Московский авиационный институт
(Национальный исследовательский университет)

Факультет: «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»

Лабораторная работа №3

по курсу «Компьютерная графика»
Тема: «Основы построения фотореалистичных изображений»

Студент: Мариничев И. А.
Группа: М8О-308Б-19
Преподаватель: Филиппов Г. С.
Оценка:

Москва
2021

1. 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 №8: Наклонный круговой цилиндр.

2. Описание программы.

Для решения задачи я решил использовать C++ и фреймворк Qt, в котором использовал библиотеку QPainter.

Я создал класс `polygon` для хранения полигонов, класс `oblique_circular_cylinder`, представляющий фигуру наклонный круговой цилиндр. Такая фигура состоит из множества полигонов. Пользователь может аппроксимировать эту фигуру по разным осям. Все преобразования для фигуры выполняются для каждого полигона, и в каждом полигоне преобразования выполняются для каждой точки. Так выполняются пространственные повороты фигуры и масштабирование фигуры.

Я использовал модель освещения, построенную как сумма трех световых составляющих: фоновая, рассеянная, зеркальная.

Фоновая составляющая:

$$I_a = k_a \times i_a \text{, где}$$

I_a – фоновая составляющая освещенности в точке,
k_a – свойство материала воспринимать фоновое освещение,
i_a – мощность фонового освещения.

Фоновая составляющая освещенности не зависит от пространственных координат освещаемой точки и источника.

Рассеянное отражение света происходит, когда свет как бы проникает под поверхность объекта, поглощается, а затем вновь испускается. При этом положение наблюдателя не имеет значения, так как диффузно отраженный свет рассеивается равномерно по всем направлениями. Интенсивность света обратно пропорциональна квадрату расстояния от источника, следовательно, объект, лежащий дальше от него, должен быть темнее.

$$I_d = \frac{k_a \times i_l}{d+K} \times \cos(\vec{L},\ \vec{N}) = \frac{k_a \times i_l}{d+K} \times \left(\vec{L} \times \vec{N}\right) \text{, где}$$

K – произвольная постоянная,
d – расстояние от центра проекции до объекта,
I_d – рассеянная составляющая освещенности в точке,
k_d – свойство материала воспринимать рассеянное освещение,

i_l – интенсивность точечного источника,
L – направление из точки на источник света,
N – вектор нормали в точке.

Зеркальная составляющая влияет на появление блика на объекте. Местонахождение блика на объекте определяется из закона равенства углов падения и отражения. Если наблюдатель находится вблизи углов отражения, яркость соответствующей точки повышается.

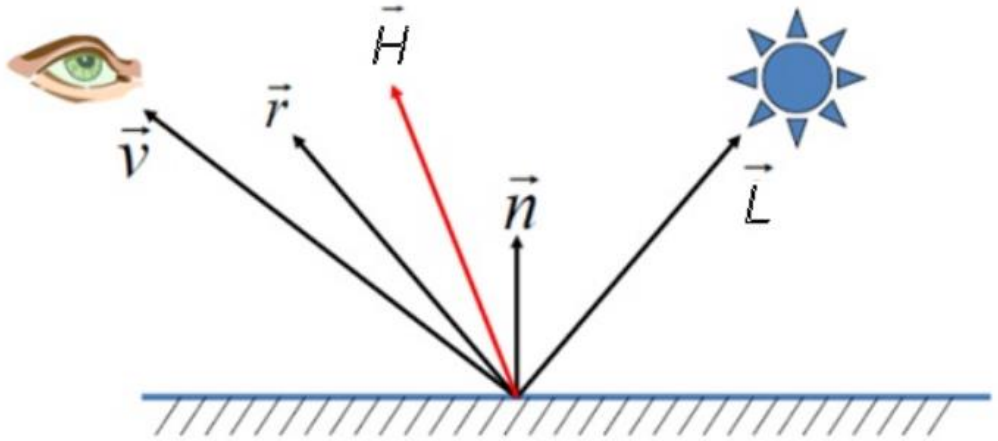


Рис. Вектора, необходимые для расчета освещенности по модели Блинна-Фонга: вектор на источник света s, вектор на наблюдателя v, отраженный вектор от источника r, средний вектор между отраженным вектором и нормалью h.

