实验二　基于窗口的 Liang-Barsky 的二维直线段裁剪算法的实现

姓名　刘子言　学号 20002462　专业班级 计 203　成绩_____

实验日期_2022.4.28__ 实验地点 线上　指导教师(签名)　李建华

一．实验目的

1、学习 Liang-Barsky 直线裁剪算法。
2、实现基于窗口的直线裁剪。

二．实验工具与设备

计算机，开发软件为 CodeBlocks （配置 OpenGL）

三、实验内容

1、在本科学习平台（s.ecust.edu.cn）资料栏 lineClipping 文件夹下，有 2 个文件 lineClipping.cpp 和 lineClipping.h。

①在 lineClipping.h 头文件中，定义了点类型 point 和矩形类型 rect

②在 lineClipping.cpp 源文件中，定义了梁友栋-Barsky 直线裁剪算法基础代码

```
int Clip_Top(float p,float q,float &umax,float &umin);
void Line_Clipping(vector<point> &points,rect & winRect);
```

2、在实验一的基础上，利用键盘橡皮筋技术交互绘制要裁剪的直线段，键盘’p’确定直线段端点；利用鼠标橡皮筋技术交互绘制裁剪窗口，鼠标左键单击确定裁剪窗口主对角线位置。

```
rect winObj;                    //标准矩形裁剪窗口对象
int iKeyPointNum = 0;   //键盘'p'确定直线段端点的数目：0-无、1-起始点、2-终止点
int xKey=0,yKey=0;                    //直线段橡皮筋时，保留鼠标移动时的坐标值
int iMousePointNum= 0;//鼠标单击确定裁剪窗口点的数目：0-无、1-起始点、2-终止点
int xMouse=0,yMouse=0;                //裁剪窗口橡皮筋时，保留鼠标移动时的坐标值
```

3、键盘‘c'实现基于窗口的直线裁剪，观察 Liang-Barsky 直线裁剪算法对直线的裁剪结果。

实验代码：

1、lineClipping.h

```
#include <iostream>
#include <vector>
#include <windows.h>
#include <GL/glut.h>
using namespace std;
```

```
//点类型 point
typedef struct Point {
    int x, y;
    Point(int a = 0, int b = 0)
    {
        x = a, y = b;
    }
}point;
//矩形类型 rect
typedef struct Rectangle{
    float w_xmin,w_ymin;
    float w_xmax,w_ymax;
    Rectangle(float xmin = 0.0, float ymin = 0.0,float xmax=0.0,float ymax=0.0){
        w_xmin = xmin;    w_ymin = ymin;
        w_xmax = xmax;    w_ymax = ymax;
    }
}rect;
int Clip_Top(float p,float q,float &umax,float &umin);
void Line_Clipping(vector<point> &points,rect & winRect);
```

