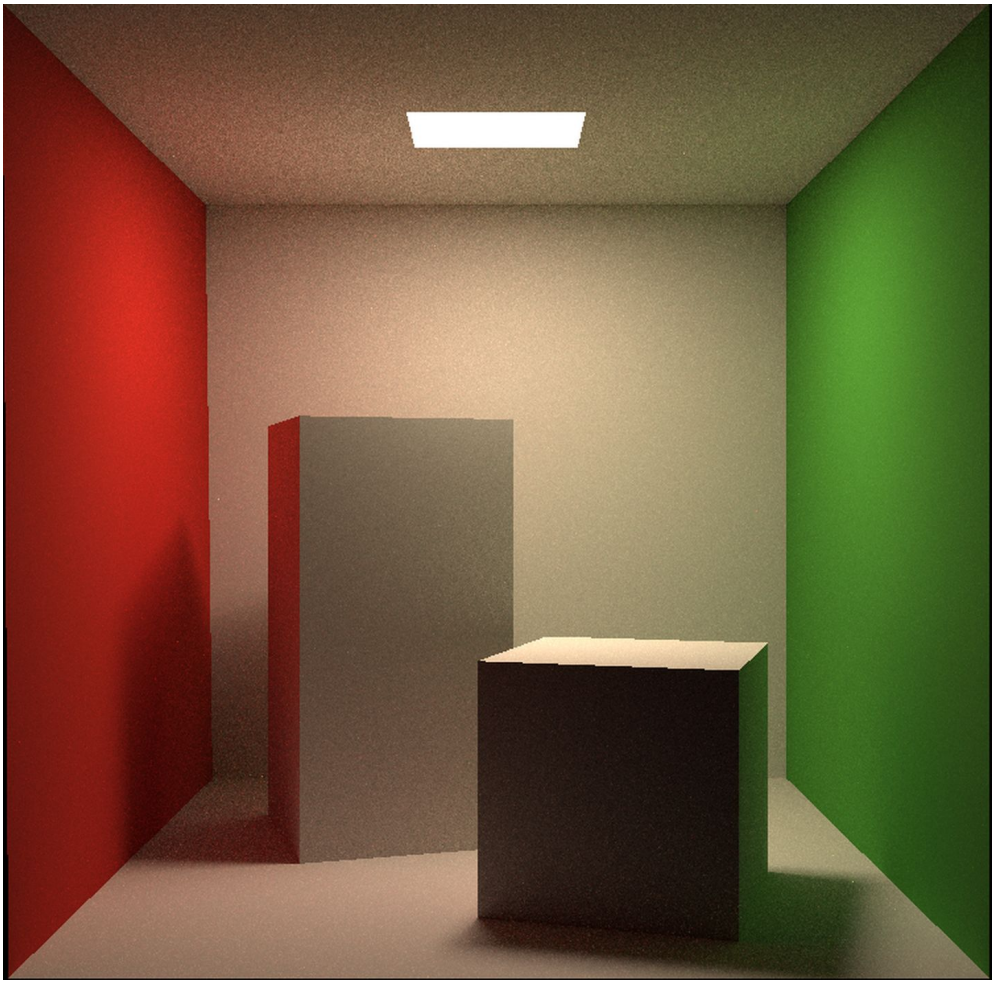
# 从零开始学图形学：写一个光线追踪渲染器（二）——微表面模型与代码实现



