Lab-1


# 图形绘制技术

Introduction and Prerequisite

**Lab1-物体求交以及加速结构**

2023 春季学期

# 目录

**1 作业提交** **3**
1.1 分数占比 . . . 3
1.2 你需要编写的部分 . . . 3
1.3 提交方式 . . . 3

**2 光线-物体求交接口** **3**
2.1 为什么需要求交 . . . 3
2.2 moer-lite 中的物体求交接口 . . . 3

**3 圆环与光线的求交** **4**
3.1 moer-lite 中的圆环 . . . 4
3.2 求交 . . . 5
3.3 填充求交信息 . . . 6
3.4 验证你的求交结果 . . . 6

**4 圆柱与光线的求交** **6**
4.1 moer-lite 中的圆柱 . . . 6
4.2 求交 . . . 7
4.3 填充求交信息 . . . 8
4.4 验证你的求交结果 . . . 8

**5 圆锥与光线的求交** **8**
5.1 moer-lite 中的圆锥 . . . 8
5.2 求交 . . . 9
5.3 填充求交信息 . . . 10
5.4 验证你的求交结果 . . . 11

**6 加速结构介绍** **11**
6.1 加速结构简介 . . . 11
6.2 BoundingBox . . . 12
6.3 加速结构接口 . . . 13

**7 Octree（选做）** **14**
7.1 Octree 介绍 . . . 14
7.2 Octree 节点结构 . . . 15
7.3 Octree 构建 . . . 15
7.4 Octree 求交 . . . 15
7.5 验证 . . . 16

**8 BVH（选做）** **16**
8.1 BVH 介绍 . . . 16

8.2 BVH 节点结构 . . . . . 16
8.3 BVH 构建 . . . . . 17
8.4 BVH 求交 . . . . . 18
8.5 验证 . . . . . 18

# Abstract

该文档是南京大学计算机系 2023 春季学期《图形绘制技术》课程实验 1 的实验手册。

Lab1 的内容主要是完成三种物体（圆环，圆柱，圆锥）的求交判断和交点信息填充。学有余力的同学可以完成加速结构。我们提供了两种加速结构的介绍，完成一种即可获得 bonous。

各位同学如果在 lab1 实验过程中有任何疑问或想法，包括但不限于

- 实验框架的出现的 Bug
- 手册中叙述不完善的部分
- 对实验安排的建议
- 更好的框架设计

主讲老师：过洁 Email：guojie@nju.edu.cn

欢迎联系助教，帮助我们完善课程的实验部分。TAs：

- 陈振宇 QQ：895761580 Email：chenzy@smail.nju.edu.cn
- 袁军平 QQ：1924188282 Email：191250189@smail.nju.edu.cn
- 周辰熙 QQ：1305845549 Email：191250210@smail.nju.edu.cn
- 王宸 QQ：1401520906 Email：chenwang@smail.nju.edu.cn

**同时，课程的框架代码以及实验文档也会不断更新改进，请同学们关注助教发布的相关信息。**

# 1. 作业提交

## 1.1 分数占比

圆环求交:30%,圆柱求交:30%, 圆锥求交:40%. 完成任意一种加速结构,bonous20%

## 1.2 你需要编写的部分

- Disk.cpp、Cylinder.cpp、Cone.cpp 文件下的 todo 方法
- 如果你要完成加速结构: bvh.cpp 或 octree.cpp 文件下的 todo 方法。

## 1.3 提交方式

在教学立方提交以下内容的压缩包:disk.cpp、cylinder.cpp、cone.cpp、加速结构的头文件和源文件（如果你完成了加速结构)。以及 **pdf** 格式实验报告，报告包括以下内容:

- 对于物体求交部分，给出简单的求交逻辑介绍和实验结果图片
- 对于选做部分，给出加速结构实现思路以及求交时间和线性求交的对比
- 对实验的建议和吐槽

# 2. 光线-物体求交接口

## 2.1 为什么需要求交

渲染器需要模拟光线在场景中的传播过程，这需要计算光线和物体的求交。比如我们生成了一条摄像机光线，我们需要知道这条光线是否击中了物体，如果击中物体，击中的第一个物体 是什么物体。如果要对物体表面进行正确的着色，我们还需要击中点的其他信息，比如法线，纹理坐标等等。

## 2.2 moer-lite 中的物体求交接口

在 moer-lite 中，物体和光线的求交接口主要有两个。

第一个函数是判断给定光线，光线是否和物体相交。如果相交，会填充 primID(图元 Id),uv 坐标。 primID 在我们本次作业中都填充 0 即可。

第二个函数是通过 primid，uv 坐标等来填充相交信息。因为光线可能会先与较远的物体计算相交，所以我们不将这两个函数合二为一来避免没有意义的填充调用

在这次作业中，我们只需要填充相交点的法线，相交点的位置即可。

```
virtual bool rayIntersectShape(Ray &ray, int *primID, float *u, float *v) const = 0;
virtual void fillIntersection(float distance, int primID, float u, float v,Intersection *
    intersection) const = 0;