2、main.cpp

```
#include <iostream>
#include <vector>
using namespace std;
#include "lineClipping.h"

int screenWidth = 600, screenHeight = 400;
int iKeyPointNum = 0;                  //键盘'p'确定直线段端点的数目：0-无、1-起始点、2-终止点
int iMousePointNum= 0;                 //鼠标单击确定裁剪窗口点的数目：0-无、1-起始点、2-终止点
int xw1=0,yw1=0;                       //绘制裁剪窗口的起始坐标值
int xw2=0,yw2=0;                       //绘制裁剪窗口时，保留鼠标移动时的坐标值
rect winObj;                               //标准矩形裁剪窗口对象
point p1,p2;                               //标准直线段端点
vector<point> points;

//以下为 Liang_Barsky 裁剪算法代码
/*********************************************
*如果 p 参数<0，计算、更新 umax，保证 umax 是最大 u 值
*如果 p 参数>0，计算、更新 umin，保证 umin 是最小 u 值
*如果 umax>umin，返回 0，否则返回 1
*********************************************/
int Clip_Top(float p,float q,float &umax,float &umin)
{
```

```
    float r=0.0;
    if(p<0.0) //线段从裁剪窗口外部延伸到内部，取最大值 r 并更新 umax
    {
        r=q/p;
        if (r>umin) return 0;       //umax>umin 的情况，弃之
        else if (r>umax)    umax=r;
    }
    else if(p>0.0)          //线段从裁剪窗口内部延伸到外部,取最小值 r 并更新 umin
        {
            r=q/p;
            if (r<umax) return 0;       //umax>umin 的情况，弃之
            else if(r<umin)    umin=r;
        }
       else          //p=0 时，线段平行于裁剪窗口
          if(q<0.0) return 0;
     return 1;
}
/*************************************************************
*已知 winRect：矩形对象，存放标准裁剪窗口 4 条边信息
*      points：点的动态数组，存放直线 2 个端点信息
*根据裁剪窗口的左、右边界，求 umax；
*根据裁剪窗口的下、上边界，求 umin
*如果 umax>umin，裁剪窗口和直线无交点，否则求裁剪后直线新端点
*************************************************************/
void Line_Clipping(vector<point> &Points,rect & winRect)
{
    //比较左、右边界，获得最大的 umax
    point &p_1=Points[0],&p_2=Points[1];
    float dx=p_2.x-p_1.x,dy=p_2.y-p_1.y,umax=0.0,umin=1.0;
    point p=p_1;
    if (Clip_Top(-dx,p_1.x- winRect.w_xmin,umax,umin))   //左边界
        if (Clip_Top(dx,winRect.w_xmax-p_1.x, umax,umin)) //右边界
            //比较下、上边界，获得最小的 umin
            if (Clip_Top(-dy,p_1.y- winRect.w_ymin, umax,umin)) //下边界
                if (Clip_Top(dy,winRect.w_ymax-p_1.y, umax,umin)) //上边界
                {//求裁剪后直线新端点（这里还是选择直接将结果赋给全局变量的方法）
                    p1.x=(int)(p.x+umax*dx);
                    p1.y=(int)(p.y+umax*dy);
                    p2.x=(int)(p.x+umin*dx);
                    p2.y=(int)(p.y+umin*dy);
                }
}

void Initial(void)//初始化窗口
```

```
{
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f); //设置窗口背景颜色为白色
}
void ChangeSize(int w, int h)
{
    screenWidth = w;
    screenHeight = h;
    glViewport(0, 0, w, h);                    //指定窗口显示区域
    glMatrixMode(GL_PROJECTION);               //设置投影参数
    glLoadIdentity();
    gluOrtho2D(0.0,screenWidth,0.0,screenHeight);
}

void Key(unsigned char key, int xMouse, int yMouse)     //指定键盘响应函数
{
    switch(key)
    {
        case 'p':     //绘制直线段
            if(iKeyPointNum == 0 || iKeyPointNum == 2)
            {
                iKeyPointNum = 1;
                p1.x = xMouse;
                p1.y = screenHeight - yMouse;
            }
            else
            {
                iKeyPointNum = 2;
                p2.x = xMouse;
                p2.y = screenHeight - yMouse;
                glutPostRedisplay();                    //窗口执行重新绘制操作
            }
            break;
        case 'c':     //进行直线的裁剪
            {
                points.push_back(p1);     //将直线两端点放入容器中
                points.push_back(p2);
                Line_Clipping(points, winObj);//调用 Liang-Barsky 直线裁剪函数
                points.pop_back();
                points.pop_back();          //将直线两端点从容器中释放
            }
            break;
        default: break;
    }
}
```

