计算机图形学

第二章：光栅图形学算法

光栅图形学算法的研究内容

- 直线段的扫描转换算法
- 多边形的扫描转换与区域填充算法
- 直线裁剪算法
- 反走样算法
- 消隐算法

主要讲述的内容：

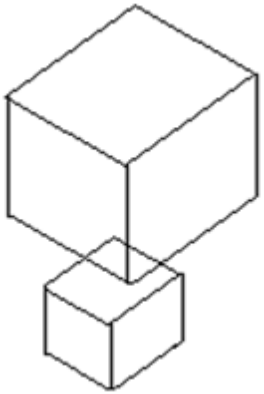
消隐的分类，如何消除隐藏线、隐藏面，主要介绍以下几个算法：

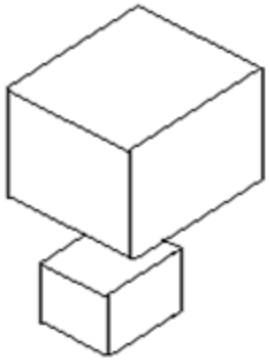
- Z缓冲区(Z-Buffer)算法
- 扫描线Z-buffer算法
- 区域子分割算法

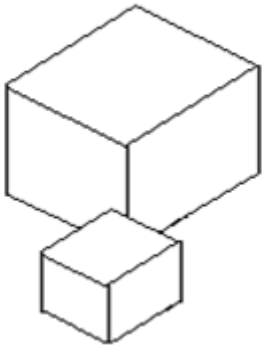
一、消隐

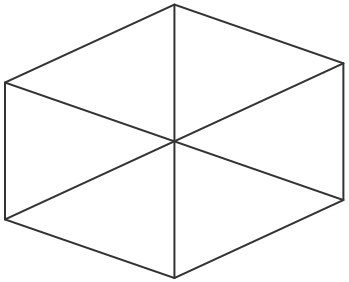
当我们观察空间任何一个不透明的物体时，只能看到该物体朝向我们的那些表面，其余的表面由于被物体所遮挡我们看不到

如果把可见和不可见的线都画出来，对视觉会造成多义性

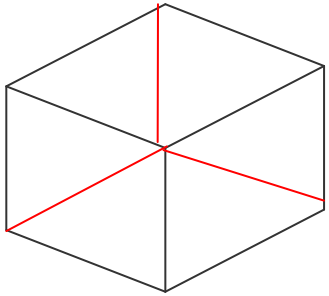




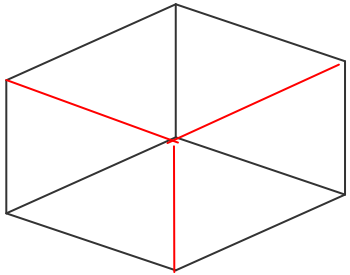




（a）



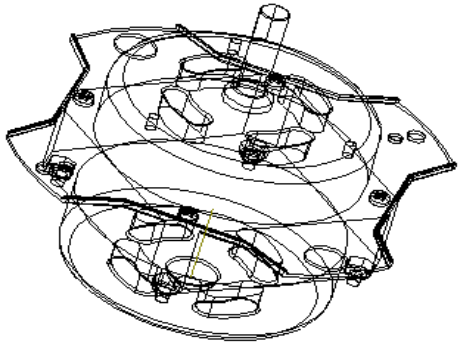
（b）



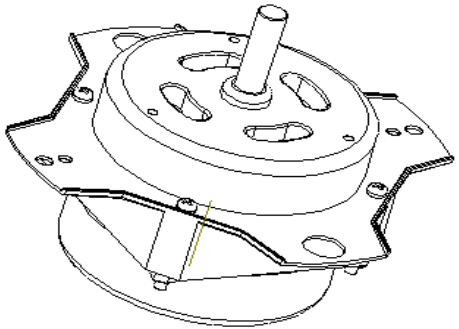
（c）

要消除二义性，就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，简称为**消隐**。

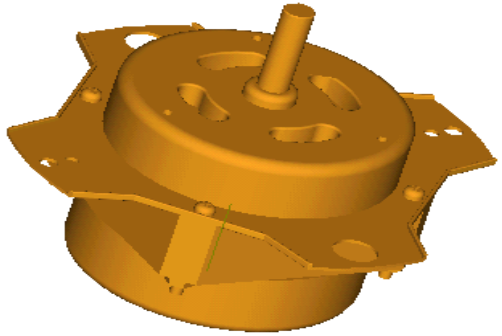
要绘制出意义明确的、富有真实感的立体图形，首先必须消去形体中的不可见部分，而只在图形中表现可见部分



线框图



消隐图

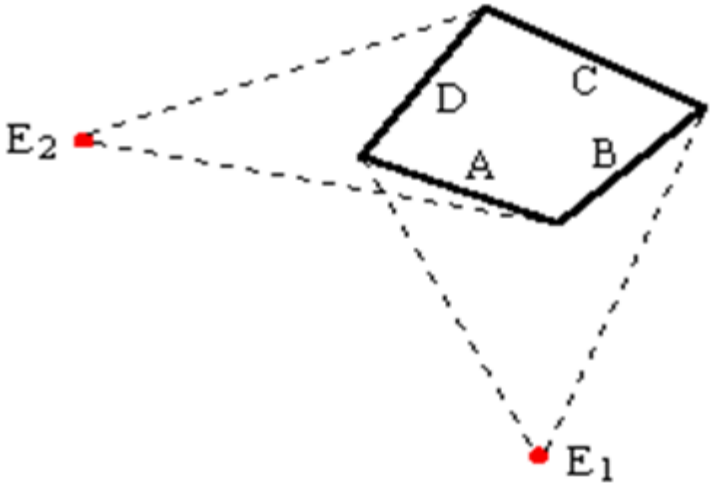


真实感图形

消隐包括消除“隐藏线”和“隐藏面”两个问题

到目前为止，虽然已有数十种算法被提出来了，但是由于物体的形状、大小、相对位置等因素千变万化，因此至今它仍吸引人们作出不懈的努力去探索更好的算法

消隐不仅与消隐对象有关，还与观察者的位置有关



消隐处理从原理上讲并不复杂，但是消隐处理的具体实现并不那么简单，它要求适当的算法及大量的运算。在60年代，消隐问题曾被认为是计算机图形学中的几大难题之一

二、消隐的分类

1、按消隐对象分类

（1）线消隐

消隐对象是物体上的边，消除的是物体上不可见的边

（2）面消隐

消隐对象是物体上的面，消除的是物体上不可见的面，通常做真实感图形消隐时用面消隐

2、按消隐空间分类

（1）物体空间的消隐算法

以场景中的物体为处理单元。假设场景中有k个物体，将其中一个物体与其余k-1个物体逐一比较，仅显示它可见表面以达到消隐的目的

此类算法通常用于线框图的消隐！

```
for（场景中的每一个物体）
  { 将该物体与场景中的其
    它物体进行比较，确定
    其表面的可见部分；
    显示该物体表面的可见
    部分；
  }
```

在物体空间里典型的消隐算法有两个： Roberts算法和光线投射法

Roberts算法数学处理严谨，计算量甚大。算法要求所有被显示的物体都是凸的，对于凹体要先分割成多个凸体的组合

Roberts算法基本步骤：

- 逐个的独立考虑每个物体自身，找出为其自身所遮挡的边和面（自消隐）；
- 将每一物体上留下的边再与其它物体逐个的进行比较，以确定是完全可见还是部分或全部遮挡（两两物体消隐）；
- 确定由于物体之间的相互贯穿等原因，是否要形成新的显示边等，从而使被显示各物体更接近现实

