计算机图形学大作业A1报告

专业：计算机科学与技术

学号：2113352 2113099

姓名：刘松卓 祝天智

2024年6月14日

1 摘要

1.1 大作业实现

在完成本学期课程后，我们基于计算机图形学课程提供的渲染框架，完成了对渲染框架的光线跟踪核心算法改进和优化，基于原有的Simple Path Tracer，通过实现导体材质着色器以及绝缘体材质着色器，增添了反射以及折射，我们实现了Path Tracing 导体材质以及Path Tracing 绝缘体材质等渲染算法，并且实现了PVH树来加速光线物体求交算法，实现了对渲染算法性能的优化。

2 增添导体材质着色器

在框架内提供的Lambertian漫反射着色器的基础上，根据gitte源代码库推荐，我决定先实现Path Tracing 导体材质，即实现导体材质着色器，先观察图片材质的属性，导体材质的属性是reflect，姑且理解为反射率，而我们要实现镜面反射的效果。在实现过程中，反射光线的计算通常遵循反射定律，即反射角等于入射角。在3D图形学中，反射光线的计算可以通过向量运算来实现。以下是反射光线计算的数学公式：

$$R = V - 2 \times (\text{dot}(V, N) \times N)$$

其中：

- R 是反射光线的方向向量。
- V 是入射光线的方向向量。
- N是表面的法向量，通常需要标准化（即长度为1）。
- $dot(V, N)$是 V和 N 的点积。

首先计算入射光线 (V) 与法向量 (N) 的点积，然后将这个点积乘以2倍的法向量 (N)，从入射光线 (V) 中减去这个结果，得到反射光线 (R)。

这是反射伪代码的实现。

```
Vec3 glm::reflect(const Vec3& I, const Vec3& N) {
    float dotProduct = glm::dot(I, N);
    return I - 2.0f * dotProduct * N;
}
```

算出反射光线的方向之后，接下来就要考虑反射光线的强度，BRDF 用于描述在不同入射和观察方向下，表面如何反射光线。

$$f_{BRDF}(x, w_i, w_o) = F_r(w_r)\frac{\delta(w_i - w_r)}{cos\theta_i}$$

BRDF通过以上公式求得，其中，

- w_r为反射光线
- $F_r(w_r)$为菲涅尔项

而菲涅尔项的计算公式我们根据此处获得的属性仅有reflect反射率一种，菲涅尔系数描述了入射光线在不同入射角度下，反射和折射的比例。菲涅尔方程（Fresnel Equation）描述了光在不同介质之间发生反射和折射时的强度变化。菲涅尔效应是光学中的一个重要现象，它解释了为什么不同角度的光线在光滑表面上的反射强度会有所不同。

菲涅尔方程通常表示为两个反射和折射系数的比率，这些系数取决于入射角和光线的偏振状态。

菲涅尔方程包括两个方程，分别是垂直入射情况下的菲涅尔方程和水平入射情况下的菲涅尔方程。

1. 垂直入射（s极性）情况下的菲涅尔方程：

反射系数：

$$R_s = \left| \frac{n_1 \cos(\theta_i) - n_2 \cos(\theta_t)}{n_1 \cos(\theta_i) + n_2 \cos(\theta_t)} \right|^2$$

透射系数：

$$T_s = 1 - R_s$$

2. 水平入射（p极性）情况下的菲涅尔方程：

反射系数：

$$R_p = \left| \frac{n_1 \cos(\theta_t) - n_2 \cos(\theta_i)}{n_1 \cos(\theta_t) + n_2 \cos(\theta_i)} \right|^2$$

透射系数：

$$T_p = 1 - R_p$$

其中，n_1和n_2分别为第一个介质和第二个介质的折射率，θ_i为入射角，θ_t为透射角。这些方程可以帮助计算光线从一个介质到另一个介质时的反射和透射的比例。对于非偏振光，菲涅尔方程可以简化为一个关于反射和折射的通用表达式。对于一个给定的入射角 θ_i（入射光线与表面法线之间的角度），菲涅尔反射比（Fresnel Reflectance）可以近似为：

$$R(\theta_i) = \left(\frac{n_t - n_i}{n_t + n_i}\right)^2 \left(\frac{1 - \cos(\theta_i)}{1 + \cos(\theta_i)}\right)^2$$