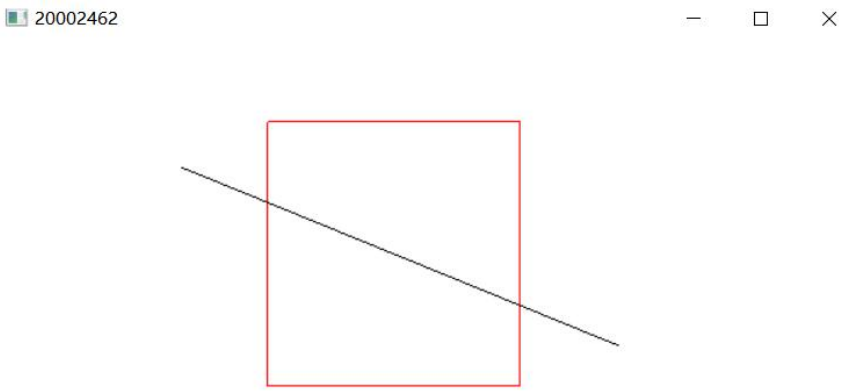
```
void MousePlot(GLint button, GLint action, GLint xMouse, GLint yMouse)   //指定鼠标响应函数
{
    if(button == GLUT_LEFT_BUTTON && action == GLUT_DOWN)//鼠标左击，绘制裁剪窗口
    {
        if(iMousePointNum == 0 || iMousePointNum == 2)
        {
            iMousePointNum = 1;
            xw1 = xMouse;
            yw1 = screenHeight - yMouse;
        }
        else
        {
            iMousePointNum = 2;
            xw2 = xMouse;
            yw2 = screenHeight - yMouse;

            //判断一下裁剪窗口边界值的大小，给 winObj 中的元素赋值
            if(xw2 > xw1){
                winObj.w_xmin = xw1;
                winObj.w_xmax = xw2;
            }
            else{
                winObj.w_xmin = xw2;
                winObj.w_xmax = xw1;
            }
            if(yw2 > yw1){
                winObj.w_ymin = yw1;
                winObj.w_ymax = yw2;
            }
            else{
                winObj.w_ymin = yw2;
                winObj.w_ymax = yw1;
            }
            glutPostRedisplay();                    //窗口执行重新绘制操作
        }
    }

    if(button == GLUT_RIGHT_BUTTON && action == GLUT_DOWN)//右击清除
    {
        iKeyPointNum = 0;
        iMousePointNum = 0;
        glutPostRedisplay();                    //窗口执行重新绘制操作
    }
}
```

```
void PassiveMouseMove (GLint xMouse, GLint yMouse)    //鼠标移动过程中函数
{
    if(iKeyPointNum == 1)//正在绘制直线段
    {
        p2.x = xMouse;
        p2.y = screenHeight - yMouse;
    }
    if(iMousePointNum == 1)//正在绘制裁剪窗口
    {
        xw2 = xMouse;
        yw2 = screenHeight - yMouse;
    }
    glutPostRedisplay();     //窗口执行重新绘制操作
}

void Display(void)
{
    glClear(GL_COLOR_BUFFER_BIT); //用当前背景色填充窗口
    glColor3f(1.0f, 0.0f, 0.0f); //设置裁剪窗口颜色为红色
    if(iMousePointNum >= 1)            //绘制裁剪窗口
    {
        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
        //参数 GL_FRONT_AND_BACK：表示显示模式将适用于物体的所有面；
        //参数 GL_LINE：表示显示线段，多边形用轮廓显示
        glRectf(xw1,yw1,xw2,yw2);
    }
    glColor3f(0.0f, 0.0f, 0.0f); //设置直线段颜色为黑色
    if(iKeyPointNum >= 1)                //绘制直线段
    {
        glBegin(GL_LINES);
        glVertex2i(p1.x,p1.y);
        glVertex2i(p2.x,p2.y);
        glEnd();
    }
    glutSwapBuffers(); //交换缓冲区
}

int main(int argc, char* argv[])      //主函数
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB); //使用“双缓存”以及 RGB 模型
    glutInitWindowSize(600,400);
    glutInitWindowPosition(200,200);
    glutCreateWindow("20002462");
```

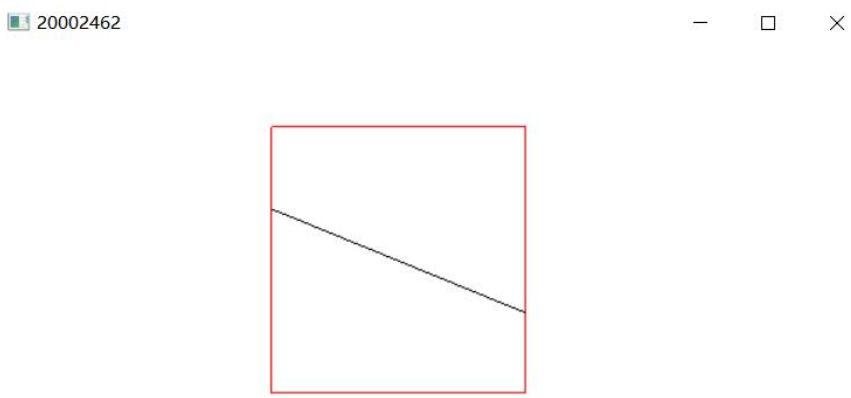
```
    glutDisplayFunc(Display);
    glutReshapeFunc(ChangeSize);                    //指定窗口在整形回调函数
    glutMouseFunc(MousePlot);                       //指定鼠标响应函数
    glutPassiveMotionFunc(PassiveMouseMove);        //指定鼠标移动响应函数
    glutKeyboardFunc(Key);                           //指定键盘响应函数

    Initial();
    glutMainLoop();
    return 0;
}
```

