**图形学部分**

1. **直线扫描转换算法**
   1. **DDA**

思想:先计算坐标差,取最大的坐标差值为e,计算坐标增量为坐标差/e,然后开始描点,每描一个点,坐标按照增量递增取整,直到画够e向下取整个点.

* + - 1. **void** DDALine(**int** x1,**int** y1,**int** x2,**int** y2) {
      2. **double** dx,dy,e,x,y;
      3. dx=x2-x1;
      4. dy=y2-y1;
      5. e=(fabs(dx)>fabs(dy))?fabs(dx):fabs(dy);
      6. dx/=e;
      7. dy/=e;
      8. x=x1;
      9. y=y1;
      10. **for**(**int** i=1;i<=e;i++){
      11. SetPixel((**int**)(x+0.5), (**int**)(y+0.5));
      12. x+=dx;
      13. y+=dy;
      14. }
      15. }
  1. **中点画线算法**

使用条件斜率在0~1之间,直线方程ax+by+c=0,判别式实际上是用来区分实际点是在(x+1,y)(x+1,y+1)这个小线段的中点的上面还是下面,依据这个来决定纵坐标是否递增,为了计算方便相关的式子乘以了2,用以避免浮点数计算.

1. **void** MidpointLine (**int** x0,**int** y0,**int** x1,**int** y1){
2. **int** a = y0 - y1;         //直线方程中的a
3. **int** b = x1 - x0;         //直线方程中的b
4. **int** d = 2 \* a + b;       //判别式的初值
5. **int** delta1 = 2 \* a;      //纵坐标不递增时的判别式增量
6. **int** delta2 = 2 \*( a + b);//纵坐标递增是的判别式增量
7. **int** x = x0;
8. **int** y = y0;
9. SetPixel(x,y);//第一个点没有争议
10. **while**(x<x1){
11. x++;//横坐标递增
12. **if**(d<0){//判别式小于0,纵坐标递增
13. y++;
14. d+=delta2;
15. }**else**{//判别式不小于0,纵坐标不递增
16. d+= delta1;
17. }
18. SetPixel(x,y);
19. }
20. }
    1. **Bresenham画线算法**

同中点划线法相似,只是判别式的原理不同,此处是将纵坐标递增和不递增两种情况下的点和直线上的真实点之间的距离做差,通过差值的正负判断这两个距离的大小,选择距离真实点较近的那个点进行描点.

1. **void** BresenhamLine(**int** x1,**int** y1,**int** x2,**int** y2){
2. **int** x=x1;
3. **int** y=y1;
4. **int** dx=x2-x1;
5. **int** dy=y2-y1;
6. **int** p=2\*dy-dx;       //判别式初值
7. **for**(;x<=x2;x++){     //横坐标递增
8. SetPixel(x,y);
9. **if**(p>=0){        //判别式非负,纵坐标递增
10. y++;
11. p+=2\*(dy-dx);//判别式非负时的增量
12. }**else**{           //判别式为负,纵坐标不变
13. p+=2\*dy;     //判别式为负时的增量
14. }
15. }
16. }
17. **圆的扫描转换**
    1. **Bresenham画圆法**

只考虑圆心在原点的第一象限角分线之上(0≤x≤y)的1/8圆周的画法,其他情况使用对称和平移操作绘制.在这个范围内,随着x的递增,y递减,每次x递增y要在y和y-1中选择,选择方法类似于Bresenham画线法

1. **void** BresenhamCircle(**int** R){
2. **int** x=0;
3. **int** y=R;
4. **int** p=3-2\*R;        //判别式初值
5. **for**(;x<=y;x++)
6. {
7. SetPixel(x,y);
8. **if**(p>=0){           //判别式非负,y递减
9. p+=4\*(x-y)+10;  //判别式增量
10. y--;
11. }**else**{              //判别式为负,y不变
12. p+=4\*x+6;
13. }
14. }
15. }
16. **区域填充算法**
    1. **种子填充**

利用区域的连通性(8联通区域和4联通区域),从事先给定的一个区域内部点开始进行递归遍历,直到不再能找到需要填充的点.

