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： 实验二 二维基本图元的生成

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 6 日

**一. 实验目的**

1. 进一步掌握OpenGL的绘制原理；

2. 直线生成算法的理解与实现；

3. 圆生成算法的理解与实现；

4. 椭圆生成算法的理解与实现。

**二. 实验内容**

仔细阅读实验指导书，基于OpenGL完成以下内容：

（1）实现DDA直线生成算法；

（2）实现Bresenham直线生成算法；

（3）实现中点圆绘制算法；

（4）实现中点椭圆绘制算法；（选做）

**三. 实验代码**

**1. 实现DDA直线生成算法**

// 标准DDA画线算法

void lineDDA(int x0, int y0, int xEnd, int yEnd) {

int dx = xEnd - x0, dy = yEnd - y0, steps, k;

float xIncrement, yIncrement, x = x0, y = y0;

// 斜率绝对值小于1，沿x轴递进，每一列画一个点

if (fabs(dx) > fabs(dy))

steps = fabs(dx);

else// 斜率绝对值大于1，沿y轴递进，每一行画一个点

steps = fabs(dy);

// 计算增量

xIncrement = float(dx) / steps;

yIncrement = float(dy) / steps;

// 先画好第一个点

setPixel(round(x), round(y));

// 逐点绘画，共steps步，加上第一个点刚好steps+1步

for (k = 0; k < steps; k++) {

x += xIncrement;

y += yIncrement;

setPixel(round(x), round(y));

}

}

**2. 实现Bresenham直线生成算法**

// bresenham算法画线

void bresenham(int x0, int y0, int x1, int y1) {

int dy = y1 - y0, dx = x1 - x0;

int stepx, stepy;

// 根据Δx和Δy的符号判断决策参数增量式()2Δy()2Δx(y(k+1)-y(k))两项前的正负

// 并确定x与y的递进方向

if (dy < 0) {

dy = -dy, stepy = -1;

} else {

stepy = 1;

}

if (dx < 0) {

dx = -dx, stepx = -1;

} else {

stepx = 1;

}

// 通过位移乘2，更快

dy <<= 1;

dx <<= 1;

// 先画起点

setPixel(x0, y0);

// 不使用round取整，使用fraction记录每次加法的小数部分，有进位时才+1

if (dx > dy) {// 斜率绝对值小于1

int fraction = dy - (dx >> 1);

while (x0 != x1) {

// 小数部分的进位操作

if (fraction >= 0) {

y0 += stepy;

fraction -= dx;

}

// x步进

x0 += stepx;

// 决策参数加上递增量

fraction += dy;

setPixel(x0, y0);

}

} else {// 斜率绝对值大于1，与上面类似

int fraction = dx - (dy >> 1);

while (y0 != y1) {

if (fraction >= 0) {

x0 += stepx;

fraction -= dy;

}

y0 += stepy;

fraction += dx;

setPixel(x0, y0);

}

}

}

**3. 实现中点圆算法**

// 依据圆的对称性，同时可以画出8个八分之一圆

void circlePlotPoints(scrPt circCtr, scrPt circPt) {

setPixel(circCtr.x + circPt.x, circCtr.y + circPt.y);

setPixel(circCtr.x - circPt.x, circCtr.y + circPt.y);

setPixel(circCtr.x + circPt.x, circCtr.y - circPt.y);

setPixel(circCtr.x - circPt.x, circCtr.y - circPt.y);

setPixel(circCtr.x + circPt.y, circCtr.y + circPt.x);

setPixel(circCtr.x - circPt.y, circCtr.y + circPt.x);

setPixel(circCtr.x + circPt.y, circCtr.y - circPt.x);

setPixel(circCtr.x - circPt.y, circCtr.y - circPt.x);

}

// 中点圆算法，以(xc, yc)为圆心，radius为半径画圆

// 从画第二象限（共分为8个象限）的八分之一圆出发，根据对称性画出其他部分

void circleMidpoint(GLint xc, GLint yc, GLint radius) {

scrPt circPt, tmp;

GLint p = 1 - radius;

