Ray Tracing: The Next Week 阅读笔记

> 在 "Ray Tracing in One Weekend"篇中，我们搭建了简单的路径追踪渲染器。在这篇中，我们会添加纹理，Volume(如雾)，rectangles，instances，灯光，并支持使用BVH的许多对象。完成上述内容后，我们将会实现一个“真正的”光线追踪器。本篇的难点在于**BVH结构和Perlin Noise的生成**，下面的文档会重点讲解这两个部分。

Chapter 2 运动模糊

在真实的相机中，快门会在很短的时间间隔内保持打开状态，在此期间，相机和外界的物体可能会移动。为了准确地再现这样的相机镜头，我们寻求相机在打开快门时感知到的平均值。

1.“时空”上的光线追踪介绍

运动模糊的意思是，现实世界中，相机快门开启的时间间隔内，相机或者物体发生了位移，画面最后呈现出来的像素，是移动过程中像素的平均值。我们可以通过随机一条光线持续的时间，最后计算出像素平均的颜色，这也是光线追踪使用了很多随机性的方法，最后的画面接近真实世界的原因。

这种方法的基础是当快门开机的时间段内，随机时间点生成光线（比如说随机生成x个时间点的光线，并对渲染得到的结果进行平均，就可以得到近似运动模糊后的效果）。

修改之前的ray类，添加一个光线存在时间的变量。

```
class ray {
public:
    ray(){}
    //修改构造函数，添加变量tm
    ray(const point3& origin, const vec3& direction) : orig(origin),
dir(direction),tm(0) {}
    ray(const point3& origin, const vec3& direction, double time=0.0):
orig(origin),dir(direction),tm(time){}
    point3 origin() const { return orig; }
    vec3 direction()const { return dir; }
    double time() const { return tm; } //++

    point3 at(double t) const {
        return orig + t * dir;
    }

public:
    point3 orig;
    vec3 dir;
    double tm; //添加一个与时间有关的变量
};
```

2.管理时间

快门计时有两个方面需要考虑：

- 从一个快门打开到下一个快门打开的时间；
- 每帧快门保持打开的时间。现代电影一般会使用24，30，48，60，120等帧率。

每一帧都可以有自己的快门速度。这个快门速度不必是——通常也不是——整个画面的最长持续时间。你可以让快门每帧打开1/1000秒，或者1/60秒。

就暂时来说，我们先用一个简单的模型。让相机在`time=[0, 1]`的区间内随机生成一根光线，并且实现一个可以移动的球（Sphere）类。

（1）随机生成一根光线

```
//camera.h
//修改camera类的get_ray函数如下：
ray get_ray(double s, double t) const {
    vec3 rd = lens_radius * random_in_unit_disk();
    vec3 offset = u * rd.x() + v * rd.y();
    auto ray_time = random_double();
    return ray(
        origin+offset, //把光线的起点定为圆盘内随机一点,因为圆盘是与u,v所成平面平行的所以偏移后的结果可以用u,v表示
        lower_left_corner + s * horizontal + t * vertical - origin - offset,
        ray_time); //在后面每个像素打出的光线都是随机的时间，平均下来之后就是运动模糊的结果
}
```

（2）添加可以移动的球体

修改Sphere类，使其添加关于是否随着时间移动的逻辑：

```
class sphere :public hittable {
public:
    sphere(){}
    //stationary Sphere
    sphere(point3 cen,double r,shared_ptr<material> m)
        :center(cen),radius(r),mat_ptr(m),is_moving(false){}
    //Moving Sphere
    sphere(point3 cen1, point3 cen2, double r, shared_ptr<material> m)
        :center(cen1), radius(r), mat_ptr(m), is_moving(true)
    {
        center_vec = cen2 - cen1;  //center_vec 指的是球心移动的向量
    }

    virtual bool hit(const ray& r, double t_min, double t_max, hit_record& rec)const override;
public:
    point3 center;
    double radius;
    shared_ptr<material> mat_ptr;
    //新增的变量的函数
    bool is_moving;
```

```
    vec3 center_vec;

    point3 center_cal(double time) const {
        return center + time * center_vec;
    }
};
```

