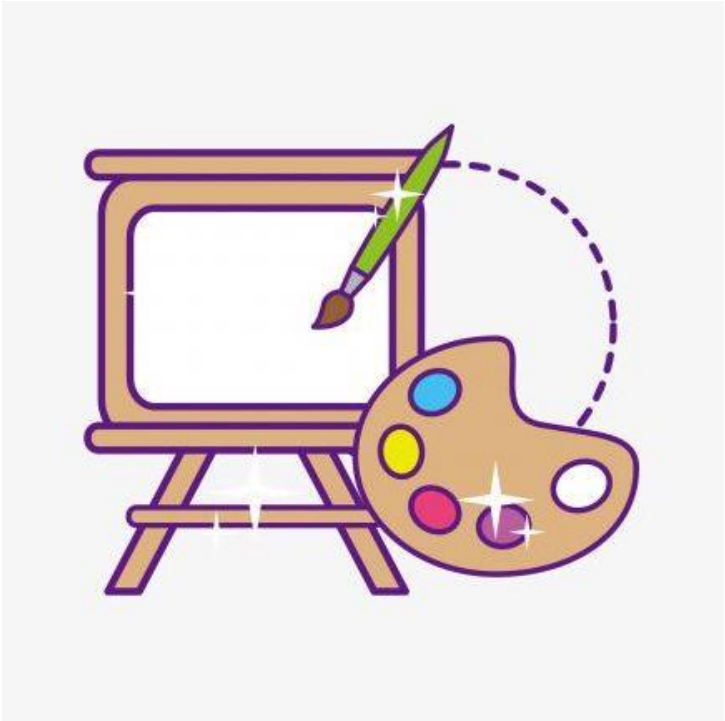
图形学在线画图工具

系统设计说明书

2018 年 05 月 31 日



目录

1.图元

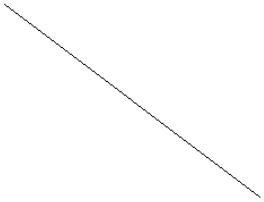
1.1 直线

这种画线算法的思想和中点画线的一致，只是在判断取哪个点时，不是看它位于中点的上边还是下边，而是将这俩个点到直线上那个点的距离相减，判断其正负，如果下边的点到直线实际点距离远则的 d1-d2>=0 那么取上边的点 y1+1,同样也是代入直线化解可以得出下面结论：

(1)当 d1-d2<0 时，d=d+2*dy.

(2)当 d1-d2>=0 时，d=d+2*dy-2*dx.

(3)d 的初始值为 d=2*dy-dx.



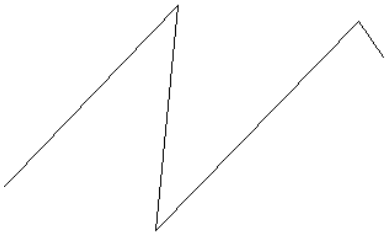
```
var pax = this.spoint.px;
        var pay = this.spoint.py;
        var pbx = this.epoint.px;
        var pby = this.epoint.py;
        var dx = pbx - pax;
        var dy = pby - pay;
        var x = pax;
        var y = pay;
        var eps;
        if (Math.abs(dx) > Math.abs(dy)) {
            eps = Math.abs(dx);
        } else {
            eps = Math.abs(dy);
        }
        var xlincre = dx * 1.0 / eps;
        var ylincre = dy * 1.0 / eps;
        for (var i = 0; i <= eps; i++) {
            var tp = new Point(parseInt(x + 0.5), parseInt(y + 0.5));
            tp.draw();//画点
            // if (down_flag == false) {
            //     this.edgepoints.push(tp);
            // }
            x += xlincre;
            y += ylincre;
```

}

其代码如上所示。运行截图如下：

1.2 折线

折线的画线方法和直线一样，本质上是添加画多个直线，代码上最主要的区别就是添加了一个鼠标右键点击事件。



```
for (var j = 0; j < this.vertices.length; j += 2) {
        var pax = this.vertices[j];
        var pay = this.vertices[j + 1];
        var pbx = this.vertices[j + 2];
        var pby = this.vertices[j + 3];
        var dx = pbx - pax;
        var dy = pby - pay;
        var x = pax;
        var y = pay;
        var eps;

        if (Math.abs(dx) > Math.abs(dy)) {
            eps = Math.abs(dx);
        } else {
            eps = Math.abs(dy);
        }
        var xlincre = dx * 1.0 / eps;
        var ylincre = dy * 1.0 / eps;
        for (var i = 0; i <= eps; i++) {
            var tp = new Point(parseInt(x + 0.5), parseInt(y + 0.5));
            tp.draw();//画点
            // if (down_flag == false) {
            //         this.edgepoints.push(tp);
            // }
            x += xlincre;
            y += ylincre;
```

```
            }
        }
```

其代码如上所示。运行截图如下：

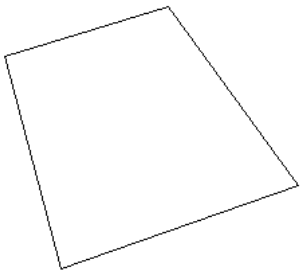
1.3 直角

画直角本质上就是画直线，竖线是将直线的所有点的 x 置为一个定值，将 y 也置为一个定值。

```
var pax = this.spoint.px;
        var pay = this.spoint.py;
        var pbx = this.epoint.px;
        var pby = this.epoint.py;
        var dx = pbx - pax;
        var dy = pby - pay;
        var x = pax;
        var y = pay;
        var eps;
        if (Math.abs(dx) > Math.abs(dy)) {
            eps = Math.abs(dx);
        } else {
            eps = Math.abs(dy);
        }
        var xlincre = dx * 1.0 / eps;
        var ylincre = dy * 1.0 / eps;
        for (var i = 0; i <= eps; i++) {
            new Point(pax, y).draw();//画点
            new Point(x, pby).draw();//画点
            x += xlincre;
            y += ylincre;
        }
}
```

1.4 任意多边形

多边形的画法是基于直线，画线方法和折线相同，相比于折线多了一个右键点击事件，在右键点击时，会自动将折线的首位两点连接形成多边形。



代码如下：

```
for (var j = 0; j < this.vertices.length; j += 2) {
                var pax = this.vertices[j];
                var pay = this.vertices[j + 1];
                var pbx = this.vertices[j + 2];
                var pby = this.vertices[j + 3];
                var dx = pbx - pax;
                var dy = pby - pay;
                var x = pax;
                var y = pay;
                var eps;

                if (Math.abs(dx) > Math.abs(dy)) {
                    eps = Math.abs(dx);
                } else {
                    eps = Math.abs(dy);
                }
                var xlincre = dx * 1.0 / eps;
                var ylincre = dy * 1.0 / eps;
                for (var i = 0; i <= eps; i++) {
                    var tp = new Point(parseInt(x + 0.5), parseInt(y + 0.5));
                    tp.draw();//画点
                    // if (down_flag == false) {
                    //         this.edgepoints.push(tp);
                    // }
                    x += xlincre;
                    y += ylincre;
                }
            }
}
```

1.5 圆形

中心点画圆法

根据圆的函数 F(x,y)=x^2+y^2-R^2;得出结论：对于平面内任意一点(x,y)，若 F(x,y)<0 则该点在圆内，若 F(x,y)=0，则该点在圆上。若 F(x,y)>0，则该点在圆外。

据此，我们可以构造判别式 d=F(x+1,y-0.5)。

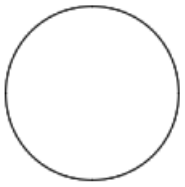
若 d<0, 判别式 d=F(x+2,y-0.5)=d+2x+3;.y 值不变

若 d>=0,判别式 f=F(x+2,y-1.5)=d+2(x-y)+5;y 值减少 1

初始是

起始点为(0,R),d=F(1,R-0.5)=1.25-R;

算法如下：



```
var cx = this.center.px;//圆心 x 坐标
        var cy = this.center.py;//圆心 y 坐标

        var da = 1.0 / (this.radius + 1);
        var radin = Math.PI / 4;//1/8 弧即 1/4pi
        var steps = parseInt(radin / da);
        var x = this.radius * Math.cos(0);
        var y = this.radius * Math.sin(0);
        for (var i = 0; i <= steps; i++) {
            new Point(x + cx, y + cy).draw();//画点 1
            new Point(x + cx, -y + cy).draw();//画点 2
            new Point(-x + cx, y + cy).draw();//画点 3
            new Point(-x + cx, -y + cy).draw();//画点 4
            new Point(y + cx, x + cy).draw();//画点 5
            new Point(-y + cx, x + cy).draw();//画点 6
            new Point(y + cx, -x + cy).draw();//画点 7
            new Point(-y + cx, -x + cy).draw();//画点 8
            x -= y * da;
            y += x * da;
        }
```

1.6 椭圆

椭圆的画线方法也是采用的中心点画圆法：

根据圆的函数 F(x,y)=x^2+y^2-R^2;得出结论：对于平面内任意一点(x,y)，若 F(x,y)<0 则该点在圆内，若 F(x,y)=0，则该点在圆上。若 F(x,y)>0，则该点在圆外。

据此，我们可以构造判别式 d=F(x+1,y-0.5)。

若 d<0, 判别式 d += b * b * (2 * x + 3);.y 值不变

若 d>=0,判别式 d += b * b * (2 * x + 3) + a * a * (-2 * y + 2);;y 值减少 1

当 y > 0)时

若 d<0, 判别式 d += b * b * (2 * a + 2) + a * a * (-2 * y + 3).y 值不变

若 d>=0,判别式 d += b * b * (2 * a + 2) + a * a * (-2 * y + 3) - b * b * (2 * a + 2);y 值减少 1。

初始是

起始点为(0,R),d=F(1,R-0.5)=1.25-R;

为了使用者能够简单的画出任意曲线，此处使用单个鼠标移动事件来画圆，我们将圆的 c 值与鼠标点的 x 相关连，这样，在移动鼠标时，可以同时改变椭圆的大小和形状。

算法如下：

```
var cx = this.center.px;//椭圆心 x 坐标
        var cy = this.center.py;//椭圆心 y 坐标
        var px = this.epoint.px;
        var py = this.epoint.py;

        var X = Math.pow(Math.abs(px - cx), 2);
        var Y = Math.pow(Math.abs(py - cy), 2);


        var B = (Y + Math.sqrt(Math.pow(Y, 2) + 4 * Y * X)) / 2;
        var A = B + X;
        var a = Math.sqrt(A);
        var b = Math.sqrt(B);


        var x = 0;
        var y = b;
        var d1 = b * b + a * a * (-b + 0.25);
        new Point(cx + x, cy + y).draw();//画点 1
        while (b * b * (x + 1) < a * a * (y - 0.25)) {
            if (d1 < 0) {
                d1 += b * b * (2 * x + 3);
                x++;
            }
            else {
                d1 += b * b * (2 * x + 3) + a * a * (-2 * y + 2);
                x++;
                y--;
            }
```

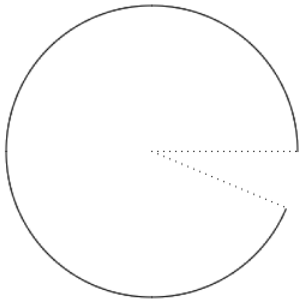
```
                new Point(x + cx, -y + cy).draw();//画点 2
                new Point(-x + cx, -y + cy).draw();//画点 3
                new Point(-x + cx, y + cy).draw();//画点 6
                new Point(x + cx, y + cy).draw();//画点 7

            }
            var d2 = Math.sqrt(b * (x + 0.5)) + Math.sqrt(a * (y - 1)) - Math.sqrt(b *
a);

            while (y > 0) {
                if (d2 < 0) {
                    d2 += b * b * (2 * a + 2) + a * a * (-2 * y + 3);
                    x++ , y--;
                }
                else {
                    d2 += b * b * (2 * a + 2) + a * a * (-2 * y + 3) - b * b * (2 * a +
2);
                    y--;
                }
                new Point(x + cx, -y + cy).draw();//画点 1
                new Point(-x + cx, -y + cy).draw();//画点 4
                new Point(-x + cx, y + cy).draw();//画点 5
                new Point(x + cx, y + cy).draw();//画点 8
            }
        /*
        求椭圆的长短轴
        */
        function Ellipse_ab(p1, p2) {
            var X = Math.pow(Math.abs(p1.px - p2.px), 2);
            var Y = Math.pow(Math.abs(p1.py - p2.py), 2);
            //a>b
            var B = (Y + Math.sqrt(Math.pow(Y) + 4 * Y * X)) / 2;
            var A = B + X;
            var a0 = Math.sqrt(A);
            var b0 = Math.sqrt(B);
            return a0, b0;
        }
```

