

河海大学



计算机图形学课程报告

专 业 计算机科学与技术

学 号 1661310314

姓 名 江聚勇

2019 年 4 月

目录

一、实验目的	3
二、实验内容	3
三、算法实现	3
1、算法原理	3
2、活性边表具体变化过程	4
3、算法伪代码	5
4、其它功能算法原理	6
四、实验结果	7
1、橡皮筋交互输入多边形	7
2、撤销上一步输入的多边形	8
3、清屏重置多边形	9
4、多边形扫描算法中的顶点处理	9
5、用类（或模版类）来表示数据结构多文档组织	10
6、填充模式设定	10
7、菜单交互	11
8、输入的多边形指定颜色填充	12
9、自相交多边形、多个多边形的扫描填充	13
10、文件存储和读出已经交互输入的多边形	14
五、总结与体会	16

一、实验目的

利用 OpenGL 实现多边形扫描填充算法，提高利用计算机图形学算法知识解决问题的编程能力。

二、实验内容

- 利用 OpenGL 实现
- 通过橡皮筋交互输入多边形
- 清屏重置多边形
- 多边形扫描算法中的顶点处理以每条边减去一个像素方法处理
- 要类（或模版类）来表示数据结构
- 多文档组织，至少要用头（.h）文件表示数据结构
- 填充模式自由设定，如等间隔、斜扫描线填充
- 通过菜单交互
- 自相交多边形、多个多边形的扫描填充
- 通过文件存储和读出已经交互输入的多边形
- 其他多边形填充算法实现，可以调用 OpenGL 函数库

三、算法实现

1、算法原理

扫描线多边形区域填充算法是按扫描线顺序（由下到上），计算扫描线与多边形的相交区间，再用要求的颜色显示这些区间的像素，即完成填充工作。对于一条扫描线，多边形的填充过程可以分为四个步骤，如图 1 所示：

- （1）求交：计算扫描线与多边形各边的交点；
- （2）排序：把所有交点按 x 值递增顺序排序；
- （3）配对：第一个与第二个，第三个与第四个等等；每对交点代表扫描线与多边形的一个相交区间；
- （4）填色：把相交区间内的像素置成多边形颜色；

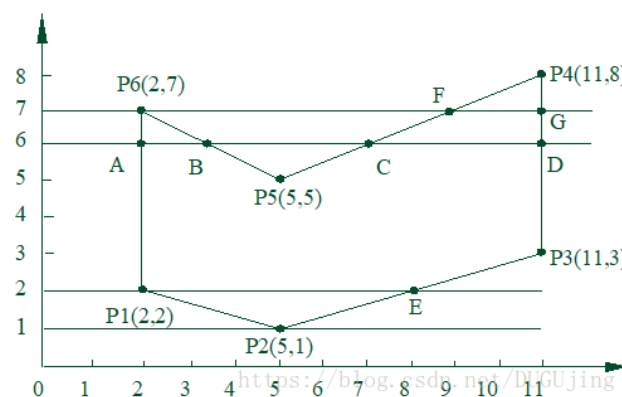


图 1 扫描线多边形区域填充算法基本原理图

2、活性边表具体变化过程

为了提高效率，在处理一条扫描线时，仅对与它相交的多边形的边进行求交运算，避免了某一扫描线对未相交的边的多余运算。把与当前扫描线相交的边称为活性边，并把它们按与扫描线交点 x 坐标递增的顺序存放在一个链表中，称此链表为活性边表（AET）。

假定当前扫描线与多边形某一条边的交点的 x 坐标为 x ，则下一条扫描线与该边的交点不用重复计算，只要加一个增量 Δx (连贯性)。通过如下公式说明：

设某一边所在的直线为： $ax+by+c=0$ ，若 $y=y_i, x=x_i$ ，则当

$x_{i+1} = x_i - \frac{b}{a}$ ，其中 $\Delta x = -\frac{b}{a}$ 为斜率的倒数。那么 Δx 可以存放在对应边的活性

边表结点中。另外，使用增量法计算时，需要知道一条边何时与下一条扫描线相交，以便及时把它从活性边表中删除出去，避免下一步进行无谓的计算。综上所述，活性边表的结点中至少应为对应边保存如下内容：

X	ΔX	Y_{\max}	
---	------------	------------	--

- 第 1 项存当前扫描线与边的交点坐标 x 值；
- 第 2 项存从当前扫描线到下一条扫描线间 x 的增量 Δx ；
- 第 3 项存该边所交的最高扫描线号 y_{\max} ；
- 第 4 项存指向下一条边的指针（ \wedge 代表一条边的退出，即结束或抛弃）；

