****选做实验二：扫描线填充算法****

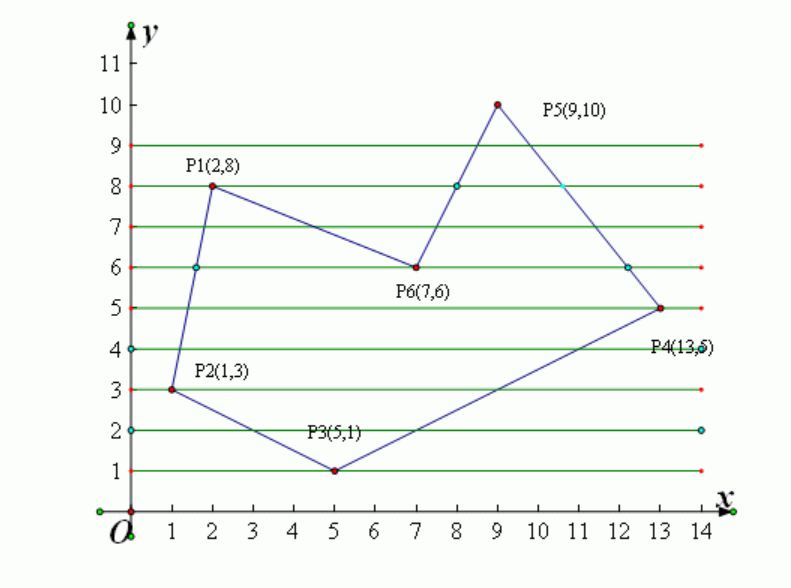
扫描线算法适合对矢量图形进行区域填充，只需要直到多边形区域的几何位置，不需要指定种子点，适合计算机自动进行图形处理的场合使用，比如电脑游戏和三维CAD软件的渲染等。

**一、扫描线算法的基本思想**

扫描线填充算法的基本思想是：用水平扫描线从上到下（或从下到上）扫描由多条首尾相连的线段构成的多边形，每根扫描线与多边形的某些边产生一系列交点。将这些交点按照x坐标排序，将排序后的点两两成对，作为线段的两个端点，以所填的颜色画水平直线。多边形被扫描完毕后，颜色填充也就完成了。扫描线填充算法也可以归纳为以下4个步骤：

（1）求交，计算扫描线与多边形的交点 （2）交点排序，对第2步得到的交点按照x值从小到大进行排序 （3）颜色填充，对排序后的交点两两组成一个水平线段，以画线段的方式进行颜色填充； （4） 是否完成多边形扫描？如果是就结束算法，如果不是就改变扫描线，然后转第1步继续处理；

整个算法的关键是第1步，需要用尽量少的计算量求出交点，还要考虑交点是线段端点的特殊情况，最后，交点的步进计算最好是整数，便于光栅设备输出显示。对于每一条扫描线，如果每次都按照正常的线段求交算法进行计算，则计算量大，而且效率底下，如图（6）所示：



观察多边形与扫描线的交点情况，可以得到以下两个特点：

（1） 每次只有相关的几条边可能与扫描线有交点，不必对所有的边进行求交计算；

（2） 相邻的扫描线与同一直线段的交点存在步进关系，这个关系与直线段所在直线的斜率有关；

第一个特点是显而易见的，为了减少计算量，扫描线算法需要维护一张由“活动边”组成的表，称为“活动边表（AET）”。例如扫描线4的“活动边表”由P1P2和P3P4两条边组成，而扫描线7的“活动边表”由P1P2、P6P1、P5P6和P4P5四条边组成。

第二个特点可以进一步证明，假设当前扫描线与多边形的某一条边的交点已经通过直线段求交算法计算出来，得到交点的坐标为（x, y），则下一条扫描线与这条边的交点不需要再求交计算，通过步进关系可以直接得到新交点坐标为（x + △x, y + 1）。前面提到过，步进关系△x是个常量，与直线的斜率有关，下面就来推导这个△x。

假设多边形某条边所在的直线方程是：ax + by + c = 0，扫描线yi和下一条扫描线yi+1与该边的两个交点分别是（xi，yi）和（xi+1，yi+1），则可得到以下两个等式：

axi + byi + c = 0 (等式 1)

axi+1 + byi+1 + c = 0 (等式 2)

