实验内容：

* 1. 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）…

* 1. 采用C/C++编写程序。
  2. 选作：

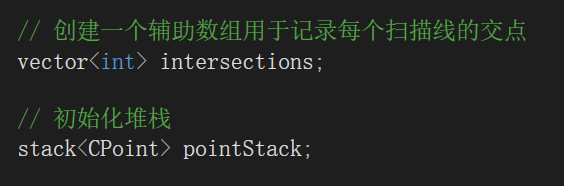
改进提供的polygonfill算法演示界面（polygonfill目录下），写入你的填充算法代码。

实验要求：

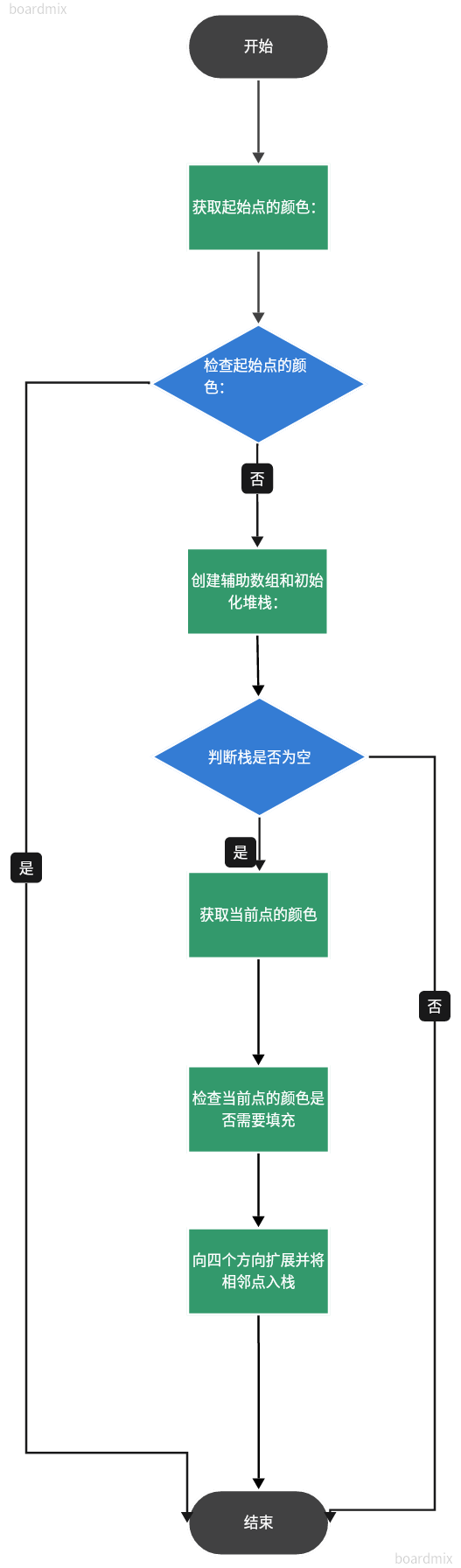
1. 调试通过算法程序；
2. 撰写实验报告（参照所附《实验报告书模板》）：

* 说明算法中所采用的数据结构；

在该程序中，使用的是栈，实现扫描线算法进行多边形区域填充。



* 说明算法各函数的功能；



算法流程图

算法的介绍：

1. 获取起始点颜色：

通过pDC->GetPixel(startPoint)获取起始点的颜色。

1. 检查起始点颜色：

如果起始点的颜色是边界颜色或者已经是填充颜色，直接返回，不进行填充。

1. 初始化堆栈和辅助数组：

创建一个堆栈pointStack用于存储待处理的点，将起始点压入堆栈。还创建了一个辅助数组intersections，用于记录每个扫描线的交点。

1. 扫描线填充循环：

不断从堆栈中取出点进行处理，填充当前点，并将相邻未处理的点入栈。这样的过程会一直进行，直到堆栈为空。

1. 向四个方向扩展：

对当前点进行右、左、上、下四个方向的扫描，将相邻的未处理点入栈，以便后续继续填充。

其主要功能是在一个图形区域内进行颜色的填充，填充的起始点由startPoint指定，填充的颜色由fillCol指定。在进行填充时，会考虑到边界颜色boundaryCol，避免填充到边界外或已经被填充的区域。

* 提供程序源代码并进行必要的注释；

void CCGPainterView::BoundaryFill\_4Connection(CDC \*pDC, CPoint startPoint, COLORREF fillCol, COLORREF boundaryCol)

{

// 获取起始点的颜色

COLORREF startCol = pDC->GetPixel(startPoint);

// 如果起始点的颜色是边界颜色或者已经是填充颜色，直接返回

if (startCol == boundaryCol || startCol == fillCol)

{

return;

}

// 创建一个辅助数组用于记录每个扫描线的交点

vector<int> intersections;

// 初始化堆栈

stack<CPoint> pointStack;

// 将起始点入栈

pointStack.push(startPoint);

// 扫描线填充

while (!pointStack.empty())

{

// 弹出栈顶点

CPoint currentPoint = pointStack.top();

pointStack.pop();

// 获取当前点的颜色

COLORREF currentCol = pDC->GetPixel(currentPoint);

// 如果当前点的颜色不是边界颜色并且不是填充颜色

if (currentCol != boundaryCol && currentCol != fillCol)

