

# 消隐算法—图像/景物空间消隐

冯结青

浙江大学 CAD&CG国家重点实验室

# 主要内容

- 图像空间消隐
  - 区间扫描线算法
  - 层次z-buffer算法
- 景物空间消隐
  - 区域细分算法(Warnock)
  - Weiler-Atherton算法

# 消隐的基本概念

- 消隐 (隐藏线或面消除): 相对于观察者, 确定场景中哪些物体是**可见的**或**部分可见的**, 哪些物体是**不可见的**
- 消隐是图形学中非常重要的一个**基本问题**
- 消隐算法分类: 算法实现时所在坐标系
  - 图像空间消隐
  - 景物空间消隐

# 图像空间消隐

- 描述

```
for(图像中每一个像素) {  
    确定视线穿过的距离观察点的最近物体;  
    用适当的颜色绘制该像素;  
}
```

- 特点：在屏幕坐标系中进行的，生成的图像一般受限于显示器的分辨率
- 算法复杂度为 $O(nN)$ ： $n$ 为物体个数， $N$ 为像素个数
- 代表方法：z缓冲器算法、扫描线算法等

# 景物空间消隐

- 描述

```
for(世界坐标系中的每一个物体) {  
    确定未被遮挡的物体或者部分物体;  
    用恰当的颜色绘制出可见部分;  
}
```

- 特点：算法精度高，与显示器的分辨率无关，适合于精密的CAD工程领域
- 算法复杂度为 $O(n^2)$ ：场景中每一个物体都要和场景中其他的物体进行排序比较， $n$ 为物体个数
- 代表方法：背面剔除、表优先级算法等

# 景物和图形空间消隐方法的比较

- 理论上，景物空间算法的计算量 $O(n^2)$ 小于图像空间算法 $O(nN)$ (在场景中物体个数小于屏幕的像素数的前提下)
- 实际应用中通常会考虑画面的连贯性，所以图像空间算法的效率有可能更高
- 景物空间和图像空间的混合消隐算法

# 主要内容

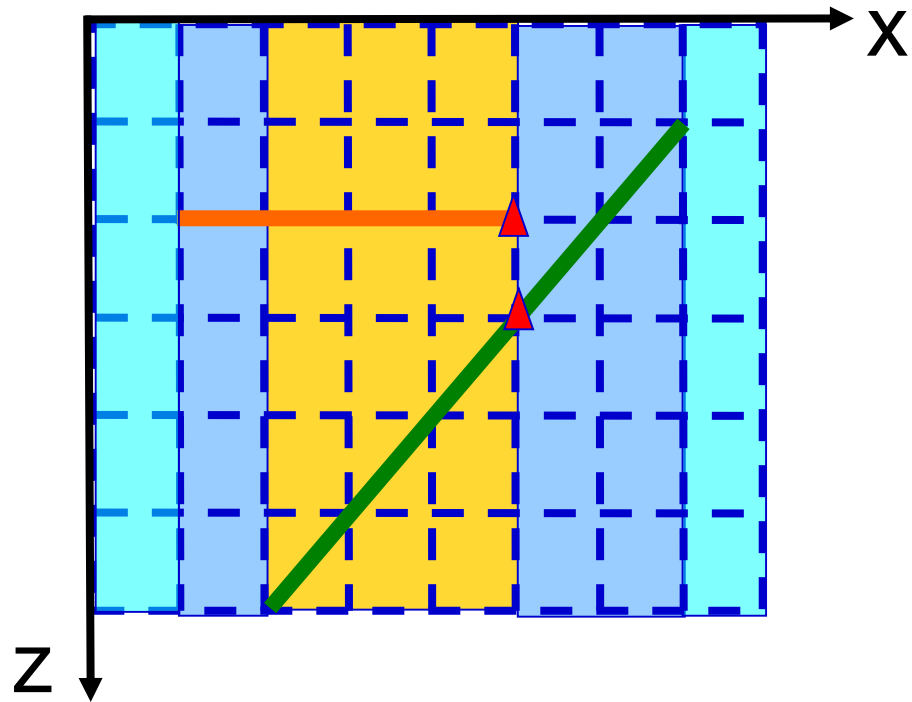
- 图像空间消隐
  - 区间扫描线算法
  - 层次z-buffer算法
- 景物空间消隐
  - 区域细分算法(Warnock)
  - Weiler-Atherton算法

# 区间扫描线算法

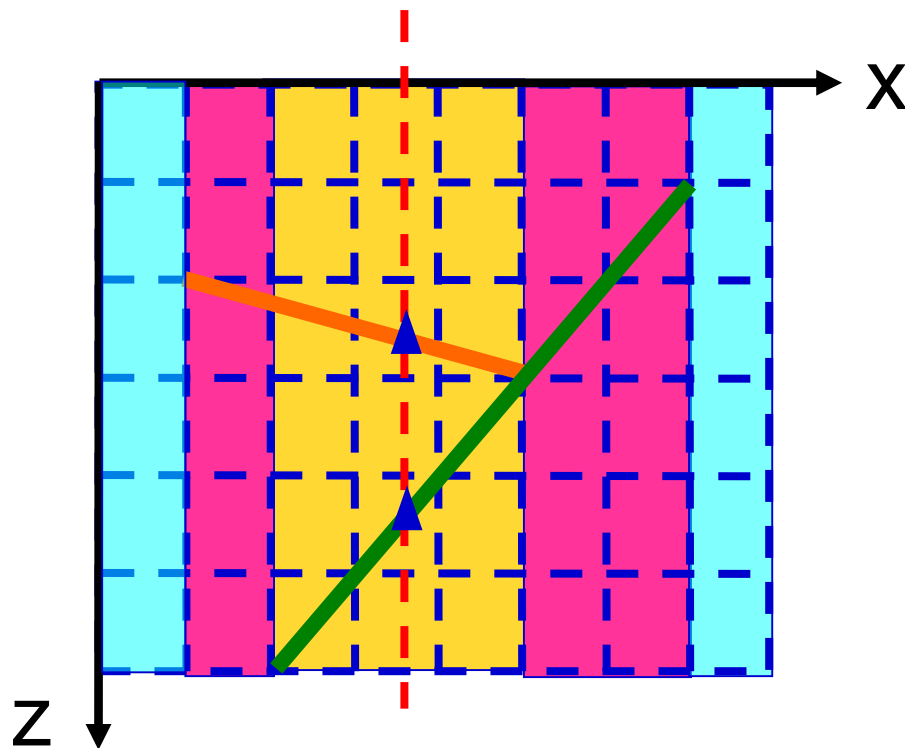
- 扫描线z-buffer算法还可以深入挖掘：
  - 扫描线内区间的连贯性
  - 多条扫描线之间的连贯性
  - 深度的连贯性
- 在扫描转换算法的基础上，引入所需数据结构，发掘并应用上述连贯性



