第二章 光栅图形学

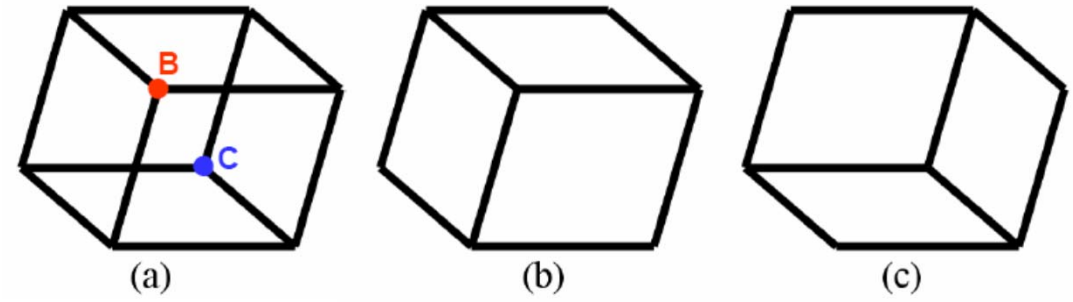
- 光栅图形学
  - 直线段的扫描转换算法
  - 圆弧的扫描转换算法
  - 多边形的扫描转换与区域填充
  - 字符
  - 裁剪
  - 反走样
  - 消隐

2.7 消隐

- 基本概念
- 消隐的分类
- 提高消隐算法效率的常见方法
- 消除隐藏线
- 消除隐藏面
  - 画家算法
  - Z缓冲区(Z-Buffer)算法
  - 扫描线Z-buffer算法
  - 区间扫描线算法
  - 区域子分割算法
  - 光线投射算法

基本概念

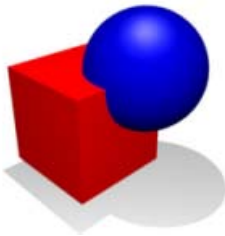
- 消隐是图形学中非常重要的一个基本问题。
- **投影变换**失去了深度信息，往往导致图形的二义性
  - 消隐：为了消除二义性，必须在绘制时消除被遮挡的不可见的线或面，习惯上称作消除隐藏线和隐藏面；
  - 经过消隐得到的投影图称为物体的真实图形。

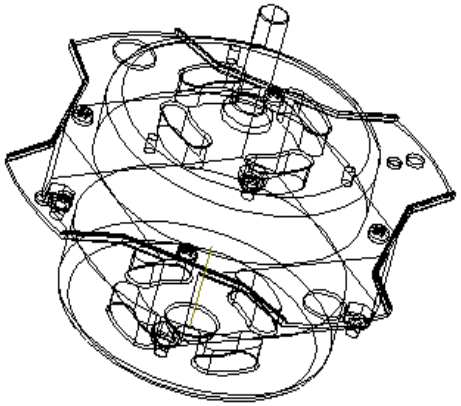


没有消隐的图形具有二义性：(a) 立方体的线框图；(b) 顶点B离视点最近时的消隐；(c) 顶点C离视点最近时的消隐

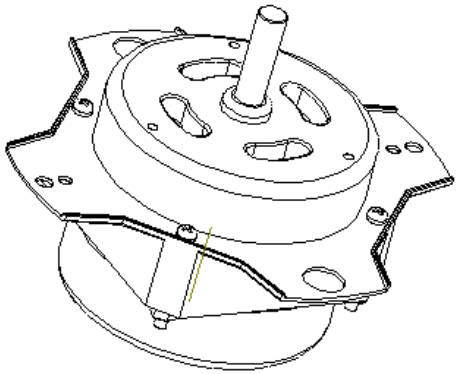
基本概念

- 消隐的对象是三维物体。三维体的表示主要有边界表示和构造实体几何表示等。
  - **最简单的方法**：用表面上的平面多边形表示。
- 消隐结果与观察物体有关，也与视点有关。

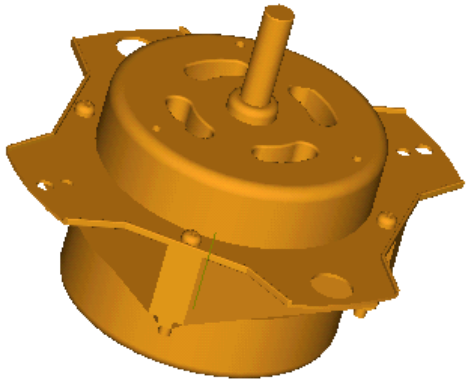




线框图



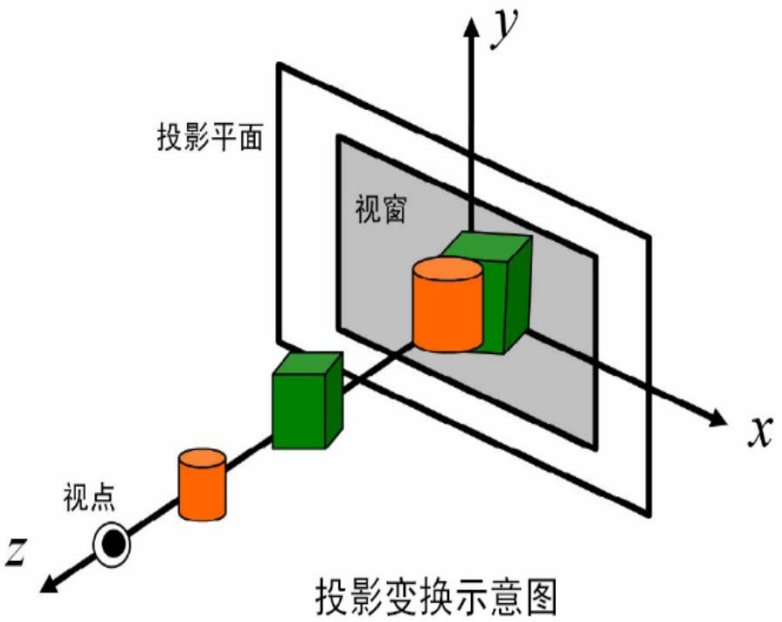
消隐图



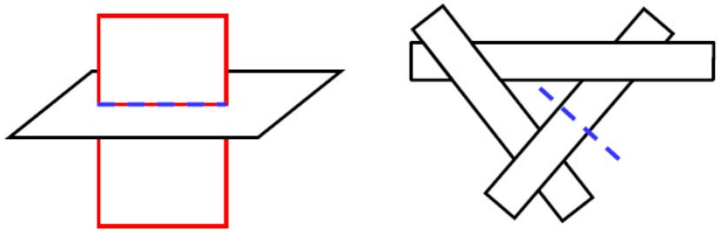
真实感图形

基本概念

- 几个假设：
  - 投影平面是oxy平面；
  - 投影方向为负z轴方向的平行投影；
    - z值越大，离视点越近；
    - 透视变换转化为平行投影。
  - 不能处理相互贯穿或循环遮挡的物体，此时应做特殊处理。



投影变换示意图


相互贯穿　　循环遮挡

基本概念

- 消隐问题的复杂性导致许多精巧的算法，不同算法适合于不同的应用环境
  - 在实时模拟过程中，要求消隐算法速度快，通常生成的图形质量一般；
  - 在真实感图形生成过程中，要生成高质量的图形，通常消隐算法速度较慢。
- 消隐算法的权衡：**消隐效率、图形质量**。

基本概念

- 与消隐与密切相关的因素：
  - **物体排序**：判断场景中的物体全部或者部分与视点之间的远近；
  - **连贯性**：场景中物体或其投影所表现出来的相似程度。
- 消隐算法的效率很大程度上取决于排序的**效率**、各种**连贯性**的利用。

提高消隐算法效率的常见方法

- 利用连贯性
- 包围盒技术
- 背面剔除
- 空间分割技术
- 物体分层表示

提高消隐算法效率的常见方法

- **利用连贯性**：相邻事物的属性之间有一定的连贯性，其属性值通常是平缓过渡的，如颜色值、空间位置关系等
  - 物体连贯性
  - 面的连贯性
  - 区域连贯性
  - 扫描线的连贯性
  - 深度连贯性

提高消隐算法效率的常见方法

- 物体连贯性
  - 如果物体A与物体B是完全相互分离的，则在消隐时，只需比较A、B两物体之间的遮挡关系就可以了，无须对它们的表面多边形逐一进行测试。
- 面的连贯性
  - 一张面内的各种属性值一般都是缓慢变化的，允许采用增量形式对其进行计算。
- 扫描线的连贯性
  - 相邻两条扫描线上，可见面的分布情况相似。

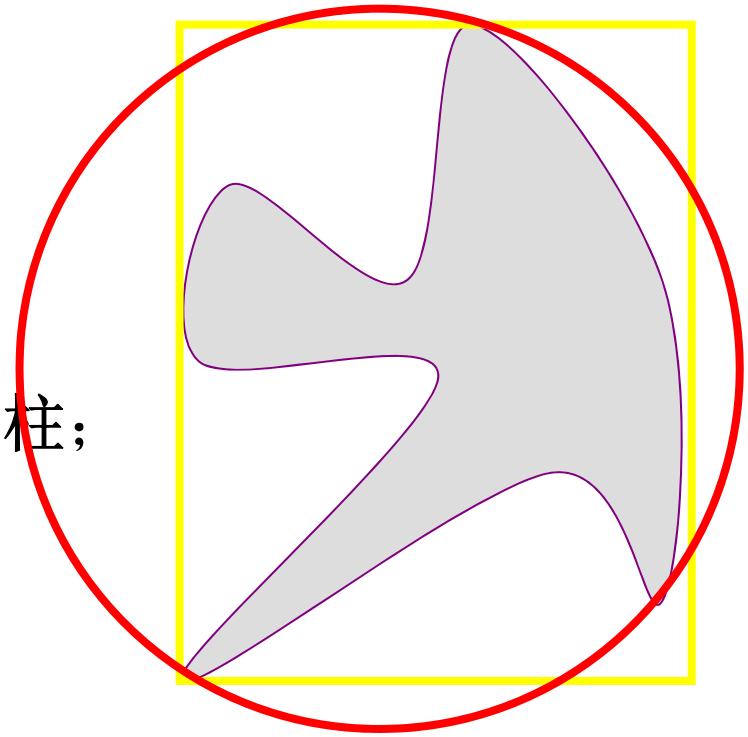
提高消隐算法效率的常见方法

- **区域连贯性**
  - 区域指屏幕上一组相邻的像素，它们通常为同一个可见面所占据，可见性相同。
  - 区域连贯性表现在一条扫描线上即为扫描线上的每个区间内只有一个面可见。
- **深度连贯性**
  - 占据屏幕上同一区域的不同表面的深度不同；
  - 但同一表面上的相邻部分深度是相近的，在判断表面间的遮挡关系时，只需取其上一点计算出深度值，比较该深度值即可得到结果。

提高消隐算法效率的常见方法

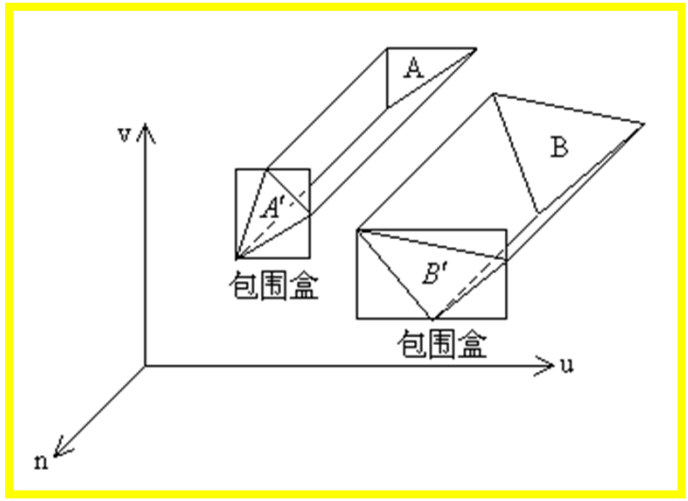
- 包围盒技术
  - 定义:一个形体的包围盒指的是包围它的简单形体。
  - 好的包围盒要具有两个条件：
    - 假设包围和充分

      紧密包围着形体；
    - 对其的测试比较简单。
  - 常用包围盒:长方体、球、圆柱；
  - 避免盲目求交。



提高消隐算法效率的常见方法

- 包围盒技术
  - 例如：两个空间多边形A、B在投影平面上的投影分别为A′，B′ ，因为A′ 、B′的矩形包围盒不相交，则A′、B′不相交，无须进行遮挡测试。

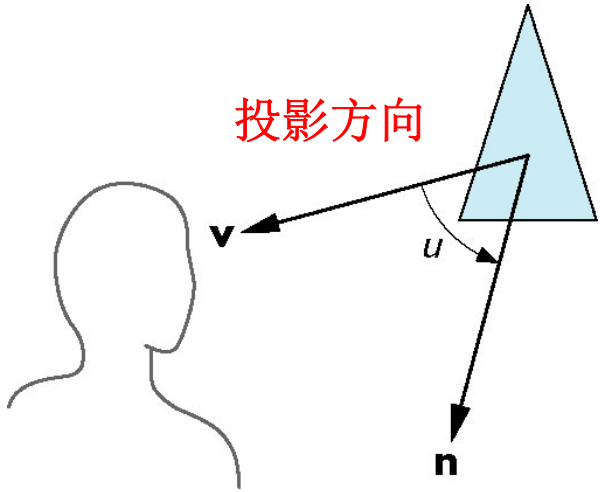


提高消隐算法效率的常见方法

- 背面剔除
  - **外法向**：规定每个多边形的外法向都是指向物体外部的。
  - **前向面**：若多边形的外法向与投影方向（观察方向）的夹角为钝角（$V \cdot N<0$），称为前向面。
  - **后向面**：若多边形的外法向与投影方向（观察方向）的夹角为锐角（$V \cdot N>0$），称为后向面（背面）。