1. /\*(x,y) 为种子位置
2. boundaryvalue是边界象素值
3. newvalue是区域内象素新值，未填充前区域内不应有值为newvalue的象素。\*/
4. **void** Boundaryfill(**int** x,**int** y,**COLORREF** boundaryvalue,**COLORREF** newvalue){
5. **if**( GetPixel(x,y)!=boundaryvalue && GetPixel(x,y)!=newvalue){
6. // 未达边界且未访问过
7. SetPixel(x,y,newvalue);//赋以新值
8. Boundaryfill(x,y-1,boundaryvalue,newvalue);  //四联通
9. Boundaryfill(x,y+1,boundaryvalue,newvalue);
10. Boundaryfill(x-1,y,boundaryvalue,newvalue);
11. Boundaryfill(x+1,y,boundaryvalue,newvalue);
12. }
13. }
    1. **扫描线种子填充算法**
14. 初始化一个空的栈用于存放种子点，将种子点(x, y)入栈；
15. 判断栈是否为空，如果栈为空则结束算法，否则取出栈顶元素作为当前扫描线的种子点(x, y)，y是当前的扫描线；
16. 从种子点(x, y)出发，沿当前扫描线向左、右两个方向填充，直到边界。分别标记区段的左、右端点坐标为xLeft和xRight；
17. 分别检查与当前扫描线相邻的y - 1和y + 1两条扫描线在区间[xLeft, xRight]中的像素，从xLeft开始向xRight方向搜索，若存在非边界且未填充的像素点，则找出这些相邻的像素点中最右边的一个，并将其作为种子点压入栈中，然后返回第2步；
18. **void** ScanLineSeedFill(**int** x, **int** y, **int** new\_color, **int** boundary\_color)
19. {
20. std::stack<Point> stk;
21. stk.push(Point(x, y)); //第1步，种子点入站
22. **while**(!stk.empty())
23. {
24. Point seed = stk.top(); //第2步，取当前种子点
25. stk.pop();
26. //第3步，向左右填充
27. **int** count = FillLineRight(seed.x, seed.y, new\_color, boundary\_color);
28. **int** xRight = seed.x + count - 1;
29. count = FillLineLeft(seed.x - 1, seed.y, new\_color, boundary\_color);
30. **int** xLeft = seed.x - count;
31. //第4步，处理相邻两条扫描线
32. SearchLineNewSeed(stk, xLeft, xRight, seed.y - 1, new\_color, boundary\_color);
33. SearchLineNewSeed(stk, xLeft, xRight, seed.y + 1, new\_color, boundary\_color);
34. }
35. }
37. **void** SearchLineNewSeed(std::stack<Point>& stk, **int** xLeft, **int** xRight,
38. **int** y, **int** new\_color, **int** boundary\_color)
39. {
40. **int** xt = xLeft;
41. **bool** findNewSeed = **false**;
42. **while**(xt <= xRight)
43. {
44. findNewSeed = **false**;
45. **while**(IsPixelValid(xt, y, new\_color, boundary\_color) && (xt < xRight))
46. {
47. findNewSeed = **true**;
48. xt++;
49. }
50. **if**(findNewSeed)
51. {
52. **if**(IsPixelValid(xt, y, new\_color, boundary\_color) && (xt == xRight))
53. stk.push(Point(xt, y));
54. **else**
55. stk.push(Point(xt - 1, y));
56. }
57. /\*向右跳过内部的无效点（处理区间右端有障碍点的情况）\*/
58. **int** xspan = SkipInvalidInLine(xt, y, xRight, new\_color, boundary\_color);
59. xt += (xspan == 0) ? 1 : xspan;
60. /\*处理特殊情况,以退出while(x<=xright)循环\*/
61. }
62. }
63. **图形变换**
    1. **二维图形的平移,缩放,旋转,对称,错切**

平移;缩放;旋转(绕原点旋转);

对称;错切

* 1. **二维视见变化**

我感觉不能考,蜜汁自信

* 1. **三维图形的平移,缩放,旋转**

平移;缩放;

绕X轴旋转:

绕Y轴旋转:

绕Z轴旋转:

* 1. **三维图形的投影变换**
     1. **平行投影**

投影中心与投影平面无限远,投影线为平行线

* + 1. **透视投影**

投影中心与投影面有限远,投影线交于一点.

平行于投影平面的平行线,投影后要么保持平行,要么重合;

不平行于投影平面的平行线,投影后交于一点,该点称为消失点.每有一组不平行于投影平面的平行线,就有一个消失点.落在坐标轴上的消失点称为主消失点,主消失点最多只有三个,所谓一点透视/两点透视/三点透视,说的是主消失点.