其中：

- $R(\theta_i)$是反射比，表示反射光强与入射光强的比例。
- n_i 是入射介质的折射率。
- n_t 是透射介质的折射率。
- θ_i 是入射角。

在实际应用中，特别是计算机图形学中，菲涅尔效应通常使用一个更简单的模型来近似，这个模型基于舒伯特（Schlick）近似。舒伯特近似提供了一个快速计算菲涅尔效应的方法，它避免了使用复杂的折射率和角度值，而是使用一个通用的菲涅尔公式：

$$F(\theta_i) = R_0 + (1 - R_0) \times (1 - \cos(\theta_i))^5$$

其中：

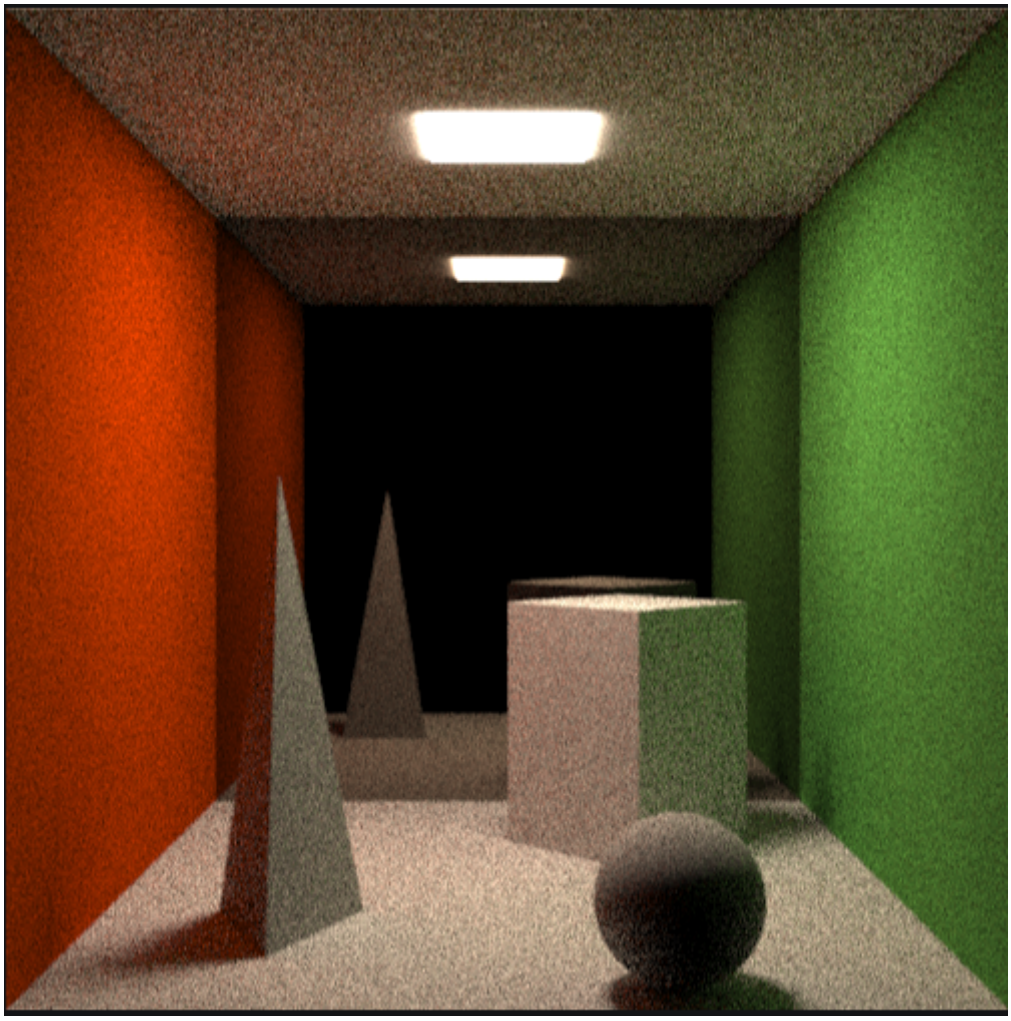
- $F(\theta_i)$ 是反射光的强度。
- R_0 是垂直入射时（即 (θ_i= 0) 度）的反射比，通常称为F0或Fresnel零阶项。
- (θ_i) 是入射角。

