<https://blog.csdn.net/qq_35312463/article/details/108299036>

<https://raytracing.github.io/books/RayTracingInOneWeekend.html>

<https://github.com/RayTracing/raytracing.github.io/>

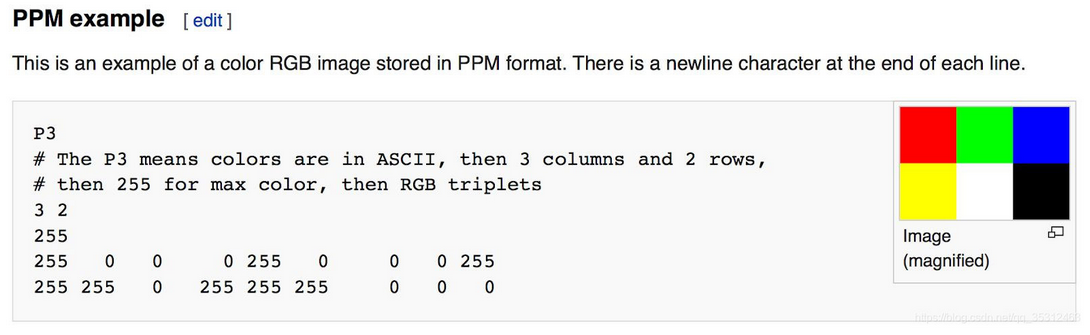
# 【光线追踪系列一】PPM输出；自定义vec3；光线、简单相机及天空采样

本次光线追踪系列从基础重新开始，主要参照 [Ray Tracing in One Weekend](https://raytracing.github.io/books/RayTracingInOneWeekend.html) ，具体实现代码框架见 <https://github.com/RayTracing/raytracing.github.io/>。本文只是主要精炼光追相关理论，具体实现可参照原文。

## 一、PPM输出

### 1.1 PPM格式

ppm是一种直接存储RGB颜色值的文件格式，第一行是p3，表示颜色值用ASCII存。第二行是图像的宽和高。接下来是每一行按顺序存放的颜色值。



具体实现可参照原文，本人实现效果如下：



## 二、自定义vec3类

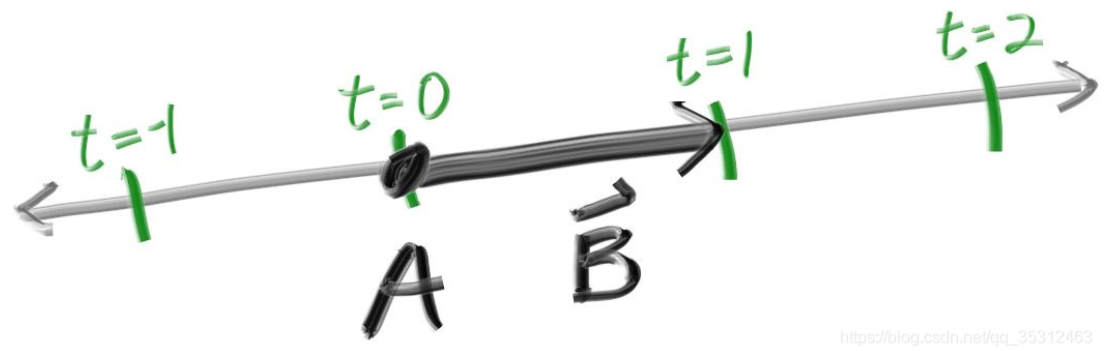
几乎所有图形程序都有一些用于存储几何矢量和颜色的类。在许多系统中，这些向量是4D（3D加上几何的齐次坐标，而RGB加上颜色的alpha透明通道）。就我们的目的而言，三个坐标就足够了。我们将对vec3颜色，位置，方向，偏移量等使用相同的类。  
原文中，除了实现一些基本的操作外，还在该类中重载了许多vec3的拓展，用户可自己前去查看。

## 三、光线、简单相机及天空采样

本部分主要实现：  
以相机所处的原点(0,0,0)为起点，对屏幕的每个像素都发射一条射线：  
①. 如果命中物体，就处理命中物体的情况，这种情况放到下节处理  
②. 如果没有命中，获取射线方向的单位向量的y值，从0到1对浅蓝色和白色进行插值，以模拟天空的颜色，最终将颜色值输出到屏幕。

### 3.1 光线类

Ray Tracing是一个光学成像的逆过程。所有光线跟踪器都具有的一件事是光线类别和对沿光线看到的颜色进行计算。让我们将射线视为函数**P(t)= A +tb**。这里 P 是沿3D线的3D位置。 一个 是射线的起源， b是射线方向。射线参数t是一个实数。插入其他t和 P(t)沿射线移动点。加上负数t值，您可以在3D线上的任何位置。对于正确的t，您只会得到在正向的t，这就是通常所说的半线或射线。



射线类代码如下：

class ray{

public:

vec3 A; //起点

vec3 B; //方向

ray() {}

ray(const vec3 &a, const vec3 &b){

A = a;

B = b;