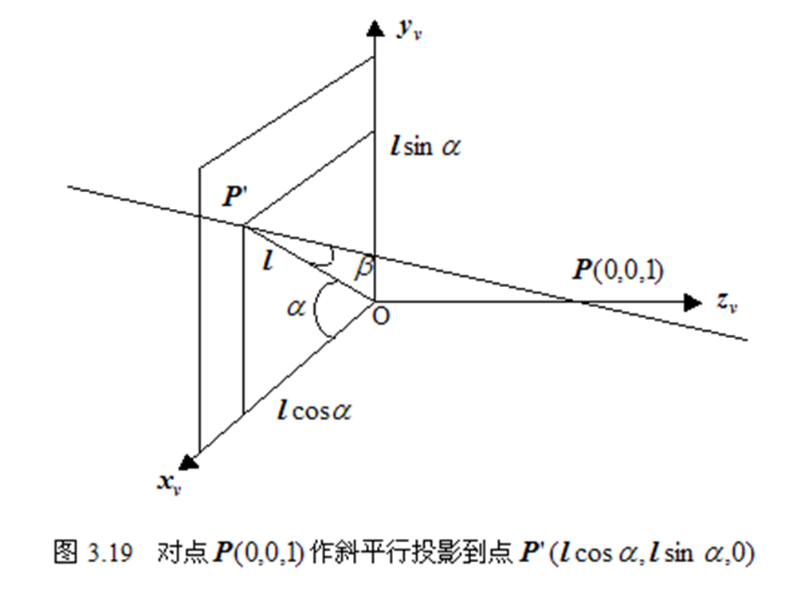
* + - 1. **正交投影**

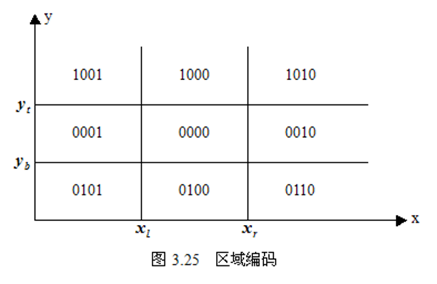
投影方向与投影平面垂直,正视投影,顶视投影,侧视投影.如果投影平面垂直于坐标轴,称为正投影,如果投影方向与三个坐标轴夹角相等,则称等轴投影,等轴投影能使得三个方向上产生相等的透视缩短.

正投影矩阵,投影方向是哪个坐标轴,xyz中对应的值为0,其余两个为1即可.

* + - 1. **斜交投影**

投影平面与投影方向不垂直,此时投影面一般取坐标平面.



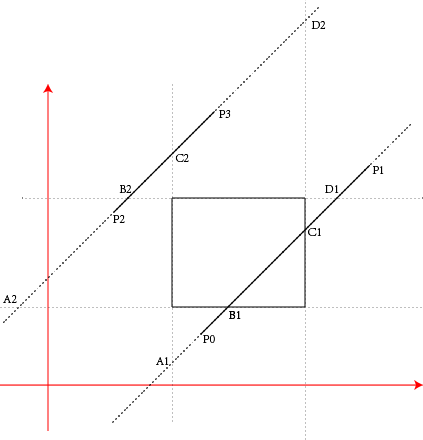
* 1. **裁剪**
     1. **直线段裁剪**
        1. **Cohen-Sutherland算法**

首先判断直线段是否完全在窗口内,是则保留,不是则在判断是否完全在窗口外,是则舍弃;如果两种情况都不符合,将此线段分割,分割之后再继续上述判断,直到所有的线段都判断完毕.

判断线段是否完全在窗口内或者窗口外时,先对区域进行编码,其中显示窗口是中间编码为0000的区域,四位数字依次代表当前区域是否在窗口区域的上下右左四个方位,当线段的两个端点的编号都是0000时,说明线段完全可见,当线段两个端点的编号进行按位与运算的结果不为0时,说明线段完全不可见,如果不符合上述两个情况,说明线段有可能部分可见,需要进行分割.

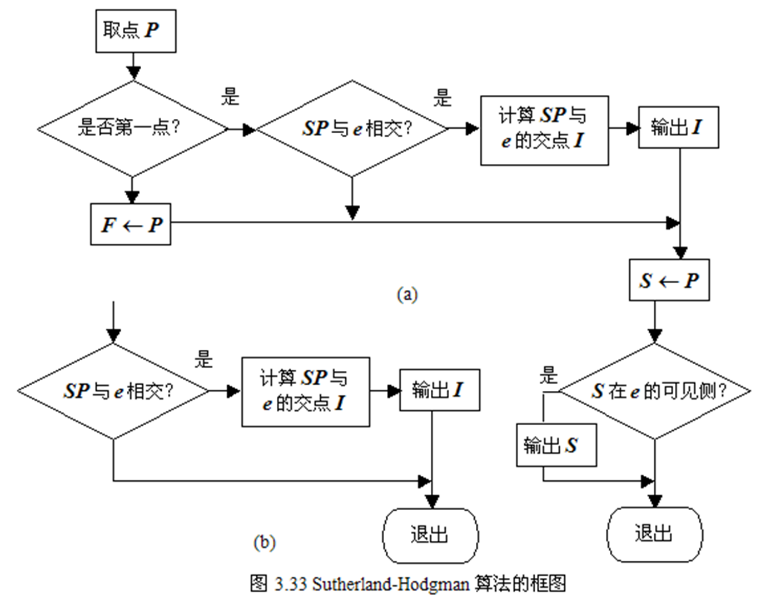
对线段进行分割时,采用的是中点分割法,首先给定一个精度值,然后对要分割的线段求中点,分别判断两个子线段,如果不能判断为完全可见或者完全不可见,则对子线段进一步切割,直到所有的子线段都能够被判断为完全可见或者完全不可见,或者线段的长度已经达到精度值时,算法终止.

