中国海洋大学
OCEAN UNIVERSITY OF CHINA

海纳百川 取则行远

计算机图形学

Lab5-7 实验报告

实验题目:	光线追踪
姓　　名:	翟一航
年　　级:	2023 级
专　　业:	软件工程
实验时间:	2026 年 1 月 12 日

目录

1 实验 5：光线与三角形相交

1.1 实验要求

在光线追踪中最重要的操作之一就是找到光线与物体的交点。一旦找到光线与物体的交点，就可以执行着色并返回像素颜色。在这次实验中，我们需要实现两个部分：光线的生成和光线与三角形的相交。

1.2 遇到的问题与解决方法

问题一： 最后生成的图片上，在阴影部分出现了一个小蓝点。如图所示

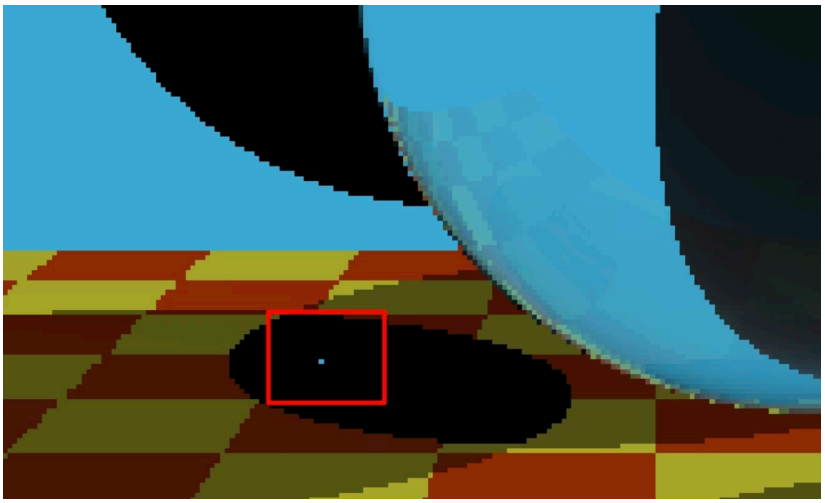

图 1: 阴影中出现小蓝点

分析与解决：这是因为在 rayTriangleIntersect 函数中对于浮点数的计算存在精度问题，在原来的代码中，我使用的判断条件是：

$$1 - b_1 - b_2 >= 0$$

但是由于 b_1 和 b_2 都是 float 类型的浮点数，在进行计算过程中会积累舍入误差，因此就导致了如图所示的情况。为了解决这个误差，将判定条件改为：

$$1 - b_1 - b_2 >= std :: numeric\_limits :: epsilon()$$

让运算结果最后大于等于一个极小的正数，如此一来就可以消除这个误差。最后的效果图为：

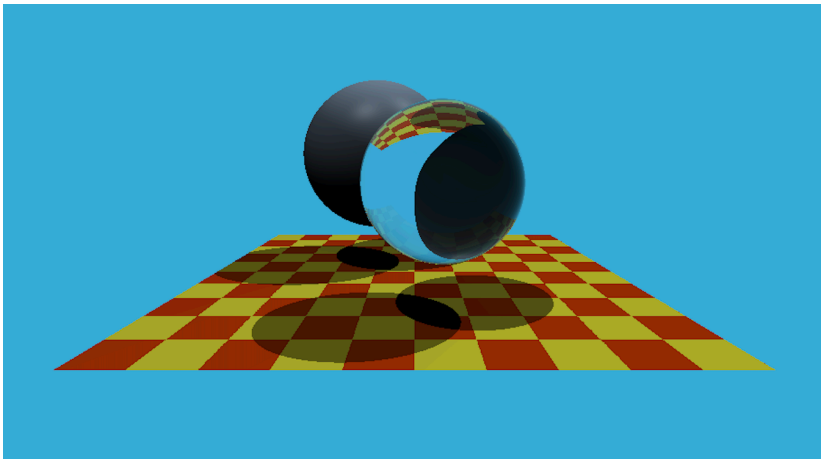

图 2: 效果图

问题二： 对于文件进行保存的代码版本太旧了，使用的还是 C 语言的实现，在我的电脑上无法正确运行。

分析与解决：使用 C++ 的 offstream 方法对文件保存代码部分进行重写，重写后的代码如下

```
std::ofstream fout("binary.ppm", std::ios::binary);
if (!fout.is_open()) {
    std::cerr << "Error: Could not open output file for writing.\n";
    return;
}

fout << "P6\n" << scene.width << " " << scene.height << "\n255\n";
for (int i = 0; i < scene.height * scene.width; ++i) {
    std::array<unsigned char, 3> color{
    static_cast<unsigned char>(255 * clamp(0, 1, framebuffer[i].x)),