提高消隐算法效率的常见方法

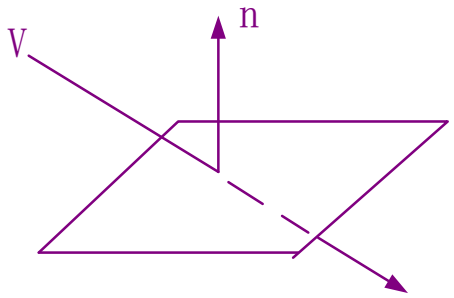
- 背面剔除

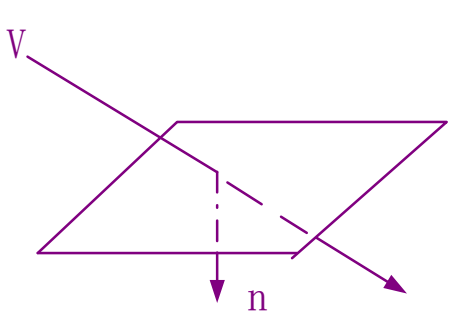


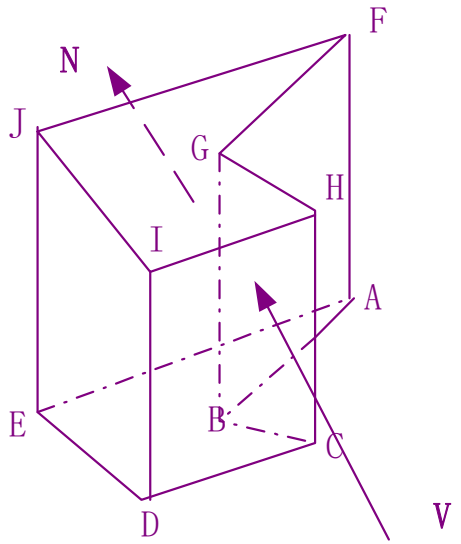
夹角为180－*u*

  - 后向面总是看不见的，不会由于后向面的遮挡，而使别的棱成为不可见的。因此计算时，可以把这些后向面全部去掉，这并不影响消隐结果。

提高消隐算法效率的常见方法







前向面　　后向面　　多面体的隐藏线消除

图3中的JEAF、HCBG和DEABC所在的面均为后向面。其它为前向面。

提高消隐算法效率的常见方法

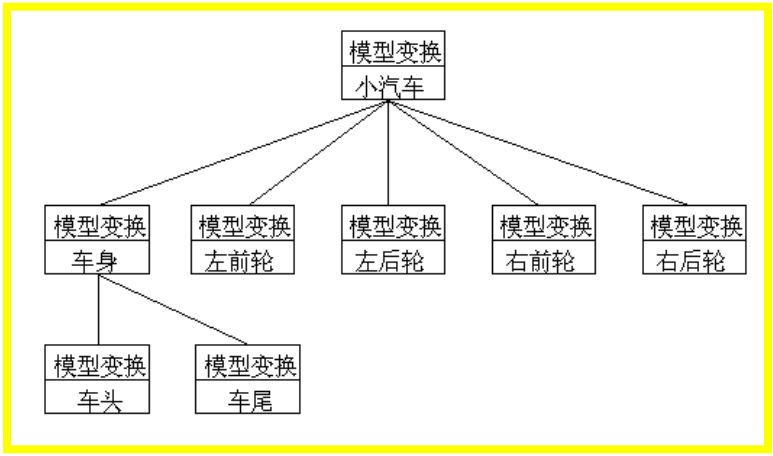
- **空间分割技术**:场景中的物体，它们的投影在投影平面上是否有重叠部分？
  - **依据**：对于根本不存在相互遮挡关系的物体，应避免这种不必要的测试。
  - **原因**：物体在场景中分散，有些物体的投影相距甚远，不存在遮挡关系。
  - **方法**：将投影平面上的窗口分成若干小区域；为每个小区域建立相关物体表，表中物体的投影于该区域有相交部分；则在小区域中判断那个物体可见时，只要对该区域的相关物体表中的物体进行比较即可。

提高消隐算法效率的常见方法

- 空间分割技术
  - **复杂度比较**：不妨假定每个小区域的相关物体表中平均有h个物体，场景中有k个物体，由于物体在场景中的分布是分散的，显然h远小于k。根据第二种消隐方法所述，其算法复杂度为O(h*h)，远小于O(k*k)。

提高消隐算法效率的常见方法

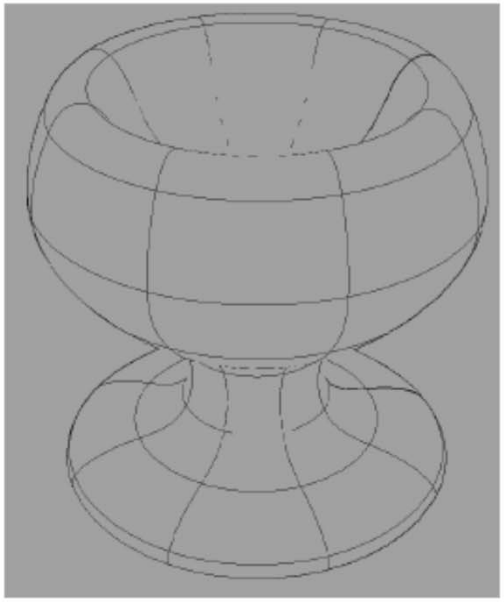
- 物体分层表示
  - 表示形式：模型变换中的树形表示方式；



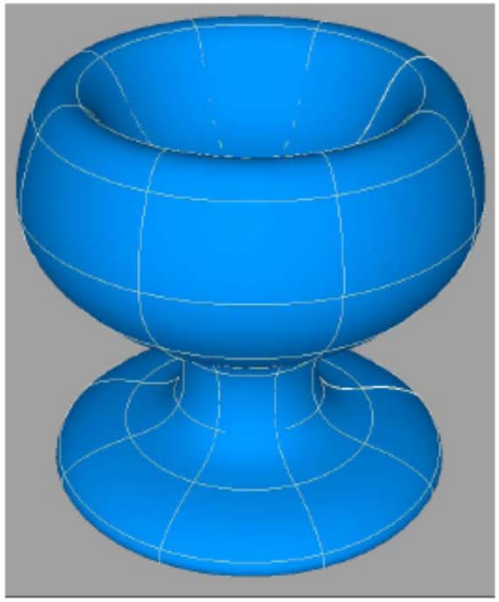
  - **依据**：减少场景中物体的个数，降低算法复杂度；
  - 将父节点所代表的物体看成子节点所代表物体的包围盒，当两个父节点之间不存在遮挡关系时，就没有必要对两者的子节点做进一步测试；
  - 父节点之间的遮挡关系可用包围盒进行预测试。

2.7.1消隐的分类

- 按消隐对象和输出结果分类
  - **线消隐**：消除的是物体上不可见的边。
  - **面消隐**：消除的是物体上不可见的面。



线消隐：输出线框图



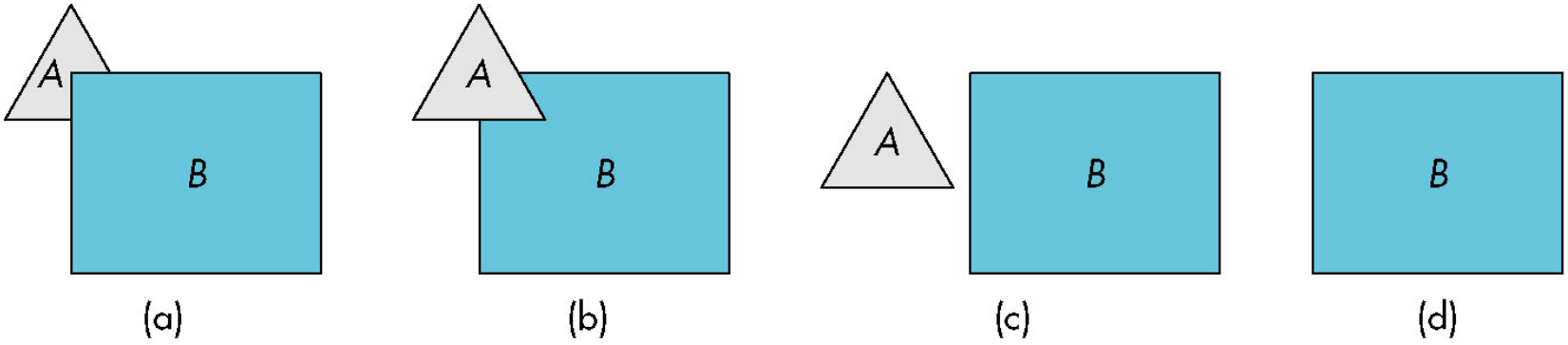
面消隐：输出着色图

2.7.1消隐的分类

- 根据消隐空间分类：
  - **物体空间的消隐算法**：以场景中的物体为处理单元；
    - for（场景中的每一个物体）

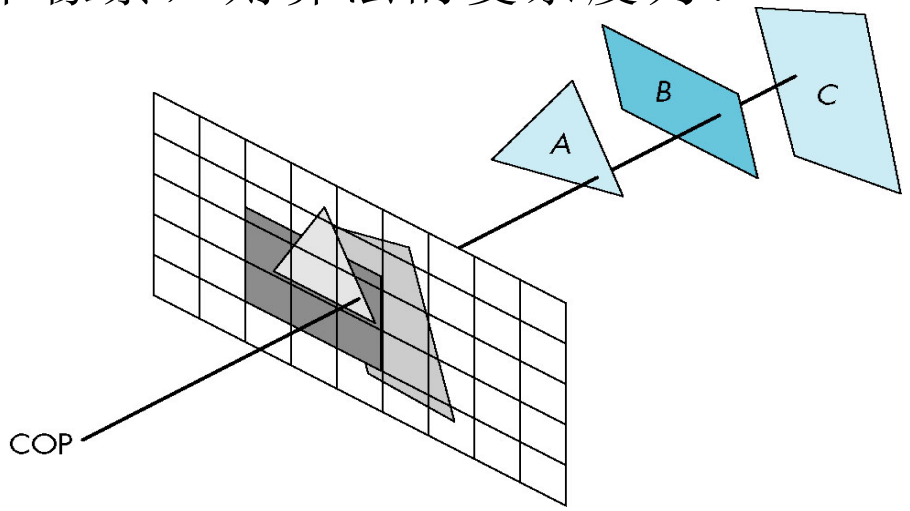
      {将其与场景中的其它物体比较，确定其表面的可见部分；

      显示该物体表面的可见部分；}
    - 假设场景中有k个物体，平均每个物体表面由h个多边形构成，显示区域中有m*n个像素，则算法的复杂度为：O((kh)*(kh))



2.7.1消隐的分类

- **图像空间的消隐算法**：以窗口内的每个像素为处理单元；
  - for(窗口内的每一个像素)
    {确定距视点最近的物体，以该物体表面的颜色来显示像素}
  - 假设场景中有k个物体，平均每个物体表面由h个多边形构成，显示区域中有m*n个像素，则算法的复杂度为：O(mnkh)

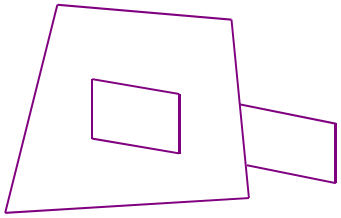


2.7.1消隐的分类

- 理论上：
  - 物体空间消隐算法的效率高于图像空间消隐算法。
  - 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。
- **物体空间和图像空间的消隐算法：**在物体空间中预先计算面的可见性优先级，再在图像空间中生成消隐图；
  - 代表性算法：画家算法。

2.7.2消除隐藏线

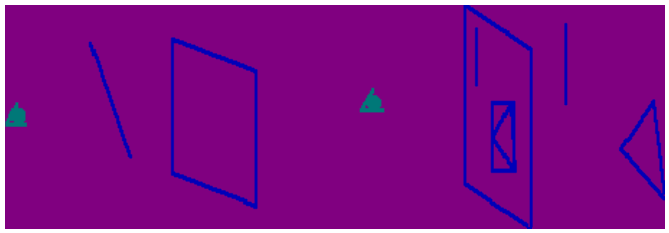
- 对造型的要求
  - 在线框表示模型中，要求造型系统中有面的信息，最好有体的信息。
- 坐标变换
  - 通过坐标变换，将视点变换到Z轴的正无穷大处，视线方向变为Z轴的负方向。



- 最基本的运算
  - 线消隐中，判断面对线的遮挡关系：体分解成面，再判断面与线关系。判断过程中需反复地进行线线、线面之间的求交运算。

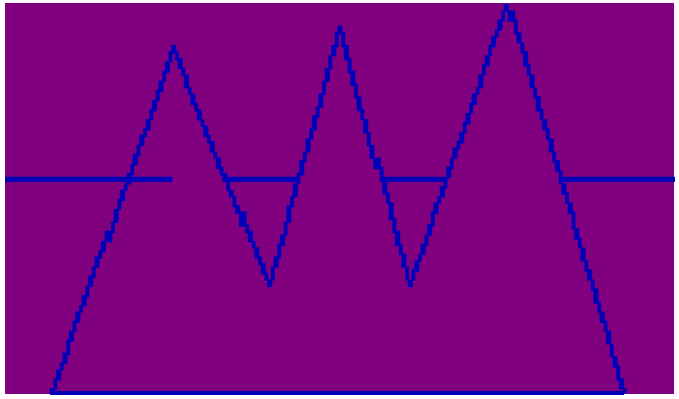
2.7.2消除隐藏线

- 平面对直线段的遮挡判断算法
  - (1)若线段的两端点及视点在给定平面的同侧，线段不被给定平面遮挡，转(7)；
  - (2)若线段的投影与平面投影的包围盒无交，线段不被给定平面遮挡，转(7)；
  - (3)求直线与相应无穷平面的交：
    - 无交点转(4)；
    - 若交点在线段内部，交点将线段分成两段，与视点同侧一段不被遮挡，另一段在视点异侧转(4)；
    - 若交点在线段外部,转(4)。



