

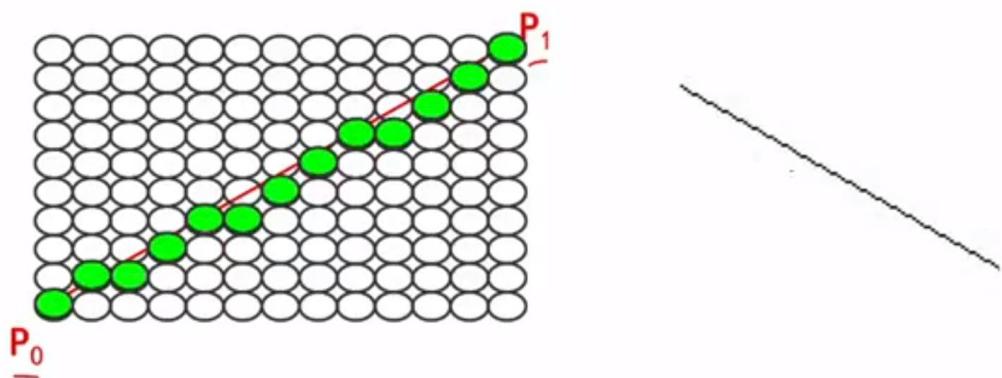
# 计算机图形学算法

- 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
- 裁剪算法
- 反走样算法
- 消隐算法

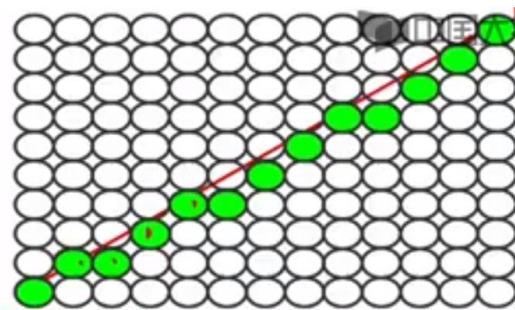


## 一：直线段的扫描转换算法

在数学上，直线上的点有无穷多个。但当在计算机光栅显示器屏幕上表示这条直线时需要做一些处理



为了在光栅显示器上用这些离散的像素点逼近这条直线，  
需要知道这些像素点的  $x$ ,  $y$   
坐标



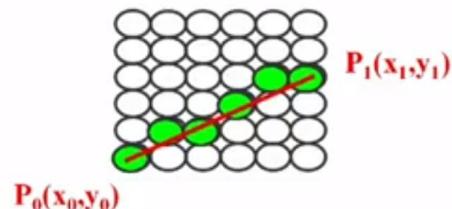
求出过  $P_0, P_1$  的直线段方程：

$$y = kx + b$$

$$k = \frac{(y_1 - y_0)}{(x_1 - x_0)} \quad (x_1 \neq x_0)$$

$$y = kx + b \quad k = \frac{(y_1 - y_0)}{(x_1 - x_0)} \quad (x_1 \neq x_0)$$

假设  $x$  已知，即从  $x$  的起点  $x_0$  开始  
，沿  $x$  方向前进一个像素（步长  
= 1），可以计算出相应的  $y$  值。



因为像素的坐标是整数，所以  $y$  值  
还要进行取整处理

## 如何把数学上的一个点扫描转换一个屏幕像素点？

如： $p(1.7, 0.8)$   $\xrightarrow{\text{取整}} p(1, 0)$

$p(1.7, 0.8)$   $\xrightarrow{+0.5} p(2.2, 1.3)$

$p(2.2, 1.3)$   $\xrightarrow{\text{取整}} p(2, 1)$

不是直接取整，而是  $+0.5$  后再取整

$$\underline{y = kx + b}$$

直线是最基本的图形，一个动画或真实感图形往往需要调用成千上万次画线程序，因此直线算法的好坏与效率将直接影响图形的质量和显示速度

$$\underline{\textcolor{red}{y}} = \underline{kx} + b$$

为了提高效率，把计算量减下来，关键问题就是如何把~~乘法~~取消？

在计算机的各种运算中，执行速度最快的就是加法运算。

## 二、直线绘制的三个著名的常用算法

1、数值微分法（DDA）

2、中点画线法

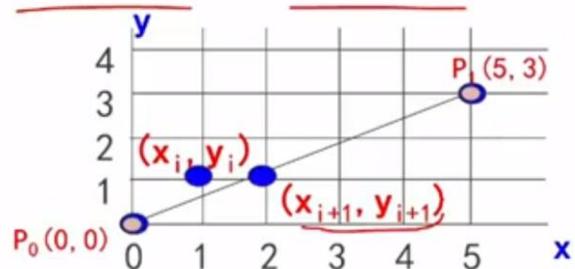
3、Bresenham算法

## 1、数值微分算法

### 1、数值微分DDA(Digital Differential Analyzer)法

引进图形学中一个很重要的思想——增量思想

$$\begin{aligned}y_i &= kx_i + b \\y_{i+1} &= kx_{i+1} + b \\&= k(x_i + 1) + b\end{aligned}$$



解释上述的推导过程，根据直线的斜截式，我们可以根据  $x_i$  的值来得到  $y_i$  的值，同样的我们也可以用  $x_{i+1}$  得到  $y_{i+1}$  的值，而由于  $x_i$  和  $x_{i+1}$  之间只差 1，即每次步进都是 1 个像素，则  $x_{i+1}$  可以写为  $x_i + 1$ 。如上图所示。

$$\begin{aligned}\cancel{y_i} &= \cancel{kx_i} + b \\y_{i+1} &= kx_{i+1} + b \\&= k(x_i + 1) + b \\&= kx_i + \cancel{k} + b \\&= \cancel{kx_i} + b + k \\&= y_i + k\end{aligned}$$

继续推导，我们将括号展开，再移项得到倒数第二步的式子，我们可以观察到，前面一部分就是  $y_i$  的值，因此我们可以将  $y_{i+1}$  表示为  $y_i$  和  $k$  的和。

$$\cancel{y_{i+1}} = \cancel{y_i} + k$$

中国大学MOOC

这个式子的含义是：当前步的  $y$  值等于前一步的  $y$  值加上斜率  $k$

这样就把原来一个乘法和加法变成了一个加法！

下面举一例：

用DDA扫描转换连接两点 $P_0(0, 0)$ 和 $P_1(5, 3)$ 的直线段。

$$k = \frac{y_1 - y_0}{x_1 - x_0} = \frac{3 - 0}{5 - 0} = 0.6 < 1 \quad y_{i+1} = y_i + k$$

x	y	$\text{int}(y + 0.5)$
0	0	0
1	0+0.6	1
2	0.6+0.6	1
3	1.2+0.6	2
4	1.8+0.6	2
5	2.4+0.6	3

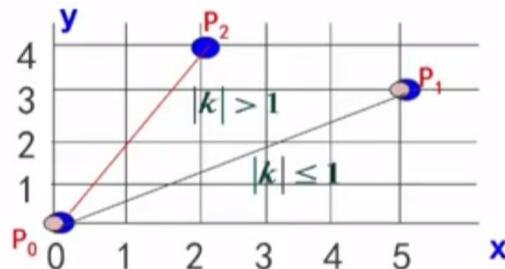


用 DDA 算法来画直线，我们可以看到，乘法只进行了一次

(1) DDA画直线算法：x每递增1，y递增斜率k。是否适合任意斜率的直线？

$$|k| \leq 1$$

$$x_{i+1} = x_i + 1 \\ y_{i+1} = y_i + k$$

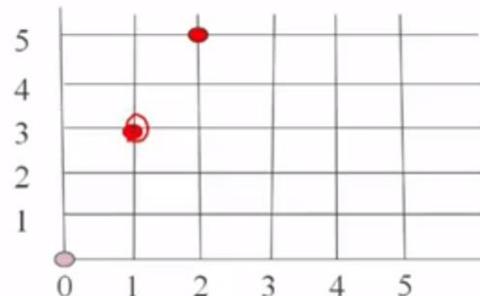


$$|k| > 1$$

用DDA方法画连接两点 $P_0(0, 0)$ 和 $P_1(2, 5)$ 的直线段

$$K=5/2=2.5>1 \quad y_{i+1} = y_i + k$$

x	y	$\text{int}(y + 0.5)$
0	0	0
1	2.5	3
2	5	5



再比如直线点从 $(0, 0)$ 到 $(2, 100)$ ，也只用3个点来表示

用DDA算法有这种问题存在，当斜率大于1时，有可能很长的一条直线只通过一个像素来表示，这显然是不合理的，录入上图中的下例。使得光栅点太稀了，没有办法产生一条连续的直线

## 2、中点画线法

(1) 改进效率。这个算法每步只做一个加法，能否再提高效率？

$$\underline{y_{i+1} = y_i + k}$$

一般情况下k与y都是小数，而且每一步运算都要对y进行四舍五入后取整。

由于整数运算效率要高于浮点运算，因此唯一改进的途径是把浮点运算变成整数加法！

(2) 第二个思路是从直线方程类型做文章

$$y = kx + b$$

而直线的方程有许多类型，如两点式、一般式等。如用其它的直线方程来表示这条直线会不会有出人意料的效果？

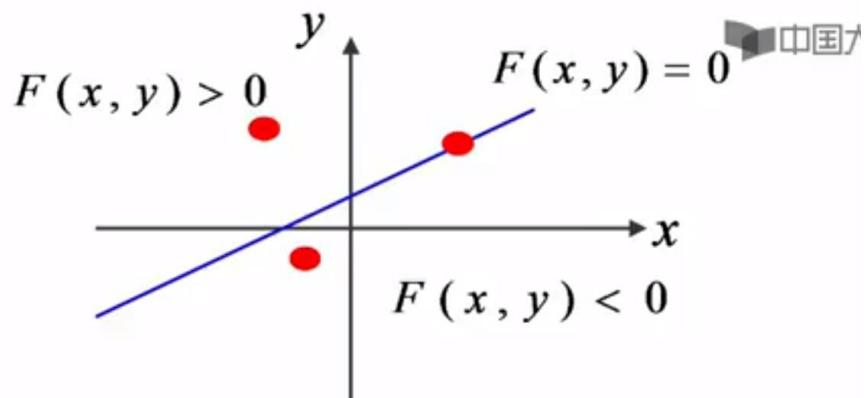
## 中点画线法

直线的一般式方程：

$$F(x, y) = 0$$

$$Ax + By + C = 0$$

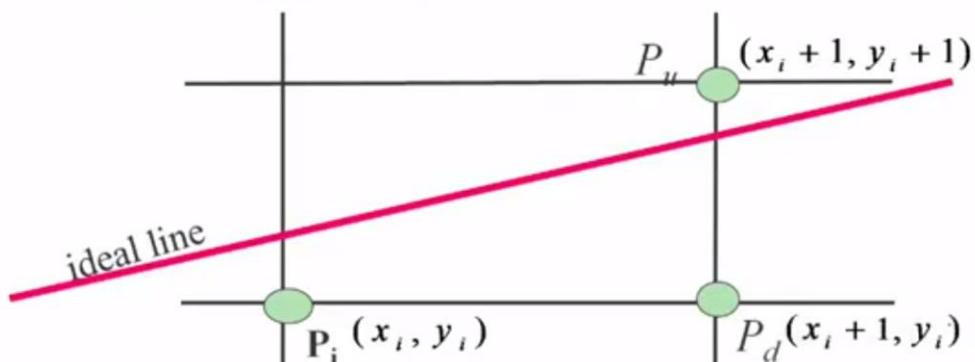
其中:  $A = -(\Delta y)$ ;  $B = (\Delta x)$ ;  $C = -B(\Delta x)$



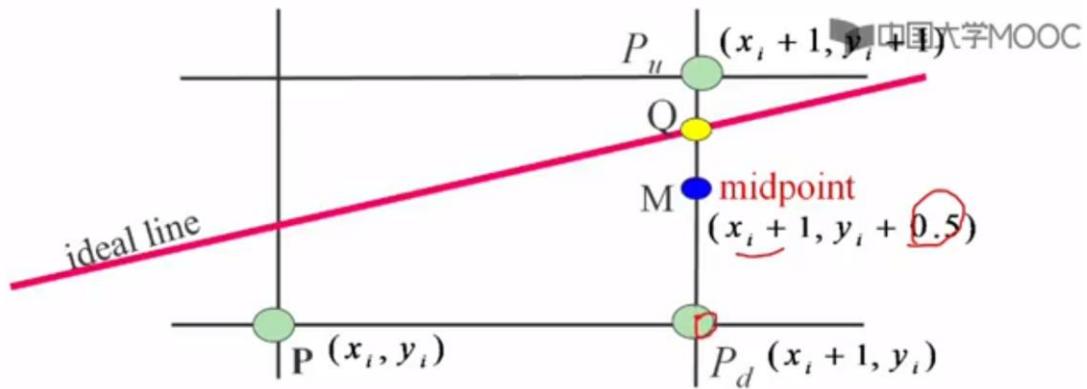
- 对于直线上的点:  $F(x, y) = 0$
- 对于直线上方的点:  $F(x, y) > 0$
- 对于直线下方的点:  $F(x, y) < 0$

每次在最大位移方向上走一步, 而另一个方向是走步还是不走步要取决于中点误差项的判断。

假定:  $0 \leq |k| \leq 1$ 。因此, 每次在x方向上加1, y方向上加1或不变需要判断。



在x前进1后, 得到的点可能是  $P_d$  或者  $P_u$ ,



当M在Q的下方，则 $P_u$ 离直线近，应为下一个象素点

当M在Q的上方，应取 $P_d$ 为下一点。

我们取 $P_uP_d$ 的中点M，其坐标如图所示，我们再设直线和 $P_uP_d$ 的交点为Q，根据M和Q的位置关系，我们能够得到上图所述的结论。如果M和Q恰好是一个点，则可以取 $P_u$ 和 $P_d$ 两点中任意一个作为下一个像素点

### 把M代入理想直线方程：

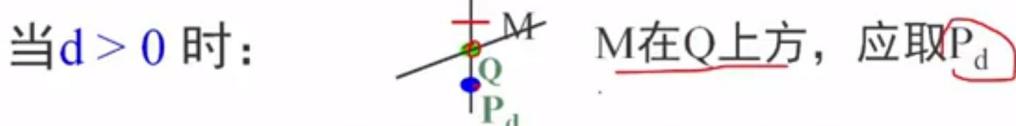
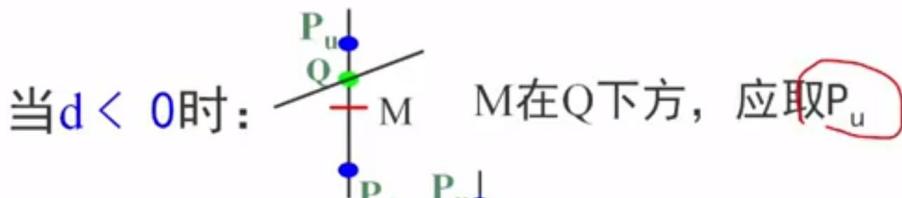
$$F(x_m, y_m) = \underbrace{Ax_m + By_m + C}_{\cdot}$$

$$\begin{aligned} d_i &= F(x_m, y_m) = F(x_i + 1, y_i + 0.5) \\ &= \underbrace{A(x_i + 1) + B(y_i + 0.5) + C}_{\cdot} \end{aligned}$$

将坐标带入后得到该式

$$d_i = F(x_m, y_m) = F(x_i + 1, y_i + 0.5)$$

$$= \underline{A(x_i + 1)} + \underline{B(y_i + 0.5)} + C$$



当  $d = 0$  时: M在直线上, 选  $p_d$  或  $p_u$  均可。

我们将 M 带入后得到的结果有如下结论: 即带入的 M 点的值和 0 的关系,  $d > 0$  时, M 在 Q 上方, 应该取  $P_d$ , 相反的,  $d < 0$  时, M 在 Q 的下方, 则应该取  $P_u$ , 当  $d = 0$  时, 两点重合, 可以任意选择

$$y = \begin{cases} \underline{y + 1} & (d < 0) \\ y & (d \geq 0) \end{cases}$$

## 这就是中点画线法的基本原理

下面来分析一下中点画线算法的计算量?

$$y = \begin{cases} y + 1 & (d < 0) \\ y & (d \geq 0) \end{cases}$$

$$d_i = \underline{A(x_i + 1)} + \underline{B(y_i + 0.5)} + C$$

为了求出  $d$  值, 需要两个乘法, 四个加法

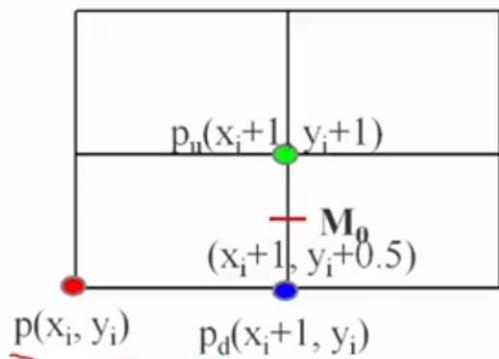
显然运算效率低下, 我们再采用增量的思想来简化算法的计算量

能否也采用增量计算，提高运算效率呢？

$$d_{i+1} = d_i + ? \quad \text{增量}$$

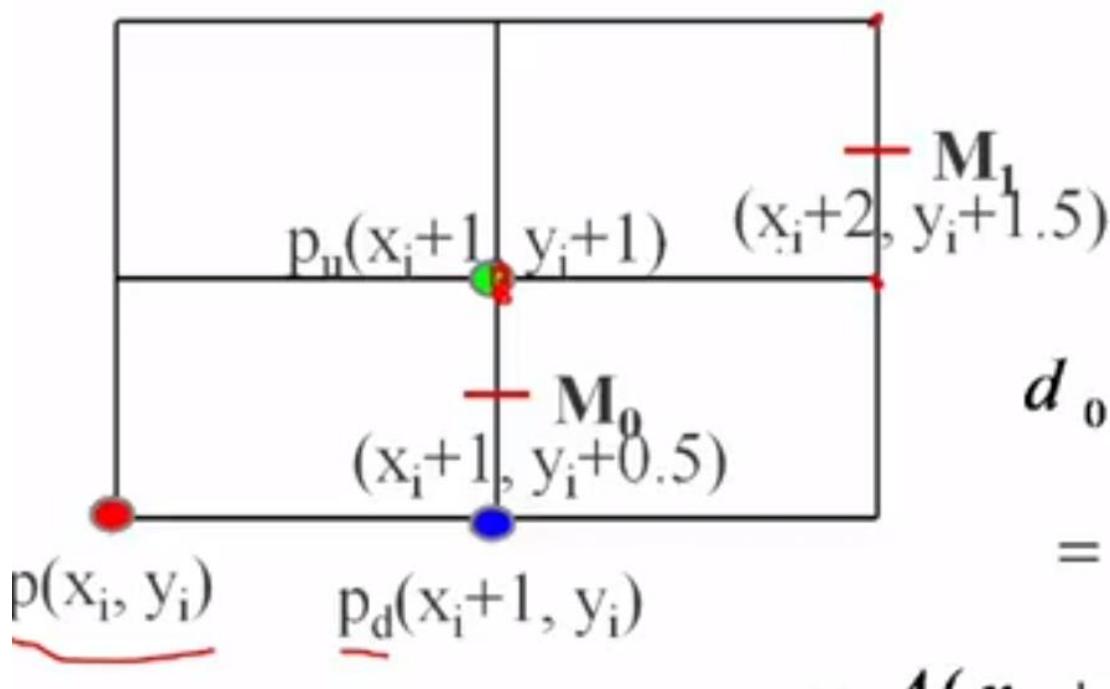
d是x, y的线性函数，采用增量计算是可行的  
如何推导出d值的递推公式？

中国大学MOOC



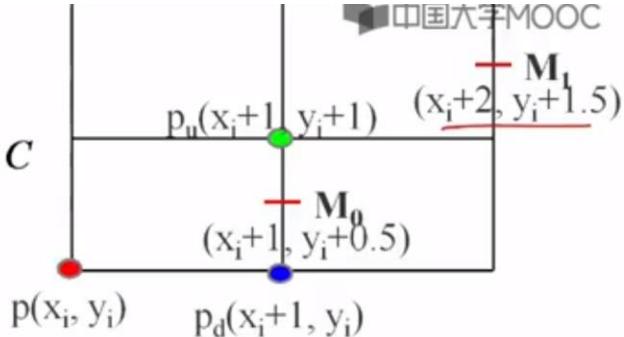
$$y = \begin{cases} y + 1 & (d < 0) \\ y & (d \geq 0) \end{cases}$$

$$\begin{aligned} d_0 &= F(x_m, y_m) \\ &= F(\underline{x_i} + 1, \underline{y_i} + 0.5) \\ &= A(x_i + 1) + B(y_i + 0.5) + C \end{aligned}$$



$$\begin{aligned}
 d_0 &= F(x_{m0}, y_{m0}) \\
 &= F(x_i + 1, y_i + 0.5) \\
 &= A(x_i + 1) + B(y_i + 0.5) + C
 \end{aligned}$$

$$\begin{aligned}
 d &= F(x_{m1}, y_{m1}) \\
 &= F(x_i + 2, y_i + 1.5) \\
 &= A(x_i + 2) + B(y_i + 1.5) + C \\
 &= A(x_i + 1) + B(y_i + 0.5) + C + \underline{A} + \underline{B}
 \end{aligned}$$

