



# Lecture 14: 几何查询加速

SSE315: 计算机图形学  
Computer Graphics

---

陈壮彬

软件工程学院

chenzhb36@mail.sysu.edu.cn

# Course roadmap

## 光栅化 Rasterization

计算机图形学介绍  
基于采样的光栅化  
空间变换  
纹理映射、深度和透明度

## 几何 Geometry

几何介绍  
曲线与曲面  
几何处理

## 材质与光线 Materials and Lighting

几何查询  
几何查询加速

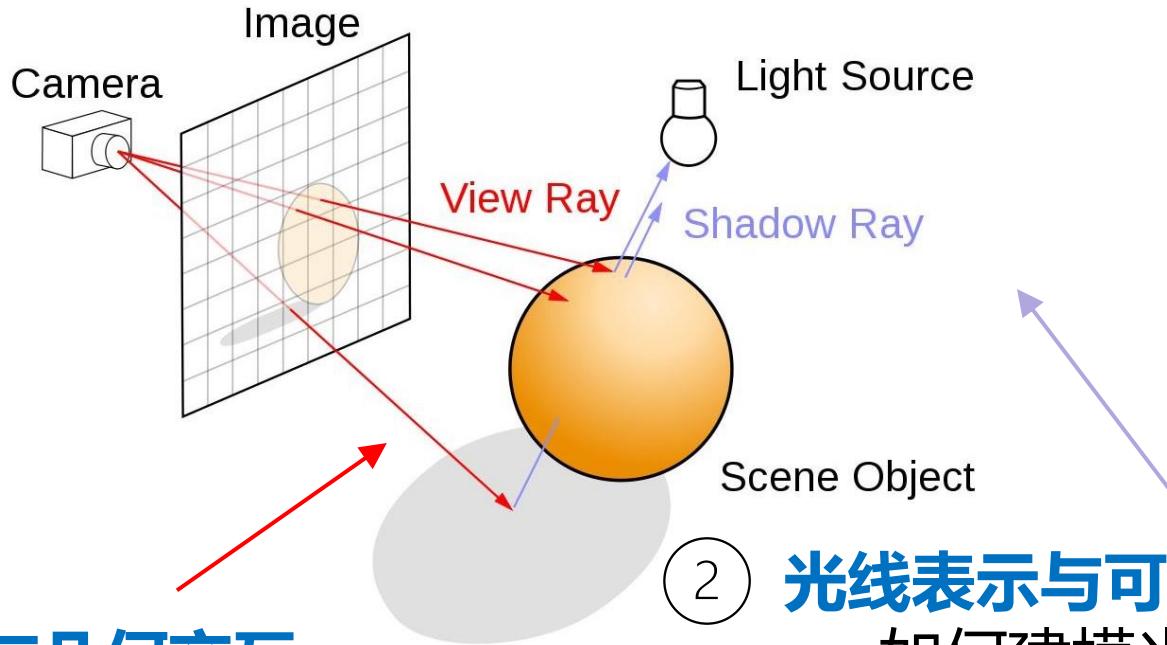
## 动画 Animation

几何查询  
几何查询加速



# 光线追踪 Ray tracing

分成 **光线与几何交互** 和 **光线表示与可视化** 两部分



## ① 光线与几何交互

- 查找光线照射在几何中的位置
- 如何加速上述过程

看到哪里

## ② 光线表示与可视化

- 如何建模光线
- 光线与物体的相互作用，包括反射、折射和散射等
- 全局光照

看到什么

# Today's topics

□ 几何查询的复杂性

□ 几种用于加速几何查询的数据结构

- 层次包围盒
- 均匀空间划分
- KD 树

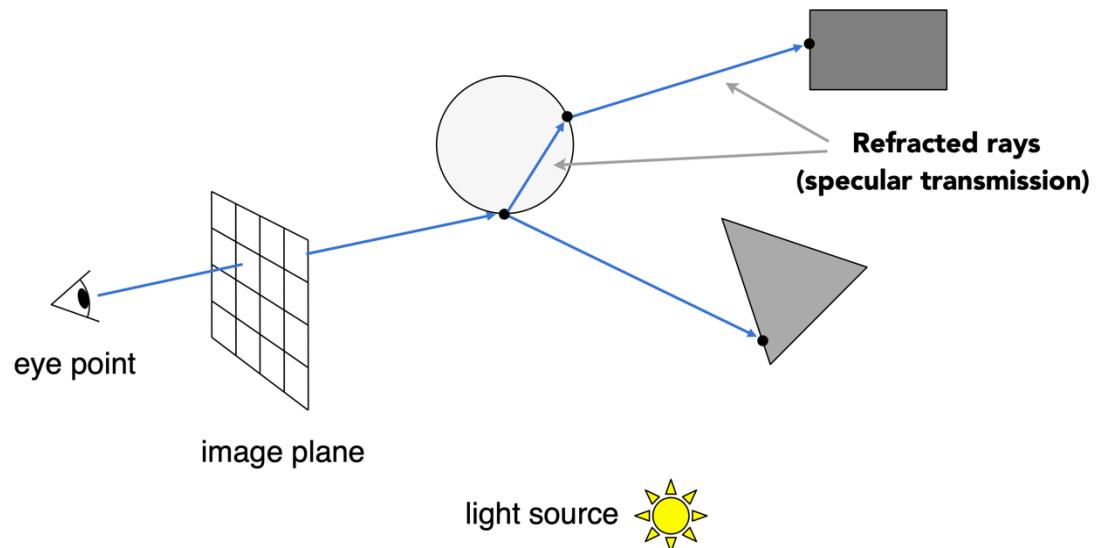
# 光线追踪 – 性能挑战

## 口简单的光线-网格交互

- 需要计算光线与所有三角形的交互
- 判断是否交叉、计算最近点

## 口挑战

- 朴素算法 (naïve algorithm)
  - 时间复杂度 = #pixels \* #triangles (\* #bounces)
- 非常慢且低效!



# 光线追踪 – 性能挑战



Jun Yan, Tracy Renderer

San Miguel Scene, 10.7M triangles

# 光线追踪 – 性能挑战



Deussen et al; Pharr & Humphreys, PBRT

**Plant Ecosystem, 20M triangles**

# 回顾：光线与三角形交叉

首先找到光线与三角形所在平面的交点

- 光线公式为

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$$

- 将光线公式带入到平面公式

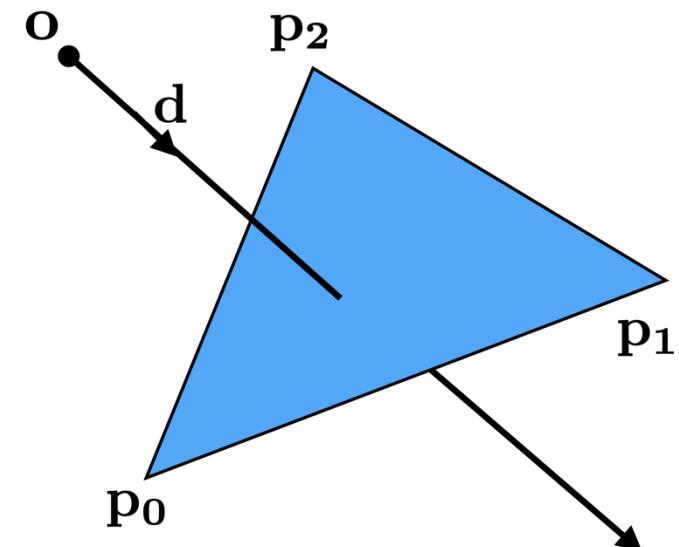
$$\mathbf{N}^T \mathbf{x} = c \quad \mathbf{N}^T \mathbf{r}(t) = c$$

$$\mathbf{N}^T (\mathbf{o} + t\mathbf{d}) = c$$

- 求解  $t$ :

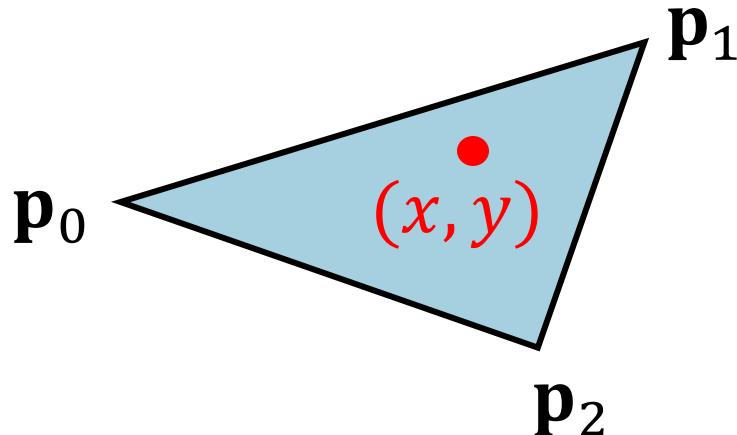
$$t = \frac{c - \mathbf{N}^T \mathbf{o}}{\mathbf{N}^T \mathbf{d}} \quad \mathbf{r}(t) = \mathbf{o} + \frac{c - \mathbf{N}^T \mathbf{o}}{\mathbf{N}^T \mathbf{d}} \mathbf{d}$$

然后判断交点是否在三角形内部



# 光线与三角形交叉 – 更快的算法

□ 利用重心坐标表示三角形内的点



$$(x, y) = w\mathbf{p}_0 + u\mathbf{p}_1 + v\mathbf{p}_2$$

$$w + u + v = 1$$

$$w = 1 - u - v$$

$$f(t) = (1 - u - v)\mathbf{p}_0 + u\mathbf{p}_1 + v\mathbf{p}_2$$

□ 变换一下位置

$$f(t) = \mathbf{p}_0 + u(\mathbf{p}_1 - \mathbf{p}_0) + v(\mathbf{p}_2 - \mathbf{p}_0)$$

# 光线与三角形交叉 – 更快的算法

□ 把光线公式代入到重心坐标表示的三角形的点

$$\mathbf{o} + t\mathbf{d} = (1 - u - v)\mathbf{p}_0 + u\mathbf{p}_1 + v\mathbf{p}_2$$

□ 求解  $u, v, t$

$$[\mathbf{p}_1 - \mathbf{p}_0 \quad \mathbf{p}_2 - \mathbf{p}_0 \quad -\mathbf{d}] \begin{bmatrix} u \\ v \\ t \end{bmatrix} = \mathbf{o} - \mathbf{p}_0$$

$\underbrace{\hspace{10em}}$

**M**

$$\Rightarrow \begin{bmatrix} u \\ v \\ t \end{bmatrix} = \mathbf{M}^{-1}(\mathbf{o} - \mathbf{p}_0)$$

□ 怎么判断交点在三角形内部?

- 检查  $u, v, 1 - u - v$  是否均大于 0

# 光线与网格相交

□ 给定一个包含  $N$  个图像基元的几何网格与一条光线  $\mathbf{r}(t)$ ，  
找到光线首个击中的点

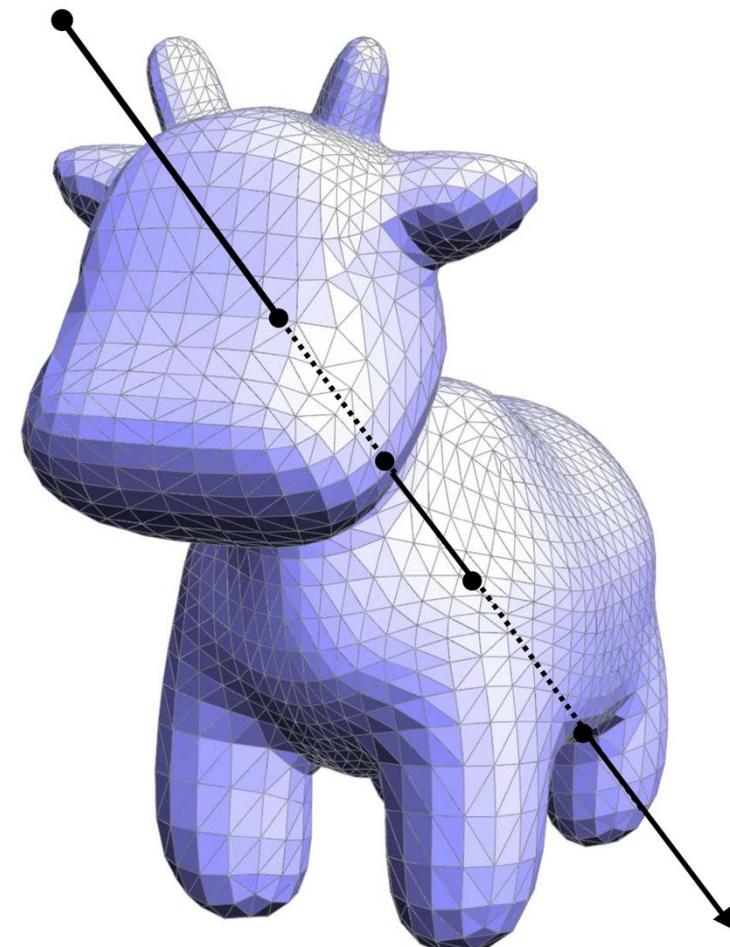
□ 最简单的算法？

- 计算光线与每一个三角形（可能）的交点
- 更新最近的交点

□ 时间复杂度

- $O(N)$

□ 是否有更好的算法？



包围盒  
Bounding box/volume

# 包围盒

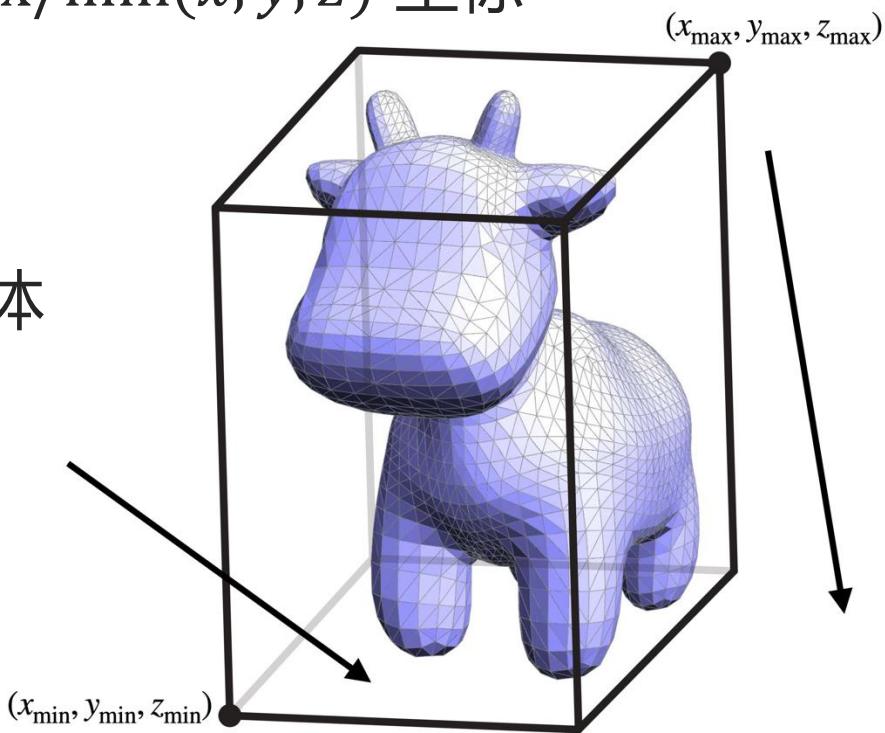
口一种快速(初步)判断交叉的方法：用最小的立方体包围复杂的几何物体(包含其所有图像基元)

- Q: 怎么构建包围盒?
- A: 循环所有的点，记录  $\max/\min(x, y, z)$  坐标
- Complexity:  $O(N)$

口检查光线是否与包围盒相交

- 若没有，则一定没有击中物体
- 若有，**检查所有三角形**找到相交的点(也可能都不相交)

避免每一条光线都要检查所有图像基元！



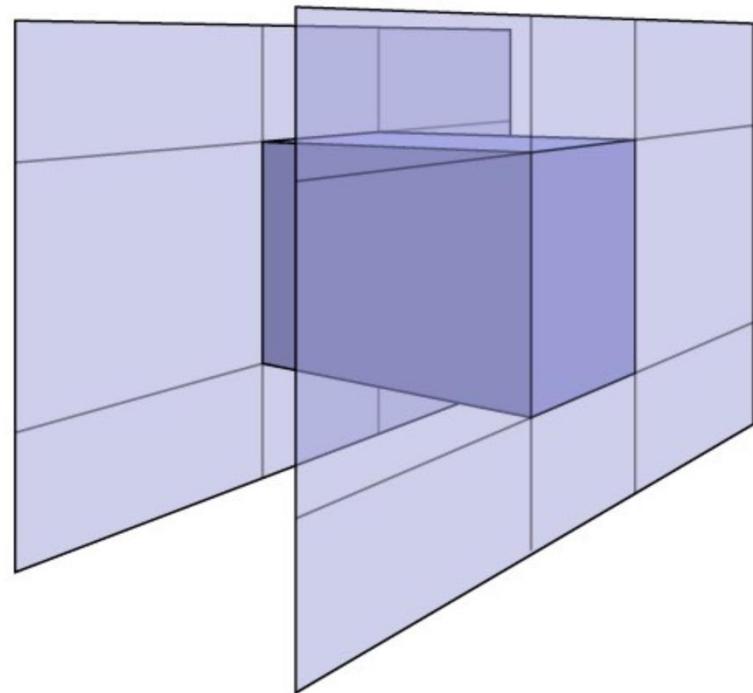
# 光线与包围盒交叉

□ 如何理解包围盒？

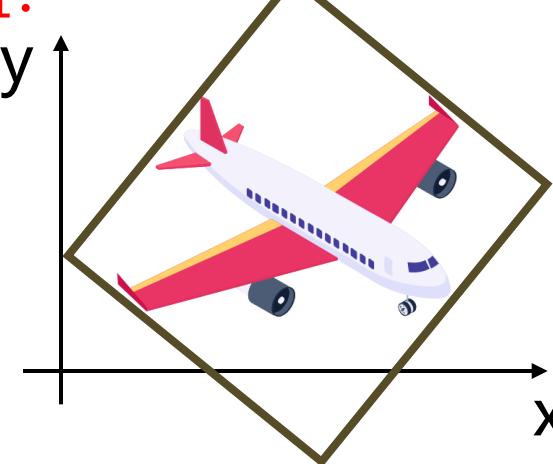
- 是三对无限平面的交集

□ 通常选择轴对齐包围盒 (Axis-Aligned Bounding Box, AABB)

- 即包围盒的任意一面都沿着相应的  $x, y, z$  轴
- **为什么选择轴对齐的包围盒？**



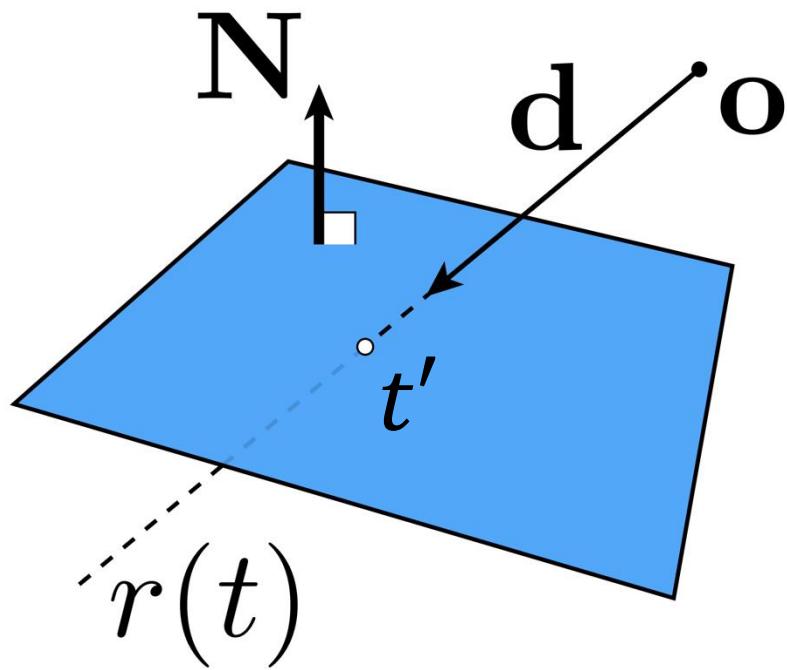
轴对齐包围盒



非轴对齐包围盒

# 为什么用轴对齐的包围盒？

□一般情况下，光线与平面的交点

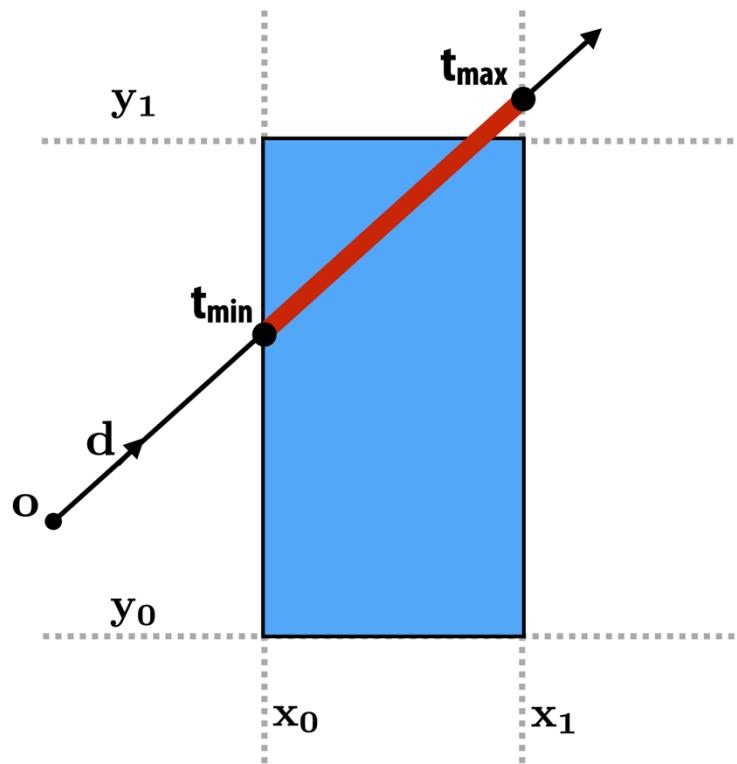


$$t' = \frac{c - \mathbf{N}^T \mathbf{o}}{\mathbf{N}^T \mathbf{d}}$$

# 为什么用轴对齐的包围盒？

## □ 光线与轴对齐平面的交点

- 考虑 2D 中垂直于  $x$  轴的平面      单位法向量为  $N^T = [1, 0]^T$ !



