Computer Graphic PA1

王子轩
2023011307
wang-zx23@mails.tsinghua.edu.cn

2025/4/11

1 Algorithm Implemention

1.1 主函数光线投射基本原理

通过模拟光线在场景中的传播来计算每个像素的颜色. 基本思想是从视点 (相机) 向场景中的每个像素发射光线, 然后追踪这些光线与场景中物体的交点, 并使用 Phong 光照模型计算这些交点处的光照效果, 最后将计算得到的颜色值赋给对应的像素.

Algorithm 1 Basic Ray Tracing Algorithm

Require: Scene description, camera parameters, output image dimensions
Ensure: Rendered image

```
Parse scene, obtain objects and light sources
Initialize output image
for each pixel (x, y) do
    Generate ray from camera toward pixel ray
    Initialize nearest intersection info hit
    Compute ray intersection with all scene objects
    if intersection exists then
        Initialize pixel color color = (0, 0, 0)
        for each light source in scene do
            Calculate light illumination at intersection point
            Compute color contribution using Phong model
            Add color contribution to color
        end for
        Assign color to pixel (x, y)
    else
        Assign background color to pixel (x, y)
    end if
end for
Output rendered image
```

Listing 1 实现主渲染循环

```
int main(int argc, char *argv[]) {
    for (int argNum = 1; argNum < argc; ++argNum) {
        std::cout << "Argument " << argNum << " is: " << argv[argNum] << std::endl;
    }
    if (argc != 3) {
        cout << "Usage: ./bin/PA1 <input scene file> <output bmp file>" << endl;
        return 1;
    }
    string inputFile = argv[1];
    string outputFile = argv[2];
    SceneParser sceneParser(inputFile.c_str());
    Camera* camera = sceneParser.getCamera();
    int width = camera->getWidth();
    int height = camera->getHeight();
    Image image(width, height);
    for (int x = 0; x < camera->getWidth(); ++x) {
        for (int y = 0; y < camera->getHeight(); ++y) {
            // 生成从相机到像素的光线
            Ray camRay = camera->generateRay(Vector2f(x, y));
            Group* baseGroup = sceneParser.getGroup();
            Hit hit;
            bool isIntersect = baseGroup->intersect(camRay, hit, 0);
            if (isIntersect) {
                // 计算交点处的光照效果
                Vector3f finalColor = Vector3f::ZERO;
                for (int li = 0; li < sceneParser.getNumLights(); ++li) {
                    Light* light = sceneParser.getLight(li);
                    Vector3f L, lightColor;
                    light->getIllumination(camRay.pointAtParameter(hit.getT()), L, lightColor);
                    finalColor += hit.getMaterial()->Shade(camRay, hit, L, lightColor);
                }
                image.SetPixel(x, y, finalColor);
            } else {
                image.SetPixel(x, y, sceneParser.getBackgroundColor());
            }
        }
    }
    image.SaveBMP(outputFile.c_str());
    return 0;
}
```

1.2 光线与物体求交

光线投射的核心是计算光线与场景中各种几何体的交点. 在本次实现中, 我们实现了三种基本几何体的求交算法: 平面, 球体和三角形.

1.2.1 光线与平面求交

平面可以用点法式方程表示:$\vec{n} \cdot \vec{p} = d$, 其中 $\vec{n}$ 是平面的法向量,d 是平面到原点的有向距离. 光线可以表示为:$\vec{r}(t) = \vec{o} + t\vec{d}$, 其中 $\vec{o}$ 是光线的起点,$\vec{d}$ 是光线的方向,t 是参数. 光线方程代入平面方程, 可以求解交点参数 t:

$$t = \frac{d - \vec{n} \cdot \vec{o}}{\vec{n} \cdot \vec{d}} \tag{1}$$

Algorithm 2 Ray-Plane Intersection Algorithm

Require: Ray ray, plane normal vector $normal$, plane constant d, minimum valid distance $tmin$
Ensure: Whether intersection exists, intersection information

Calculate denominator $denominator = normal \cdot ray.direction$
if $|denominator| < \epsilon$ **then**
  return false {Ray is parallel to plane, no intersection}
end if
Calculate intersection parameter $t = \frac{d - normal \cdot ray.origin}{denominator}$
if $t \geq tmin$ AND $t < hit.t$ **then**
  Update intersection information hit
  return true
else
  return false
end if

Listing 2 实现 Plane::intersect

```
bool intersect(const Ray &r, Hit &h, float tmin) override {
    float denominator = Vector3f::dot(normal, r.getDirection());
    // 如果光线方向与平面法线平行或几乎平行，则不相交
    if (fabs(denominator) < 1e-8) {
        return false;
    }
    // 计算交点参数 t
    float t = (d - Vector3f::dot(normal, r.getOrigin())) / denominator;
    // 如果 t 在有效范围内，则有交点
    if (t >= tmin && t < h.getT()) {
        h.set(t, material, normal);
        return true;
    }
    return false;
}
```

