蒙特卡洛光线追踪

DrinkingCoder

介绍

本报告用于介绍计算机图形学第二次大作业蒙特卡洛光线追踪算法的实现，主要分为以下几个章节：第二节介绍了蒙特卡洛光线追踪算法的理论基础，第三节介绍了我实现作业使用的整体架构、用于加速的一些算法以及实现的重点与难点，第四节对于比较关键的代码实现进行讲解，第五节展示了实现的最终成果以及使用代码的方式，第六节描述的是一些在实现蒙特卡洛光线追踪算法的总结，以及后续想要对蒙特卡洛算法增加的一些改进。

理论基础

蒙特卡洛积分法

蒙特卡洛（Monte Carlo）法是一类随机算法的统称。随着二十世纪电子计算机的出现，蒙特卡洛法已经在诸多领域展现出了超强的能力，而蒙特卡洛积分法则是建立在随机采样逼近真实结果的思想上计算函数积分的方法。

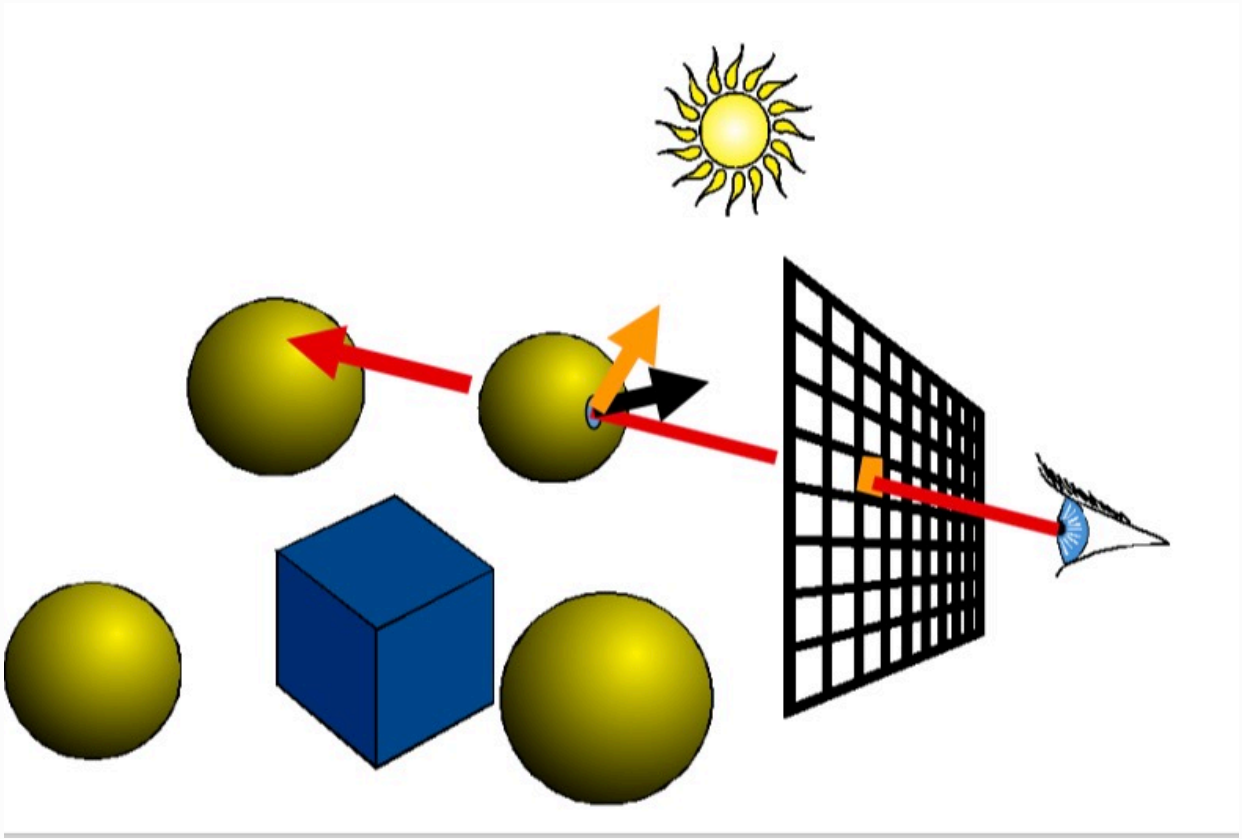
依据无意识统计学家法则（Law of the unconscious statistician，简称为LOTUS），取一组独立同分布的随机变量$\{X_i\}$，X_i在$[a,b]$上服从分布f_X，那么我们可以计算一个函数$g(x)$在区间$[a,b]$的期望：

$$(b-a)E[g(x)] = \int_a^b g(x) f_X(x) dx$$

当对于某些函数$g(x)$我们不知道其解析式或者其解析式计算代价十分高时，传统的求其积分的解析方法就失效了，而蒙特卡洛积分法可以有效避免这一尴尬情况转而直接求其积分的数值解。在我们对一个场景进行绘制时，场景的光场函数其解析式十分复杂（即麦克斯韦的电磁场方程），求每一个像素点对应的解析解是不可接受的，这个时候蒙特卡洛法就可以被用来对每个点进行光线采样和追踪以计算其幅度（radiance）了。

蒙特卡洛光线追踪算法

计算机图形学中一个十分重要的领域就是将多个三维网格组成的物体绘制，使用指定内参的相机投影成一幅二维图像，而在这种降维投影中涉及到很多技术：几何变换、消隐算法、纹理映射、阴影绘制等等。1986年Kajiya提出的光线追踪算法是一种根据光线碰撞的角度、材质等调整其成分模拟追踪光子路径的绘制算法，如下图所示：



由于它能够提供较好的环境模拟效果（阴影、反射、折射）而成为主流绘制算法之一。其算法流程主要可以由以下伪代码描述：

```
color trace_ray(ray, depth):
    intersection = get_intersection(ray) // get intersection generated by meshes and
forwarding ray

    if not intersection.hit:
        return BLACK        //ray forward to nothing

    if depth > MAX_DEPTH:
        return intersection.emission    //don't trace forever

    new_ray = montecarlo_sample(ray, intersection) // get a new ray originated from
intersection

    incoming_color = trace_ray(new_ray) // incident light
    return BRDF(intersection.material, incoming_color) //emergent light
```

这里需要注意到我们在从光线与物体的交点生成一条新的光线时使用了蒙特卡洛采样法。在理想情况下纯镜面反射时入射角与出射角相等，但是由于漫反射、折射的存在，在一个物体表面任意角度的入射光线所产生的某个角度的出射光线取决于该物体材质的BRDF函数，因此我们在交点处随机一个出射角并根据BRDF函数计算在该交点处提供的光线幅值，使用蒙特卡洛法进行多次采样可以逼近真实的幅值。

其他辅助算法

KD树

由于在光线追踪算法中存在大量的光线与网格的求交运算，如果直接使用三角网格与线的求交算法会导致整个算法十分之慢，我们可以建立一个KD树，为每个物体建立一个层次包围盒结构使用KD树管理，这样可以对求交运算大量加速。

