第3-4章二维裁剪

课程目标

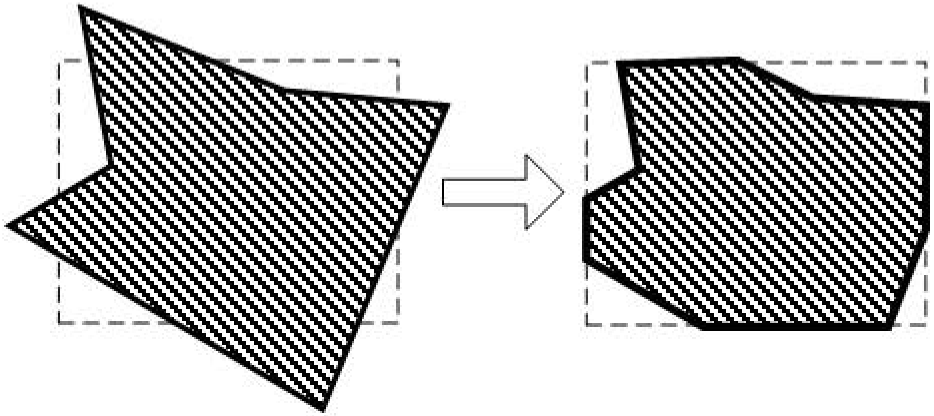
- 点的裁剪
- 直线段的裁剪
- 多边形的裁剪

何谓裁剪

确定图形中哪些部分落在显示区之内，

哪些落在显示区之外，以便只显示落在显示区内的那部分图形。

这个选择过程称为裁剪。



点的裁剪

对于一点P(x , y)，要判断其是否可见，可利用下面的不等式组来判断此点是否落在窗口范围内。

$$\begin{cases} w_{xl} \le x \le w_{xr} \\ w_{yb} \le y \le w_{yt} \end{cases}$$

满足上述不等式组的点则在窗口范围内，可见，保留；

反之，该点落在窗口外，不可见，舍弃（裁剪）

直线段的裁剪

由点的裁剪方法想到的

一种最简单的裁剪方法——逐点比较法

把图形离散成点，判断各点，若满足则在窗口内，

为可见点，否则即在窗口外被裁掉。

缺点

速度太慢，不适用

直线段的裁剪

裁剪的目的

- ✓ 判断图元是否落在裁剪窗口内，找出其位于内部的部分

裁剪的处理的基础

- ✓ 图元关于窗口内外关系的判别
- ✓ 图元与窗口的求交

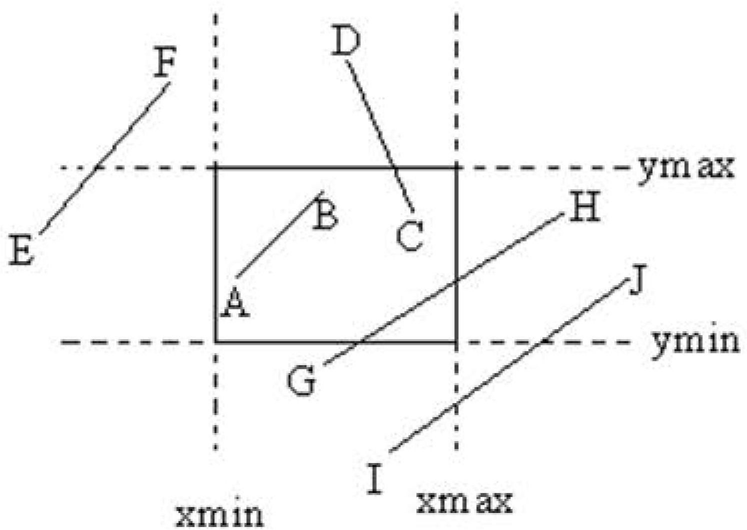
假定条件

- ✓ 矩形裁剪窗口：$[x_{min},x_{max}]*[y_{min},y_{max}]$
- ✓ 待裁剪线段： $P_0(x_0,y_0)P_1(x_1,y_1)$

