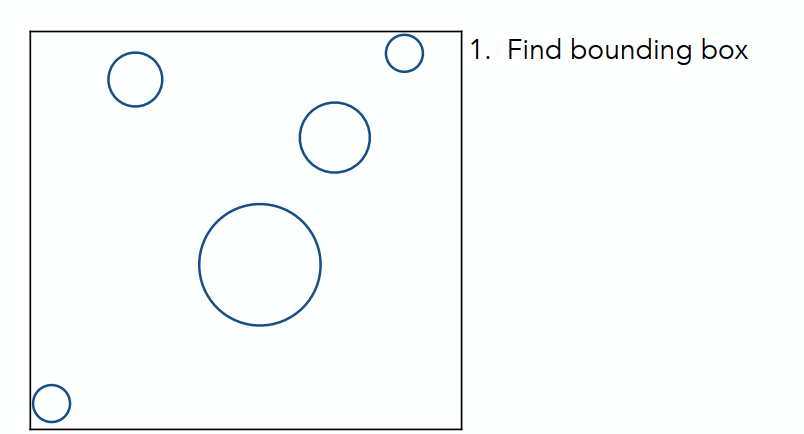
在上节课的最后我们已经讲清楚了如何用AABB包围盒与光线求交。在这节课我们就继续上节课的思路，去探索如何加快包围盒与光线的求交速度

# (一)Uniform Spatial Partitions (Grids)

## 1、Preprocess – Build Acceleration Grid

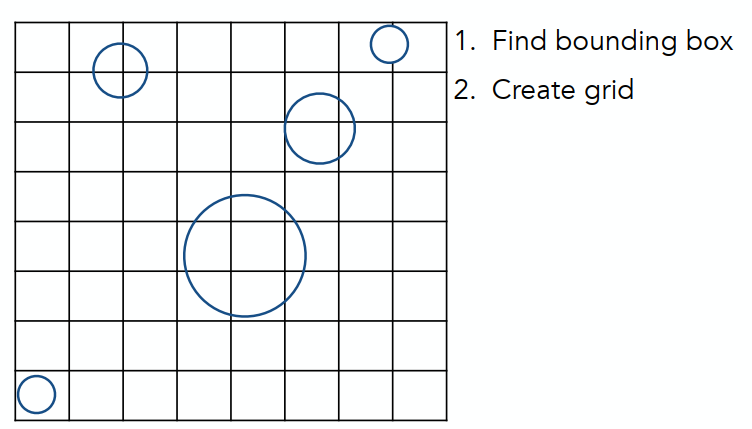
### ①. Find bounding box

找到这个包围盒



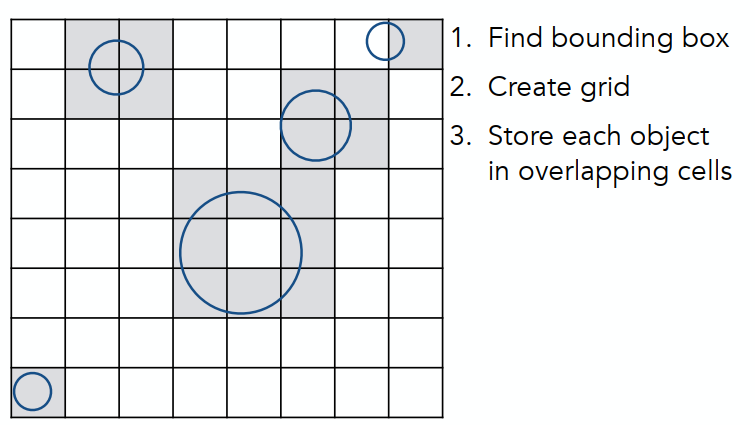
### ②. Create grid

把场景划分成很多格子（在二维理解的话是格子）



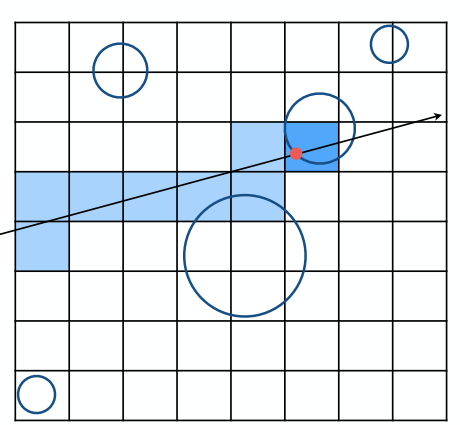
### ③.Store each object in overlapping cells