舒伯特近似公式通过一个五次幂的项来模拟菲涅尔效应，使得在接近法线（即 (θ_i) 接近0度）时反射强度较低，而在接近掠射角（即 (θ_i) 接近90度）时反射强度较高。

```
Scattered Conductor::shade(const Ray& ray, const Vec3& hitPoint,const Vec3& normal)const {
    //首先计算出射方向
    Vec3 V = glm::normalize(ray.direction);//计算入射方向
    Vec3 N = glm::normalize(normal);
    // 计算反射光线方向
    Vec3 R = glm::reflect(V, N);
    //计算导体的菲涅尔系数
    float cosTheta = glm::dot(N, V);
```

```
    Vec3 F0(1.0f); // 初始化
    F0 = F0 - this->reflect; // 逐元素减法
    F0*= pow(1 - abs(cosTheta), 5);
    F0 += this->reflect ;

    Vec3 BRDF = F0;
    BRDF*= abs(glm::dot(N, V));
    Vec3 attenuation = BRDF * abs(glm::dot(R, N)) * this->reflect;

    Scattered s;
    s.ray = Ray(hitPoint, R);
    s.attenuation = Vec3(1.0f);
    s.attenuation = attenuation;
    //getServer().logger.log("111111111");
    return s;
}
```

实现效果：



3 增添绝缘体材质材质着色器

绝缘体类似于玻璃中同时包含反射和折射,也需要通过菲涅尔系数计算反射和折射光线的强度，通过Schlick近似公式计算反射率，而公式中的$F_r(\theta_i) = R_0 + (1 - R_0)(1 - \cos(\theta_i))^5 R_0$ 可以通过以下公式计算：

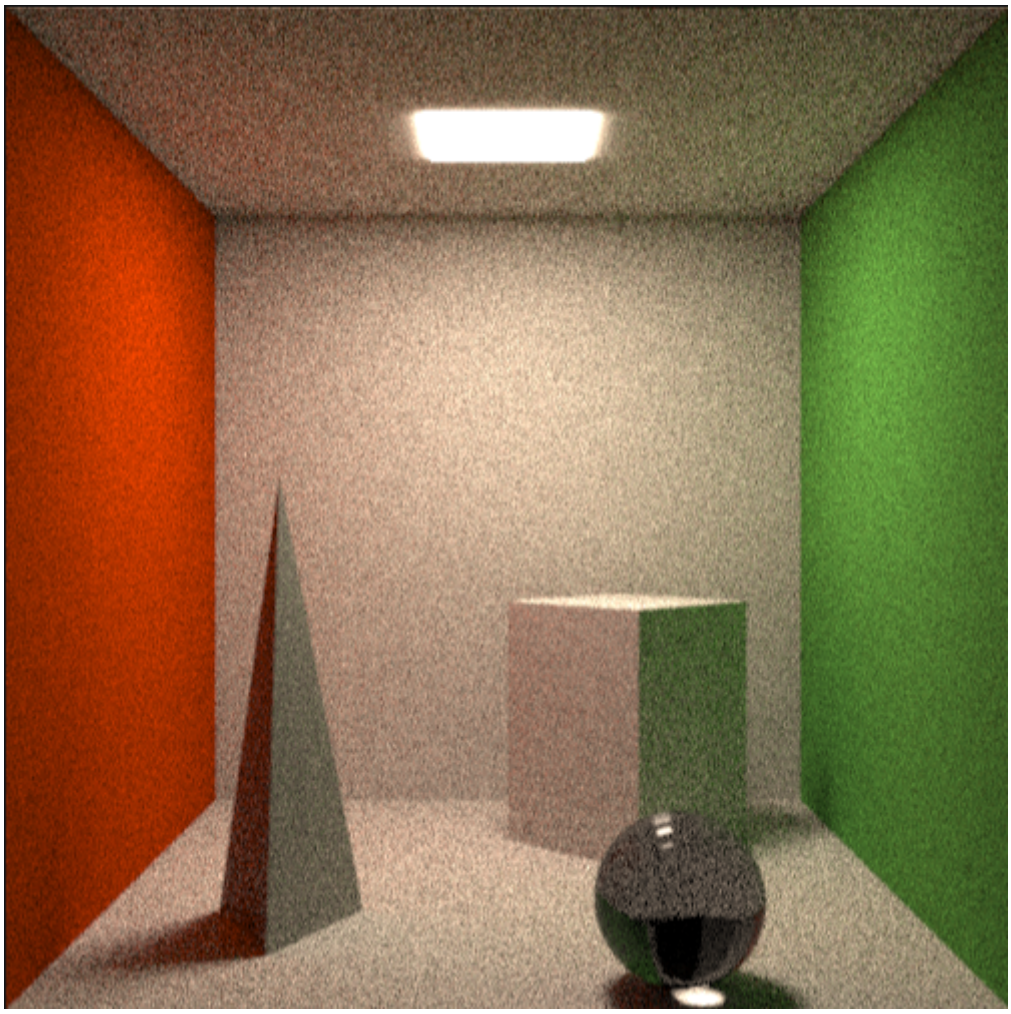
$$R_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2$$

