计算机图形学
Computer Graphics

陈中贵
chenzhonggui@xmu.edu.cn
http://graphics.xmu.edu.cn/~zgchen

Graphics@XMU

第七章 第二节

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
- 走样/反走样

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：

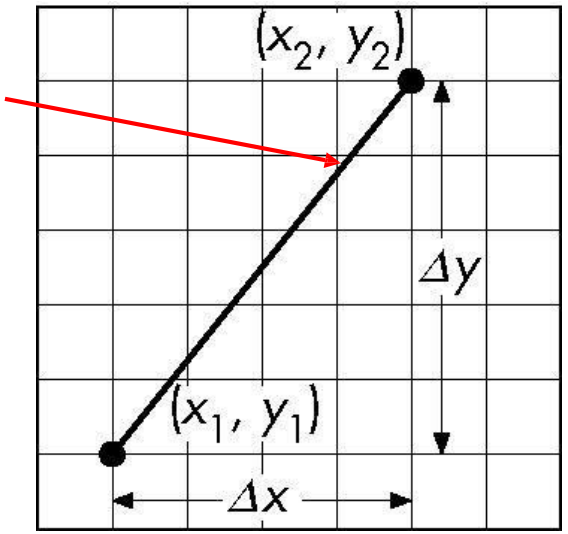
  write_pixel(int ix, int iy, int value);
- 颜色缓冲区是离散的，只讨论位于整数ix和iy位置上的像素
  - 如果片段位置为(63.4,157.9)，取(63,158)或(63.5,157.5)，取决于像素中心位于整数值还是半 整数值
  - 像素显示为正方形，中心在像素位置，边长为两个相邻像素间的距离
  - OpenGL的像素中心位于半整数值位置

直线段的扫描转换

