第02章 光栅图形学

光栅显示器上显示的图形，称之为光栅图形。光栅显示器可以看作是一个象素的矩阵，在光栅显示器上显示任何一个图形，实际上都是一些具有一种或多种颜色和灰色象素的集合。由于光栅图形只是近似的实际图形，如何使光栅图形最完美的逼近实际图形，便是光栅图形学要研究的内容。本章将主要介绍光栅图形学的基本算法，包括基本图形元素的扫描转换、区域填充、裁剪、反走样和消隐等。

2.1 直线段的扫描转换算法

数学上的直线是没有宽度、由无数个点构成的集合，显然，光栅显示器只能近地似显示直线。当我们对直线进行光栅化时，需要在显示器有限个象素中，确定最佳逼近该直线的一组象素，并且按扫描线顺序，对这些象素进行写操作，这个过程称为用显示器绘制直线或直线的扫描转换。

由于在一个图形中，可能包含成千上万条直线，所以要求绘制算法应尽可能地快。本节我们介绍一个象素宽直线绘制的三个常用算法：数值微分法（DDA）、中点画线法和Bresenham算法。

（1）数值微分(DDA)法

设过端点$P_0(x_0, y_0)$、$P_1(x_1, y_1)$的直线段为$L(P_0, P_1)$，则直线段L的斜率为$k=(y_1-y_0)/(x_1-x_0)$。要在显示器显示L，必须确定最佳逼近L的像素集合。我们从L的起点P_0的横坐标x_0向L的终点P_1的横坐标x_1步进，取步长=1(个象素)，用L的直线方程$y=kx+b$计算相应的y坐标，并取象素点$(x, \mathrm{round}(y))$作为当前点的坐标。

因为： $y_{i+1} = kx_{i+1}+b = kx_i+b+k\Delta x = y_i+k\Delta x$

所以，当Δx =1； $y_{i+1} = y_i+k$。也就是说，当x每递增1，y递增k(即直线斜率)。根据这个原理，我们可以写出DDA画线算法程序。

DDA画线算法程序：

```
void DDALine(int x0,int y0,int x1,int y1,int color)
{
   int x;
   float dx, dy, y, k;

   dx = x1-x0; dy=y1-y0;
   k=dy/dx,; y=y0;
    for (x=x0; x< x1; x++){
        drawpixel (x, int(y+0.5), color);
        y=y+k;
   }
}
```

注意：我们这里用整型变量color表示象素的颜色和灰度。

举例：用DDA方法扫描转换连接两点P0（0,0）和P1（5,2）的直线段。

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

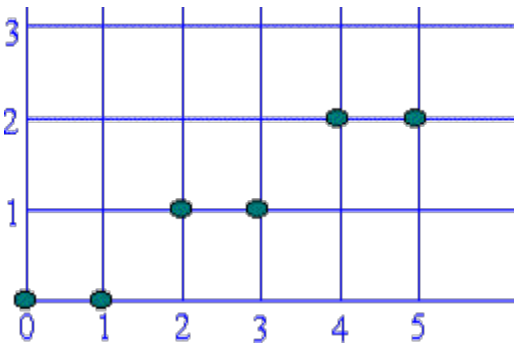


图2.1.1 直线段的扫描转换

注意：上述分析的算法仅适用于$|k| \leq 1$的情形。在这种情况下，x每增加1,y最多增加1。当 $|k| > 1$时，必须把x，y地位互换，y每增加1，x相应增加1/k。在

这个算法中，y与k必须用浮点数表示，而且每一步都要对y进行四舍五入后取整，这使得它不利于硬件实现。

（2）中点画线法

假定直线斜率k在0~1之间，当前象素点为（xp,yp），则下一个象素点有两种可选择点P1（xp+1,yp）或P2（xp+1,yp+1）。若P1与P2的中点（xp+1,yp+0.5）称为M，Q为理想直线与x=xp+1垂线的交点。当M在Q的下方时，则取P2应为下一个象素点；当M在Q的上方时，则取P1为下一个象素点。这就是中点画线法的基本原理。

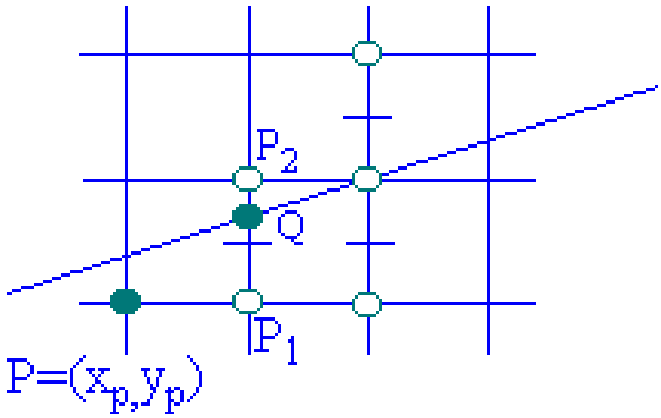


图2.1.2 中点画线法每步迭代涉及的象素和中点示意图

下面讨论中点画线法的实现。过点(x0,y0)、(x1, y1)的直线段L的方程式为F(x, y)=ax+by+c=0，其中，a=y0-y1, b=x1-x0, c=x0y1-x1y0，欲判断中点M在Q点的上方还是下方，只要把M代入F（x，y），并判断它的符号即可。为此，我们构造判别式:

d=F(M)=F(xp+1, yp+0.5)=a(xp+1)+b(yp+0.5)+c

所以：

当d<0时，M在L(Q点)下方，取P2为下一个象素；

当d>0时，M在L(Q点)上方，取P1为下一个象素；

当d=0时，选P1或P2均可，约定取P1为下一个象素；

注意到d是xp, yp的线性函数，可采用增量计算，提高运算效率：

若当前象素处于d>=0情况，则取正右方象素P1(xp+1, yp),要判下一个象素位置，应计算 d1=F(xp+2, yp+0.5)=a(xp+2)+b(yp+0.5)=d+a，增量为a。

若d<0时，则取右上方象素P2(xp+1, yp+1)。要判断再下一象素，则要计算d2= F(xp+2, yp+1.5)=a(xp+2)+b(yp+1.5)+c=d+a+b ，增量为a＋b。画线从(x0, y0)开始，d的初值 d0=F(x0+1, y0+0.5)=F(x0, y0)+a+0.5b，因 F(x0, y0)=0，所以d0=a+0.5b。

由于我们使用的只是d的符号，而且d的增量都是整数，只是初始值包含小数。因此，我们可以用2d代替d来摆脱小数，写出仅包含整数运算的算法程序。

中点画线算法程序：

```
void Midpoint Line (int x0,int y0,int x1, int y1,int color)
{   int a, b, d1, d2, d, x, y;

    a=y0-y1;  b=x1-x0; d=2*a+b;
    d1=2*a; d2=2* (a+b);
    x=x0; y=y0;
    drawpixel(x, y, color);
    while (x<x1){
    if (d<0)   {x++; y++;  d+=d2; }
        else      {x++;  d+=d1;}
        drawpixel (x, y, color);
    } /* while */
} /* mid PointLine */
```

举例：用中点画线方法扫描转换连接两点P0（0,0）和P1（5,2）的直线段。

a=y0-y1=-2; b=x1-x0=5; d0=2*a+b=1;d1=2*a=-4;d2=2*(a+b)=6

x	y	d
0	0	1
1	0	-3
2	1	3
3	1	-1
4	2	5
5	2	15

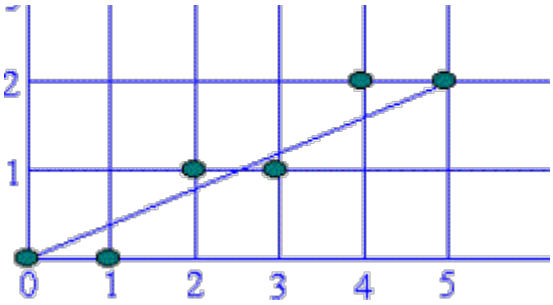


图2.1.3 中点画线法

（3）Bresenham算法

Bresenham算法是计算机图形学领域使用最广泛的直线扫描转换算法。仍然假定直线斜率在0~1之间，该方法类似于中点法，由一个误差项符号决定下一个象素点。

算法原理如下：过各行各列象素中心构造一组虚拟网格线。按直线从起点到终点的顺序计算直线与各垂直网格线的交点，然后确定该列象素中与此交点最近的象素。该算法的巧妙之处在于采用增量计算，使得对于每一列，只要检查一个误差项的符号，就可以确定该列的所求象素。

如图2.1.4所示，设直线方程为$y_{i+1}=y_i+k(x_{i+1}-x_i)+k$。假设列坐标象素已经确定为$x_i$，其行坐标为$y_i$。那么下一个象素的列坐标为$x_i+1$，而行坐标要么为$y_i$，要么递增1为$y_i+1$。是否增1取决于误差项$d$的值。误差项$d$的初值$d_0=0$，$x$坐标每增加1，$d$的值相应递增直线的斜率值$k$，即$d=d+k$。一旦　$d\geq 1$，就把它减去1，这样保证d在0、1之间。当$d\geq 0.5$时，直线与垂线$x=x_i+1$交点最接近于当前象素(x_i, y_i)的右上方象素(x_i+1, y_i+1)；而当$d<0.5$时，更接近于右方象素(x_i+1, y_i)。为方便计算，令$e=d-0.5$，e的初值为-0.5，增量为k。当$e\geq 0$时，取当前象素(x_i, y_i)的右上方象素(x_i+1, y_i+1)；而当$e<0$时，取(x_i, y_i)右方象素(x_i+1, y_i)。

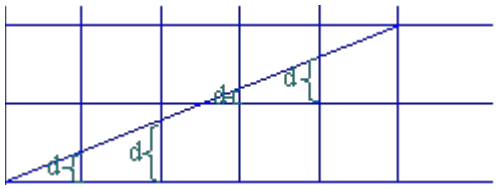


图2.1.4 Bresenham算法所用误差项的几何含义

Bresenham画线算法程序：

```
void Bresenhamline (int x0,int y0,int x1, int y1,int color)
{
    int x, y, dx, dy;
```

```
    float k, e;

    dx = x1-x0; dy = y1- y0; k=dy/dx;
    e=-0.5;  x=x0,; y=y0;
    for (i=0; i<dx; i++)
    {   drawpixel (x, y, color);
        x=x+1; e=e+k;
        if (e >= 0) { y++;  e=e-1;}
    }
}
```

★举例：用Bresenham方法扫描转换连接两点P0（0,0）和P1（5,2）的直线段。

x	y	e
0	0	-0.5
1	0	-0.1
2	1	-0.7
3	1	-0.3
4	2	-0.9
5	2	-0.5

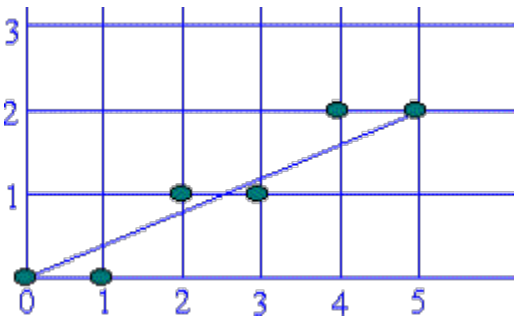


图2.1.5 Bresenham算法

上述Bresenham算法在计算直线斜率与误差项时用到小数与除法。可以改用整数以避免除法。由于算法中只用到误差项的符号，因此可作如下替换：2*e*dx。

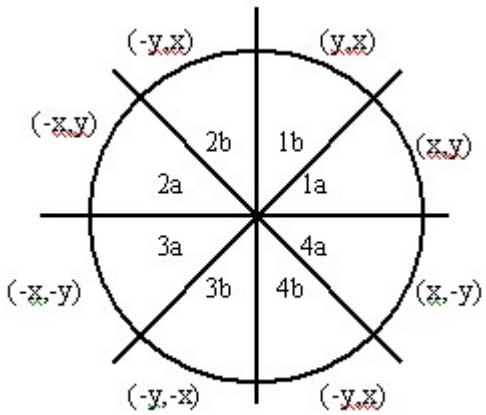
改进的Bresenham画线算法程序：

```
void InterBresenhamline (int x0,int y0,int x1, int y1,int color)
{
    int x, y, dx, dy,e;

    dx = x1-x0,; dy = y1- y0,; e=-dx;
    x=x0;  y=y0;
    for (i=0; i<dx; i++){
        drawpixel (x, y, color);
        x++;  e=e+2*dy;
        if (e >= 0) { y++; e=e-2*dx;}
    }
}
```