直线段的裁剪

待裁剪线段和窗口的关系

- ✓ 线段完全可见
- ✓ 显然不可见
- ✓ 线段至少有一端点在窗口之外，但非显然不可见



为提高效率，算法设计时应考虑：

（一）快速判断情形(1)(2)；

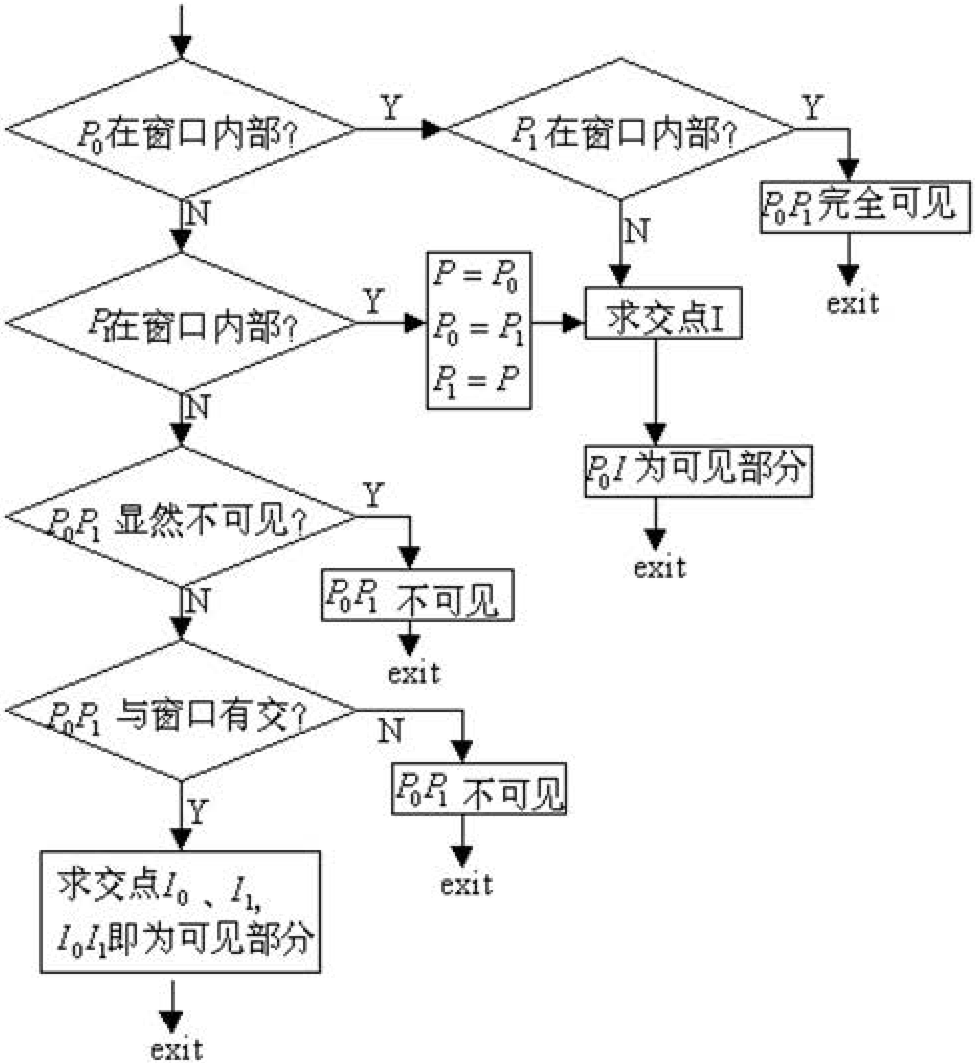
（二）设法减少情形(3)求交次数和每次求交时所需的计算量。

直线段的裁剪

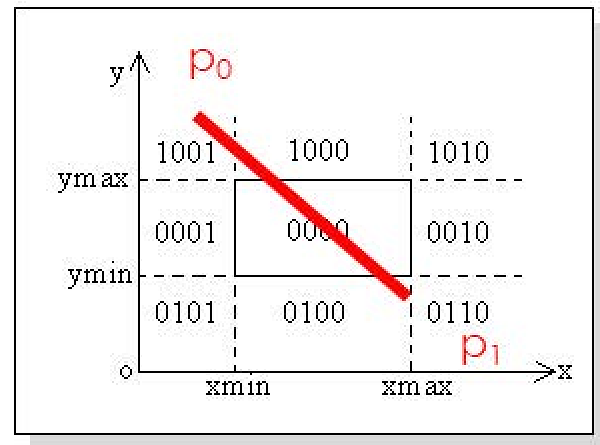
直线段的裁剪算法

- ✓直接求交算法
- ✓Cohen-Sutherland算法
- ✓中点算法

直接求交算法



P_0在窗口内部?
Y
P_1在窗口内部?
Y
P_0P_1完全可见
exit
N
N
P在窗口内部?
Y
$P = P_0$
$P_0 = P_1$
$P_1 = P$
求交点I
P_0I 为可见部分
exit
N
P_0P_1 显然不可见?
Y
P_0P_1 不可见
exit
N
P_0P_1 与窗口有交?
N
P_0P_1 不可见
exit
Y
求交点I_0 、I_1,
I_0I_1即为可见部分
exit



y
p_0
1001
1000
1010
ymax
0001
0000
0010
ymin
0101
0100
0110
p_1
o
xmin
xmax
x

Cohen-Sutherland裁剪算法

对于每条线段P1P2分为三种情况处理分为三种情况处理.