实验运行结果截图：

1、键盘键入‘p’，开始绘制直线段，鼠标移动，再起按‘p’，结束绘制；点击鼠标左键，移动鼠标，开始绘制裁剪窗口，再次点击左键，结束绘制。



2、键盘输入‘c’，进行直线裁剪，效果如下：



四、拓展实验

1、举一反三，观察 Liang-Barsky 直线裁剪算法对折线的裁剪结果，利用向量（Vector）保存折线端点序列；鼠标移动交互显示下一段折线的橡皮筋效果，键盘'p'确定折线端点，键盘'e'结束折线绘制。利用鼠标橡皮筋技术交互确定裁剪窗口，键盘'c'实现基于窗口的折线裁剪。

实验代码：

1、头文件与上一实验相同

2、main.cpp

```
#include <iostream>
#include <vector>
using namespace std;
#include "lineClipping.h"

int screenWidth = 600, screenHeight = 400;
int iKeyPointNum = 0;                  //键盘'p'确定直线段端点的数目：0-无、1-起始点、2-终止点
int iMousePointNum= 0;                 //鼠标单击确定裁剪窗口点的数目：0-无、1-起始点、2-终止点
int xw1=0,yw1=0;                       //裁剪窗口的起始坐标值
int xw2=0,yw2=0;                       //绘制裁剪窗口时，保留鼠标移动时的坐标值
//以下是针对折线做出的设计
rect winObj;                           //标准矩形裁剪窗口对象
point p[10];                           //用 point 类型的数组 记录标准折线段端点
point p_sub[10];                       //用 point 类型的数组 记录每一段裁剪出的点
int i=0;                               //记录折线段端点个数
int k=0;                               //记录裁剪过后的端点的个数
vector<point> points;                  //裁剪时承载端点的容器

//以下为 Liang_Barsky 裁剪算法代码
/*********************************************
*如果 p 参数<0，计算、更新 umax，保证 umax 是最大 u 值
*如果 p 参数>0，计算、更新 umin，保证 umin 是最小 u 值
*如果 umax>umin，返回 0，否则返回 1
*********************************************/
int Clip_Top(float p,float q,float &umax,float &umin)
{
    float r=0.0;
    if(p<0.0)  //线段从裁剪窗口外部延伸到内部，取最大值 r 并更新 umax
    {
        r=q/p;
        if (r>umin) return 0;      //umax>umin 的情况，弃之
        else if (r>umax)   umax=r;
    }
    else if(p>0.0)          //线段从裁剪窗口内部延伸到外部,取最小值 r 并更新 umin
        {
```

```
                r=q/p;
                if (r<umax) return 0;      //umax>umin 的情况，弃之
                else if(r<umin)    umin=r;
            }
          else          //p=0 时，线段平行于裁剪窗口
              if(q<0.0) return 0;
       return 1;
}
/*************************************************************
*已知 winRect：矩形对象，存放标准裁剪窗口 4 条边信息
*      points：点的动态数组，存放直线 2 个端点信息
*根据裁剪窗口的左、右边界，求 umax；
*根据裁剪窗口的下、上边界，求 umin
*如果 umax>umin，裁剪窗口和直线无交点，否则求裁剪后直线新端点
*************************************************************/
void Line_Clipping(vector<point> &Points,rect & winRect)
{
    //比较左、右边界，获得最大的 umax
    point &p_1=Points[0],&p_2=Points[1];
    float dx=p_2.x-p_1.x,dy=p_2.y-p_1.y,umax=0.0,umin=1.0;
    point p=p_1;
    if (Clip_Top(-dx,p_1.x- winRect.w_xmin,umax,umin))   //左边界
        if (Clip_Top(dx,winRect.w_xmax-p_1.x, umax,umin)) //右边界
            //比较下、上边界，获得最小的 umin
            if (Clip_Top(-dy,p_1.y- winRect.w_ymin, umax,umin)) //下边界
                if (Clip_Top(dy,winRect.w_ymax-p_1.y, umax,umin)) //上边界
                {//求裁剪后直线新端点（这里还是选择直接将结果赋给全局变量的方法）
                    p_sub[k].x=(int)(p.x+umax*dx);
                    p_sub[k].y=(int)(p.y+umax*dy);
                    p_sub[k+1].x=(int)(p.x+umin*dx);
                    p_sub[k+1].y=(int)(p.y+umin*dy);
                    k = k+2;
                }
}

void Initial(void)//初始化窗口
{
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f); //设置窗口背景颜色为白色
}
void ChangeSize(int w, int h)
{
    screenWidth = w;
    screenHeight = h;
    glViewport(0, 0, w, h);                              //指定窗口显示区域
```

```
    glMatrixMode(GL_PROJECTION);                    //设置投影参数
    glLoadIdentity();
    gluOrtho2D(0.0,screenWidth,0.0,screenHeight);
}

void Key(unsigned char key, int xMouse, int yMouse)     //指定键盘响应函数
{
    switch(key)
    {
        case 'p':   //绘制折线段
            if(iKeyPointNum == 0 || iKeyPointNum == 2)   //0 为开始画，1 为正在画，2 为画完了
            {
                if(iKeyPointNum == 2){
                    i = 0;k = 0;      //如果判断上一次已经画完了，则将 i、k 置为零，重头开始画
                }
                iKeyPointNum = 1;
                p[i].x = xMouse;
                p[i].y = screenHeight - yMouse;
                i++;
            }
            else
            {
                p[i].x = xMouse;
                p[i].y = screenHeight - yMouse;
                i++;
                glutPostRedisplay();                        //窗口执行重新绘制操作
            }
            break;
        case 'e':   //停止绘制折线段
            iKeyPointNum = 2;
            p[i].x = xMouse;
            p[i].y = screenHeight - yMouse;
            break;
        case 'c':   //进行折线的裁剪
            {
                for(int j=0;j<i;j++)  //每循环一次，进行折线段中一段的裁剪
                {
                    points.push_back(p[j]);
                    points.push_back(p[j+1]);
                    Line_Clipping(points, winObj);
                    points.pop_back();
                    points.pop_back();
                }
                break;
```