2.2 圆弧的扫描转换算法

（1）圆的特征

圆被定义为到给定中心位置(xc,yc)距离为r的点集。圆心位于原点的圆有四条对称轴x=0,y=0,x=y和x=-y。若已知圆弧上一点(x,y),可以得到其关于四条对称轴的其它7个点，这种性质称为圆的八对称性。因此，只要扫描转换八分之一圆弧，就可以求出整个圆弧的象素集。



圆心在0,0点圆周生成时的对称变换（参见：P.26图3-4）

显示圆弧上的八个对称点的算法：

```
void CirclePoints(int x,int y,int color)
{
    drawpixel(x,y,color); drawpixel(y,x,color);
    drawpixel(-x,y,color); drawpixel(y,-x,color);
    drawpixel(x,-y,color); drawpixel(-y,x,color);
    drawpixel(-x,-y,color); drawpixel(-y,-x,color);
}
```

（2）中点画圆法（参见：P.29—P.30）

如果我们构造函数 $F(x, y)=x^2+y^2-R^2$，则对于圆上的点有$F(x, y)=0$，对于圆外的点有$F(x, y)>0$，对于圆内的点$F(x, y)<0$ 。与中点画线法一样，构造判别式：

$$d=F(M)=F(x_p+1, y_p-0.5)=(x_p+1)^2+(y_p-0.5)^2-R^2$$

若 $d<0$，则应取P_1为下一象素，而且再下一象素的判别式为：

$$d=F(x_p+2, y_p-0.5)=(x_p+2)^2+(y_p-0.5)^2-R^2=d+2x_p+3$$

若$d\geqslant 0$，则应取P_2为下一象素，而且下一象素的判别式为

$$d=F(x_p+2, y_p-1.5)=(x_p+2)^2+(y_p-1.5)^2-R^2=d+2(x_p-y_p)+5$$

我们这里讨论的第一个象素是（0, R），判别式d的初始值为：

$$d_0=F(1, R-0.5)=1.25-R$$

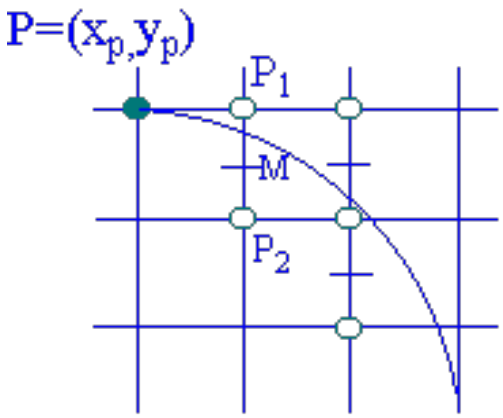


图2.2.1 当前象素与下一象素的候选者

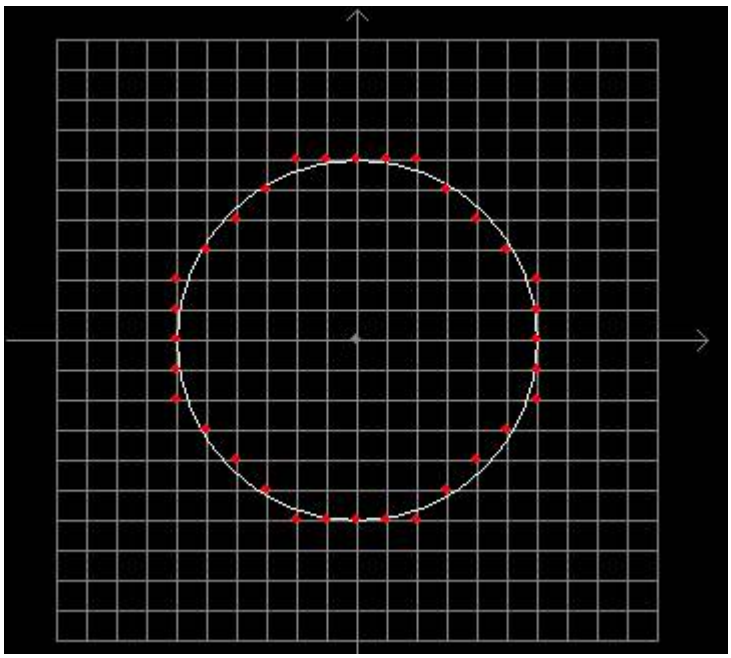
中点画圆算法：

```
MidPointCircle(int r int color)
{
    int x,y;
    float d;

    x=0; y=r; d=1.25-r;
    circlepoints (x,y,color);
    while(x<=y)
    {
```

```
        if(d<0)    d+=2*x+3;
        else    { d+=2*(x-y)+5; y--;}
        x++;
        circlepoints (x,y,color);
    }
}
```

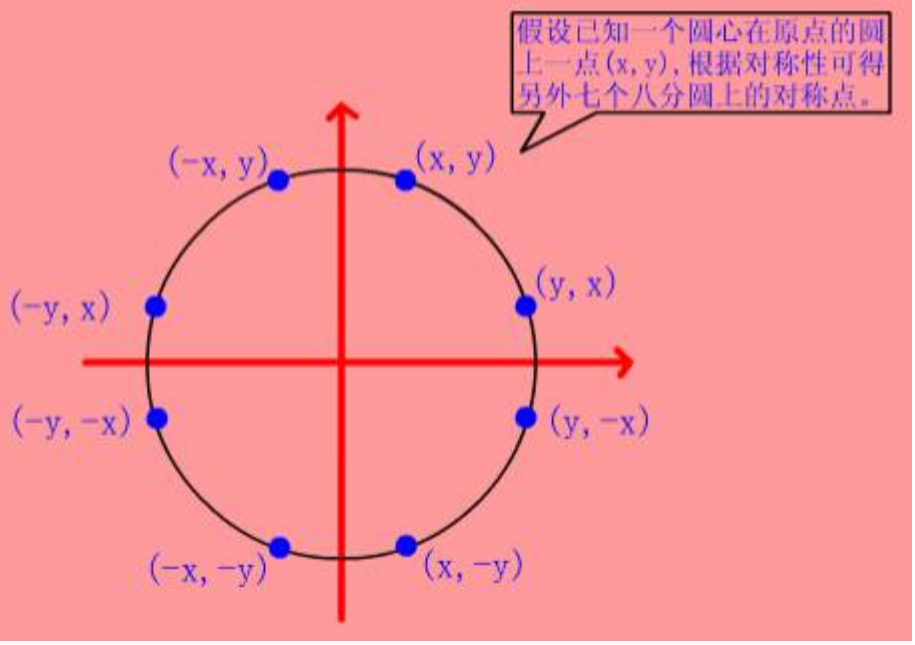
为了进一步提高算法的效率，可以将上面的算法中的浮点数改写成整数，将乘法运算改成加法运算，即仅用整数实现中点画圆法。

（3）例：半径为6，利用中点画圆法算法所画的点

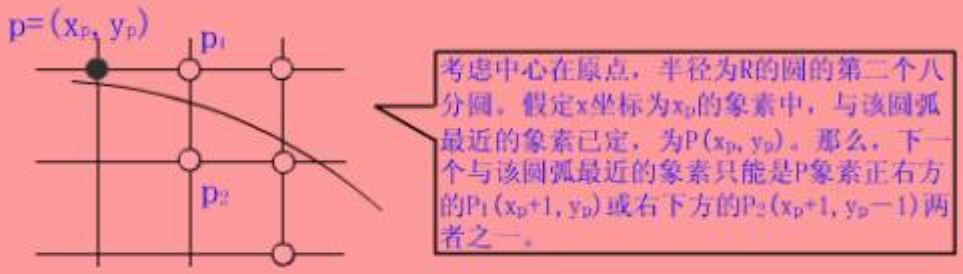


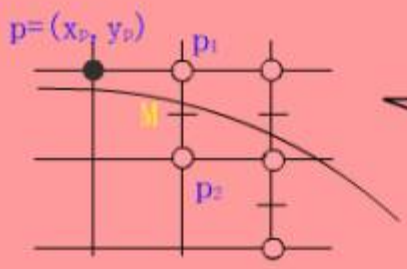
（4）中点画圆法图示

一：圆的对称性



二：中点画圆算法判别式的构造





构造函数：　　$F(x,y)=x^2+y^2-R^2$

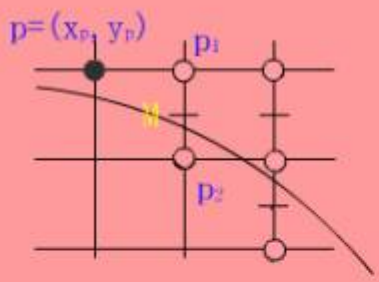
假设M是P_1和P_2的中点，即$M=(x_p+1, y_p-0.5)$
那么，当$F(M)<0$时，M在圆内，这说明P1距离圆弧更近，应取P1作为下一象素。

