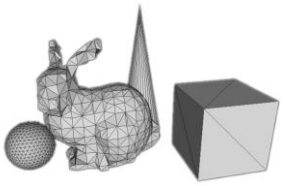


Introduction to Computer Graphics

Implement Your Pipeline 2

Ming Zeng

Software School

Xiamen University

Contact Information:

zengming@xmu.edu.cn

第七章 流水线实现
——从顶点到片段

- 主要内容
  - 框架与裁剪
    - 基本实现策略
    - 线段裁剪
    - 多边形裁剪
  - 光栅化
  - 隐藏面消除

第二部分 光栅化

- 主要内容
  - 线段的扫描转换算法
    - DDA算法
    - Bresenham算法
  - 多边形的填充算法
    - 种子填充算法
    - 扫描线填充算法

光栅化(Rasterization)

- 光栅化也称为扫描转换(scan conversion)
  - 确定哪些像素在由顶点表示的图元内部
  - 生成片段集合
  - 片段有位置值（像素位置）和由顶点属性值插值得到的颜色、纹理坐标和深度等其他属性
- 最终像素的颜色由片段颜色、纹理和其他顶点属性确定

一些假设

- 颜色缓冲区是一个n x m像素阵列，(0,0)对应于左下角
- 像素可由图形系统内部函数赋颜色值：

  write_pixel(int ix, int iy, int value);
- 颜色缓冲区是离散的，只讨论位于整数ix和iy位置上的像素
  - 如果片段位置为(63.4,157.9)，取(63,158)或(63.5,157.5)，取决于像素中心位于整数值还是半整数值
  - 像素显示为正方形，中心在像素位置，边长为两个相邻像素间的距离
  - OpenGL的像素中心位于半整数值位置

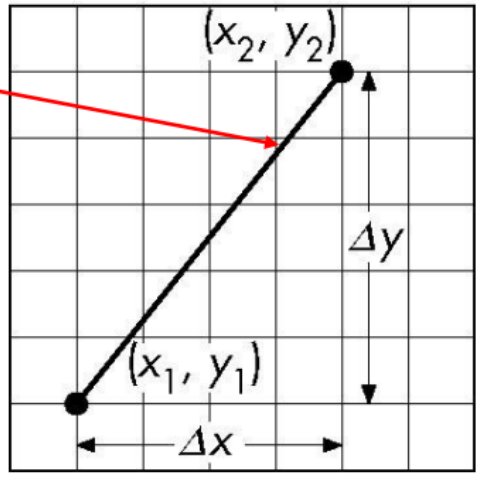
直线段的扫描转换

- 假定线段的端点为(x1,y1)和(x2,y2)，取整数值的窗口坐标
- 线段的斜率为 $m = \dfrac{y_2 - y_1}{x_2 - x_1} = \dfrac{\Delta y}{\Delta x}$

平凡算法：

```
for(x = x1; x <= x2; x++)
{
    y = m * x + h;
    write_pixel(x, round(y),
                line_color);
}
```

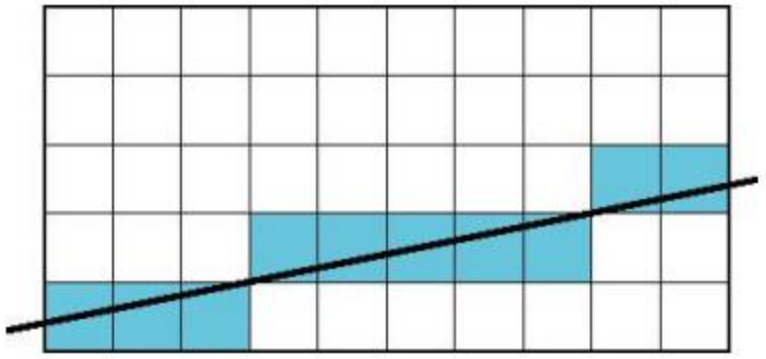
y = mx + h



DDA算法

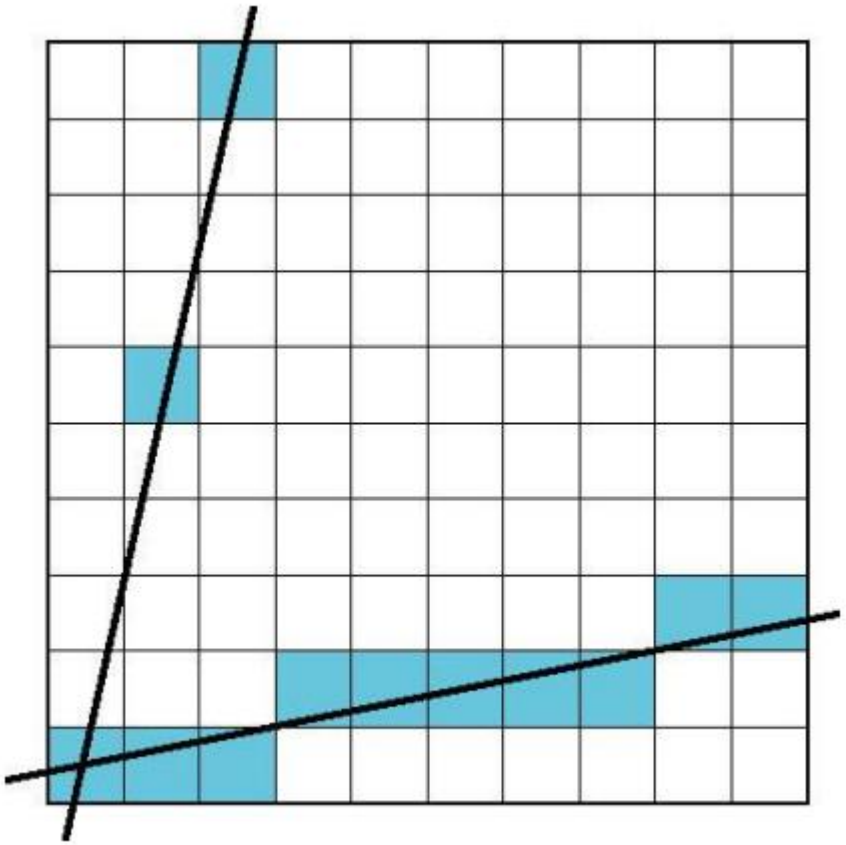
- DDA: Digital Differential Analyzer 数字微分分析器
  - DDA是早期用于微分方程数值仿真的机电设备
  - 直线 y = mx + h满足微分方程
    $dy/dx = m = \Delta y/\Delta x = (y_2-y_1) / (x_2-x_1)$
  - 沿扫描线有dx = 1, dy = m

```
for(x = x1; x <= x2; x++)
{
    y += m;
    write_pixel(x, round(y),
    line_color);
}
```



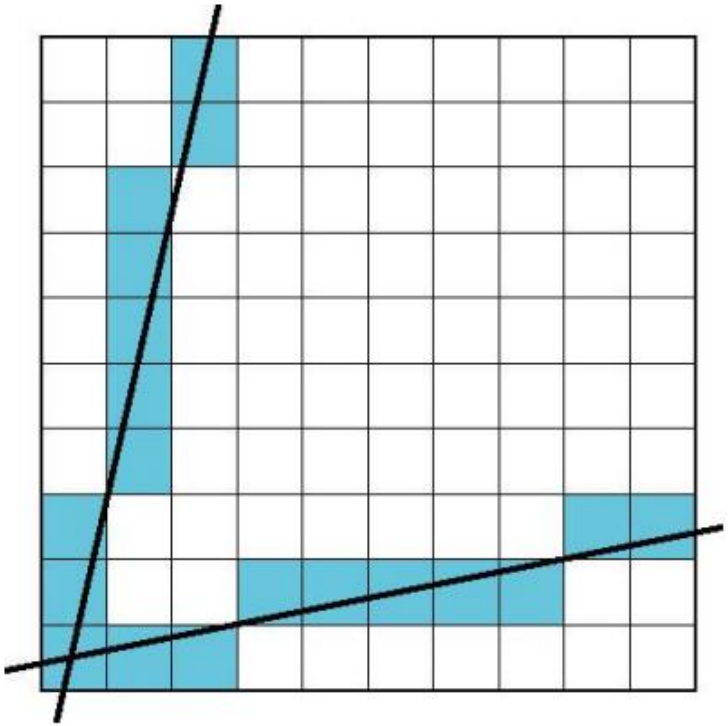
问题

- DDA = 对于每个x画出最接近的整数y
  - 对于斜率大的直线有问题



利用对称性

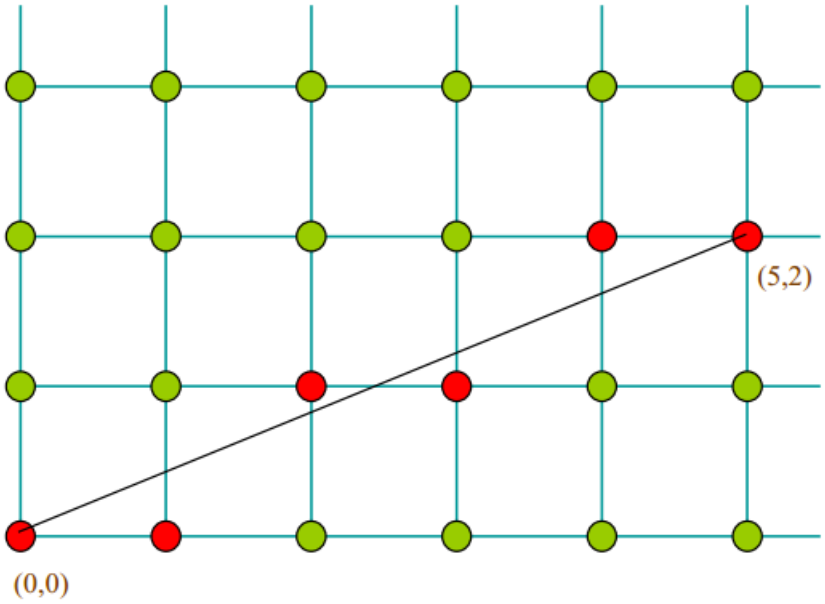
- 只对$0 \le |m| \le 1$的直线应用上述算法
- 对于$|m|>1$的直线，交换x与y的角色
  - 对于每个y，找出最接近的整数x