2.7.2消除隐藏线

- (4)求所剩线段的投影与平面边界投影的所有交点。若无交点，转(7)。
- (5)以上所求得的各交点将线段的投影分成若干段，求出第一段中点。
- (6)若第一段中点在平面的投影内，则相应的段被遮挡，否则不被遮挡；其他段的遮挡关系可依次交替取值进行判断。
- (7)结束。



2.7.2消除隐藏线

- 基本数据结构
  - 面表(存放参与消隐的面) +线表(存放待显示的线) 。
- 算法
  - 假设E为面F的一条边, 需判别F以外每一个面与E的遮挡关系。
- 假设消隐对象有n条边和m个面， 当n和m很大时， 两两求交的消隐方法工作量很大：
  - 解决方法：背面剔除。

2.7.2消除隐藏线

- 算法

```
HiddenLineRemove()
{ 坐标变换;
  for(对每个面Fj)
   for(Fj的每一条边Ei) 将二元组< Ei ,j>压入堆栈
  While(栈不空)
  { < Ei ,j0> = 栈顶;
    for(j!= j0的每一个面Fj)
{ if( Ei 被Fj 全部遮挡)
{ 将Ei 清空;   break; }
if( Ei 被Fj 部分遮挡)
{ 从Ei 中将被遮挡的部分裁掉;
   if(Ei 被分成若干段)
   { 取其中的一段作为当前段;
     将其它段及相应的j压栈
   }
}
}
if(Ei 段不为空)
显示Ei ;
    }
}
```

2.7.3消除隐藏面

- 画家算法
- Z缓冲算法
- 扫面线Z缓冲算法
- 区间扫面线算法
- 区域子分割(Warnack)算法
- 光线投射算法

2.7.3.1画家算法

- **列表优先算法：**画家的作画顺序暗示出所画物体之间的相互遮挡关系。
- **基本思想：**
  - 先把屏幕置成背景色；
  - 再把物体的各个面按其离视点的远近进行排序，排序结果存在一张深度优先级表中；
  - 然后按照从远到近的顺序逐个绘制各个面。
- 关键是如何对场景中的物体按深度排序。

2.7.3.1画家算法

- 多边形的排序算法如下：

在规范投影坐标系里 ***XYZ*** 中，投影方向是 ***Z***轴的负方向，因而 ***Z***坐标大者距观察者更近。$Z_{min}(P)$和 $Z_{max}(P)$ 分别为 ***P*** 各顶点 ***Z***坐标的最小和最大值。[投影为正平行投影]

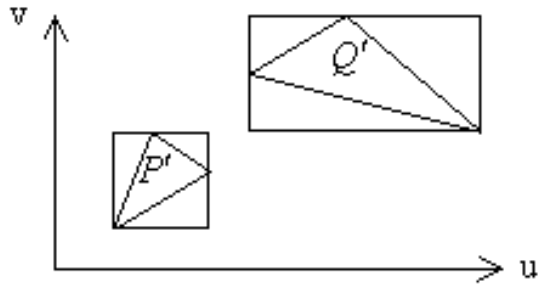
- Step 1:将场景中所有多边形存入一个线性表，记为L；
- Step 2:如果L中仅有一个多边形，算法结束；否则根据每个多边形的Z_{min}对它们预排序。不妨假定多边形P落在表首，即Z_{min}(P)为最小。再记Q为L–{P}(表中其余多边形）中任意一个；

2.7.3.1画家算法

- Step 3:判别P，Q之间的关系，有如下二种：
  - 对所有的Q，有$Z_{max}(P) < Z_{min}(Q)$，则多边形P的确距观察点最远，它不可能遮挡别的多边形。令L=L–{P}，返回第二步；
  - 存在某一个多边形Q，使$Z_{max}(P) > Z_{min}(Q)$，需进一步判别：

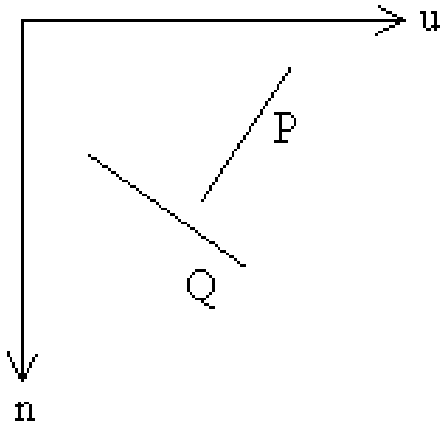
2.7.3.1画家算法

- A）若P,Q的投影P’,Q’的包围盒不相交（图a），则P,Q在表中的次序不重要，令L=L－{P}，返回step 2;否则进行下一步。



(a)

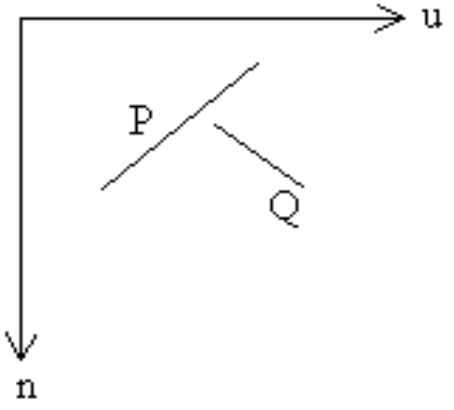
- B）若P的所有顶点位于Q所在平面的不可见的一侧(图b)，则P,Q关系正确，令L=L－{P}，返回step 2；否则进行下一步。



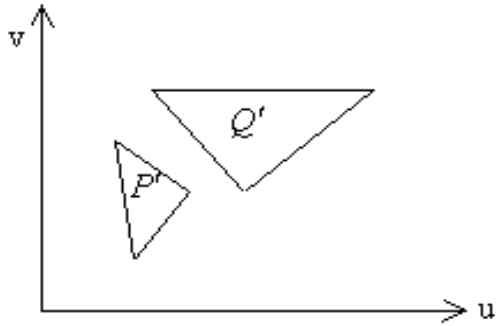
(b)

2.7.3.1画家算法

- C）若Q的所有顶点位于P所在平面的可见的一侧(图c)，则P,Q关系正确，令L=L－{P}，返回step 2；否则进行下一步。
- D）对P,Q投影P′,Q′求交，若P′,Q′不相交(图d)，则P,Q在表中的次序不重要，令L = L–{P}，返回step 2；否则在它们所相交的区域中任取一点，计算P,Q在该点的深度值，如果P的深度小，则P,Q关系正确，令L = L–{P}，返回step 2;否则交换P,Q,返回step 3.



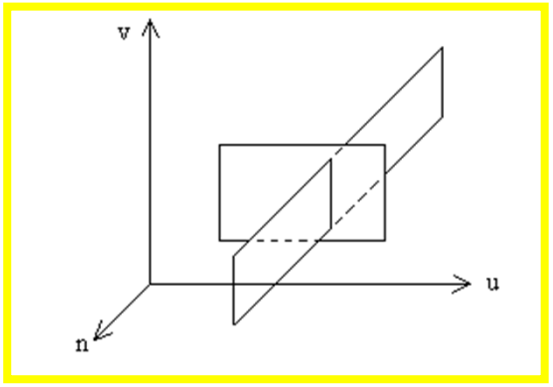
(c)

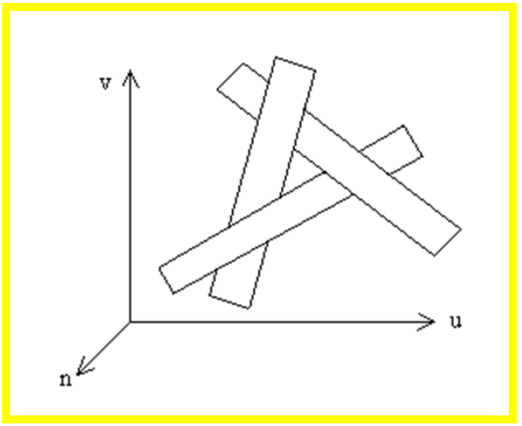


(d)

2.7.3.1画家算法

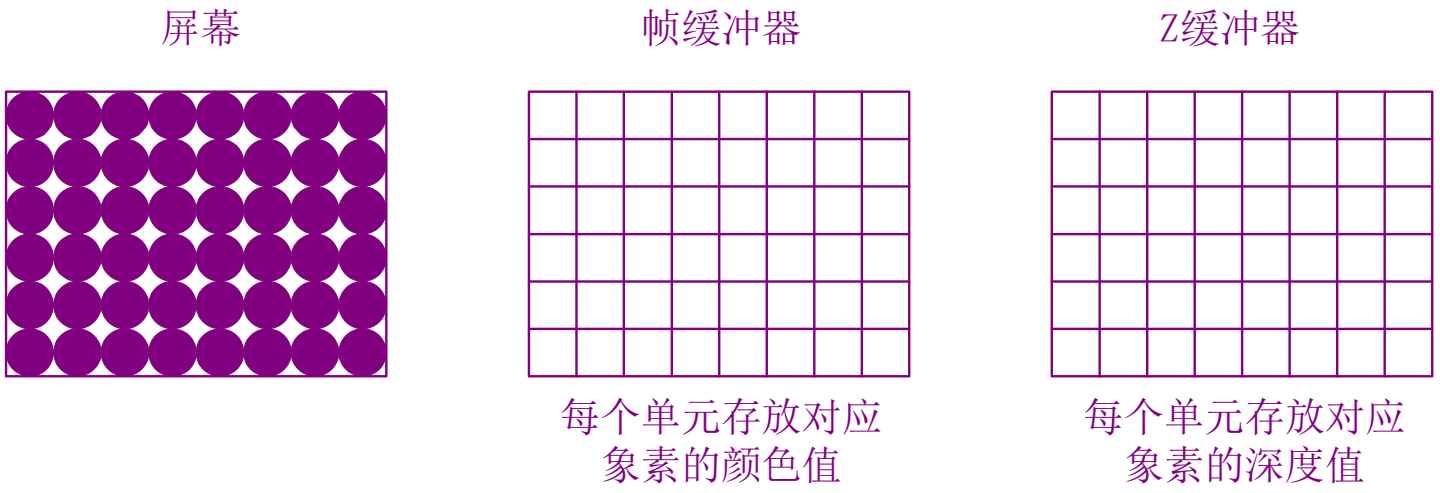
- **优点**：简单（如何对场景中的物体按深度排序）。
- **缺点**：只能处理互不相交的面，且深度优先级表中面的顺序可能出错。
- 本算法不能处理的情况：
  - 两个面相交；
  - 循环重叠。
- 解决办法：进行分割。





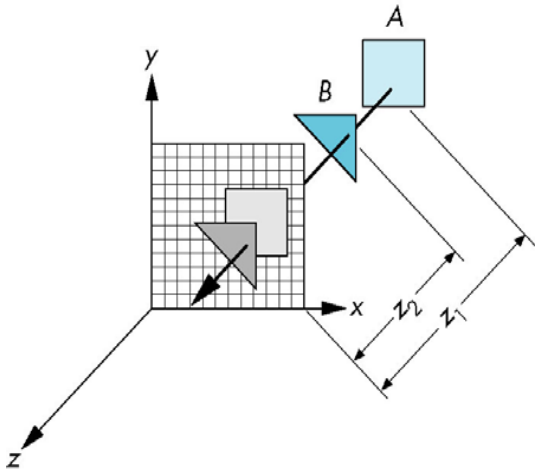
2.7.3.2 Z缓冲区算法

- Z－Buffer算法(深度缓冲depth-buffer)
- 组成：
  - 帧缓冲器--保存各像素颜色值；
  - Z 缓冲器 --保存各像素处物体深度值；
  - Z缓冲器中的单元与帧缓冲器中的单元一一对应。



2.7.3.2 Z缓冲区算法

- 算法思想
  - 将 Z 缓冲器中个单元的初始值置为最小值。
  - 多边形投影时，当要改变某个像素的颜色值时，首先检查当前多边形的深度值是否大于该像素原来的深度值：
    - 大于，说明当前多边形更靠近观察点，用它的颜色替换像素原来的颜色；同时更新深度值；
    - 否则说明在当前像素处，当前多边形被前面所绘制的多边形遮挡了，是不可见的，像素的颜色值不改变。



2.7.3.2 Z缓冲区算法

Z-Buffer算法（）

```
{   帧缓存全置为背景色
    深度缓存全置为最小Z值
    for(每一个多边形)
    {  扫描转换该多边形
       for(该多边形所覆盖的每个象素(x,y) )
       {    计算该多边形在该象素的深度值Z(x,y);
            if(Z(x,y)大于Z缓存在(x,y)的值)
            {   把Z(x,y)存入Z缓存中(x,y)处
                把多边形在(x,y)处的颜色值存入帧缓存的(x,y)处
            }
       }
    }
}
```

需要计算的像素深度值次数

=多边形个数*多边形平均占据的像素个数

2.7.3.2 Z缓冲区算法

- 所有图像空间算法中最简单的隐藏面消除算法
- 优点：
  - 简单稳定，利于硬件实现；
  - 不需要整个场景的几何数据。
- 缺点：
  - 空间：需要一个额外的Z缓冲器；
  - 时间：在每个多边形占据的每个像素处都要计算深度值。
- 只用一个深度缓存变量zb的改进算法
  - 需要开一个与图象大小相等的缓存数组ZB

2.7.3.2 Z缓冲区算法

```
{       帧缓存全置为背景色
for(屏幕上的每个象素(i,j))
{       深度缓存变量zb置最小值MinValue
        for(多面体上的每个多边形P_k)
        {       if(象素点(i,j)在P_k的投影多边形之内)
                {       计算P_k在(i,j)处的深度值depth;
                        if(depth大于zb)
                        {       zb = depth;
                                indexp = k;
                        }
                }
        }
        if(zb != MinValue)
          在交点 (i,j) 处用多边形P_indexp的颜色显示
}
}
```