1.7 任意圆弧

任意圆弧的画法是基于圆的画法，在此画法上，我们添加了对鼠标点位置的判断，使其，能够画出任意圆弧。

此算法的的优点是，基于四象限来判断的，先判断鼠标点在哪个象限，再判断点的具体位置，节省了大量的代码运行时间。

代码如下：

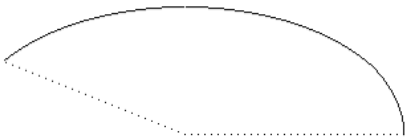


```
var cx = this.center.px;//圆心 x 坐标
        var cy = this.center.py;//圆心 y 坐标
        var sx = this.spoint.px;
        var sy = this.spoint.py;

        //添加虚线
        xp =new Point()
        xp.px=cx+this.radius;
        xp.py=cy;
        dotted_Line(this.center,xp);
        dotted_Line(this.center,this.spoint);

        var dx =sx-cx;
        var dy =sy-cy;

        console.log(objs.length);
        var da = 1.0 / (this.radius + 1);
        var radin = Math.PI / 4;//1/8 弧即 1/4pi
        var steps = parseInt(radin / da);
        var x = this.radius * Math.cos(0);
        var y = this.radius * Math.sin(0);
        for (i = 0; i <= steps; i++) {
            if(1/*1234 象限*/) {
                if((x>dx)||(!(dx>0&&dy<0))/*或非 1 象限*/){
                    new Point(x + cx, -y + cy).draw();/*画点 1*/}
                if((y>dx)||(!(dx>0&&dy<0))/*或非 1 象限*/){
                    new Point(y + cx, -x + cy).draw();/*画点 2*/}
            }
            if(!(dy<0&&dx>0)/*234 象限*/){
                if((-y>dx)||(dy>0)/*或 34 象限*/){
                    new Point(-y + cx, -x + cy).draw();/*画点 3*/}
                if((-x>dx)||(dy>0)/*或 34 象限*/){
                    new Point(-x + cx, -y + cy).draw();/*画点 4*/}
            }
            if(dy>0/*34 象限*/){
```

```
                if((-x<dx)||(dy>0&&dx>0)/*或 4 象限*/){
                    new Point(-x + cx, y + cy).draw();/*画点 5*/}
                if((-y<dx)||(dy>0&&dx>0)/*或 4 象限*/) {
                    new Point(-y + cx, x + cy).draw();/*画点 6*/}
            }
            if(dy>=0&&dx>=0){
                if(y<dx) {
                    new Point(y + cx, x + cy).draw();/*画点 7*/}
                if(x<dx) {
                    new Point(x + cx, y + cy).draw();/*画点 8*/}
            }

            x -= y * da;
            y += x * da;
        }
```

1.8 任意椭圆弧

任意圆弧的画法清参考圆的画法和任意曲线的画法。
代码如下：



```
var cx = this.center.px;//椭圆心 x 坐标
        var cy = this.center.py;//椭圆心 y 坐标
        var px = this.epoint.px;
        var py = this.epoint.py;

        var X = Math.pow(Math.abs(px - cx), 2);
        var Y = Math.pow(Math.abs(py - cy), 2);

        //直接令 x=c
        var B = (Y + Math.sqrt(Math.pow(Y, 2) + 4 * Y * X)) / 2;
        var A = B + X;
        var a = Math.sqrt(A);
        var b = Math.sqrt(B);

        //spoint 相对于圆心的坐标
        var dx,dy;
        dx=px - cx;
        dy=py - cy;
```

```
                //添加虚线
                var xp =new Point();
                xp.px=cx+a;xp.py=cy;
                dotted_Line(this.center,xp);
                dotted_Line(this.center,this.epoint);

                var x = 0;
                var y = b;
                var d1 = b * b + a * a * (-b + 0.25);
                new Point(cx + x, cy + y).draw();//画点 1
                while (b * b * (x + 1) < a * a * (y - 0.25)) {
                    if (d1 < 0) {
                        d1 += b * b * (2 * x + 3);
                        x++;
                    }
                    else {
                        d1 += b * b * (2 * x + 3) + a * a * (-2 * y + 2);
                        x++;
                        y--;
                    }
                    Arc(x,y,dx,dy,cx,cy);
                }
                var d2 = Math.sqrt(b * (x + 0.5)) + Math.sqrt(a * (y - 1)) - Math.sqrt(b *
a);

                while (y > 0) {
                    if (d2 < 0) {
                        d2 += b * b * (2 * a + 2) + a * a * (-2 * y + 3);
                        x++ , y--;
                    }
                    else {
                        d2 += b * b * (2 * a + 2) + a * a * (-2 * y + 3) - b * b * (2 * a +
2);
                        y--;
                    }
                    Arc(x,y,dx,dy,cx,cy);
            }
```

1.9 圆角矩形

圆角曲线的基本图元是矩形和圆形，将四个角的线段换成圆便形成了圆角。
代码如下：

```
var pax = this.spoint.px;
        var pay = this.spoint.py;
        var pbx = this.epoint.px;
        var pby = this.epoint.py;
        var dx = pbx - pax;
        var dy = pby - pay;
        var x = pax;
        var y = pay;
        var eps;
        if (Math.abs(dx) > Math.abs(dy)) {
            eps = Math.abs(dx);
        } else {
            eps = Math.abs(dy);
        }
        var xlincre = dx * 1.0 / eps;
        var ylincre = dy * 1.0 / eps;
        var kx = 10 * (Math.abs(pbx - pax)) / (pbx - pax);
        var ky = 10 * (Math.abs(pby - pay)) / (pby - pay);
        pax -= kx; pbx += kx; pay -= ky; pby += ky;
        Rounde(pax, pay, pbx, pby);
        for (var i = 0; i <= eps; i++) {
            new Point(pax, y).draw();//画点
            new Point(x, pby).draw();//画点
            new Point(pbx, y).draw();//画点
            new Point(x, pay).draw();//画点
            x += xlincre;
            y += ylincre;
    }
/*圆角*/
function Rounde(ax, ay, bx, by) {
    this.radius = 10;//半径     ----》此处可更改圆角大小
    var cx = (ax + bx) / 2;//圆心 x 坐标
    var cy = (ay + by) / 2;//圆心 y 坐标

    var da = 1.0 / (10);
    var radin = Math.PI / 4;//1/8 弧即 1/4pi
    var steps = parseInt(radin / da);
    var x = this.radius * Math.cos(0);
    var y = this.radius * Math.sin(0);
```

```
        var dx = Math.abs(bx - ax) / 2 - 10;
        var dy = Math.abs(by - ay) / 2 - 10;

        cx += dx; cy += dy;
        for (var i = 0; i <= 10; i++) {
            new Point(x + cx, y + cy).draw();//画点 1
            new Point(y + cx, x + cy).draw();//画点 5
            x -= y * da;
            y += x * da;
        }

        cx = (ax + bx) / 2; cy = (ay + by) / 2;
        x = this.radius * Math.cos(0); y = this.radius * Math.sin(0);
        cx += dx; cy -= dy;
        for (var i = 0; i <= 10; i++) {
            new Point(x + cx, -y + cy).draw();//画点 2
            new Point(y + cx, -x + cy).draw();//画点 7
            x -= y * da;
            y += x * da;
        }

        cx = (ax + bx) / 2; cy = (ay + by) / 2;
        x = this.radius * Math.cos(0); y = this.radius * Math.sin(0);
        cx -= dx; cy += dy;
        for (var i = 0; i <= 10; i++) {
            new Point(-x + cx, y + cy).draw();//画点 3
            new Point(-y + cx, x + cy).draw();//画点 6
            x -= y * da;
            y += x * da;
        }

        cx = (ax + bx) / 2; cy = (ay + by) / 2;
        x = this.radius * Math.cos(0); y = this.radius * Math.sin(0);
        cx -= dx; cy -= dy;
        for (var i = 0; i <= 10; i++) {
            new Point(-x + cx, -y + cy).draw();//画点 4
            new Point(-y + cx, -x + cy).draw();//画点 8
            x -= y * da;
            y += x * da;
    }
```

1.10 矩形

矩形的画法原理上和直角折线的画法一样，，本质上是将两个直角折线首位相连。

我也不多解释了

代码如下：

```
var pax = this.spoint.px;
        var pay = this.spoint.py;
        var pbx = this.epoint.px;
        var pby = this.epoint.py;
        var dx = pbx - pax;
        var dy = pby - pay;
        var x = pax;
        var y = pay;
        var eps;
        if (Math.abs(dx) > Math.abs(dy)) {
            eps = Math.abs(dx);
        } else {
            eps = Math.abs(dy);
        }
        var xlincre = dx * 1.0 / eps;
        var ylincre = dy * 1.0 / eps;
        for (var i = 0; i <= eps; i++) {
            new Point(pax, y).draw();//画点
            new Point(x, pby).draw();//画点
            new Point(pbx, y).draw();//画点
            new Point(x, pay).draw();//画点
            x += xlincre;
            y += ylincre;

        }
```

1.11 字符

每个字用点阵存储起来，1 代表黑色，0 代表白色

```
var ming=[
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,1,1,1,1,0,0,0,0,1,1,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,1,1,1,0,0,0],
    [0,0,0,0,0,0,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0],
    [0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,1,1,1,0,0,0,0,0,0,0,0,1,1,1,1,1,0],
    [0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,1,1,0,0,1,1,1,1,0,0,0,0,0,0,0,1,1,1,1,0,0,0],
    [0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0],
    [0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0],
    [0,0,0,0,1,1,1,0,0,0,0,0,1,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,1,1,1,1,0,0,0,0,0],
    [0,0,0,0,1,1,1,0,0,0,0,0,1,1,1,0,0,0,1,1,1,1,1,1,1,1,0,0,1,1,1,1,0,0,0,0,0,0],
    [0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,0,0,1,1,1,1,0,0,1,1,1,1,0,0,0,0,0,0],
    [0,0,0,1,1,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,0,1,1,1,1,0,0,0,0,0,0,0],
    [0,0,1,1,1,0,0,1,1,1,1,0,0,0,0,0,1,1,0,0,0,0,0,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0],
    [0,1,1,1,0,0,0,1,1,1,1,0,0,0,0,1,1,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0],
    [0,1,1,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,1,1,1,1,0,0,1,1,1,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,1,1,1,1,1,1,0,0,0,0,0,0,0,0,1,1,1,0,0],
    [0,0,1,1,1,1,0,1,1,1,1,0,0,0,0,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0],
    [0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0],
    [0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0],
    [0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0],
    [0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0],
    [0,0,0,0,0,0,0,1,1,1,1,0,0,0,1,1,1,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0],
    [0,0,0,0,0,0,0,1,1,1,1,0,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0],
    [0,0,0,0,0,0,0,1,1,1,1,0,0,0,1,1,1,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0],
    [0,0,0,0,0,0,0,1,1,1,1,0,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0],
    [0,0,0,0,0,0,0,1,1,1,1,1,1,1,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0],
    [0,0,0,0,0,0,0,1,1,1,1,1,1,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0],
    [0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0],
    [0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0],
    [0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0],
```

规定一个当前点，两个循环，遍历数组，当为 1 时填充像素，为 0 时不填充，遍历时当前点也同步向前遍历：

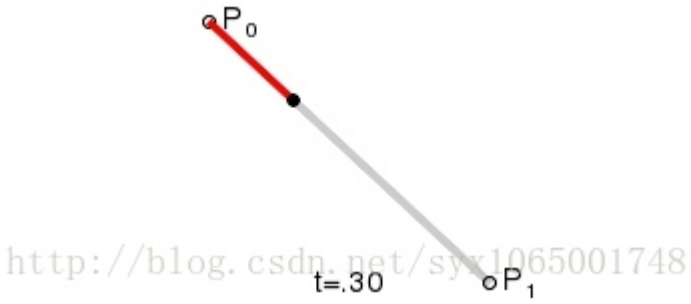
```
for(var i=0;i<gongchangz.length;i++)
{
    for (var j = 0; j <gongchangz[i].length;j++) {
        if(gongchangz[i][j]==1)
            ctx.fillRect(xxx, yyy, 1, 1);
        xxx++;

    }
    xxx=100;
    yyy++;
}

yyy=100;
```

操作文档：当点击页面上的文字，即可在画布上展现文字，效果如图：

1.12 任意曲线

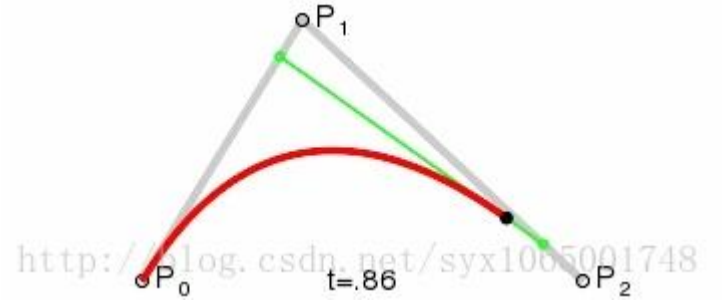
贝塞尔曲线贝塞尔曲线是计算机图形图像造型的基本工具，是图形造型运用得最多的基本线条之一。它通过控制曲线上的四个点（起始点、终止点以及两个相互分离的中间点）来创造、编辑图形。其中起重要作用的是位于曲线中央的控制线。这条线是虚拟的，中间与贝塞尔曲线交叉，两端是控制端点。移动两端的端点时贝塞尔曲线改变曲线的曲率（弯曲的程度）；移动中间点（也就是移动虚拟的控制线）时，贝塞尔曲线在起始点和终止点锁定的情况下做均匀移动。注意，贝塞尔曲线上的所有控制点、节点均可编辑。这种“智能化”的矢量线条为艺术家提供了一种理想的图形编辑与创造的工具。

以下公式中：B(t)为 t 时间下 点的坐标；
P0 为起点,Pn 为终点,Pi 为控制点
一阶贝塞尔曲线(线段)：



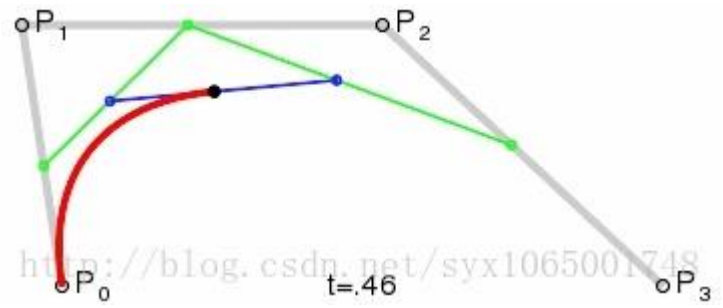
意义：由 P0 至 P1 的连续点， 描述的一条线段
二阶贝塞尔曲线(抛物线)：

$$\mathbf{B}(t) = (1-t)^2\mathbf{P}_0 + 2t(1-t)\mathbf{P}_1 + t^2\mathbf{P}_2 \ , t \in [0,1]$$