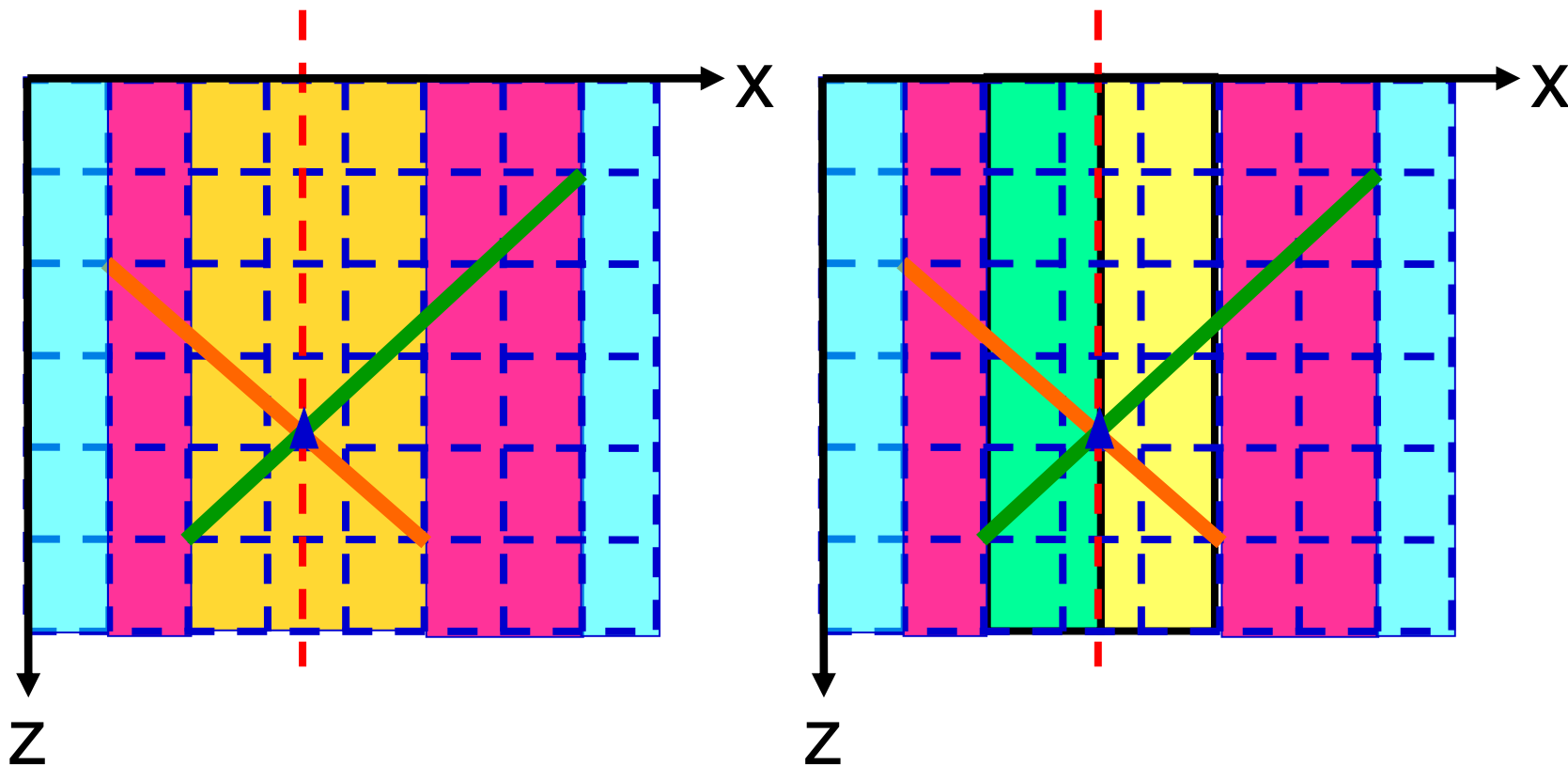
# 区间扫描线算法实例



# 区间扫描线算法实例

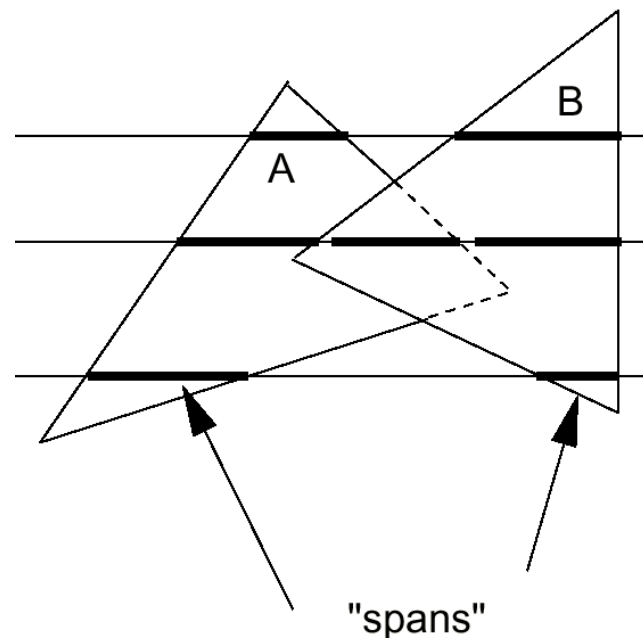


# 区间扫描线算法实例



# 区间扫描线算法的思想

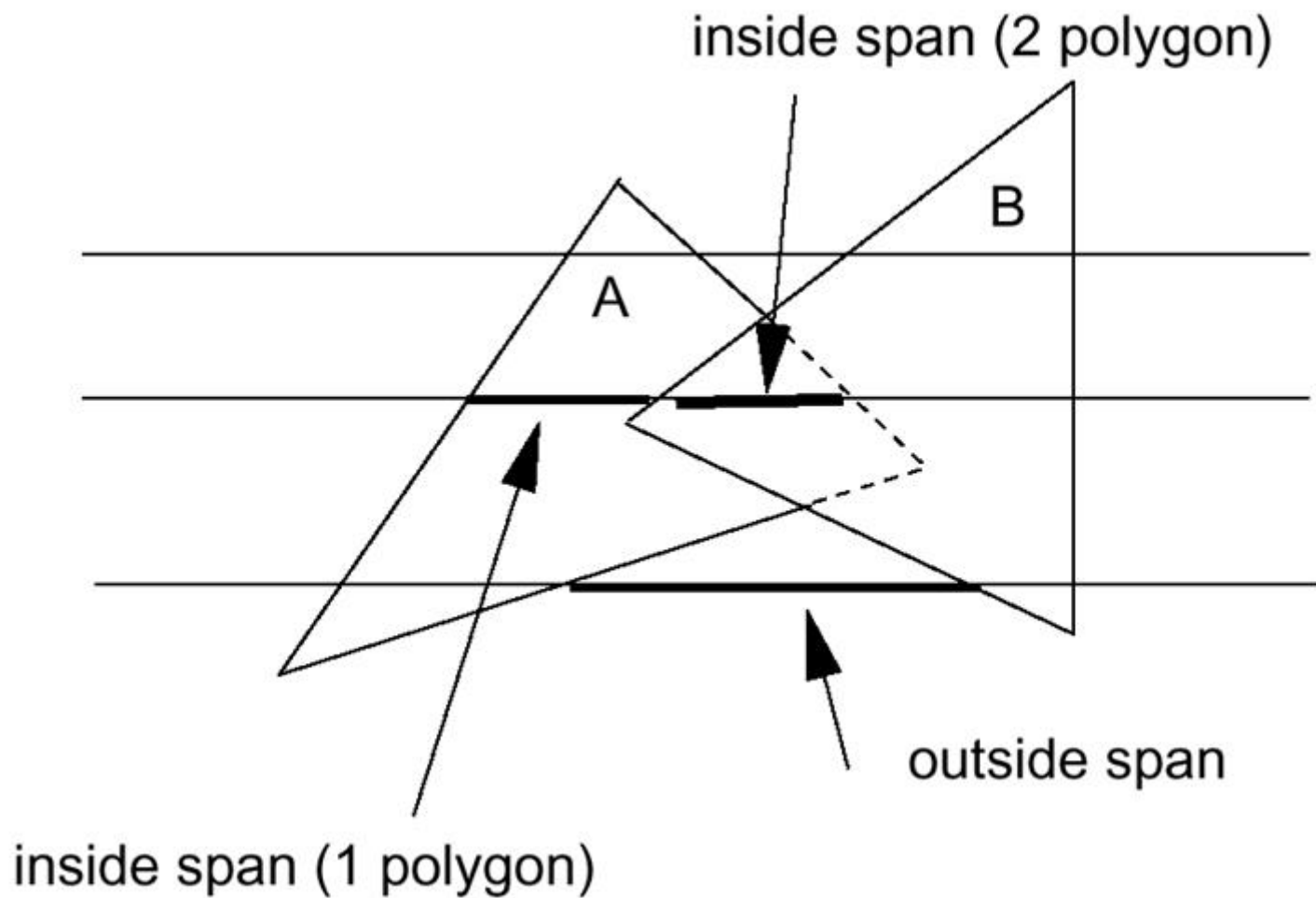
- 不采用z-buffer的逐点判断
- 每一条扫描线分割成几个区间
- 确定每一个区间属于哪个多边形
- 用多边形的颜色对离视点最近的区间着色
- 发掘区间的连贯性：对每一个区间只做一次可见性判断！