该算法主要的计算量在何处？

2.7.3.2 Z缓冲区算法

- 关键问题：
  - 计算多边形 P_k 在点（i，j）处的深度。设多边形的平面方程为：

$$ax + by + cz + d = 0$$

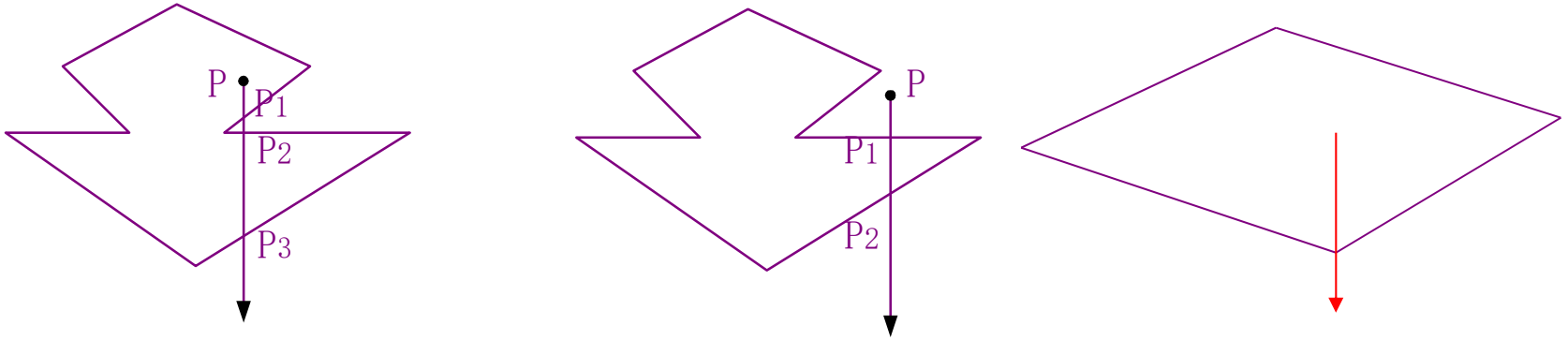
$$depth = -\frac{ai + bj + d}{c}$$

(c为0，多边形在xoy面上的投影为一条直线，不考虑)

  - 判断象素点(i, j)是否在 P_k 的投影多边形之内（点与多边形的包含性检测）

2.7.3.2 Z缓冲区算法

- 射线法



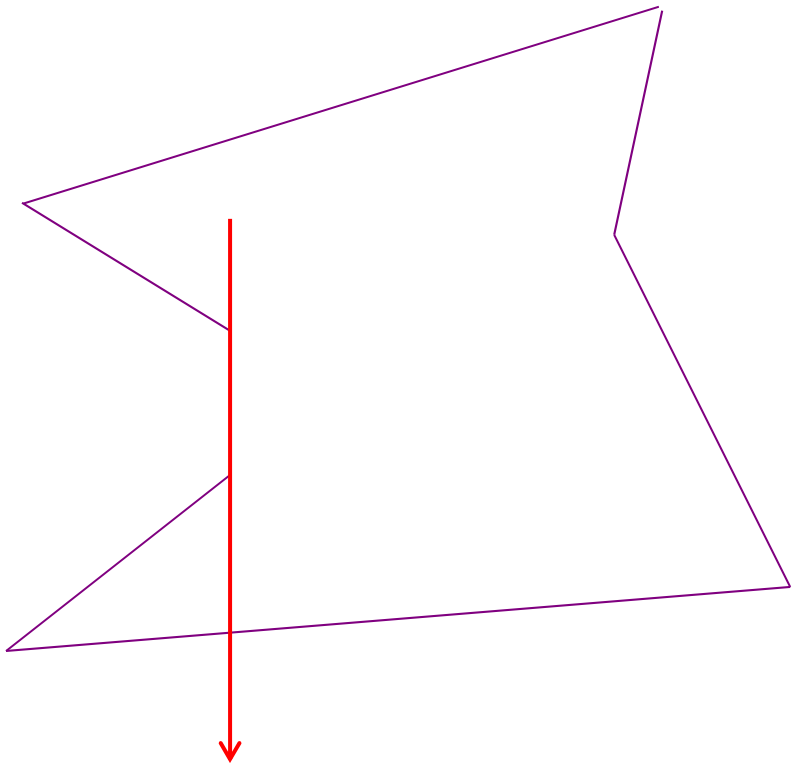
  - 由被测点P处向 y = - ∞方向作射线
    - 交点个数是奇数，则被测点在多边形内部；
    - 否则，偶数，在多边形外部；
    - 若射线正好经过多边形的顶点，则采用“左开右闭”的原则来实现。

2.7.3.2 Z缓冲区算法

- 射线与边相交判别法
  - 由$P(x,y)$向$y=-\infty$方向作射线，对P_iP_{i+1}按以下顺序检测：
    - 若$(x>x_i)$且$(x>x_{i+1})$, 点在边的右侧，射线与边无交；
    - 若$(x<=x_i)$且$(x<=x_{i+1})$, 点在边的左侧，射线与边无交；
    - 若$(y<y_i)$且$(y<y_{i+1})$, 点在边的下方，射线与边无交；
    - 若$(y>y_i)$且$(y>y_{i+1})$, 点在边的上方，射线与边有交；
    - 若$(y_i=y_{i+1})$，这时如果有$(y<y_i)$，则点在边的下方，射线与边无交；否则点在边的上方，射线与边有交；
    - 如上述检测失败，说明P点在P_iP_{i+1}的矩形包围盒内，构造函数$f(x,y)=(y-y_i)(x_{i+1}-x_i)-(x-x_i)(y_{i+1}-y_i)$。当$((x_i>x_{i+1})and(f(x,y))>0)$ or $((x_i<x_{i+1})$and$(f(x,y)<0)$时，射线与边无交，否则相交点。

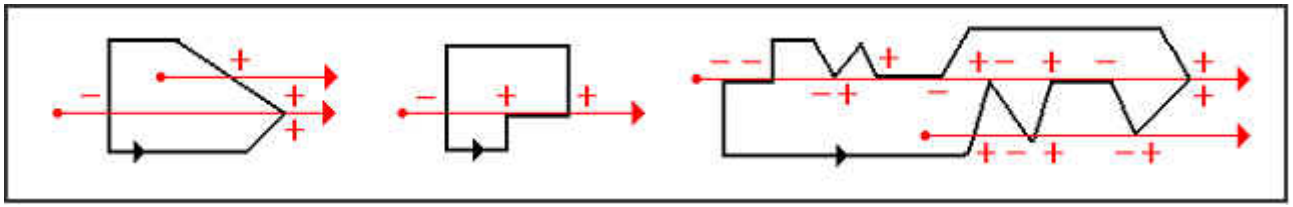
2.7.3.2 Z缓冲区算法

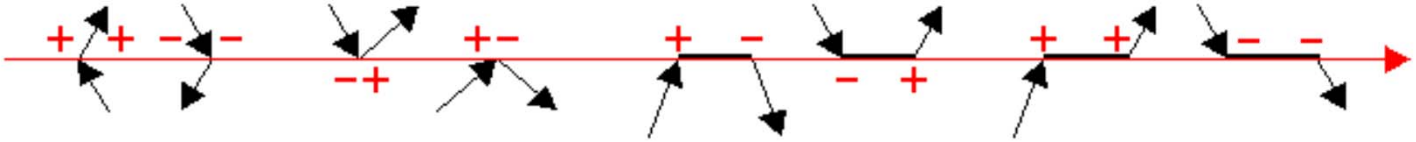
- **思考：**考虑点$P(x,y)$落在边P_iP_{i+1}边上或$P(x,y)$发出的射线与P_iP_{i+1}同一直线上射线。



2.7.3.2 Z缓冲区算法

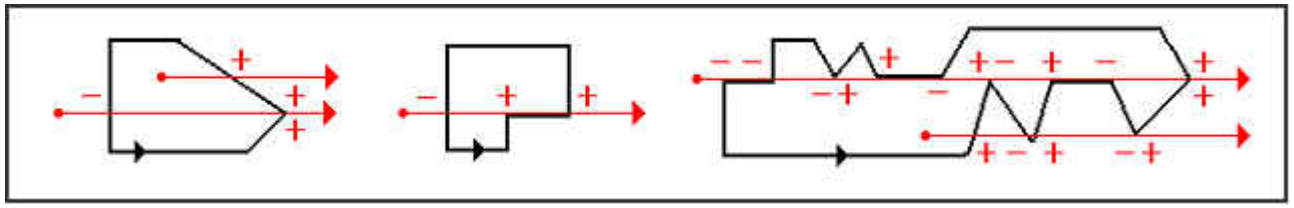
- 改进算法
  - 建立射线：
    - 由点P(x，y)向点(x = ∞)建立一射线向量。其中(x = ∞)是一个多角形顶点不可能达到的X大值。
  - 求交点：
    - 将此射线向量和多角形的各边向量求交。并记录交点几何参数和相对于射线的特征值，并将交点按其射线方向排队。





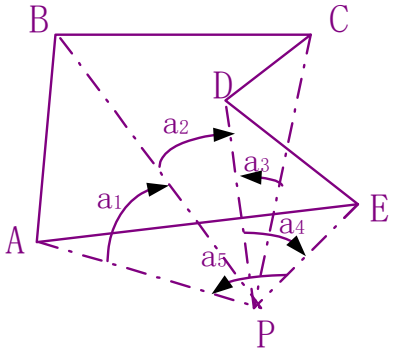
2.7.3.2 Z缓冲区算法

- 改进算法（续）
  - 合并重点：
    - 判别相邻交点的几何参数，如为重点求其特征值的代数和，如代数和为零，则取消两个交点，否则取消其中一个交点 。
  - 合并相邻同特征交点：
    - 判别相邻两个交点的特征，如果相邻两个特征相同，则取消其中一个交点。
  - 判别：
    - 计算交点个数，如为奇数，则点在多角形内部，否则在多角形外部 。

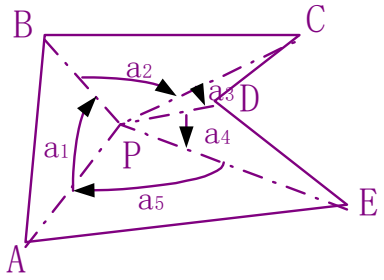


2.7.3.2 Z缓冲区算法

- 弧长法（单位圆）
  - 以被测点为圆心作单位圆，计算其在单位园上弧长的代数和
    - 代数和为0，点在多边形外部；
    - 代数和为2π，点在多边形内部；
    - 代数和为π，点在多边形边上。



（a）被测点p在多边形外



（b）被测点p在多边形内

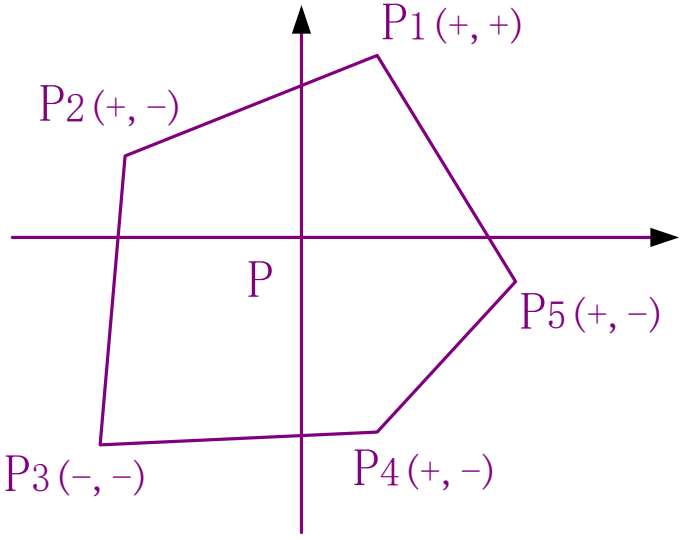
2.7.3.2 Z缓冲区算法

- 弧长累加方法：以顶点符号为基础。
  - 将坐标原点移到被测点P。各象限内点的符号对分别为(＋,＋)，（－，＋)，(－，－)，(＋，－)。
  - 算法规定：若顶点p_i的某个坐标为0,则其符号为＋。若顶点p_i的x、y坐标都为0，则说明这个顶点为被测点，我们在这之前予以排除。于是弧长变化如下表。