```

在给定光线后，场景中的加速结构承担光线和场景的求交, 逻辑如下:

```
    std::optional<Intersection> rayIntersect(Ray &ray) const {
    int primID, geomID = -1;
    float u, v;
    bool hit = rayIntersect(ray, &geomID, &primID, &u, &v);
    if (!hit)
      return std::nullopt;
    Intersection its;
    shapes[geomID]->fillIntersection(ray.tFar, primID, u, v, &its);
    return std::make_optional(its);
}
```

其中 rayIntersect 是加速结构需要实现的方法，暂时可以认为是遍历每个物体，调用物体的 rayIntersectShape 方法。场景与光线的求交逻辑便显而易见：如果没有物体相交就返回 std::nullopt 代表没有相交物体。相交的话得到最近相交物体的 id。根据 id 找到对应物体，根据 primId 和 uv 坐标等来填充相交信息结构体。下面让我们来完成三种新增物体的求交逻辑实现吧!

# 3. 圆环与光线的求交

## 3.1 moer-lite 中的圆环

在三维空间, 圆可以用圆心, 法线, 半径来定义。如果要计算光线和圆的交点, 可以通过计算光线与平面的交点并计算交点与圆心的距离来完成。但是三维空间内的旋转等是件很麻烦的事情。我们不妨将圆固定在原点，并且让它始终垂直于 z 轴。它的旋转平移等变换通过变换矩阵来完成。在 transform 类中，我们提供了 inverseRay 方法。该方法可以将光线变换到局部空间。已经实现的物体求交中,cube 便是通过这种方法完成求交的. 本次实验的圆柱，圆锥等也是通过先将光线变换到物体局部空间进行求交。

在有了以上信息，要确定一个圆其实就只需要一个半径了。为了能表示圆环，我们加入了 innerRadius 表示内半径。为了能够渲染残缺圆环，我们加入 phiMax 来表示圆环的最大角度（弧度制）

```
class Disk{
  float radius;
  float phiMax;
  float innerRadius;
```

```
}
```

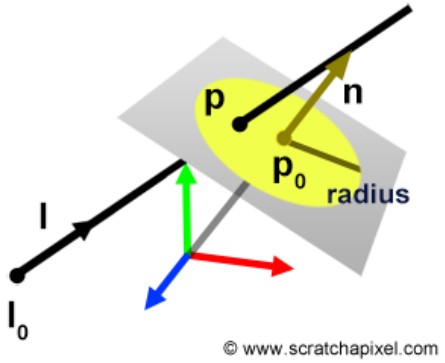



图 1: 光线和圆相交

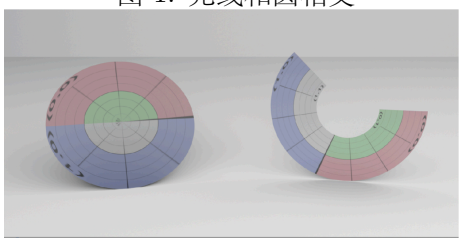


图 2: 两个圆环完整的和残缺的

### 3.2 求交

注意我们下文提到的 ray 都是转换到局部空间的 ray。在局部空间光线与圆环的求交就十分简单了, 可以通过联立方程计算也可以通过几何信息计算，这里使用后者。我们先考虑光线和 z=0 这个平面的相交。首先需要检验光线方向 z 分量是否为 0。若不为 0，计算光线到达该平面时的时间 $t = \frac{(0-ray.origin.z)}{ray.direction.z}$ 我们可以使用 ray.at 函数来得到交点坐标，在得到光线与平面的交点后，我们需要检验以下几点:

- 我们需要考虑交点是在当前光线的范围内，即判断 t 与光线的 tFar 和 tNear 的大小关系
- 交点是否在圆环半径内，我们需要计算交点与圆心的距离。判断该距离和圆环的内外半径的大小关系
- 交点与圆心的角度是否符合要求，需要计算交点和圆心所成夹角，判断夹角和 phiMax 的大小关系

对于交点 p 与圆心的夹角 phi，我们有一个简单的几何关系 $\tan phi = \frac{p.y}{p.x}$。如果能通过以上检验，那么就说明光线和物体我们的圆环是有交点的。我们还需要填充 uv 坐标即纹理坐标。纹理坐标是一个二元组 (u,v)，详细介绍可以看 lab0。这里

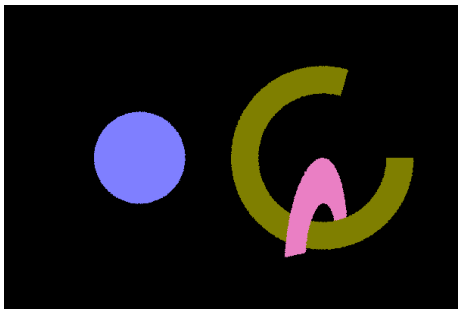
对于圆环我们计算 uv 可以用 u 表示交点在角度上的偏移，v 表示在半径上的偏移。$u = \frac{phi}{phiMax}$ ，$v = \frac{radius - minRadius}{maxRadius - minRadius}$ 这样给定 uv 坐标，我们就可以在圆环上唯一的确定一个点。