n1和n2分别是两个介质中的折射率，我在代码的实现中分别根据空气射入物体和物体射入空气分为两种情况，一种n1为空气，另一种n2为空气，则置为1来简化计算。

折射角的计算通过$n_1 \sin(\theta_1) = n_2 \sin(\theta_2)$计算，$\theta_1$ 是入射角，θ_2 是折射角，n_1 和 n_2 是两种介质的折射率。计算完菲涅尔系数后，可以得出反射光线的强度和折射光线的强度，最后返回一个 `Scattered` 结构，包含反射光线、折射光线、反射和折射的颜色衰减

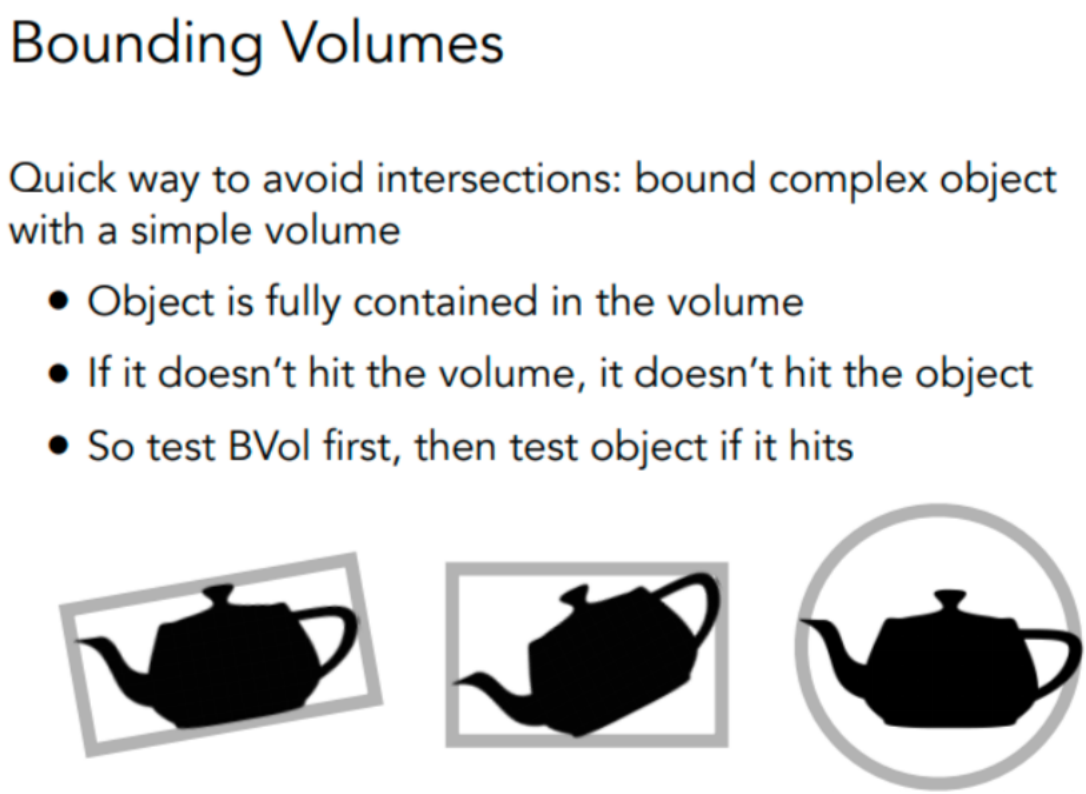
```
Scattered Dielectric::shade(const Ray& ray, const Vec3& hitPoint, const Vec3& normal)const {
        Vec3 begin = hitPoint;
        Vec3 V = glm::normalize(ray.direction);//计算入射方向
        Vec3 N = glm::normalize(normal);
        Vec3 R = glm::reflect(V, N);//反射
        float trueIor = 1.0f / ior;
        //计算导体的菲涅尔系数
        float cosTheta = glm::dot(N, V);

        float R0 = (1 - ior) / (1 + ior);
        R0 = R0 * R0;   //计算反射率
        float F0=1; //
        F0 -=  R0; //
        F0 *= pow(1 - abs(cosTheta), 5);
        F0 += R0;  //反射率

        Vec3 Reflex = F0 * absorbed;//反射率和自身颜色
        Vec3 Refraction = (1-F0) * absorbed;//除去反射和自身颜色
        if (cosTheta > 0) {//里面射到外面,所以是锐角,法向量朝向外
            N = -N;
            trueIor = ior;
        }

        Vec3 T; // 折射光线
            // 光线从外部射入
        T = glm::refract(V, N, trueIor);

        if (!T.x && !T.y && !T.z) { // 折射光线是零向量
            Refraction = Vec3(0, 0, 0); // 没有折射光线
        }
        Scattered s;
        s.ray = Ray(begin, R);
        s.attenuation = Reflex;
        s.refraction = Ray{ begin, T };
        s.refractionRate = Refraction;

        //getServer().logger.log("111111111");
        return s;
    }
```