1.2.2 光线与球体求交

球体可以表示为:$|\vec{p} - \vec{c}|^2 = r^2$, 其中 $\vec{c}$ 是球心,r 是半径. 将光线方程代入球体方程, 可以得到一个关于 t 的二次方程:

$$|\vec{o} + t\vec{d} - \vec{c}|^2 = r^2 \tag{2}$$

展开后得到:

$$|\vec{d}|^2 t^2 + 2\vec{d} \cdot (\vec{o} - \vec{c})t + |\vec{o} - \vec{c}|^2 - r^2 = 0 \tag{3}$$

通过求解这个二次方程, 可以得到光线与球体的交点参数 t.

Algorithm 3 Ray-Sphere Intersection Algorithm

Require: Ray ray, sphere center $center$, radius $radius$, minimum distance $tmin$
Ensure: Intersection result and information

$oc = ray.origin - center$
$a = |ray.direction|^2$
$b = 2(oc \cdot ray.direction)$
$c = |oc|^2 - radius^2$
$\Delta = b^2 - 4ac$
if $\Delta < 0$ **then**
  return false
end if
$t = \frac{-b-\sqrt{\Delta}}{2a}$
if $t < tmin$ **then**
  $t = \frac{-b+\sqrt{\Delta}}{2a}$
  if $t < tmin$ **then**
    return false
  end if
end if
if $t < hit.t$ **then**
  $p = ray.origin + t \cdot ray.direction$
  $normal = \frac{p-center}{radius}$
  Update hit with t, $material$, $normal$
  return true
end if
return false

Listing 3 实现 Sphere::intersect

```
bool intersect(const Ray &r, Hit &h, float tmin) override {
    Vector3f oc = r.getOrigin() - center;
    float a = r.getDirection().squaredLength();
    float b = 2.0f * Vector3f::dot(oc, r.getDirection());
    float c = oc.squaredLength() - radius * radius;
    float discriminant = b * b - 4 * a * c;
    if (discriminant < 0) {
        return false;
    }
    float sqrtDiscriminant = sqrt(discriminant);
    float t = (-b - sqrtDiscriminant) / (2.0f * a);
    if (t < tmin) {
        t = (-b + sqrtDiscriminant) / (2.0f * a);
        if (t < tmin) {
            return false;
        }
    }
    if (t < h.getT()) {
        Vector3f intersectionPoint = r.pointAtParameter(t);
        Vector3f normal = (intersectionPoint - center).normalized();
        h.set(t, material, normal);
        return true;
    }
    return false;
}
```

1.2.3 光线与三角形求交

三角形是构建复杂 3D 模型的基本单元. 我们采用 Möller-Trumbore 算法进行光线与三角形的求交计算, 这是一种不需要显式计算三角形所在平面方程的高效算法. 该算法使用重心坐标系统, 直接计算交点的重心坐标和参数值. 算法首先计算光线方向与三角形一条边的叉积, 用于判断光线是否与三角形平行. 然后通过一系列向量运算得到交点的重心坐标 (u, v), 这两个参数必须满足 $u \geq 0$,$v \geq 0$ 且 $u + v \leq 1$ 才表示交点在三角形内部. 这种方法比传统的先求平面交点再判断点是否在三角形内的方法更加高效.

Algorithm 4 Möller-Trumbore Ray-Triangle Intersection Algorithm

Require: Ray ray, triangle vertices $v0, v1, v2$, minimum distance $tmin$
Ensure: Intersection result and information

$edge1 = v1 - v0$ $edge2 = v2 - v0$
$pvec = ray.direction \times edge2$ $det = edge1 \cdot pvec$
if $|det| < \epsilon$ **then**
  return false {Ray parallel to triangle}
end if
$invDet = 1/det$
$tvec = ray.origin - v0$
$u = (tvec \cdot pvec) \cdot invDet$
if $u < 0$ OR $u > 1$ **then**
  return false
end if
$qvec = tvec \times edge1$
$v = (ray.direction \cdot qvec) \cdot invDet$
if $v < 0$ OR $u + v > 1$ **then**
  return false
end if
$t = (edge2 \cdot qvec) \cdot invDet$
if $t \geq tmin$ AND $t < hit.t$ **then**
  Update intersection information hit
  return true
else
  return false
end if