### 3.3 填充求交信息

需要填充的有相交点位置和法线，注意使用 transform 类的 toWorld 将信息转换到世界空间。

### 3.4 验证你的求交结果

运行场景 **lab1-test/lab1-test-disk** 如果你的实现没有问题，那么你将得到以下图片结果。



# 4. 圆柱与光线的求交

### 4.1 moer-lite 中的圆柱

同圆环一样，我们一样在局部空间对圆柱进行求交。我们认为在局部空间内，圆柱底面是以原点为圆心，法线为 z 轴，半径为 r 的圆。圆柱高度为 h。因此，圆柱只需要两项来定义，radius 和 height。同圆环一样，我们为了能够渲染残缺圆柱，我们引入了 phiMax 表示圆柱最大角度。注意，moer-lite 中的圆柱只包含**侧面**。

```
class cylinder{
    float height;
    float radius;
    float phiMax;
};
```

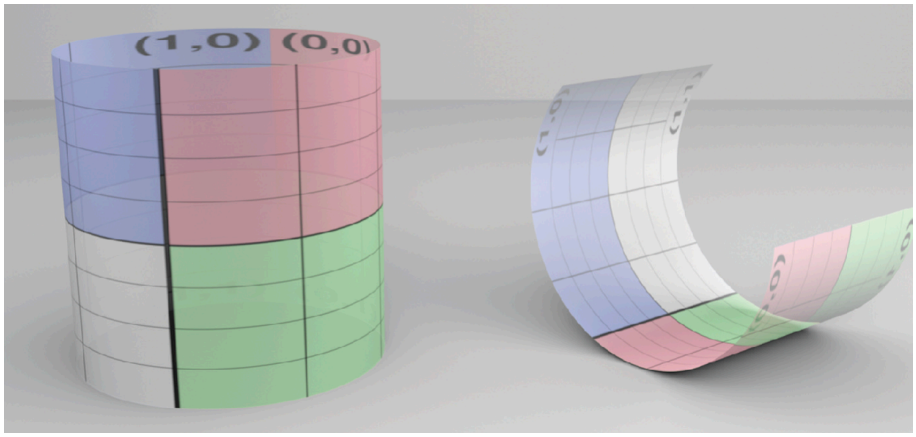



图 3: 两个圆柱完整的和残缺的

## 4.2 求交

在局部空间计算光线与圆柱的相交我们选择使用联立方程组来解决。感兴趣的同学可以考虑从几何上入手。
考虑无限高的圆柱，其方程为：$x^2+y^2=r^2$。而我们的光线方程为 $p=ray.direction*t+ray.origin$。联立二者，我们得到：

$$(ray.origin.x+ray.direction.x*t)^2+(ray.origin.y+ray.direction.y*t)^2=r^2$$

这是一个简单的一元二次方程组，我们在 Function.h 文件里面提供了一个简单的辅助函数求解。

```
11    inline bool Quadratic(float A, float B, float C, float *t0, float *t1);
```

A,B,C 分别为对应系数，$t_0$，$t_1$ 为方程的解。$t_0$<$t_1$ 但注意，我们这里求出的解是默认光线是无限长的直线，圆柱也是无限高的。得到交点后，我们需要检验 t。

- 我们需要考虑交点是在当前光线的范围内，即判断 t 与光线的 tFar 和 tNear 的大小关系
- 交点是否在圆环内，交点的 z 坐标是否在圆柱高度范围内
- 交点与圆柱对应面的圆心的角度是否符合要求，需要计算交点和圆心所成夹角，判断夹角和 phiMax 的大小关系

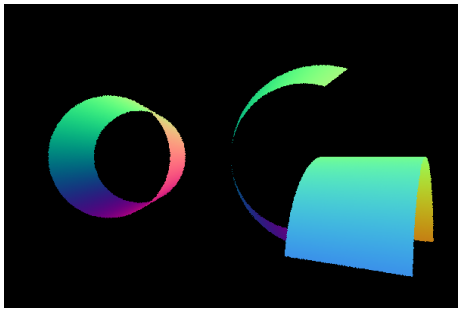
如果 t 可以通过以上检验，那么我们的光线和圆柱就有交点。要注意的一点是如果方程有两个解，并且都符合要求，我们需要选择最近的即 t0。通过 t 得到交点坐标，我们可以通过 $u=\frac{phi}{phiMax}$ $v=\frac{p.z}{hegiht}$ 计算纹理坐标。

### 4.3 填充求交信息

需要填充的有相交点位置和法线，注意使用 transform 类的 toWorld 将信息转换到世界空间。交点处的位置可以根据 uv 坐标计算，交点处的法线沿该点半径方向。$n = (\cos(phi), \sin(phi), 0)$。