1. //函数Cohen\_Sutherland用来实现算法
2. //函数makecode用来编码，利用数值位运算
3. //double xl, xr, yt, yb; (事先给出窗口的位置，四个数值是已知的)
4. **void** makecode(**double** x, y; **int** c)
5. {
6. c=0;
7. **if** (x<xl)
8. c=1;
9. **else** **if** (x>xr)
10. c=2;
11. **if** (y<yb)
12. c=c+4;
13. **else** **if** (y>yt)
14. c=c+8;
15. }
17. **void** Cohen\_Sutherland(**double** x0, y0, x2, y2)
18. {
19. **int** c, c1, c2;
20. **double** x, y;
21. makecode(x0, y0,c1);
22. makecode(x2, y2,c2);
23. **while** (c1!=0 || c2!=0){
24. **if** (c1&c2)
25. **return**;
26. c=c1;
27. **if** (c==0)
28. c=c2;
29. **if** (c&1==1) {
30. y=y0+(y2-y0)\*(x1-x0)/(x2-x0);
31. x=x1;
32. }**else** **if** (c&2==2) {
33. y=y0+(y2-y0)\*(xr-x0)/(x2-x0);
34. x=xr;
35. }**else** **if** (c&4==4) {
36. x=x0+(x2-x0)\*(yb-y0)/(y2-y0);
37. y=yb;
38. }**else** **if** (c&8==8) {
39. x=x0+(x2-x0)\*(yt-y0)/(y2-y0);
40. y=yt;
41. }
42. **if** (c==c1){
43. x0=x;
44. y0=y;
45. makecode(x, y, c1);
46. }**else**{
47. x2=x;
48. y2=y;
49. makecode(x, y, c2);
50. }
51. }
52. showline(x0, y0, x2, y2); //显示可见线段
53. }
    * + 1. **梁友栋-Barsky算法**

****

将线段和窗口边界延长为直线,会产生四个交点,如果一条线段有可视部分,那么可视部分一定是这四个交点和线段的两个端点这六个点中的某两个点之间的子线段,

1. //函数L\_Barsky实现算法
2. //函数cansee用于判断直线段是否可见。
3. //double xl, xr, yt, yb; (窗口位置)
4. **void** L\_Barsky(**double** x0, y0, x2, y2){
5. **double** t0, t1, deltax, deltay;
6. t0=0.0; t1=1.0;
7. deltax=x2-x0;
8. **if** (!cansee(-deltax, x0-x1, t0, t1))
9. **return**;
10. **if** (!cansee(deltax, xr-x0, t0, t1))
11. **return**;
12. deltay=y2-y0;
13. **if** (!cansee(-deltay, y0-yb, t0, t1))
14. **return**;
15. **if** (!cansee(deltay, yt-y0, t0, t1))
16. **return**;
17. x2=x0+t1\*deltax;
18. y2=y0+t1\*deltay;
19. x0=x0+t0\*deltax;
20. y0=y0+t0\*deltay;
21. showline(x0, y0, x2, y2);   //显示可见线段
22. }
23. **bool** cansee(**double** q, d, t0, t1){
24. **double** r;
25. **if** (q<0){
26. r=d/q;
27. **if** (r>tl) {
28. cansee=**false**;
29. **return**;
30. }**else** **if** (r>t0)
31. t0=r;
32. }**else** **if** (q>0){
33. r=d/q;
34. **if** (r<t0) {
35. cansee=**false**;
36. **return**;
37. } **else** **if** (r<t1)
38. t1=r;
39. }**else** **if** (d<0) {
40. cansee=**false**;
41. **return**;
42. }
43. cansee=**true**;
44. }
    * 1. **其他裁剪**
         1. 用于多边形剪裁的Sutherland-Hodgman算法



* + - 1. 字符裁剪

把字符看成由线段连接而成,使用线段裁剪算法处理,或者,当外界矩形的中心或左下角不可见时认为整个字符或者字符串不可见

* + - 1. 圆弧裁剪

可把圆弧和窗口四条边的交点求出来，再按交点对圆心幅角的大小排序，排序后，相邻的两个交点决定了圆弧上一段可见或不可见的弧。

* + - 1. 裁剪窗口为任意凸多边形时的直线段裁剪算法
      2. 三维图形的裁剪

1. **曲线**
   1. **曲线曲面基础知识**
   2. **Bezier曲线**

给出型值点P0，P1，…，Pn，它们所确定的n次Bezier曲线是：(0!及00约定均为1)

