主函数

```
int main(int argc, const char** argv)
{
    std::vector<Triangle*> TriangleList;

    float angle = 140.0;
    bool command_line = false;

    std::string filename = "output.png";
    objl::Loader Loader;
    std::string obj_path = "./models/spot/";//模型路径

    // Load .obj File
    bool loadout = Loader.LoadFile("./models/spot/spot_triangulated_good.obj");
    for (auto mesh : Loader.LoadedMeshes)
    {
        for (int i = 0; i < mesh.Vertices.size(); i += 3)
        {
            Triangle* t = new Triangle();
            for (int j = 0; j < 3; j++)
            {
                t->setVertex(j, Vector4f(mesh.Vertices[i + j].Position.X, mesh.Vertices[i + j].Position.Y, mesh.Vertices[i + j].Position.Z, 1.0));
                t->setNormal(j, Vector3f(mesh.Vertices[i + j].Normal.X, mesh.Vertices[i + j].Normal.Y, mesh.Vertices[i + j].Normal.Z));
                t->setTexCoord(j, Vector2f(mesh.Vertices[i + j].TextureCoordinate.X, mesh.Vertices[i + j].TextureCoordinate.Y));
            }
            TriangleList.push_back(t);
        }
    }

    rst::rasterizer r(700, 700);

    auto texture_path = "spot_texture.png";//纹理路径
    //cout << obj_path + texture_path << endl;
    r.set_texture(Texture(obj_path + texture_path));

    //主要
    //normal_fragment_shader
    //phong_fragment_shader
    //texture_fragment_shader
    //着色器替换
    std::function<Eigen::Vector3f(fragment_shader_payload)> active_shader = normal_fragment_shader;

    if (argc >= 2)
    {
        command_line = true;
```

```
        filename = std::string(argv[1]);

        if (argc == 3 && std::string(argv[2]) == "texture")
        {
            std::cout << "使用纹理着色器的光栅化\n";
            active_shader = texture_fragment_shader;
            texture_path = "spot_texture.png";
            r.set_texture(Texture(obj_path + texture_path));
        }
        else if (argc == 3 && std::string(argv[2]) == "normal")
        {
            std::cout << "使用正常着色器的光栅化\n";
            active_shader = normal_fragment_shader;
        }
        else if (argc == 3 && std::string(argv[2]) == "phong")
        {
            std::cout << "使用phong着色器的光栅化\n";
            active_shader = phong_fragment_shader;
        }
    }

    Eigen::Vector3f eye_pos = { 0,0,10 };

    r.set_vertex_shader(vertex_shader);
    r.set_fragment_shader(active_shader);

    int key = 0;
    int frame_count = 0;

    if (command_line)
    {
        r.clear(rst::Buffers::Color | rst::Buffers::Depth);
        r.set_model(get_model_matrix(angle));
        r.set_view(get_view_matrix(eye_pos));
        r.set_projection(get_projection_matrix(45.0, 1, 0.1, 50));

        r.draw(TriangleList);
        cv::Mat image(700, 700, CV_32FC3, r.frame_buffer().data());
        image.convertTo(image, CV_8UC3, 1.0f);
        cv::cvtColor(image, image, cv::COLOR_RGB2BGR);

        cv::imwrite(filename, image);

        return 0;
    }

    while (key != 27)
    {
        r.clear(rst::Buffers::Color | rst::Buffers::Depth);

        r.set_model(get_model_matrix(angle));
        r.set_view(get_view_matrix(eye_pos));
        r.set_projection(get_projection_matrix(45.0, 1, 0.1, 50));

```

```
        //r.draw(pos_id, ind_id, col_id,
rst::Primitive::Triangle);
        r.draw(TriangleList);
        cv::Mat image(700, 700, CV_32FC3,
r.frame_buffer().data());
        image.convertTo(image, CV_8UC3, 1.0f);
        cv::cvtColor(image, image, cv::COLOR_RGB2BGR);

        cv::imshow("image", image);
        cv::imwrite(filename, image);
        key = cv::waitKey(10);

        if (key == 'a')
        {
            angle -= 10;
        }
        else if (key == 'd')
        {
            angle += 10;
        }

        cout <<"角度为: " << angle << endl;
    }
    return 0;
}
```

