

# 计算机图形学大作业：光线追踪

作者：岑康瑞

学号：521021910151

日期：2024.6.10

计算机图形学大作业：光线追踪

总述

Experimental Details

`Render()` in `Renderer.cpp`

`Triangle::getIntersection()` in `Triangle.hpp`

`IntersectP(...)` in `Bound3.hpp`

`getIntersection(...)` in `BVH.cpp`

Result 1 (Before / After BVH Acceleration)

Other work (Optimization for BVH Build Tree)

将遍历过程转换成优化问题

具体实现

Result 2 (BVH vs BVH with SAH)

Conclusions

## 总述

本次实验要求我们实现基本的光线渲染以及使用BVH结构进行渲染加速，我将在实验细节部分提供我的代码，并通过一些数学公式或代码逻辑提供对代码的解读。另外，我的创新点在于修改了代码框架的BVH Recursive Build Tree部分的代码，将原有的Split By Count（两个BVH子树包含相同的总结点个数）修改成Split By SAH（最优化渲染过程的开销，综合考虑两子树各自的表面积以及两子树相交部分的表面积），实验效果显著：在渲染时间上得到了很好的优化，但在建树过程中的开销是渲染的数倍，这在单次渲染中是难以接受的。但是这种方法可能可以有如下应用：如果多次渲染不会改变BVH树结构，或者光线的微小改变不要求重建BVH树，那么Split By SAH方法或许能够在总时间开销上胜过Split By Count方法。

## Experimental Details

### `Render()` in `Renderer.cpp`

```
for (uint32_t j = 0; j < scene.height; ++j) {
    for (uint32_t i = 0; i < scene.width; ++i) {
        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));
        Ray ray(eye_pos, dir);

        if(check_mode)
            framebuffer[m++] = scene.castRay_noBVH(ray, 0);
        else
```

```

        framebuffer[m++] = scene.castRay(ray, 0);
    }
    UpdateProgress(j / (float)scene.height);
}

```

代码内容解释：

1. 计算像素在屏幕空间的归一化坐标。 $(i + 0.5)$  和  $(j + 0.5)$  表示将像素进行中心对齐。
2. 坐标归一化，即  $(i + 0.5) / (\text{float})\text{scene.width}$  和  $(j + 0.5) / (\text{float})\text{scene.height}$ 。
3. 将归一化坐标映射到  $[-1, 1]$  范围。从  $[0, 1] \Rightarrow [-1, 1]$ ，因此坐标  $\times 2 - 1$ ，由于图像通常从上到下扫描，因此纵坐标需要颠倒取负号。
4. 应用视野缩放和宽高比：

$$x_{\text{camera}} = x_{\text{screen}} \times \text{imageAspectRatio} \times \text{scale}$$

$$y_{\text{camera}} = y_{\text{screen}} \times \text{scale}$$

5. `Vector3f(x, y, -1)` 创建了一个方向向量，其中 `-1` 表示这个方向指向屏幕内。同时，`normalize` 函数将方向向量归一化。
6. `Ray ray(eye_pos, dir)` 创建射线，`eye_pos` 是相机的位置，`dir` 是从相机位置指向当前像素的方向向量。

## Triangle::getIntersection() in Triangle.hpp

代码的部分解释直接写在了注释中。

```

inline Intersection Triangle::getIntersection(Ray ray)
{
    Intersection inter;

    if (dotProduct(ray.direction, normal) > 0) return inter; //背面剔除 (Backface Culling) : 点积大于零意味着射线与三角形背面相交。

    double u, v, t = 0; //计算用于判定的向量和标量
    Vector3f ray_cross_e2 = crossProduct(ray.direction, e2);
    double det = dotProduct(e1, ray_cross_e2);

    if (fabs(det) < EPSILON) return inter; //判断光线是否和三角形平行。
    double det_inv = 1.0 / det; //这里我们计算det的倒数方便之后的计算，因为乘法是要快于除法的。

    Vector3f s = ray.origin - v0;
    u = dotProduct(s, ray_cross_e2) * det_inv;

    Vector3f s_cross_e1 = crossProduct(s, e1);
    v = dotProduct(ray.direction, s_cross_e1) * det_inv;

    if (v < 0 || u < 0 || u + v > 1) return inter; //利用重心坐标插值判断交点是否在三角形内
}