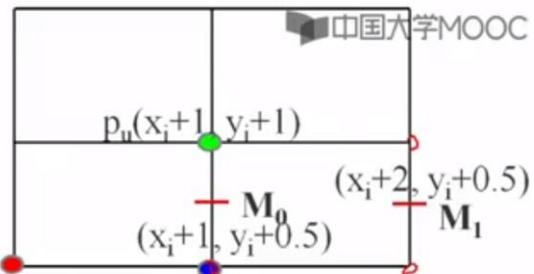


我们将  $M_1$  带入计算  $d_1$  的值，经过展开后，我们可以得到  $d_0$  和  $d_1$  之间的关系，这样我们只需在第一次进行繁杂的计算，而在后续计算都是用增量计算即可。

$$= \underline{d_0} + \underline{A + B}$$

我们得到的增量就是  $A+B$

$$y = \begin{cases} y + 1 & (d < 0) \\ y & (d \geq 0) \end{cases}$$

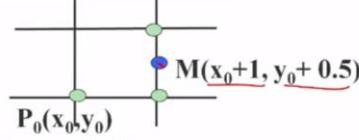


$$\begin{aligned}
 d_1 &= F(x_i + 2, y_i + 0.5) \\
 &= A(x_i + 2) + B(y_i + 0.5) + C \\
 &= \underline{A(x_i + 1)} + B(y_i + 0.5) + C + \underline{A} \\
 &= d_0 + A
 \end{aligned}$$

再记录一下  $d$ ---这个  $d$  是将中点带入直线方程之后得到的值，如果  $d>0$ ，那说明中点在直线之上，取下面的像素点，如果  $d<0$ ，说明中点在直线之上，取上面的像素点。

而根据不同的  $d$ ，我们增量计算的增量计算值也不同，当  $d<0$  时，增量为  $A+B$ ，当  $d>0$ ，增量为  $A$

下面计算d的初始值 $d_0$ , 直线的第一个像素 $P_0(x_0, y_0)$ 在直线上, 因此相应的d的初始值计算如下:



$$\begin{aligned} d_0 &= F(x_0 + 1, y_0 + 0.5) \\ &= A(x_0 + 1) + B(y_0 + 0.5) + C \\ &= \cancel{Ax_0 + By_0 + C} + A + 0.5B \\ &= \underline{\underline{A + 0.5B}} \end{aligned}$$

那如何计算第一个d呢? 这里我们代入计

算可以得到 $d_0$ 的值, 其实就是 $A+0.5B$  (上式左边=0),

$$d_{new} = \begin{cases} d_{old} + A + B & d < 0 \\ d_{old} + A & d \geq 0 \end{cases} \quad d_0 = \underline{\underline{A + 0.5B}}$$

总的中点算法如图  
所示

至此, 中点算法至少可以和DDA算法一样好!

可以用 $2d$ 代替 $d$ 来摆脱浮点运算, 写出仅包含**整数运算**的算法。

这样, 中点生成直线的算法提高到整数加法, 优于DDA算法。

$$(1) \quad \underline{\underline{Ax + By + C = 0}}$$

(2) 通过判中点的符号, 最终可以只进行整数加法

### 3、Bresenham 算法

中国大学MOOC

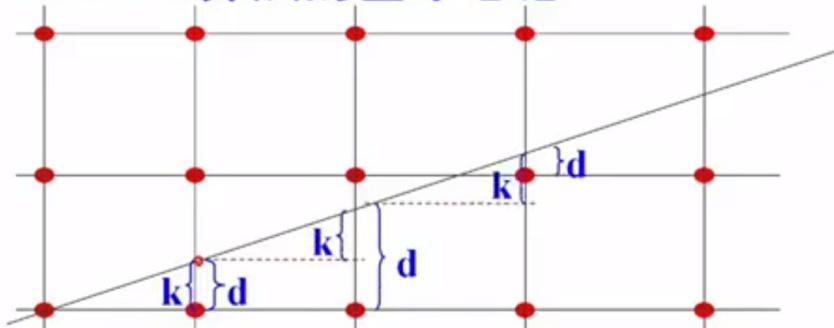
DDA把算法效率提高到每步只做一个加法。

中点算法进一步把效率提高到每步只做一个整数加法

Bresenham提供了一个更一般的算法。该算法不仅有好的效率，而且有更广泛的适用范围

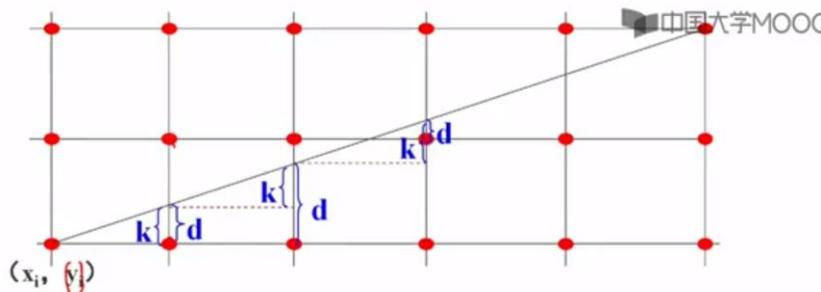
#### Bresenham算法的基本思想

中国大学MOOC



该算法的思想是通过各行、各列像素中心构造一组虚拟网格线，按照直线起点到终点的顺序，计算直线与各垂直网格线的交点，然后根据误差项的符号确定该列象素中与此交点最近的象素。

看上图，直线与虚拟网络的第二个交点，可以得知与下面像素点的距离  $d$  和斜率  $k$  相同，



假设每次  $x+1$ ,  $y$  的递增（减）量为 0 或 1，它取决于实际直线与最近光栅网格点的距离，这个距离的最大误差为 0.5。

如果  $d$  小于 0.5，则离

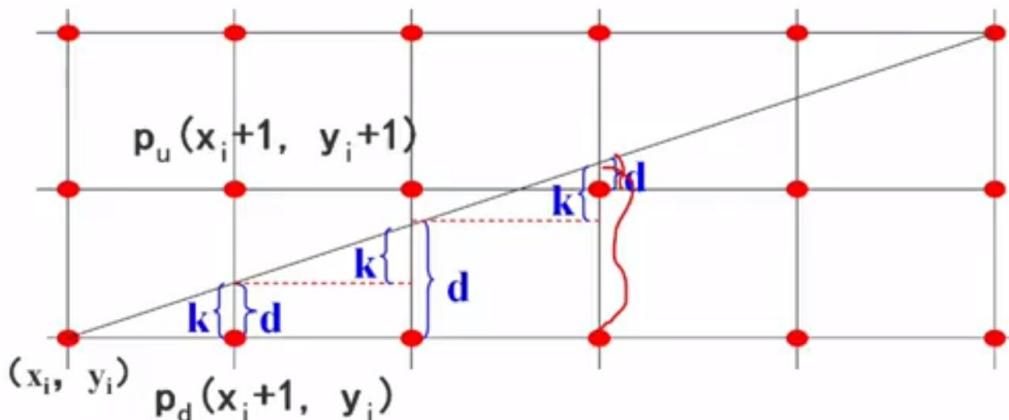
下面的像素点近，如果  $d > 0.5$ ，则离上面的像素点近

误差项d的初值  $d_0 = 0$

中国大学MOOC

$$d = d + k$$

一旦  $\underline{d} \geq 1$ , 就把它 减去1, 保证d的相对性, 且在0、1之间。



如果  $d > 0.5$ , 则下一点为  $P_u$ , 反之,  $d < 0.5$ , 下一点为  $P_d$ ,  
Bresenham 算法如下

$$\begin{cases} x_{i+1} = x_i + 1 \\ y_{i+1} = \begin{cases} y_i + 1 & (d \geq 0.5) \\ y_i & (d \leq 0.5) \end{cases} \end{cases}$$

如何把这个算法的效率也提高到整数加法?

**改进1:** 令  $e = d - 0.5$

$$\begin{cases} x_{i+1} = x_i + 1 \\ y_{i+1} = \begin{cases} y_i + 1 & (e > 0) \\ y_i & (e \leq 0) \end{cases} \end{cases}$$

$$\begin{cases} x_{i+1} = x_i + 1 \\ y_{i+1} = \begin{cases} y_i + 1 & (d > 0.5) \\ y_i & (d \leq 0.5) \end{cases} \end{cases}$$

我们用  $e$  来代替  $d$  进行判断，经过该操作之后，不再是进行数值的判断，而是符号的判断了。

$e > 0$ ,  $y$  方向递增1;  $e < 0$ ,  $y$  方向不递增

$e = 0$  时，可任取上、下光栅点显示

- $e_{\text{初}} = -0.5$
- 每走一步有  $e = \underline{e+k}$
- if ( $e > 0$ ) then  $e = \underline{e-1}$

经过该操作， $e$  初值为  $-0.5$ ，与  $d$  相同，新的  $e$  的增量仍然为  $k$ ，而当  $e > 0$  时（即  $d > 1$  时），将  $e-1$ ，得到新的  $e$ ，保证  $e$  的模在  $0-1$  之间

$$e_{\text{初}} = -0.5 \quad k = \frac{dy}{dx}$$

中国大学MOOC

**改进2：**由于算法中只用到误差项的符号，于是可以用  $e * 2 * \Delta x$  来替换  $e$ 。

经过该操作，新的  $e$  不再有浮点运算，改为判断符号，同时， $dx$  项抵消掉  $k$  的  $1/dx$ ，使得除法消失，提高效率

■  $e_{\text{初}} = -\Delta x$ ,

■ 每走一步有：  $e = e + 2\Delta y$ 。

■ if ( $e > 0$ ) then  $e = e - 2\Delta x$

同时， $e$  的初值变为  $-dx$ ，

The image shows handwritten mathematical steps on a piece of paper. At the top, it says  $e = e + k$ . Below that,  $e_{in} = -0.5 \times 2 \times dx = -dx$ . Then,  $e = e_{in} + k = -dx + \frac{dy}{dx}$ , with  $-dx$  crossed out. A large oval encloses the equation  $2e + 2dx = 2e_{in} + 2\frac{dy}{dx}$ , with  $2e$  crossed out.

具体的推导如图

实际上  $e$  的取模操作需要减去的就是  $e$  的乘项，因此减去  $2dx$ ，就可以起到和前面一样的取模效果

## 算法步骤为：

中国大学MOOC

1. 输入直线的两端点  $P_0(x_0, y_0)$  和  $P_1(x_1, y_1)$ 。
2. 计算初始值  $\Delta x$ 、 $\Delta y$ 、 $e = -\Delta x$ 、 $x = x_0$ 、 $y = y_0$ 。
3. 绘制点  $(x, y)$ 。
4.  $e$  更新为  $e + 2\Delta y$ ，判断  $e$  的符号。若  $e > 0$ ，则  $(x, y)$  更新为  $(x+1, y+1)$ ，同时将  $e$  更新为  $e - 2\Delta x$ ；否则  $(x, y)$  更新为  $(x+1, y)$ 。
5. 当直线没有画完时，重复步骤3和4。否则结束。

Bresenham算法很像DDA算法，都是加斜率

但DDA算法是每次求出一个新的y以后取整来画；而Bresenham算法是判符号来决定上下两个点。所以该算法集中了DDA和中点两个算法的优点，而且应用范围广泛

## 1、计算机科学问题的核心就是算法

$$\underline{y = kx + b}$$

## 2、领会算法中所蕴含的创新思想

改进和完善算法的过程中所体现出来的一些闪光的思想是我们所要认识和领会的

## 3、科学研究无止境，学术面前人人平等

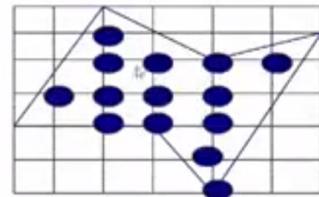
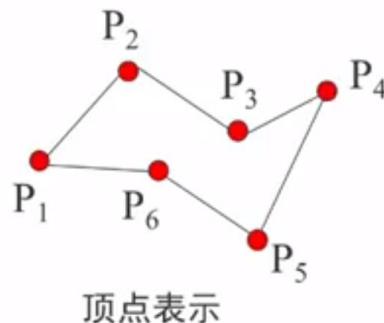
# 二：多边形的扫描转换和区域填充

## 1、多边形的扫描转换

多边形的扫描转换和区域填充这个问题是怎么样在离散的像素集上表示一个连续的二维图形



多边形有两种重要的表示方法：顶点表示和点阵表示



点阵表示

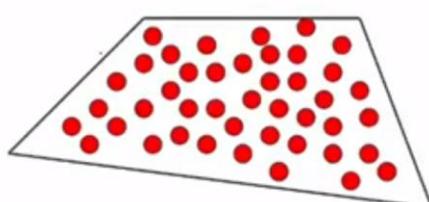
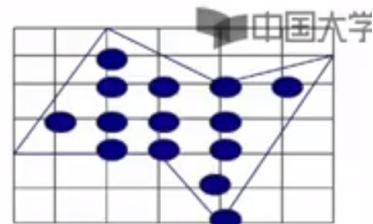
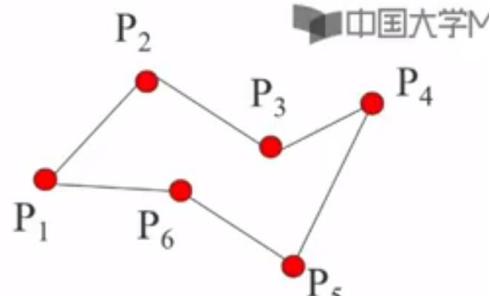
**顶点**表示是用多边形的顶点序列来表示多边形。这种表示直观、几何意义强、占内存少，易于进行几何变换

但由于它没有明确指出哪些像素在多边形内，故不能直接用于面着色

**点阵**表示是用位于多边形内的像素集合来刻画多边形。这种表示丢失了许多几何信息（如**边界**、**顶点**等），但它却是光栅显示系统显示时所需的表示形式。

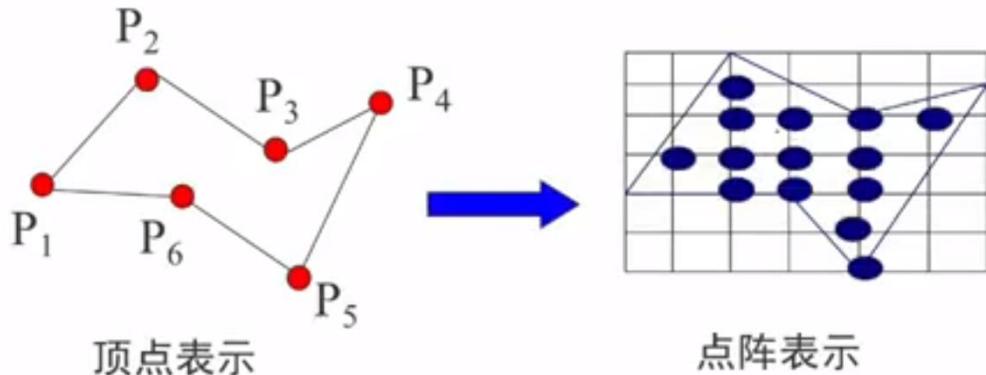
这涉及到两个问题：第一个问题是如果知道边界，能否求出哪些像素在多边形内？

第二个问题是知道多边形内部的像素，反过来如何求多边形的C边界？



第二个问题不属于计算机图形学的范畴，而是图像识别的问题，因此不考虑这个问题。

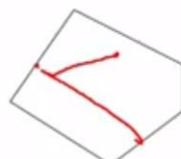
光栅图形的一个基本问题是把多边形的顶点表示转换为点阵表示。这种转换称为**多边形的扫描转换**



多边形分为凸多边形、凹多边形、含内环的多边形等。

### (1) 凸多边形

任意两顶点间的连线均在多边形内



### (2) 凹多边形

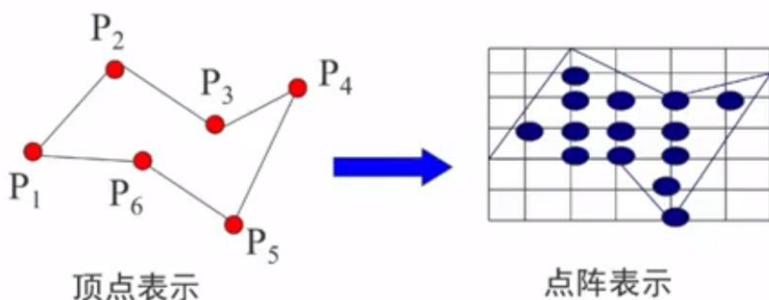
任意两顶点间的连线有不在在多边形内



### (3) 含内环的多边形

多边形内包含多边形

现在的问题是，知道**多边形的边界**，如何找到**多边形内部**的点，即把多边形内部填上颜色



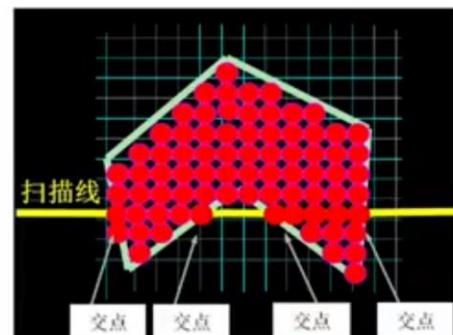
# X 扫描线算法

## 1、X-扫描线算法

中国大学MOOC

**X-扫描线算法**填充多边形的基本思想是按扫描线顺序，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像素，即完成填充工作

区间的端点可以通过计算扫描线与多边形边界线的交点获得

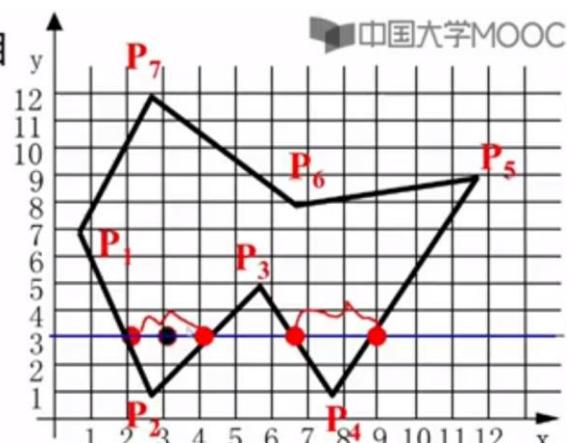


将扫描线与图形的交点之间的像素点上色，来完成填充工作，这样的扫描线有很多，这样就能够完成整个图形的填充

如扫描线 $y=3$ 与多边形的边界相交于4点：

$(2, 3)$ 、 $(4, 3)$ 、 $(7, 3)$ 、  
 $\underline{(9, 3)}$ 。

这四点定义了扫描线从 $x=2$ 到 $x=4$ ，从 $x=7$ 到 $x=9$ 两个落在多边形内的区间，该区间内的像素应取填充色



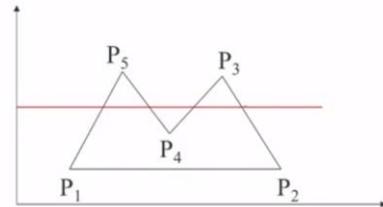
算法的核心是按X递增顺序排列交点的X坐标序列。由此，可得到X-扫描线算法步骤如下：

(1) 确定多边形所占有的最大扫描线数，得到多边形顶点的最小和最大y值 ( $y_{min}$  和  $y_{max}$ )

(2) 从  $y = y_{\min}$  到  $y = y_{\max}$ ,  
每次用一条扫描线进行填充

(3) 对一条扫描线填充的过程可分为四个步骤:

- a、求交: 计算扫描线与多边形各边的交点
- b、排序: 把所有交点按递增顺序进行排序

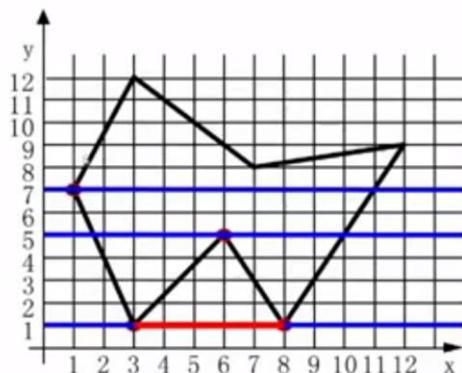


- c、交点配对: 第一个与第二个, 第三个与第四个
- d、区间填色: 把这些相交区间内的像素置成不同于背景色的填充色

为什么要进行排序?

