说明文档

项目介绍

使用C++，不调用任何库或者API，实现光线追踪算法，并以此完成对经典图形学场景（如Cornell Box）和自己构建的复杂场景的渲染。在实现过程中，需要考虑光线的反射、折射、散射现象。

项目采用面向对象的方法进行构建，类课分为7种。第一种，main.cpp负责整体调度、构建场景、扫描像素。第二种，MyCamera类分装了照相机的功能。第三种为所有类的基础，包括：vec3类实现了向量和点，重载了向量相关的运算；ray类实现了光线；color实现了颜色，以及像素点的输出。第四种为抽象类，包括：Hittable抽象了一切能被光线碰撞的物体；Hittable_list抽象了Hittable类的集合，用于储存场景。第五种为实体类，实现了一部分用于构建cornell box的简单几何形体，包括Sphere即球体；Box即长方体；aarect即三维平面。第六类为才之类，实现了郎波体、金属、电介质、光源等材质。第七类为myRayTracing.h，用于存储常数和提供最基本的运算服务。

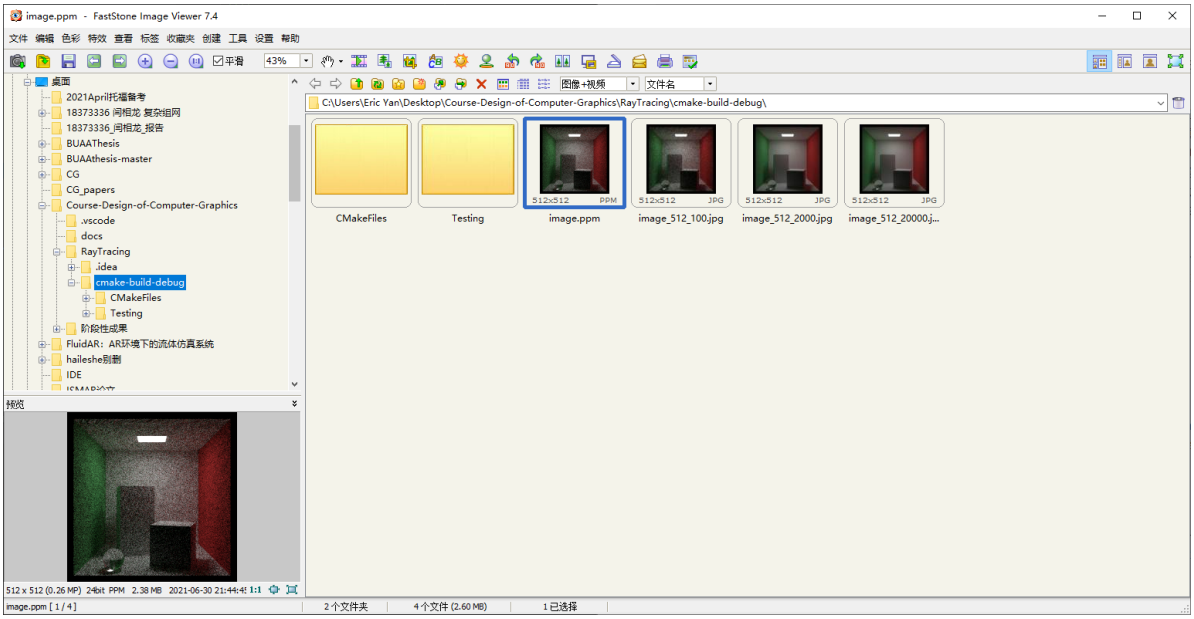
由于ppm（P3）是最简单的图片格式，因此本项目选取ppm格式的图片作为输出。除此之外，ppm格式的图片是一系列十进制(r, g, b)三元组组成的，观察起来直观、方便，便于debug，这也是选择ppm格式作为输出的原因之一。

查看方式

为了查看ppm格式的图片，请下载“FastStone Image Viewer”。软件下载地址如下：

https://www.faststone.org/FSIVDownload.htm

下载FastStone Image Viewer 7.5的exe，安装后，用FastStone Image Viewer打开Course-Design-of-Computer-Graphics\RayTracing\cmake-build-debug目录即可查看ppm文件。由于ppm格式没有经过任何压缩，图片体积较大，因此使用软件现实可能略有延迟。使用截图如下：



