实验内容：

选用任一种光栅图形学算法实现多边形区域的填充。

如：逐点判断法、扫描线算法、边缘填充算法、种子填充算法（除了边界填充算法）。

实验目的：

1. 通过实现光栅图形学经典的多边形填充算法，深入理解光栅图形学的原理。
2. 锻炼实践算法的能力。

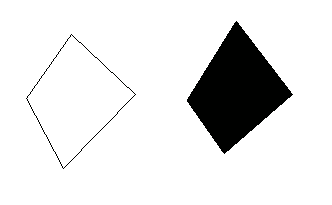
实现方法：

1. **实现方法概述和框架变动的说明：**

本次实验采用了扫描线算法完成多边形的填充。

在原有框架中，通过点击工具栏上的“PolygonFill”图标，在多边形内用鼠标定义一个点即可应用种子填充算法（即边界填充），以实现多边形的填充。由于**扫描线算法依赖多边形的顶点而非内部点**，因此对整个框架进行了以下改动：

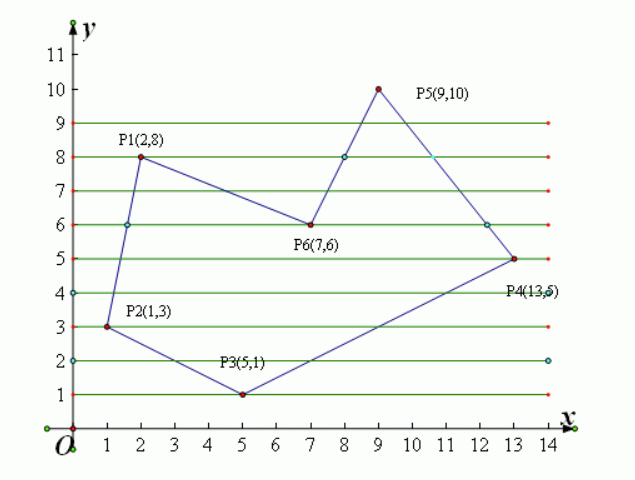
1. 点击原框架中工具栏的“PolygonFill”图标，可以实现与点击工具栏上“Polygon”图标相同的功能。不同之处在于“PolygonFill”所绘制的是实心多边形，即多边形绘制附加上多边形填充。下图中左图为“Polygon”所绘多边形，右图为“PolygonFill”所绘多边形。



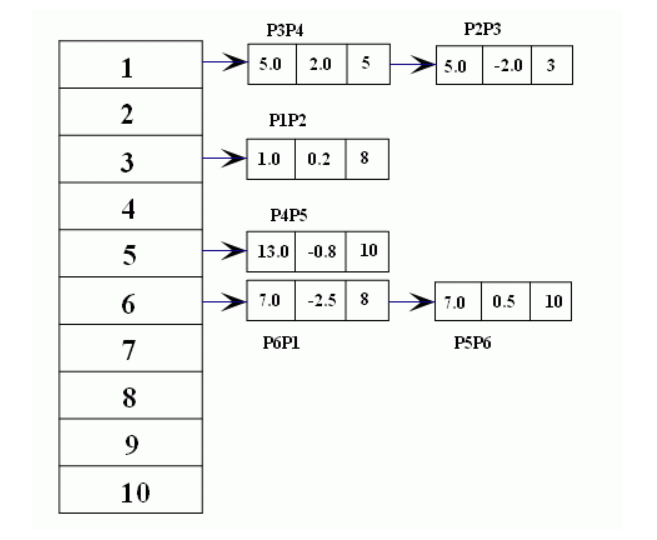
1. 在CCGPainterView类中，将MyPolygon类的成员变量从ThePolygon改为MyPolygon类的ThePolygons数组成员变量。这一调整可完成多个多边形的绘制，并将最大绘制数量定义为100。
2. 本次实验中实现的扫描线算法依赖边表和活化边表（详细的数据结构将在后文阐述）。在MyPolygon类中，新增了成员变量edgeTable和activeEdgeTable，用于分别存储ET边表和AET活化边表。同时，添加了相应的成员方法BuildEdgeTable()和SetAET(int y)来设置边表和活化边表。
3. **算法的介绍及所采用的数据结构说明：**