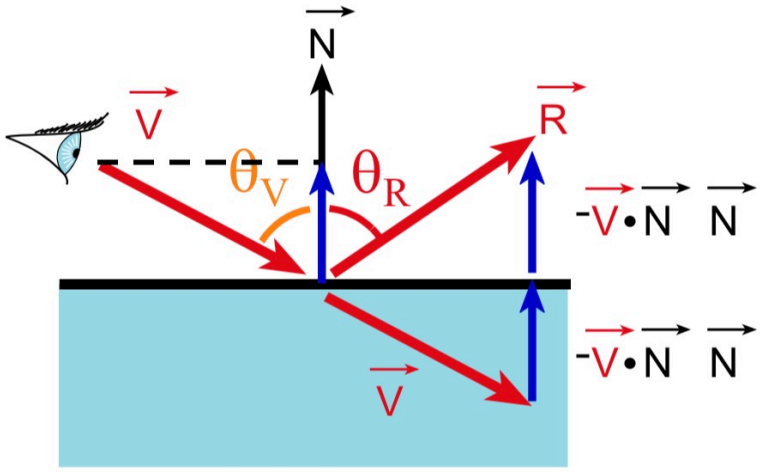
反射、折射的计算方法

这里不累赘的对光学原理进行介绍，一下两张图来自王锐老师课上的讲稿，我们直接参照这些光学公式进行计算：

反射公式：

Reflection

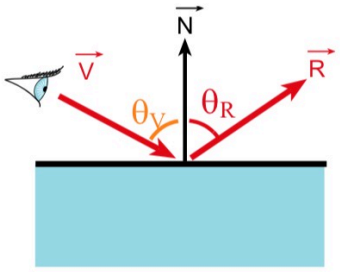
- Reflection angle = view angle
- **R** = **V** – 2 (**V** · **N**) **N**

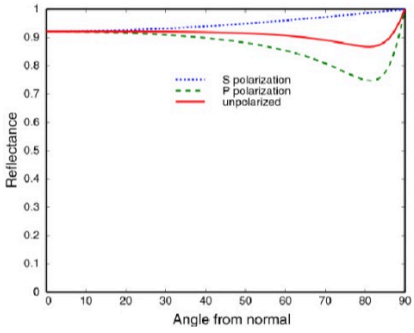


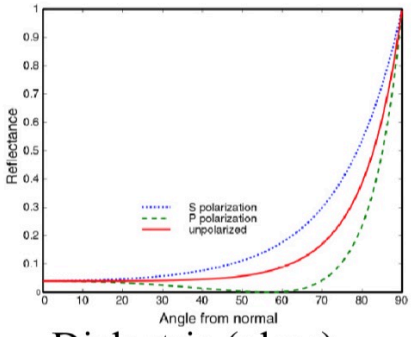
上面描述的是理想情况下的反射公式，但是真实情况与这个稍有出入，并非只有严格在反射角的方向才有光线飞出，在进行光线跟踪时，我们在严格的出射角附近随机一个出射角，并采用以下公式计算光线幅值的变化。

Amount of Reflection

- Traditional ray tracing (hack)
  - Constant **`reflectionColor`**
- More realistic:
  - Fresnel reflection term (more reflection at grazing angle)
  - Schlick's approximation: $R(\theta)=R_0+(1-R_0)(1-\cos\theta)^5$



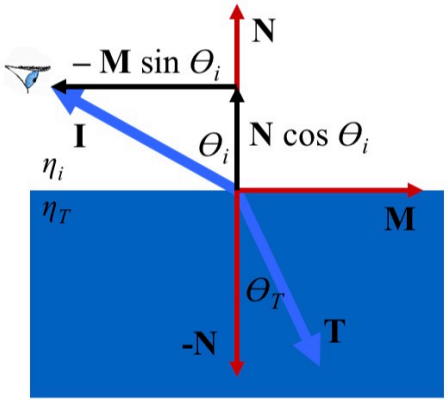




Dielectric (glass)

折射公式:

Refraction



Snell-Descartes Law:

$$\eta_i \sin\Theta_i = \eta_T \sin\Theta_T$$

$$\frac{\sin\Theta_T}{\sin\Theta_i} = \frac{\eta_i}{\eta_T} = \eta_r$$

$$\mathbf{I} = \mathbf{N}\cos\Theta_i - \mathbf{M}\sin\Theta_i$$

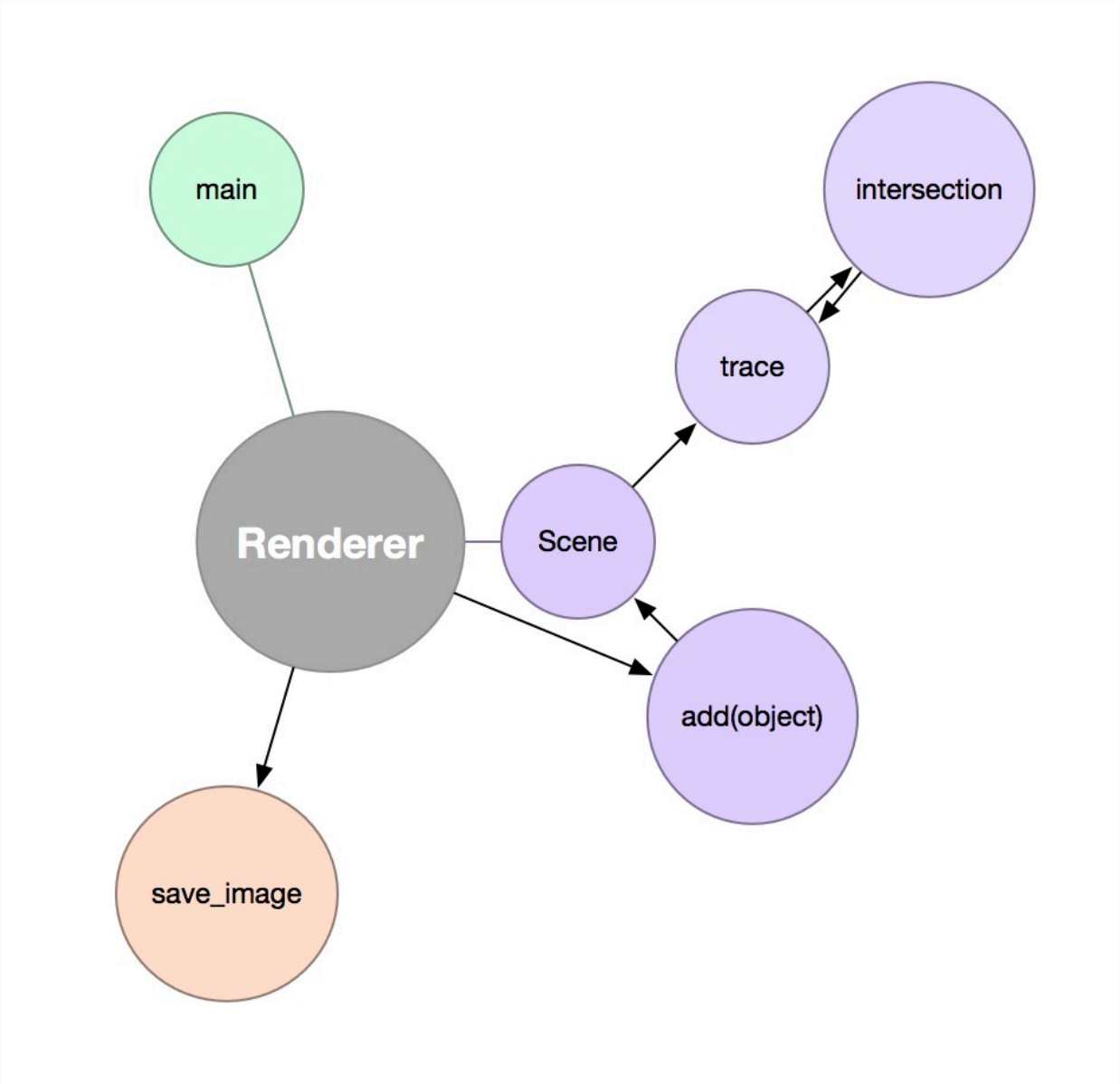
$$\mathbf{M} = (\mathbf{N}\cos\Theta_i - \mathbf{I}) / \sin\Theta_i$$