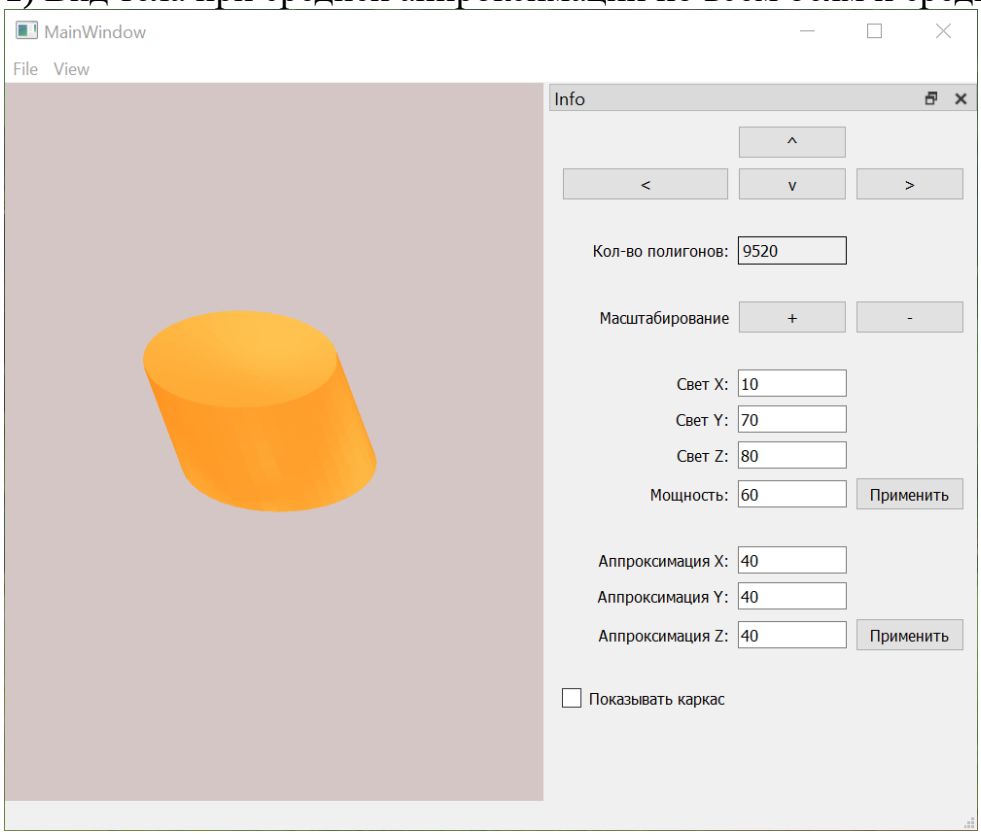
$$I_s = k_s \cos^{\alpha}(\vec{N},\ \vec{H})\, i_s \text{ , где}$$