### 4.4 验证你的求交结果

运行场景 **lab1-test/lab1-test-cylinder** 如果你的实现没有问题，那么你将得到以下结果。



# 5. 圆锥与光线的求交

### 5.1 moer-lite 中的圆锥

同样的，我们一样在局部空间对圆锥进行求交。我们认为在局部空间内，圆锥底面是以原点为圆心，法线为 z 轴，半径为 r 的圆。圆锥高度为 h。因此，圆锥只需要两项来定义 radius 和 height。同前面两个物体一样，我们为了能够渲染残缺圆锥，我们引入了 phiMax 表示圆锥最大角度。为了计算方便，我们记录圆锥夹角的 cos 值. 注意，moer-lite 中的圆锥只包含**侧面**。

```
class cylinder{
    float height;
    float radius;
    float phiMax;
    float cosTheta;
};
```

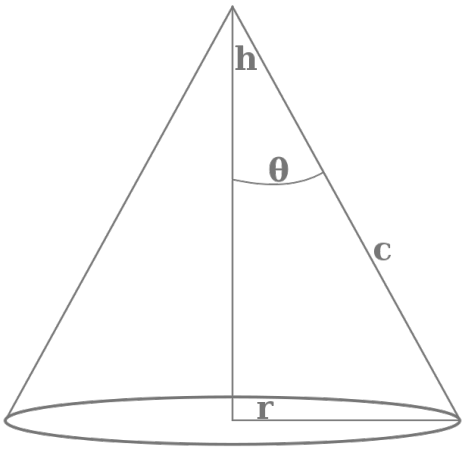



图 4: 圆锥示意图

## 5.2 求交

在局部空间计算光线与圆锥的相交我们选择使用联立方程组来解决。感兴趣的同学可以考考虑从几何上入手。

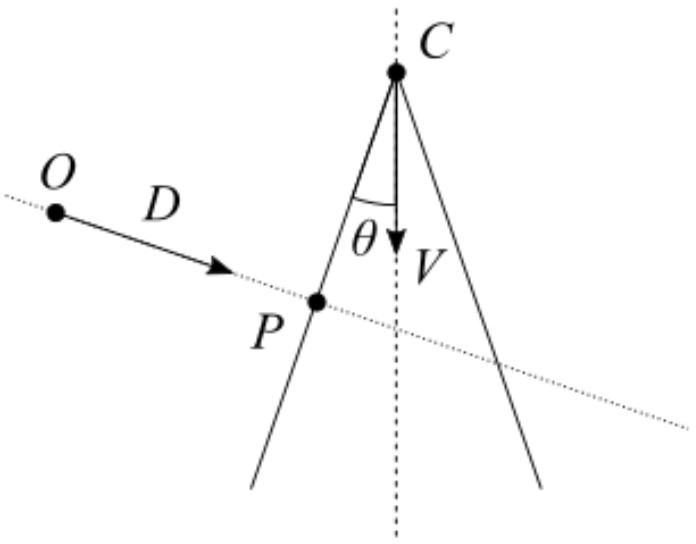



图 5: cone-ray-intersection

圆锥上的任意点 P 都满足 $(P-C)\cdot\vec{V}=|P-C|*\cos theta$ 同时光线的方程为

$P = O + t * \vec{D}$ 联立并带入我们得到方程:

$$t^2((\vec{D}\cdot\vec{V})^2-\cos^2\theta)+2t((\vec{D}\cdot\vec{V})(\vec{CO}\cdot\vec{V})-\vec{D}\cdot\vec{CO}\cos^2\theta)+(\vec{CO}\cdot\vec{V})^2-\vec{CO}\cdot\vec{CO}\cos^2\theta=0$$

同样的，我们通过解这个方程组得到 $t_0$,$t_1$。这个求交方程考虑的圆锥是完整并且高度是无限的，我们需要检验 t。

- 我们需要考虑交点是在当前光线的范围内，即判断 t 与光线的 tFar 和 tNear 的大小关系
- 交点是否在圆锥内，交点的 z 坐标是否在圆锥高度范围内
- 交点与圆锥对应面的圆心的角度是否符合要求，需要计算交点和圆心所成夹角，判断夹角和 phiMax 的大小关系

如果 t 可以通过以上检验，那么我们的光线和圆锥就有交点。同样选择最近的解 通过 t 得到交点坐标，我们可以通过 $u = \frac{phi}{phiMax}$ $v = \frac{p.z}{hegiht}$ 计算纹理坐标。