    static_cast<unsigned char>(255 * clamp(0, 1, framebuffer[i].y)),
    static_cast<unsigned char>(255 * clamp(0, 1, framebuffer[i].z))
    };
    fout.write(reinterpret_cast<const char*>(color.data()), color.size());
}
```

1.3 算法的意义与价值

Möller-Trumbore 算法是计算机图形学中一个经典且高效的射线与三角形求交算法，它将射线-三角形求交问题转化为一个简洁、高效的线性系统求解问题。在实时渲染、物理模拟、碰撞检测、光线追踪等领域具有非常重要的意义与价值。

- 速度快、效率高：相比传统的两步法，Möller-Trumbore 算法计算量更小，尤其适合在光线追踪等需要数十亿次求交计算的场景中使用。
- 数值稳定性较好：算法中使用了适当的向量运算，一定程度上减少了浮点数精度误差的影响，提高了在极端情况下的鲁棒性。
- 直接输出重心坐标：算法结果直接给出了交点的重心坐标 (b1, b2)，便于后续进行纹理插值、颜色插值、法向插值等操作，非常贴合图形学渲染流程的需求。
- 实现简洁：算法本身可以用较少的代码实现，是许多图形学库、游戏引擎、渲染器中的标准实现方案。

通过此次实验，我深入理解了 Möller-Trumbore 算法的高效设计及其在图形学中的核心地位，也认识到在工程实现中必须重视浮点精度问题，合理使用容差与高精度计算来确保算法的鲁棒性。

2 实验 6：加速结构

2.1 实验要求：

在场景中的物体数量不大时，该基础光线追踪算法可以取得良好的结果，但当物体数量增多、模型变得更加复杂，该做法将会变得非常低效。因此，我们需要加速结构来加速求交过程。在本次练习中，我们重点关注物体划分算法 Bounding Volume Hierarchy (BVH)。本练习要求你实现 Ray-Bounding Volume 求交与 BVH 查找。

提高项：实现 SAH 查找进行加速。

2.2 实验过程要点与问题

（此次实验算法思路较为复杂，在实现时遇到了很多困难，难以一一阐述，因此在此讲述我遇到问题的步骤以及解决思路）

我们想要计算每条光线和所有三角形求交，但是这样一来计算量太庞大，所以就需要使用包围盒来简化计算。

2.2.1 判断光线与包围盒是否相交

我们规定，仅当光线进入到所有成对平板时，光线进入到包围盒内:

- 光线离开任意一对平板时，即离开了包围盒
- 对每一对平板，计算 $tmin$ 和 $tmax$
- 对包围盒，$t\_enter = max\{tmin\}$，$t\_exit = min\{tmax\}$