把与物体相交的格子找出来并标记（如上图灰色的格子）



## 2、Ray-Scene Intersection

在这里我们假设光线与格子求交非常快，与物体求交非常慢。

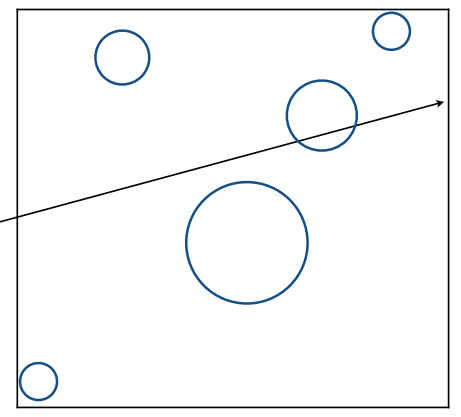


当光线在格子中走（如上图所有蓝色格子），如果光线经过的格子中有物体，这说明光线有可能与物体相交（并不意味着一定相交，有可能与格子相交，但与物体不相交），那么就做一次光线与物体的求交。当判定光线与物体相交时（如上图中深蓝色格子），问题就得到了解决。

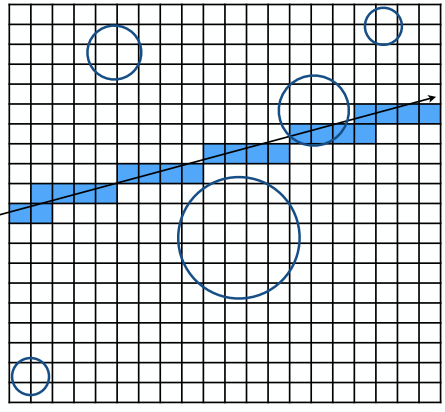
这里的加速原理就是由于光线与包围盒的求交速度比光线与物体求交速度快，因此我们多做一些光线与包围盒的求交，减少与物体的求交次数。

## 3、Grid Resolution?

那么这种方法的加速效果怎么样？

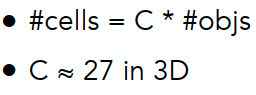


如果划分的包围盒少的话（如上图就划分为一个），则实际上没有太大效果。



如果划分的包围盒数很多，检测次数变多，效率也不会很高。

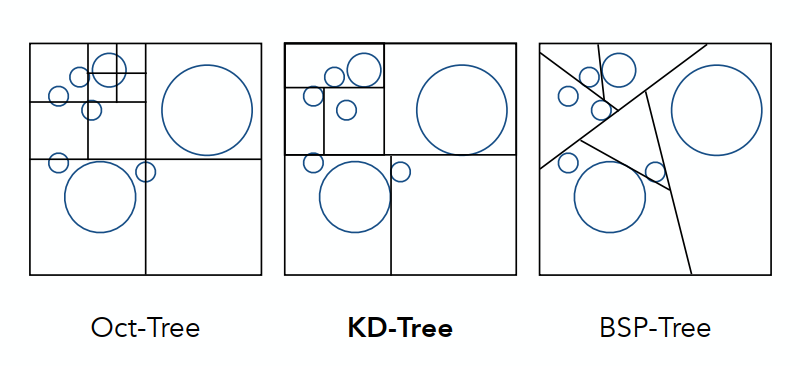
经过人们长期实践，当满足下述式子的时候，效果最好：划分的格子数=一个常数\*物体数，这个常数为27。