### 5.3 填充求交信息

需要填充的有相交点位置和法线，注意使用 transform 类的 toWorld 将信息转换到世界空间。

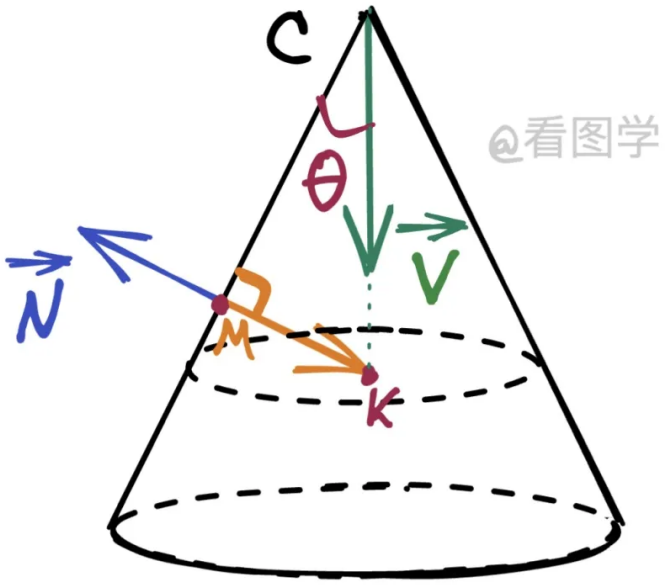
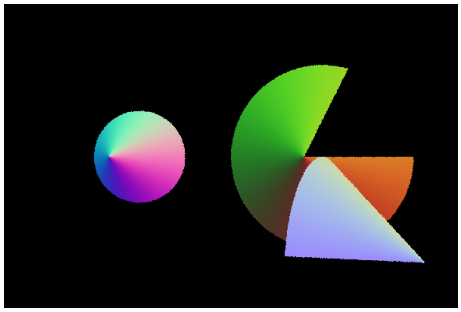



图 6: compoute-cone-normal

交点 M 处的位置可以通过 uv 坐标计算得到。让我们考虑交点 M 处的圆锥法线, 法线方向其实就是 $\vec{MK}$。M 坐标可以通过 uv 坐标计算得到，我们只需要计算 K 的坐标, 通过 $C + C\vec{K}$ 即可得到 K 的坐标进而计算法线。

### 5.4 验证你的求交结果

运行场景 **lab1-test/lab1-test-conel** 如果你的实现没有问题，那么你将得到以下图片结果。



# 6. 加速结构介绍

本部分为选做部分。**即使不完成也不会影响分数。如果你选择做，只需要选一种加速结构完成即可**

### 6.1 加速结构简介

在这次作业中，我们将实现用来加速光线与场景中物体求交的数据结构。在光线追踪渲染器中，无论是寻找着色点或者是判断光线与物体的遮挡关系等等，都需要我们计算光线与物体的相交。一个朴素的想法是，对于一条给定的光线，我们遍历场景中的每个物体, 计算光线是否与物体相交来完成这一任务。这样的想法对于 cornell-box 这样的场景是可行的。但是许多场景中可能有几十个或者更多物体。有的几何物体例如 TriangleMesh 可能由几十万个三角形面片组成, 毛发可能由几十万条曲线组成。暴力遍历的方式的时间成本是难以接受的。我们可以根据物体在三维空间内的几何分布和物体的层次结构来加速这一过程。