构造判别式为：$d=F(M)=F(X_P+1,Y_P-0.5)=(X_P+1)^2+(Y_P-0.5)^2-R^2$

若$d<0$, 则应取P1为下一象素，而且再下一个象素的判别式为

$$d=F(X_P+2,Y_P-0.5)=(X_P+2)^2+(Y_P-0.5)^2-R^2=d+2x_p+3$$

所以，沿正右方向，d 的增量为$2x_p+3$。

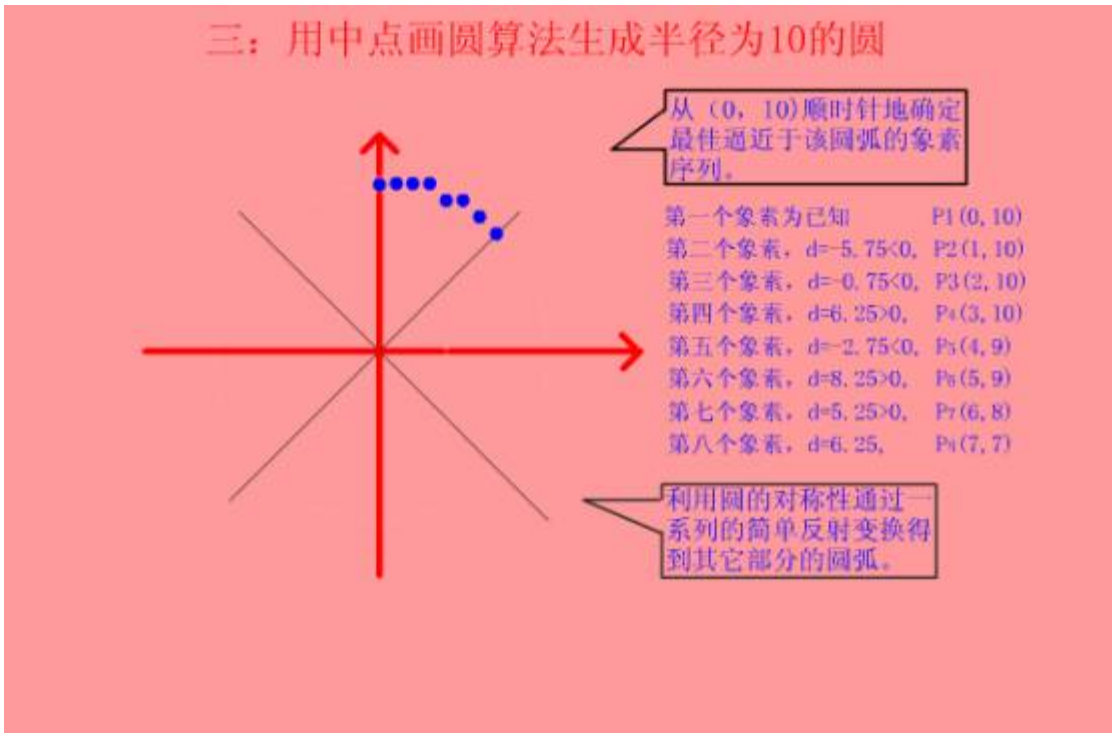


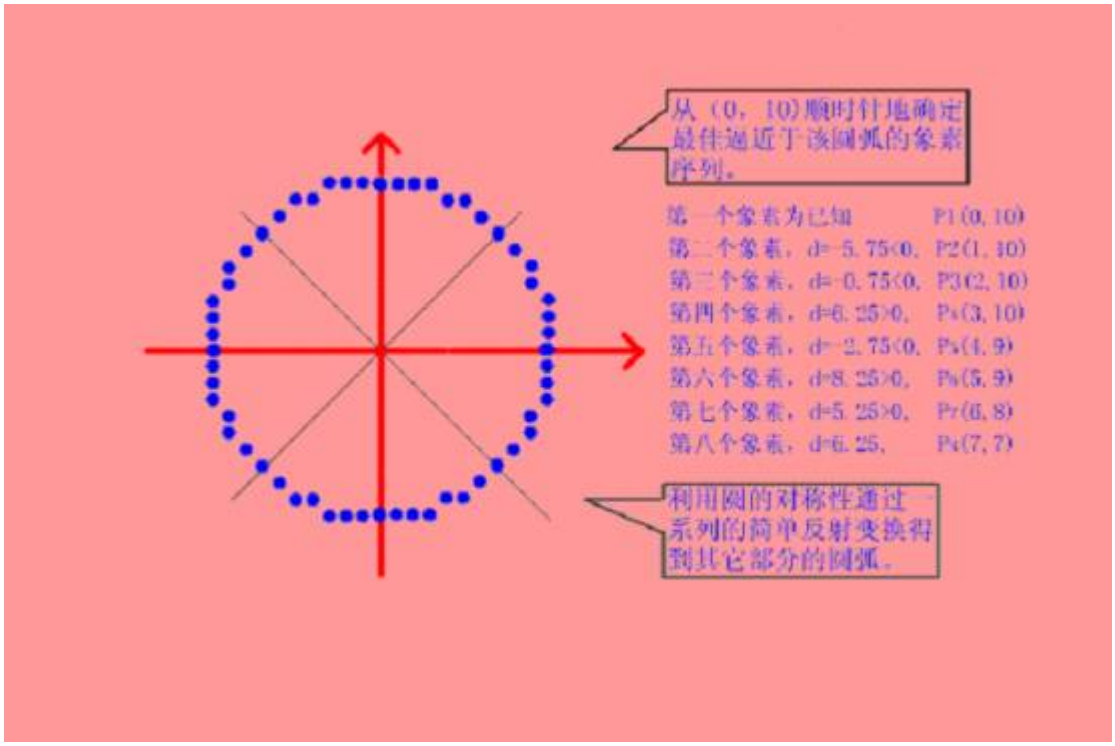
此时$F(M)>0$时，M在圆外，P_2距离圆弧更近，则应取P_2。当$F(M)=0$时，P_1与P_2任取一个；我们约定取P_2。

而若d>=0, 则P2是下一象素，而且下一象素的判别式为

$$d=F(x_p+2,y_p-1.5)=(x_p+2)^2+(y_p-1.5)^2-R^2$$
$$=d+(2x_p+3)+(-2y_p+2)$$

所以，沿右下方向，判别式d 的增量为$2(x_p-y_p)+5$。





2.3 多边形的扫描转换与区域填充

在计算机图形学中，多边形有两种重要的表示方法：顶点表示和点阵表示。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象素在多边形内，故不能直接用于面着色；点阵表示是用位于多边形内的象素集合来刻画多边形。这种表示丢失了许多几何信息，但便于帧缓冲器表示图形，是面着色所需要的图形表

示形式。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。

区域填充则是指先将在点阵表示的多边形区域内的一点（称为种子点）赋予指定的颜色和灰度，然后将这种颜色和灰度扩展到整个区域内的过程。

（1）多边形的扫描转换

多边形扫描转换算法对多边形的形状没有限制，但多边形的边界必须是封闭的，且不自交。我们可以将多边形分为三种：凸多边形、凹多边形、含内环的多边形。

- 凸多边形是指任意两顶点间的连线均在多边形内；
- 凹多边形是指任意两顶点间的连线有不在多边形内的部分；
- 含内环的多边形则是指多边形内再套有多边形，多边形内的多边形也叫内环，内环之间不能相交。

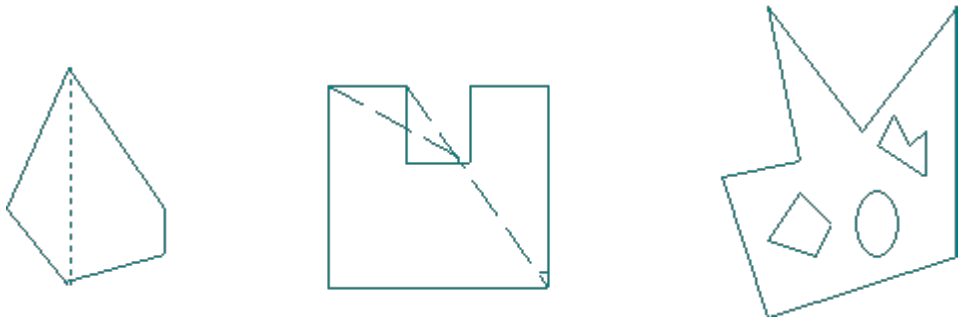


图2.3.1 多边形的种类

（a）扫描线算法

扫描线算法是按扫描线顺序，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象素，完成转换工作。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。对于一条扫描线，多边形的扫描转换过程可以分为四个步骤：

- 求交：计算扫描线与多边形各边的交点；
- 排序：把所有交点按x值递增顺序排序；
- 配对：第一个与第二个，第三个与第四个等等；每对交点代表扫描线与多边形的一个相交区间，
- 着色：把相交区间内的象素置成多边形颜色，把相交区间外的象素置成背景色。

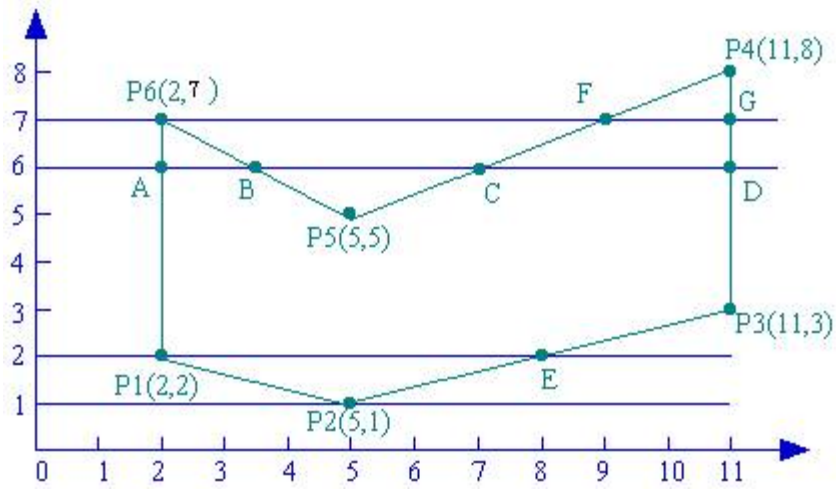
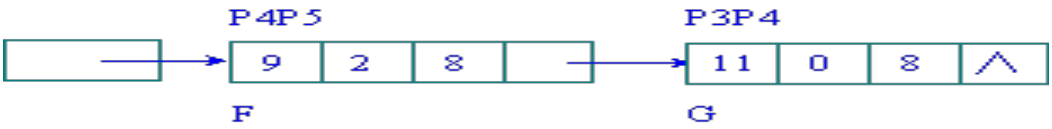


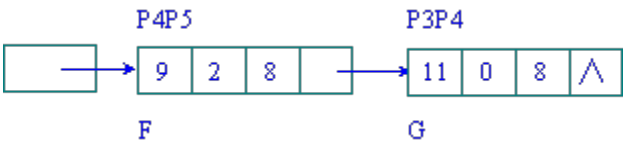
图2.3.2 一个多边形与若干扫描线

为了提高效率，在处理一条扫描线时，仅对与它相交的多边形的边进行求交运算。我们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，称此链表为活性边表(AET)。

活性边表的结点应为对应边保存如下内容：第1项存当前扫描线与边的交点坐标x值；第2项存从当前扫描线到下一条扫描线间x的增量Dx；第3项存该边所交的最高扫描线号y_{max}。



a. 扫描线6的活性边表



b. 扫描线7的活性边表

图2.3.3活性边表(AET)

假定当前扫描线与多边形某一条边的交点的横坐标为x，则下一条扫描线与该边的交点不必要重计算，只要加一个增量△x即可，下面，我们推导这个结论。

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

$ax_i+by_i+c=0$

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

由此得: $x_{i+1}=\frac{1}{a}(-b\cdot y_{i+1}-c)$

由于$y_{i+1}=y_i+1$,所以$x_{j+1}=\frac{1}{a}(-b\cdot y_{j+1}-c_j)=x_j-\frac{b}{a}$;

其中△x=-b/a 为常数，

另外使用增量法计算时，我们需要知道一条边何时不再与下一条扫描线相交，以便及时把它从活性边表中删除出去。

为了方便活性边表的建立与更新，我们为每一条扫描线建立一个新边表（NET）,存放在该扫描线第一次出现的边。也就是说，若某边的较低端点为y_{min}，则该边就放在扫描线y_{min}的新边表中。

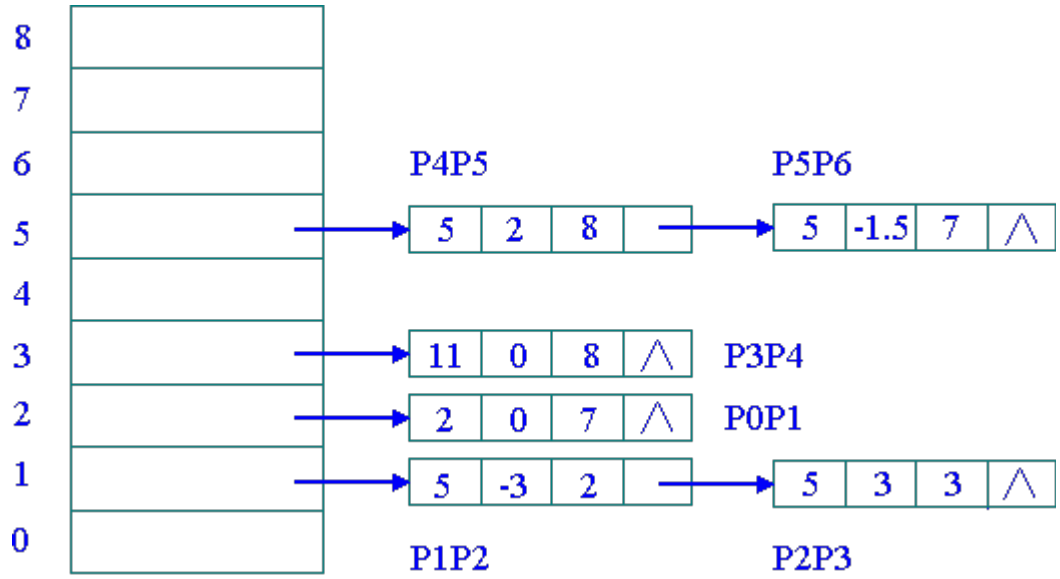


图2.3.4 上图所示各条扫描线的新边表NET

算法过程：

```
void polyfill (polygon, color)
int color;多边形 polygon;
{
    for (各条扫描线i )
    {    初始化新边表头指针NET [i];
    把y min = i 的边放进边表NET [i];
     }

    y = 最低扫描线号;
     初始化活性边表AET为空;
```

```
    for (各条扫描线i )
    {    把新边表NET[i]中的边结点用插入排序法插入AET表，使之按x坐标递增顺序排列;
         遍历AET表，把配对交点区间(左闭右开)上的象素(x, y),
                                  用drawpixel (x, y, color) 改写象素颜色值;
         遍历AET表，把ymax= i 的结点从AET表中删除，并把ymax > i结点的x值递增Dx;
         若允许多边形的边自相交，则用冒泡排序法对AET表重新排序;
    }
} /* polyfill */
```

扫描线与多边形顶点相交时，必须正确地取舍交点，如图2.3.5所示。

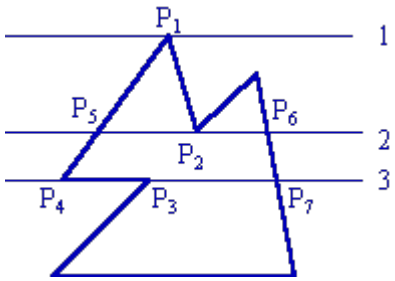


图2.3.5 扫描线与多边形相交，特殊情况的处理

✍ 扫描线与多边形相交的边分别位于扫描线的两侧，则计一个交点，如点P_5，P_6。

✍ 扫描线与多边形相交的边分别位于扫描线同侧，且 $y_i<y_i-1$，$y_i<y_i+1$，则计2个交点(填色)，如P_2。若 $y_i>y_i-1$，$y_i>y_i+1$，则计0个交点(不填色)，如P_1。

✍ 扫描线与多边形边界重合（当要区分边界和边界内区域时需特殊处理)，则计1个交点。

具体实现时，只需检查顶点的两条边的另外两个端点的y值。按这两个y值中大于交点y值的个数是0,1,2来决定。

（b）边界标志算法

边界标志算法的基本思想是：在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。对每条与多边形相交的扫描线依从左到右的顺序，逐个访问该扫描线上的象素。使用一个布尔量inside来指示当前点是否在多边形内的状态。Inside的初值为假，每当当前访问的象素为被打