运行方式

如果使用window环境，则打开cmd，依此输入前两条命令，进入到“Course-Design-of-Computer-Graphics\RayTracing\cmake-build-debug”这一目录。随后使用cmd运行RayTracing.exe，并重定向进image.ppm文件中。

cd RayTracing

cd cmake-build-debug

RayTracing.exe > image.ppm

部分所使用的技术

1多核并行

项目开发的前期，场景较为简单（阶段性成果1、2、3、4、5），运算量小，使用单核运行尚可在1s内完成渲染。当项目开发进行到后期，尤其是引入三维平面后，计算光线碰撞的运算量陡增，单核已经无法胜任渲染任务，因此选择使用omp标准库进行多核并行，使渲染速度提升了约16倍。相关代码如下：

```
    //// Render
    const int nworker = omp_get_num_procs();
    #pragma omp parallel for num_threads(nworker) schedule(dynamic, 1)
    for (int j = IMAGE_HEIGHT - 1; j >= 0; --j) {
        for (int i = 0; i < IMAGE_WIDTH; ++i) {
            color pixel_color( x: 0,  y: 0,  z: 0);
            for (int sample = 1; sample <= SAMPLES_PER_PIXEL; sample++) {
                double u = (i + random_double())/(IMAGE_WIDTH - 1);
                double v = (j + random_double())/(IMAGE_HEIGHT - 1);
                ray r = myCamera.get_ray(u, v);
                pixel_color += ray_color(r, background, world, DEPTH);
            }
            //write_color_stream_gamma2(std::cout, pixel_color / SAMPLES_PER_PIXEL);
            color_buffer[i][j] = pixel_color / SAMPLES_PER_PIXEL;
        }
    }
    std::cerr << "\nRender finished\n";

    //// Output
    std::cout << "P3\n" << IMAGE_WIDTH << ' ' << IMAGE_HEIGHT << "\n255\n";
    for (int j = IMAGE_HEIGHT - 1; j >= 0; --j) {
        std::cerr << "\r[Output]Scanlines remaining: " << j << ' ' << std::flush;
        for (int i = 0; i < IMAGE_WIDTH; ++i) {
            write_color_stream_gamma2( &: std::cout, color_buffer[i][j]);
        }
    }
    std::cerr << "\nOutput finished\n";
}
```

尽管如此，在渲染复杂的cornell box时，512 * 512分别率，20000次采样依然花费了约60分钟（CPU为Intel i7-10700）。（图片在”阶段性成果/8_cornell_box_extra“文件夹中，可以看到不同采样数的图像效果。

