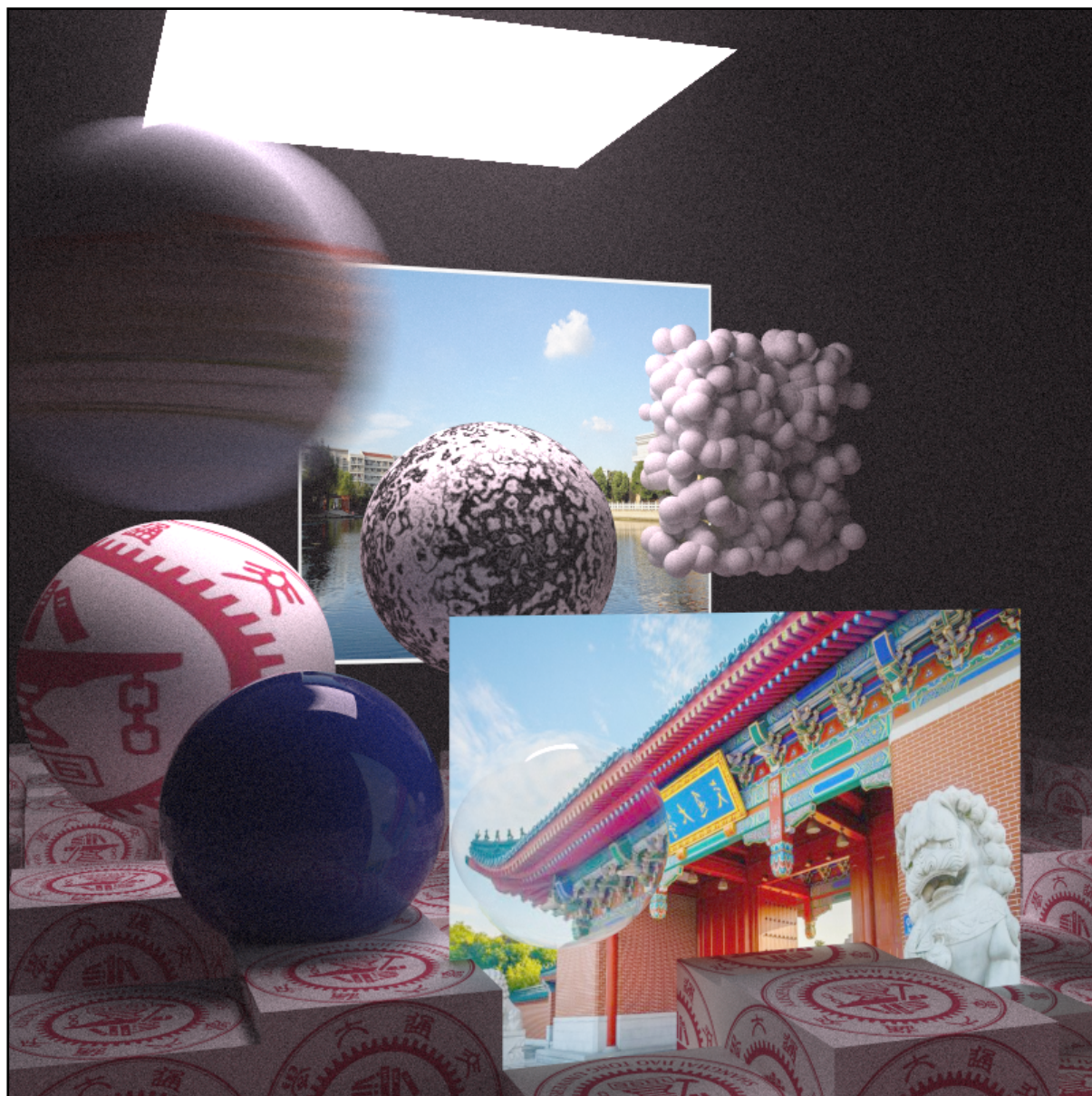


# 光线追踪

常烁晨-521021910369

## 1、介绍



光线追踪是一种通过模拟光线传播和与场景中对象的交互来生成图像的技术。这种方法能够产生高度逼真的图像，因为它能够精确地模拟光线如何在真实世界中传播，包括反射、折射、散射等物理现象。

