山东大学　计算机科学与技术　学院

计算机图形学 课程实验报告

学号：201700130011	姓名：刘建东	班级：17 级菁英班

实验题目：图元的生成

实验内容：

1. 直线生成，使用 Bresenham 方法绘制直线
2. 椭圆和圆的生成，使用 Bresenham 方法绘制
3. 区域填充，使用 X 扫描线算法实现填充

实验过程：

一、直线生成：

采用了 Bresenham 方法绘制直线的方法。Bresenham 算法的思想是通过各行、各列像素中心构造一组虚拟网格线，按照直线起点到终点的顺序，计算直线与各垂直网格线的交点，然后根据误差项的符号确定该列像素中与此交点最近的像素。

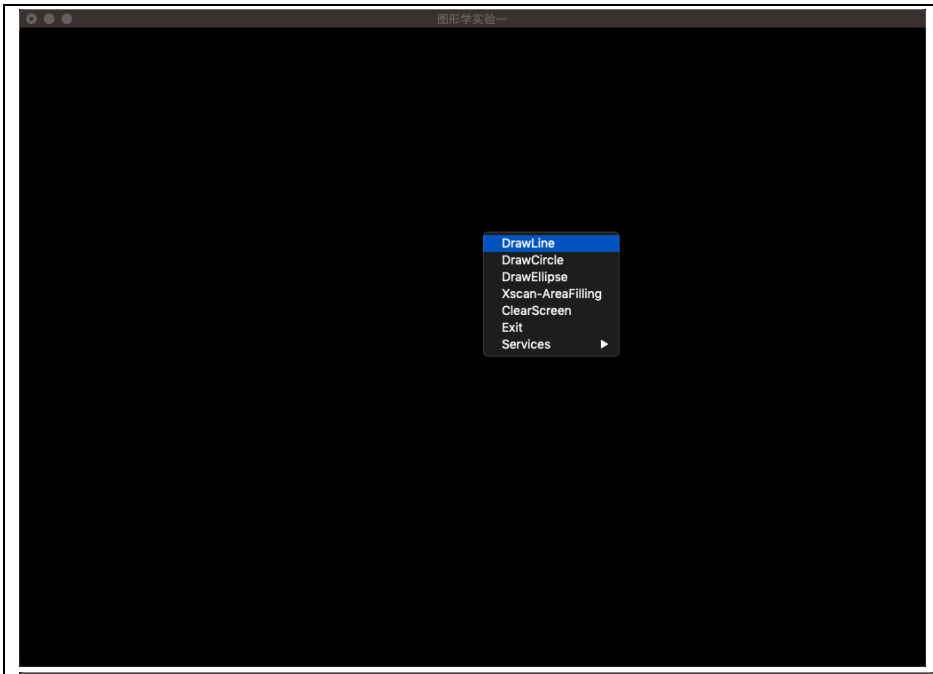
算法步骤：

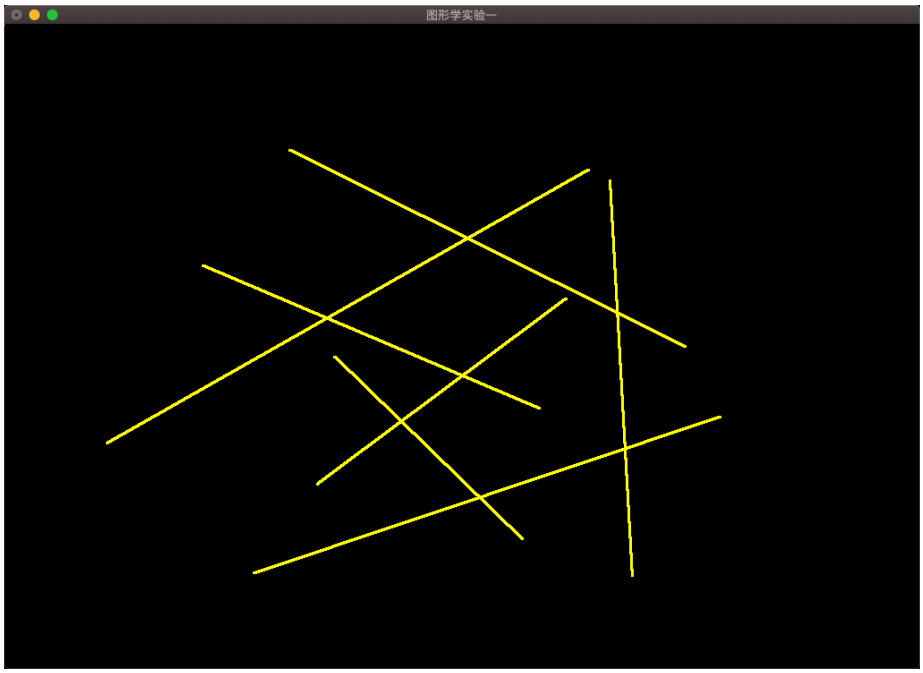
1. 输入直线的两端点 P0(x0, y0)和 P1(x1, y1)
2. 计算初始值△x、△y、e=-△x、x=x0、y=y0
3. 绘制点(x, y)
4. e 更新为 e+2△y，判断 e 的符号。若 e>0，则(x, y)更新为(x+1, y+1)，同时将 e 更新为 e-2△x，否则(x, y)更新为(x+1, y)
5. 当直线没有画完时，重复步骤 3 和 4。否则结束。

具体代码实现如下，对于斜率需要分类讨论一下。

```
void BresenhamLine(int x1,int y1,int x2,int y2){
    int x = x1, y = y1, dx = abs(x2-x1), dy = abs(y2-y1), s1 = 1, s2 = 1, e, flag = 0;
    if(x1 >= x2) s1 = -1;
    if(y1 >= y2) s2 = -1;
    if(dy > dx) {swap(dx,dy); flag = 1;}
    e = -dx;
    int DX = 2*dx, DY = 2*dy;
    for(int i = 1; i <= dx; i++){
        setPixel(x, y);
        if(e >= 0){
            if(!flag) y += s2;
            else x += s1;
            e = e-DX;
        }
        if(!flag) x += s1;
        else y += s2;
        e = e+DY;
    }
}
```

