一、目的

通过实验，深入理解并进一步学习计算机图形学基础算法，在课堂学习算法的思路后利用 `MFC` 实现这些算法。

本次两个实验均使用 `C/C++` 语言完成，利用 `MFC应用` ，应用程序类型为 `单个文档` ， `Microsoft Visual Studio Professional 2019` 版本号为 `16.7.5` 。

本次实验报告及源代码已上传至 `Github` ，

仓库地址为：https://github.com/FireSSang/20_21_1_Computer_Graphics

二、实验内容

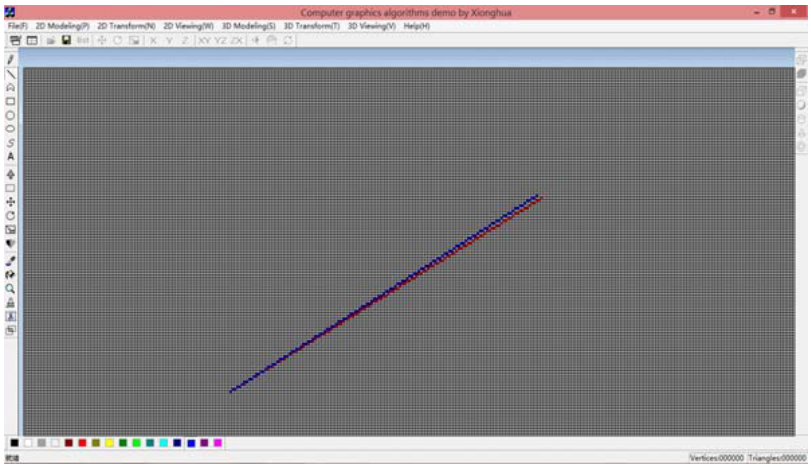
(1) 线画图元生成算法

内容：

1. 实现 `DDA` 直线生成算法
2. 实现 `Bresenham` 直线生成算法

要求：

1. 自定义直线段起点和终点坐标；
2. 采用不同的彩色显示两种算法生成的直线结果；
3. 为突出显示效果，采用网格表示像素，实现如下绘制效果。



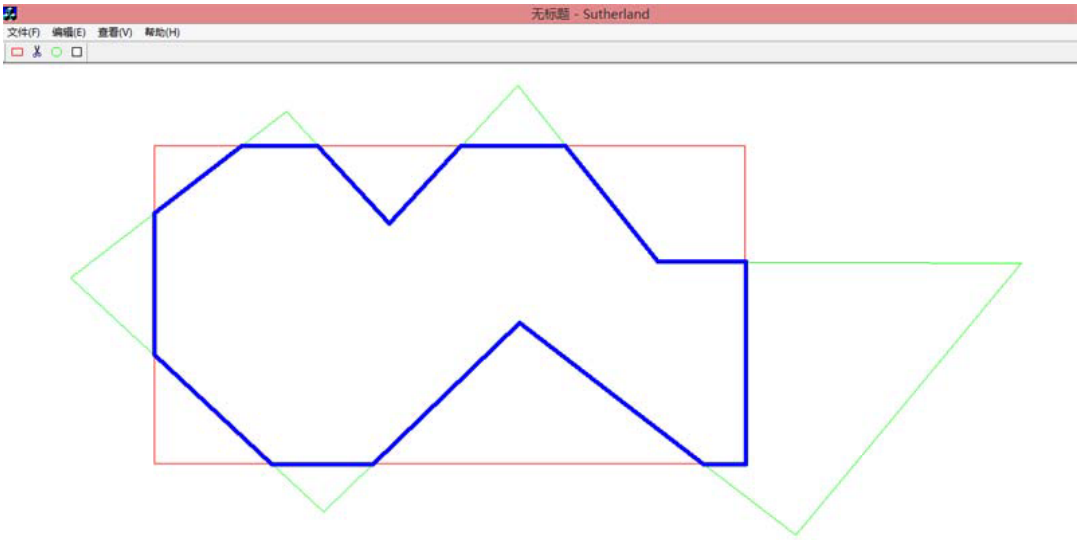
(2) 裁剪算法实验

内容：

1. 实现 `Cohen-Sutherland` 直线裁剪算法（选做）
2. 实现 `Sutherland-Hodgman` 多边形裁剪算法

要求：

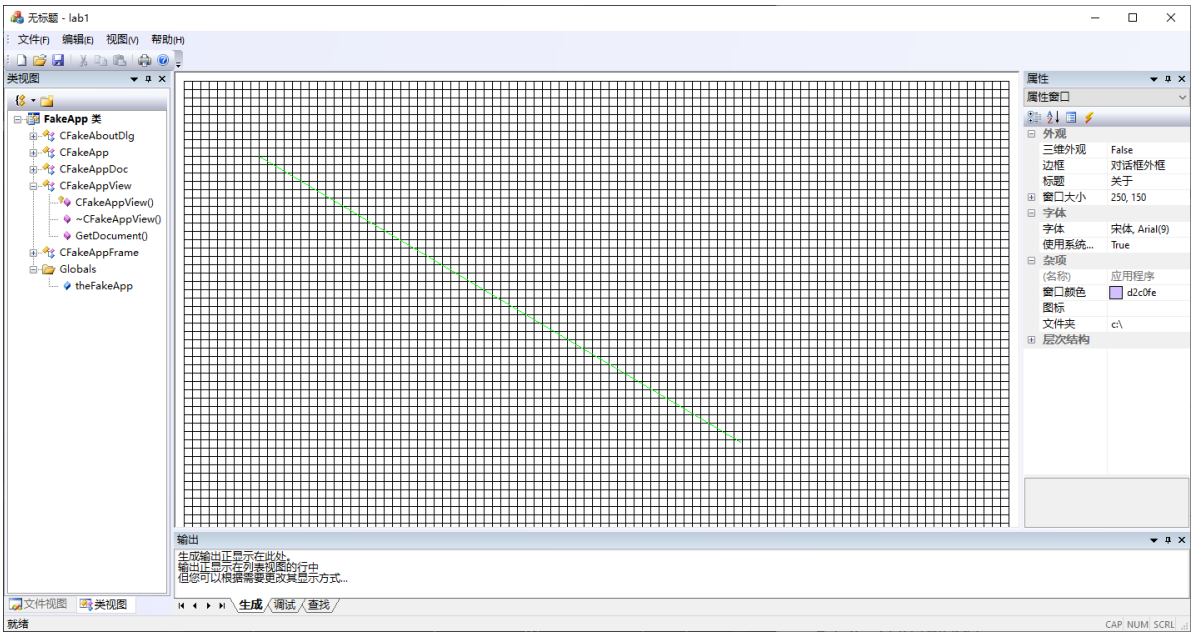
1. 自定义裁剪窗口和待裁剪直线段（或多边形）；
2. 采用不同颜色突出显示裁剪结果，如下图所示。