在C++实现的光线追踪器中，核心原理是计算从视点出发，穿过每个像素点的光线与场景中对象的交互。这些交互通常包括但不限于：光线与物体表面的相交检测、光线在不同材质上的反射和折射行为，以及光源对物体表面的照明效果。计算结果以颜色值的形式存储，形成最终图像。

本项目使用基于C++实现的RayTracing模型，为了与交大元素相结合，本项目在图片的渲染过程中加入交大元素的纹理信息，如将交大庙门、致远湖图片作为四边形光源的纹理贴图，在渲染场景中展示出“放映屏幕”的视觉效果，与此同时，使用交大校徽图案作为场景中某个球型以及复杂地板的纹理贴图，可以看到在渲染的场景中有明显的交大校徽元素。

## 2、相关头文件的实现

### 2.1 vec3.h

头文件 `vec3.h` 定义了一个用于光线追踪器的基础数学类 `vec3`，这个类是3维向量（或点）的表示。该文件是整个光线追踪器的基础内容，定义了最小粒度上的渲染行为。

以下介绍`vec3.h`文件的核心定义内容。

```

class vec3 {
public:
    double e[3];

    vec3() : e{0,0,0} {}
    vec3(double e0, double e1, double e2) : e{e0, e1, e2} {}

    double x() const { return e[0]; }
    double y() const { return e[1]; }
    double z() const { return e[2]; }

    vec3 operator-() const { return vec3(-e[0], -e[1], -e[2]); }
    double operator[](int i) const { return e[i]; }
    double& operator[](int i) { return e[i]; }

    vec3& operator+=(const vec3 &v) {
        e[0] += v.e[0];
        e[1] += v.e[1];
        e[2] += v.e[2];
        return *this;
    }
}

```

此处定义了一个类 `vec3`，它代表了一个三维空间中的向量或点。该类包含了一个三元素数组 `e`，用于存储向量的 `x`, `y`, `z` 分量。在图形学光追器中，本文件所定义的是三维空间中的点坐标，并提供了构造函数用于初始化这些值，以及 `x()`, `y()`, `z()` 成员函数用于访问这些分量。此外，本项目还在该文件中定义了一系列三维点坐标（向量）的运算效果。

## 2.2 color.h

头文件 `color.h` 专注于处理与颜色相关的操作，特别是在光线追踪中处理像素颜色的转换和输出。容易想到，我们可以将前面定义的三维点直接应用于颜色（RGB）。

```

void write_color(std::ostream &out, color pixel_color, int samples_per_pixel) {
    auto r = pixel_color.x();
    auto g = pixel_color.y();
    auto b = pixel_color.z();

    auto scale = 1.0 / samples_per_pixel;
    r *= scale;
    g *= scale;
    b *= scale;

    r = linear_to_gamma(r);
    g = linear_to_gamma(g);
    b = linear_to_gamma(b);

    static const interval intensity(0.000, 0.999);
    out << static_cast<int>(256 * intensity.clamp(r)) << ' '
        << static_cast<int>(256 * intensity.clamp(g)) << ' '
        << static_cast<int>(256 * intensity.clamp(b)) << '\n';
}

```

color.h区别于vec3.h的关键是，此处定义了一个函数 write\_color，负责将三维的RGB颜色值输出到流中。该函数首先对颜色分量进行比例缩放，使用 linear\_to\_gamma 函数进行颜色调整，最后将颜色值限制在0到255中，作为.ppm文件的输出，以便于在标准的数字图像格式中表示。

