Task 1

算法2对算法1的改进：利用扫描线，使每一条扫描线上的像素点在计算时不需要每次都代入方程计算Ei，而使用Ei+=ai的方式，大大减少了计算量。

Task 2

（1）开发和运行环境：win10系统 VS2019 OpenGL

（2）算法设计思想及实现技术

（3）程序源码

#include <math.h>

#include <GL/freeglut.h>

#include <iostream>

#include <time.h>

#define WINDOW\_WIDTH 600 //窗口宽度

#define WINDOW\_HEIGHT 600 //窗口高度

#define TRIANGLE\_QUANLITY 5//三角形数量

//三角形对象

class Triangle {

int a[3][2];//三角形三个顶点a1,a2,a3的x,y坐标

float c[3][3];//三角形三条边c1,c2,c3对应直线方程的三个系数，c1为a1相对的边，即a2a3，以此类推

public:

//利用构造函数给a数组赋值，计算c数组和b数组

Triangle(int, int, int, int, int, int);

int get\_a(int i, int j) { return a[i][j]; };

float get\_c(int i, int j) { return c[i][j]; };

};

//三角形对象的构造函数

Triangle::Triangle(int x1, int y1, int x2, int y2, int x3, int y3) {

{

//利用构造函数给a数组赋值，计算c数组和b数组

//给a数组赋值

a[0][0] = x1;

a[0][1] = y1;

a[1][0] = x2;

a[1][1] = y2;

a[2][0] = x3;

a[2][1] = y3;

//计算c数组

for (int ci = 1; ci <= 3; ci++) {//边的编号

int deltaX = a[(ci + 1) % 3][0] - a[ci % 3][0];//该边两顶点横坐标之差，用作y项的系数

int deltaY = a[(ci + 1) % 3][1] - a[ci % 3][1];//该边两顶点纵坐标之差，用作x项的系数

//约分

int x = deltaX, y = deltaY, t;

while (y != 0) {

t = x % y;

x = y;

y = t;

}

deltaX /= x;

deltaY /= x;

c[ci - 1][1] = (float)deltaX;

c[ci - 1][0] = (float)(-deltaY);

//带入边上的任意一个顶点求出c[ci-1][2]，即直线方程的常数项

c[ci - 1][2] = 0 - a[ci % 3][0] \* c[ci - 1][0] - a[ci % 3][1] \* c[ci - 1][1];

//将ci与其相对的点代入直线方程，结果>0无操作，

// <0则三个系数取相反数，保证像素点代入直线方程时>0在三角形可见侧

// =0则三角形不成立，为了方便设数时已避开

if (a[ci - 1][0] \* c[ci - 1][0] + a[ci - 1][1] \* c[ci - 1][1] + c[ci - 1][2] < 0)

for (int i = 0; i < 3; i++)

c[ci - 1][i] = -c[ci - 1][i];

}

}

}

//创建5个三角形对象

Triangle triangle[TRIANGLE\_QUANLITY] = { Triangle(100, 50, 200, 150, 300, 100),

Triangle(100,150,200,250,100,300),

Triangle(-100,-150,-200,-250,-100,-300),

Triangle(-100,-50,-200,-150,-300,-100),

Triangle(100,-100,200,-250,100,-300) };

//三角形包围盒裁剪

int bboxcut(Triangle\* triangle) {

int worldSize = 0;//世界坐标系尺寸

for (int i = 0; i < TRIANGLE\_QUANLITY; i++)

for (int j = 0; j < 3; j++)

for (int k = 0; k < 2; k++)

if (abs(triangle[i].get\_a(j, k)) > worldSize)

worldSize = triangle[i].get\_a(j, k);

//确定世界坐标系 横纵向坐标极值均为(-worldSize, worldSize)

gluOrtho2D(-worldSize, worldSize, -worldSize, worldSize);

glViewport(0, 0, WINDOW\_WIDTH, WINDOW\_HEIGHT);

return worldSize;