同时，我们也需要修改对应的`hit`函数，因为随着时间的变换球心的位置会发生变化：

```
bool sphere::hit(const ray& r, double t_min, double t_max, hit_record& rec)
const {
    point3 center1 = is_moving ? center_cal(r.time()) : center; //add

    vec3 oc = r.origin() - center1;
    auto a = r.direction().length_squared();
    auto half_b = dot(oc, r.direction());
    auto c = oc.length_squared() - radius * radius;
    //...与三部曲第一部保持一致
}
```

（3）interval类的实现

在我们继续之前，我们将实现一个区间类来管理具有最小值和最大值的实值区间。我们以后会经常用到这个类（特别是在比如AABB和BVH Tree相关的代码上）。

```
//interval.h，记得把这个头文件的引用放在rtweekend.h文件里
#ifndef INTERVAL_H
#define INTERVAL_H

#include <limits>
const double infinity = std::numeric_limits<double>::infinity();
class interval {
public:
    double min, max;

    interval() : min(+infinity), max(-infinity) {} // Default interval is empty

    interval(double _min, double _max) : min(_min), max(_max) {}

    bool contains(double x) const {
        return min <= x && x <= max;
    }

    bool surrounds(double x) const {
        return min < x&& x < max;
    }

    static const interval empty, universe;
};

const static interval empty(+infinity, -infinity);
```

```
const static interval universe(-infinity, +infinity);

#endif
```

有了这个类之后，我们就可以进一步对`hit`函数进行修改了：

```
//hittable.h
class hittable {
public:
    virtual bool hit(const ray& r, interval ray_t, hit_record& rec) const = 0;
 //修改基类，把中间的min和max用统一interval类对象替代
};

//hittable_list.h,记得把函数对应的声明也给改了
bool hittable_list::hit(const ray& r, interval ray_t, hit_record& rec) const {
    hit_record temp_rec;
    bool hit_anything = false;
    auto closest_so_far = ray_t.max;

    for (const auto& object: objects) {
        if (object->hit(r, interval(ray_t.min,closest_so_far), temp_rec)) {
            hit_anything = true;
            closest_so_far = temp_rec.t;
            rec = temp_rec;  //修改传入的引用形式的rec,这样就可以把相交点信息传出了
        }
    }
    return hit_anything;
}

//sphere.h，记得把函数对应的声明也给改了
bool sphere::hit(const ray& r, interval ray_t, hit_record& rec) const {
    //...
    //Find the nearest root that lies in the acceptable range.
    auto root = (-half_b - sqrtd) / a; //root求解出来的是最近的hit point对应的t值
    if (!ray_t.surrounds(root)) {
        root = (-half_b + sqrtd) / a;
        if (!ray_t.surrounds(root)) return false;
    }
    //...
}

//main.cpp
//根据输入的光线方向决定输出的颜色
color ray_color(const ray& r, const hittable& world, int depth) {
    //...
    if (world.hit(r, interval(0.0001,infinity), rec)) {  //修改这个函数调用
        //对于不同材质进行判断
        ray scattered;
        color attenuation;
        if (rec.mat_ptr->scatter(r, rec, attenuation, scattered)) {
            return attenuation * ray_color(scattered, world, depth - 1);
        }
    }
    //...
}
```

（4）对反射光添加时间有关的量

对于反射，镜面反射和折射材质类，都需要修改对应的`scatter`函数，将与时间有关的量同样加入到反射光线中：

```
//material.h
//以下均改在材质类的scatter函数里
scattered = ray(rec.p, scatter_direction,r_in.time()); //lambertain类
scattered = ray(rec.p, reflected+fuzz*random_in_unit_sphere(), r_in.time());
 //matal类
scattered = ray(rec.p, direction,r_in.time());  //dieletric类
```