上边标志的点，就把inside取反。对未打标志的象素，inside不变。若访问当前象素时，inside为真，说明该象素在多边形内，则把该象素置为填充颜色。

边界标志算法：

```
void edgemark_fill(polydef, color)
多边形定义 polydef;  int color;
{  对多边形polydef 每条边进行直线扫描转换;
   inside = FALSE;
   for (每条与多边形polydef相交的扫描线y )
   for (扫描线上每个象素x )
   { if(象素 x 被打上边标志)inside = ! (inside);
     if(inside! = FALSE)     drawpixel (x, y, color);
     else    drawpixel (x, y, background);
   }
}
```

用软件实现时，扫描线算法与边界标志算法的执行速度几乎相同，但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）区域填充算法

这里讨论的区域指已经表示成点阵形式的填充图形，它是象素的集合。区域可采用内点表示和边界表示两种表示形式。

内点表示：区域内的所有象素着同一颜色。

边界表示：区域的边界点着同一颜色。区域填充指先将区域的一点赋予指定的颜色，然后将该颜色扩展到整个区域的过程。

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。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。

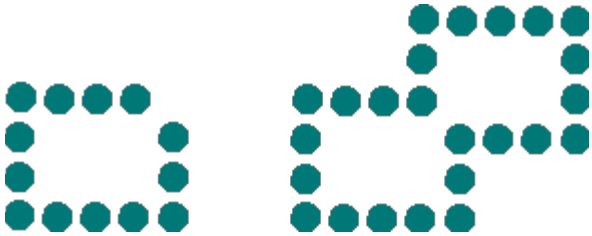


图2.3.7 四连通区域和八连通区域

图2.3.8 区域的内点表示和边界表示

（a）区域填充的递归算法

以上讨论的多边形填充算法是按扫描线顺序进行的。种子填充算法假设在多边形内有一象素已知，由此出发利用连通性找到区域内的所有象素。

设(x,y)为内点表示的4连通区域内的一点，oldcolor为区域的原色，要将整个区域填充为新的颜色newcolor。

内点表示的4连通区域的递归填充算法:

```
void FloodFill4(int x,int y,int oldcolor,int newcolor)
{
        if(getpixel(x,y)==oldcolor)
    {
                drawpixel(x,y,newcolor);
                FloodFill4(x,y+1,oldcolor,newcolor);
                FloodFill4(x,y-1,oldcolor,newcolor);
                FloodFill4(x-1,y,oldcolor,newcolor);
```

```
            FloodFill4(x+1,y,oldcolor,newcolor);
    }
}
```

边界表示的4连通区域的递归填充算法:

```
void BoundaryFill4(int x,int y,int boundarycolor,int newcolor)
{ int color;
  if(color!=newcolor && color!=boundarycolor)
  { drawpixel(x,y,newcolor);
    BoundaryFill4 (x,y+1, boundarycolor,newcolor);
    BoundaryFill4 (x,y-1, boundarycolor,newcolor);
    BoundaryFill4 (x-1,y, boundarycolor,newcolor);
    BoundaryFill4 (x+1,y, boundarycolor,newcolor);
  }
}
```

对于内点表示和边界表示的8连通区域的填充，只要将上述相应代码中递归填充相邻的4个象素增加到递归填充8个象素即可。

2.4 字符

字符：数字、字母、汉字，计算机中字符由一个数字编码唯一标识。

字符集：

- ASCII码，国际上最流行的字符集是“美国信息交换用标准代码集”简称ASCII码。它是用7位二进制数进行编码表示128个字符，包括字母、标点、运算符以及一些特殊符号。
- 汉字编码，其国家标准字符集GB2312－80。该字符集分为94个区，94个位，每个符号由一个区码和一个位码共同标识。区码和位码各用一个字节表示。为了能够区分ASCII码与汉字编码，采用字节的最高位来标识：最高位为0表示ASCII码；最高位为1表示表示汉字编码。

字库：为了在显示器等输出设备上输出字符，系统中必须装备有相应的字库。字库中存储了每个字符的形状信息，字库分为矢量和点阵型两种。

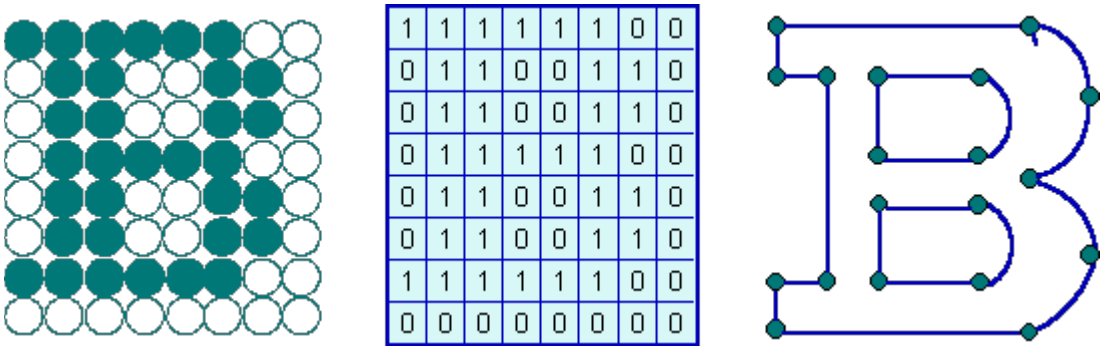
（1）点阵字符

在点阵字符库中，每个字符由一个位图表示。该位为1表示字符的笔画经过此位，对应于此位的象素应置为字符颜色。该位为0表示字符的笔画不经过此位，对应于此位的象素应置为背景颜色。

在实际应用中，有多种字体（如宋体、楷体等），每种字体又有多种大小型号，因此字库的存储空间是很庞大的。解决这个问题一般采用压缩技术。如：黑白段压缩；部件压缩；轮廓字形压缩等。其中，轮廓字形法压缩比大，且能保证字符质量，是当今国际上最流行的一种方法。轮廓字形法采用直线或二/三次bezier曲线的集合来描述一个字符的轮廓线。轮廓线构成一个或若干个封闭的平面区域。轮廓线定义加上一些指示横宽、竖宽、基点、基线等等控制信息就构成了字符的压缩数据。

点阵字符的显示分为两步：

- 从字库中将它的位图检索出来
- 将检索到的位图写到帧缓冲器中。



点阵字符　　点阵字库中的位图表示　　矢量轮廓字符

图2.4.1 字符的种类

（2）矢量字符

矢量字符记录字符的笔画信息而不是整个位图，具有存储空间小，美观、变换方便等优点。对于字符的旋转、缩放等变换，点阵字符的变换需要对表示字符位图中的每一象素进行；而矢量字符的变换只要对其笔画端点进行变换就可以了。

矢量字符的显示也分为两步：

- 从字库中将它的字符信息；
- 然后取出端点坐标，对其进行适当的几何变换，再根据各端点的标志显示出字符。

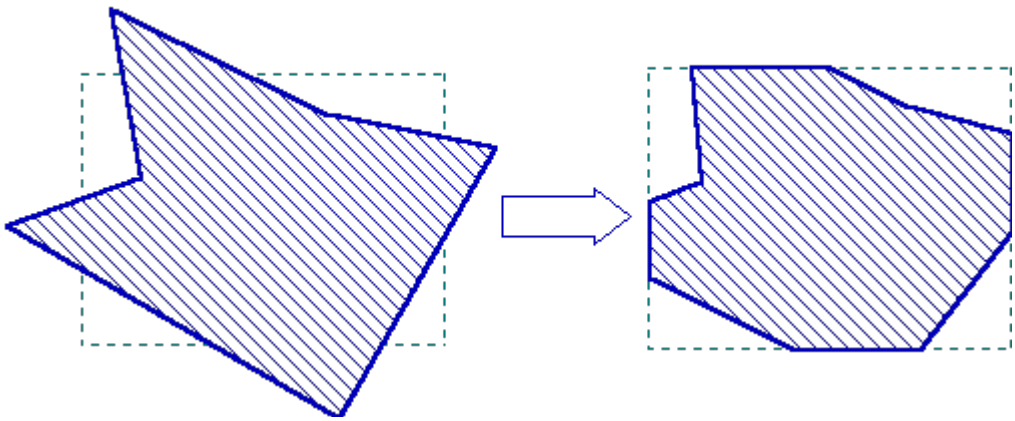
（3）字符属性

- 字体 宋体 仿宋体 楷体 **黑体** ~~隶书~~
- 字高 宋体 宋体 宋体 宋体
- 字宽因子(扩展/压缩) 大海 大海 大海 大海
- 字倾斜角 倾斜 *倾斜*
- 对齐（左对齐、中心对齐、右对齐）
- 字色

2.5 裁剪

屏幕显示的只是图的一部分，而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。

最简单的裁剪方法是把各种图形扫描转换为点之后，再判断各点是否在窗内。但那样太费时，一般不可取。这是因为有些图形组成部分全部在窗口外，可以完全排除，不必进行扫描转换。所以一般采用先裁剪再扫描转换的方法。



(a)裁剪前　　(b) 裁剪后

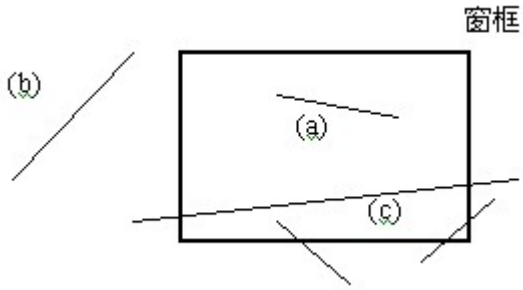
图2.5.1 多边形裁剪

（1）直线段裁剪

曲线可以通过折线段来近似，从而裁剪问题也可以化为直线段的裁剪问题。因此，直线段的裁剪是复杂图元裁剪的基础。常用的线段裁剪方法有三种：Cohen-Sutherland,中点分割算法和梁友栋－barskey算法。这里仅讨论Cohen-Sutherland算法。

直线和窗口的关系可以分为如下三类：

- 整条直线在窗口之内。此时，不需剪裁，显示整条直线。
- 整条直线在窗口之外，此时，不需剪裁，不显示整条直线。
- 部分直线在窗口之内，部分在窗口之外。此时，需要求出直线与窗框之交点，并将窗口外的直线部分剪裁掉，显示窗口内的部分。

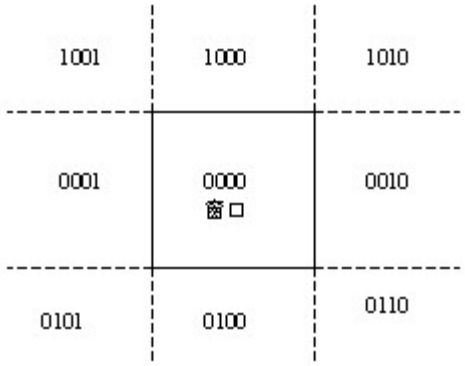


直线与窗口的关系图

直线剪裁算法有两个主要步骤：

- 将不需剪裁的直线挑出，并删去其中在窗外的直线。
- 对其余直线，逐条与窗框求交点，并将窗外部分删去。

Cohen-Sutherland直线剪裁算法以区域编码为基础，将窗口及其周围的八个方向以4 bit的二进制数进行编码。4个bit分别代表窗外上、下、右、左空间的编码值。如左上区域编码为1001，右上区域编码为1010。窗内编码为0000，如图所示。



直线剪裁算法中的区域编码

上述编码方法将窗口及其邻域分为5个区域：

- 内域：区域（0000）；
- 上域：区域（1001，1000，1010）；
- 下域：区域（0101，0100，0110）；
- 左域、区域（1001，0001，0101）；
- 右域：区域（1010，0010，0110）

这就带来二个优点：

- 容易将不需剪裁的直线挑出。规则是：如果一条直线的两端在同一区域，则该直线不需剪裁，否则，该直线为可能剪裁直线。
- 对可能剪裁的直线缩小了与之求交的边框范围。规则是：如果直线的一个端点在上（下，左，右）域，则此直线与上边框求交，然后删去上边框以上的部分。该规则对直线的另一端点也适用。这样，一条直线至多只需与两条边框求交。

该算法的思想是对于每条线段P_1P_2分为三种情况处理：

[1]若P_1P_2完全在窗口内，则显示该线段P_1P_2简称“取”之。

[2]若P_1P_2明显在窗口外，则丢弃该线段，简称“弃”之。

[3]若线段既不满足“取”的条件，也不满足“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。然后对另一段重复上述处理。