2.7.3.2 Z缓冲区算法

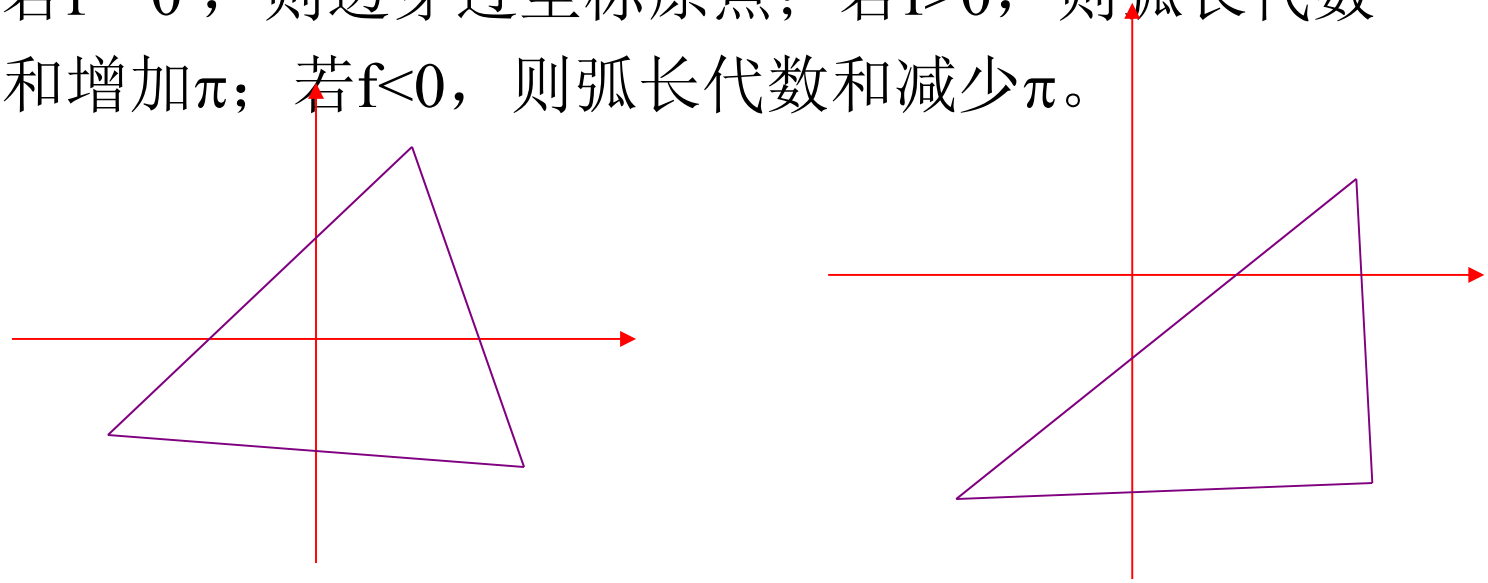
- 表 :符号对变化与弧长变化的关系

(sx_i , sy_i)	$(sx_{i+!} , sy_{I+1})$	弧长变化	象限变化
(+ +)	(+ +)	0	I → I
(+ +)	(- +)	π/2	I → II
(+ +)	(- -)	±π	I → III
(+ +)	(+ -)	-π/2	I → IV
…	…	…	…



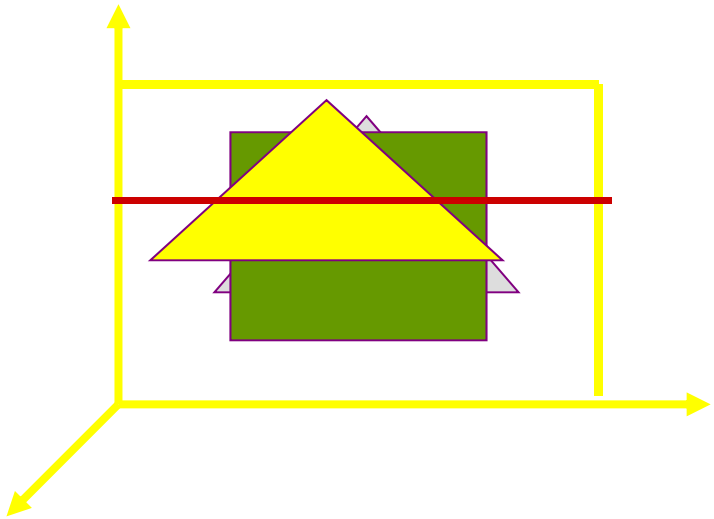
2.7.3.2 Z缓冲区算法

- 注意：当边的终点P_{i+1}在起点P_i的相对象限时，弧长变化可能增加或减少π：
  - 设(x_i,y_i)和(x_{i+1},y_{i+1})分别为边的起点和终点坐标。计算 $f = y_{i+1}x_i - x_{i+1}y_i$
  - 若f=0，则边穿过坐标原点；若f>0，则弧长代数和增加π；若f<0，则弧长代数和减少π。



2.7.3.3 扫描线Z-buffer算法

- 思想：
  - Z 缓冲器算法中所需要的Z 缓冲器容量较大，可以将整个绘图区域分割成若干个小区域，然后一个区域一个区域地显示，以减少Z 缓冲器的单元数；
  - 如果将小区域取成屏幕上的扫描线，就得到扫描线 Z 缓冲器算法。

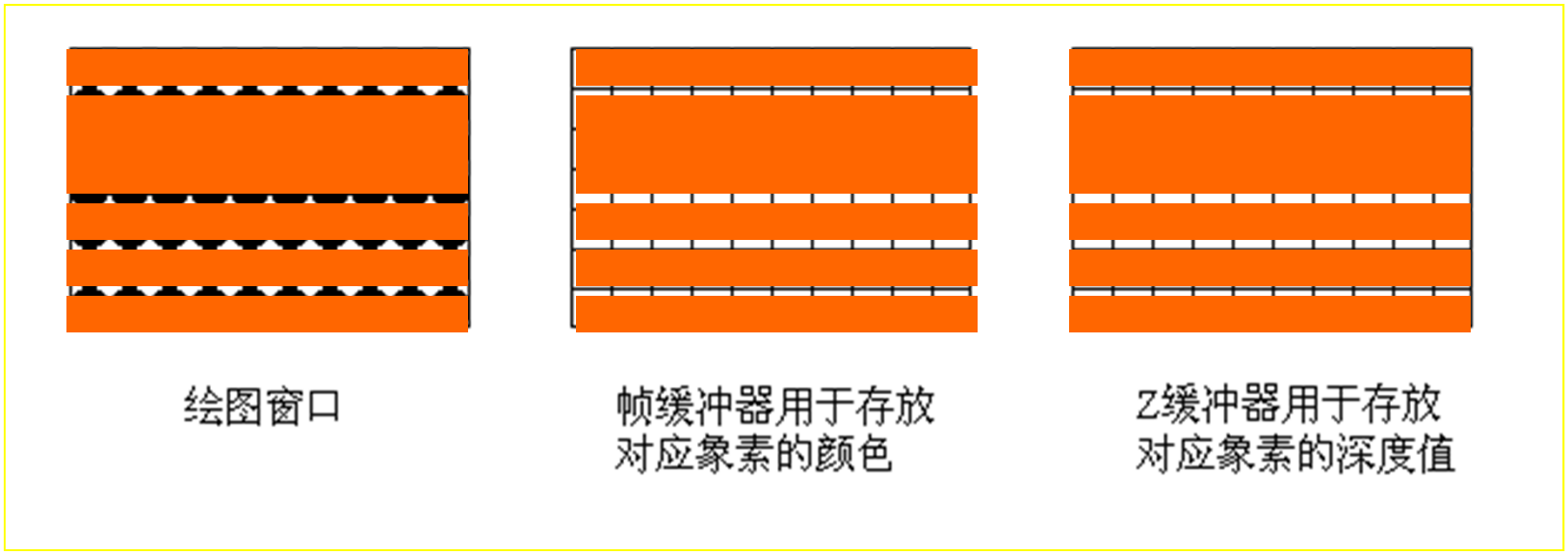


2.7.3.3 扫描线Z-buffer算法

- 算法思想:
  - 面Buffer到线Buffer;
  - 利用图形的连贯性(指深度计算);
  - 在处理当前扫描线时，开个一维数组作为当前扫描线的Z-buffer。找出与当前扫描线相关的多边形，以及每个多边形中相关的边对;
  - 对每一个边对之间的小区间上的各象素，计算深度，并与Z-buffer中的值比较，找出各象素处可见平面;
  - 确定颜色，写帧缓存：采用增量算法计算深度。

2.7.3.3 扫描线Z-buffer算法

- 四个方面的改进：
  - 将窗口分割成扫描线：Z缓冲器的单元数只要等于一条扫描线内像素的个数就可以了。



2.7.3.3 扫描线Z-buffer算法

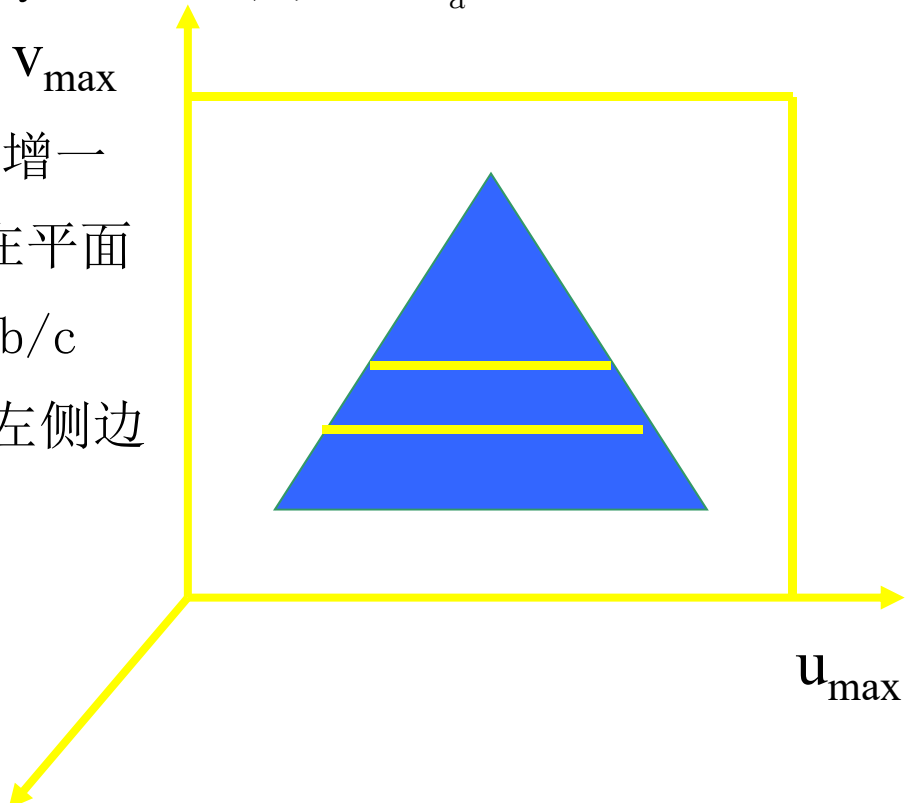
- 四个方面的改进（续）：
  - 采用多边形Y(分类)表、活化多边形表避免**多边形**与扫描线的盲目求交；
  - 利用边、边的分类表、边对、活化边对表避免**边**与扫描线的盲目求交。

2.7.3.3 扫描线Z-buffer算法

– 利用连贯性计算深度

- **水平方向**：当沿扫描线x递增一个像素时，多边形所在平面在z坐标的增量，对方程ax+by+cz+d=0来说，$\Delta z_a=-a/c$
- **竖直方向**
  - Δz_b ：当沿扫描线y递增一个像素时，多边形所在平面z坐标的增量， $\Delta z_b=-b/c$
  - 下一条扫描线与边对左侧边交点处的深度值：

$z_1=z_1+\Delta x_1\Delta z_a+\Delta z_b$

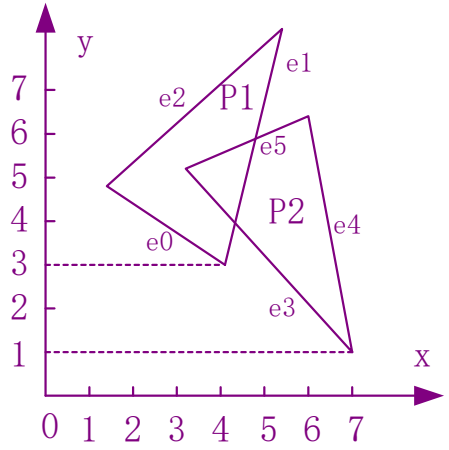


2.7.3.3 扫描线Z-buffer算法

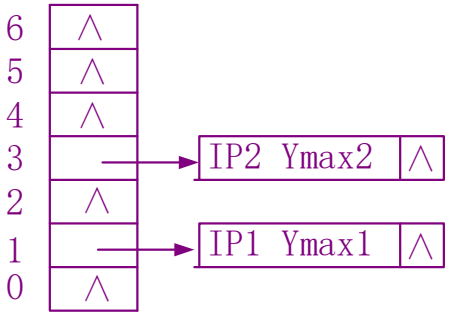
- 有效的数据结构：
  - 多边形Y表；
  - 活化多边形表(APT);
  - 边表(ET);
  - 活化边对表(AET)。

2.7.3.3 扫描线Z-buffer算法

- **多边形Y表**：存放所有多边形信息。
  - 根据多边形顶点中最小的y坐标，插入多边形Y表中的相应位置；
  - 根据序号可从定义多边形的数据结构中取多边形信息。(ax+by+cz+d=0的系数，多边形的边，顶点坐标和颜色)



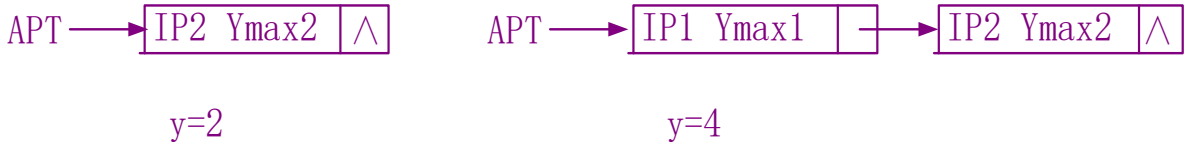
待消隐对象



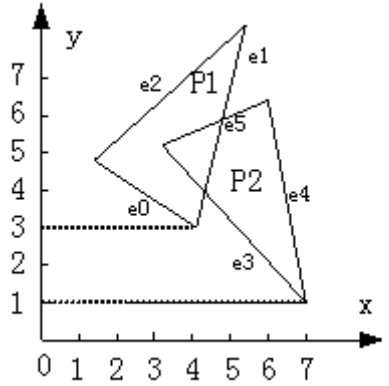
多边形y表

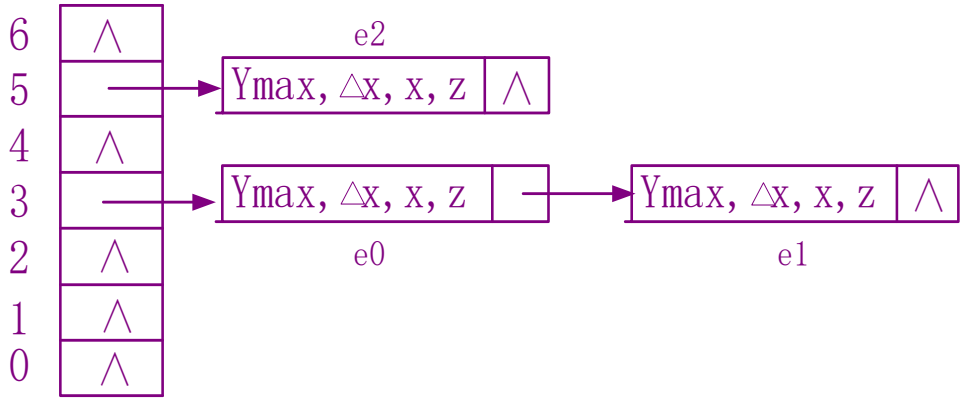
2.7.3.3 扫描线Z-buffer算法Q[

- **活化多边形表APT**：与当前扫描线相交的多边形。
  - APT是一个动态的链表。



- **边表ET**：活化多边形表中的每一个多边形都有一个边表ET(每条边端点中较大者，增量 Δx，y值较小一端的x坐标和z坐标)