}

//三角形填充算法1：每个像素点分别带入三角形各边的直线方程求解

void triangleFilling1(Triangle\* triangle, int worldSize) {

glBegin(GL\_POINTS);

for (int x = -worldSize; x <= worldSize; x++)

for (int y = -worldSize; y <= worldSize; y++)

for (int i = 0; i < TRIANGLE\_QUANLITY; i++) {

//计算某点是否在三角形的可见侧

bool visible = true;

for (int ci = 0; ci < 3; ci++) {

float result = triangle[i].get\_c(ci, 0) \* x + triangle[i].get\_c(ci, 1) \* y + triangle[i].get\_c(ci, 2);//像素点代入直线方程

//flag和b数组的值对照，如果相同则像素点在可见侧

if (result < 0) {

visible = false;

break;

}

}

if (visible)

glVertex2i(x, y);

}

glEnd();

}

//三角形填充算法2：利用扫描线求解

void triangleFilling2(Triangle\* triangle, int worldSize) {

glBegin(GL\_POINTS);

for (int y = -worldSize; y <= worldSize; y++)

for (int i = 0; i < TRIANGLE\_QUANLITY; i++) {

int x0 = -worldSize;//每条扫描线左边的起始x值

float Ei[3];//存储一个三角形内扫描线最左像素点代入三个直线方程的初值

for (int ci = 0; ci < 3; ci++)

Ei[ci] = triangle[i].get\_c(ci, 0) \* x0 + triangle[i].get\_c(ci, 1) \* y + triangle[i].get\_c(ci, 2);//像素点(x0,y)代入直线方程

for (int x = x0; x <= worldSize; x++) {

if (Ei[0] >= 0 && Ei[1] >= 0 && Ei[2] >= 0)//像素点在三角形三条边的可见侧

glVertex2i(x, y);

for (int j = 0; j < 3; j++)

Ei[j] += triangle[i].get\_c(j, 0);

}

}

glEnd();

}

//初始化

void init()

{

glClearColor(0.0, 0.0, 0.0, 0.0);//黑

}

//图像显示

void display()

{

//三角形包围盒裁剪

int worldSize = bboxcut(triangle);

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(1.0f, 1.0f, 1.0f);

double dur;//计时器

clock\_t start, end;

start = clock();

//三角形填充算法

//triangleFilling1(triangle, worldSize);

triangleFilling2(triangle, worldSize);

end = clock();

dur = (double)(end - start);

std::cout << "Use Time:" << dur / CLOCKS\_PER\_SEC << std::endl;

//画出5个三角形做轮廓方便识别

for (int i = 0; i < TRIANGLE\_QUANLITY; i++) {

glBegin(GL\_LINE\_LOOP);

for (int j = 0; j < 3; j++)

glVertex2i(triangle[i].get\_a(j, 0), triangle[i].get\_a(j, 1));

glEnd();

}

glFlush();

}

int main(int argc, char\*\* argv)

{

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGBA);

glutInitWindowSize(WINDOW\_WIDTH, WINDOW\_HEIGHT);

glutInitWindowPosition(350, 50);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluOrtho2D(-WINDOW\_WIDTH / 2, WINDOW\_WIDTH / 2, -WINDOW\_WIDTH / 2, WINDOW\_WIDTH / 2);

glutCreateWindow("Task 3");

init();

glutDisplayFunc(display);

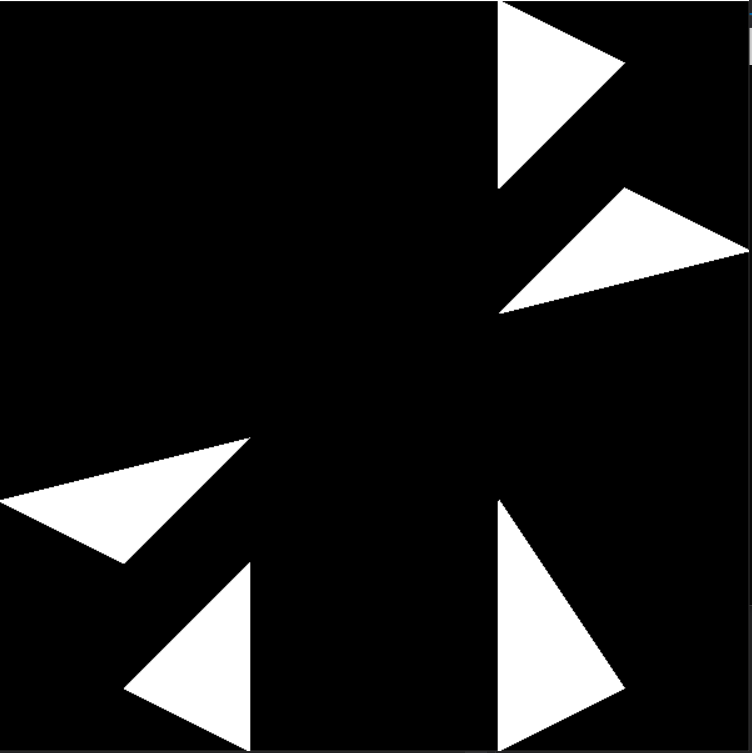
glutMainLoop();