✓若P1P2完全在窗口内，则显示该线段P1P2，“取”之

✓若P1P2明显在窗口外，则丢弃该线段，“弃”之

✓若线段不满足“取”或“弃”的条件，则在交点处把线段分为两段。其中一段完全在窗口外，可弃之。对另一段重复上述处理。

裁剪过程是递归的

Cohen-Sutherland裁剪算法

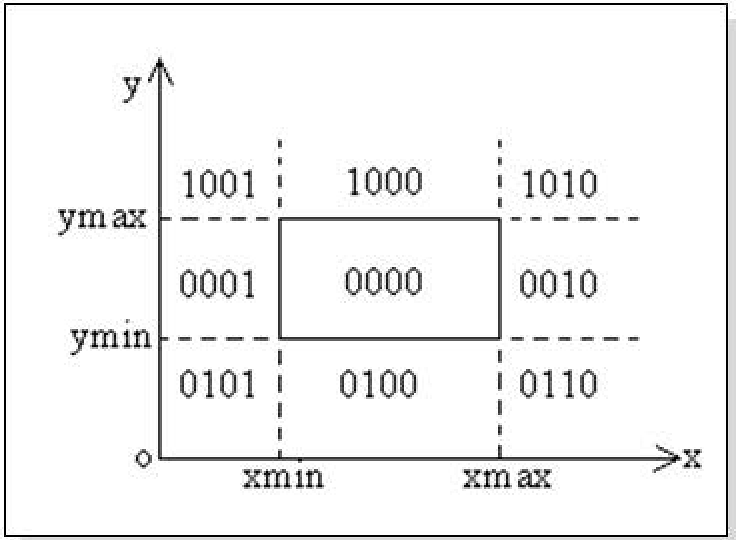
编码算法

特点：对显然不可见线段的快速判别

编码方法：由窗口四条边所在直线把二维平面分成9个区域，

每个区域赋予一个四位编码，$C_tC_bC_rC_l$，上下右左；

$$C_t=\begin{cases}1 & y>y_{max}\\0 & other\end{cases}\qquad C_b=\begin{cases}1 & y<y_{min}\\0 & other\end{cases}\qquad C_r=\begin{cases}1 & x>x_{max}\\0 & other\end{cases}\qquad C_l=\begin{cases}1 & x>x_{min}\\0 & other\end{cases}$$

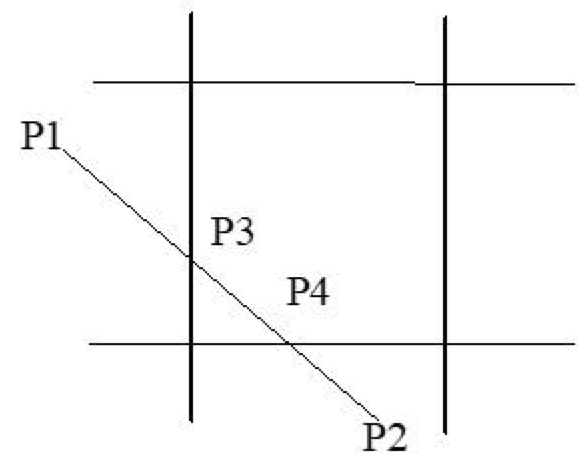


Cohen-Sutherland裁剪算法

对于每条线段P1P2分为三种情况处理分为三种情况处理:

- ✓若P1P2完全在窗口内code1=0000,且code2=0000,则“取”
- ✓若P1P2明显在窗口外code1&code2≠0，则“弃”
- ✓在交点处把线段分为两段。其中一段完全在窗口外，可弃之。对另一段重复上述处理。

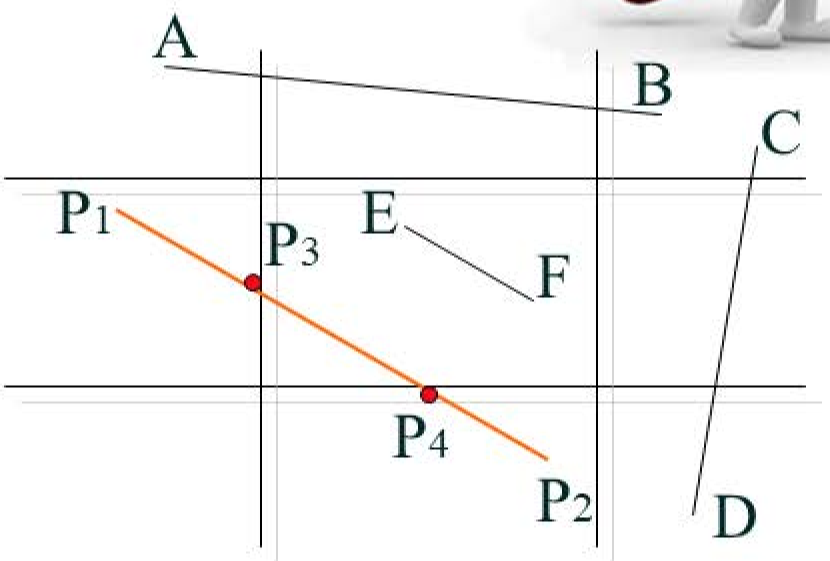
1001	1000	1010
0001	0000	0010
0101	0100	0110



Cohen-Sutherland裁剪算法

1001	1000	1010
0001	0000	0010
0101	0100	0110

裁剪空间编码状态表

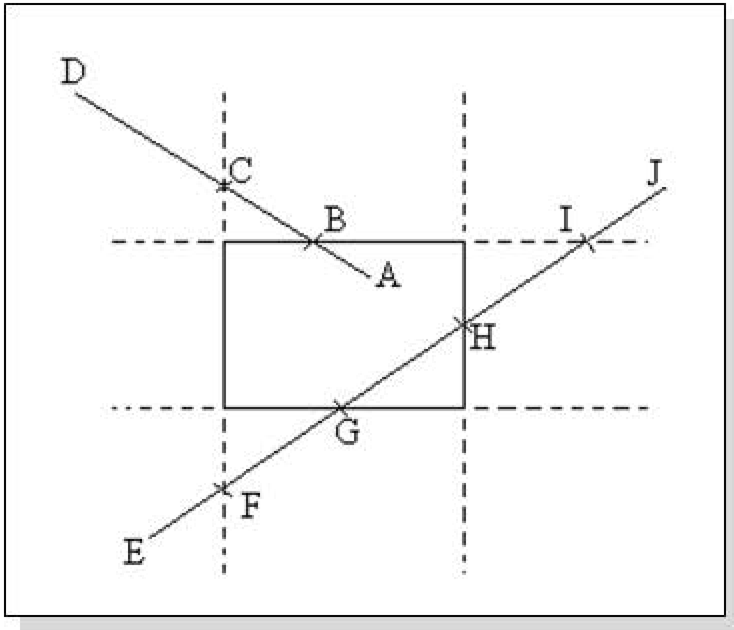


按位与运算

E	0000	A	1001	C	1010	P_1	0001
F	0000	B	1010	D	0110	P_2	0100
	0000		1000		0010		0000

Cohen-Sutherland裁剪算法

对于那些非完全可见、又非显然不可见的线段，需要求交（如，线段AD)，求交前先测试与窗口哪条边所在直线有交？(按序判断端点编码中各位的值$C_lC_rC_bC_t$）