$$\begin{aligned}
\mathbf{T} &= -\mathbf{N}\cos\Theta_T + \mathbf{M}\sin\Theta_T \\
&= -\mathbf{N}\cos\Theta_T + (\mathbf{N}\cos\Theta_i - \mathbf{I})\sin\Theta_T / \sin\Theta_i \\
&= -\mathbf{N}\cos\Theta_T + (\mathbf{N}\cos\Theta_i - \mathbf{I})\,\eta_r \\
&= [\,\eta_r\cos\Theta_i - \cos\Theta_T\,]\,\mathbf{N} - \eta_r\mathbf{I} \\
&= [\,\eta_r\cos\Theta_i - \sqrt{1-\sin^2\Theta_T}\,]\,\mathbf{N} - \eta_r\mathbf{I} \\
&= [\,\eta_r\cos\Theta_i - \sqrt{1-\eta_r^2\sin^2\Theta_i}\,]\,\mathbf{N} - \eta_r\mathbf{I} \\
&= [\,\eta_r\cos\Theta_i - \sqrt{1-\eta_r^2(1-\cos^2\Theta_i)}\,]\,\mathbf{N} - \eta_r\mathbf{I} \\
&= [\,\eta_r(\mathbf{N}\cdot\mathbf{I}) - \sqrt{1-\eta_r^2(1-(\mathbf{N}\cdot\mathbf{I})^2)}\,]\,\mathbf{N} - \eta_r\mathbf{I}
\end{aligned}$$

- **Total internal reflection when the square root is imaginary**
- **Don't forget to normalize!**

工程实现

在实现过程中，我参考了JamesGriffin的整体架构如下：



并使用了以下几个开源库：

- 使用lodepng库输出png图片呈现绘制结果
- 使用tiny_obj_loader来读取obj和mtl文件
- 使用Eigen库完成底层的线性运算
- 使用nlohmann的json库进行参数配置，方便调试
- 对原有的JamesGriffin的KD树进行修改和改进以加速求交运算
- （可选）使用openmp进行多线程加速

在工程中，我主要实现了以下几个类：

- Timer - 处理计时功能。
- Mesh, Triangle, Material - 模型的网格内容，
- KdNode、AABBox - 按照空间排列组建KD树，树的叶子节点是一组空间上临近的三角形。
- Ray - 处理光线相关，反射、折射、光线的起点、方向、前进距离等。
- Scene - 处理好需要绘制的场景内容，如相机、被绘制物体等，不断利用Kd树与光线求交之后生成新的光线并迭代追踪。
- Renderer - 使用Scene处理整体绘制流程，在每个batch的迭代都输出一张图片以供查看进度。

下面我们分别介绍他们的功能与接口：

Time.h:

```
class timer {
    inline void start(); // start counting
    inline void stop(); // stop counting
    inline double get_seconds(); // return duration
    inline void reset(); // reset timer
};  // Recording duration

enum TIMER_TYPE {
    GLOBAL_TIMER,
    KDTREE_INTERSECTION_TIMER,
    TRIANGLE_INTERSECTION_TIMER,
    TIMER_NUM
};  // Convenient to index global timer
class global_timer {
    static std::vector<timer> timers;
};  // Recording duration globally
```

Scene.h:

```
class Scene {
    Scene(const Configuration &config_); // c'tor
    void add(Object *object); // add object into scene

    Intersection intersect(Ray &ray);
    // return intersection between this scene and the specified ray

    Vector3 trace_ray(Ray &ray, int depth);
    // ray tracing iteration

    Vector3 direct_illum(Ray &ray, Intersection& intersection); //
    Ray montecarlo_sample(Ray& ray, Intersection& intersection);
```

```
    bool russian_roulette(double probability, double& survival);
    Vector3 importance_sample(const Vector3& up, double n = -1);
};
```

Camera.h:

```
class Camera {
    Ray get_ray(int x, int y);
    // get a ray originating from pixel(x, y)
}
```

KdTree.h

```
class AABBox {
    inline void expand(const AABBox &box);
    inline void expand(const Vector3 &vec);
    // expand box to fit a specified box/point

    inline int get_longest_axis();
    // Returns longest axis: 0, 1, 2 for x, y, z respectively

    inline bool intersection(const Ray &r, double &t);
    // check intersection
};

class Triangle {
    inline bool intersect(Ray ray, double &t) const;
    // check intersection
    inline Vector3 barycentric(Vector3 p);
    // return barycentric coordinate
    inline Vector3 get_normal_at(const Vector3& p);
    // return normal by interpolation
    inline Material& get_material();
    // return material
};

class KDNode {
    KDNode* build(std::vector<Triangle*> &tris, int depth);
    // build KDTree
    bool hit (KDNode* node, Ray &ray, double &t, Intersection& intersection);
    // check intersection
};
```

Ray.h

```
struct Ray {
    inline Vector3 reflect(const Vector3& normal);
    // generate reflect ray
    bool refract(const Vector3& normal, double nit, Vector3& refract_direction);
    // generate refract ray
};
```

Renderer.h

```
class Renderer {
    void setup_scene();
    // preparation
    void render();
    // start rendering
    void save_image(const char * file_path);
};
```

Type.h

```
typedef double real_type;
typedef Eigen::Matrix<real_type, 3, 1> Vector3;
typedef Eigen::Matrix<real_type, 2, 1> Vector2;

//#define TIMER_COUNT
//#undef _OPEN_MP_

#define T_MIN 0.1
#define T_MAX 5.0
#define EPSILON 1e-4f

struct Configuration {
    std::string obj;
    std::string output_name;
    int width;
    int height;
    int samples;
    int batch_size;
    Vector3 camera_position;
    Vector3 look_at_point;
    Vector3 ambient_light;
    bool light;
};
```