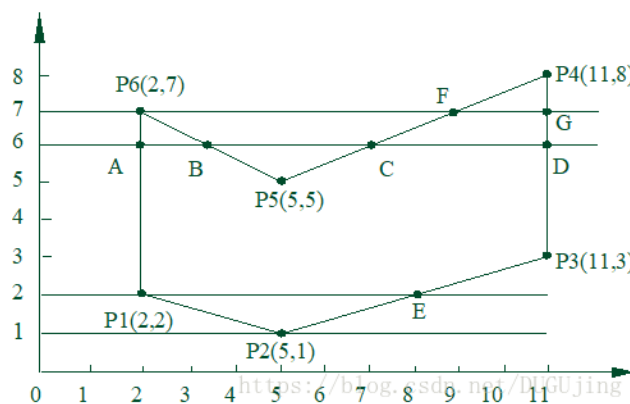


图 2 扫描线多边形区域填充具体实现过程图

扫描线 6 的活性边表：

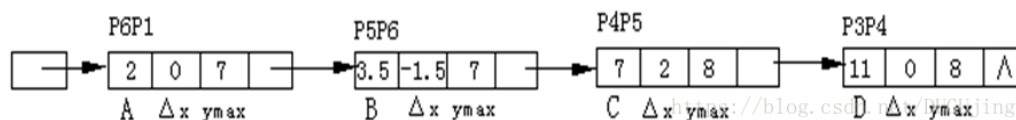


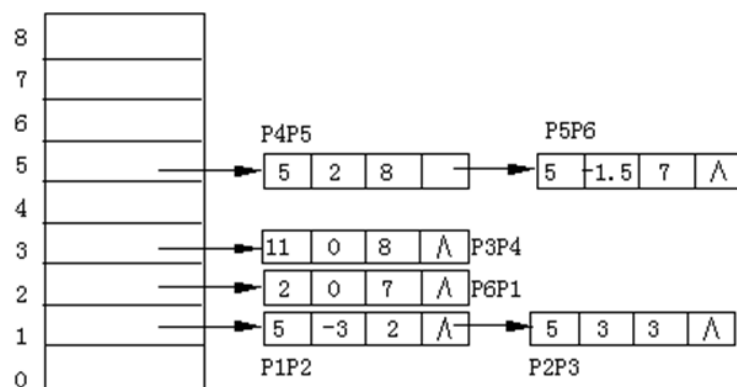
图 3 扫描线 6 的活化边表

扫描线 7 的活性边表：



图 4 扫描线 7 的活化边表

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），存放在该扫描线第一次出现的边。也就是说，若某边的较低端点为 y_{min} ，则该边就放在扫描线 y_{min} 的新边表中，如图 5 所示。



<https://blog.csdn.net/DUGUjing>

图 5 新边表

具体说明：

- (1) 图 5 结构实际上是一个指针数组，数组的每个变量是个指针，这个指针指向所有的以这个 y 值作为起点的边。
- (2) 把多边形所有的边全部填成这样的结构，插到这个指针数组里面来。
- (3) 从上面这个指针数组里面就知道多边形是从哪里开始的。在这个指针数组里只有 1、2、3、5 处有边，因此当从下往上进行扫描转换的时候，从 $y=1$ 开始做，而在 1 这条线上有两条边进来了，然后就把这两条边放进活性边表来处理。
- (4) 当处理 $y=2$ 这条扫描线时（活性边里有 2 条边），先看活性边表里是否有边要退出和加入，实际上 $p1p2$ 边要退出了（ $y=y_{max}$ ）， $p1p6$ 边要加入了。退出的边要从活性边表里去掉，不退出的边要进行处理，即把 x 值加上一个增量。（这时 $p1p2$ 变后边需要加一个 Δ ）
- (5) 后面持续步骤 3, 4，只到 $y=5$ 这扫描线结束。
- (6) 即每做一次新的扫描线时，要对已有的边进行三个处理：一是否被去除掉；如果不被去除；第二就要对它的数据进行更新，所谓更新数据就是要更新它的 x 值，即 $x+\Delta x$ ；最后，就是有没有新的边进来，新的边在 NET 里，可以插入排序插进来。

3、算法伪代码

```

void polyfill (多边形 polygon, 颜色 color)
{ for (各条扫描线 i )
    { 初始化新边表头指针 NET [i]; 把  $y_{min} = i$  的边放进边表 NET [i];      }
  y = 最低扫描线号;
  初始化活性边表 AET 为空;
  for (各条扫描线 i )
      { 把新边表 NET[i] 中的边结点用插入排序法插入 AET 表，使之按  $x$  坐标递增顺序排列;