因而，光线与轴对齐包围盒相交，当且仅当 $t_enter < t\_exit$ 且 $t_exit \geq 0$

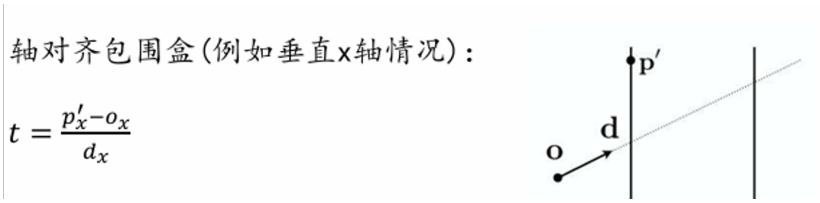

图 3: 轴对齐包围盒

在代码中，我们需要快速判断光线是否击中轴对齐包围盒，实现要点如下:

- 乘法优化: 使用预计算的 invDir（方向的倒数），将代价高昂的除法运算转换为快速的乘法。
- 区间计算: 计算光线进入/离开 x，y，z 三对平板的时间。
- 处理负方向: 通过 tMin 和 tMax 的 Min/Max 函数处理光线反向的情况，确保计算出的进入时间始终小于离开时间。
- 逻辑判定: 只有当最晚进入时间（tEnter）小于最早离开时间（tExit），且离开时间非负（tExit >= 0）时，才判定为命中。

代码实现如下:

```
inline bool Bounds3::IntersectP(const Ray& ray, const Vector3f& invDir, const std::array<int, 3>& dirIsNeg) const
{
    Vector3f t1 = (pMin - ray.origin) * invDir;
    Vector3f t2 = (pMax - ray.origin) * invDir;
    Vector3f tMin = Vector3f::Min(t1, t2);
    Vector3f tMax = Vector3f::Max(t1, t2);
    float tEnter = 0.0;
    float tExit = 0.0;
    tEnter = std::max(tMin.x, std::max(tMin.y, tMin.z));
    tExit = std::min(tMax.x, std::min(tMax.y, tMax.z));
    return tEnter < tExit && tExit >= 0;
}
```

2.2.2 BVH 递归构建

核心思想: 把一堆物体递归地二分成左右两堆（递归到只剩 1 或者 2 节点情况），给每一堆套一个包围盒，最后形成一棵二叉树。

思想虽然不难，但是在实际实现时需要考虑对与叶子结点终止条件的处理。

- 终止条件 (1 个物体): 如果只有一个物体，直接创建叶子节点，存储该物体并返回。
- 特殊处理 (2 个物体): 如果只有两个物体，直接左右各分一个，无需计算复杂的划分策略

此外，还需要考虑的是如何将大量物体分到左子树或右子树，我使用的是最长轴原则，即每次找出跨度最大的轴，在该轴上进行切分可以使子空间分布更均匀。

2.2.3 SAH-BVH

原本的 BVH 直接取包围盒最长的轴，每条光线与所有三角形求交，复杂度近似 O(N)，场景越复杂 (N 越大)，渲染越慢，因此我们需要对算法进行优化实现加速。

SAH 的核心思想是用表面积近似“命中概率”，因此需要引入关键假设。

- 对于位于凸物体 B 内部的凸物体 A，击中物体 B 的随机光线也击中 A 的概率由这些物体的表面积比值给出。
- 光线随机分布
- 光线不被遮挡
- SAH 代价函数的目标是最小化总期望代价，在左右子节点间寻找最优划分。