以上图为例, 按照顺序扫描, 那应该先扫描  $P_1P_2$ , 但这条边明显和图中的扫描线没有交点, 再用  $P_2P_3$  和扫描线相交得到一点, 但是很明显的, 剩下的相交点其实都比  $P_2P_3$  交点小, 我们遵循从小到大的顺序进行填色, 则必须根据所有的交点的  $x$  值来重新排序交点, 再按照排序后的顺序来进行填色。因此 (3) b 是必要的

当扫描线与多边形顶点相交时, 交点的取舍问题 (交点的个数应保证为偶数个)

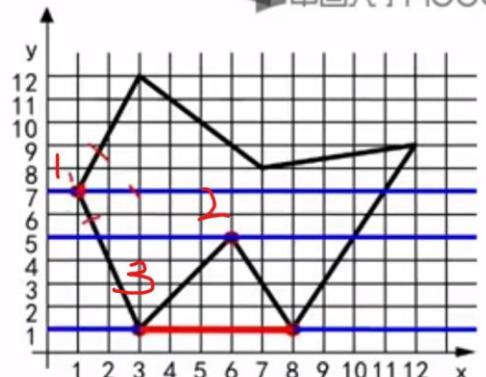


当出现上图中三条扫描线的情况时, 即两条边线的和扫描线的交点相同时, 我们要对如何取舍交点做出规范。

## 解决方案：

(1) 若共享顶点的两条边分别落在扫描线的两边，交点只算一个

(2) 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



检查共享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值的个数来决定交点数

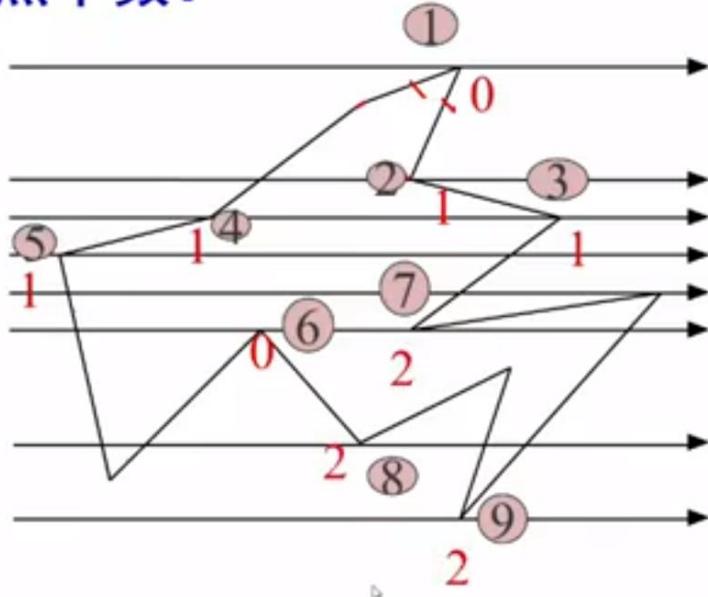
对其规定如上，我们分别举几个例子，上图中的交点 1，很明显的共享顶点的两条边分别在扫描线的两边，因此这个交点算一个。

而上图的交点 2，其共享端点的两条边都在扫描线的一边，需要再做判断，根据更具体的判断条件，我们得知，这两条边的另外一个端点的 y 值都比共享端点的 y 值小，因此我们定交点为 0 个

上图中的交点 3，其共享端点的其他两个端点的 y 值比交点大，因此交点就是 2 个。

我们也可以这么理解：交点就是两条边线的端点的 y 值比扫描线的 y 值大的个数。显然第一种情况，y 值大的端点有 1 个，交点就是 1。第二种情况较大的端点数量为 0，而第三种情况则是 2 个，这样似乎更好记忆。

## 举例计算交点个数：



为了计算每条扫描线与多边形各边的交点，最简单的方法是把多边形的所有边放在一个表中。在处理每条扫描线时，按顺序从表中取出所有的边，分别与扫描线求交

这个算法效率低，为什么？

关键问题是求交！而求交是很可怕的，求交的计算量是非常大的

## X 扫描线算法的改进

### 2、多边形的扫描转换算法的改进

扫描转换算法重要意义是提出了图形学里两个重要的思想：

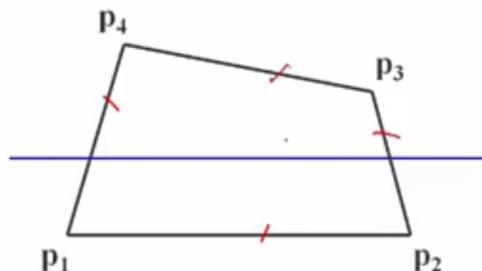
(1) 扫描线：当处理图形图像时按一条条扫描线处理

(2) 增量的思想

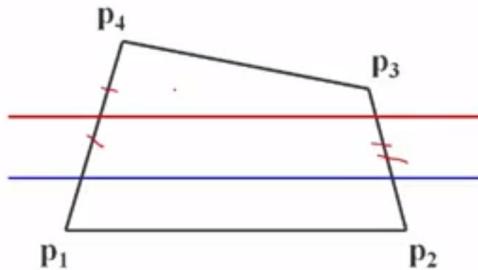
求交点的时候能不能也采取增量的方法？每条扫描线的y值都知道，关键是求x的值。x是什么？

### 可以从三方面考虑加以改进：

(1) 在处理一条扫描线时，仅对与它相交的多边形的边（有效边）进行求交运算

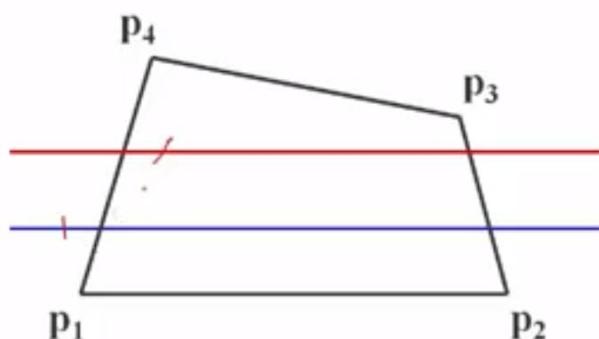


以该图为例，我们在处理上图的扫描线时，只对 P1P4 和 P2P3 做运算，其他的不做处理，这样减少了计算量。

(2) 考虑扫描线的连贯性

即当前扫描线与各边的交点顺序与下一条扫描线与各边的交点顺序很可能相同或非常相似

对此做出解释，图中蓝色的扫描线与  $P_1P_4$  和  $P_2P_3$  相交，而与之相邻的红色的扫描线也很有可能与  $P_1P_4$  和  $P_2P_3$  相交。

(3) 最后考虑多边形的连贯性

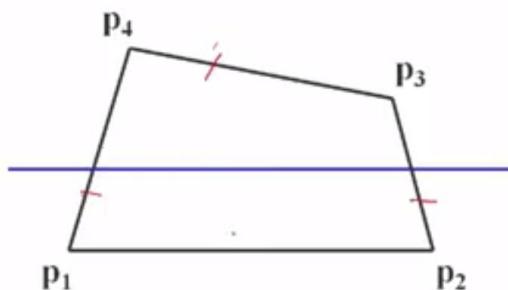
即当某条边与当前扫描线相交时，它很可能也与下一条扫描线相交

对此做出解释，当  $P_1P_4$  和  $P_2P_3$  与红扫描线相交时，那这两条边线也很可能与蓝色的扫描线相交。

## 2、多边形的扫描转换算法的改进

为了避免求交运算，需要引进一套特殊的数据结构

(1) 活性边表(AET)：把与当前扫描线相交的边称为活性边，并把它们按与扫描线交点x坐标递增的顺序存放在一个链表中。



以上图为例，P1P4 和 P2P3 为活性边，而 P3P4P1P2 为非活性边

(2) 结点内容 (一个结点在数据结构里可用结构来表示)

x: 当前扫描线与边的交点坐标  
 $\Delta x$ : 从当前扫描线到下一条扫描线间x的增量  
 $y_{max}$ : 该边所交的最高扫描线的坐标值 $y_{max}$

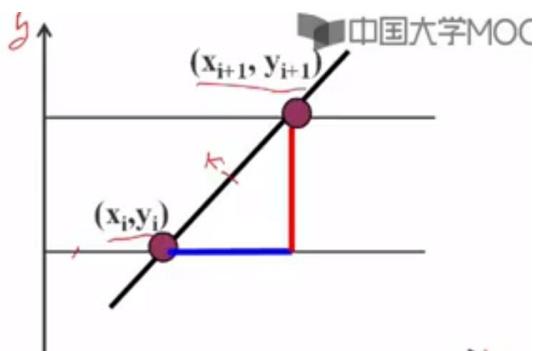
x	$\Delta x$	$y_{max}$	next
---	------------	-----------	------

随着扫描线的移动，扫描线与多边形的交点和上一次交点相关：

设边的直线斜率为k

$$k = \frac{\Delta y}{\Delta x} = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} = 1$$

$$x_{i+1} - x_i = \frac{1}{k} \rightarrow x_{i+1} = x_i + \frac{1}{k}$$



$y_{i+1}$  和  $y_i$  这两条相邻的扫描线的 y 值差为 1，则  $k=1/X_{i+1}-X_i$ ，于是有上图的下左式，再推导得下右式。

$$\text{即: } \Delta x = \frac{1}{k}$$

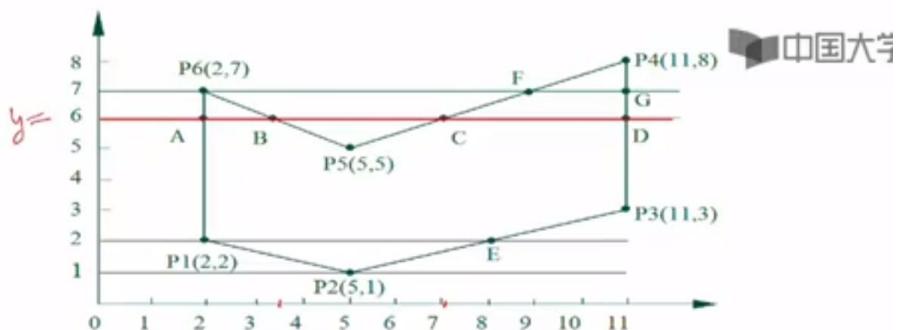
另外, 需要知道一条边何时不再与下一条扫描线相交, 以便及时把它从有效边表中删除出去, 避免下一步进行无谓的计算

X	$\Delta x$	$y_{\max}$	next
---	------------	------------	------

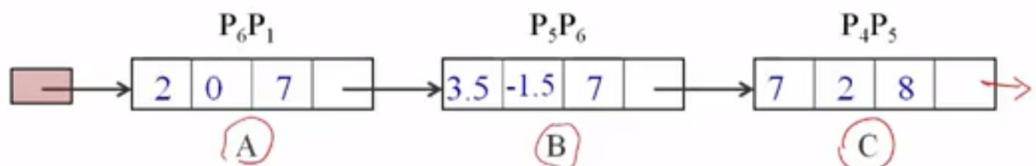
其中x为当前扫描线与边的交点,  $y_{\max}$ 是边所在的最大扫描线值, 通过它可知何时才能“**抛弃**”该边,  $\Delta x$ 表示从当前扫描线到下一条扫描线之间的x增量即斜率的倒数。next为指向下一条边的指针

也就是说x的增量dx就是斜率的倒数。

我们要不断的更新表, 将非活性边从表中去掉, 而判断条件则是扫描线的Y值是否大于边的最大扫描线值, 如果扫描线的Y值大于边的最大扫描线值, 那就将其从表中去掉。



X	$\Delta x$	$y_{\max}$	next
---	------------	------------	------

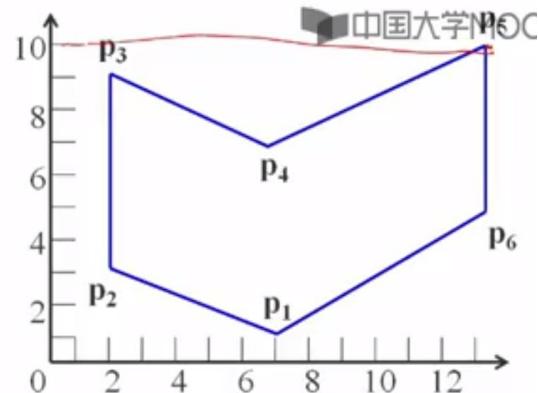


## 2、多边形的扫描转换算法的改进

为了方便活性边表的建立与更新，需构造一个新边表（NET），用来存放多边形的边的信息，分为4个步骤：

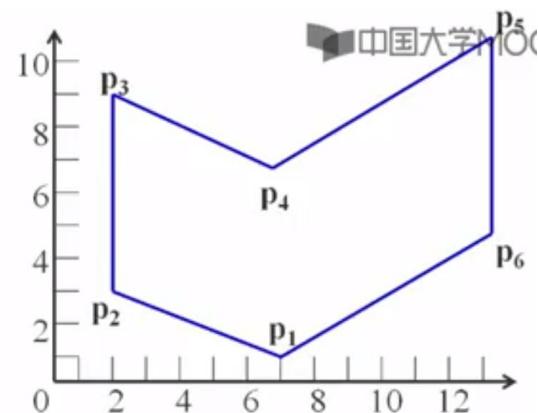
- (1) 首先构造一个纵向链表，链表的长度为多边形所占有的最大扫描线数，链表的每个结点，称为一个吊桶，对应多边形覆盖的每一条扫描线

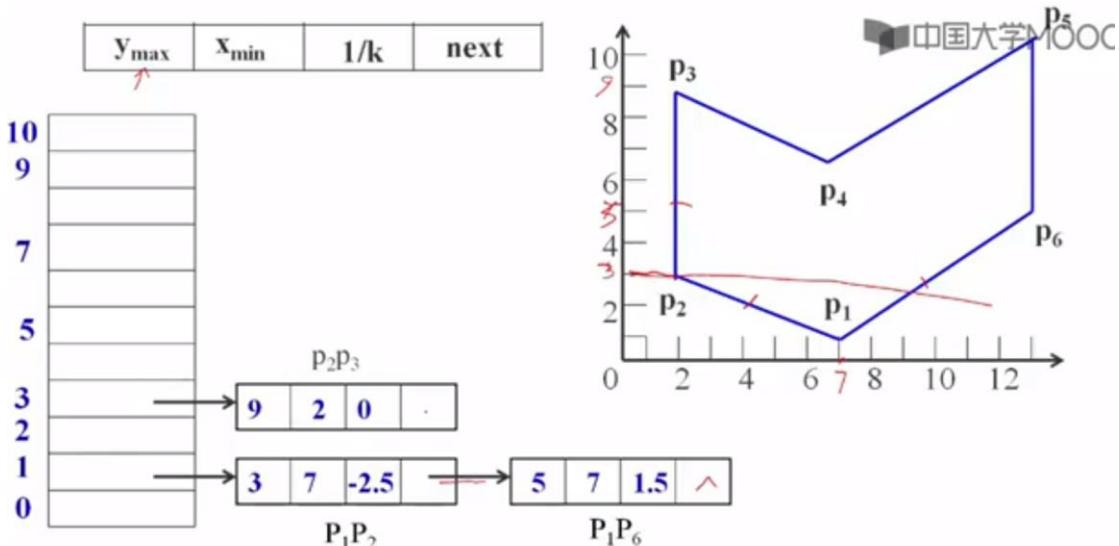
(1) 首先构造一个纵向链表，链表的长度为多边形所占有的最大扫描线数



(2) NET挂在与该边低端y值相同的扫描线桶中。也就是说，存放在该扫描线第一次出现的边

- 该边的  $y_{max}$
- 该边较低点的x坐标值  $x_{min}$
- 该边的斜率  $1/k \Delta x$
- 指向下一条具有相同较低端y坐标的边的指针

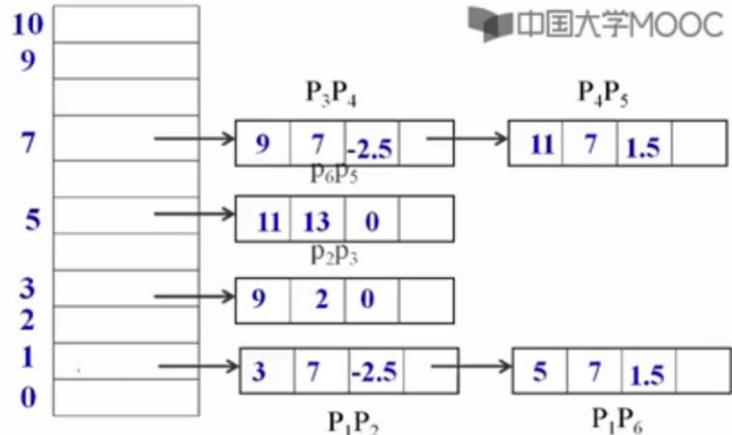




所谓 Xmin 就是边的最低点的 x 值，而非 X 的最小值，注意区分。

从右边这个NET表里就知道多边形是从哪里开始的

在这个表里只有 1、3、5、7 处有边，从  $y=1$  开始做，而在 1 这条线上有两条边进来了，然后就把这两条边放进活性边表来处理



实际上，我们所做的表中，只有扫描了第一次出现的边的扫描线才被记录，如上图的示例，扫描线 2 并没有扫描出比扫描线 1 更新的线，所以不填值，扫描线 3 只新扫描了一条边，则只写这条新边，旧的 P1P6 不写入。

每做一次新的扫描线时，要对已有的边进行 三个处理：

1、是否被去除掉；

2、如果不被去除，第二就要对它的数据进行更新。所谓更新数据就是要更新它的 x 值，即： $x+1/k$

3、看有没有新的边进来，新的边在 NET 里，可以插入排序插进来。

这个算法过程从来没有求交，这套数据结构使得你不用求交点！避免了求交运算。

做一次新的扫描线时，

(1) 先看现有的边是否被去掉---即判断扫描线是否大于该边的最大 Y 值，如果大于就去

掉，小于则保留。

- (2) 对于被保留的边，要更新他的  $x$  值，就是要增加增量  $1/x$
- (3) 再看有没有新的边被扫描，如果有，则将他的信息插入表 (NET) 内

## 多边形扫描转换算法小结



扫描线法可以实现已知任意多边形域边界的填充。该填充算法是按扫描线的顺序，计算扫描线与待填充区域的相交区间，再用要求的颜色显示这些区间的像素，即完成填充工作

为了提高算法效率：

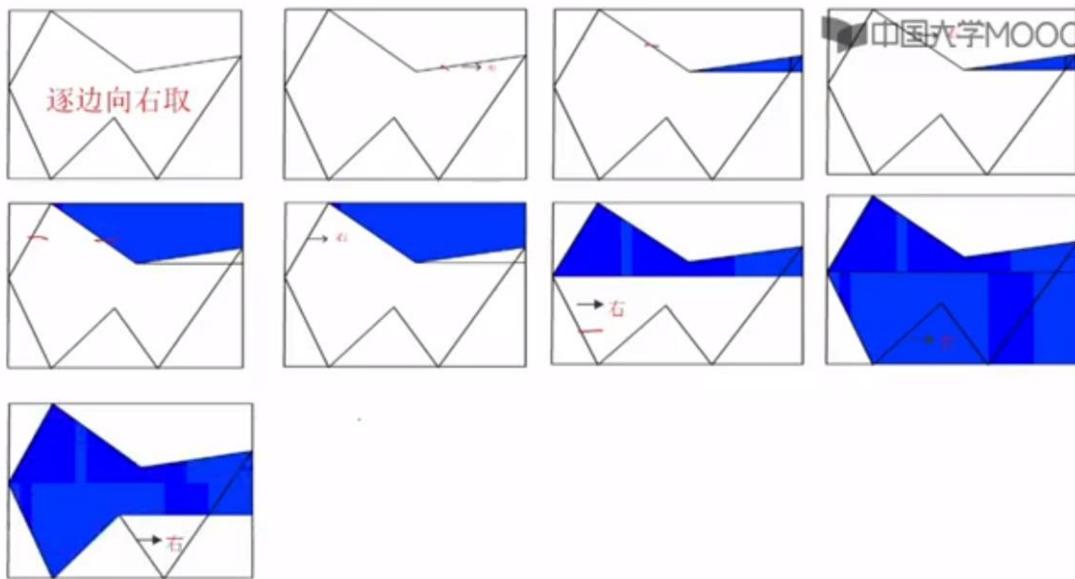
- (1) 增量的思想
- (2) 连贯性思想
- (3) 构建了一套特殊的数据结构

这里区间的端点通过计算扫描线与多边形边界  
的交点获得。所以待填充区域的边界线必须事  
先知道，因此它的缺点是无法实现对未知边界  
的区域填充

## 3、边缘填充算法



其基本思想是按任意顺序处理多边形的每条边。在处理每  
条边时，首先求出该边与扫描线的交点，然后将每一条扫  
描线上交点右方的所有像素取补。多边形的所有边处理完  
毕之后，填充即完成



