V4

assert(!"Can't load Windows V4 bitmaps");

break;

case 124:

//Windows V5

assert(!"Can't load Windows V5 bitmaps");

break;

default:

assert(!"Unknown bitmap format");

}

//Read the data

int bytesPerRow = ((width \* 3 + 3) / 4) \* 4 - (width \* 3 % 4);

int size = bytesPerRow \* height;

auto\_array<char> pixels(new char[size]);

input.seekg(dataOffset, ios\_base::beg);

input.read(pixels.get(), size);

//Get the data into the right format

auto\_array<char> pixels2(new char[width \* height \* 3]);

for (int y = 0; y < height; y++) {

for (int x = 0; x < width; x++) {

for (int c = 0; c < 3; c++) {

pixels2[3 \* (width \* y + x) + c] =

pixels[bytesPerRow \* y + 3 \* x + (2 - c)];

}

}

}

input.close();

return new Image(pixels2.release(), width, height);

}

void initRender() {

glEnable(GL\_DEPTH\_TEST);

glEnable(GL\_COLOR\_MATERIAL);

glEnable(GL\_BLEND); // turns on alpha blending

glEnable(GL\_NORMALIZE);

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA); // to see the blending

Image\* image = loadBMP("assets/floor.bmp");

\_textureId1 = loadTexture(image);

image = loadBMP("assets/wall.bmp");

\_textureId2 = loadTexture(image);

image = loadBMP("assets/sky2.bmp");

\_textureId3 = loadTexture(image);

image = loadBMP("assets/glass.bmp");

\_textureId4 = loadTexture(image);

image = loadBMP("assets/wood.bmp");

\_textureId5 = loadTexture(image);

image = loadBMP("assets/lamp.bmp");

\_textureId6 = loadTexture(image);

delete image;

glClearColor(1.0f, 1.0f, 1.0f, 1.0f); // clears background colour and put alpha value as 1

}

Light.h

#pragma once

#include "glut.h"

static GLboolean lightSwitch = GL\_TRUE;

static int directionalLight = 0;

/\* Time varying or user-controled variables. \*/

static float lightAngle = 180.0;

static GLfloat lightPosition[4];

static GLfloat lightColor[] = { 1.0f, 1.0f, 0.8f }; // green-tinted

main

#include <cstring>

#include <Windows.h>

#include <time.h>

#include <iostream>

#include "glut.h"

#define STB\_IMAGE\_IMPLEMENTATION

#include "ImageLoader.h"

#include "Light.h"

// Cube config

#define CUBE\_SIZE 32

int TIMER = 5;

bool cubeAutoRatote = false;

bool rotFlag = false;

unsigned int c[9] = { 0x00FF00, 0x0000FF, 0xFFFF00, 0xAAAAAA, 0xFF860D, 0xFF0000 };

// Cam config

int xRot = 25, yRot = 25, zRot = 0;

double maxTranslateZ = -7.0 \* CUBE\_SIZE;

double translateZ = maxTranslateZ;

// Room config

float rs = CUBE\_SIZE \* 5; // size of room

float rw = CUBE\_SIZE \* 17; // Room width

float rh = CUBE\_SIZE \* 10; // Room height (to sky)

float wd = 0.01f; ; // element's padding

float wh = CUBE\_SIZE \* 15; // Wall's height

// Wall Lamp

float lampS = CUBE\_SIZE \* 4;// Lamp Size

float lampW = lampS \* 0.1; // Elements width

float lampPad = 1.0f; // Elements width

float bch = wh/2 - lampS/2; // Bottom corner height

float lampRX = rw - lampPad;// Coords on wall

// Floor config

static GLfloat floorVertices[4][3] = {

{ -rw, -rh, rw },

{ rw, -rh, rw },

{ rw, -rh, -rw },

{ -rw, -rh, -rw },

};

// Table config

float tw = rw \* 0.5; // width of table top from centre

float tl = rw \* 0.2; // length of table top from centre

float ttop = -rh \* 0.2;

float th = -rh \* 0.8; // height of table top from centre

float tt = CUBE\_SIZE \* 0.7; // thickness of table top

float ls = CUBE\_SIZE \* 0.4; // size of table leg

float tPadCenter = CUBE\_SIZE \* 2; // padding from center

float tlPadCenter = tPadCenter - ttop; // length padding from center

float min\_v = 0.0f;

float max\_v = 1.0f;

float wll = 0.01f;

struct Small\_Cube

{

// шесть граней куба - шесть цветов

unsigned int color[6]; // (верх, низ, впереди, сзади, лево, право)

double size; // размер ребра

Small\_Cube()

{

// по умолчанию черный цвет

memset(color, 0, sizeof(color));

size = 0.0;

}

// поворот на плоскости X0Y

void rotateZ()

{

unsigned int tmp = color[5];

color[5] = color[3];

color[3] = color[4];

color[4] = color[2];

color[2] = tmp;

}

// поворот на плоскости X0Z

void rotateY()

{

unsigned int tmp = color[2];

color[2] = color[1];

color[1] = color[3];

color[3] = color[0];

color[0] = tmp;

}

// поворот на плоскости Y0Z

void rotateX()

{

unsigned int tmp = color[0];

color[0] = color[4];

color[4] = color[1];

color[1] = color[5];

color[5] = tmp;

}

void setColor(int i, int color){this->color[i] = color;}

unsigned char\* at(int i)

{

// разбиваем color[i] на 3 составляющих

// например для 0xFF0000 RGB(FF, 0, 00) - красный цвет;

unsigned char \_color[3];

\_color[0] = color[i] >> 16;

\_color[1] = color[i] >> 8;

\_color[2] = color[i];

return \_color;

}

// отрисовка куба:

void draw(int isNotMirrow = 1)

{

glPushMatrix();

glBegin(GL\_QUADS);

// верх

glNormal3f(0, 1 \* isNotMirrow, 0);

glColor3ubv(at(0));

glVertex3f(size, size, size);

glVertex3f(0, size, size);

glVertex3f(0, 0, size);

glVertex3f(size, 0, size);

// низ

glNormal3f(0, 1 \* isNotMirrow, 0);

glColor3ubv(at(1));

glVertex3f(size, 0, 0);

glVertex3f(0, 0, 0);

glVertex3f(0, size, 0);

glVertex3f(size, size, 0);

// спереди

glNormal3f(0, 1 \* isNotMirrow, 0);

glColor3ubv(at(2));

glVertex3f(size, 0, size);

glVertex3f(0, 0, size);

glVertex3f(0, 0, 0);

glVertex3f(size, 0, 0);