为使计算机能够快速判断一条直线段与窗口属何种关系，采用如下编码方法。延长窗口的边，将二维平面分成九个区域。每个区域赋予4位编码$C_tC_bC_rC_l$。其中各位编码的定义如下：

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

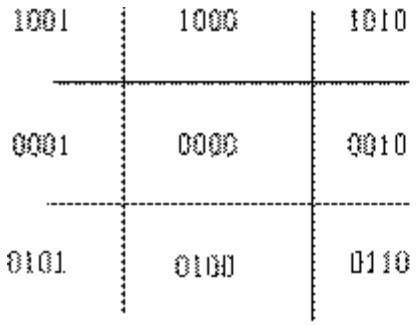


图2.5.2 多边形裁剪区域编码

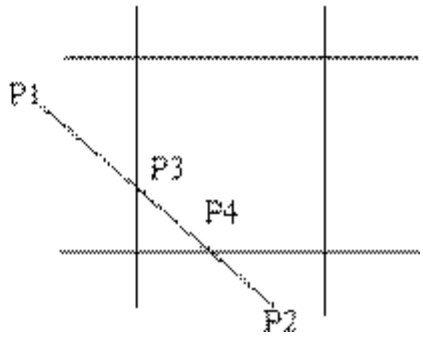


图2.5.3 线段裁剪

裁剪一条线段时，先求出P_1P_2所在的区号code1,code2。

- 若code1=0,且code2=0，则线段P_1P_2在窗口内，应取之。
- 若按位与运算code1&code2≠0，则说明两个端点同在窗口的上方、下方、左方或右方。可判断线段完全在窗口外，可弃之。
- 否则，按第三种情况处理。求出线段与窗口某边的交点，在交点处把线段一分为二，其中必有一段在窗口外，可弃之。在对另一段重复上述处理。在实现本算法时，不必把线段与每条窗口边界依次求交，只要按顺序检测到端点的编码不为0，才把线段与对应的窗口边界求交。

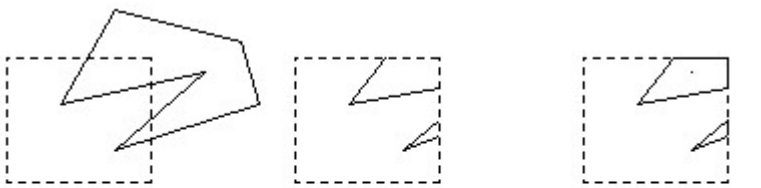
Cohen-Sutherland裁减算法

```
#define LEFT 1
#define RIGHT 2
#define BOTTOM 4
#define TOP 8
int encode(float x,float y)
{
    int c=0;
    if(x<XL) c|=LEFT;
    if(x>XR) c|=RIGHT;
    if(y<YB) c|=BOTTOM;
    if(y<YT) c|=TOP;
    retrun c;
}
void  CS_LineClip(x1,y1,x2,y2,XL,XR,YB,YT)
float x1,y1,x2,y2,XL,XR,YB,YT;
//(x1,y1)(x2,y2)为线段的端点坐标，其他四个参数定义窗口的边界
{    int code1,code2,code;
      code1=encode(x1,y1);
      code2=encode(x2,y2);
      while(code1!=0 ||code2!=0)
      { if(code1&code2 !=0) return;
```

```
            code = code1;
            if(code1==0) code = code2;
            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);
        }

           if(code ==code1)

        {  x1=x;y1=y; code1 =encode(x,y);}
        else
        { x2=x;y2=y; code2 =encode(x,y);}

    }
     displayline (x1,y1,x2,y2);
}
```

（2）多边形裁剪

多边形的剪裁比直线剪裁复杂。如果套用直线剪裁算法对多边形的边作剪裁的话，剪裁后的多边形之边就会成为一组彼此不连贯的折线，从而给填色带来困难（图2.6.3(b)）。多边形剪裁算法的关键在于：通过剪裁，不仅要保持窗口内多边形的边界部分，而且要将窗框的有关部分按一定次序插入多边形之保留边界之间，从而使剪裁后的多边形之边仍旧保持封闭状态，填色算法得以正确实现（图2.6.3(c)）。



(a)剪裁的多边形　　(b)按直线剪裁的多边形　　(c)按多边形剪裁后的多边形

图2.6.3 多边形剪裁

下面介绍Sutherland-Hodgeman算法，该算法的基本思想是一次用窗口的一条边裁剪多边形。

算法的每一步，考虑窗口的一条边以及延长线构成的裁剪线。该线把平面分成两个部分：一部分包含窗口，称为可见一侧；另一部分称为不可见一侧。依序考虑多边形的各条边的两端点S、P。它们与裁剪线的位置关系只有四种。(1)S、P均在可见一侧；(2)S、P均在不可见一侧；(3)S可见,P不可见；(4)S不可见,P可见。

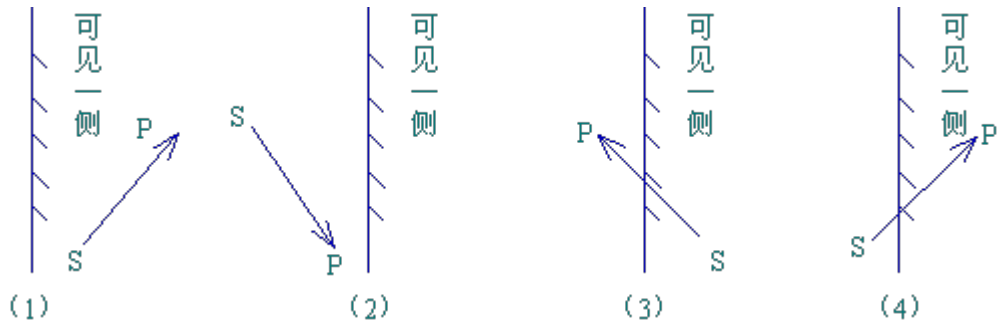


图2.5.5 S、P与裁剪线的四种位置关系

每条线段端点S、P与裁剪线比较之后，可输出0至两个顶点。对于情况(1)仅输出顶点P；情况(2)输出0个顶点；情况(3)输出线段SP与裁剪线的交点I；情况(4)输出线段SP与裁剪线的交点I和终点P。

上述算法仅用一条裁剪边对多边形进行裁剪，得到一个顶点序列，作为下一条裁剪边处理过程的输入。对于每一条裁剪边，算法框图一样，只是判断点在窗口哪一侧以及求线段SP与裁剪边的交点算法应随之改变。

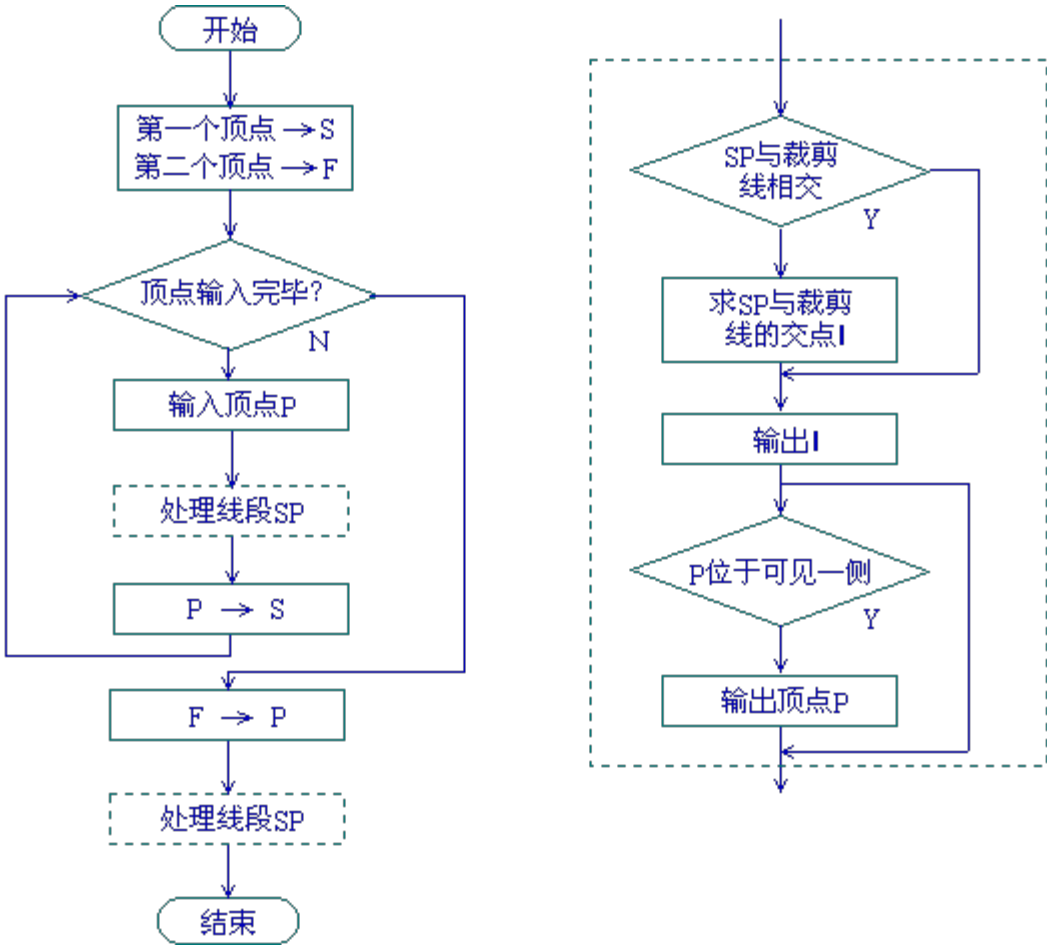


图2.5.6 仅用一条裁剪边时，逐次多边形裁剪算法框图

基本思想：

- 1）令多边形的顶点按边线逆时针走向排序：p1，p2，…，pn。如图2.6.4(a)。各边先与上窗边求交。求交后删去多边形在窗之上的部分，并插入上窗边及其延长线的交点之间的部分（图2.6.4(b)中的（3，4）），从而形成一个新的多边形。然后，新的多边形按相同方法与右窗边相剪裁。如此重复，直至与各窗边都相剪裁完毕。图2.6.4(c)、(d)、(e)示出上述操作后所生成的新多边形的情况。

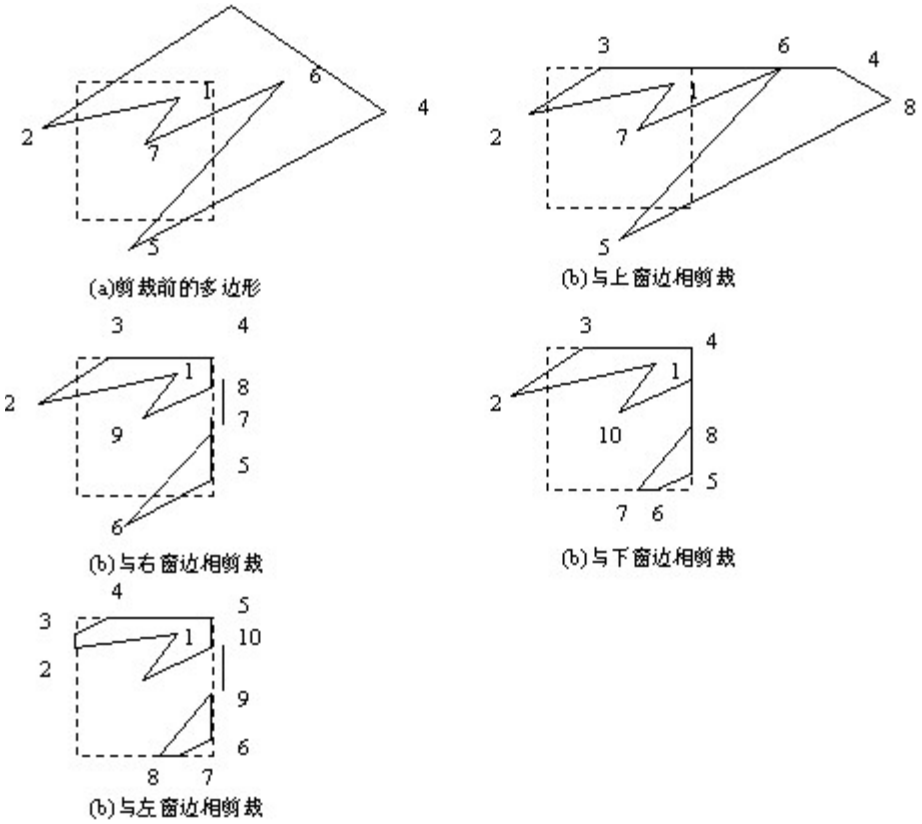


图2.6.4 多边形剪裁的步骤

- 2）多边形与每一条窗边相交，生成新的多边形顶点序列的过程，是一个对多边形各顶点依次处理的过程。设当前处理的顶点为p，先前顶点为s，多边形各顶点的处理规则如下：
  - 如果s，p均在窗边之内侧，那么，将p保存。
  - 如果s在窗边内侧，p在外侧，那么，求出sp边与窗边的交点I，保存I，舍去p。
  - 如果s，p均在窗边之外侧，那么，舍去p。
  - 如果s在窗边之外侧，p在内侧，那么，求出sp边与窗边的交点I，依次保存I和p。