扫描线算法是一种用于多边形填充的方法。它基于扫描线从多边形的最低点到最高点的扫描，通过边表和活化边表的结构，实现多边形填充。下图为扫描线

扫描到每个y值的示意图。



通过点击工具栏上的“PolygonFill”图标并绘制多边形，用户能够定义多边形的顶点，扫描线算法会根据多边形的顶点建立边表。边表的数据结构如下所示：

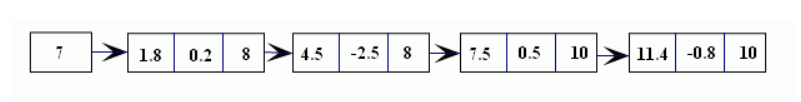


每一条边用一个结构体Edge类型的对象表示。其中x表示下端点的横坐标，Δx表示斜率的倒数（即每次扫描线移动时x顺着线段移动的距离），ymax为上端点的纵坐标。



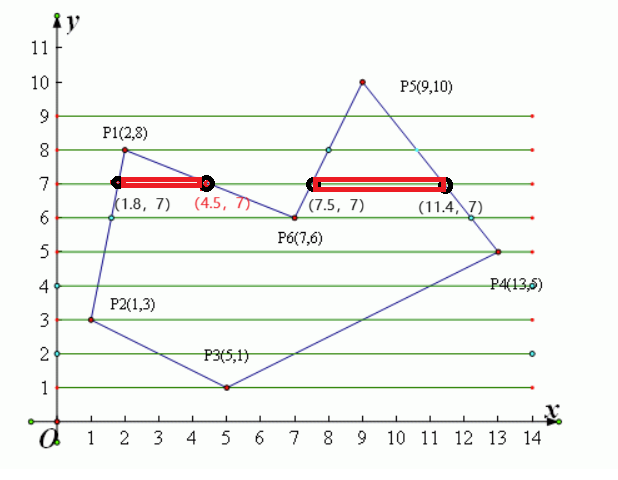
所有边按下端点的y坐标归类，每个类别建立一个链表，上图中的next为下一条边的指针，所有链表储在以y坐标为数组下标的链表数组中。在代码中，该表即为MyPolygon中的成员变量edgeTable[MAX\_Y\_NUM]，其中MAX\_Y\_NUM代表定义的最大的y值（这里采用1024）。

有了边表后，在遍历y值时，我们可以建立活化边表来确定有哪些边和当前y值所对应的扫描线相交，并且获得相交的x坐标值。代码中的活化边表AET即为MyPolygon中的成员变量activeEdgeTable。如下所示为y=7时的AET。



在建立和更新AET时，每次迭代时，将扫描线对应的y值移动一格，用x+Δx替代x来更新交点的x值，将ymax等于当前y值的边删除，因为该边已经不和扫描线相交了，从边表中寻找新的相交边添加进AET中，确定完所有相交边后，按照x从小到大的顺序对链表进行排序（如果x相等那么按照Δx从小到大排序）。

排序完成后，我们发现链表中每两个节点的x之间的部分都是需要填充的区域。我们可以根据这个特性确定每个y值上的需要填充的区域，从而完成整个多边形的填充。



本次实验中，由于应用MFC定义的坐标系，将“从多边形的最低点到最高点”改为“从多边形的最高点到最低点”，以顺应按照y轴增大的方向移动扫描线。对应的地方都应进行更改，如Edge中的x也更改为上端点的横坐标、ymax为下端点的纵坐标等。

1. **代码中各函数的功能：**

以下介绍实现扫描线算法的三个主要函数。提供部分核心代码片段，完整代码和注释见附件项目。主要实现代码在CGPainterView.h，CGPainterView.cpp，GlobalVariables.h和GlobalVariables.cpp中。

* 1. void MyPolygon::BuildEdgeTable()

用于构建边表（ET），步骤如下：

1. 初始化边表，将其各项设为空指针。
2. 逐个遍历顶点，相邻顶点确定一条边，注意下一个相邻顶点的索引用(i + 1) % m\_VerticeNumber表示，以防从尾接到头的时候发生访问越界。将非水平边添加到边表中。水平边和扫描线平行无交点，在扫描线算法中不发挥作用。对于每条非水平边，根据其上端点和下端点，计算斜率倒数（dx），并将边信息（x，dx，ymax）存储到边表中相应的位置。设置next为null空指针。

newEdge->x = upperVertex.x;

float t1 = lowerVertex.x - upperVertex.x;

float t2 = lowerVertex.y - upperVertex.y;

float t3 = t1 / t2;

newEdge->dx = t3;

newEdge->ymax = lowerVertex.y;

newEdge->next = nullptr;

1. 对于上顶点y值相同的边，存储在同一链表中。向边表数组中添加边的时候，如果该y值下表对应的链表为空，则新建链表，若不为空，则将链表尾的next指针指向该边。

if (edgeTable[y] != nullptr) {

Edge\* current = edgeTable[y];

while (current->next != nullptr) {

current = current->next;

}

current->next = newEdge;

}

else {

edgeTable[y] = newEdge;

}

* 1. void MyPolygon::SetAET(int y)

用于更新活化边表（AET），步骤如下：

* + - 1. 遍历活化边表，检查边界情况，删除达到 ymax 的边。

if (current->ymax == y) {

//ymax = y 的边进行删除

if (previous != nullptr) {

//如果删除的不是第一个节点

previous->next = current->next;

current = previous->next;