□ 大大简化了计算！

求光线与平面  $x = x_0$  的  
交点，即  $t_{min}$

$$c = x_0, N^T = [1, 0]^T$$

$$t = \frac{c - \mathbf{N}^T \mathbf{o}}{\mathbf{N}^T \mathbf{d}}$$

1 次减法  
4 次乘法  
1 次除法

$$t = \frac{x_0 - \mathbf{o}_x}{\mathbf{d}_x}$$

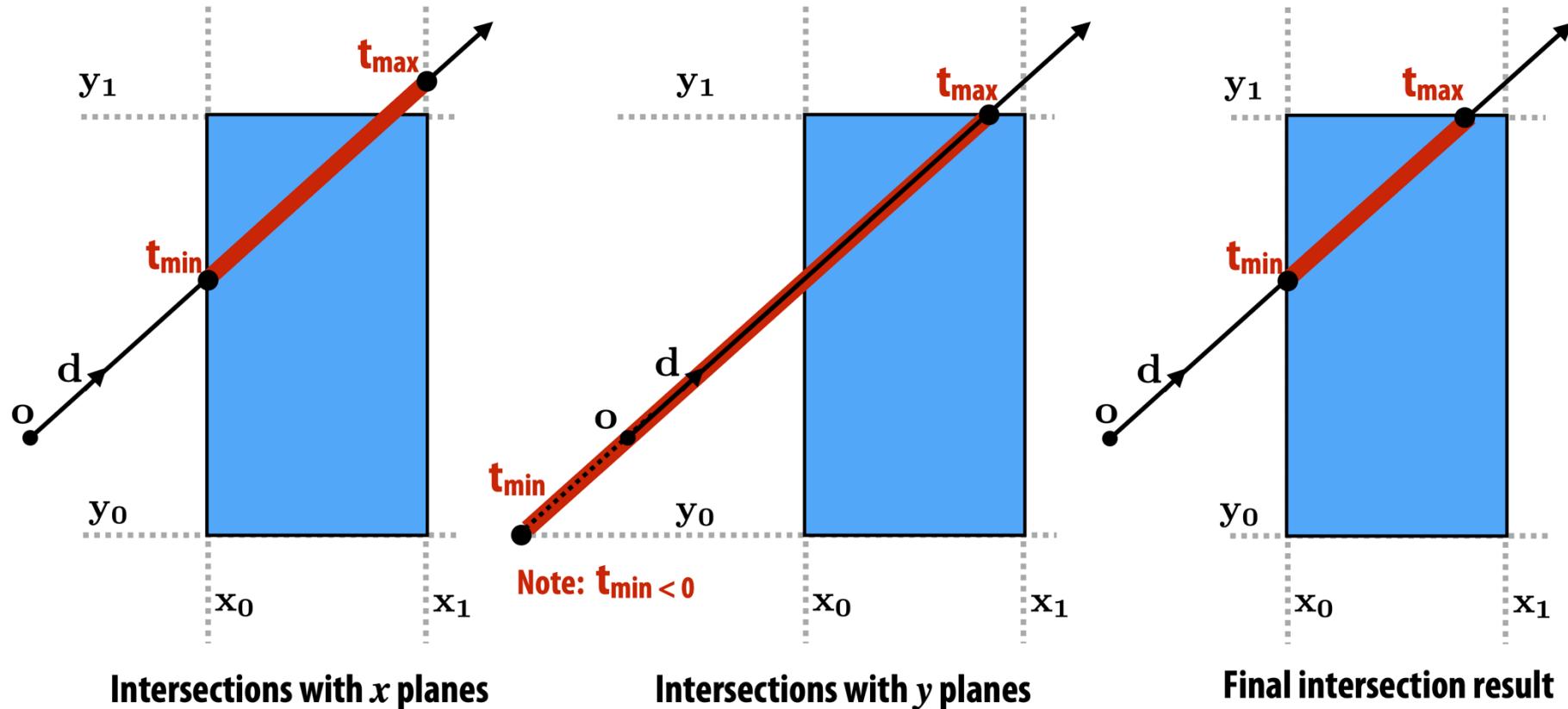
1 次减法  
1 次除法

# 光线与轴对齐包围盒相交

□怎么判断光线与盒子相交?

□2D 的例子 (3D 的情况也一样)

□计算与所有平面对的交点，并取  $t_{min}/t_{max}$  的交集



# 光线与轴对齐包围盒相交

□ 回忆一下 3D 盒子是由三对平面组成的

□ 算法关键

- 光线只有在进入所有平面对时才进入盒子
- 只要光线离开任一平面对，就会离开了盒子

□ 对每一对平面，计算  $t_{min}$  和  $t_{max}$

- 负数也没关系
- 平行于平面则得到  $-\infty$  或  $+\infty$

□ 对于 3D 盒子，令  $t_{enter} = \max\{t_{min}\}$ ,  $t_{exit} = \min\{t_{max}\}$

□  $t_{enter} < t_{exit}$  表明光线在盒子里待了一段时间 (即二者相交)

□ 是吗？

# 光线与轴对齐包围盒相交

□ 要注意光线不是直线，而是射线

- 因此需要检查时间  $t$  的正负性确保正确的物理意义

□  $t_{exit} < 0$  代表什么？

- 表明盒子在光线的后面，因此二者没有相交

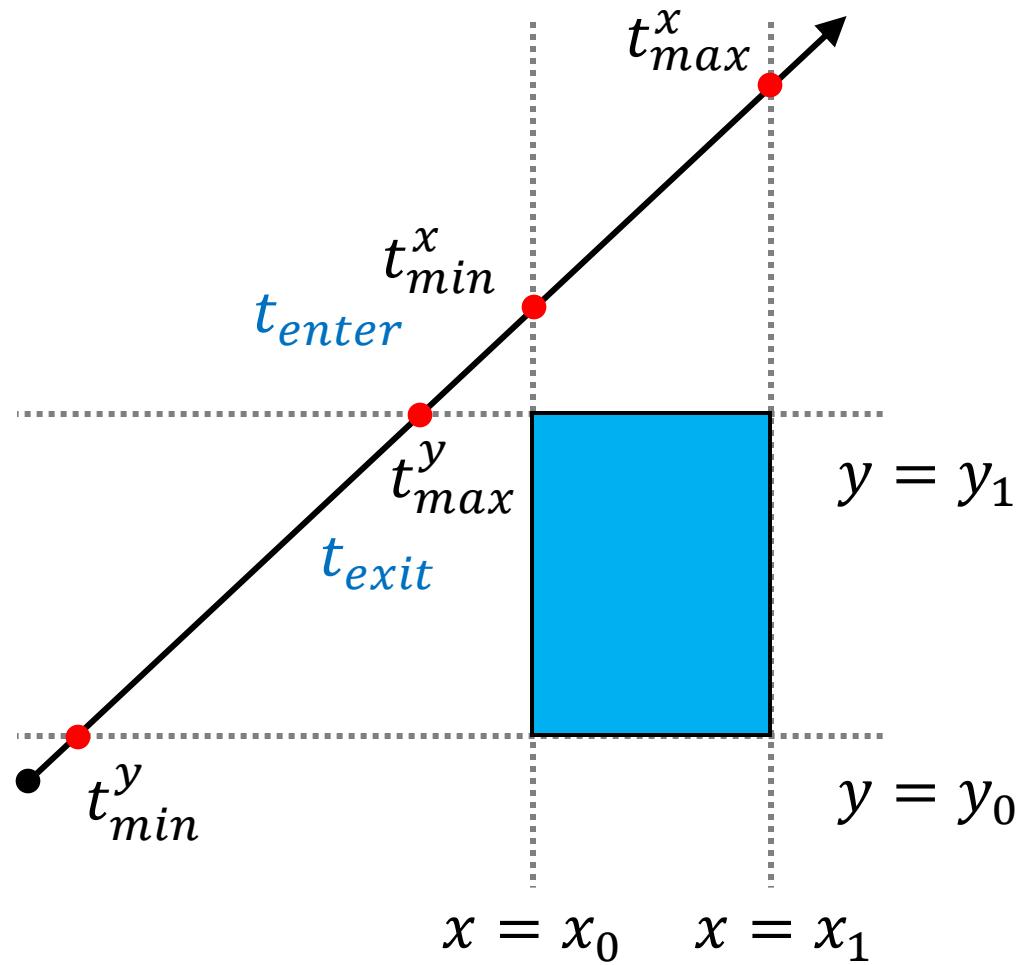
□  $t_{exit} \geq 0$  and  $t_{enter} < 0$  代表什么？

- 表明光线在盒子内部，因此二者相交

□ 因此，光线与盒子相交，当且仅当

- $t_{enter} < t_{exit}$  and  $t_{exit} \geq 0$

# 判断光线是否与盒子相交



$$t_{min}^y < t_{max}^y < t_{min}^x < t_{max}^x$$

$$\begin{aligned} t_{enter} &= \max\{t_{min}^x, t_{min}^y\} \\ &= t_{min}^x \end{aligned}$$

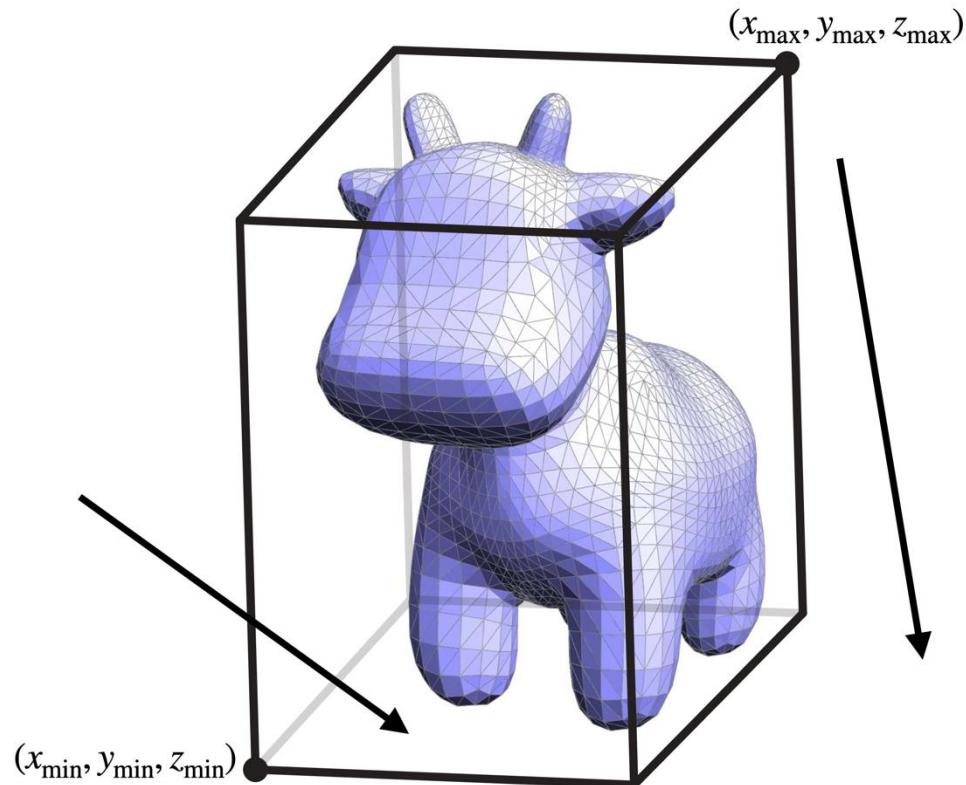
$$\begin{aligned} t_{exit} &= \min\{t_{max}^x, t_{max}^y\} \\ &= t_{max}^y \end{aligned}$$

不满足  $t_{enter} < t_{exit}$ !

# 包围盒性能

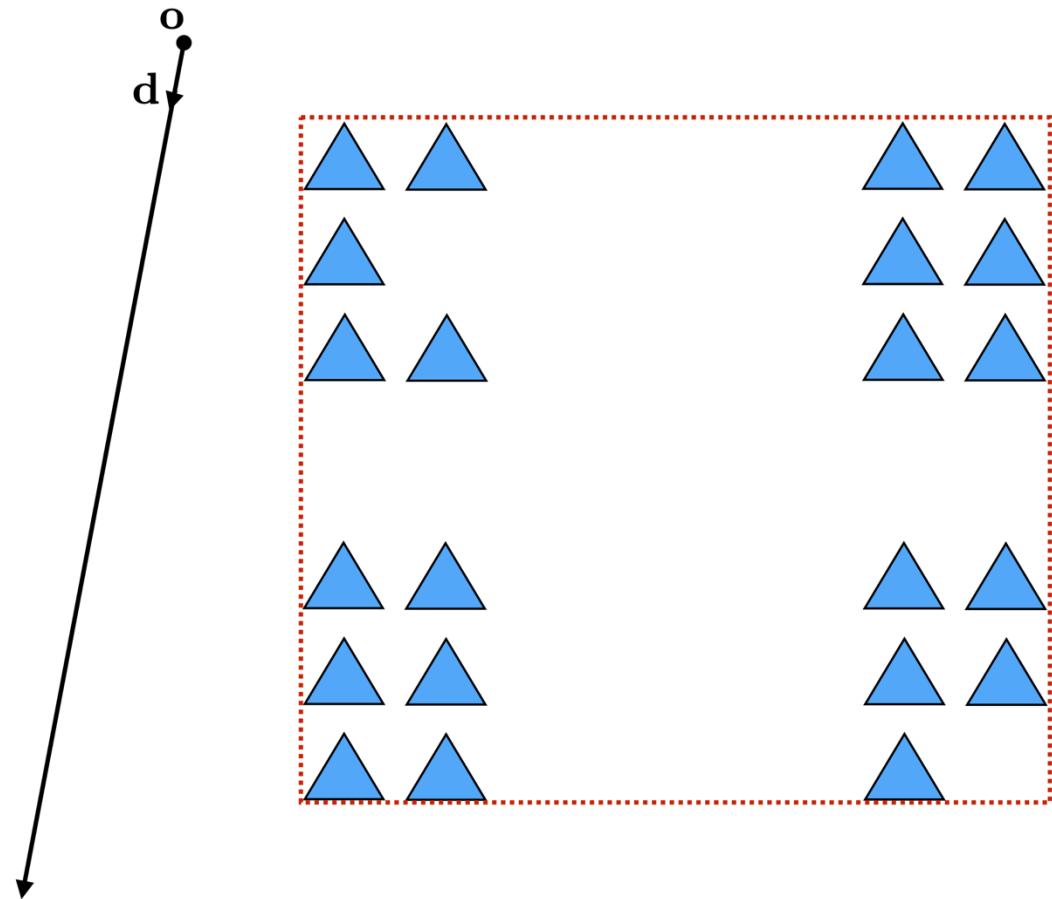
□ 我们是否做的更好?

- No, 算法最差情况还是  $O(N)$



# 不相交的情况

光线与包围盒不相交



当前的复杂度

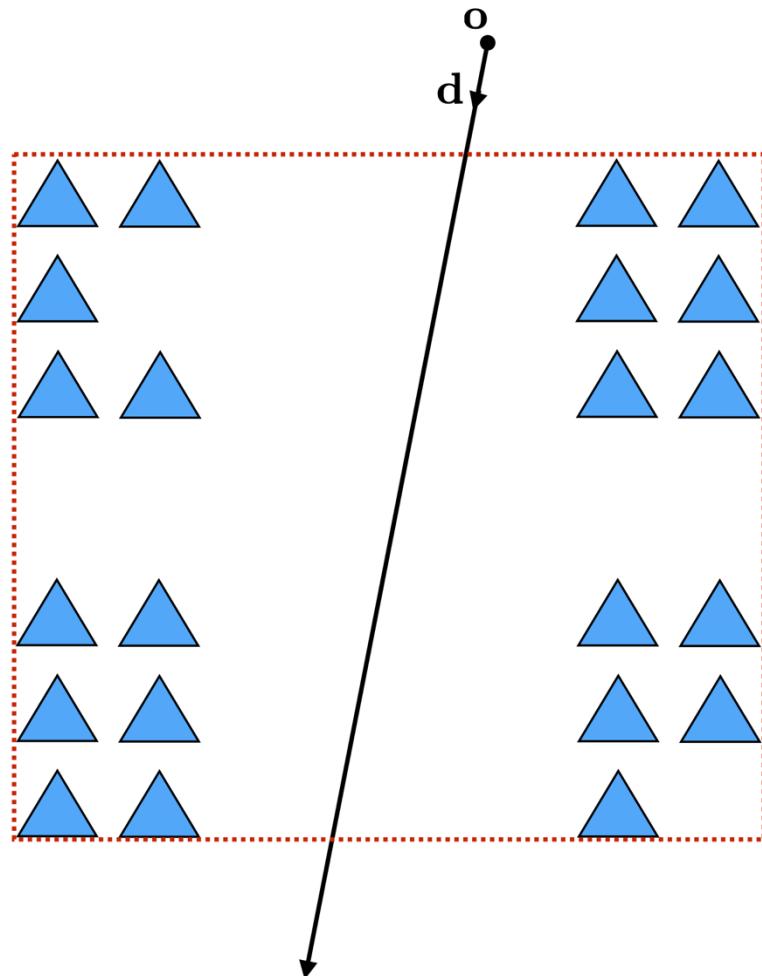
- 计算包围盒:  $O(n)$
- 测试光线与包围盒是否相交:  $O(1)$
- 多次摊销复杂度:  $O(1)$

# 相交的情况

光线与包围盒相交

当前的复杂度

- 计算包围盒:  $O(n)$
- 测试光线与包围盒是否相交:  $O(1)$
- 测试光线与所有三角形是否相交:  $O(n)$
- 多次摊销复杂度:  $O(n)$



需要测试所有三角形，没有比最简单的算法更好！

那么，要如何加速最近点计算？

# 一个更简单的问题

□ 假设我现在有一个整数数组  $S$

10 123 2 100 6 25 64 11 200 30 950 111 20 8 1 80

□ 给定一个整数，比如  $k = 18$ ，找到  $S$  中最接近  $k$  的元素

□ 根据集合的大小  $N$  找到  $k$  的复杂度是多少？

□ 我们能否做的更好？

□ 假设我们已经对数组进行了排序

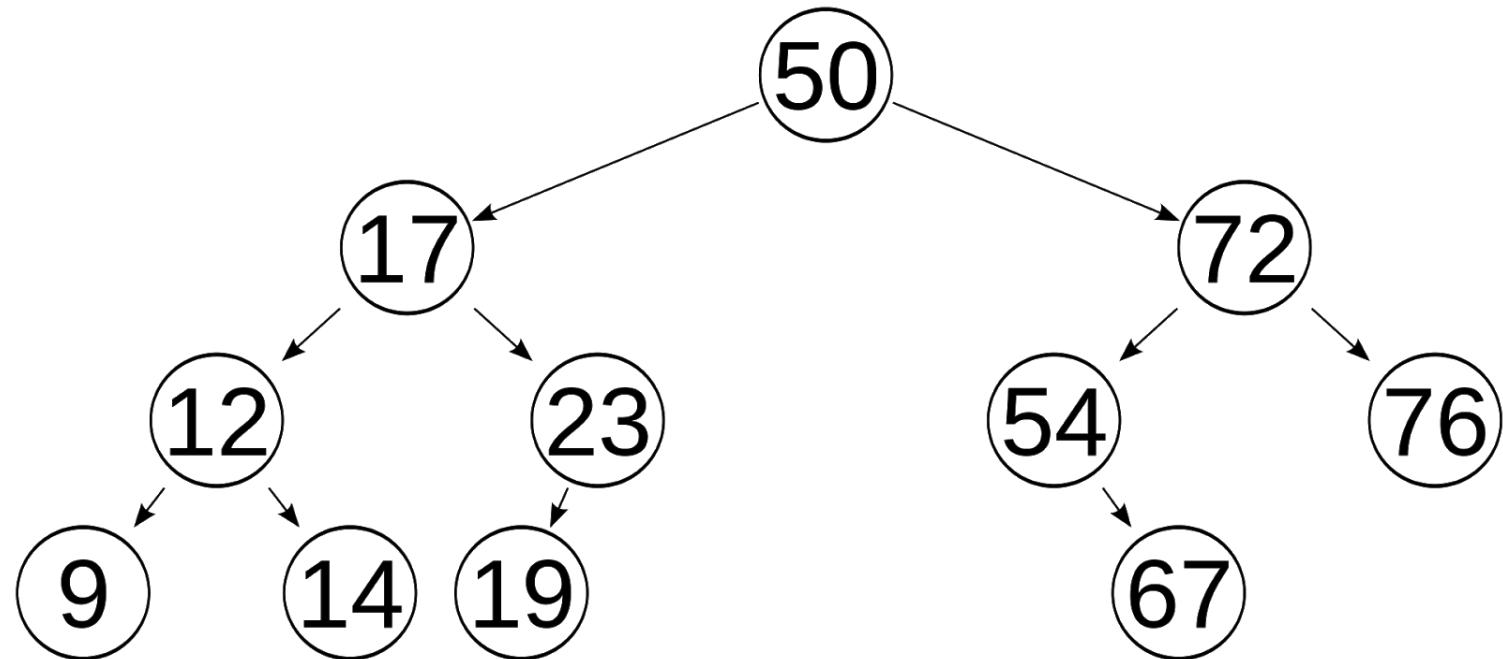
1 2 6 8 10 11 20 25 30 64 80 100 111 123 200 950

□ 现在的时间复杂度是多少（包括排序）？

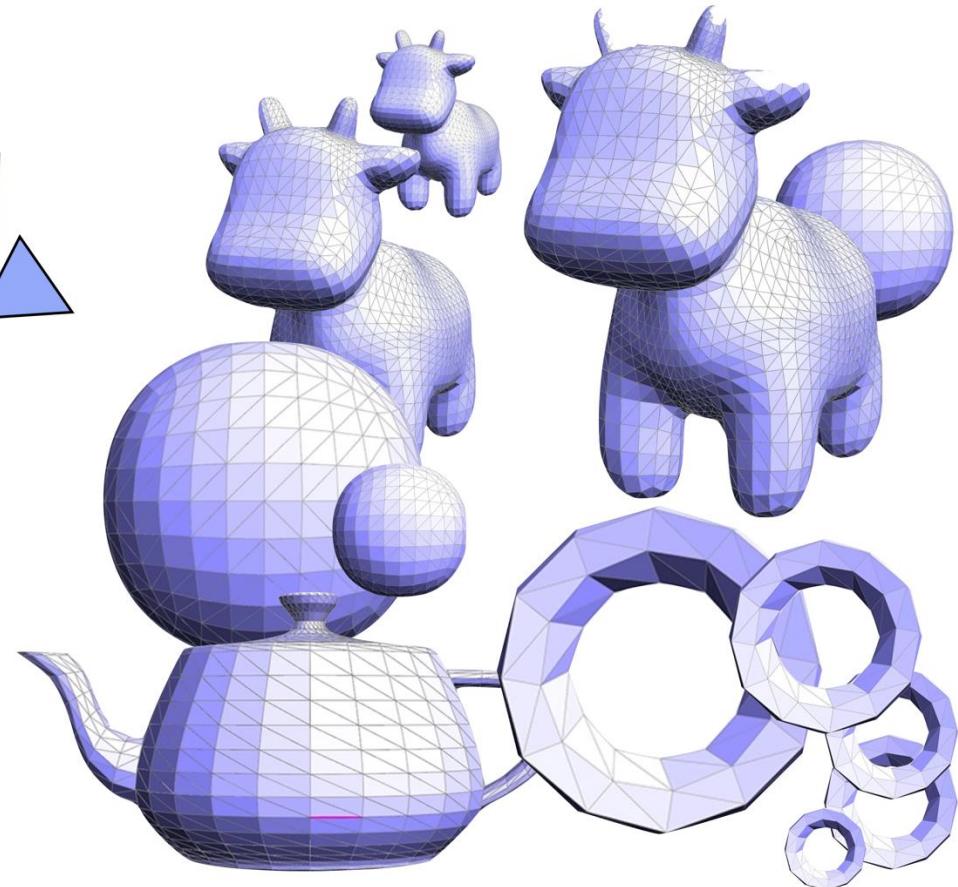
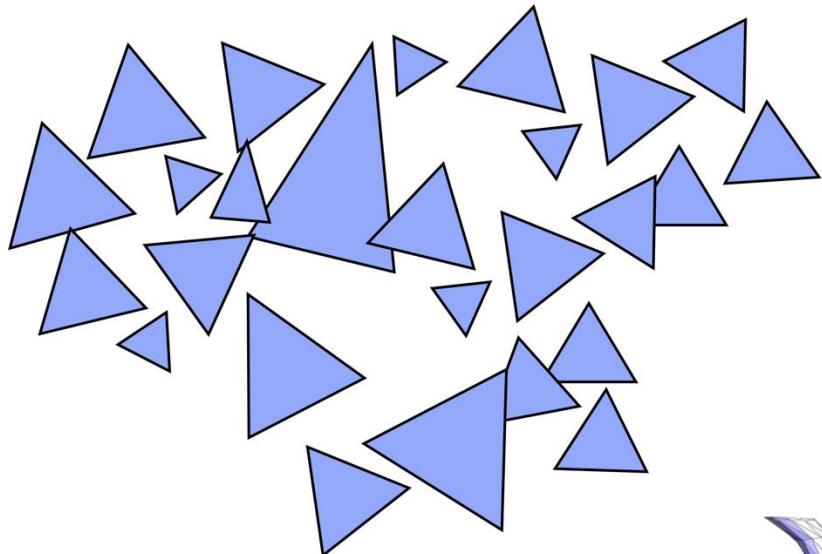
只做一次查询：  $O(n \log n)$  均摊后的复杂度：  $O(\log n)$