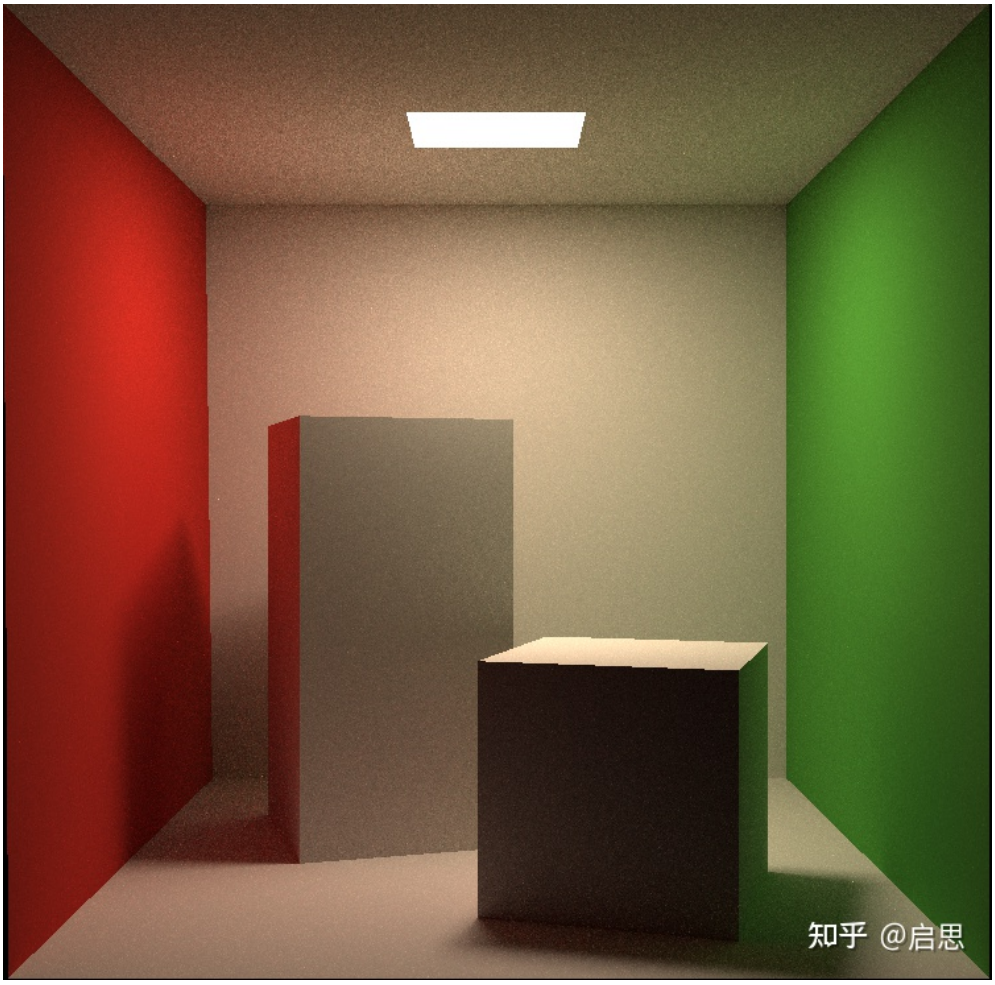
整个系列的文章归档整理于 **归档整理目录** 中。由于笔者能力有限，欢迎指正。