}

vec3 origin() const { return A; }

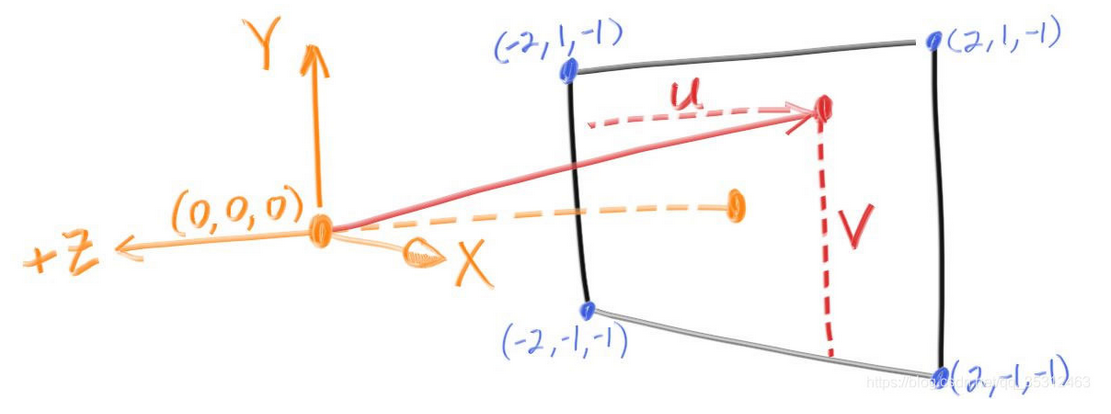
vec3 direction() const { return B; }

vec3 point\_at\_parameter(float t) const { return A + t \* B; } //终点的坐标

};

### 3.2 简单相机和背景输出

首先用一种非常简单的方式定义相机和屏幕：相机就是位于原点的一个点，屏幕就是一个4x2的矩形。



如图所示，假定Y轴指向上方，Z轴反方向面向屏幕，X轴指向右边。  
u为像素在水平方向上的比值，v为像素在竖直方向上的比值，范围为[0,1]。  
将相机的位置放置在原点(0,0,0)处，屏幕的左下角为(-2,-1,-1)，右上角为(2,1,-1)，宽为4，高为2。

接下来实现颜色输出函数，该函数的功能：**发射一条射线，并采样该射线最终输出到屏幕的颜色值。**

//发射一条射线，并采样该射线最终输出到屏幕的颜色值

vec3 color(const ray &r){

vec3 blue = vec3(0.5, 0.7, 1.0);

vec3 white= vec3(1.0, 1.0, 1.0);

vec3 unit\_direction = unit\_vector(r.direction());

float t = 0.5 \* (unit\_direction.y() + 1.0); //确保t的范围为[0,1]

return (1.0 - t) \* white + t \* blue ; //对白色和浅蓝色插值

}

**函数解释**：由于目前我们还没有定义任何会被射线命中的物体，在没有命中物体的情况下，所以上面我们直接用线性插值公式，根据射线方向的单位向量的y，对白色和浅蓝色进行插值，以模拟天空的颜色。  
该现行插值公式如下：



其中C为输出的颜色，A为起始颜色，B为目标颜色。

说明：使用这种对射线方向的单位向量的y分量进行插值的方式，可以实现任意方向平行光一致的效果。也就是说，无论射线的起点在哪里，射线采样到的颜色只和仰角（即射线和XOZ平面的夹角）有关。举个例子，在这种情况下，在屋顶的东面45°仰望天空，和在南面45°仰望天空，观察到的颜色值是一样的。因为后面会涉及射线命中物体后的反射行为，反射后，光线的起点和方向会被修改，但这不影响射线对最终天空颜色值的正常采样，因此要事先特别说明一下。后面我们将会看到绝对光滑的金属材质，可以像镜子一样反射得到天空的颜色。

接下来，我们会在入口函数中，从原点出发，朝着屏幕上的每一个像素发射一条射线，然后用 color( r ) 函数返回该射线对应的颜色值，输出到屏幕上。

vec3 lower\_left\_corner(-2.0, -1.0, -1.0); //左下角

vec3 horizontal(4.0, 0.0, 0.0);

vec3 vertical(0.0, 2.0, 0.0);

vec3 origin(0.0, 0.0, 0.0);// 相机原点

void RayTracing(){

for (int j = ny - 1; j >= 0; j--)

{

for (int i = 0; i < nx; i++)

{

float u = float(i) / float(nx); //像素在水平方向上的比值，范围[0,1]

float v = float(j) / float(ny); //像素在竖直方向上的比值，范围[0,1]

ray r(origin, lower\_left\_corner + u\*horizontal + v\*vertical);

vec3 col = color(r);

int ir = int(255.99 \* col[0]);

int ig = int(255.99 \* col[1]);

int ib = int(255.99 \* col[2]);

DrawPixel(i, j, ir, ig, ib);//根据像素点i.j位置绘制rgb颜色

}

}

}

最终可得到这张类似于天空的图像：



# 【光线追踪系列二】添加球体与表面法向量

本次光线追踪系列从基础重新开始，主要参照 [Ray Tracing in One Weekend](https://raytracing.github.io/books/RayTracingInOneWeekend.html) ，具体实现代码框架见 <https://github.com/RayTracing/raytracing.github.io/>。本文只是主要精炼光追相关理论，具体实现可参照原文。

上一节中，我们以相机所处的原点（0,0,0)为起点，对屏幕的每个像素都发射一条射线，并实现了光线没有命中物体时，返回天空颜色的效果。本部分处理如果命中物体的情况。