光线投射是求光线与场景的交点，该光线就是所谓的视线（如视点与像素连成的线）

一条视线与场景中的物体可能有许多交点，求出这些交点后需要排序，在前面的才能被看到。人的眼睛可以一目了然，但计算机做需要大量的运算

（2）图像空间的消隐算法

以屏幕窗口内的每个像素为处理单元。确定在每一个像素处，场景中的k个物体哪一个距离观察点最近，从而用它的颜色来显示该像素

```
for（窗口中的每一个像素）
  {确定距视点最近的物体，
   以该物体表面的颜色来显
   示像素；
  }
```

这类算法是消隐算法的主流！

因为最后看到的图像是在屏幕上的，所以就拿屏幕作为处理对象。针对屏幕上的像素来进行处理，算法的思想是围绕着屏幕的。对屏幕上每个象素进行判断，决定哪个多边形在该象素可见。

三、图像空间的消隐算法

这类算法是消隐算法的主流！

Z一buffer算法

扫描线算法

Warnock消隐算法

1、Z缓冲区(Z-Buffer)算法

1973年，犹他大学学生艾德·卡姆尔（Edwin Catmull）独立开发出了能跟踪屏幕上每个像素深度的算法 Z-buffer

Z-buffer让计算机生成复杂图形成为可能。Ed Catmull目前担任迪士尼动画和皮克斯动画工作室的总裁

Z缓冲器算法也叫深度缓冲器算法，属于图像空间消隐算法

该算法有帧缓冲器和深度缓冲器。对应两个数组：

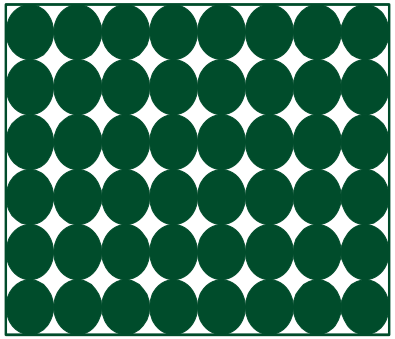
intensity（x，y）——属性数组（帧缓冲器）

存储图像空间每个可见像素的光强或颜色

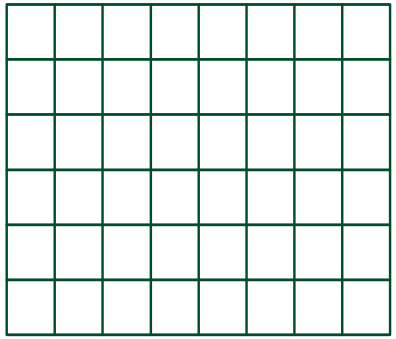
depth（x，y）——深度数组（z-buffer）

存放图像空间每个可见像素的z坐标

屏幕

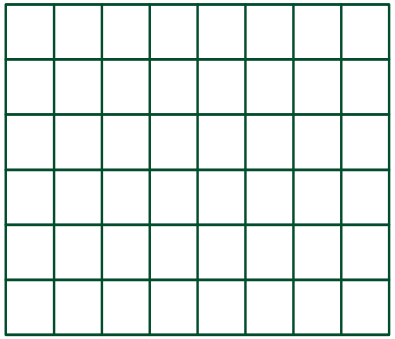


帧缓冲器



每个单元存放对应
象素的颜色值

Z缓冲器

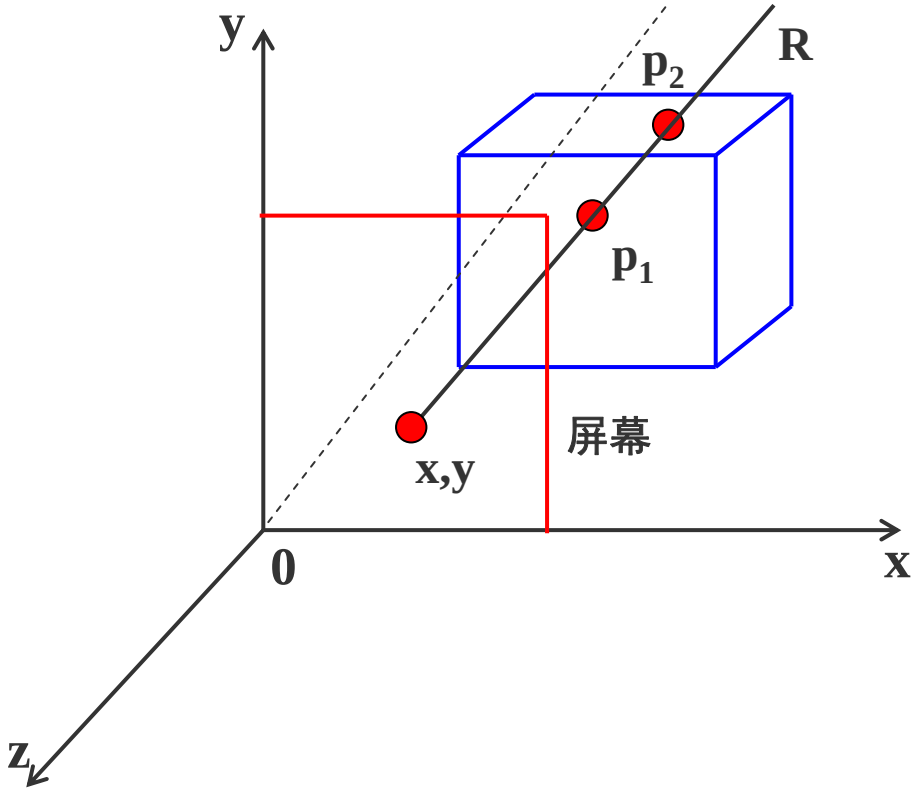


每个单元存放对应
象素的深度值

假定xoy面为投影面，z轴为观察方向

过屏幕上任意像素点（x，y）作平行于z轴的射线R，与物体表面相交于p_1和p_2点

p_1和p_2点的z值称为该点的深度值

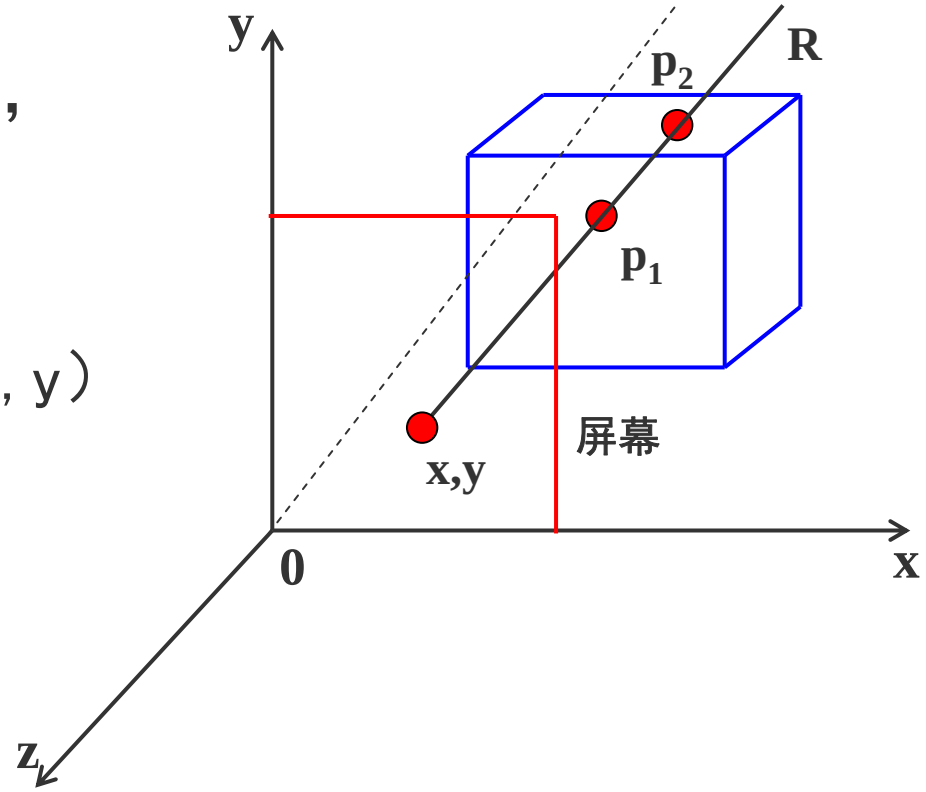


z-buffer算法比较p_1和p_2的z值，

将最大的z值存入z缓冲器中

显然，p_1在p_2前面，屏幕上（x, y）

这一点将显示p_1点的颜色