Listing 4 实现 Triangle::intersect

```
    bool intersect(const Ray& ray, Hit& hit, float tmin) override {
        Vector3f edge1 = vertices[1] - vertices[0];
        Vector3f edge2 = vertices[2] - vertices[0];
        Vector3f pvec = Vector3f::cross(ray.getDirection(), edge2);
        float det = Vector3f::dot(edge1, pvec);
        if (fabs(det) < 1e-8) return false;
        float invDet = 1.0f / det;
        Vector3f tvec = ray.getOrigin() - vertices[0];
        float u = Vector3f::dot(tvec, pvec) * invDet;
        if (u < 0.0f || u > 1.0f) return false;
        Vector3f qvec = Vector3f::cross(tvec, edge1);
        float v = Vector3f::dot(ray.getDirection(), qvec) * invDet;
        if (v < 0.0f || u + v > 1.0f) return false;
        float t = Vector3f::dot(edge2, qvec) * invDet;
        if (t >= tmin && t < hit.getT()) {
            hit.set(t, material, normal);
            return true;
        }
        return false;
    }
```

1.3 相机模型

基类 Camera 定义基本框架, 子类 PerspectiveCamera 实现透视投影功能. 相机构造时建立了一个正交坐标系: 通过输入的中心位置, 观察方向和上方向, 计算出三个相互垂直的单位向量 (观察方向, 水平方向和上方向) 作为相机的局部坐标系. 透视相机额外存储视场角及其正切值以提高计算效率. 光线生成过程首先将像素坐标转换为归一化设备坐标, 考虑了视场角和宽高比的影响. 然后将这些坐标映射到相机坐标系中, 通过线性组合观察方向, 水平方向和上方向向量, 计算出光线的方向向量. 最后创建一条从相机中心出发, 沿计算方向的光线. 这种方法准确模拟了透视效果, 使远处物体看起来比近处物体小, 产生真实的 3D 视觉效果.

Listing 5 实现 PerspectiveCamera

```
    class PerspectiveCamera : public Camera {

public:
    PerspectiveCamera(const Vector3f &center, const Vector3f &direction,
            const Vector3f &up, int imgW, int imgH, float angle) : Camera(center, direction, up, imgW, i
        this->angle = angle;
        this->tanAngle = tan(angle / 2);
    }
    Ray generateRay(const Vector2f &point) override {
        float x = (2.0f * point.x() / width - 1) * tanAngle * width / height;
        float y = -(1 - 2.0f * point.y() / height) * tanAngle;
        Vector3f rayDir = direction + x * horizontal + y * up;
        rayDir = rayDir.normalized();
        return Ray(center, rayDir);
    }
```

1.4 光照模型

1.5 物体组场景管理

为了高效地管理场景中的多个物体, 我们实现了 Group 类, 它可以包含多个 Object3D 对象, 并提供统一的求交接口.

Listing 6 实现 Group::intersect

```
    bool intersect(const Ray &r, Hit &h, float tmin) override {
        bool result = false;
        for (auto obj : objects) {
            if (obj && obj->intersect(r, h, tmin)) {
                result = true;
            }
        }
        return result;
    }
```

1.6 光照模型

我们使用 Phong 光照模型来计算交点处的光照效果.Phong 模型包括三个组成部分: 环境光, 漫反射和镜面反射.

Algorithm 5 Phong Illumination Model

Require: Incident ray ray, intersection information hit, light direction $dirToLight$, light color $lightColor$
Ensure: Computed color

1: Get surface normal $N = hit.normal$
2: Calculate diffuse coefficient $diffuseTerm = \max(0, dirToLight \cdot N)$
3: Calculate reflection vector $R = 2(dirToLight \cdot N)N - dirToLight$
4: Calculate view vector $V = -ray.direction$
5: Calculate specular coefficient $specularTerm = \max(0, V \cdot R)^{shininess}$
6: Compute final color $color = lightColor \cdot diffuseColor \cdot diffuseTerm + lightColor \cdot specularColor \cdot specularTerm$
7: **return** $color$

Listing 7 实现 Material::Shade

```
Vector3f Shade(const Ray &ray, const Hit &hit,
               const Vector3f &dirToLight, const Vector3f &lightColor) {
    Vector3f shaded = Vector3f::ZERO;
    Vector3f N = hit.getNormal();
    float diffuseTerm = std::max(0.0f, Vector3f::dot(dirToLight, N));
    Vector3f L = dirToLight;
    Vector3f R = (2.0f * Vector3f::dot(L, N) * N - L).normalized();
    Vector3f V = -ray.getDirection().normalized();
    float specularTerm = std::pow(std::max(0.0f, Vector3f::dot(V, R)), shininess);
    shaded = lightColor * diffuseColor * diffuseTerm +
             lightColor * specularColor * specularTerm;

    return shaded;
}
```

2 Honor Code

本次实现参考了课程提供的框架代码和 PPT, 在实现过程中, 我与 Claude Sonnet 3.7 模型进行了关于算法思路的交互，并独立完成了所有 TODO 部分的代码编写, 通过自己的理解实现各个组件.

3 Test

通过了 OJ 自动评测, 并自己构造了如下测试例子