H – вектор “медиана” угла между векторами V и L, вычисляется по формуле

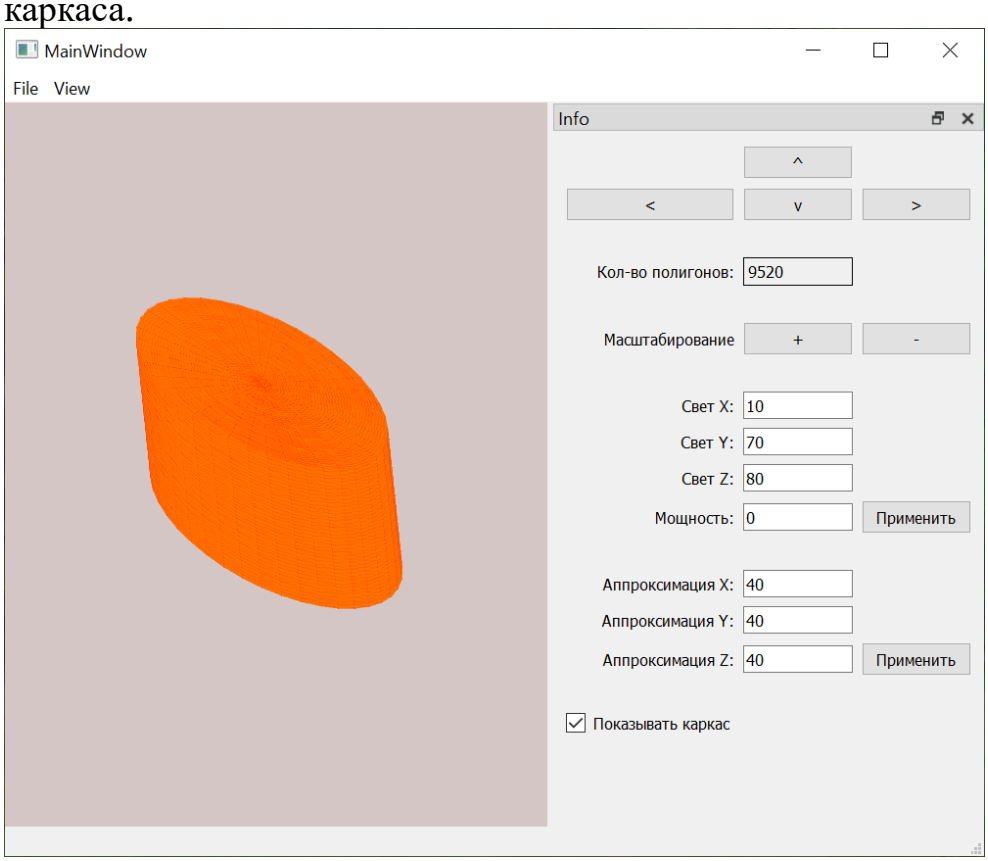
$$H = \frac{L + V}{|L + V|}$$

3. Демонстрация работы программы.

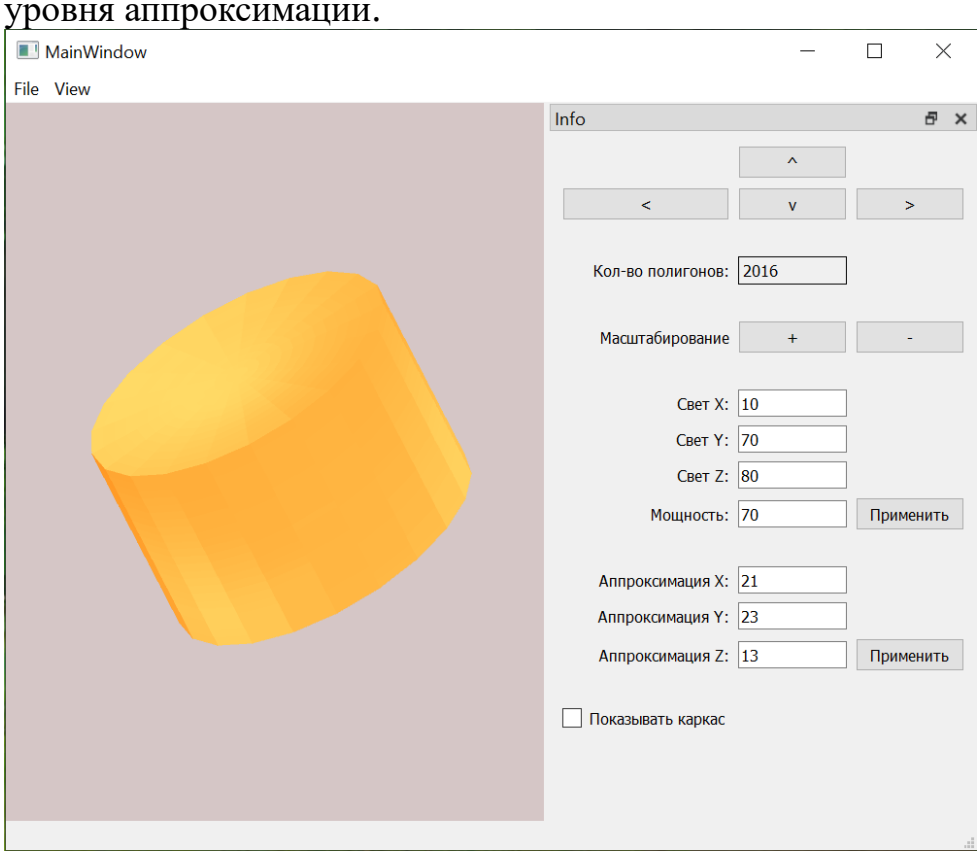
1) Вид тела при средней аппроксимации по всем осям и средней освещенности.



2) Виды тела после нескольких поворотов и отсутствии света, но с прорисовкой каркаса.



3) Вид тела при увеличенной мощности света, масштабировании и снижении уровня аппроксимации.



4. Основной код программы.

