中山大学计算机学院

计算机图形学

本科生实验报告

(2023学年秋季学期)

课程名称：Computer Graphic

组长	段尧
组员	李骏唯、张礼贤、唐成文

任务：完善光线追踪渲染器

功能实现

本次的期末项目主要是基于本学期图形学课程Homework 4 的整体框架继续编写代码。在完成了《Ray Tracing in One Weekend》以及《Ray Tracing_ The Next Week》以后，我们小组在网上找到了与前两个教程一起配套的第三个教程《Ray Tracing_ The Rest Of Your Life》。借助该教程，我们实现了用蒙特卡洛方法给图像降噪。此外，我们还实现了添加纹理、多线程执行以及加载网格模型等功能。下面我们将逐一结合基本原理介绍我们组实现的一些功能。

1. 蒙特卡洛光线追踪在康奈尔盒子中的噪声优化

1. 引言

光线追踪作为一种生成逼真图像的强大技术，在球面上的蒙特卡洛积分计算、光散射、重要采样材质等方面有着广泛的应用。本报告将详细讨论这些技术在康奈尔盒子场景中的应用，并提供相关公式和解释。

2. 球面上的蒙特卡洛积分计算

在球面上进行蒙特卡洛积分计算时，我们需要考虑球面上的积分点，可以使用球坐标系下的积分元素来表示：

$$I = \frac{1}{N} \sum_{i=1}^{N} f(\theta_i, \phi_i) \sin(\theta_i) \Delta\theta_i \Delta\phi_i \tag{15}$$

其中，(f(\theta_i, \phi_i)) 是在球面上某点的函数值，(\theta_i) 和 (\phi_i) 是球坐标系下的角度，(\Delta \theta_i) 和 (\Delta \phi_i) 是相邻积分点之间的角度差。

3. 光散射

光散射是光线在物体表面发生反射或散射的过程。在蒙特卡洛光线追踪中，光散射可以通过BRDF（双向散射分布函数）来描述：

$$L_o(\omega_o) = \int_{\Omega} f(\omega_i, \omega_o) L_i(\omega_i) \cos(\theta_i) d\omega_i \tag{16}$$

其中，(L_o(\omega_o)) 是出射辐射度，(f(\omega_i, \omega_o)) 是BRDF，(L_i(\omega_i)) 是入射辐射度，(\cos(\theta_i)) 是入射方向的余弦值。

4. 散射

在光线追踪中，散射是指光线在与物体表面发生相互作用后改变方向的现象。在散射过程中，我们可以使用蒙特卡洛积分来估算散射后的光强：

$$L_o(x, \omega_o) = \frac{1}{N} \sum_{i=1}^{N} f(\omega_i, \omega_o) L_i(x, \omega_i) \cos(\theta_i) \tag{17}$$

5. 散射PDF

为了进行重要采样，我们需要计算散射的概率密度函数（PDF）。对于均匀散射，PDF可以表示为：

$$p(\omega_i) = \frac{1}{2\pi} \tag{18}$$

其中，(\omega_i) 是入射方向。

6. 重要采样材质

重要采样通过调整样本的采样概率，提高对重要路径的采样效率。在蒙特卡洛光线追踪中，可以使用重要采样来优化材质的采样过程。

```
// quad.h
double pdf_value(const point3& origin, const vec3& v) const override {
        hit_record rec;
        if (!this->hit(ray(origin, v), interval(0.001, infinity), rec))
            return 0;

        auto distance_squared = rec.t * rec.t * v.length_squared();
        auto cosine = fabs(dot(v, rec.normal) / v.length());

        return distance_squared / (cosine * area);
    }
```

函数首先创建了一个 `hit_record` 对象 rec，然后使用 this->hit 函数来判断从 origin 沿着方向 v 是否会与物体相交。如果光线与物体相交，hit 函数将填充 rec 对象，记录相交的信息。