求交测试顺序固定(左右下上)
最坏情形，线段求交四次

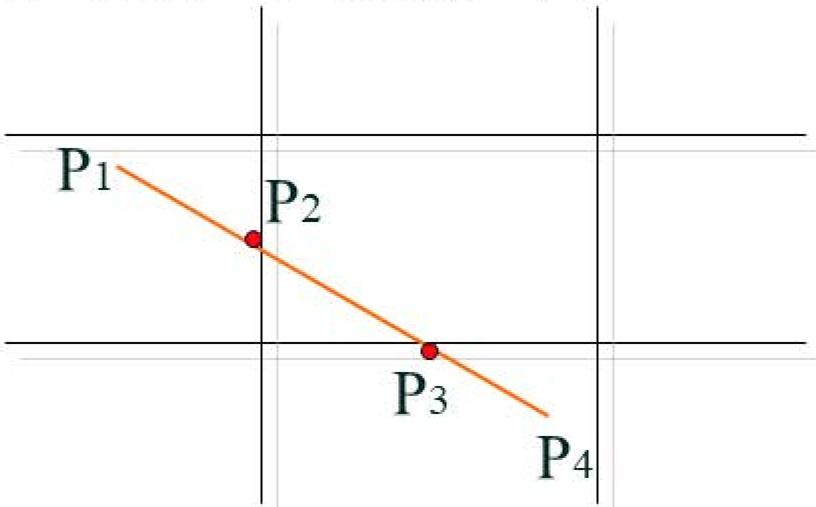
Cohen-Sutherland裁剪算法

在编程时，按照左右下上固定顺序检测区号的各位是否为0。舍弃在窗外的子线段，只要用交点的坐标值代替被舍弃端点的坐标即可实现

P1: Code1=0001　P1在窗口左边,计算与左边界的交点P2
P1,P2在窗口外,弃之.

计算与下边界的交点P3,因为P4:Code4=0100　P4在窗口外.
用P3丢弃P3P4.

最后剩下P2P3在窗口中



Cohen-Sutherland裁剪算法

在算法实现时，不必让直线与每条窗口边界求交，只要按照左右下上顺序检测到端点区码的某位不为0时（等于1），才把直线与对应的窗口边界进行求交

计算线段P1(x1,y1)P2(x2,y2)与窗口边界的交点

```
if(LEFT&code !=0)             {x=XL; y=y1+(y2-y1)*(XL-x1)/(x2-x1);}
else if(RIGHT&code !=0)       {x=XR; y=y1+(y2-y1)*(XR-x1)/(x2-x1);}
else if(BOTTOM&code !=0)      {y=YB; x=x1+(x2-x1)*(YB-y1)/(y2-y1);}
else if(TOP & code !=0)       {y=YT; x=x1+(x2-x1)*(YT-y1)/(y2-y1);}
```

Cohen-Sutherland裁剪算法

如何进行编码?

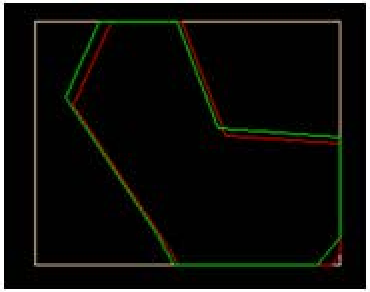
```
#define  LEFT 1
#define  RIGHT   2
#define  BOTTOM 4
#define  TOP 8
int Encode (int x,int y,int XL,int XR,int YB,int YT)
{
   int c=0;
   if(x<XL)              c=c|LEFT;
   else if(x>XR)         c=c|RIGHT;
   if (y<YB)             c=c|BOTTOM;
   else if (y>YT)        c=c|TOP;
   return c;
}
```