# 区间扫描线算法的思想

- 一条扫描线分解为一系列区间扫描线段
- 每一个区间扫描线段位于多边形“内部”和“外部”
  - “外部”扫描线区间：直接着背景颜色
  - “内部”扫描线区间：位于一个/多个多边形内部
    - 一个多边形内部：该线段上的像素着多边形的颜色；
    - 多个多边形内部：则比较扫描线段端点的 $z$ 值，以确定该线段上像素的颜色

# 区间扫描线算法



# 扫描线段与单个多边形内外关系

- 当扫描线与多边形的一条边相交时
  - 第一个交点(奇数)表示扫描线段进入多边形
  - 第二个交点(偶数)表示扫描线段离开多边形
- 对每一个多边形，用一个“in/out”标记符flag跟踪记录当前扫描线段的状态
  - 初始：标记符flag为out(e.g. flag=0)
  - 进入多边形：更新flag为in(e.g. flag=1)
  - 离开多边形：更新flag为out(e.g. flag=0)

# 扫描线段与多个多边形内外关系

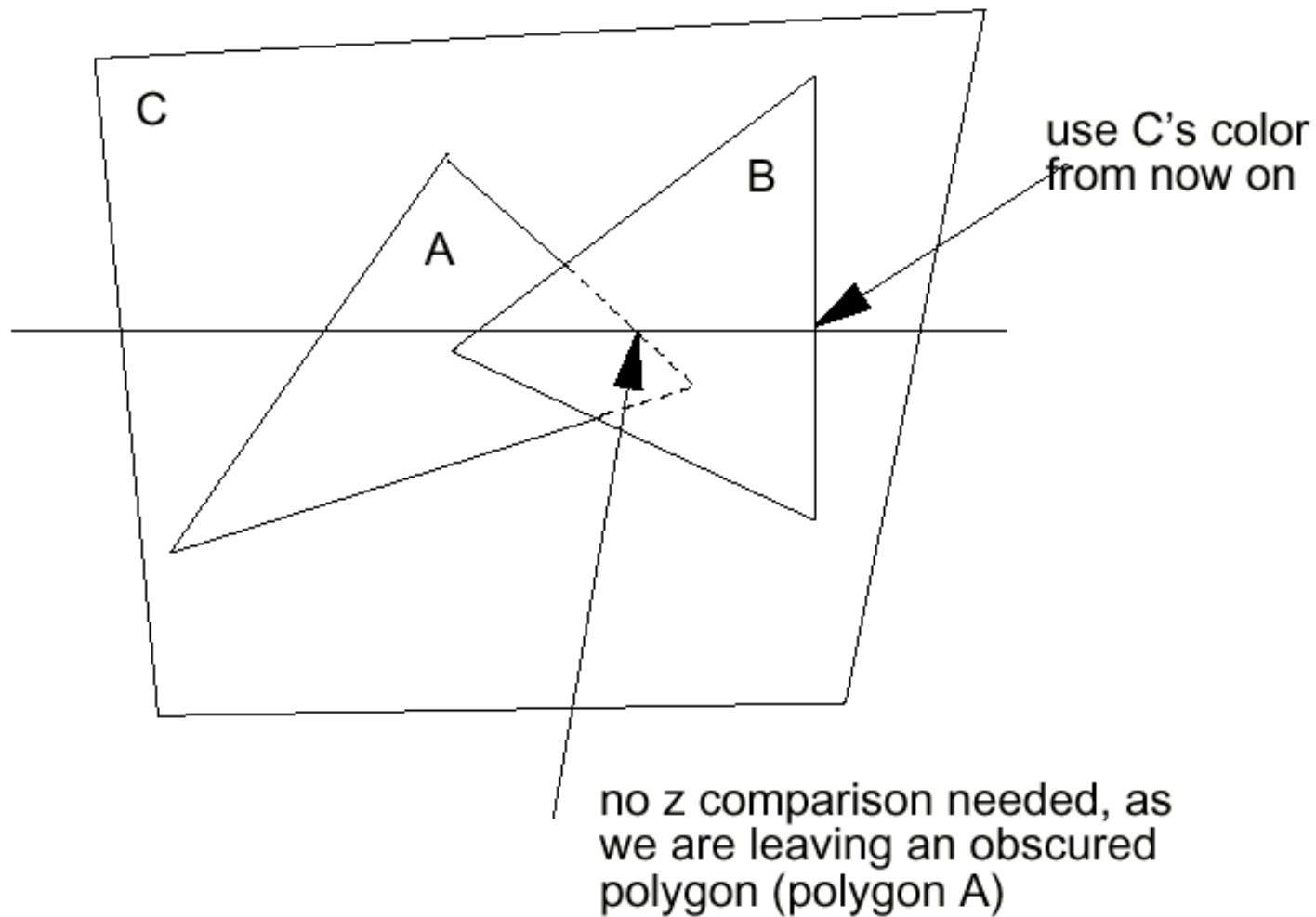
- 每一个多边形有其“in/out”标记符flag
  - 对于扫描线上一个给定区间，可能有多个多边形的flag值为“in”
- 在当前扫描线上，跟踪有多少个多边形的标记符为“in”
  - 如果有多于一个多边形flag的值为“in”，需要比较扫描线区间的z值以确定其颜色



# z-值比较

- 扫描线与一条边相交并且离开一个多边形
  - 如果剩余多边形中只有一个标记符flag值为“in”，那么就采用该多边形的颜色着色；
  - 如果离开的多边形为被遮挡多边形，则着色颜色保持不变；
  - 如果如果离开的多边形为非遮挡多边形，且剩余多边形有多个值为“in”，则需要进行z-值比较