// circPt起始于圆的上边界点

circPt.x = 0, circPt.y = radius;

tmp.x = xc, tmp.y = yc;

circlePlotPoints(tmp, circPt);

// 只画八分之一圆，即画到y=x直线处即可

while (circPt.x < circPt.y) {

// 沿x方向递进

(circPt.x)++;

// 根据决策参数p确定要画的点

if (p < 0) {// p<0，画(x(k+1), y(k))，y不用变

// 决策变量加上增量式

p += 2 \* circPt.x + 1;

} else {// p>0，画(x(k+1), y(k)-1)

(circPt.y)--;

// 决策变量加上增量式

p += 2 \* (circPt.x - circPt.y) + 1;

}

//画出k+1号点

circlePlotPoints(tmp, circPt);

}

}

**4. 实现中点椭圆算法**

// 根据椭圆的对称性，可以一次画4个点

void ellipsePlotPoints(int xc, int yc, int x, int y) {

setPixel(xc + x, yc + y);

setPixel(xc - x, yc + y);

setPixel(xc + x, yc - y);

setPixel(xc - x, yc - y);

}

// 使用与中点圆类似的思想，但由于对称性不同，需要画四分之一个

// 右上四分之一个椭圆由于斜率不同，需要递进的方向也不同，因此分为两部分来画

void ellipseMidpoint(int xc, int yc, int rx, int ry) {

int rx2 = rx \* rx;

int ry2 = ry \* ry;

int twoRx2 = 2 \* rx2;

int twoRy2 = 2 \* ry2;

int p;

// 与中点圆相同，从上方的点开始画，从右上四分之一个出发，根据对称性画出其余部分

int x = 0, y = ry;

// 决策变量中要用到的项

int px = 0, py = twoRx2 \* y;

ellipsePlotPoints(xc, yc, x, y);

// 计算决策变量的初始值

p = round(ry2 - (ry2 \* ry) + (0.25 \* rx2));

// 画“区域1”即靠近y轴的半个四分之一椭圆

while (px < py) {

// 沿x轴递增

x++;

px += twoRy2;

if (p < 0) {// p<0，画(x(k+1), y(k))，y不用变

p += ry2 + px;

} else {// p>0，画(x(k+1), y(k)-1)

y--;

py -= twoRx2;

p += ry2 + px - py;

}

ellipsePlotPoints(xc, yc, x, y);

}

// 根据“区域1”中的最后一个点来计算“区域2”即靠近x轴部分的半个四分之一椭圆的决策参数初始值

p = round(ry2 \* (x + 0.5) \* (x + 0.5) + rx2 \* (y - 1) \* (y - 1) - rx2 \* ry2);

// 与上面类似

while (y > 0) {

y--;

py -= twoRx2;

if (p > 0) {

p += rx2 - py;

} else {

x++;

px += twoRy2;

p += rx2 - py + px;

}

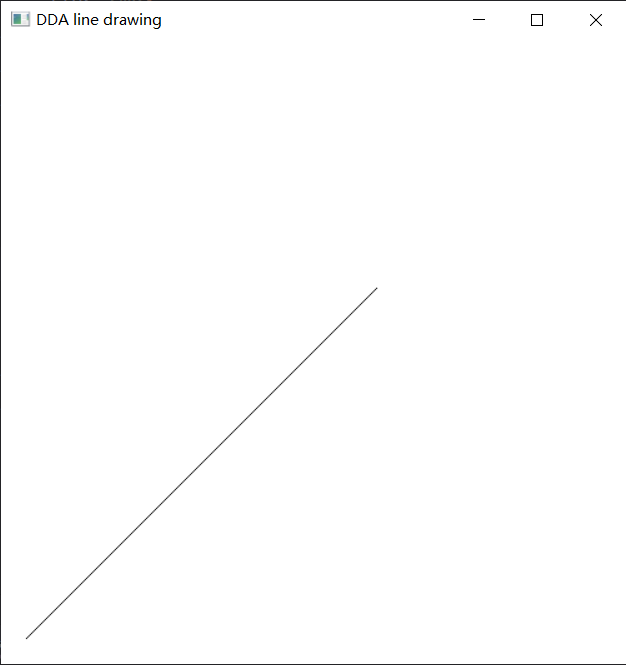
ellipsePlotPoints(xc, yc, x, y);