- 假定线段的端点为(x_1,y_1)和(x_2,y_2)，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);
}
```



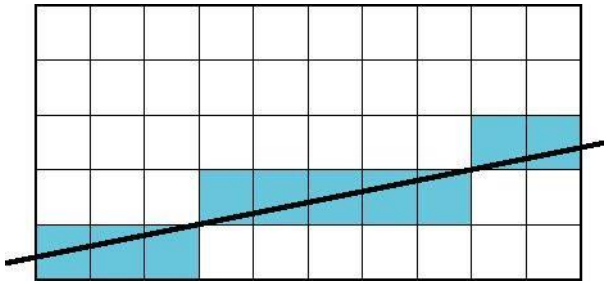
DDA算法

- 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

$$dy/dx = m = \Delta y/\Delta x = (y_2-y_1) / (x_2-x_1)$$

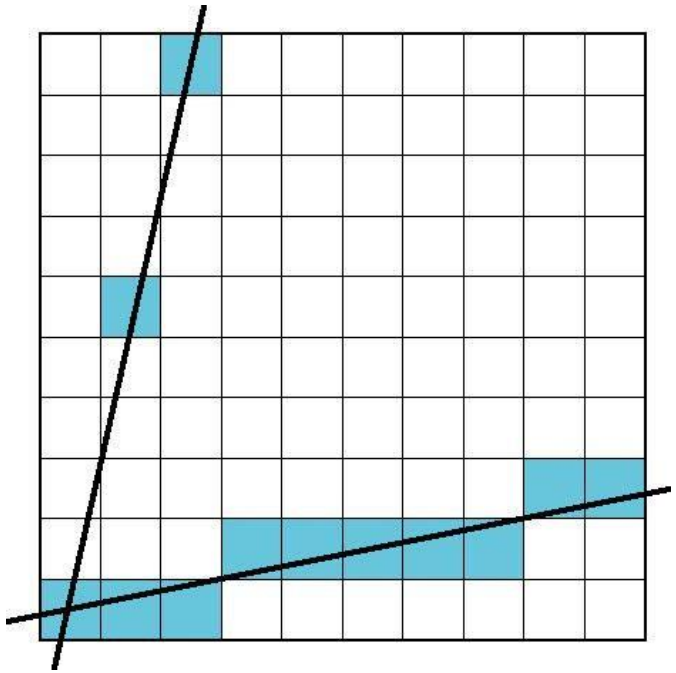
  - 沿扫描线有dx = 1, dy = m

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



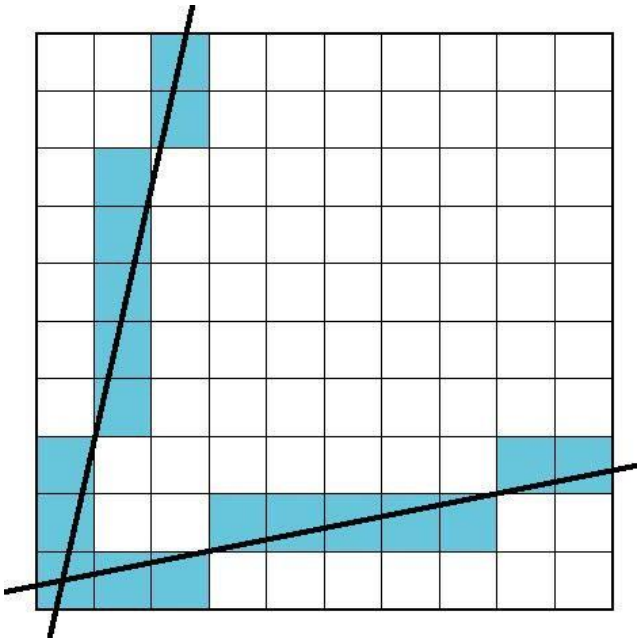
问题

- 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

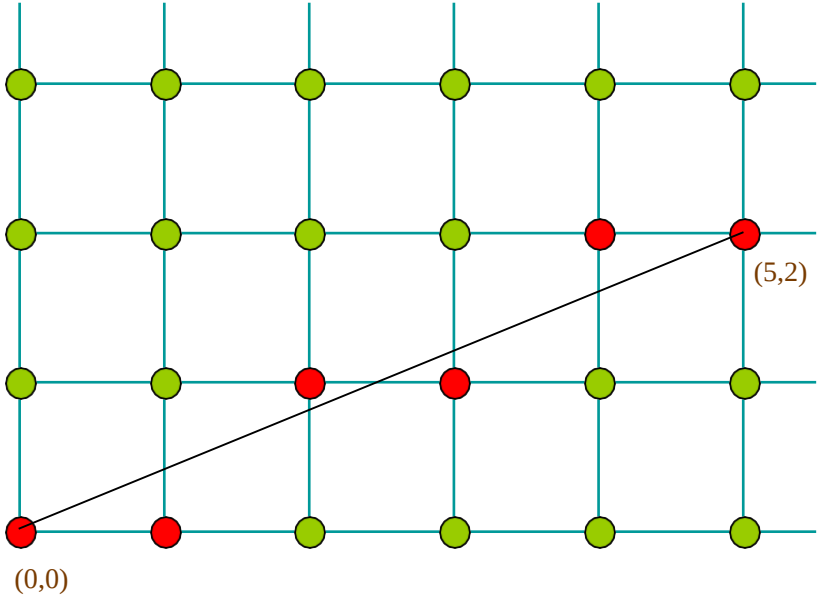


例子