三、结果

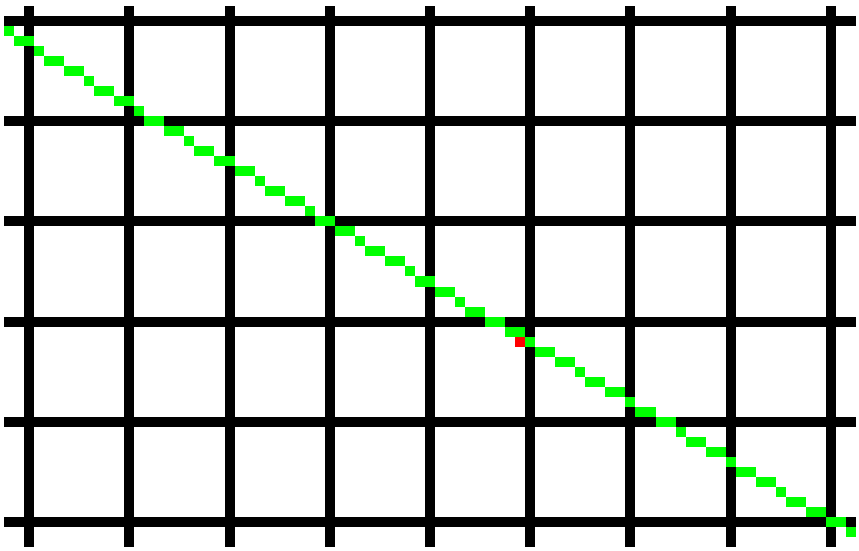
(1) 线画图元生成算法

实验结果如下：



其中，`DDA`算法所画直线使用红色（255,0,0），`Bresenham`算法所画直线使用绿色（0,255,0），先使用`DDA`算法，后使用`Bresenham`算法，故实验结果所画直线主要为绿色（0,255,0）。