}

else {

//如果删除的是第一个节点

current = current->next;

activeEdgeTable = current;

}

}

* + - 1. 更新边表中节点的x值。

current->x += current->dx;

* + - 1. 将新的边表连接到活化边表的末尾。

if (activeEdgeTable != nullptr) {

......(遍历迭代)

// 将边表接在活化边表的末尾

if (previous != nullptr) previous->next = edgeTable[y];

else activeEdgeTable = edgeTable[y];

}

else {

//若活化边表为空，直接将边表赋值给活化边表

activeEdgeTable = edgeTable[y];

}

* + - 1. 对活化边表按照 x 坐标和 dx 进行排序，确保其按照 x 坐标的递增顺序排列。

if (sortedList == nullptr || edge->x < sortedList->x || (edge->x == sortedList->x && edge->dx < sortedList->dx))

{

//如果待添加边的x比已排序链表中的第一个边的x小，

//或者已排序链表为空，直接添加到头部

edge->next = sortedList;

sortedList = edge;

}

else {

Edge\* currentSorted = sortedList; // 遍历链表的指针

// 确定插入位置

while (currentSorted->next != nullptr && (edge->x > currentSorted->next->x || (edge->x == currentSorted->next->x && edge->dx > currentSorted->next->dx)))

{

currentSorted = currentSorted->next;

}

edge->next = currentSorted->next;

currentSorted->next = edge;

}

* 1. CCGPainterView::

ScanlineFill(CDC\*pDC, MyPolygon polygon, COLORREF fillCol)

实现了扫描线填充算法。通过活化边表和扫描线的方式，逐行扫描多边形的区域并进行填充。步骤如下：

* + - 1. 通过 GetMinY() 和 GetMaxY() 方法（新添加的Mypolygon类的成员方法）获取多边形的最小和最大 Y 坐标。
      2. 从最小 Y 坐标扫描到最大 Y 坐标的范围内，依次对每条扫描线进行处理。
      3. 在每次扫描线扫描过程中：使用 SetAET(y) 构建当前扫描线的活化边表（AET）。遍历 AET 中的边，每两条相邻的边确定了一个需要填充的区域，在这个区间内设置像素点的颜色。在遍历 AET 的过程中，每次迭代跳过了两条边，这样保证了相邻边的处理。

Edge\* currentEdge = polygon.activeEdgeTable;

while (currentEdge != nullptr && currentEdge->next != nullptr) {

// 遍历 AET 中的边，填充每一对边之间的像素点

float xStart = currentEdge->x;

float xEnd = currentEdge->next->x;

// 从左端点开始到右端点，给每个像素点设置颜色

for (int x = xStart; x <= xEnd; ++x) {

// 使用 SetPixelV 设置像素点颜色

pDC->SetPixelV(x, y, fillCol);

}

currentEdge = currentEdge->next->next; // 跳过两条边

}

该函数在OnRButtonUp（鼠标右键抬起）函数中调用，该函数中实现了多边形最后一条边的绘制，随后判断当前绘制状态是否为DRAW\_POLYGON\_FILL（按下了“PolygonFill”图标按键），如果是绘制实心多边形就将当前多边形传入ScanlineFill函数进行填充。

注意：这里需要重复两次，一次是绘制到屏幕上，第二次是绘制到内存中存储，在两次绘制中间需要重置边表和活化边表以免前一次绘制时表中发生污染影响后一次绘制。

//如果选择的是填充多边形状态，就填充所绘制的多边形

if (G\_iDrawState == DRAW\_POLYGON\_FILL)

{

ThePolygons[numPolygons].BuildEdgeTable();

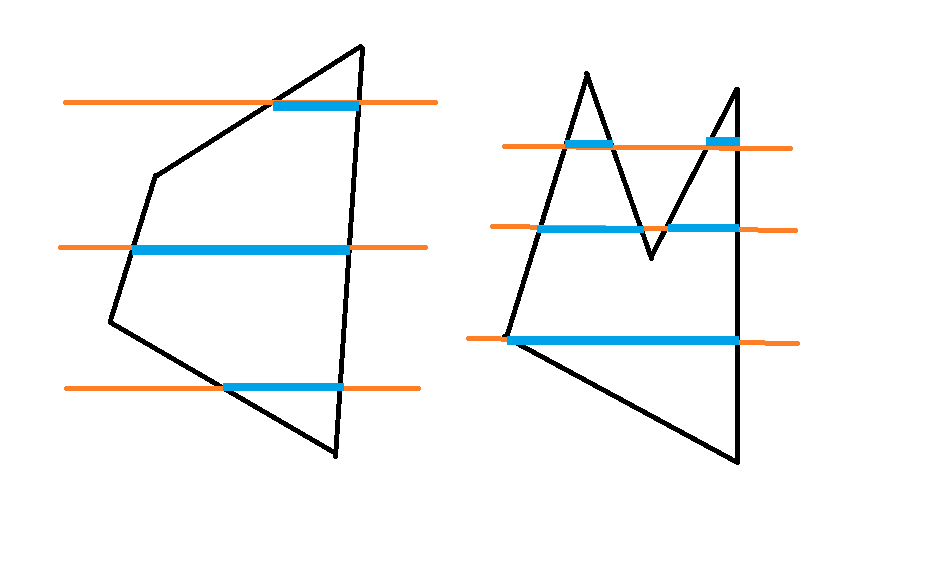
ScanlineFill(pDC, ThePolygons[numPolygons], G\_cLineColor);