由等式1可以得到等式3：

xi = -(byi + c) / a (等式 3)

同样，由等式2可以得到等式4：

xi+1 = -(byi+1 + c) / a (等式 4)

由等式 4 – 等式3可得到

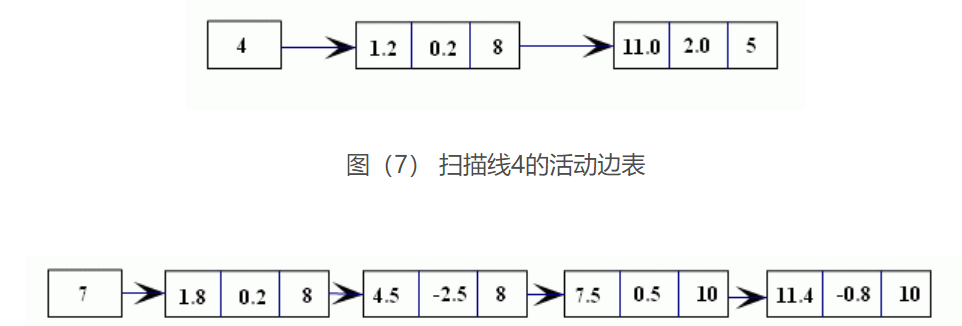
xi+1 – xi = -b (yi+1 - yi) / a

由于扫描线存在yi+1 = yi + 1的关系，将代入上式即可得到：

xi+1 – xi = -b / a

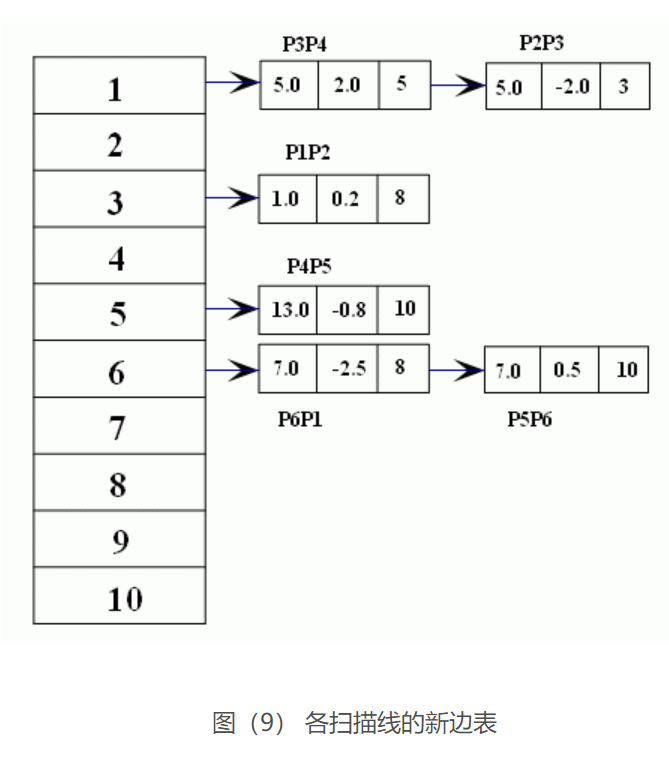
即△x = -b / a，是个常量（直线斜率的倒数）。

“活动边表”是扫描线填充算法的核心，整个算法都是围绕者这张表进行处理的。要完整的定义“活动边表”，需要先定义边的数据结构。每条边都和扫描线有个交点，扫描线填充算法只关注交点的x坐标。每当处理下一条扫描线时，根据△x直接计算出新扫描线与边的交点x坐标，可以避免复杂的求交计算。一条边不会一直待在“活动边表”中，当扫描线与之没有交点时，要将其从“活动边表”中删除，判断是否有交点的依据就是看扫描线y是否大于这条边两个端点的y坐标值，为此，需要记录边的y坐标的最大值。根据以上分析，边的数据结构可以定义如下：根据EDGE的定义，扫描线4和扫描线7的“活动边表”就分别如图（7）和图(8)所示：