原理：由 P0 至 P1 的连续点 Q0，描述一条线段。
由 P1 至 P2 的连续点 Q1，描述一条线段。
由 Q0 至 Q1 的连续点 B(t)，描述一条二次贝塞尔曲线。
经验：P1-P0 为曲线在 P0 处的切线。
三阶贝塞尔曲线：

$$\mathbf{B}(t) = \mathbf{P}_0(1-t)^3 + 3\mathbf{P}_1 t(1-t)^2 + 3\mathbf{P}_2 t^2(1-t) + \mathbf{P}_3 t^3 \ , t \in [0,1]$$



通用公式：

$$P_i^k = \begin{cases} P_i & k=0 \\ (1-t)P_i^{k-1} + tP_{i+1}^{k-1} & k=1,2,\cdots,n,\ i=0,1,\cdots,n-k \end{cases}$$

高阶以此类推。

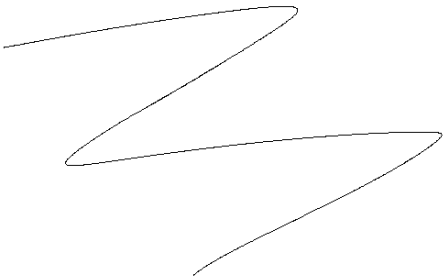
在此基础上为实现圆弧曲线的随鼠标移动绘制，记录鼠标移动的轨迹点，将轨迹点分组，对每一组采用中点降阶的贝尔塞曲线法绘制。中点降阶使用分段递归实现。

代码截图：

设置/取消书签(M) (Ctrl+Alt+B)　pitch.js | yu　items.js

```
146
147 //n为数组长度-1
148 //p为2维点数
149 function Curvelist(pp,n)
150 {
151     var p = [];
152     for(var i = 0;i<=pp.length-1;i++){
153         p.push(pp[i]);
154     }
155     if (n<= 1)
156         return null;
157     if((p[n-1]<p[0]+1)&&(p[n-1]>p[0]-1)&&(p[n]<p[1]+1)&&(p[n]>p[1]-1))
158     {
159 //      new Point(parseInt(p[0]), parseInt(p[1])).draw();
160         ctx.fillRect(parseInt(p[0]), parseInt(p[1]), 1, 1);
161         return null;
162     }
163     p1 = [];
164     var i, j;
165     p1.push(p[0],p[1]);
166     for(i=2; i<=n; i+=2)
167     {
168         for(j=0; j<=n-i;j+=2)
169         {
170             p[j] = (p[j] + p[j+2])/2;
171             p[j+1] = (p[j+1] + p[j+3])/2;
172         }
173         p1.push(p[0],p[1]);
174     }
175     Curvelist(p1,p1.length-1);
176     Curvelist(p,p.length-1);
177 }
178
179 /*直角*/
180 function Rightangle() {
181     Drawable.apply(this);//继承父类
182     this.spoint = new Point();//起始点
183     this.epoint = new Point();//终止点
```

操作截图：

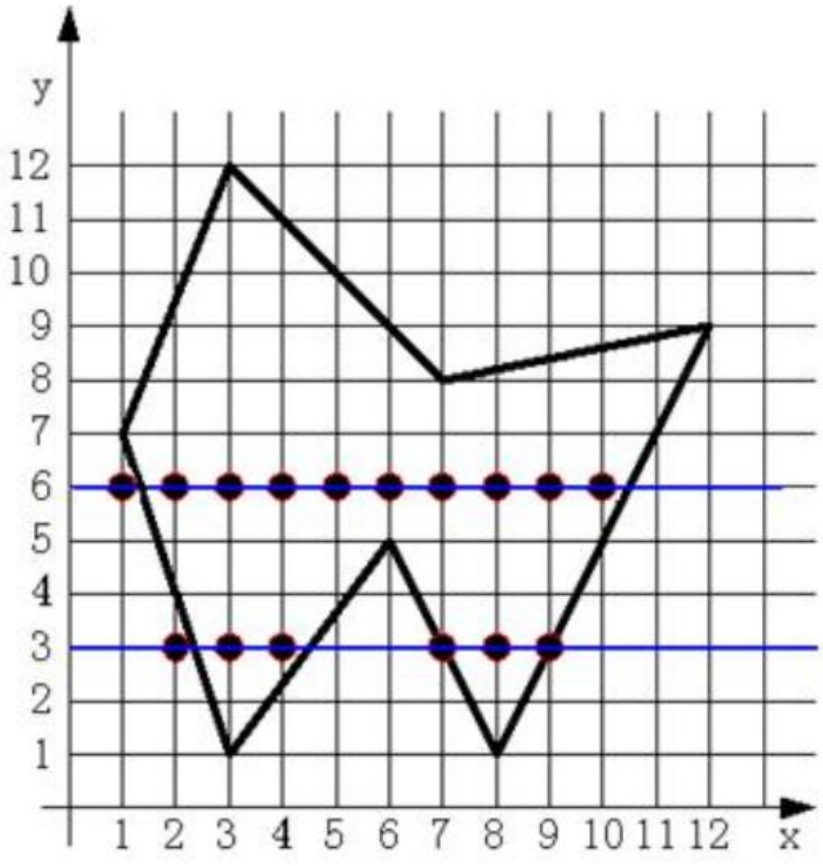


2.图元填充

图元首先选中，然后进行填充。填充分为三种，一、有序边表填充算法（扫描线）

1、算法思想

简单的来说，就是计算出屏幕上有哪些像素点是在多边形内部的，重复利用多变形内部区域的连续性。算出每一条水平线上有哪些点是在该多边形内的，然后改变这些点的像素值。如图所示：



2、算法步骤

a、确定多边形顶点的最小和最大 y 值（ymin 和 ymax）。

b、从 y=ymin 到 y=ymax，每次用一条扫描线进行填充。

c、对一条扫描线填充的过程可分为三个步骤：

(1)求交(能否只对有效边求交？)

有效边(ActiveEdge)：与当前扫描线相交的多边形边。

(2)按交点的 x 坐标排序(有序链表)

有效边表->有序的有效边表

(3)交点配对，区间填色

二、边填充算法

1、算法思想

利用异或运算，反复填充水平线：

对于多变形边上的每一个点，都以改点为初始点，向右画水平线，水平线的颜色就是所要填充的颜色。根据上一篇博文《点与多边形的关系》中的原理，我们知道处于多边形内部的区域会被水平线画到奇数次，而处于多边形外部的区域会被画到偶数次，由于采用的是异或画法，所以奇数部分的颜色会被保存下来，而偶数部分的颜色会被清除。即多边形内部会被填充颜色。

2、算法步骤

a、选择多边形的一条边，计算出其处于上方的点 up 和处于下方的点 down，并且计算该条边的斜率 K

b、根据斜率 K，遍历直线上的所有点，对于直线上的每一个点，向右画出水平线，用指定的颜色填充。

c、重复以上步骤直到所有的边都已经被处理过了。

3、评价

该算法实现起来非常的简单,而且适用范围广，凸、凹多边形，甚至可以带孔的多边形都可以用来填色。

同时，确定很明显，它需要大量的重复运算，效率不高。

三、种子填充算法

1、算法思想

在多边形内部指定一个种子点，从这个点开始，向四周蔓延填色，直到碰到边界。

多边形内部的每一个像素点看成图中一个节点,每个节点只与自己上下左右的四个像素点有边相连，这样，多边形的填色问题就演变成了一个图的遍历问题。自然而然地我们就想到了图论中的深度遍历和广度遍历。

2、算法步骤

这个算法有多种实现方法，其中最常见的就是深度遍历和广度遍历。分别对应着栈和堆的特性。这里我就再多说了。直接给出代码：

实现代码：

```
var polygonFilling = function( vertices ) {

    // "global" variables
    var verticesCount = vertices.length,
        edgesEntered,
        // table cols
        yMax, yMin, xInt, invNegSlope;

// ************************

    // 多边形坐标数组长度小于 4，不是多边形
    if( verticesCount <= 4 ) {
```

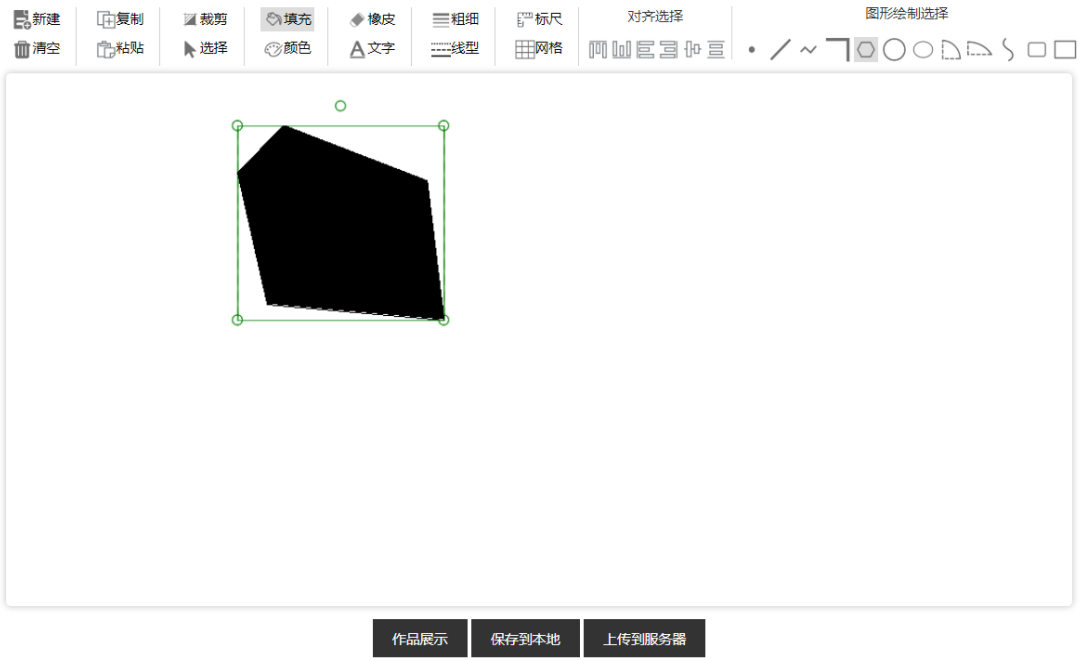
```
            console.error( 'polygon has insuffient number of vertices' );
            return;
        }
        // 多边形坐标数组长度为奇数，错误
        if( verticesCount % 2 !== 0 ) {
            console.error( 'polygon has an incorrect number of vertices' );
            return;
        }

        // find out table length
        // y1:数组最后一个元素
        var y1 = Math.round( vertices[verticesCount-1] ), y2,
            i;

        // 顶点个数
        edgesEntered = 0;
        for( i = 1; i < verticesCount; i += 2 ) {
            // y2 纵坐标
            y2 = Math.round( vertices[i] );
            if( y1 !== y2 ) {
                y1 = y2;
                edgesEntered++;
            }
        }
        // 顶点数小于 2，非多边形
        if( edgesEntered < 2 ) {
            console.error( 'malformed polygon' );
            return;
        }

        // setup table length
        // arraybuffer 视图 根据顶点个数
        xInt = new Float64Array( edgesEntered );
        yMin = new Uint32Array( edgesEntered );
        yMax = new Uint32Array( edgesEntered );
        invNegSlope = new Float64Array( edgesEntered );

// *************************

        var insertTableEntry = function( x1,y1, x2,y2, edgesEntered ) {
            var biggerY = Math.max( y1, y2 ),
                i = edgesEntered;
            for( ; i !== 0 && yMax[i-1] < biggerY; i-- ) { // insertion sort mechanism
                // shift elements to right
```

```
                yMax[i] = yMax[i-1];
                yMin[i] = yMin[i-1];
                xInt[i] = xInt[i-1];
                invNegSlope[i] = invNegSlope[i-1];
            }
            yMax[i] = biggerY;
            invNegSlope[i] = -(x2-x1)/(y2-y1);
            if( biggerY === y1 ) {
                yMin[i] = y2;
                xInt[i] = x1;
            } else {
                yMin[i] = y1;
                xInt[i] = x2;
            }
        };

        var x1 = Math.round( vertices[verticesCount-2] ),
            y1 = Math.round( vertices[verticesCount-1] ),
            x2, y2,
            i;

        edgesEntered = 0;
        for( i = 0; i < verticesCount; i += 2 ) {
            x2 = Math.round( vertices[i] );
            y2 = Math.round( vertices[i+1] );
            if( y1 !== y2 ) { // avoid horizontal edges
                insertTableEntry( x1,y1, x2,y2, edgesEntered );
                edgesEntered++;
            }
            x1 = x2;
            y1 = y2;
        }

// ************************

        // helpers functions
        var exclude = function( leftMostEdge, rightMostEdge, scan ) {
            var i, j;
            for( i = leftMostEdge; i <= rightMostEdge; i++ ) {
                if( yMin[i] > scan ) {
                    leftMostEdge++;
                    for( j = i; j >= leftMostEdge; j-- ) {
                        // shift elements to right
                        yMin[j] = yMin[j-1];
```

```
                    xInt[j] = xInt[j-1];
                    invNegSlope[j] = invNegSlope[j-1];
                }
            }
        }
        return leftMostEdge;
    };

    var updateXInt = function( leftMostEdge, rightMostEdge ) {
        for( var i = leftMostEdge; i <= rightMostEdge; i++ ) {
            xInt[i] += invNegSlope[i];
        }
    };

    var include = function( rightMostEdge, scan, edgesEntered ) {
        for( ; rightMostEdge + 1 < edgesEntered && yMax[rightMostEdge + 1] > scan;
rightMostEdge++ );
        return rightMostEdge;
    };

    var sortXInt = function( leftMostEdge, rightMostEdge ) { // bubble sort mechanism
        var i, j, tmp;
        for( i = leftMostEdge; i <= rightMostEdge - 1; i++ ) {
            for( j = i + 1; j <= rightMostEdge; j++ ) {
                if( xInt[i] > xInt[j] ) {
                    tmp = yMin[i];
                    yMin[i] = yMin[j];
                    yMin[j] = tmp;
                    tmp = xInt[i];
                    xInt[i] = xInt[j];
                    xInt[j] = tmp;
                    tmp = invNegSlope[i];
                    invNegSlope[i] = invNegSlope[j];
                    invNegSlope[j] = tmp;
                }
            }
        }
    };

    var fillScan = function( leftMostEdge, rightMostEdge, scan ) {
        var x1, x2, i;
        for( i = leftMostEdge; i <= rightMostEdge; i += 2 ) {
            x1 = Math.round( xInt[i] );
            x2 = Math.round( xInt[i+1] );
```

```
                // console.log( 'drawing line from', x1, 'to', x2, 'at y =', scan );
                for( ; x1 <= x2; x1++ ) {
                    // plot pixel (x1,scan)
                    new Point(x1, scan).draw();
                }
            }
        };

// ************************

        // start filling
        var scan = yMax[0],
            leftMostEdge = 0,
            rightMostEdge = -1;

        while( leftMostEdge < edgesEntered ) {
            scan--;
            leftMostEdge = exclude( leftMostEdge, rightMostEdge, scan );
            updateXInt( leftMostEdge, rightMostEdge );
            rightMostEdge = include( rightMostEdge, scan, edgesEntered );
            sortXInt( leftMostEdge, rightMostEdge );
            fillScan( leftMostEdge, rightMostEdge, scan );
        }
};

/*var vertices = new Float64Array( [
        3,8,
        6,3,
        1,1
] );*/
```