实验功能如下。拖拽鼠标画取直线。





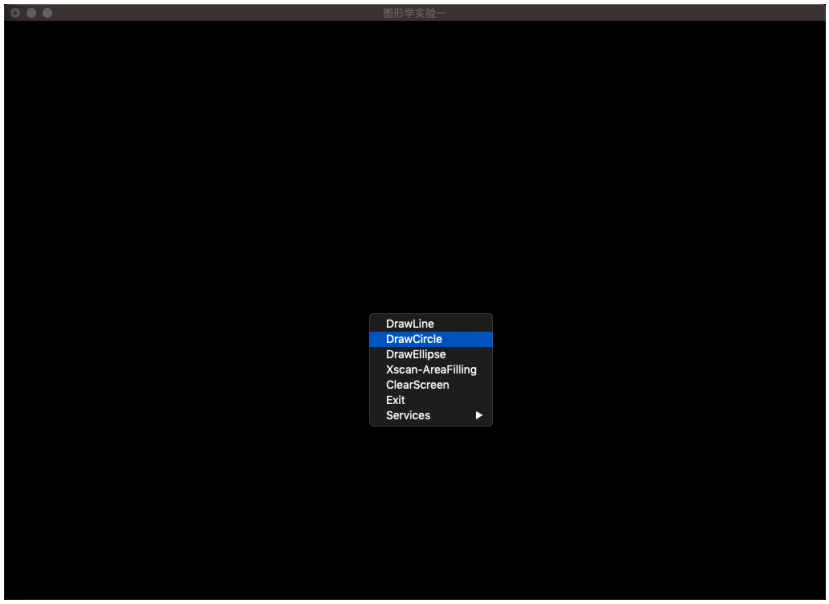
二、圆的生成：

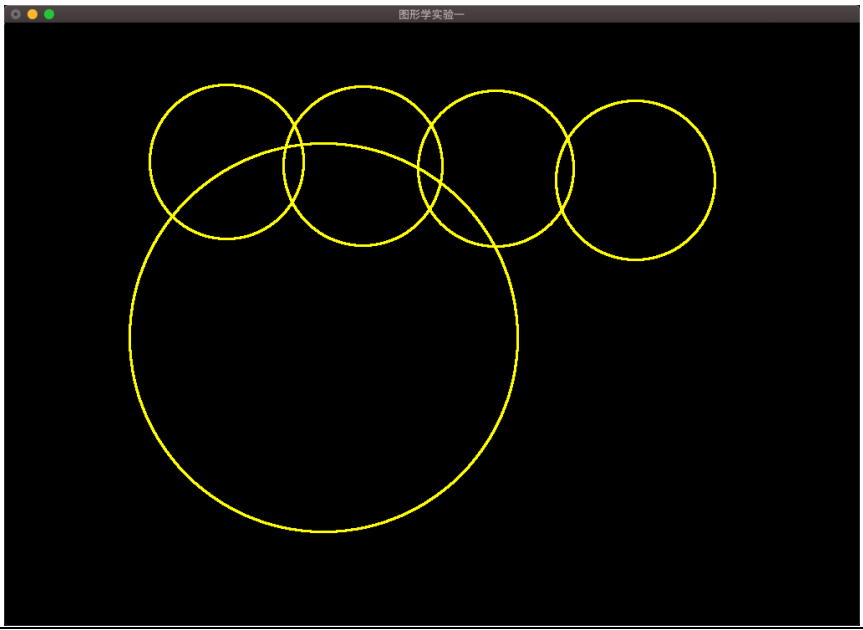
采用了八分法画圆的思想，将圆分为了 8 个部分，对称画圆。实验中采取了两点确定一个圆的方法，即第一个点为圆心，第二个点为圆弧上一点，两点距离即为圆的半径。知道圆心和半径即可确定一个圆。

具体代码如下，由于圆具有极高对称性因此不需要分类讨论，直接 8 分法对称画圆即可。

```
void Bresenham_Circle(int xc,int yc,int r){
    int x, y, d;
    x = 0; y = r; d = 5 - 4 * r;
    setPixel(x+xc, y+yc);
    while(x < y)
    {
        if(d < 0) d = d + 8 * x + 12;
        else {d = d + 8 * ( x - y ) + 20; y--;}
        x++;
        setPixel(x+xc,y+yc); setPixel(y+xc,x+yc);
        setPixel(y+xc,-x+yc); setPixel(x+xc,-y+yc);
        setPixel(-x+xc,-y+yc); setPixel(-y+xc,-x+yc);
        setPixel(-x+xc,y+yc); setPixel(-y+xc,x+yc);
    }
}
```