前面提到过，扫描线算法的核心就是围绕“活动边表（AET）”展开的，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）”，存放该扫描线第一次出现的边。当算法处理到某条扫描线时，就将这条扫描线的“新边表”中的所有边逐一插入到“活动边表”中。“新边表”通常在算法开始时建立，建立“新边表”的规则就是：如果某条边的较低端点（y坐标较小的那个点）的y坐标与扫描线y相等，则该边就是扫描线y的新边，应该加入扫描线y的“新边表”。上例中各扫描线的“新边表”如下图所示：

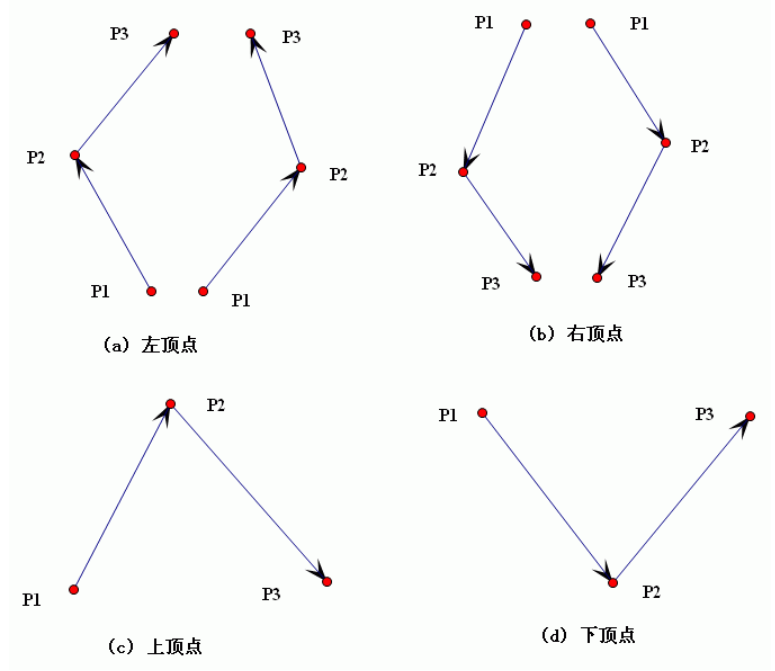


讨论完“活动边表（AET）”和“新边表（NET）”，就可以开始算法的具体实现了，但是在进一步详细介绍实现算法之前，还有以下几个关键的细节问题需要明确：

**（1） 多边形顶点处理**

在对多边形的边进行求交的过程中，在两条边相连的顶点处会出现一些特殊情况，因为此时两条边会和扫描线各求的一个交点，也就是说，在顶点位置会出现两个交点。当出现这种情况的时候，会对填充产生影响，因为填充的过程是成对选择交点的过程，错误的计算交点个数，会造成填充异常。

假设多边形按照顶点P1、P2和P3的顺序产生两条相邻的边，P2就是所说的顶点。多边形的顶点一般有四种情况，如图（10）所展示的那样，分别被称为左顶点、右顶点、上顶点和下顶点：



左顶点――P1、P2和P3的y坐标满足条件 ：y1 < y2 < y3；

右顶点――P1、P2和P3的y坐标满足条件 ：y1 > y2 > y3；

上顶点――P1、P2和P3的y坐标满足条件 ：y2 > y1 && y2 > y3；

下顶点――P1、P2和P3的y坐标满足条件 ：y2 < y1 && y2 < y3；

对于左顶点和右顶点的情况，如果不做特殊处理会导致奇偶奇数错误，常采用的修正方法是修改以顶点为终点的那条边的区间，将顶点排除在区间之外，也就是删除这条边的终点，这样在计算交点时，就可以少计算一个交点，平衡和交点奇偶个数。结合前文定义的“边”数据结构：EDGE，只要将该边的ymax修改为ymax – 1就可以了。

对于上顶点和下顶点，一种处理方法是将交点计算做0个，也就是修正两条边的区间，将交点从两条边中排除；另一种处理方法是不做特殊处理，就计算2个交点，这样也能保证交点奇偶个数平衡。

