Московский авиационный институт

(Национальный исследовательский университет)

**Институт «Информационных технологий и прикладной математики»**

**Лабораторная работа №3**

Основы построения фотореалистичных изображений

Работу выполнил:

Рябыкин Алексей Сергеевич

Группа: М8О-309Б-18

Преподаватель: Филиппов Г.С.

Оценка:

Дата:

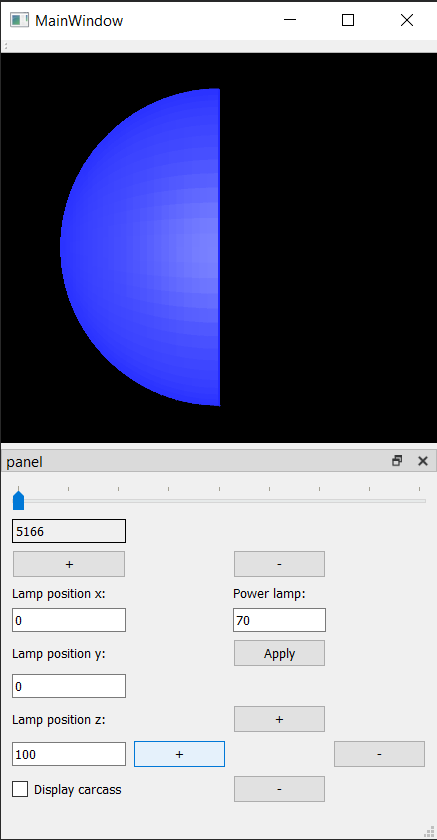
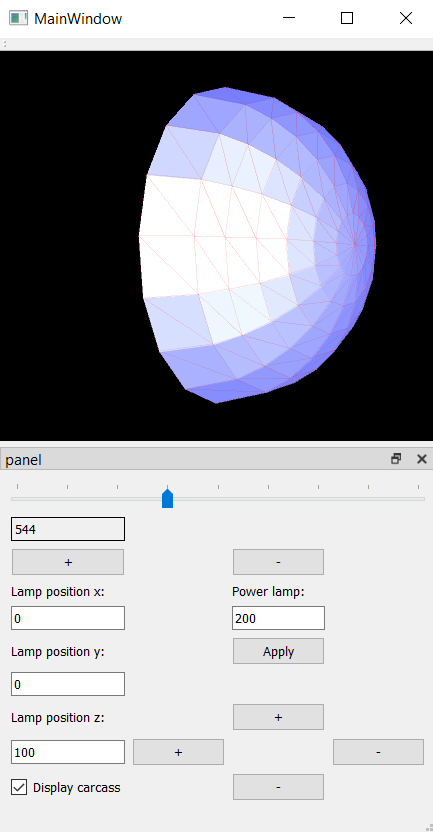
**Москва, 2020**

**Постановка задачи**

Используя результаты Л.Р.№2, аппроксимировать заданное тело выпуклым многогранником. Точность аппроксимации задается пользователем. Обеспечить возможность вращения и масштабирования многогранника и удаление невидимых линий и поверхностей. Реализовать простую модель закраски для случая одного источника света. Параметры освещения и отражающие свойства материала задаются пользователем в диалоговом режиме.

**Вариант многогранника: 4. Полушарие**

**С**криншоты работы программы:

Фрагменты кода:

**Функция Draw в классе Polygon:**

void Polygon::draw(QPainter \*ptr, int center\_x, int center\_y, double step\_pixels,

int window\_center\_x, int window\_center\_y, Lamp \*lamp,

bool displayCarcass) {

QPen oldPen = ptr->pen();

int resCalcAmbientComponent = calc\_ambient\_component(lamp);

int resCalcDiffuseComponent = calc\_diffuse\_component(center\_x - window\_center\_x,

center\_y - window\_center\_y, lamp);

int resCalcSpecularComponent = calc\_specular\_component(center\_x - window\_center\_x,

center\_y - window\_center\_y, lamp);

int r = rgb['r'] + resCalcAmbientComponent + resCalcDiffuseComponent + resCalcSpecularComponent;

int g = rgb['g'] + resCalcAmbientComponent + resCalcDiffuseComponent + resCalcSpecularComponent;

int b = rgb['b'] + resCalcAmbientComponent + resCalcDiffuseComponent + resCalcSpecularComponent;

*if* (r > 255) {

r = 255;

}

*if* (g > 255) {

g = 255;

}

*if* (b > 255) {

b = 255;

}

QPen newPen(QColor(r, g, b), 0.5, Qt::*SolidLine*, Qt::*FlatCap*, Qt::*RoundJoin*);

ptr->setPen(newPen);

ptr->setBrush(QColor(r, g, b));

QPolygonF pol;