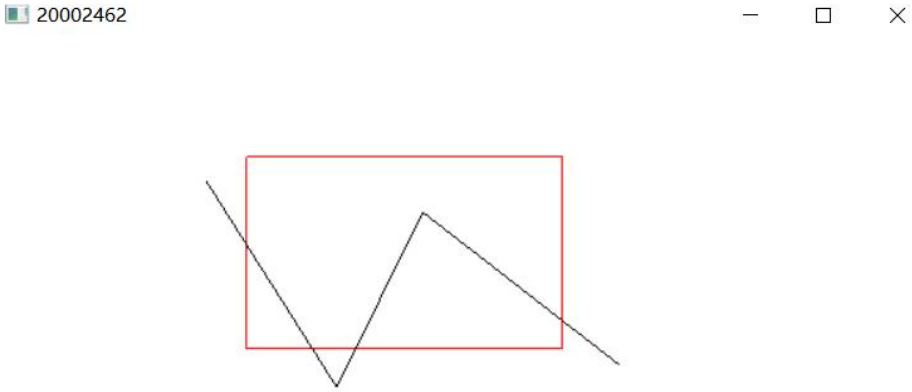
```
            }
        default: break;
    }
}

void MousePlot(GLint button, GLint action, GLint xMouse, GLint yMouse)   //指定鼠标响应函数
{
    if(button == GLUT_LEFT_BUTTON && action == GLUT_DOWN)//鼠标左击 绘制裁剪窗口
    {
        if(iMousePointNum == 0 || iMousePointNum == 2)
        {
            iMousePointNum = 1;
            xw1 = xMouse;
            yw1 = screenHeight - yMouse;
        }
        else
        {
            iMousePointNum = 2;
            xw2 = xMouse;
            yw2 = screenHeight - yMouse;

            //判断一下裁剪窗口的边界值大小，给 winObj 中的元素赋值
            if(xw2 > xw1){
                winObj.w_xmin = xw1;
                winObj.w_xmax = xw2;
            }
            else{
                winObj.w_xmin = xw2;
                winObj.w_xmax = xw1;
            }
            if(yw2 > yw1){
                winObj.w_ymin = yw1;
                winObj.w_ymax = yw2;
            }
            else{
                winObj.w_ymin = yw2;
                winObj.w_ymax = yw1;
            }
            glutPostRedisplay();                    //窗口执行重新绘制操作
        }
    }

    if(button == GLUT_RIGHT_BUTTON && action == GLUT_DOWN)     //右击清空屏幕
    {
```

```
            iKeyPointNum = 0;
            i = 0;k = 0;
            iMousePointNum = 0;
            glutPostRedisplay();                              //窗口执行重新绘制操作
        }
}

void PassiveMouseMove (GLint xMouse, GLint yMouse)    //鼠标移动过程中函数
{
    if(iKeyPointNum == 1)    //正在绘制折线
    {
        p[i].x = xMouse;
        p[i].y = screenHeight - yMouse;
    }
    if(iMousePointNum == 1)//正在绘制裁剪窗口
    {
        xw2 = xMouse;
        yw2 = screenHeight - yMouse;
    }
    glutPostRedisplay();     //窗口执行重新绘制操作
}

void Display(void)
{
    glClear(GL_COLOR_BUFFER_BIT); //用当前背景色填充窗口
    glColor3f(1.0f, 0.0f, 0.0f); //设置裁剪窗口的颜色为红色
    if(iMousePointNum >= 1)                  //绘制裁剪窗口
    {
        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
        //参数 GL_FRONT_AND_BACK：表示显示模式将适用于物体的所有面；
        //参数 GL_LINE：表示显示线段，多边形用轮廓显示
        glRectf(xw1,yw1,xw2,yw2);
    }

    glColor3f(0.0f, 0.0f, 0.0f); //设置折线颜色为黑色
    if(k == 2*i)     //判断，若裁剪完毕时，画裁剪以后得到的线段
    {
        for(int j=0;j<k;j=j+2)
        {
            if(iKeyPointNum >= 1)
            {
                glBegin(GL_LINES);
                glVertex2i(p_sub[j].x,p_sub[j].y);
                glVertex2i(p_sub[j+1].x,p_sub[j+1].y);
```