- 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

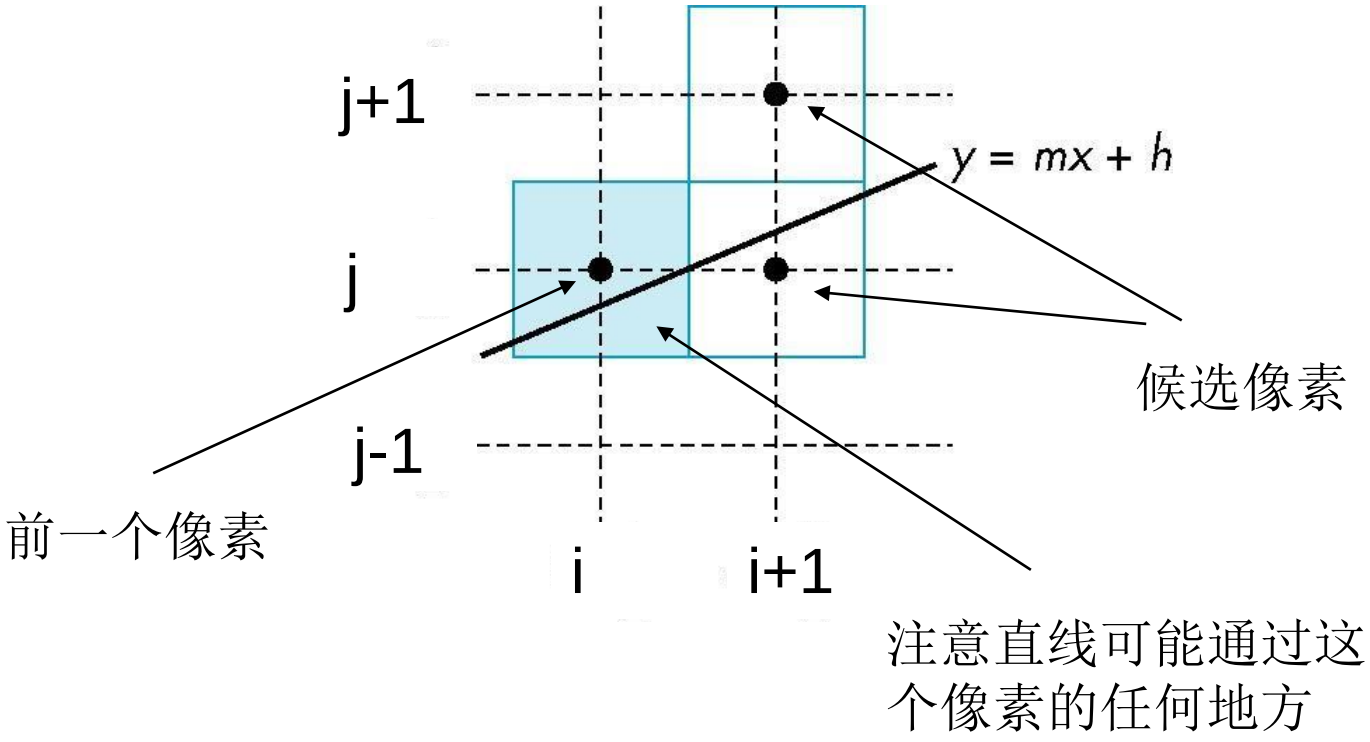


Bresenham算法

- 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$0 \le m \le 1$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
$0 \le m \le 1$，$m = \Delta y/\Delta x$，$\Delta y=y_2-y_1$，$\Delta x=x_2-x_1$

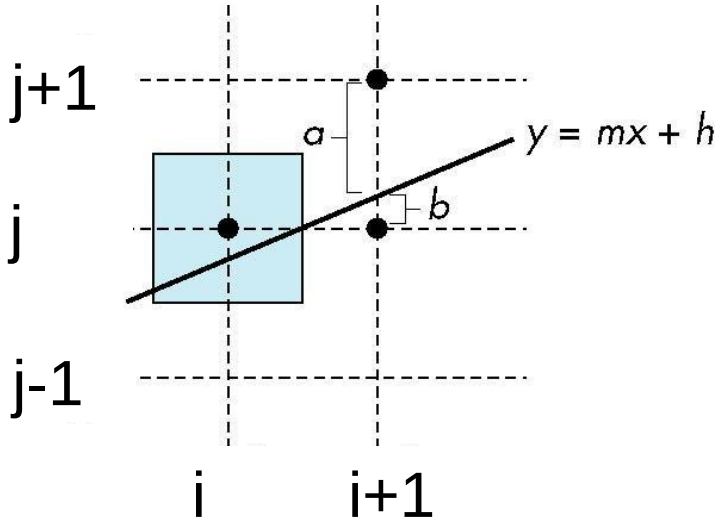


决策变量

$d = (x_2-x_1)(a-b) = \Delta x (a-b)$，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)

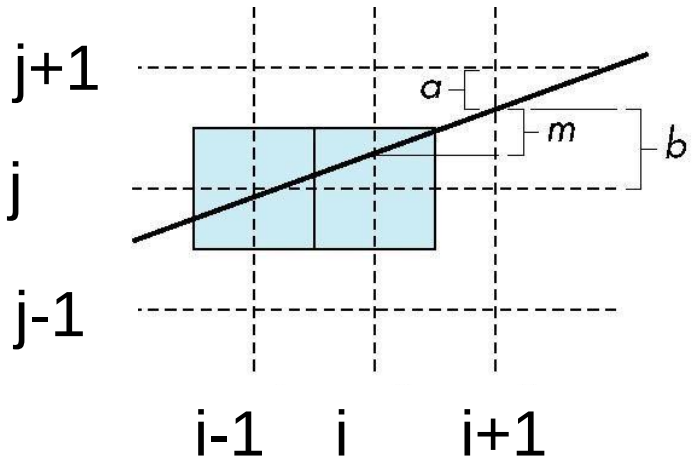


增量形式

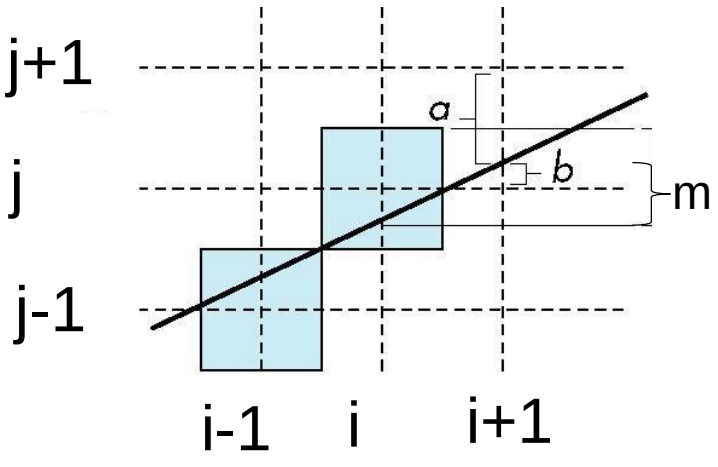