当场景中物体的分布比较平均时，用这种划分格子的方法比较好，但是对于分布不均匀的场景，这种方法就不是那么的理想

# (二)Spatial Partitions

## Spatial Partitioning Examples



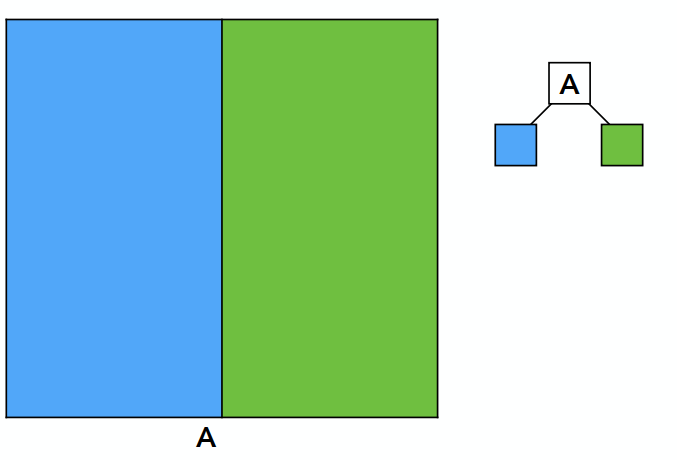
Oct-Tree：八叉树，把空间中一个大的包围盒切成八份（横竖平三刀切成八份），对于每一个子节点都再切成八块，当每个盒子中有足够少数量的物体则停止切割。但是维度高不好计算。

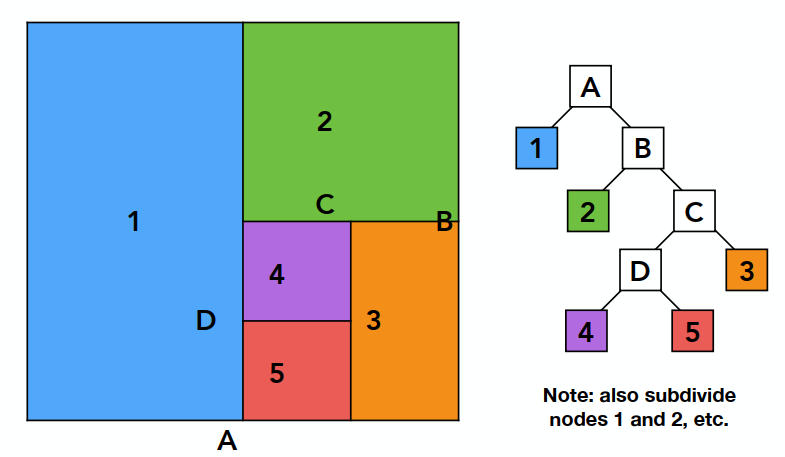
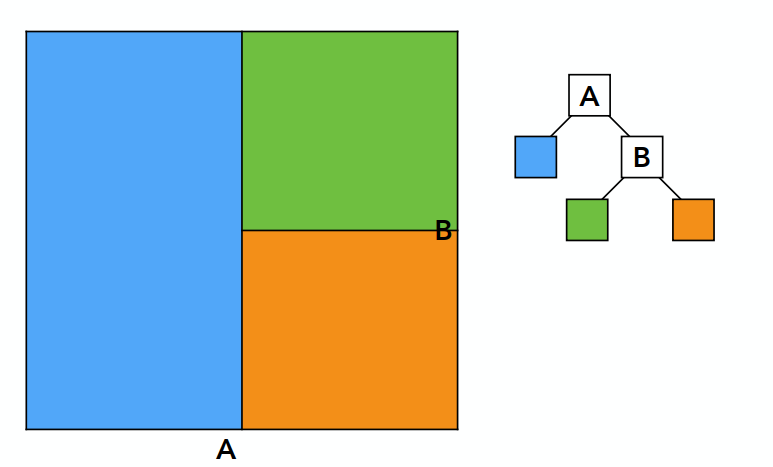
KD-Tree：与八叉树类似，但是每次只沿某个轴砍一刀。一般是三个方向交替切割。