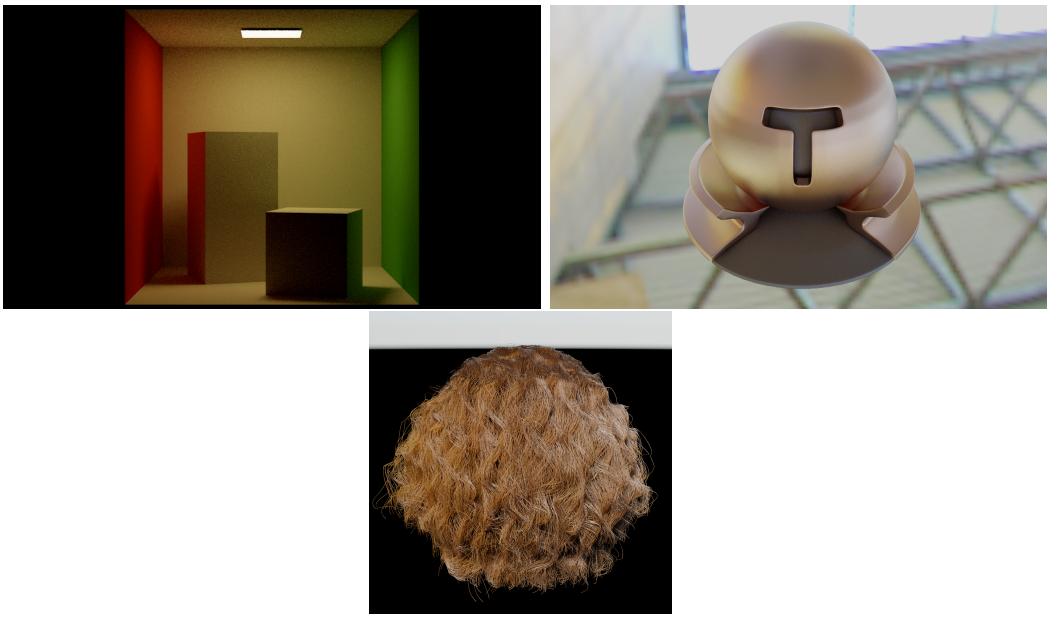


图 7: cornell-box test-ball curly-hair

## 6.2 BoundingBox

要加速求交，显然需要从物体的几何结构入手。然而不同几何结构性质不同，很难统一考虑。我们为物体引入包围盒这一概念。光线与包围盒的求交是非常容易计算的。容易想到，如果光线与包围盒不相交，那么光线与物体一定不相交。我们可以利用这一点进行加速。在 moer-lite 中，BoudingBox 这个类成员很简单，只需要记录 box 的最小点和最大点就可以。

```
class AABB{
    Point3f pMin, pMax;
    bool RayIntersect(const Ray &ray, float *tMin = nullptr,
                float *tMax = nullptr) const;
    ...
}

```

这里挑出 BoundingBox 的一个方法进行说明 (RayIntersect)。要计算光线是否与包围盒相交，我们遍历 xyz 三个轴对应的两个面，计算光线在对应两个面进出包围盒的时间，更新光线与包围盒相交 最小相交时间与最大相交时间。光线与包围盒没有相交当且仅当最小相交时间大于最大相交时间。

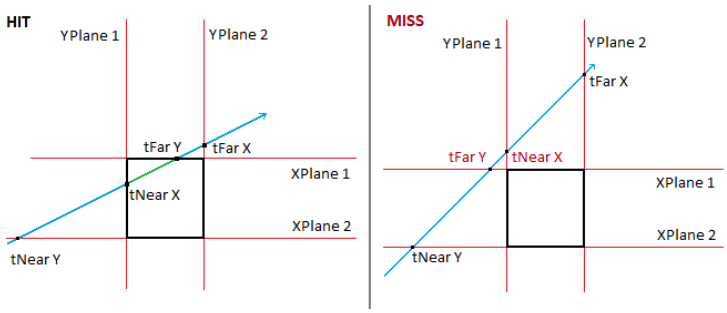



图 8: ray intersect with AABB

BoudingBox 类的余下成员方法可以在 BoudingBox 类文件里面查看。

## 6.3 加速结构接口

在 moer-lite 中，加速结构主要接口如下

```
    //* 所有空间加速结构需要实现的接口
class Acceleration {
    virtual bool rayIntersect(Ray &ray, int *geomID, int *primID, float *u,
    float *v) const = 0;
    virtual void build() = 0;
    std::optional<Intersection> rayIntersect(Ray &ray) const {
        int primID, geomID = -1;
        float u, v;
        bool hit = rayIntersect(ray, &geomID, &primID, &u, &v);
        if (!hit)
        return std::nullopt;
        Intersection its;
        shapes[geomID]->fillIntersection(ray.tFar, primID, u, v, &its);
        return std::make_optional(its);
  }
  ...
 }

```

我们首先来思考一个问题，除了判断光线是否与物体相交我们还需要记录哪些信息。我们需要记录着色点的法线，纹理信息等。这些做法是不需要在求交时做的，因为无法判断当前求交的物体是不是最近的那个物体。所以在求交过程中，我们只需要记录最终求交物体在场景中的 shapeId 和 primId。二者分别是相交物体在场景中的唯一标识 和具体和物体的哪个图元（一般为 0，除非像 triangleMesh 这样的多图元几何结构）相交。

有了这样的思路，理解加速结构的接口就很简单了。在非虚函数 rayIntersect 方法中，我们调用每个加速结构自己实现的求交方法，记录 shapeId 和 primId。然后找到对应的物体，只要我们 给它提供 primId 和 uv 坐标, 那么它就一定能帮我们填充好 Intersection 信息。

Build 函数则是每个加速结构需要在场景构建的过程中调用的方法。每种加速结

构构建的方法不一样，所以设置为虚函数，也是我们这次作业中需要实现的。

在 moer-lite 中，主场景拥有加速结构，每个 mesh 都会拥有自己的加速结构。同学们不妨想一想，这样的设计比将 mesh 分成独立的小三角形，然后只用一个主场景的一个加速结构 完成场景求交有什么优势。

# 7. Octree（选做）

## 7.1 Octree 介绍

本节进入我们本次作业的第一个加速结构-Octree（八叉树）介绍。这种加速结构通过对空间进行划分来完成求交加速。八叉树它将三维空间分割成 8 个同等大小的子空间，为什么是分成八份呢，因为我们空间是三维的，每一个轴分成两份。如果我们在二维空间构建，我们就只需要使用四叉树。我们递归地将每个子空间分割成更小的子空间, 每个子空间包含和它有相交的物体 id。直到每个子空间中包含的物体数小于某个阈值或到达了最大深度，在求交时，如果光线和当前节点对应的空间没有相交的话，我们就可以直接跳过当前节点，这样的剪枝大大提高了光线求交的效率。如果光线和当前节点有相交的话，就可以递归遍历当前节点 的子节点完成求交。