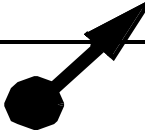
# z-值比较



# 相关数据结构

- ET: 边链表, 类似与扫描线z-buffer算法
- PT: 多边形链表, 每个条目对应一个多边形

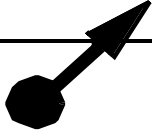
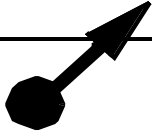
**ET**

x	$y_{\max}$	$\Delta x$	poly-ID	
---	------------	------------	---------	---

**PT**

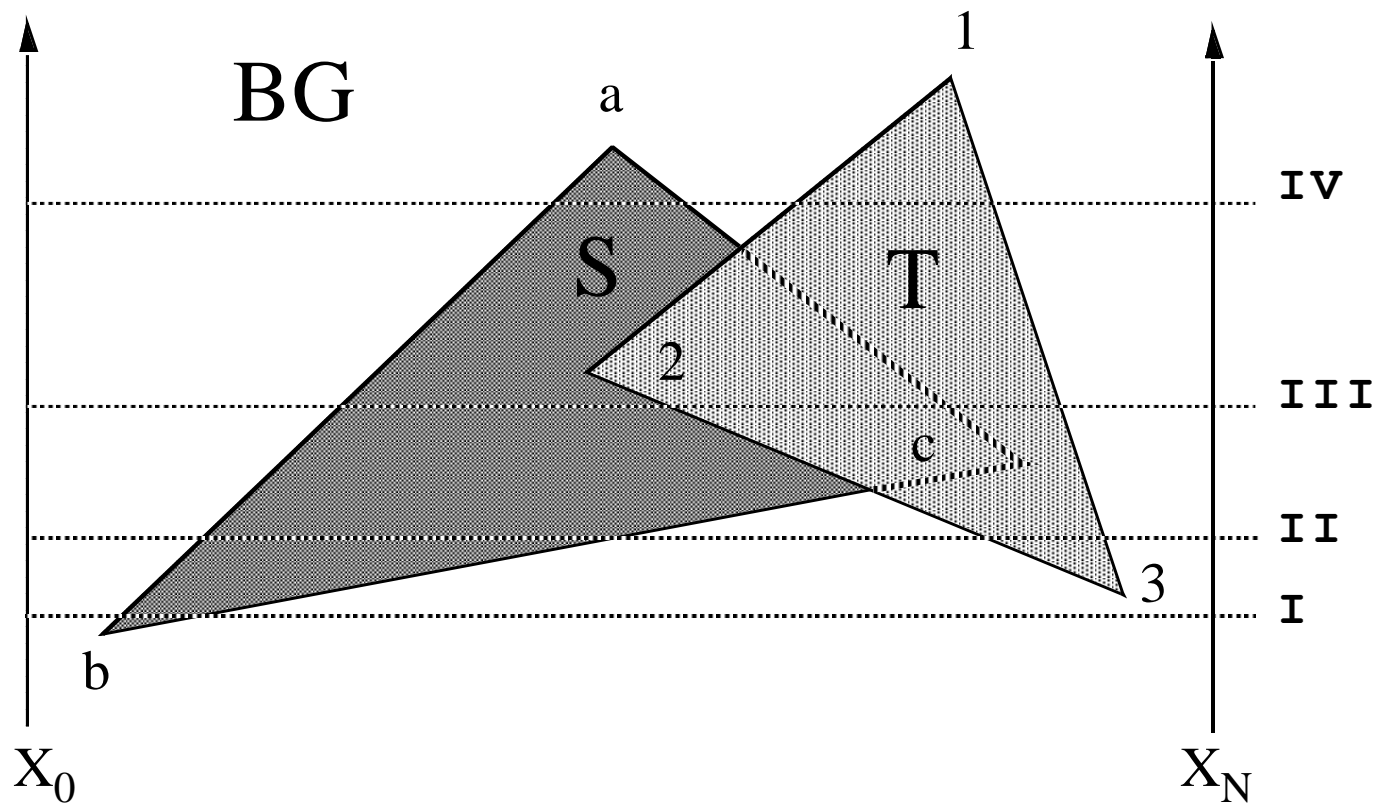
poly-ID	A,B,C,D	color	in/out flag	
---------	---------	-------	-------------	---

# 相关数据结构

<b>ET</b>	x	$y_{\max}$	$\Delta x$	poly-ID	
<b>PT</b>	poly-ID	A,B,C,D	color	in/out flag	

- 进入当前多边形时，其flag值为 “in”/“true”
  - 可以有多个多边形的flag值为 “in”/“true”
- 用IPL记录活化多边形表(active In-Polygon List)

# 例子



BG: 背景; S、T: 多边形, T部分遮挡S; abc、123: 分别为S、T的顶点; I、II、III和IV: 四条扫描线

# 区间扫描线实例

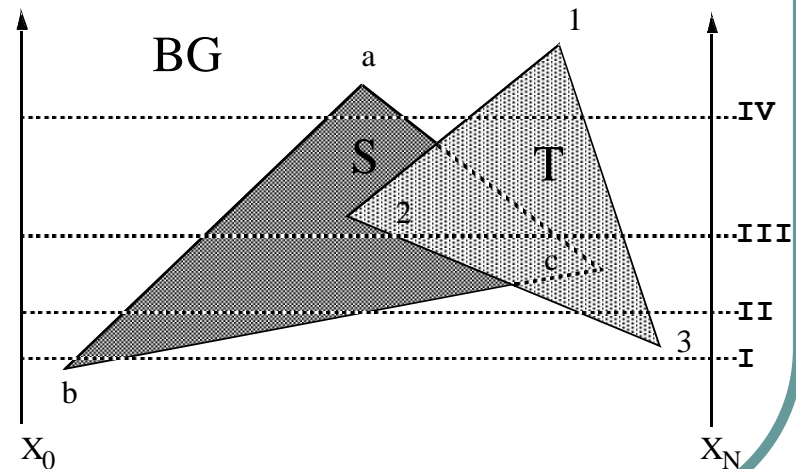
Y	AET	IPL
I	$x_0, ba, bc, x_N$	BG, BG+S, BG
II	$x_0, ba, bc, 32, 13, x_N$	BG, BG+S, BG, BG+T, BG
III	$x_0, ba, 32, ca, 13, x_N$	<i>BG, BG+S, BG+S+T, BG+T, BG</i>
IV	$x_0, ba, ac, 12, 13, x_N$	BG, BG+S, BG, BG+T, BG

I: S的标识符  $\text{flag}_S = \text{true}$

II: S和T的flag值均为true, 但扫描线段不重叠

III: S和T的flag值均为true, 扫描线段重叠

IV: 与扫描线II 情形相同



# 区间扫描线算法描述

```
build ET, PT          -- all polys+BG poly  
AET := IPL := Nil;  
for y :=  $y_{\min}$  to  $y_{\max}$  do  
    e1 := first_item ( AET ); IPL := BG;  
    while (e1.x <> MaxX) do  
        e2 := next_item (AET);  
        poly := closest poly in IPL at  $[(e1.x+e2.x)/2, y]$   
        draw_line(e1.x, e2.x, poly-color);  
        update IPL (flags); e1 := e2;  
    end-while;  
    IPL := NIL; update AET;  
end-for;
```

# 深度连贯性

- 多条扫描线间的深度连贯性：从一条扫描线跨入另一条扫描线时，深度关系不一定改变
- 实现：通过跟踪AET和PT表实现

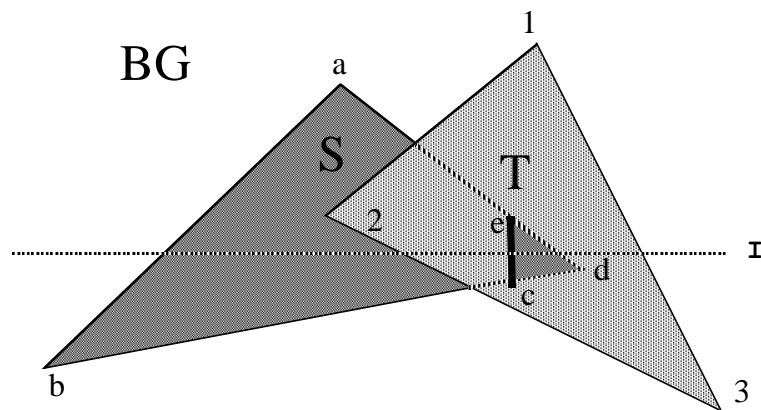


# 贯穿多边形

Y	AET	IPL
I	$x_0, ba, 23, ad, 13, x_N$	$BG, BG+S, BG+S+T, BG+T, BG$
I'	$x_0, ba, 23, \underline{ec}, ad, 13, x_N$	$BG, BG+S, BG+S+T, \mathbf{BG+S+T}, BG+T, BG$