曲线通过型值点列的起点和中点,





n=1时: (线段)

n=2时: (二次函数)

n=3时: 

几何作图法:



* 1. **B样条曲线**

给定n+1个控制点P0，P1，…，Pn，它们所确定的k阶B样条曲线是：

这里u0，u1，…，un+k，是一个非递减的序列，称为节点，(u0，u1，…，un+k)称为节点向量。定义中可能出现0/0，这时约定为0。n+1个控制点P0，P1，…，Pn所确定的最高阶的B样条曲线是k=n+1阶的，这时由节点向量(0，0，…，0，1，1，…，1)所确定的B样条曲线，与该n+1个控制点所确定的Bezier曲线相同。

1. **图形运算**
   1. **线段的交点计算**
      1. 判断条线段是否相交

线段AB和线段CD的端点分别是(xa,ya)(xb,yb)和(xc,yc)(xd,yd),如果

****

成立则说明线段无交点,否则相交,

* + 1. 求交点参数值



需要注意,只有参数均在[0,1]时两线段才真正相交。否则,交点在两线段或其中某一条线段的延长线上,这时仍然认为是两线段不相交。

* + 1. **多条线段求交点**

平面扫描法求多条线段的交点,首先用一个方法判断出哪些线段可能相交了,再对这些相交的线段求取交点.

在平面上,从待计算线段组的最左端到最右端,x依次递增,扫描线垂直于X轴从左到右依次扫描,在某个扫面线状态下,扫面线与线段的交点有的在上面有的在下面,我们就说在x处某线段在某线段的上面,如果我们限定,没有垂直于X轴的线段,并且最多只存在两两相交的线段,那么可以得出结论,如果有两条线段相交,则必存在某个扫描线位置,把该位置的线段从上到下排列,他们两个在序列中相邻.

基于上面的发现,我们设置三个个数据结构,一个进度表,其中存储所有的待扫描点,形如(x,y,描述),其中含有这个点是哪条线段的左端点还是右端点或是线段之间的交点的描述,另一个数据结构是状态表,其中存放当前扫描线压着的线段,进度表的顺序按照扫描顺序存放,状态表的顺序按照从上到下的顺序存放,最后有一个交点集合,这样我们能得到一个算法:

1. 初始化进度表为全部线段的端点并排序,初始化状态表为空;
2. 若进度表已空,算法结束.否则进行3
3. 从进度表头取出一个元素:
   1. 如果这个元素是某条线段的左端点,向状态表的合适位置插入这条线段,并判断此时刚刚插入的线段的两侧有没有其他线段,如果有,计算该线段与前后两条线段的交点,并把交点插入交点集合
   2. 如果这个元素是某条线段的右端点,判断在状态表中该线段的两边有没有其他线段,如果有,求取交点,并把交点插入交点集合,并从状态表中删除这个线段;
   3. 如果这个元素是线段A和线段B的交点,且在进度表中A在B的前面,则判断A的前面和B的后面是否有其他线段,如果情况为XABY,计算X与B的交点和Y与A的交点,并把交点插入交点集合,然后在状态表中交换A和B的顺序.
   4. 把交点集合中所有不在进度表中的交点输出,并插入到进度表的适当位置
   5. 返回2
   6. **多边形表面的交线计算**

根据顶点坐标求出两个多边形表面分别所在平面的方程,再根据平面方程计算交线,最后,还要确定出交线同时在两个多边形表面内部的部分.

* 1. **平面中的凸壳算法**

包含一个平面点集的最小凸区域叫做凸壳,有两个算法:

* + 1. Graham扫描算法

处理的思路是设想有一内点O并且不妨设想O就是坐标原点,这时点集S中所有各点相对轴OX有一个倾角。所有各点按倾角递增排序后,如果某一点不是壳上顶点,则它必然在两个壳顶点与点O形成的三角形内部。计算是可以取x坐标最小的点为O点,若这样的点不唯一则再由其中选出y坐标最小的点.

由于按照逆时针方向扫描,一旦在某个点处发生右转,就可以认定这个点是内点应该去掉,对于三个连续点,可以通过计算下列的行列式判断是右转还是左转,行列式等于0时是三点共线,大于0是左转,小于0是右转:



* + 1. Jarvis行进算法

想法是若相继两点是一条凸壳多边形的边,则对于过该边的直线,所有点集中的凸壳中的顶点在该直线同侧。因此若找到pq是壳上一边,则以q为端点的下一条壳边qr可以如下求出：计算点集中其余各点相对q点发出沿向量pq向的射线的倾角,若倾角最小者对应的点是r,则qr是下一条壳边. 寻找开始行进的第一条壳边,可以选出点集中按x,y坐标字典式次序的最低点,该点必定是一个壳顶点。可从该点引一条竖直向下的射线,在此做一个行进步就找到了第一条壳边。

* 1. **多边形的包含算法**

平面上的简单多边形是不相邻的边不能相交的多边形,简单多边形包含检验是求平面上一点是否在某个简单多边形的内部的算法.