实现效果：



4 BVH加速

在原本计算光线物体交点的方法中是遍历所有物体来寻找光线与物体的交点，在物体足够多的时候会造成极大的时间复杂度，影响算法效率，BVH通过将场景中的几何体组织成树状结构，可以快速排除那些光线不可能与之相交的几何体，减少必要的相交测试数量。BVH的学习原理如下。

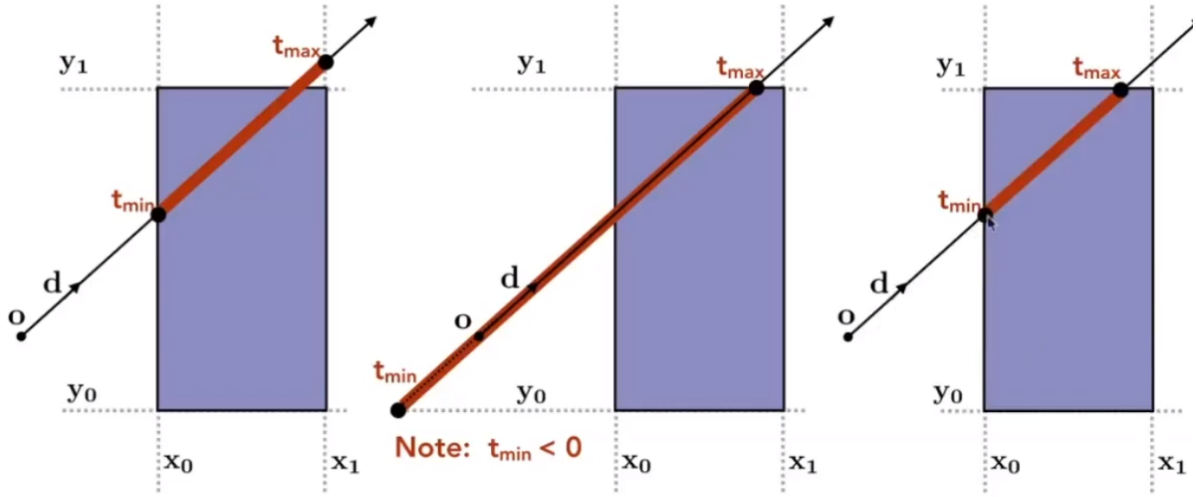


如果一个物体被包围在包围盒中，那么如果光线不会碰到包围盒也就更不会碰到物体本身，换言之，如果一个物体的包围盒都不会碰到，那么就不需要考虑物体，而我在代码中使用的包围盒是AABB(Axis-Aligned Bounding Box)，轴对齐包围盒，它的边界框与坐标轴对齐，易于计算。