Cohen-Sutherland裁剪算法

```
void C_S_Line(POINT p1,POINT p2,int XL,int XR,int YB,int YT)
{
    int x1,x2,y1,y2,x,y;
    int code1,code2,code;
    x1=p1.x;x2=p2.x;y1=p1.y;y2=p2.y;
    Encode(x1,y1,&code1,XL,XR,YB,YT);
    Encode(x2,y2,&code2,XL,XR,YB,YT);
    while(code1!=0 || code2!=0) /* 非第一种情况*/
    {
        if ((code1 & code2) !=0) return; //窗口外，为第二种情况
        code=code1;
        if (code1==0) code=code2;
        if ((LEFT & code) !=0){/* 线段与左边界相交*/
            x=XL;   y=y1+(y2-y1)*(XL-x1)/(x2-x1);
        }
        else if((RIGHT & code)!=0){/* 线段与右边界相交*/
            x=XR;   y=y1+(y2-y1)*(XR-x1)/(x2-x1);
        }
```

Cohen-Sutherland裁剪算法

```
        else if((BOTTOM & code) != 0){/* 线段与下边界相交*/
            y=YB;   x=x1+(x2-x1)*(YB-y1)/(y2-y1);
        }
        else if((TOP & code)!=0){/* 线段与上边界相交*/
            y=YT;   x=x1+(x2-x1)*(YT-y1)/(y2-y1);
        }
        if(code==code1){
            x1=x; y1=y;
            Encode(x,y,&code1,XL,XR,YB,YT);
        }
        else{
            x2=x; y2=y;
            Encode(x,y,&code2,XL,XR,YB,YT);
        }
    }
    p1.x=x1;p1.y=y1;p2.x=x2;p2.y=y2;
    moveto(p1.x,p1.y);lineto(p2.x,p2.y);
}
```



Cohen-Sutherland裁剪算法

优点

✓直观方便，速度较快

缺点

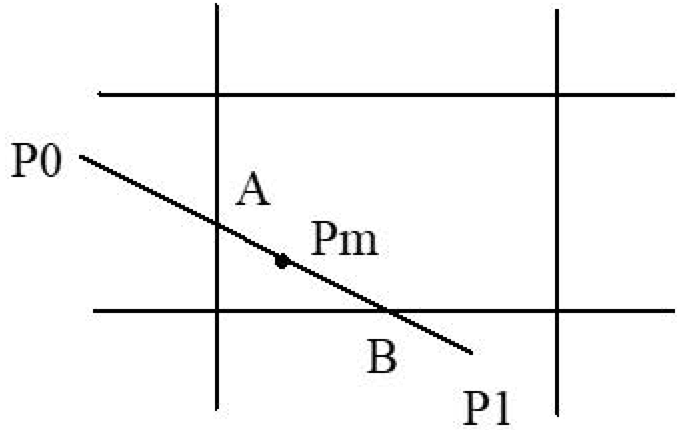
✓位逻辑乘的运算，有些高级语言中是不便进行的；

✓全部舍弃的判断只适合于那些仅在窗口同侧的线段，对于跨越三个区域的线段，就不能一次作出判别而舍弃。

中点分割裁剪算法

与Cohen-Sutherland算法一样首先对线段端点进行编码，并把线段与窗口的关系分为三种情况：

- ✓ 全在、完全不在和线段和窗口有交。
- ✓ 对前两种情况，进行一样的处理。
- ✓ 对于第三种情况，用中点分割法求出线段与窗口的交点。

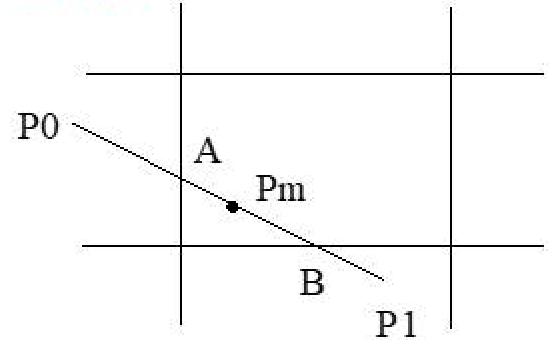


A、B分别为距P0、P1最近的可见点，Pm为P0P1中点

中点分割裁剪算法

从P0出发找最近可见点采用中点分割方法:

- ✓ 先求出P0P1的中点Pm。
- ✓ 若P0Pm非显然不可见的，且P0P1在窗口中有可见部分，则距P0最近的可见点一定落在P0Pm上，故用P0Pm代替P0P1；
- ✓ 否则取PmP1代替P0P1；
- ✓ 再对新的P0P1求中点Pm。重复上述过程，直到PmP1长度小于给定的控制常数为止，此时Pm收敛于交点。



中点分割裁剪算法

中点分割方法的特点：

✓算法过程只用到加法和除2运算，所以特别适合硬件实现，同时也适合于并行计算。

直线段的裁剪

多边形的裁剪算法

- ✓ 逐边裁剪算法
- ✓边界裁剪算法

多边形裁剪的特点

- ✓ 问题:用线段裁剪方法处理多边形时，其裁剪后的多边形边界为一系列不连接的直线段，并不是裁剪后有封闭边界的区域。
- ✓ 要求1:对于多边形裁剪，需要能产生一个或多个封闭区域，以便能够进行扫描转换实现区域填充。
- ✓要求2:多边形裁剪后的输出应是定义裁剪后多边形边界的顶点序列。

逐边裁剪算法

用窗口的每一条边界裁剪

- ✓ 对多边形顶点集初始化
- ✓ 首先用窗口左边界裁剪多边形，产生新的顶点集
- ✓ 新的顶点集依次传给窗口的右边界、下边界和上边界处理
- ✓ 每一步产生的新的顶点序列作为下一个窗口边界的输入顶点序列。

逐边裁剪算法

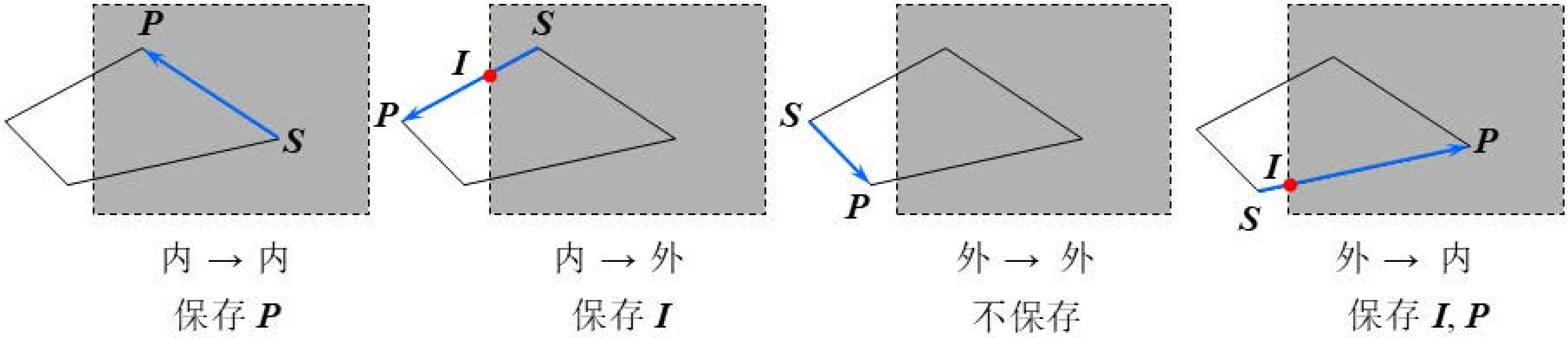
沿着多边形依次处理顶点时会遇到四种情况

✓ 若两点都在窗口边界内，则只将第二点加入到输出顶点集；

✓ 若第一点在窗口边界内，而第二点在窗口边界外，则只有该边与窗口边界的交点加入到输出顶点表中；

✓ 若两点都在窗口边界外，输出顶点表中不增加任何顶点。

✓ 若第一点在窗口边界外而第二点在窗口边界内，则该边与窗口边界的交点和第二点加入到输出顶点集；



逐边裁剪算法

逐边裁剪过程

✓设置裁剪窗口的四条边界为裁剪器，按左、下、右、上的顺序用这些裁剪器处理多边形顶点序列

✓每一裁剪器产生的新的顶点序列作为下一裁剪器的输入顶点序列

✓最后得到的顶点序列即为裁剪后的多边形顶点序列

初始多边顶点序列 → 左裁剪器 → 裁剪后的顶点序列 → 下裁剪器 → 裁剪后的顶点序列 → 右裁剪器 → 裁剪后的顶点序列 → 上裁剪器 → 裁剪多边形顶点序列