3.综合

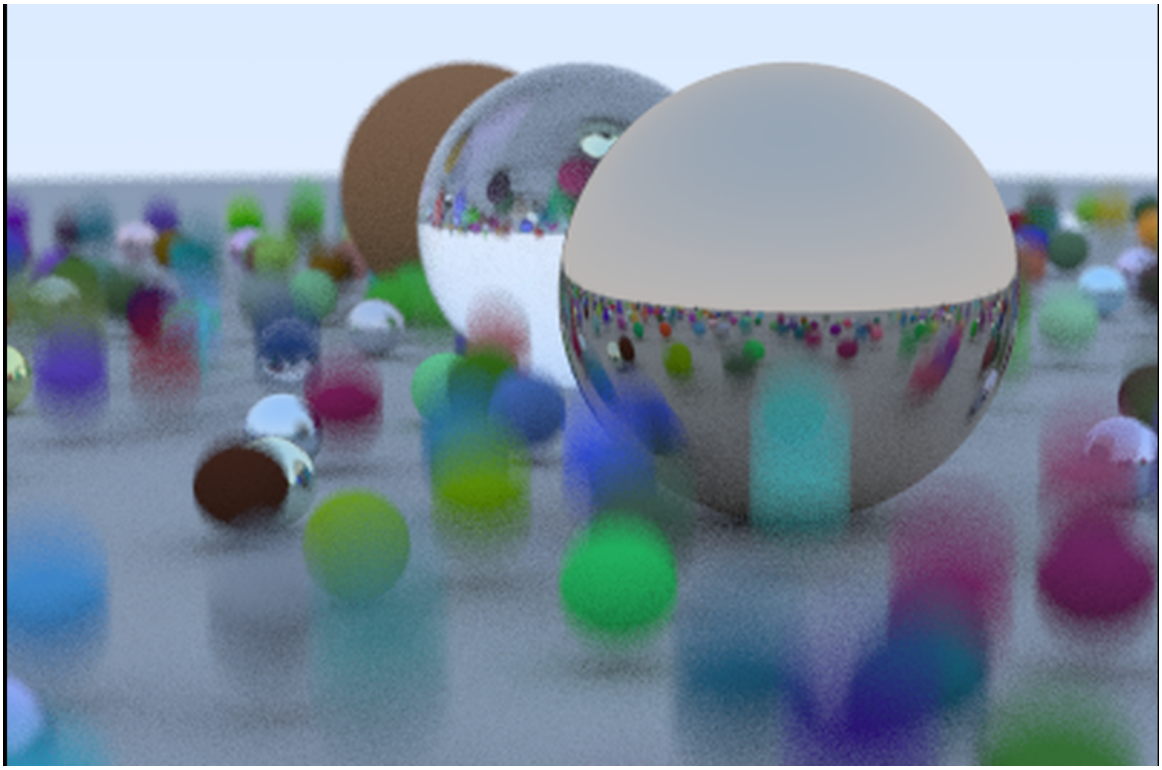
以下代码提供一个场景的示例，在t=0的时候每个球心所处的位置是C_i，而t=1的时候对应球心的位置是$C_i + vec3(0, randomDouble(0.5), 0)$。

```
hittable_list random_scene() {
    //...
    for (int a = -11; a < 11; a++) {
        for (int b = -11; b < 11; b++) {
            auto choose_mat = random_double();
            point3 center(a + 0.9 * random_double(), 0.2, b + 0.9 *
random_double());
            if ((center - point3(4, 0.2, 0)).length() > 0.9) {
                shared_ptr<material> sphere_material;
                if (choose_mat < 0.8) {
                    // diffuse,albedo选择随机的颜色
                    auto albedo = color::random() * color::random();
                    sphere_material = make_shared<lambertian>(albedo);

                    //新增的代码++++
                    auto center2 = center + vec3(0, random_double(0, 0.5), 0);
                    world.add(make_shared<sphere>(center, center2, 0.2,
sphere_material));
                }
                //.......
        }
   }

//...
//Image
const auto aspect_ratio = 3.0 / 2.0;
int image_width = 400;
int image_height = static_cast<int>(image_width/aspect_ratio);
const int samples_per_pixel = 100;
const int max_depth = 5;  //运算速度有限，这个可以调大
```