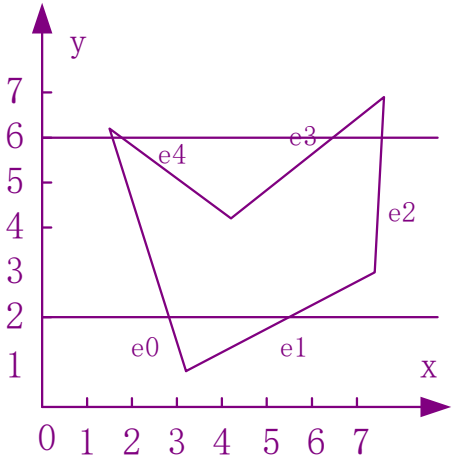




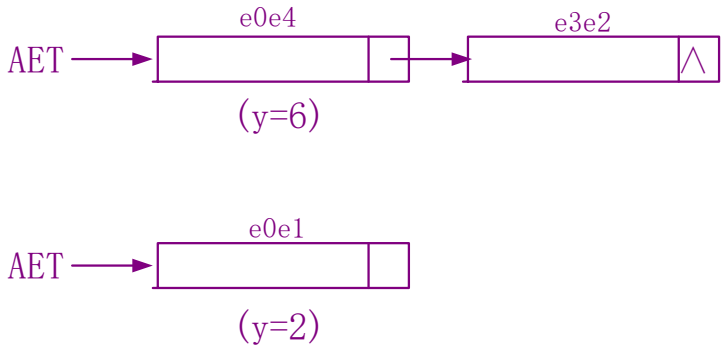
多边形P1的边表ET

2.7.3.3 扫描线Z-buffer算法

- 活化边对表AET
  - 在一条扫描线上，同一多边形的相邻两条边构成一个**边对**。活化边表AET中存放当前多边形中与当前扫描线相交的各边对的信息。



a 活化多边形



b 边对与活化边对表

2.7.3.3 扫描线Z-buffer算法

- AET的每个节点包括边对中如下信息:
  - x_l 左侧边与扫描线交点的x坐标
  - Δx_l 左侧边在扫描线加1时的x坐标增量
  - y_{lmax} 左侧边两端点中最大的y值
  - x_r 右侧边与扫描线交点的x坐标
  - Δx_r 右侧边在扫描线加1时的x坐标增量
  - y_{rmax} 右侧边两端点中最大的y值
  - z_l 左侧边与扫描线交点处的多边形深度值
  - IP 多边形序号
  - Δz_a 沿扫描线方向增加1个象素时，多边形所在平面的z坐标增量，为-a/c
  - Δz_b 扫描线加1时，多边形所在平面的z坐标增量，为-b/c

2.7.3.3 扫描线Z-buffer算法

扫描线Z-buffer算法()

{ 建多边形y表；根据多边形顶点最小y值，将多边形置入多边形y表。

活化多边形表APT，活化边对表AET初始化为空。

For(每条扫描线i，i从小到大)

{ 1. 帧缓存CB置为背景色。

2. 深度缓存ZB (一维数组) 置为负无穷大。

3. 将对应扫描线i的，多边形Y表中的多边形加入到活化多边形表APT中。

4. 对新加入的多边形，生成其相应的边表。

5. 对APT中每一个多边形，若其边表中对应扫描线i增加了新的边，将新的边配对，加到活化边对表AET中。

2.7.3.3 扫描线Z-buffer算法

6. 对AET中的每一对边：

6.1 对$x_l < j < x_r$ 的每一个象素，按增量公式$z = z + \Delta z_a$计算各点depth 。

6.2 与ZB中的量比较，depth >ZB(j), 则令ZB(j) =depth，并确定颜色值，写帧缓存。

7. 删除APT中多边形顶点最大y坐标为i的多边形，并删除相应的边。

8. 对AET中的每一个边对，作如下处理：

8.1 删除y_{lmax}或y_{rmax} 已等于i的边。若一边对中只删除了其中一边，需对该多边形的边重新配对。

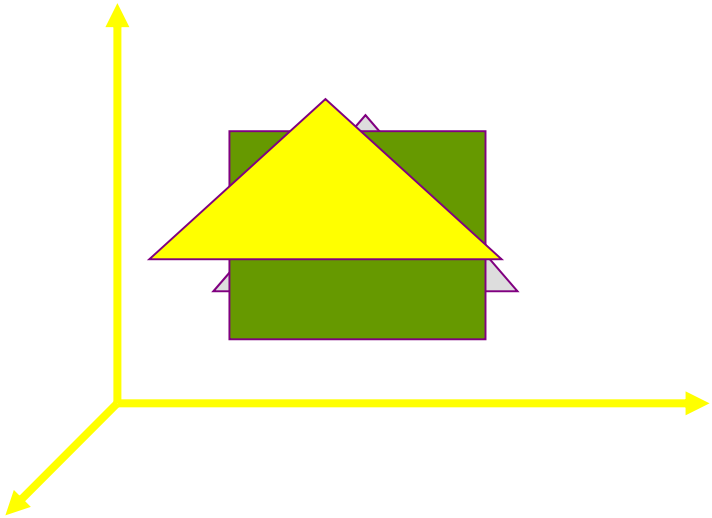
8.2 用增量公式计算新的x_l 、 x_r 和z_l

}

}

2.7.3.3 扫描线Z-buffer算法

- 缺点:
  - 在每一个被多边形覆盖像素处需要计算深度值;
  - 被多个多边形覆盖的像素需要多次计算深度值。



2.7.3.4 区间扫描线算法

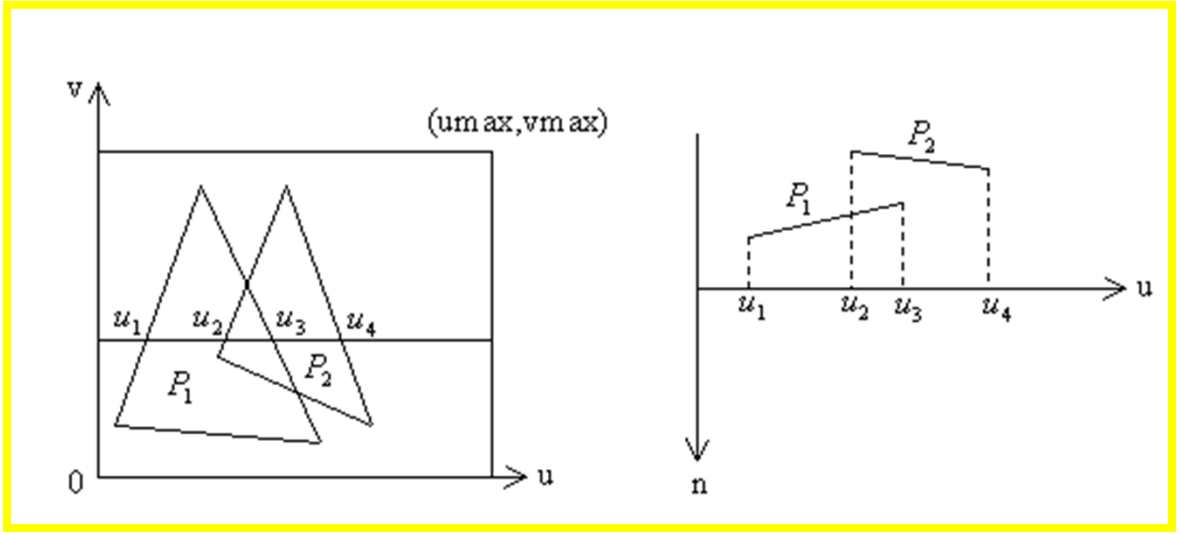
- 与Z-Buffer算法相比，扫描线算法有了很大改进。
  - 优点：
    - 将整个绘图窗口内的消隐问题分解到一条条扫描线上解决，使所需的Z-Buffer大大减少；
    - 计算深度值时，利用了面连贯性，只用了一个加法 。
  - 缺点：
    - 每个像素处都计算深度值，甚至不止一次的计算（多边形重叠区域），运算量仍然很大。
  - 改进：
    - 在一条扫描线上，每个区间只计算一次深度，即区间扫描线算法，又称扫描线算法。

2.7.3.4区间扫描线算法

- 基本思想：
  - 把当前扫描线与各多边形在投影平面的投影的交点进行排序后，使扫描线分为若干子区间。只要在区间任一点处找出在该处z值最大的一个面，这个区间上的每一个象素就用这个面的颜色来显示。

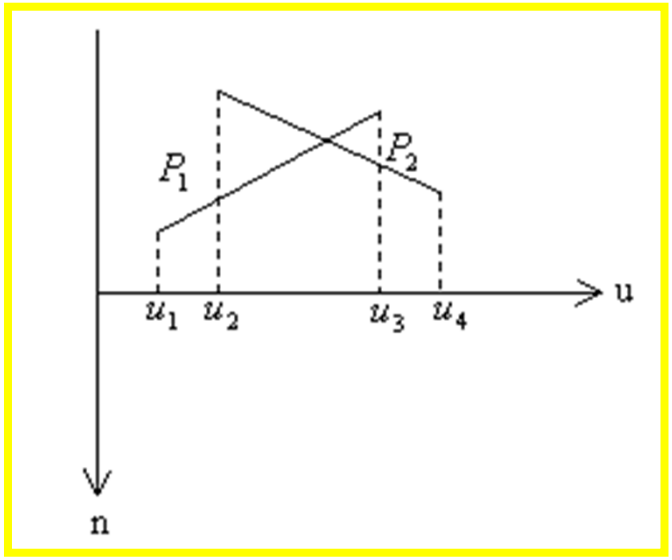
2.7.3.4区间扫描线算法

- 如下图，多边形P_1、P_2的边界在投影平面上的投影将一条扫描线划分成若干个区间$[0, u_1][u_1, u_2][u_2, u_3][u_3, u_4], [u_4, u_{max}]$，覆盖每个区间的有0个、1个或多个多边形，但仅有一个可见。在区间上任取一个像素，计算该像素处各多边形的深度值，深度值最大者即为可见多边形，用它的颜色显示整个区间。



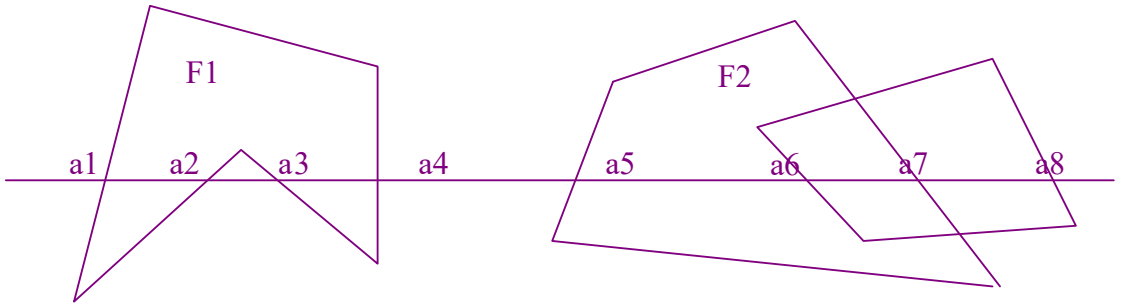
2.7.3.4区间扫描线算法

- **注意**：该算法要求多边形不能相互贯穿，否则在同一区间上，多边形深度值的次序会发生变化。
- 如图：在区间$[u_1, u_2]$上，多边形P_1的深度值大，在区间$[u_3, u_4]$上，多边形P_2的深度值大，而在区间$[u_2, u_3]$上，两个多边形的深度值次序发生交替。

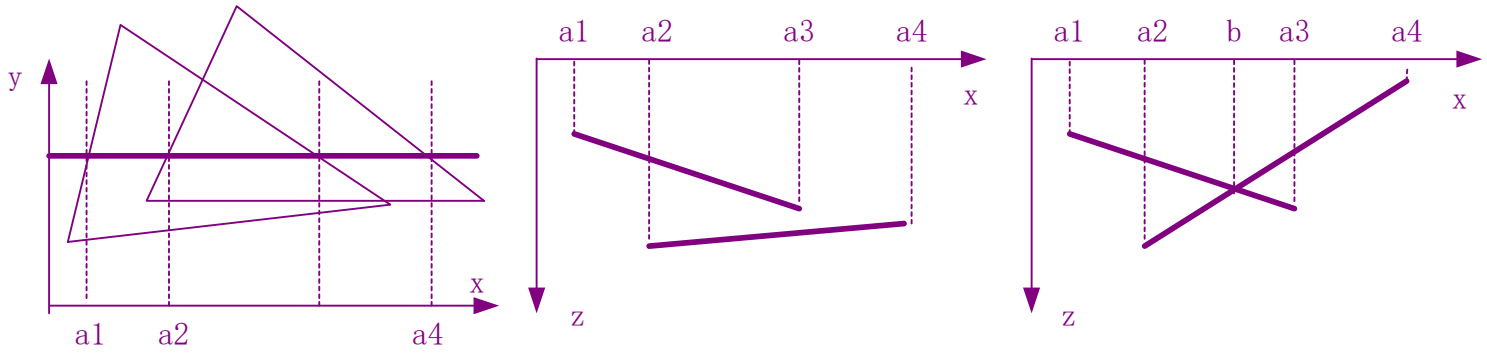


2.7.3.4区间扫描线算法

- 如何确定小区间的颜色可分为三种情况：
  - 小区间上没有任何多边形，如$[a_4, a_5]$，这时该小区间用背景色显示；
  - 小区间上只有一个多边形，如$[a_1, a_2][a_5, a_6]$这时可以对应多边形在该处的颜色显示；
  - 小区间上存在两个或两个以上的多边，形如$[a_6, a_7]$，必须通过深度测试判断哪个多边形可见。