而我在代码针对可能出现的不同物体进行不同的包围盒建立，圆的话就根据圆心和半径建立一个包住圆的正方体，三角形也根据不同的定点求出来x，y，z的最大值最小值建立轴对齐包围盒，平面也同理。

```
AABB(Sphere* sp) {
    this->sphere = sp;
    type = Type::SPHERE;
    //根据球创建Box
    Vec3 nodesp = sp->position;
    nodesp -= sp->radius;//中心减去半径
    min = nodesp;
    nodesp = sp->position;
    nodesp += sp->radius;
    max = nodesp;
}
AABB(Triangle* tr) {
    this->triangle = tr;
    type = Type::TRIANGLE;
    float x_min = std::min(std::min(tr->v1.x, tr->v2.x), tr->v3.x);
    float y_min = std::min(std::min(tr->v1.y, tr->v2.y), tr->v3.y);
    float z_min = std::min(std::min(tr->v1.z, tr->v2.z), tr->v3.z);
    float x_max = std::max(std::max(tr->v1.x, tr->v2.x), tr->v3.x);
    float y_max = std::max(std::max(tr->v1.y, tr->v2.y), tr->v3.y);
    float z_max = std::max(std::max(tr->v1.z, tr->v2.z), tr->v3.z);
    min = Vec3(x_min, y_min, z_min);
    max = Vec3(x_max, y_max, z_max);
}
AABB(Plane* pl) {
    this->plane = pl;
    type = Type::PLANE;

    Vec3 pos1 = pl->position;
    Vec3 u = pl->u;
    Vec3 v = pl->v;
    Vec3 pos2 = pos1 + u;
    Vec3 pos3 = pos1 + v;
    Vec3 pos4 = pos1 + v + u;
    Vec3 normal = pl->normal;//法向量

    pos1 -= normal * 0.01f;
    pos2 -= normal * 0.01f;
    pos3 += normal * 0.01f;
    pos4 += normal * 0.01f;
    float x_min = std::min(pos1.x, std::min(std::min(pos2.x, pos3.x),
pos4.x));
    float y_min = std::min(pos1.y, std::min(std::min(pos2.y, pos3.y),
pos4.y));
    float z_min = std::min(pos1.z, std::min(std::min(pos2.z, pos3.z),
pos4.z));
    float x_max = std::max(pos1.x, std::max(std::max(pos2.x, pos3.x),
pos4.x));
```

```
    float y_max = std::max(pos1.y, std::max(std::max(pos2.y, pos3.y),
pos4.y));
    float z_max = std::max(pos1.z, std::max(std::max(pos2.z, pos3.z),
pos4.z));

    min = Vec3(x_min, y_min, z_min);
    max = Vec3(x_max, y_max, z_max);

}
```



上图是如何判断光线是否会进入包围盒，一道光线射入任何一个对面算进入包围盒，射出任何一个对面算离开包围盒，所以对于一个立方体，有三个对面，要满足同时三个对面都要射入，而且三个对面都没有射出，才算光线穿过包围盒中，所以光线进入包围盒的时间是射入三个对面的最大值，这样就进入了三个对面，而射出包围盒的时间是射出三个对面的最小值，这样二者中间的时间就是光线在包围盒的时间，让对面的射入时间都小于射出时间，因为光线是射线而非直线，如果射出时间小于0，说明光线的方向不打向包围盒，那么自然不经过包围盒，如果进入时间小于0而离开时间大于0说明光线在包围内部，这样肯定会经过包围盒，如果最后离开时间大于进入时间说明光线会经过包围盒。