例子

- 例： 画直线段(0,0) →(5,2)
  dx=5, dy=2, 斜率m=0.4

x	y	int(y+0.5)
0	0	0
1	0.4	0
2	0.8	1
3	1.2	1
4	1.6	2
5	2.0	2

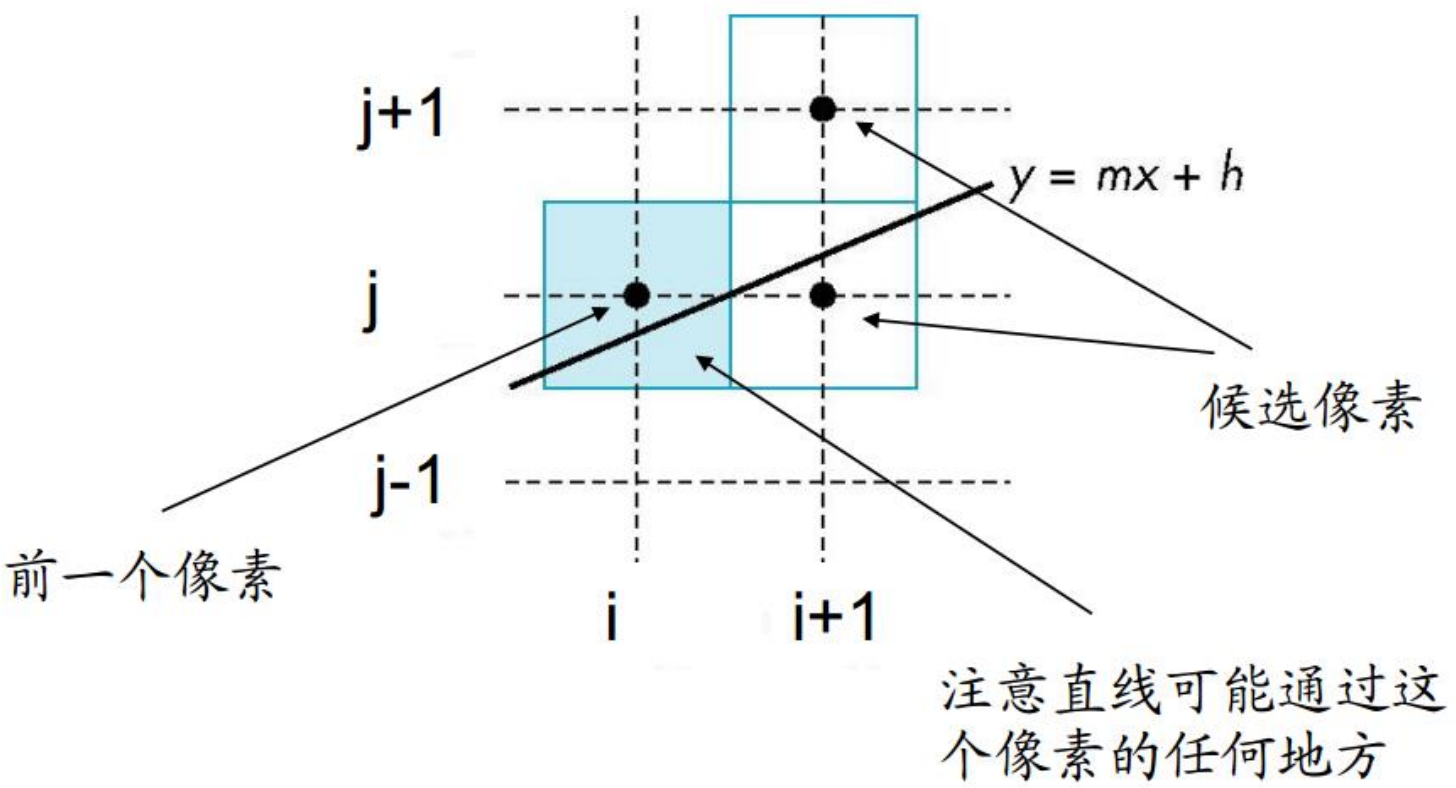


Bresenham算法

- DDA算法中每一步需要一次浮点加法
- Bresenham算法中可以不出现任何浮点运算，是硬件和软件光栅化器的标准算法
- 只考虑$0 \leq m \leq 1$的情形
  - 其它情形利用对称性处理
- 如果从一个已被确定激活的像素出发，那么下一像素的可能位置只会有两种可能

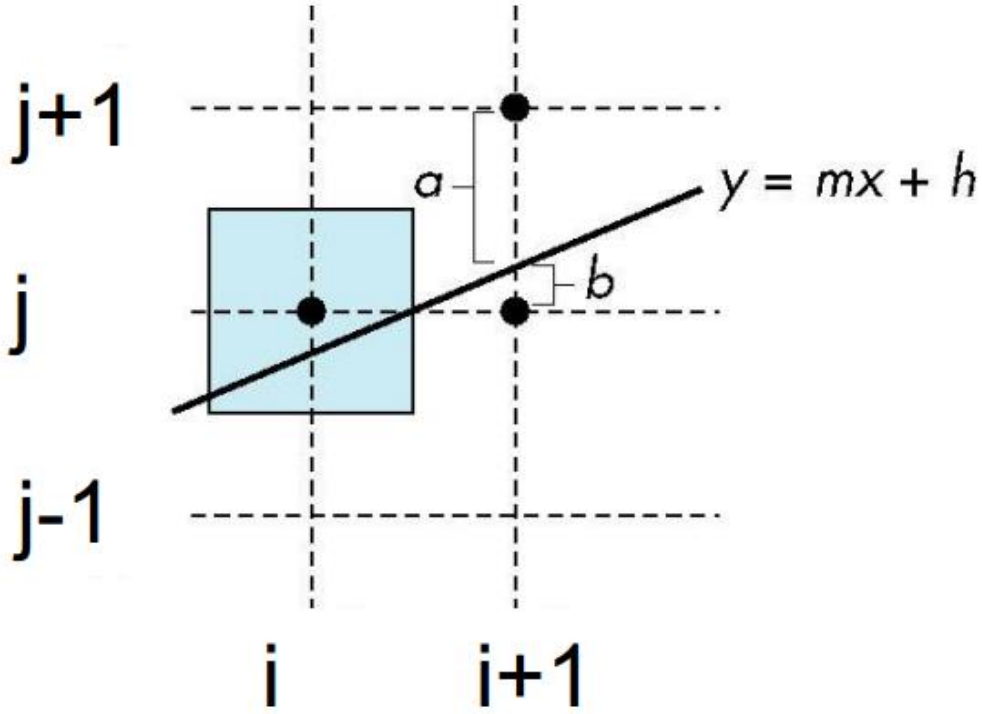
候选像素

- 线段端点在整数值(x_1,y_1)和(x_2,y_2)，斜率$0 \le m \le 1$，$m=\Delta y/\Delta x$，$\Delta y=y_2-y_1$，$\Delta x=x_2-x_1$



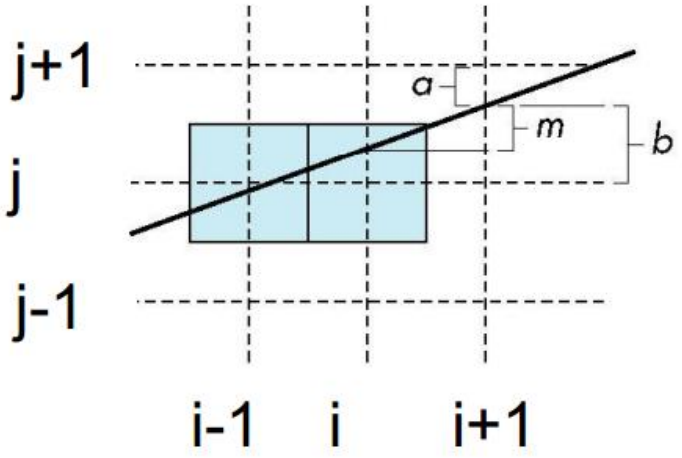
决策变量

- d = (x2-x1) (a-b) = Δx (a-b)， d为整数
  d > 0，采用下像素(i+1,j)
  d ≤ 0，采用上像素(i+1,j+1)