```

```

        遍历 AET 表, 把  $y_{max} = i$  的结点从 AET 表中删除, 并把  $y_{max} > i$  结点的
        x 值递增  $D_x$ ;
        若允许多边形的边自相交, 则用冒泡排序法对 AET 表重新排序;
        遍历 AET 表, 把配对交点区间(左闭右开)上的像素(x, y), 用 drawpixel
        (x, y, color) 改写像素颜色值;
    }
}

```

4、其它功能算法原理

- 为每一个多边形单独填充颜色, 利用菜单交互, 输入当前所绘制的多边形的颜色, 从而改变了全局颜色变量, 从而颜色的信息和多边形各个顶点的信息一起被保存在多边形中。

```

class polygon //多边形类, 存了一个多边形
{
public:
    vector<point> p; //多边形的顶点
    GLfloat red, green, blue; //用于保存每一个多边形所填充的颜色
};

```

- 撤销上一个所绘制的多边形, 仅仅是利用通过菜单交互, 将存放多边形的向量容器删除掉最后一个元素, 那么在循环绘制多边形形式可以实现撤销了上一个所绘制多边形的效果。

```

}
void clearpolygon() {
    s.pop_back();
    cout << "Last polygon has been cleared!" << endl;
    glFlush(); //刷新缓存
}

```

- 重置窗口的所有多边形, 即将存放多边形的容器向量清空即可。

```

}
void clearpolygonAll() {
    s.clear();
    cout << "All polygons have been cleared!" << endl;
    glFlush(); //刷新缓存
}

```

- 文件存储和文件读出利用将存放多边形的容器向量信息, 包括每一个多边形的顶点和颜色信息。利用文件的输入和输出流, 即可将得到的多边形信息存放到文本文档中, 同时可以将多边形信息从文本文档中输出重新格式化容器向量的形式, 即可绘制出多边形。

```

176 //根据指定的文件从文件中读取多边形，将读多边形容器向量，并将多边形显示出来
177 void GetPolygon()
178 {
179     ifstream in("savepolygon.txt");
180     vector<polygon> temp;
181     string bufline;//保存从文件中读出的字符串
182     if (!in.is_open())
183         cout << "Error:无法打开该文件! ";
184     else
185         while (!in.eof())
186         {
187             while (getline(in, bufline))
188             {
189                 polygon poly;
190                 vector<string> line = SplitString(bufline, "#");//根据#进行字符串的分割，隔离出一行中的颜色和坐标
191                 vector<string> coord = SplitString(line[0], " ");//根据空格得到所有每个点的坐标
192                 vector<string> color = SplitString(line[1], ",");
193                 poly.red = (float)atoi(color[0].c_str());
194                 poly.green = (float)atoi(color[1].c_str());
195                 poly.blue = (float)atoi(color[2].c_str());
196                 for(int i=0;i<coord.size()-1;i++)
197                 {
198                     point getp;
199                     getp.x=atoi(SplitString(coord[i], " ")[0].c_str());
200                     getp.y = atoi(SplitString(coord[i], " ")[1].c_str());
201                     poly.p.push_back(getp);
202                 }
203                 temp.push_back(poly);
204             }
205         }
206         //从文件中读出多边形，然后进行赋值显示
207         for (int i = 0; i < temp.size(); i++)
208             s.push_back(temp[i]);
209         in.close();
210         cout << "成功读取多边形文件! ";

```

```

//将保存多边形的容器向量，保存到文件中，保存的是每个多边形的顶点坐标值
void SavePolygon()
{
    int polygonnum = s.size();//获得多边形的数量，看一保存
    ofstream out;//开辟文件输出流
    out.open("savepolygon.txt");
    if (!out.is_open())//不存在该文件时
        cout << "error: 不存在该文件，无法打开! " << endl;
    else
        for (int i = 0; i < polygonnum; i++)
        {
            polygon poly = s[i];
            int pointsize = poly.p.size();
            for (int j = 0; j < pointsize; j++)
                out << poly.p[j].x << " " << poly.p[j].y << " ";//多边形点与点之间用空格间隔开
            out << "#" << poly.red << " " << poly.green << " " << poly.blue;//保存多边形的颜色值
            out << endl;//当一个多边形输入完毕，换行输入下一个多边形
        }
    out.close();
    cout << "成功保存多边形! ";
}