BSP-Tree：二分法。每次选一个方向把节点砍开，由于不是横平竖直，不好算。而且维度高不好计算

这里我们主要讨论KD-Tree。

## KD-Tree Pre-Processing





求KD树的过程在做光线追踪前先把场景划分好，再进行下面的操作。如上图（通过降维在二维上解释和理解），每一次只切一刀。第一次沿竖直方向砍A区域一刀，得到了左侧蓝色区域和右侧B区域（绿色部分），第二次水平方向砍B区域一刀（蓝色区域也应该被砍一刀，但是这里为了举例子只处理了B区域绿色部分，往后的操作也是，实际上都要砍开）得到了C区域，第三次再竖直方向砍C区域一刀…往后以此类推。

这样划分完以后，只需要在叶子结点存储多边形信息即可。

### Data Structure for KD-Trees

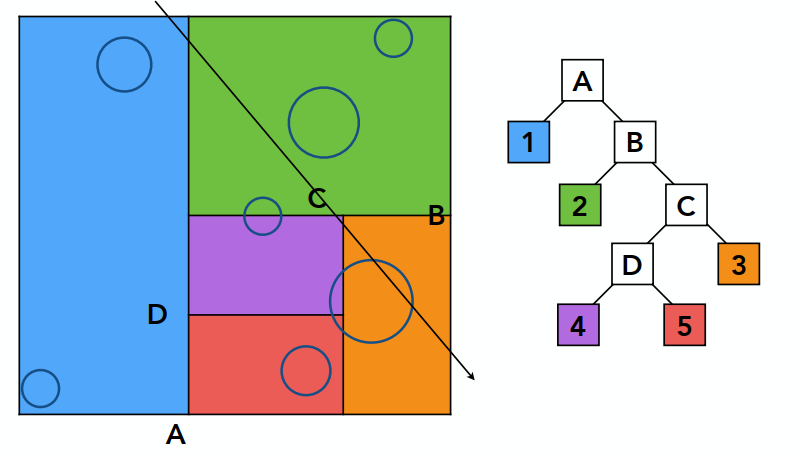
KD树的数据结构：

划分方向:沿X Y Z哪个轴

