Lab-1

图形绘制技术

Introduction and Prerequisite

Lab1-物体求交以及加速结构

2025 春季学期

Abstract

该文档是南京大学计算机系 2025 春季学期《图形绘制技术》课程实验 1 的实验手册。

Lab1 的内容主要是完成常用物体（三角面片和轴对齐包围盒）的求交判断和交点信息填充。同时实现一个被广泛使用的 BVH（层次包围盒）加速结构。我们提供了 BVH 的介绍，完成最基础的 BVH 划分形式即可，如果你实现了更多样的 BVH 划分方式和组织形式，可以获得额外加分。

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：

- 杨铭 QQ：925882085 Email：925882085@qq.com
- 孙奇 QQ：745356675 Email：745356675@qq.com

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。

目录

1. 作业提交

1.1 分数占比

- 三角面片求交: 15%
- 轴对齐包围盒求交: 15%

- BVH 的构建: 35%
- BVH 的求交: 35%
- 完成任意一种 BVH 的优化机制, 额外加分: 20%

1.2 你需要编写的部分

- **你需要使用 moer-lite 的 2025 分支完成本次作业!**
- Triangle.cpp、AABB.cpp、BVH.cpp 文件下的 TODO 方法
- 如果你要实现 BVH 的优化机制，主要修改 BVH.cpp 文件下的 TODO 方法。

1.3 提交方式

在教学立方提交 **pdf** 格式实验报告，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
- 对于加速结构部分，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。

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，我们可能需要填充相交点的位置，法线，uv 坐标等信息。

```
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。下面让我们来完成常见物体的求交逻辑实现吧！

3. 三角面片与光线的求交

3.1 moer-lite 中的三角面片

在三维空间，三角面片一般使用顶点坐标来定义。

```
class Triangle {
  int primID;
  int vtx0Idx, vtx1Idx, vtx2Idx;
  const TriangleMesh *mesh = nullptr;
}
```

其中 mesh 表示该三角面片隶属的网格对象，对应的 vtxIdx 对应顶点数据的索引。