```

四、实验结果

1、橡皮筋交互输入多边形

利用鼠标移动的位置和已经绘制好的多边形存放的顶点坐标，进行画线，达到橡皮筋交互输入多边形的功能。

核心代码：

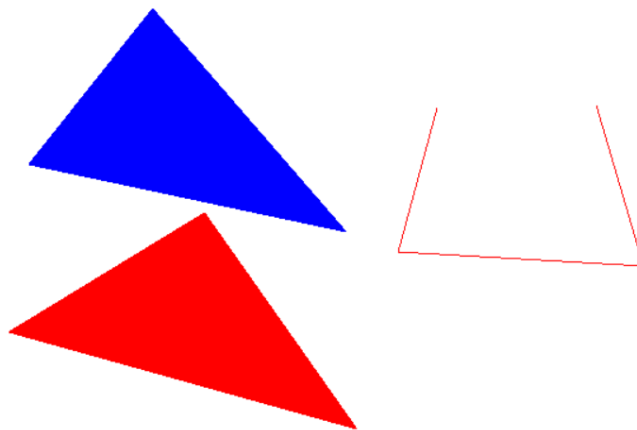
```

//看多边形类向量是否为空，即判断除了当前正在画的多边形是否还有曾经已经画好的多边形
if (!s.empty())
{
    for (i = 0; i < s.size(); i++) //对多边形类向量循环，该向量中的每个元素代表一个多边形
    {
        GLfloat a=s[i].red,b=s[i].green,c=s[i].blue;
        glColor3f(a, b, c);
        int k = s[i].p.size(); //将一个多边形的点的个数
        for (j = 0; j < k; j++) //画多边形
        {
            glBegin(GL_LINES); //将当前的点与后一个点连线
            glVertex2i(s[i].p[j].x, s[i].p[j].y);
            glVertex2i(s[i].p[(j + 1) % k].x, s[i].p[(j + 1) % k].y); //，通过取模操作来避免越界问题，该思路来源于循环队列
            glEnd();
        }
        PolygonFill(s[i]);
    }
}

//将确定的最后一个点与当前鼠标所在位置连线，即动态画线 如同橡皮筋
if (!p.empty())
{
    glBegin(GL_LINES);
    glVertex2i(p[j].x, p[j].y);
    glVertex2i(move_x, move_y);
    glEnd();
}

```

运行结果：



2、撤销上一步输入的多边形

只需要对存放已经绘制好的多边形矢量容器，删除最后一个元素，即可实现撤销上一个实现完成的多边形。

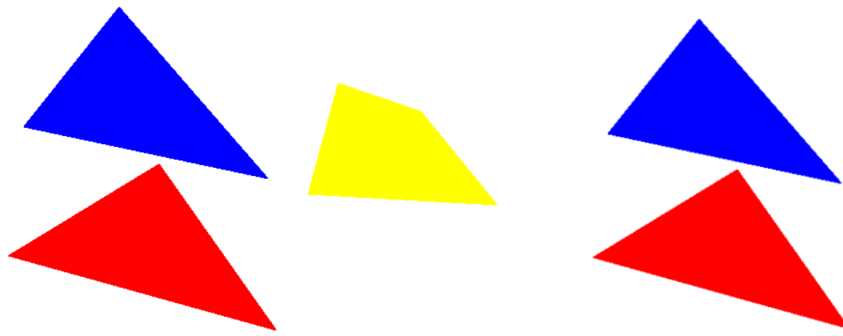
核心代码：

```

void clearpolygon() {
    s.pop_back();
    cout << "Last polygon has been cleared!" << endl;
    glFlush(); //刷新缓存
}

```

运行结果：



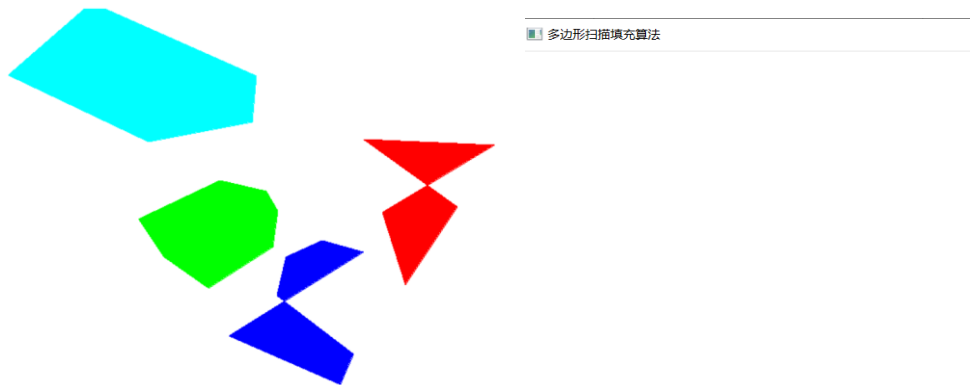
3、清屏重置多边形

将存放已经绘制好的多边形矢量容器全部清空，即可实现清屏重置多边形的功能。

核心代码：

```
void clearpolygonAll() {  
    s.clear();  
    cout << "All polygons have been cleared!" << endl;  
    glFlush(); //刷新缓存  
}
```