phong模型

```
Eigen::Vector3f phong_fragment_shader(const
fragment_shader_payload& payload)
{
    Eigen::Vector3f ka = Eigen::Vector3f(0.005, 0.005, 0.005);
    Eigen::Vector3f kd = payload.color;
    Eigen::Vector3f ks = Eigen::Vector3f(0.7937, 0.7937,
0.7937);

    auto l1 = light{ {20, 20, 20}, {500, 500, 500} };
    auto l2 = light{ {-20, 20, 0}, {500, 500, 500} };

    std::vector<light> lights = { l1, l2 };
    Eigen::Vector3f amb_light_intensity{ 10, 10, 10 };
    Eigen::Vector3f eye_pos{ 0, 0, 10 };

    float p = 150;

    Eigen::Vector3f color = payload.color;
    Eigen::Vector3f point = payload.view_pos;
    Eigen::Vector3f normal = payload.normal;

    Eigen::Vector3f result_color = { 0, 0, 0 };
    for (auto& light : lights)
    {
        // 光的方向
        Eigen::Vector3f light_dir = light.position - point;
        // 视线方向
        Eigen::Vector3f view_dir = eye_pos - point;
```

```
        // 衰减因子
        float r = light_dir.dot(light_dir);

        // ambient
        Eigen::Vector3f La = ka.cwiseProduct(amb_light_intensity);
        // diffuse
        Eigen::Vector3f Ld = kd.cwiseProduct(light.intensity / r);
        Ld *= std::max(0.0f, normal.normalized().dot(light_dir.normalized()));
        // specular
        Eigen::Vector3f h = (light_dir + view_dir).normalized();
        Eigen::Vector3f Ls = ks.cwiseProduct(light.intensity / r);
        Ls *= std::pow(std::max(0.0f, normal.normalized().dot(h)), p);

        result_color += (La + Ld + Ls);

    }

    return result_color * 255.f;
}
```

纹理映射

```

Eigen::Vector3f texture_fragment_shader(const fragment_shader_payload& payload)
{
    Eigen::Vector3f return_color = { 0, 0, 0 };
    if (payload.texture)
    {
        // TODO: 在当前的片元获取一个纹理坐标系的值
        // 获取纹理坐标的颜色
        return_color = payload.texture->getColor(payload.tex_coords.x(), payload.tex_coords.y());
    }
    Eigen::Vector3f texture_color;
    texture_color << return_color.x(), return_color.y(), return_color.z();

    Eigen::Vector3f ka = Eigen::Vector3f(0.005, 0.005, 0.005);//环境
    Eigen::Vector3f kd = texture_color / 255.f;//漫反射
    Eigen::Vector3f ks = Eigen::Vector3f(0.7937, 0.7937, 0.7937);//镜面反射

    auto l1 = light{ {20, 20, 20}, {500, 500, 500} };
    auto l2 = light{ {-20, 20, 0}, {500, 500, 500} };

    std::vector<light> lights = { l1, l2 };
    Eigen::Vector3f amb_light_intensity{ 10, 10, 10 };
    Eigen::Vector3f eye_pos{ 0, 0, 10 };
```

```
    float p = 150;

    Eigen::Vector3f color = texture_color;
    Eigen::Vector3f point = payload.view_pos;
    Eigen::Vector3f normal = payload.normal;

    Eigen::Vector3f result_color = { 0, 0, 0 };

    for (auto& light : lights)
    {
        // TODO: 对于代码中的每一个光源，计算漫反射，镜面反射，计算结果颜色
        Eigen::Vector3f light_dir = light.position - point;
        Eigen::Vector3f view_dir = eye_pos - point;
        float r = light_dir.dot(light_dir);
        // 环境光
        Eigen::Vector3f La = ka.cwiseProduct(amb_light_intensity);
        // 漫反射
        Eigen::Vector3f Ld = kd.cwiseProduct(light.intensity / r);
        Ld *= std::max(0.0f, normal.normalized().dot(light_dir.normalized()));
        // 镜面反射
        Eigen::Vector3f h = (light_dir + view_dir).normalized();
        Eigen::Vector3f Ls = ks.cwiseProduct(light.intensity / r);
        Ls *= std::pow(std::max(0.0f, normal.normalized().dot(h)), p);

        result_color += (La + Ld + Ls);

    }

    return result_color * 255.f;
}
```

通用着色器

```
Eigen::Vector3f normal_fragment_shader(const fragment_shader_payload& payload)
{
    Eigen::Vector3f return_color = (payload.normal.head<3>().normalized() + Eigen::Vector3f(1.0f, 1.0f, 1.0f)) / 2.f;
    Eigen::Vector3f result;
    result << return_color.x() * 255, return_color.y() * 255, return_color.z() * 255;
    return result;
}
```