关键代码分析

```
/*
   KD树节点与光线的求交主要分为两种：
      当我们遍历到它的叶子节点时直接对光线和该叶子节点包含的三角形面片求交。
      当我们遍历到它的中间节点时，我们对它的左右儿子节点分别判断求交：
         如果两个儿子节点都没有与光线相交则该节点不与光线相交
         如果两个儿子节点都与光线相交则取t值较小(更近)的相交节点
         如果只有一个交点，则使用该交点
*/

bool KDNode::hit(KDNode *node, Ray &ray, double &t, Intersection& intersection) {
   double dist;
   if (node->box.intersection(ray, dist)){
      if (dist + EPSILON > ray.tmax) {
         return false;
      }

      bool hit_tri = false;
      bool hit_left = false;
      bool hit_right = false;
      long tri_idx;

      if (!node->leaf) {  // check intersectio with each KDNode
         hit_left = hit(node->left, ray, t, intersection);
         hit_right = hit(node->right, ray, t, intersection);

         return hit_left || hit_right;
      }
      else {  // check intersection with the triangles
         long triangles_size = node->triangles.size();
         for (long i=0; i<triangles_size; i++) {
#ifdef TIMER_COUNT
            //global_timer::timers(TRIANGLE_INTERSECTION_TIMER).start();
#endif
            if (node->triangles(i)->intersect(ray, t)){
               hit_tri = true;
               ray.tmax = t;
               tri_idx = i;
            }
#ifdef TIMER_COUNT
            //global_timer::timers(TRIANGLE_INTERSECTION_TIMER).stop();
#endif
         }
         if (hit_tri) {
            Vector3 p = ray.origin + ray.direction * ray.tmax;
            intersection.point = p;
            intersection.n = node->triangles(tri_idx)->get_normal_at(p);
            intersection.u = ray.tmax;
            intersection.hit = true;
            intersection.m = node->triangles(tri_idx)->get_material();
            return true;
         }
      }
```

```
    }
    return false;
}
```

```
/*
    Ray中的reflect和refract函数，用来计算严格的反射和折射方向，相关公式在第二节中已经说明。
*/
    inline Vector3 reflect(const Vector3& normal)
    {
        return direction - 2.0f * normal.dot(direction) * normal;
    }

    bool refract(const Vector3& normal, double nit, Vector3& refract_direction)
    {
        double ndoti = normal.dot(direction),
               k = 1.0f - nit * nit * (1.0f - ndoti * ndoti);
        if (k >= 0.0f) {
            refract_direction.block(0, 0, 3, 1) = nit * direction - normal * (nit * ndoti + sqrt(k));
            return true;
        }
        // total internal reflection. There is no refraction in this case
        else return false;
    }
```

```
/*
    绘制的主要内容：
        1.根据宏选择是否使用OpenMP加速并给出提示
        2.从configuration中取出用户配置的参数进行配置
        3.从图像的每一个pixel位置中生成一条条光线进行光线追踪
        4.将中间结果以及最终结果进行保存
*/
void Renderer::render() {
#ifdef _OPEN_MP_    // open mp notification
    printf("Accelerate with openmp.\n");
#else
    printf("No openmp found.\n");
#endif
    const auto width = config.height;
    const auto height = config.width;

    const auto samples = config.samples;
    const auto batch_size = config.batch_size;
    const auto batch_recp = 1./batch_size;

    timer render_timer = timer();
    timer batch_timer = timer();
    char fn(100);
```

```
    setup_scene();
    // Main Loop
    int itr_times = config.samples/config.batch_size;
    for (int itr = 0; itr < itr_times; itr++) {
        render_timer.start();
        batch_timer.reset();
        batch_timer.start();
        fprintf(stderr, "\rRendering (%i samples): %.2f%% ", config.samples,
(double)itr/itr_times*100);
#ifdef _OPEN_MP_    // open mp acceleration
#pragma omp parallel for schedule(dynamic, 1)       // OpenMP
#endif
        for (int y=0; y<height; y++){
            for (int x=0; x<width; x++){
                Vector3 col = Vector3(0, 0, 0);

                for (int a=0; a<batch_size; a++){
                    Ray ray = m_camera->get_ray(x, y);
                    col = col + m_scene->trace_ray(ray, 0);
                }
                auto& pixel = m_pixel_buffer((y)*width + x);
                pixel = (pixel * itr + col * batch_recp)/(itr + 1);
            }
        }
        sprintf(fn, "%s_%d.png", config.output_name.c_str(), itr * batch_size);
        batch_timer.stop();
        printf("%lf secs per batch.\n", batch_timer.get_seconds());
        render_timer.stop();
#ifdef TIMER_COUNT
        printf("kdtree intersection per patch: %lf secs.\n",
global_timer::timers(KDTREE_INTERSECTION_TIMER).get_seconds());
        //printf("triangle intersection per patch: %lf secs.\n",
global_timer::timers(TRIANGLE_INTERSECTION_TIMER).get_seconds());
        global_timer::timers(KDTREE_INTERSECTION_TIMER).reset();
        global_timer::timers(TRIANGLE_INTERSECTION_TIMER).reset();
#endif
        save_image(fn); // termporal output
    }
    printf("\rRendering (%i samples): Complete!\nTime Taken: %lf secs\n\n",
config.samples, render_timer.get_seconds());
}
```

```
/*
    迭代的进行光线追踪，每一次获取当前光线与网格的交点，然后继续跟踪，使用depth来限
制跟踪的深度。
    每次新光线的生成使用蒙特卡洛采样。
*/
Vector3 Scene::trace_ray(Ray &ray, int depth) {
    Intersection intersection = intersect(ray);
    if (!intersection.hit) {
```

```
        return Vector3(0, 0, 0);
    }

    auto& m = intersection.m;
    if (depth > 5) {
        return m.get_emission() + m.m_ambient.cwiseProduct(ambient_light);
    }

    Vector3 color = Vector3(0, 0, 0);
    Ray newRay = montecarlo_sample(ray, intersection);

    if (newRay.type !=  INVALID) {
        color = trace_ray(newRay, depth+1);

        Vector3 reflect_direction = (newRay.direction - intersection.n * 2 *
newRay.direction.dot(intersection.n)).normalized();

        switch (newRay.type) {
            case DIFFUSE_REFLECT:
                color = m.m_diffuse.cwiseProduct(color);
                break;
            case SPECULAR_REFLECT:
                color = m.m_specular.cwiseProduct(color);
            case TRANS_REFRACT:
            default:
                break;

        }
    }

    return m.m_emission + color + m.m_ambient.cwiseProduct(ambient_light);
}
```

```
/*
    根据交点的材质、法向性质使用蒙特卡洛法生成一条新的光线，总共分为两种情况进行处
理：透射材质和非透射材质。
    透射材质：根据入射光线和表面法向的角度，确定是进入透射材质还是离开透射材质(即确定
ni和nt)，然后根据俄罗斯赌盘决定是反射还是透射。
    非透射材质：根据俄罗斯赌盘决定漫反射还是全反射，并利用重要性采样决定反射方向以提
升图像成像速度。
*/
Ray Scene::montecarlo_sample(Ray& ray, Intersection& intersection) {
    auto& m = intersection.m;
    Vector3 direction;
    double diff = m.m_diffuse.dot(Vector3(1, 1, 1));
    double spec = m.m_specular.dot(Vector3(1, 1, 1)) + diff;

    if (spec <= 0) {
        return Ray(intersection.point, direction);
    }
```

```
    if (m.m_type == TRANS) {
        //printf("mior = %lf\n",m.m_ior);
        double ni, nt;
        double costheta = ray.direction.dot( intersection.n );

        Vector3 n = costheta <= 0 ? intersection.n : -intersection.n;

        if (costheta > 0) {
            ni = m.m_ior;
            nt = 1.0;
        } else {
            ni = 1.0;
            nt = m.m_ior;
        }

        double f = (ni - nt) / (ni + nt);
        f = f*f;
        f = f + (1 - f) * pow(1 - fabs(costheta), 5);
        //printf("ni = %lf, nt = %lf, cos = %lf, f = %lf\n", ni, nt, costheta, f);
        double transmission_survival;
        if (russian_roulette(f, transmission_survival)) {
            if (ray.refract(n, ni / nt, direction)) {
                //printf("transparent.\n");
                return Ray(intersection.point, direction, TRANS_REFRACT);
            } else {
                direction = ray.reflect(n);
                return Ray(intersection.point, direction, SPECULAR_REFLECT);
            }
        }
    }

    double spec_survival;
    if (russian_roulette(diff / spec, spec_survival)) {
        Vector3 reflect_direction = ray.reflect(intersection.n);
        direction = importance_sample(reflect_direction, m.m_shiness);
        return Ray(intersection.point, direction, SPECULAR_REFLECT);
    }
    else {
        direction = importance_sample(intersection.n);
        return Ray(intersection.point, direction, DIFFUSE_REFLECT);
    }

}
```