ThePolygons[numPolygons].BuildEdgeTable();//上面的边表被改动，重新建立边表

ScanlineFill(m\_pMemDC, ThePolygons[numPolygons], G\_cLineColor);

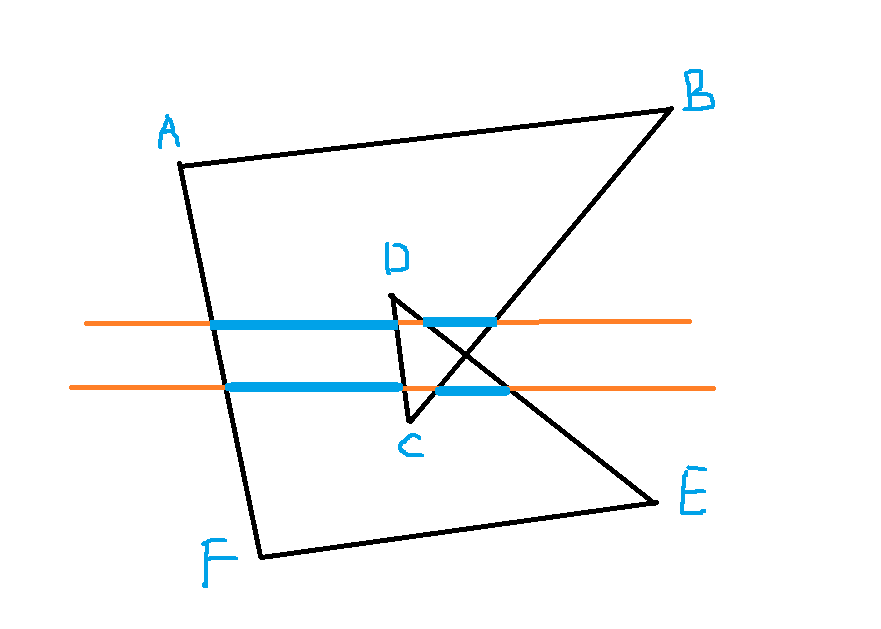
}

1. **针对各种情况进行算法检验**
2. 凸多边形、凹多边形



如上图所示，对于基本多边形，扫描线算法正常运作。

1. 带孔的多边形



如上图所示，对于带孔的多边形，扫描线算法依然适用。

但是要注意的是，在上面的扫描线上，扫描线先与DE相交再和BC相交，在下面的扫描线上，扫描线先与BC相交再和DE相交。对应活化边表，如果表中两条边的位置不对换，在计算前后两边的x之间区域时会出现前面的x比后面的x还要大的情况。

这是因为活化边表中x更新为x+Δx后，前面的节点的x值变得比后面节点的x值大，递增顺序被打乱，所以在活化边表中，我们不采用“只在添加新边的时候按照x和dx递增的顺序添加”，而是“在添加完新边并更新完已存在的边的x值后统一排序”，这样可以将扫描线算法推广适用到一些自相交多边形上。

1. 多个多边形

在项目中的CCGPainterView类中，即整个画板的显示区域上定义一个数组来存放多个多边形。定义MyPolygon ThePolygons[MAX\_POLYGONS]多边形数组，并添加int numPolygons成员变量来追踪创建的多边形个数。