接着，函数计算了散射点到光源的距离的平方 distance_squared，以及散射点法线方向与入射方向的夹角的余弦值 cosine。最后，函数返回散射点到光源的距离平方与余弦值的比值再除以物体表面的面积 area。这个比值表示在给定方向上散射的概率密度。

同理，我们也可以在`hittables_list.h`中创建相应的函数，实现重要性采样，其具体代码与上面的类似，这里不做展示

6.1 随机半球抽样

为了在材质表面上进行随机半球抽样，我们可以使用球坐标系的随机角度来获得入射方向：

$$\begin{aligned} \theta_i &= \cos^{-1}(1 - 2\xi_1) \\ \phi_i &= 2\pi\xi_2 \end{aligned} \tag{19}$$

其中，(\xi_1) 和 (\xi_2) 是在[0, 1)范围内的随机数。

7. 生成随机方向

生成随机方向在光线追踪中经常用于采样入射光线的方向，可以通过球坐标系的随机角度生成：

$$\begin{aligned} \theta &= \cos^{-1}(1 - 2\xi_1) \\ \phi &= 2\pi\xi_2 \end{aligned} \tag{20}$$

7.1 相对于Z轴的随机方向

为了生成相对于Z轴的随机方向，我们可以使用球坐标系的随机角度，并将生成的方向转换为直角坐标系：

$$\begin{aligned} x &= \sin(\theta)\cos(\phi) \\ y &= \sin(\theta)\sin(\phi) \\ z &= \cos(\theta) \end{aligned} \tag{21}$$

7.2 对半球进行均匀采样

对半球进行均匀采样可以通过在球坐标系中生成随机方向来实现，然后将其转换为直角坐标系。

```
class cosine_pdf : public pdf {
  public:
    cosine_pdf(const vec3& w) { uvw.build_from_w(w); }

    double value(const vec3& direction) const override {
        auto cosine_theta = dot(unit_vector(direction), uvw.w());
        return fmax(0, cosine_theta/pi);
    }

    vec3 generate() const override {
        return uvw.local(random_cosine_direction());
    }

  private:
    onb uvw;
};
```

其中random_cosine_direction: 该函数生成在余弦加权分布的半球上的随机方向。通常用于重要性采样漫反射光照。

8. 标准正交基

标准正交基是计算机图形学中常用的数学工具，用于表示坐标系、相机方向和光线方向等。以下简要介绍相关概念：

8.1 相对坐标

相对坐标是描述对象相对于基准点或基准坐标系的坐标表示方式。在计算机图形学中，相对坐标的使用使得对象的描述更加灵活。

8.2 生成标准正交基

生成标准正交基通常包括选择基准向量，正规化第一个基向量，通过叉乘生成第二个和第三个基向量。这样构建的标准正交基在图形学计算中有广泛应用。

8.3 ONB类

ONB 类是一个管理标准正交基的实用工具。通过该类，我们能够方便地生成正交基，进行坐标变换，并在计算机图形学算法中灵活应用。其详细定义见`onb.h`类

9. 直接采样灯光

直接采样灯光是通过随机采样光源上的点来估算直接光照。为了获得光源的PDF，我们可以使用光源表面上的面积除以总面积。

9.1 获得光源的PDF

$$p(\omega_i) = \frac{A_{\text{light}}}{A_{\text{total}}} \tag{22}$$

其中，(A{\text{light}}) 是光源表面上的面积，(A{\text{total}}) 是场景中所有表面的总面积。

9.2 光线采样

通过在光源上随机选择一点，并计算光线方向，我们可以估算直接光照的贡献。

$$L_o(x, \omega_o) = \frac{1}{N} \sum_{i=1}^{N} \frac{f(\omega_i, \omega_o) L_i(x, \omega_i) \cos(\theta_i)}{p(\omega_i)} \tag{23}$$