本文附有 GAMES101 实验七详解。

## 引入

**上一篇文章**中，我们在原理上介绍了如何实现一个光线追踪渲染器，包括渲染方程、蒙特卡洛积分、BxDF 相关内容。

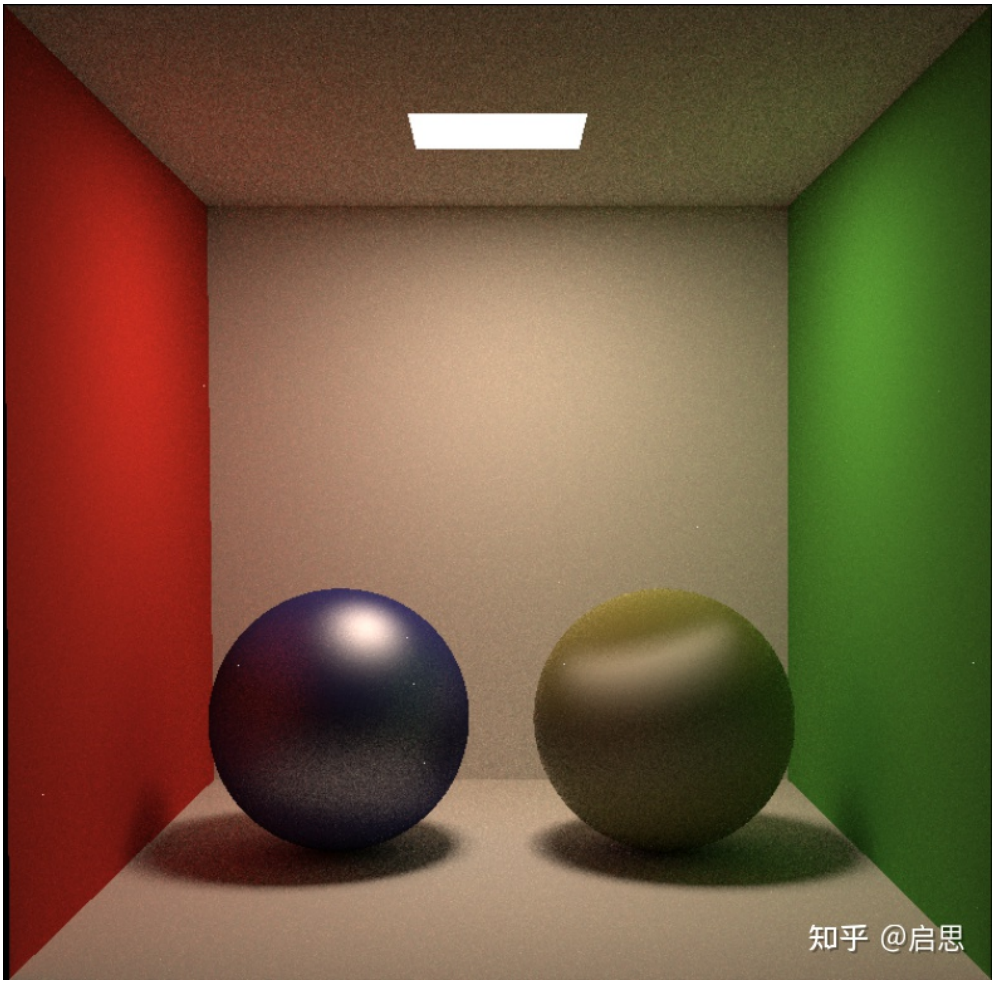
按照之前的介绍，正确实现后应该是这个样子：




256-spp, Cornell box

然而，仅仅有那些理论上的介绍还不够，真正实现时需要考虑并行加速等问题。由于本文仅介绍基于 CPU 的 path tracer，所以不涉及 GPU。

此外，对于物体的高光，微表面模型（Microfacet Model）给出了比 Blinn-Phong 模型更好的高光项。下图是二者的区别：




64-spp, 左侧是 Blinn-Phong model, 右侧是 Microfacet model, 参数都是瞎写的

本文中将先介绍 Microfacet Model ， 然后说一下实验代码。

# 微表面模型（Microfacet Model）

当物体表面平整光滑时，也就是我们平常使用的镜子，光线照上去，会产生完美的**镜面反射**。初中课堂上，关于**漫反射**现象，我们认为是光照射在由若干微小镜面构成的不规则表面上，形成的反射现象。

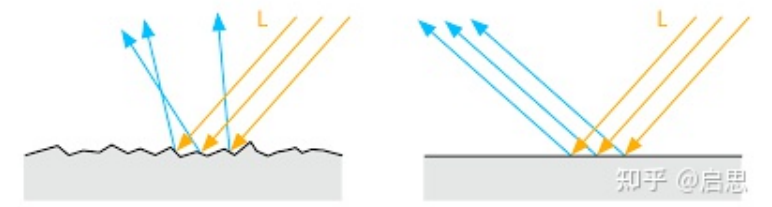



Image result for microfacet

漫反射现象是向四面八方均匀反射的。如果反射方向不那么均匀，或者说反射的方向与观察方向重合度很高，那是不是就会形成**高光**?