算法简单，但对于复杂图型，每一象素可能被访问多次。  
 输入和输出量比有效边算法大得多。

为了减少边缘填充法访问像素的次数，可采用栅栏填充算法

## 4、栅栏填充算法

**栅栏**指的是一条过多边形顶点且与扫描线垂直的直线。它把多边形分为两半。在处理每条边与扫描线的交点时，将交点与栅栏之间的像素取补

## 5、边界标志算法

帧缓冲器中对多边形的每条边进行直线扫描转换，亦即对多边形边界所经过的象素打上标志

然后再采用和扫描线算法类似的方法将位于多边形内的各个区段着上所需颜色

由于边界标志算法不必建立维护边表以及对它进行排序，所以边界标志算法更适合硬件实现，这时它的执行速度比有序边表算法快一至两个数量级。

## 2、区域填充

### 二、区域填充

中国大学MOOC

**区域**---指已经表示成点阵形式的填充图形，是象素的集合

**区域填充**是指将区域内的一点(常称**种子点**)赋予给定颜色，然后将这种颜色扩展到整个区域内的过程。



区域可采用**内点**表示和**边界**表示两种表示形式

中国大学MOOC



**内点表示**: 枚举出区域内部的所有像素，内部的所有像素着同一个颜色，边界像素着与内部像素不同的颜色

**边界表示**: 枚举出边界上的所有像素，边界上的所有像素着同一个颜色，内部像素着与边界像素不同的颜色

区域填充算法要求**区域是连通的**，因为只有在连通区域中，才可能将种子点的颜色扩展到区域内的其它点。

区域可分为**4向连通区域**和**8向连通区域**



四个方向运动

八个方向运动

四连通区域

八连通区域

**4向连通**区域指的是从区域上一点出发，可通过四个方向，即上、下、左、右移动的组合，在不越出区域的前提下，到达区域内的任意象素

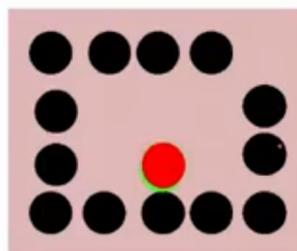
**8向连通**区域指的是从区域内每一象素出发，可通过八个方向，即上、下、左、右、左上、右上、左下、右下这八个方向的移动的组合来到达

### 简单四连通种子填充算法（区域填充递归算法） 中国大学MOOC

种子填充算法的原理是：假设在多边形区域内部有一像素已知，由此出发找到区域内的所有像素，用一定的颜色或灰度来填充

假设区域采用边界定义，即区域边界上所有像素均具有某个特定值，区域内部所有像素均不取这一特定值，而边界外的像素则可具有与边界相同的值

考虑区域的四向连通，即从区域上一点出发，可通过四个方向，即上、下、左、右移动的组合，在不越出区域的前提下，到达区域内的任意像素。

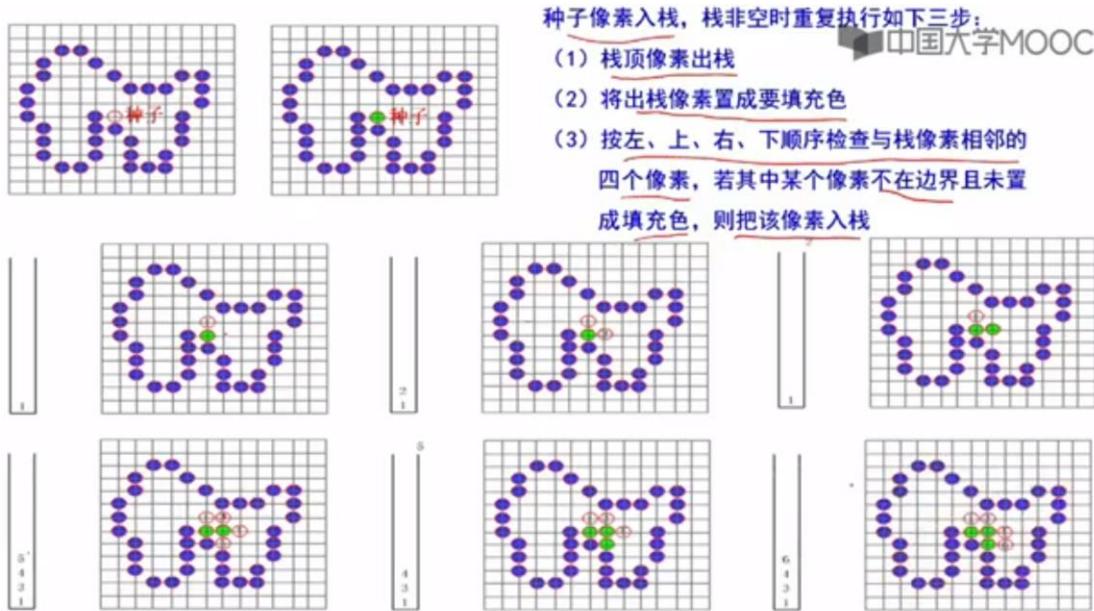


## 使用栈结构来实现简单的种子填充算法

算法原理如下：

种子像素入栈，当栈非空时重复执行如下三步操作：

- (1) 栈顶像素出栈
- (2) 将出栈像素置成要填充色
- (3) 按左、上、右、下顺序检查与栈像素相邻的四个像素，若其中某个像素不在边界且未置成填充色，则把该像素入栈



我们以上图为例，讲解这个算法：

图1中的种子像素先入栈，将其填充颜色。然后检查左边像素，为边界不入栈；再检查上边像素，不是边界入栈；检查右边入栈；检查下边入栈。然后我们将栈顶的像素作为种子像素，再按照上面的步骤，将其填充颜色后，在进行检查，最终实现整个区域的填色。

- (1) 有些像素会入栈多次，降低算法效率；栈结构占空间
- (2) 递归执行，算法简单，但效率不高。区域内每一像素都引进一次递归，进/出栈，费时费内存
- (3) 改进算法，减少递归次数，提高效率

### 可以采用区域填充的扫描线算法

## 三、多边形的扫描转换与区域填充算法小结

- **基本思想不同**
  - 多边形扫描转换是指将多边形的顶点表示转化为点阵表示
  - 区域填充只改变区域的填充颜色，不改变区域表示方法
- **基本条件不同**
  - 在区域填充算法中，要求给定区域内一点作为种子点，然后从这一点根据连通性将新的颜色扩散到整个区域
  - 扫描转换多边形是从多边形的边界(顶点)信息出发，利用多种形式的连贯性进行填充的

扫描转换区域填充的核心是知道多边形的边界，要得到多边形内部的像素集，有多种方法。其中扫描线算法是利用一套特殊的数据结构，避免求交，然后一条条扫描线确定

区域填充条件更强一些，不但知道边界，而且还知道区域内的一点，可以利用四连通或八连通区域不断往外扩展

填充一个定义的区域的选择包括：

- 选择实区域颜色或图案填充方式
- 选择某种颜色和图案

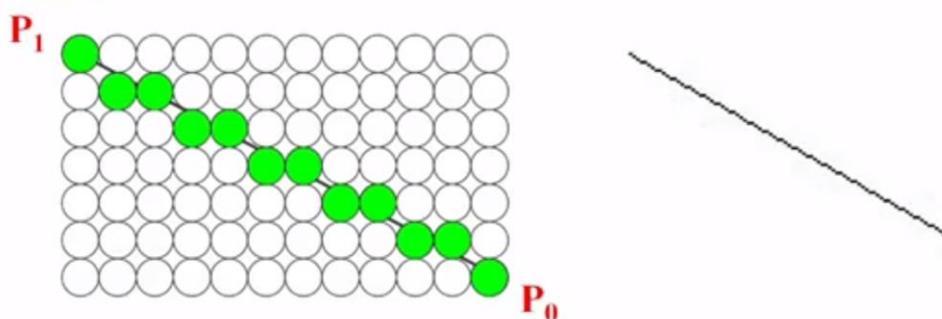
这些填充选择可应用于多边形区域或用曲线边界定义的区域；此外，区域可用多种画笔、颜色和透明度参数来绘制

### 三：反走样

对直线、圆及椭圆这些最基本元素的生成速度和显示质量的改进，在图形处理系统中具有重要的应用价值

但它们生成的线条具有明显的“锯齿形”即会发生走样（Liasing）现象

#### 一、走样



“锯齿”是“走样”（aliasing）的一种形式。而走样是光栅显示的一种固有性质。产生走样现象的原因是像素本质上是离散的

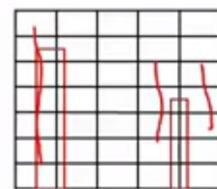
走样是数字化的必然产物。

## 走样现象：

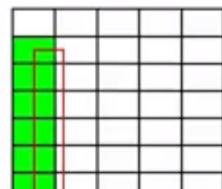
一是光栅图形产生的阶梯形（锯齿形）

二是图形中包含相对微小的物体时，  
这些物体在静态图形中容易被丢弃或忽略

小物体由于“走样”而消失

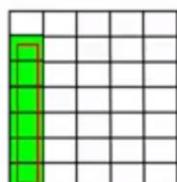


(a) 需显示的矩形

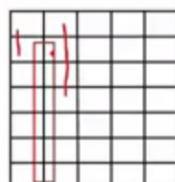


(b) 显示结果

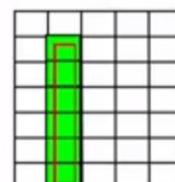
在动画序列中时隐时现，产生闪烁



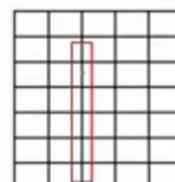
(a) 显示



(b) 不显示



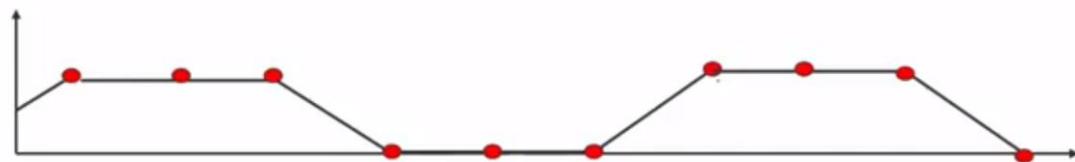
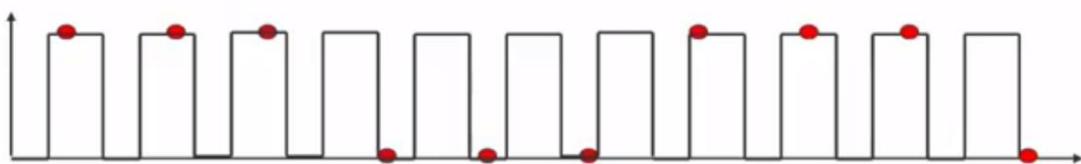
(c) 显示



(d) 不显示

矩形从左向右移动，当其覆盖某些像素中心时，矩形被显示出来，当没有覆盖像素中心时，矩形不被显示

简单地说，如果对一个快速变化的信号采样频率过低，所得样本表示的会是低频变化的信号：原始信号的频率看起来被较低的“走样”频率所代替



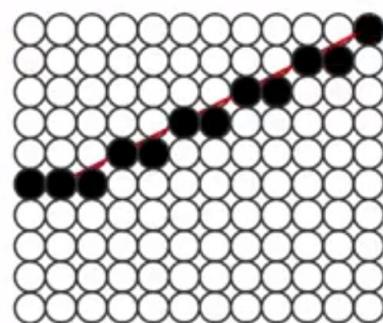
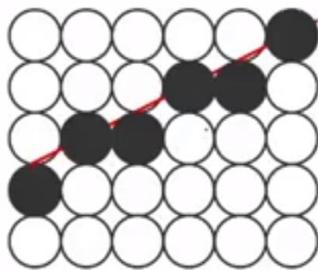
## 二、反走样技术

如何降低由于采样不足而产生的走样现象呢？

用于减少或消除走样效果的技术，称为反走样  
(Antialiasing) 技术

由于图形的走样现象对图形的质量有很大影响，几乎所有图形处理系统都要对基本图形进行反走样处理

采用分辨率更高的显示设备，对解决走样现象有所帮助，因为可以使锯齿相对物体会更小一些



该反走样方法是以4倍的存储器代价和扫描转换时间获得的

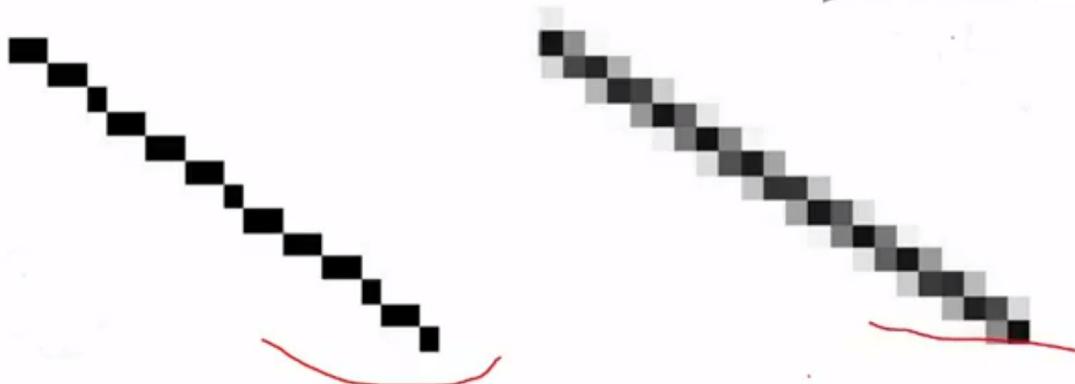
为了稳定屏幕上的图象，电子枪至少要 $1/24$ 秒时间轰击屏幕上所有像素一次，如果像素提高一倍，电子枪就要快4倍！

反走样技术涉及到某种形式的“模糊”来产生更平滑的图像

对于在白色背景中的黑色矩形，通过在矩形的边界附近掺入一些灰色像素，可以柔化从黑到白的尖锐变化

从远处观察这幅图像时，人眼能够把这些缓和变化的暗影融合在一起，从而看到了更加平滑的边界

中国大学MOOC



左边直线未经过模糊处理，右边直线经过模糊处理，很明显的，右边比左边的要光滑很多

## 两种反走样方法：

1、非加权区域采样方法

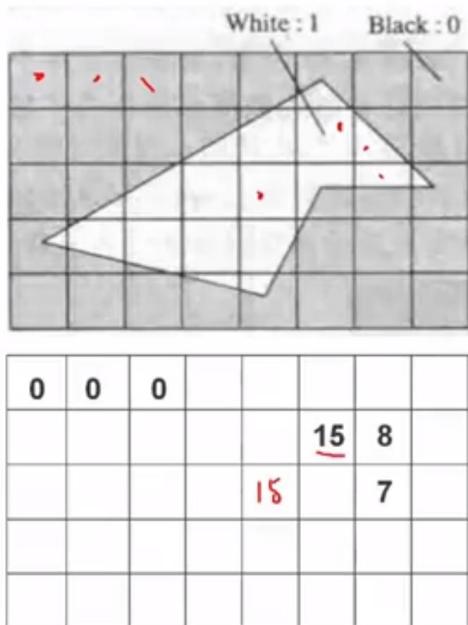
2、加权区域采样方法

## 1、非加权区域采样方法

### 1、非加权区域采样方法

中国大学MOOC

根据物体的覆盖率（coverage）计算像素的颜色。覆盖率是指某个像素区域被物体覆盖的比例



把这个多边形放在方格线中，其中每个正方形的中心对应显示器上一个像素中心。被多边形覆盖了一半的像素的亮度值赋为1/2，覆盖三分之一的像素赋值为1/3；以此类推

如果帧缓冲区的每个像素有4个比特位，那么0表示黑色，。。。15表示白色

### 非加权区域采样方法有两个缺点：

中国大学MOOC

- 1、象素的亮度与相交区域的面积成正比，而与相交区域落在象素内的位置无关，这仍然会导致锯齿效应
- 2、直线条上沿理想直线方向的相邻两个象素有时会有较大的灰度差

每个像素的权值是一样的，这是它的主要缺点。所以也称非加权区域采样方法

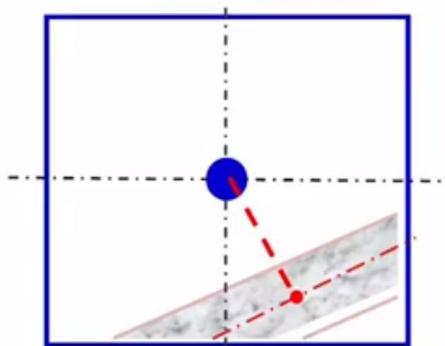
关于1, 做出解释：在该方法中，一个像素的亮度是通过多边形覆盖该像素的面积来决定的，而我们认为，如果多边形覆盖了像素的中心区域，那应该其亮度应该更高，也就是说我们认为像素的不同区域的权重应该不同。

## 2、加权区域采样方法

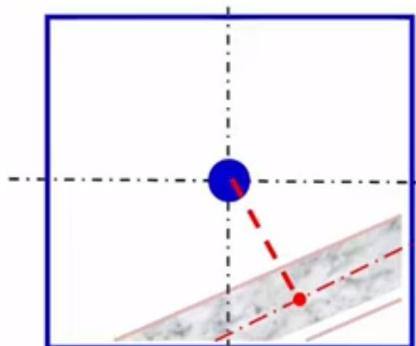
### 2、加权区域采样方法



这种方法更符合人视觉系统对图像信息的处理方式，反走样效果更好



将直线段看作是具有一定宽度的狭长矩形；当直线段与像素有交时，根据相交区域与像素中心的距离来决定其对像素亮度的贡献



直线段对一个像素亮度的贡献正比于相交区域与像素中心的距离

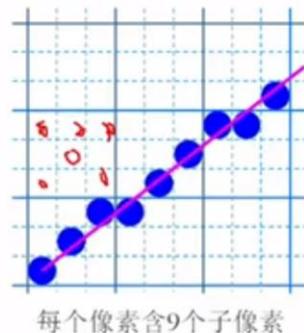
设置相交区域面积与像素中心距离的权函数（高斯函数）反映相交面积对整个像素亮度的贡献大小

利用权函数积分求相交区域面积，用它乘以像素可设置的最大亮度值，即可得到该像素实际显示的亮度值

## 可采用离散计算方法

将一个像素划分为  $n = 3 \times 3$  个子像素，加权表可以取作：

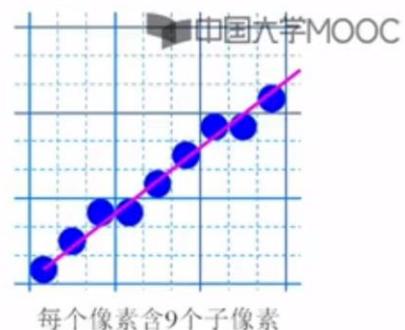
$$\begin{bmatrix} w1 & w2 & w3 \\ w4 & w5 & w6 \\ w7 & w8 & w9 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$



**加权方案：**中心子像素的加权是角子像素的4倍，是其它像素的2倍，对九个子像素的每个网格所计算出的亮度进行平均

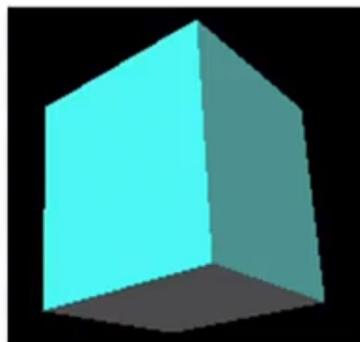
将一个像素分为九个子像素，中央像素的加权为四个角的四倍，是其余像素的两倍，如上矩阵所示

$$\begin{bmatrix} w1 & w2 & w3 \\ w4 & w5 & w6 \\ w7 & w8 & w9 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

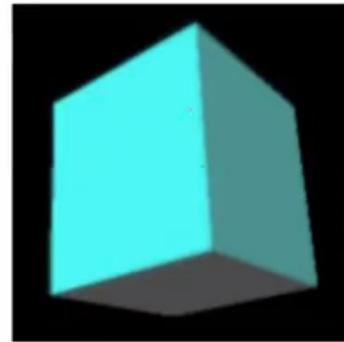


- 然后求出所有中心落于直线段内的子像素。
- 最后计算所有这些子像素对原像素亮度贡献之和

反走样是图形学中的一个根本问题，不可能避免；是图形学中的一个永恒问题



原图

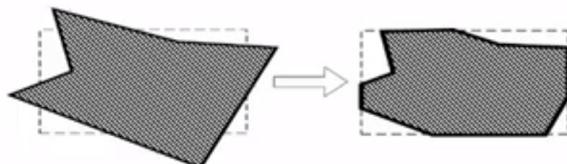


反走样图

## 四：直线裁剪算法

### 一、裁剪

使用计算机处理图形信息时，计算机内部存储的图形往往比较大，而屏幕显示的只是图形的一部分。



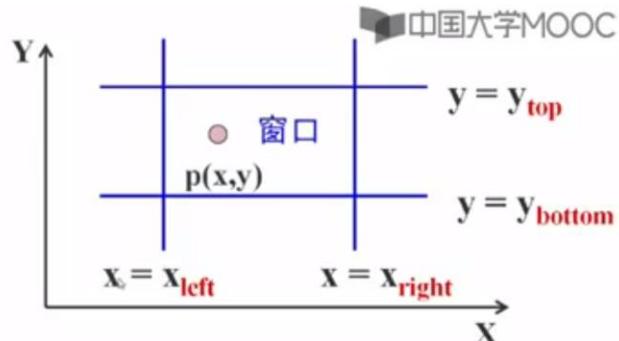
因此需要确定图形哪些部分落在显示区之内，哪些落在显示区之外。这个选择的过程就称为**裁剪**。

