云南大学

本科实验报告

课程名称： 计算机图形学实验

实验名称： 实验三 二维图元的填充

学院（系）： 信息学院

专 业： 计算机科学与技术

年 级： 2020级

姓 名： 胡诚皓

学 号： 20201060330

指导教师： 吴昊

成 绩：

2022年 4 月 13 日

**一. 实验目的**

1. 熟悉OpenGL中对颜色的设置

2. 边界填充算法的理解与实现；

3. 泛滥填充算法的理解与实现；

4. 扫描线填充算法的理解与实现；

**二. 实验内容**

仔细阅读实验指导书，基于OpenGL完成以下内容：

（1）颜色的设置；

（2）实现边界填充算法；

（3）实现泛滥填充算法；

（4）实现扫描线填充算法。

**三. 实验代码**

**1. 颜色的设置**

// 测试颜色设置的回调函数

void colorTest(void) {

int x = 0;

glBegin(GL\_LINES);

// 测试蓝色渐变的效果，从左往右，蓝色成分越来越多

for (int i = 0; i < 250; i++) {

glColor3f(0.0f, 0.0f, (float) i / 255);

glVertex2i(x, 0);

glVertex2i(x++, 500);

glVertex2i(x, 0);

glVertex2i(x++, 500);

}

glEnd();

glFlush();

}

**2. 实现边界填充算法**

typedef float Color[3];

void setPixel(GLint x, GLint y) {

glBegin(GL\_POINTS);

glVertex2i(x, y);

glEnd();

}

void getPixel(GLint x, GLint y, Color c) {

Color tmp={0, 0, 0};

glReadPixels(x, y, 1, 1, GL\_RGB, GL\_FLOAT, tmp);

c[0] = tmp[0], c[1] = tmp[1], c[2] = tmp[2];

}

// 四连通的边界填充算法

void boundaryFill4(int x, int y, Color fillColor, Color borderColor) {

Color interiorColor;

// 获取当前点的颜色值以判断是否填充

getPixel(x, y, interiorColor);

// 没到边界且没填充过，就分别往右、左、上、下填充

if (!cmpColor(interiorColor, borderColor) && !cmpColor(interiorColor, fillColor)) {

setPixel(x, y);

// 每填一个点都进行Flush，以展示出填充的过程

glFlush();

// 分别递归边界填充右、左、上、下四个点

boundaryFill4(x + 1, y, fillColor, borderColor);

boundaryFill4(x - 1, y, fillColor, borderColor);

boundaryFill4(x, y + 1, fillColor, borderColor);

boundaryFill4(x, y - 1, fillColor, borderColor);

}

}

// 用以测试4连通边界填充算法的回调函数

void testBoundaryFill(void) {

// 用来填充的颜色，十六进制为#66CCFF

Color fill = {0.4f, 0.8f, 1.0f};

// 作为边界颜色，为黑色

Color border = {0.0f, 0.0f, 0.0f};

// 先画好待填充的区域的边界（一个五角星）

int points[] = {85, 160, 109, 117, 156, 108, 123, 73, 129, 24, 85, 45, 41, 24, 47, 73, 14, 108, 61, 117};

glColor3fv(border);

// 五角星由10个点组成，使用GL\_LINE\_LOOP画首位相接的线

glBegin(GL\_LINE\_LOOP);

for (int i = 0; i < 10; i++) {

glVertex2i(points[2 \* i], points[2 \* i + 1]);

}

glEnd();

glFlush();

printf("boundary draw completed.\n");

// 设置绘图颜色为fill

glColor3fv(fill);

// 选择的起始点位三角形的中心点

boundaryFill4(85, 85, fill, border);

glFlush();

printf("boundary fill done.\n");

}

**3. 实现泛滥填充算法**

void floodFill4(int x, int y, Color fillColor, Color interiorColor) {

Color current;

// 获取当前点颜色值

getPixel(x, y, current);

// 当前颜色值与interiorColor相等，说明当前像素需要被替换为fillColor色

if (cmpColor(interiorColor, current)) {

setPixel(x, y);

// 每填一个点都进行Flush，以展示出填充的过程

glFlush();