# 二分搜索树

- 每个节点均大于其左子树的节点
- 每个节点均小于其右子树的节点



能否以类似的方式重新组织图像基元，  
以加速光线与网格交叉的查询？



# 层次包围盒

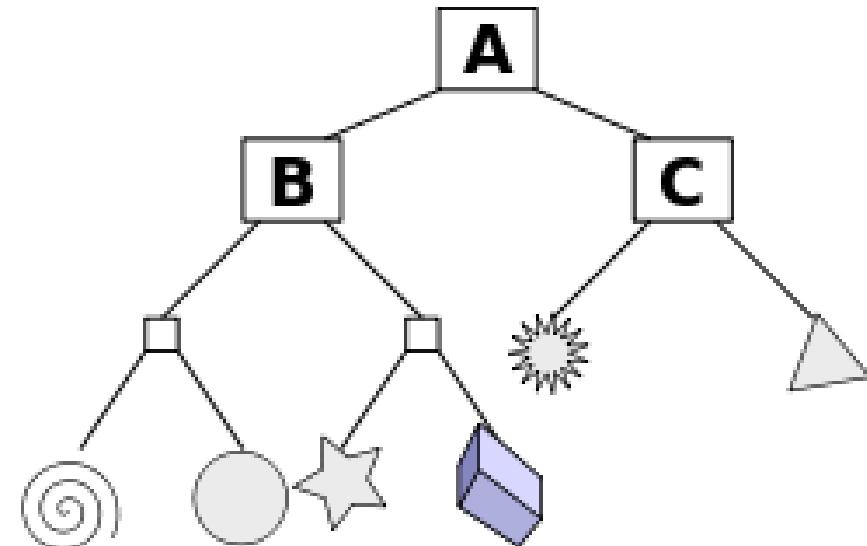
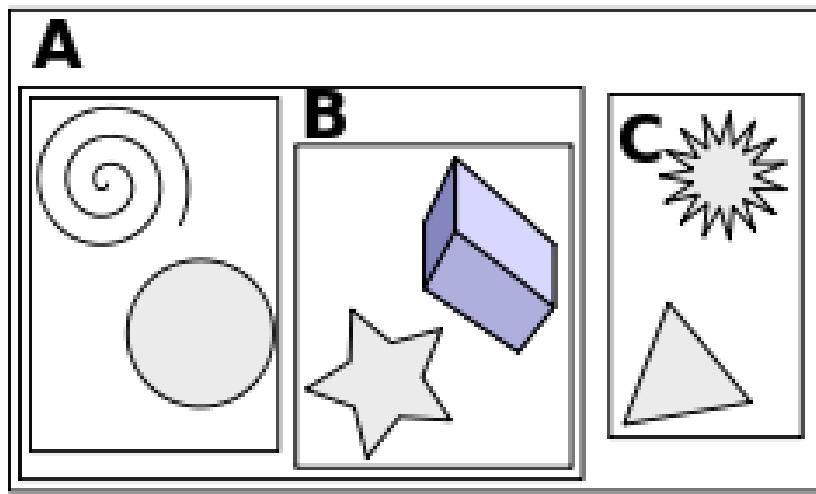
# Bounding volume hierarchy (BVH)

# 层次包围盒

□一种树形结构，对空间中的几何对象进行分层划分

□查询时

- 从树的顶部开始，逐级检查光线是否与节点的包围盒相交
- 若不是，则跳过包围盒内的所有几何对象
- 大大减少需要进行**精确交叉检查**的几何对象数量



# 层次包围盒

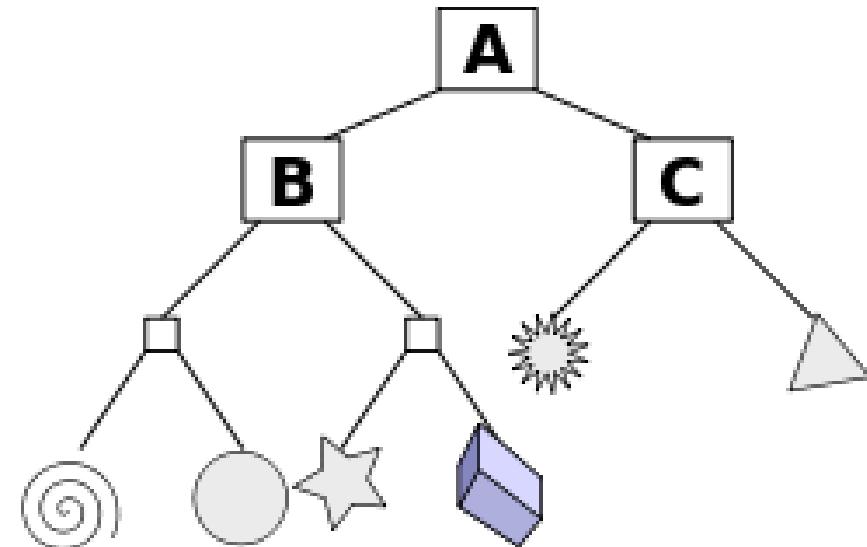
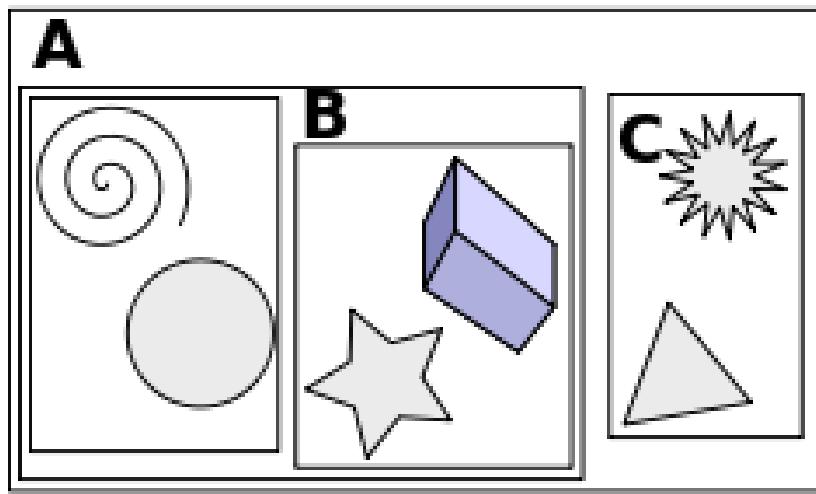
□ 层次包围盒 (BVH) 包含两种节点

□ 叶子节点

- 包含一个图像基元列表

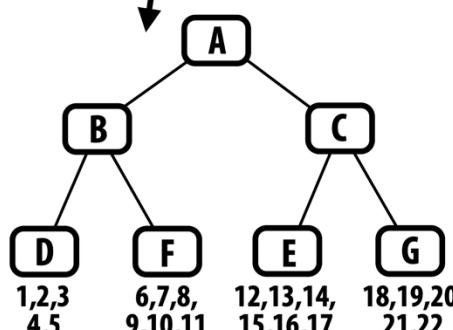
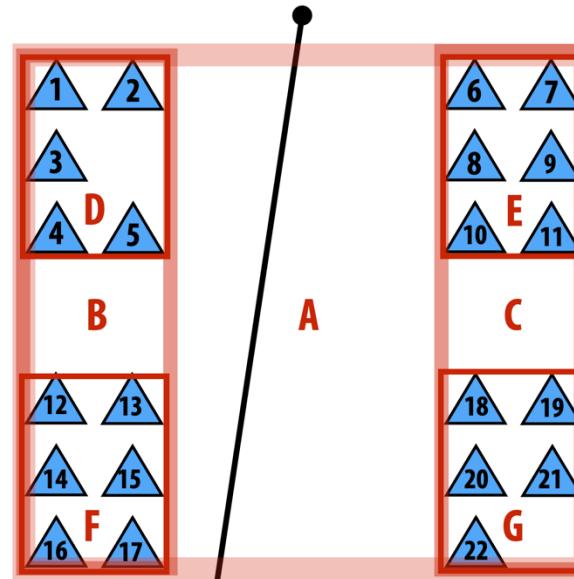
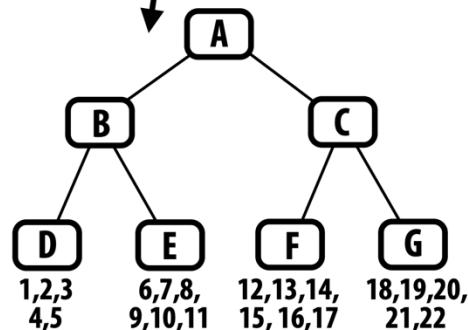
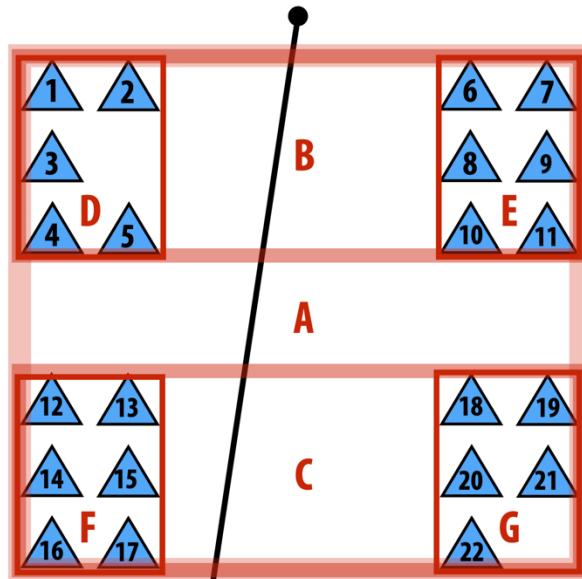
□ 中间节点

- 通向多个图像基元列表 (即多个叶子结点) 的代理
- 存储子树中所有图像基元的包围盒



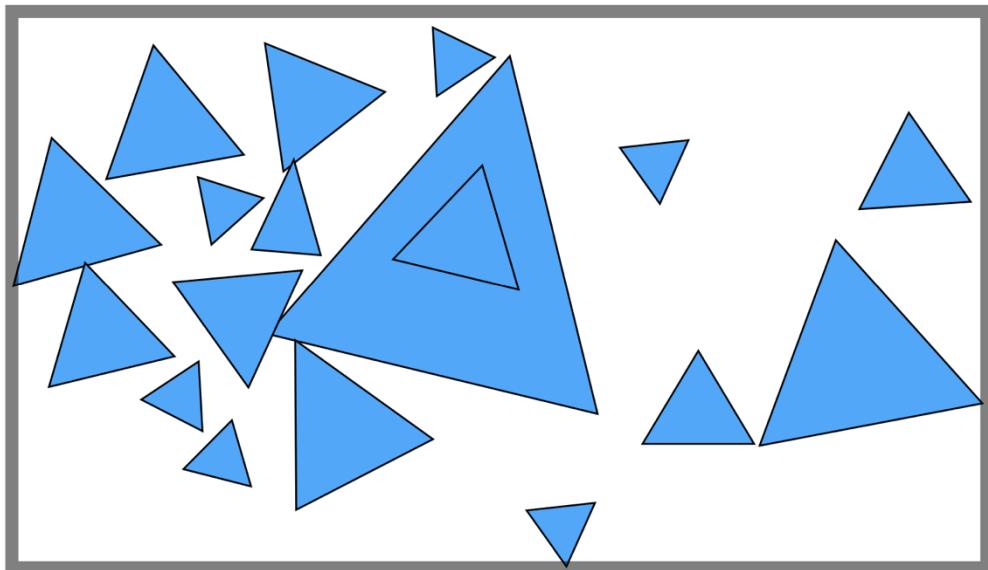
# 层次包围盒例子

- 两个不同结构的层次包围盒，包含相同的 22 个三角形
- 如何构造层次包围盒？

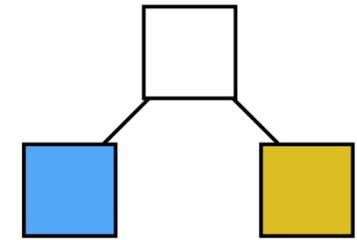
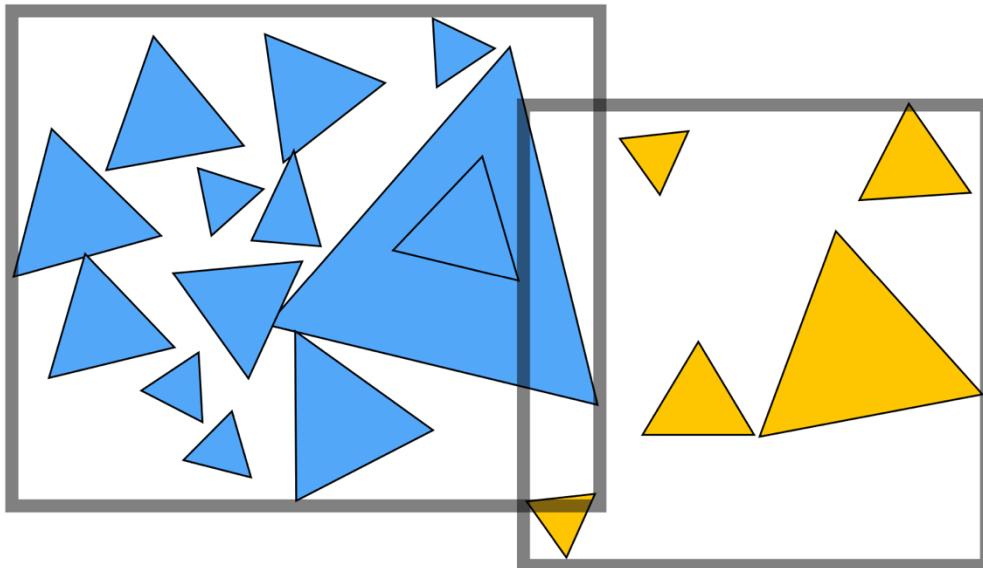


哪一个更好？

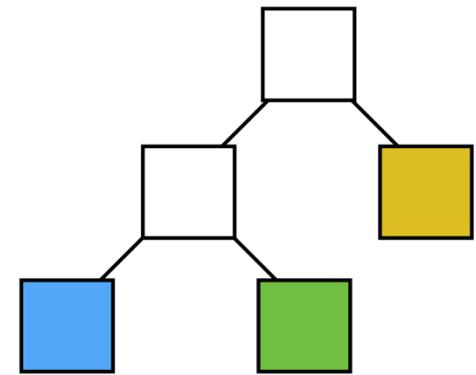
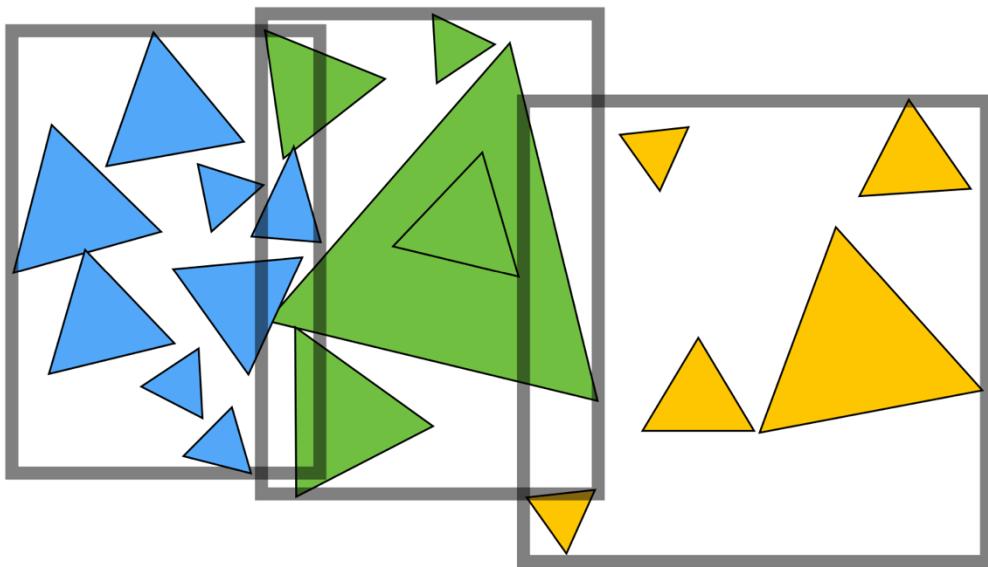
# 层次包围盒的构建



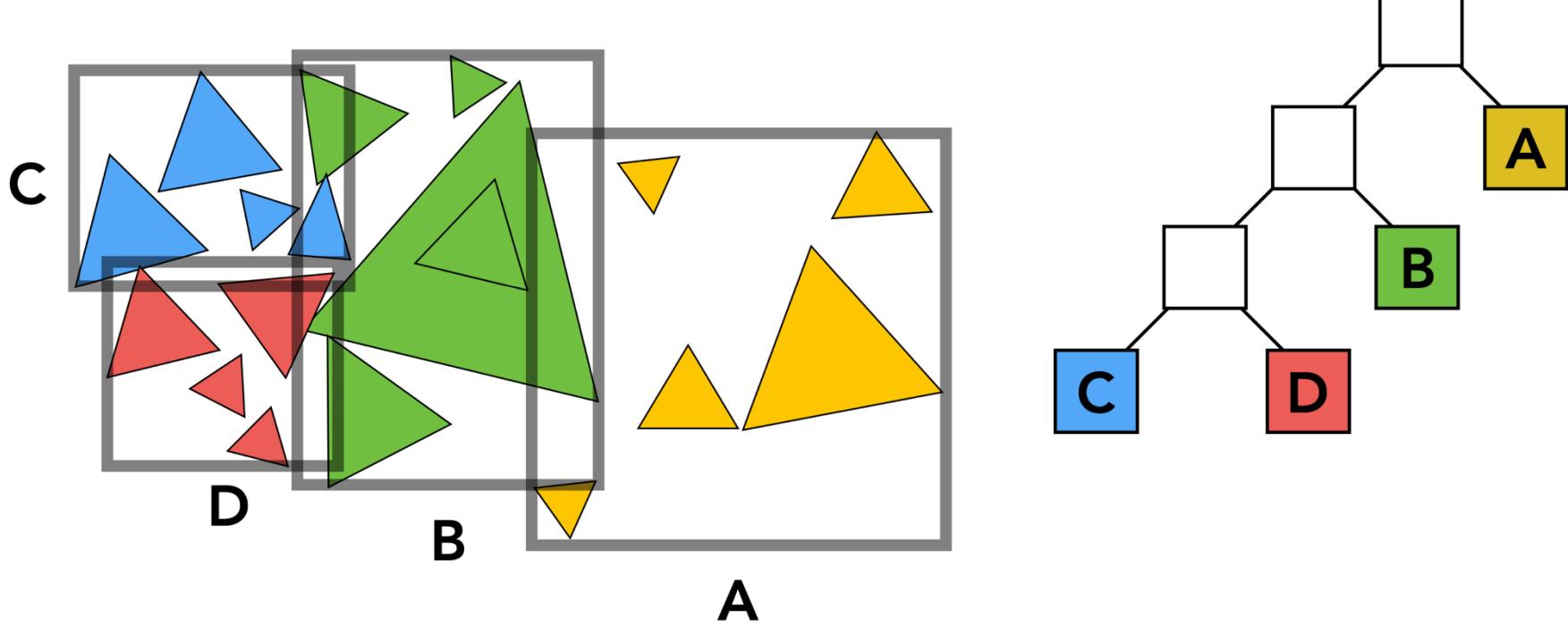
# 层次包围盒的构建



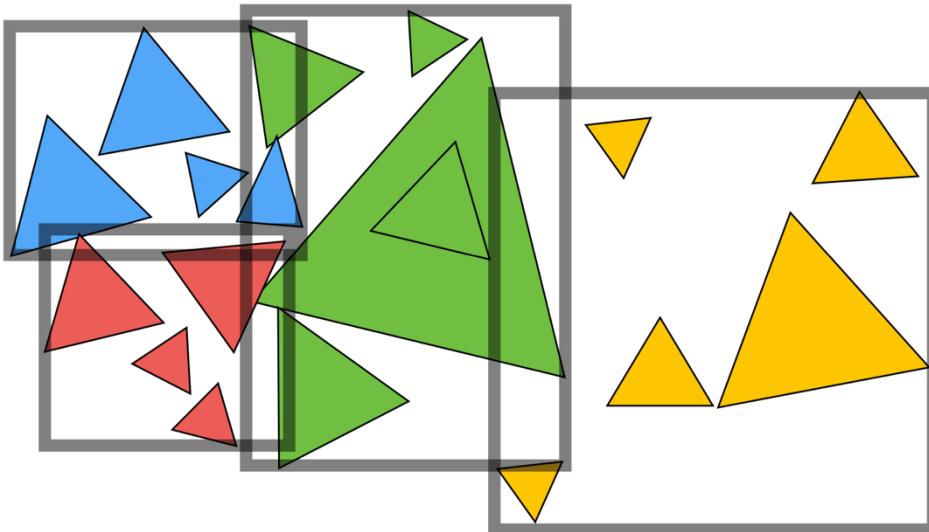
# 层次包围盒的构建



# 层次包围盒的构建



# 层次包围盒的构建



1. 找到包围盒
2. 迭代地把图像基元分成两个子集
3. 重新计算子集的包围盒
4. 在合适的时候停止
5. 字节点存储相应的图像基元列表

不同节点的包围盒可能会重叠，会导致什么问题？

# 构建层次包围盒的启发式规则

□ 如何划分节点？

- 启发式规则 1：总是选择节点中最长的维度划分
- 启发式规则 2：在中间基元的位置划分

□ 终止划分的规则

- 当节点包含合适数量的图像基元（比如 5 个）

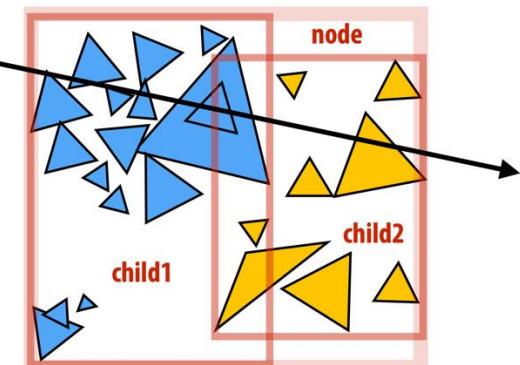
进一步思考：如何优化  
基于 BVH 的几何查询？

# 优化1：尽早终止查询

# 基于 BVH 的最近点查询 (算法1)

```
intersect(ray, box) // 测试光线 ray 是否与包围盒 box 相交
```

```
find_closest_hit(ray, node) {  
    bool hit = intersect(ray, node); // 判断当前是否相交  
    if (not hit) { return }  
    if (node is a leaf) { // 当前节点为叶子结点  
        check intersection with its primitives  
    }  
    else { // 当前节点为中间结点，递归检测左右子树  
        find_closest_hit(node->left_child);  
        find_closest_hit(node->right_child);  
    }  
}
```