算法思想：先将Z缓冲器中各单元的初始值置为最小值。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（保存在该像素所对应的Z缓冲器的单元中）

如果大于原来的z值，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

```
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
               }
           }
       }
 }
```

z-Buffer算法的优点：

（1）Z-Buffer算法比较简单，也很直观

（2）在象素级上以近物取代远物。与物体在屏幕上的出现顺序是无关紧要的，有利于硬件实现

z-Buffer算法的缺点：

（1）占用空间大

（2）没有利用图形的相关性与连续性，这是z-buffer算法的严重缺陷

（3）更为严重的是，该算法是在像素级上的消隐算法

2、只用一个深度缓存变量zb的改进算法

一般认为，z-Buffer算法需要开一个与图象大小相等的缓存数组ZB，实际上，可以改进算法，只用一个深度缓存变量zb

```
z-Buffer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Pk)
           {
              if(象素点(i,j)在pk的投影多边形之内)
           {
                    计算Pk在(i,j)处的深度值depth;
                       if(depth大于zb)
                    { zb = depth;
                   indexp = k;（记录多边形的序号）
                    }
                 }
            }
        If(zb != MinValue)  计算多边形P_indexp在交点 (I,j) 处的光照
             颜色并显示
       }
 }
```

关键问题：判断象素点(i，j)是否在pk的投影多边形之内，不是一件容易的事。节省了空间但牺牲了时间。计算机的很多问题就是在时间和空间上找平衡

另一个问题计算多边形Pk在点（i，j）处的深度。设多边形Pk的平面方程为：

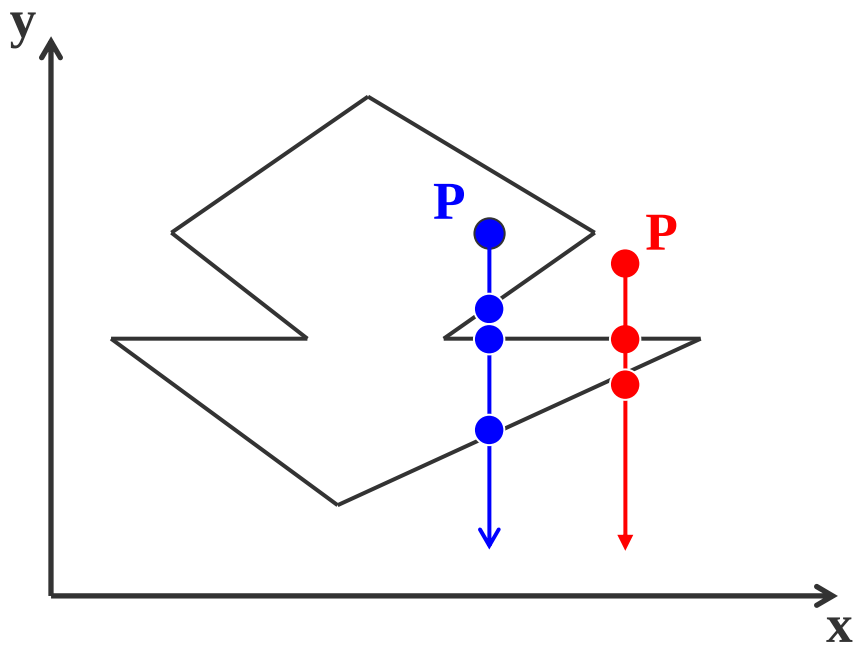
$$ax + by + cz + d = 0 \qquad depth = -\frac{ai + bj + d}{c}$$