// 分别递归泛滥填充右、左、上、下四个点

floodFill4(x+1, y, fillColor, interiorColor);

floodFill4(x-1, y, fillColor, interiorColor);

floodFill4(x, y-1, fillColor, interiorColor);

floodFill4(x, y+1, fillColor, interiorColor);

}

}

// 用以测试泛滥填充算法的回调函数

void testFloodFill(void) {

Sleep(5000);

// 用来填充的颜色，十六进制为#66CCFF

Color fill = {0.4f, 0.8f, 1.0f};

// 多边形原来的颜色，为黑色

Color origin = {0.0f, 0.0f, 0.0f};

// 绘制一个多边形用来泛滥填充

int points[] = {50, 100, 100, 130, 150, 100, 150, 50, 50, 50};

glColor3fv(origin);

glBegin(GL\_POLYGON);

for (int i = 0; i < 5; i++) {

glVertex2i(points[2 \* i], points[2 \* i + 1]);

}

glEnd();

glFlush();

printf("Polygon draw completed.\n");

// 切换至填充颜色

glColor3fv(fill);

floodFill4(100, 100, fill, origin);

glFlush();

printf("floodFill done.\n");

}

**4. 实现扫描线填充算法**

#include <GL/freeglut.h>

#include <cmath>

#include <vector>

#define WINDOW\_HEIGHT 500

#define WINDOW\_WIDTH 500

void setPixel(GLint x, GLint y) {

glBegin(GL\_POINTS);

glVertex2i(x, y);

glEnd();

}

typedef struct tEdge {

int yUpper;

float xIntersect, dxPerScan;

struct tEdge \*next;

} Edge;

typedef struct dcPt {

int x, y;

} dcPt;

// 在边表list中插入边edge，ET与AET通用

void insertEdge(Edge \*list, Edge \*edge) {

Edge \*p, \*q = list;

p = q->next;

while (p != NULL) {

// ET、AET中某一项链表记录的点的x要从左往右，将边edge插入到链表中的合适位置

// 寻找一个比p的x小，比q的x大的位置

if (edge->xIntersect < p->xIntersect) {

p = NULL;

} else {

q = p;

p = p->next;

}

}

// 在q、p之间插入edge

edge->next = q->next;

q->next = edge;

}

int yNext(int k, int cnt, dcPt \*pts) {

int j;

// 由于此处 k+1∈[1,cnt]，这段if-else相当于j=(k+1)%cnt

if ((k + 1) > (cnt - 1)) {

j = 0;

} else {

j = k + 1;

}

// 在点的数组中循环找下一个和k点不在一条水平上的点

while (pts[k].y == pts[j].y) {

if ((j + 1) > (cnt - 1)) {

j = 0;

} else {

j++;

}

}

return pts[j].y;

}

// 处理边edge（包括顶点算一次还是算两次），并将其加入边表edges

void makeEdgeRec(dcPt lower, dcPt upper, int yComp, Edge \*edge, Edge \*edges[]) {

// 计算这条边的斜率

edge->dxPerScan = (float) (upper.x - lower.x) / (upper.y - lower.y);

// 与扫描线的第一次相交交点的x，一定是下方点的x

edge->xIntersect = lower.x;

// 在边的奇偶处理中，对于顶点需要进行特殊处理

if (upper.y < yComp) {// 单调的，只算一次，将较低一段的较高点，即终点（此处即为点upper）

edge->yUpper = upper.y - 1;

} else {// 不单调的不动，即算两次

edge->yUpper = upper.y;

}

// 将处理好的边edge，即(lower->upper)，放入ET表中

insertEdge(edges[lower.y], edge);

}

// 构建ET表

void buildEdgeList(int cnt, dcPt \*pts, Edge \*edges[]) {

Edge \*edge;

dcPt v1, v2;

// yPrev记录前一个点的前一个点的y坐标，初始值显然为倒数第二个点

int i, yPrev = pts[cnt - 2].y;

// v1记录当前遍历的点的前一个点，初始值显然为倒数第一个点

v1.x = pts[cnt - 1].x, v1.y = pts[cnt - 1].y;

// 遍历各个点，同样是遍历边v1-v2

for (int i = 0; i < cnt; i++) {

// v2用以记录当前遍历点

v2 = pts[i];