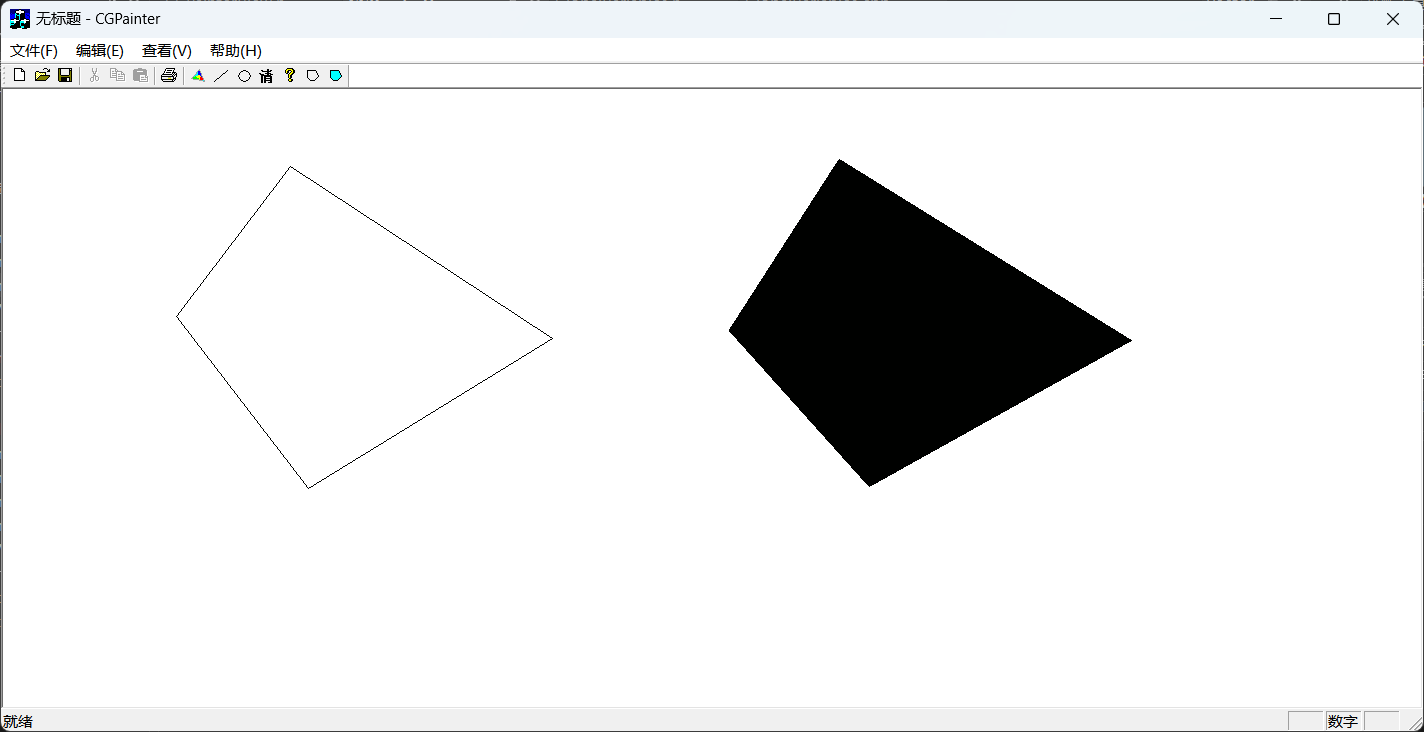
在OnRButtonUp函数中（鼠标右键抬起），原框架实现了绘制多边形的最后一条边的操作，该操作也标志一个多边形创建完毕，在函数中添加语句numPolygons++表示多边形数量加一。