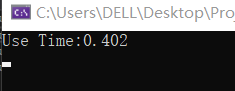
return 0;

}

（4）运行结果

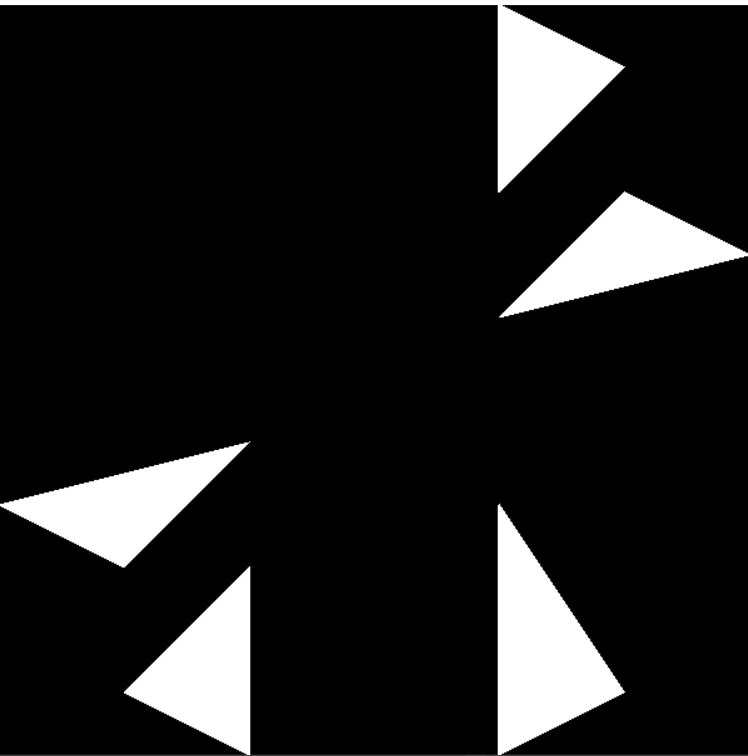
算法1：

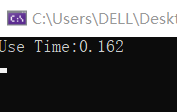




如图所示，用时0.402s。

算法2：





如图所示，用时0.162s，效率大大提高。

（5）分析

遇到的第一个较大的困难是直线方程系数的存储。起初因为是使用结构体存储的三角形顶点，导致直线方程系数的存储和三角形顶点坐标是割裂的，后来想到使用类，能够较好地将二者归到同一个三角形当中；后来又在直线方程系数的计算上出现了问题，因为数组的编号从0开始，一条边又和两个顶点对应，再加上还要考虑到约分的问题，并且要使系数尽可能简单，在数学上不太好处理。

第二个问题是像素点对直线方程的代入。起初我在对象里设置了一个bool类型数组，来记录将像素点代入直线方程后的正负对应三角形的可见侧与不可见侧；项目即将完成的时候发现可以提前处理，在系数处理的时候保证可见侧对应直线方程>0即可，浪费了许多时间。

另外，在编写第一个算法时为了简便我曾将算法的主体部分直接写入对象作为它的一个成员函数，后来考虑到计时的问题又把它拿了出来，但这一版在运行效率上要远高于在对象外单独建立函数，算法运行时间在0.088s左右，可见在类里的成员函数直接调用类成员的效率要远高于外部函数利用get成员函数调用类成员。