实验功能如下，拖拽鼠标即可改变圆的半径。





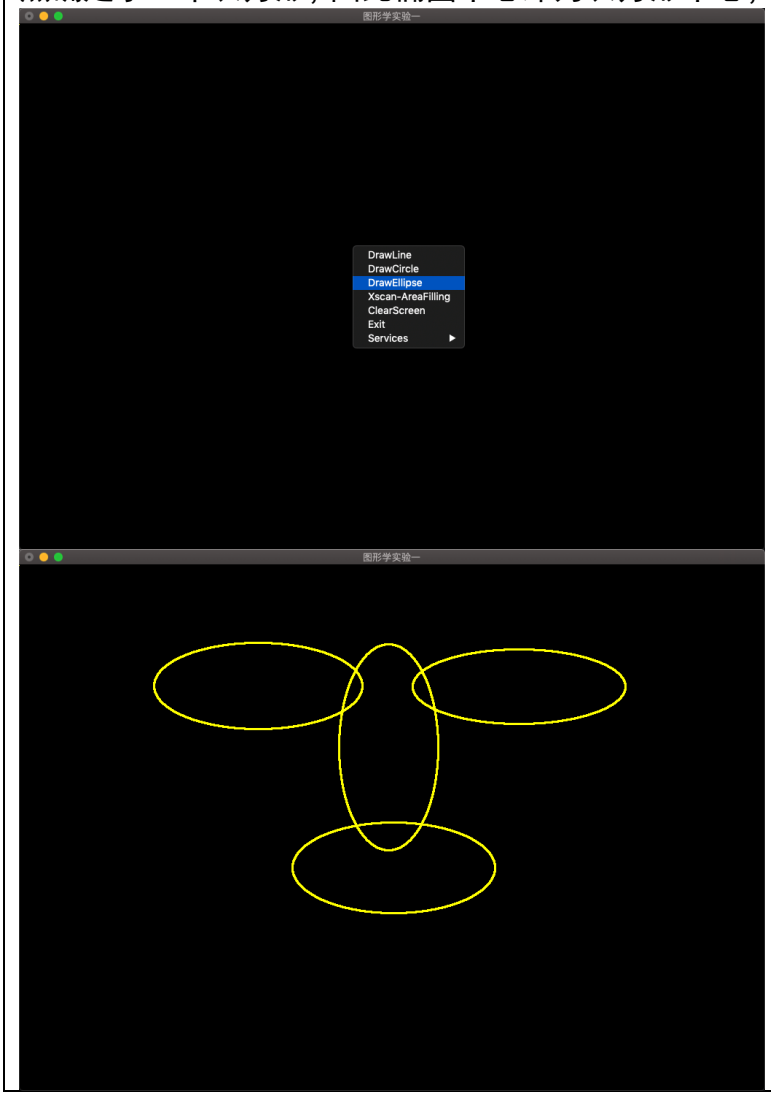
三、椭圆的生成：

基本思想与画圆区别不大，由八分法对称画圆转为了四分法对称画椭圆。但是要对椭圆切线斜率小于-1 和大于-1 的情况进行分类讨论，因此切线斜率小于-1 的时候 x 变化比较缓慢，但是斜率大于-1 时，x 变化非常迅速。因此需要进行分类讨论画椭圆。

具体代码如下，需要对斜率进行分类讨论。

```
void Bresenham_Ellipse(int xc,int yc,int a,int b){
    int x = 0, y = b, d1 = 4*b*b+a*a*(-4*b+1), d2; //一开始*4倍，浮点转整数
    setPixel(xc+x,yc+y); setPixel(xc+x, yc-y); setPixel(xc-x, yc+y); setPixel(xc-x, yc-y);
    while(2*b*b*(x+1) < a*a*(2*y-1)){ //先处理斜率 > -1的情况
        if(d1 < 0) d1 += 4*b*b*(2*x+3);
        else {d1 += 4*b*b*(2*x+3)+4*a*a*(-2*y+2); y--;}
        x++;
        setPixel(xc+x,yc+y); setPixel(xc+x, yc-y); setPixel(xc-x, yc+y); setPixel(xc-x, yc-y);
    }
    d2 = b*b*(2*x+1)*(2*x+1)+4*a*a*(y-1)*(y-1)-4*a*a*b*b;
    while(y > 0){ //再处理斜率 < -1的情况
        if(d2 < 0) {d2 += 4*b*b*(2*x+2)+4*a*a*(-2*y+3); x++;}
        else d2 += 4*a*a*(-2*y+3);
        y--;
        setPixel(xc+x,yc+y); setPixel(xc+x, yc-y); setPixel(xc-x, yc+y); setPixel(xc-x, yc-y);
    }
}
```

实验功能如下，拖拽鼠标即可画椭圆。拖拽时两点确定了一个椭圆，因为两点确定了一个长方形，因此椭圆中心即为长方形中心，椭圆长轴、短轴也可得到。



四、X-扫描线算法：

该算法基本思想为按扫描线顺序，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。

算法过程：

1. 求交：计算扫描线与多边形各边的交点
2. 排序：把所有交点按递增顺序进行排序
3. 交点配对：第一个交点与第二个，第三个与第四个
4. 区间填色：把这些相交区间内的像素置成不同于背景色的填充色