### 2.3 ray.h

头文件 ray.h 定义了一个对光线追踪至关重要的类 ray。这个类用于表示和操作光线，这是光线追踪算法的核心。光线追踪算法的原理正是从摄像机出发，向渲染空间中的光线行为进行追踪计算（如折射、反射等光线行为）。

```

class ray {
public:
    ray() {}

    ray(const point3& origin, const vec3& direction) : orig(origin), dir(direction), tm(0)
    {}

    ray(const point3& origin, const vec3& direction, double time)
    : orig(origin), dir(direction), tm(time)
    {}

    point3 origin() const { return orig; }
    vec3 direction() const { return dir; }
    double time() const { return tm; }

    point3 at(double t) const {
        return orig + t*dir;
    }
}

```

这部分代码定义了类 ray，它用于表示光线。光线在光追器中被定义为一个起点 orig 朝向方向 dir 的一条射线。在这里 vec3 类型（分别作为三维点、三维向量）用于表示光线的起点和方向。同时，时间变量 tm 用于模拟时间相关的效果，例如运动模糊。

成员函数 at 根据参数 t 来计算并返回光线上的一点。在光线追踪中，这个函数用于确定某个时刻 t 下，光线与场景中对象的相交点，即通过改变 t 的值来沿光线追踪的行为。

## 2.4 material.h

头文件 material.h 定义了一系列材质类，这些类是实现光线追踪中不同物体表面效果的关键。

首先是基类：



```

class material {
public:
    virtual ~material() = default;

    virtual color emitted(double u, double v, const point3& p) const {
        return color(0,0,0);
    }

    virtual bool scatter(
        const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered
    ) const = 0;
};

```

material 类是一个抽象基类，它定义了所有材质必须实现的接口。每种材质需要定义如何散射（scatter）入射光线，以及如何发射（emitted）光线。这些函数对于模拟不同类型的物理材质（如漫反射、金属、玻璃等）的光学特性至关重要。scatter 函数负责确定光线击中物体后的行为，如反射、折射或吸收，并相应地更新散射后的光线和颜色衰减。emitted 函数用于模拟材质自身发光的特性。

漫反射类：

```

class lambertian : public material {
public:
    lambertian(const color& a) : albedo(make_shared<solid_color>(a)) {}
    lambertian(shared_ptr<texture> a) : albedo(a) {}

    bool scatter(const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered)
    const override {
        auto scatter_direction = rec.normal + random_unit_vector();

        // Catch degenerate scatter direction
        if (scatter_direction.near_zero())
            scatter_direction = rec.normal;

        scattered = ray(rec.p, scatter_direction, r_in.time());
        attenuation = albedo->value(rec.u, rec.v, rec.p);
        return true;
    }

private:
    shared_ptr<texture> albedo;
};

```

这种材质以相同的强度向所有方向散射光线，其散射模型基于朗伯定律（Lambert's cosine law）。在 `scatter` 方法中，`lambertian` 材质计算一个新的散射方向，这个方向在击中点的法线方向附近随机选择。这种随机性给出了典型的漫反射外观。

金属类：

```
class metal : public material {
public:
    metal(const color& a, double f) : albedo(a), fuzz(f < 1 ? f : 1) {}

    bool scatter(const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered)
    const override {
        vec3 reflected = reflect(unit_vector(r_in.direction()), rec.normal);
        scattered = ray(rec.p, reflected + fuzz*random_in_unit_sphere(), r_in.time());
        attenuation = albedo;
        return (dot(scattered.direction(), rec.normal) > 0);
    }

private:
    color albedo;
    double fuzz;
};
```

这种材质主要进行反射光线。反射角度基于入射光线和表面法线，遵循反射定律。在其 `scatter` 方法中，`metal` 材质首先计算光线的反射方向，此后在此方向上添加一些随机性（由 `fuzz` 参数控制），模拟微小的表面粗糙度。

透明材料：

```

class dielectric : public material {
public:
    dielectric(double index_of_refraction) : ir(index_of_refraction) {}

    bool scatter(const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered)
    const override {
        attenuation = color(1.0, 1.0, 1.0);
        double refraction_ratio = rec.front_face ? (1.0/ir) : ir;

        vec3 unit_direction = unit_vector(r_in.direction());
        double cos_theta = fmin(dot(-unit_direction, rec.normal), 1.0);
        double sin_theta = sqrt(1.0 - cos_theta*cos_theta);

        bool cannot_refract = refraction_ratio * sin_theta > 1.0;
        vec3 direction;

        if (cannot_refract || reflectance(cos_theta, refraction_ratio) > random_double())
            direction = reflect(unit_direction, rec.normal);
        else
            direction = refract(unit_direction, rec.normal, refraction_ratio);

        scattered = ray(rec.p, direction, r_in.time());
        return true;
    }
};