最简单的裁剪方法是把各种图形扫描转换为点之后，再判断点是否在窗口内。

#### 1、点的裁剪

对于任意一点P(x, y)，若满足下列两对不等式：

$$\begin{cases} x_{left} \leq x \leq x_{right} \\ y_{bottom} \leq y \leq y_{top} \end{cases}$$



则点P在矩形窗口内；否则，点P在矩形窗口之外

判断图形中每个点是否在窗口内，太费时，一般不可取

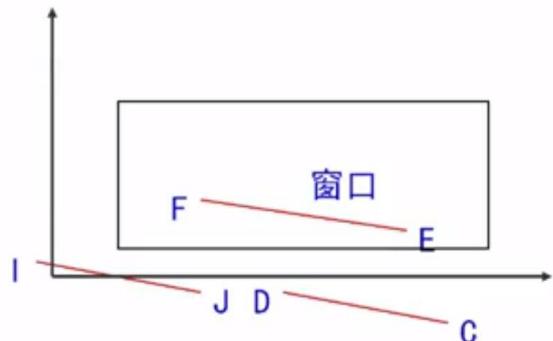
## 2、直线段的裁剪

中国大学MOOC

直线段裁剪算法复杂图形裁剪的基础

直线段和剪裁窗口的可能关系：

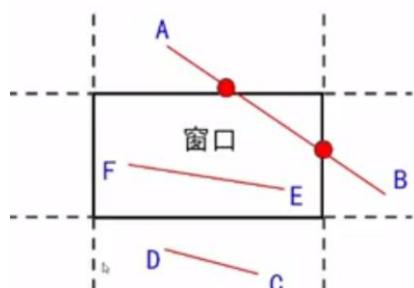
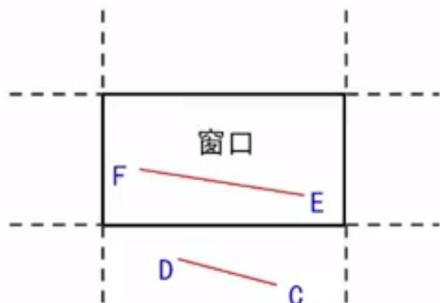
- 完全落在窗口内
- 完全落在窗口外
- 与窗口边界相交



要裁剪一条直线段，首先要判断：

中国大学MOOC

- (1) 它是否完全落在裁剪窗口内？
- (2) 它是否完全在窗口外？
- (3) 如果不满足以上两个条件，则计算它与一个或多个裁剪边界的交点



如图所示，直线与边界的交点称为实交点，而将直线的延长线和边界的延长线的交点称为虚交点。

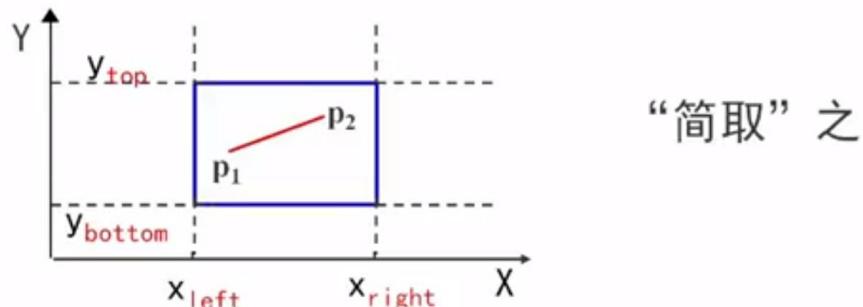
常用的裁剪算法有三种，即Cohen-Sutherland、中点分割法和Liang-Barsky裁剪算法

## 1、Cohen-Sutherland 算法

### 1、Cohen-Sutherland算法

本算法又称为编码裁剪算法，算法的基本思想是对每条直线段分三种情况处理：

(1) 若点 $p_1$ 和 $p_2$ 完全在裁剪窗口内



所谓“简取”就是将其完全保留该线段，因其完全在窗口内

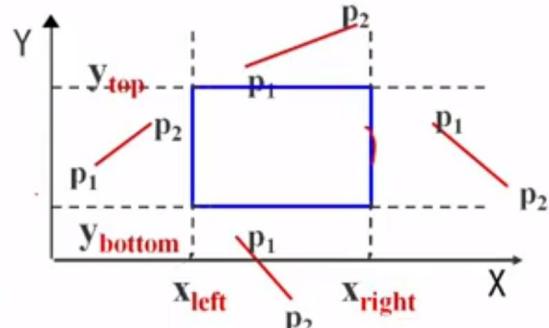
(2) 若点 $p_1(x_1, y_1)$ 和 $p_2(x_2, y_2)$ 均在窗口外，且满足下列四个条件之一：

$$x_1 < x_{left} \text{ 且 } x_2 < x_{left}$$

$$x_1 > x_{right} \text{ 且 } x_2 > x_{right}$$

$$y_1 < y_{bottom} \text{ 且 } y_2 < y_{bottom}$$

$$y_1 > y_{top} \text{ 且 } y_2 > y_{top}$$

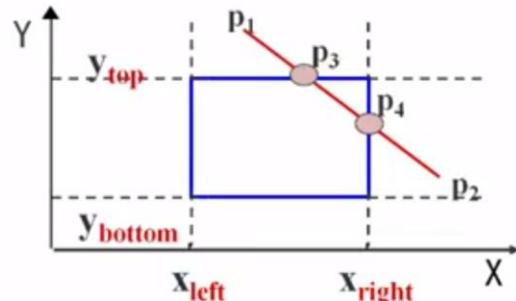


对这四种类型的直线，“简弃”之

所谓“简弃”就是将线段直接抛弃，因其完全在窗口外

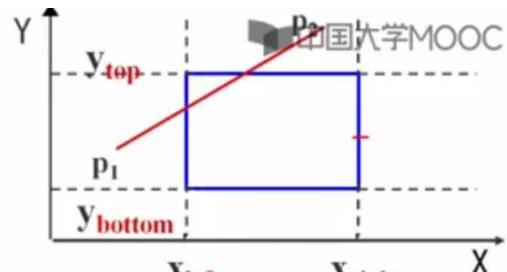
### (3) 如果直线段既不满足“简取”的条件，也不满足“简弃”的条件？

需要对直线段按交点进行分段，分段后判断直线是“简取”还是“简弃”。



我们将上图的直线分为三段，其中 P3 为上边界交点，P4 为右边界交点，我们再对 P1P3, P3P4, P4P2 分开进行判断

每条线段的端点都赋以四位二进制码  $D_3D_2D_1D_0$ ，编码规则如下：



- 若  $x < x_{left}$ , 则  $D_0 = 1$ , 否则  $D_0 = 0$
- 若  $x > x_{right}$ , 则  $D_1 = 1$ , 否则  $D_1 = 0$
- 若  $y < y_{bottom}$ , 则  $D_2 = 1$ , 否则  $D_2 = 0$
- 若  $y > y_{top}$ , 则  $D_3 = 1$ , 否则  $D_3 = 0$

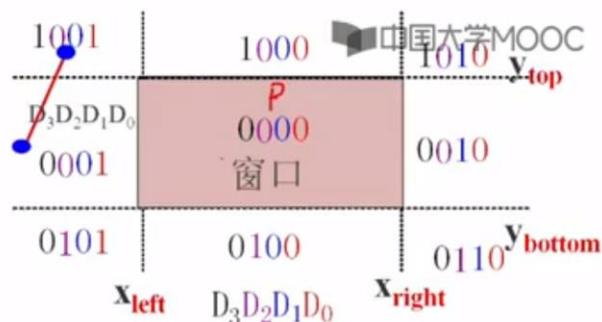
窗口及其延长线所构成了9个区域。根据该编码规则：

$D_0$  对应窗口左边界

$D_1$  对应窗口右边界

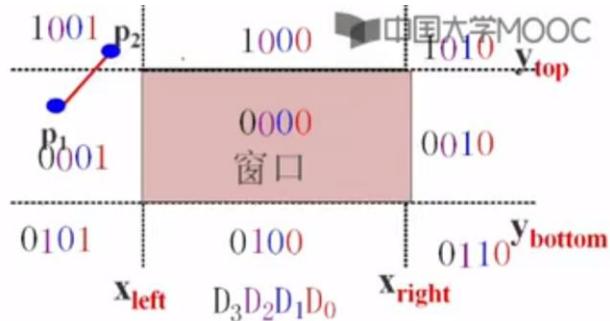
$D_2$  对应窗口下边界

$D_3$  对应窗口上边界



裁剪一条线段时，先求出端点 $p_1$ 和 $p_2$ 的编码 $code_1$ 和 $code_2$

然后进行二进制“或”运算和“与”运算



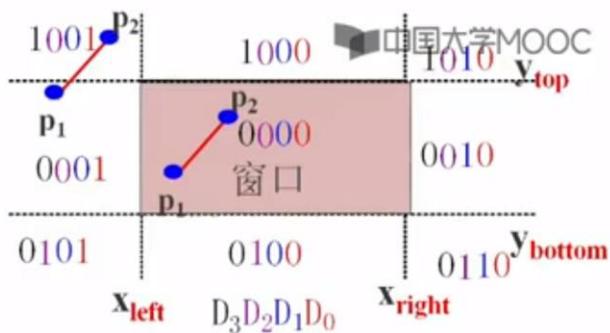
## 二进制运算



运算符	名称	例子	运算功能
$\sim$	位反	$\sim b$	求b的位反
	与运算	$b \& c$	$b$ 和 $c$ 位与
	或运算	$b   c$	$b$ 和 $c$ 位或
$\wedge$	异或运算	$b \wedge c$	$B$ 和 $c$ 位异或

(1) 若 $code_1 | code_2 = 0$ , 对直线段应简取之

$$\begin{array}{r} \text{或} \\ \hline 0000 \\ 0000 \\ \hline 0000 \end{array}$$



(2) 若 $code_1 \& code_2 \neq 0$ , 对直线段可简弃之

$$\begin{array}{r} \text{与} \\ \hline 1001 \\ 0001 \\ \hline 0001 \end{array}$$

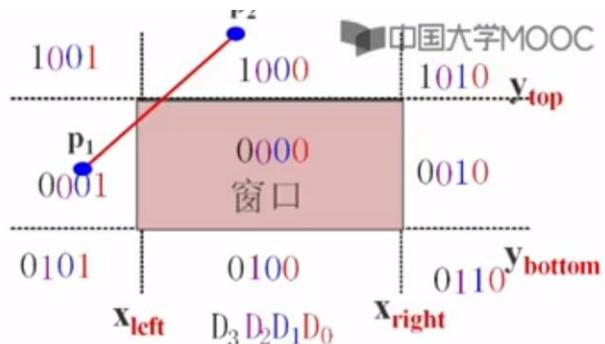
我们来分析一下上图的两种情况:

- (1) 两个点取或操作, 如果结果为 0, 也就是说-----两个点对应的编码的各位都为 0, 也就是两个点都在边界内, 那我们可以完全取入该线段。
- (2) 两个点的编码做与操作, 如果结果不为 0, 也就是说-----两个点对应的编码各位对应的起码有一对都为 1, 也就是说, 他们最起码完全在一条边界外, 也就是

我们在前面讨论的简弃的情况，完全舍弃这条线段。

若上述两条件均不成立

$$\begin{array}{c} \text{或} \\ \begin{array}{r} 0001 \\ 1000 \end{array} & \text{与} & \begin{array}{r} 0001 \\ 1000 \end{array} \\ \hline \underline{\underline{1001}} & & \underline{\underline{0000}} \end{array}$$

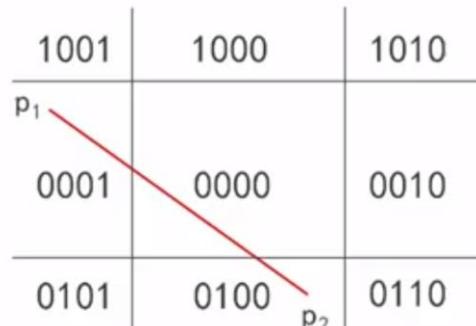


则需求出直线段与窗口边界的交点在交点处把线段一分为二

下面根据该算法步骤来裁剪如图所示的直线段P<sub>1</sub>P<sub>2</sub> 中国大学MOOC

首先对P<sub>1</sub>P<sub>2</sub>进行编码

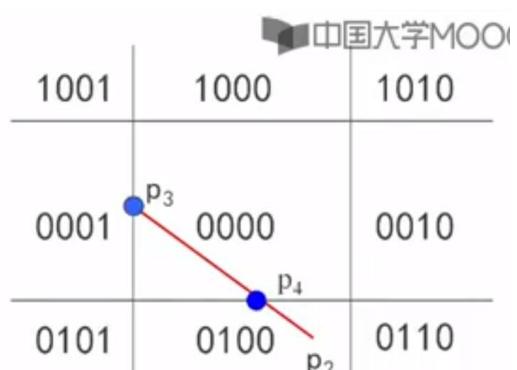
$$\begin{array}{c} \text{或} \\ \begin{array}{r} 0001 \\ 0100 \end{array} & \text{与} & \begin{array}{r} 0001 \\ 0100 \end{array} \\ \hline \underline{\underline{0101}} & & \underline{\underline{0000}} \end{array}$$



按左、右、下、上的顺序求出直线段与窗口左边界的交点为P<sub>3</sub>，P<sub>1</sub>P<sub>3</sub>必在窗口外，可简弃

对P<sub>2</sub>P<sub>3</sub>重复上述处理

$$\begin{array}{c} \text{或} \\ \begin{array}{r} 0000 \\ 0100 \end{array} & \text{与} & \begin{array}{r} 0000 \\ 0100 \end{array} \\ \hline \underline{\underline{0100}} & & \underline{\underline{0000}} \end{array}$$



剩下的直线段（P<sub>3</sub>P<sub>4</sub>）再进行进一步判断，code<sub>1</sub>|code<sub>2</sub>=0，全在窗口中，简取之。

Cohen-Sutherland算法用编码的方法实现了对直线段的裁剪

**编码的思想**在图形学中甚至在计算机科学里也是非常重要的，一个很简单的思想可以带来很了不起的作用。

比较适合两种情况：一是大部分线段完全可见；二是大部分线段完全不可见。

存在的问题：

$$\begin{array}{r} \text{或} \quad 0001 \\ 0100 \\ \hline 0101 \end{array} \qquad \begin{array}{r} \text{与} \quad 0001 \\ 0100 \\ \hline 0000 \end{array}$$

最坏情况下，被裁剪线段与窗口4条边计算交点，然后所得的裁剪结果却可能是全部舍弃。

1001	1000	中国大学MOOC 1010
0001	窗口 0000	0010
0101	0100	p <sub>2</sub> 0110

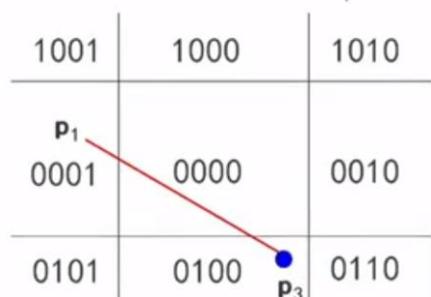
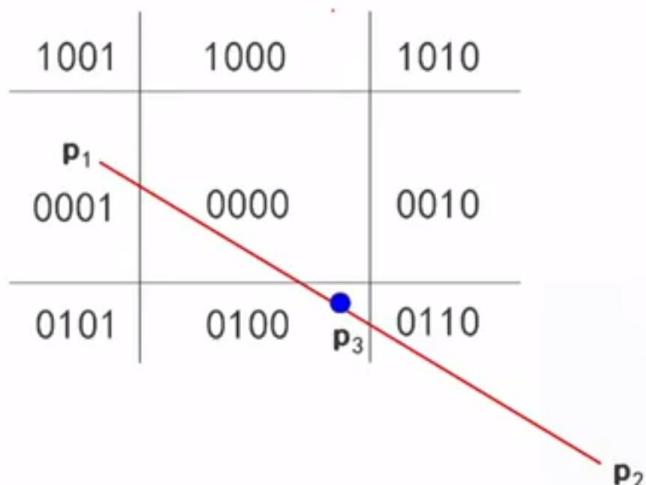
## 2、中点分割法

和上面讲到的Cohen-Sutherland算法一样，首先对直线段的端点进行编码。

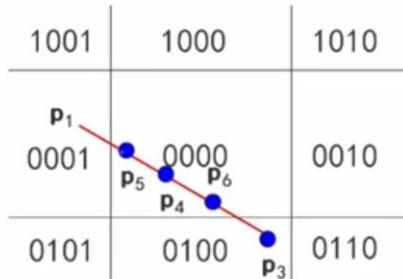
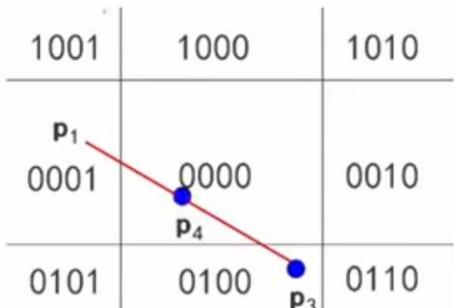
把线段和窗口的关系分成三种情况：

- 1、完全在窗口内
- 2、完全在窗口外
- 3、和窗口有交点

中点分割算法的**核心思想**是通过二分逼近来确定直线段与窗口的交点。

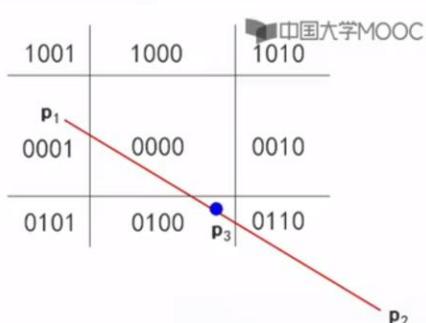


P3P2 完全在边界外，将其舍弃

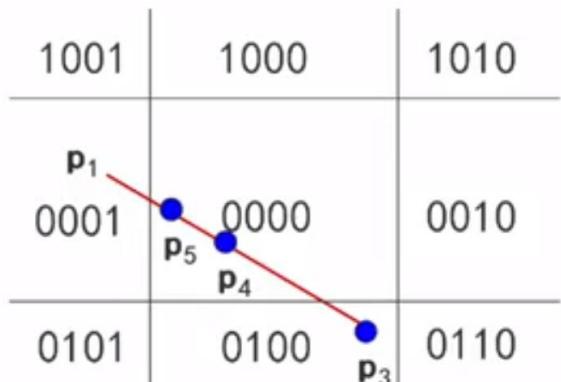


**注意：**

- 若中点不在窗口内，则把中点和离窗口边界最远点构成的线段丢掉，以线段上的另一点和该中点再构成线段求其中点



2、如中点在窗口内，则又以中点和最远点构成线段，并求其中点，直到中点与窗口边界的坐标值在规定的误差范围内相等



### 3、Liang-Barsky 算法

#### 三、Liang-Barsky算法

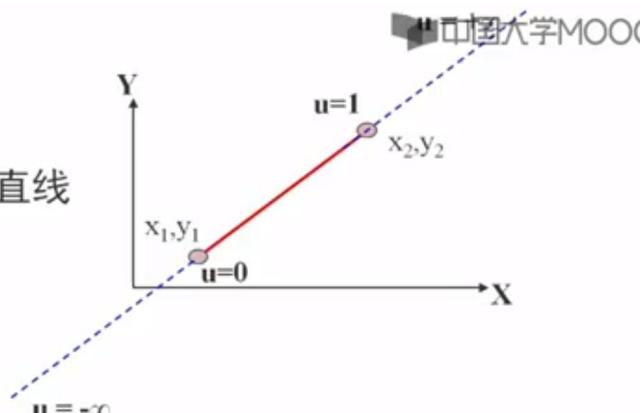
中国大学MOOC

在Cohen-Sutherland算法提出后，梁友栋和Barsky又针对标准矩形窗口提出了更快的Liang-Barsky直线段裁剪算法。

上世纪80年代，梁友栋先生提出了著名的Liang-Barsky算法，至今仍是计算机图形学中最经典的算法之一，也是写进国内外主流《计算机图形学》教科书里的唯一一个以中国人命名的算法。