扫描线段间遮挡关系发生改变

- 避免贯穿：多边形求交与裁剪
- 处理贯穿：扫描线段求交



# 贯穿多边形

- 记两个扫描线区间的深度值分别为 $(z_{1l}, z_{1r})$ 和 $(z_{2l}, z_{2r})$ ，判断如下关系  
 $\text{Sign}(z_{1l} - z_{2l})$  和  $\text{Sign}(z_{1r} - z_{2r})$ 
  - 同号：两个线段不相交
  - 为零：线段的一个端点重合
  - 异号：两个线段相交，计算交点，分别将两个扫描线段一分为二

# 区间扫描线算法

- 充分发挥
  - 扫描线上的(深度)连贯性：逐点深度判断→区间深度判断
  - 扫描线之间的(深度)连贯性：多边形的flag和边表排序关系未发生改变时
- 改进后的算法可以处理贯穿情形

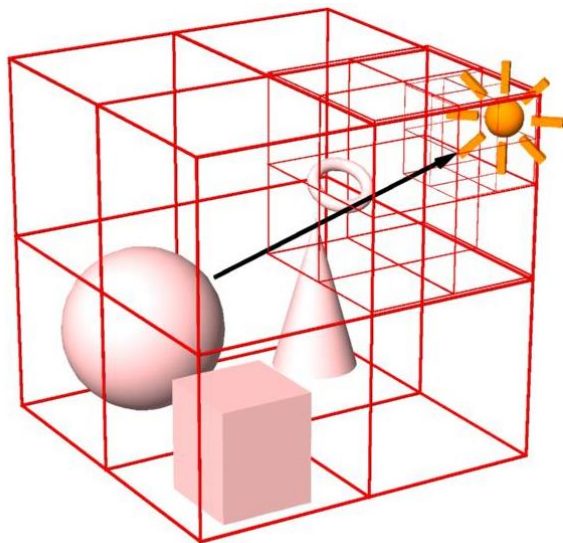
# 主要内容

- 图像空间消隐
  - 区间扫描线算法
  - 层次z-buffer算法
- 景物空间消隐
  - 区域细分算法(Warnock)
  - Weiler-Atherton算法

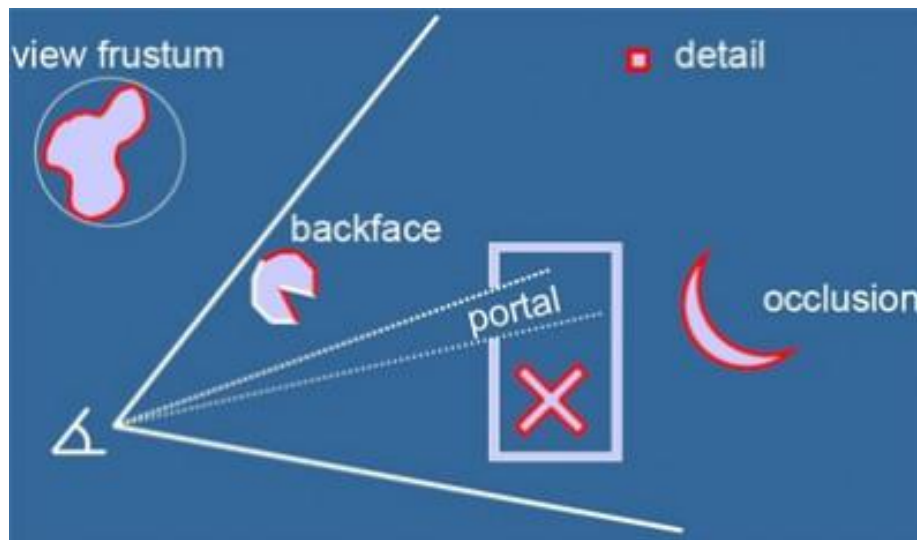
# 层次z-buffer算法

- 原则：不绘制 “看不见” 和 “被遮挡” 的物体
- 方法：“快速拒绝”
  - 景物深度**排序**：八叉树...
  - 景物**剔除**(Culling)：背面剔除、视域四棱台剔除...

# 空间排序和各种剔除



场景物体的八叉树  
剖分



场景中潜在不可见  
物体的快速剔除

如何快速拒绝被遮挡物体？

# 已有方法存在的问题

- z-buffer: 记录每个像素的深度值
- 基于场景空间八叉树剖分的预排序
  - 将节点在投影屏面上投影: 很少有节点能被一个像素所遮挡
  - 逐像素地快速拒绝判断效率低
- 场景: 森林中树木? 商场中拥挤的人群?

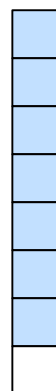
# 层次z-buffer算法

- 进一步发掘各种连贯性
  - **图像空间连贯性**：被一个像素遮挡的物体，也倾向于被其邻近像素遮挡；
  - **景物空间连贯性**：位于遮挡物附近的物体也倾向于被遮挡；
  - **时间连贯性**：动态场景中，当前帧中被遮挡的物体在下一帧中倾向于被遮挡



# 层次z-buffer算法

- 将z-buffer替换成层次z-buffer
  - 最底层：全分辨率z-buffer
  - 上一层：每一个像素是当前层四个像素深度的最小值(离视点最远)
- 层次z-buffer为一棵四叉树：MIP-Map z-buffer

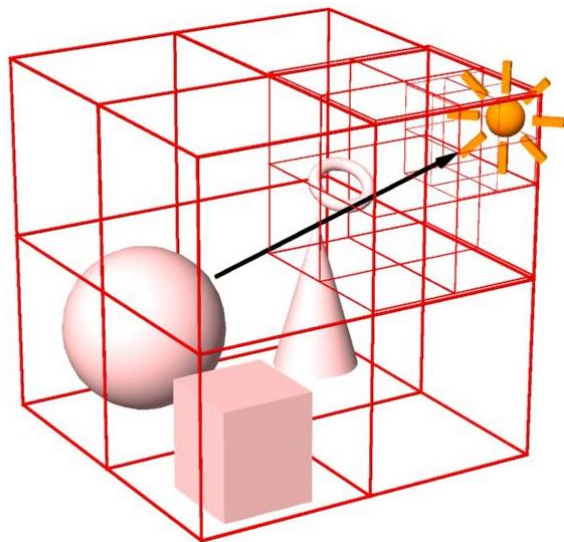


-.2	-.3	-.5	-far
-.3			
-.4	-.5		
-.5			
-.6	-.7	-far	
-.7			
-.8	-far		
-far			

hierarchical zbuffer

# 层次z-buffer算法

- 场景：八叉树



- 算法的基本思想：

- 层次化光栅化场景
- 当遇到被遮挡的物体时，尽可能早地停止光栅化

# 基本的层次z-buffer算法

- 在光栅化一个多边形之前
  - 比较多边形离视点最近的z-值与四叉树(层次z-buffer)中包含此多边形的最小节点中的z-值;
  - 如果z-测试失败, 则整个多边形被遮挡, 不必光栅化;
  - 否则, 与四叉树的下一层节点进行z-测试
  - 如果在最底层的z-测试仍然通过, 则光栅化多边形并更新层次z-buffer

# 场景的八叉树递归剖分

- 选择场景物体的包围盒作为根立方体
- 递归剖分立方体
  - 如果立方体中包含的多边形很少，则这些多边形属于该立方体，并退出；
  - 否则
    - 与至少一个分割面相交的多边形属于该立方体
    - 递归剖分立方体，并确定位于该立方体中的多边形与子立方体的归属关系