## 一、球体

### 1.1 判断射线是否命中球体

该项目的所有物体都是球体，因为射线命中球体的计算方式非常简单粗暴。  
对于球心坐标为C=(Cx,Cy,Cz)，半径为RR的一个球体，其方程如下：

设有一点P=(x,y,z)，则从球心到点P的向量为 (p−C)。  
因为

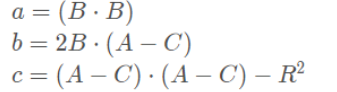
所以球的方程可以写成

对于射线：p(t)=A+tB，假设t取某个值的时候，射线可能会击中这个球体。 将光线方程带入(1)，有:

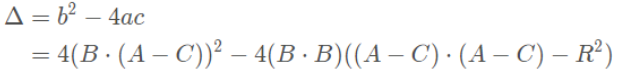
展开得:

变形得:

设



则有判别式:

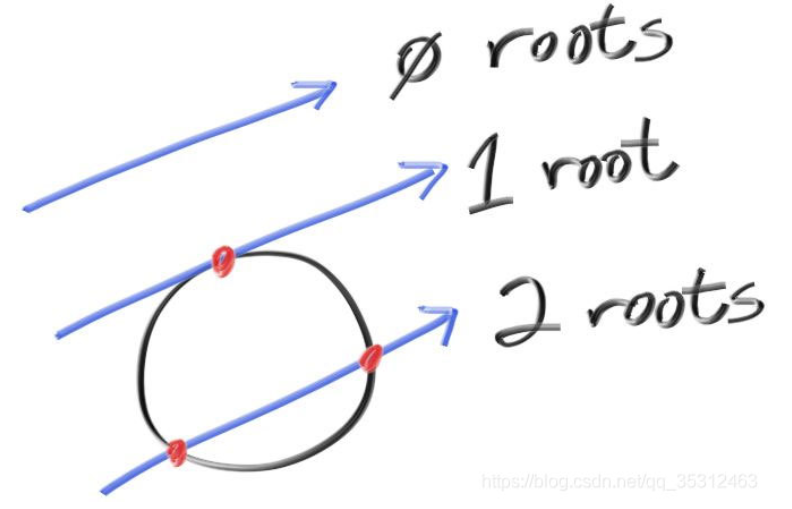


在这里插入图片描述

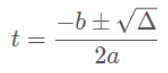
①.当 Δ>0时，方程有两个不相等的实数根，即射线穿过;

②.当Δ=0时，方程有两个相等的实数根，即射线与球体相切;

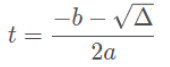
③.当Δ<0时，方程无实数根，即射线没有命中球体.



方程的解为:



如果有正根，则取较小的实数根:



此时，射线的终点坐标p(t)=A+tB，刚好就是射线前进方向，命中球体的位置。  
代码表示如下：

//输入球（球心坐标和半径）、射线

//若射线命中球，返回较小的实数根，否则返回-1

float hit\_sphere(const vec3 &center, float radius, const ray &r){

vec3 oc = r.origin() - center;

float a = dot(r.direction(), r.direction());

float b = 2.0 \* dot(oc, r.direction());

float c = dot(oc, oc) - radius \* radius;

float discriminant = b \* b - 4 \* a \* c;

if (discriminant < 0) { //没有实数根

return -1.0;

} else {

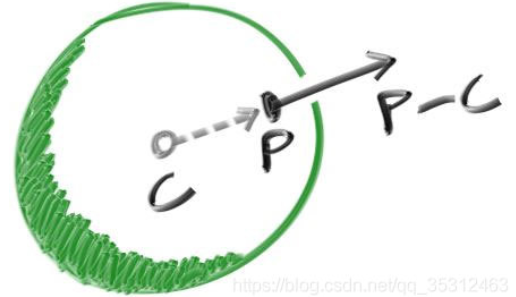
return (-b - sqrt(discriminant)) / (2.0 \* a); //返回较小的实数根

}

}

## 二、表面法向量

在图形学中，表面法向量可以用于实现光照中的镜面反射。  
一个球的表面法向量，就是一个起点在切点且与相切平面相垂直的单位向量。



假设射线在点P=(x,y,z)P=(x,y,z)命中球体，球心坐标为C=(Cx,Cy,Cz)，则此时的表面法向量为 (p−C)。

接下来，我们进行表面法向量的可视化。假设有一个球心坐标为(0,0,-1)，半径为0.5的球，我们依旧在上一节的位置进行光追采样，如果没命中球，则按照上节的处理方式，返回天空的颜色。如果击中球，就计算击中点的表面法向量，并将该表面法向量的y值转换到RGB的G通道，用绿色进行可视化。

代码如下：

//发射一条射线，并采样该射线最终输出到屏幕的颜色值

vec3 color(const ray &r){

vec3 center = vec3(0, 0, -1);

float radius = 0.5;

float t = hit\_sphere(center, radius, r);

if (t > 0.0) //如果有正实数根

{

vec3 p = r.point\_at\_parameter(t);

vec3 N = unit\_vector(p - center);

// vec3 res = 0.5 \* vec3(0, (N.y() + 1), 0); //只可视化G通道

vec3 res = 0.5 \* vec3((N.x() + 1), (N.y() + 1), (N.z() + 1)); //可视化RGB通道

return res;

}

vec3 unit\_direction = unit\_vector(r.direction());

t = 0.5 \* (unit\_direction.y() + 1.0);

return (1.0 - t) \* vec3(1.0, 1.0, 1.0) + t \* vec3(0.5, 0.7, 1.0);