运行结果：



4、多边形扫描算法中的顶点处理

为了保证相邻多边形中的像素不会相互覆盖，而保证了对指定多边形的内部的正确填充，将所有顶点处理以每条边减去一个最高的端点的像素点的办法。

核心代码：

```

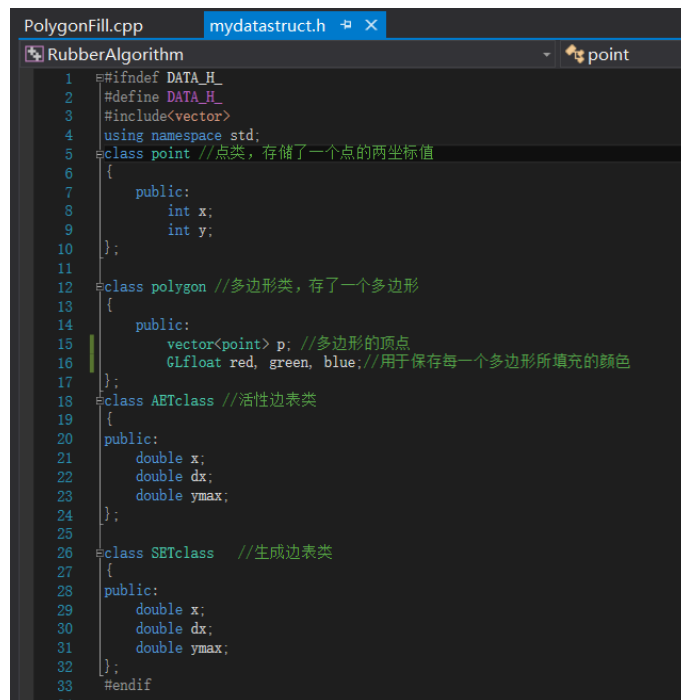
//更新活性边表
int count = AETs.size();
if (count > 0) {
    vector<AETclass>::iterator it;
    for (it = AETs.begin(); it != AETs.end(); it++) {
        it->x = it->x + it->dx; //利用扫描线的连贯性，进行增量计算
        if (it->ymin == i) { //删除ymin=i的结点，即减去一个像素，删除最高顶点的一个像素，防止交点计数错误
            it = AETs.erase(it);
        }
        else {
            it++;
        }
    }
}
}

```

5、用类（或模版类）来表示数据结构多文档组织

通过.h文件定义了 point 类、polygon 类、AETclass 类和 SETclass 类，分别用于顶点、多边形、活化边表，有序边表。

核心代码：



```

PolygonFill.cpp  mydatastruct.h
RubberAlgorithm
1  #ifndef DATA_H
2  #define DATA_H
3  #include<vector>
4  using namespace std;
5  class point //点类，存储了一个点的两坐标值
6  {
7      public:
8          int x;
9          int y;
10 };
11
12 class polygon //多边形类，存了一个多边形
13 {
14     public:
15         vector<point> p; //多边形的顶点
16         GLfloat red, green, blue; //用于保存每一个多边形所填充的颜色
17 };
18 class AETclass //活性边表类
19 {
20     public:
21         double x;
22         double dx;
23         double ymax;
24 };
25
26 class SETclass //生成边表类
27 {
28     public:
29         double x;
30         double dx;
31         double ymax;
32 };
33 #endif

```

6、填充模式设定

填充的模式有通用扫描线填充算法、边界填充算法和泛滥填充算法。

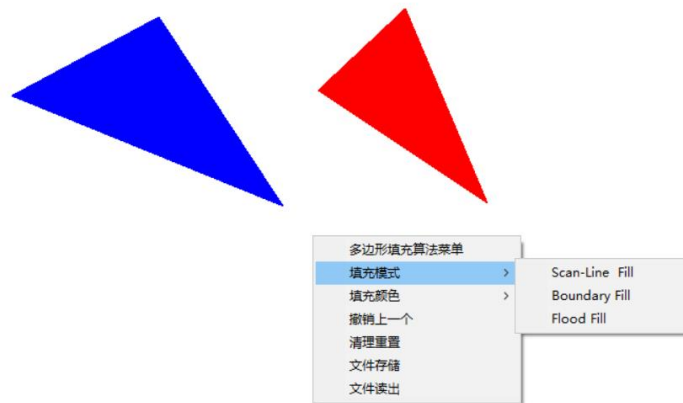
核心代码：

```

//对传入的多边形进行填充颜色
void PolygonFill(polygon s)
{
    vector<point> polypoint = s.p;
    glColor3f(s.red, s.green, s.blue); //根据多边形保存的颜色，进行不同的颜色填充
    int pointnum = polypoint.size(); //每一个多边形点数
    int i, maxy = 0, miny = 5000;
    vector<AETclass> AETs; //建立活化边表
    vector<SETclass> SETs[1024]; //建立新边表
    //先找到所有顶点的y坐标的上下边界即最大和最小值
    for (i = 0; i < pointnum; i++)
    {
        if (polypoint[i].y > maxy)
            maxy = polypoint[i].y;
        else if (polypoint[i].y < miny)
            miny = polypoint[i].y;
    }
    //保存每一条边的信息
    for (i = miny; i <= maxy; i++)
        for (int j = 0; j < pointnum; j++)
            if (polypoint[j].y == i)