```
class Ray {
  Point3f origin;
  Vector3f direction;
  float tFar, tNear;
}
```

而光线一般由原点和方向来定义，其中 t 值用于表示光线沿对应方向行进的距离。因此，我们的目标就是找到一个**合法的 t 值**，判断射线在对应 t 值的位置是否位于三角形内。

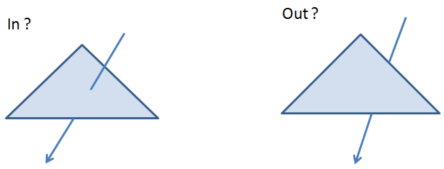

图 1: 光线和三角面片相交

3.2 求交

你需要实现 Triangle 类的 rayIntersect 方法。

一个简单而直观的方法：首先判断射线是否与三角形所在的平面相交，如果相交，再判断交点是否在三角形内，大致步骤如下：

- 联立射线与三角形所在平面的方程，可以求得光线到达三角形所在平面的 t 值。
- 根据 t 值可以求得射线与三角形所在平面的交点。
- 判断交点是否在三角形内，如果相交，则更新光线的 tFar 信息，防止其与其他物体求交。

同学们可以想一想还有什么其他的求交方式。

3.3 填充求交信息

你需要实现 TriangleMesh 类的 fillIntersection 方法。

本次作业中，三角形图元隶属于一个网格对象 TriangleMesh，像法线等信息往往存储在这个对象中，因此我们填充求交信息时，应该去实现 TriangleMesh 的填充方法，具体根据光线与三角形求交得到的最近的交点，填充相交信息，需要填充的有相交点位置和法线等信息，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/torus** 如果你的实现没有问题，那么你将得到以下图片结果。

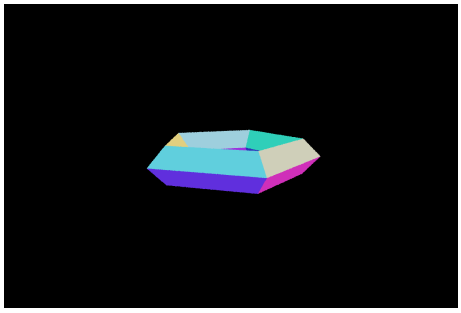

图 2: Torus 的法线

4. 加速结构介绍

本部分将会介绍一下物体求交所用的加速结构，并针对加速结构通用的包围盒结构进行求交。

4.1 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 CornellBox 这样几何面片数较少的场景是可行的，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。因此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。

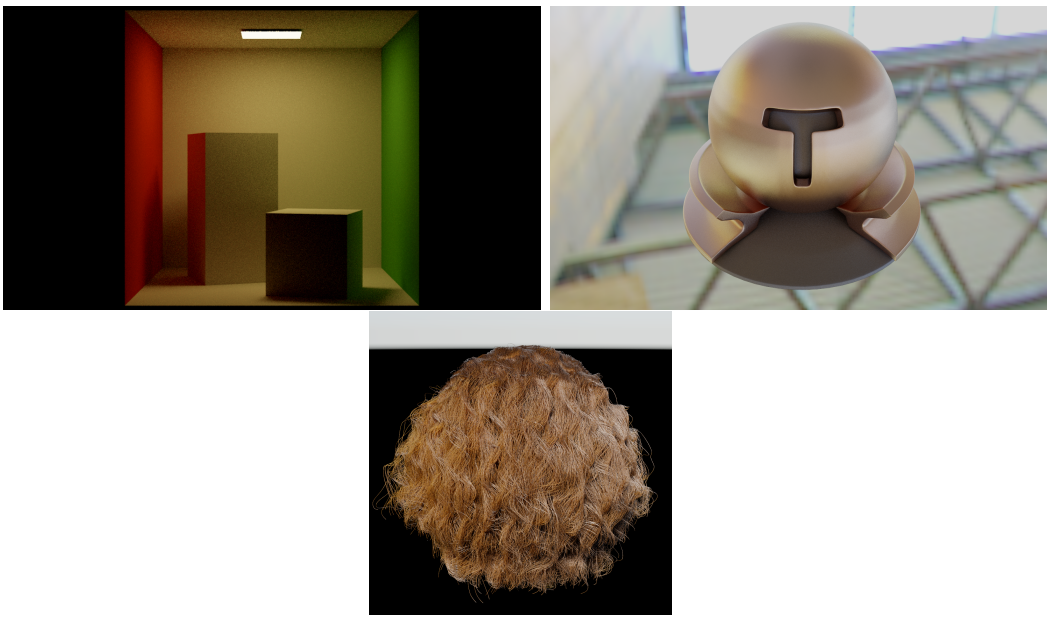

图 3: cornell-box test-ball curly-hair

4.2 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，我们使用简单的轴对齐包围盒，即 AABB 类，该类成员很简单，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;
  ...
}
```

你需要实现 AABB 的求交方法 (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。

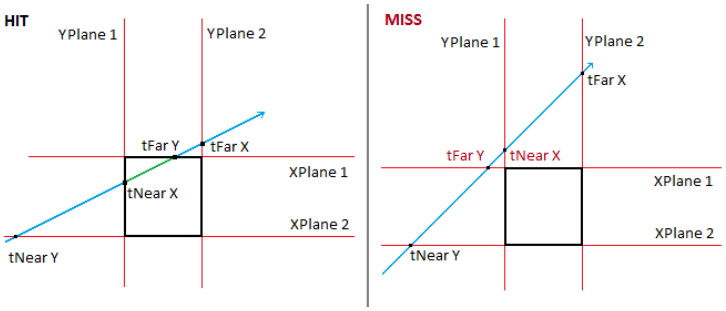

图 4: ray intersect with AABB

AABB 类的余下成员方法可以在 AABB 类文件里面查看。

4.3 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。

5. BVH

5.1 BVH 介绍

本节介绍我们本次作业的要实现的加速结构，也是目前实时光线追踪中最常用的加速结构 BVH (Bounding Volume Hierarchies)。

BVH 通过对物体进行划分来完成加速，其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逐层深入的相交测试，就可以越来越逼近实际对象。最简单的 BVH 的构成就是一棵二叉树，相交测试就是在进行二叉树的搜索，直到命中叶子节点。