2.7.3.4区间扫描线算法



两个平面在屏幕上的投影　　无贯穿的情形　　相互贯穿的情形

– 若允许物体表面相互贯穿时，还必须求出它们在扫描平面的交点。用这些交点把该小区间分成更小的子区间，在这些间隔上决定哪个多边形可见。如将$[a_2,a_3]$区间分成$[a_2,b][b,a_3]$两个子区间。

2.7.3.4区间扫描线算法

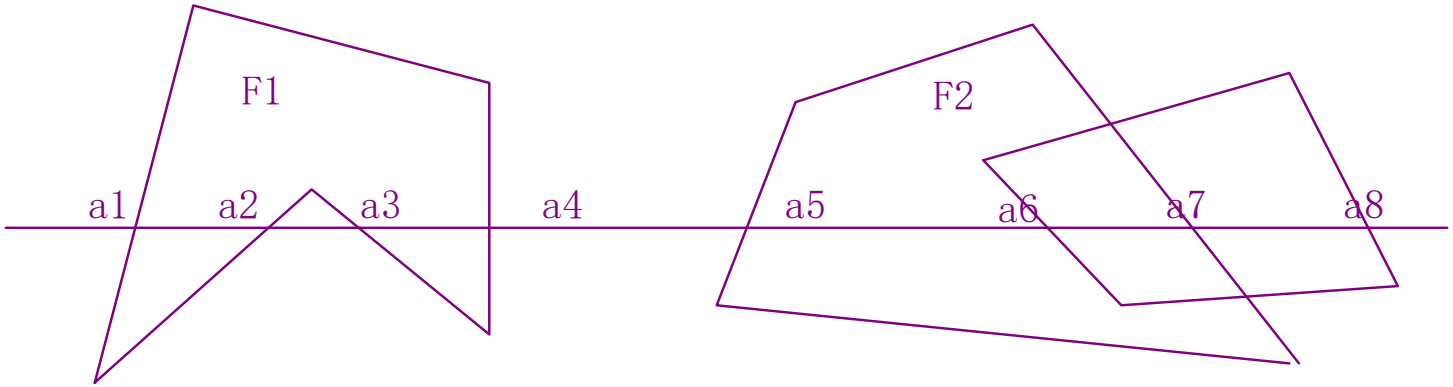
- 确定某间隔内哪一多边形可见：
  - 可在间隔内任取一采样点(如中点)，分析该点处哪个多边形离视点最近，该多边形即是在该间隔内可见的多边形。

2.7.3.4区间扫描线算法

- 类似于扫描线Z-Buffer算法中的数据结构
  - 多边形分类表
  - 活化多边形表
  - 边表
  - 活化边表
- 活化边表中的结点是边，而非边对。
- 如何知道每一个区间中，有几个相关的多边形？是哪几个？

2.7.3.4区间扫描线算法

- 解决方案：活化多边形表中增加一个标志，flag=0, 每遇到它的边，flag取反。



2.7.3.4区间扫描线算法

- 算法描述

for (绘图窗口内的每一条扫描线）

{ 求投影与当前扫描线相交的所有多边形

求上述多边形中投影与当前扫描线相交的所有边，将它们记录在活化边表AEL中

求AEL中每条边的投影与扫描线的交点；

按交点的u坐标将AEL中各边从左到右排序， 两两配对组成一个区间；

for （AEL中每个区间）

{

求覆盖该区间的所有多边形，将它们记入活化多边形表APL中；

在区间上任取一点，计算APL中各多边形在该点的深度值，记深度最大者为P；

用多边形P的颜色填充该区间

}

}

2.7.3.4区间扫描线算法

– 算法的优点：将逐点(象素、Pixel)计算
改为逐段计算

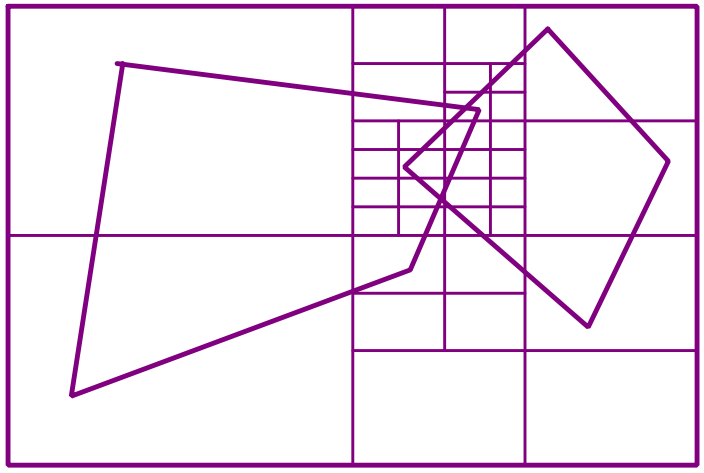
效率大大提高！

2.7.3.5区域子分割算法

- **Z缓冲器算法**与**扫描线Z缓冲器算法**中都是将像素孤立来考虑，未利用相邻像素之间存在的属性的连贯性，即区域的连贯性，所以算法效率不高。
- 实际上，可见多边形至少覆盖了绘图窗内的一块区域，这块区域由多边形在投影平面上的投影的边界围成。
  - 如果能将这类区域找出来，再用相应的多边形颜色加以填充则避免了在每个像素处计算深度值，消隐问题也就解决了。

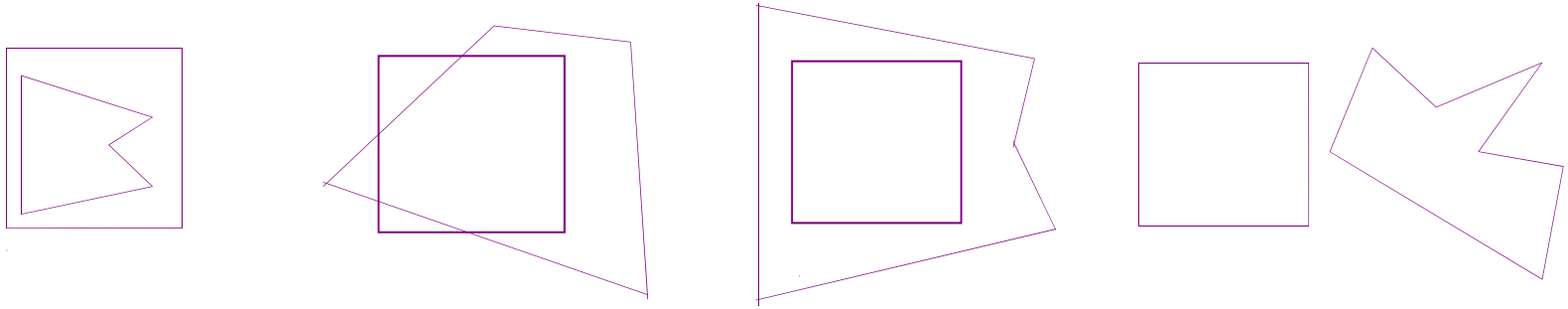
2.7.3.5区域子分割算法

- **基本思路：** 首先将场景中的多边形投影到绘图窗口内，判断窗口是否足够简单，若是则算法结束；否则将窗口进一步分为四块。对此四个小窗口重复上述过程，直到窗口仅为一个像素大小。此时可能有多个多边形覆盖了该像素，计算它们的深度值，以最近的颜色显示该像素即可。



2.7.3.5区域子分割算法

- 何谓窗口足够简单？
  - 窗口为空，即多边形与窗口的关系是分离的；
  - 窗口内仅含一个多边形，即有一个多边形与窗口的关系是包含或相交。此时先对多边形投影进行裁剪，再对裁剪结果进行填充；
  - 有一个多边形的投影包围了窗口，并且它是最靠近观察点的，以该多边形颜色填充窗口。

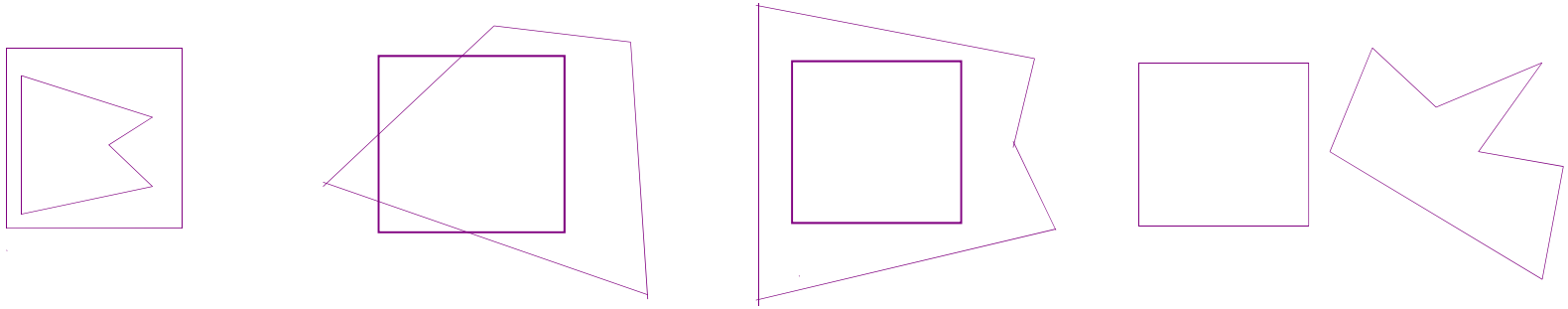


内含　　多边形与窗口相交　　包围　　窗口和多边形分离

2.7.3.5区域子分割算法

- 分离和包围多边形的判别方法：
  - 射线检查法；
  - 转角累计检查法；
  - 区域检查法。

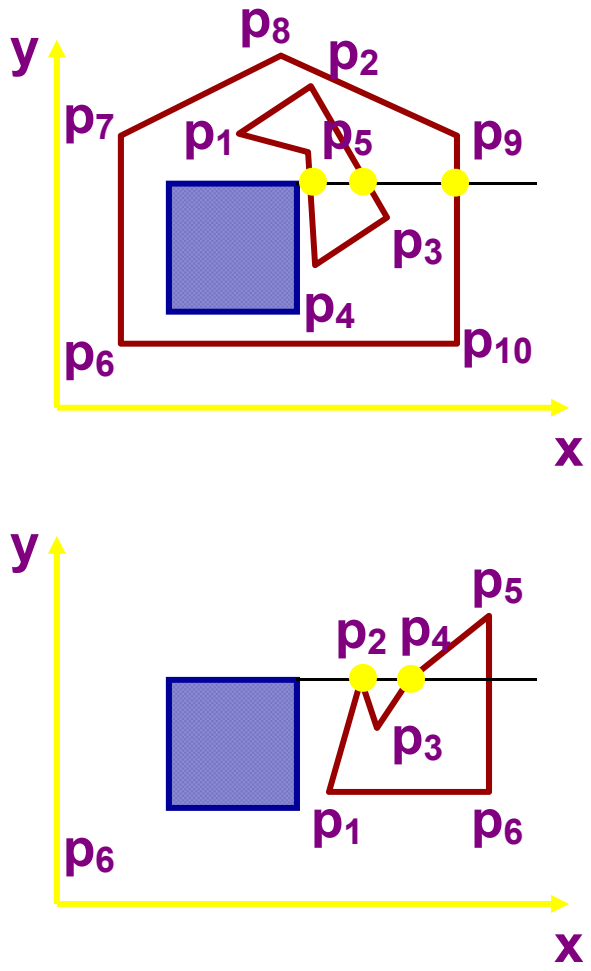
这三种检查都假定相交的和内含的多边形已经事先判定了！



内含　　多边形与窗口相交　　包围　　窗口和多边形分离

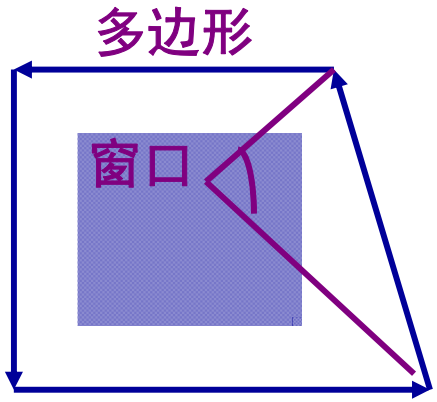
2.7.3.5区域子分割算法

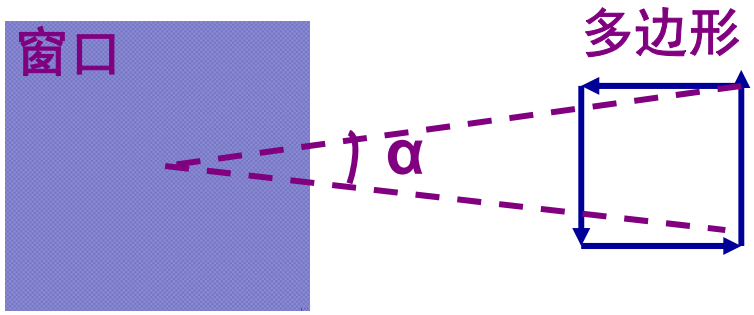
- 射线检查法
  - 从窗口内的任意点，画一条射线至无穷远处，累计射线与多边形交点的个数。如果为偶数，则此多边形与窗口分离；否则，此多边形包围窗口。但当射线通过多边形的顶点时，会得出错误结论。
  - **解决方法**：当射线通过多边形的局部极值顶点时，记入两个交点；当射线通过多边形的非局部极值顶点时，记入一交点。