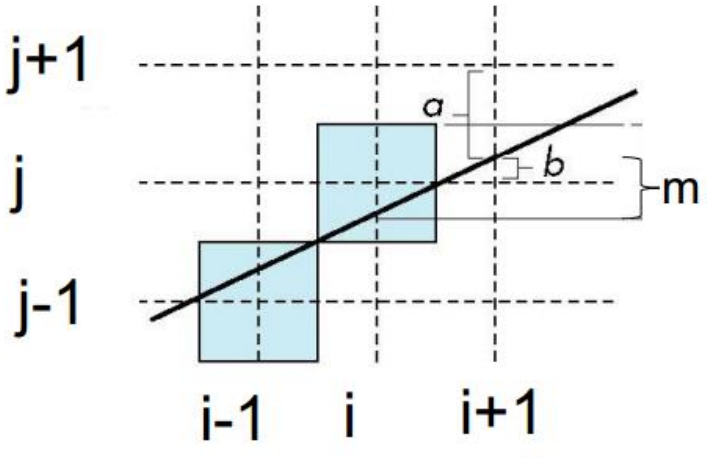


增量形式

- x=k处的决策变量为$d_k = \Delta x\,(a_k - b_k)$，则
  $d_{k+1} = d_k - 2\,\Delta y$，如果 $d_k > 0$ (左图)
  $d_{k+1} = d_k - 2(\Delta y - \Delta x)$，其他 （右图）



$a_{i+1}=a_i-m,\ b_{i+1}=b_i+m$



$a_{i+1}=a_i+1-m,\ b_{i+1}=b_i+m-1$

决策变量d的初值

- 在$x=x_1$处， $a=1, b=0, d = \Delta x > 0$
- 在$x=x_1+1$处， $a=1-m, b=m, d = \Delta x-2\Delta y$
- 算法优点：
  - 对每个x值，只需要进行整数加法以及符号判断
  - 可以在图形芯片上用单个指令实现

多边形的扫描转换

- 对于多边形，扫描转换就是填充
- 早期的光栅系统
  - 可以显示被填充的多边形
  - 无法实时给多边形内部每个点着以不同颜色
- 直线的光栅化算法就是Bresenham算法，而多边形的填充算法有许多种
  - 具体选择与系统的实现框架有关

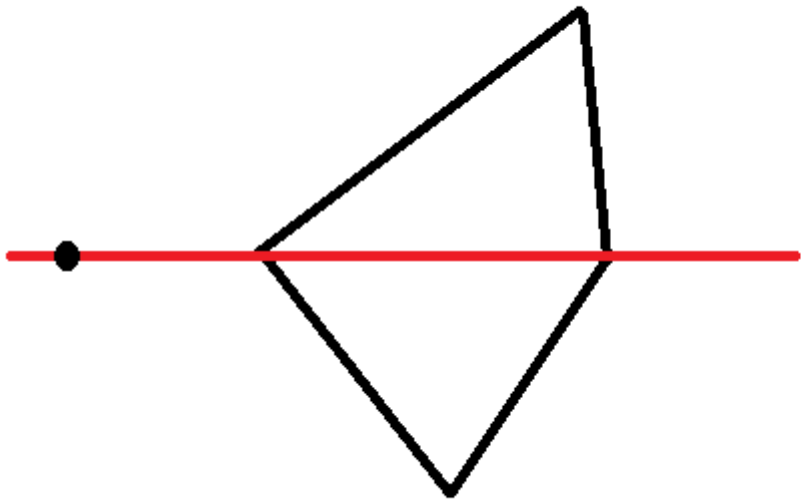
内外检测

- 对多边形内部区域的填充过程等价于判断多边形所在平面上哪些点是内部点
  - 多边形填充是一个分类问题
- 如何区分内部与外部？
  - 对于非平面多边形，无法定义它的内部区域
  - 对于凸多边形，很容易做到 （如何做到）
  - 对于非简单多边形，就非常困难
  - 可以采用奇偶检测的方法
    - 统计与多边形边的交点数
  - 环绕数 (winding number)

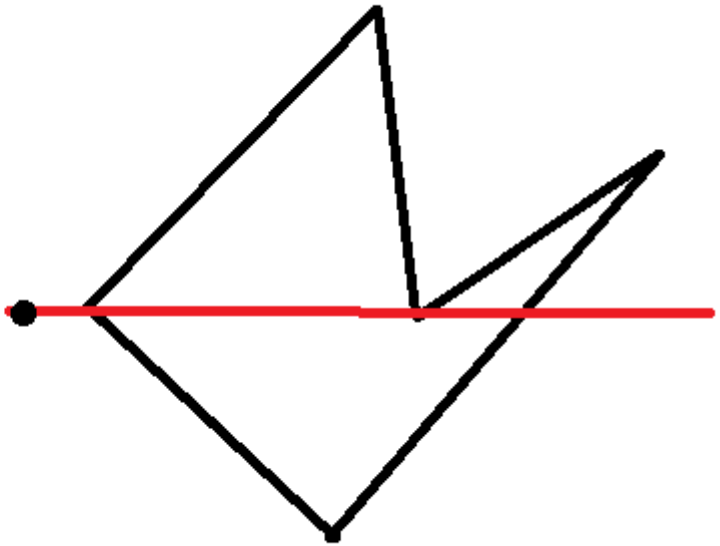
奇偶检测

- 奇偶检测(odd-even test), 也称为射线法
  - 判断多边形内-外区域的最常用方法
- 从一点p引射线，如果与多边形的交点数为偶数，则p在多边形外；否则p在多边形内
- 如果交点为顶点，需要特别处理
- 通常使用扫描线代替射线

奇偶检测



与4条边相交



与5条边相交

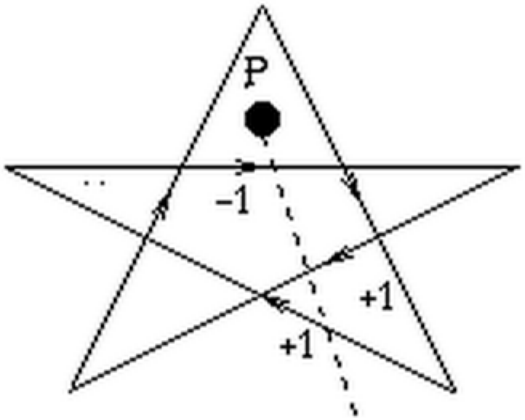
实际上，如果交到顶点，应该根据实际情况计算

1) 顶点两侧的直线分居扫描线两侧，只算1次

2）顶点两侧的直线在扫描线同侧，算2次

环绕数

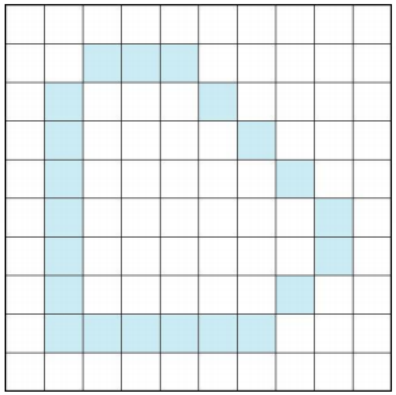
- 判断点p是否在多边形内，从点p向外做一条射线（可以任意方向），多边形的边从左到右经过射线时环数减1，多边形的边从右往左经过射线时环数加1，最后环数不为0，即表示在多边形内部。



种子填充算法

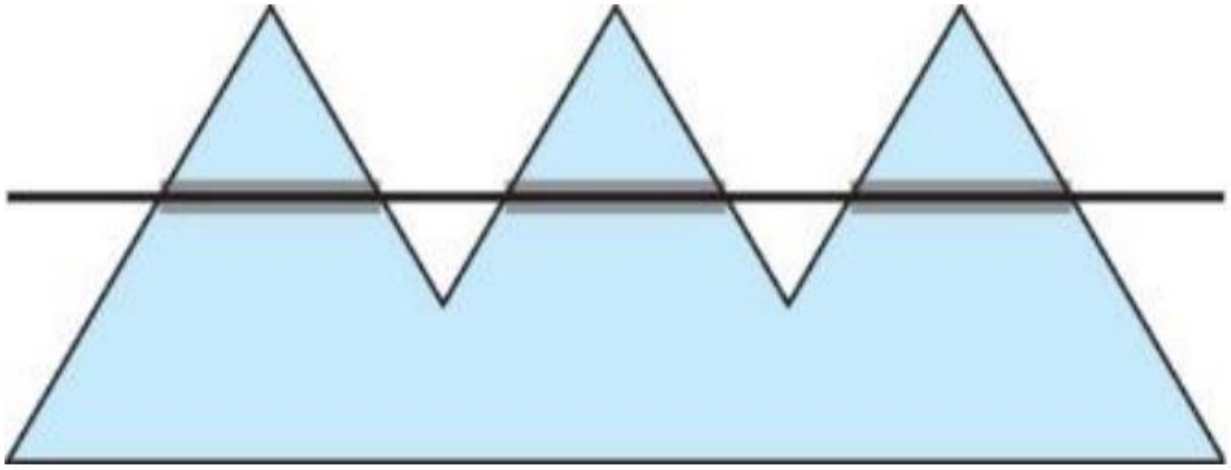
- 用Bresenham算法将多边形的边扫描转换为帧缓冲区中像素，颜色置成边/内部填充颜色(BLACK)
- 如果已知位于多边形内部的一个种子点(WHITE)，那么可以递归填充

```
flood_fill(int x, int y)
{
    if(read_pixel(x,y) == WHITE)
    {
        write_pixel(x,y,BLACK);
        flood_fill(x-1, y);
        flood_fill(x+1, y);
        flood_fill(x, y+1);
        flood_fill(x, y-1);
    }
}
```