```

运行结果:



7、菜单交互

利用 OpenGL 的图形用户界面中的菜单功能实现了主菜单和子菜单的功能，方便进行交互选择指定的要求和功能。

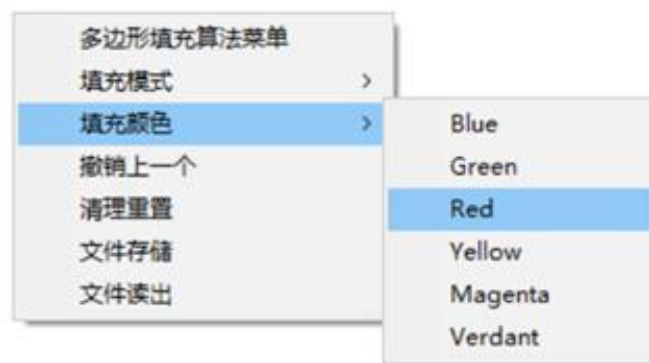
核心代码:

```

//菜单交互的设计
subMenu1 = glutCreateMenu(renderSubMenu);
    glutAddMenuEntry("Scan-Line Fill", 1); //扫描线填充算法
    glutAddMenuEntry("Boundary Fill", 2); //边界填充算法
    glutAddMenuEntry("Flood Fill", 3); //泛滥填充算法
subMenu2 = glutCreateMenu(colorSubMenu);
    glutAddMenuEntry("Blue", 1);
    glutAddMenuEntry("Green", 2);
    glutAddMenuEntry("Red", 3);
    glutAddMenuEntry("Yellow", 4);
    glutAddMenuEntry("Magenta", 5);
    glutAddMenuEntry("Verdant", 6);
glutCreateMenu(mainMenu);
    glutAddMenuEntry("多边形填充算法菜单", 0);
    glutAddSubMenu("填充模式", subMenu1);
    glutAddSubMenu("填充颜色", subMenu2);
    glutAddMenuEntry("撤销上一个", 2);
    glutAddMenuEntry("清理重置", 3);
    glutAddMenuEntry("文件存储", 4);
    glutAddMenuEntry("文件读出", 5);
glutAttachMenu(GLUT_MIDDLE_BUTTON); //将菜单与鼠标的右键绑定起来

```

运行结果:



8、输入的多边形指定颜色填充

通过在将多边形的各个顶点保存时，利用菜单交互的颜色指定，将全局变量 red、green、blue 的颜色值，修改为指定颜色的值

`glColor(red, green, blue)`, 从而改变了填充的颜色。多边形的颜色信息，必须作为多边形如同顶点一样的自身属性，才能在每次显示多边形的时候设定的填充颜色显示出来。

核心代码:

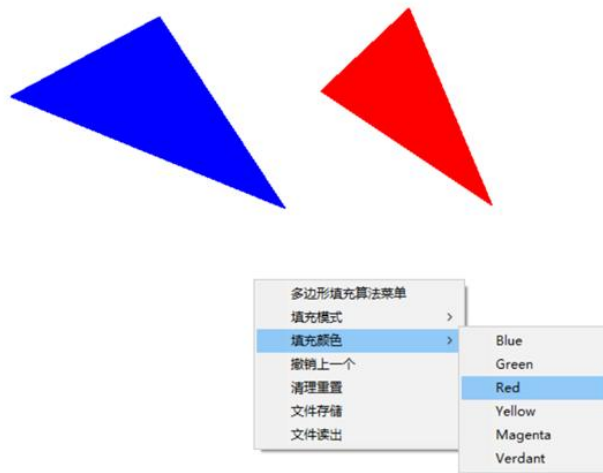
```

class polygon //多边形类，存了一个多边形
{
public:
    vector<point> p; //多边形的顶点
    GLfloat red, green, blue; //用于保存每一个多边形所填充的颜色
};

if (select) //判断右键是否被按下
{
    select = false; //将状态值置为假
    if (!p.empty())
    {
        glBegin(GL_LINES); //将多边形封闭
        glVertex2i(p[j].x, p[j].y);
        glVertex2i(p[0].x, p[0].y);
        glEnd();
        polygon sq; //当按下了鼠标右键才会将完整的该多边形的所有点集作为一个多边形放入多边形类中
        //将封闭了的多边形保存到多边形类中
        for (i = 0; i < p.size(); i++)
            sq.p.push_back(p[i]);
        sq.red = red;
        sq.green = green;
        sq.blue = blue;
        s.push_back(sq); //将绘成的多边形存入多边形类向量中
    }
    p.clear(); //p每次存完一个完整的多边形以后就进行清空，以备后面的多边形点集存放使用
}