**方案一:**从定点竖直向下做射线,统计一共与多边形的边产生了多少个交点, 当由点P竖直向下的射线恰好通过多边形的顶点或某一边时,交点计数可采取简单的"左闭右开"法来处理,即:当多边形一边的两个顶点的x坐标都小于或等于点P的x坐标时,相应交点不计算在内**。**

**方案二:**沿着逆时针方向行进,依次询问每条边,定点是否在边的左侧,如果全是,那就在内部.

* 1. **凸多边形重叠计算**

求两个凸多边形相交形成的多边形, 有两个问题需要解决,一个是如何有次序地求出各边的所有交点,一个是如何排列求出交点和原凸多边形的顶点,形成交得凸多边形的顶点序列。为了有次序地求出交点,可以在两个多边形边上交替地前进,原则是在哪个多边形的边上可能有交点就等待,在另一个多边形的边上前进**。**

* 1. **简单多边形的三角剖分**

简单多边形做三角剖分,是要求选出完全在内部又互不相交的一组对角线，把整个多边形划分成一些三角形。这里对角线是不相邻顶点间的连线,选出的对角线的集合称为是简单多边形的三角剖分。三角剖分是不唯一的.

简单多边形三角剖分的算法:考查连续三个顶点A,B,C,若AC完全在多边形内部,则可输出△ABC为一个剖分后形成的三角形,删除点B后再对少了一个顶点的多边形继续进行。