```

```

    t = dotProduct(e2, s_cross_e1) * det_inv;
    if (t < EPSILON) return inter;

    inter.happened = true;
    inter.coords = ray(t);
    inter.normal = normal;
    inter.distance = t;
    inter.obj = this;
    inter.m = m;

    return inter;
}

```

这里用到了 Möller-Trumbore intersection algorithm。

最主要原理：设三角形三顶点为  $P_0, P_1, P_2$ ，根据重心坐标插值：

$$o + t\vec{d} = (1 - u - v)P_0 + uP_1 + vP_2 \quad (1)$$

只需要算出  $u, v$ ，根据重心坐标插值的原理，直接就可以知道交点是否在三角形内。

而Möller-Trumbore定理告诉我们：

$$\begin{aligned}
 u &= \frac{S_1 \cdot S}{S_1 \cdot E_1} \\
 v &= \frac{S_2 \cdot d}{S_1 \cdot E_1}
 \end{aligned} \quad (2)$$

其中

$$\begin{aligned}
 S_1 &= d \times E_2 \\
 S_2 &= S \times E_1, \quad S = o - P_0
 \end{aligned}$$

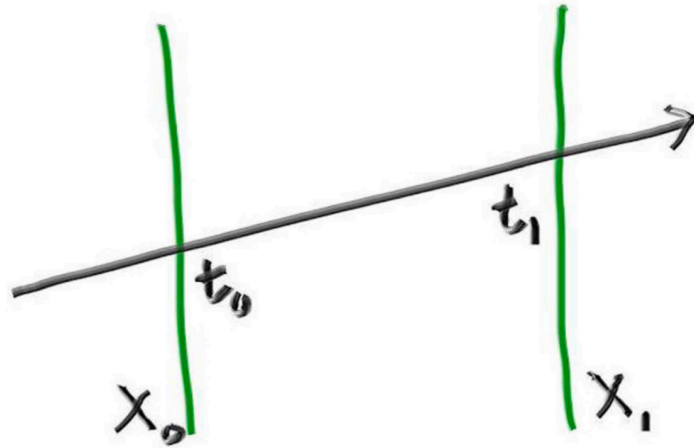
## IntersectP(...) in Bound3.hpp

```

// x dimension
t1 = (pMin.x - ray.origin.x) * invDir.x;
t2 = (pMax.x - ray.origin.x) * invDir.x;
double txmin = dirIsNeg[0] > 0 ? t1 : t2;
double txmax = dirIsNeg[0] > 0 ? t2 : t1;
// y dimension
t1 = (pMin.y - ray.origin.y) * invDir.y;
t2 = (pMax.y - ray.origin.y) * invDir.y;
double tymin = dirIsNeg[1] > 0 ? t1 : t2;
double tymax = dirIsNeg[1] > 0 ? t2 : t1;
// z dimension
t1 = (pMin.z - ray.origin.z) * invDir.z;
t2 = (pMax.z - ray.origin.z) * invDir.z;
double tzmin = dirIsNeg[2] > 0 ? t1 : t2;
double tzmax = dirIsNeg[2] > 0 ? t2 : t1;

```

```
double t_enter = std::max({txmin, tymin, tzmin});
double t_exit = std::min({txmax, tymax, tzmax});
// Check for intersection
return (t_enter < t_exit) && (t_exit >= 0);
```



总思路：考虑光线经过Bounding Box时，在x轴、y轴、z轴分别经过AABB的时间区间，如果这三个时间区间存在重合，那么就存在intersection。

## getIntersection(...) in BVH.cpp

```
Intersection BVHAccel::getIntersection(BVHBuildNode* node, const Ray& ray) const
{
    std::array<int, 3> dirIsNeg = {int(ray.direction.x > 0), int(ray.direction.y > 0),
int(ray.direction.z > 0)};
    Vector3f invDir = Vector3f(1.0f / ray.direction.x, 1.0f / ray.direction.y, 1.0f /
ray.direction.z);

    // Intersects current node
    if (!node->bounds.IntersectP(ray, invDir, dirIsNeg))
        return Intersection();

    // If leaf node => return the intersection with the object
    if (node->left == nullptr && node->right == nullptr)
        return node->object->getIntersection(ray);

    // Otherwise => recurse in the child nodes
    Intersection hitLeft = getIntersection(node->left, ray);
    Intersection hitRight = getIntersection(node->right, ray);

    // Return the closer one
    if (hitLeft.distance < hitRight.distance)
        return hitLeft;
    else
        return hitRight;
```

```
}
```