划分位置:不一定非要在中间划分

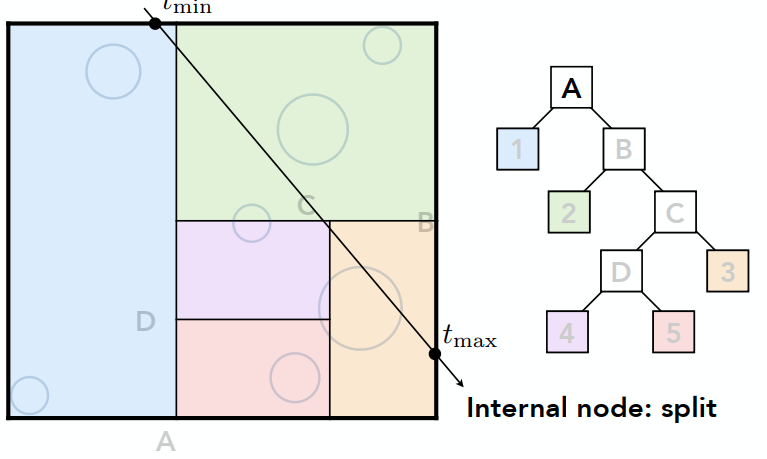
中间节点必有子结点，实际上的图元只存在于叶子节点上

## Traversing a KD-Tree(KD树的建立)

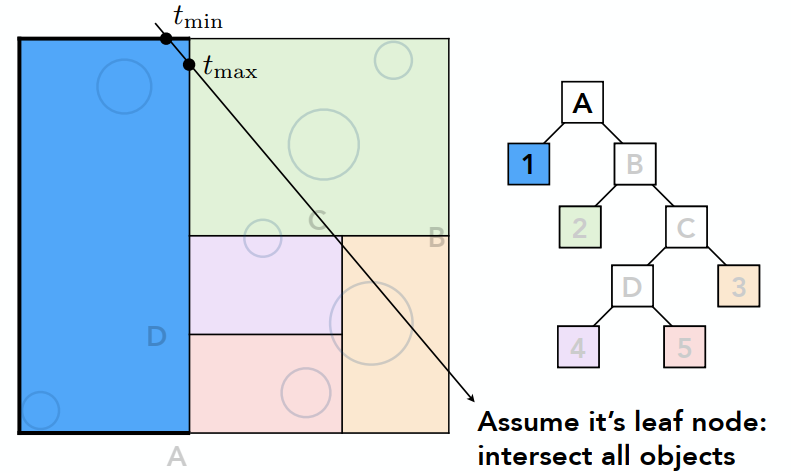


在光线进入之前，就需要把KD树建立好，那么应该如何建立呢？

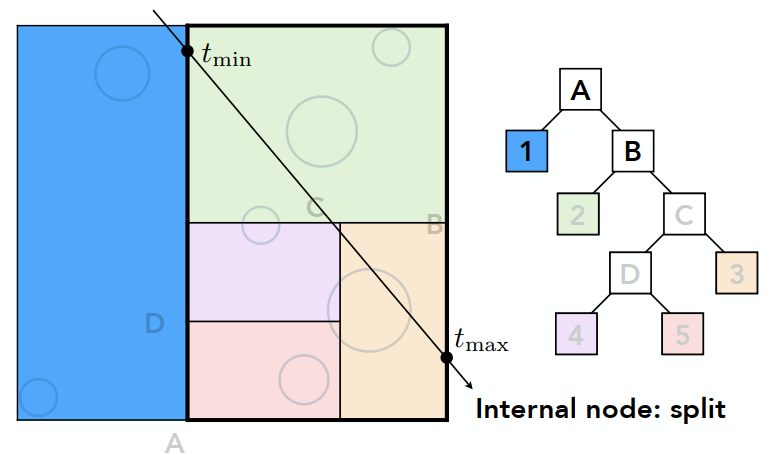
光线进入，首先判定是否和A(下图黑框)有交点(很明显存在交点)，继而考虑是否和A的子节点有交点（A的子节点是1和B）



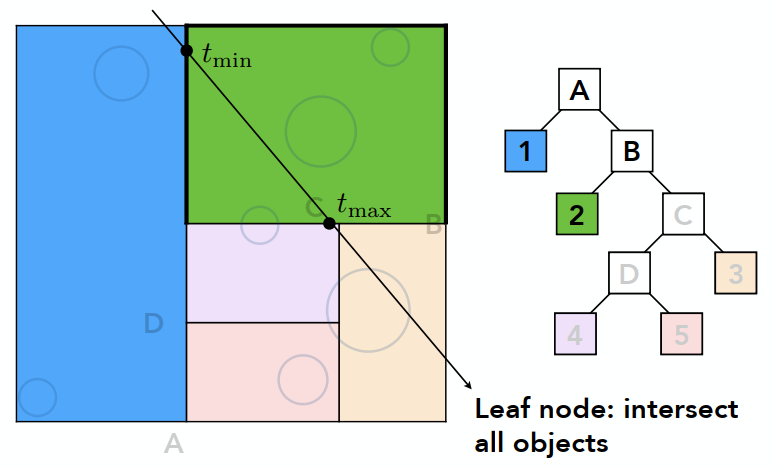
发现和蓝色区域（1）有交点（假设这里蓝色区域1不再划分，就是个叶子结点），那么由于已经找到了叶子结点，光线就要与这个叶子结点代表的AABB盒子中所包围的所有物体求交