#### 梁算法的主要思想：

(1) 用参数方程表示一条直线



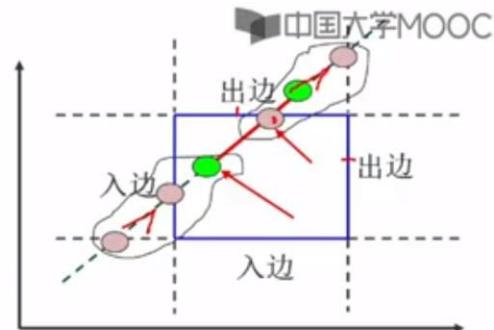
$$\begin{aligned} x &= x_1 + u \cdot (x_2 - x_1) = x_1 + \Delta x \cdot u \\ y &= y_1 + u \cdot (y_2 - y_1) = y_1 + \Delta y \cdot u \end{aligned} \quad 0 \leq u \leq 1$$

直线段的  $u$  取值在 0 到 1 之间，而如果是直线，则  $u$  取值为  $-\infty$  到  $+\infty$ 。

## 梁算法的主要思想：

(2) 把被裁剪的红色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：

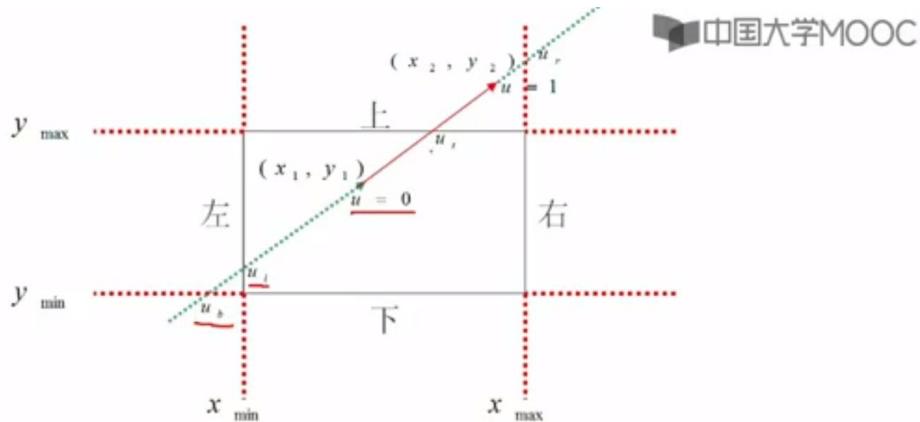
入边和出边



裁剪结果的线段起点是直线和两条入边的交点以及始端点三  
个点里最前面的一个点，即参数u最大的那个点；

裁剪线段的终点是和两条出边的交点以及端点最后面的一个  
点，取参数u最小的那个点。

我们可以看到有这些交点：两个入边交点，两个出边交点，还有两个直线段的端点。  
我们将其分为两组，其中第一组的包括两个入边交点和线段起点，第二组包括两个出边交点  
和线段终点。  
而对于我们要的结果，即裁剪完的线段，我们做如下的规定：将第一组内 u 最大的点作为起  
点，将第二组内最小的点作为终点。



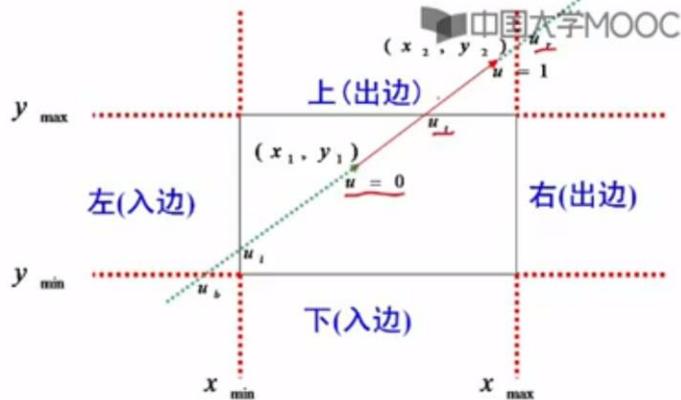
值得注意的是，当u从 $-\infty$ 到 $+\infty$ 遍历直线时，首先对裁剪窗口的两条边界直线（下边和左边）从外面向里面移动，再对裁剪窗口两条边界直线（上边和右边）从里面向外面移动。

遍历整条直线时，方向是u从 $-\infty$ 到 $+\infty$ ，即从入边进入内部，再从出边出去

如果用 $u_1$ ,  $u_2$ 分别表示线段 $(u_1 \leq u_2)$ 可见部分的开始和结束

$$u_1 = \max(0, u_l, u_b)$$

$$u_2 = \min(1, u_t, u_r)$$



$u_1$ 即为0 (线段起点),  $u_l$ ,  $u_b$ 入边交点的最大值。

$u_2$ 即为1 (线段终点),  $u_t$ ,  $u_r$ 出边交点的最小值。

Liang-Barsky算法的基本出发点是直线的参数方程

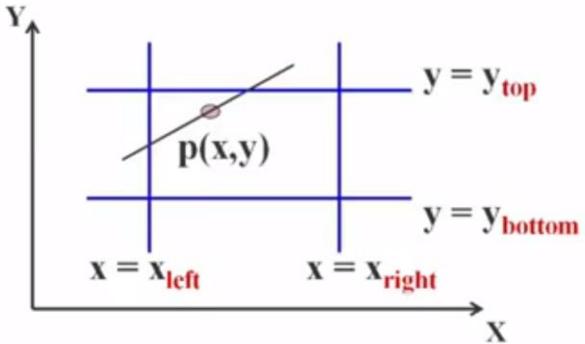
$$x = x_1 + u \cdot (x_2 - x_1)$$

$$y = y_1 + u \cdot (y_2 - y_1)$$

$$0 \leq u \leq 1$$

$$x_{left} \leq x_1 + u \cdot \Delta x \leq x_{right}$$

$$y_{bottom} \leq y_1 + u \cdot \Delta y \leq y_{top}$$



$$x_{left} \leq x_1 + u \cdot \Delta x \leq x_{right}$$

$$y_{bottom} \leq y_1 + u \cdot \Delta y \leq y_{top}$$

$$u \cdot (-\Delta x) \leq x_1 - x_{left}$$

$$u \cdot \Delta x \leq x_{right} - x_1$$

$$u \cdot (-\Delta y) \leq y_1 - y_{bottom}$$

$$u \cdot \Delta y \leq y_{top} - y_1$$

$$u \cdot (-\Delta x) \leq x_1 - x_{left}$$

$$u \cdot \Delta x \leq x_{right} - x_1$$

$$u \cdot (-\Delta y) \leq y_1 - y_{bottom}$$

$$u \cdot \Delta y \leq y_{top} - y_1$$

令：

$$\underline{p}_1 = -\underline{\Delta x} \quad q_1 = x_1 - x_{left}$$

$$\underline{p}_2 = \underline{\Delta x} \quad q_2 = x_{right} - x_1$$

$$\underline{p}_3 = -\underline{\Delta y} \quad q_3 = y_1 - y_{bottom}$$

$$\underline{p}_4 = \underline{\Delta y} \quad q_4 = y_{top} - y_1$$

于是有：  $\underline{u \cdot p_k} \leq q_k$  其中，  $k = 1, 2, 3, 4$

$$\underline{P}_1 = -\Delta x \quad q_1 = x_1 - x_{left}$$

$$\underline{P}_3 = -\Delta y \quad q_3 = y_1 - y_{bottom}$$

入边：左边和下边

出边：右边和上边

$$\underline{P}_2 = \Delta x \quad q_2 = x_{right} - x_1$$

$$\underline{P}_4 = \Delta y \quad q_4 = y_{top} - y_1$$

$$p_1 = -\Delta x \quad q_1 = x_1 - x_{left}$$

$$p_2 = \Delta x \quad q_2 = x_{right} - x_1$$

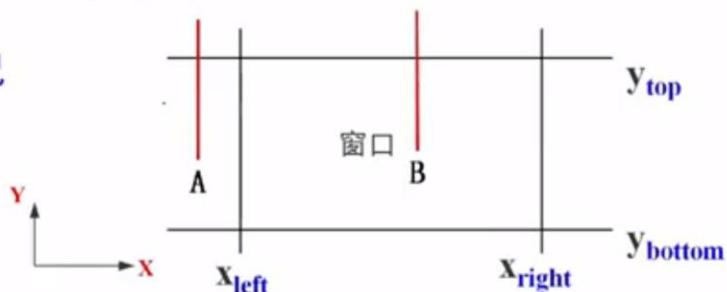
$$p_3 = -\Delta y \quad q_3 = y_1 - y_{bottom}$$

$$p_4 = \Delta y \quad q_4 = y_{top} - y_1$$

$u \cdot p_k \leq q_k$  其中，  $k = 1, 2, 3, 4$

(1) 分析  $\underline{P_k}=0$  的情况

若  $P_1 = P_2 = 0$

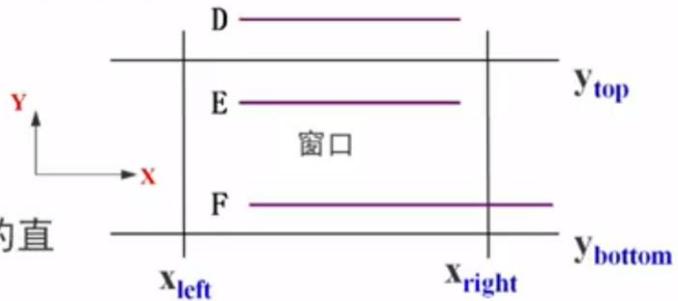


在何时  $u \cdot P_k = Q_k$ ？ 线段与边界的交点取得的  $u$  值就可以使得等式成立。

### (1) 分析 $P_k=0$ 的情况

若  $P_3=P_4=0$

任何平行于窗口某边界的直线，其  $p_k=0$



$$\begin{array}{ll} p_1 = -\Delta x & q_1 = x_1 - x_{left} \\ p_2 = \Delta x & q_2 = x_{right} - x_1 \\ p_3 = -\Delta y & q_3 = y_1 - y_{bottom} \\ p_4 = \Delta y & q_4 = y_{top} - y_1 \end{array}$$

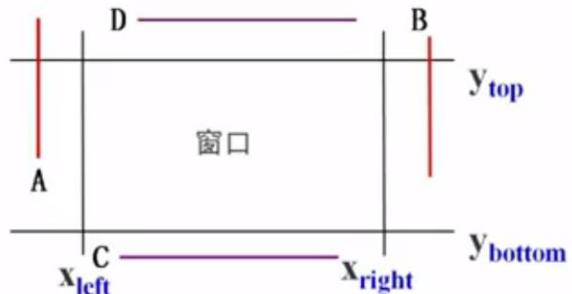
中国大学MOOC

$$u \cdot p_k \leq q_k \quad \text{其中, } k = 1, 2, 3, 4$$

### (1) 分析 $P_k=0$ 的情况

如果还满足  $q_k < 0$

则线段完全在边界外，应舍弃该线段



$$\begin{array}{ll} p_1 = -\Delta x & q_1 = x_1 - x_{left} \\ p_2 = \Delta x & q_2 = x_{right} - x_1 \\ p_3 = -\Delta y & q_3 = y_1 - y_{bottom} \\ p_4 = \Delta y & q_4 = y_{top} - y_1 \end{array}$$

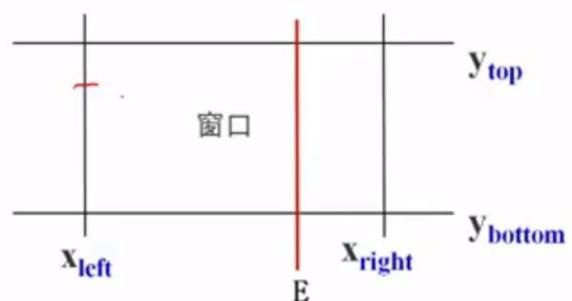
中国大学MOOC

$$u \cdot p_k \leq q_k \quad \text{其中, } k = 1, 2, 3, 4$$

### (1) 分析 $P_k=0$ 的情况

如果  $q_k \geq 0$

则进一步判断



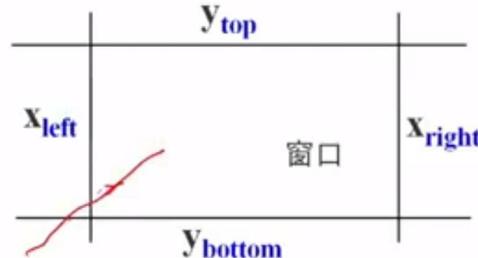
$$\begin{array}{ll}
 p_1 = -\Delta x & q_1 = x_1 - x_{left} \\
 p_2 = \Delta x & q_2 = x_{right} - x_1 \\
 p_3 = -\Delta y & q_3 = y_1 - y_{bottom} \\
 p_4 = \Delta y & q_4 = y_{top} - y_1
 \end{array}$$

$u \cdot p_k \leq q_k$  其中,  $k = 1, 2, 3, 4$

(2) 当  $p_k \neq 0$  时:

当  $p_k < 0$  时

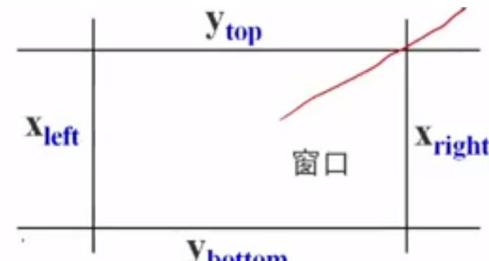
线段从裁剪边界延长线的外部延伸到内部, 是入边交点



(2) 当  $p_k \neq 0$  时:

当  $p_k > 0$  时

线段从裁剪边界延长线的内部延伸到外部, 是出边交点



根据上面的定义, 我们可以知道负值的  $k$  是入边交点, 正值的  $k$  是出边交点。我们需要注意, 在该算法下的  $dx$  并非斜率, 而是  $x_2 - x_1$  的差值。

线段和窗口边界一共有四个交点, 根据  $p_k$  的符号, 就知道哪两个是入交点, 哪两个是出交点

当  $p_k < 0$  时: 对应入边交点

当  $p_k > 0$  时: 对应出边交点

一共四个 u 值, 再加上 u=0、u=1 两个端点值, 总共 六个 值

把  $p_k < 0$  的两个 u 值和 0 比较去找最大的, 把  $p_k > 0$  的两个 u 值和 1 比较去找最小的, 这样就得到两个端点的参数值

如何理解这一步, 首先线段与窗口边界有四个交点, 这样我们就有四个 u 值, 然后再加上 u=1 和 u=0, 一共有六个交点。然后根据我们的规定, 通过判断  $P_k$  的值确定边界的交点的属性, 具体关系如上。

然后我们判断入边的端点, 找到  $(u_1, u_2, 0)$  中最大的值, 那他就是裁剪完的线段的起始点, 再判断出边的端点, 找到  $(u_3, u_4, 1)$  中的最小值, 这个点就是裁剪完的线段的终点。这样的

操作之后，我们得到裁剪完线段的两个端点。

$$\underbrace{u_k}_{\text{ }} = \frac{q_k}{p_k} \quad (\underbrace{p_k \neq 0}_{\text{ }} , k = 1, 2, 3, 4)$$

$u_k$ 是窗口边界及其延长线的交点的对应参数值

分别计算  $\underline{u_{\max}}$  和  $\underline{u_{\min}}$ ：

$$\underline{u_{\max}} = \max (0, \underline{u_k|_{p_k < 0}}, \underline{u_k|_{p_k > 0}})$$

$$\underline{u_{\min}} = \min (\underline{u_k|_{p_k > 0}}, \underline{u_k|_{p_k < 0}}, 1)$$

注意： $\underline{p_k < 0}$ , 代表入边； $\underline{p_k > 0}$ 代表出边

若  $\underline{u_{\max} > u_{\min}}$ , 则直线段在窗口外, 删除该直线

中国大学MOOC

若  $\underline{u_{\max} \leq u_{\min}}$ , 将  $\underline{u_{\max}}$  和  $\underline{u_{\min}}$  代回直线参数方程, 即求出直线与窗口的两实交点坐标。

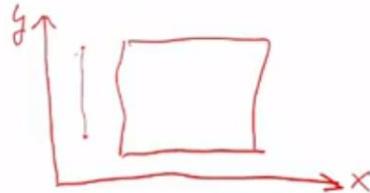
$$\begin{cases} x = x_1 + \underline{u} \cdot (x_2 - x_1) \\ y = y_1 + \underline{u} \cdot (y_2 - y_1) \end{cases}$$

注意：因为对于实交点  $0 \leq u \leq 1$ , 因此  $u_{\max}$  不能小于0,  $u_{\min}$  不能大于1

为何要做这种操作？如果直线在窗口内，则  $U_{\max}$  应该为线段的起点, 而  $U_{\min}$  为线段的终点, 必有  $U_{\max} > U_{\min}$ , 否则, 直线段不在窗口中。

## 下面写出Liang-Barsky裁剪算法步骤： 中国大学MOOC

(1) 输入直线段的两端点坐标  $(x_1, y_1)$ 、 $(x_2, y_2)$ ，以及窗口的四条边界坐标：wxl、wxr、wyb和wyt



(2) 若  $\Delta X=0$ ，则  $p_1=p_2=0$ ，此时进一步判断是否满足  $q_1<0$  或  $q_2<0$ ，若满足，则该直线段不在窗口内，算法转 (7) -结束。否则，满足  $q_1 \geq 0$  且  $q_2 \geq 0$ ，则进一步计算  $u_{\max}$  和  $u_{\min}$ ：

$$\begin{aligned} u_{\max} &= \max(0, u_k \mid p_k < 0) \\ u_{\min} &= \min(u_k \mid p_k > 0, 1) \end{aligned}$$

其中， $u_k = \frac{q_k}{p_k}$  ( $p_k \neq 0, k = 3, 4$ )。算法转 (5)

(3) 若  $\Delta y=0$ ，则  $p_3=p_4=0$ ，此时进一步判断是否满足  $q_3 \geq 0$  或  $q_4 \geq 0$ ，若满足，则该直线段不在窗口内，算法转 (7)。否则，满足  $q_3 \geq 0$  且  $q_4 \geq 0$ ，则进一步计算  $u_{\max}$  和  $u_{\min}$ ：

$$\begin{aligned} u_{\max} &= \max(0, u_k \mid p_k < 0) \\ u_{\min} &= \min(u_k \mid p_k > 0, 1) \end{aligned}$$

其中， $u_k = \frac{q_k}{p_k}$  ( $p_k \neq 0, k = 1, 2$ )。算法转 (5)

(4) 若上述两条均不满足，则有  $p_k \neq 0$  ( $k=1, 2, 3, 4$ )，此时计算  $u_{\max}$  和  $u_{\min}$ ：

$$u_{\max} = \max (0, u_k|_{p_k < 0}, u_k|_{p_k > 0})$$

$$u_{\min} = \min (u_k|_{p_k > 0}, u_k|_{p_k < 0}, 1)$$

其中， $u_k = \frac{q_k}{p_k}$  ( $p_k \neq 0, k = 1, 2, 3, 4$ )

(5) 求得  $u_{\max}$  和  $u_{\min}$  后，进行判断：若  $u_{\max} > u_{\min}$ ，则直线段在窗口外，算法转 (7)。若  $u_{\max} \leq u_{\min}$ ，利用直线的参数方程：