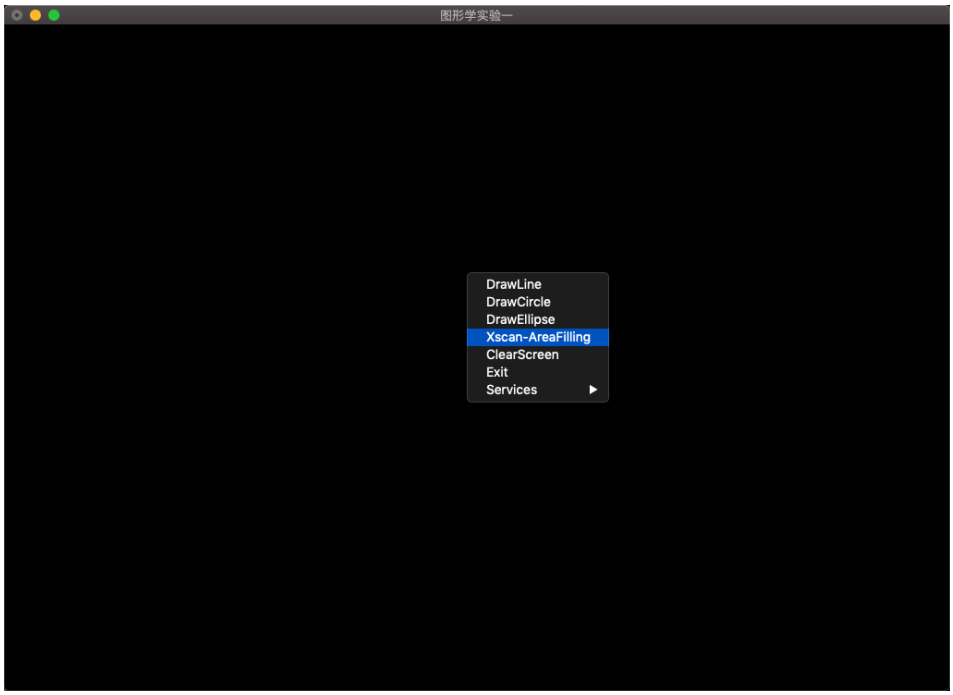
交点取舍问题：

1. 若共享顶点的两条边分别落在扫描线的两边，交点只算一个
2. 若共享顶点的两条边在扫描线的同一边，这时交点作为 0 个或 2 个。检查共享顶点的另外两个端点的 y 值，按这两个 y 值中大于交点 y 值的个数来决定交点数。

具体代码如下，需要对交点进行判断。

```
void Xscan_Algorithm(){
    int MaxY = 0, MinY = 1e5, n = (int)Point.size();
    for(int i = 0; i < n; i++){
        MaxY = max(MaxY, (int)Point[i].second);
        MinY = min(MinY, (int)Point[i].second);
    }
    vector<Node> NET[2048];
    for(int i = MinY; i <= MaxY; i++) NET[i].clear();
    list<Node> AET; AET.clear();
    for(int i = 0; i < n; i++){
        if(Point[i].second == Point[(i+1)%n].second) continue;
        pair<int,int> tmp; db b = 0.0;
        if(Point[i].second > Point[(i+1)%n].second) tmp = Point[(i+1)%n];
        else tmp = Point[i];
        if(Point[i].first != Point[(i+1)%n].first)
            b = tmp.second-((db)(Point[i].second-Point[(i+1)%n].second)*(db)(tmp.first)/(db)(Point[i].first-Point[(i+1)%n].first));
        Node thp = {(db)tmp.first,(db)(Point[i].first-Point[(i+1)%n].first)/(db)(Point[i].second-Point[(i+1)%n].second),b,max(Point[i].second,Point[(i+1)%n].second)};
        NET[min(Point[i].second,Point[(i+1)%n].second)].push_back(thp);
    }
    for(int i = MinY; i <= MaxY; i++){
        for(int j = 0; j < NET[i].size(); j++) AET.push_back(NET[i][j]);
        AET.sort(); list<Node>::iterator it = AET.begin();
        int flag = 0, flag2 = 0, flag3 = 0, ct = 0, yy[4];
        db xx[4];
        while(it != AET.end()){
            xx[++ct] = (*it).x; yy[ct] = (*it).ymax; it++;
            if(ct == 2 && yy[1] == yy[2] && yy[1] == i && (sign(xx[2]-xx[1]) == 0)) {ct = 0; flag2++;}
            if(ct == 2 && yy[1] > i && yy[2] > i && (sign(xx[2]-xx[1]) == 0)) {flag3++;}
            if(ct == 2 && (sign(xx[2]-xx[1]) == 0))
                if((yy[1] > i && yy[2] == i) || (yy[1] == i && yy[2] > i)) {flag++; ct = 1; xx[1] = xx[2]; yy[1] = yy[2];}
            if(ct == 2){
                int jud = 0;
                if(it != AET.end()){
                    jud = 1; xx[3] = (*it).x; yy[3] = (*it).ymax; it++;
                    if((sign(xx[3]-xx[2]) == 0)){
                        if((yy[3] > i && yy[2] == i) || (yy[2] > i && yy[3] == i)){ct = 2; jud = 0; flag++;}
                        else if(yy[2] == yy[3] && yy[2] == i){flag2++; ct = 1; continue;}
                        else if(yy[2] > i && yy[3] > i) flag3++;
                    }
                }
                for(int k = ceil(xx[1]); k <= floor(xx[2]); k++){
                    if(k < 0) continue;
                    setPixel(k, i);
                }
                if(jud) {xx[1] = xx[3]; yy[1] = yy[3];}
                ct = jud;
            }
        }
        it = AET.begin();
        while(it != AET.end()){
            if(sign((*it).dx-0.0) != 0) (*it).x = (((db)i+1.0-(*it).b)*(*it).dx);
            if((*it).ymax == i) it = AET.erase(it);
            else it++;
        }
    }
}
```