}

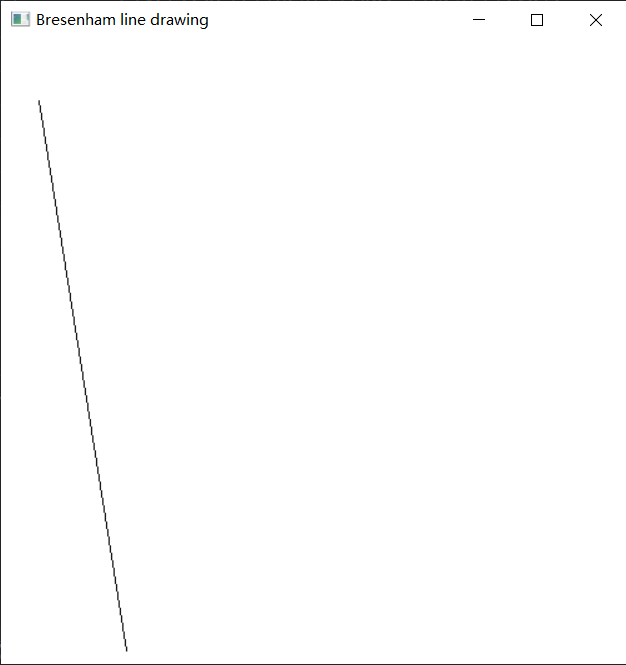
}

**四. 实验结果**

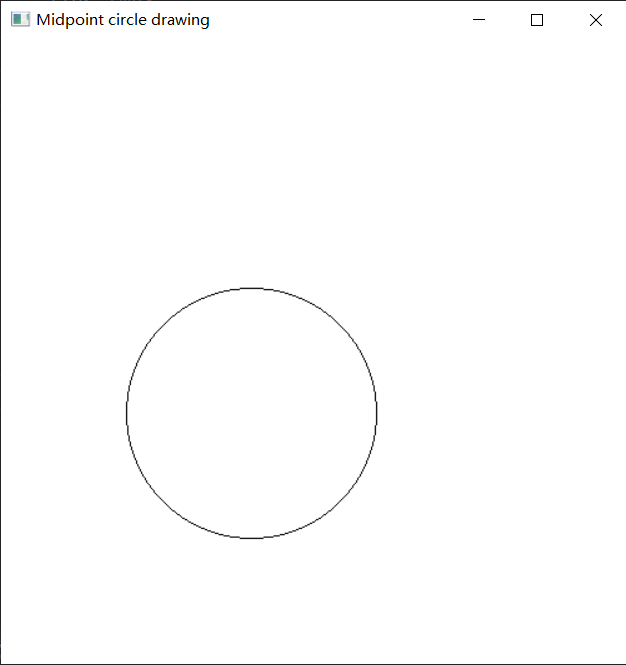
**1. 实现DDA直线生成算法**



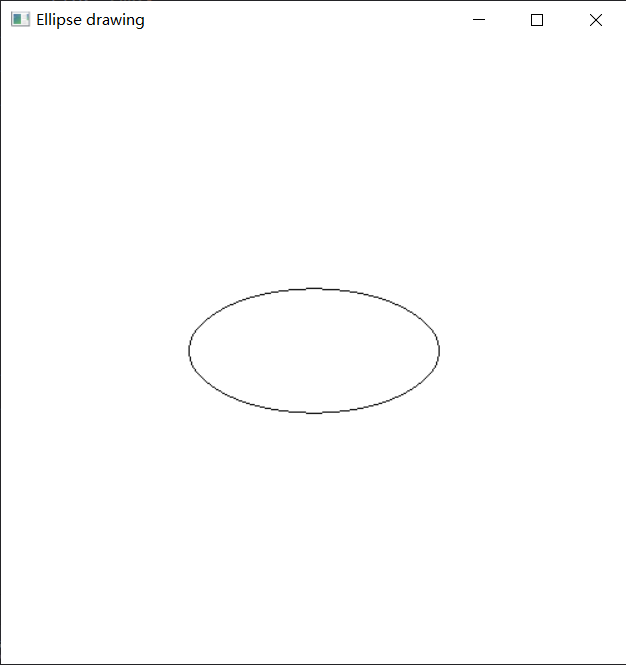
**2. 实现Bresenham直线生成算法**



**3. 实现中点圆算法**



**4. 实现中点椭圆算法**



**五. 实验分析**

**1. 出现的问题及解决方法**

一开始，绘画的图形始终不显示，在反复尝试之后，发现是没有调用glFlush函数。该函数可以让强制让OpenGL在一段限定的时间内执行所有需要执行的GL函数并且将缓冲区的内容加载到屏幕上来。

另外，在尝试采用Bresenham算法中不进行四舍五入及取整运算优化DDA算法时，一开始的程序无法正确处理斜率小于0的情况，实际运行效果无论如何都是一条和起点y值相等的水平直线。据此可以排查出是y方向上的递增量没有正确计算的原因。最终写出来的DDA improved算法如下

void lineDDA\_improved(int x0, int y0, int x1, int y1) {

int dx = x1 - x0, dy = y1 - y0;

int xIncrement, yIncrement;

int x = x0, y = y0;

float k;

float res = 0.5f;

if (fabs(dx) > fabs(dy)) {

k = (float) dy / dx;

xIncrement = dx > 0 ? 1 : -1;

yIncrement = dy > 0 ? 1 : -1;

res = xIncrement < 0 ? -0.5f : 0.5f;

setPixel(x, y);

while (x != x1) {

x += xIncrement;