MVP变换

```
Eigen::Matrix4f get_view_matrix(Eigen::Vector3f eye_pos)
```

```
{
    Eigen::Matrix4f view = Eigen::Matrix4f::Identity();

    Eigen::Matrix4f translate;
    translate << 1, 0, 0, -eye_pos[0],
        0, 1, 0, -eye_pos[1],
        0, 0, 1, -eye_pos[2],
        0, 0, 0, 1;

    view = translate * view;

    return view;
}

Eigen::Matrix4f get_model_matrix(float angle)
{
    Eigen::Matrix4f rotation;
    angle = angle * MY_PI / 180.f;
    rotation << cos(angle), 0, sin(angle), 0,
        0, 1, 0, 0,
        -sin(angle), 0, cos(angle), 0,
        0, 0, 0, 1;

    Eigen::Matrix4f scale;
    scale << 2.5, 0, 0, 0,
        0, 2.5, 0, 0,
        0, 0, 2.5, 0,
        0, 0, 0, 1;

    Eigen::Matrix4f translate;
    translate << 1, 0, 0, 0,
        0, 1, 0, 0,
        0, 0, 1, 0,
        0, 0, 0, 1;

    return translate * rotation * scale;
}

Eigen::Matrix4f get_projection_matrix(float eye_fov, float
aspect_ratio, float zNear, float zFar)
{

    Eigen::Matrix4f projection = Eigen::Matrix4f::Identity();
    Eigen::Matrix4f M_trans;
    Eigen::Matrix4f M_persp;
    Eigen::Matrix4f M_ortho;
    M_persp <<
        zNear, 0, 0, 0,
        0, zNear, 0, 0,
        0, 0, zNear + zFar, -zFar * zNear,
        0, 0, 1, 0;

    float alpha = 0.5 * eye_fov * MY_PI / 180.0f;
    float yTop = -zNear * std::tan(alpha); //
    float yBottom = -yTop;
    float xRight = yTop * aspect_ratio;
    float xLeft = -xRight;

```

```
    M_trans <<
        1, 0, 0, -(xLeft + xRight) / 2,
        0, 1, 0, -(yTop + yBottom) / 2,
        0, 0, 1, -(zNear + zFar) / 2,
        0, 0, 0, 1;
    M_ortho <<
        2 / (xRight - xLeft), 0, 0, 0,
        0, 2 / (yTop - yBottom), 0, 0,
        0, 0, 2 / (zNear - zFar), 0,
        0, 0, 0, 1;

    M_ortho = M_ortho * M_trans;
    projection = M_ortho * M_persp * projection;
    return projection;
}
```

光栅化

```
//平面坐标系的光栅
void rst::rasterizer::rasterize_triangle(const Triangle& t,
const std::array<Eigen::Vector3f, 3>& view_pos)
{



     //选取采样点
    auto v = t.toVector4();
    int min_x = INT_MAX;
    int max_x = INT_MIN;
    int min_y = INT_MAX;
    int max_y = INT_MIN;
    for (auto point : v)
    {
        if (point[0] < min_x) min_x = point[0];
        if (point[0] > max_x) max_x = point[0];
        if (point[1] < min_y) min_y = point[1];
        if (point[1] > max_y) max_y = point[1];
    }
    for (int y = min_y; y <= max_y; y++)
    {
        for (int x = min_x; x <= max_x; x++)
        {
            if (insideTriangle((float)x + 0.5, (float)y + 0.5,
t.v)) //以像素中心点作为采样点
            {

                //得到这个点的重心坐标
                auto abg = computeBarycentric2D((float)x + 0.5,
(float)y + 0.5, t.v);
                float alpha = std::get<0>(abg);
                float beta = std::get<1>(abg);
                float gamma = std::get<2>(abg);
                //z-buffer插值
                float w_reciprocal = 1.0 / (alpha / v[0].w() +
beta / v[1].w() + gamma / v[2].w()); //归一化系数
```

```
                float z_interpolated = alpha * v[0].z() / v[0].w() + beta * v[1].z() / v[1].w() + gamma * v[2].z() / v[2].w();
                z_interpolated *= w_reciprocal;

                if (z_interpolated < depth_buf[get_index(x, y)])
                {
                    //重点
                    Eigen::Vector2i p = { (float)x,(float)y };
                    // 颜色插值
                    auto interpolated_color = interpolate(alpha, beta, gamma, t.color[0], t.color[1], t.color[2], 1);
                    // 法向量插值
                    auto interpolated_normal = interpolate(alpha, beta, gamma, t.normal[0], t.normal[1], t.normal[2], 1);
                    // 纹理颜色插值
                    auto interpolated_texcoords = interpolate(alpha, beta, gamma, t.tex_coords[0], t.tex_coords[1], t.tex_coords[2], 1);
                    // 内部点位置插值
                    auto interpolated_shadingcoords = interpolate(alpha, beta, gamma, view_pos[0], view_pos[1], view_pos[2], 1);
                    fragment_shader_payload payload(interpolated_color, interpolated_normal.normalized(), interpolated_texcoords, texture ? &*texture : nullptr);
                    payload.view_pos = interpolated_shadingcoords;
                    auto pixel_color = fragment_shader(payload);
                    set_pixel(p, pixel_color); //设置颜色
                    depth_buf[get_index(x, y)] = z_interpolated;//更新z值
                }
            }
        }
    }
}
```