实验结果

除了老师给出的scene01和scene02两个场景外，我还尝试了在free3D中找到的一些其他可用的物体，由于有些材质没有光源，自发光看起来很奇怪，我在外部场景中加入了一个球型光源以增强视觉效果。老师给的2017ACG的网站上提供的两个场景的obj和mtl文件都或多或少有问题（法向、材质等），我对给定的文件内容都进行了一定的手动修改。

本工程使用了OpenMP进行并行加速，如果希望耗时减少建议在Ubuntu或者Windows下先自行配置OpenMP。由于Mac平台的LLVM不支持openmp，CMakeLists.txt中已自动检测是否要启用openmp，因此没有openmp也可以编译运行。

我们可以使用以下方法编译工程，之后${PROJECT_SOURCE_DIR}/bin会出现可执行文件：

```
cd ${PROJECT_SOURCE_DIR}
mkdir build
cd build
cmake .. -DCMAKE_BUILD_TYPE=RELEASE #You can choose Debug
make
```

注意：

如果需要查看更多的时间损耗信息，可以在Type.h中启用这两个宏（关掉openmp是为了检测时间）：

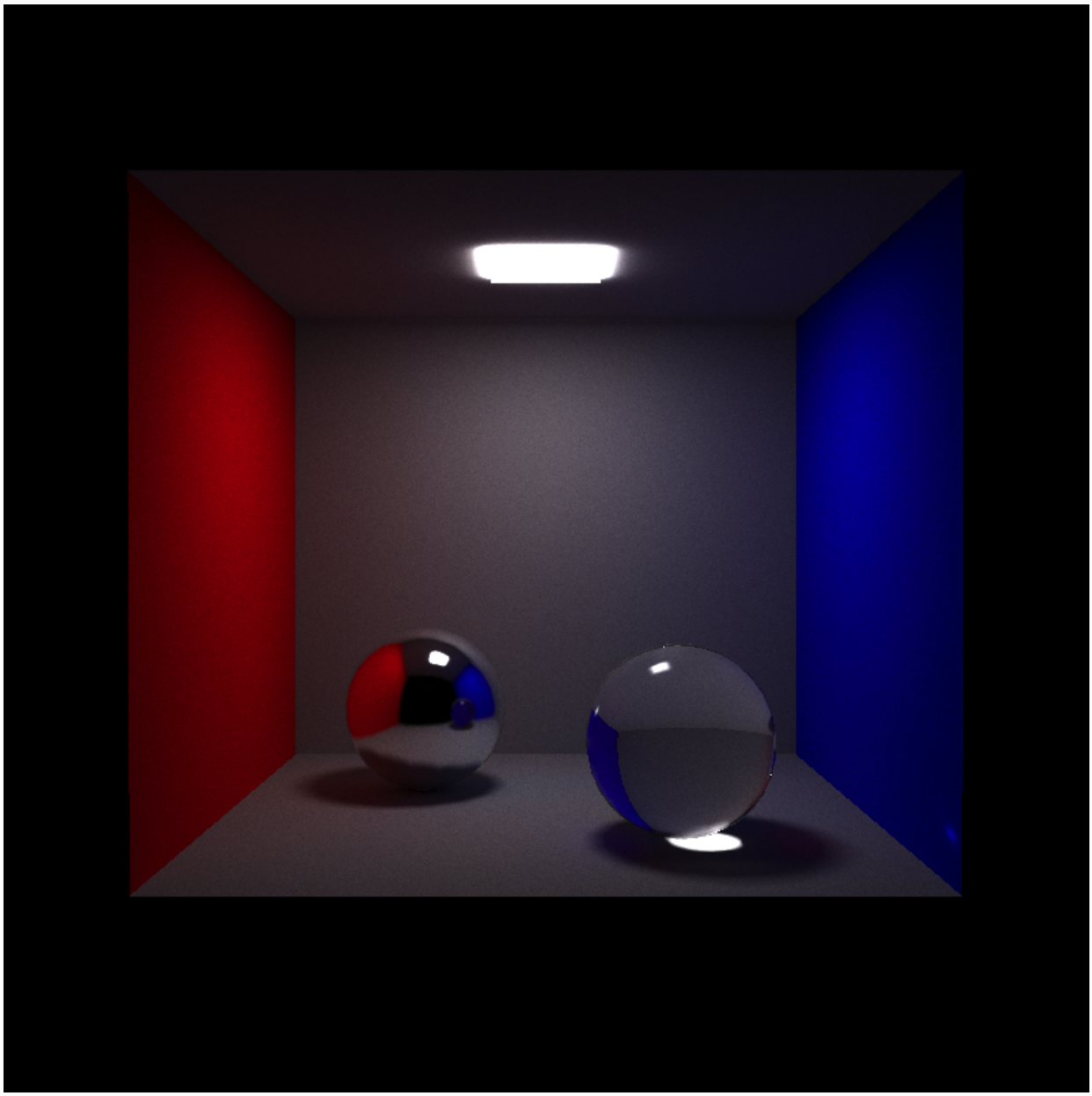
```
#define TIMER_COUNT
#undef _OPEN_MP_
```