```

运行结果：



9、自相交多边形、多个多边形的扫描填充

对于绘制的任意多边形的形状，通过处理各种可能会出现填充错误的地方，从而达到任何形状下，扫描线和边的交点能正确的进行填充。

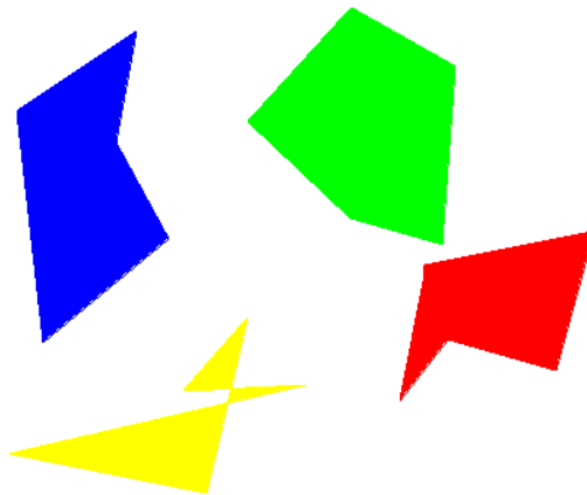
核心代码：

```

//更新活性边表AET，每次遍历画一次线
for (i = miny; i <= maxy; i++)
{
    //更新活性边表
    int count = AETs.size();
    if (count > 0) {
        vector<AETclass>::iterator it;
        for (it = AETs.begin(); it != AETs.end(); it++) {
            it->x = it->x + it->dx; //利用扫描线的连贯性，进行增量计算
            if (it->ymin == i) { //删除ymax==i的结点，即减去一个像素，删除最高顶点的一个像素，防止交点计数错误
                it = AETs.erase(it);
            }
            else {
                it++;
            }
        }
    }
    //通过x的大小对交点进行排序
    sort(AETs.begin(), AETs.end(), cmp);
    vector<SETclass> vSET = SETs[i];
    //判断是否遇到端点，遇到端点则更新AET表
    int setcount = vSET.size();
    if (setcount > 0) {
        for (int seindex = 0; seindex < setcount; seindex++) {
            AETclass aet;
            aet.dx = vSET[seindex].dx;
            aet.x = vSET[seindex].x;
            aet.ymin = vSET[seindex].ymin;
            AETs.push_back(aet);
        }
    }
}
}

```

运行结果：



10、文件存储和读出已经交互输入的多边形

通过将窗口中绘制好的所有多边形，点击鼠标的中间，弹出交互式菜单，选择文件存储，即可将图中所有多边形的顶点坐标，以及多边形的颜色 red, green, blue 分量的值，主要是利用文件的输出和输入流进行功能实现。对于读出已经存放好的多边形文件，要进行一系列的字符串分割函数的处理，已经进制的转化，实现一一对应。

核心代码：

```

//将保存多边形的容器向量，保存到文件中，保存的是每个多边形的顶点坐标值
void SavePolygon()
{
    int polygonnum = s.size();//获得多边形的数量，着一保存
    ofstream out;//开辟文件输出流
    out.open("savepolygon.txt");
    if (!out.is_open())//不存在该文件时
        cout << "error: 不存在该文件，无法打开！" << endl;
    else
        for (int i = 0; i < polygonnum; i++)
        {
            polygon poly = s[i];
            int pointsize = poly.p.size();
            for (int j = 0; j < pointsize; j++)
                out << poly.p[j].x << "," << poly.p[j].y << " ";//多边形点与点之间用空空间隔开
            out << "#" << poly.red << "," << poly.green << "," << poly.blue;//保存多边形的颜色值
            out << endl;//当一个多边形输入完毕，换行输入下一个多边形
        }
    out.close();
    cout << "成功保存多边形！";
}