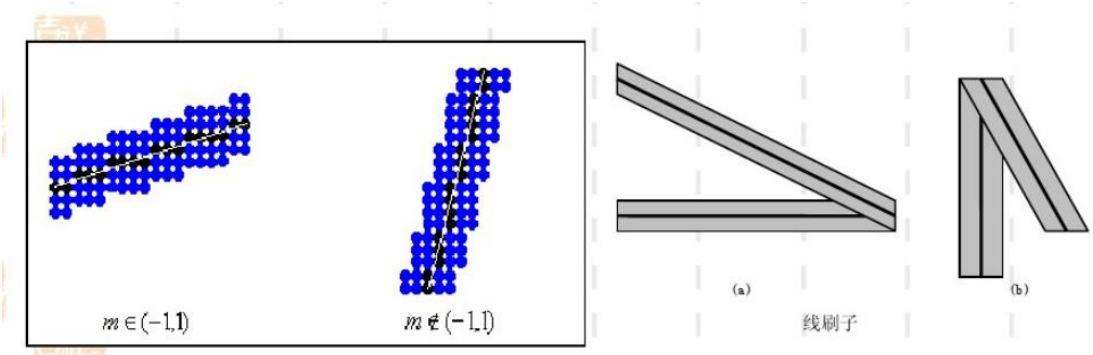
结果截图：



3.线型线宽

3.1 线宽

直线的线宽采用一定宽度的“线刷子”来实现，当直线斜率在[-1,1]时，将线刷子定成垂直方向，并将线刷子中心对准直线上某一像素点，然后将线刷子沿直线移动画出一条有宽度的直线；当直线斜率不在[-1,1]上时，将线刷改为水平方向即可：



实现伪代码：

水平线刷子：

```
LineBrush（int x, int y）:
{
    Drawpixel(x,y);
    Drawpixel(x-1,y);
    Drawpixel(x+1,y);
    Drawpixel(x-2,y);
```

```
        Drawpixel(x+2,y);
        Drawpixel(x-3,y);
        Drawpixel(x+3,y);
    }
    垂直线刷子：
    LineBrush（int x, int y）:
    {
        Drawpixel(x,y);
        Drawpixel(x,y-1);
        Drawpixel(x,y+1);
        Drawpixel(x,y-2);
        Drawpixel(x,y+2);
        Drawpixel(x,y-3);
        Drawpixel(x,y+3);
    }
```

每一个元素对象有 weight 属性，用 this.weight 规定每一个画布上直线对象特有的宽度，以实现画布上画出不同线宽的任意线型的直线对象。

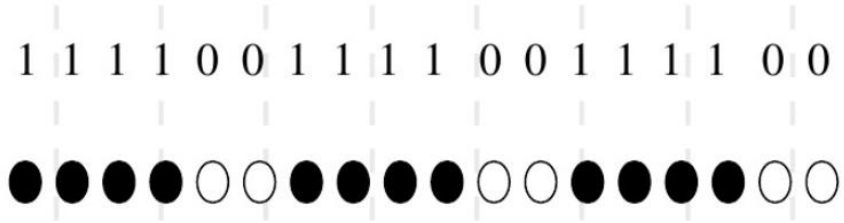
实现代码：（以实直线对象为例）：

```
    /*
    实线对象
 */
function Line() {
    Drawable.apply(this);//继承父类
    this.spoint = new Point();//起始点
    this.epoint = new Point();//终止点
    this.weight= localStorage.getItem("width");//获取用户输入的宽度
    this.draw = function () {
        var pax = this.spoint.px;
        var pay = this.spoint.py;
        var pbx = this.epoint.px;
        var pby = this.epoint.py;
        var dx = pbx - pax;
        var dy = pby - pay;
        var x = pax;
        var y = pay;
        var eps;
        if(this.weight==null){
            this.weight=0;
        }
        console.log(this.weight);
        var l=dy/dx;//计算直线斜率
        if (Math.abs(dx) > Math.abs(dy)) {
            eps = Math.abs(dx);
        } else {
```

```
            eps = Math.abs(dy);
        }
        var xlincre = dx * 1.0 / eps;
        var ylincre = dy * 1.0 / eps;
        //当斜率为[-1,1]之间时，采用垂直线刷
        if(l>=-1&&l<=1){
                for (var i = 0; i <= eps; i++) {
                for(var k = 0;k <= this.weight; k++){
                var tp1 = new Point(parseInt(x + 0.5), parseInt(y + 0.5-(k/2)));
                var tp2 = new Point(parseInt(x + 0.5), parseInt(y + 0.5));
                var tp3 = new Point(parseInt(x + 0.5), parseInt(y + 0.5+(k/2)));
                tp1.draw();
                tp2.draw();
                tp3.draw();

                }//画线刷
                x += xlincre;
                y += ylincre;

                }
            }
            //当斜率不在[-1,1]之间时，采用水平线束
            else{
                for (var i = 0; i <= eps; i++) {
                for(var k = 0;k <= this.weight; k++){
                var tp1 = new Point(parseInt(x + 0.5-(k/2)), parseInt(y + 0.5));
                var tp2 = new Point(parseInt(x + 0.5), parseInt(y + 0.5));
                var tp3 = new Point(parseInt(x + 0.5+(k/2)), parseInt(y + 0.5));
                tp1.draw();
                tp2.draw();
                tp3.draw();

                }//画线刷

                }
            }
    }
}
```

3.2 线型

线型才采用布尔数组来存放，当数组中，对应数值为 1，则画点，为 0 则不画点。通过一定长度的布尔数组定义线型，线型必须以该长度的像素进行周期重复。如图所示：



实现伪代码：

Drawpixel（x,y）-> if(位串%线型周期长度){Drawpixel（x,y）};

本次课程设计有 5 中线型：实线，短虚线，长虚线，点横线和两点横线 5 种，对应的布尔数组分别为：

实现默认为 1，不需要设定数组；

短虚线：[1,1,1,1,1,1,1,1,1,0,0,0];

长虚线：[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,,0,0,0,0];

点横线：[1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,1,1,1,1,0,0,0,0];

两点横线：[1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,1,1,1,1,0,0,0,0,0,1,1,1,1,0,0,0,0];

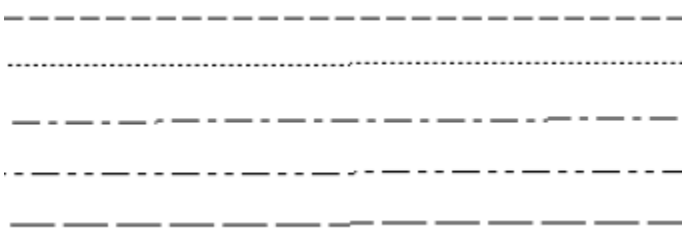
5 种线型的实现代码为：（关键部分：）

```
//短虚线
var linkmark = [1,1,1,1,1,1,1,1,1,0,0,0];//短虚线数组
if(linkmark[i%12]==1){
                    var tp = new Point(parseInt(x + 0.5), parseInt(y + 0.5));
                    tp.draw();
                   }

//长虚线
var linkmark =    [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,,0,0,0,0]//长虚线数组
if(linkmark[i%25]==1){
                    var tp = new Point(parseInt(x + 0.5), parseInt(y + 0.5));
                    tp.draw();
}

//点线
var linkmark = [1,1,0,0];//点线数组
if(linkmark[i%4]==1){

                    var tp = new Point(parseInt(x + 0.5), parseInt(y + 0.5));
                    tp.draw();
                   }
//点横线
var linkmark = [1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0];//点横线数组
if(linkmark[i%25]==1){
                    var tp = new Point(parseInt(x + 0.5), parseInt(y + 0.5));
                    tp.draw();
                   }
//两点横线
```

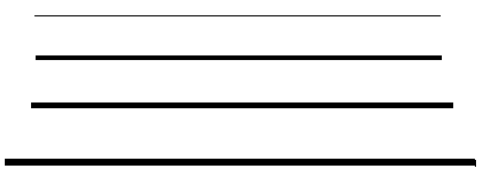
```
var linkmark = [1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0];//两点横线数组
if(linkmark[i%33]==1){
                var tp = new Point(parseInt(x + 0.5), parseInt(y + 0.5));
                tp.draw();
              }
```

操作文档：

点击相应的线型图标，在画布上画出相应的线型，线型分别有：实线，短虚线，长虚线，点横线和两点横线 5 种，在画布上可以画出相应线型。输入任意线宽，获取用户输入的线宽值，在画布上画出对应宽度的所选线型。显示结果如图：



线宽功能结果如图：



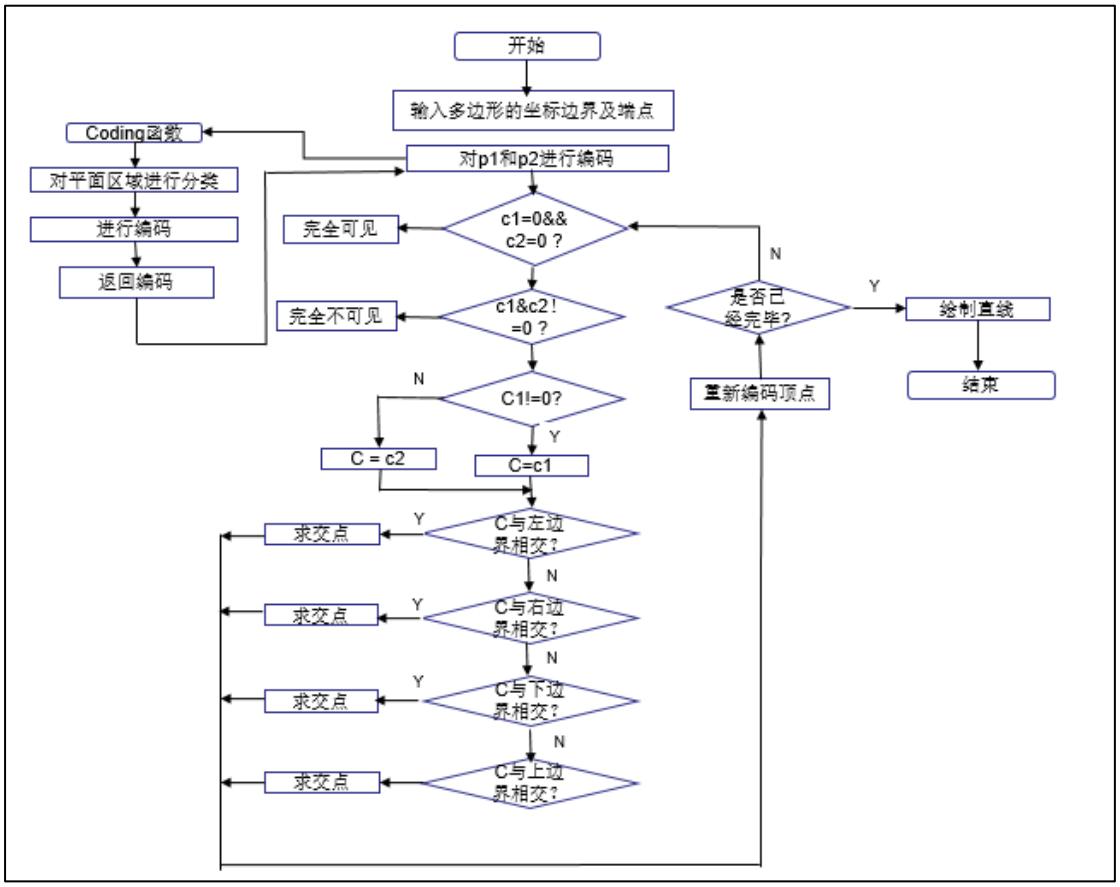
4.裁剪

1. 线段内外裁剪

一、 Cohen－Sutherland 算法思想

该算法也称为编码算法，首先对线段的两个端点按所在的区域进行分区编码，根据编码可以迅速地判明全部在窗口内的线段和全部在某边界外侧的线段。只有不属于这两种情况的线段，才需要求出线段与窗口边界的交点，求出交点后，舍去窗外部分。对剩余部分，把它作为新的线段看待，又从头开始考虑。两遍循环之后，就能确定该线段是部分截留下来，还是全部舍弃。