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

    color emitted(const ray& r_in, const hit_record& rec, double u, double v,
const point3& p)
    const override {
        if (!rec.front_face)
            return color(0,0,0);
        return emit->value(u, v, p);
    }

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

10. 混合密度

混合密度技术通过使用混合概率密度函数，同时考虑平均照明和反射，以减少噪声。
代码实现如下：

```
class mixture_pdf : public pdf {
  public:
    mixture_pdf(shared_ptr<pdf> p0, shared_ptr<pdf> p1) {
        p[0] = p0;
        p[1] = p1;
    }

    double value(const vec3& direction) const override {
        return 0.5 * p[0]->value(direction) + 0.5 *p[1]->value(direction);
    }

    vec3 generate() const override {
        if (random_double() < 0.5)
            return p[0]->generate();
        else
            return p[1]->generate();
    }
```

```
  private:
    shared_ptr<pdf> p[2];
};
```

10.1 平均照明和反射

$$p(\omega_i) = w_{\text{light}} \cdot p_{\text{light}}(\omega_i) + w_{\text{reflect}} \cdot p_{\text{reflect}}(\omega_i) \tag{24}$$

其中，(w{\text{light}}) 和 (w{\text{reflect}}) 是混合权重，(p{\text{light}}(\omega_i)) 和 (p{\text{reflect}}(\omega_i)) 是光源和反射的PDF。

10.2 对Hittable的方向采样

在光线追踪中，对Hittable对象的方向进行采样时，可以使用PDF来调整采样方向：

$$p(\omega_i) = \frac{1}{2\pi} \tag{25}$$

当需要在场景中的物体表面进行随机采样时，通过这个概率密度函数来确定采样方向

```
class hittable_pdf : public pdf {
  public:
    hittable_pdf(const hittable& _objects, const point3& _origin)
      : objects(_objects), origin(_origin)
    {}

    double value(const vec3& direction) const override {
        return objects.pdf_value(origin, direction);
    }

    vec3 generate() const override {
        return objects.random(origin);
    }

  private:
    const hittable& objects;
    point3 origin;
};
```

11. 一些材质的pdf调用

在material类代码中，pdf 的调用主要涉及到两个虚函数：scatter 函数和 scattering_pdf 函数。这些函数用于模拟材质的散射和计算散射事件的概率密度函数。

1. **scatter 函数：**

   ```
   virtual bool scatter(const ray& r_in, const hit_record& rec, scatter_record&
   srec) const {
       return false;
   }
   ```

   在 material 基类中，定义了 scatter 函数，该函数负责计算散射。具体的散射行为由派生类实现。在 lambertian 类中，对 scatter 函数进行了重写：

```
bool scatter(const ray& r_in, const hit_record& rec, scatter_record& srec)
const override {
    srec.attenuation = albedo->value(rec.u, rec.v, rec.p);
    srec.pdf_ptr = make_shared<cosine_pdf>(rec.normal);
    srec.skip_pdf = false;
    return true;
}
```

在这里，scatter 函数为漫反射材质，计算了反射光线的颜色衰减和与表面法线相关的概率密度函数（cosine_pdf）。

在 metal 类中也有scatter 函数的重写：

```
bool scatter(const ray& r_in, const hit_record& rec, scatter_record& srec)
const override {
    srec.attenuation = albedo;
    srec.pdf_ptr = nullptr;
    srec.skip_pdf = true;
    vec3 reflected = reflect(unit_vector(r_in.direction()), rec.normal);
    srec.skip_pdf_ray = ray(rec.p, reflected + fuzz*random_in_unit_sphere(),
r_in.time());
    return true;
}
```

对于金属材质，scatter 函数计算了反射光线的颜色衰减和随机漫反射的光线。

在 dielectric 类中，同样有 scatter 函数的实现：

```
bool scatter(const ray& r_in, const hit_record& rec, scatter_record& srec)
const override {
    // ... (省略部分代码)
    srec.skip_pdf_ray = ray(rec.p, direction, r_in.time());
    return true;
}
```