# 层次z-buffer算法描述

1. 初始化四叉树所有层次的节点深度为无穷远
2. 从场景八叉树根节点开始，选择恰当层次z-buffer，对八叉树节点(立方体的面)进行深度测试
  1. 如果测试失败，则整个节点被遮挡，停止
  2. 否则，剖分该八叉树节点
3. 如果叶节点没有通过深度测试
  1. 将叶节点中的物体绘制到z-buffer中
  2. 根据新的z-buffer更新层次z-buffer

# 层次z-buffer算法中的时间连贯性

- 层次z-buffer中
  - 在上一帧中影响z-buffer的多边形，在这一帧中也倾向于影响z-buffer
  - 当层次z-buffer已经建立时，层次z-buffer算法的效率最高！
- 在每一帧中，首先绘制上一帧可见的八叉树节点！

# 层次z-buffer算法中的实现

- 硬件实现层次z-buffer算法更高效
- 硬件开始支持
  - 深度拒绝测试 (z-query)
  - 物体空间连贯性(包围盒)
  - 时间连贯性(最后绘制的列表)

# 主要内容

- 图像空间消隐
  - 区间扫描线算法
  - 层次z-buffer算法
- 景物空间消隐
  - 区域细分算法(Warnock算法)
  - Weiler-Atherton算法



# 区域细分算法(图像空间算法)

- 分而治之(Divide and Conquer): 把一个复杂的问题分解为一系列简单的问题, 对简单问题求解, 然后将结果合成
- 区域细分算法(Warnock算法): 把物体投影到全屏幕窗口上, 然后递归分割窗口(不分割多边形), 直到窗口内目标足够简单, 可以显示为止

# 区域细分算法(Warnock算法)

将场景中的多边形投影到初始窗口内；

判断投影多边形与窗口的关系；

如果足够简单或者窗口为一个像素大小

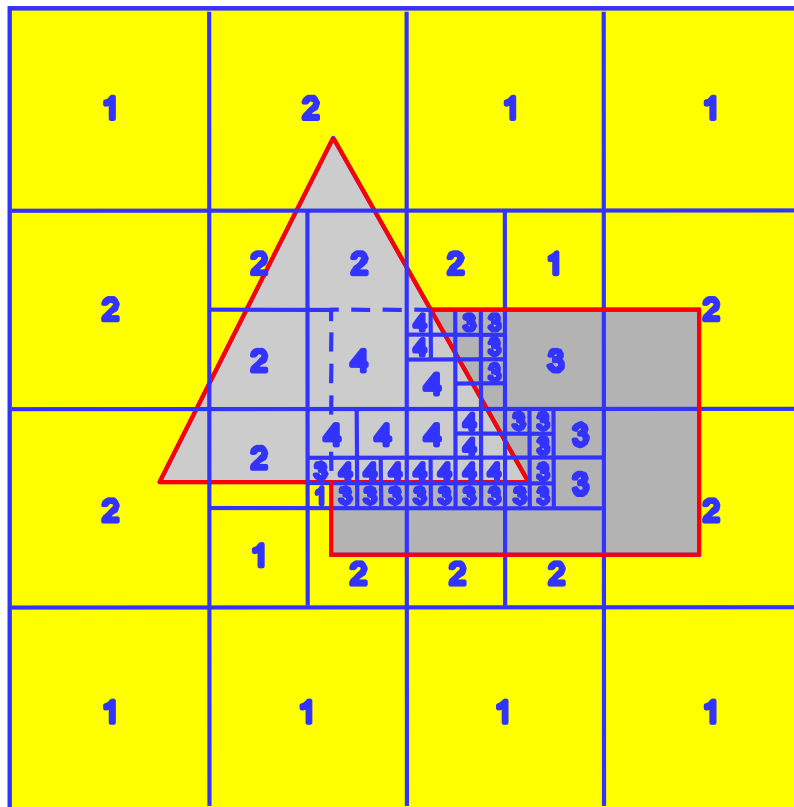
绘制该窗口内的多边形；

否则

把窗口等分成四个子窗口；

递归执行上述步骤；

# 区域细分算法举例

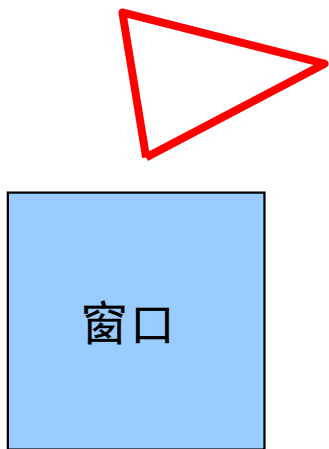


图中1、2、3、4表示多边形与窗口的关系，见下一页  
多边形与窗口关系分类。

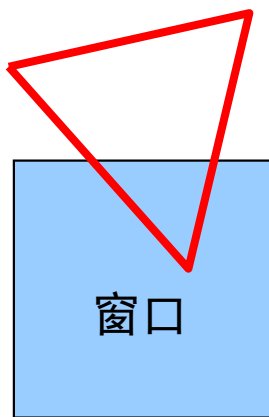
# 窗口和投影多边形的关系

- 窗口内目标足够简单？
  1. 所有多边形都和窗口分离
  2. 只有一个多边形和窗口相交或包含在窗口内
  3. 只有一个多边形把窗口整个包围
  4. 有几个多边形和窗口有交叠，最靠近观察者的多边形包围整个窗口