这种建模思维，其实就是微表面模型。


地球表面凹凸不平，但在更大的尺度上观察，凹凸不平的表面形成了微表面，产生了镜面反射的行为。https://twitter.com/Cmdr_Hadfield/status/318986491063828480

微表面模型尝试对微小镜面的统计现象进行建模，通过参数来调整属性。最经典的 Microfacet BRDF 来建模 Specular BRDF 的公式就是 Cook-Torrance 模型：

$$f(w_i, w_o) = \frac{D(h)F(w_i, h)G(w_i, w_o, h)}{4(n, w_i)(n, w_o)}$$

其中：

1. D：法线分布函数（Normal Distribution Function）。关于半程向量 h 的一个函数，反应了微表面的法线方向的分布函数。
2. F：菲涅尔项（Fresnel term）。通常，从不同角度看，物体表面反射的光的量也不同。例如公交车的车玻璃，从正面直视则几乎看不到反射，然而以一个几乎与之平行的角度去看则反射相当明显。该项通常与物体是否是导体有关。
3. G：几何衰减项（Geometric attenuation term），或者阴影遮蔽项（shadowing-masking term），指微表面形成的凹凸不平的形状会遮挡一部分光，即有一部分光会被表面本身遮挡，无法反射进入摄像机。
4. 分母是在校正。


Fresnel term。第三张图的反射更明显。 GAMES101 pdf17

D、F、G 三个函数有多种定义方式，并被多次改良。原始公式详见 **wikipedia** 。

# 代码实现

首先来整理一下目前的光线追踪实现步骤。

1. 摄像机对每个像素多次取样，每次取样射出一个射线。
2. 射线打在物体表面上，由某种概率分布（此处为均匀分布）在半球面上取样 N 次，每一次发射一根射线，递归计算。
3. 回溯后，对于 N 次的结果，收集起来，根据 BxDF 和概率分布将其加权累加。直到回溯到摄像机时，更新该像素的值。
4. 对于递归边界：设置最大深度或者俄罗斯轮盘（RR）法。

通常情况下我们取 N=1 ，这样可以防止递归爆炸。显然，就算这么做，我们只要对像素取样次数够多，答案仍然收敛到真实结果。

实际上，这里有一个问题：光线追踪方法是逆着光路来的，也就是说直到射线打到光源上，否则 shading 结果永远是 0。如果光源面积非常非常小，那么射线击中光源的概率也非常小，这样我们可能需要很多次采样才能得到一张不那么黑的图。

这促使我们**对光源单独采样。**我们将“以均匀分布在半球面上采样”修改为“**分别对光源和非光源进行均匀采样**”，便解决了这个问题。由于对非光源单独采样需要扣掉光源，比较麻烦，所以我们直接对半球面均匀采样，如果遇到光源则返回 0 。

即修改为：

1. 摄像机对每个像素多次取样，每次取样射出一个射线。
2. 如果射线打在了光源上，返回 0 。否则，射线打在物体表面上，考虑采样：
    1. 在所有光源上，按光源面积均匀采样一次，取着色点与光源取样点的连线为入射光方向，做 shading。此记为**直接光照**的 radiance。
    2. 在半球面上均匀采样一条射线，递归计算。
    3. 回溯后，对于 N 次的结果，收集起来，根据 BxDF 和概率分布将其加权累加。此记为**间接光照**的 radiance。
    4. 将直接光照和间接光照的 radiance 加起来，返回。
3. 直到回溯到摄像机时，更新该像素的值。
4. 对于递归边界：设置最大深度或者俄罗斯轮盘（RR）法。

伪代码如下：

```
shade(p, wo)
    # Contribution from the light source.
    Uniformly sample the light at x' (pdf_light = 1 / A)
    L_dir = L_i * f_r * cos θ * cos θ' / |x' - p|^2 / pdf_light

    # Contribution from other reflectors.
    L_indir = 0.0
    Test Russian Roulette with probability P_RR
    Uniformly sample the hemisphere toward wi (pdf_hemi = 1 / 2pi)
    Trace a ray r(p, wi)
    If ray r hit a non-emitting object at q
        L_indir = shade(q, -wi) * f_r * cos θ / pdf_hemi / P_RR

    Return L_dir + L_indir
```