**（2） 水平边的处理**

水平边与扫描线重合，会产生很多交点，通常的做法是将水平边直接画出（填充），然后在后面的处理中就忽略水平边，不对其进行求交计算。

**（3） 如何避免填充越过边界线**

边界像素的取舍问题也需要特别注意。多边形的边界与扫描线会产生两个交点，填充时如果对两个交点以及之间的区域都填充，容易造成填充范围扩大，影响最终光栅图形化显示的填充效果。为此，人们提出了“左闭右开”的原则，简单解释就是，如果扫描线交点是1和9，则实际填充的区间是[1,9），即不包括x坐标是9的那个点。

**二、扫描线算法实现**

#include<GL\glut.h>

#include <stdio.h>

#include <math.h>

#include <malloc.h>

#define COLOR\_NEW 1.0,0.0,0.0

#define FALSE 0

#define TRUE 1

struct Point

{

int x;

int y;

Point(int a=0,int b=0)

{

x=a;

y=b;

}

};

struct Bian\_list

{

float jx;

float dx;

int ymin;

int ymax;

bool sp;

};

struct Xb

{

float x[10];

int num;

};

struct Huo\_list

{

int num;

Bian\_list \* next[10];

};

void InitiateHuo\_list(Huo\_list \* H)

{

H->num=0;

H->next[0]=NULL;

}

void InsertHuo\_list(Huo\_list \* H,Bian\_list \* b\_list)

{

H->next[H->num]=b\_list;

H->num++;

}

void Deleteb\_list(Huo\_list \* H,int j)

{

int i;

for(i=j;i<H->num;i++)

{

H->next[i]=H->next[i+1];

}

H->num--;

}

void pai\_xuHuo\_list(Xb \* xb)

{

int n = xb->num;

int i,j;

float temp;

for (i=0;i<n-1;i++)

{

temp=xb->x[i+1];

j=i;

while(j>-1&&temp<xb->x[j])

{

xb->x[j+1]=xb->x[j];

j--;

}

xb->x[j+1]=temp;

}

}

//ScanLine用于设多边形存储的顶点序列，顶点数等于边数

void ScanLine(Point \* Polygon\_point,int num\_point)

{

//point\_ymax,point\_ymin,num\_smx用来确定扫描线链表的个数

int i,j,point\_ymax,point\_ymin,num\_smx;

point\_ymax=Polygon\_point[1].y;

point\_ymin=Polygon\_point[1].y;

for(i=0;i<num\_point;i++)

{

if (Polygon\_point[i].y>point\_ymax)

point\_ymax=Polygon\_point[i].y;

if (Polygon\_point[i].y<point\_ymin)

point\_ymin=Polygon\_point[i].y;

}

num\_smx=point\_ymax-point\_ymin;

//建立、初始化边表

Bian\_list \* b\_list=(Bian\_list \* )malloc(num\_point \* sizeof(Bian\_list));//n个点n-1条边

for(i=0;i<num\_point-1;i++)

{

//jx其实是保存扫描线与边的当前交点值，即ymin对应的jx,Cpoint(jx,ymin)

if(Polygon\_point[i].y<Polygon\_point[i+1].y)

b\_list[i].jx=Polygon\_point[i].x;

else

b\_list[i].jx=Polygon\_point[i+1].x;

if(Polygon\_point[i].y!=Polygon\_point[i+1].y)

{

//忽略斜率为无穷，即忽略水平线

b\_list[i].dx=(float)(Polygon\_point[i].x-Polygon\_point[i+1].x)/

(Polygon\_point[i].y-Polygon\_point[i+1].y);

b\_list[i].sp=FALSE;

}

else

{

//dx取较大的x

b\_list[i].dx=(Polygon\_point[i].x>Polygon\_point[i+1].x)?Polygon\_point[i].x:Polygon\_point[i+1].x;

//jx取较小的x

b\_list[i].jx=(Polygon\_point[i].x<Polygon\_point[i+1].x)?Polygon\_point[i].x:Polygon\_point[i+1].x;

b\_list[i].sp=TRUE;

}

b\_list[i].ymax=(Polygon\_point[i].y>Polygon\_point[i+1].y)?Polygon\_point[i].y:Polygon\_point[i+1].y;

b\_list[i].ymin=(Polygon\_point[i].y<Polygon\_point[i+1].y)?Polygon\_point[i].y:Polygon\_point[i+1].y;