- x=k处的决策变量为$d_k = \Delta x\,(a_k - b_k)$，则

  $d_{k+1} = d_k - 2\,\Delta y$，　　如果 $d_k > 0$

  $d_{k+1} = d_k - 2(\Delta y - \Delta x)$，其他



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



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

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

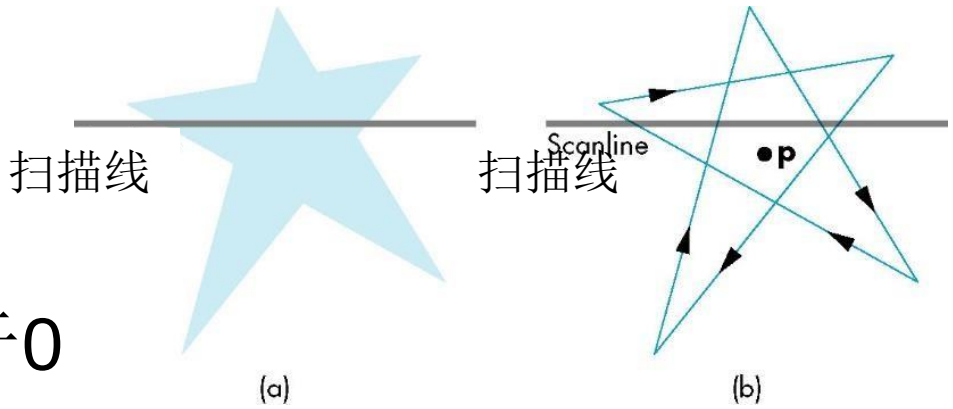
- 对多边形内部区域的填充过程等价于判断多边形所在平面上哪些点是内部点
  - 多边形填充是一个分类问题
- 如何区分内部与外部？
  - 对于非平面多边形，无法定义它的内部区域
  - 对于凸多边形，很容易做到
  - 对于非简单多边形，就非常困难
  - 可以采用奇偶检测的方法
    - 统计与多边形边的交点数