}

如下图：



解释：在取得射线命中的球体表面的法向量之后，可知球的“北极”附近的法向量大约为(0,1,0)，“南极”附近的法向量大约为(0,-1,0)，加一并减半处理后，分别缩放为(0,1,0)和(0,0,0)，也就是顶部绿色和下部黑色，如上图展示结果

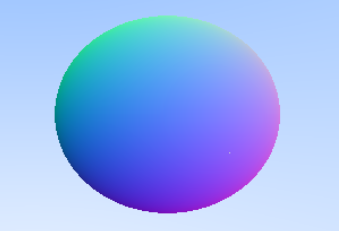
假设将表面法向量的x,y,z值分别都转换到相应的RGB，只需将上面第11行代码修改为：

vec3 res = 0.5 \* vec3((N.x() + 1), (N.y() + 1), (N.z() + 1));

如下图：



你也可以尝试改变球体圆心坐标及半径尝试不同效果：



# 【光线追踪系列三】场景对象基类封装

本次光线追踪系列从基础重新开始，主要参照 [Ray Tracing in One Weekend](https://raytracing.github.io/books/RayTracingInOneWeekend.html) ，具体实现代码框架见 <https://github.com/RayTracing/raytracing.github.io/>。本文只是主要精炼光追相关理论，具体实现可参照原文。



接着上一节定义单个球体功能后，我们可以想一下场景中，如果有很多个球体怎么办呢？如果光线命中特殊材质后发射了反射怎么办？  
我们可以通过以下解决方案来适用：  
1.将球体封装成类(为了场景中多模型适用)。  
2.一条射线的路径上可能会命中好几个球，因此要选出射线能命中的最近的球(直观上需要展示的颜色值)。  
3.用数组存储这些球（方便后续再利用）。  
4.如果发射了反射等行为，则要记录上一条射线的命中信息，例如命中终点、命中点的法向量等（多次反射使用）。

## 一、场景记录及抽象类

### 1.1 命中信息记录

struct hit\_record{

float t; //命中射线的长度

vec3 p; //命中终点坐标

vec3 normal; //命中点的法向量};

如果想选出射线所命中的最近的球，可以循环遍历所有能命中的球，然后每一趟更新t值，最终能得到最小的t值。

### 1.2 物体的虚基类

物体的虚基类hittable只定义了一个虚函数hit()，该函数用于记录射线的命中信息，并返回是否击中。

class hittable{

public:

virtual bool hit(const ray &r, float t\_min, float t\_max, hit\_record &rec) const = 0;};

其中，参数t\_min和t\_max区间用于限定射线的长度，用于排除我们不想被命中的较远处的物体。

## 二、实现球类

球类sphere继承了物体hittable，并实现了hit()函数，作用是记录射线在t\_min和t\_max区间内的命中信息。

class sphere : public hittable{

public:

vec3 center;

float radius;

sphere() {};

sphere(vec3 cen, float r) : center(cen), radius(r){};

//如果命中了，命中记录保存到rec

virtual bool hit(const ray &r, float t\_min, float t\_max, hit\_record &rec) const

{

vec3 oc = r.origin() - center;

float a = dot(r.direction(), r.direction());

float b = dot(oc, r.direction());

float c = dot(oc, oc) - radius \* radius;

float discriminant = b \* b - a \* c;

if (discriminant > 0)

{

float temp = (-b - sqrt(discriminant)) / a; //小根，物体上靠近光源点

if (temp < t\_max && temp > t\_min)

{

rec.t = temp;

rec.p = r.point\_at\_parameter(rec.t);

rec.normal = (rec.p - center) / radius;

return true;

}

temp = (-b + sqrt(discriminant)) / a; //大根，物体上相对远光源点

if (temp < t\_max && temp > t\_min)

{

rec.t = temp;

rec.p = r.point\_at\_parameter(rec.t);

rec.normal = (rec.p - center) / radius;

return true;

}

}

return false;

}

};

## 三、实现物体列表类

物体列表类hittable\_list同样继承自hittable，成员函数有一个物体虚基类数组的指针。实现了hit()函数，作用是遍历数组中的所有物体，记录目前当前射线命中的最近的球，然后返回是否命中。

class hittable\_list : public hittable{

public:

hittable \*\*list;

int list\_size;

hittable\_list() {}

hittable\_list(hittable \*\*l, int n)

{

list = l;

list\_size = n;

}

//如果命中了，命中记录保存到rec

virtual bool hit(const ray &r, float t\_min, float t\_max, hit\_record &rec) const

{

hit\_record temp\_rec;

bool hit\_anything = false;

double closest\_so\_far = t\_max; //记录目前最近的t值

for (int i = 0; i < list\_size; i++)

{

if (list[i]->hit(r, t\_min, closest\_so\_far, temp\_rec))

{

hit\_anything = true;

closest\_so\_far = temp\_rec.t;

rec = temp\_rec; //只记录打到的最近的球

}

}

return hit\_anything;

}

};

## 四、实现场景

### 4.1 多个球体布局及记录

hittable \*list[2]; //new

hittable \*world; //new

vec3 lower\_left\_corner(-2.0, -1.0, -1.0); //左下角

vec3 horizontal(4.0, 0.0, 0.0);

vec3 vertical(0.0, 2.0, 0.0);

vec3 origin(0.0, 0.0, 0.0); // 相机原点

void RayTracing(){

list[0] = new sphere(vec3(0, 0, -1), 0.5); //new

list[1] = new sphere(vec3(0, -100.5, -1), 100); //new

world = new hittable\_list(list, 2); //new

for (int j = ny - 1; j >= 0; j --)

{

for (int i = 0; i < nx; i++)

{

float u = float(i) / float(nx);

float v = float(j) / float(ny);

vec3 dir = lower\_left\_corner + u \* horizontal + v \* vertical - origin;

ray r(origin, dir);

vec3 col = color(r, world); //new

...