判断点位置

```
static bool insideTriangle(int x, int y, const Vector4f* _v) {
    Vector3f v[3];
    for (int i = 0; i < 3; i++)
        v[i] = { _v[i].x(),_v[i].y(), 1.0 };
    Vector3f f0, f1, f2;
    f0 = v[1].cross(v[0]);
    f1 = v[2].cross(v[1]);
    f2 = v[0].cross(v[2]);
    Vector3f p(x, y, 1.);
    if ((p.dot(f0) * f0.dot(v[2]) > 0) && (p.dot(f1) * f1.dot(v[0]) > 0) && (p.dot(f2) * f2.dot(v[1]) > 0))
        return true;
```

```
    return false;
}

```

三角形类

```

Triangle::Triangle() {
    v[0] << 0, 0, 0, 1;
    v[1] << 0, 0, 0, 1;
    v[2] << 0, 0, 0, 1;

    color[0] << 0.0, 0.0, 0.0;
    color[1] << 0.0, 0.0, 0.0;
    color[2] << 0.0, 0.0, 0.0;

    tex_coords[0] << 0.0, 0.0;
    tex_coords[1] << 0.0, 0.0;
    tex_coords[2] << 0.0, 0.0;
}

void Triangle::setVertex(int ind, Vector4f ver) {
    v[ind] = ver;
}
void Triangle::setNormal(int ind, Vector3f n) {
    normal[ind] = n;
}
void Triangle::setColor(int ind, float r, float g, float b) {
    if ((r < 0.0) || (r > 255.) ||
        (g < 0.0) || (g > 255.) ||
        (b < 0.0) || (b > 255.)) {
        fprintf(stderr, "ERROR! Invalid color values");
        fflush(stderr);
        exit(-1);
    }

    color[ind] = Vector3f((float)r / 255., (float)g / 255.,
(float)b / 255.);
    return;
}
void Triangle::setTexCoord(int ind, Vector2f uv) {
    tex_coords[ind] = uv;
}

std::array<Vector4f, 3> Triangle::toVector4() const
{
    std::array<Vector4f, 3> res;
    std::transform(std::begin(v), std::end(v), res.begin(), []
(auto& vec) { return Vector4f(vec.x(), vec.y(), vec.z(), 1.f);
});
    return res;
}

void Triangle::setNormals(const std::array<Vector3f, 3>&
normals)
{

```

```
    normal[0] = normals[0];
    normal[1] = normals[1];
    normal[2] = normals[2];
}

void Triangle::setColors(const std::array<Vector3f, 3>& colors)
{
    auto first_color = colors[0];
    setColor(0, colors[0][0], colors[0][1], colors[0][2]);
    setColor(1, colors[1][0], colors[1][1], colors[1][2]);
    setColor(2, colors[2][0], colors[2][1], colors[2][2]);
}
```

纹理插值

```
static Eigen::Vector2f interpolate(float alpha, float beta, float gamma, const Eigen::Vector2f& vert1, const Eigen::Vector2f& vert2, const Eigen::Vector2f& vert3, float weight)
{
    auto u = (alpha * vert1[0] + beta * vert2[0] + gamma * vert3[0]);
    auto v = (alpha * vert1[1] + beta * vert2[1] + gamma * vert3[1]);

    u /= weight;
    v /= weight;

    return Eigen::Vector2f(u, v);
}
```

重心计算

```
static Eigen::Vector3f interpolate(float alpha, float beta, float gamma, const Eigen::Vector3f& vert1, const Eigen::Vector3f& vert2, const Eigen::Vector3f& vert3, float weight)
{
    return (alpha * vert1 + beta * vert2 + gamma * vert3) / weight;
}
```