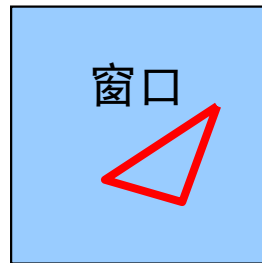
# 窗口和投影多边形的关系



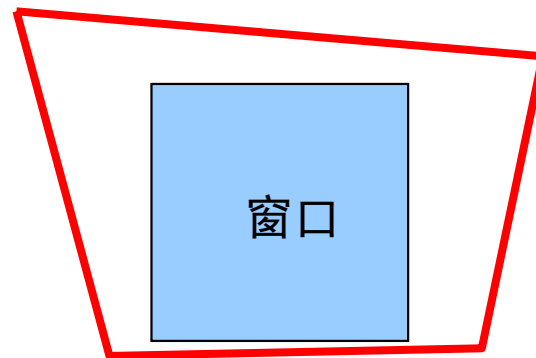
分离



相交

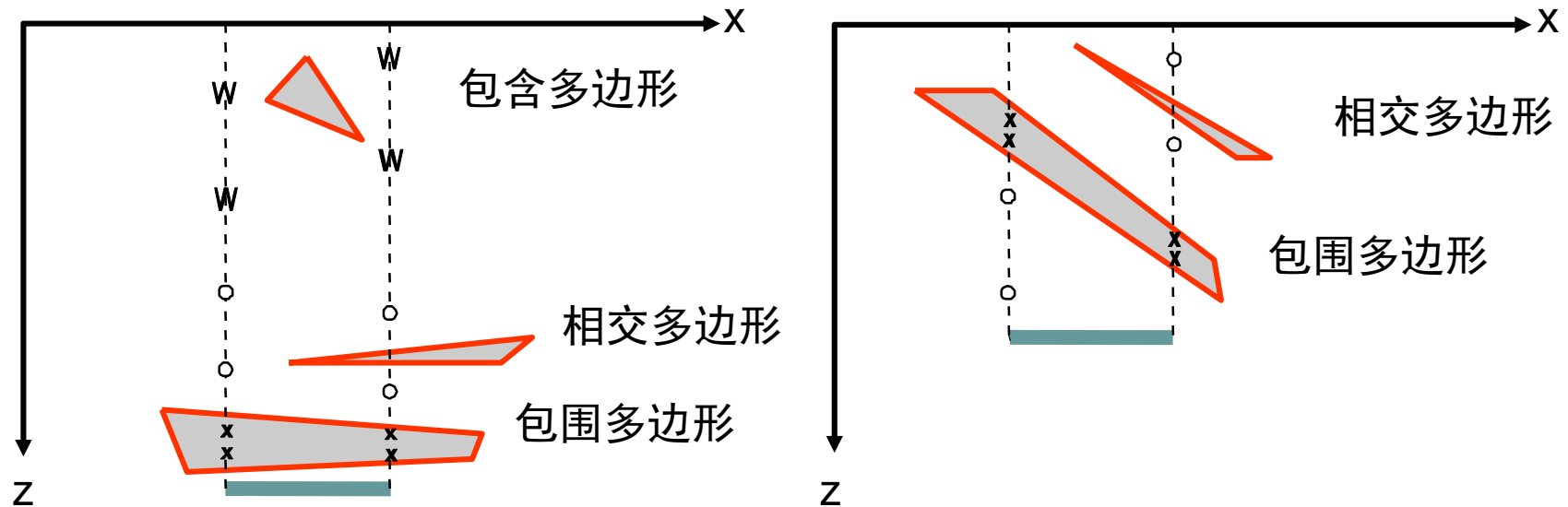


包含



包围

# 窗口和投影多边形的关系



# 区域细分算法的讨论

- 利用包围盒技术判断分离关系
- P不与窗口相交，P不与其子窗口相交
- P包含窗口，P包含其子窗口
- P包含窗口，被P遮挡的多边形可以在子窗口的多边形序列中删除
- 二维多边形裁减技术判断相交关系

# 区域细分算法的进一步发展

- 区域细分法：分割窗口
- Weiler-Atherton算法：用多边形来分割窗口，减少窗口剖分



# 主要内容

- 图像空间消隐
  - 区间扫描线算法
  - 层次z-buffer算法
- 景物空间消隐
  - 区域细分算法(Warnock)
  - Weiler-Atherton算法

# Weiler–Atherton算法

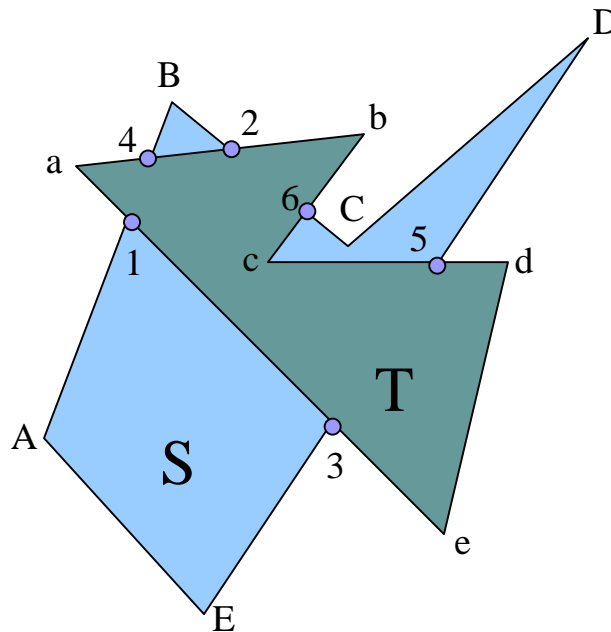
- Weiler-Atherton算法是物体空间的消隐算法
- 类似于 Warnock算法
  - 空间剖分：基于多边形而非坐标轴进行剖分
  - 分而治之
- 输出：具有任意精度的多边形
  - 对比Warnock算法：多边形的逼近

# Weiler-Atherton 裁剪算法

- 更为一般的多边形裁剪算法
- 裁剪窗口和裁剪多边形均可以是非凸（凹）多边形

# Weiler-Atherton 裁剪算法

- 首先计算两个多边形边之间的所有交点

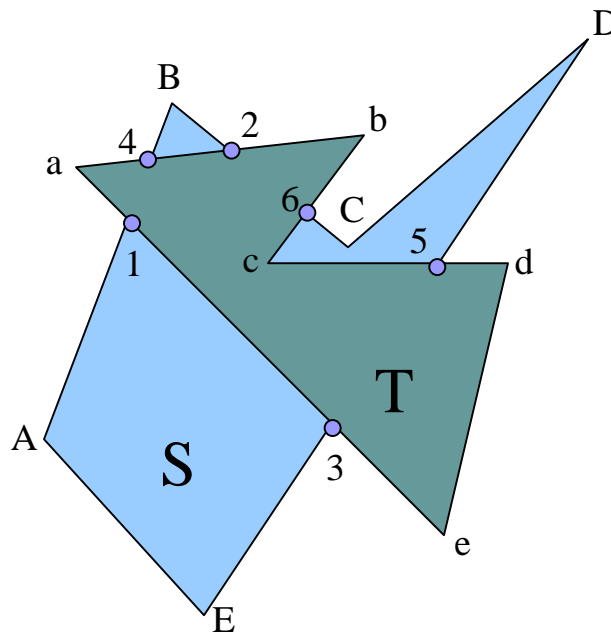


S: A,B,C,D,E

T: a,b,c,d,e

# Weiler-Atherton 裁剪算法

- 然后将交点逆时针插入多边形中，重构多边形（原始多边形顶点是逆时针排列的）

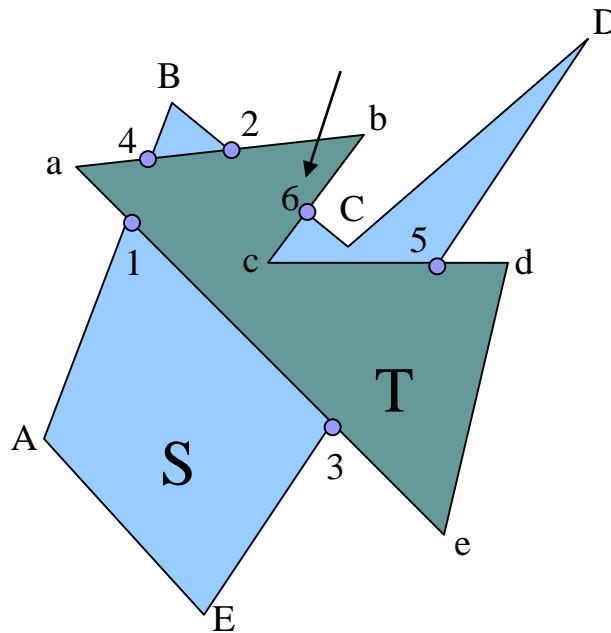


S: A,1,4,B,2,6,C,D,5,3,E

T: a,4,2,b,6,c,5,d,e,3,1

# Weiler-Atherton 裁剪算法

- 找到裁剪多边形的一个交点，该交点的所在边的两个顶点是从裁剪窗口外到裁剪窗口内的
- 遍历裁剪多边形，至下一个交点



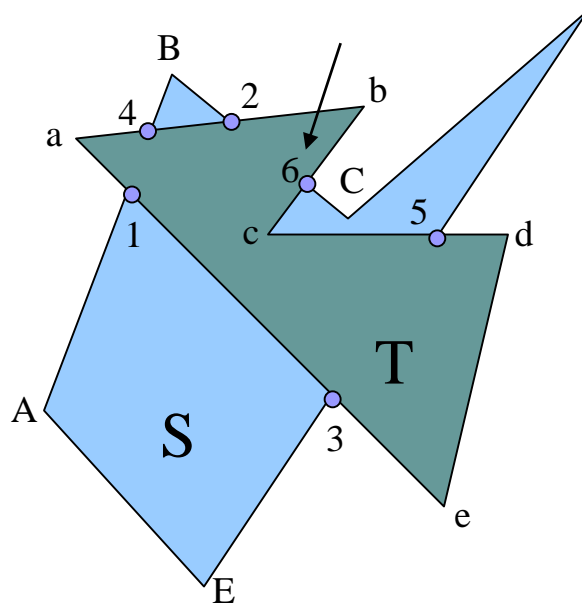
**S:** A,1,4,B,2,6,C,D,5,3,E

**T:** a,4,2,b,6,c,5,d,e,3,1

Clip: 6,c,5,...