// сзади

glNormal3f(0, 1 \* isNotMirrow, 0);

glColor3ubv(at(3));

glVertex3f(size, size, 0);

glVertex3f(0, size, 0);

glVertex3f(0, size, size);

glVertex3f(size, size, size);

// слева

glNormal3f(0, 1 \* isNotMirrow, 0);

glColor3ubv(at(4));

glVertex3f(0, size, size);

glVertex3f(0, size, 0);

glVertex3f(0, 0, 0);

glVertex3f(0, 0, size);

// справа

glNormal3f(0, 1 \* isNotMirrow, 0);

glColor3ubv(at(5));

glVertex3f(size, size, 0);

glVertex3f(size, size, size);

glVertex3f(size, 0, size);

glVertex3f(size, 0, 0);

glEnd();

glPopMatrix();

}

// отрисовка куба со смещением (x, y, z)

void draw(double x, double y, double z, int isNotMirrow = 1)

{

glPushMatrix();

glTranslated(x, y, z);

draw(isNotMirrow);

glPopMatrix();

}

};

class Cube

{

// 27 частей

Small\_Cube a[3][3][3];

// храним угол поворота каждой грани

int rotate[6];

// размер кубика-рубика

double size;

// цвета граней

unsigned int color[6];

public:

// храним номер грани, которая в данный момент поварачивается, или -1 если ничего не поварачивается

int current;

Cube(){}

Cube(double size, unsigned int\* color){clear(size, color);}

void clear(double size, unsigned int\* color)

{

memset(rotate, 0, sizeof(rotate));

this->size = size;

current = -1;

int i, j, k;

for (i = 0; i < 6; i++) this->color[i] = color[i];

for (i = 0; i < 3; i++) for (j = 0; j < 3; j++) a[i][j][2].setColor(0, color[0]); // верх

for (i = 0; i < 3; i++) for (j = 0; j < 3; j++) a[i][j][0].setColor(1, color[1]); // низ

for (k = 0; k < 3; k++) for (j = 0; j < 3; j++) a[j][0][k].setColor(2, color[2]); // спереди

for (k = 0; k < 3; k++) for (j = 0; j < 3; j++) a[j][2][k].setColor(3, color[3]); // сзади

for (i = 0; i < 3; i++) for (k = 0; k < 3; k++) a[0][k][i].setColor(4, color[4]); // слева

for (i = 0; i < 3; i++) for (k = 0; k < 3; k++) a[2][k][i].setColor(5, color[5]); // справа

// устанавливаем размеры мелких деталей

// это будет треть всего размера, умноженная на коэффициент немного

//меньший еденицы

// (чтобы детали не были слишком плотно)

for (i = 0; i < 3; i++) for (j = 0; j < 3; j++)

for (k = 0; k < 3; k++) a[i][j][k].size = (size / 3.0) \* 0.95;

}

void draw(int isNotMirrow = 1)

{

const double K = 0.65;

// рисуем корпус - это просто куб черного цвета, размер которого равен

//K\*size

glPushMatrix();

glColor3f(0, 0, 0);

glTranslatef(((1.0 - K) / 2) \* size + K \* size / 2, ((1.0 - K) / 2) \* size +

K \* size / 2, ((1.0 - K) / 2) \* size + K \* size / 2);

glutSolidCube(size \* K);

glPopMatrix();

// ok[i][j][k] показывает, находится ли в состоянии покоя деталь с

//координатами (i, j, k)

memset(ok, true, sizeof(ok));

if (current != -1)

{

glPushMatrix();

int i, j, k;

if (current == 0 || current == 1)

{

// 0 <= current <= 1 показывает, что сейчас крутится грань на

//плоскости X0Y

// current = 0 - нижняя часть

// current = 1 - верхняя часть

k = (current & 1) \* 2;

// следовательно ok слоя k устанавливаем в false

for (i = 0; i < 3; i++) for (j = 0; j < 3; j++) ok[i][j][k] = false;

// теперь нужно покрутить грань под номером current на угол rotate

//[current]

// относительно центра этой грани

// для этого сдвинемся к центру, покрутим, сдвинемся обратно

glTranslated(size / 2, size / 2, 0); // сдвигаемся к центру

glRotatef(rotate[current], 0, 0, 1); // крутим

glTranslated(-size / 2, -size / 2, 0); // сдвигаемся обратно

// рисуем

for (i = 0; i < 3; i++) for (j = 0; j < 3; j++) a[i][j][k].draw(size / 3 \* i, size / 3 \* j, size / 3 \* k, isNotMirrow);

}

// аналагично с остальными четырмя гранями

else if (current == 2 || current == 3)

{

j = (current & 1) \* 2;

for (i = 0; i < 3; i++) for (k = 0; k < 3; k++) ok[i][j][k] = false;

glTranslated(size / 2, 0, size / 2);

glRotatef(rotate[current], 0, 1, 0);

glTranslated(-size / 2, 0, -size / 2);

for (i = 0; i < 3; i++) for (k = 0; k < 3; k++) a[i][j][k].draw(size / 3 \* i, size / 3 \* j, size / 3 \* k, isNotMirrow);

}

else if (current == 4 || current == 5)

{

i = (current & 1) \* 2;

for (j = 0; j < 3; j++) for (k = 0; k < 3; k++) ok[i][j][k] = false;

glTranslated(0, size / 2, size / 2);

glRotatef(rotate[current], 1, 0, 0);

glTranslated(0, -size / 2, -size / 2);

for (j = 0; j < 3; j++) for (k = 0; k < 3; k++) a[i][j][k].draw(size / 3 \* i, size / 3 \* j, size / 3 \* k, isNotMirrow);

}

glPopMatrix();

}

for (int i = 0; i < 3; i++)

for (int j = 0; j < 3; j++)

for (int k = 0; k < 3; k++) if (ok[i][j][k])

// теперь рисуем те детали, которые не поварачивались выше,

// они отмечены ok[i][j][k] = true

a[i][j][k].draw(size / 3 \* i, size / 3 \* j, size / 3 \* k, isNotMirrow);

}

public:

void rot90(int idx, int sign)

{

int i, j, k;

// sign задаётся в зависимости он направления

// sign = -1, sign = 1

// если sign = -1, значит крутим 3 раза

if (sign == -1) sign = 3;

while (sign--)

{

if (idx == 0 || idx == 1)

{

// низ/верх

k = (idx & 1) \* 2;