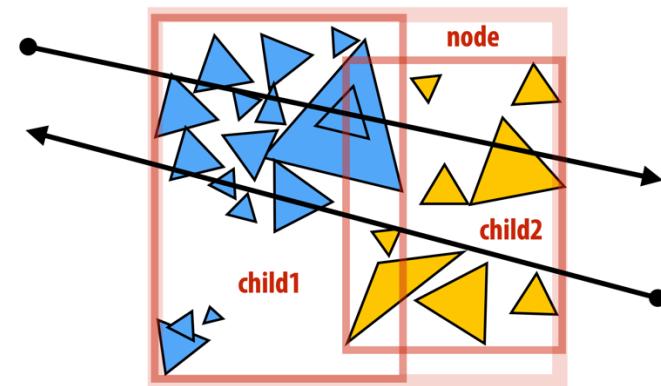


# 基于 BVH 的最近点查询 (算法2)

## 口优化策略

- 以可能 “提前” 终止的方式进行遍历 BVH

```
find_closest_hit(ray, node) {  
    // same as before  
    else { // 当前节点为中间结点，递归检测左右子树  
        // 找到光线与左右子树相交的先后顺序 (检查相交时间)  
        first_node, second_node = find_hit_order(ray, left_child,  
                                                right_child)  
  
        find_closest_hit(ray, first_node);  
        find_closest_hit(ray, second_node);  
    }  
}
```



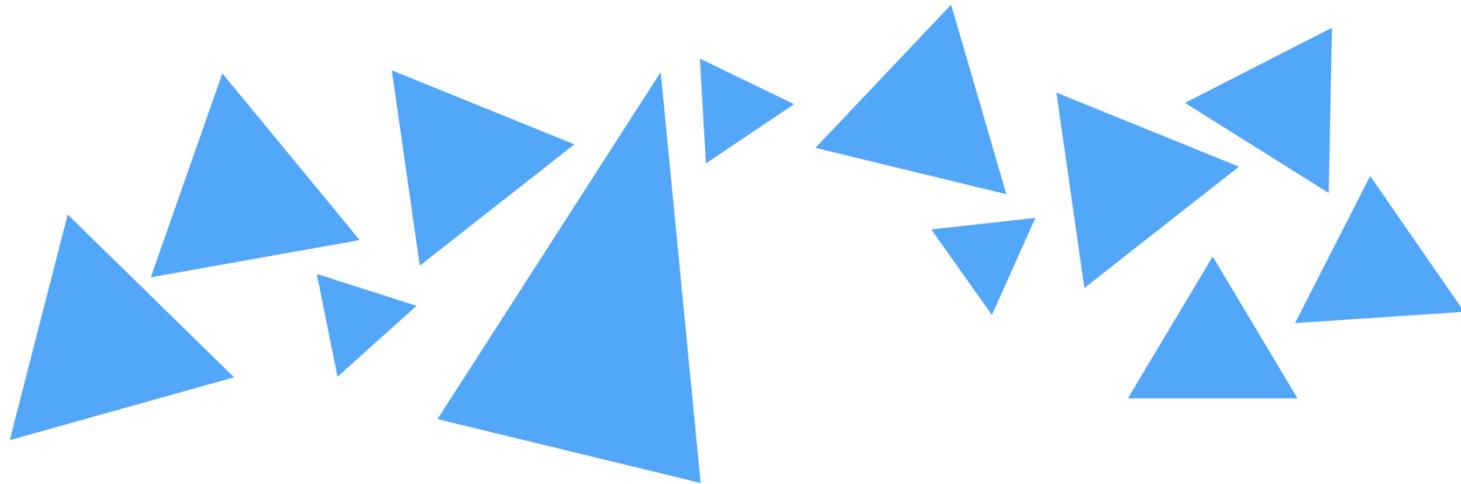
# 优化2：更好的 BVH 结构

# BVH 结构众多

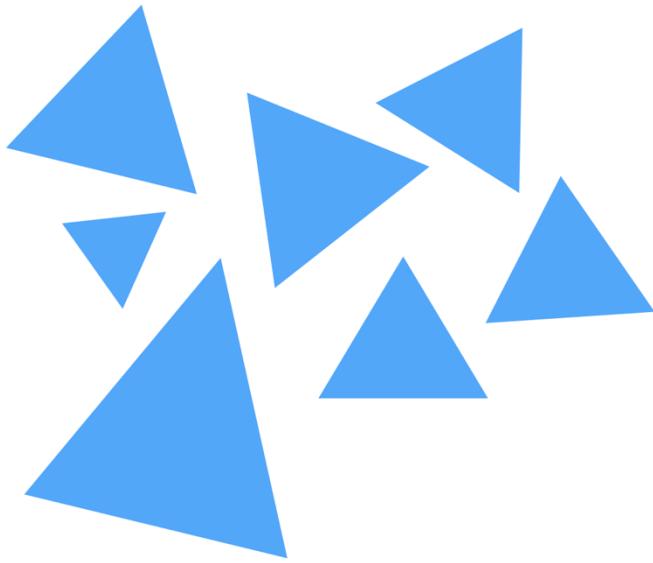
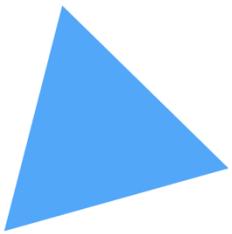
- 但对于给定的基元集，存在许多可能的 BVH 结构
- 比如，总共有  $2^N / 2$  种方法将  $N$  个基元划分为两组

**我们如何快速地构建高质量的 BVH (不仅仅是基于简单的启发式规则)？**

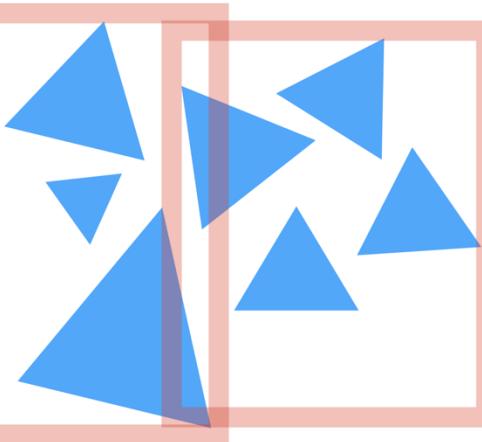
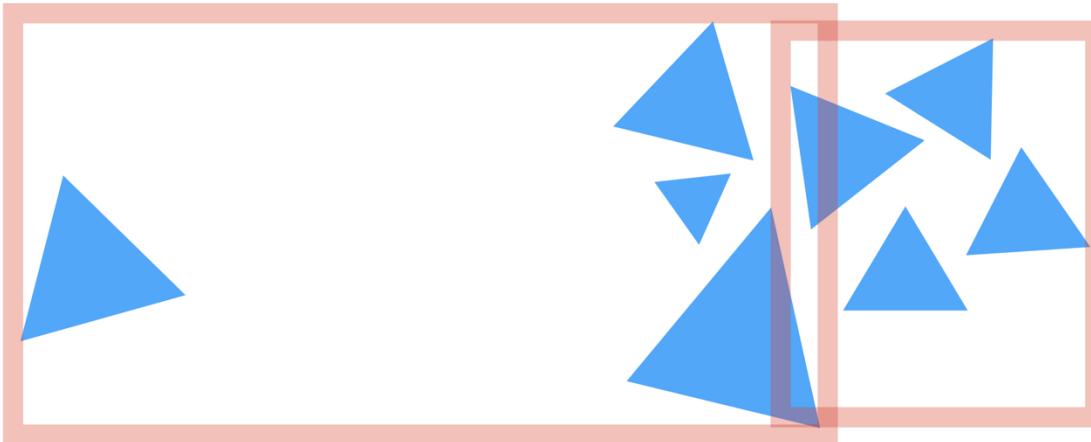
你会怎么将下列三角形划分成两组？



这个呢？

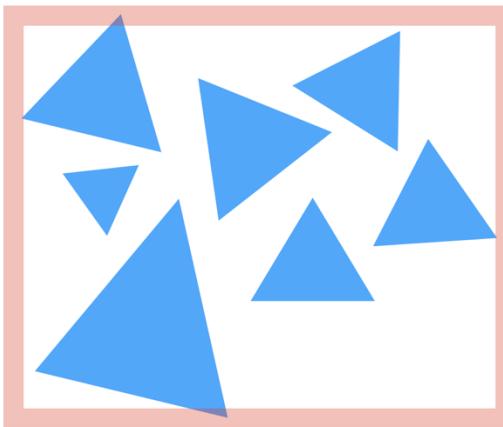
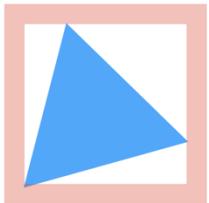


# 直观上，什么是好的划分？



两个字节点有相同  
数量的三角形？

接近平衡树，查  
找速度快 



还是考虑三角形之  
间的空隙？

避免浪费空间，  
提升查找效率 

# 如何衡量划分的质量？

□ 一个好的划分应尽可能减少在树中找到与光线最接近的图像基元的开销

如何计算开销  $C$ ？

□ 对叶子结点

$$C = \sum_{i=1}^N C_{isect}(i)$$

其中， $N$  为叶子节点中图像基元的个数， $C_{isect}(i)$  为测试与第  $i$  个图像基元相交的开销

$$C = NC_{isect}$$

假定测试与所有图像基元相交的开销相同，即  $C_{isect}$

# 如何衡量划分的质量？

口对中间节点，需要计算光线与内部节点相交的**期望成本**

口假设中间节点有  $A$  和  $B$  两个子节点，期望成本为

$$C = C_{trav} + p_A C_A + p_B C_B$$

- $C_{trav}$  为穿过一个内部节点的开销 (比如包围盒相交测试)
- $C_A$  和  $C_B$  为与子树进行相交测试的开销
- $p_A$  和  $p_B$  分别为光线与子节点  $A$  和  $B$  相交的概率

口如何计算？

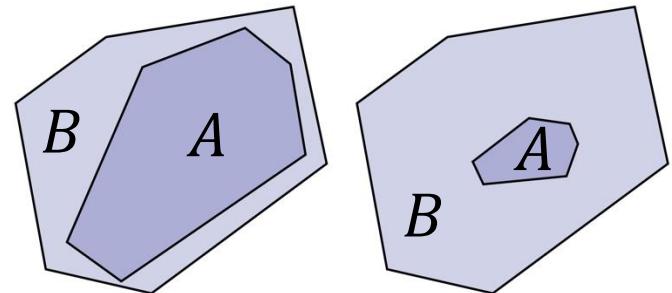
- $C_{trav}$  已知，即光线与包围盒的相交测试
- $C_A$  和  $C_B$  通常被启发式地计作**节点图像基元的个数**

$$C = C_{trav} + p_A N_A C_{isect} + p_B N_B C_{isect}$$

# 相交的概率怎么算？

口若凸对象 A 在凸对象 B 内部，则随机射线既命中 B 也命中 A 的概率可通过它们的面积比计算

$$p(\text{hit } A \mid \text{hit } B) = \frac{S_A}{S_B}$$



口因此，节点访问概率可启发式地利用节点内图像基元的面积之和与整个网格的表面积的比值来计算

$$C = C_{trav} + \frac{S_A}{S_N} N_A C_{isect} + \frac{S_B}{S_N} N_B C_{isect}$$

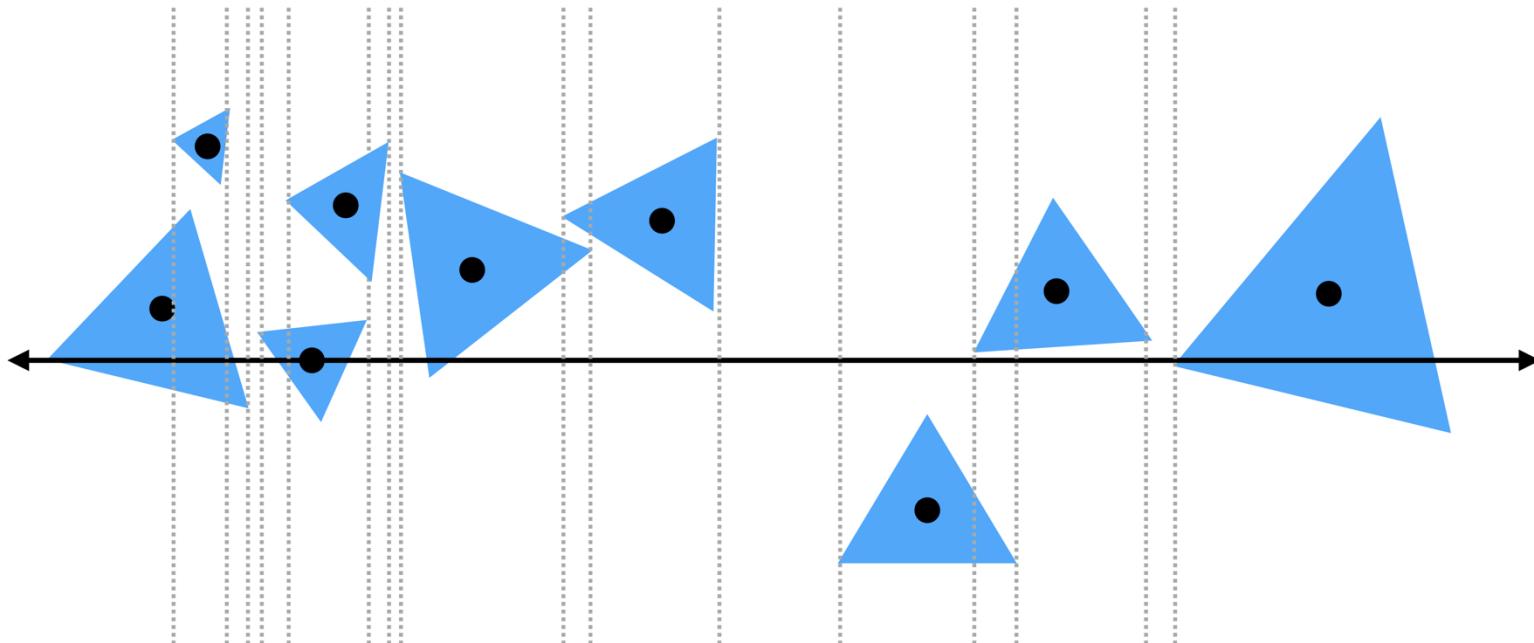
口该假设在现实中不一定成立

- 需要光线是随机的
- 同时图像基元之间没有相互遮挡

# 实现分区

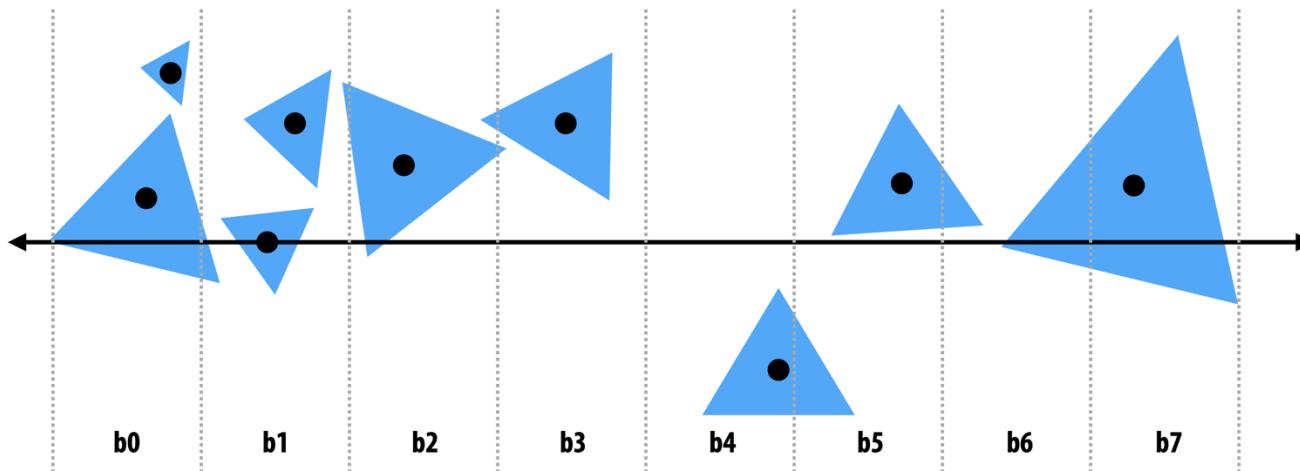
□ 将寻找好的分区限制在与坐标轴对齐的空间分区上：

- 选择一个坐标轴，搜索一个垂直于该轴的平面作为分割平面
- 根据物体的重心所在的位置，将其分配到分割平面的某一侧
- 注意当分割平面移动超过三角形的边界时，成本估算才会变化
- **需要考虑相当多的可能分割平面...**



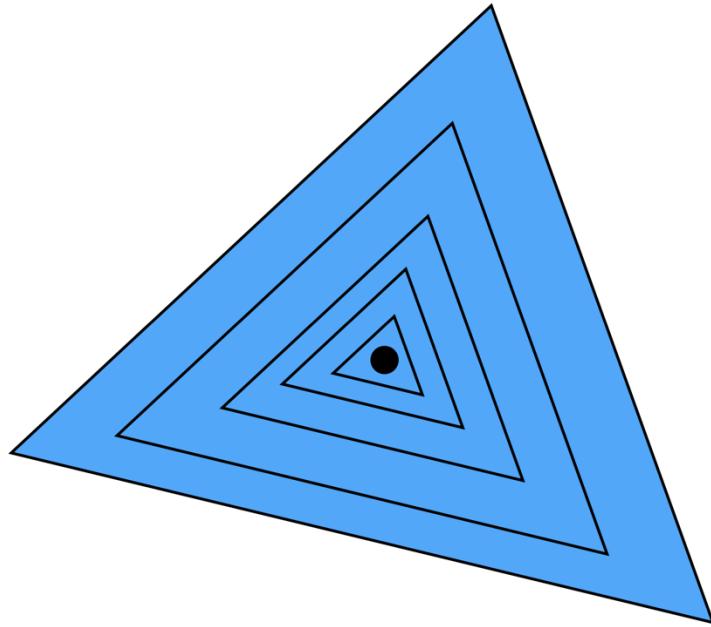
# 高效地实现分区

口现代的高效近似方法：将物体的空间范围分成  $B$  个桶，  
 $B$  通常较小： $B < 32$

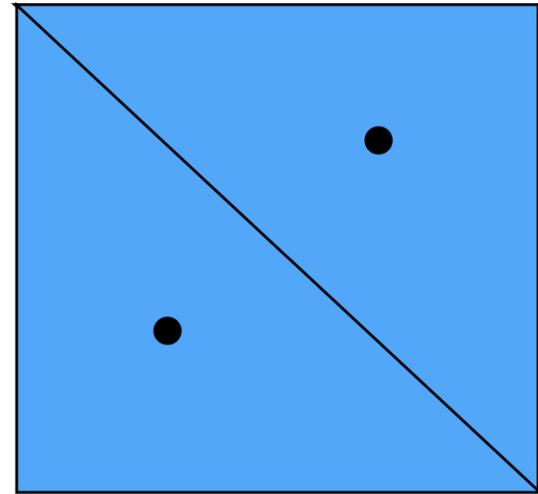


```
For each axis x,y,z:  
    initialize buckets  
For each primitive p in node:  
    b = compute_bucket(p.centroid)  
    b.bbox.union(p.bbox);  
    b.prim_count++;  
For each of the B-1 possible partitioning planes  
    Evaluate cost, keep track of lowest cost partition  
Recurse on lowest cost partition found (or make node a leaf)
```

# 棘手的情况



所有物体的重心都相同（所有物体最终都会被分到同一个分区）



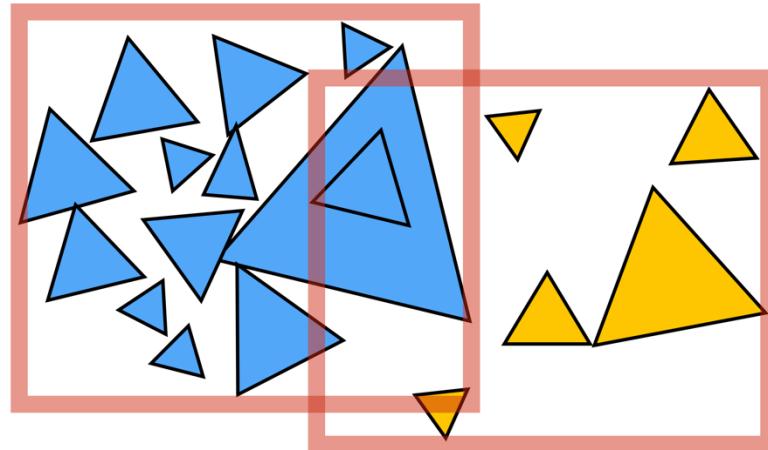
所有物体的包围盒都相同（光线常常会访问两个分区）

口一般来说，不同的策略可能更适合不同类型的几何形状或不同的物体分布情况

# 其他图形基元划分方法

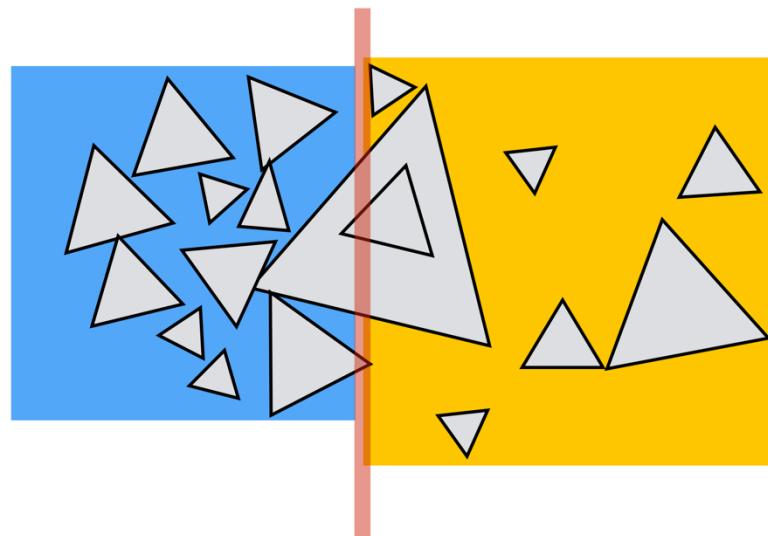
口基于图像基元的划分：

- 将节点的基元划分为不相交的包围盒 (包围盒在空间上可能重叠)
- 比如层次包围盒 (BVH)