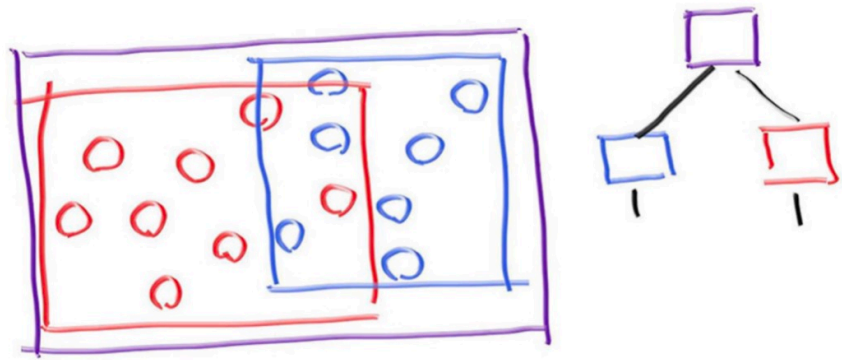
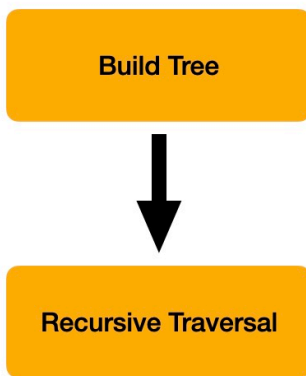


Figure 1: Bounding volume hierarchy

BVH数据结构本质上是一种二叉树（包含两种结点 Interior Node / Leaf Node）

- Leaf Node 是最终存放物体的地方；
- Interior Node 存放着代表该 Partition 的包围盒信息，下面还有两个子树有待遍历。

这段代码的逻辑如下：

- 如果当前光线与当前结点的Bounding Box不相交，直接返回，无需遍历左右子树（这也是 BVH 树结构为什么能够高效渲染的最直接原因）。否则：
  1. 如果当前结点是叶子结点，直接返回叶子结点中交点的信息。
  2. 如果当前结点是中间结点，则分别递归查找左右子节点的交点，比较两个交点的距离，返回距离较小的交点。

## Result 1 (Before / After BVH Acceleration)

```
kr@Krs-MacBook-Pro 选题1-code % ./RayTracing check
BVH Generation complete:
Time Taken: 0 hrs, 0 mins, 0 secs

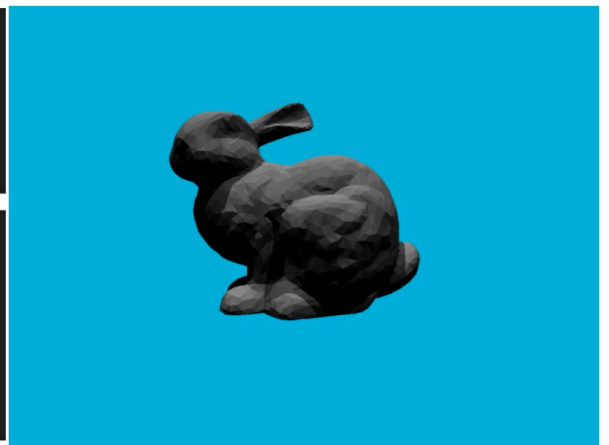
Rendering using Check mode
Render complete: =====] 100 %
Time taken: 0 hours
: 6 minutes
: 411 seconds
kr@Krs-MacBook-Pro 选题1-code %
```

```
kr@Krs-MacBook-Pro 选题1-code % ./RayTracing
BVH Generation complete:
Time Taken: 0 hrs, 0 mins, 0 secs

- Generating BVH...

BVH Generation complete:
Time Taken: 0 hrs, 0 mins, 0 secs

Render complete: =====] 100 %
Time taken: 0 hours
: 0 minutes
: 5 seconds
kr@Krs-MacBook-Pro 选题1-code %
```