// 要添加到ET中的边不能是水平的

if (v1.y != v2.y) {

edge = (Edge \*) malloc(sizeof(Edge));

// 要把y较小的点作为起点

if (v1.y < v2.y) {// 当前点的前一个点v1的y较小

makeEdgeRec(v1, v2, yNext(i, cnt, pts), edge, edges);

} else {// 当前点v2的y较小

makeEdgeRec(v2, v1, yPrev, edge, edges);

}

}

// 传递变量

yPrev = v1.y;

v1 = v2;

}

}

// 将扫描线y=scan涉及的新边加入到AET中

void buildActiveList(int scan, Edge \*active, Edge \*edges[]) {

Edge \*p, \*q;

p = edges[scan]->next;

while (p) {

q = p->next;

// 在ET中取得的新边，插入AET

insertEdge(active, p);

p = q;

}

}

// 扫描线y=scan按AET进行填充

void fillScan(int scan, Edge \*active) {

Edge \*p1, \*p2;

p1 = active->next;

// 在第奇数个与第偶数个点之间填充即可

// 即AET中1、2之间，3、4之间...进行填充

while (p1) {

p2 = p1->next;

for (int i = p1->xIntersect; i < p2->xIntersect; i++) {

setPixel((int) i, scan);

}

p1 = p2->next;

}

}

// 在链表中删除q->next

void deleteAfter(Edge \*q) {

Edge \*p = q->next;

q->next = p->next;

free(p);

}

// 在完成一条扫描线后，更新AET的操作

void updateActiveList(int scan, Edge \*active) {

Edge \*q = active, \*p = active->next;

while (p) {

if (scan >= p->yUpper) {// 下一条扫描线已经不会与该边相交，从AET中删除

p = p->next;

deleteAfter(q);

} else {// 根据边的dx信息更新下一条扫描线和该边的交点x坐标

p->xIntersect = p->xIntersect + p->dxPerScan;

q = p;

p = p->next;

}

}

}

// 在

void resortActiveList(Edge \*active) {

Edge \*q, \*p = active->next;

active->next = NULL;

while (p) {

q = p->next;

insertEdge(active, p);

p = q;

}

}

// 扫描线填充的主函数，cnt为多边形的顶点个数，pts为顶点数组

void scanFill(int cnt, dcPt \*pts) {

Edge \*edges[WINDOW\_HEIGHT], \*active;

int i, scan;

// 初始化一个空的ET

for (int i = 0; i < WINDOW\_HEIGHT; i++) {

edges[i] = (Edge \*) malloc(sizeof(Edge));

edges[i]->next = NULL;

}

// 构建ET

buildEdgeList(cnt, pts, edges);

// 初始化AET，各条扫描线之间互不干扰，只需要一条链表记录当前扫描线的AET即可

active = (Edge \*) malloc(sizeof(Edge));

active->next = NULL;

// 从下到上遍历扫描线进行填充

for (scan = 0; scan < WINDOW\_HEIGHT; scan++) {

buildActiveList(scan, active, edges);

if (active->next) {

fillScan(scan, active);

updateActiveList(scan, active);

resortActiveList(active);

}

}

}

void testScanLineFill() {

// 三种多边形用于测试，分别是复杂多边形、凹多边形、凸多边形

dcPt complex[] = {

WINDOW\_WIDTH / 4, WINDOW\_HEIGHT / 4,

3 \* WINDOW\_WIDTH / 4, 3 \* WINDOW\_HEIGHT / 4,

3 \* WINDOW\_WIDTH / 4, WINDOW\_HEIGHT / 4,

WINDOW\_WIDTH / 4, 3 \* WINDOW\_HEIGHT / 4

};

dcPt concave[] = {

WINDOW\_WIDTH / 4, WINDOW\_HEIGHT / 4,

3 \* WINDOW\_WIDTH / 4, WINDOW\_HEIGHT / 4,

3 \* WINDOW\_WIDTH / 4, 3 \* WINDOW\_HEIGHT / 4,

WINDOW\_WIDTH / 2, WINDOW\_HEIGHT / 2,

WINDOW\_WIDTH / 4, 3 \* WINDOW\_HEIGHT / 4,

};

dcPt convex[] = {

WINDOW\_WIDTH / 4, WINDOW\_HEIGHT / 4,

3 \* WINDOW\_WIDTH / 4, WINDOW\_HEIGHT / 4,

3 \* WINDOW\_WIDTH / 4, 3 \* WINDOW\_HEIGHT / 4,

WINDOW\_WIDTH / 2, 4 \* WINDOW\_HEIGHT / 5,

WINDOW\_WIDTH / 4, 3 \* WINDOW\_HEIGHT / 4,

};

glColor3f(0.4f, 0.8f, 1.0f);