口基于空间的划分：

- 将空间划分为不相交的区域 (同一个基元可以包含在空间的多个区域中)
- 比如均匀空间划分, KD 树

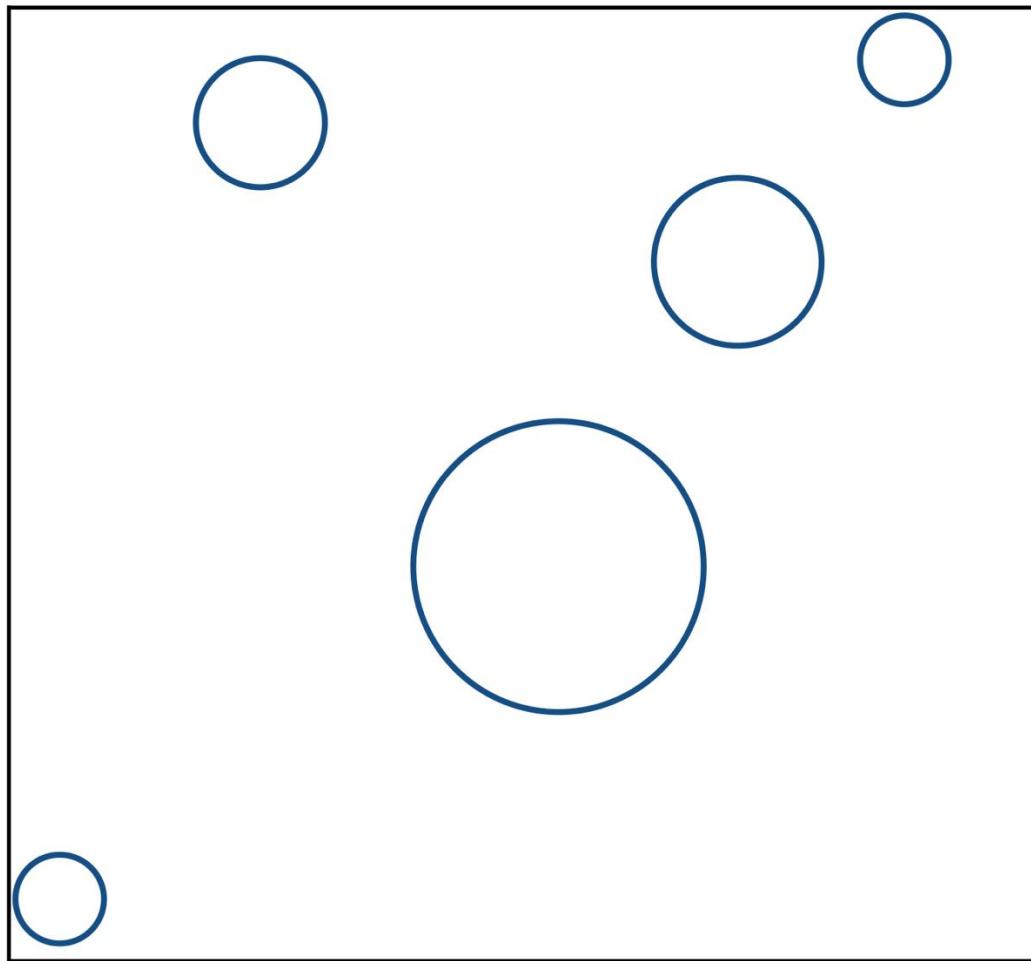


# 基于空间的图像基元 划分方法

# 均匀网格划分

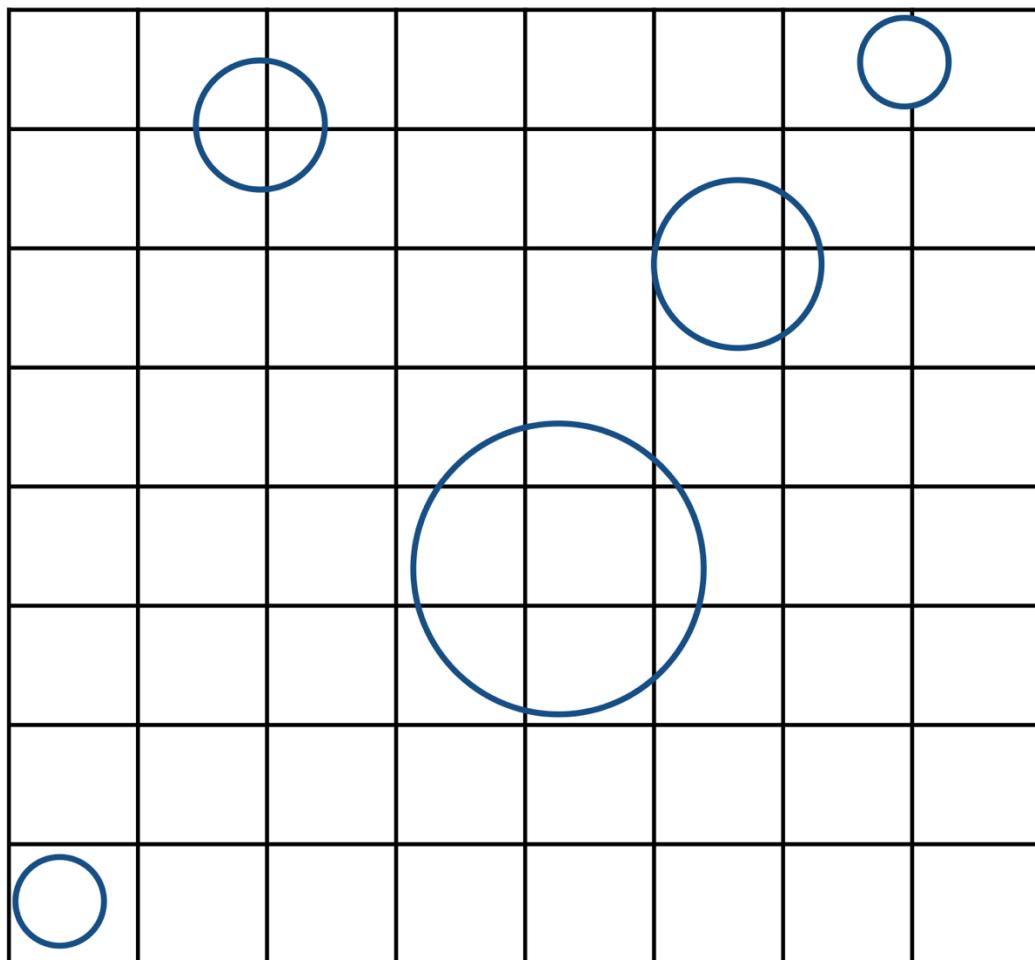
# Uniform grid

# 均匀网格划分



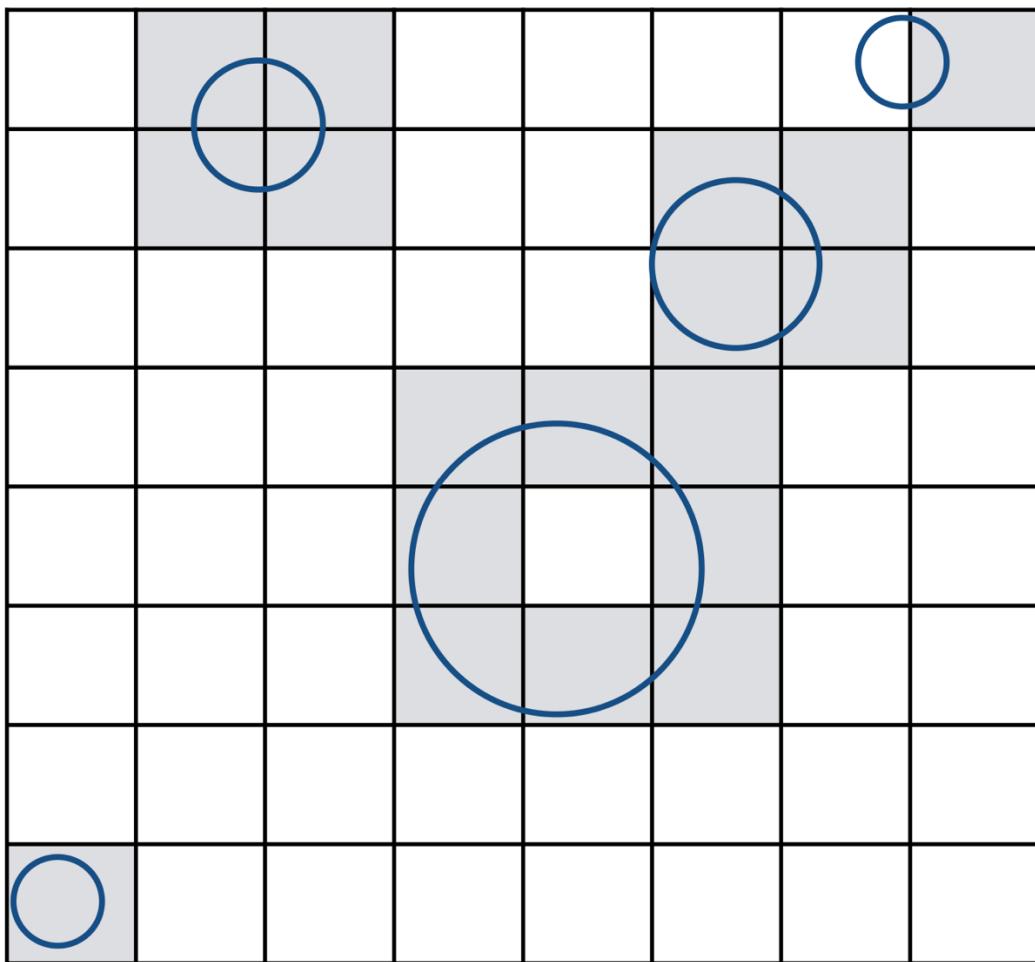
1. 找到包围盒

# 均匀网格划分



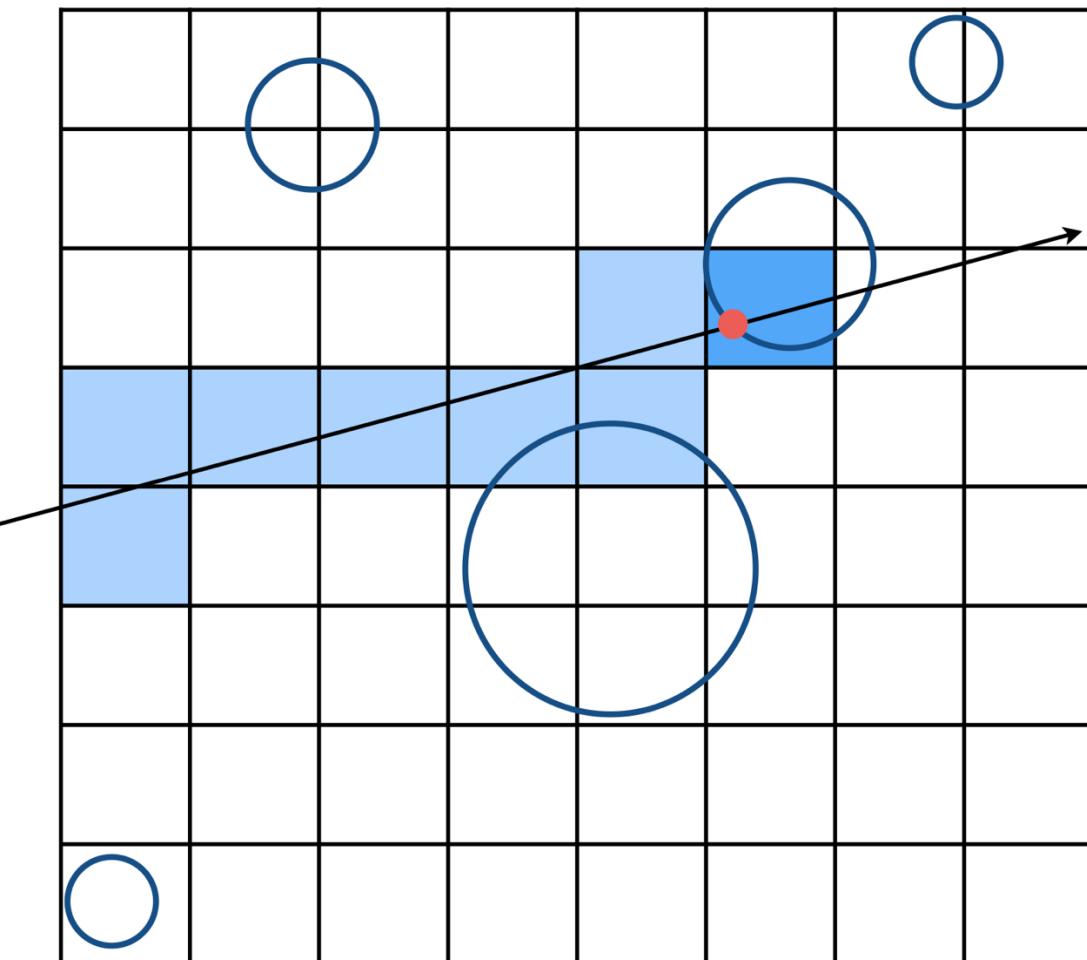
1. 找到包围盒
2. 创建网格

# 均匀网格划分



1. 找到包围盒
2. 创建网格
3. 每个格子保存重叠的图像基元

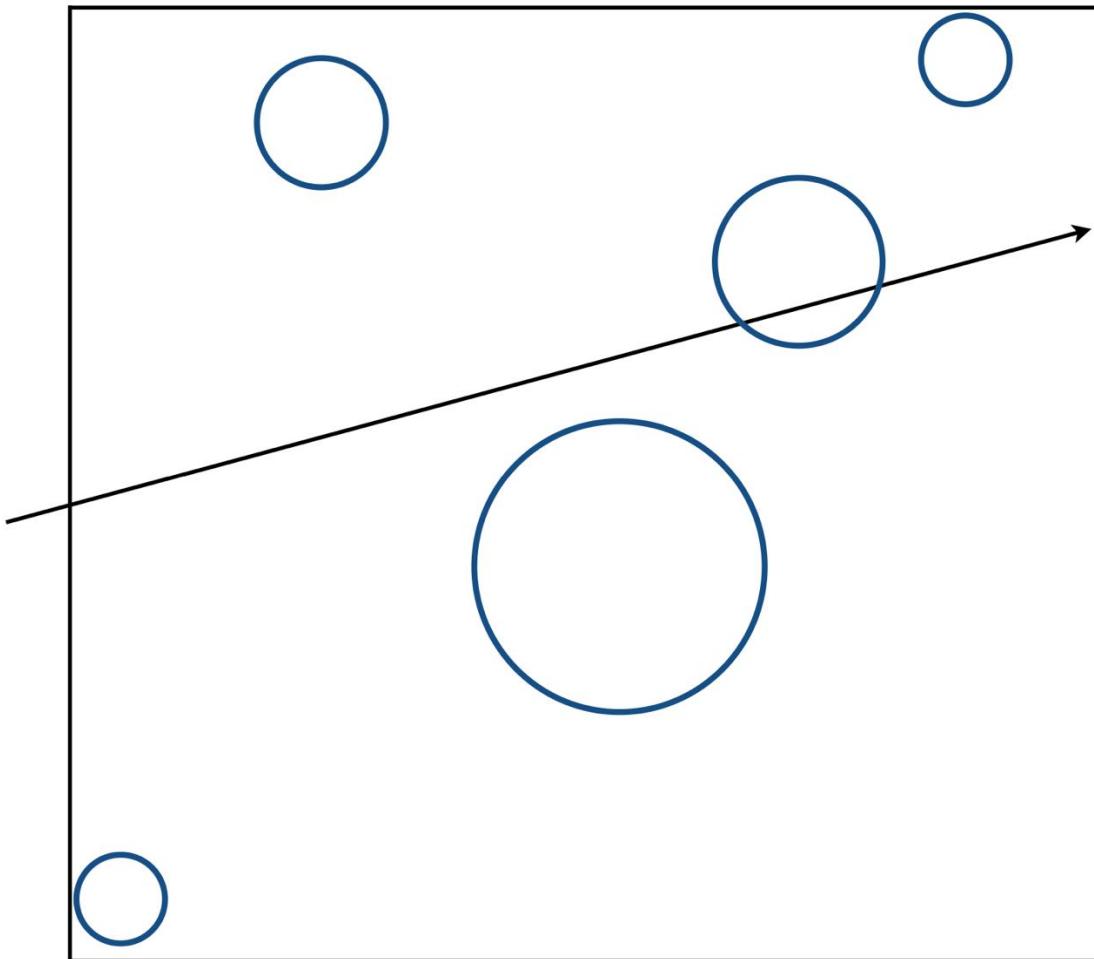
# 光线网格相交



按光线的走向依次  
穿过网格内的格子

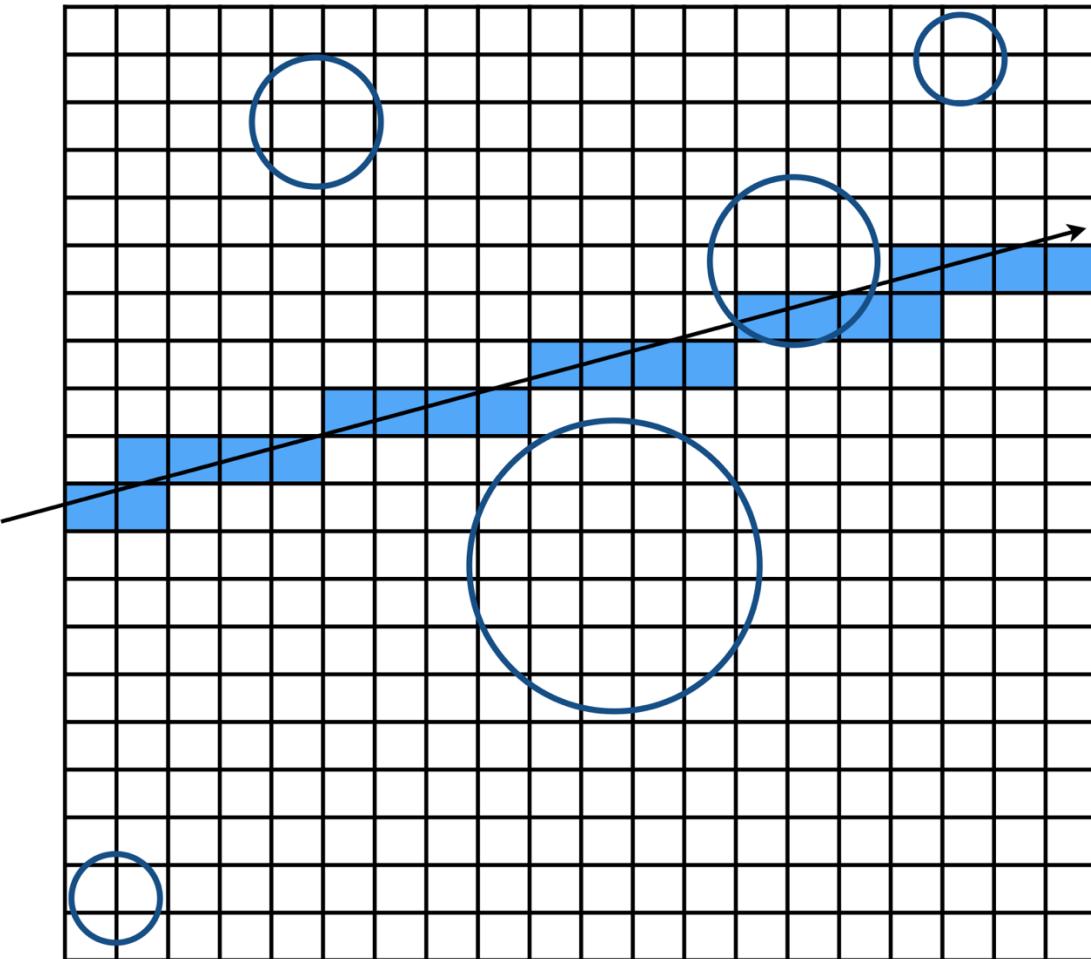
对于每一个格子  
测试是否与格子  
中保存的图像基  
元相交

# 如何确定网格的密度？



只有一个格子  
• 没有加速作用

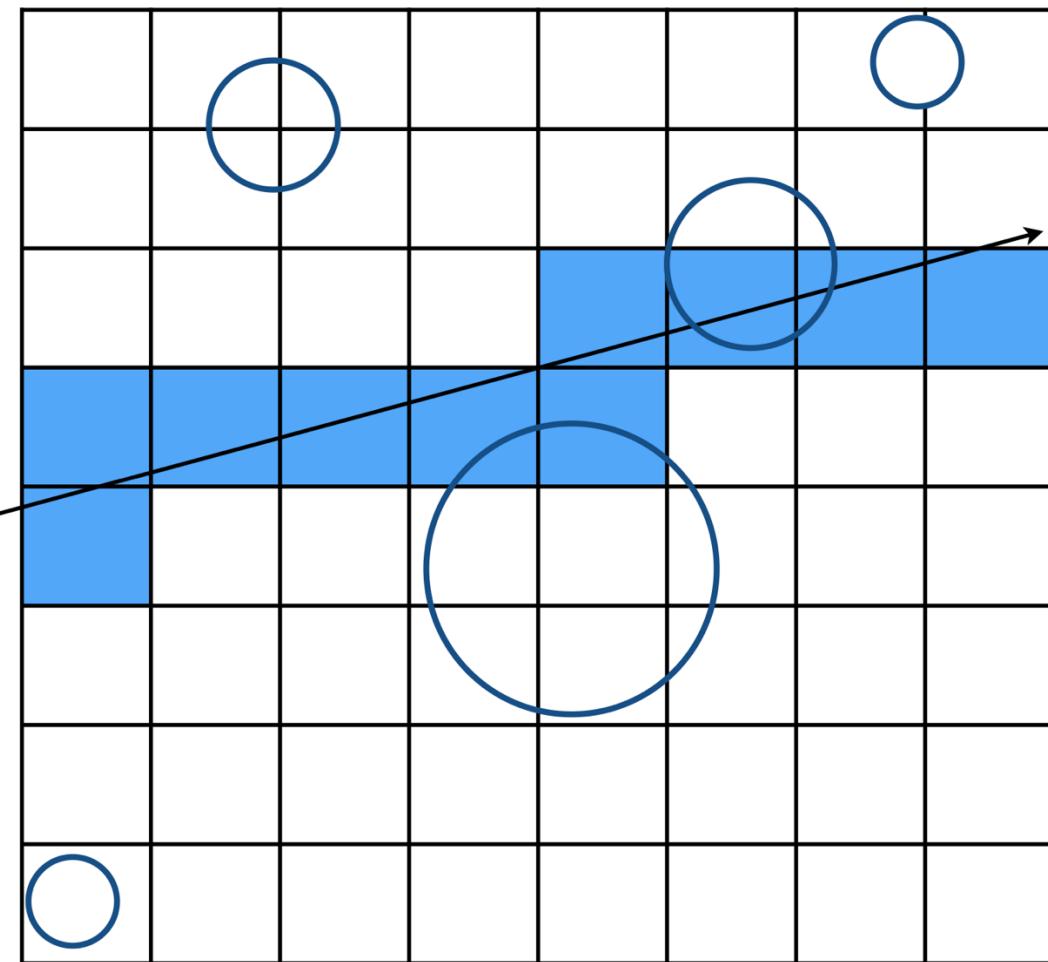
# 如何确定网格的密度?