  - 环绕数 (winding number)

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

环绕数

- 首先根据多边形的边界建立一条环路
  - 从一个顶点出发，依次遍历各边，最后回到该顶点
- 一点的环绕数计算
  - 多边形各边按上述环路遍历绕该点的次数
  - 顺时针方向为正
  - 逆时针方向为负
- 内外检测
  - 如果环绕数不等于0就是内部



OpenGL与凹多边形

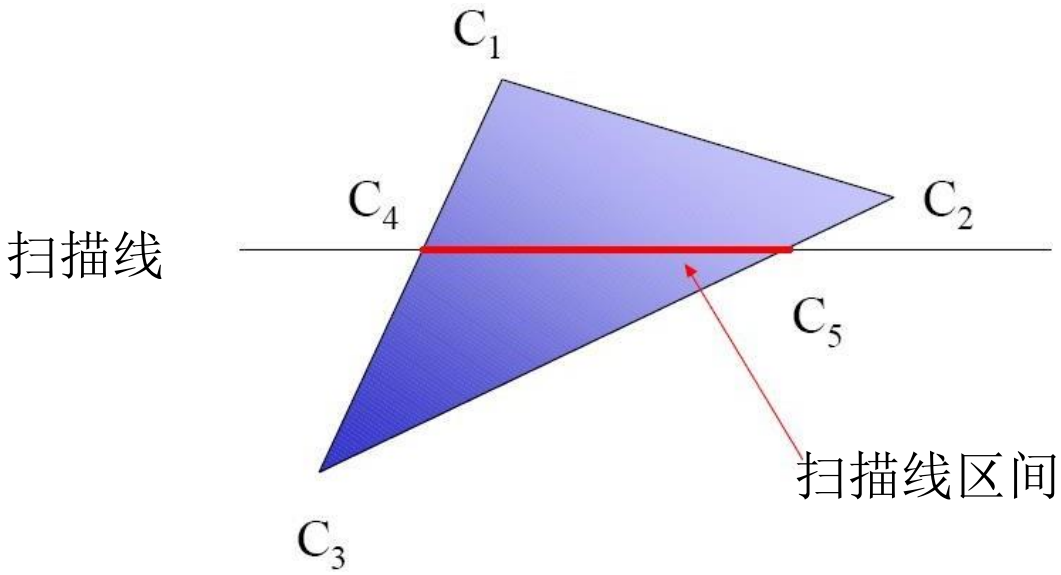
- OpenGL只保证正确填充凸多边形
- 实际应用时由用户保证这条约定被遵守，或者用其它软件把给定多边形剖分为凸多边形
  - 一般结果就是三角形的集合
  - 好的剖分算法应当不生成过长或过细的三角形
  - GLU库提供了剖分函数

在帧缓冲区中的填充

- 在流水线尾部进行填充
  - 只接受凸多边形
  - 非凸多边形需要已被剖分
  - 顶点处的亮度(颜色)已经计算出来(Gouraud明暗处理算法)
  - 与z缓冲区算法结合在一起
    - 跟踪扫描线，插值亮度
    - 增量方法的作用不大