最终渲染出的结果如下：

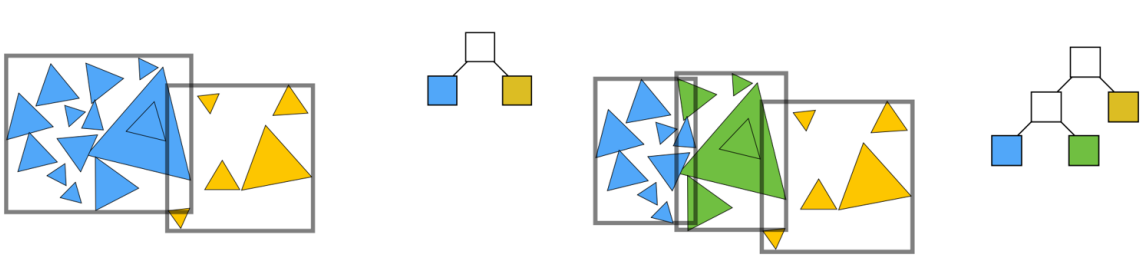


Chapter 3 BVH 数据结构

这一部分的难度还是比较大的，我们需要搭建一个BVH数据结构来做空间加速。有两种常见的加速方法：（1）分割空间，个人理解是类似与KD Tree；（2）分割物体，比如接下来的BVH数据结构。

1.核心思想

采用包围盒进行空间加速的核心思想在于，**如果光线不与包围盒相交，那么一定不会和里面的物体相交。**这样就可以先判断光线是否与包围盒相交，如果相交的话再求解子包围盒，以下是一个BVH Tree的结构和基本说明（**更详细的见第2和第3部分**）：



(1)按照物体划分成蓝色和黄色的组,当然每组要重新计算包围盒.

(2)继续划分,与KD Tree类似,这里的黄色部分也需要进行划分

与KD-Tree类似,在空间划分的时候也可以依次按照不同的轴进行.注意到,**包围盒会重叠，但一个三角形面只会被存储在唯一的包围盒内，而这也就解决了KD-Tree的缺点.**

构建BVH结构的步骤可以总结如下：

- （1）找到包围盒；
- （2）递归地将包围盒里的物体分成两个子集，设定终止条件（比如可设定为当前包围盒内三角形数量足够少）；
- （3）分别重新计算每个子集的包围盒；
- （4）在每个叶子节点当中存储物体的信息。

如何划分节点呢？

- 每次划分一般选择最长的那一轴划分；
- 选择位置在中位数的三角形作为划分依据（比如可以找**重心的中位数**，这里涉及到找中位数的算法，可以排序但是效率不高，也可以使用**快速选择算法（时间复杂度为O(n)）**。），这样可以使两边的三角形数目差不多。

以下图的BVH Tree为例，可以写出伪代码：

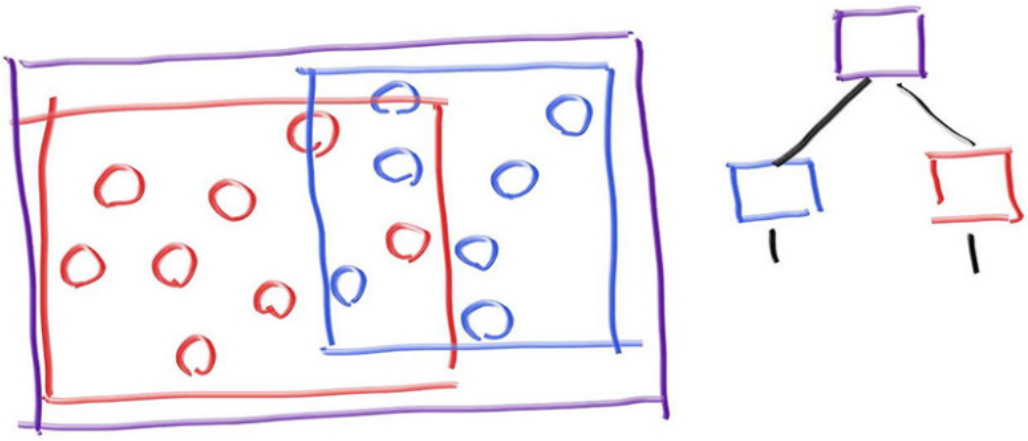


Figure 1: *Bounding volume hierarchy*

```
if(hits purple)
{
    hit0 = hits blue enclosed objects;
    hit1 = hits red enclosed objects;
    if(hit0 or hit1)
        return true and info of closer hit;
}
return false;
```