实验功能如下，点击屏幕，上一个点就会和当前点连成直线，当形成多边形之后，即可按下鼠标中键，最后一个点就会和第一个点连成直线并且对多边形包围的区域进行区域填充。


图形学实验一
DrawLine
DrawCircle
DrawEllipse
Xscan-AreaFilling
ClearScreen
Exit
Services


图形学实验一

实验总结：

1. Bresenham 画直线算法，需要对斜率大于 1、小于 1、以及是正是负分类讨论，但是将所有情况列出之后可以归纳四种情况大量简化代码。

2. Bresenham 画圆算法，将圆八等分对称画圆。可以很方便的画出这个圆。

3. Bresenham 画椭圆算法，一开始采用和画圆一样的方法，利用椭圆的四等分对称性，然后会发现发出的椭圆在切线斜率小于 -1，即 x 快速变化的那一段会出现失真情况。因此意识到需要对切线斜率大于 -1 和小于 -1 进行特判，即可画出椭圆。

4. 在椭圆的交互上面，一开始画的椭圆都是给定椭圆中心，然后确定长轴和短轴画椭圆。后来在画图软件上发现椭圆的交互都是用两点确定了一个长方形，然后长方形中心就是椭圆中心，长方形长宽分别是椭圆长轴和短轴。由此修改了椭圆的交互方式。

5. 对于 X-扫描线算法。我将其中的 NET 链表用 STL 中的 vector 代替，并且使用了 STL 中的 list 来构建 AET。由于算法中需要涉及浮点数比较，因此使用了 long double 增加精度，并且写了 sign 比较函数来保证精度控制在 1e-15。

```
typedef long double db;
const db EPS = 1e-15;
inline int sign(db a) {return a < -EPS ? -1 : a > EPS; } //返回-1表示a < 0, 1表示a > 0, 0表示a = 0
```

6. X-扫描线算法中，AET 中每个节点的 x 值每次递增△x，但由于 x 与△x 均为浮点数，当多边形形状比较复杂时，浮点数误差会不断积累，最后在判断交点时会发生错误。因此我在节点中存储了直线的一般式方程，将每次的递增△x 修改为了根据 y 坐标直接求出 x 坐标，虽然丢失了一些效率，但是严格保证了算法正确性。

7. X-扫描线算法中，我将 AET 中的插入排序直接改为了 STL 中的 sort，原因在于每次插入排序时间复杂度为 O(n)，如果插入元素过多，时间复杂度会很高。而 STL 中的 sort 不单只是快速排序，还会根据不同的数量级别以及不同情况，结合插入排序和堆排序进行排序。虽然复杂度为 O(nlogn)，但是实际表现与插入排序并无差别，而且在 y 递增的过程中，如果有大量直线插入的话，sort 表现的会优于直接插入排序。