插值

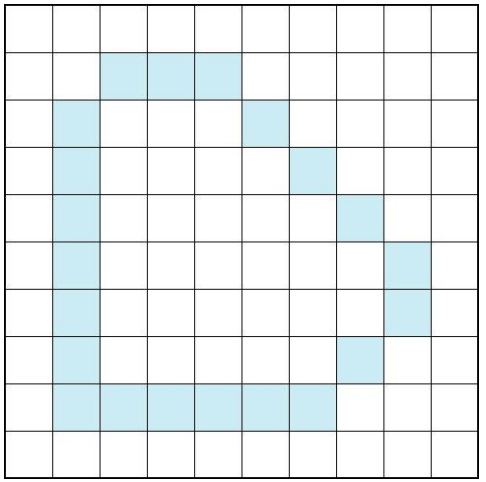
C_1 C_2 C_3 由glColor或顶点光照计算指定，C_4 由C_1和C_3插值得到，C_5由C_2和C_3插值得到，红线上各点的颜色由C_4和C_5沿扫描线区间插值得到



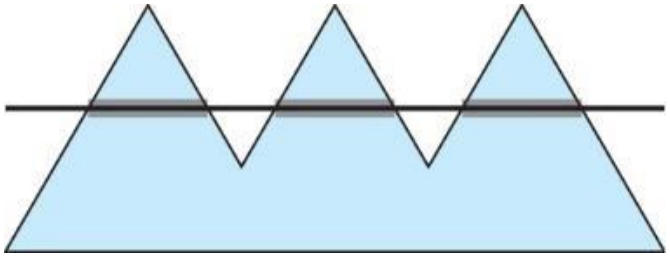
种子填充算法

- 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

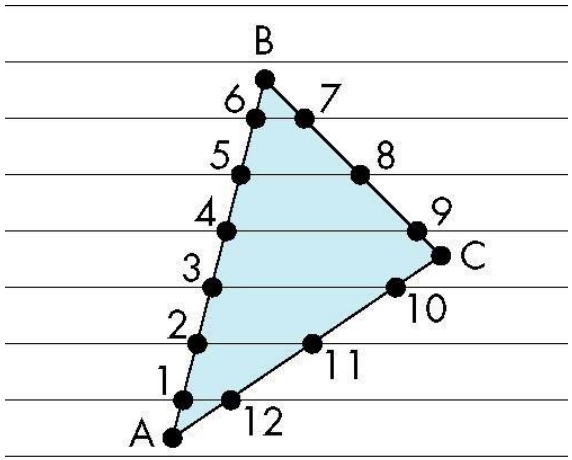


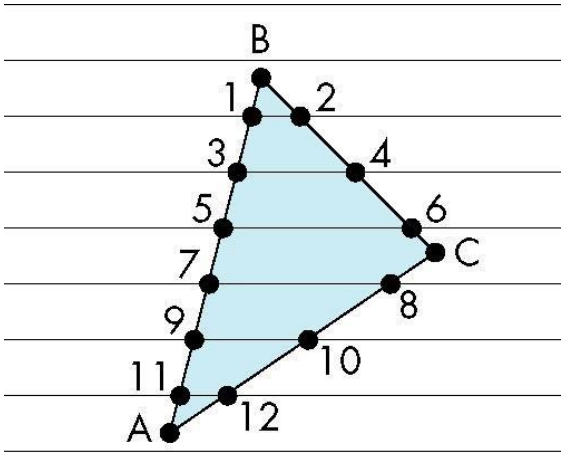
- 根据奇偶检测规则，可知扫描线上有3个填充区间位于多边形的内部
  - 填充区间由扫描线与多边形的交点集合确定
  - 沿扫描线，逐区间填充
  - 每个填充区间可以独立地进行光照或深度计算

扫描线填充

- 通过逐边处理来得到扫描线与多边形的交点序列
  - 可采用增量算法来计算交点
- 进行扫描线填充时，交点首先按扫描线排序，然后每条扫描线上按x坐标排序