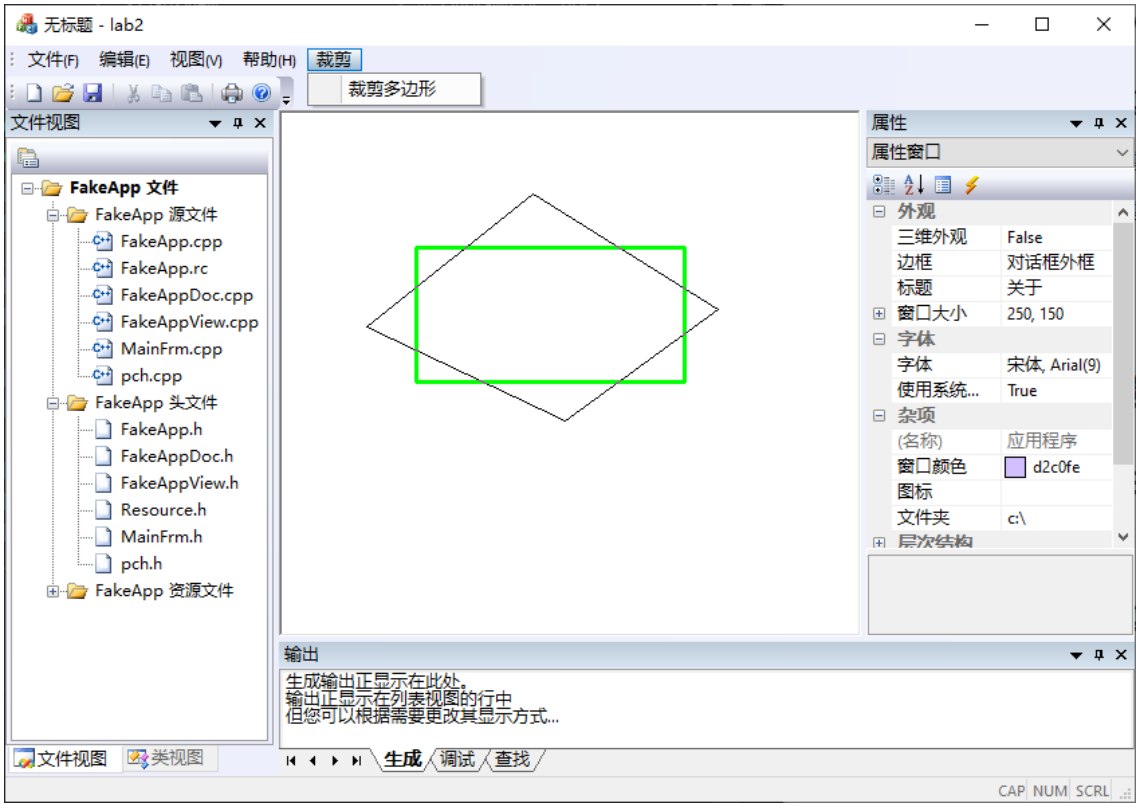
通过仔细观察，发现两算法仅在一点出现差异，以下为放大后的局部图片：



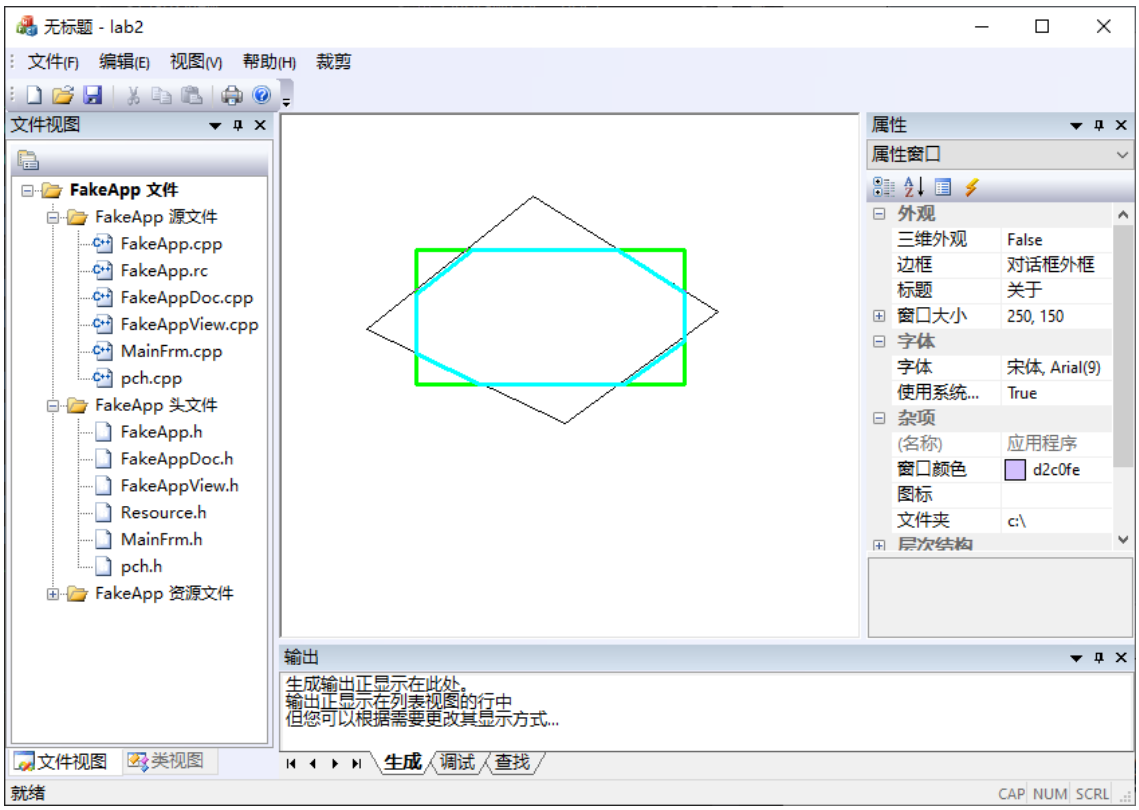
（2）裁剪算法实验

为简化操作，我固定了裁剪区域，并以绿色（0,255,0）表示，使用橡皮筋技术绘制待裁剪多边形，以黑色表示（0,0,0）。

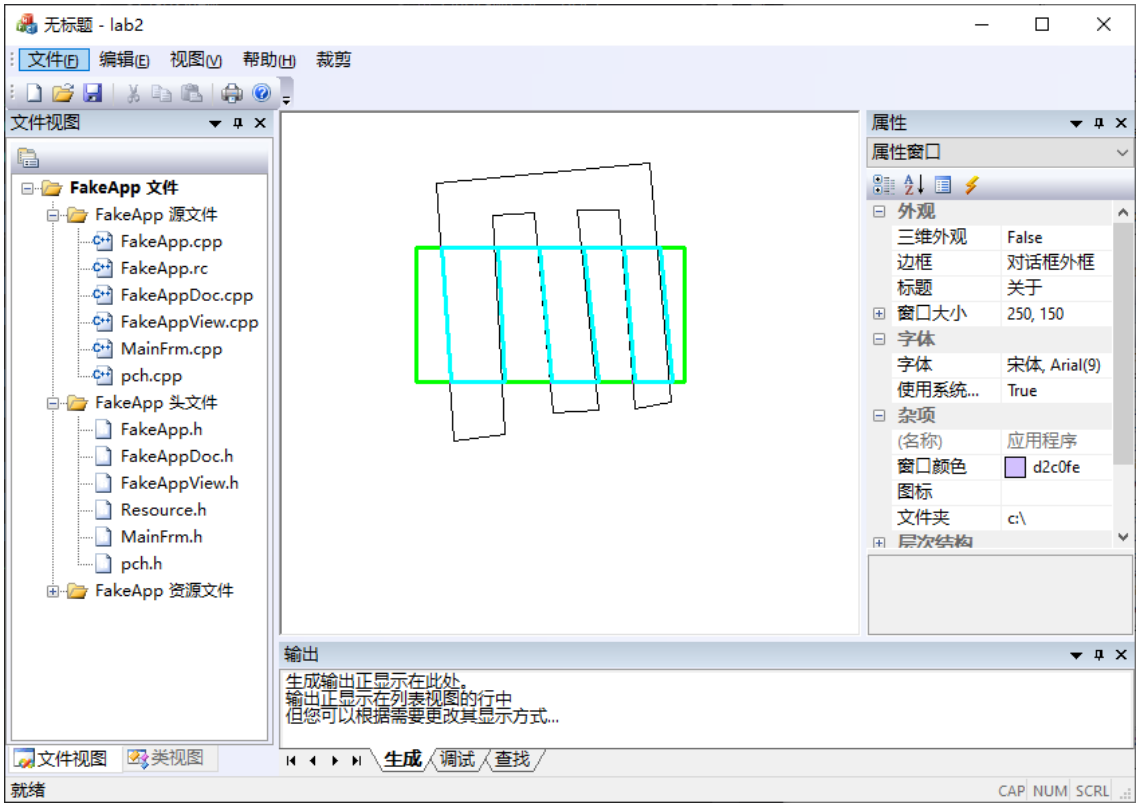
绘制待裁剪多边形后，裁剪前：

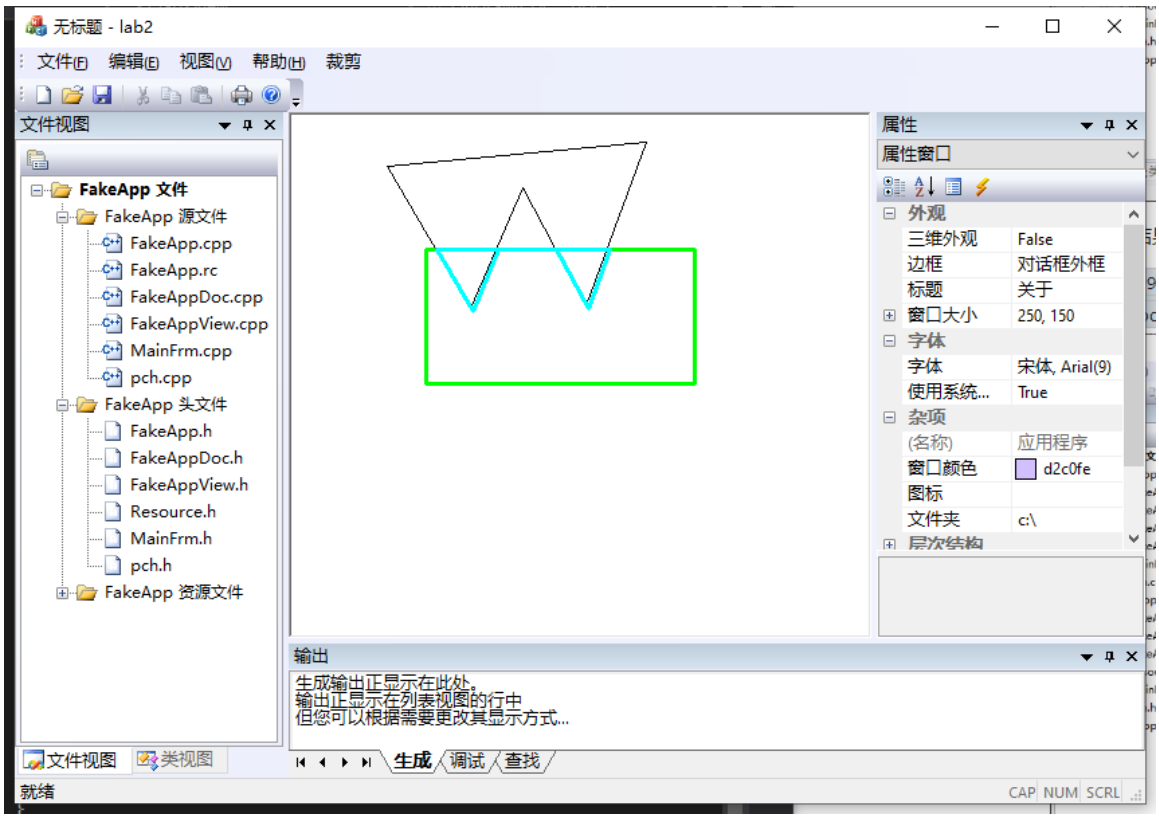


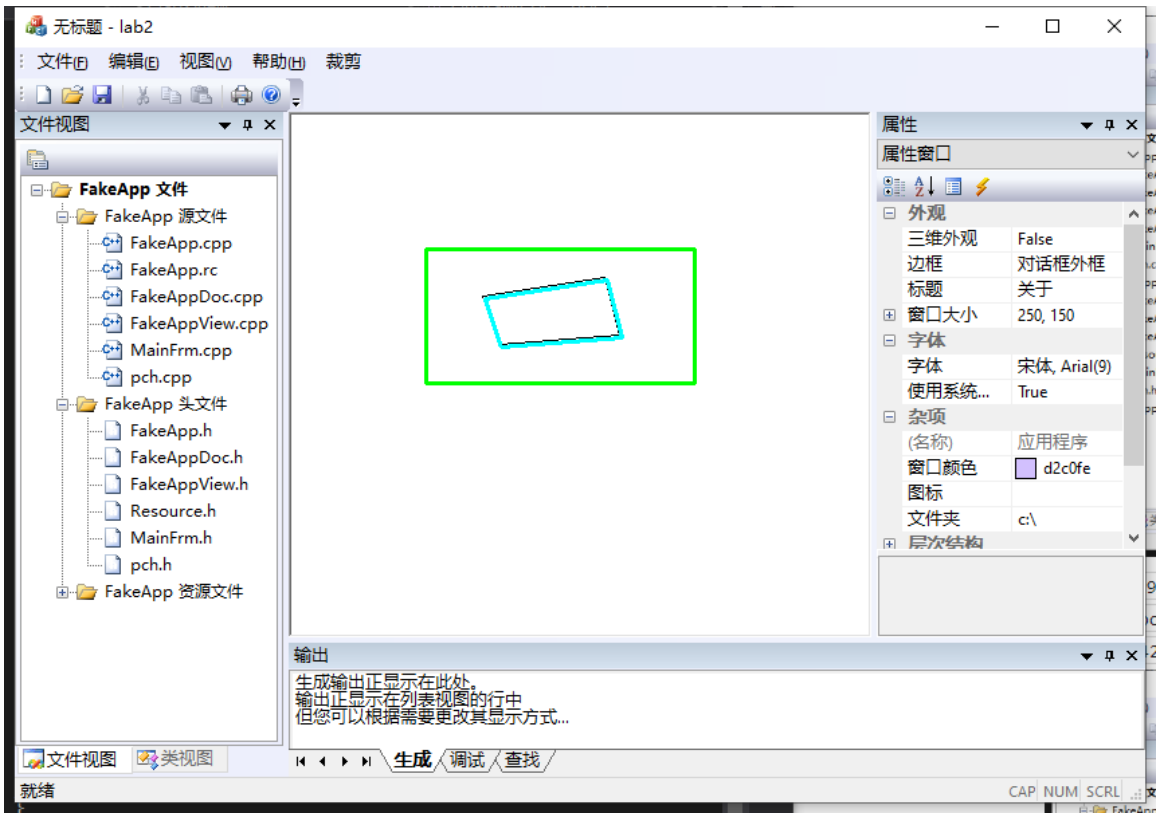
使用设定好的时间处理程序进行裁剪，裁剪后区域以青色绘制（0,255,255）：

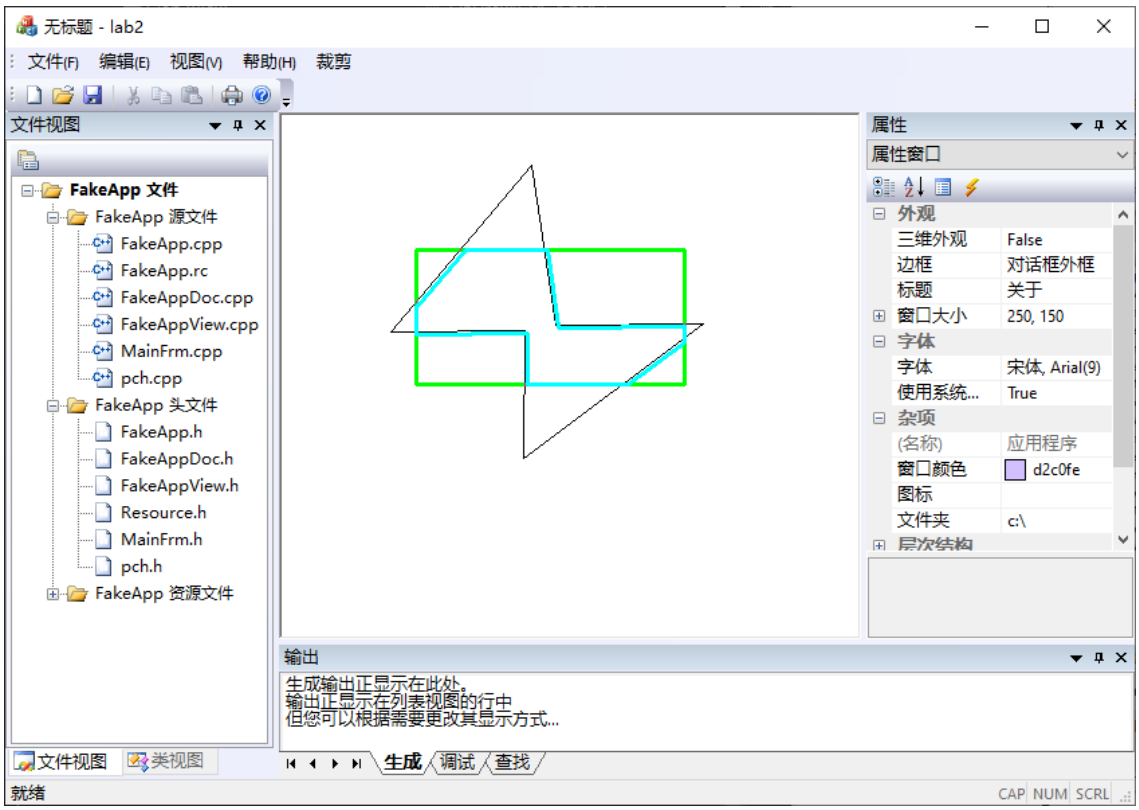


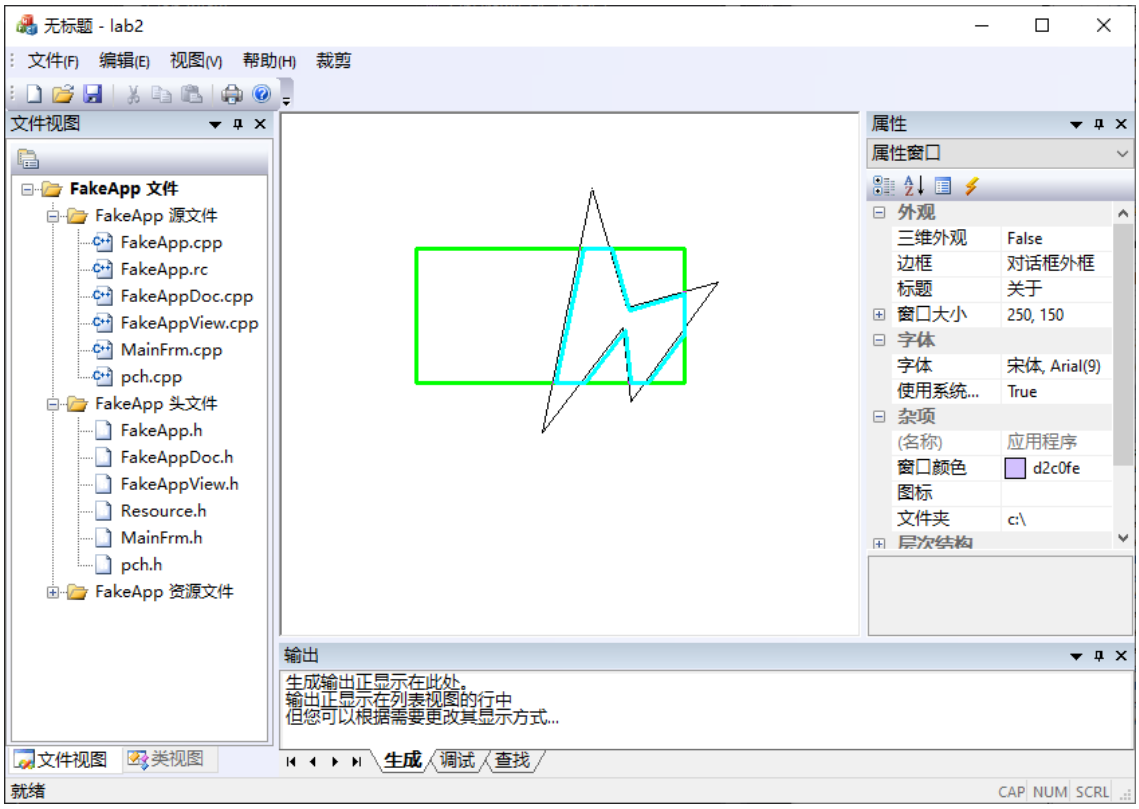
以下是其他多边形的裁剪结果展示：




无标题 - lab2
文件(F) 编辑(E) 视图(V) 帮助(H) 裁剪
文件视图
FakeApp 文件
FakeApp 源文件
FakeApp.cpp
FakeApp.rc
FakeAppDoc.cpp
FakeAppView.cpp