二、 Cohen－Sutherland 算法步骤：



1.分区编码

延长裁剪边框将二维平面分成九个区域，每个区域各用一个四位二进制代码标识。各区代码值如图中所示。

1001	1000	1010
0001	0000	0010
0101	0100	0110

四位二进制代码的编码规则是：第一位置 1：区域在左边界外侧；第二位置 1：区域在右边界外侧；第三位置 1：区域在下边界外侧；第四位置 1：区域在上边界外侧。裁剪窗口内（包括边界上）的区域，四位二进制代码均为 0。设线段的两个端点为 P_1（x_1，y_1）和 P_2（x_2，y_2），根据上述规则，可以求出 P_1 和 P_2 所在区域的分区代码 C_1 和 C_2。

2．判别

根据 C_1 和 C_2 的具体值，可以有三种情况：

- $C_1=C_2=0$，表明两端点全在窗口内，因而整个线段也在窗内，应予保留。
- $C_1\&C_2\neq0$（两端点代码按位作逻辑乘不为 0），即 C_1 和 C_2 至少有某一位同时为 1，表明两端点必定处于某一边界的同一外侧，因而整个线段全在窗外，应予舍弃。
- 不属于上面两种情况，均需要求交点。

3．求交点

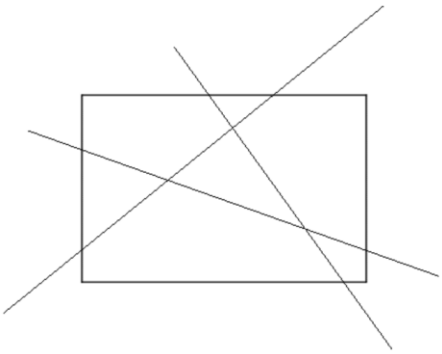
假设算法按照：左、右、下、上边界的顺序进行求交处理，对每一个边界求完交点，并相关处理后，算法转向第 2 步，重新判断，如果需要接着进入下一边界的处理。为了规范算法，令线段的端点 P_1 为外端点，如果不是这样，就需要 P_1 和 P_2 交换端点。当条件 $(C_1\&0001\neq0)$成立时，表示端点 P_1 位于窗口左边界外侧，按照前面介绍的求交公式，进行对左边界的求交运算。依次类推，对位于右、下、上边界外侧的判别，应将条件式中的 0001 分别改为 0010、0100、1000 即可。求出交点 P 后，用 $P_1=P$ 来舍去线段的窗外部分，并对 P_1 重新编码得到 C_1，接下来算法转回第 2 步继续对其它边界进行判别。

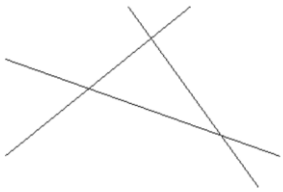
三、 算法编码

```
function Inner_cohenSutherland(startPoint, endPoint, r1, r2) {
    // console.log("wahais " + r1, r2);
    // console.log(startPoint,endPoint)
    let outerPoints = []; //在外部的点
    //计算直线斜率
    let gradient = (startPoint.py - endPoint.py) / (startPoint.px - endPoint.px);
    //初始化区域码
    let positionCodeOfStart = initPositionCode(startPoint, r1, r2),
        positionCodeOfEnd = initPositionCode(endPoint, r1, r2);
    // console.log(positionCodeOfStart);
    // console.log(positionCodeOfEnd);
    //画线
    //drawLine(startPoint,endPoint);
    let count = 0;
    for (let i = 0; i < 4; i++) {
        if (positionCodeOfStart[i] == false && positionCodeOfEnd[i] == false) {
            count++;
        }
    }
    if (count == 4) {
        drawLine(startPoint, endPoint, 'normal');
        return;
    }
    //循环四次，按上右下左的顺序的顺序
    for (let i = 0; i < 4; i++) {
        let positionFlag = positionCodeOfStart[i] || positionCodeOfEnd[i];
        // console.log(positionFlag)
        if (positionFlag == true) {
            if (positionCodeOfStart[i] == true && positionCodeOfEnd[i] == true) {
                // 调用画线方法 外部点
                // drawLine(startPoint, endPoint, 'outer');
                return;
            } else {
                //这种情况是有一个点在相对应边界之外
                if (i == 0) {
                    outerPoints[0] = (calculatePoint(gradient, startPoint, r2.py, 'y'));
                } else if (i == 1) {
                    outerPoints[1] = (calculatePoint(gradient, startPoint, r2.px, 'x'));
                } else if (i == 2) {
                    outerPoints[2] = (calculatePoint(gradient, startPoint, r1.py, 'y'));
                } else if (i == 3) {
                    outerPoints[3] = (calculatePoint(gradient, startPoint, r1.px, 'x'));
```

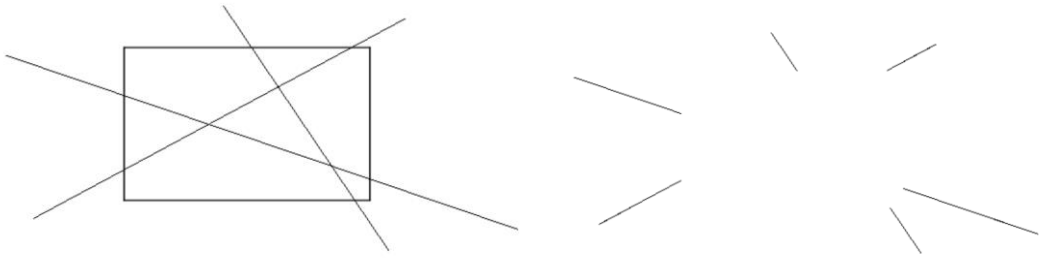
四、 实现结果

1. 线段内裁剪





2. 线段外裁剪

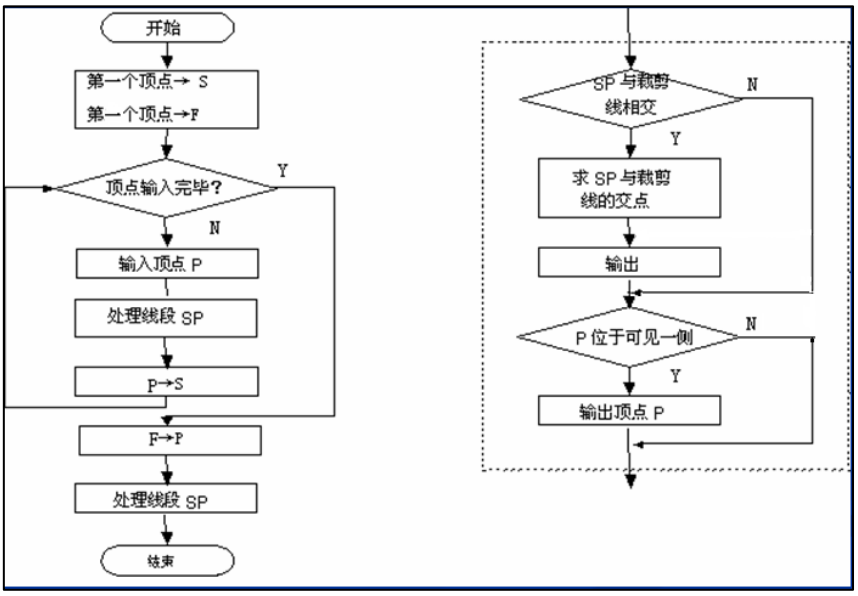


2. 多边形内裁剪

一、 Sutherland-Hodgman 算法思想

将多边形作为一个整体，每次用窗口的一条边对要裁剪的多边形和中间结果多边形进行裁剪，体现一种分而治之的思想。

二、 Sutherland-Hodgman 算法步骤：



从裁剪得到的结果，可发现多边形的顶点由两部分组成：

- 落在可见一侧的原多边形顶点
- 多边形的边与裁剪窗口边界的交点。

根据多边形每一边与窗口边所形成的位置关系，沿着多边形依次处理顶点会遇到四种情况：

1.第一点 S 在不可见侧面，而第二点 P 在可见侧。处理方法：交点 I 和点 P 均被加入到输出顶点表中。

2. S 和 P 都在可见侧。处理方法：P 被加入到输出顶点表中。

3. S 在可见侧，而 P 在不可见侧。处理方法：交点 I 被加入到输出顶点表中。

4. 如果 S 和 P 都在不可见侧。处理方法：输出顶点表中不增加任何顶点。

在窗口的一条裁剪边界处理完所有顶点后，其输出顶点表将用窗口的下一条边界继续裁剪。

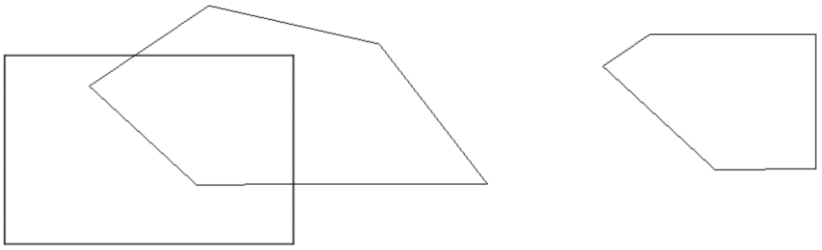
分割处理策略：将多边形关于矩形窗口的裁剪分解为多边形关于窗口四边所在直线的裁剪。流水线过程(左上右下)：前边的结果是后边的输入。

- 一次用窗口的一条边裁剪多边形。
- 考虑窗口的一条边以及延长线构成的裁剪线,把平面分成两个部分:可见一侧；不可见一侧
- 多边形的各条边的两端点 S、P。它们与裁剪线的位置关系只有四种
- 情况（1）仅输出顶点 P；
- 情况（2）输出 0 个顶点；
- 情况（3）输出线段 SP 与裁剪线的交点 I；
- 情况（4）输出线段 SP 与裁剪线的交点 I 和终点 P

三、算法编码

```
function Mul_Inner_SutherlandHodgman(polygon, r1, r2) {
    //接收多边形顶点集
    //每条边逐一考虑 1左 2下 3右 4上
    new_polygon = polygon; //裁剪新点集
    for (var i = 1; i < 5; i++) {
        new_polygon = clipEdge(i, new_polygon, r1, r2);
    }
    for (var i = 0; i < new_polygon.length; i++) {
        //描边
        TheLine(new_polygon[i], new_polygon[(i + 1) % new_polygon.length]);
    }
}

function clipEdge(edge, new_polygon, r1, r2) {
    index = 0;
    polygonTemp = [];
    var isP1In, isP2In;
    for (var i = 0; i < new_polygon.length; i++) {
        isP1In = inSide(edge, new_polygon[i], r1, r2);
        isP2In = inSide(edge, new_polygon[(i + 1) % new_polygon.length], r1, r2);
        //多边形一边的两个顶点都在内侧 加入第二个顶点
        if (isP1In && isP2In)
            polygonTemp[index++] = new_polygon[(i + 1) % new_polygon.length];
        //多边形一边的第一个顶点在内侧 加入边界交点
        else if (isP1In)
            polygonTemp[index++] = findIntersection(edge, new_polygon[i], new_polygon[(i + 1) % new_polygon.length], r1, r2);
        //多边形一边的第二个顶点在内侧 加入边界交点和第二个顶点
        else if (isP2In) {
            polygonTemp[index++] = findIntersection(edge, new_polygon[i], new_polygon[(i + 1) % new_polygon.length], r1, r2);
            polygonTemp[index++] = new_polygon[(i + 1) % new_polygon.length];
        }
    }
    //更新新点集合
    // new_polygon.length = index;
    // for (var i = 0; i < new_polygon.length; i++)
    //     new_polygon[i] = polygonTemp[i];
    return polygonTemp;
}
```

四、 实现结果：



3. 多边形外裁剪

一、 Weiler－Atherton 算法思想

Sutherland－Hodgeman 算法解决了裁剪窗口为凸多边形窗口的问题，但一些应用需要涉及任意多边形窗口（含凹多边形窗口）的裁剪。Weiler-Atherton 多边形裁剪算法正是满足这种要求的算法。

裁剪窗口和被裁剪多边形处于完全对等的地位，这里我们称被裁剪多边形为主多边形，记为 A；裁剪窗口为裁剪多边形，记为 B。主多边形 A 和裁剪多边形 B 的边界将整个二维平面分成了四个区域，分别为 A∩B（交：属于 A 且属于 B）；A－B（差：属于 A 不属于 B）；B－A（差：属于 B 不属于 A）；A∪B(并：属于 A 或属于 B，取反；即：不属于 A 且不属于 B)。内裁剪即通常意义上的裁剪，取图元位于窗口之内的部分，结果为 A∩B。外裁剪取图元位于窗口之外的部分，结果为 A－B。