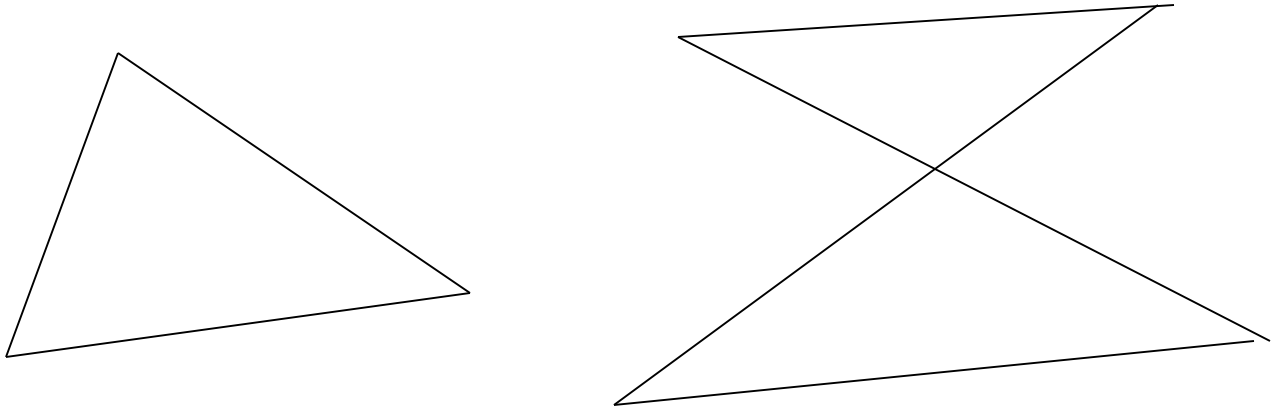
扫描线填充算法

- 根据奇偶检测规则，可知扫描线上有3个填充区间位于多边形的内部
  - 填充区间由扫描线与多边形的交点集合确定
  - 沿扫描线，逐区间填充
  - 每个填充区间可以独立地进行光照或深度计算



扫描线填充算法

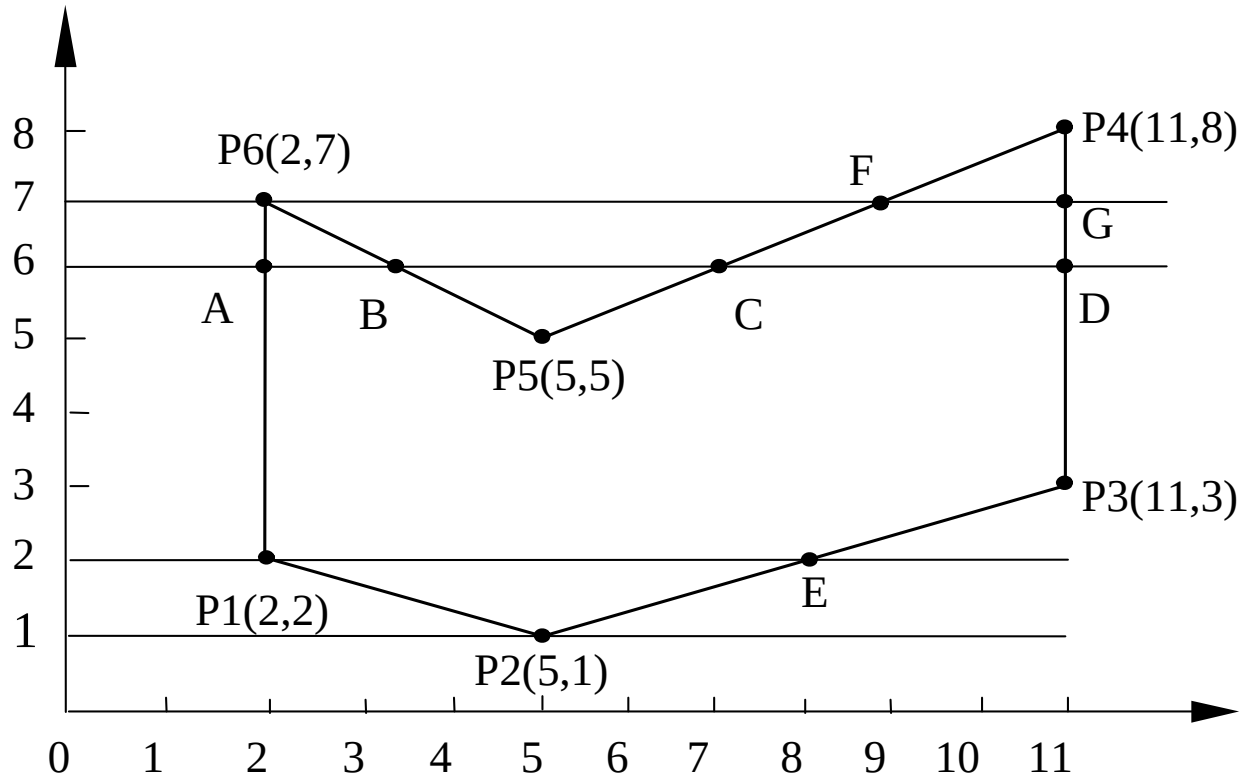
- 目标：利用相邻像素之间的连贯性，提高算法效率
- 处理对象：非自交多边形（边与边之间除了顶点外无其它交点）



基本思想

- 基本思想：
  - 按扫描线顺序，计算扫描线与多边形的相交区间，再用要求的颜色显示这些区间的象素，即完成填充工作。
- 对于一条扫描线填充过程可以分为四个步骤：
  1. 求交：计算扫描线与多边形各边的交点；
  2. 排序：把所有交点按x值递增顺序排序；
  3. 配对：第一个与第二个，第三个与第四个等等；每对交点代表扫描线与多边形的一个相交区间，
  4. 着色：把相交区间内的象素置成多边形颜色，把相交区间外的象素置成背景色。