点与多边形的包含性检测：

（1）射线法

由被测点P处向 y = − ∞方向作射线

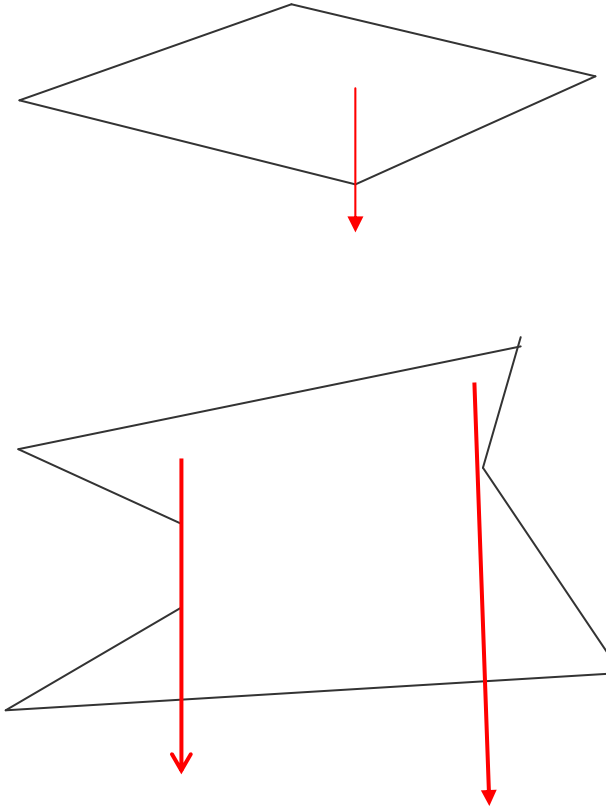
交点个数是奇数，则被测点在多边形内部



交点个数是偶数表示在多边形外部

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

即：当射线与某条边的顶点相交时，若边在射线的左侧，交点有效，计数；若边在射线的右侧，交点无效，不计数

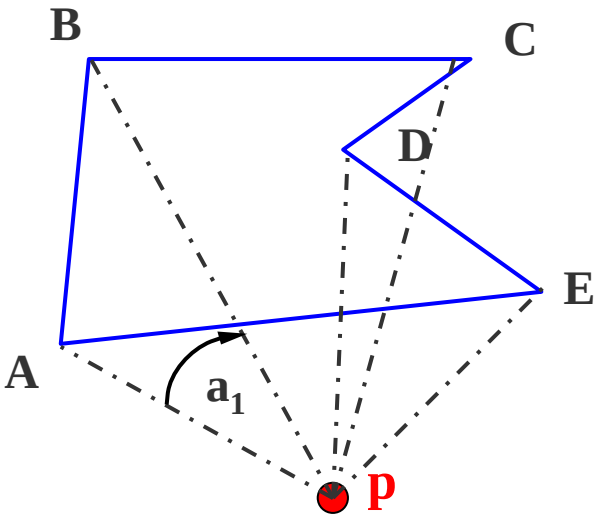


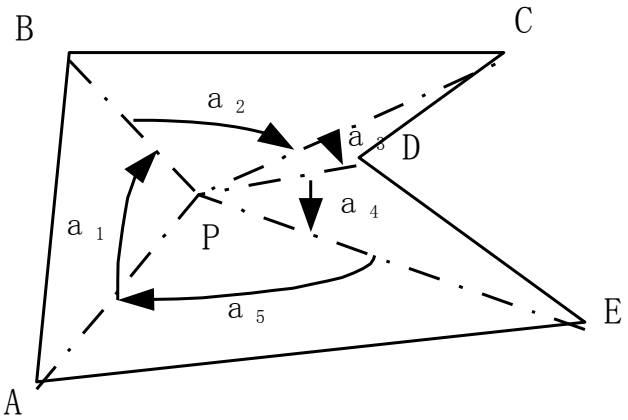
用射线法来判断一个点是否在多边形内的弊端：

（1）计算量大

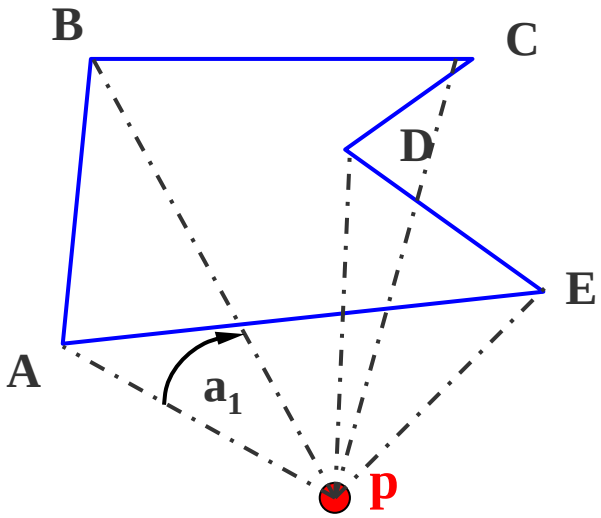
（2）不稳定

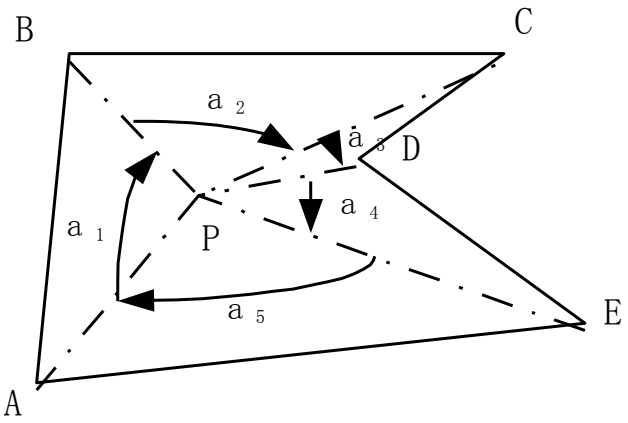
（2）弧长法





以p点为圆心，作单位圆，把边投影到单位圆上，对应一段段弧长，规定逆时针为正，顺时针为负，计算弧长代数和





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

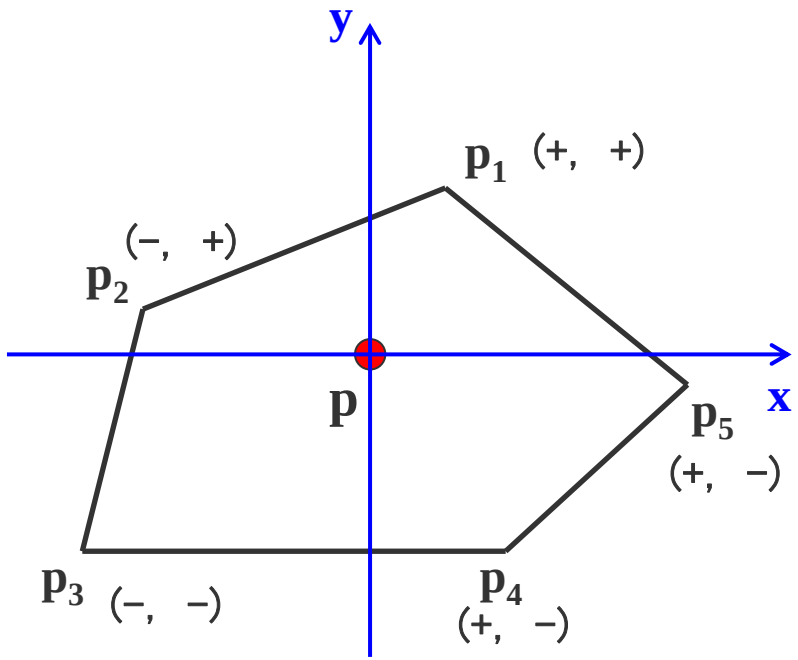
这个算法为什么是稳定的？假如算出来后代数和不是0，而是0.2或0.1，那么基本上可以断定这个点在外部，可以认为是有计算误差引起的，实际上是0。

但这个算法效率也不高，问题是算弧长并不容易，因此又派生出一个新的方法—以顶点符号为基础的弧长累加方法

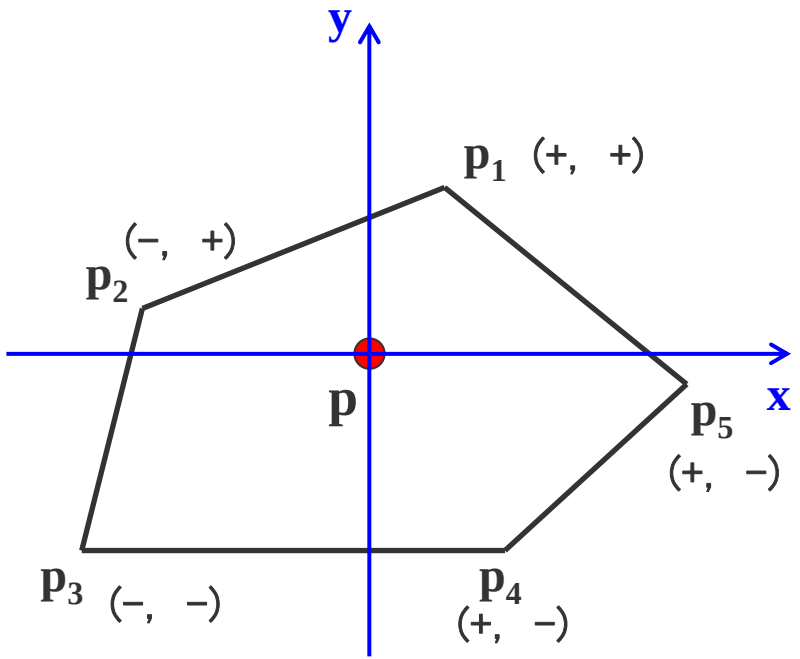
（3）以顶点符号为基础的弧长累加方法

p是被测点，按照弧长法，p点的代数和为2π

不要计算角度，做一个规定来取代原来的弧长计算



		弧长变化	象限变化
(+ +)	(+ +)	0	Ⅰ → Ⅰ
(+ +)	(− +)	π/2	Ⅰ → Ⅱ
(+ +)	(− −)	±π	Ⅰ → Ⅲ
(+ +)	(+ −)	−π/2	Ⅰ → Ⅳ
…	…	…	…