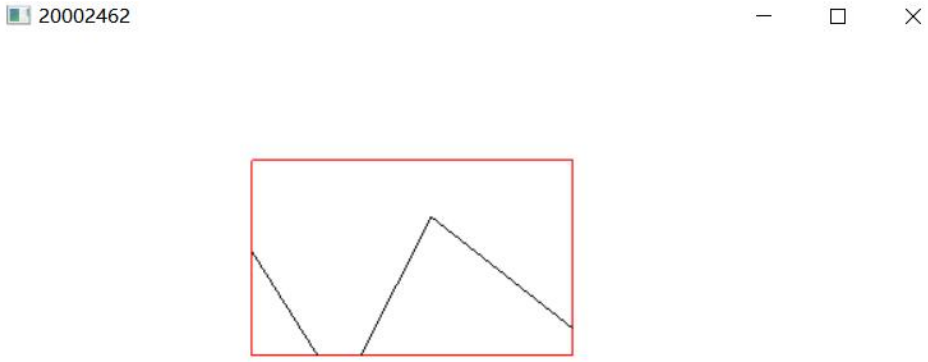
```
                glEnd();
            }
        }
    }
    else //否则，即为未裁剪完毕，就画完整的折线
    {
        for(int j=0;j<i;j++)
        {
            if(iKeyPointNum >= 1)
            {
                glBegin(GL_LINES);
                glVertex2i(p[j].x,p[j].y);
                glVertex2i(p[j+1].x,p[j+1].y);
                glEnd();
            }
        }
    }
    glutSwapBuffers(); //交换缓冲区
}

int main(int argc, char* argv[])
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB); //使用“双缓存”以及 RGB 模型
    glutInitWindowSize(600,400);
    glutInitWindowPosition(200,200);
    glutCreateWindow("20002462");

    glutDisplayFunc(Display);
    glutReshapeFunc(ChangeSize);                    //指定窗口在整形回调函数
    glutMouseFunc(MousePlot);                       //指定鼠标响应函数
    glutPassiveMotionFunc(PassiveMouseMove);        //指定鼠标移动响应函数
    glutKeyboardFunc(Key);                          //指定键盘响应函数

    Initial();
    glutMainLoop();
    return 0;
}
```