*for* (size\_t i = 0; i < 3; i++) {

pol << QPointF(

*static\_cast*<double>(vertices[i][0]) \* step\_pixels + center\_x,

*static\_cast*<double>(vertices[i][1]) \* step\_pixels + center\_y

);

}

ptr->drawPolygon(pol);

*if* (displayCarcass) {

ptr->setPen(oldPen);

*for* (size\_t i = 0; i < 3; i++) {

ptr->drawLine(

*static\_cast*<int>(*static\_cast*<double>(vertices[i][0]) \* step\_pixels + center\_x),

*static\_cast*<int>(*static\_cast*<double>(vertices[i][1]) \* step\_pixels + center\_y),

*static\_cast*<int>(*static\_cast*<double>(vertices[(i + 1) % 3][0]) \* step\_pixels + center\_x),

*static\_cast*<int>(*static\_cast*<double>(vertices[(i + 1) % 3][1]) \* step\_pixels + center\_y)

);

}

}

**Создание сферы из полигонов:**

void sphere::create() {

std::vector<QVector4D> prevPoints{};

QVector4D firstIter{0, 0, *static\_cast*<float>(r \* cos(0)), 1};

QVector4D lastIter{0, 0, 0, 1};

bool connectToOnePoint = *true*;

*for* (double theta = step / 2.; theta < M\_PI / 2.; theta += step / 2.) {

*if* (connectToOnePoint) {

QVector4D prevVertex;

QVector4D firstVertex;

*for* (double phi = 0.; phi < 2 \* M\_PI; phi += step) {

*if* (phi == 0.) {

firstVertex = {

*static\_cast*<float>(r \* sin(theta) \* cos(phi)),

*static\_cast*<float>(r \* sin(theta) \* sin(phi)),

*static\_cast*<float>(r \* cos(theta)),

1

};

prevVertex = firstVertex;

prevPoints.push\_back(prevVertex);

*continue*;

}

std::vector<QVector4D> toPushBack;

QVector4D newVertex;

newVertex = {

*static\_cast*<float>(r \* sin(theta) \* cos(phi)),

*static\_cast*<float>(r \* sin(theta) \* sin(phi)),

*static\_cast*<float>(r \* cos(theta)),

1

};

toPushBack = {

firstIter,

prevVertex,

newVertex

};

polygons.push\_back(toPushBack);

prevVertex = newVertex;

prevPoints.push\_back(prevVertex);

*if* (phi + step >= 2 \* M\_PI) {

toPushBack = {

firstIter,

prevVertex,

firstVertex

};

polygons.push\_back(toPushBack);

prevPoints.push\_back(firstVertex);

connectToOnePoint = *false*;

}

}

} *else* *if* (theta + step/2. > M\_PI/2.){

theta = M\_PI/2.;

QVector4D prevVertex;

QVector4D firstVertex;

std::vector<QVector4D> newPrevPoints{};

size\_t cnt = 0;

*for* (double phi = 0; phi < 2 \* M\_PI; phi += step, cnt++) {

*if* (phi == 0.) {

firstVertex = {

*static\_cast*<float>(r \* sin(theta) \* cos(phi)),

*static\_cast*<float>(r \* sin(theta) \* sin(phi)),

*static\_cast*<float>(r \* cos(theta)),

1

};

prevVertex = firstVertex;

newPrevPoints.push\_back(prevVertex);

*continue*;

}

std::vector<QVector4D> toPushBack;

QVector4D newVertex;

newVertex = {

*static\_cast*<float>(r \* sin(theta) \* cos(phi)),

*static\_cast*<float>(r \* sin(theta) \* sin(phi)),

*static\_cast*<float>(r \* cos(theta)),

1

};

toPushBack = {

prevPoints[cnt - 1],

prevVertex,

newVertex

};

polygons.push\_back(toPushBack);

toPushBack = {

prevPoints[cnt - 1],

newVertex,

prevPoints[cnt]

};

polygons.push\_back(toPushBack);

prevVertex = newVertex;

newPrevPoints.push\_back(prevVertex);

*if* (phi + step > 2 \* M\_PI) {

cnt++;

toPushBack = {

prevPoints[cnt - 1],

prevVertex,

firstVertex

};

polygons.push\_back(toPushBack);

toPushBack = {

prevPoints[cnt - 1],

firstVertex,

prevPoints[cnt]

};

polygons.push\_back(toPushBack);

newPrevPoints.push\_back(firstVertex);

prevPoints = newPrevPoints;

*if* (theta + step / 2. > M\_PI) {

connectToOnePoint = *true*;

}

}

}

}

*else* {

QVector4D prevVertex;

QVector4D firstVertex;

std::vector<QVector4D> newPrevPoints{};

size\_t cnt = 0;