同一个象限认为是0，跨过一个象限是π/2，跨过二个象限是π。这样当要计算代数和的时候，就不要去投影了，只要根据点所在的象限一下子就判断出多少度，这样几乎没有什么计算量，只有一些简单的判断，效率非常高

z-buffer算法是非常经典和重要的，在图形加速卡和固件里都有。只用一个深度缓存变量zb的改进算法虽然减少了空间，但仍然没考虑相关性和连贯性

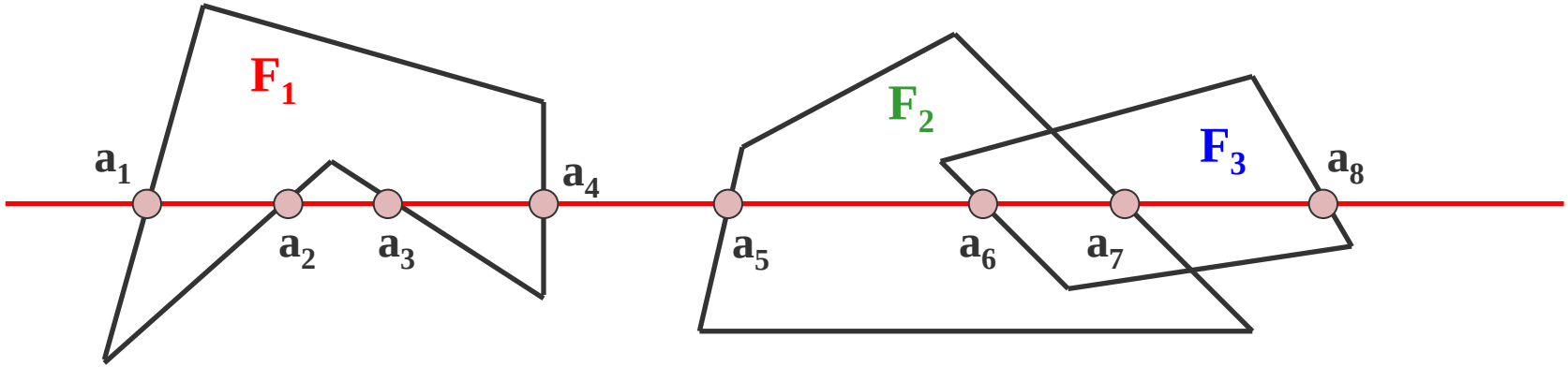
二、区间扫描线算法

前面介绍了经典的z-buffer算法，思想是开一个和帧缓存一样大小的存储空间，利用空间上的牺牲换取算法上的简洁

还介绍了只开一个缓存变量的z-buffer算法，是把问题转化成判别点在多边形内，通过把空间多边形投影到屏幕上，判别该像素是否在多边形内

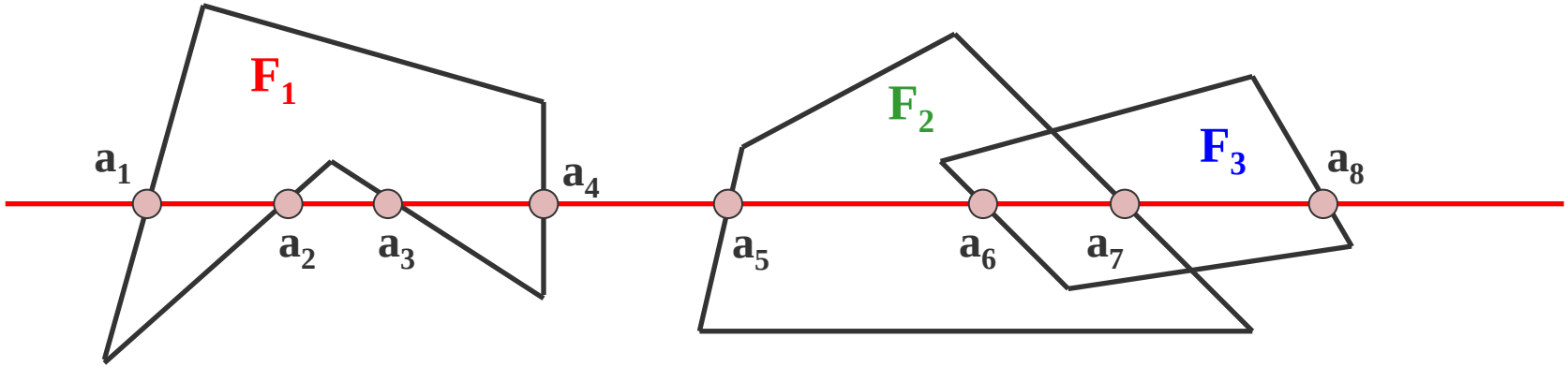
下面介绍区间扫描线算法。该算法放弃了z-buffer的思想，是一个新的算法，这个算法被认为是消隐算法中最快的

因为不管是哪一种z-buffer算法，都是在像素级上处理问题，要进行消隐，每个像素都要进行计算判别，甚至一个像素要进行多次（一个像素可能会被多个多边形覆盖）



扫描线的交点把这条扫描线分成了若干个区间，每个区间上必然是同样一种颜色

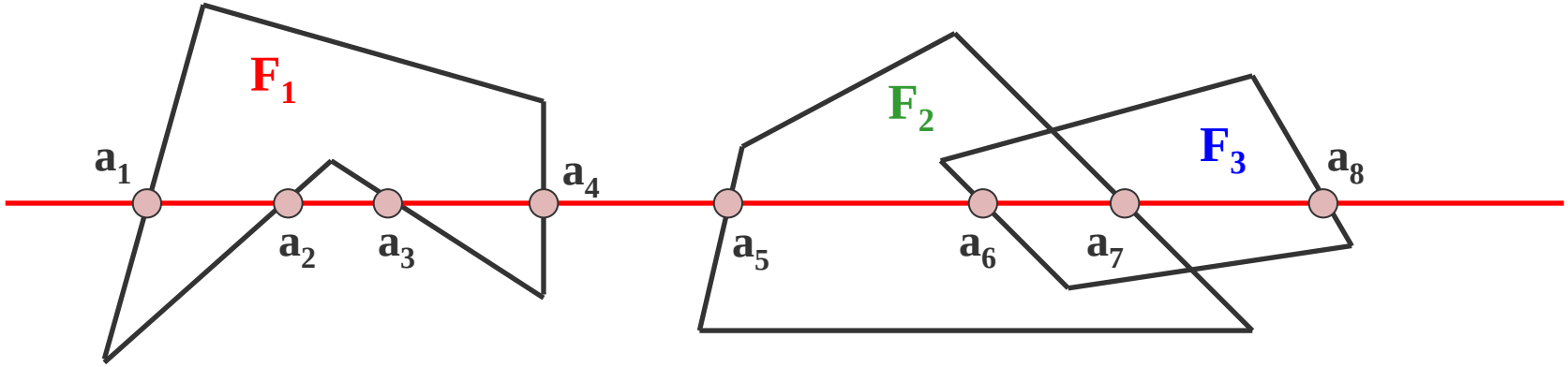
对于有重合的区间，如a6a7这个区间，要么显示F2的颜色，要么显示F3的颜色，不会出现颜色的跳跃