每次创建多边形都用ThePolygons[numPolygons]来操作，numPolygons作为数组下标可以指定控制当前绘制的多边形。

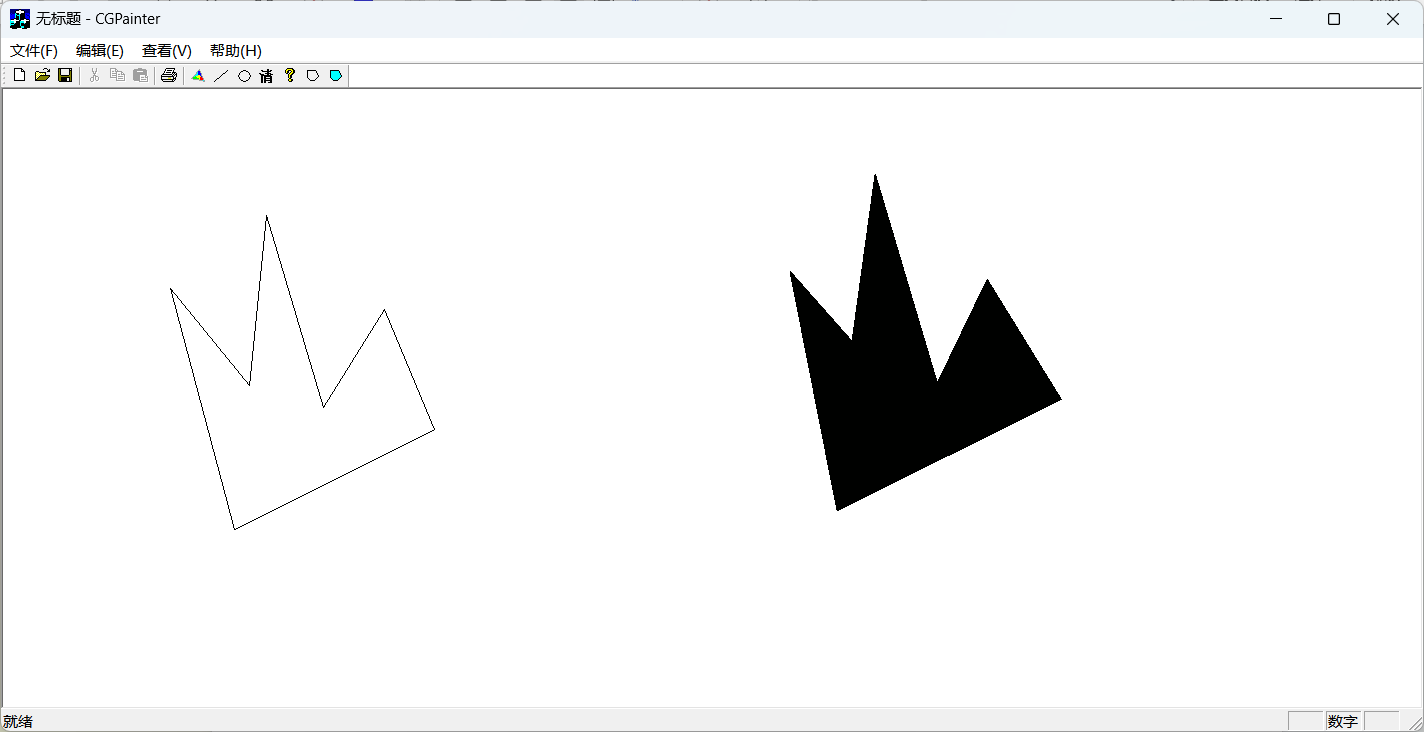
实验结果：

以下为上述多种情况的程序运行结果。

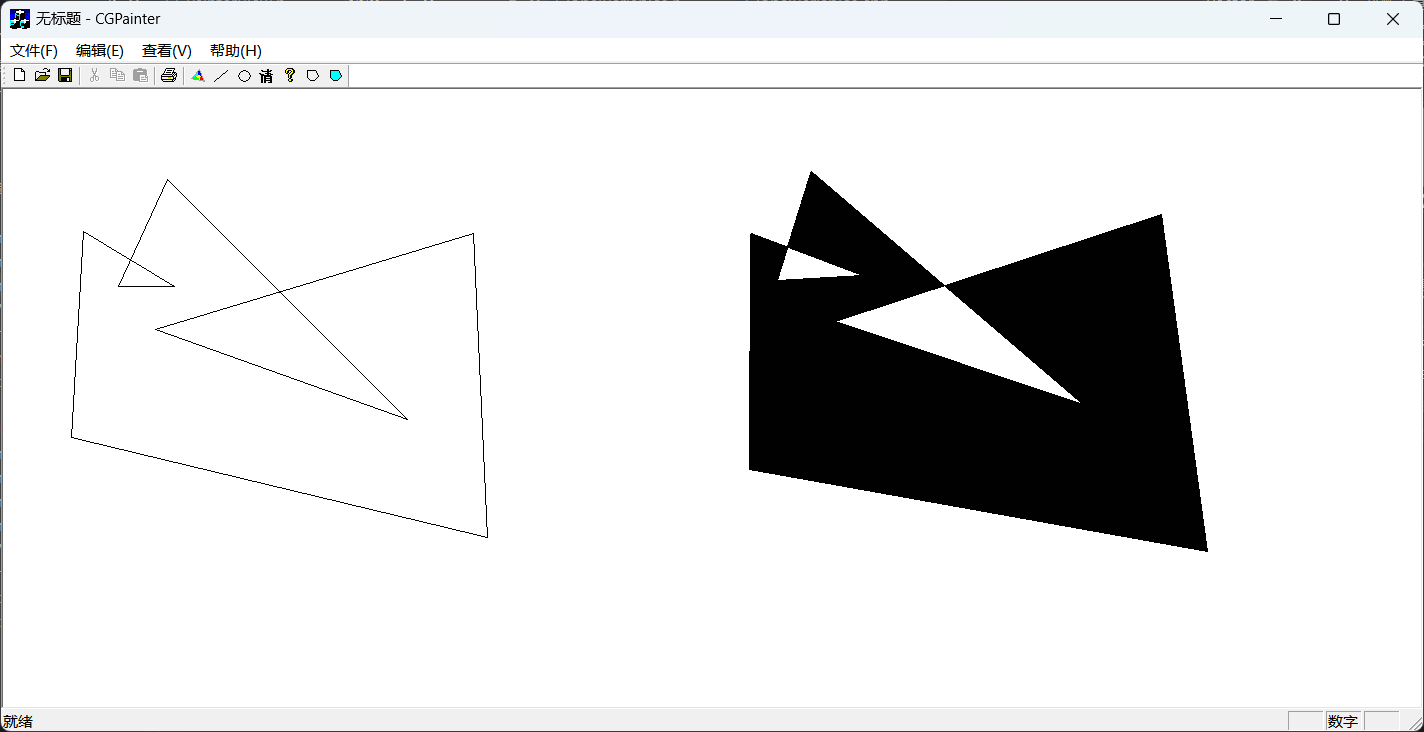
1. 凸多边形：（左为空心绘制，右为实心绘制，下同。）



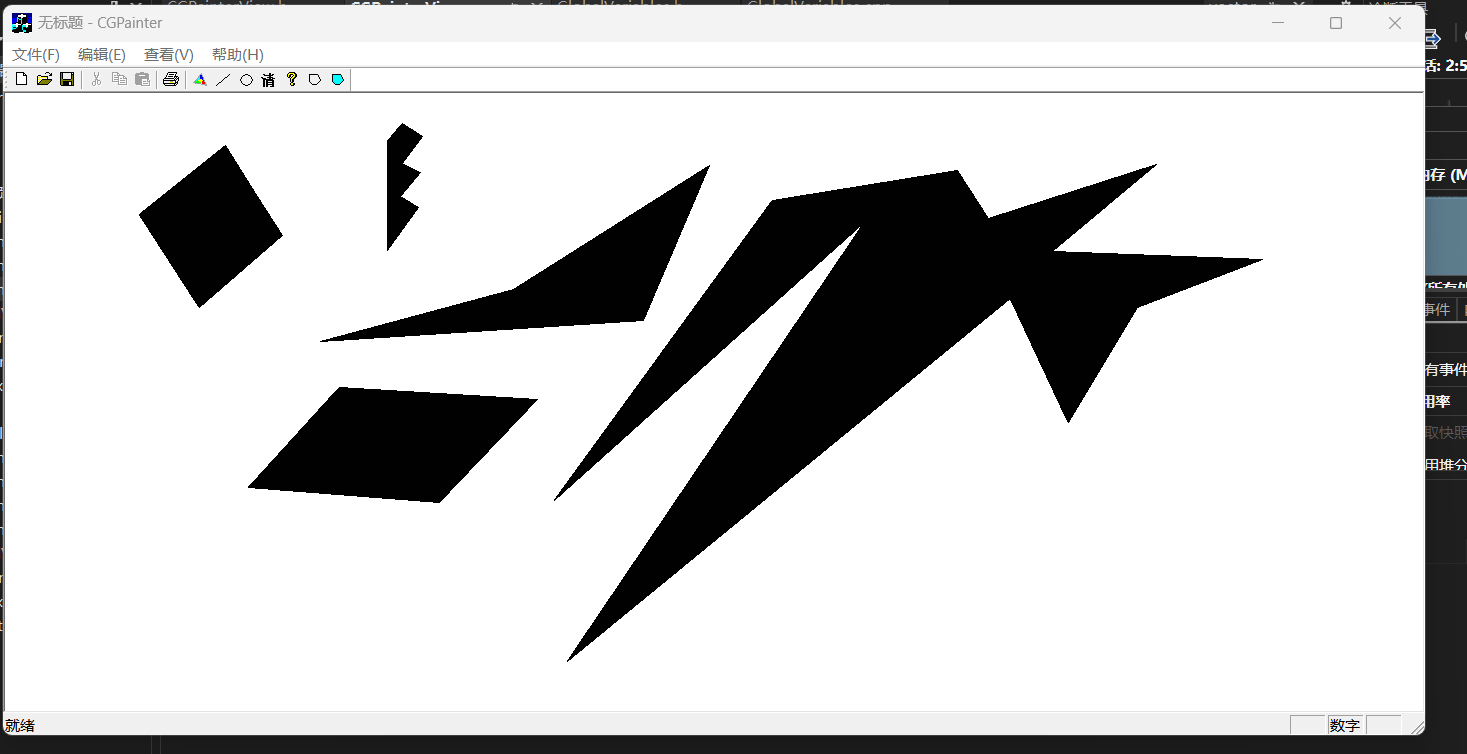
1. 凹多边形：

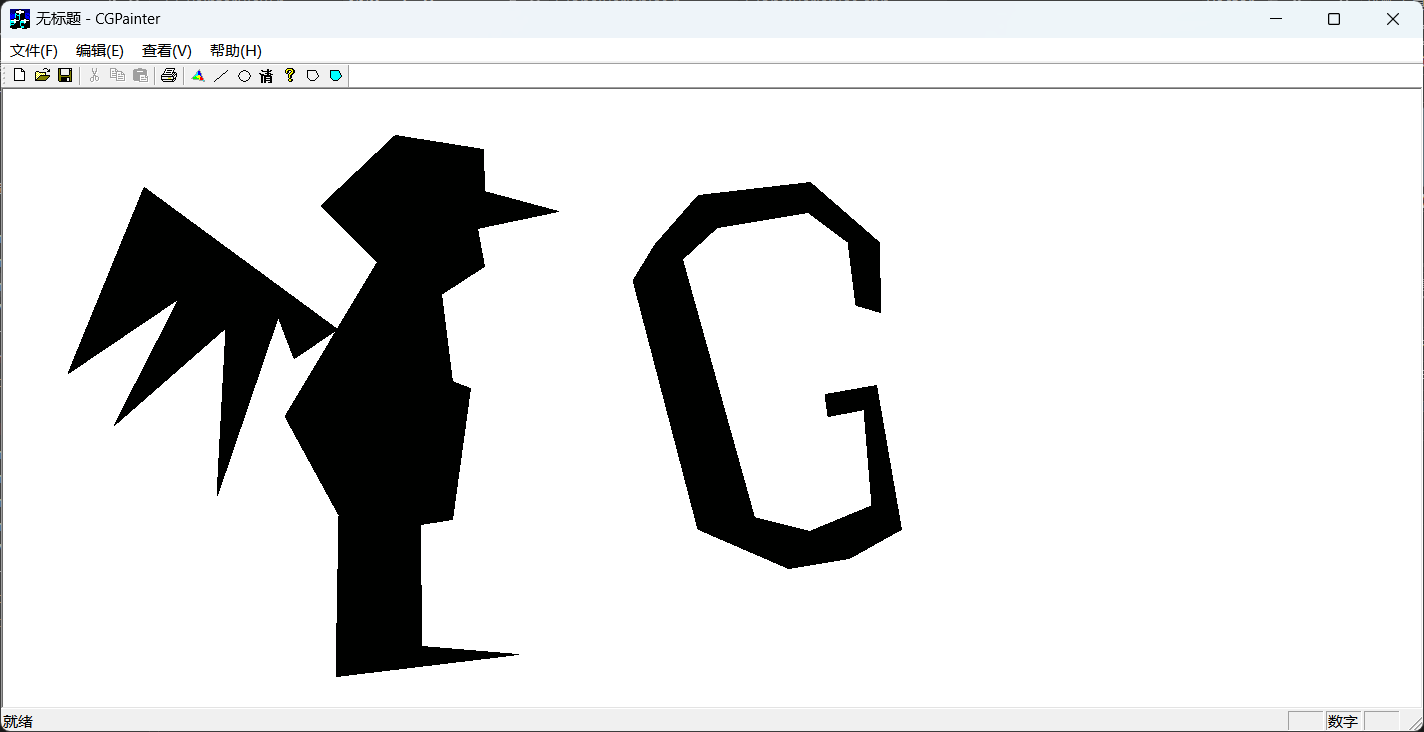