// scanFill(4, complex);

scanFill(5, concave);

// scanFill(5, convex);

glFlush();

glColor3f(0.0f, 0.0f, 0.0f);

glBegin(GL\_LINES);

for (int i = 0; i < 5; i++) {

glVertex2i(concave[i].x, concave[i].y);

glVertex2i(concave[(i + 1) % 5].x, concave[(i + 1) % 5].y);

}

glEnd();

glFlush();

}

int main(int argc, char \*argv[]) {

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_RGB | GLUT\_SINGLE);

glutInitWindowPosition(100, 100);

glutInitWindowSize(WINDOW\_WIDTH, WINDOW\_HEIGHT);

glutCreateWindow("scan-line Fill algorithm");

glClearColor(1.0, 1.0, 1.0, 0.0);

gluOrtho2D(0, WINDOW\_WIDTH, 0, WINDOW\_HEIGHT);

glMatrixMode(GL\_PROJECTION);

glClear(GL\_COLOR\_BUFFER\_BIT);

glFlush();

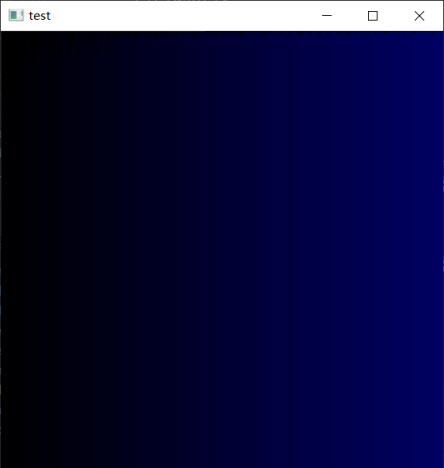
glutDisplayFunc(testScanLineFill);

glutMainLoop();