```

透明材料既可以折射光线也可以反射光线。折射比率取决于材料的折射指数（ir）。在 scatter 方法中，dielectric 材质根据斯涅尔定律（Snell's law）计算折射角度，同时使用菲涅尔方程（Fresnel equations）来确定在特定角度下光线是反射还是折射。

光源：

```

class diffuse_light : public material {
public:
    diffuse_light(shared_ptr<texture> a) : emit(a) {}
    diffuse_light(color c) : emit(make_shared<solid_color>(c)) {}

    bool scatter(const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered)
    const override {
        return false;
    }

    color emitted(double u, double v, const point3& p) const override {
        return emit->value(u, v, p);
    }

private:
    shared_ptr<texture> emit;
};

```



这种材质本身发出光线，而不是从其他光源反射光线。实现：diffuse\_light 重写了 emitted 方法来返回光源的颜色，而其 scatter 方法通常返回 false，表示光线不从此材质反射。

云雾：

```
class isotropic : public material {
public:
    isotropic(color c) : albedo(make_shared<solid_color>(c)) {}
    isotropic(shared_ptr<texture> a) : albedo(a) {}

    bool scatter(const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered)
    const override {
        scattered = ray(rec.p, random_unit_vector(), r_in.time());
        attenuation = albedo->value(rec.u, rec.v, rec.p);
        return true;
    }

private:
    shared_ptr<texture> albedo;
};
```

isotropic 材质是一种特殊的材质，它在所有方向上均匀地散射光线。这种材质通常用于模拟均匀的雾或云雾等效果。在 scatter 方法中，isotropic 材质通过在单位球内随机选择一个新方向，产生一个全方向均匀的散射。

## 2.5 hittable.h

hittable.h 是光线追踪器中的一个关键组成部分，它定义了可被光线击中（hit）的对象的接口和相关处理。

首先代码中定义了 hit\_record，作为光线击中对象的存储信息。

```

class hit_record {
public:
    point3 p;
    vec3 normal;
    shared_ptr<material> mat;
    double t;
    double u;
    double v;
    bool front_face;

    void set_face_normal(const ray& r, const vec3& outward_normal) {
        front_face = dot(r.direction(), outward_normal) < 0;
        normal = front_face ? outward_normal : -outward_normal;
    }
};

```

record中存储了击中点的位置（p）、击中点处的表面法线（normal）、击中的材质（mat）、光线参数 t（确定击中位置）、纹理坐标（u 和 v）、法线的朝向函数set\_face\_normal（），用于确定光线从物体外部或者内部击中。

```

class hittable {
public:
    virtual ~hittable() = default;

    virtual bool hit(const ray& r, interval ray_t, hit_record& rec) const = 0;

    virtual aabb bounding_box() const = 0;
};

```

hittable 是一个抽象类，它定义了所有可被光线击中对象的基本接口。这个接口包括一个 hit 函数，它接收一条光线和一个时间区间，然后返回是否有击中发生，并在有击中时填充 hit\_record 结构。此外还有hittable\_list提供了一种方式来管理一组可被光线击中的对象（即 hittable 对象）。

```

class hittable_list : public hittable {
public:
    std::vector<shared_ptr<hittable>> objects;

    hittable_list() {}
    hittable_list(shared_ptr<hittable> object) { add(object); }

    void clear() { objects.clear(); }

    void add(shared_ptr<hittable> object) {
        objects.push_back(object);
        bbox = aabb(bbox, object->bounding_box());
    }

    bool hit(const ray& r, interval ray_t, hit_record& rec) const override {