为介电材质计算了衰减颜色和折射/反射光线。

在 diffuse_light 和 isotropic 类中，scatter 函数的实现分别处理发光材质和各向同性散射材质的散射行为。

2. **scattering_pdf 函数：**

```
virtual double scattering_pdf(const ray& r_in, const hit_record& rec, const
ray& scattered) const {
    return 0;
}
```

在material 基类中定义了 scattering_pdf 函数，用于计算散射事件的概率密度函数。在 lambertian 类中，对 scattering_pdf 函数进行了重写：

```
double scattering_pdf(const ray& r_in, const hit_record& rec, const ray&
scattered)
const override {
    auto cos_theta = dot(rec.normal, unit_vector(scattered.direction()));
    return cos_theta < 0 ? 0 : cos_theta/pi;
}
```

这里，scattering_pdf 函数返回了与表面法线相关的概率密度函数，用于重要性采样。

在 isotropic 类中，scattering_pdf 函数返回了各向同性散射的概率密度函数：

```
double scattering_pdf(const ray& r_in, const hit_record& rec, const ray&
scattered)
const override {
    return 1 / (4 * pi);
}
```

这样的设计可以使得材质的散射行为和概率密度函数的计算更加模块化，方便在不同的材质之间切换和扩展。

12. 清理PDF管理

在代码实现中，清晰地管理PDF是保证算法正确性和性能的关键。在光线追踪中，我们需要考虑漫反射光、反射光、镜面高光和球体对象的采样过程。

12.1 漫射光和反射光

对于漫反射光和反射光，需要计算其PDF以进行重要采样：

$$p(\omega_i) = \frac{1}{2\pi} \tag{26}$$

12.2 处理镜面高光

对于镜面高光，需要根据具体的BRDF来计算其PDF。

12.3 采样球体对象

在光线追踪中，对球体对象进行采样时，可以使用球坐标系下的随机角度。

12.4 更新球体代码

在代码实现中，需要相应地更新球体对象的代码，确保正确处理光线的相交和反射。

12.5 添加PDF功能到Hittable

在Hittable对象中，需要添加计算PDF的功能，以便在光线追踪中正确进行重要采样。

13. 结果与讨论

通过实施上述技术，我们在康奈尔盒子场景中成功地降低了噪声水平，提高了渲染图像的质量。在具体应用中，需要仔细调整参数和权重，以达到最佳的图像效果。

14. 结论

蒙特卡洛光线追踪技术在康奈尔盒子场景中的应用涉及到多个方面的数学计算和模型设计。通过详细的说明和公式解释，我们展示了如何在光线追踪算法中应用这些技术，为图像渲染提供了理论支持。技术的实际应用需要结合具体场景进行调优和优化，以满足不同需求。

2. 添加纹理

我们创建了一个纹理类（虚基类），其中核心是`value`函数，它的作用是输入纹理坐标(u, v)返回对应纹理值。

纹理类有两个继承：实体纹理和图像纹理。

实体纹理是只有一种颜色的纹理，多用于构造墙或其他纯色物体。其`value`函数只会返回一种颜色。

图像纹理用于加载外部图像作为纹理，其基本原理如下：

- 构造函数接受一个文件路径加载纹理图像，并用借助`rtw_stb_image.h`库初始化用于存储图片的`rtw_image`对象。
- 在`value`函数中：
    - 首先检查图像高度是否小于等于零，如果是，则返回纯青色（为调试提供帮助）。
    - 对输入的纹理坐标进行范围限制，确保它们在 [0,1] x [0,1] 的范围内。
    - 将经过范围限制后的纹理坐标 u 和 v 分别乘以图像纹理的宽度和高度，得到在图像纹理中的像素位置 (i, j)。
    - 使用`rtw_stb_image.h`库的`pixel_data`函数从图像中获取像素数据，该函数接受像素在图像中的位置 (i, j)，返回一个包含颜色信息的数据。
    - 将像素值范围从 [0, 255] 映射到 [0, 1]，并返回颜色向量。