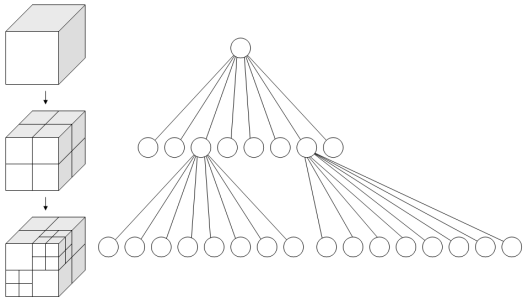


图 9: octree

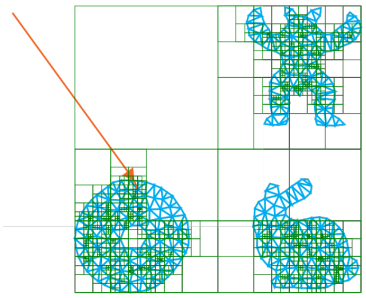


图 10: octree

## 7.2 Octree 节点结构

有了这样的求交思路，我们就可以设计出 OctNode 的结构。对于非叶子结点，我们需要存储 8 个子节点的指针用来递归遍历，以及节点的包围盒用来判断光线有没有可能和当前节点下的物体相交。对于叶子结点，我们需要记录叶子结点的存储的物体 Id 这里给出一个参考。

```
struct Octree::OctreeNode {
      BoundingBox box;
      std::unique_ptr < std::vector <int > > indexBuffer;
      OctreeNode * subNodes[nSub];
      ...
  }

```

## 7.3 Octree 构建

在有了结点结构后，我们需要考虑如何构建八叉树。根据我们上文提到的节点结构，构建时我们需要节点的包围盒以及当前节点需要处理的索引数组。如果递归深度到达最大值或者当前节点数小于一定值时，当前节点作为叶子节点。否则我们将当前节点八等分。用八个子节点表示. 子节点的包围盒通过空间划分得到。子节点的索引数组通过寻找和子节点包围盒有重叠区域的物体得到。这里给出伪代码。

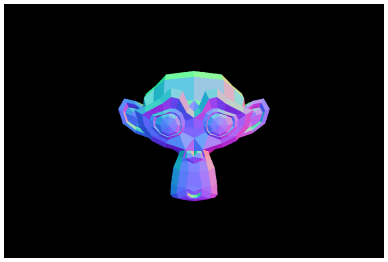
```
    Node recuriseveBuild(box,indexBuffers,depth):
      if(indexBuffers.size()<MaxLeafSize or depth > maxDepth)
          reutrn node(indexBuffers)
      else :
         Node node(box)
         subBoxes = getSubBoxes(box)
         vector<vector<int>> subBuffers(8)
         for(i = 0; i<8 ;i++)
         for(index : indexBuffers)
              if(shapes[index].getAABB().overlaps(subBoxes[i]))
                  subBuffers[i].push(index)
          for(int i = 0;i<8;i++)
              node.subNodes[i] = recuriseveBuild(subBoxes[i],subBuffers[i])
          return node

```

## 7.4 Octree 求交

八叉树求交逻辑在八叉树介绍时我们就已经提到过了。主要是一个递归过程，通过节点的包围盒对与光线是否相交进行剪枝来进行加速。对于非叶子节点，首先判断当前空间是否和光线相交，如果不相交可以直接返回, 否则遍历每个子节点。遍历到叶子节点时我们遍历叶子节点下的所有物体进行求交即可。

### 7.5 验证

运行场景 **lab1-test/lab1-test-octree** 如果你的 octree 实现没有问题，那么你将在较短时间内（不超过 5s）完成渲染得到 monkey.obj 的法线图



# 8. BVH（选做）

### 8.1 BVH 介绍

本节进入我们本次作业的二个加速结构-BVH (Bounding Volume Hierarchies) 介绍。与 Octree 对空间进行划分不同，BVH 通过对物体进行划分来完成加速。BVH 的核心思想就是用体积略大而几何特征简单的包围盒来近似描述复杂的几何对象，并且这种包围盒是嵌套的，我们只需要对包围盒进行进一步的相交测试，就可以越来越逼近实际对象。在求交时，如果光线和当前节点对应的包围盒没有相交的话，我们就可以直接跳过当前节点，这样的剪枝提高了光线求交的效率。如果光线和当前节点有相交的话，就可以递归遍历当前节点 的左右子节点完成求交。