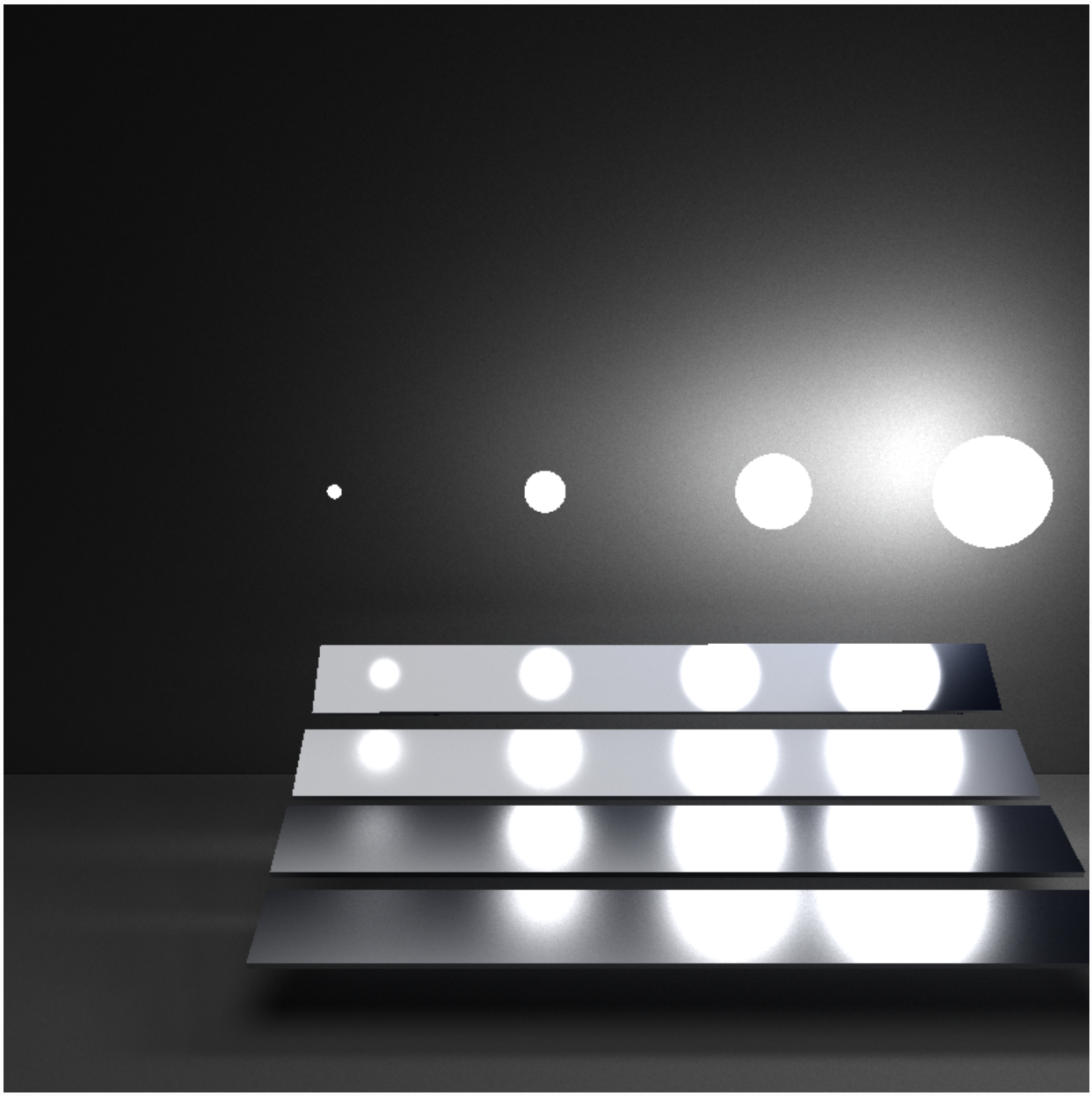
在bin目录下执行`./pathtracer`，pathtracer将读取该目录下的config.json文件作为配置进行绘制，下面我们简单解释一下配置文件的选项：

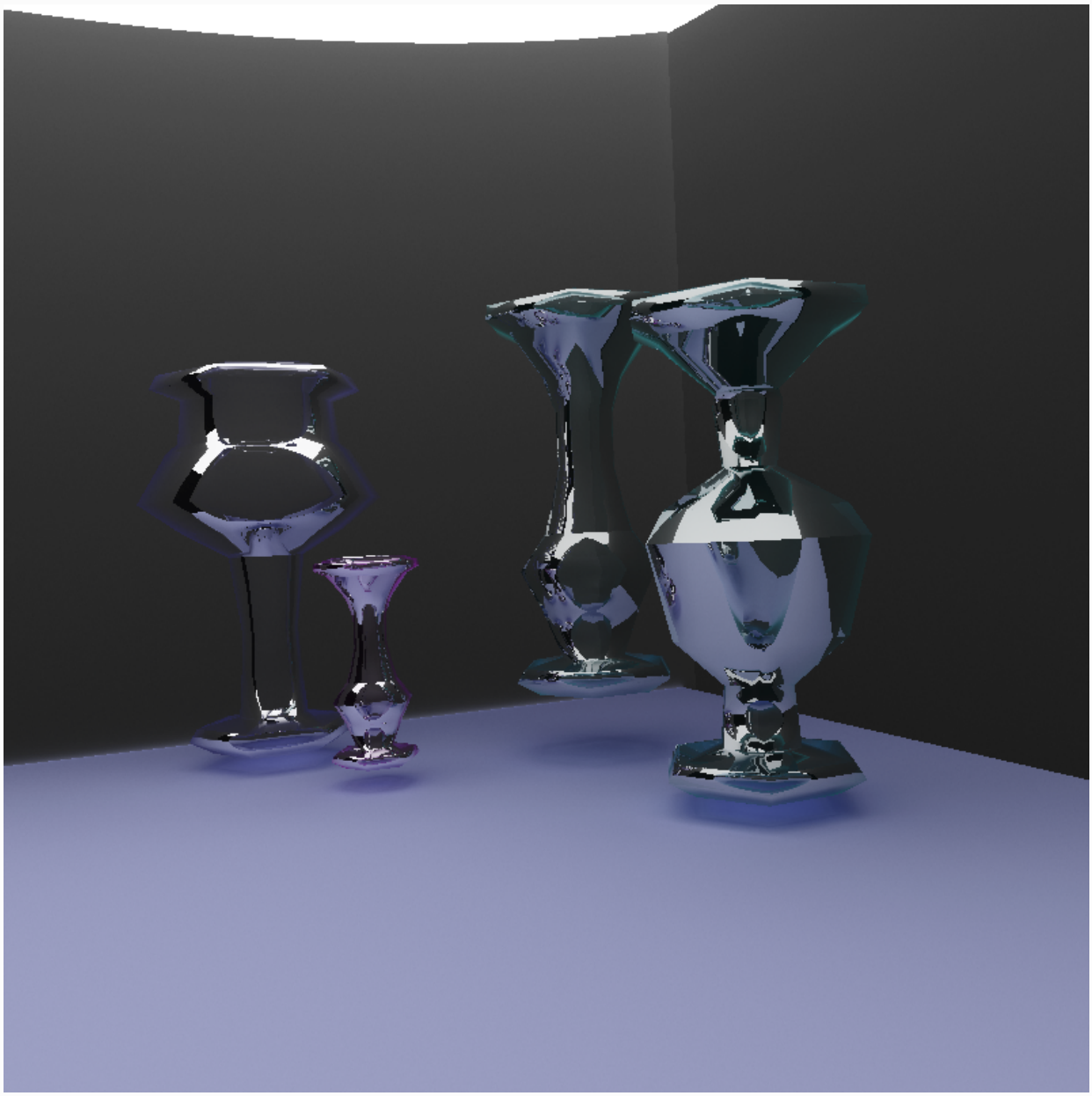
```
obj - 模型名称，该模型对应的材质文件请务必保证相同名称
output_name - 输出名称
width, height - 绘制图像结果的宽高

camera_x, camera_y, camera_z - 相机摆放的位置
look_at_x, look_at_y, look_at_z - 相机的朝向
ambient_light_r, ambient_light_g, ambient_light_b - 环境光幅值

light - 是否摆放外部光源

samples - 采样总数
batch_size - 由于等待整体绘制完成再查看结果太漫长，这里的batch_size指的采样多少次之后输出一次结果，输出的结果可以在本目录下找到。
```