}

}

}

我们声明了一个名为world的变量，类型为hittable\_list，用于存放的场景中的所有球类。  
更新了之后，物体数组中除了上一节的小球，还添加一个半径为100的大球体，放在小球体的下面，且恰好跟小球相切，以突出球体顶部附近法向量坐标对应的颜色。

### 4.2 更新采样函数

更新之后，参数多了个world，也就是刚刚入口函数定义的一个hittable\_list变量。对一条射线执行world->hit()，从而将该射线命中的最近的物体信息，记录到rec中，进行相应的处理，目前的处理仍然为上一节的法向量可视化。

//发射一条射线，并采样该射线最终输出到屏幕的颜色值

vec3 color(const ray &r, hittable \*world){

hit\_record rec;

if (world->hit(r, 0.001, MAXFLOAT, rec)) //射线命中物体

{

return 0.5 \* vec3(rec.normal.x() + 1, rec.normal.y() + 1, rec.normal.z() + 1); //可视化RGB通道

}

else

{

vec3 unit\_direction = unit\_vector(r.direction());

float t = 0.5 \* (unit\_direction.y() + 1.0);

return (1.0 - t) \* vec3(1.0, 1.0, 1.0) + t \* vec3(0.5, 0.7, 1.0);

}

}

实现后的效果见开头图片。  
你也可以自行改动生成多个场景数据：



# 【光线追踪系列四】抗锯齿

# 【光线追踪系列五】漫反射(Diffuse)

# 【光线追踪系列六】反射与金属类特性

# 【光线追踪系列七】折射与电介质材质

# 【光线追踪系列八】动态相机、散焦模糊及最终光追效果

# 【光线追踪系列九】物体动态模糊

# 【光线追踪系列十】光追加速结构(BVH树)

本文主要参照 [Ray Tracing: The Next Week](https://raytracing.github.io/books/RayTracingTheNextWeek.html)，其中只是主要精炼光追相关理论，具体实现可参照原文。

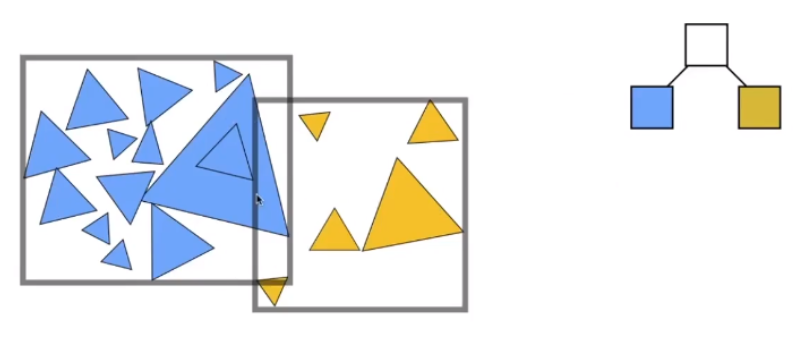
经过之前几部分的光追实现，我们大致可以实现了光追的效果，但其中有一个致命的bug就是：慢！！！本部分我们就来解决这个问题。  
首先我们分析下：射线和物体的相交检测，是光线追踪器的主要时间瓶颈，目前时间复杂度是线性的。有两种方法可以将时间复杂度降低到对数级别：一是空间划分，二是物体划分。本节将用BVH实现物体划分，对当前的项目进行加速。

## 一、层次包围体 BVH

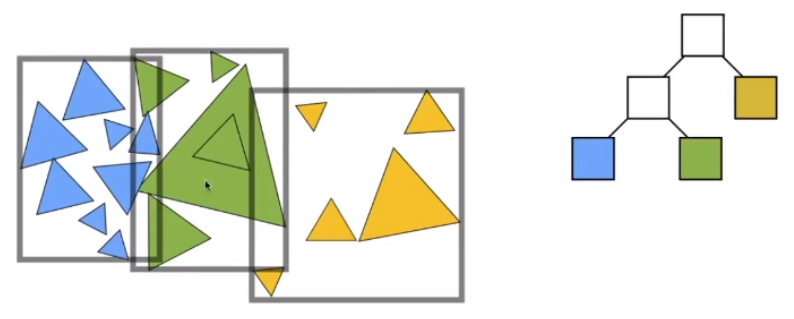
Bounding volume hierarchy (BVH)即层次包围体，在BVH中，所有的几何物体都会被包在bounding volume的叶子节点里面，bounding volume外面继续包着一个更大的bounding volume，递归地包裹下去，最终形成的根节点会包裹着整个场景。常用于碰撞检测和光线追踪的加速，能够将时间复杂度降低到关于物体数量的对数级别。