逐边裁剪算法

```
typedef struct {
    float x, y
}Vertex；
typedef  Vertex Edge[2];
typedef  Vertex VertexArray[MAX];
```

逐边裁剪算法

```
void Intersect (Vertex S, Vertex P, Edge clipBoundary, Vertex &IntersectPt) {
 //计算由顶点S和P确定的多边形的边与一条裁剪边界的交点，
 //裁剪边界由裁剪窗口的两个顶点定义
   if( clipBoundary[0].y == clipBoundary[1].y) { //水平裁剪边界
     IntersectPt.y = clipBoundary[0].y;
     IntersectPt.x = S.x + (clipBoundary[0].y – S.y)*(P.x – S.x)/(P.y – S.y);
    }
    else { //垂直裁剪边界
         IntersectPt.x = clipBoundary[0].x;
         IntersectPt.y = S.y + (clipBoundary[0].x – S.x)*(P.y – S.y)/(P.x – S.x);
      }
    }
```

逐边裁剪算法

```
boolean Inside(Vertex TestPt, Edge ClipBoundary) {
//顶点在裁剪边界的内侧，函数 Inside( ) 返回值 TRUE。
   if(ClipBoundary[1].x > ClipBoundary[0].x) // 裁剪边界为窗口下边界
      if(TestPt.y >= ClipBoundary[0].y) return TRUE;
   if(ClipBoundary[1].x < ClipBoundary[0].x) // 裁剪边界为窗口上边界
      if(TestPt.y <= ClipBoundary[0].y) return TRUE;
   if(ClipBoundary[1].y > ClipBoundary[0].y) // 裁剪边界为窗口右边界
      if(TestPt.x <= ClipBoundary[0].x) return TRUE;
   if(ClipBoundary[1].y < ClipBoundary[0].y) // 裁剪边界为窗口左边界
      if(TestPt.x >= ClipBoundary[0].x) return TRUE;
   return FALSE;
}
```

逐边裁剪算法

```
void Output(Vertex newVertex, int *outLength, Vertex *OutVertexArray)
{//将一个顶点放入裁剪后的顶点数组OutVertexArray 之中
   (*outLength) ++;
   OutVertexArray[*outLength – 1].x = newVertex.x;
   OutVertexArray[*outLength – 1].y = newVertex.y;
}
```

逐边裁剪算法

```
void SutherlandHodgemanClip(VertexArray InVertexArray, VertexArray
        OutVertexArray, Edge ClipBoundary, int inLength, int *outLength )
{//输入为多边形顶点数组InVertexArray
 //输出为裁剪后的顶点数组OutVertexArray
   Vertex S, P, ip;      int j;    *outLength = 0;
   S = InVertexArray[inLength – 1]; // 从最后一个顶点开始
   for(j = 0; j < inLength; j ++){
      P = InVertexArray[j];
      if(Inside(P, ClipBoundary)){
         if(Inside(S, ClipBoundary)) // S，P 在窗口内
            Output(P, outLength, OutVertexArray);
```

逐边裁剪算法

```
else{//S 在窗口外，P在窗口内
        Intersect(S, P, ClipBoundary, &ip);
        Output(ip, outLrngth, OutVertexArray);
        Output(P, outLength, OutVertexArray);
  }
}
else if(Inside(S, ClipBoundary)){//S 在窗口内，P在窗口外
        Intersect(S, P, ClipBoundary, &ip);
        Output(ip, outLength, OutVertexArray);
} //S，P 在窗口外是没有输出
S = P; //转到下一对顶点继续运行
}
}
```

逐边裁剪算法

凹多边形的裁剪

课后习题

1. 编程实现编码裁剪算法。
2. 有 n 个顶点的凸多边形被一个矩形窗口裁剪