伪代码。GAMES101课件

# 实验

1. 实现基本的光线追踪渲染器。
2. 多线程优化速度。
3. 实现微表面模型。

# 解析

首先修改两个原代码框架的错误。

1. global.cpp 中的 get_random_float 函数，关于随机数生成器的对象每次都要新建，相当慢。需要改成 static ，提速相当明显。
2. 测试微表面模型需要用到球，但是 Sphere.hpp 中的 getIntersection 函数判断相交有问题，精度不够，需要改成较大的判断条件，例如 t0>0.5 。

实现 path tracer 就是 Scene.cpp 的 castRay 函数，跟着伪代码来就行。注意，摄像机的射线照到光源需要特殊处理，即 depth 为 0 时，返回 emit。

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

    Intersection intersection = intersect(ray);

    if(!intersection.happened)
        return Vector3f(0, 0, 0);
    if(intersection.emit.norm()>0) {
        if(depth == 0)
            return intersection.emit;
        else
            return Vector3f(0, 0, 0);
    }

    Vector3f& p = intersection.coords;
    Vector3f wo = normalize(-ray.direction);
    Vector3f normal = normalize(intersection.normal);
    Material*& material = intersection.m;
```

```
    auto format = [](Vector3f &a) {
        if(a.x < 0) a.x = 0;
        if(a.y < 0) a.y = 0;
        if(a.z < 0) a.z = 0;
    };

    // direct
    Vector3f L_direct;
    {
        Intersection inter_dir;
        float pdf_dir;
        sampleLight(inter_dir, pdf_dir);

        Vector3f& x = inter_dir.coords;
        Vector3f ws = normalize(x - p);
        Vector3f light_normal = normalize(inter_dir.normal);

        auto pws = intersect(Ray(p, ws));
        if(pws.happened && (pws.coords-x).norm() < 1e-2) {
            L_direct = inter_dir.emit * material->eval(ws, wo, normal) * dotProduct(normal, w
                * dotProduct(light_normal, -ws) / (dotProduct((x-p), (x-p)) * pdf_dir);
            format(L_direct);
        }
    }
    // indirect
    Vector3f L_indirect;
    {
        float RR = this->RussianRoulette;
        if(get_random_float() < RR) {
            Vector3f wi = normalize(material->sample(wo, normal));
            L_indirect = castRay(Ray(p, wi), depth+1)
                * material->eval(wi, wo, normal) * dotProduct(wi, normal)
                / (material->pdf(wi, wo, normal) * RR);
            format(L_indirect);
        }
    }
    return L_direct + L_indirect;
}
```

多线程只需要在摄像机发射射线时用 std::thread 把图像分为几个块去做即可。

```
    int spp = 64;
    std::cout << "SPP: " << spp << "\n";

    int process = 0;
    auto deal = [&](int lx,int rx,int ly,int ry) {
        for (uint32_t j = ly; j <= ry; ++j) {
            int m = j * scene.width + lx;
            for (uint32_t i = lx; i <= rx; ++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++;
                process ++;
            }
            {
                std::lock_guard<std::mutex> g1(mutex_ins);
                UpdateProgress(1.0*process / scene.width / scene.height);
            }
        }
    };
    int minx = 0, maxx = scene.width-1;
    int miny = 0, maxy = scene.height-1;

    int bx = 5, by = 5;
    int nx = (scene.width+bx-1) / bx, ny = (scene.height+by-1) / by;
    std::thread th[bx * by];
```

```
    for(int i = 0,id = 0;i < scene.width;i += nx) {
        for(int j = 0;j < scene.height;j += ny) {
            th[id] = std::thread(deal, i, std::min(i+nx, scene.width)-1,
                                       j, std::min(j+ny, scene.height)-1);
            id ++;
        }
    }

    for(int i = 0;i < bx*by;i ++) th[i].join();