举例：找个volume包围10个物体，如果射线没击中这个volume，那么射线肯定没击中这10个物体。

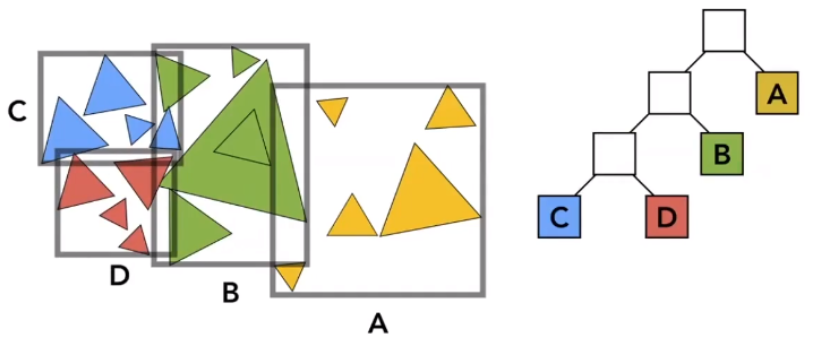
我们来看一下划分流程：  
步骤一：



步骤二：



步骤三：

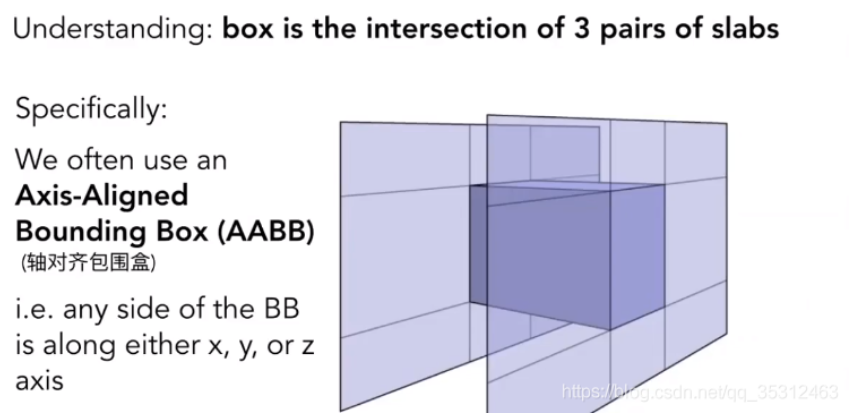


注意：把物体分成子集，可能会bounding volums重叠的情况。  
例如上图中用方框来包围物体，一堆物体最终被分成ABCD四组，这四组物体又被其之上根节点更大的方框包围。

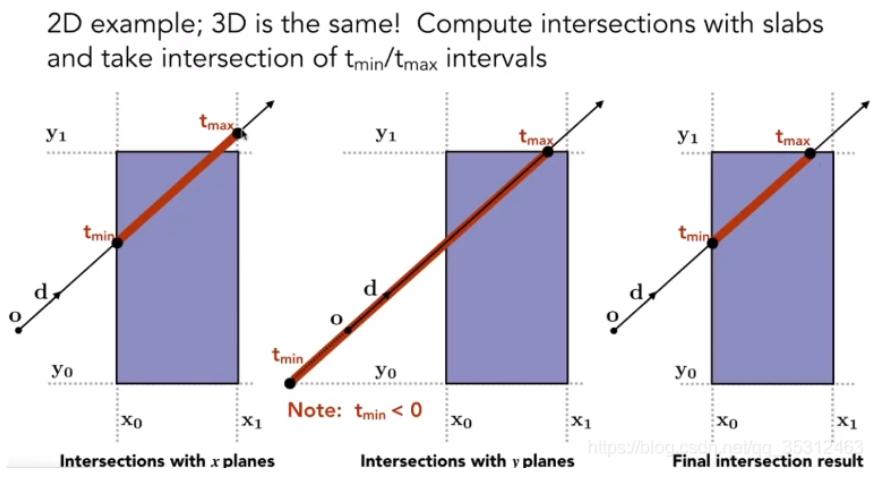
最常见的bounding volume就是AABB。

## 二、轴对齐包围盒 AABB

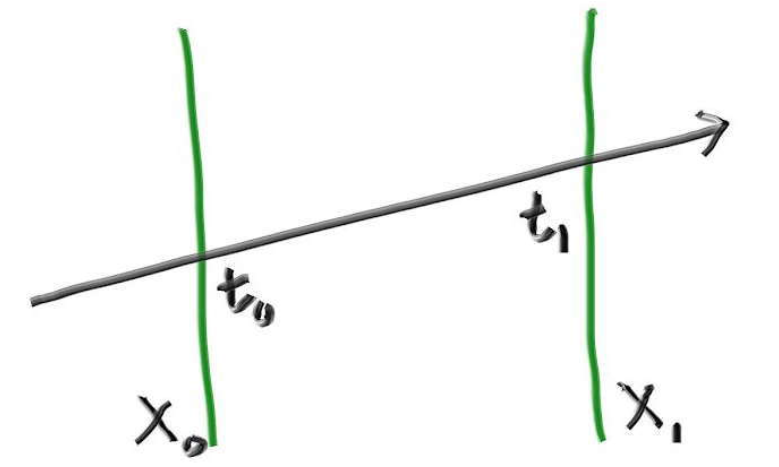
轴对齐包围盒：axis-aligned bounding box, 缩写为 AABB