// копируем повёрнутую на 90 градусов верхнюю/нижнюю грань

// в массив tmp, затем грани присваиваем tmp

// и не забываем повернуть каждую деталь этой грани

for (i = 0; i < 3; i++) for (j = 0; j < 3; j++) tmp[j][2 - i] = a[i][j][k];

for (i = 0; i < 3; i++) for (j = 0; j < 3; j++) tmp[i][j].rotateZ(), a[i][j][k] = tmp[i][j];

}

// аналогично с остальными четырмя гранями

else if (idx == 2 || idx == 3)

{

// лево/право

j = (idx & 1) \* 2;

for (i = 0; i < 3; i++) for (k = 0; k < 3; k++) tmp[k][2 - i] = a[i][j][k];

for (i = 0; i < 3; i++) for (k = 0; k < 3; k++) tmp[i][k].rotateX(), a[i][j][k] = tmp[i][k];

}

else if (idx == 4 || idx == 5)

{

// впереди/сзади

i = (idx & 1) \* 2;

for (j = 0; j < 3; j++) for (k = 0; k < 3; k++) tmp[k][2 - j] = a[i][j][k];

for (j = 0; j < 3; j++) for (k = 0; k < 3; k++) tmp[j][k].rotateY(), a[i][j][k] = tmp[j][k];

}

}

}

// крутит грань под номером idx на угол angle (в градусах)

void Rotate(int idx, int angle, int reverse)

{

// мы пытаемся покрутить грань с номером idx

// значит нужно проверить что другая грань уже не крутится

if (current == -1 || current == idx)

{

// обновляем поворот

if (idx == 1 || idx == 3 || idx == 5) angle \*= -1;

if (reverse == true) angle \*= -1;

rotate[idx] += angle;

if (rotate[idx] % 90 != 0) current = idx;

else

{

// если угол стал кратным 90, то поварачиваем на массиве

if ((rotate[idx] < 0) ^ (current == 2 || current == 3)) rot90(idx, 1);

else rot90(idx, -1);

rotate[idx] = 0;

current = -1;

rotFlag = false;

}

}

}

private:

int \_angle[4];

bool ok[4][4][4];

Small\_Cube tmp[4][4];

};

// кубик-рубик

Cube cube;

void drawTableTop() {

glPushMatrix(); // 1 set where to start the current object transformation

glTranslatef(0.0f, -2.0f, 0.0f); // move downwards to lie on the floor

glBegin(GL\_QUADS);

// bottom: normal pointing outwards

glTexCoord2f(min\_v, min\_v);

glVertex3f(-tw, ttop - tPadCenter, tl);

glTexCoord2f(max\_v, min\_v);

glVertex3f(-tw, ttop - tPadCenter, -tl);

glTexCoord2f(max\_v, max\_v);

glVertex3f(tw, ttop - tPadCenter, -tl);

glTexCoord2f(min\_v, max\_v);

glVertex3f(tw, ttop - tPadCenter, tl);

// Top:normal pointing outwards

// front:normal pointing outwards

glTexCoord2f(min\_v, min\_v);

glVertex3f(-tw, ttop + tt - tPadCenter, tl);

glTexCoord2f(max\_v, min\_v);

glVertex3f(-tw, ttop - tPadCenter, tl);

glTexCoord2f(max\_v, max\_v);

glVertex3f(tw, ttop - tPadCenter, tl);

glTexCoord2f(min\_v, max\_v);

glVertex3f(tw, ttop + tt - tPadCenter, tl);

// back:normal pointing outwards

glTexCoord2f(min\_v, min\_v);

glVertex3f(tw, ttop + tt - tPadCenter, -tl);

glTexCoord2f(max\_v, min\_v);

glVertex3f(tw, ttop - tPadCenter, -tl);

glTexCoord2f(max\_v, max\_v);

glVertex3f(-tw, ttop - tPadCenter, -tl);

glTexCoord2f(min\_v, max\_v);

glVertex3f(-tw, ttop + tt - tPadCenter, -tl);

// right:normal pointing outwards

glTexCoord2f(min\_v, min\_v);

glVertex3f(tw, ttop + tt - tPadCenter, tl);

glTexCoord2f(max\_v, min\_v);

glVertex3f(tw, ttop - tPadCenter, tl);

glTexCoord2f(max\_v, max\_v);

glVertex3f(tw, ttop - tPadCenter, -tl);

glTexCoord2f(min\_v, max\_v);

glVertex3f(tw, ttop + tt - tPadCenter, -tl);

// left:normal pointing outwards

glTexCoord2f(max\_v, max\_v);

glVertex3f(-tw, ttop + tt - tPadCenter, tl);

glTexCoord2f(min\_v, max\_v);

glVertex3f(-tw, ttop + tt - tPadCenter, -tl);

glTexCoord2f(min\_v, min\_v);

glVertex3f(-tw, ttop - tPadCenter, -tl);

glTexCoord2f(max\_v, min\_v);

glVertex3f(-tw, ttop - tPadCenter, tl);

// top

glTexCoord2f(min\_v, min\_v);

glVertex3f(-tw, ttop + tt - tPadCenter, tl);

glTexCoord2f(min\_v, max\_v);

glVertex3f(tw, ttop + tt - tPadCenter, tl);

glTexCoord2f(max\_v, max\_v);

glVertex3f(tw, ttop + tt - tPadCenter, -tl);

glTexCoord2f(max\_v, min\_v);

glVertex3f(-tw, ttop + tt - tPadCenter, -tl);

glEnd();

glPopMatrix();

}

void firstLeg() {

glPushMatrix(); // 1 set where to start the current object transformation

glTranslatef(0.0f, -2.0f, 0.0f); // move downwards to lie on the floor

glBegin(GL\_QUADS);

// front:normal pointing outwards

glNormal3f(0, 0, 1);

glTexCoord2f(min\_v, min\_v);

glVertex3f(-tw + ls, th - tlPadCenter, tl - ls);

glTexCoord2f(max\_v, min\_v);

glVertex3f(-tw + ls, -tlPadCenter, tl - ls);

glTexCoord2f(max\_v, max\_v);

glVertex3f(-tw + ls + ls, -tlPadCenter, tl - ls);

glTexCoord2f(min\_v, max\_v);

glVertex3f(-tw + ls + ls, th - tlPadCenter, tl - ls);

// back:normal pointing outwards

glNormal3f(0, 0, -1);

glTexCoord2f(1, 1);

glVertex3f(-tw + ls, th - tlPadCenter, tl - ls - ls);

glTexCoord2f(0, 1);

glVertex3f(-tw + ls + ls, th - tlPadCenter, tl - ls - ls);

glTexCoord2f(0, 0);

glVertex3f(-tw + ls + ls, -tlPadCenter, tl - ls - ls);

glTexCoord2f(1, 0);

glVertex3f(-tw + ls, -tlPadCenter, tl - ls - ls);