MainFrm.cpp
pch.cpp
FakeApp 头文件
FakeApp.h
FakeAppDoc.h
FakeAppView.h
Resource.h
MainFrm.h
pch.h
FakeApp 资源文件
属性
属性窗口
外观
三维外观 False
边框 对话框外框
标题 关于
窗口大小 250, 150
字体
字体 宋体, Arial(9)
使用系统... True
杂项
(名称) 应用程序
窗口颜色 d2c0fe
图标
文件夹 c:\
层次结构
输出
生成输出正显示在此处。
输出正显示在列表视图的行中
但您可以根据需要更改其显示方式...
文件视图 类视图
生成 调试 查找
就绪
CAP NUM SCRL


无标题 - lab2
文件(F) 编辑(E) 视图(V) 帮助(H) 裁剪
文件视图
FakeApp 文件
FakeApp 源文件
FakeApp.cpp
FakeApp.rc
FakeAppDoc.cpp
FakeAppView.cpp
MainFrm.cpp
pch.cpp
FakeApp 头文件
FakeApp.h
FakeAppDoc.h
FakeAppView.h
Resource.h
MainFrm.h
pch.h
FakeApp 资源文件
属性
属性窗口
外观
三维外观 False
边框 对话框外框
标题 关于
窗口大小 250, 150
字体
字体 宋体, Arial(9)
使用系统... True
杂项
(名称) 应用程序
窗口颜色 d2c0fe
图标
文件夹 c:\
层次结构
输出
生成输出正显示在此处。
输出正显示在列表视图的行中
但您可以根据需要更改其显示方式...
文件视图 类视图
生成 调试 查找
就绪
CAP NUM SCRL





四、分析和总结

• （1）线画图元生成算法

本实验主要难点在于两算法处理各种斜率的直线时需要分类讨论，总体实现难度不大。

通过本实验初步了解了`MFC应用`，能够实现简单的计算机图形学算法。

（2）裁剪算法实验

本实验我使用 `橡皮筋技术` 绘制待裁剪多边形，网络上关于 `MFC应用` 的教程过少并且描述太过简略，在 `消息处理程序` 和 `事件处理程序` 的添加、修改中遇到了很大的困难，耗费了很多时间和精力，好在通过摸索最终完成了本次实验。经过这次实验，我深入了解了 `MFC应用` 的实现原理，熟悉了 `MFC应用` 的使用方法，通过自己摸索留下的印象或许会更加深刻。

在 `Sutherland-Hodgman` 算法的编程中，由于编程水平不足，我的裁剪算法写的过于冗杂，好在结果正确，这暴露出我对该算法的理解不够深入、编程水平不足两个问题，在日后的学习研究工作中我需要继续提升编程水平，避免此类问题再次发生。