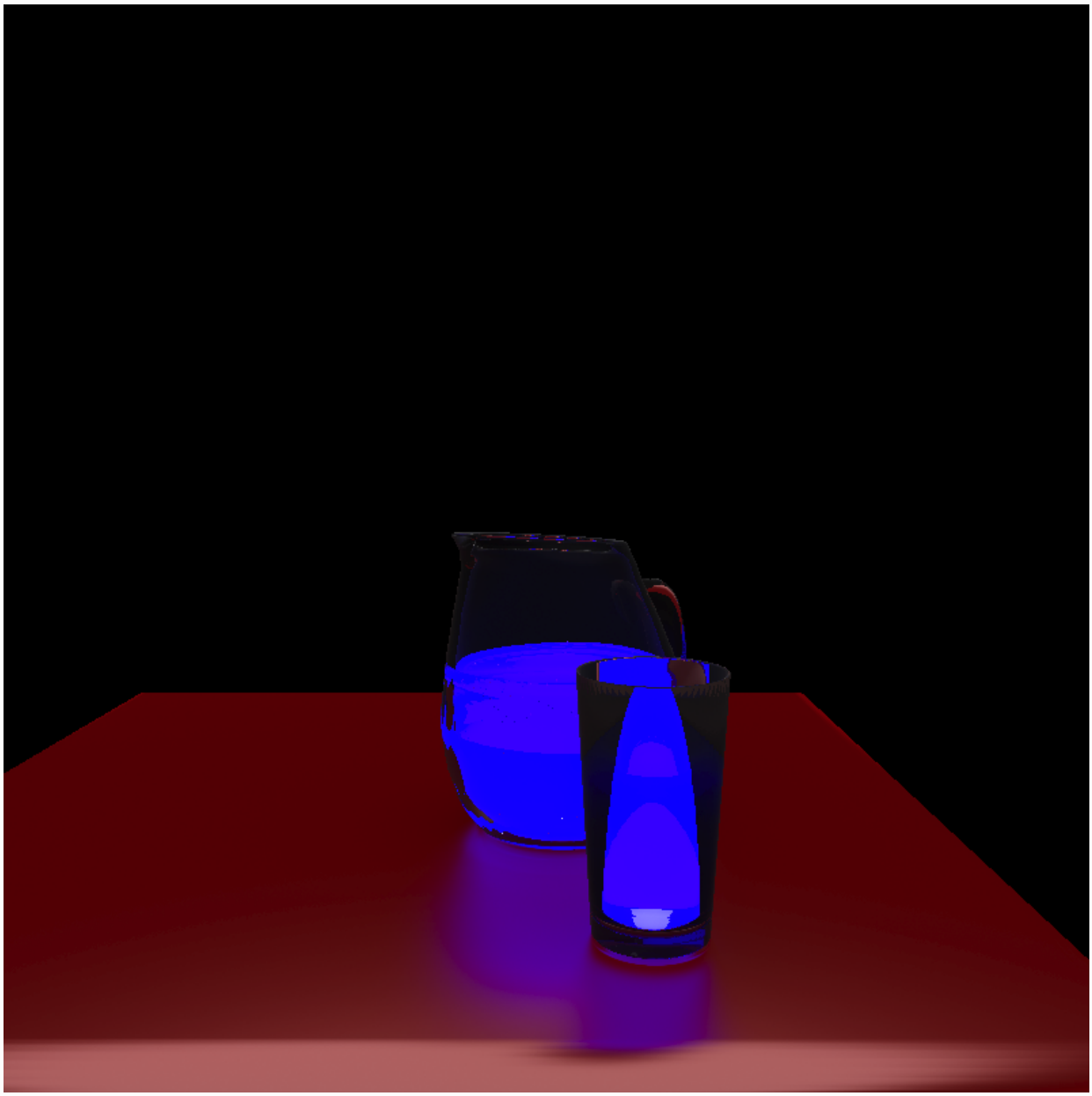
严格来说相机应该是用外参进行配置（Rotation, Translation），但是在调试过程中外参比较不方便，这里使用这种比较直观的方法进行配置。

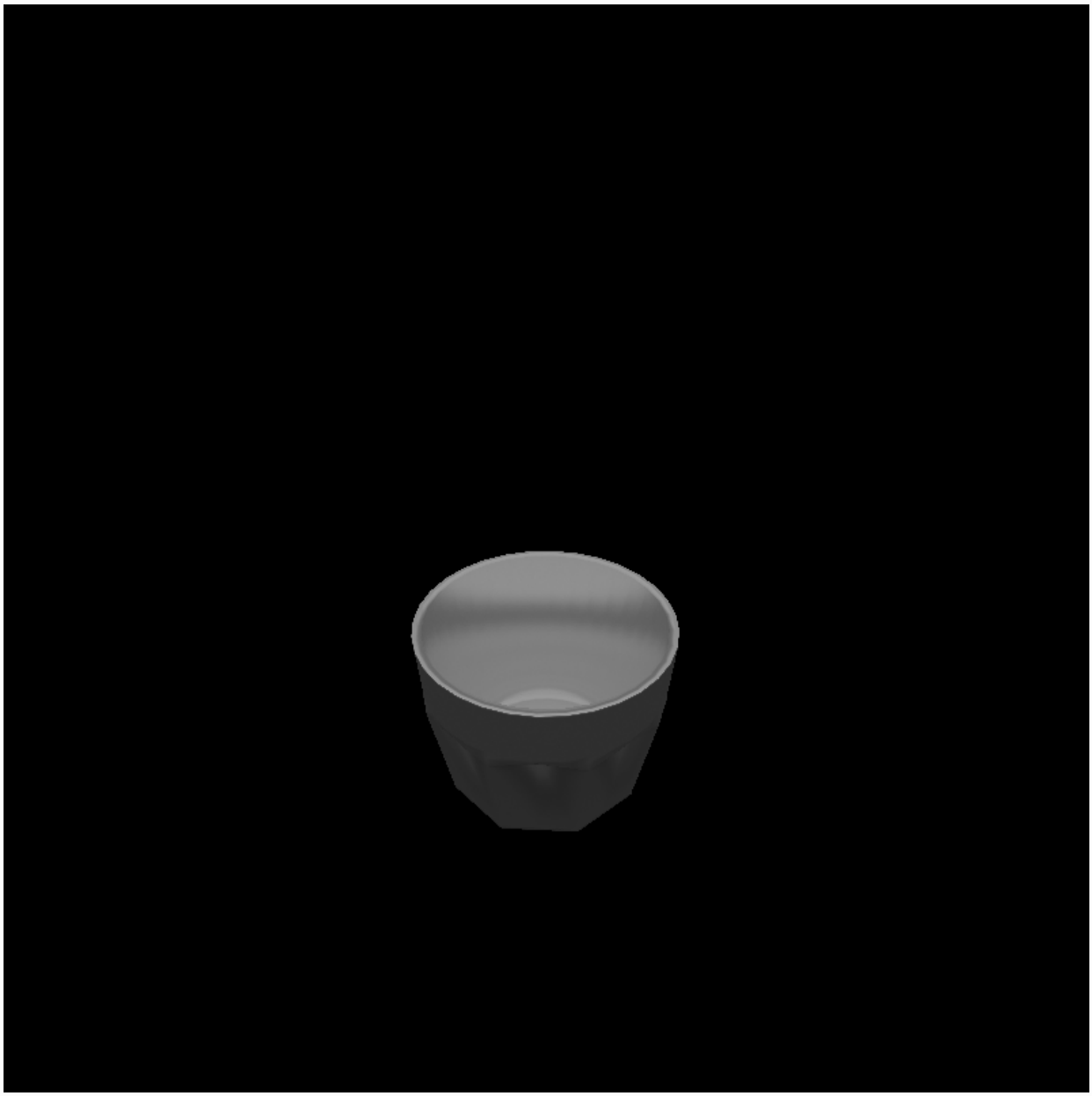
所有的模型都可以在${PROJECT_SOURCE_DIR}/obj下可以找到，所有的相关配置文件都可以在${PROJECT_SOURCE_DIR}/bin目录下可以找到，以下为实验结果展示：