2.AABB创建

对于大多数的模型来说，AABB作为包围盒表现都是很不错的。但需要注意的是特殊情况也需要特殊考虑，这个遇到再说。对于光线和AABB求交来说，我们并不需要知道交点信息以及法线方向，只需要能够判断是否相交即可。这里以一组对边为例：

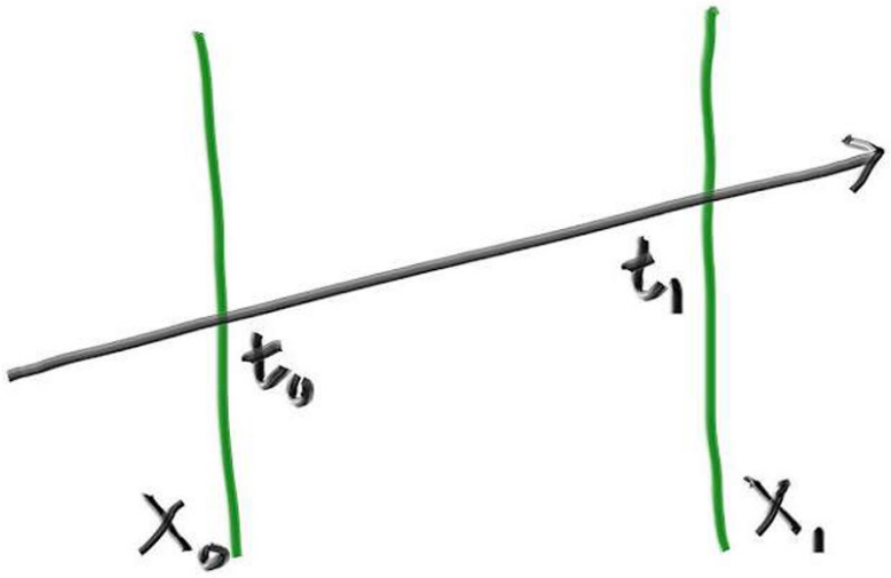


Figure 3: *Ray-slab intersection*

已知光线方程可以表示为：$P(t) = A + tb$。

上述方程对于x，y，z轴都是适配的，比如说：$x(t) = A_x + tb_x$。

当光线与$x = x_0$相交时，有$X_0 = A_x + t_0 b_0$,解得$t_0 = \frac{x_0 - A_x}{b_x}$；同理有$t_1 = \frac{x_1 - A_x}{b_x}$。

根据上面的推导，对于3D的AABB包围盒来说，只需要对三组对边的t_0和t_1进行求解，然后**依据某些判断条件判断光线和AABB是否相交即可**。简单来说伪代码可以书写如下：

```
compute(tx0, tx1);
compute(ty0, ty1);
compute(tz0, tz1); //这些tx1之类的公式参考上面，这六个值是指光线和三组对面相交时计算出的t值
return overlap? ((tx0,tx1), (ty0,ty1), (tz0,tz1))
```

看上去还是比较清晰的，但是**需要考虑到以下的特殊情况**：

- （1）想象一下光线是沿着-x方向传播的，此时(tx0，tx1)可能就需要进行翻转，比如（3，7）翻转为（7，3）；
- （2）由于使用了除法，就需要考虑除0等情况。比如如果光线的原点在某个AABB的面上，就会得到t的结果为NaN。在不同的光线追踪器中有不同的解决方案。
- （3）还有矢量化问题，如SIMD，我们在这里不讨论。Ingo Wald的论文是一个很好的起点，如果你想在矢量化方面做得更快。