```

```

//根据指定字符对字符串进行分割，从而能从文件中得到所需要的坐标值，返回的是一个存放分割后的每一个字符串向量容器
vector<string> SplitString(string srcStr, const string& delim)
{
    int nPos = 0;
    vector<string> vec;
    nPos = srcStr.find(delim.c_str());
    while (-1 != nPos)
    {
        string temp = srcStr.substr(0, nPos);
        vec.push_back(temp);
        srcStr = srcStr.substr(nPos + 1);
        nPos = srcStr.find(delim.c_str());
    }
    vec.push_back(srcStr);
    return vec;
}

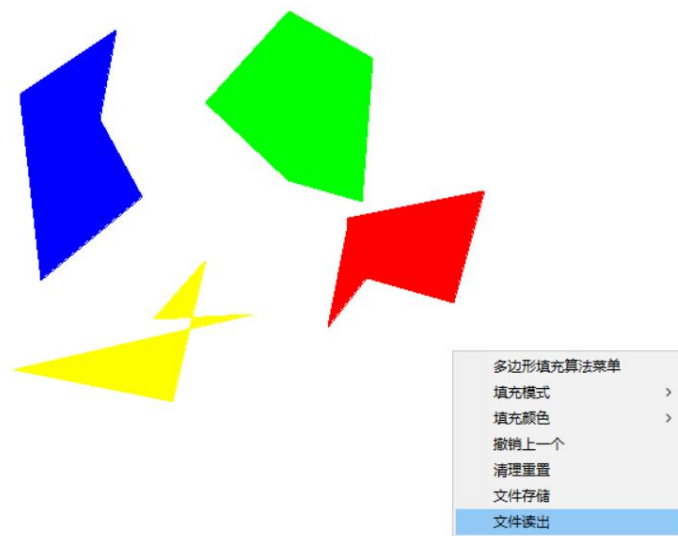
```

```

void GetPolygon()
{
    ifstream in("savepolygon.txt");
    vector<polygon> temp;
    string bufline;//保存从文件中读出的字符串
    if (!in.is_open())
        cout << "Error:无法打开该文件！";
    else
        while (!in.eof())
        {
            while (getline(in, bufline))
            {
                polygon poly;
                vector<string> line = SplitString(bufline, "#");//根据#进行字符串的分割，隔离出一行中的颜色和坐标
                vector<string> coord = SplitString(line[0], ",");//根据空格得到所有每个单坐标
                vector<string> color = SplitString(line[1], ",");
                poly.red = (float)atoi(color[0].c_str());
                poly.green = (float)atoi(color[1].c_str());
                poly.blue = (float)atoi(color[2].c_str());
                for (int i=0; i<coord.size()-1; i++)
                {
                    point getp;
                    getp.x=atoi(SplitString(coord[i], ",")[0].c_str());
                    getp.y = atoi(SplitString(coord[i], ",")[1].c_str());
                    poly.p.push_back(getp);
                }
                temp.push_back(poly);
            }
        }
    //从文件中读出多边形，然后进行赋值显示
    for (int i = 0; i < temp.size(); i++)
        s.push_back(temp[i]);
    in.close();
    cout << "成功读取多边形文件！";
}

```

运行结果：



保存所有多边形的.txt 文件

```
savepolygon.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
294,803 312,645 398,716 362,782 376,858 #0,0,1
452,796 523,874 594,834 585,712 522,730 #0,1,0
573,698 689,721 663,626 589,647 556,605 572,687 #1,0,0
408,612 452,662 424,542 289,569 492,616 #1,1,0
```

五、总结与体会

通过这样一次的多边形扫描线填充算法的计算机图形学实验，我对利用 OpenGL 去编写图形学的算法能力有了很大的提高以及对其本身也有了很大的了解和更加的直观认识。算法实现过程中，如何利用橡皮筋算法得到的多边形顶点，去建立活化边表，并利用新边表更加方便活化边表的更新，通过连贯性，解决了某一条边和扫描线交点的问题，于此同时，为了防止交点的计算错误，将每一条边的较高顶点减去了一个像素，从而避免了重复计算而导致绘制区间的错误。计算机图形学从一开始的晦涩难懂，却通过这样一次的编程算法实践，攻破了心底深处的难关，可以看出学好计算机图形学与大量的编程实践是分不开的。总的来说，从一开始的懊恼，通过查阅资料、阅读已有的橡皮筋算法程序和课本中的扫描线填充算法原理，渐渐的有了更加的直观理解，到最后成功实现了各项多边形填充的功能后的豁然开朗，是一次着实提高自己编程能力和加深计算机图形学算法原理理解的实践，受益匪浅。