上述四种情况在图2.6.5(a)、(b)、(c)、(d)中分别示出。基于这四种情况，可以归纳对当前点p的处理方法为：1、p在窗边内侧，则保存p；否则不保存。2、p和s在窗边非同侧，则求交点I，并将I保存，并插入p之前，或s之后。

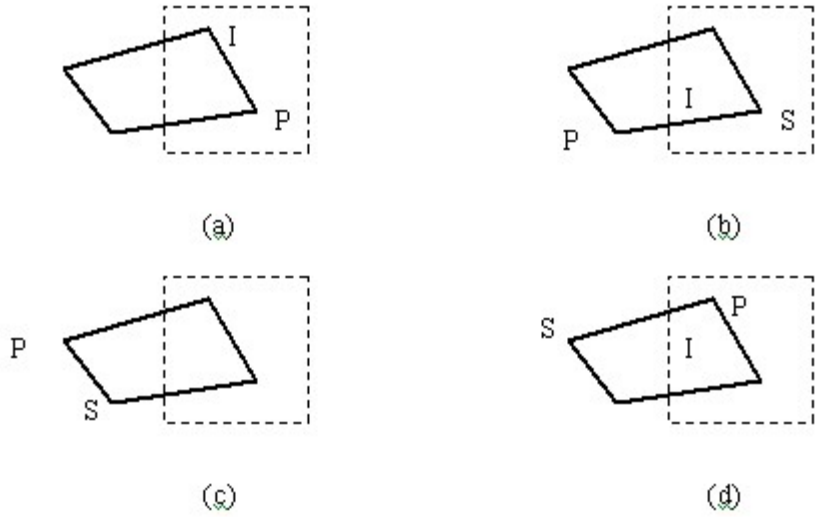


图2.6.5 多边形的新顶点序列的生成规则

（3）字符裁剪

当字符和文本部分在窗口内，部分在窗口外时，就提出了字符裁剪问题。

字符串裁剪可按三个精度来进行：串精度、字符精度、以及笔画\象素精度。

采用串精度进行裁剪时，将包围字串的外接矩形对窗口作裁剪。当字符串方框整个在窗口内时予以显示，否则不显示。

采用字符精度进行裁剪时，将包围字的外接矩形对窗口作裁剪，某个字符方框整个落在窗口内予以显示，否则不显示。

采用笔画/象素精度进行裁剪时，将笔划分解成直线段对窗口作裁剪，处理方法同上。

(a)待裁剪字符串　(b)串精度裁剪　(c)字符精度裁剪　(d)象素精度裁剪

图2.5.7 字符裁剪

2.6 反走样

在光栅显示器上显示图形时，直线段或图形边界或多或少会呈锯齿状。

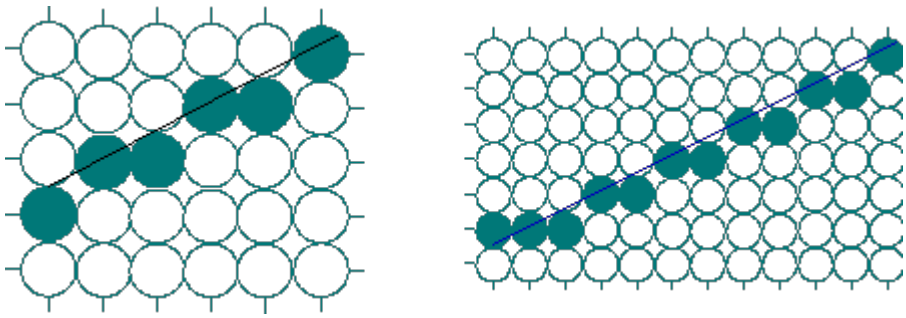
原因：图形信号是连续的，而在光栅显示系统中，用来表示图形的却是一个个离散的象素。这种用离散量表示连续量引起的失真现象称之为走样(aliasing)；用于减少或消除这种效果的技术称为反走样(antialiasing)。

光栅图形的走样现象除了阶梯状的边界外，还有图形细节失真(图形中的那些比象素更窄的细节变宽)，狭小图形遗失等现象。

常用的反走样的主要方法：提高分辨率、区域采样和加权区域采样。

（1）提高分辨率

把显示器分辨率提高一倍，直线经过两倍的象素，锯齿也增加一倍，但同时每个阶梯的宽度也减小了一倍，所以显示出的直线段看起来就平直光滑了一些。这种反走样方法是以4倍的存储器代价和扫描转换时间获得的。因此，增加分辨率虽然简单，但是不经济的方法，而且它也只能减轻而不能消除锯齿问题。



(a)用中点算法扫描转换的一条直线 (b)把显示器分辨率提高一倍后的结果

图2.6.1 不同分辨率下的直线显示

（2）区域采样

区域采样方法假定每个象素是一个具有一定面积的小区域，将直线段看作具有一定宽度的狭长矩形。当直线段与象素有交时，求出两者相交区域的面积，然后根据相交区域面积的大小确定该象素的亮度值。假设一条直线段的斜率为m(0≤m≤1)，且所画直线为一个象素单位，则直线段与象素相交有五种情况，见图。

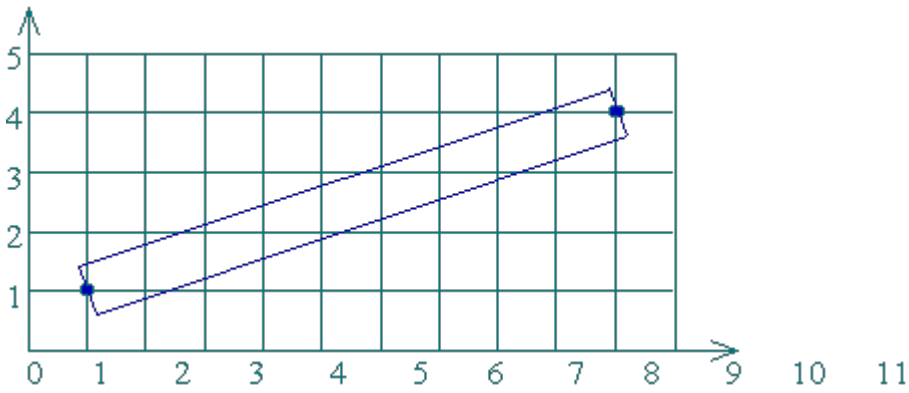


图2.6.2 有宽度的线条轮廓

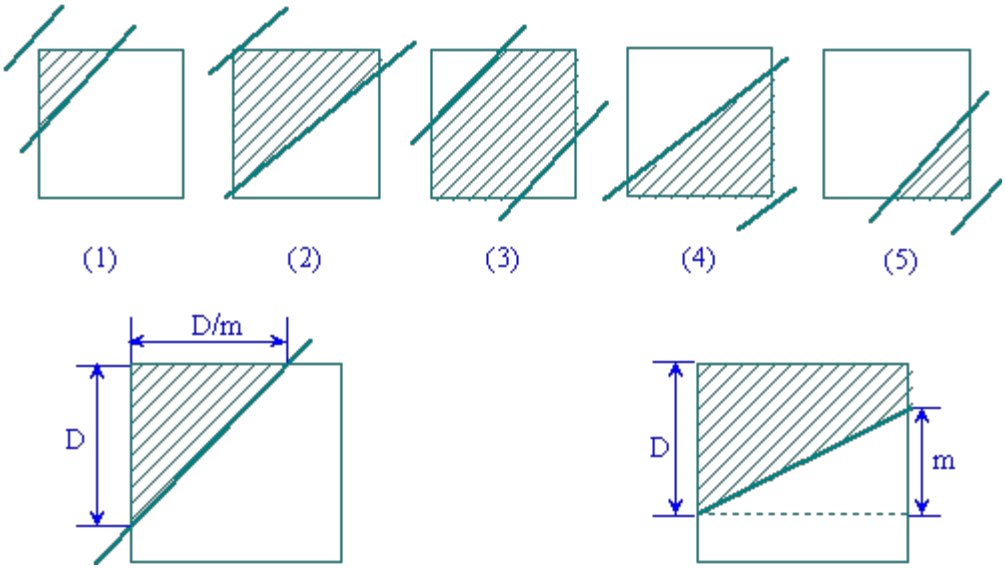


图2.6.3 线条与象素相交的五种情况及用于计算面积的量

在计算阴影区面积时，(1)与(5)，(2)与(4)类似，(3)可用正方形面积区减去二个三角形面积。

如图，情况(1)阴影面积为：$D^2/2m$；

情况(2)阴影面积为：$D - m/2$；

情况(3)阴影面积为：$1 - D^2/m$ 。

上述阴影面积是介于0-1之间的正数，用它乘以象素的最大灰度值，再取整，即可得到象素的显示灰度值。这种区域取样法的反走样效果较好。有时为了简化计算可以采用离散的方法。首先将屏幕象素均分成n个子象素，然后计算中心点落在直线段内的子象素的个数k。最后将屏幕该象素的亮度置为相交区域面积的近似值可k/n。

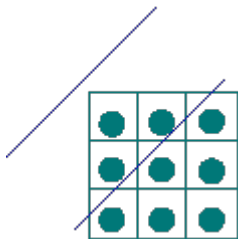


图2.6.4 n=9,k=3近似面积为1/3

从取样理论的角度，区域取样方法相当于使用盒式滤波器进行前置滤波后再取样。

非加权区域采样方法的缺点：

- 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。
- 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。

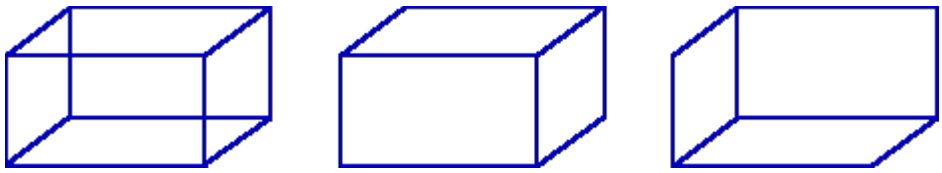
（3）加权区域取样

为了克服上述两个缺点，可以采用加权区域取样方法，使相交区域对象素亮度的贡献依赖于该区域与象素中心的距离。当直线经过该象素时，该象素的亮度F是在两者相交区域A'上对滤波器（函数w）进行积分的积分值。滤波器函数w可以取高斯滤波器。

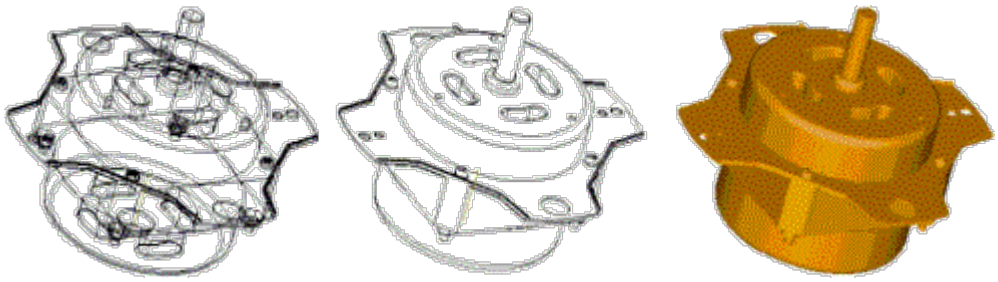
2.7 消隐

✍ 消隐的概念

在用显示设备描述物体的图形时，必须把三维信息经过某种投影变换，在二维的显示表面上绘制出来。由于投影变换失去了深度信息，往往导致图形的二义性（如图1所示）。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，即消隐。经过消隐得到的投影图称为物体的真实图形。



长方体线框投影图的二义性



线框图　　消隐图　　真实感图形

（1）消隐的分类

消隐的对象是三维物体。最简单的表示方式是用表面上的平面多边形表示。如物体的表面是曲面，则将曲面用多个平面多边形近似。 消隐结果与观察物体有关，也与视点有关。

按消隐对象分类

- 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。
- 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。

消隐算法分为三类

- 物体空间的消隐算法 （光线投射、Roberts）：将场景中每一个面与其他每个面比较，求出所有点、边、面遮挡关系。
- 图像空间的消隐算法 (Z－buffer、扫描线、warnock)：对屏幕上每个象素进行判断，决定哪个多边形在该象素可见。
- 物体空间和图像空间的消隐算法 (画家算法)：在物体空间中预先计算面的可见性优先级，再在图像空间中生成消隐图。