}

//建立、初始化新边表

Xb xb;

Huo\_list \* h\_list=new(Huo\_list);

h\_list->num=0;

for(i=point\_ymin;i<point\_ymax;)

{

for(j=0;j<num\_point-1;j++)

if(i==b\_list[j].ymin)

{

InsertHuo\_list(h\_list,&b\_list[j]);

}

int n=0;

xb.num=0;

while(n<h\_list->num)

{

if(h\_list->next[n]->sp)

{

xb.x[xb.num]=h\_list->next[n]->jx;

xb.num++;

xb.x[xb.num]=h\_list->next[n]->dx;

xb.num++;

}

else

{

xb.x[xb.num]=h\_list->next[n]->jx+h\_list->next[n]->dx \* (i-h\_list->next[n]->ymin);

xb.num++;

}

n++;

}

pai\_xuHuo\_list(&xb);

bool tc=FALSE;

for(j=0;j<xb.num;j++,j++)

{

int x,x1,xr;

x1=xb.x[j];

xr=xb.x[j+1];

x=x1;

while(x<=xr)

{

glColor3f(COLOR\_NEW);

glVertex2i(x,i);

x++;

}

}

if(i==point\_ymin)

i--;

i++;

for(j=0;j<h\_list->num;j++)

if(i==h\_list->next[j]->ymax)

{

Deleteb\_list(h\_list,j);

}

if(i==point\_ymin)

i++;

}

}

void LineDDA(int x0,int y0,int x1,int y1)

{

float dy,dx,x,y,m;

dx=x1-x0;

dy=y1-y0;

if(dx!=0)

{

m=dy/dx;

if(m<=1&&m>=-1)

{

y=y0;

for(x=x0;x<=x1;x++)

{

y=y0;

for(x=x0;x<=x1;x++)

{

glVertex2i(x,int(y+0.5));

y+=m;

}

}

if(m>1||m<-1)

{

m=1/m;

x=x0;

for(y=y0;y<=y1;y++)

{

glVertex2i(int(x+0.5),y);

x+=m;

}

}

}else

{

int x=x0,y;

y=(y0<=y1)?y0:y1;

int d=fabs((double)(y0-y1));

while(d>=0)

{

glVertex2i(x,y);

y++;

d--;

}

}

}

}

void ScanLineSegment()

{

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(0.0,0.0,0.0);

glBegin(GL\_POINTS);

Point spt[8];

spt[0]=Point(230,230);

spt[1]=Point(290,230);

spt[2]=Point(275,280);

spt[3]=Point(230,280);

spt[4]=Point(245,260);

spt[5]=Point(250,254);

spt[6]=Point(235,245);

spt[7]=Point(230,230);

ScanLine(spt,8);

glEnd();

glFlush();

}

void main(int argc, char \* argv[])

{

glutInit(&argc,argv);

glutInitDisplayMode(GLUT\_SINGLE|GLUT\_RGB);

glutInitWindowPosition(50,100);

glutInitWindowSize(500,500);

glutCreateWindow("扫描填充算法");

glClearColor(1.0,1.0,1.0,0.0);

glMatrixMode(GL\_PROJECTION);