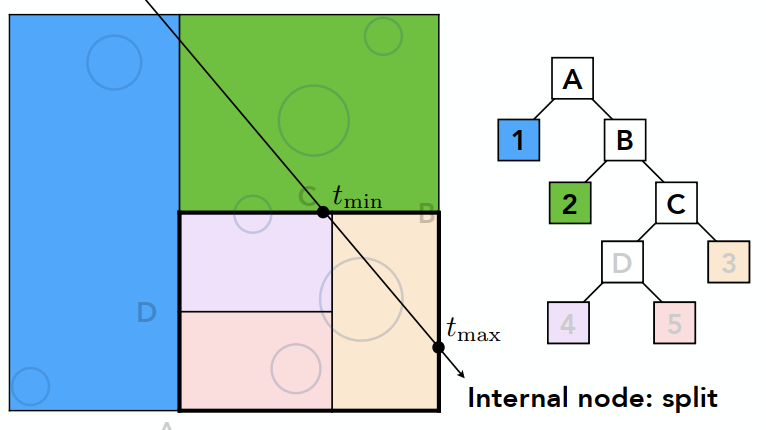
另一个需要判断的子节点就是B节点(黑框)，我们发现光线和B节点也有交点，那么就要考虑是否和B的子节点有交点（B的子节点是2和C）



发现和绿色区域（2）有交点（假设这里绿色区域2不再划分，就是个叶子结点），那么由于已经找到了叶子结点，光线就要与这个叶子结点代表的AABB盒子中所包围的所有物体求交

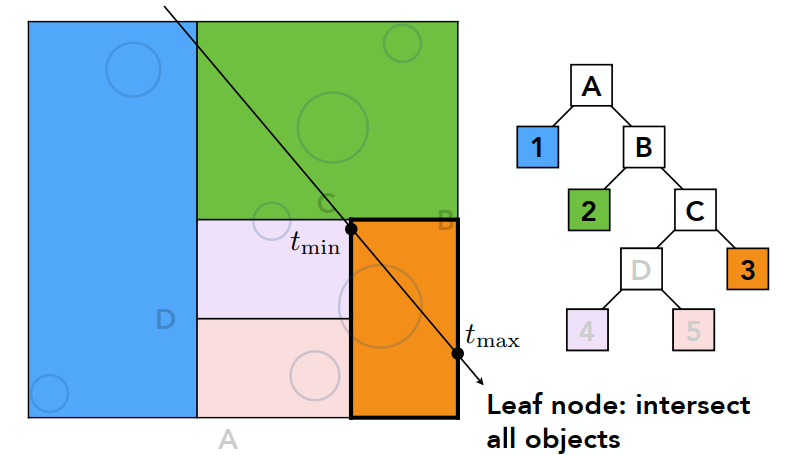


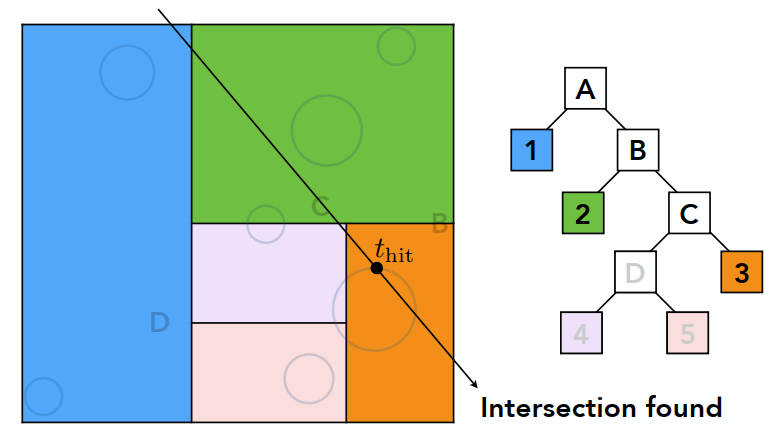
另一个需要判断的子节点就是C节点，我们发现光线和C节点也有交点，那么就要考虑是否和C的子节点有交点（C的子节点是3和D）