2反走样

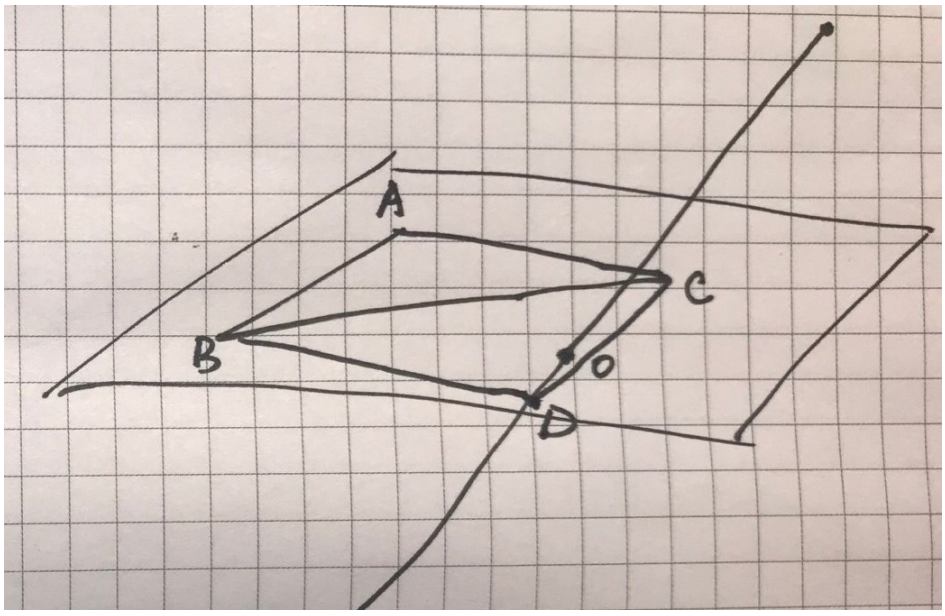
考虑到只使用了CPU进行渲染，因此画面的分辨率不宜过高，否则可能就要花费几个小时对画面进行渲染。但是，较低的分辨率带来了一个问题：画面走样。为了避免画面感走样，我对每个像素进行多次采样，每次采样时在像素内部加入一个微小的扰动（random_double()），最后求平均值得出像素最终的颜色，成功解决了这一问题。相关代码如下：

```
//// Render
const int nworker = omp_get_num_procs();
#pragma omp parallel for num_threads(nworker) schedule(dynamic, 1)
for (int j = IMAGE_HEIGHT - 1; j >= 0; --j) {
    for (int i = 0; i < IMAGE_WIDTH; ++i) {
        color pixel_color( x: 0,  y: 0,  z: 0);
        for (int sample = 1; sample <= SAMPLES_PER_PIXEL; sample++) {
            double u = (i + random_double())/(IMAGE_WIDTH - 1);
            double v = (j + random_double())/(IMAGE_HEIGHT - 1);
            ray r = myCamera.get_ray(u, v);
            pixel_color += ray_color(r, background, world, DEPTH);
        }
        //write_color_stream_gamma2(std::cout, pixel_color / SAMPLES_PER_PIXEL);
        color_buffer[i][j] = pixel_color / SAMPLES_PER_PIXEL;
    }
}
```

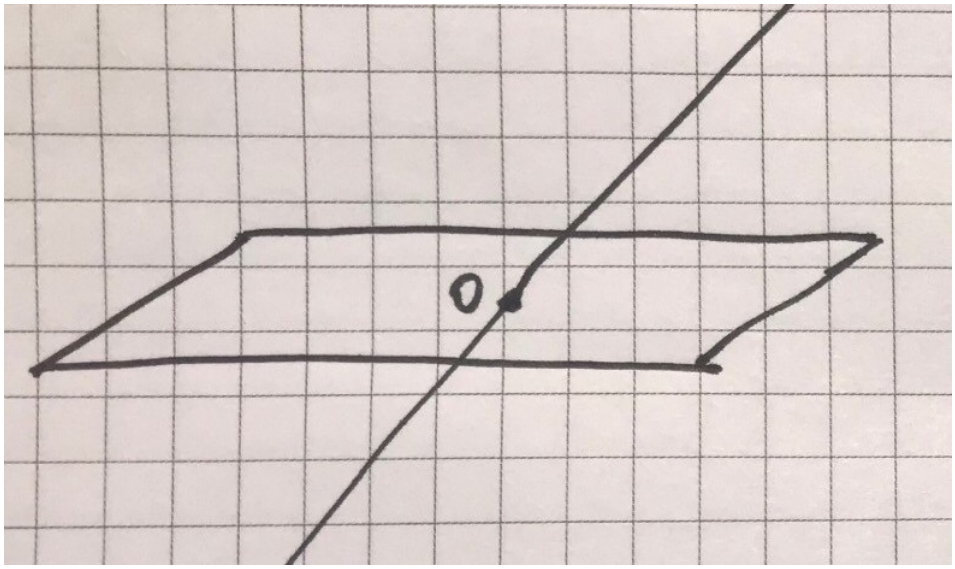
3三角形建模的改进

利用三角形能够统一、简单地完成复杂几何场景的建模，因此三角形建模时目前主流的建模方式。但是在本项目中，由于cornell box由简单几何体构成，且利用简单几何体建模能够在计算碰撞时实现更高的效率，因此采用平面和长方体构建cornell box。三位平面建模效率高于三角形建模的原因如下：