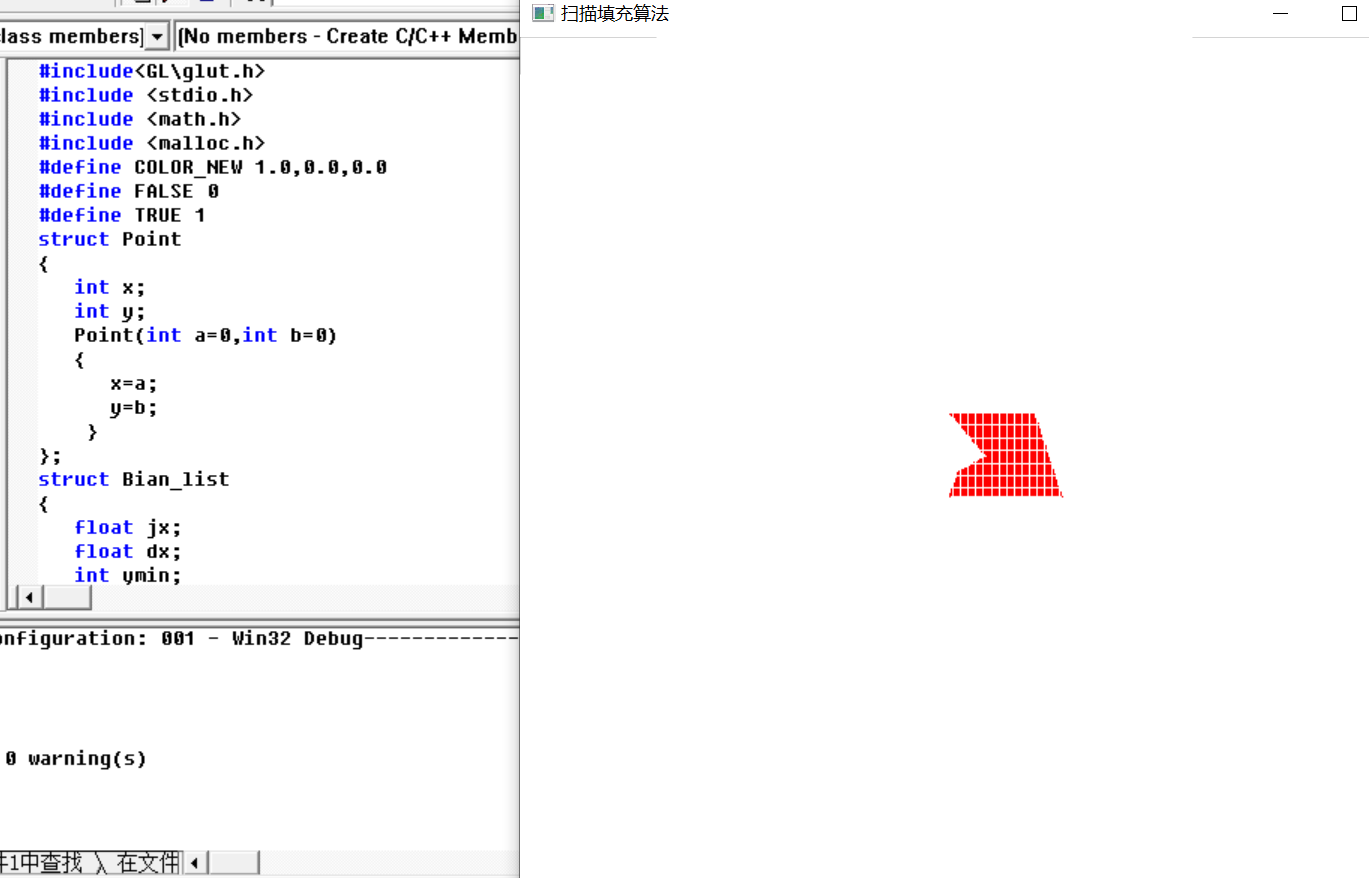
gluOrtho2D(0.0,500.0,0.0,500.0);

glutDisplayFunc(ScanLineSegment);

glutMainLoop();

}

三、实验结果

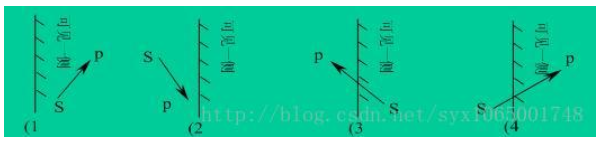


选做实验四：Sutherland-Hodgeman算法

**一、基本思想**

是一次用窗口的一条边裁剪多边形。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：可见一侧；不可见一侧。  
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，