五、源代码

（1）线画图元生成算法

`lab1View.h` 中添加以下函数定义：

```
    void swap(int a, int b);
    void DDA_line(CDC* pDC, int x0, int y0, int x1, int y1, COLORREF color);
    void Bre_line(CDC* pDC, int x1, int y1, int x2, int y2, COLORREF color);
    void grid(CDC* pDC);
```

`lab1View.cpp` 中添加及修改以下函数：

```
// Clab1View swap函数
void Clab1View::swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}

// Clab1View DDA算法

void Clab1View::DDA_line(CDC* pDC, int x0, int y0, int x1, int y1, COLORREF color)
{
    double k;//斜率
    if (x1 != x0 && fabs(k = (y1 - y0) * 1.0 / (x1 - x0)) <= 1) //斜率<=1时
    {
        //交换
        if (x0 > x1)
        {
            swap(x0, x1);
            swap(y0, y1);
        }

        double y = y0;
        for (int x = x0; x <= x1; x++)
        {
            pDC->SetPixel(x, int(y + 0.5), color);//绘制像素点
            y += k;
        }
    }
    else //斜率>1时
    {
        k = (x1 - x0) * 1.0 / (y1 - y0);
```

```
        //交换
        if (y0 > y1)
        {
            swap(x0, x1);
            swap(y0, y1);
        }

        double x = x0;
        for (int y = y0; y <= y1; y++)
        {
            pDC->SetPixel(int(x + 0.5), y, color);
            x += k;
        }
    }
}

// Clab1View Bresenham算法

void Clab1View::Bre_line(CDC* pDC, int x1, int y1, int x2, int y2, COLORREF color)
{
    int dx, dy; // 横轴纵轴差量
    int e;
    int x, y;
    dx = abs(x2 - x1);
    dy = abs(y2 - y1);
    y = y1;
    x = x1;
    int cx, cy; // 表明x、y方向的增量
    if (x1 > x2)
        cx = -1; // x递减方向
    else
        cx = 1; // x递增方向
    if (y1 > y2)
        cy = -1; // y递减方向
    else
        cy = 1; // y递增方向
    if (dx == 0 && dy == 0)
    {
        printf("The input is not a line. It is just a point. Please input two
different points.!\n");
    }
    else if (dy == 0)
    {
        for (x = x1; (cx == 1 ? x <= x2 : x >= x2); x += cx)
        {
            pDC->SetPixel(x, y, color);

        }
    }
    else if (dx == 0)
    {
        for (y = y1; (cy == 1 ? y <= y2 : y >= y2); y += cy)
        {
            pDC->SetPixel(x, y, color);
        }
    }
    else if (dx >= dy) {
        e = -dx;
        for (x = x1; (cx == 1 ? x <= x2 : x >= x2); x += cx)
        {
            pDC->SetPixel(x, y, color);
```

```
            e += dy << 1;
            if (e > 0)
            {
                y += cy;
                e -= dx << 1;
            }
        }
    }
    else {
        e = -dy;
        for (y = y1; (cy >= 0 ? y <= y2 : y >= y2); y += cy)
        {
            pDC->SetPixel(x, y, color);
            e += dx << 1;
            if (e > 0)
            {
                x += cx;
                e -= dy << 1;
            }
        }
    }
}

// Clab1grid 画个网格
void Clab1View::grid(CDC* pDC)
{
    for (int i = 10; i <= 1000; i += 10)
    {
        for (int j = 10; j <= 1000; j++)
        {
            pDC->SetPixel(i, j, RGB(0, 0, 0));
            pDC->SetPixel(j, i, RGB(0, 0, 0));
        }
    }
}


// Clab1View 绘图

void Clab1View::OnDraw(CDC* pDC)
{
    Clab1Doc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
        return;

    // TODO: 在此处为本机数据添加绘制代码
    grid(pDC);
    DDA_line(pDC, 100, 100, 678, 443, RGB(255, 0, 0));
    Bre_line(pDC, 100, 100, 678, 443, RGB(0, 255, 0));
}
```

（2）裁剪算法实验

选择 `类视图` ，右键单击 `Clab2View` ，选择 `类向导` ，选择 `消息` ，添加以下两个消息处理程序（自动在 `Clab2View.h` 中添加定义，在 `Clab2View.cpp` 中创建函数定义），利用橡皮筋技术实现绘制待裁剪多边形。

`OnLButtonDown` ：

```
void Clab2View::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: 在此添加消息处理程序代码和/或调用默认值
    if (if_Press == 1 || Number == 0)
    {
        if_Press = 1; //说明已经点过了一个点
        start_point = point;
        end_point = start_point;
        Point[Number] = start_point;//把这个点存入数组里
        if (Number == 0) //表示第一次左击的点为第一个点，
            first = start_point;//把第一个点保存下来
        Number++;//多边形的点数加一
    }
    CView::OnLButtonDown(nFlags, point);
}
```

`OnMouseMove`：

```
void Clab2View::OnMouseMove(UINT nFlags, CPoint point)
{
    // TODO: 在此添加消息处理程序代码和/或调用默认值
    CDC* pDC = this->GetWindowDC();
    if (if_Press)
    {
        pDC->SetROP2(R2_NOT);  //添加橡皮绳（只要没左击，鼠标在桌面上移动，线条就会跟到哪去）
        pDC->MoveTo(start_point); //将点坐标移到上一个点击的点的位置
        pDC->LineTo(end_point);  //没有画出线，此时的start_point = end_point
        if ((point.x - first.x) < 0.1 && (point.y - first.y) < 0.1)//当鼠标指着的点离第一个点的距离小于一个值的时候，两点重合（这里两个点会自动重合，不用再次点击左键）
        {
            point = first;
            if_Press = 0;
        }
        pDC->MoveTo(start_point); //画出直线
        pDC->LineTo(point);
        end_point = point;
    }
    CView::OnMouseMove(nFlags, point);
}
```

在`Clab2View.h`中添加如下定义，包括待裁剪多边形的存储、裁剪结果的存储以及裁剪区域的边界值：

```
    CPoint start_point;//起始点
    CPoint end_point;//终止点
    CPoint first;//第一个点
    CPoint Point[100];
    int if_Press = 0;
    int Number = 0; //多边形的点数

    CPoint New_Point[100];

    int XL = 100, XR = 300, YB = 100, YT = 200;//分别是左边的x值，右边的x值，下边的y值，上边的y值
```

在`Clab2View.cpp`中修改`OnDraw`函数，设定裁剪区域，并以绿色（0,255,0）着色：

```
void Clab2View::OnDraw(CDC* pDC)
{
    Clab2Doc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
        return;

    // TODO: 在此处为本机数据添加绘制代码
    CPen newpen(PS_SOLID, 3, RGB(0, 255, 0));
    CPen* old = pDC->SelectObject(&newpen);
    pDC->MoveTo(100, 100);
    pDC->LineTo(100, 200);

    pDC->MoveTo(100, 200);
    pDC->LineTo(300, 200);

    pDC->MoveTo(300, 200);
    pDC->LineTo(300, 100);

    pDC->MoveTo(300, 100);
    pDC->LineTo(100, 100);
    pDC->SelectObject(old);
    //ReleaseDC(pDC);
}
```