考虑需要对一个垂直于某一坐标轴的长方形进行建模。如果使用三角形建模就需要至少2个三角形对这一长方形进行表示，判断一束光线是否与这一长方形相交时需要判断这一光线是否与两个三角形相交（如下图所示）。判断方法详细过程如下：首先通过1次浮点数出发求出光线与长方形所在平面的交点。随后，需要依此判断这个交点是否在三角形内。判断一个点是否在三角形内需要进行3次向量叉乘，也就是18次浮点数乘法，9次加法。这一过程共计1次浮点数除法、36次浮点数乘法、18次加法，计算量较大。



如果使用垂直于y轴的三维平面进行建模，则首先需要求出光线与平面的交点，随后判断这个点是否在长方形内（如下图所示）。计算光线与平面的交点时需要1次浮点数除法，而判断这个点是否在长方形内只需要进行4次比较运算（姑且算作是减法运算）。因此，这一过程共计需要1次浮点数除法、4次减法，效率远远高于三角形建模法。



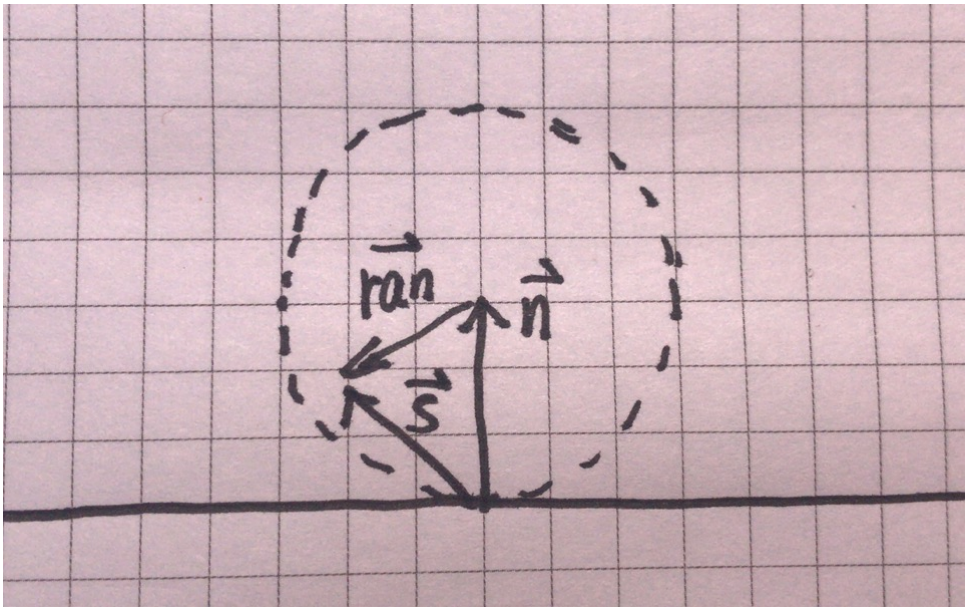
该算法的代码实现见下图（以垂直于y轴，四边平行于x轴和z轴的正方形为例）：

```cpp
//// algorithm reference: GAMES101-现代计算机图形学入门-闫令琪 Lecture 13 Ray Tracing 1 1:13:49
bool xz_rect::hit(const ray& r, double t_min, double t_max, hit_record& rec) const {
    auto t = (k - r.origin().y()) / r.direction().y();
    if (t < t_min || t > t_max)
        return false;

    auto x = r.origin().x() + t * r.direction().x();
    auto z = r.origin().z() + t * r.direction().z();
    if (x < x0 || x > x1 || z < z0 || z > z1)
        return false;

    rec.u = (x - x0) / (x1 - x0);
    rec.v = (z - z0) / (z1 - z0);
    rec.distance = t;
    auto outward_normal = vec3( x: 0,  y: 1,  z: 0);
    rec.set_face_normal(r, outward_normal);
    rec.mat_ptr = mat_ptr;
    rec.hit_point = r.at(t);

    return true;
}
```

4利用随机数实现郎波体材质

朗伯体是指当入射能量在所有方向均匀反射，即入射能量以入射点为中心，在整个半球空间内向四周各向同性的反射能量的现象，称为漫反射，也称各向同性反射，一个完全的漫射体称为朗伯体。为了实现这样的性质，我们不得不结合随机以及采样的思想进行实现。