高密度的格子

- 大量的网格遍历  
影响效率

# 如何确定网格的密度？



一个启发式的法则

- $\# \text{cells} = C * \# \text{objs}$
- $C \approx 27 \text{ in 3D}$

# 均匀网格划分优缺点

- 优点：实现简单
- 缺点：对于大部分场景，网格中的大多数单元格都是空的
- 哪些场景适合用均匀网格划分？



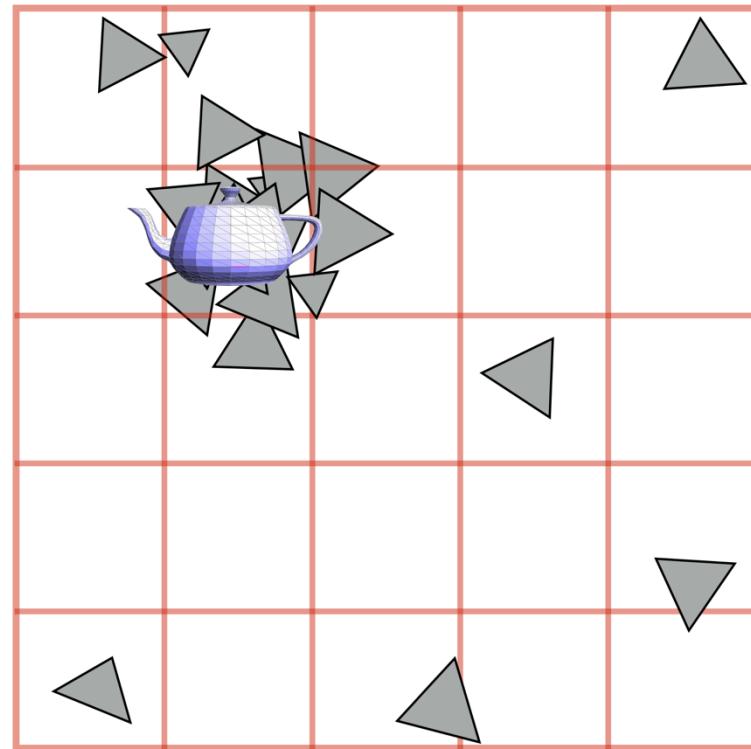
大量图像基元在大小和空间上分布均匀

# 哪些场景不适合用均匀网格划分？



Jun Yan, Tracy Renderer

体育场里的茶壶问题



□ 均匀网格无法适应场景中非均匀分布的几何物体

- 比如，具有高分辨率的物体（包含大量图像基元）仅分布在空间中的一个小部分

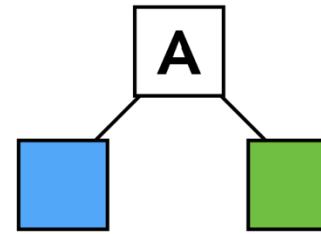
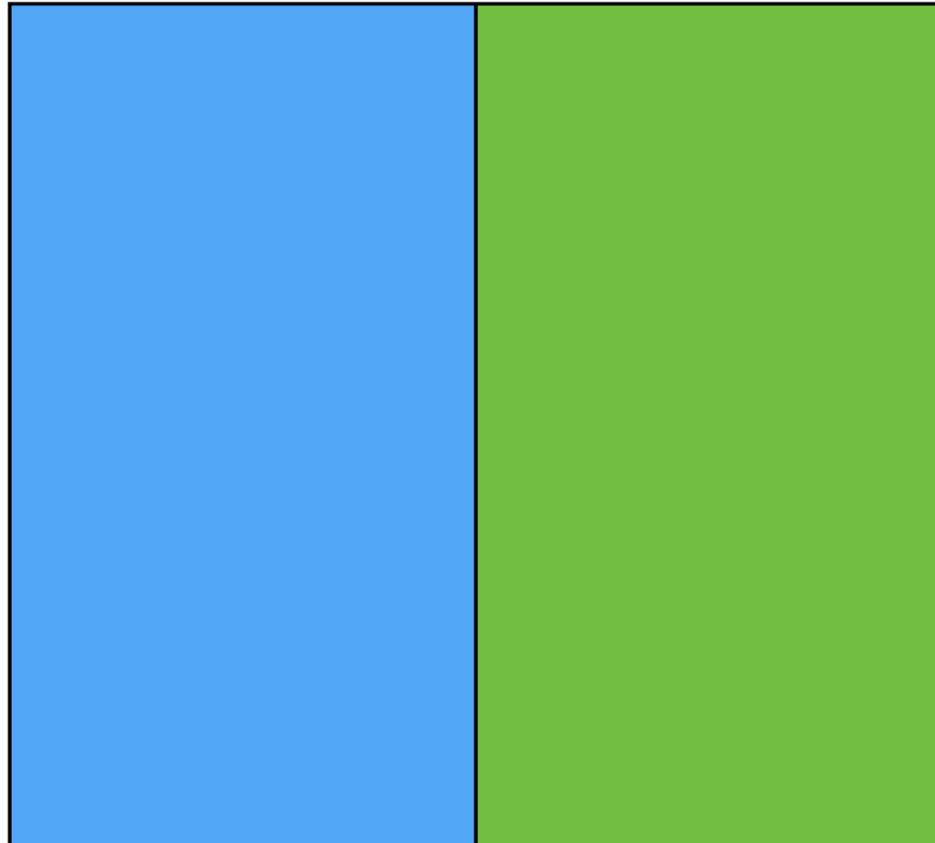
# 几何细节分布不均匀例子



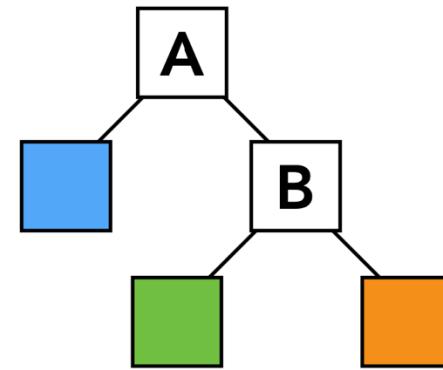
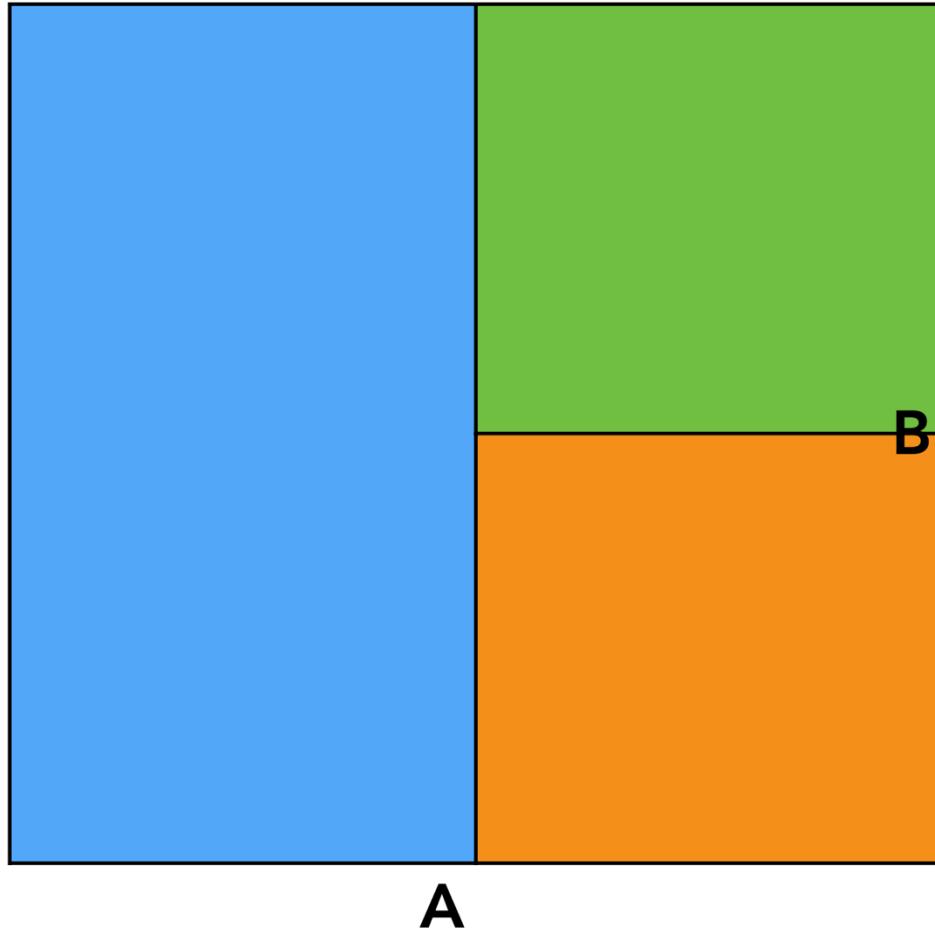
# KD 树划分

# KD Tree

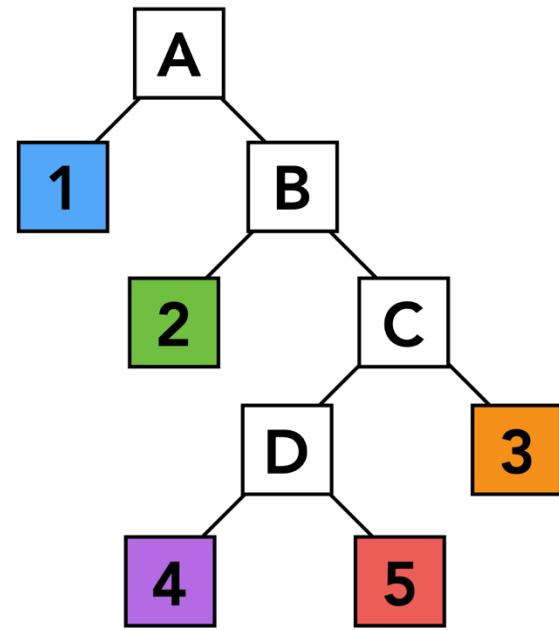
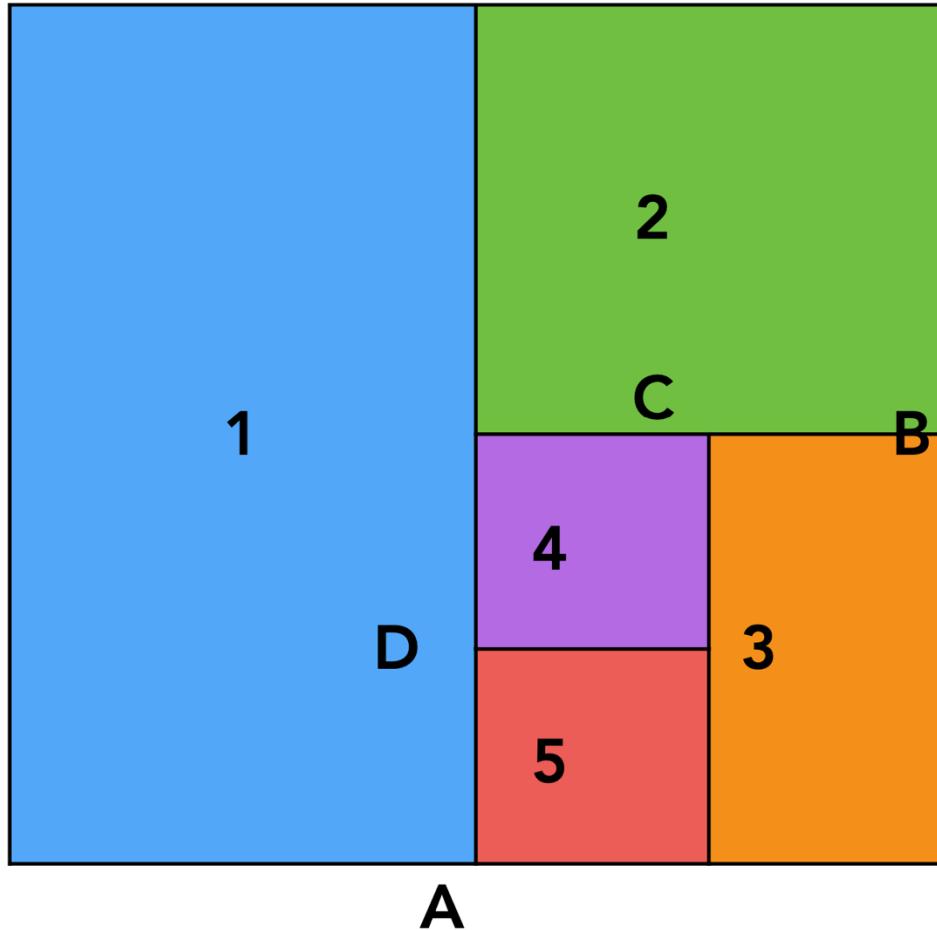
# KD 树划分



# KD 树划分

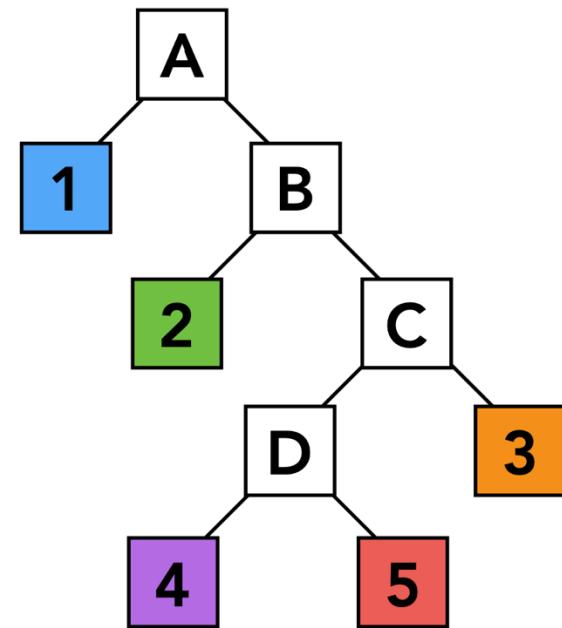
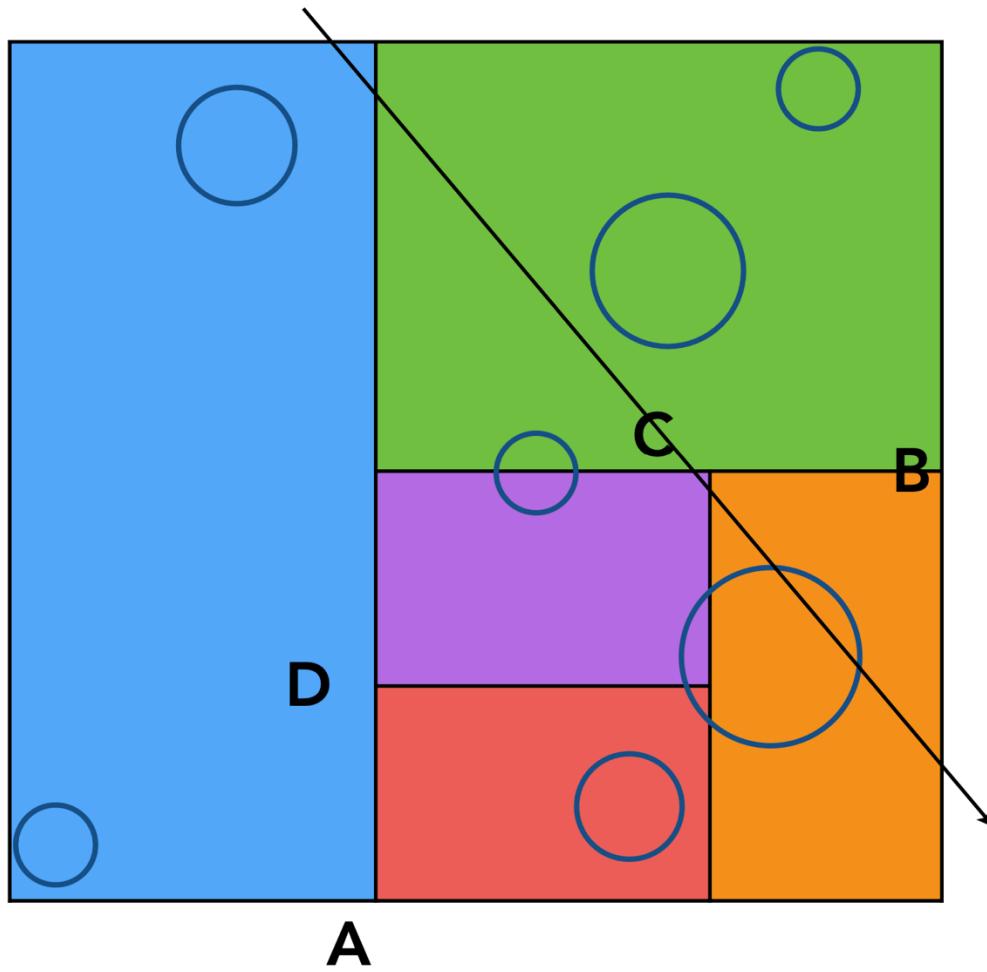


# KD 树划分

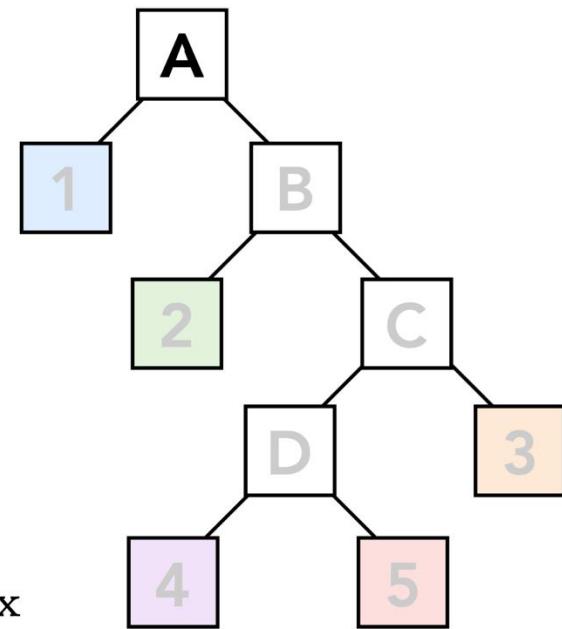
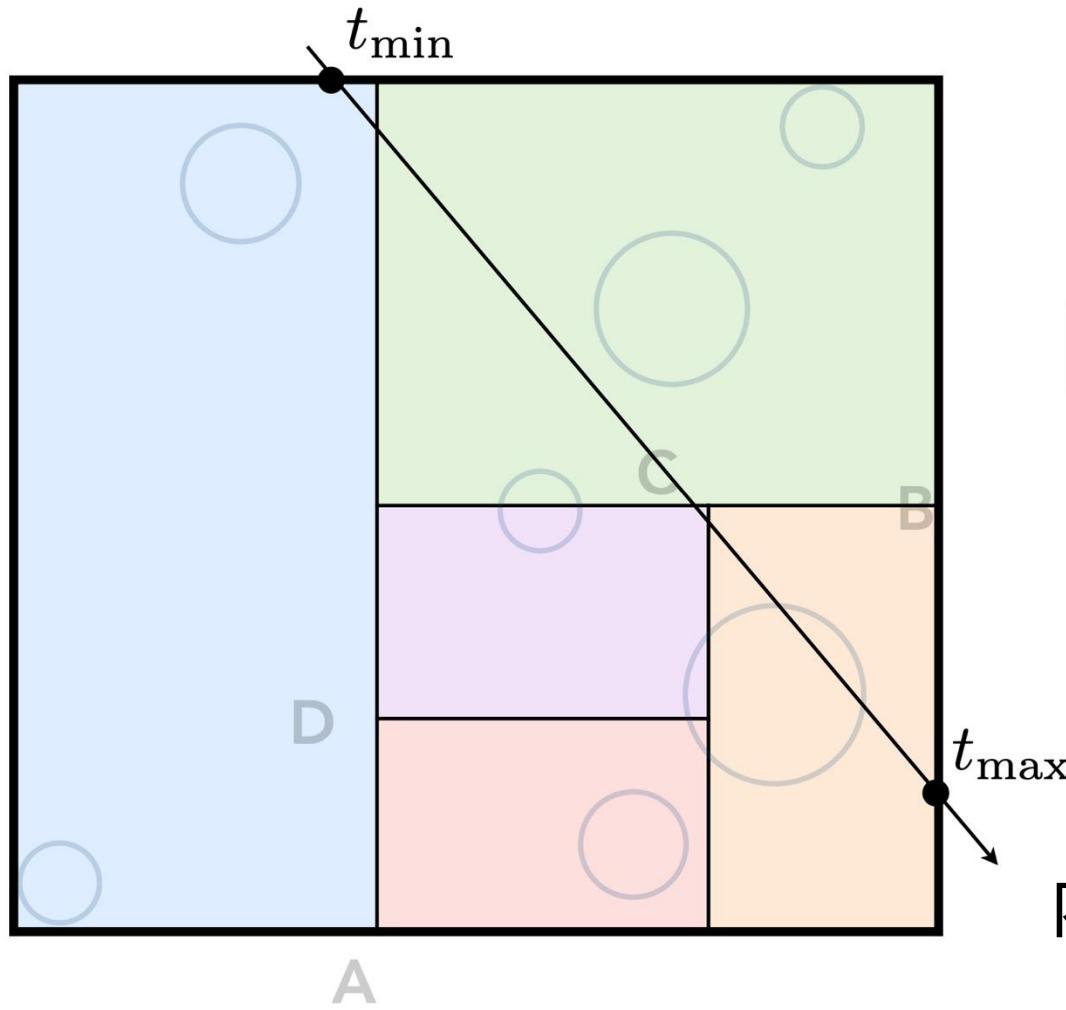


Note: also subdivide  
nodes 1 and 2, etc.