BVH加速的效果十分显著。

## Other work (Optimization for BVH Build Tree)

### 将遍历过程转换成优化问题

BVH建树主要包含两个步骤：

1. 选择切分轴
2. 在轴上挑选合适的切分点，将切分点左右两部分分别设为左右子树

一般来说，第一步【选择切分轴】是基本没有争议的，即选择物体在x轴、y轴、z轴上分落的最“散”的那一条，具体实现时通常选择【在轴上最大坐标与最小坐标之差】最大的那一条。

```
Bounds3 centroidBounds;  
for (int i = 0; i < objects.size(); ++i)  
    centroidBounds =  
        Union(centroidBounds, objects[i]->getBounds().Centroid());  
int dim = centroidBounds.maxExtent();
```

然后将最大扩展轴(maxExtent)上的物体按照该坐标轴进行排序。

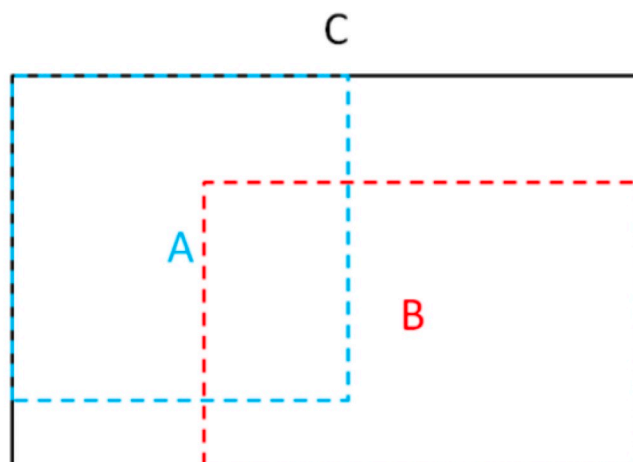
接下来便是BVH建树的关键，**如何拆分左右子树**，在这里我提供三种方案：

1. SPLIT MIDDLE (在中间划分)
2. SPLIT BY COUNT (根据数量划分)
3. SPLIT SAH (Surface Area Heuristic, 表面积启发式算法)

**INSIGHT 1：**显然SPLIT MIDDLE方案是极不合理的，因为在中间划分后，**左右子树的结点数量是不平衡的**，在极端情况下，n个物体在划分后可能左子树包含n-1个物体、右子树只包含1个物体，那么BVH树结构的优势荡然无存。并且SPLIT MIDDLE还会遇到一个问题，如果左右子树的**包围盒重合面积过大 (Bounding Box Overlap)**，可能导致光线经常性地穿过这块重合区域，这会导致左右子树都需遍历一遍，大大降低效率。

**INSIGHT 2：**SPLIT BY COUNT方案是比较可行的，它保证了左右子树在数量上的平衡，但这也不能解决包围盒重合的问题。

于是这就引出了我们SAH表面积启发式算法，我们先看如何推导：



假设光线交  $A$  的概率为  $p(A)$ , 交  $B$  的概率为  $p(B)$ , 我们可以计算光线经过  $C$  时的计算开销:

$$\text{cost}(A, B) = p(A) \sum_{i \in A} t_i + p(B) \sum_{i \in B} t_i$$

其中  $t_i$  表示判定  $i$  物体与光线相交所需要的开销, 由于它是一个相对值, 我们可以将所有的  $t_i$  看成常量1。则:

$$\text{cost}(A, B) = p(A) \cdot n_A + p(B) \cdot n_B, \text{ 其中 } n_A, n_B \text{ 代表左右子树中物体的个数。}$$

于是我们就可以得到以下公式

$$\begin{aligned} \text{SAH cost} &= \mathbb{E}[\text{cost}(A, B)] \\ &= \frac{n_A \cdot S_A}{S_C} + \frac{n_B \cdot S_B}{S_C} \end{aligned}$$

其中  $S_A, S_B, S_C$  代表包围盒的面积。

我们的优化目标可以确定为  $GOAL : \min\{\text{SAH cost}\}$ 。

## 具体实现

原代码, 关键信息 `middling = objects.begin() + (objects.size() / 2)` 可以看出采用的方法是SPLIT BY COUNT。

```
auto beginning = objects.begin();
auto middling = objects.begin() + (objects.size() / 2);
auto ending = objects.end();

auto leftshapes = std::vector<Object*>(beginning, middling);
auto rightshapes = std::vector<Object*>(middling, ending);
```