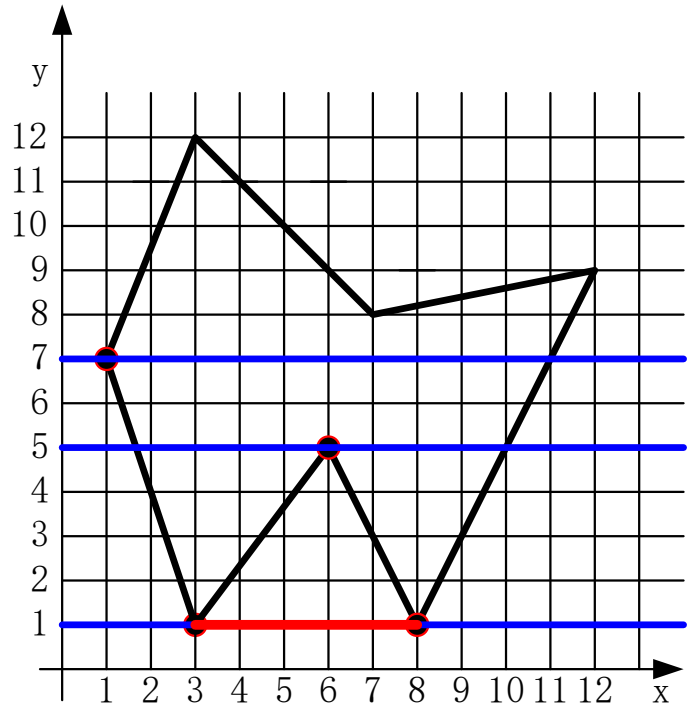
一个多边形与若干条扫描线



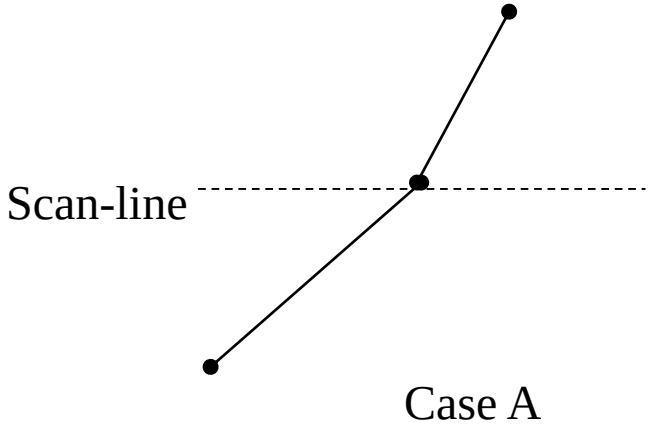
1. 求交
2. 排序
3. 配对
4. 着色

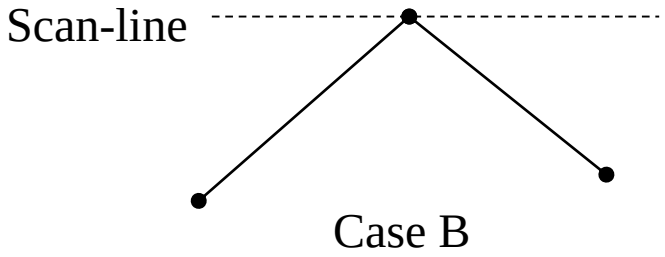
存在问题

- 当扫描线与多边形顶点相交时，交点的取舍问题。



与多边形顶点相交的交点的处理





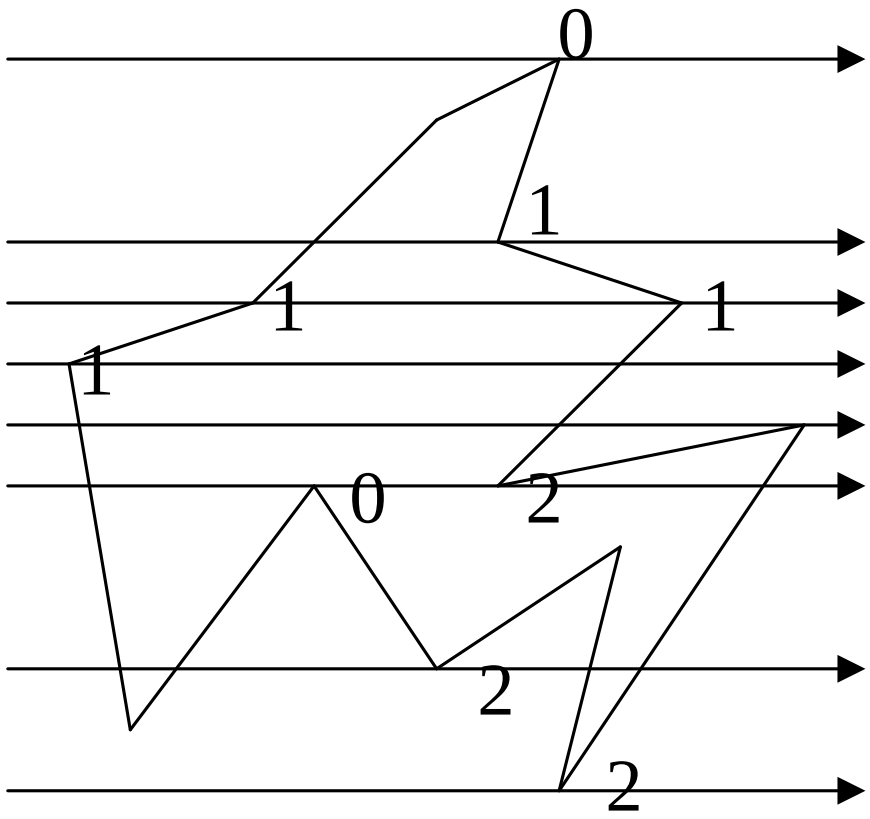
解决方法

解决方法：

当扫描线与多边形的顶点相交时，

- 若共享顶点的两条边分别落在扫描线的两边，交点只算一；
- 若共享顶点的两条边在扫描线的同一边，这时交点作为零个或两个。
- 具体实现时，只需检查顶点的两条边的另外两个端点的y值。按这两个y值中大于交点y值的个数是0, 1, 2来决定。

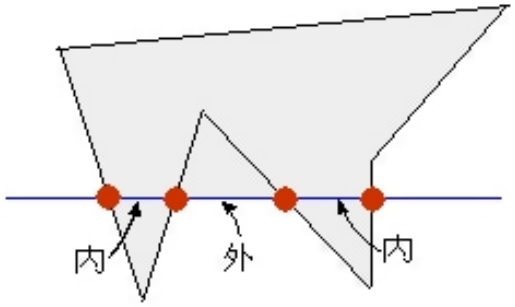
解决方法



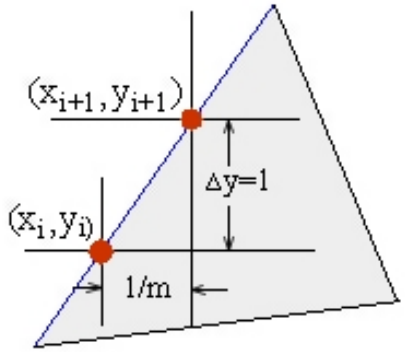
与扫描线相交的多边形顶点的交点数

利用相关性改进算法：边相关扫描线算法

- 扫描线的相关性：某条扫描线上相邻的象素，几乎都具有同样的内外性质，这种性质只有遇到多边形边线与该扫描线的交点时才会发生改变。
- 边的相关性：由于相邻扫描线上的交点是与多边形的边线相关的。对同一条边，前一条扫描线yi与该边的交点为xi，而后一条扫描线 $y_{i+1}=y_i+1$ 与该边的交点则为 $X_{i+1}=X_i+\frac{1}{m}$ ，利用这种相关性可以省去大量的求交运算。



(a) 扫描线的相关性



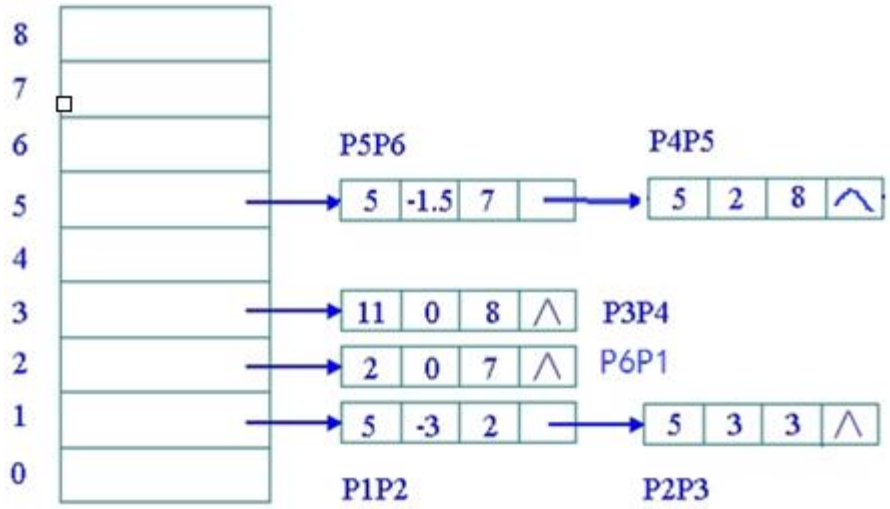
(b) 边的相关性

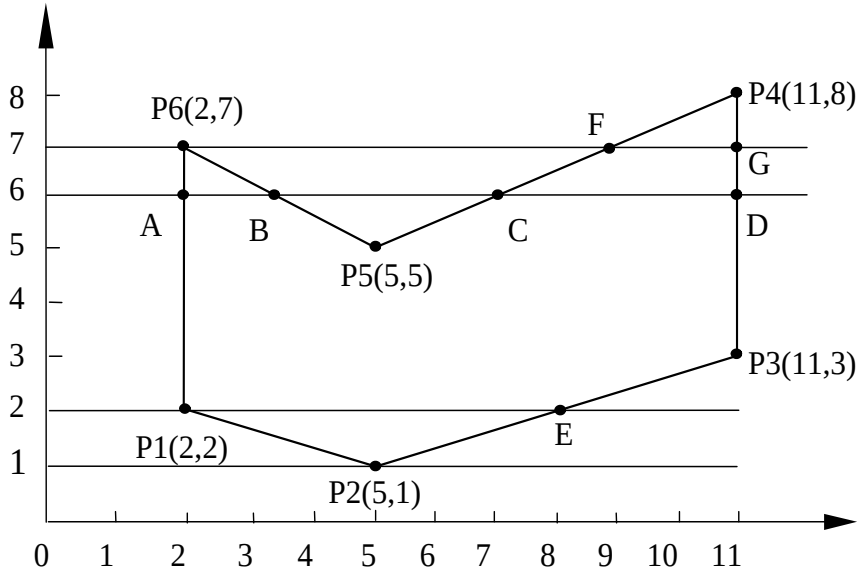
新边表和活动边表

- 边相关扫描线填充算法的实现需要建立两个表：
  新边表（NET）和活动边表（AET）

1 新边表（New Edge Table)

用来对除水平边外的所有边进行登记，来建立边的记录。边的记录定义为：





x： 当前扫描线与边的交点坐标

△x： 从当前扫描线到下一条扫描线间x的增量

ymax： 该边所交的最高扫描线号

x	△x	ymax	

新边表构造

(1) 首先构造一个纵向扫描线链表，链表的长度为多边形所占有的最大扫描线数，链表的每个结点称为一个桶，对应多边形覆盖的每一条扫描线。

(2) 将每条边的信息链入与该边最小y坐标（ymin）相对应的桶处。也就是说，若某边的较低端点为ymin，则该边就放在相应的扫描线桶中。

新边表构造

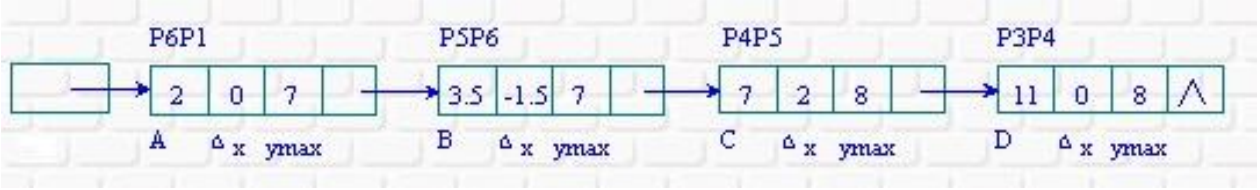
(3) 每条边的数据形成一个结点，内容包括：该扫描线与该边的初始交点x（即较低端点的x值），1/k，以及该边的最大y值y_{max}。