那纹理坐标是怎么来的呢？对于不同形状的物体，将笛卡尔坐标映射到纹理坐标的方式不同。以球为例，我们知道笛卡尔坐标系和球坐标的关系为：

$$\begin{aligned} y &= -\cos(\theta) \\ x &= -\cos(\phi)\sin(\theta) \\ z &= \sin(\phi)\sin(\theta) \end{aligned} \tag{27}$$

这里假设球是单位球，θ 是从 -Y 开始到 Y 的角度，ϕ 是绕 Y 轴的角度（从 -X 到 +Z 到 +X 到 -Z 再到 -X）。于是有：

$$\begin{aligned} \phi &= \operatorname{atan2}(-z, x) + \pi \\ \theta &= \arccos(-y) \end{aligned} \tag{28}$$

将 θ 和 ϕ 都映射到 [0, 1]，就得到了纹理坐标 (u, v)：
$u = \frac{\phi}{2\pi}$
$v = \frac{\theta}{\pi}$
从直观上来看，上述步骤实际上是把均匀的经度和纬度映射到均匀的纹理图像的长宽值。

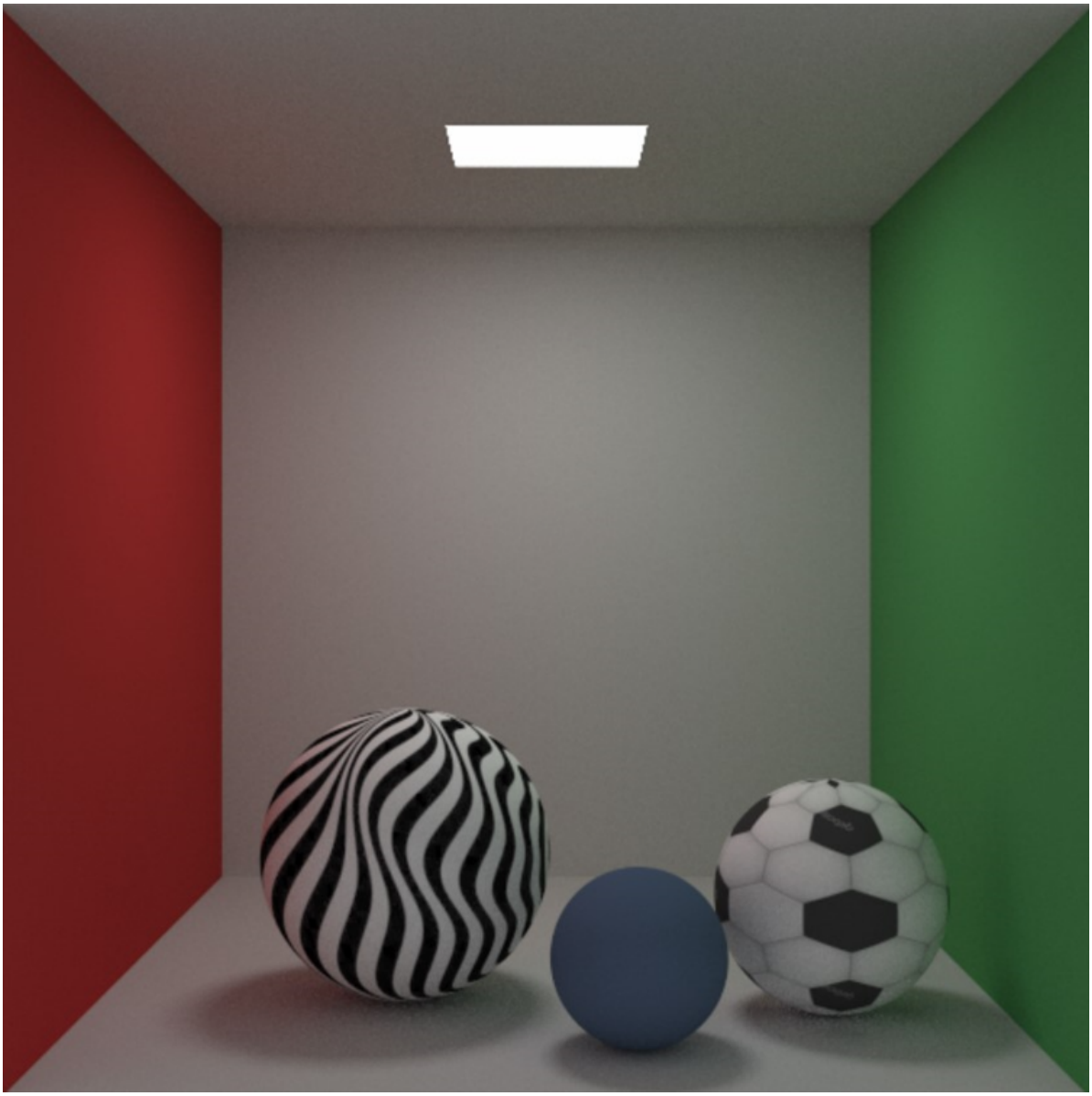
于是，我们在`hit_record`结构体中加上纹理坐标变量u、v，使得计算碰撞点时还得到纹理坐标。
同时在sphere类中加上计算纹理坐标的`get_sphere_uv`函数，并在计算碰撞的`hit`函数例调用它。

```
static void get_sphere_uv(const point3& p, double& u, double& v) {
    auto theta = acos(-p.y());
    auto phi = atan2(-p.z(), p.x()) + pi;

    u = phi / (2*pi);
    v = theta / pi;
}
```