修改后的代码, 采用表面积启发式算法:

```
auto beginning = objects.begin();
auto middling = objects.begin();
auto ending = objects.end();
float min_cost = std::numeric_limits<float>::infinity(); // Initialize with infinity to
ensure first cost update
Bounds3 a;
Bounds3 b;

for (int i = 0; i < objects.size() - 1; ++i) {
    for (int j = 0; j < i; ++j) {a = Union(a, objects[j]->getBounds());}
    for (int j = i + 1; j < objects.size(); ++j) {b = Union(b, objects[j]-
>getBounds());}

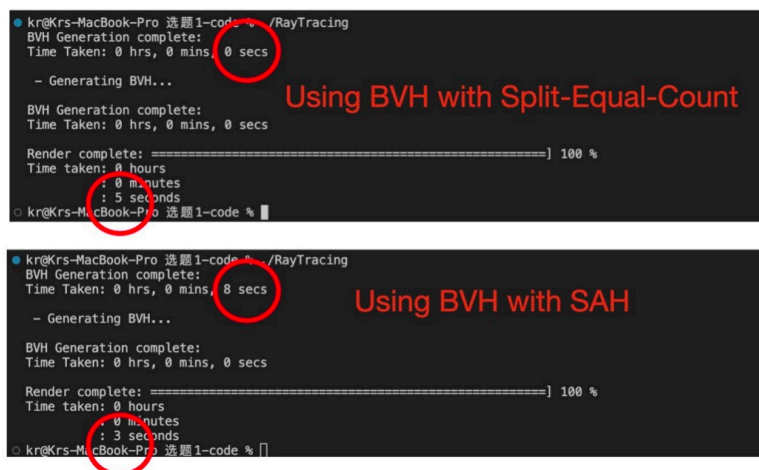
    float temp_cost = (i + 1) * a.SurfaceArea() + (objects.size() - i - 1) *
b.SurfaceArea();

    if (temp_cost < min_cost) {
        middling = objects.begin() + i;
        min_cost = temp_cost;
    }
}
```

```
}  
  
}  
  
auto leftshapes = std::vector<Object*>(beginning, middling);  
auto rightshapes = std::vector<Object*>(middling, ending);
```

主要思路：遍历objects list所有划分的可能性，如果当前划分下cost更小，则记录当前划分为目前的最优划分。由于这个过程是保证每一步都是当下最优的，“启发式”由此得来。

## Result 2 (BVH vs BVH with SAH)



```
kr@Krs-MacBook-Pro: 选题1-code % ./RayTracing  
BVH Generation complete:  
Time Taken: 0 hrs, 0 mins, 0 secs  
- Generating BVH...  
BVH Generation complete:  
Time Taken: 0 hrs, 0 mins, 0 secs  
Render complete: =====] 100 %  
Time taken: 0 hours  
: 0 minutes  
: 5 seconds  
kr@Krs-MacBook-Pro: 选题1-code %  
  
kr@Krs-MacBook-Pro: 选题1-code % ./RayTracing  
BVH Generation complete:  
Time Taken: 0 hrs, 0 mins, 0 secs  
- Generating BVH...  
BVH Generation complete:  
Time Taken: 0 hrs, 0 mins, 0 secs  
Render complete: =====] 100 %  
Time taken: 0 hours  
: 0 minutes  
: 3 seconds  
kr@Krs-MacBook-Pro: 选题1-code %
```



我们发现渲染过程优化比较明显，但同时建树开销增大，可能还需要剪枝（从目前的算法实现可以看出，SAH建树复杂度是非常大的）。

## Conclusions

1. 实现了基础的光线生成算法（提供光线、Möller-Trumbore定理判定相交）；
2. 使用BVH加速渲染（判定包围盒与光线相交、使用BVH结构加速求交），效果显著；
3. 尝试使用BVH with SAH，将光线与包围盒相交所产生的时间代价的期望值作为一个优化目标，将问题转换成了一个优化问题。SAH算法减少渲染时间的同时但增加了建树开销，需要权衡，但其在算法实现上可能有比较大的优化空间。