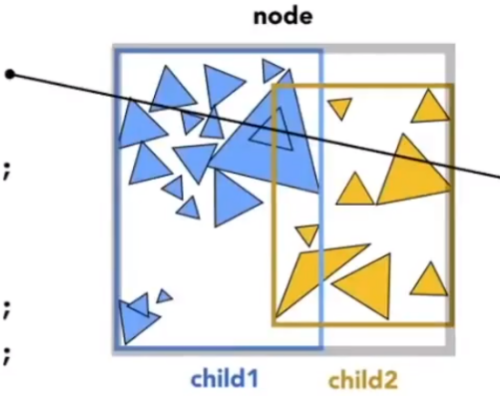
在代码中实现方法就是这样，在代码中的时间之比用光的方向向量反比来做比，因为方向向量的比就类比于速度之比，就是时间反比。

```
inline bool AABB::Intersect(const Ray& ray) {
    Vec3 pos = ray.origin;
    Vec3 dir = ray.direction;
    dir = 1.0f / dir;
    Vec3 dis_max = max;
    dis_max -= pos;
    Vec3 dis_min = min;
    dis_min -= pos;
    float t_xmin = dis_min.x * dir.x;//时间比
    float t_ymin = dis_min.y * dir.y;//时间比
    float t_zmin = dis_min.z * dir.z;//时间比
    float t_xmax = dis_max.x * dir.x;//时间比
    float t_ymax = dis_max.y * dir.y;//时间比
    float t_zmax = dis_max.z * dir.z;//时间比
    if (t_xmin > t_xmax)std::swap(t_xmin, t_xmax);
    if (t_ymin > t_ymax)std::swap(t_ymin, t_ymax);
    if (t_zmin > t_zmax)std::swap(t_zmin, t_zmax);
    float t_enter = std::max(t_xmin, std::max(t_ymin, t_zmin));
    float t_exit = std::min(t_xmax, std::min(t_ymax, t_zmax));
```

```
    if (t_exit < 0) {//说明射线不打向盒子
        return false;
    }
    else if (t_exit >= 0 && t_enter < 0) {//说明在内部
        return true;
    }
    else if (t_exit > t_enter) {
        return true;
    }
    else {
        return false;
    }

}
```

而我实现BVH的划分方式就是将所有的物体一直分成两块，直到最后化成最小的包围盒，所以每个叶节点存的都是物体，中间节点是多个物体合成的包围盒，这样判断光线与物体有无交点可以直接从根节点开始，即所有物体，如果和所有物体的包围盒都没交点，那就不能和物体有交点，然后再拆分成两堆物体，对于我的拆分方式是，找到x，y，z包围盒跨度最大的一组，然后找到对应轴坐标中间的物体进行划分，再继续判断和他们的包围盒有无交点，依次往下判断，最后找到最小的包围盒，在和里面的物体进行判断，这样大大减少了和物体判断有无交点的次数，提升了效率。这就实现了BVH结构。

```
Intersect(Ray ray, BVH node) {
  if (ray misses node.bbox) return;

  if (node is a leaf node)
     test intersection with all objs;
     return closest intersection;

  hit1 = Intersect(ray, node.child1);
  hit2 = Intersect(ray, node.child2);

  return the closer of hit1, hit2;
}
```




```
inline BVHNode* buildBVH(vector<AABB> bounds) {//建树
    BVHNode* node = new BVHNode;
    node->bounds = bounds;
    if (bounds.size() == 1) {
        node->box = bounds[0];
        node->left = nullptr;
        node->right = nullptr;
        return node;
    }
    if (bounds.size() == 2) {
        node->box = Merge(bounds[1], bounds[0]);
        node->left = buildBVH({ bounds[0] });
        node->right = buildBVH({ bounds[1] });
        return node;
    }
    if (bounds.size() > 2) {
        auto max_x = bounds.begin()->centroid().x;
        auto min_x = bounds.begin()->centroid().x;
```

```
        auto max_y = bounds.begin()->centroid().y;
        auto min_y = bounds.begin()->centroid().y;
        auto max_z = bounds.begin()->centroid().z;
        auto min_z = bounds.begin()->centroid().z;
        /* auto max_x = std::max(bounds.begin(), bounds.end(), AABBcomparex());
        auto min_x = std::min(bounds.begin(), bounds.end(), AABBcomparex());
        auto max_y = std::max(bounds.begin(), bounds.end(), AABBcomparey());
        auto min_y = std::min(bounds.begin(), bounds.end(), AABBcomparey());
        auto max_z = std::max(bounds.begin(), bounds.end(), AABBcomparez());
        auto min_z = std::min(bounds.begin(), bounds.end(), AABBcomparez());*/
        for (auto it = bounds.begin(); it != bounds.end(); ++it) {
            max_x = std::max(max_x, it->centroid().x);
        }
        for (auto it = bounds.begin(); it != bounds.end(); ++it) {
            min_x = std::min(min_x, it->centroid().x);
        }
        for (auto it = bounds.begin(); it != bounds.end(); ++it) {
            max_y = std::max(max_y, it->centroid().y);
        }
        for (auto it = bounds.begin(); it != bounds.end(); ++it) {
            min_y = std::min(min_y, it->centroid().y);
        }
        for (auto it = bounds.begin(); it != bounds.end(); ++it) {
            max_z = std::max(max_z, it->centroid().z);
        }
        for (auto it = bounds.begin(); it != bounds.end(); ++it) {
            min_z = std::min(min_z, it->centroid().z);
        }
        float stride_x = max_x - min_x;
        float stride_y = max_y - min_y;
        float stride_z = max_z - min_z;

        if (stride_x == std::max({ stride_x ,stride_y ,stride_z })) {
            std::sort(bounds.begin(), bounds.end(), []
            (const AABB& b1, const AABB& b2) {
                    return b1.centroid().x < b2.centroid().x;
                }
            );
        }
        else if (stride_y == std::max({ stride_x ,stride_y ,stride_z })) {
            std::sort(bounds.begin(), bounds.end(), []
            (const AABB& b1, const AABB& b2) {
                    return b1.centroid().y < b2.centroid().y;
                }
            );
        }
        else if (stride_z == std::max({ stride_x ,stride_y, stride_z })) {
            std::sort(bounds.begin(), bounds.end(), []
            (const AABB& b1, const AABB& b2) {
                    return b1.centroid().z < b2.centroid().z;
```

```
                }
            );
        }

        vector<AABB> boundsleft(bounds.begin(), bounds.begin() +
(bounds.size() / 2) + 1);
        vector<AABB> boundsright(bounds.begin() + (bounds.size() / 2) + 1,
bounds.end());
        node->left = buildBVH(boundsleft);
        node->right = buildBVH(boundsright);
        node->box = Merge(node->left->box, node->right->box);
        return node;
    }
    if (stride_x == std::max({ stride_x ,stride_y ,stride_z })) {
        std::sort(bounds.begin(), bounds.end(), []
        (const AABB& b1, const AABB& b2) {
                return b1.centroid().x < b2.centroid().x;
            }
        );
    }
    else if (stride_y == std::max({ stride_x ,stride_y ,stride_z })) {
        std::sort(bounds.begin(), bounds.end(), []
        (const AABB& b1, const AABB& b2) {
                return b1.centroid().y < b2.centroid().y;
            }
        );
    }
    else if (stride_z == std::max({ stride_x ,stride_y, stride_z })) {
        std::sort(bounds.begin(), bounds.end(), []
        (const AABB& b1, const AABB& b2) {
                return b1.centroid().z < b2.centroid().z;
            }
        );
    }

    vector<AABB> boundsleft(bounds.begin(), bounds.begin() + (bounds.size()
/ 2) + 1);
    vector<AABB> boundsright(bounds.begin() + (bounds.size() / 2) + 1,
bounds.end());
    node->left = buildBVH(boundsleft);
    node->right = buildBVH(boundsright);
    node->box = Merge(node->left->box, node->right->box);
    return node;
}
```