(4) 同一桶中若干条边按$X|_{ymin}$由小到大排序，若$X|_{ymin}$相等，则按照1/k由小到大排序。

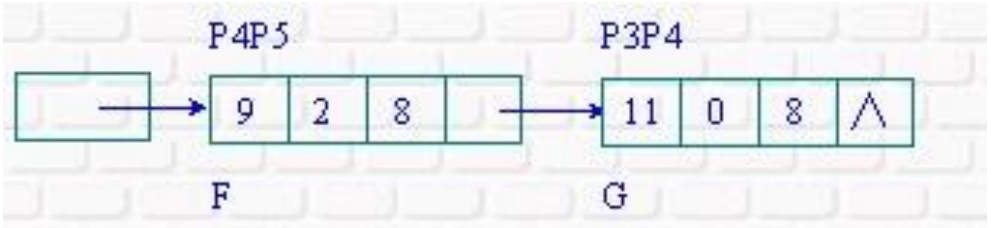
活动边表

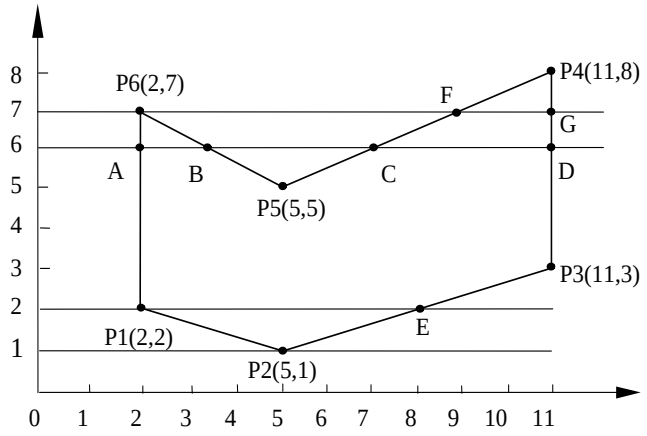
- NET表建立以后，就可以开始扫描转换了。对不同的扫描线，与之相交的边线也是不同的，当对某一条扫描线进行扫描转换时，我们只需要考虑与它相交的那些边线，为此需要建立一个只与当前扫描线相交的边记录链表，称之为活动边表。

(a) 扫描线6的活动边表：



(b) 扫描线7的活动边表：





相邻扫描线的相关性

设该边的直线方程为：$ax+by+c=0$，当前扫描线及下一条扫描线与边的交点分别为(x_i, y_i)，(x_{i+1}, y_{i+1}) 则：

$$aX_i + by_i + c = 0$$

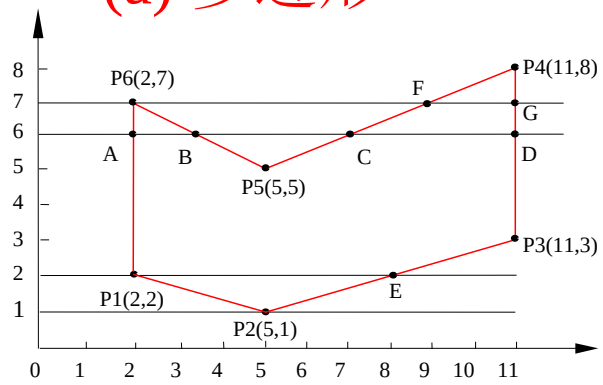
$$aX_{i+1} + by_{i+1} + c = 0$$

由此可得：$x_{i+1} = \frac{1}{a}(-b \cdot y_{i+1} - c_i) = x_i - \frac{b}{a} = \frac{1}{m}$

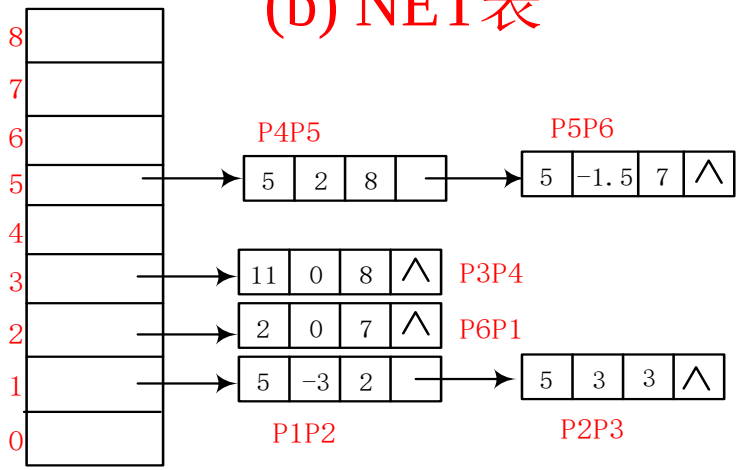
- 其中 $y_{i+1} = y_i + 1$，$\Delta x = -b/a = \frac{1}{m}$ 为常数。

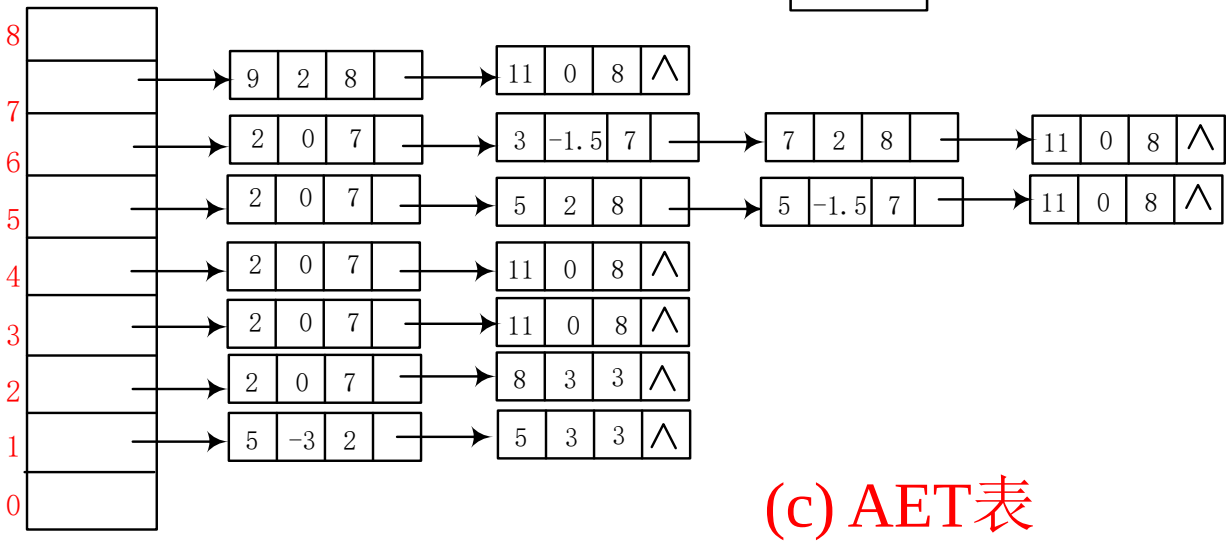
如何由NET->AET

(a) 多边形



(b) NET表





(c) AET表

算法流程

1、根据给出的多边形顶点坐标，建立NET表；

求出顶点坐标中最大y值ymax和最小y值ymin。

2、初始化AET表指针，使它为空。

3、执行下列步骤直至NET和AET都为空．从y=ymin开始扫描

3.1、如NET中第y类非空，则将其中的所有边取出并插入AET中；

3.2、如果有新边插入AET，则对AET中各边排序；

3.3、对AET中的边两两配对，（1和2为一对，3和4为一对，…），将每对边中x坐标按规则取整，获得有效的填充区段，再填充.

3.4、将当前扫描线纵坐标y值递值1；

3.5、如果AET表中某记录的ymax=yj，则删除该记录（因为每条边被看作下闭上开的）；

3.6、对AET中剩下的每一条边的x递增1/k，即x = x+ 1/k .