选择 资源视图 ，展开 lab2 ，展开 lab2.rc ，展开 Menu ，双击 IDR_MAINFRAME ，在 帮助 后添加一个 裁剪 控件，下拉菜单添加一个选项并命名为 裁剪多边形 ，右键 裁剪多边形 控件，选择 添加事件处理程序 ，类列表中选择 Clab2View ，消息类型选择 COMMAND ，函数名命名为 OnSutherHodgman ，在 Clab2View.cpp 中完成该函数的定义：

```
void Clab2View::OnSutherHodgman()
{
    // TODO: 在此添加命令处理程序代码
    {
        // TODO: 在此添加命令处理程序代码
        int temp_num; //这个临时的变量存储的就是多边形的点个数，也就是前面画多边形时的Number
        int new_num = 0; //这个变量是用来存储通过算法裁剪后的点的个数
        CPoint temp_point; //定义一个临时点，在后面是用来求交点用的
        int i;
        int j;
        temp_num = Number;

        //先说明下，画线的时候是从起始点到终止点，在后面的解释中我会把起始点说成头点，终止点说成尾点，也就是从头到尾，还有一点，如果这个点在可视区外，在后面就直接简称为外了
          //////////////
          //对矩形框上边处理
        for (i = 0; i < temp_num; i++)//temp_num 为多边形的边数，执行次数也就是边数
        {
            if (Point[i % temp_num].y > YT) //如果头点在上边之上，即头点在不可视区
            {    //这样要注意下Point数组选取元素是加的%号的含义，因为头号是i，尾号是i+1，可是到了最后一个点的时候，头号还是i，可是尾却不是i+1了，此时的头要和最初的第一个点相连，才能形成一个闭合的多边形，所以整除（点的个数），刚好能让最后一个的值变成0
            //本来%只是最后一个点才用得着，这里为了不乱掉，索性全加上，反正对前面的点也没什么影响。。。
                if (Point[(i + 1) % temp_num].y > YT) //尾点也在上边之上，即两点都在不可视区，均不进入新数组，这种情况就是（外到外），所以什么都不用写.
                {