// right: normal pointing outwards

glNormal3f(1, 0, 0);

glTexCoord2f(0, 1);

glVertex3f(-tw + ls + ls, th - tlPadCenter, tl - ls);

glTexCoord2f(0, 0);

glVertex3f(-tw + ls + ls, -tlPadCenter, tl - ls);

glTexCoord2f(1, 0);

glVertex3f(-tw + ls + ls, -tlPadCenter, tl - ls - ls);

glTexCoord2f(1, 1);

glVertex3f(-tw + ls + ls, th - tlPadCenter, tl - ls - ls);

// left:normal pointing outwards

glNormal3f(-1, 0, 0);

glTexCoord2f(1, 1);

glVertex3f(-tw + ls, th - tlPadCenter, tl - ls);

glTexCoord2f(0, 1);

glVertex3f(-tw + ls, th - tlPadCenter, tl - ls - ls);

glTexCoord2f(0, 0);

glVertex3f(-tw + ls, -tlPadCenter, tl - ls - ls);

glTexCoord2f(1, 0);

glVertex3f(-tw + ls, -tlPadCenter, tl - ls);

glEnd();

glPopMatrix();

}

void secondLeg() {

glPushMatrix(); // 1 set where to start the current object transformation

glTranslatef(0.0f, -2.0f, 0.0f); // move downwards to lie on the floor

glBegin(GL\_QUADS);

// front:Normals pointing outwards

glNormal3f(0, 0, 1);

glTexCoord2f(1, 1);

glVertex3f(tw - ls, th - tlPadCenter, tl - ls);

glTexCoord2f(0, 1);

glVertex3f(tw - ls - ls, th - tlPadCenter, tl - ls);

glTexCoord2f(0, 0);

glVertex3f(tw - ls - ls, -tlPadCenter, tl - ls);

glTexCoord2f(1, 0);

glVertex3f(tw - ls, -tlPadCenter, tl - ls);

// back:normals pointing outwards

glNormal3f(0, 0, -1);

glTexCoord2f(0, 1);

glVertex3f(tw - ls, th - tlPadCenter, tl - ls - ls);

glTexCoord2f(0, 0);

glVertex3f(tw - ls, -tlPadCenter, tl - ls - ls);

glTexCoord2f(1, 0);

glVertex3f(tw - ls - ls, -tlPadCenter, tl - ls - ls);

glTexCoord2f(1, 1);

glVertex3f(tw - ls - ls, th - tlPadCenter, tl - ls - ls);

// left:normal pointing outwards

glNormal3f(-1, 0, 0);

glTexCoord2f(1, 1);

glVertex3f(tw - ls - ls, th - tlPadCenter, tl - ls);

glTexCoord2f(0, 1);

glVertex3f(tw - ls - ls, th - tlPadCenter, tl - ls - ls);

glTexCoord2f(0, 0);

glVertex3f(tw - ls - ls, -tlPadCenter, tl - ls - ls);

glTexCoord2f(1, 0);

glVertex3f(tw - ls - ls, -tlPadCenter, tl - ls);

// right:normal pointing outwards

glNormal3f(1, 0, 0);

glTexCoord2f(0, 1);

glVertex3f(tw - ls, th - tlPadCenter, tl - ls);

glTexCoord2f(0, 0);

glVertex3f(tw - ls, -tlPadCenter, tl - ls);

glTexCoord2f(1, 0);

glVertex3f(tw - ls, -tlPadCenter, tl - ls - ls);

glTexCoord2f(1, 1);

glVertex3f(tw - ls, th - tlPadCenter, tl - ls - ls);

glEnd();

glPopMatrix();

}

void thirdLeg() {

glPushMatrix(); // 1 set where to start the current object transformation

glTranslatef(0.0f, -2.0f, 0.0f); // move downwards to lie on the floor

glBegin(GL\_QUADS);

// front:normal pointing outwards

glNormal3f(0, 0, 1);

glTexCoord2f(min\_v, min\_v);

glVertex3f(-tw + ls, th - tlPadCenter, -tl + ls);

glTexCoord2f(max\_v, min\_v);

glVertex3f(-tw + ls, -tlPadCenter, -tl + ls);

glTexCoord2f(max\_v, max\_v);

glVertex3f(-tw + ls + ls, -tlPadCenter, -tl + ls);

glTexCoord2f(min\_v, max\_v);

glVertex3f(-tw + ls + ls, th - tlPadCenter, -tl + ls);

// back:normal pointing outwards

glNormal3f(0, 0, -1);

glTexCoord2f(1, 1);

glVertex3f(-tw + ls, th - tlPadCenter, -tl + ls + ls);

glTexCoord2f(0, 1);

glVertex3f(-tw + ls + ls, th - tlPadCenter, -tl + ls + ls);

glTexCoord2f(0, 0);

glVertex3f(-tw + ls + ls, -tlPadCenter, -tl + ls + ls);

glTexCoord2f(1, 0);

glVertex3f(-tw + ls, -tlPadCenter, -tl + ls + ls);

// left:normal pointing outwards

glNormal3f(-1, 0, 0);

glTexCoord2f(1, 1);

glVertex3f(-tw + ls, th - tlPadCenter, -tl + ls);

glTexCoord2f(0, 1);

glVertex3f(-tw + ls, th - tlPadCenter, -tl + ls + ls);

glTexCoord2f(0, 0);

glVertex3f(-tw + ls, -tlPadCenter, -tl + ls + ls);

glTexCoord2f(1, 0);

glVertex3f(-tw + ls, -tlPadCenter, -tl + ls);

// right: normal pointing outwards

glNormal3f(1, 0, 0);

glTexCoord2f(0, 1);

glVertex3f(-tw + ls + ls, th - tlPadCenter, -tl + ls);

glTexCoord2f(0, 0);

glVertex3f(-tw + ls + ls, -tlPadCenter, -tl + ls);

glTexCoord2f(1, 0);

glVertex3f(-tw + ls + ls, -tlPadCenter, -tl + ls + ls);

glTexCoord2f(1, 1);

glVertex3f(-tw + ls + ls, th - tlPadCenter, -tl + ls + ls);

glEnd();

glPopMatrix();