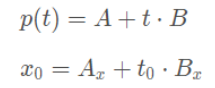
可用slab方法检测射线于AABB相交。



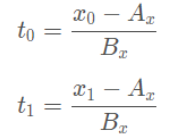
下边我们来看一下公式求解：



其中：



则x轴方向上的t区间为：



射线和AABB是否相交，可以转化为不同轴方向上的t区间是否重叠。如上图中的tmin与tmax之间的区间。

2D情况写成伪代码：

compute (tx0, tx1)

compute (ty0, ty1)

return overlap?( (tx0, tx1), (ty0, ty1))

3D情况写成伪代码：

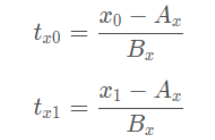
compute (tx0, tx1)

compute (ty0, ty1)

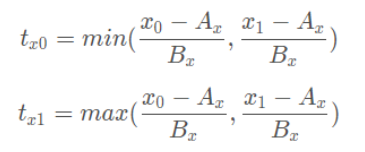
compute (tz0, tz1)

return overlap?( (tx0, tx1), (ty0, ty1), (tz0, tz1))

**注意2个问题：**  
问题1：  
1.射线沿着x轴的反方向发射，区间会反过来，例如(7,3)  
2.分母可能为0



Bx为0的情况，tx0和 tx1都会是正无穷或者负无穷，tx0和 tx1可以分别取两者之间的较小者和较大者。



问题2：  
Bx=0的同时，x0−Ax=0或 x1−Ax=0，这个问题稍后再处理。

假设没有反转区间，只需处理两对t区间是否重叠，伪代码如下：

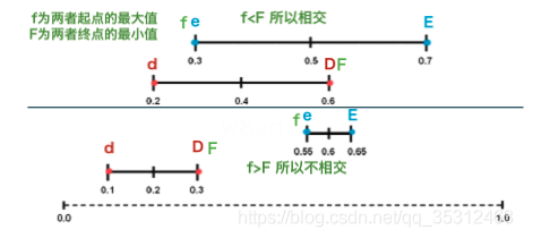
bool overlap(d, D, e, E){

f = max(d, e) //f为两者起点的最大值

F = min(D, E) //F为两者终点的最小值

return (f < F) //区间相交条件：f<F}

f为两者起点的最大值，F为两者终点的最小值，区间相交条件：f<F ( 即光线射入时只有最后一个方向进入才算进入，射出时只要有方向出去即为出区域 )。  
可参考下图理解：



3D的情况，则需要多判断一次，写成代码如下：

inline float ffmin(float a, float b) { return a < b ? a : b; }

inline float ffmax(float a, float b) { return a > b ? a : b; }

class aabb {

public:

vec3 \_min; //左下角顶点

vec3 \_max; //右上角顶点

aabb() {}

aabb(const vec3& min, const vec3& max) { \_min = min; \_max = max;}

vec3 min() const {return \_min; }

vec3 max() const {return \_max; }

//判断射线是否在t区间

bool hit(const ray &r, float tmin, float tmax) const

{

//xyz三个方向分别判断

for (int a = 0; a < 3; a++)

{

float invD = 1.0f / r.direction()[a];

float t0 = (min()[a] - r.origin()[a]) \* invD;

float t1 = (max()[a] - r.origin()[a]) \* invD;

if (invD < 0.0f)

std::swap(t0, t1);

tmax = t1 < tmax ? t1 : tmax; //F为两者终点的最小值

tmin = t0 > tmin ? t0 : tmin; //f为两者起点的最大值

if (tmax <= tmin) //F <= f

return false;

}

return true;

}

};

此外我们还需要计算2个box的包围盒：

/计算2个box的包围盒

aabb surrounding\_box(aabb box0, aabb box1){

vec3 small(ffmin(box0.min().x(), box1.min().x()),

ffmin(box0.min().y(), box1.min().y()),

ffmin(box0.min().z(), box1.min().z()));

vec3 big(ffmax(box0.max().x(), box1.max().x()),

ffmax(box0.max().y(), box1.max().y()),

ffmax(box0.max().z(), box1.max().z()));

return aabb(small, big);

}

用bounding\_box来设置物体的包围盒，目前场景中的物体只有sphere，但后续会新增其他物体（例如无边界的平面），这些物体不一定都会有包围盒，所以该函数返回bool值。对于动态模糊球体moving\_sphere则需要时间间隔参数来模拟相机快门曝光，所以参数里面带上t0和t1。  
在基类hittable中定义虚函数bounding\_box()，并在子类中分别实现：

class hittable {

public:

...

virtual bool bounding\_box(float t0, float t1, aabb& box) const = 0;};

bool moving\_sphere::bounding\_box(float t0, float t1, aabb &box) const{

aabb box0(center(t0) - vec3(radius, radius, radius), center(t0) + vec3(radius, radius, radius));

aabb box1(center(t1) - vec3(radius, radius, radius), center(t1) + vec3(radius, radius, radius));

box = surrounding\_box(box0, box1);

return true;}

bool sphere::bounding\_box(float t0, float t1, aabb &box) const{

box = aabb(center - vec3(radius, radius, radius), center + vec3(radius, radius, radius));

return true;}

bool hittable\_list::bounding\_box(float t0, float t1, aabb &box) const{

if (list\_size < 1)

return false;

aabb temp\_box;

bool first\_true = list[0]->bounding\_box(t0, t1, temp\_box);

if (!first\_true)

return false;

else

box = temp\_box;

for (int i = 1; i < list\_size; i++)