```

## 2.6 camera.h

头文件 camera.h 定义了一个名为 camera 的类，它用于在光线追踪器中创建和管理虚拟相机的设置。

首先头文件中定义了一系列相机参数，封装了虚拟相机的所有属性和行为，包括视角、像素大小、采样率、递归深度等。

```

public:
    double aspect_ratio    = 1.0;    // Ratio of image width over height
    int    image_width     = 100;    // Rendered image width in pixel count
    int    samples_per_pixel = 10;    // Count of random samples for each pixel
    int    max_depth       = 10;    // Maximum number of ray bounces into scene
    color  background;      // Scene background color

    double vfov    = 90;    // Vertical view angle (field of view)
    point3 lookfrom = point3(0,0,-1); // Point camera is looking from
    point3 lookat  = point3(0,0,0);  // Point camera is looking at
    vec3   vup     = vec3(0,1,0);    // Camera-relative "up" direction

    double defocus_angle = 0; // Variation angle of rays through each pixel
    double focus_dist    = 10; // Distance from camera lookfrom point to plane of perfect focus

```

紧接着是相机的渲染核心函数功能实现。render 方法是类的核心功能，它负责生成整个场景的图像。该方法遍历每个像

素，对每个像素点采样多次（由 `samples_per_pixel` 控制），并计算每个采样点的光线颜色。然后，这些颜色被平均并写入输出文件，生成最终的图像。

```
void render(const hittable& world) {
    initialize();

    outFile << "P3\n" << image_width << ' ' << image_height << "\n255\n";

    for (int j = 0; j < image_height; ++j) {
        std::clog << "\rScanlines remaining: " << (image_height - j) << ' ' << std::flush;
        for (int i = 0; i < image_width; ++i) {
            color pixel_color(0,0,0);
            for (int sample = 0; sample < samples_per_pixel; ++sample) {
                ray r = get_ray(i, j);
                pixel_color += ray_color(r, max_depth, world);
            }
            write_color(outFile, pixel_color, samples_per_pixel);
        }
    }
    outFile.close();
    std::clog << "\rDone.          \n";
}
```

## 2.7 sphere.h等

头文件 `sphere.h` 定义了一个名为 `sphere` 的类，它是 `hittable` 类的一个派生类，专门用于表示和处理球体对象。

```
class sphere : public hittable {
public:
    // Stationary Sphere
    sphere(point3 _center, double _radius, shared_ptr<material> _material)
        : center1(_center), radius(_radius), mat(_material), is_moving(false)
    {
        auto rvec = vec3(radius, radius, radius);
        bbox = aabb(center1 - rvec, center1 + rvec);
    }

    // Moving Sphere
    sphere(point3 _center1, point3 _center2, double _radius, shared_ptr<material> _material)
        : center1(_center1), radius(_radius), mat(_material), is_moving(true)
    {
        auto rvec = vec3(radius, radius, radius);
        aabb box1(_center1 - rvec, _center1 + rvec);
        aabb box2(_center2 - rvec, _center2 + rvec);
        bbox = aabb(box1, box2);

        center_vec = _center2 - _center1;
    }

    bool hit(const ray& r, interval ray_t, hit_record& rec) const override {
```

sphere 类表示一个三维空间中的球体，可以是静止的或者在两个点之间移动的（表示动态球体）。该类包含球体的中心位置、半径、材质等属性。其构造函数可以创建静态球体或者动态球体。

头文件 quad.h 定义了一个名为 quad 的类，以及一个用于创建立方体（由六个 quad 对象组成）的辅助函数。

```
class quad : public hittable {
public:
    quad(const point3& _Q, const vec3& _u, const vec3& _v, shared_ptr<material> m)
        : Q(_Q), u(_u), v(_v), mat(m)
    {
        auto n = cross(u, v);
        normal = unit_vector(n);
        D = dot(normal, Q);
        w = n / dot(n,n);

        set_bounding_box();
    }

    virtual void set_bounding_box() {
        bbox = aabb(Q, Q + u + v).pad();
    }

    aabb bounding_box() const override { return bbox; }

    bool hit(const ray& r, interval ray_t, hit_record& rec) const override {
```

quad 类是光线追踪中实现四边形对象的重要工具，它允许在场景中添加具有明确边界的平面对象。通过使用 quad 类，可以构建出墙面、地板、天花板等平面元素，进而创建出丰富和逼真的三维场景。