    UpdateProgress(1.f);
```

修改微表面模型，只需要修改 eval 函数（该函数计算 BxDF 值），套公式即可。

```
Vector3f Material::cookTorrance(const Vector3f &wi, const Vector3f &wo, const Vector3f &N) {
    // return 0;

    auto V = wo;
    auto L = wi;
    auto H = normalize(V + L);
    auto type = m_type;

    if(!(dotProduct(N, V) > 0 && dotProduct(N, L) > 0)) return 0;

    auto getFresnel = [&]() {
        static double n_air = 1, n_diff = 1.2, n_glos = 1.2;
        double n1 = n_air, n2;
        if(type == MaterialType::MICROFACET_DIFFUSE) n2 = n_diff;
        else n2 = n_glos;
        auto costheta = dotProduct(N, V);
        double r0 = (n1-n2)/(n1+n2); r0*=r0;
        return r0+(1-r0)*pow(1-costheta, 5);
    };


    float F;
    double G, D; // Fresnel, Geometry, Distribution
    {
        // double F2 = getFresnel();
       fresnel(wi, N, 1.2f, F);
    //    std::cout << F << " F " << F2 << std::endl;
        // F = F2;
    }
    {
        double G1 = 2 * dotProduct(N, H) * dotProduct(N, V) / dotProduct(V, H);
        double G2 = 2 * dotProduct(N, H) * dotProduct(N, L) / dotProduct(V, H);
        G = clamp(0, 1, std::min(G1, G2));
    }
    {
        double m; // lager, more diffused
        if(type == MaterialType::MICROFACET_DIFFUSE) m = 0.6;
        else m = 0.2;
        double alpha = acos(dotProduct(H, N));
        D = //(dotProduct(N,H) > 0) *
            exp(-pow(tan(alpha)/m, 2)) / (M_PI*m*m*pow(cos(alpha), 4));
    }
    auto ans = F * G * D / (dotProduct(N, L) * dotProduct(N, V) * 4);
    return ans;
}

static Vector3f eval_diffuse(const Vector3f &wi, const Vector3f &wo, const Vector3f &N) {
    return 1.0 / M_PI;
}

Vector3f Material::eval(const Vector3f &wi, const Vector3f &wo, const Vector3f &N){
    switch(m_type){
        case DIFFUSE: //case MICROFACET_DIFFUSE: case MICROFACET_GLOSSY:
        {
            float cosalpha = dotProduct(N, wo);
            if (cosalpha > 0.0f) {
                return Kd * eval_diffuse(wi, wo, N);
            }
```

```
                else
                    return Vector3f(0.0f);
                break;
            }
            case MICROFACET_DIFFUSE:
            {
                float cosalpha = dotProduct(N, wo);
                if (cosalpha > 0.0f) {
                    auto ans = Ks * cookTorrance(wi, wo, N) + Kd * eval_diffuse(wi, wo, N);
                  //  clamp(0, 1, ans.x); clamp(0, 1, ans.y); clamp(0, 1, ans.z);
                    return ans;
                }
                else
                    return Vector3f(0.0f);
                break;
            }
            case MICROFACET_GLOSSY:
            {
                float cosalpha = dotProduct(N, wo);
                if (cosalpha > 0.0f) {
                    double p = 25;
                    auto h = normalize(wi + wo);
                    double spec = pow(std::max(0.0f, dotProduct(N, h)), p);
                    auto ans = Ks * spec + Kd * eval_diffuse(wi, wo, N);
                  //  clamp(0, 1, ans.x); clamp(0, 1, ans.y); clamp(0, 1, ans.z);
                    return ans;
                }
                else
                    return Vector3f(0.0f);
                break;
            }

        }
}
```

本文首发于知乎专栏**图形图像与机器学习**，以后会常更新，欢迎大家的关注与催更。