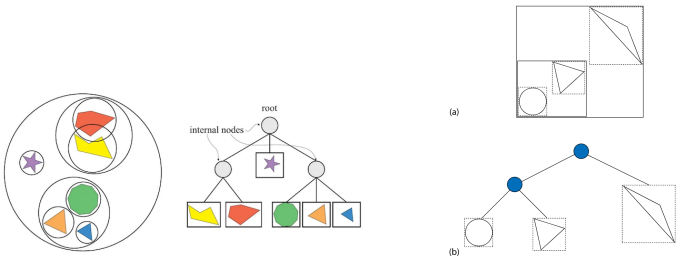



图 11: bvh

### 8.2 BVH 节点结构

同样的，bvh 也是树形结构。分为叶子结点和非叶子结点，叶子结点存储物体索引，非叶子结点存储节点包围盒以及左右节点索引。（该结构仅为讲解方便作示例，实际实现不必拘泥）

```
1     struct BVH::BVHNode {
2       BVHNode * left;
3       BVHNode * right;
```

```
        BoundingBox box;
        int firstShapeOffset;
        int nShape = 0;
        int splitAxis;
    }

```

其中 firstShapeOffset 和 nShape 是为叶子结点存结点索引使用（代表第一个物体的索引和叶子结点存了多少物体）。 splitAxis 代表非叶子结点的分割轴（在后续求交部分会用到）

### 8.3 BVH 构建

在有了结点结构后，我们需要考虑如何构建 BVH。对于非叶子节点在最开始，我们需要处理整个场景的物体索引。现在我们要想一种分割方式，将这一组物体分成两组，再交给左右子节点分别构建。这种分割方式要保证我们在相交时能够尽量少的进行相交测试。分割方式很大程度上影响 BVH 的性能。一种常见的方式是按中点分割，将所有物体的中心在某个坐标轴上排列，按照中心在该坐标轴的大小均匀分成两部分。

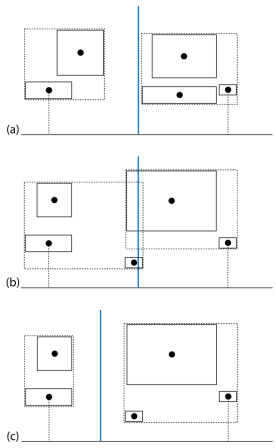



图 12: 中点分割

这种方式简单，但分割效果不一定好。如上图所示。在（a）中所示的情况，根据质心沿所选轴线的中点来进行分割是有效的。由此产生的两个形状组的包围盒用虚线表示。然而，对于像（b）中所示的这种分布，中点选择是欠优的，两个中间包围盒的相互重叠会导致性能下降。因此，（c）中选择了一个不同的分割平面（绿色），导致得到的包围盒更小且完全不重叠，从而在渲染时获得更好的性能.

感兴趣的同学可以考虑实现 SAH，并且比较下 SAH 和中点分割的性能差异。SAH 会考虑不同分割方式的求交成本。 SAH 分割参考

这里我们只实现中点分割就可以。按照物体中心分割，物体中心可以认为是物体

的包围盒中心。在 bvh 构建函数接受的索引数组小于我们设定的叶子结点最大物体数时 我们就可以结束递归，返回一个叶子结点。

```
BVHNode * BVH::recursiveBuild(std::vector < BVHShapeInfo > & shapeInfos, int l, int r,
                              std::vector < std::shared_ptr < Shape>> & orderedShapes) {
if ( r-l <= LeafMaxSize ) {
      node->firstShapeOffset = orderedShapes.size();
      node->nShape = nShapes;
      for ( int i = l ; i < r ; ++ i ) {
          orderedShaps.push_back(shapes[shapeInfos[i].id]);
      }
      return node;
  }
  \... handle no-left case
  }
```

叶子结点需要存储节点内所有物体的索引，我们可以像八叉树一样存一个索引数组。但对于 bvh 来说，在分割构建时我们会对物体进行排序, 此时叶子结点接受的序列列表是有序的，我们存储第一个物体的索引和物体个数就可以。每次初始化叶子结点，我们都会向 orderedShapes 里存储排好的物体。在 bvh 构建完成后，我们会将 shapes 重置为 orderedShapes，以保证叶子结点中存储的索引和 bvh 中物体列表可以对应.

## 8.4 BVH 求交

BVH 求交逻辑非常简单。给定光线，我们首先判断节点是否为叶子节点，如果是叶子节点，我们依次遍历节点物体即可。对于非叶子节点，我们之前就递归 调用左右子节点的求交函数即可。当然求交也可以不使用递归，用一个数组存储待处理的节点迭代处理是更好的选择。这里我们要注意的一点是，我们是先调用左子节点还是右子节点呢。不同的调用顺序显然会影响求交效率。例如光线因为和左子节点相交的而更新了 tFar 而不会与右子节点的包围盒相交. 这种情况下先调用左子节点的求交就可以减少求交计算次数。还记得我们的 splitAxis 这个属性吗。我们在某一个轴上将物体坐标从小到大排列。我们可以根据光线在这个轴的方向上选择调用顺序。如果光线在这个方向上的分量大于 0，我们就先调用左子节点。

## 8.5 验证

运行场景 **lab1-test/lab1-test-bvh** 如果你的 bvh 实现没有问题，那么你将在较短时间内（不超过 5s）完成渲染 得到 monkey.obj 的法线图