而对于平面，要求其纹理坐标，首先把笛卡尔坐标转换为该平面上坐标alpha、beta，并判定是否在边界内（这里的平面是有界的），如果在，那就把alpha、beta赋值给纹理坐标u、v。

现在构建一幅场景验证我们的成果。首先在四周建立固体纹理的平面作为密闭空间墙，然后放置两个图像纹理的小球和一个固体纹理的小球。



如图所示，纹理成功作用到物体上，完成！

2. 多线程执行

在本项目中实现了使用 `openmp` 与 `thread` 两种多线程加速方式。

OpenMP 实现

`OpenMP`是一种并行编程接口，用于编写多线程并行程序，通常用于共享内存体系结构上的并行计算。它基于指令集，允许开发者在C，C++和Fortran中以简单的方式编写并行程序。

原项目中，在 `camera::render()` 中实现最终场景的渲染，在该函数中存在如下四层循环：

```
void render(const hittable& world, const hittable& lights) {
    //初始化代码
    #pragma omp parallel for
    for (int j = image_height-1 ; j >= 0 ; --j) {
        for (int i = 0; i < image_width; ++i) {
            color pixel_color(0,0,0);
            for (int s_j = 0; s_j < sqrt_spp; ++s_j) {
                for (int s_i = 0; s_i < sqrt_spp; ++s_i) {
                    //计算像素值
                }
            }
```

```
            //写入像素值
        }
    }
}
```

其中，前两层循环是独立的，内部循环需要叠加到同一变量 `pixel_color` 上不独立。所以本项目实现中考虑对于外层两层循环使用 `OpenMP` 进行加速处理。实际测试中，由于渲染场景像素高度远大于计算机支持的线程数目，故仅加速最外层循环 即可，这样也可以减少线程之间的耦合，如果使用两层循环的加速，效果反而不如一层循环加速。

在本项目代码实现中，只需要在 `camera::render()` 中循环体之前加上 `#pragma omp parallel for` 并在该类开头引入 `#include <omp.h>` 即可。项目中并没有使用编译指令规定线程数目，这样项目渲染时可以根据不同硬件自适应选择线程数。