…

最终会找完所有节点，完成求交。





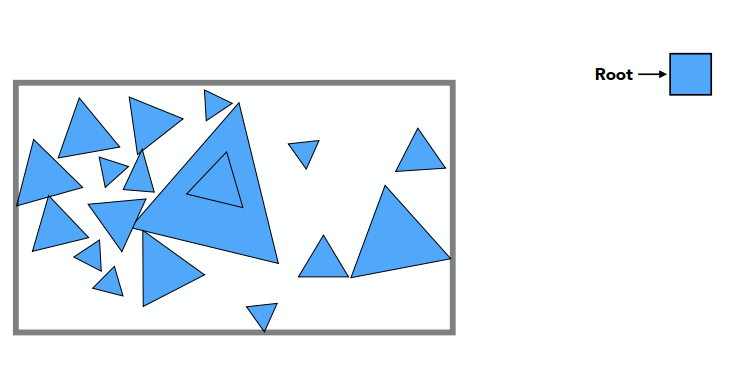
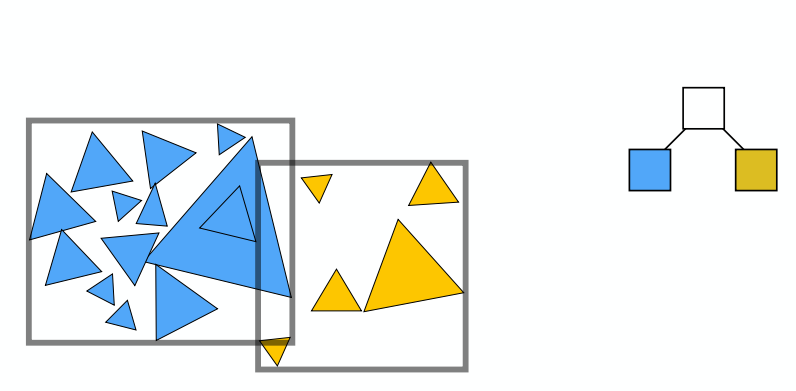
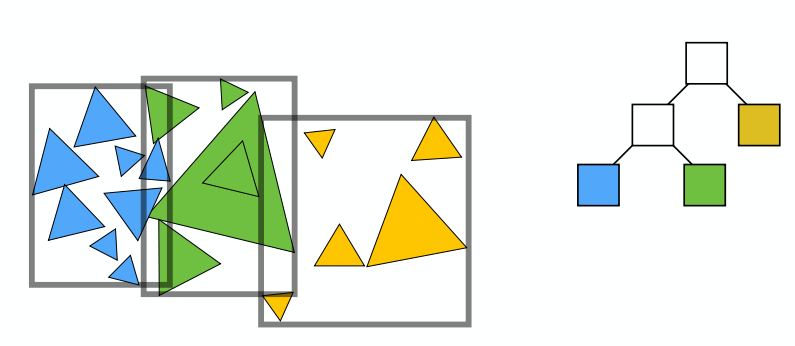
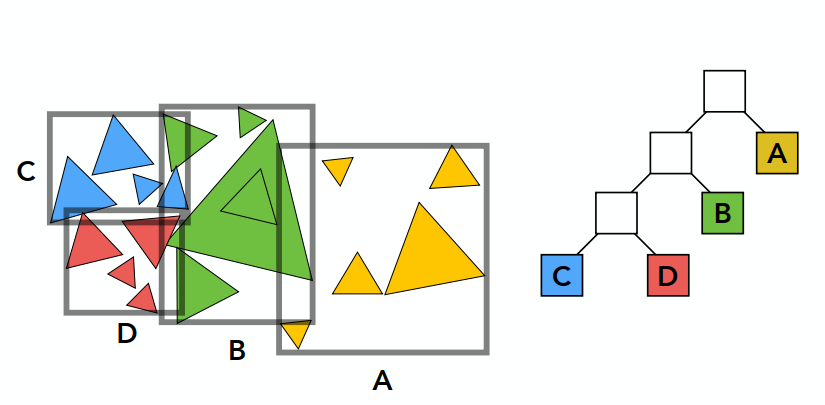
对于KD-Tree，有一个明显的问题，如果对于场景来说（假设全是三角形构成），那么通过空间划分的方式，就会产生一个问题，给定一个包围盒，我要知道它与哪些物体（三角形）有交集。这个判定是很复杂的。另一个问题就是，对于一个物体，可能与很多包围盒有交集，那么每个包围盒中都要存这个物体。一个物体可能出现在多个叶子结点中。