}

void fourthLeg() {

glPushMatrix(); // 1 set where to start the current object transformation

glTranslatef(0.0f, -2.0f, 0.0f); // move downwards to lie on the floor

glBegin(GL\_QUADS);

// front:Normals pointing outwards

glNormal3f(0, 0, 1);

glTexCoord2f(1, 1);

glVertex3f(tw - ls, th - tlPadCenter, -tl + ls);

glTexCoord2f(0, 1);

glVertex3f(tw - ls - ls, th - tlPadCenter, -tl + ls);

glTexCoord2f(0, 0);

glVertex3f(tw - ls - ls, -tlPadCenter, -tl + ls);

glTexCoord2f(1, 0);

glVertex3f(tw - ls, -tlPadCenter, -tl + ls);

// back:normals pointing outwards

glNormal3f(0, 0, -1);

glTexCoord2f(0, 1);

glVertex3f(tw - ls, th - tlPadCenter, -tl + ls + ls);

glTexCoord2f(0, 0);

glVertex3f(tw - ls, -tlPadCenter, -tl + ls + ls);

glTexCoord2f(1, 0);

glVertex3f(tw - ls - ls, -tlPadCenter, -tl + ls + ls);

glTexCoord2f(1, 1);

glVertex3f(tw - ls - ls, th - tlPadCenter, -tl + ls + ls);

// left:normal pointing outwards

glNormal3f(-1, 0, 0);

glTexCoord2f(1, 1);

glVertex3f(tw - ls - ls, th - tlPadCenter, -tl + ls);

glTexCoord2f(0, 1);

glVertex3f(tw - ls - ls, th - tlPadCenter, -tl + ls + ls);

glTexCoord2f(0, 0);

glVertex3f(tw - ls - ls, -tlPadCenter, -tl + ls + ls);

glTexCoord2f(1, 0);

glVertex3f(tw - ls - ls, -tlPadCenter, -tl + ls);

// right:normal pointing outwards

glNormal3f(1, 0, 0);

glTexCoord2f(0, 1);

glVertex3f(tw - ls, th - tlPadCenter, -tl + ls);

glTexCoord2f(0, 0);

glVertex3f(tw - ls, -tlPadCenter, -tl + ls);

glTexCoord2f(1, 0);

glVertex3f(tw - ls, -tlPadCenter, -tl + ls + ls);

glTexCoord2f(1, 1);

glVertex3f(tw - ls, th - tlPadCenter, -tl + ls + ls);

glEnd();

glPopMatrix();

}

void drawTable() {

glEnable(GL\_TEXTURE\_2D);//enabling texture

glBindTexture(GL\_TEXTURE\_2D, \_textureId5);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

drawTableTop();

firstLeg();

secondLeg();

thirdLeg();

fourthLeg();

glDisable(GL\_TEXTURE\_2D);

}

void drawSky(int isNotMirrow = 1) {

glEnable(GL\_TEXTURE\_2D);//enabling texture

glBindTexture(GL\_TEXTURE\_2D, \_textureId3);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glEnable(GL\_BLEND);//turns on alpha blending

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);

glColor4f(1.0f, 1.0f, 1.0f, 0.5f);

glBegin(GL\_QUADS);//normals pointing inwards

glNormal3f(0, 1 \* isNotMirrow, 0);

glTexCoord2f(0, 0);

glVertex3f(-rw \* 6, rh \* 3, -rw);

glTexCoord2f(1, 0);

glVertex3f(rw \* 6, rh \* 3, -rw);

glTexCoord2f(1, 1);

glVertex3f(rw \* 6, rh \* 3, rw);

glTexCoord2f(0, 1);

glVertex3f(-rw \* 6, rh \* 3, rw);

glEnd();

glDisable(GL\_BLEND);

glDisable(GL\_TEXTURE\_2D);

}

void drawGlass(int isNotMirrow = 1) {

float pad = 0.05f;

glEnable(GL\_TEXTURE\_2D);//enabling texture

glBindTexture(GL\_TEXTURE\_2D, \_textureId4);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glEnable(GL\_BLEND);//turns on alpha blending

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);

glColor4f(1.0f, 1.0f, 1.0f, 0.5f);

glBegin(GL\_QUADS); //normals pointing outwards

glNormal3f(0, 1, 0);

//front

glTexCoord2f(0, 0);

glVertex3f(-rw + pad , -rh + pad, -rw + pad);

glTexCoord2f(1, 0);

glVertex3f(rw - pad, -rh + pad, -rw + pad);

glTexCoord2f(1, 1);

glVertex3f(rw - pad, rh - pad, -rw + pad);

glTexCoord2f(0, 1);

glVertex3f(-rw + pad, rh - pad, -rw + pad);

////back

glTexCoord2f(0, 0);

glVertex3f(rw - pad, -rh + pad, rw - pad);

glTexCoord2f(1, 0);

glVertex3f(-rw + pad, -rh + pad, rw - pad);

glTexCoord2f(1, 1);

glVertex3f(-rw + pad, rh - pad, rw - pad);

glTexCoord2f(0, 1);

glVertex3f(rw - pad, rh - pad, rw - pad);

glEnd();

glDisable(GL\_BLEND);

glDisable(GL\_TEXTURE\_2D);

}

GLfloat lampLightPos[4] = { - lampRX + 0.3, 0.0, 0.0, .0 };

void drawWallLamp(int isNotMirrow = 1) {

glEnable(GL\_TEXTURE\_2D);//enabling texture

glBindTexture(GL\_TEXTURE\_2D, \_textureId6);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glBegin(GL\_QUADS); //normals pointing outwards

glNormal3f(0, -1 \* isNotMirrow, 0);

glTexCoord2f(0, 0);

glVertex3f(-lampRX, -lampS, -bch);

glTexCoord2f(1, 0);

glVertex3f(-lampRX, -lampS, bch);

glTexCoord2f(1, 1);

glVertex3f(-lampRX, lampS, bch);

glTexCoord2f(0, 1);

glVertex3f(-lampRX, lampS, -bch);

glEnd();

glDisable(GL\_TEXTURE\_2D);

glPushMatrix();

glEnable(GL\_LIGHT1);

glLightfv(GL\_LIGHT1, GL\_POSITION, lampLightPos);

GLfloat lampColor[] = { .0f, 1.0f, 0.8f };

glLightfv(GL\_LIGHT1, GL\_SHININESS, lampColor);

glLightf(GL\_LIGHT1, GL\_CONSTANT\_ATTENUATION, 0.0);

glLightf(GL\_LIGHT1, GL\_LINEAR\_ATTENUATION, 0.2);

glLightf(GL\_LIGHT1, GL\_QUADRATIC\_ATTENUATION, 0.4);

glPopMatrix();

