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:

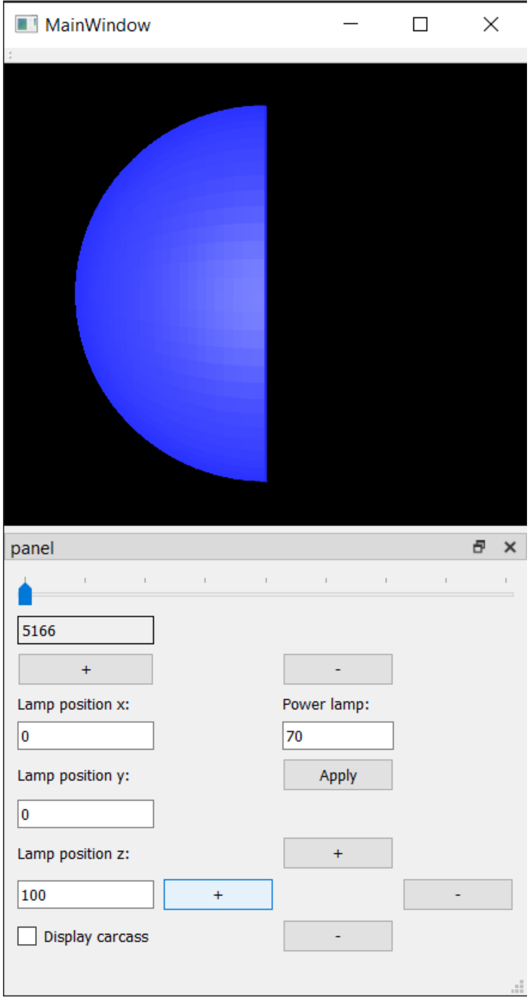
Москва, 2020

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

Скриншоты работы программы:



5166

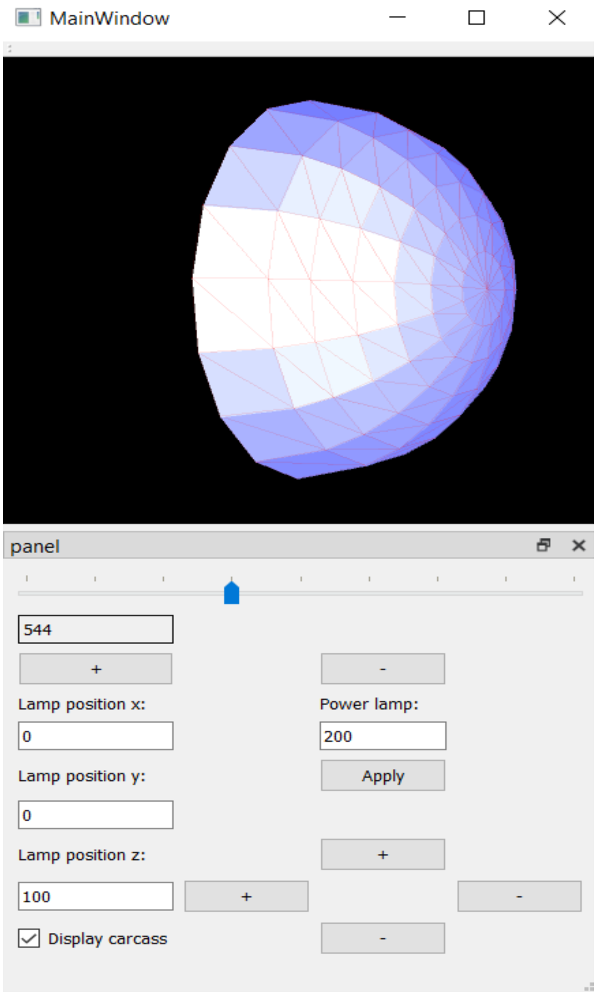
Lamp position x: 0

Power lamp: 70

Lamp position y: 0

Lamp position z: 100

□ Display carcass



544

Lamp position x: 0

Power lamp: 200

Lamp position y: 0

Lamp position z: 100

☑ Display carcass

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.