（2）消除隐藏线

- 对造型的要求

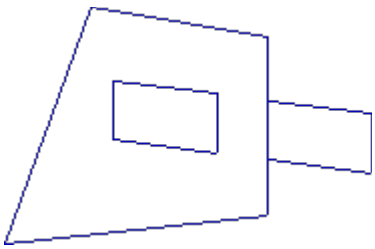
在线框显示模型中，用边界线表示有界平面，用边界线及若干参数曲线表示参数曲面，所以待显示的所有实体均为线。但线不可能对线有遮挡关系，只有面或体才有可能对线形成遮挡。故消隐算法要求造型系统中有面的信息，最好有体的信息。正则形体的消隐可利用其面的法向量，这样比一般情况快的多。

- 坐标变换

为运算方便，一般通过平移、旋转、透视等各种坐标变换，将视点变换到Z轴的正无穷大处，视线方向变为Z轴的负方向。变换后，坐标Z值反映了相应点到视点的距离，可以作为判断遮挡的依据。另外，对视锥以外的物体应先行虑掉，以减少不必要的运算。

- 最基本的运算

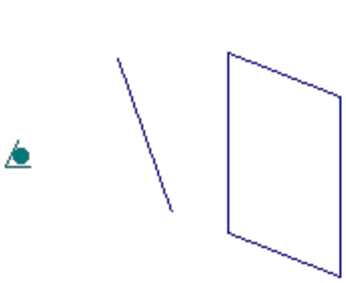
线消隐中，最基本的运算为：判断面对线的遮挡关系。体也要分解为面，再判断面与线的遮挡关系。在遮挡判断中，要反复地进行线线、线面之间的求交运算。



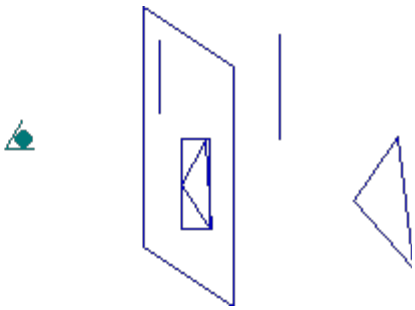
遮挡关系

平面对直线段的遮挡判断算法

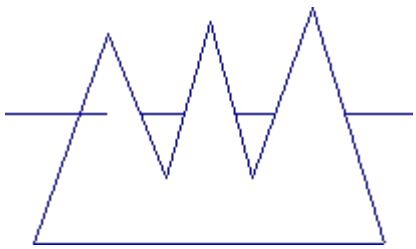
1) 若线段的两端点及视点在给定平面的同侧，线段不被给定平面遮挡，转7
2) 若线段的投影与平面投影的包围盒无交，线段不被给定平面遮挡，转7
3) 求直线与相应无穷平面的交。若无交点，转4。否则，交点在线段内部或外部。若交点在线段内部，交点将线段分成两段，与视点同侧的一段不被遮挡，另一段在视点异侧，转4再判；若交点在线段外部，转4。
4) 求所剩线段的投影与平面边界投影的所有交点，并根据交点在原直线参数方程中的参数值求出Z值(即深度)。若无交点，转5。
5) 以上所求得的各交点将线段的投影分成若干段，求出第一段中点。
6) 若第一段中点在平面的投影内，则相应的段被遮挡，否则不被遮挡；其他段的遮挡关系可依次交替取值进行判断。
7) 结束。



视点与线段同侧



包围盒不交



分段交替取值

（3）消除隐藏面

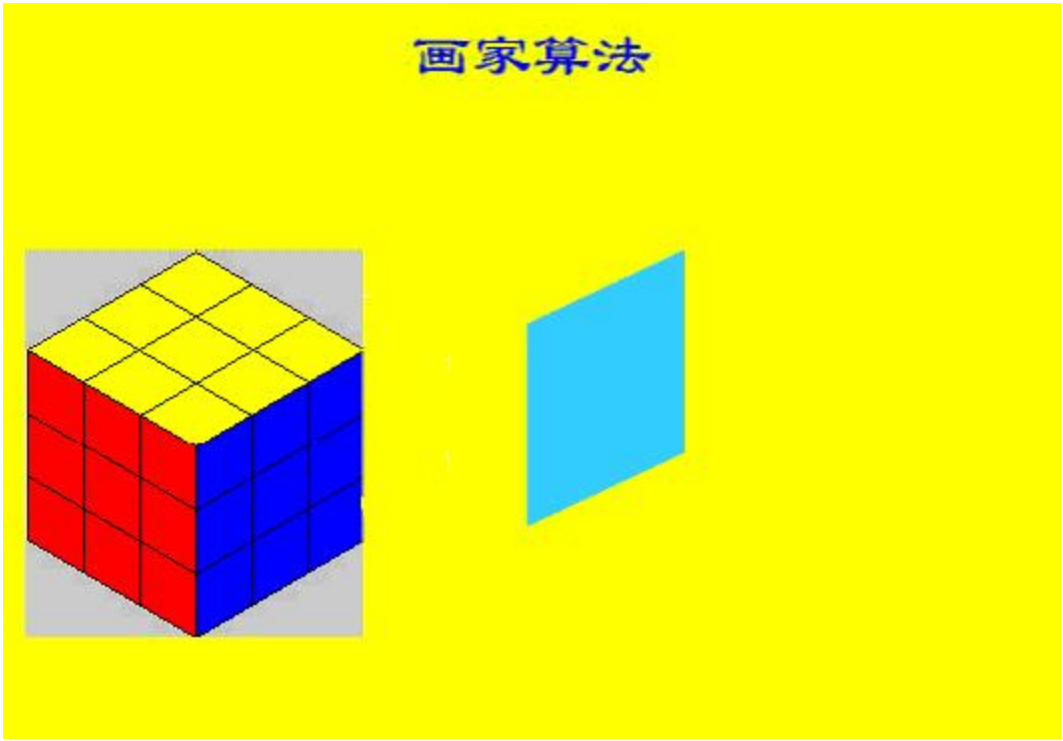
在使用光栅图形显示器绘制物体的真实图形时，必须解决消除隐藏面的问题。这方面的几个常用算法：

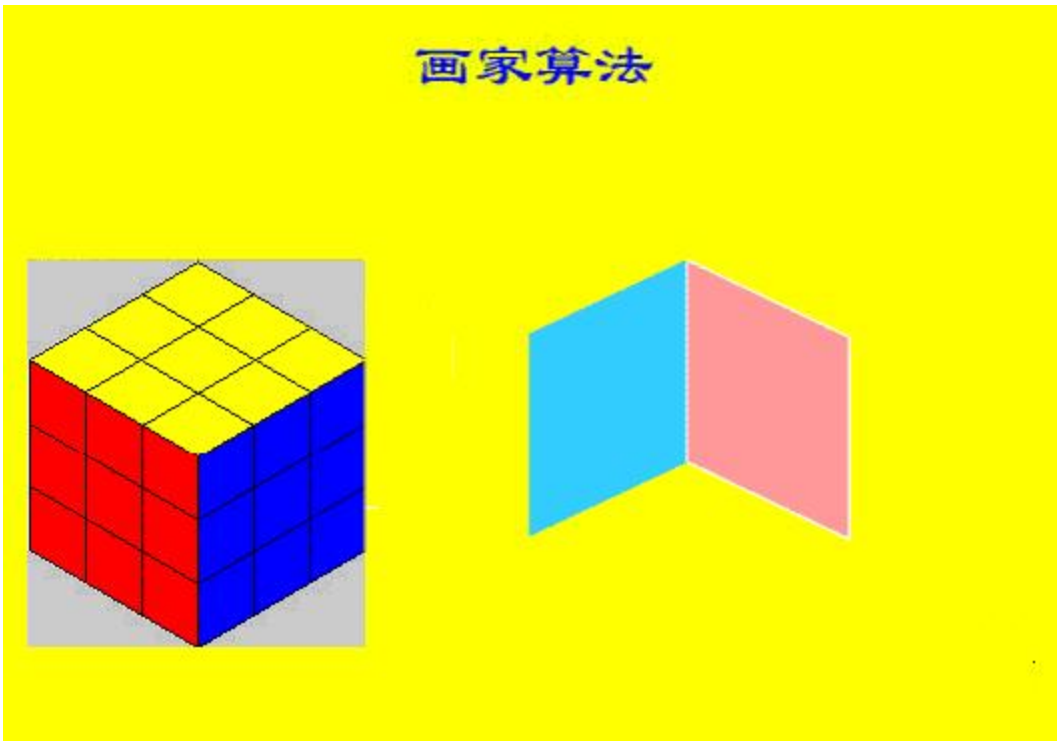
画家算法

画家算法的原理：先把屏幕置成背景色，再把物体的各个面按其离视点的远近进行排序，离视点远者在表头，离视点近者在表尾，排序结果存在一张深度优先级表中。然后按照从表头到表尾的顺序逐个绘制各个面。由于后显示的图形取代先显示的画面，而后显示的图形所代表的面离视点更近，所以由远及近的绘制各面，就相当于消除隐藏面。这与油

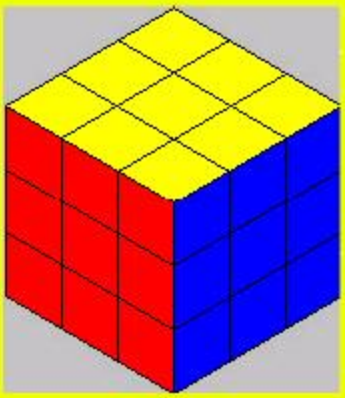
画作家作画的过程类似，先画远景，再画中景，最后画近景。所以，该算法习惯上称为画家算法或列表优先算法。画家算法原理简单。其关键是如何对场景中的物体按深度排序。

画家算法原理的缺点：只能处理互不相交的面，而且深度优先级表中面的顺序可能出错。在两个面相交，三个以上的面重叠的情形，用任何排序方法都不能排出正确的序。这时只能把有关的面进行分割后再排序。

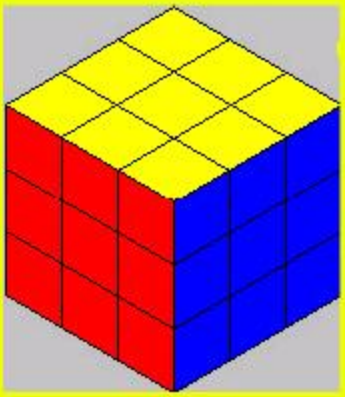




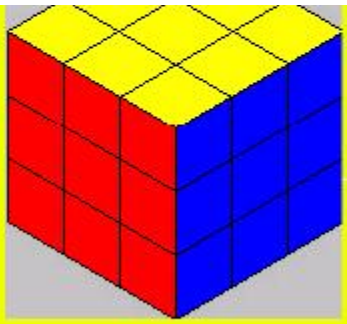
画家算法

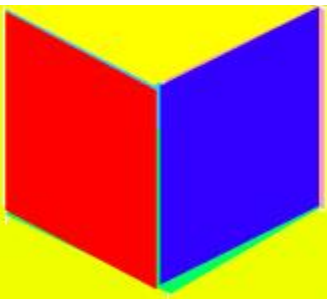


画家算法

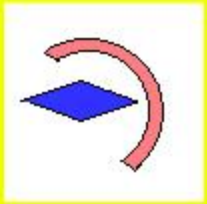


画家算法

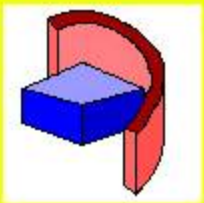




但必须注意，物体的整体结构必须是凸多边形，因为物体实际都是作为一个平面处理（除非将其按3D格式存放）本身都为一次绘成， 凹的多边形容易造成物体间没有固定的前后关系，产生相互遮挡的结果. 此时， 应当将多边形切成几个凸多变形的块，分别处理.