{

if (list[i]->bounding\_box(t0, t1, temp\_box))

{

box = surrounding\_box(box, temp\_box);

}

else

return false;

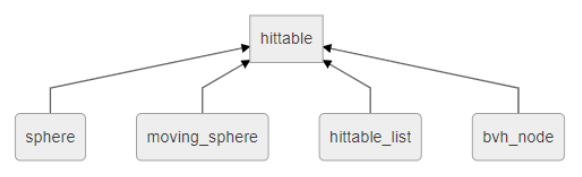
}

return true;

}

## 二、BVH树的实现

下面开始实现BVH树，定义BVH树的节点为bvh\_node类，继承自hittable。目前hittable子类的继承关系如下：



每个bvh\_node都有一个AABB和左右孩子指针，左右孩子同样指向hittable的指针，可能指向bvh\_node，也可能指向sphere或其他hittable子类。

hit()函数非常简单粗暴：检查当前节点的box是否命中，若是，则检查该节点的左右孩子（可能是bvh\_node或sphere）是否命中。

**对于所有的加速结构（包括BVH），最复杂的部分就是创建。使用自顶向下创建BVH树的方式如下**：

1. 随机选择x,y,z轴中的一个；
2. 对该轴正方向的顺序对物体排序，并分成前后两半；
3. 前一半物体放左子树，后一半物体放右子树，继续对左右子树递归创建。如果左右子树的物体数量等于2，直接左右节点分别指向物体。如果等于1，则左右节点均指向该物体。最后将当前节点的aabb设置为包含左右子树的box。

class bvh\_node : public hittable{

public:

bvh\_node() {}

hittable \*left;

hittable \*right;

aabb box;

//t0, t1为moving\_sphere中使用

bool bounding\_box(float t0, float t1, aabb &b) const

{

b = box;

return true;

}

bool hit(const ray &r, float t\_min, float t\_max, hit\_record &rec) const

{

if (box.hit(r, t\_min, t\_max))

{

hit\_record left\_rec, right\_rec;

bool hit\_left = left->hit(r, t\_min, t\_max, left\_rec);

bool hit\_right = right->hit(r, t\_min, t\_max, right\_rec);

if (hit\_left && hit\_right)

{

if (left\_rec.t < right\_rec.t)

rec = left\_rec;

else

rec = right\_rec;

return true;

}

else if (hit\_left)

{

rec = left\_rec;

return true;

}

else if (hit\_right)

{

rec = right\_rec;

return true;

}

else

return false;

}

else

return false;

}

bvh\_node(hittable \*\*l, int n, float time0, float time1)

{

int axis = int(3 \* random\_double());

if (axis == 0)

qsort(l, n, sizeof(hittable \*), box\_x\_compare);

else if (axis == 1)

qsort(l, n, sizeof(hittable \*), box\_y\_compare);

else

qsort(l, n, sizeof(hittable \*), box\_z\_compare);

if (n == 1)

{

left = right = l[0];

}

else if (n == 2)

{

left = l[0];

right = l[1];

}

else

{

left = new bvh\_node(l, n / 2, time0, time1);

right = new bvh\_node(l + n / 2, n - n / 2, time0, time1);

}

aabb box\_left, box\_right;

if (!left->bounding\_box(time0, time1, box\_left) ||

!right->bounding\_box(time0, time1, box\_right))

{

std::cerr << "no bounding box in bvh\_node constructor\n";

}

box = surrounding\_box(box\_left, box\_right);

}

};

x轴、y轴和z轴上的qsort()比较函数如下：

int box\_x\_compare(const void \* a, const void \* b) {

aabb box\_left, box\_right;

hittable \*ah = \*(hittable\*\*)a;

hittable \*bh = \*(hittable\*\*)b;

if (!ah->bounding\_box(0, 0, box\_left) || !bh->bounding\_box(0, 0, box\_right))

std::cerr << "no bounding box in bvh\_node constructor\n";

if (box\_left.min().x() - box\_right.min().x() < 0.0)

return -1;

else

return 1;}

int box\_y\_compare(const void \* a, const void \* b) {

aabb box\_left, box\_right;

hittable \*ah = \*(hittable\*\*)a;

hittable \*bh = \*(hittable\*\*)b;

if (!ah->bounding\_box(0, 0, box\_left) || !bh->bounding\_box(0, 0, box\_right))

std::cerr << "no bounding box in bvh\_node constructor\n";

if (box\_left.min().y() - box\_right.min().y() < 0.0)

return -1;

else

return 1;}

int box\_z\_compare(const void \* a, const void \* b) {

aabb box\_left, box\_right;

hittable \*ah = \*(hittable\*\*)a;

hittable \*bh = \*(hittable\*\*)b;

if (!ah->bounding\_box(0, 0, box\_left) || !bh->bounding\_box(0, 0, box\_right))

std::cerr << "no bounding box in bvh\_node constructor\n";

if (box\_left.min().z() - box\_right.min().z() < 0.0)

return -1;

else

return 1;

}

修改random\_scene的返回值：

hittable \*random\_scene() {

...

// return new hittable\_list(list,i);

return new bvh\_node(list, i, 0.0, 1.0);

}

对比显示，使用BVH后，生成速度提升一倍左右。