res += k;

if (res >= 1) {

res--;

y += yIncrement;

} else if (res <= -1) {

res++;

y += yIncrement;

}

setPixel(x, y);

}

} else {

k = (float) dx / dy;

yIncrement = dy > 0 ? 1 : -1;

xIncrement = dx > 0 ? 1 : -1;

res = yIncrement < 0 ? -0.5f : 0.5f;

setPixel(x, y);

while (y != y1) {

y += yIncrement;

res += k;

if (res >= 1) {

res--;

x += xIncrement;

} else if (res <= -1) {

res++;

x += xIncrement;

}

setPixel(x, y);

}

}

}

**2. 算法性能分析**

**a. 较为理想的测试**

先放上实际运行的时间测试，先生成50000对随机点作为待画的直线两端点坐标，使用这50000对随机点测试各算法并计时。

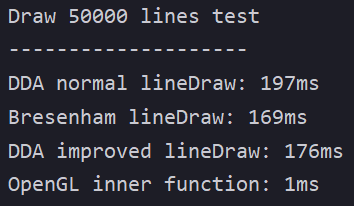
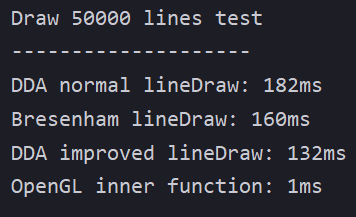
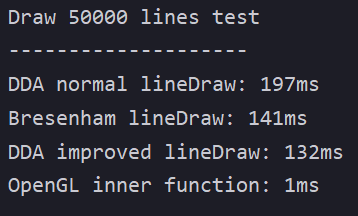
为了尽量避免误差，从而体现算法本身的效率差异，对测试做了如下设计

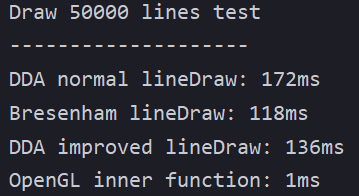
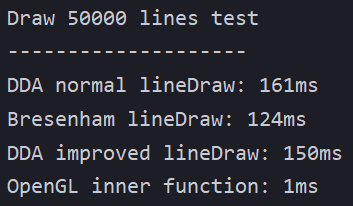
（1）将setPixel函数中的glBegin与glEnd移到测试函数中，除了OpenGL内置的画线算法需要独立使用glBegin(GL\_LINES)外，只进行一次glBegin与glEnd