}

void drawWall(int isNotMirrow=1) { //enclosing the walls of the room{

glEnable(GL\_TEXTURE\_2D);//enabling texture

glBindTexture(GL\_TEXTURE\_2D, \_textureId2);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glBegin(GL\_QUADS); //normals pointing outwards

////left

glNormal3f(0, -1 \* isNotMirrow, 0);

glTexCoord2f(0, 0);

glVertex3f(-rw, -rh, -rw);

glTexCoord2f(5, 0);

glVertex3f(-rw, -rh, rw);

glTexCoord2f(5, 5);

glVertex3f(-rw, rh, rw);

glTexCoord2f(0, 5);

glVertex3f(-rw, rh, -rw);

////right

glTexCoord2f(0, 0);

glVertex3f(rw, -rh, rw);

glTexCoord2f(5, 0);

glVertex3f(rw, -rh, -rw);

glTexCoord2f(5, 5);

glVertex3f(rw, rh, -rw);

glTexCoord2f(0, 5);

glVertex3f(rw, rh, rw);

glEnd();

glDisable(GL\_TEXTURE\_2D);

}

static void drawFloor(int isNotMirrow = 1) {

glEnable(GL\_TEXTURE\_2D);//enabling texture

glBindTexture(GL\_TEXTURE\_2D, \_textureId1);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glBegin(GL\_QUADS);

glNormal3f(0, 1 \* isNotMirrow, 0);

glTexCoord2f(0, 0);

glVertex3fv(floorVertices[0]);

glTexCoord2f(15, 0);

glVertex3fv(floorVertices[1]);

glTexCoord2f(15, 15);

glVertex3fv(floorVertices[2]);

glTexCoord2f(0, 15);

glVertex3fv(floorVertices[3]);

glEnd();

glDisable(GL\_TEXTURE\_2D);

}

void drawInMirrow() {

drawSky(-1);

drawWall(-1);

drawWallLamp(-1);

drawFloor(-1);

drawTable();

cube.draw(-1);

}

void drawMirrors() {

GLint buffers = GL\_NONE; //get current color buffer from being drawn

glGetIntegerv(GL\_DRAW\_BUFFER, &buffers); // set the clear value

glClearStencil(0x00); // clear the stencil buffer

glEnable(GL\_STENCIL\_TEST);

glColorMask(0, 0, 0, 0); //Disable drawing colors to the screen

// always pass the stencil test

glStencilOp(GL\_REPLACE, GL\_REPLACE, GL\_REPLACE); // disable drawing to the color buffer

glStencilFunc(GL\_ALWAYS, 1, 0xffffffff); // set the operation to modify the stencil buffer

//this would be the stencil mask-->

glDisable(GL\_DEPTH\_TEST);

glBegin(GL\_QUADS);

//back wall

glNormal3f(0.0f, 0.0f, 1.0f);

glVertex3f(-rs, -wh, -rs);

glVertex3f(rs, -wh, -rs);

glVertex3f(rs, wh, -rs);

glVertex3f(-rs, wh, -rs);

glEnd();

glBegin(GL\_QUADS);

////Frontwall

glVertex3f(-rs, wh, rs);

glVertex3f(rs, wh, rs);

glVertex3f(rs, -wh, rs);

glVertex3f(-rs, -wh, rs);

glEnd();

////////////////////////////////////////////////

// reenable drawing to color buffer

glDrawBuffer((GLenum)buffers);

glColorMask(1, 1, 1, 1); //Enable drawing colors to the screen

glStencilFunc(GL\_EQUAL, 1, 0xffffffff); // draw only where the stencil buffer == 1

glStencilOp(GL\_KEEP, GL\_KEEP, GL\_KEEP); // make stencil buffer read only

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT); // clear the color and depth buffers

glEnable(GL\_DEPTH\_TEST);

glPushMatrix(); // draw the mirror image

glScalef(1.0f, 1.0f, -1.0f);

for (int k = 1; k < 35; k += 2) {

glTranslatef(0.0f, 0.0f, k \* rw\*2);

drawInMirrow();

glTranslatef(0.0f, 0.0f, -(k+1) \* rw\*2);

drawInMirrow();

}

glPopMatrix();

glDisable(GL\_STENCIL\_TEST); // disable the stencil buffer

glDrawBuffer(GL\_NONE); // disable drawing to the color buffer

glDrawBuffer((GLenum)buffers);

}

void drawScene() {

drawMirrors();

glClearColor(1.0f, 1.0f, 1.0f, 1.0f); // clears background colour and put alpha value as 1

drawSky();

drawGlass();

drawWall();

drawWallLamp();

drawFloor();

drawTable();

}

void drawSunLight() {

lightAngle += 0.005f;

lightPosition[0] = 1.5 \* rw \* cos(lightAngle);

lightPosition[1] = 1.5 \* rw \* sin(lightAngle);

lightPosition[2] = -CUBE\_SIZE / 2;

lightPosition[3] = .0;

std::cout << lightPosition[0] << " " << lightPosition[1] << " " << lightPosition[2] << " " << std::endl;

glPushMatrix();

glEnable(GL\_LIGHT0);

glLightfv(GL\_LIGHT0, GL\_POSITION, lightPosition);

glLightfv(GL\_LIGHT0, GL\_DIFFUSE, lightColor);

glLightf(GL\_LIGHT0, GL\_CONSTANT\_ATTENUATION, 0.0);

glLightf(GL\_LIGHT0, GL\_LINEAR\_ATTENUATION, 0.2);

glLightf(GL\_LIGHT0, GL\_QUADRATIC\_ATTENUATION, 0.4);

glPopMatrix();

}

void display()

{

glLoadIdentity(); // Reset the drawing perspective

glPushMatrix();

glEnable(GL\_LIGHTING);

glLightf(GL\_LIGHT0, GL\_SHININESS, 64);

glLightModelf(GL\_LIGHT\_MODEL\_TWO\_SIDE, GL\_TRUE);

glLightModelf(GL\_LIGHT\_MODEL\_LOCAL\_VIEWER, 1);

glEnable(GL\_NORMALIZE);