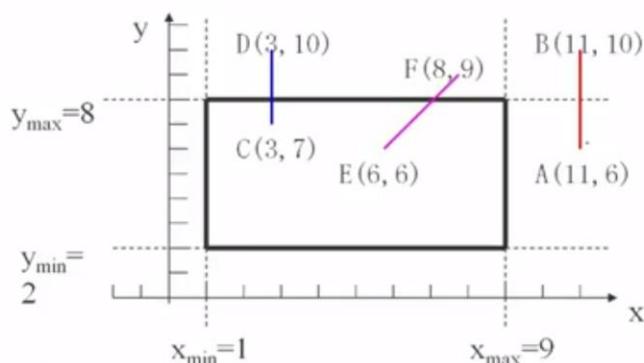
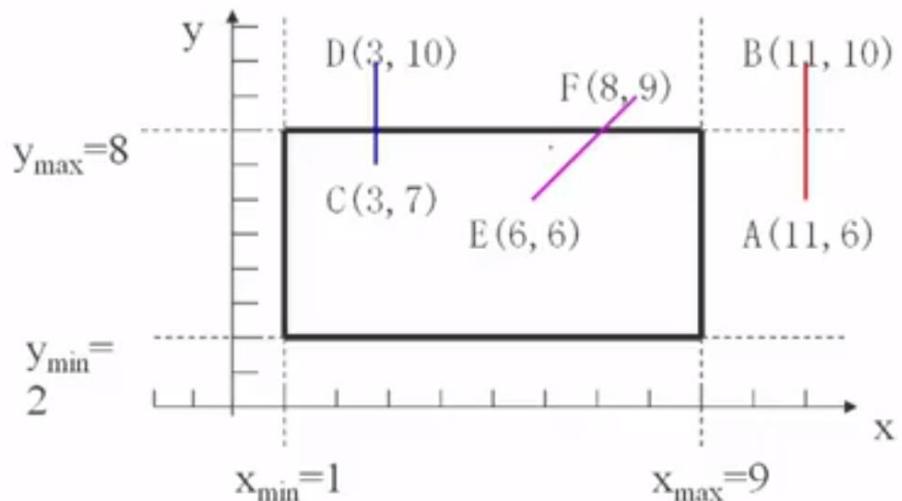
$$x = x_1 + u \cdot (x_2 - x_1)$$

$$y = y_1 + u \cdot (y_2 - y_1)$$

(6) 利用直线的扫描转换算法绘制在窗口内的直线段。

(7) 算法结束。

## 用Liang-Barsky算法裁减直线段举例



令：

$$\begin{array}{ll} p_1 = -\Delta x & q_1 = x_1 - x_{left} \\ p_2 = \Delta x & q_2 = x_{right} - x_1 \\ p_3 = -\Delta y & q_3 = y_1 - y_{bottom} \\ p_4 = \Delta y & q_4 = y_{top} - y_1 \end{array}$$

中国大学MOOC

对于直线AB，有：

$$\begin{array}{ll} p_1 = 0 & q_1 = 10 \\ p_2 = 0 & q_2 = -2 \\ p_3 = -4 & q_3 = 4 \\ p_4 = 4 & q_4 = 2 \end{array}$$

AB完全在右边界之右

对于直线CD，有：

$$p_1 = 0 \quad q_1 = 2$$

$$p_2 = 0 \quad q_2 = 6$$

$$p_3 = -3 \quad q_3 = 5$$

$$p_4 = 3 \quad q_4 = 1$$

$$u_3 = -\frac{5}{3} \quad u_4 = \frac{1}{3}$$

$$u_{\max} = \max(0, -\frac{5}{3}) = 0$$

$$u_{\min} = \min(1, \frac{1}{3}) = \frac{1}{3}$$

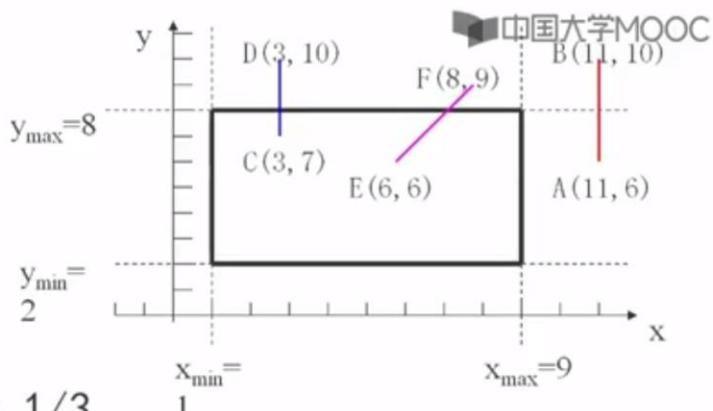
注意  $U_k = Q_k/P_k$ , 这个等式只能用于边界交点求参数

$$u_{\max} = 0$$

$$u_{\min} = \frac{1}{3}$$

$$x = x_1 + u \cdot (x_2 - x_1)$$

$$y = y_1 + u \cdot (y_2 - y_1)$$



裁减后直线的两个端点是  $(3, 7)$  和  $(3, 8)$ 。

比较  $U_{\max}$  和  $U_{\min}$  的值，如果  $U_{\max} < U_{\min}$ ，则为实交点，否则为虚交点

对于直线EF，有：

$$\underline{p_1} = -2 \quad q_1 = 5$$

$$\underline{p_2} = 2 \quad q_2 = 3$$

$$\underline{p_3} = -3 \quad q_3 = 4$$

$$\underline{p_4} = 3 \quad q_4 = 2$$

$$\underline{u_1} = -5/2 \quad \underline{u_2} = 3/2$$

$$\underline{u_3} = -4/3 \quad \underline{u_4} = 2/3$$

$$\text{因此: } \underline{u_{\max}} = \max(0, -5/2, -4/3) = 0 \quad u_{\max} < u_{\min}$$

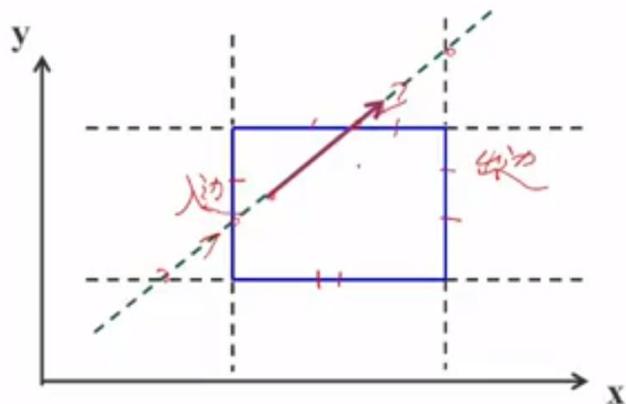
$$u_{\min} = \min(1, 3/2, 2/3) = 2/3$$

将得到的  $U_{\max}$  和  $U_{\min}$  分别带入参数方程，得到两个端点的坐标。

小结

## Liang-Barsky算法小结

### 1、直线段看成是有方向的



## 2、直线参数化

$$\begin{aligned}x &= x_1 + u \cdot (x_2 - x_1) \\y &= y_1 + u \cdot (y_2 - y_1)\end{aligned}\quad 0 \leq u \leq 1$$

3、判断线段上一点是否在窗口内，需满足下面两个不等式

$$\left\{ \begin{array}{l} x_{left} \leq x_1 + u \cdot \Delta x \leq x_{right} \\ y_{bottom} \leq y_1 + u \cdot \Delta y \leq y_{top} \end{array} \right.$$

$$u \cdot p_k \leq q_k$$

## 4、线段和窗口边界一共有四个交点

$$\underline{u_k} = \frac{q_k}{p_k} \quad (p_k \neq 0, k = 1, 2, 3, 4)$$

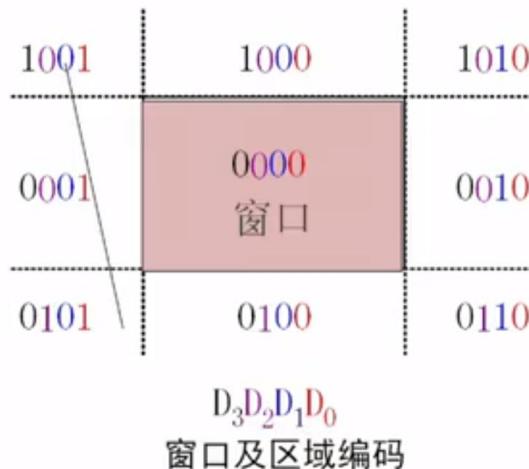
$$\left\{ \begin{array}{l} \underline{u_{max}} = \max (0, u_k|_{p_k < 0}, u_k|_{p_k > 0}) \\ \underline{u_{min}} = \min (u_k|_{p_k > 0}, u_k|_{p_k < 0}, 1) \end{array} \right. \quad \text{U}_{max} < \text{U}_{min}$$

$$\left\{ \begin{array}{l} x = x_1 + u \cdot (x_2 - x_1) \\ y = y_1 + u \cdot (y_2 - y_1) \end{array} \right.$$

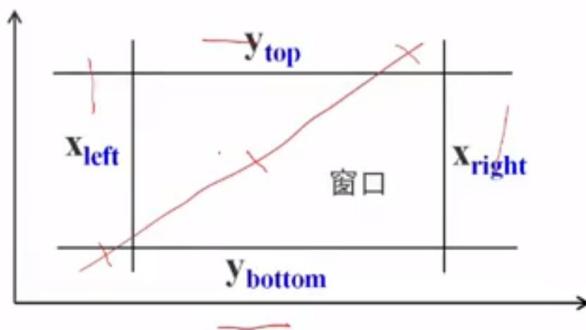
## 4、Cohen-Sutherland 和 Liang-Barsky 比较

### Cohen-Sutherland PK Liang-Barsky裁剪算法比较

1、Cohen-Sutherland算法的核心思想是编码



2、如果被裁剪的图形大部分线段要么在窗口内或者要么完全在窗口外，很少有贯穿窗口的。Cohen-Sutherland算法效果非常好



3、在一般情况下，Liang-Barsky裁剪算法的效率则优于 Cohen-Sutherland算法

4、Cohen-Sutherland和Liang-Barsky只能应用于矩形窗口

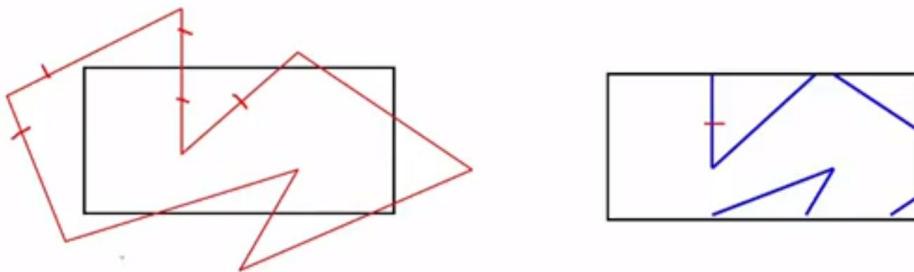
裁剪算法是非常底层的算法，任何一个图形显示的算法和软件都离不开这些底层算法

# 五：多边形裁剪算法

1、Sutherland-Hodgeman 算法

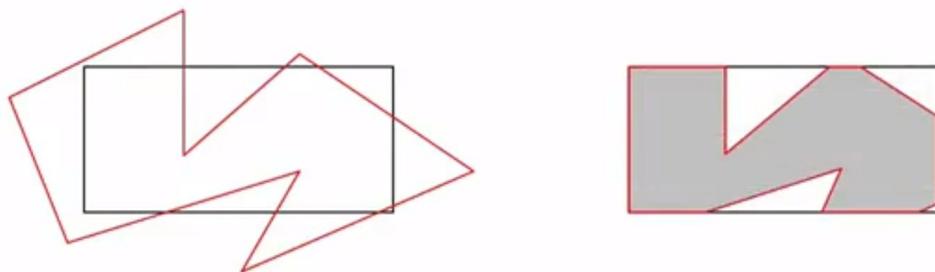
## 一、多边形的裁剪

中国大学MOOC



即得到一系列不连续的直线段！

而实际上，应该得到的是下图所示的有边界的区域！



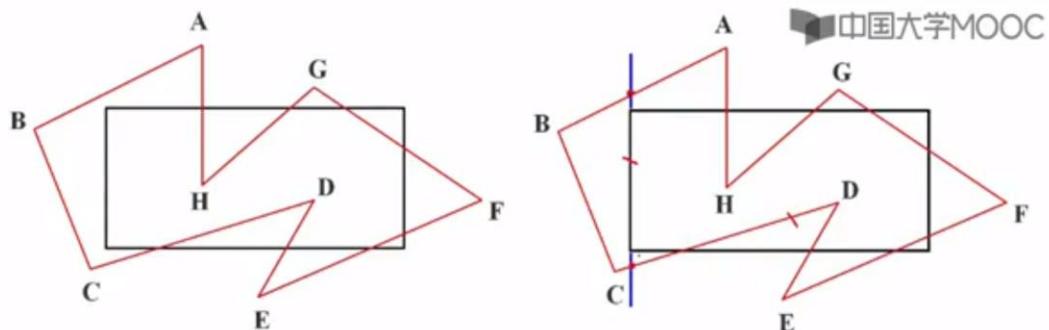
多边形裁剪算法的输出应该是裁剪后的多边形边界的顶点序列！

需要构造能产生一个或多个封闭区域的多边形裁剪算法

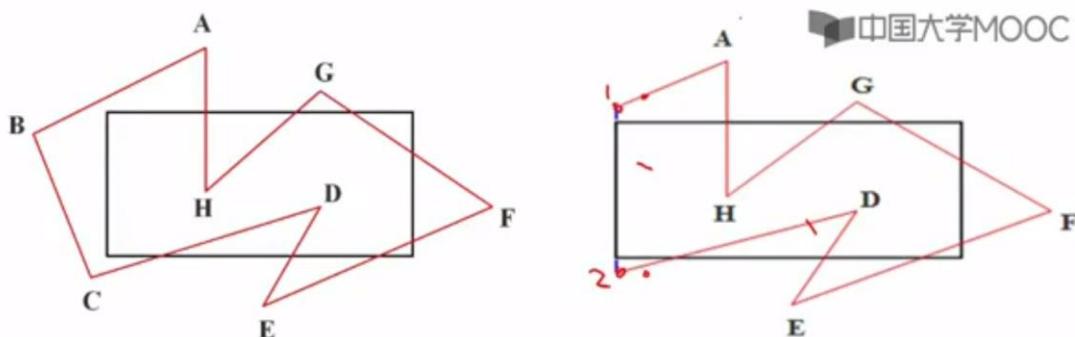
## 二、Sutherland-Hodgeman多边形裁剪

中国大学MOOC

该算法的基本思想是将多边形边界作为一个整体，每次用窗口的一条边对要裁剪的多边形和中间结果多边形进行裁剪，体现一种分而治之的思想

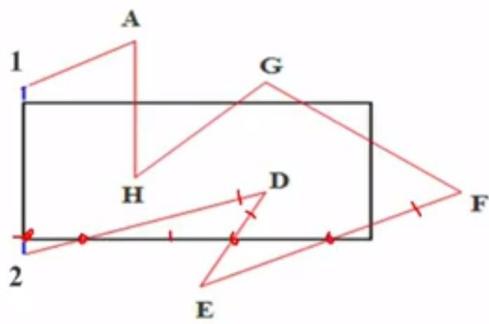


算法的输入： ABCDEFGH

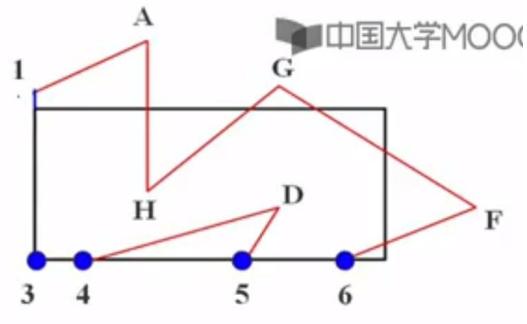


算法的输入： ABCDEFGH

输出： A12DEFGHA



输入：A12DEFGH



输出：A134D56FGH

裁剪得到的结果多边形的顶点有两部分组成：

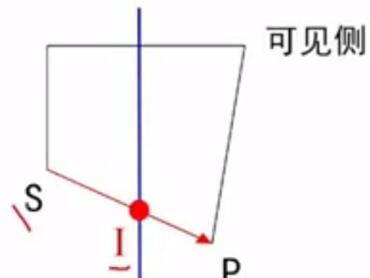
- (1) 落在可见一侧的原多边形顶点
- (2) 多边形的边与裁剪窗口边界的交点

再继续做这种操作，就可以得到想要的裁剪后多边形

根据多边形每一边与窗口边所形成的位置关系，沿着多边形依次处理顶点会遇到四种情况：

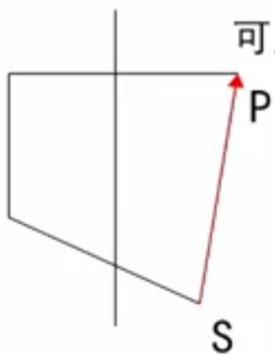
(1) 第一点S在不可见侧面，而第二点P在可见侧

交点I与点P均被加入到输出顶点表中



(2) 是S和P都在可见侧

则P被加入到输出顶点表中

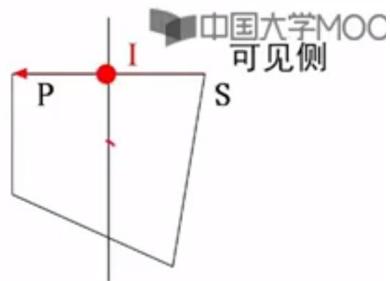


(2) 输出P

为何不将 P 加入呢？因为在此操作之前，就已经将 S 加入输出顶点表了。

(3) S在可见侧，而P在不可见侧

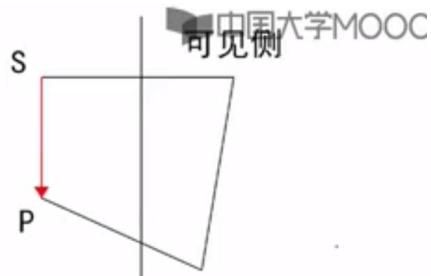
则交点I被加入到输出顶点表中



(3) 输出I

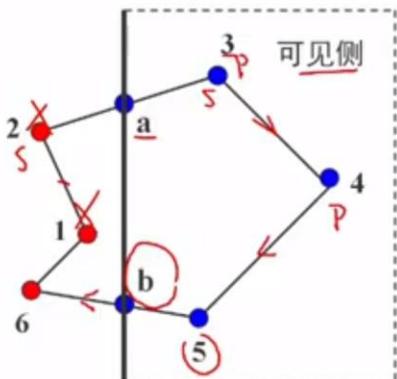
(4) 如果S和P都在不可见侧

输出顶点表中不增加任何顶点



输入: 1 2 3 4 5 6

输出:

a 3 4 5 b

```
while 对于每一个窗口边或面 do
```

```
begin
```

```
if  $P_1$  在窗口边的可见一侧 then 输出  $P_1$ 
```

```
for i=1 to n do
```

```
begin
```

```
if  $P_1$  在窗口边的可见一侧 then
```

```
if  $P_{i+1}$  在窗口边的可见一侧 then 输出  $P_{i+1}$ 
```

```
else 计算交点并输出交点
```

```
else if  $P_{i+1}$  在窗口可见一侧, then 计算交点
```

```
并输出交点, 同时输出  $P_{i+1}$ 
```