  - 时间复杂度O(n log n)

通过顶点列表生成的顶点顺序

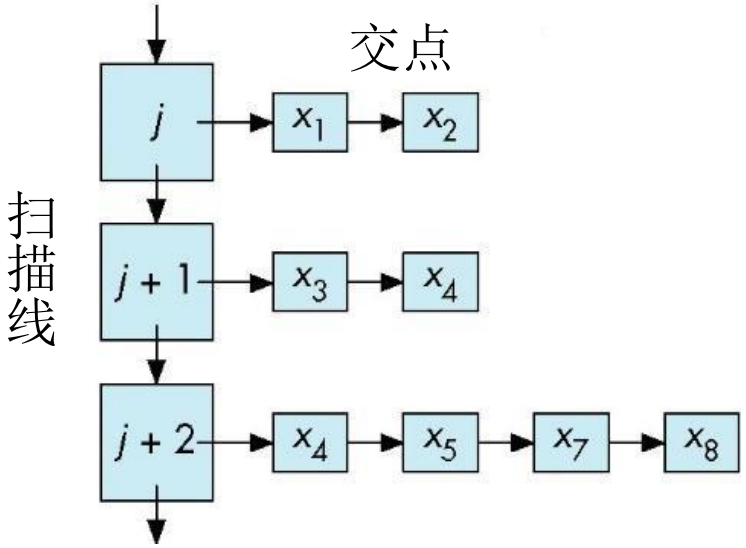




所期望的顺序

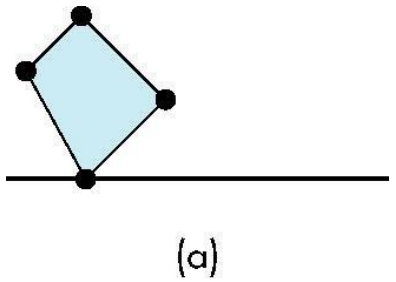
数据结构

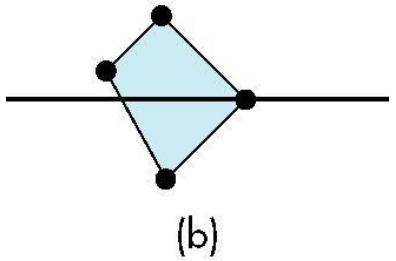
- y-x算法：
  - 为每条扫描线建一个桶
  - 对多边形每条边，把它与各扫描线的交点放入对应的桶
  - 在桶内，按x坐标值对交点进行插入排序



奇异情形

- 可以把大多数的多边形填充算法拓展到对其他封闭曲线的填充
  - 必要的细节考虑
- 即使对于多边形，对于某些算法也需要仔细考虑
- 对奇偶检测方法，顶点可能正好位于扫描线上
  - (a) 顶点处交点计0次或2次
  - (b) 顶点处交点计1次



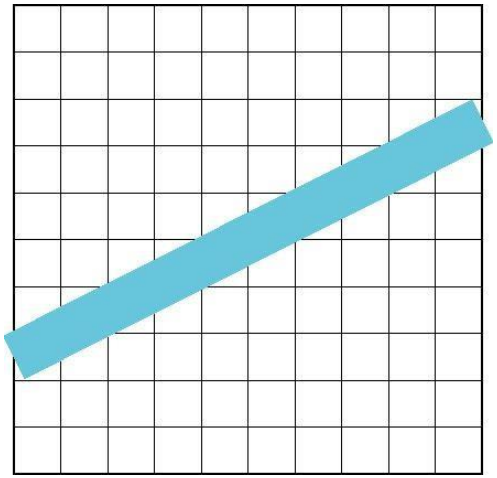


走样

- 线段和多边形边线光栅化的结果呈锯齿状
- 走样(aliasing)源于把连续表示的对象（无限分辨率）转换为离散的采样近似表示（有限分辨率）
- 走样误差是由帧缓冲区的3个问题所引起