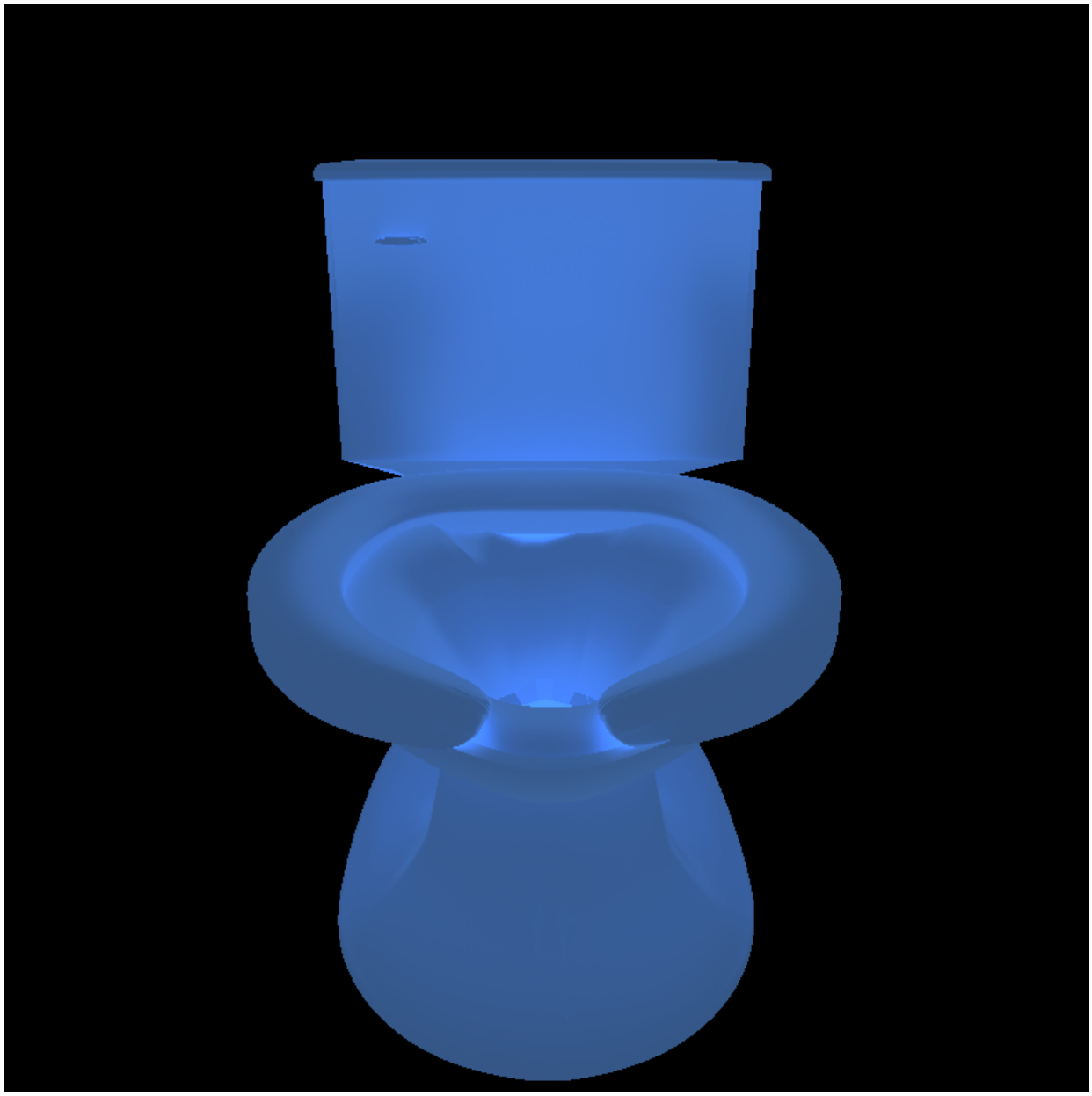












总结

首先阐述以下我在实现过程中遇到的问题：

1. 在实现过程中的debug相当之麻烦，因为采样光线很多，然而人眼并不能一眼看出数值是否正确，只能够看着公式一步步手动算数值，在debug的时候使用了固定采样方便自己的计算验证，当然得到的结果并不好，有一些花纹出现，之后修改成了使用russian roulette算法。
2. 显示屏的好坏相当影响人对绘制结果的判断，一开始使用了一个实验室淘汰下来的显示器导致跟同学进行结果对比时一直觉得不对，换了个显示器查看发现结果就没问题了。
3. 由于存在比较多的参数需要对不同的模型进行适配，比如相机的位置、朝向、是否放置额外光源等等，如果需要每一次都改动代码并重新编译运行则开发周期相当长，因此使用了json进行配置，不必要每次都修改代码，也方便老师检查工作。
4. 由于使用了tiny_obj_loader进行文件读取，很多从网站上load下来的模型都使用不了，之后我会尝试使用更加鲁棒的库来进行模型读取。

之后的工作：

1. 使用其他模型读取库来替换tiny_obj_loader获得更鲁棒的性能。
2. 目前的代码无法读取并绘制纹理，之后应该将纹理的处理加进去。
3. 在某些视角进行观察时可以发现绘制出来的图像存在走样问题，之后可以继续改进。