{

// 填充当前点

pDC->SetPixel(currentPoint, fillCol);

// 向右扫描

CPoint rightPoint(currentPoint.x + 1, currentPoint.y);

pointStack.push(rightPoint);

// 向左扫描

CPoint leftPoint(currentPoint.x - 1, currentPoint.y);

pointStack.push(leftPoint);

// 将当前点的上下两个相邻点入栈

if (currentPoint.y - 1 >= 0)

{

CPoint upPoint(currentPoint.x, currentPoint.y - 1);

pointStack.push(upPoint);

}

if (currentPoint.y + 1 < pDC->GetDeviceCaps(VERTRES))

{

CPoint downPoint(currentPoint.x, currentPoint.y + 1);

pointStack.push(downPoint);

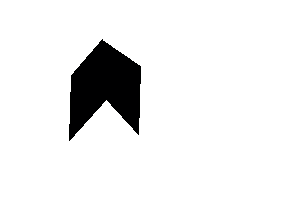
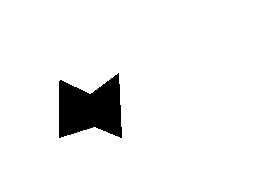
}

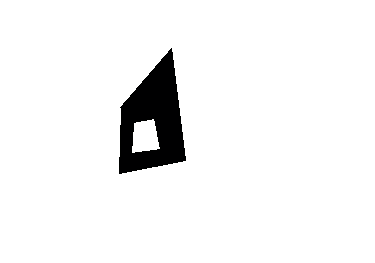
}

}

}

* 针对各种情况进行算法检验（定义任意多边形如：凸多边形、凹多边形、带孔的多边形、多个多边形…），并对结果进行说明。





* 若在程序中，有任何创新，请注明。将视情况获得加分。
* **选作：**

在原来代码给定的基础上，我修改了原来代码中的递归算法，利用了队列这种数据结构，采用广度优先搜索（BFS）的方式，避免了潜在的栈溢出问题。这种方法更适合处理大面积的填充。

改进算法的思路如下：

1. 获取绘图区域的最大宽度和高度：

使用GetDeviceCaps函数获取水平（HORZRES）和垂直（VERTRES）方向上的最大分辨率，以确定绘图区域的大小。

1. 初始化队列和加入起始点：

创建一个队列q用于保存待填充的点。

将起始点(startPoint)加入队列。

1. 循环处理队列中的点：

使用while (!q.empty())循环，直到队列为空。

从队列中取出一个点，即队列头部的点。

1. 获取当前点的颜色：

使用GetPixel获取当前点的颜色。

1. 填充当前点：

如果当前点的颜色既不是边界色又不是填充色，将当前点的颜色设置为填充色(fillCol)。

1. 将相邻点加入队列：

检查相邻的上、下、左、右四个方向的点，如果在绘图区域内且未被访问过，将这些点加入队列。

1. 重复步骤3~6，直到队列为空：

不断从队列中取出点，填充该点并将其相邻未填充过的点加入队列。

实验代码如下：

// 获取绘图区域的最大宽度和高度

int maxX = pDC->GetDeviceCaps(HORZRES);

int maxY = pDC->GetDeviceCaps(VERTRES);

// 使用队列进行广度优先搜索，避免栈溢出的问题

std::queue<std::pair<int, int>> q;

// 将起始点加入队列

q.push(std::make\_pair(startPoint.x, startPoint.y));

// 循环处理队列中的点

while (!q.empty())

{

// 取出队列头部的点

std::pair<int, int> fillPoint = q.front();

q.pop();

// 获取当前点的颜色

COLORREF currentCol = pDC->GetPixel(fillPoint.first, fillPoint.second);

// 如果颜色不是边界色和填充色，则进行填充

if (currentCol != boundaryCol && currentCol != fillCol)

{

// 设置当前点的颜色为填充色

pDC->SetPixelV(fillPoint.first, fillPoint.second, fillCol);

// 将相邻点加入队列

if (fillPoint.second + 1 < maxY) // 检查边界条件

q.push(std::make\_pair(fillPoint.first, fillPoint.second + 1)); // (x, y+1)

if (fillPoint.second - 1 >= 0)

q.push(std::make\_pair(fillPoint.first, fillPoint.second - 1)); // (x, y-1)

if (fillPoint.first - 1 >= 0)

q.push(std::make\_pair(fillPoint.first - 1, fillPoint.second)); // (x-1, y)

if (fillPoint.first + 1 < maxX)

q.push(std::make\_pair(fillPoint.first + 1, fillPoint.second)); // (x+1, y)

}

}

实验结果及结论分析：

1-代码中扫描线是按照先左后右的方式进行填充，这个顺序可以变，但要注意种子点是否填充过。

2- 在画线过程中需要严格按照逆时针方向选取顶点，因为该工程中包含内外判断算法，改算法是按照逆时针存储顺序设计，若不这样做将会无法填充多边形。

3-经过以上测试，该算法能够适应任何多边形填充且效果较好。

4-在改进的算法中，利用了队列和BFS广度优先算法，算法会逐步填充与起始点相邻的区域，使得填充的速度大大加快，同时也有效的避免了由于递归调用而导致栈溢出的问题。