  - 帧缓冲区的像素数固定，近似一条线段只有有限的模式
    - 多条不同的连续线段可能被相同的像素序列来近似表示
  - 像素位置固定于均匀分布栅格上
    - 不能把像素放于任意位置
  - 像素的大小和形状固定

线段的走样

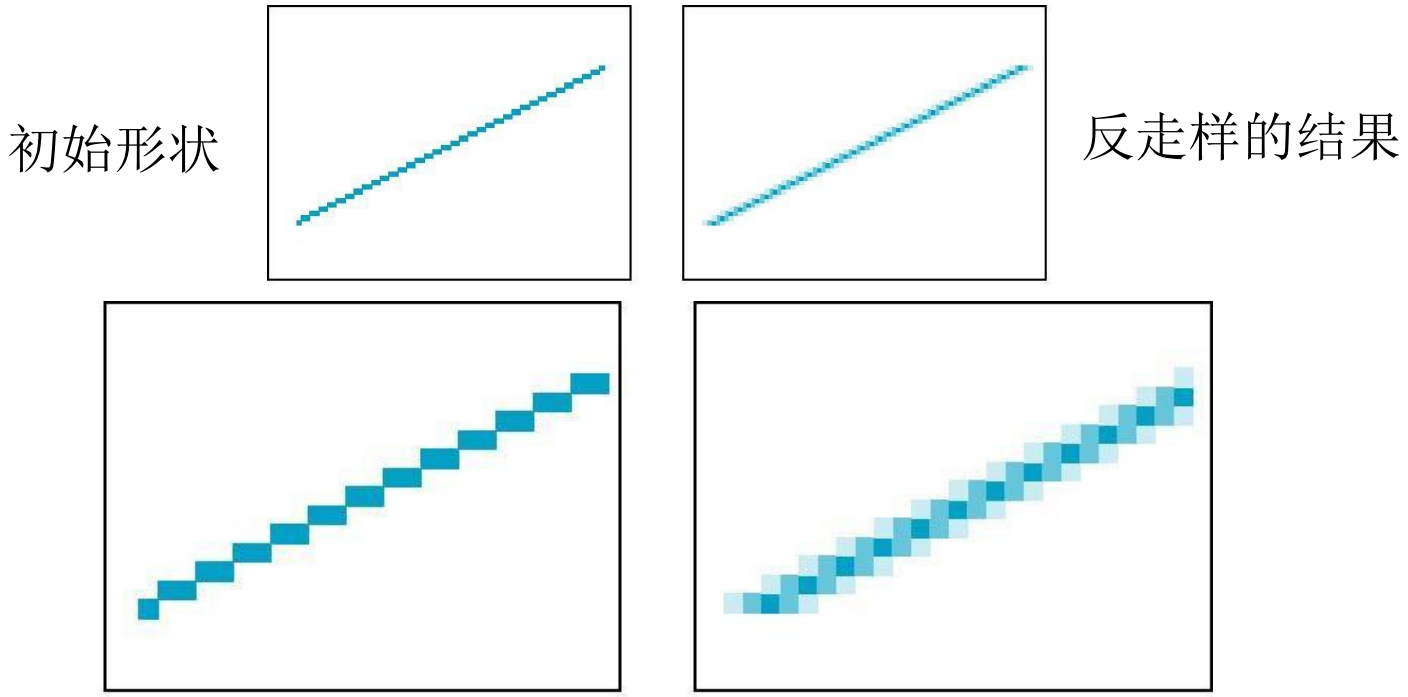
- 理想的线段应当是一个像素宽，但不能直接绘制，必须离散成正方形像素序列



- 对于每个x选择最佳的y(或者反过来)会导致走样的光栅化直线

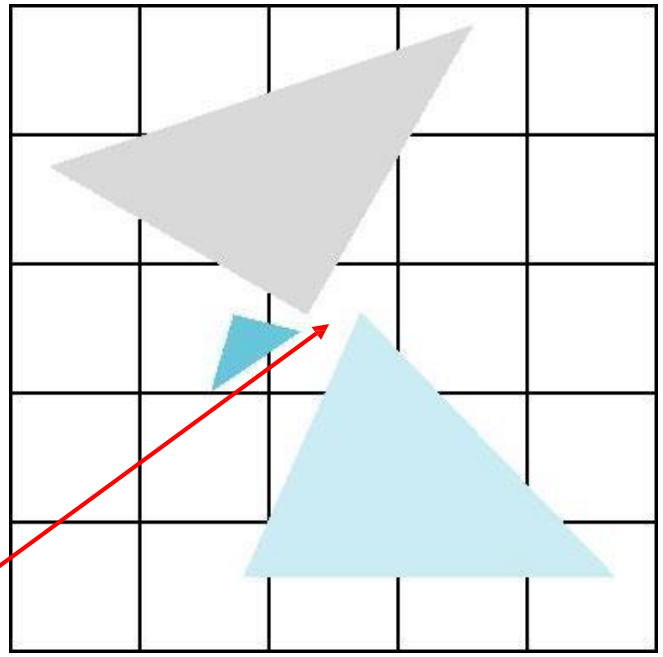
面积平均反走样

- 对每个x，根据理想直线所覆盖的面积对多个像素着色



多边形的走样

- 对于多边形，走样问题可能非常严重
  - 边的锯齿形状
  - 小多边形被忽略
  - 需要进行颜色组合，从而可能一个多边形并不完全确定一个像素的颜色



三个多边形对该像素的颜色都应该有贡献

时间域走样

- 当生成图像序列，例如动画时，除了之前讨论的空间域走样，还要考虑时间域走样
  - 如果对象的投影非常小，那么移动过程中可能不会在投影平面成像，从而观察者会看到该对象在屏幕上闪烁