1) Метод класса `oblique_circular_cylinder` для формирования полигонов наклонного кругового цилиндра.

```
void oblique_circular_cylinder::create() {
    int edges = static_cast<int>(approximation_x);
    int radius = 10;
    double height = 10;
    QVector4D shift{2, 2, 0, 0};

    if (edges <= 0 || radius <= 0 || height <= 0) return;

    int layersNum = static_cast<int>(approximation_y);
    int circlesNum = static_cast<int>(approximation_z);

    // base points without relation to top or bottom
    QList<QVector2D> prismBasePoints;

    for (int i = 1; i <= edges; ++i)
    {
        double phi = (M_PI * 2 * i) / edges;
        prismBasePoints.append(radius * QVector2D {static_cast<float>(cos(phi)),
static_cast<float>(sin(phi))});
    }
    QList<QVector4D> toPushBack;
    // top base
    for (int i = 0; i < edges; i++) {
        // middle part
        polygons.push_back(std::vector<QVector4D>{
            QVector4D{0, 0, static_cast<float>(height / 2), 1} + shift,
            QVector4D{(prismBasePoints[(i + 1) % edges] / circlesNum),
static_cast<float>(height / 2), 1} + shift,
```

```
            QVector4D{(prismBasePoints[i]             /             circlesNum),
static_cast<float>(height / 2), 1} + shift});

        // area around middle point
        for (int c = 1; c < circlesNum; ++c) {
           polygons.push_back(std::vector<QVector4D>{
                QVector4D{(prismBasePoints[i]     /     circlesNum     *     c),
static_cast<float>(height / 2), 1} + shift,
                QVector4D{(prismBasePoints[(i + 1) % edges] / circlesNum * c),
static_cast<float>(height / 2), 1} + shift,
                QVector4D{(prismBasePoints[(i + 1) % edges] / circlesNum * (c +
1)), static_cast<float>(height / 2), 1} + shift});
            polygons.push_back(std::vector<QVector4D>{
                QVector4D{(prismBasePoints[i]     /     circlesNum     *     c),
static_cast<float>(height / 2), 1} + shift,
                QVector4D{(prismBasePoints[(i + 1) % edges] / circlesNum * (c +
1)), static_cast<float>(height / 2), 1} + shift,
                QVector4D{(prismBasePoints[i]   /   circlesNum   *   (c   +   1)),
static_cast<float>(height / 2), 1} + shift});
        }
    }

    // bottom base
    for (int i = 0; i < edges; i++) {
        // middle part
        polygons.push_back(std::vector<QVector4D>{
             QVector4D{0, 0, static_cast<float>(-height / 2), 1} - shift,
             QVector4D{(prismBasePoints[i]  /  circlesNum),  static_cast<float>(-
height / 2), 1} - shift,
             QVector4D{(prismBasePoints[(i   +   1)   %   edges]   /   circlesNum),
static_cast<float>(-height / 2), 1} - shift});

        // area around middle point
        for (int c = 1; c < circlesNum; ++c) {
            polygons.push_back(std::vector<QVector4D>{
                QVector4D{(prismBasePoints[i]     /     circlesNum     *     c),
static_cast<float>(-height / 2), 1} - shift,
                QVector4D{(prismBasePoints[(i + 1) % edges] / circlesNum * (c +
1)), static_cast<float>(-height / 2), 1} - shift,
                QVector4D{(prismBasePoints[(i + 1) % edges] / circlesNum * c),
static_cast<float>(-height / 2), 1} - shift});
            polygons.push_back(std::vector<QVector4D>{
                QVector4D{(prismBasePoints[i]     /     circlesNum     *     c),
static_cast<float>(-height / 2), 1} - shift,
                QVector4D{(prismBasePoints[i]   /   circlesNum   *   (c   +   1)),
static_cast<float>(-height / 2), 1} - shift,
                QVector4D{(prismBasePoints[(i + 1) % edges] / circlesNum * (c +
1)), static_cast<float>(-height / 2), 1} - shift});
        }
    }

    double heightStep = height / layersNum; // step of change in height
    QVector4D shiftStep = shift * 2 / layersNum; // step of shift change

    // polygons that form side surfaces (top is looking up)
    for (int i = 0; i < edges; ++i) {
        QVector4D s = -shiftStep * layersNum / 2;
        for (double h = 0; h < height - heightStep / 4; h += heightStep) {
            polygons.push_back(std::vector<QVector4D>{
                QVector4D{prismBasePoints[i], static_cast<float>(height / 2 - h),
1} - s,
                QVector4D{prismBasePoints[(i      +      1)      %      edges],
static_cast<float>(height / 2 - h - heightStep), 1} - s - shiftStep,
```

```
                QVector4D{prismBasePoints[i], static_cast<float>(height / 2 - h
- heightStep), 1} - s - shiftStep});
            s += shiftStep;
        }
    }

    // polygons that form side surfaces (top is looking down)
    for (int i = 0; i < edges; ++i) {
        QVector4D s = -shiftStep * layersNum / 2;
        for (double h = 0; h < height - heightStep / 4; h += heightStep) {
            polygons.push_back(std::vector<QVector4D>{
                QVector4D{prismBasePoints[i], static_cast<float>(height / 2 - h),
1} - s,
                QVector4D{prismBasePoints[(i      +      1)      %      edges],
static_cast<float>(height / 2 - h), 1} - s,
                QVector4D{prismBasePoints[(i      +      1)      %      edges],
static_cast<float>(height / 2 - h - heightStep), 1} - s - shiftStep});
            s += shiftStep;
        }
    }
}
```