（2）考虑各算法调用的先后顺序的差异，调用glFlush将Buffer中内容加载到屏幕上时调用的数据硬件来源可能不同，因此不调用glFlush将Buffer中的内容加载到屏幕上

（3）避免随机误差的影响，进行5次测试，取平均值作为最终测试结果

（4）每次测试后都进行内存清理

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 平均时间 |
| 普通DDA | 197ms | 182ms | 197ms | 172ms | 161ms | 182ms |
| Bresenham | 169ms | 160ms | 141ms | 118ms | 124ms | 142ms |
| DDA改进 | 176ms | 132ms | 132ms | 136ms | 150ms | 145ms |
| OpenGL自带 | 1ms | 1ms | 1ms | 1ms | 1ms | 1ms |

可见，Bresenham算法的效率确实高于普通的DDA算法，而避免了四舍五入和取整运算的DDA改进算法也确实优于普通的DDA算法。

以下为用于测试的回调绘制函数

void lineDraw\_test(void) {

//获取随机数发生器以生成随机种子

std::random\_device rd;

//生成种子

std::mt19937 gen(rd());

//随机数生成器，生成[0,500]中的随机整数

std::uniform\_int\_distribution<> dis(0, 500);

//生成50000对随机点，作为50000条线的起点与终点

for (int i = 0; i < 50000; i++) {

lines[i] = line(dis(gen), dis(gen), dis(gen), dis(gen));

}

glBegin(GL\_POINTS);

//普通DDA算法画线

clock\_t start, finish;

glClear(GL\_COLOR\_BUFFER\_BIT);

start = clock();

for (int i = 0; i < 50000; i++) {

lineDDA(lines[i].x0, lines[i].y0, lines[i].x1, lines[i].y1);

}

finish = clock();

glFlush();

std::cout << "DDA normal lineDraw: " << (double) (finish - start) / CLOCKS\_PER\_SEC \* 1000 << "ms" << std::endl;

//bresenham算法画线

glClear(GL\_COLOR\_BUFFER\_BIT);

start = clock();

for (int i = 0; i < 50000; i++) {

bresenham(lines[i].x0, lines[i].y0, lines[i].x1, lines[i].y1);

}

finish = clock();

std::cout << "Bresenham lineDraw: " << (double) (finish - start) / CLOCKS\_PER\_SEC \* 1000 << "ms" << std::endl;

//改进的DDA算法画线

start = clock();

for (int i = 0; i < 50000; i++) {

lineDDA\_improved(lines[i].x0, lines[i].y0, lines[i].x1, lines[i].y1);

}

finish = clock();

std::cout << "DDA improved lineDraw: " << (double) (finish - start) / CLOCKS\_PER\_SEC \* 1000 << "ms" << std::endl;

glEnd();

//OpenGL自带画线函数画线

glClear(GL\_COLOR\_BUFFER\_BIT);

glBegin(GL\_LINES);

start = clock();

for (int i = 0; i < 50000; i++) {

glVertex2i(lines[i].x0, lines[i].y0);

glVertex2i(lines[i].x1, lines[i].y1);

}

finish = clock();

glEnd();

glFlush();

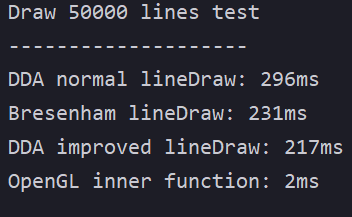
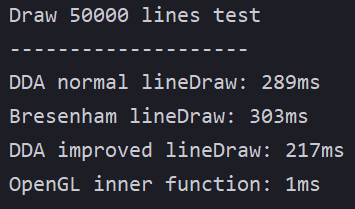
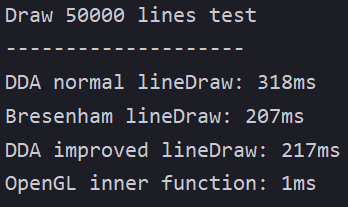
std::cout << "OpenGL inner function: " << (double) (finish - start) / CLOCKS\_PER\_SEC \* 1000 << "ms" << std::endl;