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

//glColor3f(1, 0, 0);

glTranslatef(0, 0, translateZ);

glRotatef(xRot, 1, 0, 0);

glRotatef(yRot, 0, 1, 0);

drawSunLight();

drawScene();

glTranslatef(CUBE\_SIZE / -2.0, CUBE\_SIZE / -2.0, CUBE\_SIZE / -2.0);

cube.draw();

glPushMatrix();

glDisable(GL\_LIGHTING);

glColor3f(1.0, 1.0, 0.0);

glTranslatef(lightPosition[0], lightPosition[1], lightPosition[2]);

glutSolidSphere(2\*CUBE\_SIZE, 75, 75);

glPopMatrix();

glPopMatrix();

glutSwapBuffers();

}

void init()

{

glEnable(GL\_NORMALIZE);

glClearColor(0.7, 0.7, 0.7, 0.7);

srand(time(0));

cube.clear(CUBE\_SIZE, c);

}

//управление кубом посредством клавиш

void specialKeys(int key, int, int)

{

void timer(int);

void keys(unsigned char key, int, int);

// клавиши влево/вправо вращают по Y

// клавиши вверх/вниз вращают по X

if (key == GLUT\_KEY\_DOWN)

{

xRot += 3;

if (xRot >= 360) xRot -= 360;

glutPostRedisplay();

}

if (key == GLUT\_KEY\_UP)

{

xRot -= 3;

if (xRot < 0) xRot += 360;

glutPostRedisplay();

}

if (key == GLUT\_KEY\_RIGHT)

{

yRot += 3;

if (yRot >= 360) yRot -= 360;

glutPostRedisplay();

}

if (key == GLUT\_KEY\_LEFT)

{

yRot -= 3;

if (yRot < 0) yRot += 360;

glutPostRedisplay();

}

//восстановление состояния куба

if (key == GLUT\_KEY\_HOME)

{

cube.clear(CUBE\_SIZE, c);

translateZ = maxTranslateZ;

glutPostRedisplay();

xRot = 25;

yRot = 25;

zRot = 0;

}

if (key == GLUT\_KEY\_PAGE\_DOWN)

{

translateZ -= 10;

glutPostRedisplay();

}

if (key == GLUT\_KEY\_PAGE\_UP)

{

translateZ += 10;

if (translateZ > maxTranslateZ) translateZ = maxTranslateZ;

glutPostRedisplay();

}

//алгоритмы поворотов (смотри «Алгоритмы для новичков»

if (key == GLUT\_KEY\_F1) {

rotFlag = true; keys('R', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

rotFlag = true; keys('D', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

keys('R', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

keys('D', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

}

if (key == GLUT\_KEY\_F2) {

keys('D', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

keys('L', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

rotFlag = true; keys('D', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

rotFlag = true; keys('L', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

keys('F', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

rotFlag = true; keys('L', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

rotFlag = true; keys('F', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

keys('L', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

}

if (key == GLUT\_KEY\_F3) {

rotFlag = true; keys('D', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

rotFlag = true; keys('R', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

keys('D', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

keys('R', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

rotFlag = true; keys('F', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

keys('R', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

keys('F', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

rotFlag = true; keys('R', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

}

if (key == GLUT\_KEY\_F4) {

keys('F', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

keys('R', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

keys('U', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

rotFlag = true; keys('R', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

rotFlag = true; keys('U', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

rotFlag = true; keys('F', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

}

if (key == GLUT\_KEY\_F5) {

keys('R', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

keys('U', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

rotFlag = true; keys('R', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

keys('U', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

keys('R', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

keys('U', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

keys('U', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

rotFlag = true; keys('R', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

}

if (key == GLUT\_KEY\_F6) {

rotFlag = true; keys('R', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

rotFlag = true; keys('F', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

rotFlag = true; keys('L', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

keys('F', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

keys('R', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

rotFlag = true; keys('F', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

keys('L', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

keys('F', 0, 0); while (cube.current != -1) { cube.Rotate(cube.current, 3, rotFlag); display(); Sleep(TIMER); };

}

}

//повороты сторон куба в зависимости от повернутого кубика

//1) определяем как повернут кубик –> какая грань фронтальная. 2)выполняем по-вороты относительно фронта

class KeyChecker {

public:

static bool is\_Y(unsigned char key) { return (key == 'Y' || key == 'y' || key == 'Н' || key == 'н'); }

static bool is\_Q(unsigned char key) { return (key == 'Q' || key == 'q' || key == 'Й' || key == 'й'); }

static bool is\_W(unsigned char key) { return (key == 'W' || key == 'w' || key == 'Ц' || key == 'ц'); }

static bool is\_F(unsigned char key) { return (key == 'F' || key == 'f' || key == 'А' || key == 'а'); }

static bool is\_B(unsigned char key) { return (key == 'B' || key == 'b' || key == 'И' || key == 'т'); }

static bool is\_U(unsigned char key) { return (key == 'U' || key == 'u' || key == 'Г' || key == 'г'); }

static bool is\_D(unsigned char key) { return (key == 'D' || key == 'd' || key == 'В' || key == 'в'); }

static bool is\_R(unsigned char key) { return (key == 'R' || key == 'r' || key == 'К' || key == 'к'); }

static bool is\_L(unsigned char key) { return (key == 'L' || key == 'l' || key == 'Д' || key == 'д'); }

};

void keys(unsigned char key, int, int)

{

//повороты

if (KeyChecker::is\_Y(key)) {

for (int i = 0; i < 180; i++) {

xRot += 1;

if (xRot >= 360) xRot -= 360;

display();

Sleep(10);

}

}

else

if (KeyChecker::is\_Q(key)) {

int coef = 0;

if (xRot <= 30 || xRot >= 330) coef = -1;

if (xRot >= 150 || xRot <= 210) coef = 1;

for (int i = 0; i < 90; i++) {

yRot += coef;

if (yRot >= 360) yRot -= 360;

display();

Sleep(10);

}

}

else if (KeyChecker::is\_W(key)) {

int coef = 0;

if (xRot <= 30 || xRot >= 330) coef = -1;

if (xRot >= 150 || xRot <= 210) coef = 1;

for (int i = 0; i < 90; i++) {

yRot -= coef;

if (yRot < 0) yRot += 360;

display();

Sleep(10);

}

}

if (key == '+') TIMER += 2;

else if (key == '-' && TIMER != 1) TIMER -= 2;

if ((xRot >= 320 || xRot <= 40) && (yRot >= 320 || yRot <= 40)) {

if (KeyChecker::is\_F(key)) cube.Rotate(1, 3, rotFlag);

else if (KeyChecker::is\_B(key)) cube.Rotate(0, 3, rotFlag);

else if (KeyChecker::is\_U(key)) cube.Rotate(3, 3, rotFlag);

else if (KeyChecker::is\_D(key)) cube.Rotate(2, 3, rotFlag);

else if (KeyChecker::is\_R(key)) cube.Rotate(5, 3, rotFlag);

else if (KeyChecker::is\_L(key)) cube.Rotate(4, 3, rotFlag);