# Weiler-Atherton 裁剪算法

- 当检测到一个新的交点时，切换遍历多边形S(T)至遍历多边形T(S)
- 直至遇到初始交点



S: A,1,4,B,2,6,C,D,5,3,E

T: a,4,2,b,6,c,5,d,e,3,1

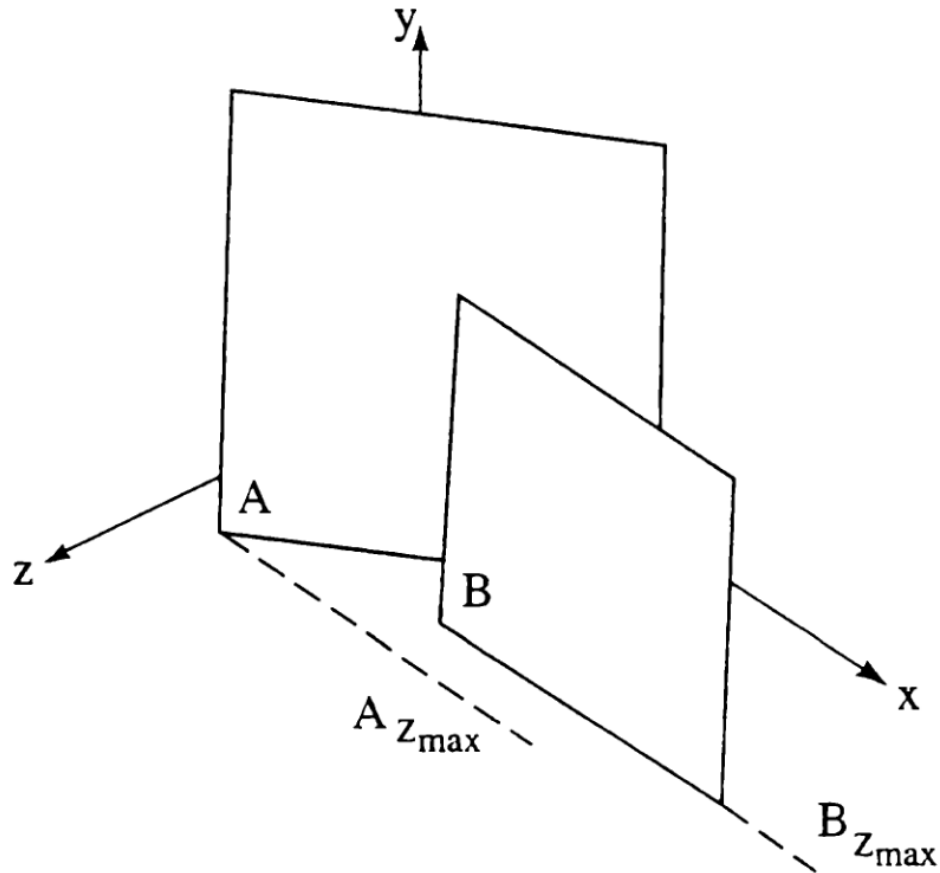
Clip: 6,c,5,3,1,4,2,6

# Weiler–Atherton算法

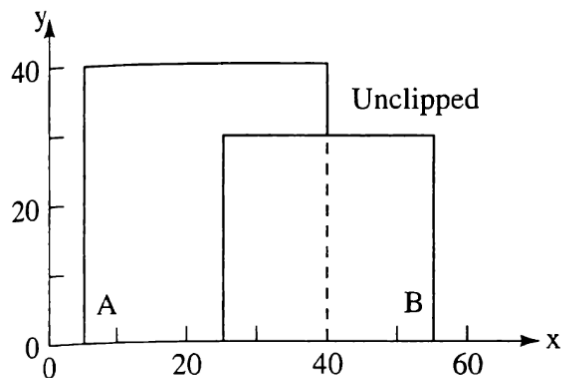
1. 基于多边形的最大 $z$ -值递减排序；
2. 选取排序队列中第一个多边形 $P$
3. 以 $P$ 为窗口，裁剪其它多边形，产生两个队列
  - 内部队列：位于 $P$ 内的裁剪多边形(包含 $P$ )
  - 外部队列：位于 $P$ 外的裁剪多边形
4. 删除内部队列中所有位于多边形 $P$ 背后的裁剪多边形。如果内部队列中存在多边形位于 $P$ 的前面，将此多边形设为 $P$ ，并执行步骤3
5. 显示 $P$ ，处理外部队列，并返回步骤2



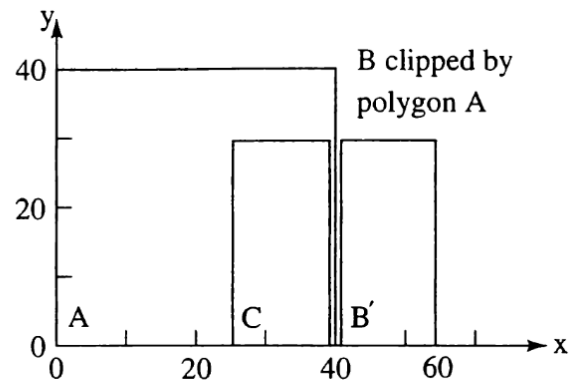
# Weiler–Atherton算法



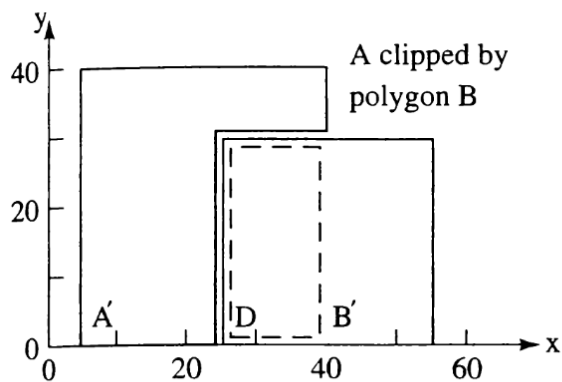
# Weiler–Atherton算法



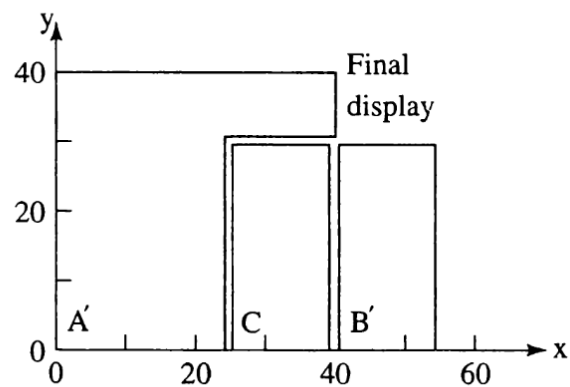
(a)



(b)



(c)



(d)

# 小结

- 图像空间消隐
  - 区间扫描线算法
  - 层次z-buffer算法
- 景物空间消隐
  - 区域细分算法(Warnock)
  - Weiler-Atherton算法