Thread 实现

上述 `OpenMP` 的多线程加速是基于编译级别的多线程加速，我们也实现了使用 `thread` 的线程级别加速。值得注意的是，项目中的实现为了兼容原有的项目代码以及增强项目在不同平台的适用性，并没有使用 `pthread` 这一POSIX线程库实现，而是使用了 `thread` 这一微软基于 `pthread` 二次开发的线程库，该库可以在不同平台上使用，并且本项目的原始代码中，使用 `thread` 来创建线程用于开启渲染，我们使用 `thread` 可以更好地兼容原有代码。

使用 `thread` 实现的多线程加速代码在 `camera::muti_thread_render()` 函数中，该函数与原本的 `render` 函数接口参数一致，在函数体中，首先通过预定义数目创建线程，将渲染场景按照高度分割为不相关联的部分，通过 `threads[t] = std::thread([&]() {}` 调用接口实现每个线程的渲染，在 `{}` 内的是渲染的逻辑代码。具体如下：

```
        //分割任务
        const int chunksize = image_height / tNumber;
        for (int t = 0; t < tNumber; t++)
        {
            int start = t * chunksize;
            int end = (t == tNumber - 1) ? image_height : (t + 1) * chunksize;
            //执行线程函数
            threads[t] = std::thread([&]() {
                std::cout << "thread id: " << std::this_thread::get_id() <<
std::endl;
                for (int j = end-1; j >=0; --j) {
                    for (int i = 0; i < image_width; i++) {
                        //渲染像素的代码
                        color pixel_color(0, 0, 0);
                        for (int s_j = 0; s_j < sqrt_spp; ++s_j) {
                            for (int s_i = 0; s_i < sqrt_spp; ++s_i) {
                                ray r = get_ray(i, j, s_i, s_j);
                                pixel_color += ray_color(r, max_depth, world,
lights);
                            }
                        }
                        write_color(i, j, pixel_color, samples_per_pixel);
                    }
                }
            });
```

该函数在 `windows visual studio` 中调用会创建16个线程同时用于不同区域的渲染，但（经过咨询老师）由于 `windows` 下对于该库的限制，实际被操作系统调度中无法同时调度全部线程，一般只有2-5个线程在前台运行，移植到 `linux` 下可以取得更好的效果。

3. obj 网格模型加载

本项目已经实现obj网格模型文件的读取与加载。主要思路是创建`triangle.h`实现三角面元的渲染，创建`readobj.h`实现OBJ文件解析，提取顶点与法线信息。使得用户可以将OBJ模型加载到现有场景中。

除此之外，实际上代码实现中还保留了对于OBJ文件其他信息的读取接口，方便后续基于本代码实现二次开发，进一步支持MTL纹理模型文件等。

triangle 类

该文件实现了对三角面元的渲染。原项目代码中，采用 `quad.h` 中实现的四边形面进行渲染，只能支持平行四边形（包含长方形）的渲染，而现有业界多用三角面元实现渲染，OBJ模型文件也以三角面元记录模型信息。为实现OBJ模型的加载，需要先实现三角面元的渲染功能，为了提高代码可扩展性，以及为后续更多模型文件加载提供接口，在实现中将三角面元渲染单独拆分作为单独类实现。

成员变量

```
private:
    point3 v0, v1, v2;//顶点
    vec3 normal;//法向
    shared_ptr<material> mat_ptr;//指向材质对象的指针
    const double epsilon = 0.0000001; // 极小值，用于控制数值计算精度
```

`triangle`类中的成员变量如上所示，分别表示渲染三角面元所需的基本信息。`mat_ptr`是为兼容原有代码框架使用的智能指针，将在渲染时由调用者决定传入的渲染材质。`epsilon`是控制数值计算精度的成员变量，众所周知在点与三角形位置计算中存在众多舍入，使用该参数可以控制误差范围，这个参数主要在`triangle::hit`函数中使用。