此外，本项目还给出了 aabb.h, bvh.h 等工具头文件，其中轴对齐包围盒（AABB）是一个简单的几何形状，用于包围更

复杂的几何形状。在光线追踪中，AABB 通常用于快速排除与光线不相交的物体，从而优化性能。而有界体积层次（BVH）是一个树状结构，用于组织场景中的对象，以便高效地进行光线交点查询。BVH 通常由许多层次的 AABB 组成，每个节点包含对子节点的引用和节点所包含的 AABB。

aabb.h 和 bvh.h 提供的数据结构对于光线追踪中的性能优化至关重要。通过使用 AABB，可以快速确定光线是否有可能与一个复杂对象相交。而 BVH 结构进一步提高了这种查询的效率。

### 3、main函数的实现

main 函数组成了光线追踪程序的核心。程序在运行到 main 函数时，根据所给出的参数，对 main 中设计的场景进行光线追踪渲染。

首先第一部分渲染出地板的效果，地板由高低不平的立方体方块所组成，同时此处引入纹理贴图（“1.jpg”）是交大校徽图案。

```
hittable_list boxes1;
auto ground = make_shared<lambertian>(make_shared<image_texture>("1.jpg"));

int boxes_per_side = 20;
for (int i = 0; i < boxes_per_side; i++) {
    for (int j = 0; j < boxes_per_side; j++) {
        auto w = 100.0;
        auto x0 = -1000.0 + i*w;
        auto z0 = -1000.0 + j*w;
        auto y0 = 0.0;
        auto x1 = x0 + w;
        auto y1 = random_double(1,101);
        auto z1 = z0 + w;

        boxes1.add(box(point3(x0,y0,z0), point3(x1,y1,z1), ground));
    }
}
```



接下来代码中设置了三个光源，分别是一张纯色光源与两张贴图光源，此处引入交大元素，使用庙门和致远湖的照片作为纹理贴图，在渲染后的场景中起到一个“放映屏幕”的视觉效果。

```
auto light = make_shared<diffuse_light>(color(9, 7, 8));
world.add(make_shared<quad>(point3(123,554,147), vec3(300,0,0), vec3(0,0,265), light));

auto light2 = make_shared<diffuse_light>(make_shared<image_texture>("3.jpg"));
world.add(make_shared<quad>(point3(23,204,400), vec3(300,0,0), vec3(0,270,0), light2));

auto light3 = make_shared<diffuse_light>(make_shared<image_texture>("5.jpg"));
world.add(make_shared<quad>(point3(10, 0, 50), vec3(300, 0, 0), vec3(0, 250, 0), light3));
```

此外，main 中还创建了一系列球体等模型，用于展示光线追踪效果，包括动态球体、透明球、金属球、雾化球、柏林噪声球、旋转平移球集等等，光追逐一在相应位置展示出对应像素点应有的颜色。