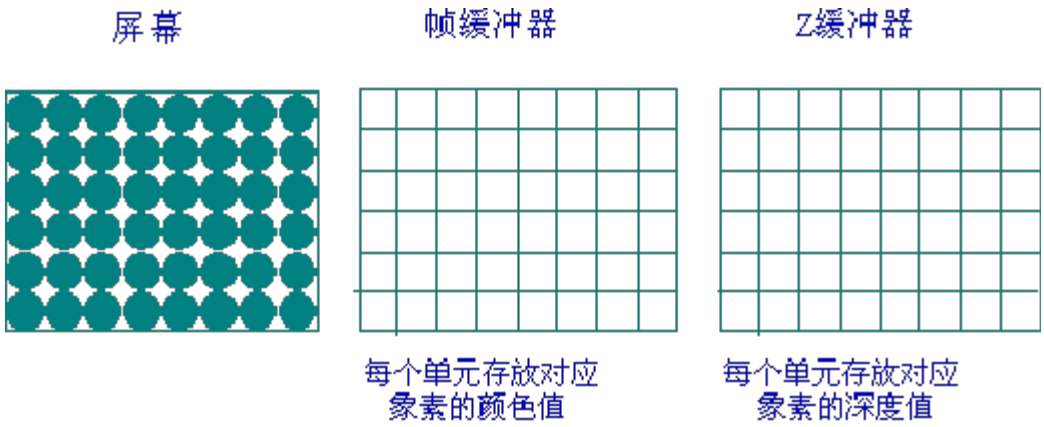
如图，是一个半圆形的墙和一个矩形物体的底面，从物体的底面来看，墙是个凹多边形



在3D视图上，墙就和六面体发生了相互遮挡的现象. 这时，不能简单的先画六面体，或是先画墙，来得到正确的图像.

Z缓冲区(Z-Buffer)算法

画家算法中，深度排序计算量大，而且排序后，还需再检查相邻的面，以确保在深度优先级表中前者在前，后者在后。若遇到多边形相交，或多边形循环重叠的情形，还必须分割多边形。为了避免这些复杂的运算，人们发明了Z缓冲区(Z-Buffer)算法。在这个算法里，不仅需要有帧缓存来存放每个象素的颜色值，还需要一个深度缓存来存放每个象素的深度值。



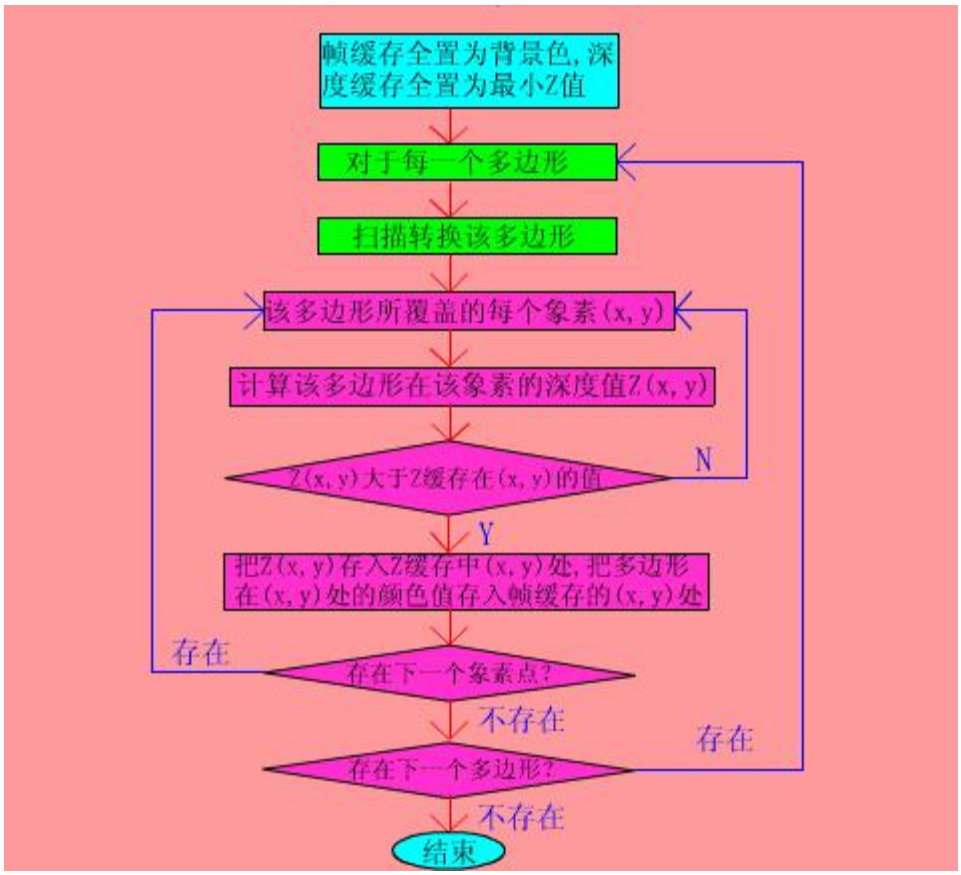
Z缓冲区示意图

Z缓冲器中每个单元的值是对应象素点所反映对象的z坐标值。Z缓冲器中每个单元的初值取成z的极小值，帧缓冲器每个单元的初值可放对应背景颜色的值。图形消隐的过程就是给帧缓冲器和Z缓冲器中相应单元填值的过程。在把显示对象的每个面上每一点的属性（颜色或灰度）值填入帧缓冲器相应单元前，要把这点的z坐标值和z缓冲器中相应单元的值进行比较。只有前者大于后者时才改变帧缓冲器的那一单元的值，同时z缓冲器中相应单元的值也要改成这点的z坐标值。如果这点的z坐标值小于z缓冲器中的值，则说明对应象素已经

显示了对象上一个点的属性，该点要比考虑的点更接近观察点。对显示对象的每个面上的每个点都做了上述处理后，便可得到消除了隐藏面的图。

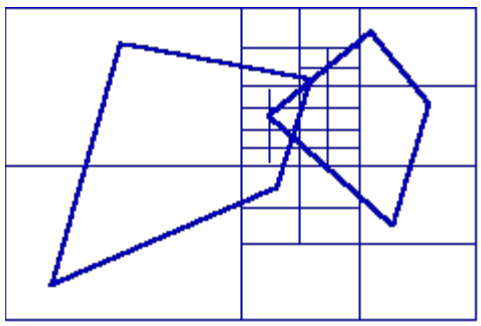
```
Z-Buffer算法（）
{
   帧缓存全置为背景色
   深度缓存全置为最小Z值

       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算法在象素级上以近物取代远物。形体在屏幕上的出现顺序是无关紧要的。这种取代方法实现起来远比总体排序灵活简单，有利于硬件实现。然而Z-Buffer算法存在缺点：占用空间大，没有利用图形的相关性与连续性。 Z-Buffer算法以算法简单著称，但也以占空间大而闻名。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。



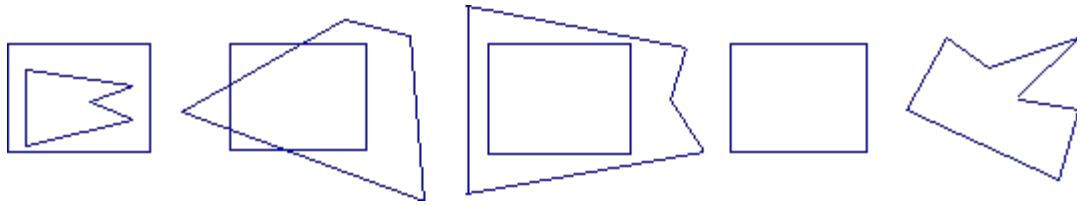
区域子分割算法(Warnack算法)

区域子分割算法的基本思想：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。首先，该算法把初始窗口取作屏幕坐标系的矩形，将场景中的多边形投影到窗口内。如果窗口内没有物体则按背景色显示；若窗口内只有一个面，则把该面显示出来。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。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，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。



区域子分的过程

窗口与多边形的覆盖关系有四种：



(a)内含　　(b)多边形与窗口相交　　(c)包围　　(d)窗口和多边形分离

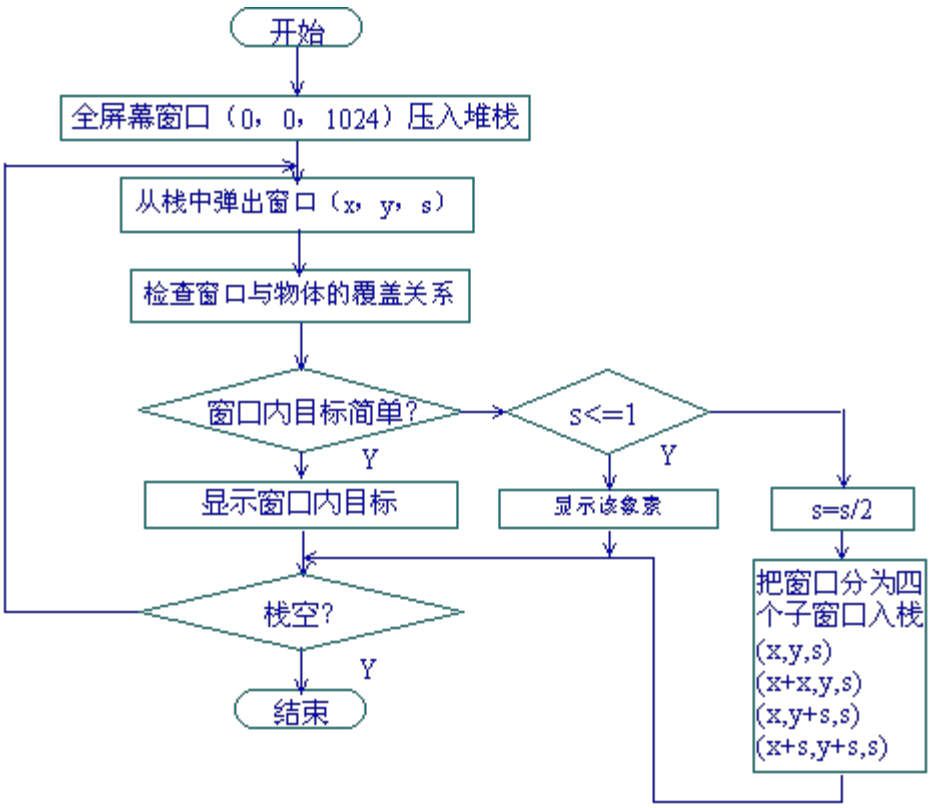
下列情况之一发生时，窗口足够简单，可以直接显示：

(1)所有多边形均与窗口分离。该窗口置背景色。

(2)只有一个多边形与窗口相交，或该多边形包含窗口，则先整个窗口置背景色，在对多边形在窗口内部分扫描线算法绘制。

(3)有一个多边形包围了窗口，或窗口与多个多边形相交，但有一个多边形包围窗口，而且在最前面最靠近观察点。

假设全屏幕窗口分辨率为1024×1024。窗口以左下角点（x，y）和边宽s定义。下图为使用栈结构实现的区域子分割算法流图。由于算法中每次递归的把窗口分割成四个与原窗口相似的小窗口，故这种算法通常称为四叉树算法。

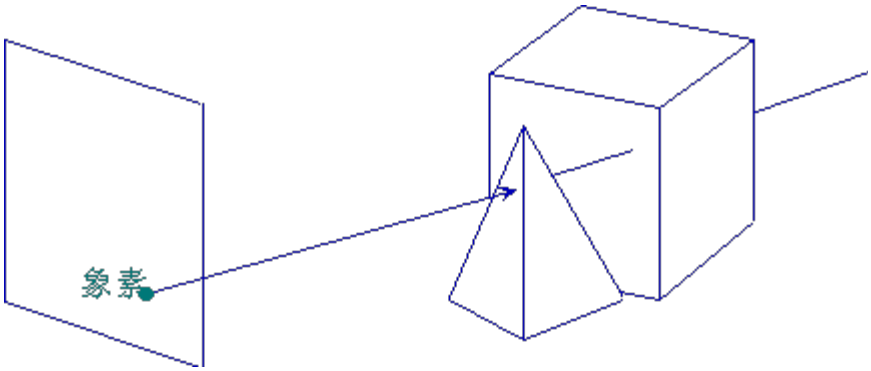


区域子分割算法流图

光线投射算法

光线投射算法的思想：考察由视点出发穿过观察屏幕的一象素而射入场景的一条射线，则可确定出场景中与该射线相交的物体。在计算出光线与物体表面的交点之后，离象

素最近的交点的所在面片的颜色为该象素的颜色；如果没有交点，说明没有多边形的投影覆盖此象素，用背景色显示它即可。



将通过屏幕各象素的投影线与场景中的物体表面求交

算法描述

```
for（屏幕上的每一象素）
{
   形成通过该屏幕象素(u,v)的射线；
  for（场景中的每个物体）
        将射线与该物体求交；

  if（存在交点）
        以最近的交点所属的颜色显示象素(u,v)
  else
   以背景色显示象素(u,v)
}
```

光线投射算法与Z缓冲器算法相比，它们仅仅是内外循环颠倒了一下顺序，所以它们的算法复杂度类似。区别在于光线投射算法不需要Z缓冲器。为了提高本算法的效率可以使用包围盒技术，空间分割技术以及物体的层次表示方法等来加速。