如图：

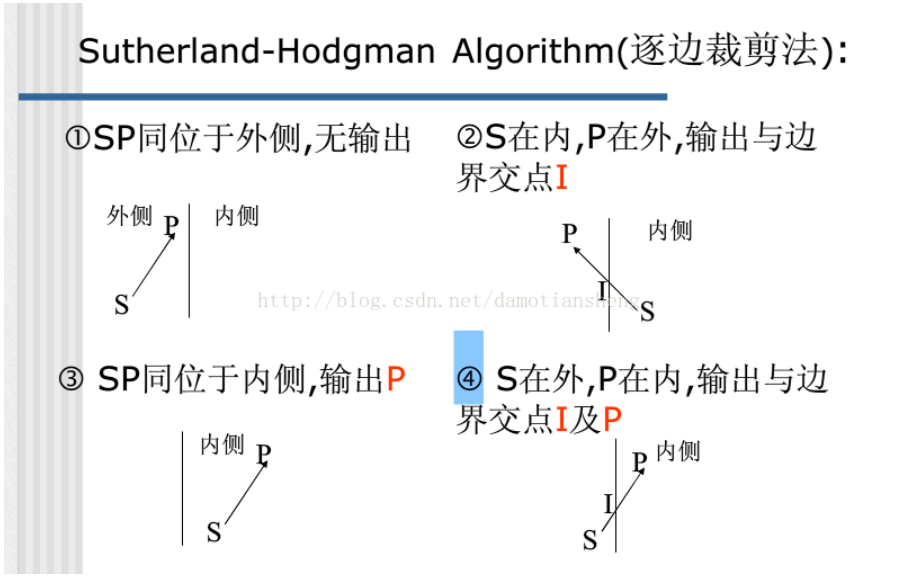


情况（1）仅输出1个顶点P；

情况（2）输出0个顶点；

情况（3）输出线段SP与裁剪线的1个交点I；

情况（4）输出线段SP与裁剪线的1个交点I和1个终点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，只是判断点在窗口哪一侧改变求线段SP与裁剪边的交点的算法。

代码过程：

1首先画一个多边形，然后记录边数和边的2个顶点。

2.运行算法。

**二、算法实现：**

#include<iostream>

#include<GL\glut.h>

using namespace std;

int result[50][2] = { {10, 80},{70, 10},{80, 80},{90, 10},{130, 80},{80, 130}};

enum mark { L, B, R, T};

int c = -1;

struct node {

float data[2];

node\* next;

node\* pre;

};

void initNode(node \*&head) {

head = new node;

head->data[0] = 0;

head->data[1] = 0;

head->next = NULL;

head->pre = NULL;

node \*p = head;

node \*q = NULL;

//此处直接输入的点的起始数目进行初始化

for (int i = 0; i < 6; ++i) {

q = new node;

q->data[0] = result[i][0];

q->data[1] = result[i][1];

q->pre = p;

p->next = q;

p = q;

}

p->next = head;

head->pre = p;

}

//分别对应每个边裁剪

void testPoint(node \*&first, node \*&head, int& edg, float xl, float xr, float yb, float yt) {

if (edg == 4) {

return;

}

node\* p = first;

node\* q = p->pre;

//依次对每个点处理，在线内的就存入result数组

while (p != head) {

int x = 0;

int y = 0;

if (q == head) {

q = p->pre->pre;

}

float k = 0;

if (q->data[0] - p->data[0] != 0) {

k = (q->data[1] - p->data[1])\*1.0 / (q->data[0] - p->data[0]);

}

switch (edg) {

case L:

//在边界之外的重新计算点

if (((p->data[0] <= xl) && (q->data[0] >= xl)) || ((p->data[0] >= xl) && (q->data[0] <= xl))) {

if ((q->data[0] >= xl) && (q->data[0] <= xr)) {

c++;

result[c][0] = q->data[0];

result[c][1] = q->data[1];

}

//相连的点在边界内则计算新的交点

x = xl;

y = p->data[1] + (x - p->data[0]) \* k;

c++;

result[c][0] = x;

result[c][1] = y;

}

//处理的点再边界内,如果相连点也在边界内则把相连点插入

else{

c++;

result[c][0] = q->data[0];

result[c][1] = q->data[1];

}

break;

case R:

if (((p->data[0] >= xr) && (q->data[0] <= xr)) || ((p->data[0] <= xr) && (q->data[0] >= xr))) {

if ((q->data[0] >= xl) && (q->data[0] <= xr)) {

c++;

result[c][0] = q->data[0];

result[c][1] = q->data[1];

}

x = xr;

y = p->data[1] + (x - p->data[0])\*k;

c++;

result[c][0] = x;

result[c][1] = y;