接口说明

在开始介绍其他各类成员函数之前，应该首先说明最需要掌握的两个函数。

```
static hittable_list load_obj_model(const std::string& filename,
std::shared_ptr<material> mat)
static hittable_list load_obj_model(const std::string & filename,
std::shared_ptr<material> mat, hittable_list & world)
```

上述两个函数实现了从一个obj文件中加载模型到现有的渲染场景中，我们推荐使用第二个函数，第二个函数使用了引用传参，传入了当前的场景指针，并在函数中将obj文件载入组织成三角面元后添加到场景中。第一个函数由于作者对于智能指针的实现并不熟练，使用该函数可能需要更改该函数的返回值类型为指向 `hittable` 的智能指针。

其次是该类的构造函数，我们提供了带参数与不带参数的构造函数，其中后者需要使用顶点与材质参数指针进行初始化：

```
triangle()
triangle(const point3& v0, const point3& v1, const point3& v2,
shared_ptr<material> m)
```

为了与 `quad` 类实现用法的兼容，定义了虚函数 `hit` ，这个函数将会在渲染时用于计算相交。在 `triangle::hit` 的实现中，采用 `Möller–Trumbore` 算法实现，如果对该算法并不熟悉，可以参考该博客 [数学] Möller–Trumbore 算法 - 知乎 (zhihu.com)。我们的实现接口如下。实际上在渲染时你并不需要实际了解该函数：

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

除此之外，剩下的是一些功能性函数，为了加速渲染或者采样：

```
virtual aabb bounding_box() const override;
virtual double pdf_value(const point3& o, const vec3& v) const override;
virtual vec3 random(const point3& o) const override;
```

第一个函数返回原来项目中的 `aabb` 类，目的是构建一个包含该可渲染对象的最小盒子，用于加速射线与三角形的相交检测。

第二个函数计算指定点与向下方向的概率密度函数值，它首先检测光线与三角形相交情况，如果相交则计算投影面积，以此为基础计算概率密度函数，该函数将在实现蒙特卡洛加速中使用。

第三个函数用于在三角形上生成一个随机样本点。

readobj 类

该类用于读取 OBJ 文件并将所有信息存入调用者指定的 `vector<point3>` 中，其中 `point3` 是原代码框架在 `vec3.h` 中定义的别名。

该类中只有一个函数，接口如下：

```
static bool loadOBJ(const std::string & filename, std::vector<point3>&vertices,
std::vector<vec3>&normals, std::vector<vec3>&texCoords,
std::vector<std::vector<int>>&faces)
```

通过读取 `filename` 指向的obj文件路径，将其中的顶点信息，法向信息，纹理坐标信息（纹理坐标信息需要结合mtl文件使用），面元结构信息读取出来。需要注意，此处读取信息存储的 `vector` 是由外部引用传参的，也就是需要外部调用者创建。

调用说明

如需在 `main.cpp` 中创建并渲染一个三角形，可以参照：

```
    //triangle
    point3 v0(0, 0, 100);
    point3 v1(0, 100, 0);
    point3 v2(100,0,0);
    // 创建一个 light 材质
    auto light = make_shared<diffuse_light>(color(15, 15, 15));
    // 创建三角形对象，传入三个顶点和材质
    auto light_triangle = make_shared<triangle>(v0, v1, v2, light);
    // 将三角形添加到场景中
    world.add(light_triangle);
```

首先定义三个顶点，然后定义材质属性，之后创建三角形对象（调用 `triangle.h` 的构造函数），最后添加到渲染场景 `world` 中。

如需从OBJ文件中加载模型，直接使用：

```
world = triangle::load_obj_model("E:/obj/lifangti1003.obj", light,world);
```

即可将OBJ文件中的模型加载到场景中。

最终实现结果展示：