```

```
                }
                else  //尾点在上边之下，即尾点在可视区，尾点和与上边的交点进入新数组（外到内）
                {

                    temp_point.x = Point[i % temp_num].x - (Point[i % temp_num].y - YT) * (Point[i % temp_num].x - Point[(i + 1) % temp_num].x)
                        / (Point[i % temp_num].y - Point[(i + 1) % temp_num].y); //这一大串求交点的x坐标值是通过相似的原理来表示的，想深究的朋友可以画个实例图研究，接下来的很多求交点都是通过这个原理
                    temp_point.y = YT; //求交点的y坐标，因为交点在上边，所以上边的y值就是交点的y值
                    New_Point[new_num] = temp_point; //把这个临时的交点存到New_point数组里，同时点的个数加一，接下来同样每送到新数组一个点，new_num（新多边形的点个数)就加一
                    new_num += 1;
                    /*New_Point[new_num] = Point[(i + 1) % temp_num]; //这里要注意下，按理来说，画一条从外到内的线应该要把与边的交点和内点都要进入新数组里，但是内点却不在这段代码中加到新数组里面，而是到下一个循环，这个内点从尾点变成头点的时候加到数组里
                    new_num += 1;*/                                   //如果在这里加到数组里会导致这样一种情况，这个点作为尾点加到数组，到下一个循环，这个点变成了头点，难道又要再加一遍？这会导致同一个点在数组内存在了两次
                    //凡是都有例外，其中有一种情况是要在一段代码中添加两个点的，接下来会解释到

                }
            }
            else  //头点在上边之下，即在可视区
            {
                if (Point[(i + 1) % temp_num].y <= YT) //尾点也在可视区，头尾两点均入新数组（内到内）
                {
                    New_Point[new_num] = Point[i % temp_num];
                    new_num += 1;
                    /*New_Point[new_num] = Point[(i + 1) % temp_num];
                    new_num += 1;*/
                }
                else //尾点在不可视区，所以头点和交点进入新数组（内到外），这种情况就是上面说到的要添加两个点，大家可以拿笔画个草图看看，内到外，肯定先把内点加到新数组里，如果说把那个交点留着下一个循环来处理会导致什么样的情况，下一次肯定是从外到内
                {    //或者从外到外，不管哪种情况，能确定的是和这个交点已经没什么关系了，画图很容易看出来，所以如果不在这种情况下把交点加到新数组里，就会导致这个交点漏掉，即使在下面的循环也没法把这个交点加到数组了
                    New_Point[new_num] = Point[i % temp_num];
                    new_num += 1;
                    //计算交点坐标
                    temp_point.x = Point[i % temp_num].x + (YT - Point[i % temp_num].y) * (Point[(i + 1) % temp_num].x - Point[i % temp_num].x)
                        / (Point[(i + 1) % temp_num].y - Point[i % temp_num].y);
                    temp_point.y = YT;
                    New_Point[new_num] = temp_point;
                    new_num += 1;

                }
            }
        }
        ////////////////////////

        //重新整理处理上边后的图形


        for (j = 0; j < new_num; j++)
```

```
68          {
69              Point[j] = New_Point[j]; //将新数组的点调回Point，此时的新数组已经保留了裁剪上边后的新的点，这时，把这个经过上边处理后的多边形调回Point,可以这么说，把这个新的多边形重新作为待处理的多边形，进行下边处理
70          }
71          Number = new_num; //此时的多边形点数为上边裁剪后的新多边形的点数
72          new_num = 0; //归0，进行下边处理
73          temp_num = Number;
74          //对矩形框下边处理，接下来的对下边，左边，右边和上边处理类似，只是在一些细节方面做了一些改变，例如在求交点的相似处理上，有所改变，而且上下两边和左右两边的处理也稍有不同，不过大同小异，把上边处理弄懂后，下面的都差不多
75          for (i = 0; i < temp_num; i++)//temp_num 为多边形的边数，执行次数也就是边数
76          {
77              if (Point[i % temp_num].y > YB) // 头点在下边上，即头点在可视区内
78              {
79                  if (Point[(i + 1) % temp_num].y > YB) // 尾点也在可视区内，两点均在可视区内（内到内）
80                  {
81                      New_Point[new_num] = Point[i % temp_num];
82                      new_num += 1;
83                      /*New_Point[new_num] = Point[(i + 1) % temp_num];
84                      new_num += 1;*/
85                  }
86                  else  //尾点在可视区外 ，头点和交点进入新数组（内到外）
87                  {
88                      New_Point[new_num] = Point[i % temp_num];
89                      new_num += 1;
90                      //求交点
91                      temp_point.x = Point[i % temp_num].x - (Point[i % temp_num].y - YB) * (Point[i % temp_num].x - Point[(i + 1) % temp_num].x)
92                          / (Point[i % temp_num].y - Point[(i + 1) % temp_num].y);
93                      temp_point.y = YB;
94                      New_Point[new_num] = temp_point;
95                      new_num += 1;
96
97                  }
98              }
99              else //头在可视区外
100             {
101                 if (Point[(i + 1) % temp_num].y >= YB) //尾在可视区内（外到内）
102                 {
103                     //计算交点坐标
104                     temp_point.x = Point[i % temp_num].x + (YB - Point[i % temp_num].y) * (Point[(i + 1) % temp_num].x - Point[i % temp_num].x)
105                         / (Point[(i + 1) % temp_num].y - Point[i % temp_num].y);
106                     temp_point.y = YB;
107                     New_Point[new_num] = temp_point;
108                     new_num += 1;
109                     /*New_Point[new_num] = Point[(i + 1) % temp_num];
110                     new_num += 1;*/
111                 }
112                 else //头尾两点均在可视区外（外到外）
113                 {
114
115
116                 }
117             }
118         }
119         //重新整理处理下边后的图形
120         for (j = 0; j < new_num; j++)
121         {
```

```
            Point[j] = New_Point[j]; //将新数组的点调回Point
        }
        Number = new_num; //此时的点数为改变上边后的图形的点数
        new_num = 0; //归0，进行下边处理
        temp_num = Number;
        //对矩形框左边处理
        for (i = 0; i < temp_num; i++)//temp_num 为多边形的边数，执行次数也就是边数
        {
            if (Point[i % temp_num].x >= XL) // 头点在可视区内
            {
                if (Point[(i + 1) % temp_num].x >= XL) // 尾点也在可视区内，两点均在可视区内(内到内）
                {
                    New_Point[new_num] = Point[i % temp_num];
                    new_num += 1;
                    /*New_Point[new_num] = Point[(i + 1) % temp_num];
                    new_num += 1;*/
                }
                else  //尾点在可视区外 ，头点和交点进入新数组（内到外）
                {
                    New_Point[new_num] = Point[i % temp_num];
                    new_num += 1;
                    //求交点
                    temp_point.y = Point[i % temp_num].y - (Point[i % temp_num].x - XL) * (Point[i % temp_num].y - Point[(i + 1) % temp_num].y)
                        / (Point[i % temp_num].x - Point[(i + 1) % temp_num].x);
                    temp_point.x = XL;
                    New_Point[new_num] = temp_point;
                    new_num += 1;

                }
            }
            else //头在可视区外
            {
                if (Point[(i + 1) % temp_num].x >= XL) //尾在可视区内（外到内）
                {
                    //计算交点坐标
                    temp_point.y = Point[i % temp_num].y + (XL - Point[i % temp_num].x) * (Point[(i + 1) % temp_num].y - Point[i % temp_num].y)
                        / (Point[(i + 1) % temp_num].x - Point[i % temp_num].x);
                    temp_point.x = XL;
                    New_Point[new_num] = temp_point;
                    new_num += 1;
                    /*New_Point[new_num] = Point[(i + 1) % temp_num];
                    new_num += 1;*/
                }
                else //头尾两点均在可视区外（外到外）
                {


                }
            }
        }
        //重新整理处理左边后的图形
        for (j = 0; j < new_num; j++)
        {
            Point[j] = New_Point[j]; //将新数组的点调回Point
        }
        Number = new_num; //此时的点数为改变上边后的图形的点数
        new_num = 0; //归0，进行下边处理
        temp_num = Number;
```

```
        //对矩形框右边处理
        for (i = 0; i < temp_num; i++)//temp_num 为多边形的边数，执行次数也就是边数
        {
            if (Point[i % temp_num].x <= XR) // 头点在可视区内
            {
                if (Point[(i + 1) % temp_num].x <= XR) // 尾点也在可视区内，两点均在可视区内(内到内）
                {
                    New_Point[new_num] = Point[i % temp_num];
                    new_num += 1;
                    /*New_Point[new_num] = Point[(i + 1) % temp_num];
                    new_num += 1;*/
                }
                else  //尾点在可视区外 ，头点和交点进入新数组（内到外）
                {
                    New_Point[new_num] = Point[i % temp_num];
                    new_num += 1;
                    //求交点
                    temp_point.y = Point[i % temp_num].y + (XR - Point[i % temp_num].x) * (Point[(i + 1) % temp_num].y - Point[i % temp_num].y)
                        / (Point[(i + 1) % temp_num].x - Point[i % temp_num].x);
                    temp_point.x = XR;
                    New_Point[new_num] = temp_point;
                    new_num += 1;

                }
            }
            else //头在可视区外
            {
                if (Point[(i + 1) % temp_num].x <= XR) //尾在可视区内（外到内）
                {
                    //计算交点坐标
                    temp_point.y = Point[i % temp_num].y - (Point[i % temp_num].x - XR) * (Point[i % temp_num].y - Point[(i + 1) % temp_num].y)
                        / (Point[i % temp_num].x - Point[(i + 1) % temp_num].x);
                    temp_point.x = XR;
                    New_Point[new_num] = temp_point;
                    new_num += 1;
                    /*New_Point[new_num] = Point[(i + 1) % temp_num];
                    new_num += 1;*/
                }
                else //头尾两点均在可视区外（外到外）
                {


                }
            }
        }
        //这里是最后一步，经过上下左右四条边的处理，此时的多边形已经裁剪完成，并且储存在New_Point在，下面的工作就是用循环一条边一条边得画出来
        CDC* pDC = this->GetDC();
        CPen newpen(PS_SOLID, 3, RGB(0, 255, 255));
        CPen* old = pDC->SelectObject(&newpen);
        for (j = 0; j < new_num; j++)
        {
            pDC->MoveTo(New_Point[j % new_num].x, New_Point[j % new_num].y);
            pDC->LineTo(New_Point[(j + 1) % new_num].x, New_Point[(j + 1) % new_num].y);
        }
        pDC->MoveTo(New_Point[0].x, New_Point[0].y);
        pDC->LineTo(New_Point[1].x, New_Point[1].y);
```

```
236         pDC->SelectObject(old);
237         ReleaseDC(pDC);
238 
239     }
240 }
```