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。

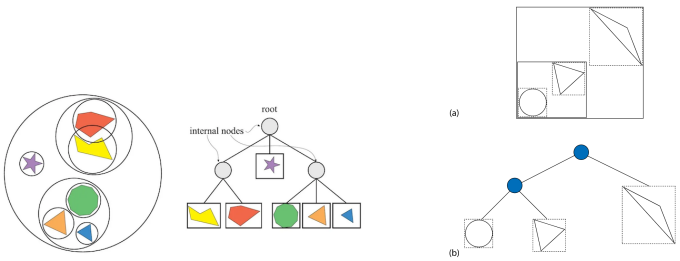

图 5: BVH

5.2 BVH 节点结构

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）

```
struct BVH::BVHNode {
    BVHNode * left;
    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）

5.3 BVH 构建

你需要实现 BVH 类的 build 方法。

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。

我们需要考虑一种场景分割依据，以便将一组物体分成两组，再交给左右子节点递归此过程，从而构建出场景的 BVH。一般来说，我们希望场景分割依据能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。

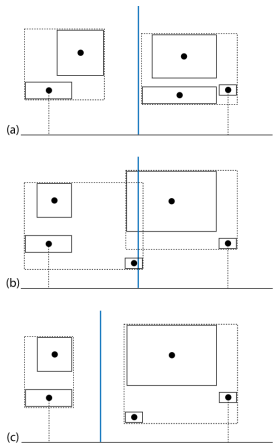

图 6: 中点分割

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.

这里我们先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，我们可以将其存入一个索引数组。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，因此我们**只需要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.

5.4 BVH 求交

你需要实现 BVH 类的 rayIntersect 方法。

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，我们可以递归 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，使用基于栈的遍历方案存储待处理的节点迭代处理是更好的选择，能够减少函数调用。

这里我们需要考虑，求交时应当先调用左子节点还是右子节点，不同的调用顺序显然会影响求交效率。假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，之前在进行划分时，我们在某一个轴上将物体坐标**从小到大排列**，因此可以根据光线在这个轴的方向上选择调用顺序。简单来说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。

5.5 验证

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

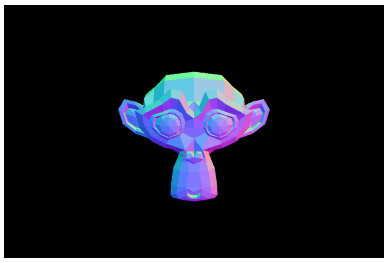

图 7: Monkey 的法线

运行场景 **lab1-test/bunny-bvh** 如果你的 BVH 实现没有问题，那么你将在几分钟完成渲染，而如果不使用加速结构，你会发现渲染的进度条几乎不变，速度极慢，渲染结果如下：

图 8: Bunny

5.6 BVH 优化 *

我们前面提到，BVH 的分割方式很大程度上影响 BVH 的性能。作为 Bonus，我们希望针对 BVH 的性能做优化。

一方面，当前的 BVH 分割方式是简单的中点分割，该分割方式构建简单，但面对许多情境存在划分重叠率高、求交重复的问题，我们需要更有效的分割算法。以下资料可供参考：

How to create awesome accelerators: The Surface Area Heuristic

Spatial Splits in Bounding Volume Hierarchies

Fast BVH Construction on GPUs

Bounding Volume Hierarchies

另一方面，当前的 BVH 仅仅是二叉树（BVH2）的组织形式，这在处理大场景时在遍历速度上相比于多叉树不占优势，且无法利用现代 CPU/GPU 的 SIMD 特性。在实际的实时渲染应用中，BVH 往往是四叉树或者八叉树的实现，我们需要多叉树的实现来进一步优化求交性能。以下资料可供参考：

Efficient SIMD Single-Ray Traversal using Multi-branching BVHs

Efficient Incoherent Ray Traversal on GPUs Through Compressed Wide BVHs

同学们如果感兴趣的话，可以任选其中之一实现一个对应的 BVH 优化，并比较一下与我们实现的 BVH 有哪些性能上的优势，可以通过求交次数等指标来衡量。