# KD 树划分

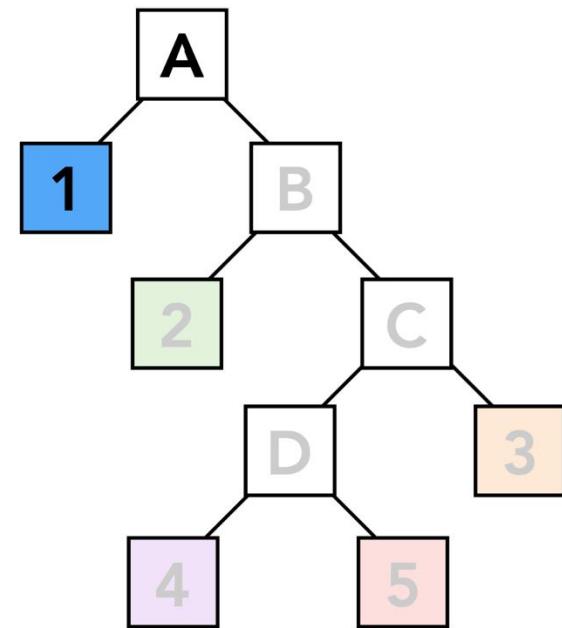
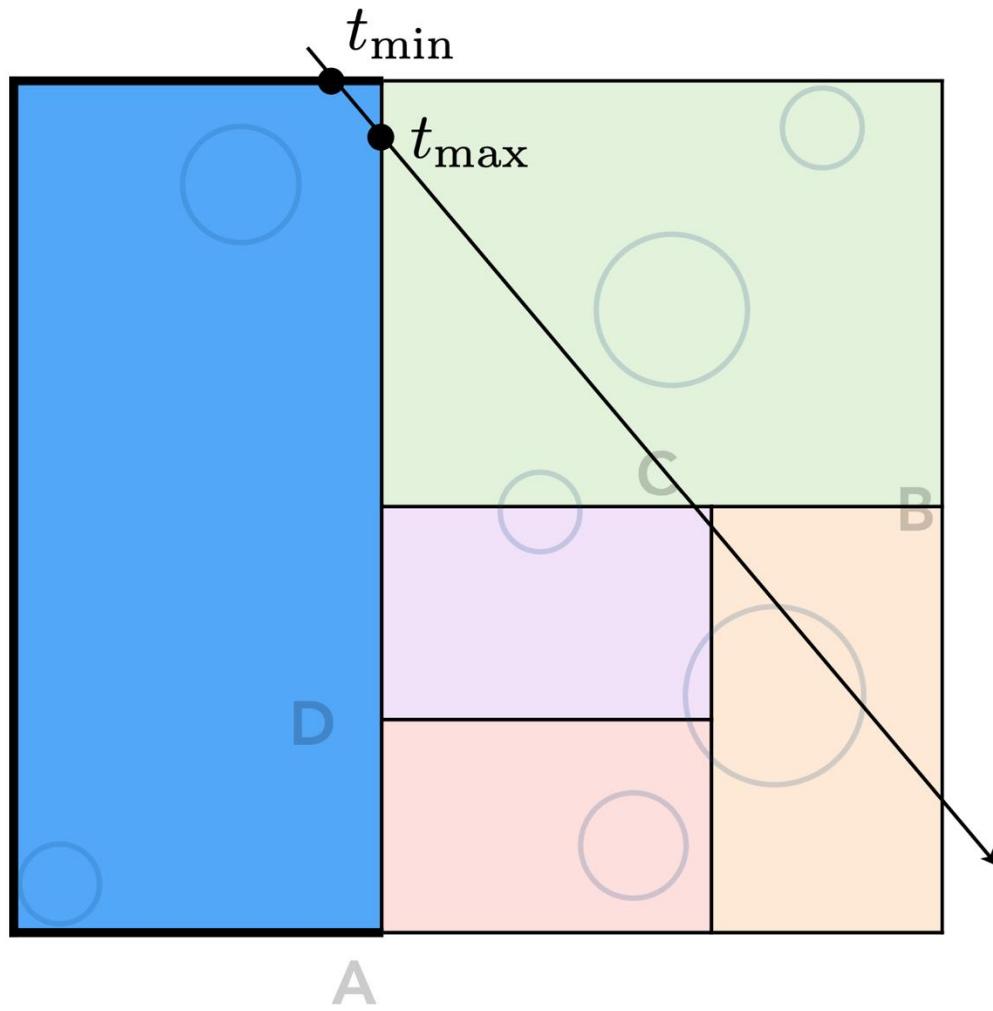


# KD 树划分



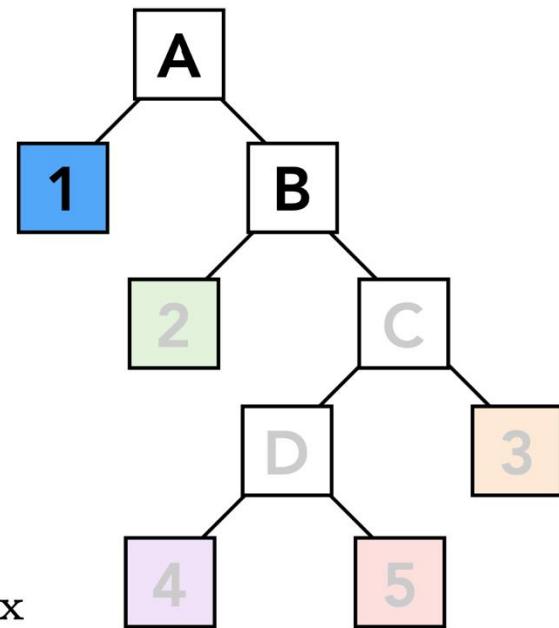
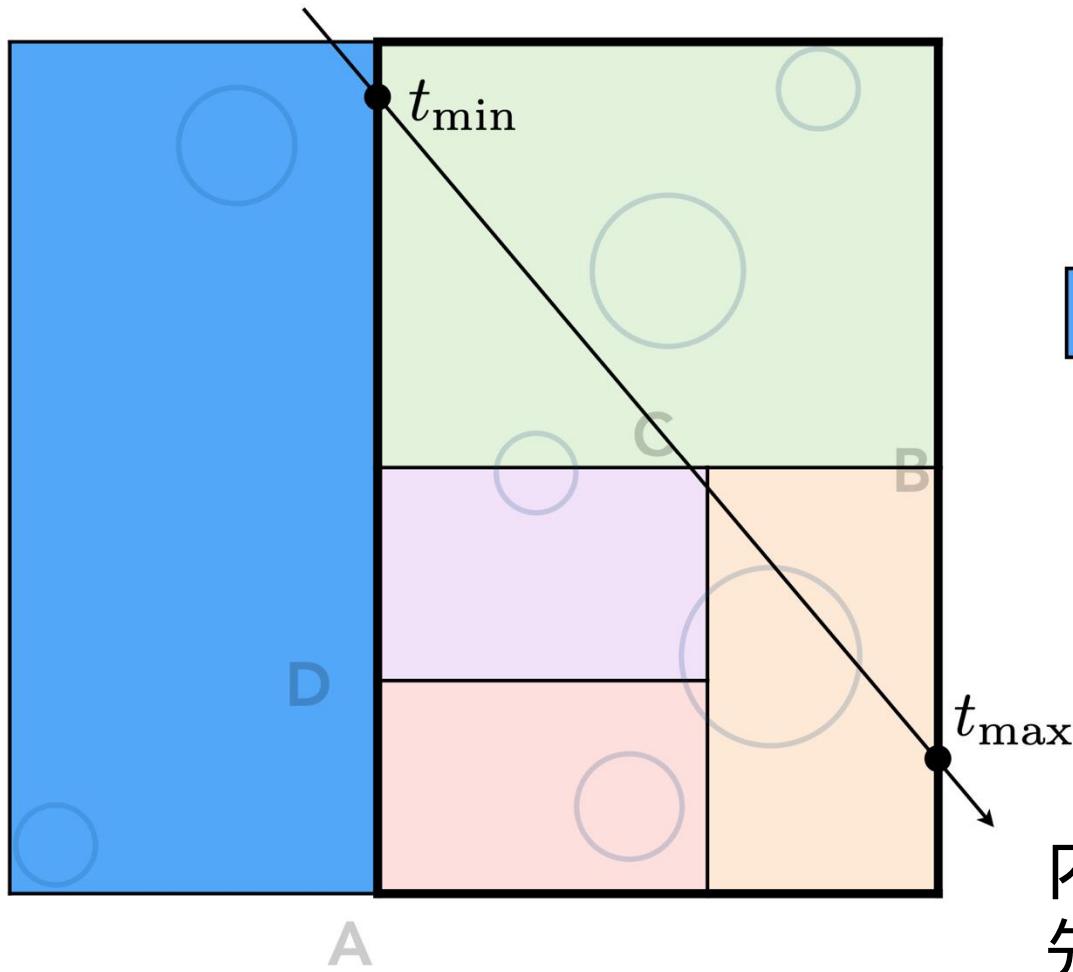
内部节点，选择先与  
光线相交的分支

# KD 树划分



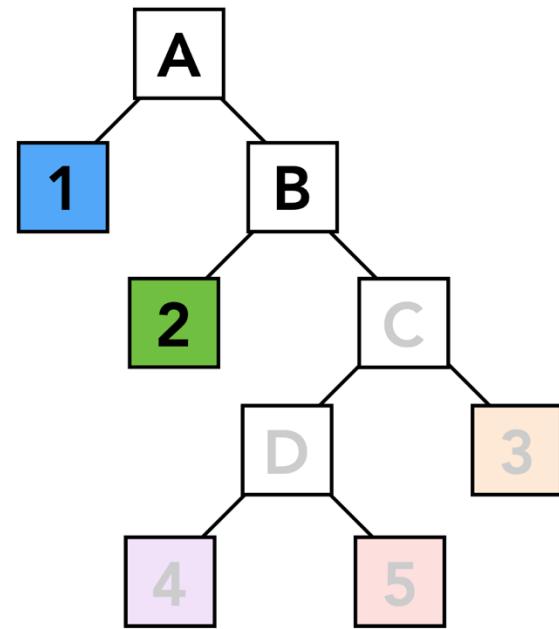
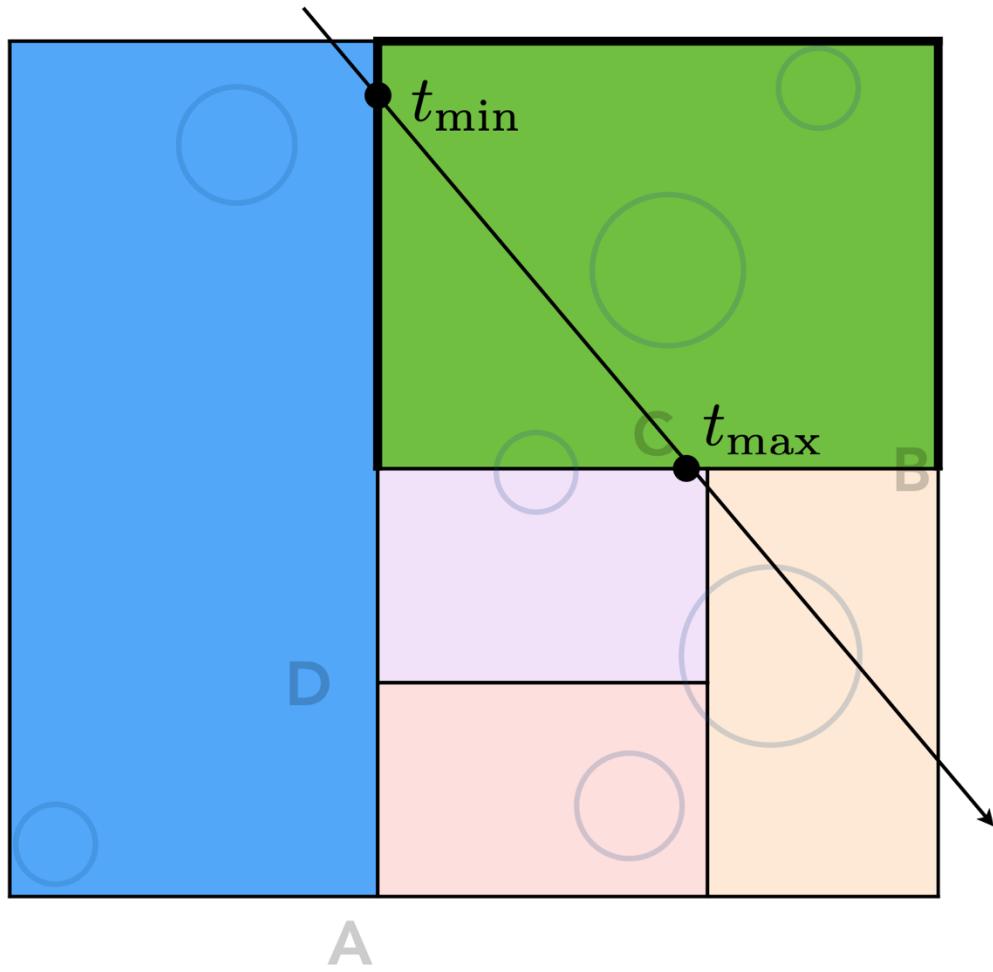
检查光线是否跟与 1  
接触的图像基元相交

# KD 树划分



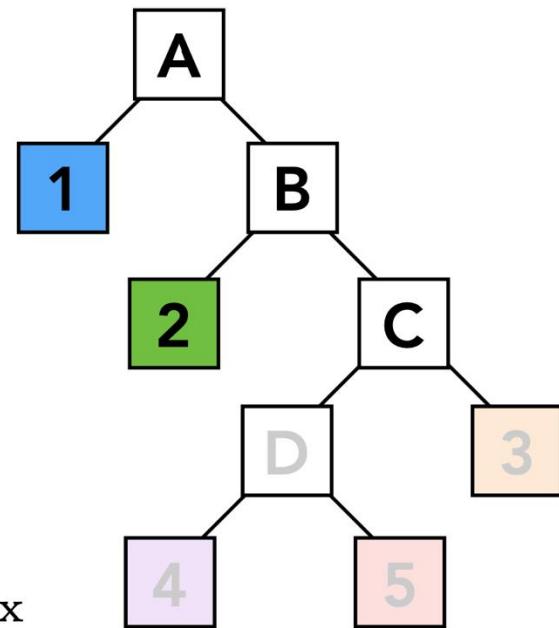
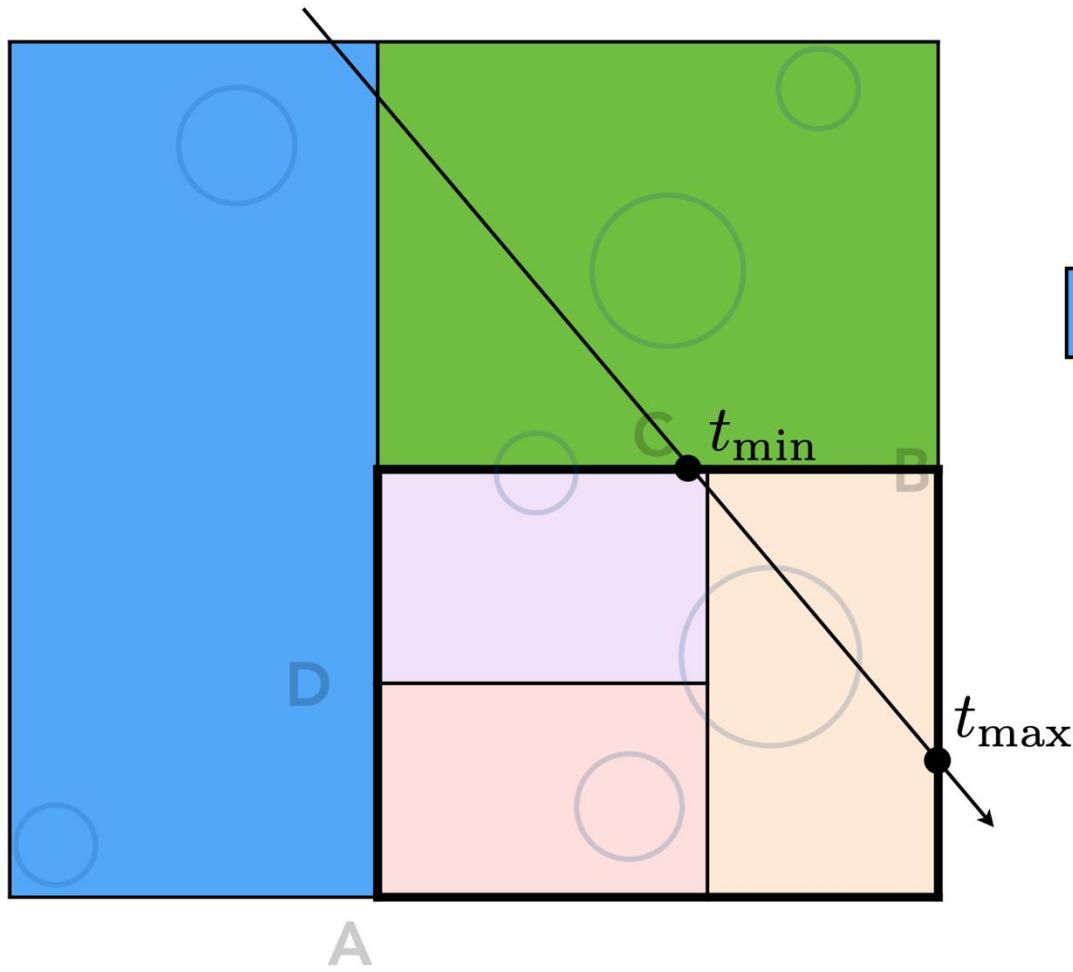
内部节点，同样选择  
先与光线相交的分支

# KD 树划分



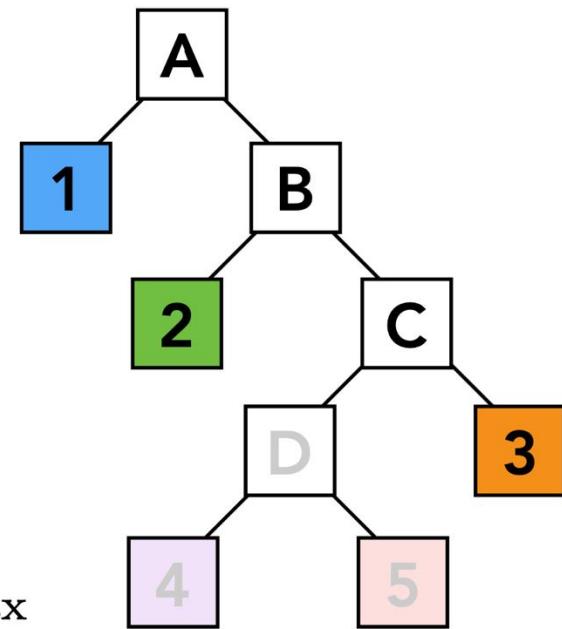
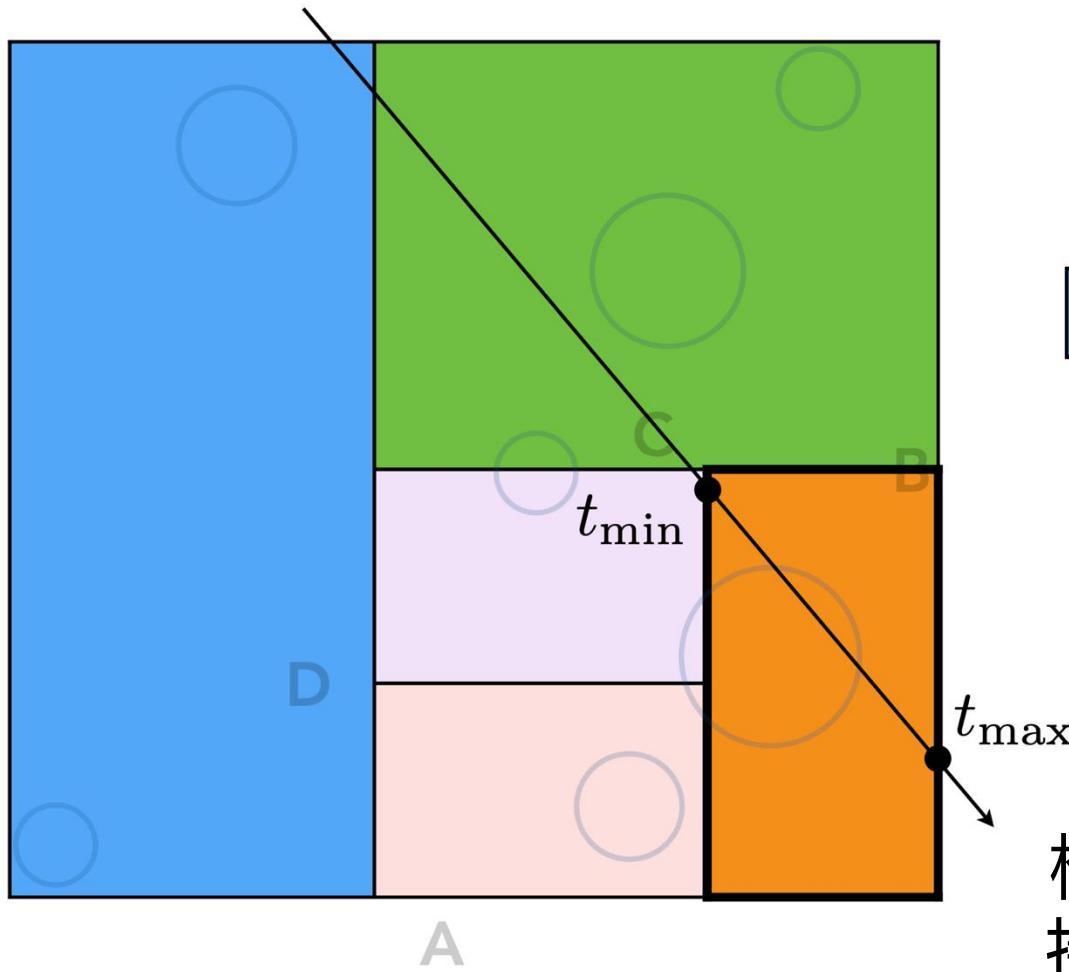
检查光线是否跟与 2  
接触的图像基元相交

# KD 树划分



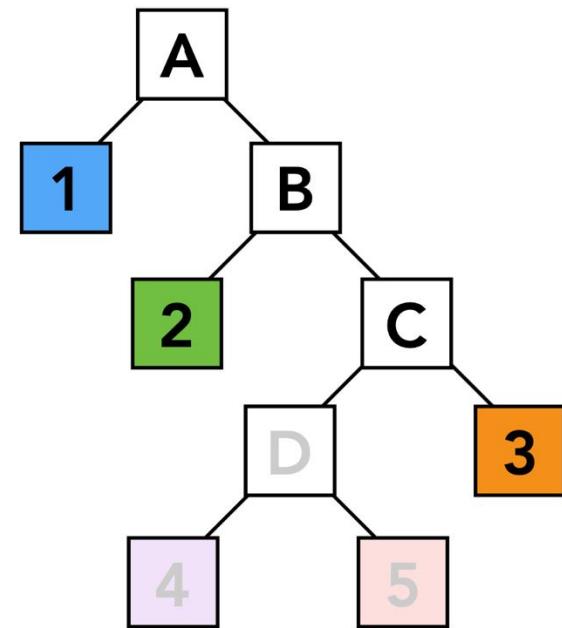
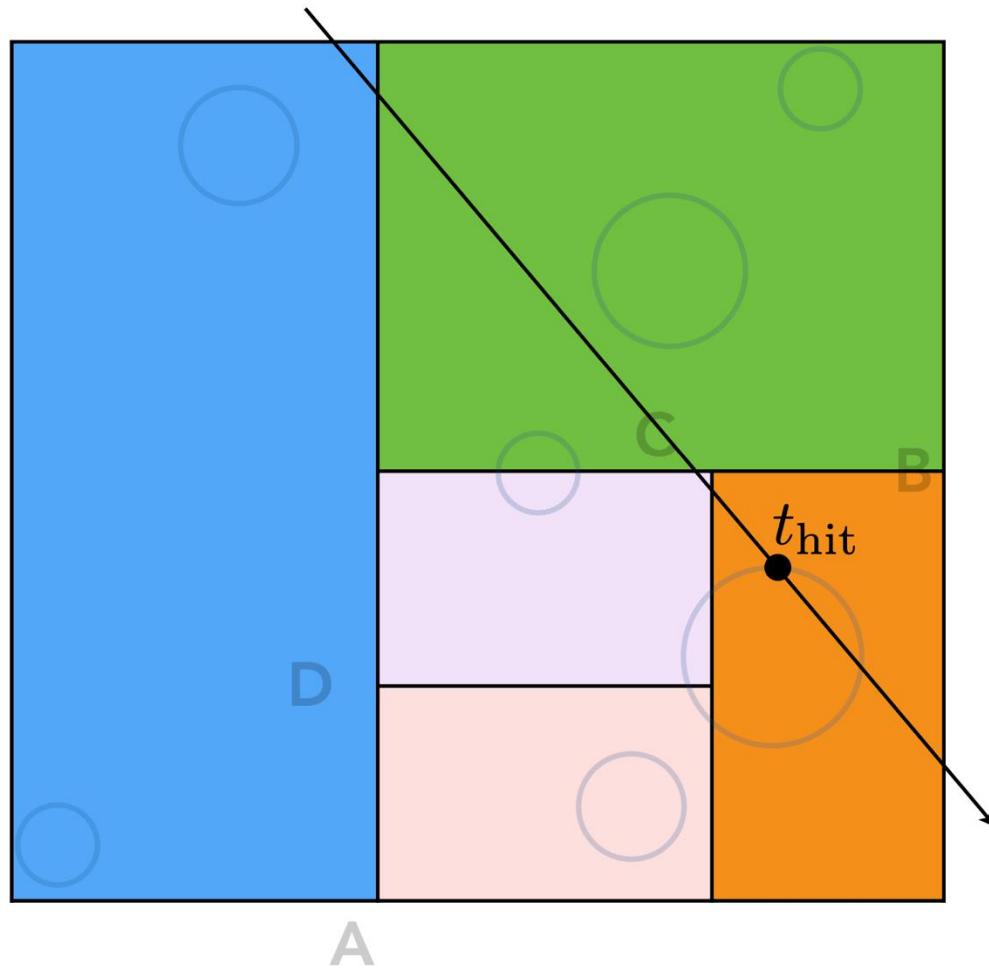
内部节点，进  
一步划分