阅读定义后，我们可以采样下图所示的思想对郎波体进行实现。由于每次发生反射，郎波体朝各个方向辐射能量的概率都是一样的，因此我们可以通过$\vec{s} = \overrightarrow{normal} + \overrightarrow{random}$的方式，随机生成一个出射方向。其中$\vec{s}$为出射方向，$\vec{n}$即为$\overrightarrow{normal}$为碰撞点的法向量，$\overrightarrow{ran}$即为随机产生的单位球内的向量，用于给出射防线添加扰动。



这一方法的代码实现如下图：

```
class lambertian : public Material {
    public:
        lambertian(const color& a) { albedo = a; }

        //// diffuse reflection
        virtual bool scatter(
                const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered
        ) const override {
            vec3 scatter_direction = rec.normal + random_unit_vector(); //diffuse

            if (scatter_direction.near_zero())
                scatter_direction = rec.normal;

            scattered = ray(rec.hit_point, scatter_direction);
            attenuation = albedo;
            return true;
        }

    public:
        color albedo;
};
```

代码中random_unit_vector()即为生成一个单位求内的向量的函数。

5光源的实现

项目中，我将光源看作是一种材质，继承自Material类。考虑到一种材质即可能发光，有可能对打到材质上的光线起散射作用，因此在抽象类Material类中声明两个虚函数emit、scatter，分别用于表示材质的发光和对光线的散射作用。Material类的声明如下：

```
class Material {
    public:
        ////@preset: the ray r_in hits the material's owner(and object)
        ////@input: the ray r_in
        ////@function: this function return whether the ray will scatter after hitting the object that possesses this material
        ////          is yes, return true, and pass hit_point information through rec, color through attenuation, the ray scatters out throught scattered
        ////@output: true if scatters, false is not
        virtual bool scatter(
                const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered
        ) const = 0;

        ////@preset: a ray hits the material's owner(and object)
        ////@output: return the color of light emitted by the material
        virtual color emitted(const point3& p) const {
            return color( x: 0, y: 0, z: 0);
        }
};
```

随后我实现了light类，用于模拟光源，并忽略光源对外来光照的散射，只考虑光源发出的自身颜色的光，其代码如下：

```cpp
class light : public Material {
    public:
        light(color light_color) {
            this->light_color = light_color;
        }

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

        //// light source
        virtual color emitted(const point3& p) const override {
            return this->light_color;
        }

    public:
        color light_color;
};
```

随后，在渲染时使用着色函数对像素的颜色进行计算，就可以实现图像的渲染了。着色函数代码如下：

```cpp
color ray_color(const ray& r, const color& background, const Hittable& world, int depth) {
    if (depth <= 0) {
        return color( x: 0,  y: 0,  z: 0);
    }

    hit_record rec;

    if (world.hit(r,  min_distance: 0.001, infinity,  &: rec)) { //hit an object
        ray scattered_out;
        color attenuation;
        color emitted;

        emitted = rec.mat_ptr->emitted(rec.hit_point);

        if (rec.mat_ptr->scatter(r, rec,  &: attenuation,  &: scattered_out)) {
            return emitted + attenuation * ray_color(scattered_out, background, world,  depth: depth - 1);
        } else {
            return emitted;
        }
    } else { //hitting nothing. return the background light
        return background;
    }
}
```

注：depth是用于控制光线散射次数的量，每散射一次，depth减一，其作用与计网数据报中的ttl字段的功能有异曲同工之处。

着色函数的思路为首先判断这一光线是否与模型实体碰撞（第71行），若发生碰撞则计算其颜色，否则返回背景色background。第76行计算碰撞点材质的自发光。78行判断该光线在碰到材质后是否会发生散射，如果发生散射则返回散射光颜色与材质自发光颜色的混合色，否则进返回材质自发光的颜色，由此实现场景的着色。

最终成果

图片像素512 * 512，采样数20000，使用Intel i7-10700渲染60分钟后可得下图。老师可以按照文章开始时提供的运行方法运行RayTracing.exe，并将输出重定向到image.ppm中，并使用FastStone Image Viewer查看渲染出的图像。为了节省老师的时间，我编译时采样数改为了100（myRayTracing.h中的const int SAMPLES_PER_PIXEL = 100;），图像质量可能不如下图，但渲染过程只需要3s。