代码实现的方式就是一直拆分成子结点，直到叶节点是物体，然后中间划分两堆物体的方式就是先找到最大值和最小值就是O(n)的复杂度，进行比较，然后找到跨度最大的坐标轴，O(nlogn)排序找到中间的结点，然后分成子节点继续分。最后依靠上文的伪代码实现找到光线与物体的最近交点。因为所给例图物体太过简单，实现的加速效果不明显，于是是在网上所找图片进行渲染进行性能分析。在普通path racing所花80min的情况下，BVH加速的path racing仅花22min左右，加速足足接近4倍。其他我还将线程数扩大，也实现了速度的提升。

5 迷中遇到的一些小bug

在做bvh的时候，我发现我的图片渲染出来每次都少一般，长方体变成少几个面，我百思不得其解，检查好多遍代码终究找不到错误，最后发现出现在一个很有意思的地方，迭代器的使用，

```
vector<AABB> boundsleft(bounds.begin(), bounds.begin() + (bounds.size() / 2) + 1);
vector<AABB> boundsright(bounds.begin() + (bounds.size() / 2) + 1, bounds.end());
```

这里是在建立bvh树的过程中，要把一堆物体拆分成两堆，此时已经经过排序，我一开始left写的是bounds.begin() + (bounds.size() / 2)，后来才发现，这样的话最后一个物体是不计入的，迭代器的end是指迭代器所指向的最后一个元素的下一个位置。所以最后找了半天bug才把这个改成下一个元素，才成功渲染。

部分菲涅尔方程的资料借助了AI生成理解。

BVH加速的截图和公式来源闫令琪老师games101课程 Ray Tracing 1 理解也是观看网课之后所进行的理解
https://www.bilibili.com/video/BV1X7411F744?p=13