第三部分 隐藏面消除

- 主要内容
  - 对象空间方法
    - 画家算法
    - 背向面剔除
  - 图像空间方法
    - Z缓冲区算法

隐藏面消除

- 尽管光栅化生成的每个片段都对应颜色缓冲区中的某个位置，但不希望显示来自于被其他不透明对象所遮挡的对象上的片段

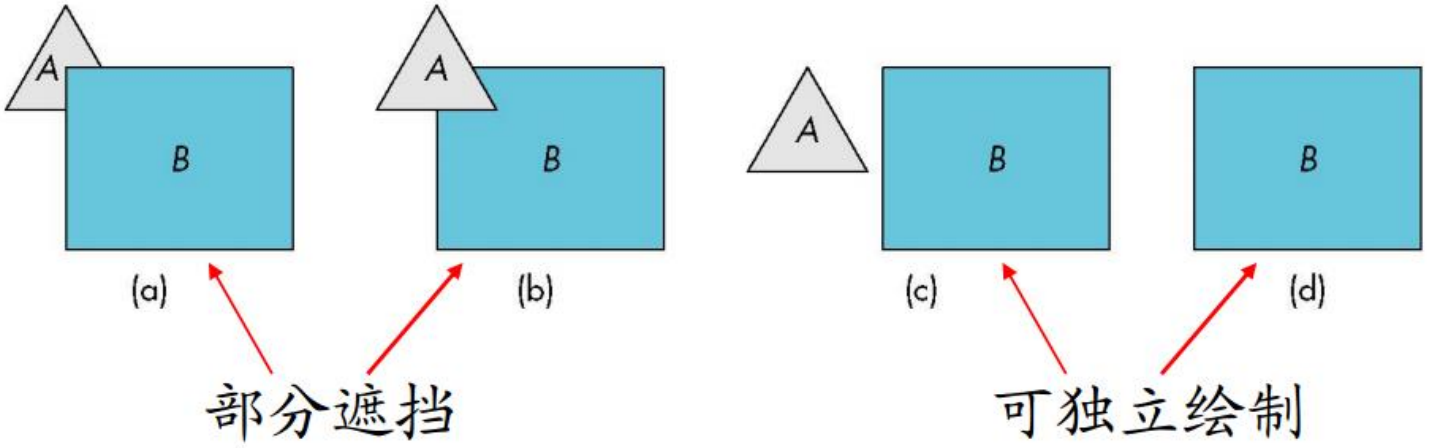
- 隐藏面消除（或可见面确定）用来确定视景体内每个对象的哪些部分是可见的，或者在视线上被其他对象所遮挡
  - 对象空间算法
  - 图像空间算法

裁剪与可见性

- 裁剪和隐藏面消除有许多共同点
- 实际上，对于裁剪与隐藏面消除，都是希望把看不到的对象从视野中去掉
- 通过在处理过程中提早应用可见性或者遮挡检测，使得在进入流水线体系之前消除尽可能多的多边形

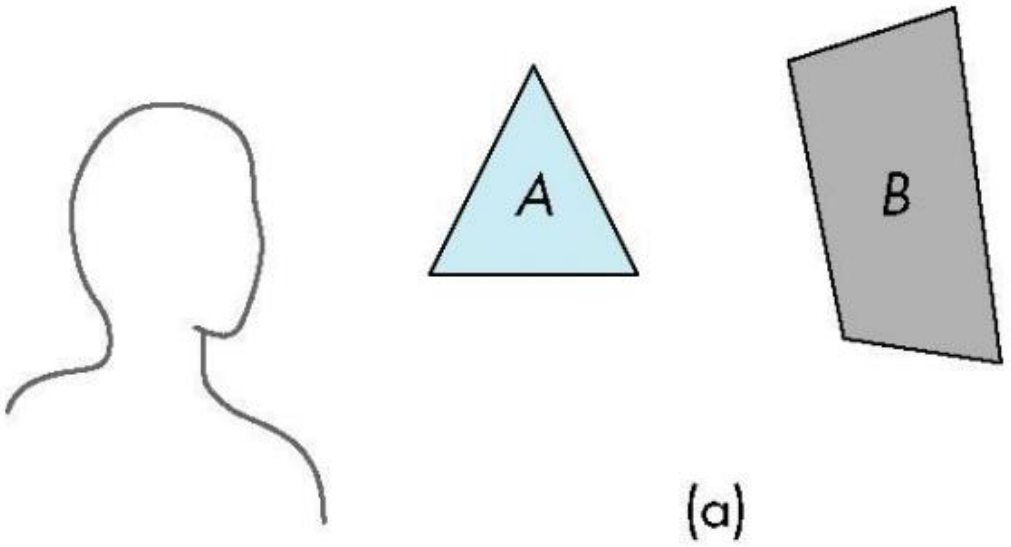
对象空间算法

- 考虑由k个三维不透明多边形构成的场景
- 每个多边形认为是单独的一个对象
- 两两比较，检测相互之间的位置



画家算法

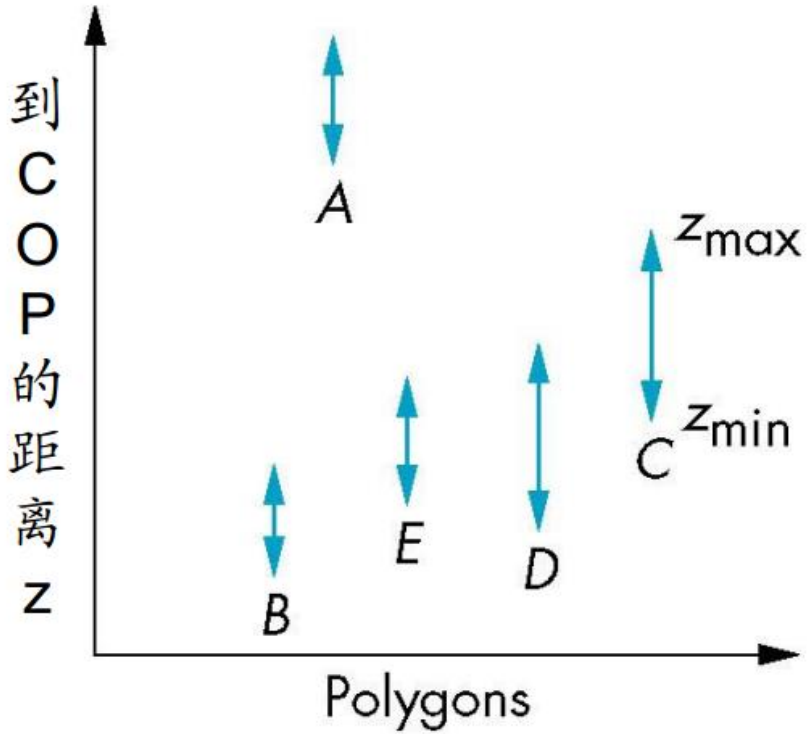
- 假设多边形已按离视点的远近排序，按从后到前的顺序把每个多边形完全绘制出来



从观察者的角度来说，B在A后面　　先画B，再画A

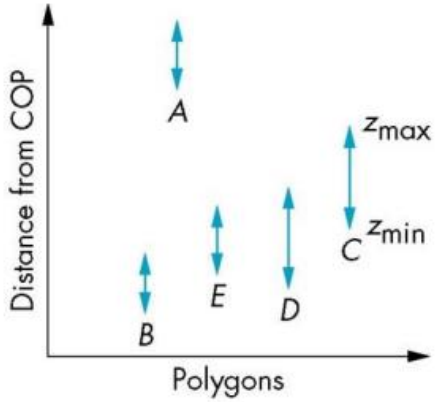
深度排序

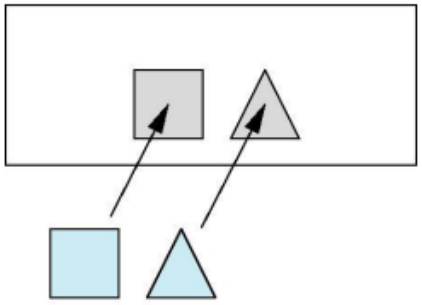
- 根据离视点的z值对多边形进行排序
  - 时间复杂度至少为O(k log k)
  - 并不是所有的多边形都完全在其它多边形的前面或后面
- 先处理简单情形，再处理困难情形

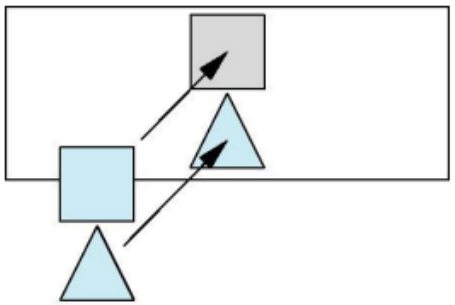


简单情形

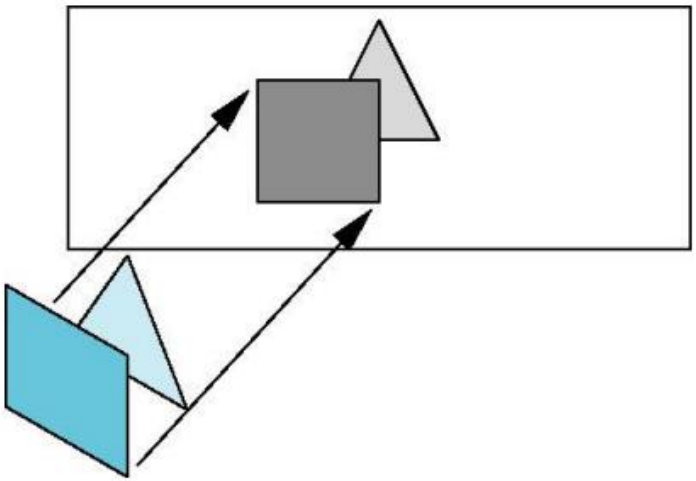
- 多边形A位于所有其它多边形后面
  - 可以绘制出来
- 多边形在z方向有重叠，但在x或y方向没有重叠
  - 可以分别绘制出来