裁剪结果区域的边界由被裁剪多边形的部分边界和裁剪窗口的部分边界两部分构成，并且在交点处边界发生交替，即由被裁剪多边形的边界转至裁剪窗口的边界，或者反之。由于多边形构成一个封闭的区域，所以，如果被裁剪多边形和裁剪窗口有交点，则交点成对出现。这些交点分成两类：一类称“入”点，即被裁剪多边形由此点进入裁剪窗口；一类称“出”点，即被裁剪多边形由此点离开裁剪窗口。

Weiler－Atherton 任意多边形裁剪算法思想可以描述为：假设被裁剪多边形和裁剪窗口的顶点序列都按顺时针方向排列。当两个多边形相交时，交点必然成对出现，其中一个是从被裁剪多边形进入裁剪窗口的交点，称为“入点”，另一个是从被裁剪多边形离开裁剪窗口的交点，称为“出点”。算法从被裁剪多边形的一个入点开始，碰到入点，沿着被裁剪多边形按顺时针方向搜集顶点序列；而当遇到出点时，则沿着裁剪窗口按顺时针方向搜集顶点序列。按上述规则，如此交替地沿着两个多边形的边线行进，直到回到起始点。这时，收集到的全部顶点序列就是裁剪所得的一个多边形。由于可能存在分裂的多边形，因此算法要考虑：将搜集过的入点的入点记号删去，以免重复跟踪。将所有的入点搜集完毕后算法结束。

二、 Weiler－Atherton 算法外裁剪步骤：

1、顺时针输入被裁剪多边形顶点序列I放入数组 1 中。

2、顺时针输入裁剪窗口顶点序列II放入数组 2 中。

3、求出被裁剪多边形和裁剪窗口相交的所有交点，并给每个交点打上“入”、“出”标记。然后将交点按顺序插入序列I得到新的顶点序列III，并放入数组 3 中；同样也将交点按顺序插入序列II得到新的顶点序列IV，放入数组 4 中；

4、初始化输出数组 Q，令数组 Q 为空。接着从数组 3 中寻找“出”点。

如果“出”点没找到，程序结束。

5、如果找到“出”点，则将“出”点放入 S 中暂存。

6、将“出”点录入到输出数组 Q 中。并从数组 3 中将该“出”点的“出”点标记删去。

7、沿数组 3 逆序取顶点：

如果顶点不是“入点”，则将顶点录入到输出数组 Q 中，流程转第 7 步。

否则，流程转第 8 步。

8、沿数组 4 顺序取顶点：

如果顶点不是“出点”，则将顶点录入到输出数组 Q 中，流程转第 8 步。

否则，流程转第 9 步。

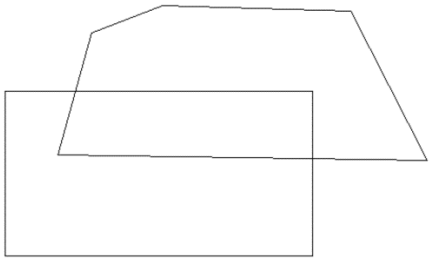
9、如果顶点不等于起始点 S，流程转第 6 步，继续跟踪数组 3。

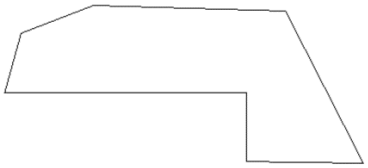
否则，将数组 Q 输出；流程转第 4 步，寻找可能存在的分裂多边形。

三、 算法编码

```
function Mul_Outer_WeilerAtherton(subject, r1, r2) {
    var subjectListSave = new polygonPro();
    var clipListSave = new polygonPro();
    var OutList = new polygonPro;
    //获得含交点的裁剪框序列和多边形顶点序列
    WeilerAtherton_GetList(subject, r1, r2, subjectListSave, clipListSave)
    //内裁剪 从多边形顶点序列的第一个入点开始加入顶点集并消去入点标记 从左至右 遍历将非

    //外裁剪 从裁剪框顶点序列的第一个出点开始加入顶点集并消去出点标记 从左至右 遍历将非
    // ------------------------------2019.05.27_modify-----------------------
        var p = clipListSave.first;
        var flag = false;
        var end = false;
        while (p != null) {
            //找到第一个出点 将其加入顶点集 并消去出点标记
            if (p.point.type == "false") {
                OutList.first = new PolygonNode(p.point);
                p.point.type = null;
                OutList.length++;
                flag = true;
                p = p.next;
                if (p == null) {
                    p = clipListSave.first;
                }
                continue;
            }
            //将出点后的非入点加入输出顶点集
            if (OutList.first) {
                if (p.point.type != "true") {
                    if (flag) {
                        OutList.insertAfter(p.point, OutList.first);
                        flag = false;
                    } else {
                        OutList.insertAfter(p.point, OutList.last);
                    }
                } else {
                    //到多边形顶点序列中找到当前入点位置
                    var substart = subjectListSave.find(p)
                    var q = substart;
                    while (q != null) {
```

四、 实现结果





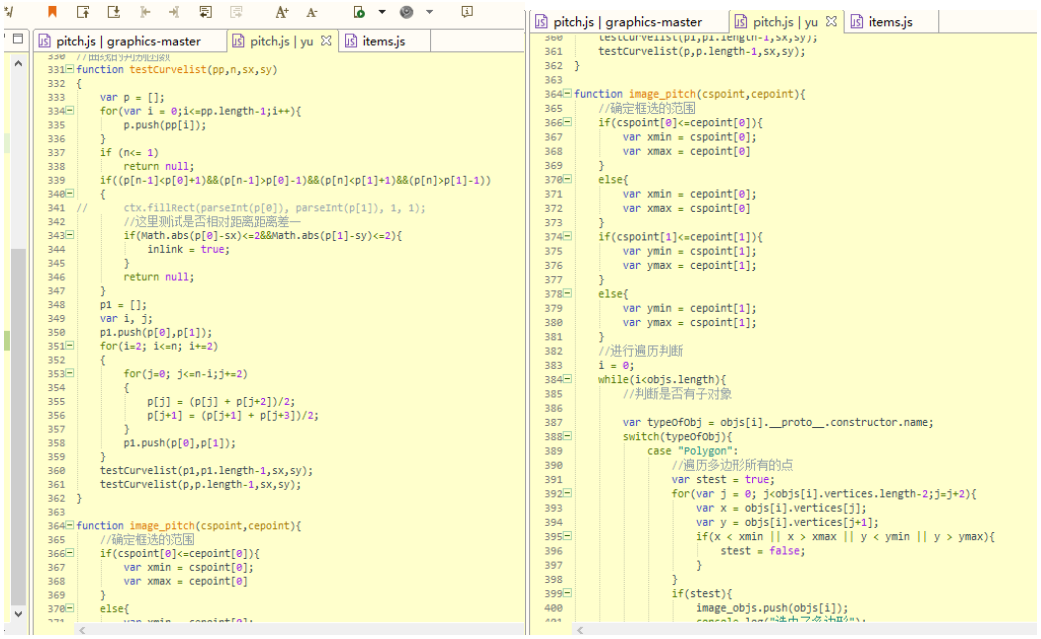
5.选中

鼠标按下与移动，确定框选矩形的范围。遍历所有图元，得到框选矩形范围内的图元列表，返回框选图元列表。

各图元的框选识别方法：

1. 直线--两点在框选矩形范围内。
2. 曲--曲线的绘制点全部在框选矩形范围内。
3. 圆--圆心到框选矩形各边的距离大于半径，且圆心在框选矩形范围内。
4. 椭圆--椭圆的四个关键顶点全部在框选矩形范围内。
5. 多边形--多边形的绘制点全部在框选矩形范围内。
6. 矩形--矩形的四个关键顶点全部在框选矩形范围内。
7. 圆角矩形--圆角矩形的四个关键顶点全部在框选矩形范围内。
8. 直角--两顶点和拐点全部在框选矩形范围内。

代码截图：

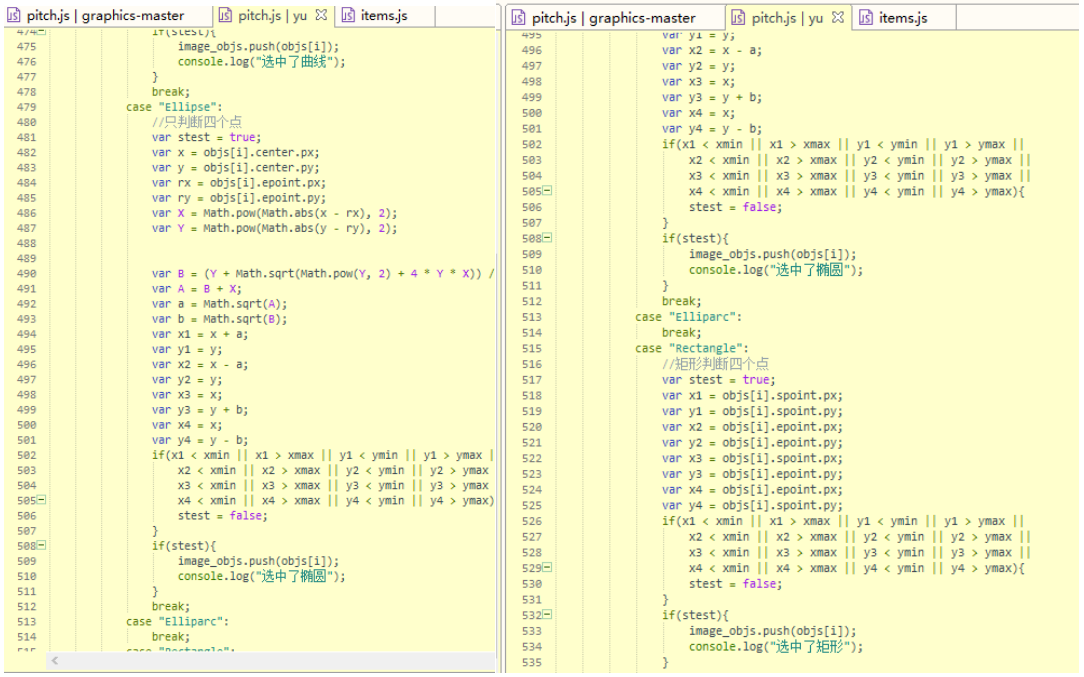


```
pitch.js | graphics-master | pitch.js | yu | items.js

function testCurvelist(pp,n,sx,sy)
{
    var p = [];
    for(var i = 0;i<=pp.length-1;i++){
        p.push(pp[i]);
    }
    if (n<= 1)
        return null;
    if((p[n-1]<p[0]+1)&&(p[n-1]>p[0]-1)&&(p[n]<p[1]+1)&&(p[n]>p[1]-1))
    {
//      ctx.fillRect(parseInt(p[0]), parseInt(p[1]), 1, 1);
        //这里测试是否相对距离距离差一
        if(Math.abs(p[0]-sx)<=2&&Math.abs(p[1]-sy)<=2){
            inlink = true;
        }
        return null;
    }
    p1 = [];
    var i, j;
    p1.push(p[0],p[1]);
    for(i=2; i<=n; i+=2)
    {
        for(j=0; j<=n-i;j+=2)
        {
            p[j] = (p[j] + p[j+2])/2;
            p[j+1] = (p[j+1] + p[j+3])/2;
        }
        p1.push(p[0],p[1]);
    }
    testCurvelist(p1,p1.length-1,sx,sy);
    testCurvelist(p,p.length-1,sx,sy);
}

function image_pitch(cspoint,cepoint){
    //确定框选的范围
    if(cspoint[0]<=cepoint[0]){
        var xmin = cspoint[0];
        var xmax = cepoint[0]
    }
    else{
```

```
pitch.js | graphics-master | pitch.js | yu | items.js

    testCurvelist(p1,p1.length-1,sx,sy);
    testCurvelist(p,p.length-1,sx,sy);
}

function image_pitch(cspoint,cepoint){
    //确定框选的范围
    if(cspoint[0]<=cepoint[0]){
        var xmin = cspoint[0];
        var xmax = cepoint[0]
    }
    else{
        var xmin = cepoint[0];
        var xmax = cspoint[0]
    }
    if(cspoint[1]<=cepoint[1]){
        var ymin = cspoint[1];
        var ymax = cepoint[1];
    }
    else{
        var ymin = cepoint[1];
        var ymax = cspoint[1];
    }
    //进行遍历判断
    i = 0;
    while(i<objs.length){
        //判断是否有子对象

        var typeOfObj = objs[i].__proto__.constructor.name;
        switch(typeOfObj){
            case "Polygon":
                //遍历多边形所有的点
                var stest = true;
                for(var j = 0; j<objs[i].vertices.length-2;j=j+2){
                    var x = objs[i].vertices[j];
                    var y = objs[i].vertices[j+1];
                    if(x < xmin || x > xmax || y < ymin || y > ymax){
                        stest = false;
                    }
                }
                if(stest){
                    image_objs.push(objs[i]);
```

```
pitch.js | graphics-master | pitch.js | yu | items.js

                if(stest){
                    image_objs.push(objs[i]);
                    console.log("选中了多边形");
                }
                break;
            case "Line":
                //直线比较两个点
                var stest = true;
                var x1 = objs[i].spoint.px;
                var y1 = objs[i].spoint.py;
                var x2 = objs[i].epoint.px;
                var y2 = objs[i].epoint.py;
                if(x1 < xmin || x1 > xmax || y1 < ymin || y1 > ymax ||
                    x2 < xmin || x2 > xmax || y2 < ymin || y2 > ymax){
                    stest = false;
                }
                if(stest){
                        image_objs.push(objs[i]);
                        console.log("选中了直线");
                }
                break;
            case "Brokenline":
                break;
            case "Rightangle":
                //直角判断三个点
                var stest = true;
                var x1 = objs[i].spoint.px;
                var y1 = objs[i].spoint.py;
                var x2 = objs[i].epoint.px;
                var y2 = objs[i].epoint.py;
                var x3 = objs[i].spoint.px;
                var y3 = objs[i].epoint.py;
                if(x1 < xmin || x1 > xmax || y1 < ymin || y1 > ymax ||
                    x2 < xmin || x2 > xmax || y2 < ymin || y2 > ymax ||
                    x3 < xmin || x3 > xmax || y3 < ymin || y3 > ymax){
                    stest = false;
                }
                if(stest){
                        image_objs.push(objs[i]);
                        console.log("选中了直角");
                }
                break;
```

```
pitch.js | graphics-master | pitch.js | yu | items.js

                }
                break;
            case "Circle":
                //判断圆心到四条边的距离
                var stest = false;
                var x = objs[i].center.px;
                var y = objs[i].center.py;
                var r = objs[i].radius;
                var dup = odistance_4(x,y,x,ymin);
                var ddown = odistance_4(x,y,x,ymax);
                var dleft = odistance_4(x,y,xmin,y);
                var dright = odistance_4(x,y,xmax,y);
                if(dup >= r && ddown >= r && dleft >= r && dright >= r&&
                    x > xmin && x < xmax && y > ymin && y < ymax){
                    stest = true;
                }
                if(stest){
                    image_objs.push(objs[i]);
                    console.log("选中了圆形");
                }
                break;
            case "Curve":
                //曲线遍历所有点
                var stest = true;
                var pointlist = objs[i].plist;
                console.log(pointlist);
                for(var k = 0;k<pointlist.length;k++){
                    for(var j = 0; j<pointlist[k].length;j=j+2){
                        var x = pointlist[k][j];
                        var y = pointlist[k][j+1];
                        if(x < xmin || x > xmax || y < ymin || y > ymax){
                            stest = false;
                        }
                    }
                }
                if(stest){
                    image_objs.push(objs[i]);
                    console.log("选中了曲线");
                }
                break;
```