值得一提的是，原始的 SAH 需要枚举所有划分代价太高，所以选择使用分桶的方法，将空间划分为 B 个桶，只评估 B-1 个候选切分，这样虽然效果会有所下降，但是运行效率会大大提高，算法伪代码如下:

```
BuildBVH(objects):
  # 0) 终止条件: 物体很少时直接作为叶子 (或拆成两个叶子)
  if |objects| <= 2:
      return MakeLeaf(objects)

  # 1) 选择划分轴: 用“质心包围盒”的最长轴 (x/y/z)
  axis = LongestAxis( CentroidBounds(objects) )

  # 2) 分桶: 把该轴上的质心范围划成 B 个 bucket (B 通常 < 32)
  Init buckets[0..B-1] with (count=0, bounds=empty)

  # 3) 把每个物体放入对应 bucket, 并统计:
  #    - bucket.count: 桶里有多少物体
  #    - bucket.bounds: 桶内所有物体的 AABB 并集
  for each obj in objects:
      b = BucketIndex( Centroid(obj)[axis] )     # 质心 -> 桶编号
      buckets[b].count += 1
      buckets[b].bounds = Union(buckets[b].bounds, Bounds(obj))
  # 4) 枚举 B-1 个切分位置 i (把桶 [0..i] 归左, [i+1..B-1] 归右)
  #    用 SAH 计算该切分的期望代价: 代价越小越好
  bestSplit = -1
  bestCost  = +INF
  for i = 0 .. B-2:
      L = UnionBounds(buckets[0..i]);  NL = SumCount(buckets[0..i])
      R = UnionBounds(buckets[i+1..]); NR = SumCount(buckets[i+1..])
      cost = Ctrav + (NL*Area(L) + NR*Area(R)) / Area(ParentBounds)
      if cost < bestCost: bestCost = cost; bestSplit = i

  # 5) 按 bestSplit 真正把 objects 划分为左右两组 (基于它们所属 bucket)
  (leftObjs, rightObjs) = PartitionByBucket(objects, axis, bestSplit)

  # 6) 递归构建左右子树, 返回内部节点
  return Node( BuildBVH(leftObjs), BuildBVH(rightObjs) )
```

最终运行程序，比较是否使用 SAH 加速对于程序运行速度的影响，发现两者的差距并不是很大，大约只有百分之十的差距。思考后可能是因为原本给出的图像其实就已经比较规则，因而才导致 SAH 的加速效果并没有那么明显。

2.3 算法的意义与价值

- BVH 将复杂场景组织成树状空间划分结构，通过对大量物体进行空间分区，大幅减少光线追踪中不必要的相交测试，将计算复杂度从 O(N) 降低到 O(log N) 级别，使复杂场景的实时光线追踪成为可能。

- SAH 通过数学建模来量化不同划分方式的预期开销，从而在构建 BVH 时自动选择最优的空间划分平面，解决了传统中点或均匀划分可能导致的树结构不平衡问题，显著提升了光线查询的效率。

3 实验 7: 路径追踪

3.1 实验要求

在之前的练习中，我们实现了 Whitted-Style Ray Tracing 算法，并且用 BVH 等加速结构对于求交过程进行了加速。在本次实验中，我们将在上一次实验的基础上实现完整的 Path Tracing 算法。

3.2 遇到的问题与解决方法

助教建议此次实验可以使用多线程来加速渲染，但是由于不熟悉多线程编程，在开始时对于如何实现一直没有思路。在询问 AI 和上网查阅资料后，我发现可以使用 OpenMP 这一基于多线程的共享内存并行编程 API，充分利用 CPU 的计算能力。对实验代码中的渲染部分做出如下修改：

```
1  #pragma omp parallel for collapse(2) schedule(dynamic)
2  for (uint32_t j = 0; j < scene.height; ++j) {
3      for (uint32_t i = 0; i < scene.width; ++i) {
4          int local_m = j * scene.width + i;
5
6          float x = (2 * (i + 0.5) / (float) scene.width - 1) * imageAspectRatio * scale;
7          float y = (1 - 2 * (j + 0.5) / (float) scene.height) * scale;
8
9          Vector3f dir = normalize(Vector3f(-x, y, 1));
10         Vector3f color(0.0f);
11         for (int k = 0; k < spp; k++) {
12             color += scene.castRay(Ray(eye_pos, dir), 0) / spp;
13         }
14
15         // 使用互斥锁保护对 framebuffer 的写操作
16         #pragma omp critical
17         {
18             framebuffer[local_m] += color;
19         }
20     }
21     UpdateProgress(j / (float) scene.height);
22 }
23 UpdateProgress(1.f);
```