# (三)Object Partitions & Bounding Volume Hierarchy (BVH)

BVH不是按空间划分包围盒，而是按物体划分。

每次把一个盒子中的物体划分成两部分，每部分都重新求包围盒。

这样一个物体就只会出现在一个包围盒中。而且省去了物体中多边形与包围盒求交的问题。（完美解决了KD-Tree的弊端）

由于BVH是对物体划分包围盒，所以空间上允许重叠，不一定划分的很开。

## Building BVHs流程

Find bounding box找到包围盒

Recursively split set of objects in two subsets递归地将目标划分为两个群体

Recompute the bounding box of the subsets重新计算两个群体的包围盒

Stop when necessary包围盒足够小，比如每个包围盒只有一个目标时，结束递归

Store objects in each leaf node将目标存储在叶子结点上

Tip1:为了让包围盒（节点）在空间中均匀分布，每一次都只沿着最长的轴将其划分。（比如三角形横向排列，这时候就需要按横向进行划分）

Tip2:取一个包围盒中间的物体（这个中间指的是数量上的中间，是物体数的中位数所代表的物体），以这个物体为分界线将这一堆物体划分成两部分。（划分的两边的物体数量差不多，树更平衡，搜索的次数更少）（涉及到排序问题，但是目的是要快速找出中位数，可以用快速选择算法（时间复杂度仅为O(n) ） ）

## Termination criteria?终止条件

Heuristic: stop when node contains few elements (e.g. 5)

当一个包围盒中物体足够少时（例如5个），就不再划分。如果场景变化，则需要重新计算整个过程。

## Data Structure for BVHs

中间节点存储：

--Bounding box

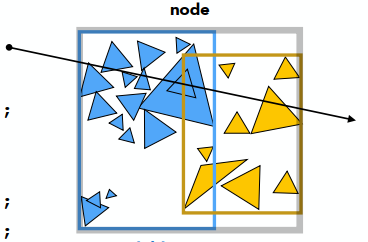
--Children: pointers to child nodes

叶子结点存储：

--Bounding box

--List of objects物体

## BVH Traversal(遍历)



Intersect(Ray ray, BVH node) {

Intersect(Ray ray, BVH node) {

 if (ray misses node.bbox) //光线与所有BVH不相交

     return;

 if (node is a leaf node){//相交 要与该包围盒内的所有叶子结点求交

     test intersection with all objs;

    return closest intersection;}//返回最近的那个

//相交但是不与叶子结点相交，递归计算左右子节点

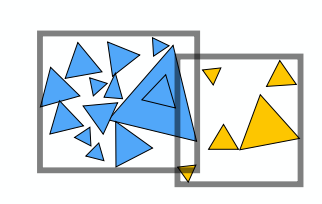
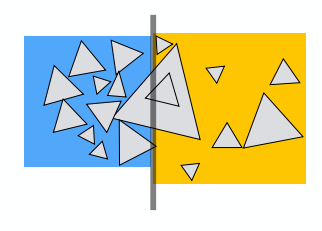
 hit1 = Intersect(ray, node.child1);

 hit2 = Intersect(ray, node.child2);

 return the closer of hit1, hit2;

}

## Spatial vs Object Partitions



空间分区（例如 KD 树）：将空间划分为不重叠的区域 一个对象可以包含在多个区域中

对象分区（例如 BVH）：将对象集划分为不相交的子集 每个集合的边界框在空间中可能重叠