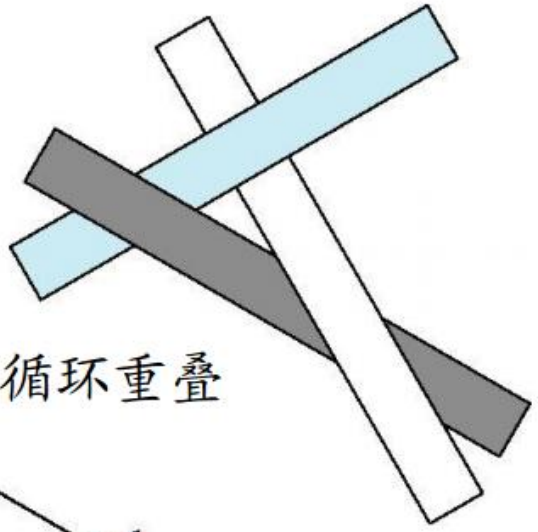




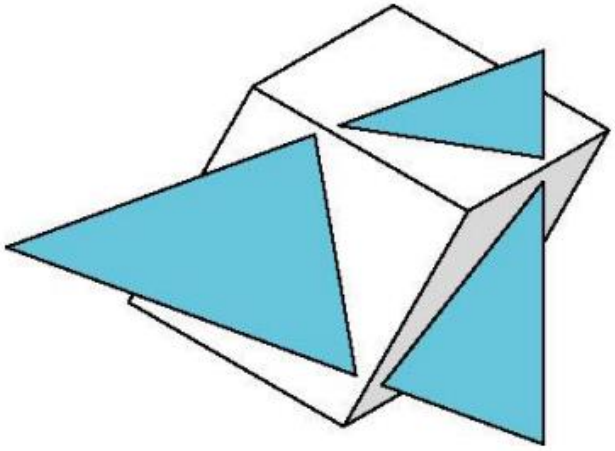
困难情形



在所有方向都有重叠，但其中一个完全在另一个的一侧



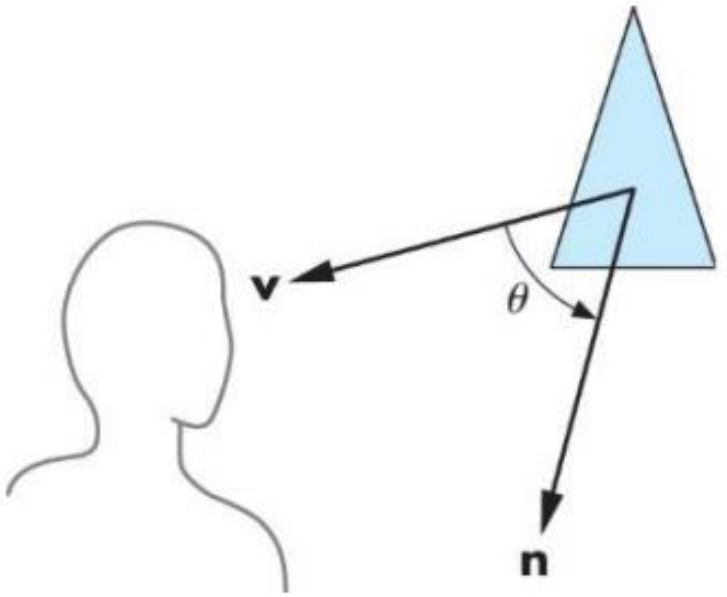
循环重叠



贯穿

背向面剔除

- 在OpenGL中，可以选择只绘制对象的正面多边形，从而可以在隐藏面消除之前把所有的背面多边形剔除掉面是可见的，如果
  $-90° \leq \theta \leq 90°$
  等价于
  $\cos\theta \geq 0$
  或者 $\mathbf{v} \cdot \mathbf{n} \geq 0$



背向面剔除

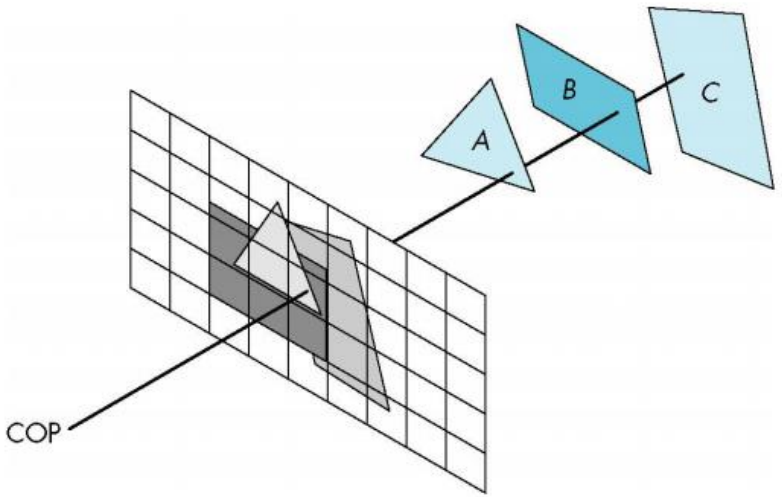
- 背向面剔除通常在多边形变换到规范化的设备坐标系之后进行
- 在规范化设备坐标系中，所有投影规范化为正交投影，投影方向沿z轴方向
  - 视线向量 $\mathbf{v} = [0, 0, 1]^T$
- 多边形所在平面的方程为

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

  - 只需检测c的符号

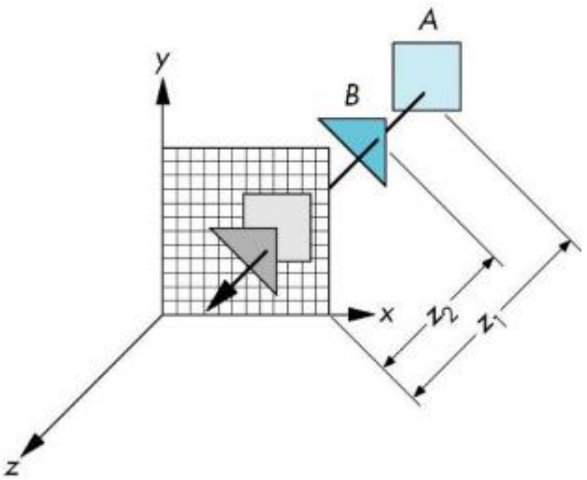
图像空间算法

- 对每条投影线（对于n x m分辨率的帧缓冲区，共有nm条投影线），找到与k个多边形的交点中离投影中心最近的那个交点
- 用多边形在该交点的颜色给对应像素着色
- 基本运算是直线与多边形的求交运算
  - 时间复杂度O(k)



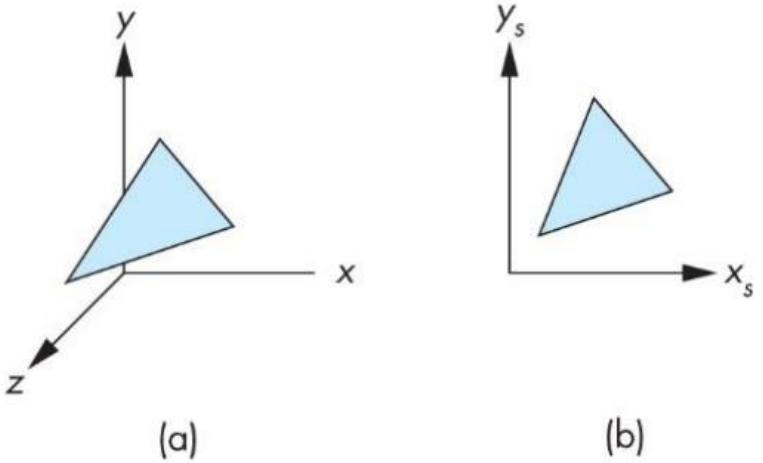
Z缓冲区算法

- z缓冲区算法是最常使用的隐藏面消除算法
  - 适合渲染流水线结构的图像空间消隐算法
- 流水线逐个多边形进行处理，光栅化给定多边形时并不知道其他多边形的信息，所以需要保留每个片段的深度信息
- z缓冲区或深度缓冲区存储，到目前为止在每个像素位置离视点最近的多边形的深度
  - 空间分辨率和颜色缓冲区相同
  - 深度分辨率通常是32位，以浮点数形式存储



基于Z缓冲区的扫描转换

- 在渲染流水线中，灵活使用z缓冲区算法可以同时完成3项任务：隐藏面消除、明暗处理和最终的正交投影变换
- 同一多边形的两种表示： (a) 三维规范化设备坐标系， (b) 二维屏幕坐标系
  - 用于在规范化坐标上比较深度



基于z缓冲区的扫描转换

- 策略：逐个多边形，逐条扫描线处理
- 沿扫描线每移动一个像素，增量式地计算对应点的深度值
- 判断屏幕坐标系中的像素是否对应多边形上的可见点
- 如果可见，从顶点明暗值插值得到像素的颜色值