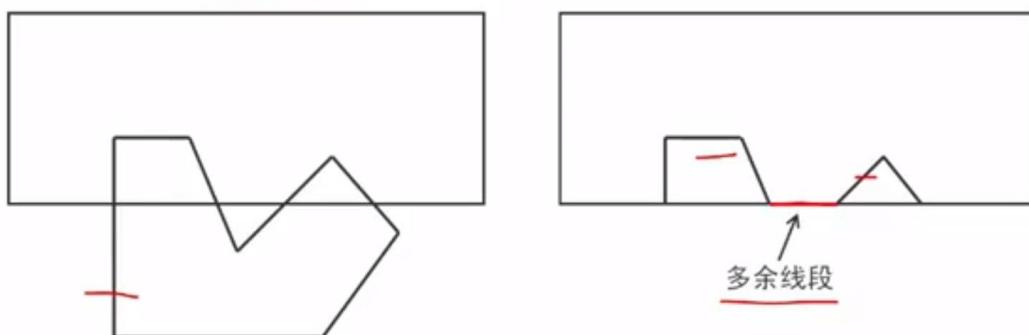
```
end
```

```
end
```

```
end
```

### Sutherland-Hodgeman算法不足之处

利用Sutherland-Hodgeman裁剪算法对凸多边形进行裁剪可以获得正确的裁剪结果，但是。。。



# 六：文字裁剪

## 三、文字裁剪



屏幕上显示的不仅仅是多边形和直线等，还显示字符。字符如何来处理？

字符并不是由直线段组成的。文字裁剪包括以下几种：

串精度裁剪

字符精度裁剪

笔划/像素精度裁剪



### 1、串精度裁剪



当字符串中的所有字符都在裁剪窗口内时，就全部保留它，否则舍弃整个字符串



### 2、字符精度裁剪



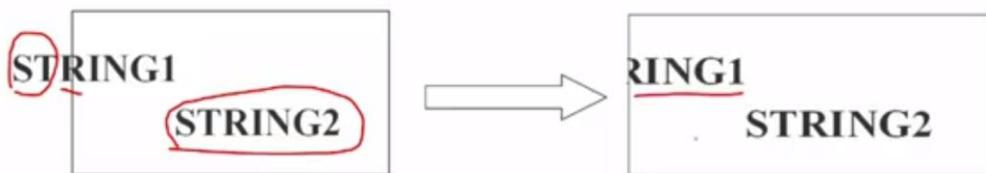
在进行裁剪时，任何与窗口有重叠或落在窗口边界以外的字符都被裁剪掉



### 3、笔画/象素精度裁剪



将笔划分解成直线段对窗口作裁剪。需判断字符串中各字符的哪些像素、笔划的哪一部分在窗口内，保留窗口内部分，裁剪掉窗口外的部分



## 七：消隐算法

### 1、Z-Buffer 算法

#### 主要讲述的内容：



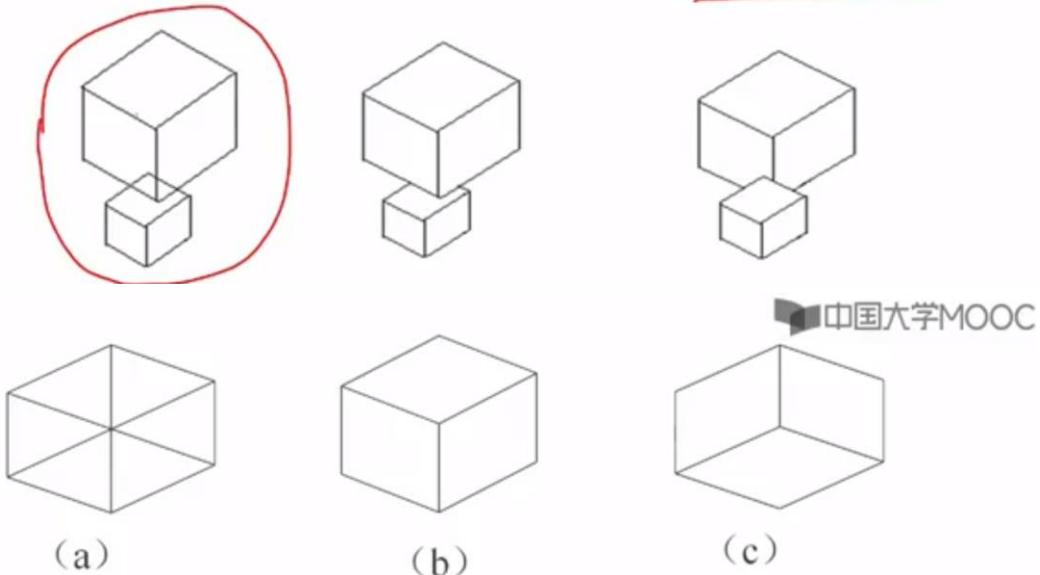
消隐的分类，如何消除隐藏线、隐藏面，主要介绍以下几个算法：

- 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

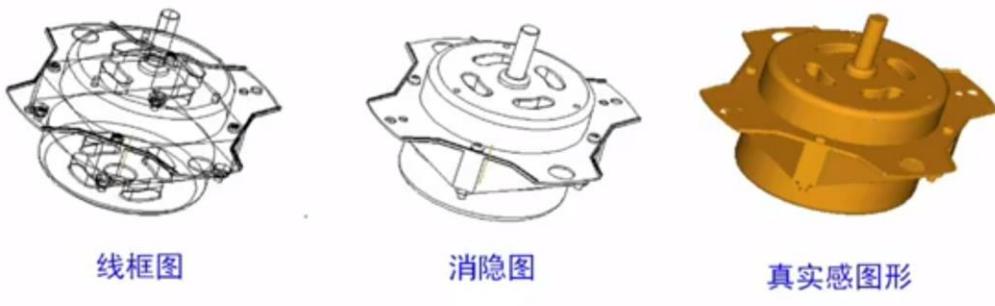
当我们观察空间任何一个不透明的物体时，只能看到该物体朝向我们的那些表面，其余的表面由于被物体所遮挡我们看不到

如果把可见和不可见的线都画出来，对视觉会造成多义性



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。

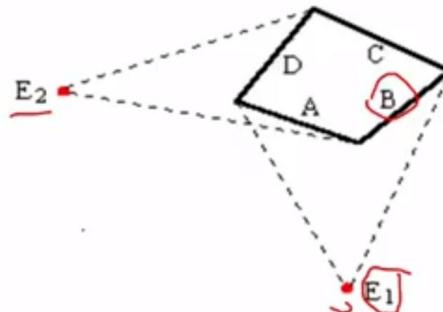
要绘制出意义明确的、富有真实感的立体图形，~~首先必须消去形体中的不可见部分~~，而只在图形中表现可见部分



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



## 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！

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）

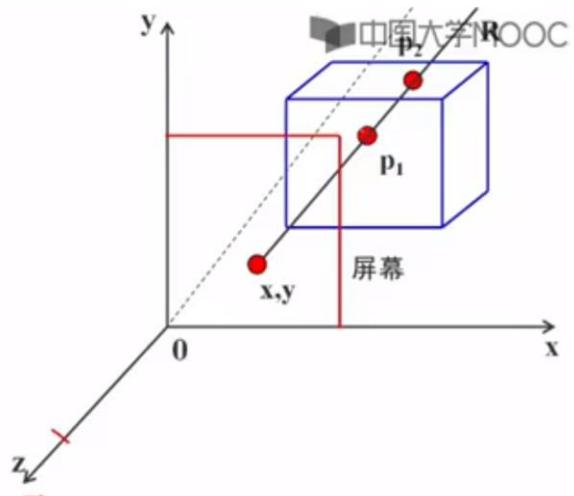
存放图像空间每个可见像素的z坐标



假定xoy面为投影面，z轴为观察方向

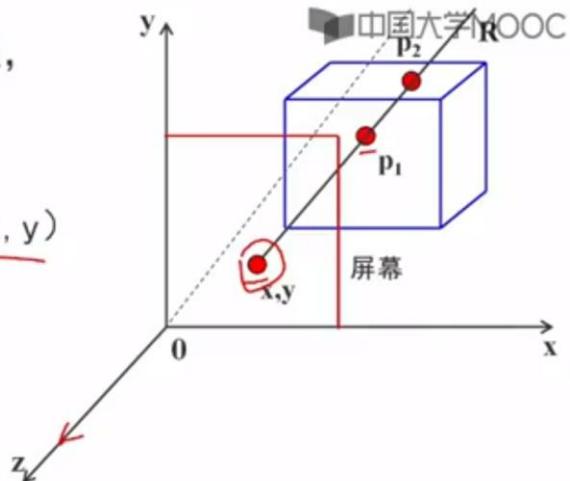
过屏幕上任意像素点  $(x, y)$  作平行于 z轴的射线 R，与物体表面相交于  $p_1$  和  $p_2$  点

$p_1$  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



## 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Pindexp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：

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

## 点与多边形的包含性检测：

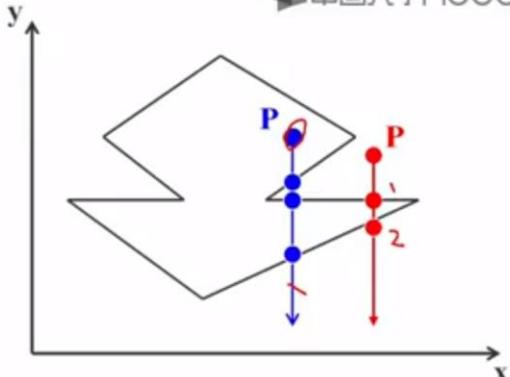
中国大学MOOC

### (1) 射线法

由被测点P处向  $y = -\infty$  方向作射线

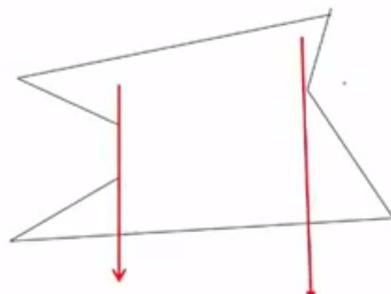
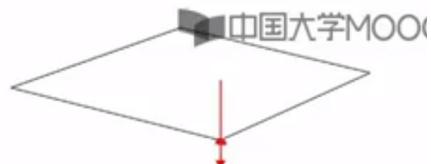
交点个数是奇数，则被测点在多边形内部

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

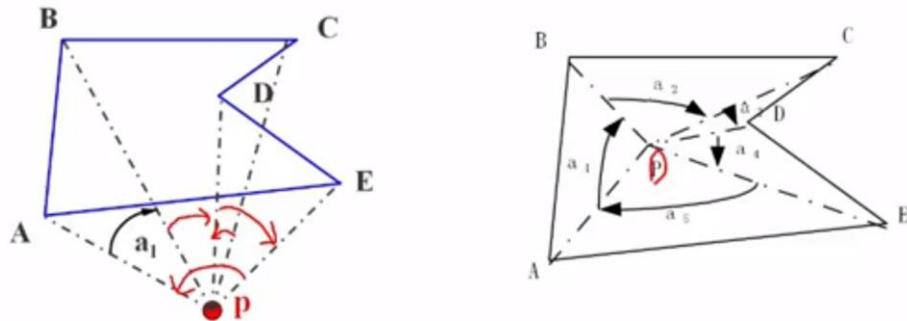


用射线法来判断一个点是否在多边形内的弊端：

(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 $2\pi$ ，点在多边形内部

代数和为 $\pi$ ，点在多边形边上

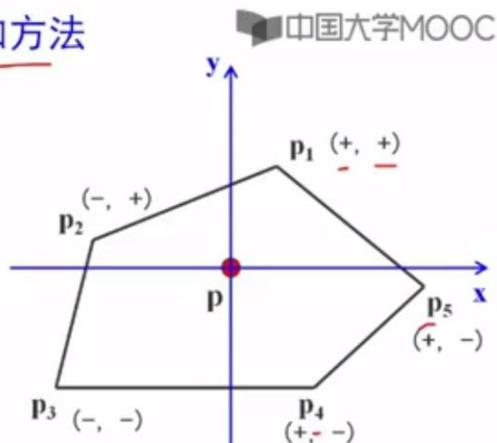
这个算法为什么是稳定的？假如算出来后代数和不是0，而是0.2或0.1，那么基本上可以断定这个点在外部，可以认为是有计算误差引起的，实际上0。

但这个算法效率也不高，问题是算弧长不容易，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

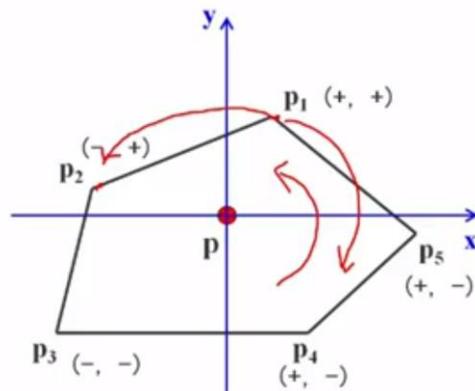
## (3) 以顶点符号为基础的弧长累加方法

p是被测点，按照弧长法，p点的代数和为 $2\pi$ 。

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
(+ +)	(- +)	$\pi/2$	I → II
(+ +)	(- -)	$\pm\pi$	I → III
(+ +)	(+ -)	$-\pi/2$	I → IV
...	...	...	...



同一个象限认为是0，跨过一个象限是 $\pi/2$ ，跨过两个象限是 $\pi$ 。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

## 2、区间扫描线算法

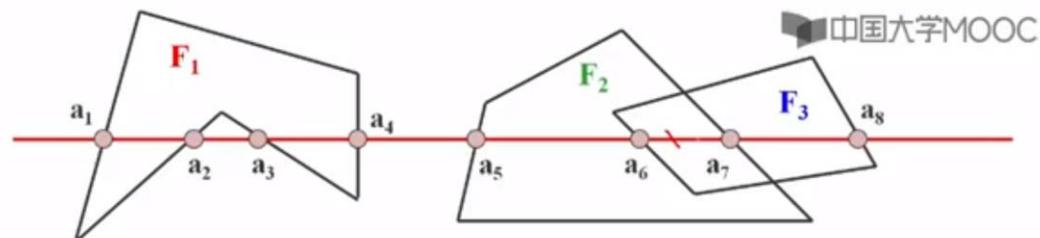
### 二、区间扫描线算法

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

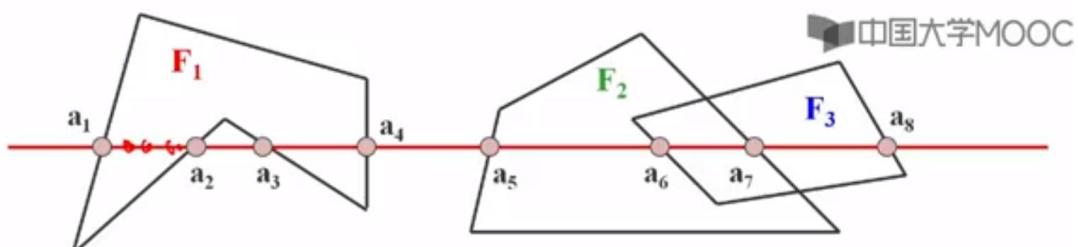
下面介绍区间扫描线算法。该算法放弃了z-buffer的思想，是一个新的算法，这个算法被认为是消隐算法中最快的之一

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

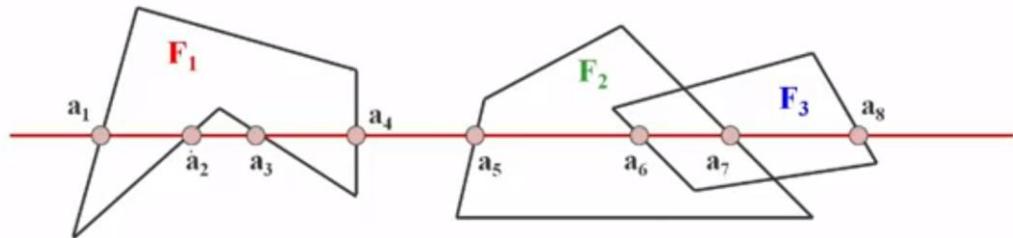


如果把扫描线和多边形的这些交点都求出来，对每个区间，只要判断一个像素的要画什么颜色，那么整个区间的颜色都解决了，这就是区间扫描线算法的主要思想

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？

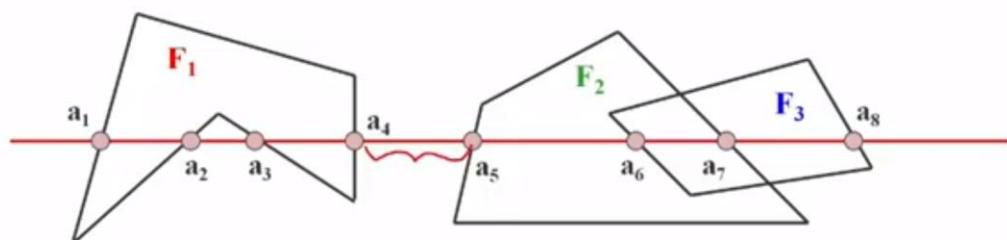
中国大学MOOC



首先要有投影多边形，然后求交点，然后交点进行排序，排序的结果就分成了一个区间

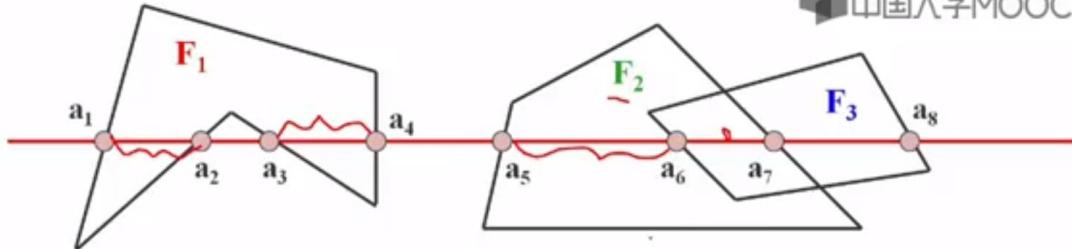
如何确定小区间的颜色？

中国大学MOOC



(1) 小区间上没有任何多边形，如[\[a4, a5\]](#)，用背景色显示

中国大学MOOC



(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形，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：

中国大学MOOC

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？

### 3、Warnock 消隐算法

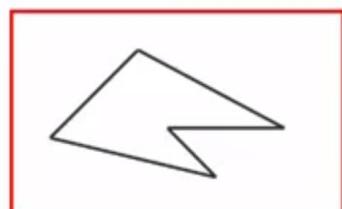
Warnock算法是图像空间中非常经典的一个算法

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

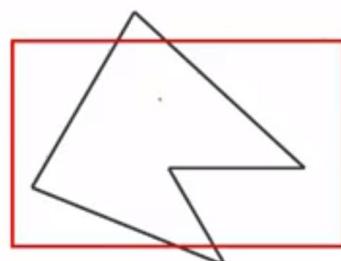
把物体投影到全屏幕窗口上，然后递归分割窗口，直到窗口内目标足够简单，可以显示为止

#### 一、什么样的情况下，画面足够简单可以立即显示？中国大学MOOC

(1) 窗口中仅包含一个多边形



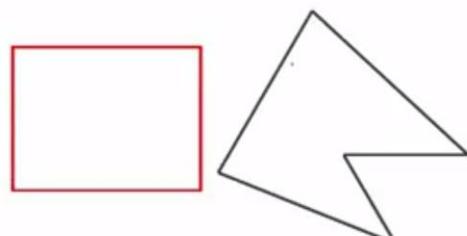
(2) 窗口与一个多边形相交，且  
窗口内无其它多边形



(3) 窗口为一个多边形所包围



(4) 窗口与一个多边形相分离

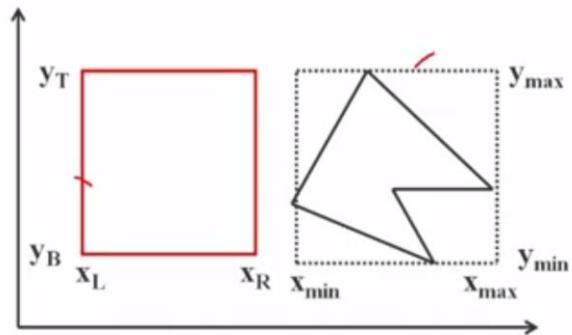


## 如何判别一个多边形和窗口是分离的？

中国大学MOOC

当满足下列条件时，多边形和窗口分离：

$$\begin{array}{l} \underline{x_{\min}} > x_R \text{ or } \underline{x_{\max}} < x_L \\ \text{or} \\ \underline{y_{\min}} > y_T \text{ or } \underline{y_{\max}} < y_B \end{array}$$



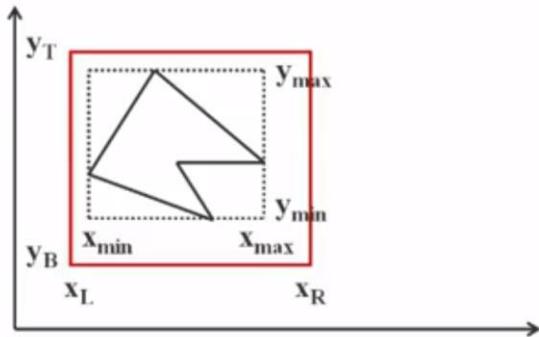
我们做出多边形的最小包围和，所谓最小包围和，就是以  $X_{\min}$ ,  $X_{\max}$ ,  $Y_{\max}$ ,  $Y_{\min}$  为边界的矩形，再根据这几个坐标来判断多边形和窗口的位置。

## 如何判别一个多边形在窗口内？

中国大学MOOC

当满足下列条件时，多边形被窗口包含：

$$\begin{array}{l} \underline{x_{\min}} \geq x_L \text{ &} \underline{x_{\max}} \leq x_R \\ \text{and} \\ \underline{y_{\min}} \geq y_B \text{ &} \underline{y_{\max}} \leq y_T \end{array}$$

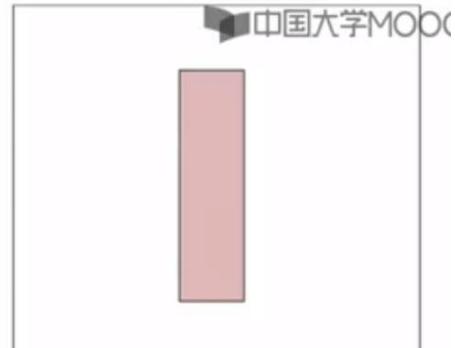


四个条件都要满足

### 算法步骤：

(1) 如果窗口内没有物体则按背景色显示

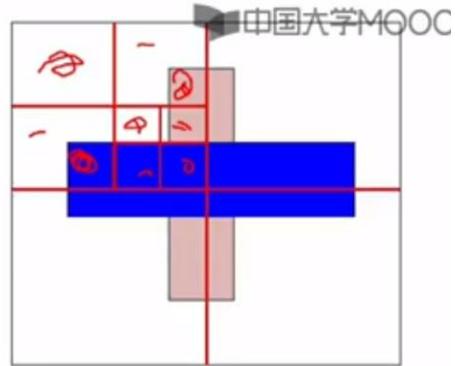
(2) 若窗口内只有一个面，则把该面显示出来



(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)。



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

## 八、光栅图形学算法小结

### 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。

## 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

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

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算法的主要问题是不断地判一个点是否在多边形内

即射线法、代数弧长累加法、以顶点符号的弧长累加方法

**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

(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