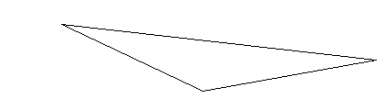
```
pitch.js | graphics-master | pitch.js | yu | items.js

                if(stest){
                    image_objs.push(objs[i]);
                    console.log("选中了曲线");
                }
                break;
            case "Ellipse":
                //只判断四个点
                var stest = true;
                var x = objs[i].center.px;
                var y = objs[i].center.py;
                var rx = objs[i].epoint.px;
                var ry = objs[i].epoint.py;
                var X = Math.pow(Math.abs(x - rx), 2);
                var Y = Math.pow(Math.abs(y - ry), 2);


                var B = (Y + Math.sqrt(Math.pow(Y, 2) + 4 * Y * X)) /
                var A = B + X;
                var a = Math.sqrt(A);
                var b = Math.sqrt(B);
                var x1 = x + a;
                var y1 = y;
                var x2 = x - a;
                var y2 = y;
                var x3 = x;
                var y3 = y + b;
                var x4 = x;
                var y4 = y - b;
                if(x1 < xmin || x1 > xmax || y1 < ymin || y1 > ymax |
                    x2 < xmin || x2 > xmax || y2 < ymin || y2 > ymax
                    x3 < xmin || x3 > xmax || y3 < ymin || y3 > ymax
                    x4 < xmin || x4 > xmax || y4 < ymin || y4 > ymax)
                    stest = false;
                }
                if(stest){
                    image_objs.push(objs[i]);
                    console.log("选中了椭圆");
                }
                break;
            case "Elliparc":
                break;
```

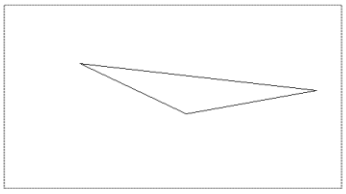
```
pitch.js | graphics-master | pitch.js | yu | items.js

                var y1 = y;
                var x2 = x - a;
                var y2 = y;
                var x3 = x;
                var y3 = y + b;
                var x4 = x;
                var y4 = y - b;
                if(x1 < xmin || x1 > xmax || y1 < ymin || y1 > ymax ||
                    x2 < xmin || x2 > xmax || y2 < ymin || y2 > ymax ||
                    x3 < xmin || x3 > xmax || y3 < ymin || y3 > ymax ||
                    x4 < xmin || x4 > xmax || y4 < ymin || y4 > ymax){
                    stest = false;
                }
                if(stest){
                    image_objs.push(objs[i]);
                    console.log("选中了椭圆");
                }
                break;
            case "Elliparc":
                break;
            case "Rectangle":
                //矩形判断四个点
                var stest = true;
                var x1 = objs[i].spoint.px;
                var y1 = objs[i].spoint.py;
                var x2 = objs[i].epoint.px;
                var y2 = objs[i].epoint.py;
                var x3 = objs[i].spoint.px;
                var y3 = objs[i].epoint.py;
                var x4 = objs[i].epoint.px;
                var y4 = objs[i].spoint.py;
                if(x1 < xmin || x1 > xmax || y1 < ymin || y1 > ymax ||
                    x2 < xmin || x2 > xmax || y2 < ymin || y2 > ymax ||
                    x3 < xmin || x3 > xmax || y3 < ymin || y3 > ymax ||
                    x4 < xmin || x4 > xmax || y4 < ymin || y4 > ymax){
                    stest = false;
                }
                if(stest){
                    image_objs.push(objs[i]);
                    console.log("选中了矩形");
                }
```

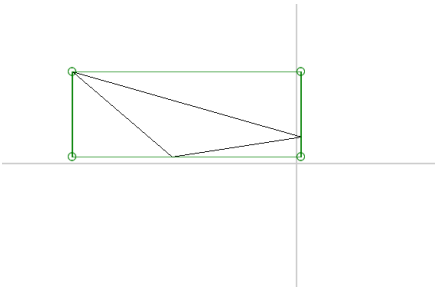
pitch.js | graphics-master　pitch.js | yu　items.js

```
            }
            break;
        case "Roundedrectangle":
            //矩形判断四个点
            var stest = true;
            var x1 = objs[i].spoint.px;
            var y1 = objs[i].spoint.py;
            var x2 = objs[i].epoint.px;
            var y2 = objs[i].epoint.py;
            var x3 = objs[i].spoint.px;
            var y3 = objs[i].epoint.py;
            var x4 = objs[i].epoint.px;
            var y4 = objs[i].spoint.py;
            if(x1<x2){
                x1 = x1-10;
                x2 = x2+10;
            }
            else{
                x1 = x1+10;
                x2 = x2-10;
            }
            if(y1<y2){
                y1 = y1-10;
                y2 = y2+10;
            }
            else{
                y1 = y1+10;
                y2 = y1-10;
            }
            if(x1 < xmin || x1 > xmax || y1 < ymin || y1 > ymax ||
                x2 < xmin || x2 > xmax || y2 < ymin || y2 > ymax ||
                x3 < xmin || x3 > xmax || y3 < ymin || y3 > ymax ||
                x4 < xmin || x4 > xmax || y4 < ymin || y4 > ymax){
                stest = false;
            }
            if(stest){
                image_objs.push(objs[i]);
                console.log("选中圆角了矩形");
            }
            break;
```

操作截图：







6.图形变换

以上操作均在已有图元并处于选中状态的基础上进行。

6.1 对称：

关于 X 轴对称：选中图元后按下 ctrl/cmd+x；
关于 Y 轴对称：选中图元后按下 ctrl/cmd+y；
关于原点对称：选中图元后按下 ctrl/cmd+o；

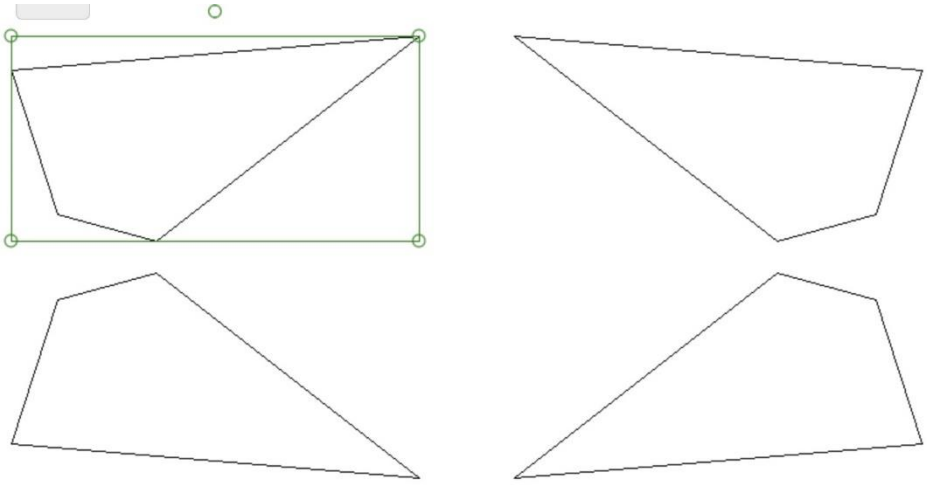
主要代码：

```
// cmd/ctrl + x 快捷键 ： 关于 x 轴对称
    if(ctrlKey && keyCode == 88) {
        // alert('x');
        e.preventDefault();
        if(option.obj){
            const obj = option.obj;
            // 获取当前选中图元的原型名称
            const kindOfObj = obj.__proto__.constructor.name;
            console.log(kindOfObj);

            switch (kindOfObj){
                case "Circle":
                    console.log(obj);
                    break;
                case "Polygon":
                    console.log(obj);
                    var copy = new Polygon();
                    var vertices = [];
                    for(var i=0; i<obj.vertices.length; i+=2){
                        // 这里是关于 x 轴对称，改变 y 坐标,x 不变
                        vertices.push(obj.vertices[i])
                        var ty = trans(obj.vertices[i+1],'x');
                        vertices.push(ty);
                    }
                    copy.vertices = vertices;
                    copy.draw();
                    objs.push(copy);
                    break;
                default:
                    alert("没有选中图元");

            }
        }
    }
    // cmd/ctrl + y 快捷键 ： 关于 y 轴对称
    if(ctrlKey && keyCode == 89){
        // alert('y');
        e.preventDefault();
        if(option.obj){
            const obj = option.obj;
            // 获取当前选中图元的原型名称
            const kindOfObj = obj.__proto__.constructor.name;
            console.log(kindOfObj);
```

```
        switch (kindOfObj){
            case "Circle":
                console.log(obj);
                break;
            case "Polygon":
                console.log(obj);
                var copy = new Polygon();
                var vertices = [];
                for(var i=0; i<obj.vertices.length; i+=2){
                    // 这里是关于 y 轴对称，改变 x 坐标
                    var tx = trans(obj.vertices[i],'y');
                    vertices.push(tx);
                    // y 轴不变
                    vertices.push(obj.vertices[i+1])
                }
                copy.vertices = vertices;
                copy.draw();
                objs.push(copy);
                break;
            default:
                alert("没有选中图元");

        }
    }
}

// cmd/ctrl + o 快捷键 ： 关于原点对称
if(ctrlKey && keyCode == 79){
    // alert('o');
    e.preventDefault();
    if(option.obj){
        const obj = option.obj;
        // 获取当前选中图元的原型名称
        const kindOfObj = obj.__proto__.constructor.name;
        console.log(kindOfObj);

        switch (kindOfObj){
            case "Circle":
                console.log(obj);
                break;
            case "Polygon":
                console.log(obj);
                var copy = new Polygon();
                var vertices = [];
```

```
            for(var i=0; i<obj.vertices.length; i+=2){
              // 这里是关于原点对称，改变 x 坐标和 y 坐标
              var tx = trans(obj.vertices[i],'y');
              var ty = trans(obj.vertices[i+1],'x');
              vertices.push(tx);
              // y 轴不变
              vertices.push(ty);
            }
            copy.vertices = vertices;
            copy.draw();
            objs.push(copy);
            break;
          default:
            alert("没有选中图元");

        }
      }
    }
```