}

else if ((xRot >= 320 || xRot <= 40) && (yRot >= 50 && yRot <= 130)) {

if (KeyChecker::is\_F(key)) cube.Rotate(4, 3, rotFlag);

else if (KeyChecker::is\_B(key)) cube.Rotate(5, 3, rotFlag);

else if (KeyChecker::is\_U(key)) cube.Rotate(3, 3, rotFlag);

else if (KeyChecker::is\_D(key)) cube.Rotate(2, 3, rotFlag);

else if (KeyChecker::is\_R(key)) cube.Rotate(1, 3, rotFlag);

else if (KeyChecker::is\_L(key)) cube.Rotate(0, 3, rotFlag);

}

else if ((xRot >= 320 || xRot <= 40) && (yRot >= 140 && yRot <= 220)) {

if (KeyChecker::is\_F(key)) cube.Rotate(0, 3, rotFlag);

else if (KeyChecker::is\_B(key)) cube.Rotate(1, 3, rotFlag);

else if (KeyChecker::is\_U(key)) cube.Rotate(3, 3, rotFlag);

else if (KeyChecker::is\_D(key)) cube.Rotate(2, 3, rotFlag);

else if (KeyChecker::is\_R(key)) cube.Rotate(4, 3, rotFlag);

else if (KeyChecker::is\_L(key)) cube.Rotate(5, 3, rotFlag);

}

else if ((xRot >= 320 || xRot <= 40) && (yRot >= 230 && yRot <= 310)) {

if (KeyChecker::is\_F(key)) cube.Rotate(5, 3, rotFlag);

else if (KeyChecker::is\_B(key)) cube.Rotate(4, 3, rotFlag);

else if (KeyChecker::is\_U(key)) cube.Rotate(3, 3, rotFlag);

else if (KeyChecker::is\_D(key)) cube.Rotate(2, 3, rotFlag);

else if (KeyChecker::is\_R(key)) cube.Rotate(0, 3, rotFlag);

else if (KeyChecker::is\_L(key)) cube.Rotate(1, 3, rotFlag);

}

else if ((xRot >= 140 && xRot <= 220) && (yRot >= 140 && yRot <= 220)) {

if (KeyChecker::is\_F(key)) cube.Rotate(1, 3, rotFlag);

else if (KeyChecker::is\_B(key)) cube.Rotate(0, 3, rotFlag);

else if (KeyChecker::is\_U(key)) cube.Rotate(2, 3, rotFlag);

else if (KeyChecker::is\_D(key)) cube.Rotate(3, 3, rotFlag);

else if (KeyChecker::is\_R(key)) cube.Rotate(4, 3, rotFlag);

else if (KeyChecker::is\_L(key)) cube.Rotate(5, 3, rotFlag);

}

else if ((xRot >= 140 && xRot <= 220) && (yRot >= 230 && yRot <= 310)) {

if (KeyChecker::is\_F(key)) cube.Rotate(4, 3, rotFlag);

else if (KeyChecker::is\_B(key)) cube.Rotate(5, 3, rotFlag);

else if (KeyChecker::is\_U(key)) cube.Rotate(2, 3, rotFlag);

else if (KeyChecker::is\_D(key)) cube.Rotate(3, 3, rotFlag);

else if (KeyChecker::is\_R(key)) cube.Rotate(0, 3, rotFlag);

else if (KeyChecker::is\_L(key)) cube.Rotate(1, 3, rotFlag);

}

else if ((xRot >= 140 && xRot <= 220) && (yRot >= 320 || yRot <= 40)) {

if (KeyChecker::is\_F(key)) cube.Rotate(0, 3, rotFlag);

else if (KeyChecker::is\_B(key)) cube.Rotate(1, 3, rotFlag);

else if (KeyChecker::is\_U(key)) cube.Rotate(2, 3, rotFlag);

else if (KeyChecker::is\_D(key)) cube.Rotate(3, 3, rotFlag);

else if (KeyChecker::is\_R(key)) cube.Rotate(5, 3, rotFlag);

else if (KeyChecker::is\_L(key)) cube.Rotate(4, 3, rotFlag);

}

else if ((xRot >= 140 && xRot <= 220) && (yRot >= 50 && yRot <= 130)) {

if (KeyChecker::is\_F(key)) cube.Rotate(5, 3, rotFlag);

else if (KeyChecker::is\_B(key)) cube.Rotate(4, 3, rotFlag);

else if (KeyChecker::is\_U(key)) cube.Rotate(2, 3, rotFlag);

else if (KeyChecker::is\_D(key)) cube.Rotate(3, 3, rotFlag);

else if (KeyChecker::is\_R(key)) cube.Rotate(1, 3, rotFlag);

else if (KeyChecker::is\_L(key)) cube.Rotate(0, 3, rotFlag);

}

if (key == 32) rotFlag = true;

}

//повороты случайных граней

void mouse(int key, int state, int x, int y)

{

if (key == GLUT\_RIGHT\_BUTTON && state == GLUT\_DOWN) cubeAutoRatote = !cubeAutoRatote; // переключаем флаг

}

void timer(int)

{

glutTimerFunc(TIMER, timer, 0);

if (cubeAutoRatote)

{

// если включен автоматический поворот, и смотрим

// если сейчас никакая грань не крутится, то начинаем крутить случайную,

// иначе крутим текущую

if (cube.current == -1) {

char k[6] = {'F', 'B', 'U', 'D', 'R', 'L'};

keys(k[rand() % 6], 0, 0);

}

else cube.Rotate(cube.current, 3, rotFlag);

}

else if (cube.current != -1) cube.Rotate(cube.current, 3, rotFlag);

display();

}

void reshape(int w, int h)

{

glViewport(0, 0, w, h);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

GLfloat fAspect = (GLfloat)w / (GLfloat)h;

gluPerspective(60, fAspect, 1, 90000.0);

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

}

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

{

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGB | GLUT\_DEPTH);

glutInitWindowSize(800, 800);

glutInitWindowPosition(1, 1);

glutCreateWindow("Rube Cube");

initRender(); //initialize rendering

init();

glutDisplayFunc(display);

glutReshapeFunc(reshape);

glutKeyboardFunc(keys);

glutMouseFunc(mouse);

glutTimerFunc(TIMER, timer, 0);

glutSpecialFunc(specialKeys);

glutMainLoop();

return 0;

}