如果把扫描线和多边形的这些交点都求出来，对每个区间，只要判断一个像素的要画什么颜色，那么整个区间的颜色都解决了，这就是区间扫描线算法的主要思想

算法的优点：将象素计算改为逐段计算，效率大大提高！

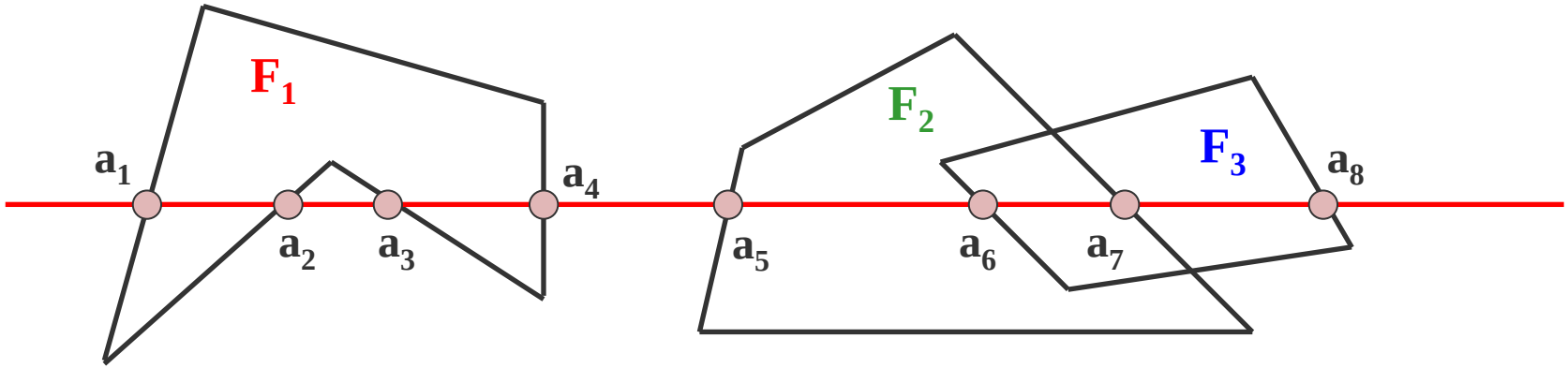
如何实现这个算法？



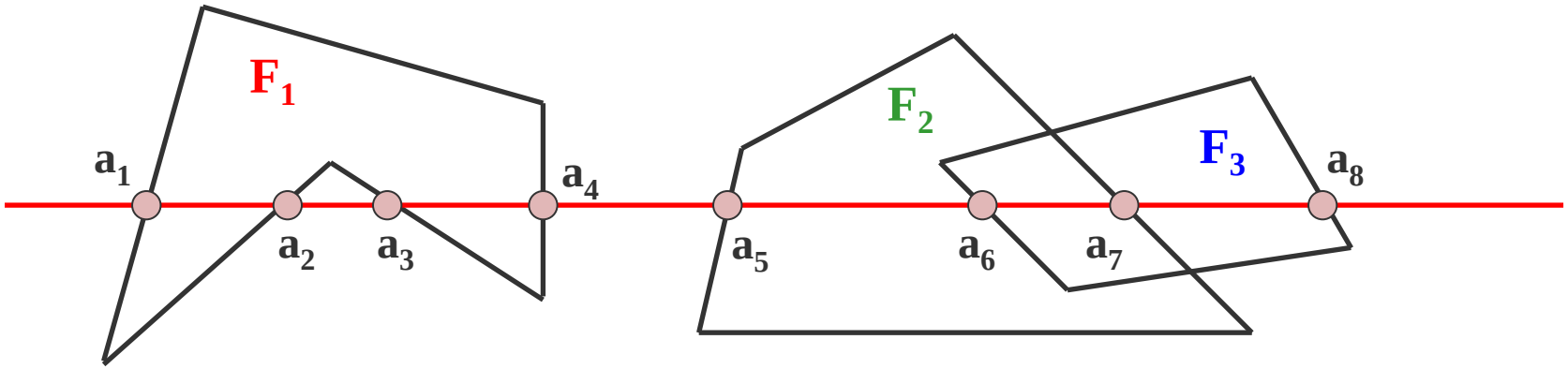
首先要有投影多边形，然后求交点，然后交点进行排序排序

排序的结果就分成了一个个区间，然后在每个区间找当中的一个像素（i，j），在（i，j）处计算每个相关面的z值，对相关深度值z进行比较，其中最大的一个就表示是可见的。整个这段区间就画这个z值最大面的颜色

如何确定小区间的颜色?



（1）小区间上没有任何多边形，如[a4, a5]，用背景色显示



（2）小区间只有一个多边形，如[a1, a2]，显示该多边形的颜色

（3）小区间上存在两个或两个以上的多边，比如[a6, a7]， 必须通过深度测试判断哪个多边形可见

这个算法存在几个问题：

1、真的去求交点吗？ 利用增量算法简化求交！

2、每段区间上要求z值最大的面，这就存在一个问题。如何知道在这个区间上有哪些多边形是和这个区间相关的？

三、区域子分割算法（ Warnock算法）

John E. Warnock 博士，Adobe创始人之一，曾担任董事会主席

In his 1969 doctoral thesis, Warnock invented the Warnock algorithm for hidden surface determination in computer graphics

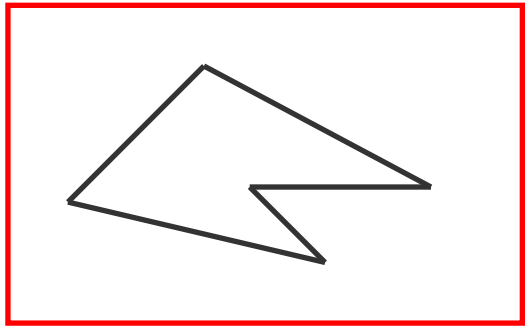
Warnock算法是图像空间中非常经典的一个算法

Warnock算法的重要性不在于它的效率比别的算法高，而在于采用了分而治之的思想，利用了堆栈的数据结构

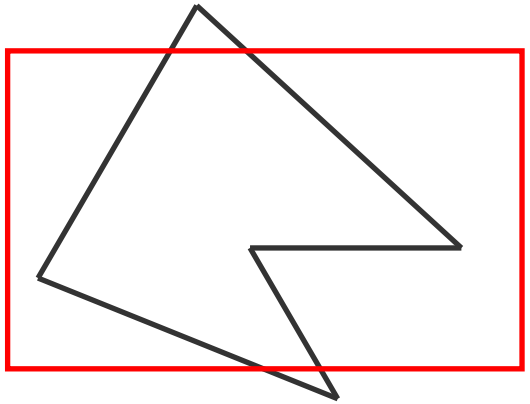
把物体投影到全屏幕窗口上，然后递归分割窗口，直到窗口内目标足够简单，可以显示为止