}

**四. 实验结果**

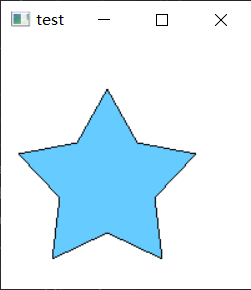
**1. 颜色的设置**



**2. 实现边界填充算法**



双击该图标可以打开算法运行演示视频



**3. 实现泛滥填充算法**



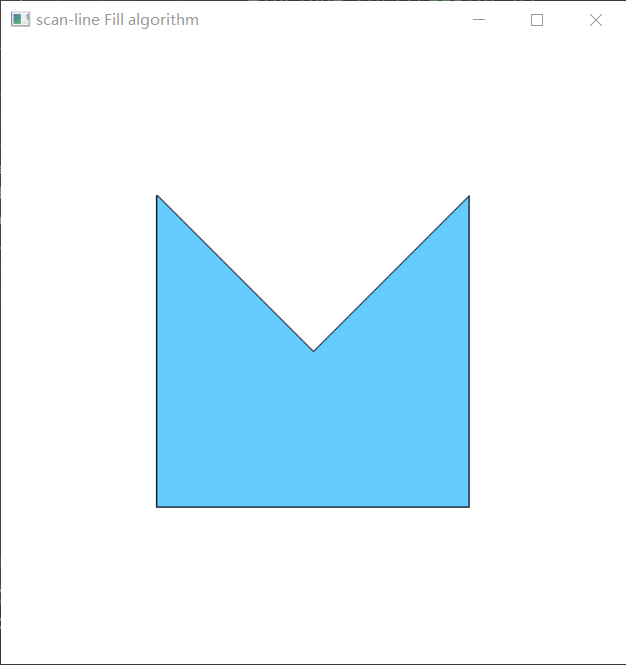
双击该图标可以打开算法运行演示视频

**4. 实现扫描线填充算法**



双击该图标可以打开算法运行演示视频

由于扫描线填充算法较为复杂，将代码单独写在一个文件中进行调试与运行。



**五. 实验分析**

1. 对边界填充算法的优化。

经过老师上课的优化提示，从getPixel的优化入手进行优化。先对未改进过的边界填充算法进行测试，作为基准值。为了便于比较，保留填充时一边画点一边flush的操作。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 平均值 |
| 不优化 | 8788ms | 8880ms | 8900ms | 8813ms | 8796ms | 8835ms |

getPixel优化

提前使用一个数组作为全局变量，其中记录与实际帧缓存中的内容相同，在帧缓存中画点时，也同时在这个数组中改变记录值。进行五次填充，取平均值作为最终结果。代码中的加红为与原来的不同的部分

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 平均值 |
| getPixel优化 | 2487ms | 2326ms | 2466ms | 2293ms | 2417ms | 2398ms |

#define WINDOW\_WIDTH 200

#define WINDOW\_HEIGHT 200

typedef float Color[3];

Color mask[WINDOW\_WIDTH][WINDOW\_HEIGHT];

void setPixel(GLint x, GLint y, Color color) {

glBegin(GL\_POINTS);

glVertex2i(x, y);

glEnd();

mask[y][x][0] = color[0], mask[y][x][1] = color[1], mask[y][x][2] = color[2];

}

void getPixel(GLint x, GLint y, Color c) {

c[0] = mask[y][x][0], c[1] = mask[y][x][1], c[2] = mask[y][x][2];

}

void testBoundaryFill(void) {

// 用来填充的颜色，十六进制为#66CCFF

Color fill = {0.4f, 0.8f, 1.0f};

// 作为边界颜色，为黑色

Color border = {0.0f, 0.0f, 0.0f};

// 先画好待填充的区域的边界（一个五角星）

int points[] = {85, 160, 109, 117, 156, 108, 123, 73, 129, 24, 85, 45, 41, 24, 47, 73, 14, 108, 61, 117};

glColor3fv(border);

// 五角星由10个点组成，使用GL\_LINE\_LOOP画首位相接的线

glBegin(GL\_LINE\_LOOP);

for (int i = 0; i < 10; i++) {

glVertex2i(points[2 \* i], points[2 \* i + 1]);

}

glEnd();

glFlush();

printf("boundary draw completed.\n");

// 画好边界后，将帧缓存放入mask中

glReadPixels(0, 0, WINDOW\_WIDTH, WINDOW\_HEIGHT, GL\_RGB, GL\_FLOAT, mask);

// 设置绘图颜色为fill

glColor3fv(fill);

clock\_t end, start = clock();

// 选择的起始点位三角形的中心点

boundaryFill4(85, 85, fill, border);

end = clock();

glFlush();

printf("boundary fill done.\n");

printf("use %.2fms", (float) (end-start)/CLOCKS\_PER\_SEC\*1000);

}

2. 用同样的方法对泛滥填充算法进行优化，测试后得到如下结果：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 平均值 |
| 未优化 | 6852ms | 7113ms | 7519ms | 7481ms | 7461ms | 7285ms |
| getPixel优化 | 1952ms | 2110ms | 1992ms | 1376ms | 2141ms | 1914ms |

可以看到性能得到显著的提升。

3. 通用扫描线填充算法

通用扫描线填充算法，没有使用递归，也没有使用getPixel这样获取已绘图形目标点的颜色值，并且适用于凸多边形、凹多边形、复杂多边形所有三种多边形，是一种真正通用的算法。

其核心思想是根据每条扫描线与多边形边的交点的“奇偶”来判断是否要在某两个交点之间进行颜色填充，关键在于对交点是奇是偶的判别。在具体的算法实现中，通过遍历多边形的顶点来构建ET有序边表与AET活动边表，通过当前点、前一个点、前一个点的前一个点之间的单调性来判断该点是奇是偶。

另外，还使用dx增量式提前计算y+1到扫描线时交点的x坐标，进一步提高了算法的效率。