}

else{

c++;

result[c][0] = q->data[0];

result[c][1] = q->data[1];

}

break;

case B:

if (((p->data[1] <= yb) && (q->data[1] >= yb)) || ((p->data[1] >= yb) && (q->data[1] <= yb))) {

if ((q->data[1] >= yb) && (q->data[1] <= yt)) {

c++;

result[c][0] = q->data[0];

result[c][1] = q->data[1];

}

if (k == 0) { //处理两点垂直和平行情况

if (p->data[1] != q->data[1]) {

y = yb;

x = p->data[0];

}

else {

break;

}

}

if (k != 0) {

y = yb;

x = p->data[0] + (y - p->data[1]) / k;

}

c++;

result[c][0] = x;

result[c][1] = y;

}

else{

c++;

result[c][0] = q->data[0];

result[c][1] = q->data[1];

}

break;

case T:

if (((p->data[1] >= yt) && (q->data[1] <= yt)) || ((p->data[1] <= yt) && (q->data[1] >= yt))) {

if ((q->data[1] >= yb) && (q->data[1] <= yt)) {

c++;

result[c][0] = q->data[0];

result[c][1] = q->data[1];

}

if (k == 0) {

if (p->data[1] != q->data[1]) {

y = yb;

x = p->data[0];

}

else {

break;

}

}

if (k != 0) {

y = yt;

x = p->data[0] + (y - p->data[1]) / k;

}

c++;

result[c][0] = x;

result[c][1] = y;

}

else{

c++;

result[c][0] = q->data[0];

result[c][1] = q->data[1];

}

break;

default:

break;

}

//对处理点更新

p = p->next;

q = p->pre;

}

//对输入点进行更新

p = head;

node\* qq = NULL;

for (int i = 0; i <= c; ++i) {

qq = new node;

qq->data[0] = result[i][0];

qq->data[1] = result[i][1];

qq->pre = p;

p->next = qq;

p = qq;

}

p->next = head;

head->pre = p;

c = -1;

edg++;

p = p->next->next;

testPoint(p, head, edg, xl, xr, yb, yt);

}

//对每个边进行裁剪

void getPoint(node\* &head, float xl, float xr, float yb, float yt) {

int n = 0;

node\* p = head->next;

testPoint(p, head, n, xl, xr, yb, yt);

}

void drawPol() {

//处理多边形顶点

node \*h = new node;

initNode(h);

//绘制原多边形

node \*p = h->next;

glColor3f(1.f, 0.0f, 0.0f);

glBegin(GL\_LINE\_LOOP);

while (p != h) {

glVertex2f(p->data[0], p->data[1]);

p = p->next;

}

glEnd();

//绘制

getPoint(h, 20, 120, 20, 120);

//绘制裁剪后的多边形

p = h->next;

glColor3f(1.f, 1.0f, 0.0f);

glLineWidth(3);

glBegin(GL\_LINE\_LOOP);

while (p != h) {

glVertex2f(p->data[0], p->data[1]);

p = p->next;

}

glEnd();

}

void initPlo() {

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(0.f, 1.0f, 1.0f);

//绘制矩形线框

glBegin(GL\_LINE\_LOOP);

glVertex2f(20.f, 20.f);

glVertex2f(20.f, 120.f);

glVertex2f(120.f, 120.f);

glVertex2f(120.f, 20.f);

glEnd();

drawPol();

glFlush();

}

void changeSize(int w, int h) {

if (h == 0) {

h = 1;

}

glViewport(0, 0, w, h);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

if (w <= h) {

glOrtho(0.0f, 200.0f, 0.0f, 200.0f \* h / w, 1.0f, -1.0f);

}

else {

glOrtho(0.0f, 200.0f \* w / h, 0.0f, 200.0f, 1.0f, -1.0f);

}

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

}

void setUpRc() {

glClearColor(0.f, 0.5f, 0.5f, 1.f);

}

void main(int argc, char \*\*argv) {

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

glutInitWindowSize(600, 600);

glutCreateWindow("Plo");

glutDisplayFunc(initPlo);

glutReshapeFunc(changeSize);

setUpRc();

glutMainLoop();

}

**三、结果示例：**

