# 作业7

## 基础部分

castRay()函数，按照实验指导书的提示完成即可

```
Vector3f Scene::castRay(const Ray &ray, int depth) const
{
    // TO DO Implement Path Tracing Algorithm here

    Intersection inter = intersect(ray);

  if (inter.happened)
  {
    // 如果射线第一次打到光源，直接返回
    if (inter.m->hasEmission())
    {
      if (depth == 0)
      {
        return inter.m->getEmission();
      }
      else return Vector3f(0, 0, 0);
    }

    Vector3f L_dir(0, 0, 0);
    Vector3f L_indir(0, 0, 0);

    // 随机 sample 灯光，用该 sample 的结果判断射线是否击中光源
    Intersection lightInter;
    float pdf = 0.0f;
    sampleLight(lightInter, pdf);
    // 物体表面法线
    Vector3f& N = inter.normal;
    // 灯光表面法线
    Vector3f& NN = lightInter.normal;
    Vector3f& objPos = inter.coords;
    Vector3f& lightPos = lightInter.coords;
    Vector3f diff = lightPos - objPos;
    float lightDistance = dotProduct(diff,diff);
    Vector3f lightDir = diff.normalized();
    Ray light(objPos, lightDir);
    Intersection light2obj = intersect(light);

    // 直接光源
    if (light2obj.happened && (light2obj.coords - lightPos).norm() < 1e-2)
    {
      Vector3f f_r = inter.m->eval(ray.direction, lightDir, N);
      L_dir = lightInter.emit * f_r * dotProduct(lightDir, N) * dotProduct(-lightDir, NN) / lightDistance / pdf;
    }

    if (get_random_float() < RussianRoulette)
    {
      Vector3f nextDir = inter.m->sample(ray.direction, N).normalized();
      Ray nextRay(objPos, nextDir);
      Intersection nextInter = intersect(nextRay);
      //间接光源
      if (nextInter.happened && !nextInter.m->hasEmission())
      {
        float pdf = inter.m->pdf(ray.direction, nextDir, N);
        Vector3f f_r = inter.m->eval(ray.direction, nextDir, N);
        L_indir = castRay(nextRay, depth + 1) * f_r * dotProduct(nextDir, N) / pdf / RussianRoulette;
      }
    }

    return L_dir + L_indir;
  }
```

```
  return Vector3f(0, 0, 0);
}
```

- 需要注意的是，向量的归一化，以及函数调用的参数问题，指导书已经说明参数与课堂上老师讲解的不一样

# 提高部分

- 实现多线程

```
void Renderer::Render(const Scene& scene)
{
    std::vector<Vector3f> framebuffer(scene.width * scene.height);

    float scale = tan(deg2rad(scene.fov * 0.5));
    float imageAspectRatio = scene.width / (float)scene.height;
    Vector3f eye_pos(278, 273, -800);
    int m = 0;

    // change the spp value to change sample ammount
    int spp = 256;
    std::cout << "SPP: " << spp << "\n";
    // for (uint32_t j = 0; j < scene.height; ++j) {
    //     for (uint32_t i = 0; i < scene.width; ++i) {
    //         // generate primary ray direction
    //         float x = (2 * (i + 0.5) / (float)scene.width - 1) *
    //                   imageAspectRatio * scale;
    //         float y = (1 - 2 * (j + 0.5) / (float)scene.height) * scale;

    //         Vector3f dir = normalize(Vector3f(-x, y, 1));
    //         for (int k = 0; k < spp; k++){
    //             framebuffer[m] += scene.castRay(Ray(eye_pos, dir), 0) / spp;
    //         }
    //         m++;
    //     }
    //     UpdateProgress(j / (float)scene.height);
    // }

    int process = 0;
    //匿名函数
    auto castRayMultiThread = [&](uint32_t rowStart,uint32_t rowEnd,
                                  uint32_t colStart, uint32_t colEnd)
    {
        for(uint32_t j = colStart;j <= colEnd;j++)
        {
            int m = j * scene.width + rowStart;
            for(uint32_t i = rowStart;i<=rowEnd;i++)
            {
                float x = (2 * (i + 0.5) / (float)scene.width - 1) *
                          imageAspectRatio * scale;
                float y = (1 - 2 * (j + 0.5) / (float)scene.height) * scale;

                Vector3f dir = normalize(Vector3f(-x, y, 1));
                for (int k = 0; k < spp; k++)
                {
                    framebuffer[m] += scene.castRay(Ray(eye_pos, dir), 0) / spp;
                }
                m++;
                process++;
            }
            std::lock_guard<std::mutex> lock1(mutex_prog);  //互斥锁，用于打印处理进度
            UpdateProgress(1.f*process/scene.width/scene.height);
        }
    };

    int id = 0;
    const int dx = 8;
```

```
    const int dy = 8;
    std::thread my_thread[dx*dy];
    int x_block = (scene.width+dx-1) / dx;
    int y_block = (scene.height+dy-1) / dy;
    //分块进行计算路径追踪
    for(int i=0;i<scene.width;i+=x_block)
    {
        for(int j=0;j<scene.height;j+=y_block)
        {
            my_thread[id] = std::thread(castRayMultiThread,
                            i,std::min(i+x_block,scene.width)-1,
                            j,std::min(j+y_block,scene.height)-1);
            id++;
        }
    }
    for(int i =0;i<dx*dy;i++)
        my_thread[i].join();

    UpdateProgress(1.f);

    // save framebuffer to file
    FILE* fp = fopen("256spp.ppm", "wb");
    (void)fprintf(fp, "P6\n%d %d\n255\n", scene.width, scene.height);
    for (auto i = 0; i < scene.height * scene.width; ++i) {
        static unsigned char color[3];
        color[0] = (unsigned char)(255 * std::pow(clamp(0, 1, framebuffer[i].x), 0.6f));
        color[1] = (unsigned char)(255 * std::pow(clamp(0, 1, framebuffer[i].y), 0.6f));
        color[2] = (unsigned char)(255 * std::pow(clamp(0, 1, framebuffer[i].z), 0.6f));
        fwrite(color, 1, 3, fp);
    }
    fclose(fp);
}
```

- 加上头文件以及对应的声明

```
#include <thread>
#include <mutex>

std::mutex mutex_prog;
```

- 在linux系统下，直接采用多线程编译会出现“undefined reference to pthread_create”，解决办法是修改CMakeList文件的配置，如下所示

```
cmake_minimum_required(VERSION 3.10)
project(RayTracing)

set(CMAKE_CXX_STANDARD 17)
find_package(Threads) //引入外部依赖包

add_executable(RayTracing main.cpp Object.hpp Vector.cpp Vector.hpp Sphere.hpp global.hpp Triangle.hpp Scene.cpp
        Scene.hpp Light.hpp AreaLight.hpp BVH.cpp BVH.hpp Bounds3.hpp Ray.hpp Material.hpp Intersection.hpp
        Renderer.cpp Renderer.hpp)

target_link_libraries (${PROJECT_NAME} ${CMAKE_THREAD_LIBS_INIT}) //链接 Thread 库
```

- 至于实现微表面模型，需要修改的框架比较多，笔者暂时先不处理