对于我们的渲染器来说，尽可能采用简单的方式来实现，讨论如下：

根据前文，解出$t_{x0} = \frac{x_0 - A_x}{b_x}$；$t_{x1} = \frac{x_1 - A_x}{b_x}$。此时如果$b_x = 0$，则会出现除0的情况，解决方案是取min和max操作，这样可以规避由分母为0带来的问题（当分母为0时，结果为正/负无穷，取min或max操作后依然可以最终得到正确的结果）：

$$t_{x0} = \min\left(\frac{x_0 - A_x}{b_x}, \frac{x_1 - A_x}{b_x}\right)$$
$$t_{x1} = \max\left(\frac{x_0 - A_x}{b_x}, \frac{x_1 - A_x}{b_x}\right)$$

对于两组包围盒对面来说，假定interval的第一个值小于第二个值，可以写出如下的伪代码：

```
bool overlap(d,D,e,E,f,F)
{
    f = max(d,e); //两组对边都进入才算进入
    F = min(D,E); //只要有一组对边离开就算离开
    return (f < F);
}
```

这里还有个问题，就是如果$x_0 - A_x$和b_x同时为0的时候，还是会出现NaN的情况，解决方案是可以做一个padding包围盒的操作。

```
class interval {
public:
    //...
    double size() const {
        return max - min;
    }

    interval expand(double delta) const {
        auto padding = delta / 2;
        return interval(min - padding, max + padding);
    }
    //...
};
```

AABB 代码

```
//AABB.h
#ifndef AABB_H
#define AABB_H
#include "rtweekend.h"

class aabb {
public:
    interval x, y, z;
    aabb(){}  //the default AABB is empty, since intervals are empty by default
    aabb(const interval& ix, const interval& iy, const interval& iz):
x(ix),y(iy),z(iz){}
    aabb(const point3& a, const point3& b)
    {
        //传入的两个点是包围盒的“左下角”和“右上角”的值，即两个点代表了三个轴的最大最小值
        //求解出的是三组对立面的x，y，z范围
        x = interval(fmin(a[0], b[0]), fmax(a[0], b[0]));
        y = interval(fmin(a[1], b[1]), fmax(a[1], b[1]));
        z = interval(fmin(a[2], b[2]), fmax(a[2], b[2]));
    }

    const interval& axis(int n) const {
        if (n == 1) return y;
        if (n == 2) return z;
        return x;
    }

    bool hit(const ray& r, interval ray_t) const{
```

```
        for (int a = 0; a < 3; a++) {
            //实现上述公式：针对x，y，z轴分别计算t0（tenter）和t1（texit），然后求
ray_t.min=max(tenter)，ray_t.max=min(exit)
            auto t0 = fmin((axis(a).min - r.origin()[a]) / r.direction()[a],
                (axis(a).max - r.origin()[a]) / r.direction()[a]);
            auto t1 = fmax((axis(a).min - r.origin()[a]) / r.direction()[a],
                (axis(a).max - r.origin()[a]) / r.direction()[a]);
            ray_t.min = fmax(t0, ray_t.min);
            ray_t.max = fmin(t1, ray_t.max);
            if (ray_t.max <= ray_t.min)
                return false;
        }
        return true;
    }
};
#endif
```

一个优化之后的Hit函数

皮克斯的Andrew Kensler做了一些实验，并提出了以下版本的代码，在许多编译器上都执行良好：

```
//aabb.h
bool hit(const ray& r, interval ray_t) const {
    for (int a = 0; a < 3; a++) {
        auto invD = 1 / r.direction()[a];
        auto orig = r.origin()[a];

        auto t0 = (axis(a).min - orig) * invD;
        auto t1 = (axis(a).max - orig) * invD;

        if (invD < 0)
            std::swap(t0, t1);
        if (t0 > ray_t.min) ray_t.min = t0;  //最终记录的ray_t.min是max
        if (t1 < ray_t.max) ray_t.max = t1;

        if (ray_t.max <= ray_t.min)
            return false;
    }
    return true;
}
```