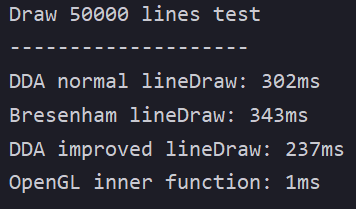
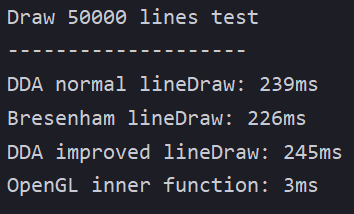
}

**b. 较贴近实际使用的测试**

由于在实际使用画图的时候，不可能只在Buffer中画好图像而不加载到屏幕上，此处追加一个测试，即对每个测试单独进行glBegin与glEnd，并进行glFlush，并将这些时间都进行计时（下面测试函数代码中标红的部分与上面不同）。

同样进行5次测试并取平均值

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 平均时间 |
| 普通DDA | 296ms | 289ms | 318ms | 302ms | 239ms | 289ms |
| Bresenham | 231ms | 303ms | 207ms | 343ms | 226ms | 262ms |
| DDA改进 | 217ms | 217ms | 217ms | 237ms | 245ms | 227ms |
| OpenGL自带 | 2ms | 1ms | 1ms | 1ms | 3ms | 1.6ms |

从上面的测试中可见，除了在第2组合第4组中，Bresenham算法甚至没普通DDA算法快，但差距均在40ms内，可能是由于硬件状态略有不同导致的实验随机误差，也可能是CPU自身硬件层面的优化导致的。但总体来说，效率仍是Bresenham算法高。

以下为测试函数代码

void lineDraw\_test(void) {

//获取随机数发生器以生成随机种子

std::random\_device rd;

//生成种子

std::mt19937 gen(rd());

//随机数生成器，生成[0,500]中的随机整数

std::uniform\_int\_distribution<> dis(0, 500);

//生成50000对随机点，作为50000条线的起点与终点

for (int i = 0; i < 50000; i++) {

lines[i] = line(dis(gen), dis(gen), dis(gen), dis(gen));

}

clock\_t start, finish;

//普通DDA算法画线

glClear(GL\_COLOR\_BUFFER\_BIT);

glBegin(GL\_POINTS);

start = clock();

for (int i = 0; i < 50000; i++) {

lineDDA(lines[i].x0, lines[i].y0, lines[i].x1, lines[i].y1);

}

glFlush();

glEnd();

finish = clock();

std::cout << "DDA normal lineDraw: " << (double) (finish - start) / CLOCKS\_PER\_SEC \* 1000 << "ms" << std::endl;

//bresenham算法画线

glClear(GL\_COLOR\_BUFFER\_BIT);

glBegin(GL\_POINTS);

start = clock();

for (int i = 0; i < 50000; i++) {

bresenham(lines[i].x0, lines[i].y0, lines[i].x1, lines[i].y1);

}

glEnd();

glFlush();

finish = clock();

std::cout << "Bresenham lineDraw: " << (double) (finish - start) / CLOCKS\_PER\_SEC \* 1000 << "ms" << std::endl;

//改进的DDA算法画线

glClear(GL\_COLOR\_BUFFER\_BIT);

glBegin(GL\_POINTS);

start = clock();

for (int i = 0; i < 50000; i++) {

lineDDA\_improved(lines[i].x0, lines[i].y0, lines[i].x1, lines[i].y1);

}

glEnd();

glFlush();

finish = clock();

std::cout << "DDA improved lineDraw: " << (double) (finish - start) / CLOCKS\_PER\_SEC \* 1000 << "ms" << std::endl;

//OpenGL自带画线函数画线

glClear(GL\_COLOR\_BUFFER\_BIT);

glBegin(GL\_LINES);

start = clock();

for (int i = 0; i < 50000; i++) {

glVertex2i(lines[i].x0, lines[i].y0);

glVertex2i(lines[i].x1, lines[i].y1);

}

glEnd();

glFlush();

finish = clock();

std::cout << "OpenGL inner function: " << (double) (finish - start) / CLOCKS\_PER\_SEC \* 1000 << "ms" << std::endl;

}