实验运行结果截图：

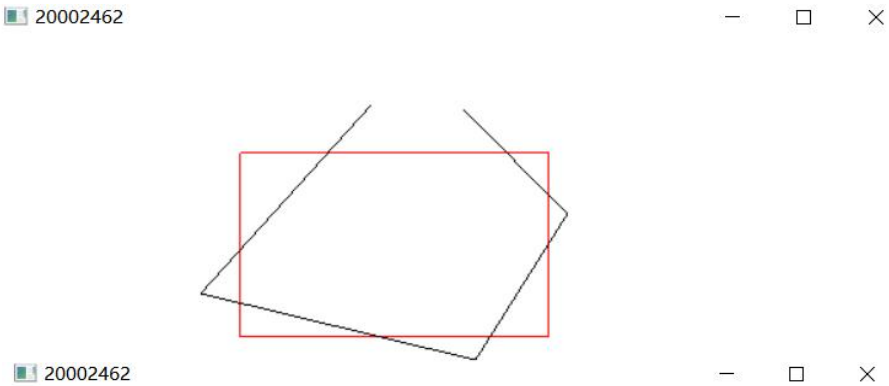
1、键盘键入‘p’，开始绘制折线，鼠标移动，每按下一个‘p’，折线定下一处拐点，直到键盘输入‘e’，结束折线的绘制。点击鼠标左键，移动鼠标，开始绘制裁剪窗口，再次点击左键，结束裁剪窗口的绘制。鼠标右击为清空重画。

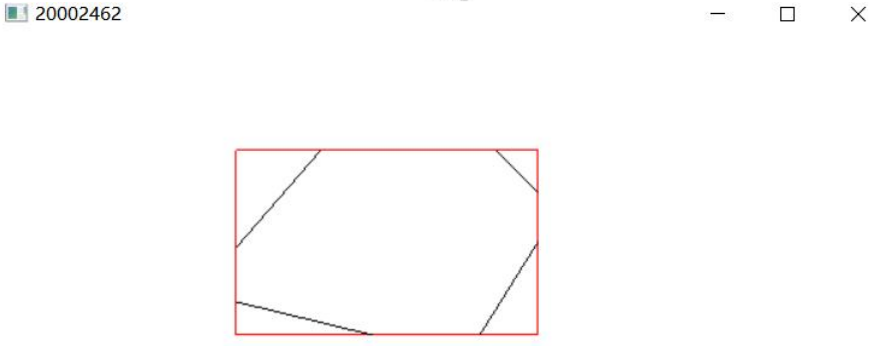


2、键盘输入‘c’，进行折线的裁剪，结果如下：



3、本实验代码还支持多次绘制，即绘制过程中，允许重新绘制折线、裁剪窗口，裁剪之后，可以接着进行下一次折线的绘制和裁剪工作，如以下折线就是在上一张截图的基础上重新绘制的折线段，运用原来的裁剪窗口进行裁剪，结果如下：





五．思考题

1．Liang-Barsky 的核心思想是什么？

答：Liang-Barsky 算法的核心思想，是将二维裁剪问题转化为二次一维的裁剪问题。

该算法的基本出发点，是直线的参数方程。算法将直线段与窗口边界的实交点和虚交点分为两组：下限组以 $p_k < 0$ 为特征，表示在该处直线段从裁剪边界延长线的外部延伸到内部；上限组以 $p_k > 0$ 为特征，表示在该处直线段从裁剪边界延长线的内部延伸到外部。在有交点的情况下，下限组分布于直线段的起点一侧，上限组则分布于直线段终点一侧，则下限组的最大值和上限值的最小值就分别对应于直线段在窗口内部分的端点(假定存在）。（其中 $k = 1,2,3,4$， $p_1 = -(x_2 - x_1)$，$p_2 = x_2 - x_1$， $p_3 = -(y_2 - y_1)$， $p_4 = y_2 - y_1$，而 $(x_1, x_2)(y_1, y_2)$ 则为直线段的端点坐标）