3.构建场景物体的包围盒

对于interval类来说，默认是empty的，在书写类的时候考虑到了这种情况，因此如果`hittable_list`里面没有物体也不影响运算。

同样需要考虑的是，有些物体是会运动的，此时AABB会发生变化。

对代码做的改动如下：

```
//hittable.h
#include "aabb.h"
class hittable {
public:
    virtual bool hit(const ray& r, interval ray_t, hit_record& rec) const = 0;
    //新增:包围盒计算
    virtual aabb bounding_box() const = 0;
};
```

对于静态的球体，包围盒计算函数是比较简单的。而对于移动的球体来说，可以把t=0和t=1时刻的包围盒都求解出来，然后对两个包围盒再求解出一个包围盒。代码如下：

```
class sphere :public hittable {
public:
    sphere(){}
    //stationary Sphere
    sphere(point3 cen,double r,shared_ptr<material> m)
        :center(cen),radius(r),mat_ptr(m),is_moving(false)
    {
         //++++++++++对于静态的球只需要解出包围盒即可++++++++++
        auto rvec = vec3(radius, radius, radius);
        bbox = aabb(center - rvec, center + rvec);
    }

    //Moving Sphere
    sphere(point3 cen1, point3 cen2, double r, shared_ptr<material> m)
        :center(cen1), radius(r), mat_ptr(m), is_moving(true)
    {
         //对于动态的球，可以分别求出两个时刻的包围盒，再对两个包围盒取一次包围盒操作
         auto rvec = vec3(radius, radius, radius);
        aabb box1(cen1 - rvec, cen1 + rvec);
        aabb box2(cen2 - rvec, cen2 + rvec);
        bbox = aabb(box1, box2);
        center_vec = cen2 - cen1;  //center_vec 指的是球心移动的向量
    }
    virtual bool hit(const ray& r, interval ray_t, hit_record& rec)const
override;
    aabb bounding_box() const override { return bbox; }
public:
    point3 center;
    double radius;
    shared_ptr<material> mat_ptr;  //hittable.h里面声明了material,因此不需要引入新的头
文件
    bool is_moving;
    vec3 center_vec;
    aabb bbox;

    point3 center_cal(double time) const {
        return center + time * center_vec;
    }
};
```

对应的aabb类里应该添加一个将两个aabb作为参数的构造函数，并为interval类添加一个构造函数：

```
class interval{
    //+++  两个包围盒求包围盒的话只需要求解两个包围盒的最小的x，最大的x，其他轴同理
    interval(const interval& a, const interval&
b):min(fmin(a.min,b.min)),max(fmax(a.max,b.max)){}
}

class aabb{
    //+++
    aabb(const aabb& box0, const aabb& box1) {
        x = interval(box0.x, box1.x);
        y = interval(box0.y, box1.y);
        z = interval(box0.z, box1.z);
    }
}
```

接下来，就是对hittable_list类提供构建包围盒的函数：

```
//hittable_list.h
#include "aabb.h"
class hittable_list :public hittable {
public:
    //...
    void add(shared_ptr<hittable> object) {
        objects.push_back(object);
        bbox = aabb(bbox, object->bounding_box()); //相当于每加入一个新物体就更新一次
包围盒
    }
    virtual bool hit(
        const ray& r, interval ray_t, hit_record& rec) const override;
    aabb bounding_box()const override { return bbox; }
public:
    std::vector<shared_ptr<hittable>> objects;
    aabb bbox;
};
```