2.7.3.5区域子分割算法

- **转角累计检查法**:按顺时针方向或逆时针方向绕多边形依次累加多边形各边起点与终点对窗口内任意一点所张的夹角。按累计角度之和可以判定:
  - 若角度之和等于0，则表示多边形与窗口分离;
  - 若角度之和等于±360°·n，则表示多边形包围窗口(n次)。





2.7.3.5区域子分割算法

- 区域检查法
  - 区域编码
  - 多边形顶点编码
  - 多边形边的编码
  - 多边形的编码

2.7.3.5区域子分割算法

- 区域编码
  - 窗口四条边所在直线将屏幕划分成9个区域，对窗口以外的8个区域按逆时针进行编码，编码为0~7
- 多边形顶点编码
  - 多边形顶点编码。多边形$v_0v_1\ldots v_n$的顶点v_i的投影落在哪个区域，那个区域的编码便作为该顶点的编码，记为I_i。

3	2	1
4	窗口	0
5	6	7

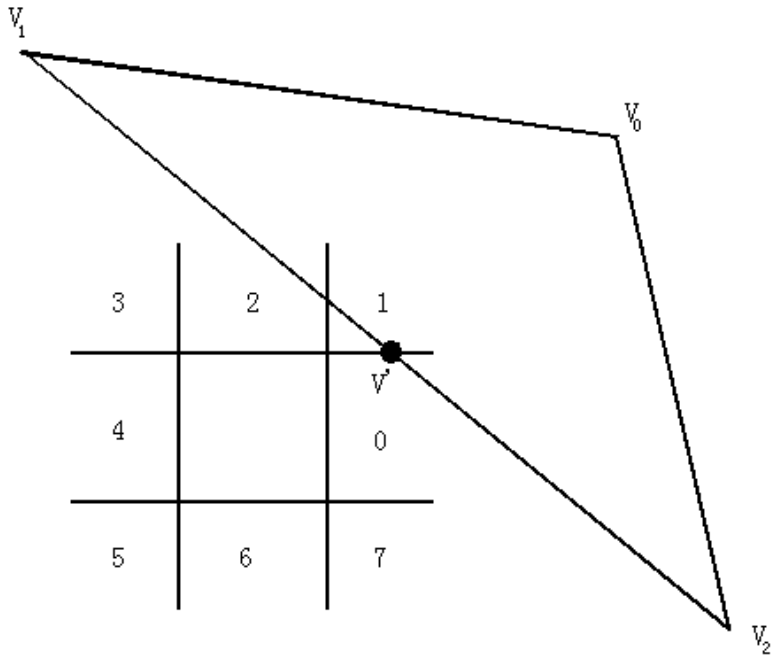
2.7.3.5区域子分割算法

- 多边形边的编码
  - 多边形的边v_iv_{i+1}的编码定义为$\Delta_i=I_{i+1}-I_i$，i=0,1…n；其中，令$I_{n+1}=I_0$，并且，当$\Delta_i>4$时，取$\Delta_i=\Delta_i-8$；当$\Delta_i<-4$时，取$\Delta_i=\Delta_i+8$；
  - 当$\Delta_i=+/-4$时，取该边与窗口边的延长线的交点将该边分为两段，对两段分别按上面的规则编码，再令Δ_i等于两者之和。
- 多边形的编码
  - 其边的编码之和。

$$\begin{cases} \text{窗口与多边形分离} & \text{当} \sum_{i=0}^{n} \Delta_i = 0 \\ \text{多边形包围窗口} & \text{当} \sum_{i=0}^{n} \Delta_i = \pm 8 \end{cases}$$

2.7.3.5区域子分割算法

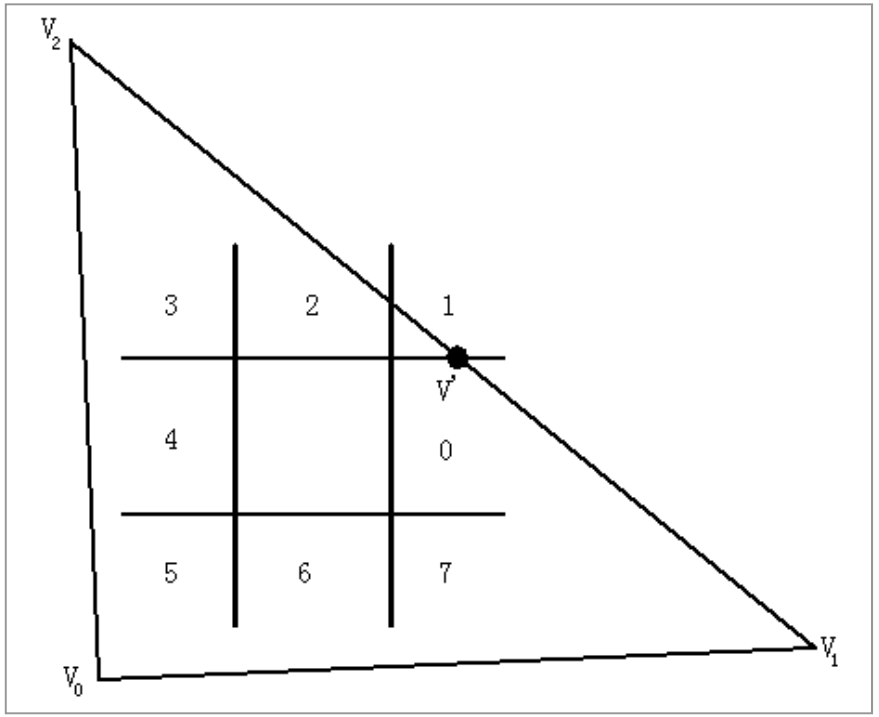
- 例如：I_0=1,I_1=3,I_2=7, Δ_0=3-1=2, Δ_1=7-3=4, Δ_2=1-7=-6。按第3步处理规则，取v_1v_2与窗口上边所在直线的交点v′将其分为两段，两段的编码分别为-2，-2，从而Δ_1=-2+（-2）=-4。而Δ_2=-6+8=2。最终求出多边形的编码为$\Delta_0+\Delta_1+\Delta_2$=2+（-4）+2=0。
- 结论：多边形与窗口分离。



2.7.3.5区域子分割算法

- 例如：I_0=5,I_1=7,I_2=3, Δ_0=7-5=2, Δ_1=3-7=-4, Δ_2=5-3=2, 因为Δ_1=-4,按第三步的处理规则，取边v_1v_2与窗口上边所在直线的交点将v_1v_2分为两段，两段的编码分别为2，2，从而Δ_2=4。最终求出多边形的编码为

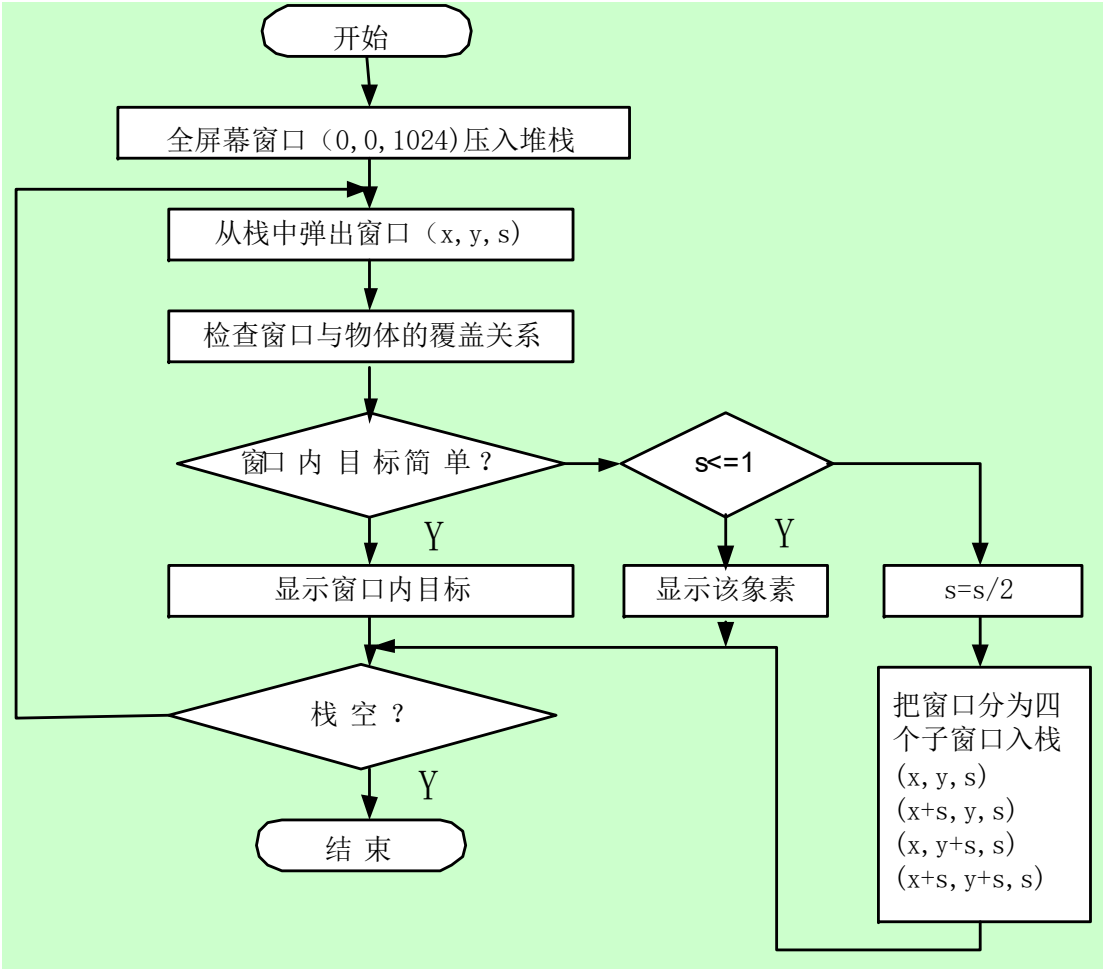
  $\Delta_0+\Delta_1+\Delta_2$=2+4+2=8，因此，该三角形包围窗口。



2.7.3.5区域子分割算法

- 假设全屏幕窗口分辨率1024*1024，定义窗口左下角点(x,y)和边宽s。
- 使用堆栈结构实现区域子分割算法流程图。
- 由于算法中每次递归地把窗口分割成4个小窗口，故又称为四叉树算法。

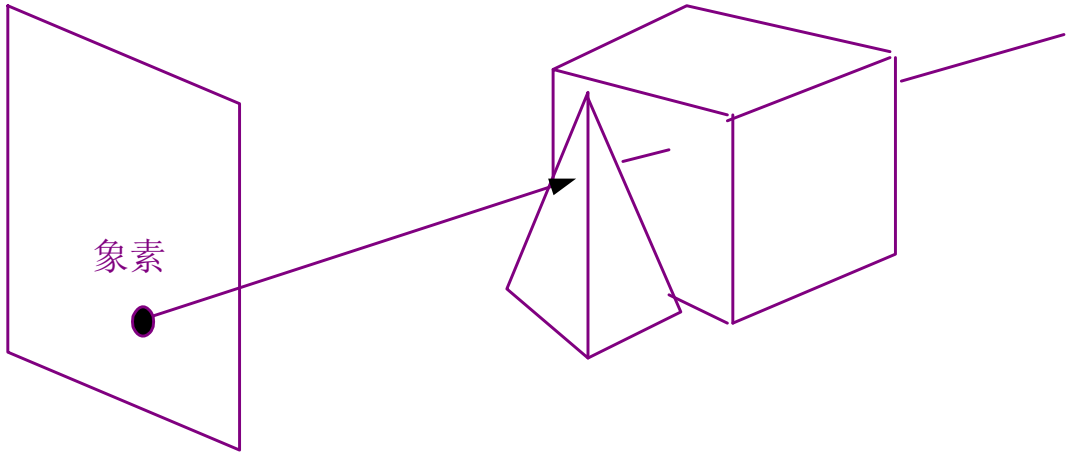
2.7.3.5区域子分割算法



区域子分割算法流程图

2.7.3.6光线投射算法

- 基本思想：
  - 考察由视点出发穿过观察屏幕一象素而射入场景的一条射线，则可确定出场景中与该射线相交的物体；
  - 在计算出光线与物体表面的交点之后，离象素最近的交点的所在面片的颜色为该象素的颜色；
  - 如果没有交点，说明没有多边形的投影覆盖此象素，用背景色显示它即可。



2.7.3.6光线投射算法

- 算法流程：

```
for屏幕上的每一象素
 {形成通过该屏幕象素(u,v)的射线；
   for 场景中的每个物体
    {将射线与该物体求交；
      if 存在交点
          以最近的交点所属的颜色显示象素(u,v)
       else
          以背景色显示象素(u,v)
    }
 }
```

2.7.3.6光线投射算法

- 光线投射算法与Z缓冲器算法相比，它们仅仅是内外循环颠倒了一下顺序，算法复杂度类似
  - 区别在于光线投射算法不需要Z缓冲器；
  - 为了提高本算法的效率可以使用包围盒技术，空间分割技术以及物体的层次表示方法等来加速。

本章总结

- 本章总结：
  - 直线段的扫描转换算法
  - 圆弧的扫描转换算法
  - 多边形的扫描转换与区域填充
  - 字符
  - 裁剪
  - 反走样
  - 消隐

本章总结

- 第二章的核心思想
  - 增量思想
  - 编码思想
  - 符号判别—> 整数算法
  - 图形连贯性