一、什么样的情况下，画面足够简单可以立即显示？

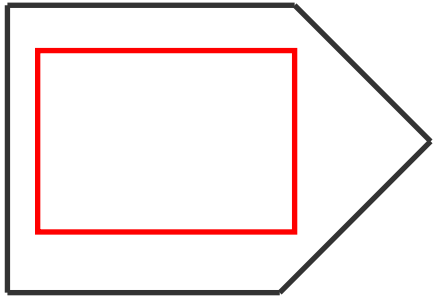
（1）窗口中仅包含一个多边形



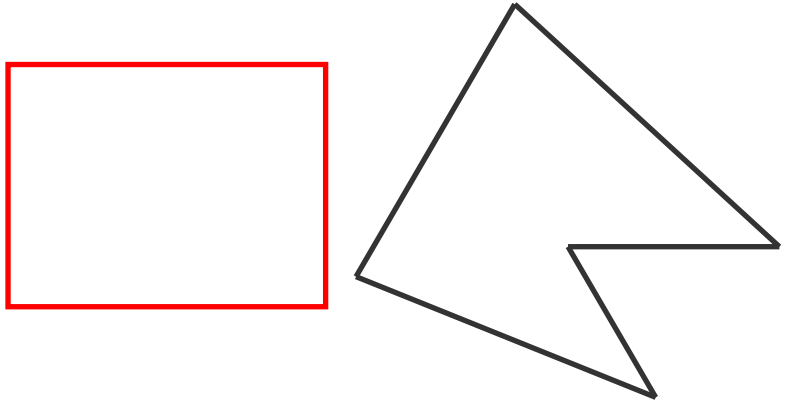
（2）窗口与一个多边形相交，且
窗口内无其它多边形



（3）窗口为一个多边形所包围



（4）窗口与一个多边形相分离

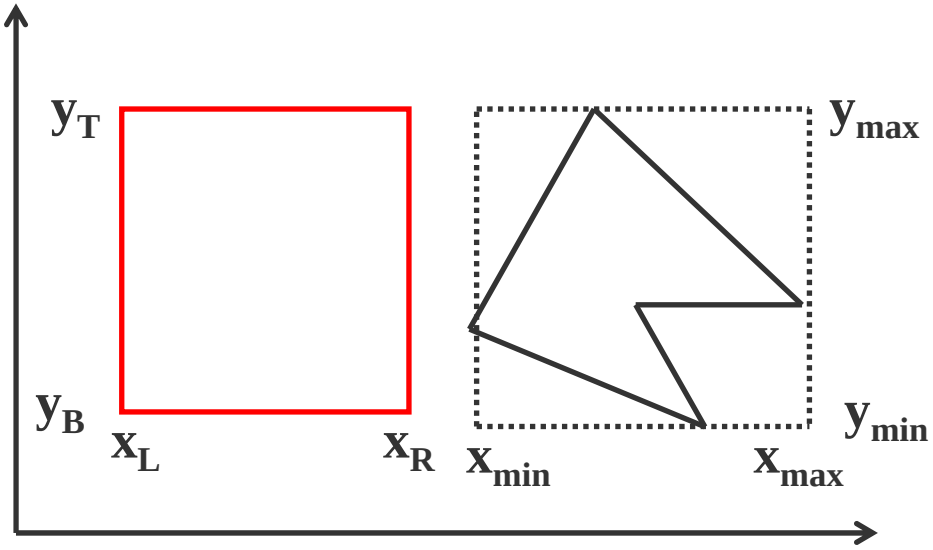


如何判别一个多边形和窗口是分离的？

当满足下列条件时，多边形和窗口分离：

$$x_{\min} > x_R \quad \text{or} \quad x_{\max} < x_L$$

$$y_{\min} > y_T \quad \text{or} \quad y_{\max} < y_B$$

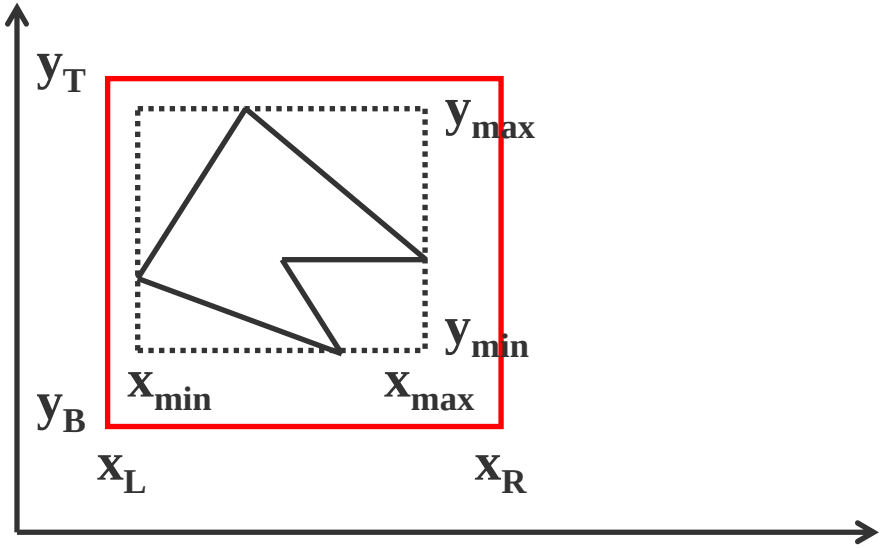


如何判别一个多边形在窗口内？

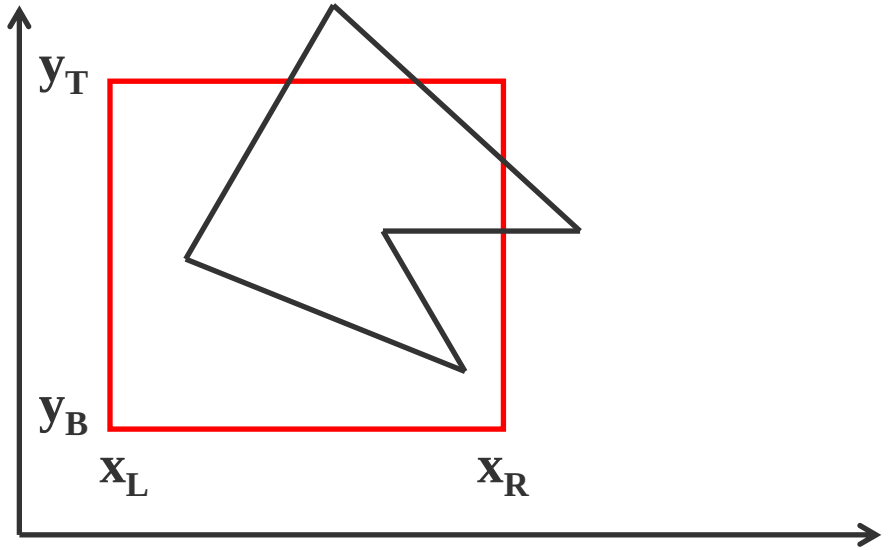
当满足下列条件时，多边形被窗口包含：

$$x_{\min} \geq x_L \quad \& \quad x_{\max} \leq x_R$$

$$y_{\min} \geq y_B \quad \& \quad y_{\max} \leq y_T$$



多边形与窗口相交的判别，可以采用直线方程作为判别函数来判定一个多边形是否与窗口相交



二、窗口有多个多边形投影面，如何显示？

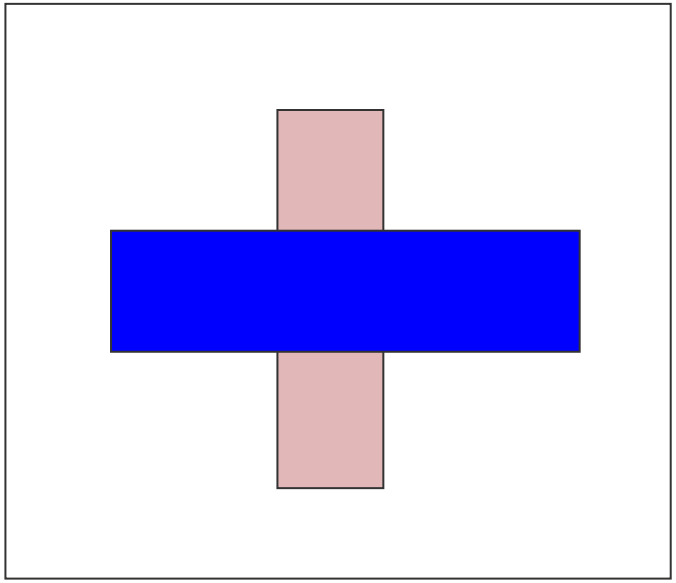
Warnock算法的重要性不在于它的效率比别的算法高，而在于采用了分而治之的思想，利用了堆栈的数据结构

把物体投影到全屏幕窗口上，然后递归分割窗口，直到窗口内目标足够简单，可以显示为止

算法步骤：

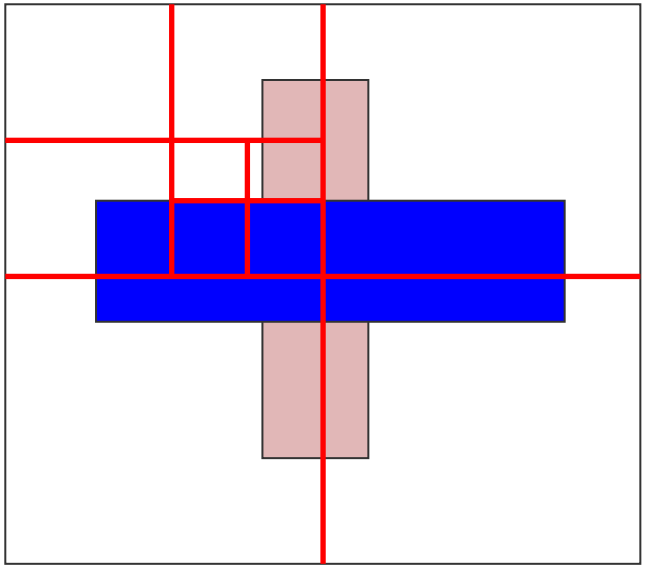
（1）如果窗口内没有物体则按背景色显示

（2）若窗口内只有一个面，则把该面显示出来



（3）否则，窗口内含有两个以上的面，则把窗口等分成四个子窗口。对每个小窗口再做上述同样的处理。这样反复地进行下去

（3）窗口内含有两个以上的面，则把窗口等分成四个子窗口。对每个小窗口再做上述同样的处理。这样反复地进行下去



把四个子窗口压在一个堆栈里（后进先出）。

假设显示器分辨率（1024*1024），窗口最多分几次？

如果到某个时刻，窗口仅有象素那么大，而窗口内仍有两个以上的面，如何处理？

这时不必再分割，只要取窗口内最近的可见面的颜色或所有可见面的平均颜色作为该象素的值

光栅扫描算法小结

1、直线段的扫描转换算法

这是一维图形的显示基础

（1）DDA算法主要利用了直线的斜截式方程（y=kx+b），在这个算法里引进了增量的思想，结果把一个乘法和加法变成一个加法

（2）中点法是采用的直线的一般式方程，也采用了增量的思想，比DDA算法的优点是采用了整数加法