---

4. BVH节点类

对于BVH Tree数据结构而言，有两种构建方式：一是将BVH树与其对应节点拆成两个类来写，而是合并到一个类当中。这里采用第二种方式。注意bvh_node类也需要继承基本的hittable类。可以写出基本的代码结构如下：

```
//bvh.h
#ifndef BVH_H
#define BVH_H
#include "rtweekend.h"
#include "hittable.h"
#include "hittable_list.h"

class bvh_node :public hittable {
public:
```

```
    bvh_node(const hittable_list& list): bvh_node(list.objects, 0,
list.objects.size()){}
    bvh_node(const std::vector<shared_ptr<hittable>>& src_objects, size_t start,
size_t end) {
        //To be implemented later
    }

    bool hit(const ray& r, interval ray_t, hit_record& rec) const override {
        if (!box.hit(r, ray_t)) //与包围盒不相交,直接返回false
            return false;
        bool ht_left = left->hit(r, ray_t, rec);
        bool hit_right = right->hit(r, interval(ray_t.min, hit_left ? rec.t :
ray_t.max), rec); //如果有hit_left,还要比较一下最近的交点,用rec来记录
        return hit_left || hit_right;
    }

    aabb bounding_box() const override { return bbox; }

private:
    shared_ptr<hittable> left;
    shared_ptr<hittable> right;
    aabb bbox;
};

#endif // !BVH_H
```

5.创建BVH树

关于BVH数据结构，最难的部分就是如何构建一颗BVH树，这里采用的算法如下：

- （1）随机选择一个轴；
- （2）使用 `std::sort` 函数对所有的物体进行排序；
- （3）选择排序后中间的物品进行分开，分成左右两棵子树。

如果只剩下了一个物体的话，就把它分别放在两棵subtree当中。这颗树的创建过程放在bvh_node类的构造函数中，如下：

```
#include <algorithm>
bvh_node(const std::vector<shared_ptr<hittable>>& src_objects, size_t start,
size_t end) {
    auto objects = src_objects;
    int axis = random_int(0, 2); //见rtweekend.h
    auto comparator = (axis == 0) ? box_x_compare
        : (axis == 1) ? box_y_compare
            : box_z_compare;  //这几个compare是类似函数指针,决定comparator是哪个函数

    size_t object_span = end - start; //BVH节点里有几个物体
    if (object_span == 1) {
        left = right = objects[start];
    }
    else if (object_span == 2) {
        if (comparator(objects[start], objects[start + 1])) {
```

```
            left = objects[start];
            right = objects[start + 1];
        }
        else {
            left = objects[start + 1];
            right = objects[start];
        }
    }
    else {
        std::sort(objects.begin() + start, objects.begin() + end, comparator);
        auto mid = start + object_span / 2;
        left = make_shared<bvh_node>(objects, start, mid);
        right = make_shared<bvh_node>(objects, mid, end);
    }
    bbox = aabb(left->bounding_box(), right->bounding_box());
}
```

其中random_int函数写在了rtweekend.h头文件里，如下：

```
//rtweekend.h
inline int random_int(int min, int max) {
    return static_cast<int>(random_double(min, max + 1));
}
```

比较函数如下所示：

```
class bvh_node :public hittable {
//...
private:
    static bool box_compare(
        const shared_ptr<hittable> a, const shared_ptr<hittable> b, int
axis_index
    ) {
        return a->bounding_box().axis(axis_index).min < b-
>bounding_box().axis(axis_index).min;
    }

    static bool box_x_compare(const shared_ptr<hittable> a, const
shared_ptr<hittable> b) {
        return box_compare(a, b, 0);
    }

    static bool box_y_compare(const shared_ptr<hittable> a, const
shared_ptr<hittable> b) {
        return box_compare(a, b, 1);
    }

    static bool box_z_compare(const shared_ptr<hittable> a, const
shared_ptr<hittable> b) {
        return box_compare(a, b, 2);
    }
}
```

在创建BVH树的时候，只需要增添一行代码即可：

```
//main.cpp
#include "bvh.h"
hittable_list random_scene() {
    //...
    auto material3 = make_shared<metal>(color(0.7, 0.6, 0.5), 0.0);
    world.add(make_shared<sphere>(point3(4, 1, 0), 1.0, material3));

    //加入bvh树进行管理
    world = hittable_list(make_shared<bvh_node>(world));

    return world;
}
```

经过实际测试，添加了BVH Tree空间加速结构的代码执行速度确实会快不少。

Chapter 4 纹理映射