2) Метод класса `polygon` для вычисления фоновой составляющей света.

```
int polygon::calc_ambient_component(light *light) {
    return static_cast<int>(ambient_coef * light->power);
}
```

3) Метод класса `polygon` для вычисления рассеянной составляющей света.

```
int polygon::calc_diffuse_component(int dx, int dy, light *light) {
    QVector3D tolight = QVector3D{
        light->position.x() - (vertices[0].x() + dx),
        light->position.y() - (vertices[0].y() + dy),
        light->position.z() - vertices[0].z()
    };
    QVector3D tolightNormalized = tolight.normalized();
    QVector3D normal = this->get_normal().normalized();
    double scalarProduct = static_cast<double>(tolightNormalized.x() * normal.x()
+
                                               tolightNormalized.y() * normal.y()
+
                                               tolightNormalized.z()           *
normal.z());

    int res = static_cast<int>(diffuse_coef * scalarProduct * light->power * 100
/ pow((static_cast<double>(tolight.length())), 1.2));
    if (res < 0) {
        res = 0;
    }
    return res;
}
```

4) Метод класса `polygon` для вычисления зеркальной составляющей света.

```
int polygon::calc_specular_component(int dx, int dy, light *light) {
    QVector3D tolight = QVector3D{
        light->position.x() - (vertices[0].x() + dx),
        light->position.y() - (vertices[0].y() + dy),
        light->position.z() - vertices[0].z()
    };
    QVector3D tolightNormalized = tolight.normalized();
    QVector3D   toObserver   =   QVector3D{0   -   (vertices[0].x()),   0   -
(vertices[0].y()), vertices[0].z()}.normalized();
```

```
    QVector3D median = (tolightNormalized + toObserver) / (tolightNormalized +
toObserver).length();
    QVector3D normal = this->get_normal().normalized();
    float scalarProduct = median.x() * normal.x() + median.y() * normal.y() +
median.z() * normal.z();
    int          res          =          static_cast<int>(specular_coef          *
pow(static_cast<double>(scalarProduct), gloss_coef) *
                                light->power          *          100          /
pow((static_cast<double>(tolight.length())), 1.2));
    if (res < 0) {
        res = 0;
    }
    return res;
}
```

5) Метод класса `polygon` для отрисовки с учётом трёх составляющих света.

```
void polygon::draw(QPainter *ptr, int center_x, int center_y, double
step_pixels_x, double step_pixels_y,
                   int window_center_x, int window_center_y, light *light,
                   bool displayCarcass) {
    QPen oldPen = ptr->pen();
    int resCalcAmbientComponent = calc_ambient_component(light);
    int    resCalcDiffuseComponent    =    calc_diffuse_component(center_x    -
window_center_x,
                                                              center_y              -
window_center_y, light);
    int   resCalcSpecularComponent   =   calc_specular_component(center_x   -
window_center_x,
                                                                center_y            -
window_center_y, light);
    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_x + center_x,
            static_cast<double>(vertices[i][1]) * step_pixels_y + 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_x + center_x),
```

```
                static_cast<int>(static_cast<double>(vertices[i][1])           *
step_pixels_y + center_y),
                static_cast<int>(static_cast<double>(vertices[(i + 1) % 3][0]) *
step_pixels_x + center_x),
                static_cast<int>(static_cast<double>(vertices[(i + 1) % 3][1]) *
step_pixels_y + center_y)
            );
        }
    }
}
```

5. Выводы.

В ходе данной лабораторной работы я смог аппроксимировать наклонный круговой цилиндр при помощи выпуклого многогранника. Кроме того, реализовал простую модель закраски полигонов тела при наличии в сцене одного источника света.