（3）Bresenham算法也采用了增量和整数算法，优点是这个算法还能用于其它二次曲线

2、多边形的扫描转换和区域填充

如何把边界表示的多边形转换成由像素逐点描述的多边形
这是二维图形显示的基础

有四个步骤：求交、排序、配对、填色。这里引进了一个新的思想—图形的连贯性。手段就是利用增量算法和特殊的数据结构（多边形y表、边y表、活化多边形表、活化边表），2个指针数组和2个指针链表

3、直线和多边形裁剪

关于裁剪算法主要讲了两个经典算法：

cohen-Sutherland算法、梁-barsky算法

cohen-Sutherland算法的核心思想是编码。把屏幕（窗口，场景空间）分成9个部分，用4位编码来描述这9个区域，通过4位编码的“与”、“或”运算来判断直线段是否在窗口内或外

梁算法主要的思想：

（1）直线方程用参数方程表示

（2）把被裁剪的直线段看成是一条有方向的线段，把窗口的四条边分成两类：入边和出边

这也是中国人的算法第一次出现在了所有图形学教科书都必须提的一个算法。这就叫原始创新！

4、走样、反走样

因为用离散量表示连续量，有限的表示无限的自然会导致一些失真，这种现象称为走样

反走样主要有三种方法：

提高分辨率

区域采样

加权区域采样。

提高分辨率无非是把分辨率增加，这样可以提高反走样的效果；但这个方法是有物理上的限制的—分辨率不能无限增加

区域采样算法是在关键的直线段、关键的区域上绘制的时候并非非黑即白，可以把关键部位变得模糊一点，有颜色的过渡区域，这样会产生一种好的视觉效果

加权区域是不但要考虑区域采样，而且要考虑不同区域的权重，用积分、滤波等技巧来做

5、消隐

在绘制场景时消除被遮挡的不可见的线或面，称作消除隐藏线和隐藏面，简称为**消隐**

消隐算法按消隐空间分类：

（1）物体空间　以场景中的物体为处理单元

（2）图像空间　以屏幕窗口内的每个像素为处理单元

首先介绍了经典的z-buffer算法。为了避免一个z-buffer二维数组的开销，引进了单个变量的z-buffer算法，把数组变成了单个变量

单个变量的z-buffer算法的主要问题是要不断地判一个点是否在多边形内。为了解决这个问题，又讲了3个算法：即如何判点在多边形内部，有射线法、代数弧长累加法、以顶点符号为基础的弧长累加方法。

区间扫描线算法：发现扫描线和多边形的交点把扫描线分成若干区间，每个区间只有一个多边形可以显示。利用这个特点可以把逐点处理变成逐段处理，提高了算法效率

Warnock消隐算法：采用了分而治之的思想，利用了堆栈的数据结构

把物体投影到全屏幕窗口上，然后递归分割窗口，直到窗口内目标足够简单，可以显示为止

核心思想

（1）增量思想：通过增量算法可以减少计算量

（2）编码思想

（3）符号判别—> 整数算法 尽可能的提高底层算法的效率，底层上提高效率才是真正解决问题

核心思想

（4）图形连贯性：利用连贯性可以大大减少计算量

（5）分而治之：把一个复杂对象进行分块，分到足够简单再进行处理