*for* (double phi = 0; phi < 2 \* M\_PI; phi += step, cnt++) {

*if* (phi == 0.) {

firstVertex = {

*static\_cast*<float>(r \* sin(theta) \* cos(phi)),

*static\_cast*<float>(r \* sin(theta) \* sin(phi)),

*static\_cast*<float>(r \* cos(theta)),

1

};

prevVertex = firstVertex;

newPrevPoints.push\_back(prevVertex);

*continue*;

}

std::vector<QVector4D> toPushBack;

QVector4D newVertex;

newVertex = {

*static\_cast*<float>(r \* sin(theta) \* cos(phi)),

*static\_cast*<float>(r \* sin(theta) \* sin(phi)),

*static\_cast*<float>(r \* cos(theta)),

1

};

toPushBack = {

prevPoints[cnt - 1],

prevVertex,

newVertex

};

polygons.push\_back(toPushBack);

toPushBack = {

prevPoints[cnt - 1],

newVertex,

prevPoints[cnt]

};

polygons.push\_back(toPushBack);

prevVertex = newVertex;

newPrevPoints.push\_back(prevVertex);

*if* (phi + step > 2 \* M\_PI) {

cnt++;

toPushBack = {

prevPoints[cnt - 1],

prevVertex,

firstVertex

};

polygons.push\_back(toPushBack);

toPushBack = {

prevPoints[cnt - 1],

firstVertex,

prevPoints[cnt]

};

polygons.push\_back(toPushBack);

newPrevPoints.push\_back(firstVertex);

prevPoints = newPrevPoints;

*if* (theta + step / 2. > M\_PI) {

connectToOnePoint = *true*;

}

}

}

}

}

connectToOnePoint = *true*;

prevPoints.clear();

*for* (double f = 0.; f < r; f += r / 10.) {

*if* (connectToOnePoint) {

QVector4D prevVertex, firstVertex;

*for* (double phi = 0.; phi < 2. \* M\_PI; phi += step) {

*if* (phi == 0.) {

firstVertex = {

*static\_cast*<float>(f \* cos(phi)),

*static\_cast*<float>(f \* sin(phi)),

0.,

1.

};

prevVertex = firstVertex;

prevPoints.push\_back(prevVertex);

*continue*;

}

std::vector<QVector4D> toPushBack;

QVector4D newVertex = {

*static\_cast*<float>(f \* cos(phi)),

*static\_cast*<float>(f \* sin(phi)),

0.,

1.

};

toPushBack = {

lastIter,

newVertex,

prevVertex

};

polygons.push\_back(toPushBack);

prevVertex = newVertex;

prevPoints.push\_back(prevVertex);

*if* (phi + step >= 2 \* M\_PI) {

toPushBack = {

lastIter,

firstVertex,

prevVertex

};

polygons.push\_back(toPushBack);

prevPoints.push\_back(firstVertex);

}

}

connectToOnePoint = *false*;

} *else* {

*if* (f + step >= r) {

f = r;

}

QVector4D prevVertex, firstVertex;

std::vector<QVector4D> newPrevPoints;

size\_t cnt = 0;

*for* (double phi = 0.; phi < 2 \* M\_PI; phi += step, cnt++) {

*if* (phi == 0.) {

firstVertex = {

*static\_cast*<float>(f \* cos(phi)),

*static\_cast*<float>(f \* sin(phi)),

0.,

1.

};

prevVertex = firstVertex;

newPrevPoints.push\_back(prevVertex);

*continue*;

}

std::vector<QVector4D> toPushBack;

QVector4D newVertex = {

*static\_cast*<float>(f \* cos(phi)),

*static\_cast*<float>(f \* sin(phi)),

0.,

1.

};

toPushBack = {

prevPoints[cnt - 1],

newVertex,

prevVertex

};

polygons.push\_back(toPushBack);

toPushBack = {

prevPoints[cnt - 1],

prevPoints[cnt],

newVertex

};

polygons.push\_back(toPushBack);

prevVertex = newVertex;

newPrevPoints.push\_back(prevVertex);

*if* (phi + step >= 2 \* M\_PI) {

cnt++;

toPushBack = {

prevPoints[cnt - 1],

firstVertex,

prevVertex

};

polygons.push\_back(toPushBack);

toPushBack = {

prevPoints[cnt - 1],

prevPoints[cnt],

firstVertex

};

polygons.push\_back(toPushBack);

newPrevPoints.push\_back(firstVertex);

prevPoints = newPrevPoints;

}

}

}

}

}

**Среда разработки:** Qt Creator 4.10.1

**Вывод:** В процессе выполнения лабораторной работы научился отрисовывать, масштабировать, центрировать при изменении окна, вращать и удалять невидимые линии для отрисовки выпуклых тел для одного источника света.