# KD 树划分



检查光线是否跟与 3  
接触的图像基元相交

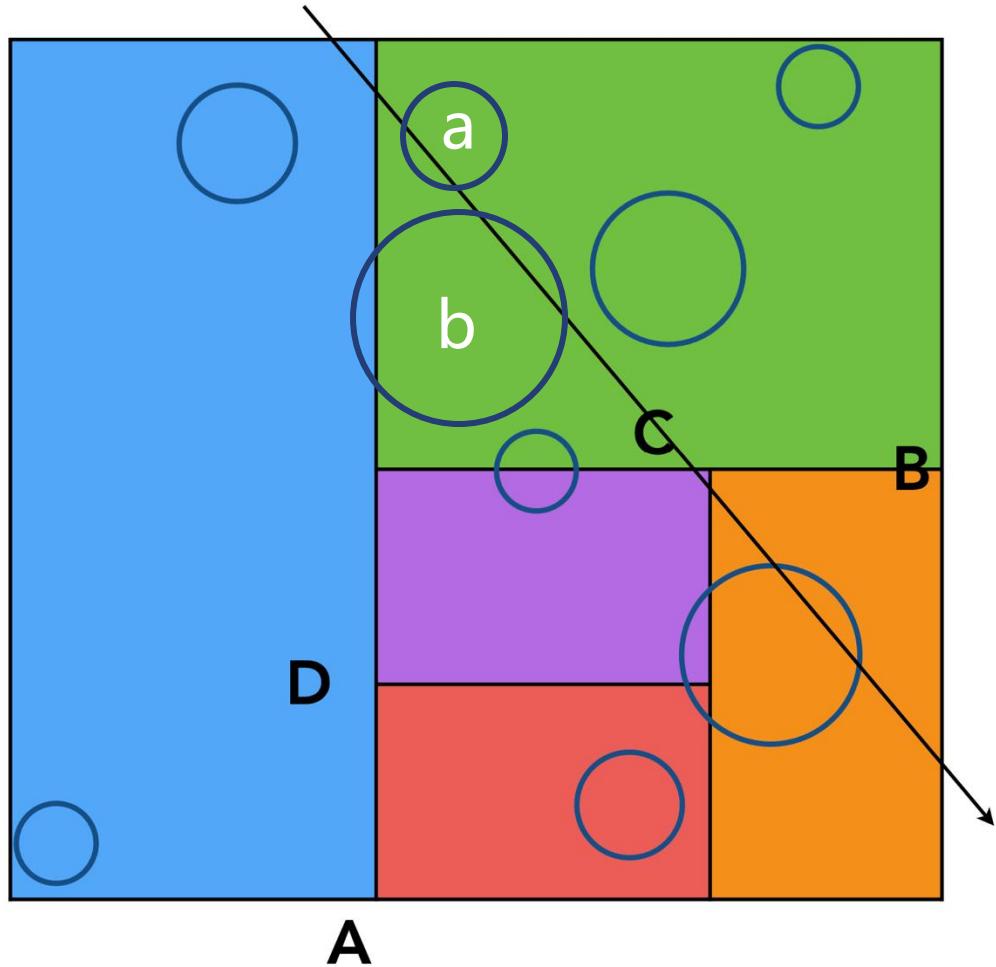
# KD 树划分



找到与光线相交  
的图像基元!

# KD 树划分的挑战

口在 KD 树中，一个图像基元可以被多个节点包含

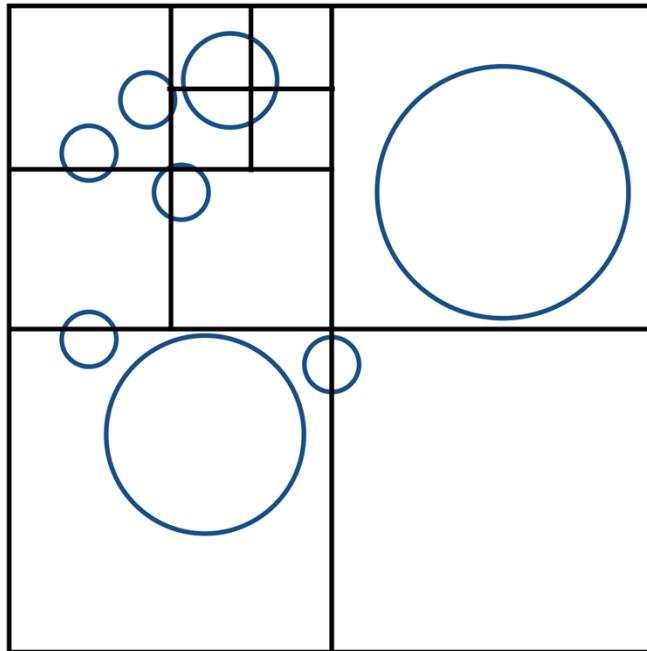


光线先与 a 还是 b 相交？

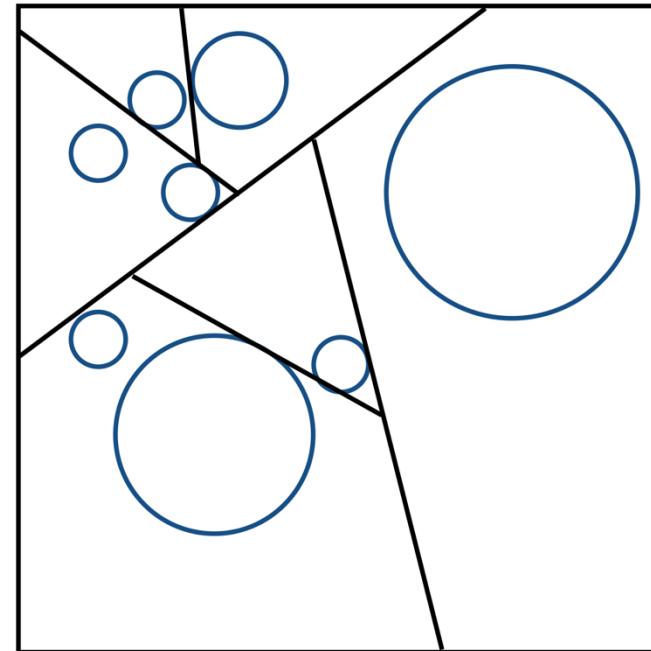
按照先前的算法，光线先穿过 1，检查与 1 接触的图像基元发现与 b 相交，然而实际上光线先穿过 a

解决方法：要求交点必须在相同区域内  
已经访问过的图像基元可以缓存下来

# 其他空间划分方法



Oct-Tree



BSP-Tree

# 不同加速结构的比较

结构	优点	缺点
均匀空间划分	<ul style="list-style-type: none"><li>1. 结构简单，易于实现</li><li>2. 访问速度快</li><li>3. 适合处理分布均匀的物体</li></ul>	<ul style="list-style-type: none"><li>1. 分布不均匀时，会有空格子，效率低下</li><li>2. 大对象需要跨越多个格子，增加计算复杂性</li></ul>
KD 树	<ul style="list-style-type: none"><li>1. 适合处理分布不均的场景</li><li>2. 查询效率高，特别是高维数据</li></ul>	<ul style="list-style-type: none"><li>1. 构建 KD 树较耗时</li><li>2. 动态场景效果不好，需要重新构建 KD 树</li></ul>
BVH 树	<ul style="list-style-type: none"><li>1. 适合处理任意分布的物体</li><li>2. 动态场景效果较好，更新成本低</li><li>3. BVH 与 GPU 并行架构匹配度高</li></ul>	<ul style="list-style-type: none"><li>1. 构建 BVH 树较耗时</li><li>2. 对于特定查询，如找最近物体，效率可能不如 KD 树</li></ul>

# 空间加速结构总结： 选择合适的结构来完成任务！

□ 基元分区 vs. 空间分区：

- 基元分区：将一组物体进行分割
  - BVH 节点数量有限
  - 如果场景中的物体位置发生变化，更新起来比较简单
- 空间分区：将空间进行分割
  - 按顺序遍历空间（第一个相交点就是最近的相交点）
  - 可能会多次与同一个物体相交

□ 自适应结构（如 BVH、K-D 树）：

- 构建成本较高（需要通过多次几何查询来摊销成本）
- 在物体分布不均匀的情况下，具有更好的相交查询性能

□ 非自适应加速结构（如均匀网格）：

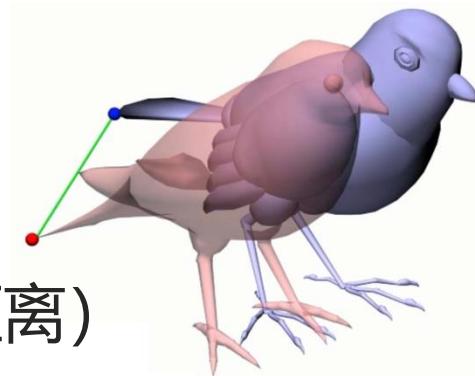
- 简单且构建成本低
- 如果场景中的物体分布均匀，则具有良好的相交性能

□ 这其中有很多很多的组合方式

# 图形学中的层次加速

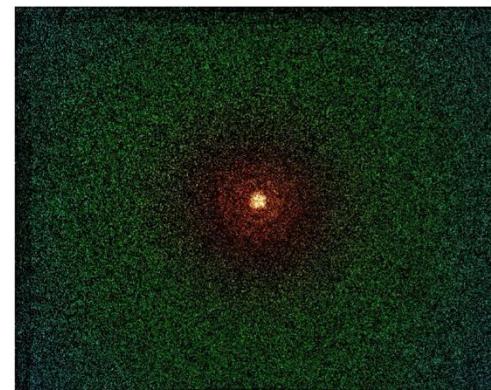
## 口几何 Geometry

- 判断物体内部或外部 (例如, 生成网格)
- 最近点测试 (如计算两个物体之间的最大距离)



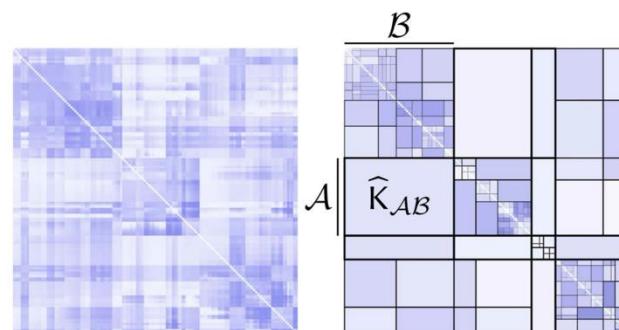
## 口动画/模拟 Animation/Simulation

- 粒子系统
- 多体动力学, 流体模拟
- Barnes-Hut 算法
- 快速多极子方法



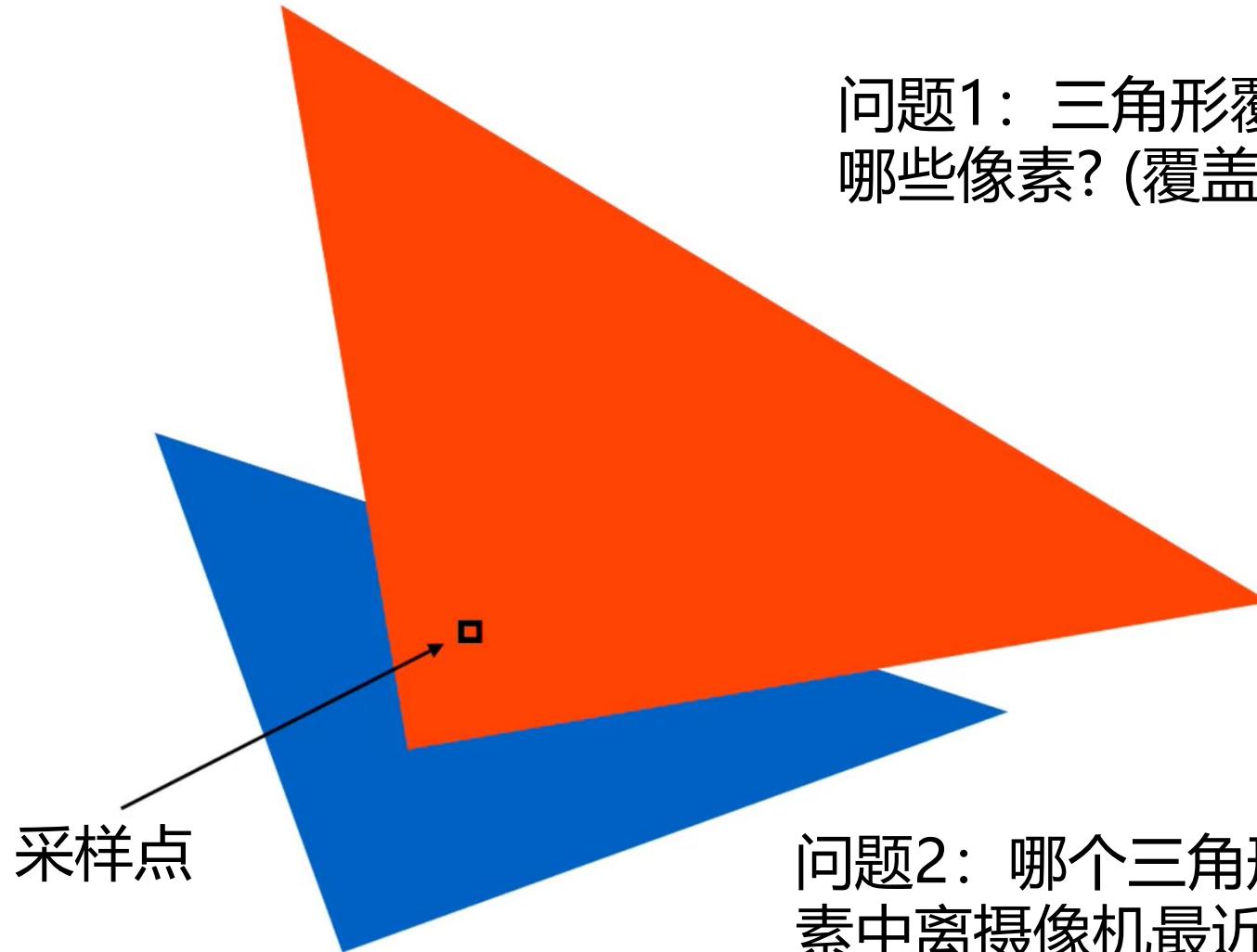
## 口渲染 Rendering

- 可见性计算
- 基于物理的光线追踪



我们如何利用光线相  
交查询来生成图像？

# 回顾三角形可见性问题



我们先前通过光栅化加  
深度缓冲来解决这个问题

但我们也可以通过光线查询来实现！

# 基本光栅化算法

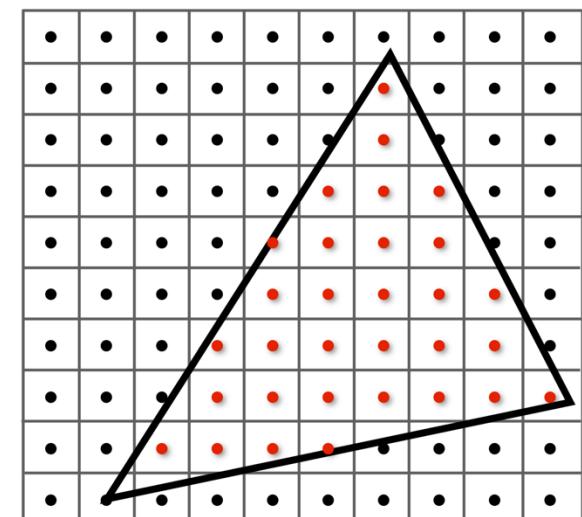
对于每一个三角形，找到它所覆盖的样本点

**Sample = 2D point**

**Coverage: 2D triangle/sample tests (does projected triangle cover 2D sample point?)**

**Occlusion: depth buffer**

```
initialize z_closest[] to ∞.           // store closest-surface-so-far for all samples
initialize color[]
for each triangle t in scene:          // loop 1: triangles
    t_proj = project_triangle(t)
    for each 2D sample s in frame buffer: // loop 2: visibility samples
        if (t_proj covers s)
            compute color of triangle at sample
            if (depth of t at s is closer than z_closest[s])
                update z_closest[s] and color[s]
```



# 基本光线追踪算法

□ 对于每一个样本点，找到被它覆盖的三角形基元

**Sample = a ray in 3D**

**Coverage: 3D ray-triangle intersection tests (does ray “hit” triangle)**

**Occlusion: closest intersection along ray**

```
initialize color[]                                // store scene color for all samples
for each sample s in frame buffer:                // loop 1: visibility samples (rays)
    r = ray from s on sensor through pinhole aperture
    r.min_t = ∞                                     // only store closest-so-far for current ray
    r.tri = NULL;
    for each triangle tri in scene:                // loop 2: triangles
        if (intersects(r, tri)) {                   // 3D ray-triangle intersection test
            if (intersection distance along ray is closer than r.min_t)
                update r.min_t and r.tri = tri;
        }
    color[s] = compute surface color of triangle r.tri at hit point
```

□ 两种方案都采样了加速技术

- 光栅化：将测试点限制在三角形的包围盒内
- 光线追踪：使用层次加速

# 基本的光栅化 vs. 光线追踪

## □ 光栅化 (Rasterization) :

- 按照三角形的顺序处理
- 使用深度缓冲区来记录深度信息 (可以随意访问固定大小的结构)
- 不需要把整个场景存入内存，因此可以自然支持无限大的场景

## □ 光线投影 (Ray casting) :

- 按照屏幕像素的顺序处理
  - 不需要为整个屏幕存储最近的深度信息 (只记录当前光线的信息)
  - 天然适合渲染透明表面 (按光线遇到的顺序处理表面：从前到后或从后到前)
- 必须存储整个场景
- 性能更依赖于场景中物体的分布

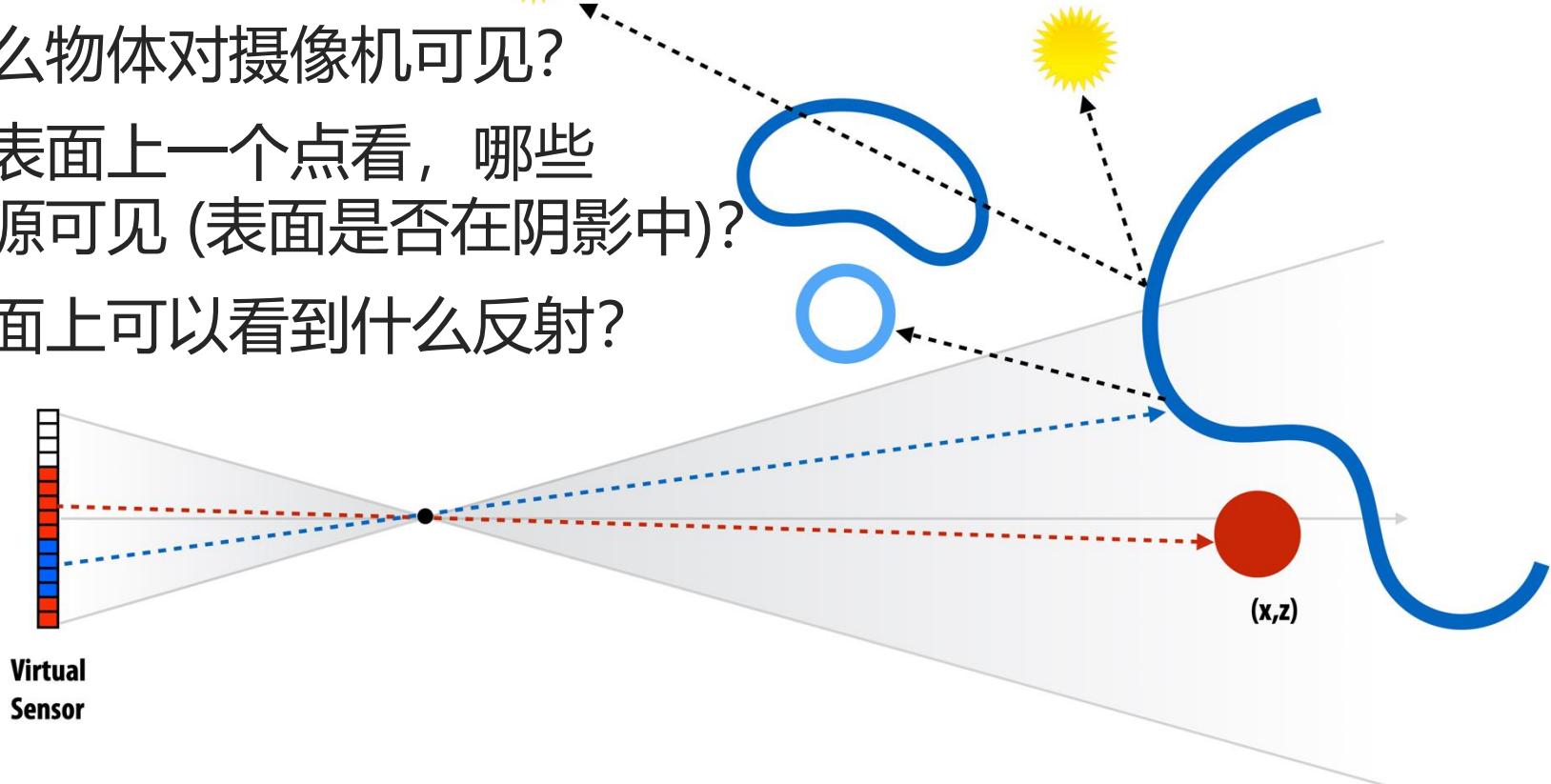
## □ 高性能实现中使用的类似技术：

- 光线/样本的层次结构
- 几何体的层次结构
- 延迟着色
- ...

# 这里有一个重要的区别...

光线投影可以用在非常多的任务上：

- 什么物体对摄像机可见？
- 从表面上一个点看，哪些光源可见（表面是否在阴影中）？
- 表面上可以看到什么反射？



相比之下，光栅化是一种高度专门化的解决方案，用于计算从同一点（通常是摄像机）发出的均匀分布光线的可见性



中山大學 软件工程学院  
SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

谢谢

陈壮彬  
软件工程学院  
[chenzhb36@mail.sysu.edu.cn](mailto:chenzhb36@mail.sysu.edu.cn)