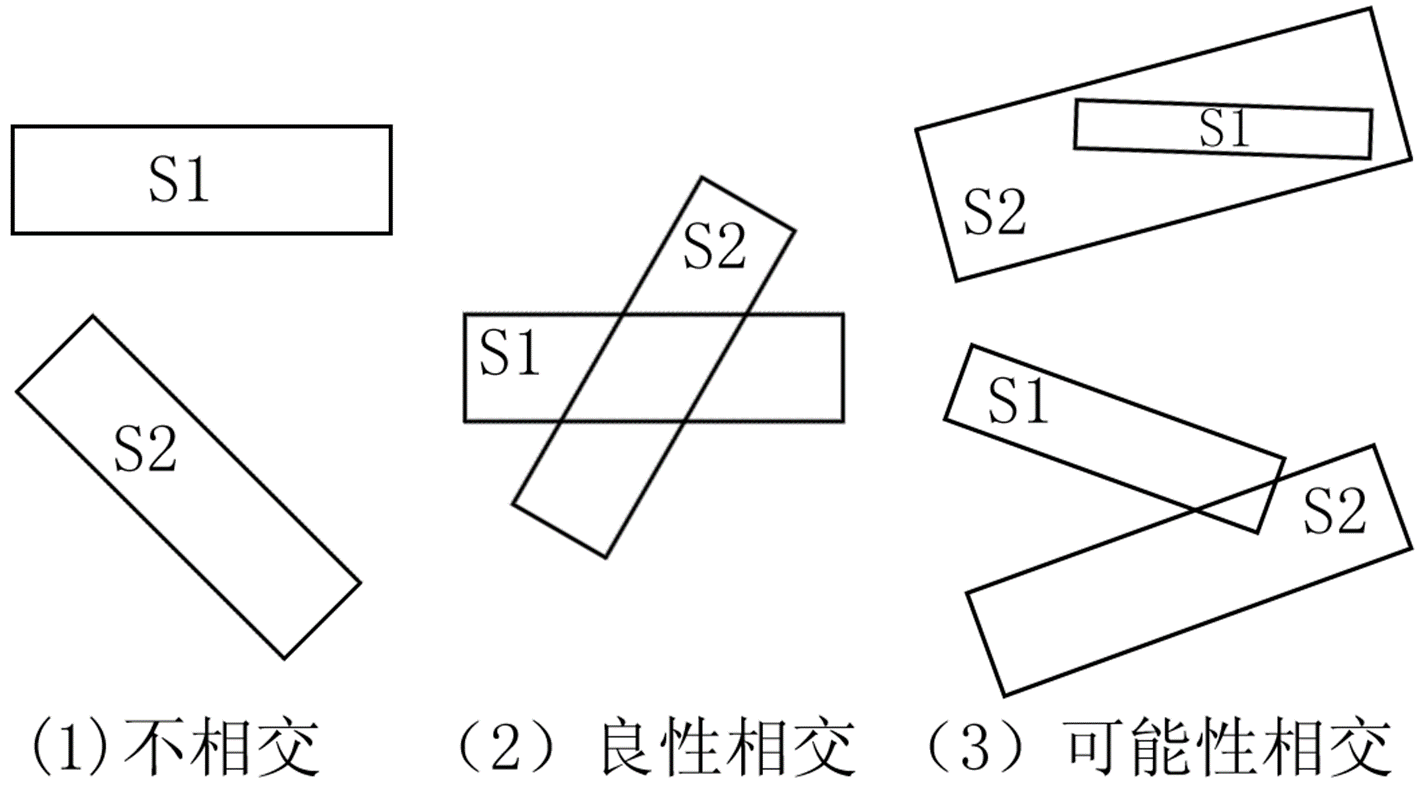
1. **形体表示**
   1. **带树法**

带树法是用来存储曲线段的,给定分辨率W0时,曲线被离散化成P0………Pn这一系列点,带树是一颗二叉树,每个节点有八个字段,分别是起点终点的坐标(占四个字段),带左侧的宽度Wl,带右侧的宽度Wr,指向左右子节点的指针,生成带树的算法如下:

* + 1. 找出由起点P0,终点Pn确定的矩形带段,其中包含P0至Pn的全部点,构造此矩形带段的对应结点并令为根。
    2. 找出P0至Pn之间距离连线P0Pn为最远的点Pk,然后对P0至Pk和Pk至Pn这两组点分别做与步1中相同的构造矩形带段及对应结点的操作,产生的两个结点,分别是根的左右子结点。
    3. 反复执行上述操作,直到所产生结点的w=wl+wr ≤w0。这样的结点是叶结点。
  1. **带树显示**

**从根节点开始遍历带树,并计算**wl+wr,若小于等于需要的分辨率,则显示当前所表示的带段,若当前的分辨率仍大于需要的分辨率,则继续遍历,在这里分辨率实际上是误差的意思,分辨率越小显示的越接近于真实的曲线.

* 1. **带树求交**

****

设表示要求交两曲线的带树己构造得足够精确,即在树叶一层,来自不同带树的矩形带段或是不相交或是良性相交,而没有可能性相交出现。

若T1和T2对应的矩形带段互不相交,那么它们代表的曲线不相交;

若T1和T2对应的矩形带段良性相交,那么它们代表的曲线相交;

若T1和T2对应的矩形带段可能性相交,且T1的面积大于或等T2的面积,那么分别执行T2与T1的左右两个儿子结点的相交性检查。若T1的面积小于T2的面积,则把它们位置对换一下再如上进行两个检查。若两个检查的结果都是不相交,则认为所表示曲线不相交;若两个检查中有一个是良性相交,则认为所表示曲线相交;若不是上述两情形,即出现可能性相交,则对可能性相交的两个矩形带段中面积较大者,取其对应结点的两个子结点,如此进行可直到树叶那一层。

1. **消除隐藏线和隐藏面**
   1. **线面比较法消除隐藏线**

先利用外法线判断出所有可能的可见面，外法线与观察方向的夹角在[0,90°)范围时面是可能可见的,等于90°时退化成线,可能可见面上的线段是可能可见线。要依次用每一条可能可见线，与每一个可能可见面比较，从而确定出可见线、隐藏线及可见线上的隐藏部分。

线面比较的意义是确定线在面的前面还是后面,因为处于前面的可见面会挡住后面的线,比较时,先进行范围检查,即按照x方向和y方向,比价线段和多边形投影的最大值和最小值,以排除那些肯定不会存在遮挡关系的线,然后在沿着z方向做粗略的深度检验,排除那些完全处于面前面的线段,和完全处于面后面的线段,这些线段和面的关系已经完全确定,不需要进一步计算

对于余下的线段,进行精确的深度比较,计算端点在平面上的投影点,比较相应的坐标,这样又能排除一部分不会被遮挡和完全被遮挡的线段.

剩下的线段与平面相交,需要求交点来判断线段的那一部分是可见的,求得交点,线段被分成了若干个子线段,在子线段上取一点,判断它是否被面的投影多边形所包围,就能判断出这个子线段是否可见.

可见性判断完毕.

* 1. **曲面隐藏线消除的浮动水平线算法.**

我猜不考这个!蜜汁自信

* 1. **深度排序算法**

1. 把所有的多边形按顶点最大z坐标值进行排序。

2. 解决当多边形z范围发生交迭时出现的不明确问题。

3. 按最大z坐标值逐渐减小的次序，对每个多边形进行扫描转换**。**

**其中所谓的不明确问题,指的是所有多边形按顶点最大z坐标值排序,其中P排在后面,Q排在P前面,但是P和Q的z坐标范围有交叉,此时我们没有十足的把握说,Q一定在P前面,需要进行进一步判断.**

以下五个情况是Q完全处于P前面的情况:

1．多边形的x坐标范围不相交迭，所以多边形不相交迭。

2．多边形的y坐标范围不相交迭，所以多边形不相交迭。

3. P整个在Q远离观察点的一侧。

4. Q整个在P的靠近观察点的一侧。

5. 多边形在z=0平面上的投影本身不相交迭。

**如果五个检查全都没有通过,就假定P遮挡了Q,并且交换P和Q在排序表中的位置,并对Q做上标记,以免循环移动,当再次移动带标记的多边形时,把另一个多边形剪裁成两个.**

* 1. **画家算法**

是深度排序算法的一种具体实现。先画远景，再画中景，最后画近景。(不能考)

* 1. **z缓冲算法**

需要有一个更新缓冲器存储各点的象素值，而且还需要有一个z缓冲存储器存储相应的z值。帧缓冲存储器初始化为背景值，z缓冲存储器初始化为可以表示的最大z值。对每一个多边形，不必进行深度排序算法要求的初始排序，立即就可以逐个进行扫描转换。

帧缓冲存储器和Z缓冲存储器相当于两个大小等于绘图区域的矩阵,其中帧缓冲存储器上存储像素值,直接对应即将要画到屏幕上的图像,Z缓冲存储器存储对应位置的最新Z值

算法对于每个多边形内部的任一点(x,y),执行已下步骤:

计算(x,y)处所属多边形的Z值,

将Z值与对应位置的Z缓冲中的Z值进行比较,如果刚算出来的这个更小(意味着更靠前),就更新Z缓冲,并把这个点的像素值更新到帧缓冲存储的相应位置;如果刚算出来的更大,则不发生任何更新,

遍历完全部的点,把帧缓冲中的图像显示出来,算法结束.

Z的计算使用平面方程

* 1. **扫描线算法**
  2. **区域分割算法**

区域分割算法将投影平面分割成区域，考察区域内的图象。如果容易决定在这个区域内某些多边形是可见的，那么就可以显示那些可见的多边形，完成对这一区域的显示任务。否则，就将区域再分割成小的区域，对小的区域递归地进行判断。由于区域逐渐变小，在每个区域内的多边形逐渐变少，最终总可以判定哪些多边形是可见的。这个算法利用的区域的相关性，这种相关性是指位于适当大小的区域内的所有象素，表示的其实是同一个表面。

1. **光照模型**
   1. **环境光**

**I=Ia· Ka**

其中I是可见表面的亮度，Ia是环境光线的总亮度，κa是物体表面对环境光线的反射系数，它在0到1之间.

* 1. **漫反射**

**Id=Ip · Kd ·COSθ**

其中Id是漫反射引起的可见表面上一点的亮度。Ip是点光源发出的入射光线引起的亮度。κd是漫反射系数，它的取值在0到1之间，随物体材料不同而不同。 θ是可见表面法向N和点光源方向L之间的夹角，即入射角，它应该在0°到90°之间。

为了简化公式中余弦值的实际计算，可以假定向量N和L都已经正规化，即已经是长度为1的单位向量，这样就可以使用向量的数量积或内积。

因为这时Cosθ=L·N，于是得: Id=Ip · Id ·(L · N)

将环境光线和漫反射的效果结合起来，计算亮度的公式应该写成：

**I=Ia· κa + Ip · Id ·(L·N)**

设R是光线从光源发出到达表面再返回的距离，则:

**I=Ia· κa + Ip · Id ·(L·N)/R2**

可通过用r+k代替R2来获得：其中r是光源到表面的距离，k是根据经验选取的一个常数。

**I=Ia· Ka + Ip · Id ·(L·N)/(r+k)**

* 1. **镜面反射**

****

* 1. **光的衰减**

**平方衰减:F(d)=1/d2**

**改进:f(d)=min(1/(C0+C1d+C2d2),1)**

* 1. **冯氏模型**
     1. **考虑环境光,镜面反射光,漫反射光,光源到物体表面的衰减,冯氏光照模型为:**

*:*环境光反射系数

*:*环境光总亮度

光线衰减函数, f(d)=min(1/(C0+C1d+C2d2),1)

*:* 点光源发出的入射光线引起的亮度

漫反射系数

*:* 点光源方向L,(正规化)

*:* 表面法向N(正规化)

*:*镜面反射经验常数

*:* 反射光线方向向量(正规化)

*V:*观察视线方向向量(正规化)

*n:*表面的镜面反射性质常数,越理想的镜面,n越大

* + 1. **考虑光从物体表面到人眼的衰减,最终亮度*I’*为:**

权重系数

用户指定的一个常数

* + 1. **冯氏方法绘制流程**
       1. **计算多边形的单位法向量**
       2. **计算多边形顶点的单位法向量**
       3. **在扫描线消隐算法中,对多边形定点的法向量进行双线性插值,计算出多边形内部各点的法向量**
       4. **利用光照模型计算多边形上每一点的亮度.**