line1 语句创建一个并行区域，启动一个线程团队，将外层和内层两个循环（j 和 i 循环）合并成一个大的迭代空间，线程在执行过程中动态获取迭代块。最终实现将两层像素循环并行化，采用动态调度以应对渲染时间不均匀，最大化 CPU 利用率。

line16 语句则是相当于加了一个互斥锁，framebuffer 是临界资源，如果不互斥访问会导致数据竞争产生无法预料的后果。

分别在 SPP 为 128 和 512 下进行 Release 多线程渲染，发现相比于使用单线程渲染速度确实有很大提升。

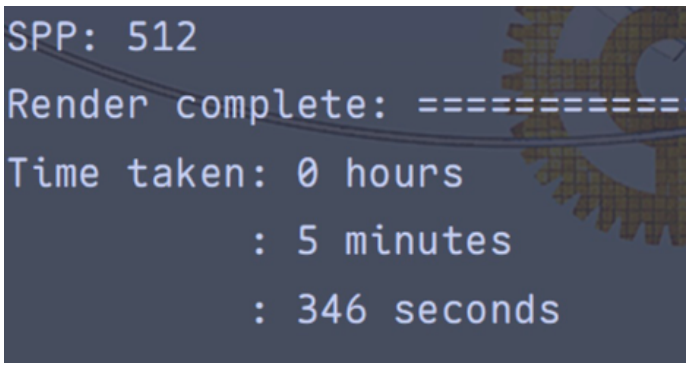

图 4: SPP512

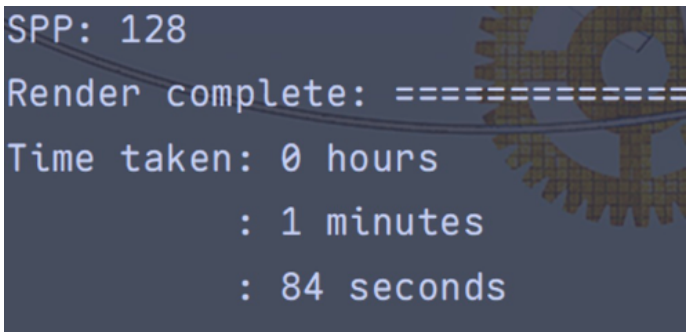

图 5: SPP128

3.3 算法的意义与价值

路径追踪首次基于物理真实的光传输方程，实现了全局光照的完整模拟，能够精确计算光线在场景中的多次弹射、漫反射、镜面反射、折射等复杂光路，从而生成具有真实感的光照效果。

- 物理正确性：严格基于辐射度学原理，确保能量守恒，生成的图像符合真实物理规律
- 无偏估计：通过蒙特卡洛方法对渲染方程进行无偏估计，理论上随着采样数增加，结果会收敛到精确解
- 处理复杂光照：能够自然模拟间接光照、漫反射互反射、环境光遮蔽等 Whitted 算法无法处理的效果

路径追踪算法奠定了现代光线追踪技术的理论基础，使得计算机生成的图像能够达到照片级真实感，推动了电影特效、虚拟现实等领域的跨越式发展。

4 收获与体会

通过这三次实验，我对于上课讲的光线追踪理论有了更深的理解。实验 5 让我直观理解了 Möller-Trumbore 算法的简洁高效。实验 6 的加速结构让我认识到优化算法的重要性，而 BVH 和 SAH 等数据结构与启发式方法通过智能的空间划分，将指数级复杂度降为对数级，可以大大提升效率。

实验 7 的路径追踪让我真正接触到现代渲染的核心，学会了如何用蒙特卡洛方法解渲染方程。多线程加速的实践更让我认识到并行计算在图形学中的必要性。它们不仅教会了我具体的图形学技术，更培养了我解决复杂问题的思维模式。