1. 带孔多边形：



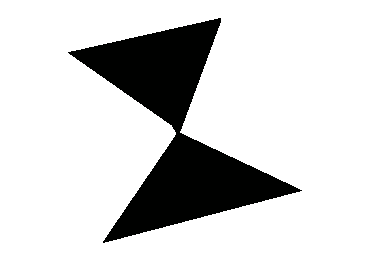
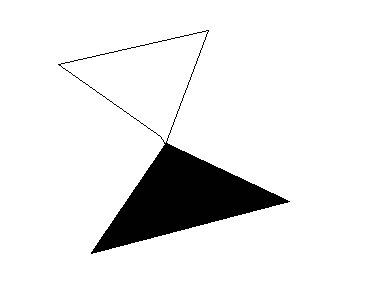
1. 多个多边形：（全部为填充多边形）





结论分析：

扫描线算法进行多边形填充是一个高效且实用的方法。通过对比多边形内部进行种子填充算法，扫描线算法能应对更多特殊情况，比如自相交多边形、带孔多边形。如下图所示，简单种子填充得到填充效果如左图，扫描线填充得到结果为右图。



简单的种子填充算法只填充到边界为特定颜色就停止填充，而扫描线算法通过顶点信息能得到全局视角，不会出现上面的只填充一部分的情况。当然，将种子填充算法进行相应的改进也能实现较好的功能。

扫描线算法基于活化边表和边表，能够有效地对多边形区域进行填充。在实现过程中，对于活化边表的排序和边表的构建采取了相应的优化策略，这有助于提升填充算法的效率及应对复杂多边形（自相交多边形、带孔多边形等）填充。

通过此次实验，我对多边形填充算法有了更深入的了解，并成功地将其应用到项目中。同时，对项目框架进行了改动，实现了更多的功能扩展。针对目前实现的功能，可以考虑进一步的优化和改进，比如针对填充算法的性能提升、界面的友好性优化等方面。同时也可以考虑实现类似于种子填充算法的操作，在屏幕上绘制多边形后，点击内部点即可填充对应多边形，这需要升级判断内外点的算法以适用于复杂多边形。