接下来是相机参数的设置。

```
camera cam;

cam.aspect_ratio      = 1.0;
cam.image_width       = image_width;
cam.samples_per_pixel = samples_per_pixel;
cam.max_depth         = max_depth;
cam.background        = color(0,0,0);

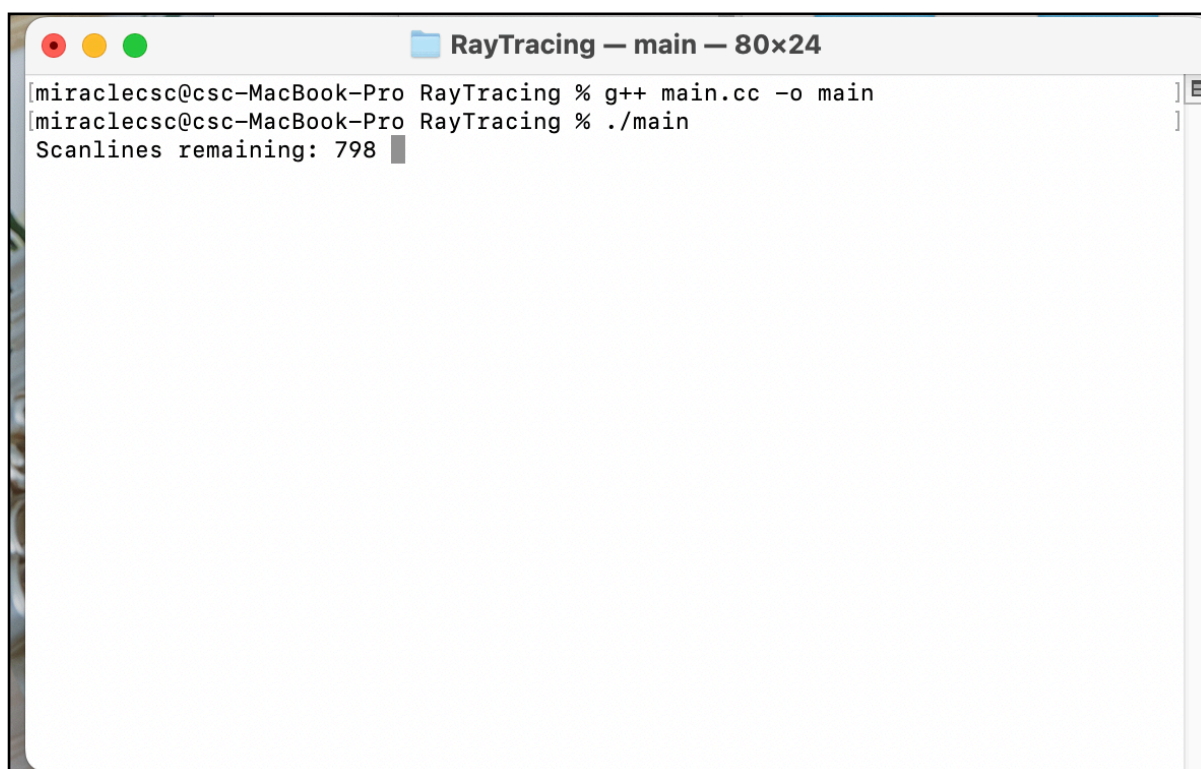
cam.vfov              = 40;
cam.lookfrom          = point3(478, 278, -600);
cam.lookat            = point3(278, 278, 0);
cam.vup               = vec3(0,1,0);

cam.defocus_angle     = 0;

cam.render(world);
```

调用 `render` 对场景进行渲染。最终展示的效果图如报告开头所示。

在终端中执行编译运行命令即可。文件输入输出流会将数据输出到 `image.ppm` 文件之中，在 MacOS 直接打开ppm文件即可展示输出效果。本项目具体渲染大约需要数个小时。



```
RayTracing — main — 80x24
[miraclecsc@csc-MacBook-Pro RayTracing % g++ main.cc -o main
[miraclecsc@csc-MacBook-Pro RayTracing % ./main
Scanlines remaining: 798
```

## 4、总结

本项目成功实现了一个高级光线追踪器，它能够生成逼真的三维图像，展现复杂场景和多样化的物理材质效果。通过精心设计的光线追踪算法和高效的数据结构，该追踪器不仅能模拟复杂光学现象，如反射、折射、散射和阴影，还能以高性能和高准确度处理大量的几何对象。

项目支持多种几何形状，包括球体、四边形和自定义形状。通过 hittable 类的派生，我们能够方便地加入新的形状，如 sphere 和 quad。camera.h 实现了一个灵活的相机模型，支持调整视野角度、焦距和深度场效果等参数，使得从不同角度和条件下渲染场景成为可能。

总的来说，此项目通过综合应用先进的计算机图形学原理和高效的编程实践，成功实现了一个功能丰富、性能优越的光线追踪器，为后续相关领域的研究和应用打下了坚实的基础。