实现结果：



6.2 旋转

选中图元后，移动鼠标到旋转热点（位于选中框上方的小绿圈），按住鼠标左键并拖动，图形即可进行随机不规则变化型旋转。

6.3 移动

自由移动：选中图元后按住鼠标左键并拖动；
水平移动：选中图元后按住 x 和鼠标左键并拖动；
竖直移动：选中图元后按住 y 和鼠标左键并拖动；

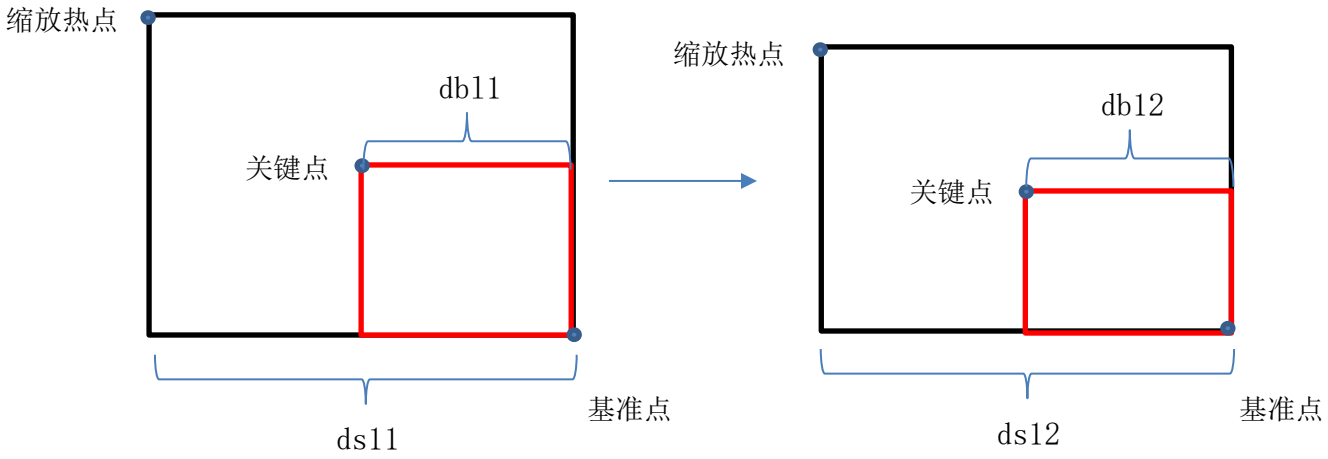
主要代码：

```
/**
    * @desc 选中并自由移动
    * @author kf
    * @date 2019.5.23
    */
   if (option.name == "pitch" && down_flag) {
      // console.log("移动");
      // 计算移动距离
      // console.log(downPoint);
      var px = e.clientX - of_left;
      var py = e.clientY - of_top;
      var dx = px - downPoint[0];
      var dy = py - downPoint[1];

      // 如果是多边形，将多边形顶点存入当前操作对象，深拷贝
      if (option.obj instanceof Polygon) {
         // 起始位置
         option.s_vertices = option.obj.vertices.slice(0);
      }
      // 当前操作为选中
      if (option.name == "pitch" && option.obj) {
         // 将多边形数组的值加上移动的距离
         for (var i = 0; i < option.obj.vertices.length; i += 2) {
            option.obj.vertices[i] += dx;
            option.obj.vertices[i + 1] += dy;
            // 更新边界点
            option.obj.limit_4 = getLimit_4(option.obj);
            // 根据边界点更新选中框
            option.obj.chooseRectangle = getChooseRec(option.obj.limit_4);
         }

         rePaint();
         paintChooseRec(option.obj);
```

```
            // 将移动后的终点位置深拷贝给 e_vertices
            option.e_vertices = option.obj.vertices.slice(0);
            option.obj.vertices = option.s_vertices;
            // console.log(option);
        }
    }
```

6.4 多边形缩放

每一个多边形是以多个关键点按顺序连接绘制的，为达到多边形缩放，需要得到每一个关键点缩放后的坐标，在此基础上定一个缩放规则：每个多边形具有一个矩形选中框，矩形选中框的顶点为缩放热点，选中的缩放热点对称点为基准点，以关键点，缩放热点和基准点构建出关键点-基准点矩形，缩放热点-基准点矩形两个矩形，这两个矩形的对应边比是恒定不变的，以此用来求解缩放点坐标。



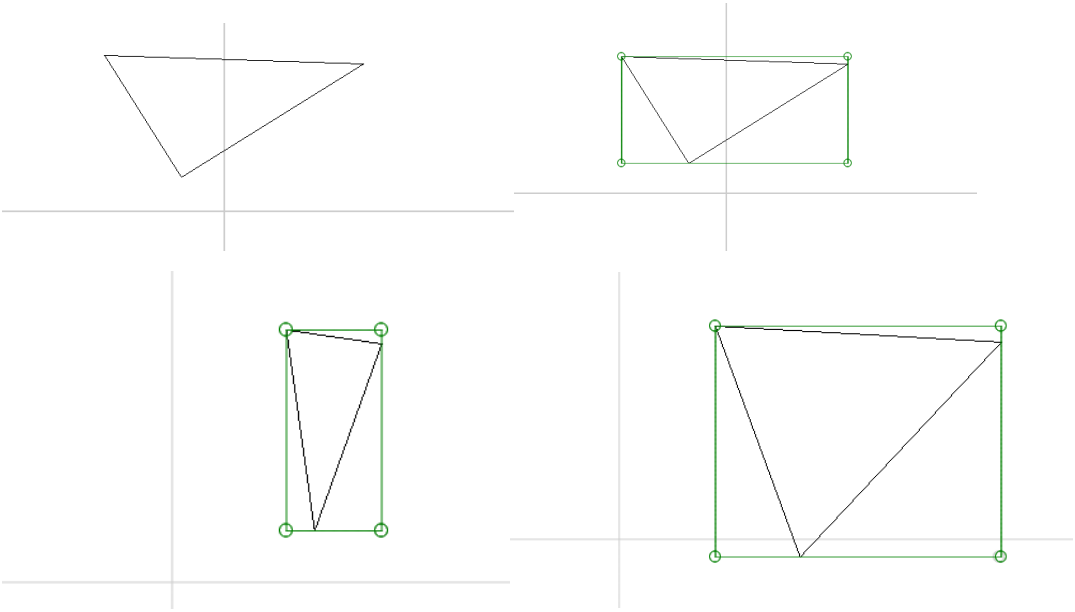
$$\frac{dbl1}{dsl1} = \frac{dbl2}{dsl2}$$

代码截图：

pitch.js | graphics-master | pitch.js | yu | items.js | mouseDown.js | scaling

```
//用于多边形缩放的容点比例求解
//参数：缩放点的初始坐标，缩放点的结束坐标，缩放的基本点坐标，缩放对象的某个关键点
//返回：缩放对象的某个关键点缩放后的新坐标

function polygon_scaling_ratio(sx,sy,ex,ey,bx,by,kx,ky){

//  console.log(sx,sy,ex,ey,bx,by,kx,ky);
    //缩放后的坐标
    var x = 0;
    var y = 0;

    //得到缩放前对象的包裹矩形框的一些基本值
    //长宽
    var sweith = (Math.abs(sx - bx));
    var sheight = (Math.abs(sy - by));

    //得到缩放后对象的包裹矩形框的一些基本值
    //长宽
    var eweith = (Math.abs(ex - bx));
    var eheight = (Math.abs(ey - by));

    //得到缩放前对象的某个关键点与缩放的基本点构成的矩形框的一些基本值
    //长宽
    var bweith = (Math.abs(bx - kx));
    var bheight = (Math.abs(by - ky));

    //得到缩放后对象的某个关键点与缩放的基本点构成的矩形框的一些基本值
    var kweith = (eweith*bweith/sweith);
    var kheight = (eheight*bheight/sheight);
//  console.log(kweith,kheight)

    //计算变化量
    //要考虑缩放点的初始坐标与缩放的基本点坐标
    //要考虑缩放方向
    //左上到右下
    if(sx < bx && sy < by){
        if(sx != ex){
            if(ex < bx){
                x = bx - kweith;
            }
            else{
                x = kx;
            }
        }
        if(sy != ey){
            if(ey < by){
                y = by - kheight;
            }
            else{
                y = by;
            }
        }
        else{
            y = ky;
        }
    }
    //右上到左下
    else if(bx < sx && sy < by){
        if(ex != sx){
            if(bx < ex){
                x = bx + kweith;
            }
            else{
                x = bx;
            }
        }
        else{
            x = ky;
        }
        if(sy != ey){
            if(ey < by){
                y = by - kheight;
            }
            else{
                y = by;
            }
        }
        else{
            y = ky;
        }
    }
    //右下到左上
    else if(bx < sx && by < sy){
```

操作截图：



点击选中思路：

鼠标点击获取点击位置坐标建立识别点。遍历所有图元，得到识别点在里面的图元列表，判断列表中各图元的图层编号大小，图层编号大小越大选中优先度越高，返回优先度最高的图元。

各图元判断识别点在里面的方法：

1. 直线--建立两点方程式，使用方程判别。
2. 曲线--多点模拟绘制曲线，逐点判别。
3. 圆--建立圆的标准方程式，使用方程判断。
4. 椭圆--建立椭圆标准方程式，使用判别式判断。
5. 多边形--以识别点的 y 坐标建立射线，用射线判别法判断。
6. 矩形--以识别点的 y 坐标建立射线，用射线判别法判断。

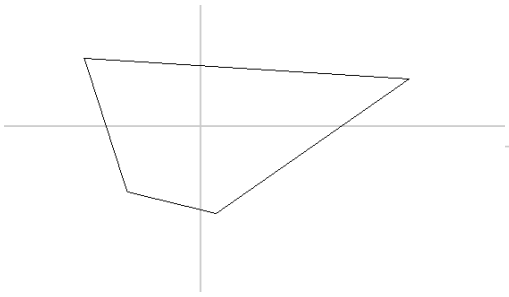
7. 圆角矩形--以识别点的 y 坐标建立射线，用射线判别法判断。
8. 直角--建立两顶点与拐点的两点方程式，使用方程判别。

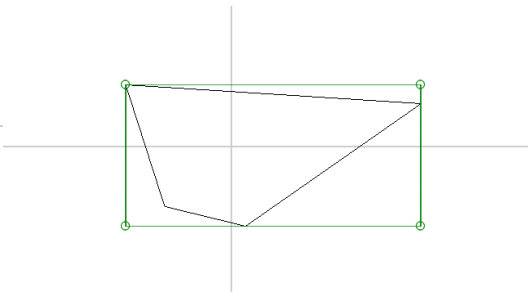
代码截图：

```
}
//未选中绘制类型，按左键走这里
if (click && e.button == 0){
    //进行图形选中判断
    //满足选中条件的图形
    var selectobj = [];
    //获取当前鼠标的坐标
    var sx = e.clientX - of_left;
    var sy = e.clientY - of_top;
    //写入状态点
    cspoint[0] = sx;
    cspoint[1] = sy;
    //进入框选状态
    choice = true;
    console.log("当前坐标: "+sx+"_"+sy);
    //遍历同一图层所有已绘制图形对象
    i = 0;
    while(i<objs.length){
        //这里先判别一下如果存在子对象

        var typeOfObj = objs[i].__proto__.constructor.name;
        switch(typeOfObj){
            case "Polygon":
                var tobj = Polygonpitch(sx,sy,objs[i]);
                if(tobj != null){
                    selectobj.push(tobj);
                }
                break;
            case "Line":
                var tobj = StraightLinepitch(sx,sy,objs[i]);
                if(tobj != null){
                    selectobj.push(tobj);
                }
                break;
            case "Brokenline":
```

```
                    selectobj.push(tobj);
                }
                break;
            case "Ellipsarc":
                var tobj = Ellipsarcpitch(sx,sy,objs[i]);
                if(tobj != null){
                    selectobj.push(tobj);
                }
                break;
            case "Rectangle":
                var tobj = Rectanglepitch(sx,sy,objs[i]);
                if(tobj != null){
                    selectobj.push(tobj);
                }
                break;
            case "Roundedrectangle":
                var tobj = Roundedrectanglepitch(sx,sy,objs[i]);
                if(tobj != null){
                    selectobj.push(tobj);
                }
                break
            case "Positivearc":
                var tobj = Positivearcpitch(sx,sy,objs[i]);
                if(tobj != null){
                    selectobj.push(tobj);
                }
                break;
        }
        i++;
    }
    //比较获取对象的图层
    var maxlevel = 0;
    var maxobj = null;
    if(selectobj.length != 0){
        for(var i=0;i<selectobj.length;i++){
            //遭到最大图层
            if(maxlevel<=selectobj[i].level){
                maxlevel = selectobj[i].level;
                maxobj = selectobj[i];
            }
        }
```

操作截图：

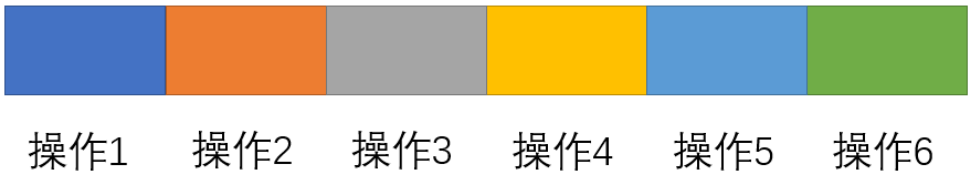




7 撤销、重复操作

7.1 撤销

撤销操作主要针对当前画布上的上一步操作进行退回，为实现该功能，我们将对 web 页面上的画布上的操作均当作操作对象，存储在一个全局的操作对象数组中，通过对操作对象数组的相关操作实现画布的撤销功能。



实现代码：

```
var objs = [];
//撤销函数
function revocation() {
    objs.pop();//移除最后一个元素
    rePaint();//画布重绘
}
```

结果截图

7.2 重复

重复操作和撤销操作的原理比较相同，在全局操作对象数组中，通过获取最后一次操作对象，并复制，加入对应的操作对象数组，实现重复操作。

实现代码：

```
function repeat() {
    //获取对象数组中的最后一个对象
     var objnew = objs[objs.length - 1];
    //对象向右下方移动 10 个像素
    //获取数组中最后一个元素的类型
```

```
var name = objnew.__proto__.constructor.name;
switch(name){
    //矩形
    case "Rectangle":
    //圆角矩形
    case "Roundedrectangle":
    //直角矩形
    case "Rightangle":
    //直线
    case "Line":
        objnew.spoint.px = objnew.spoint.px + 10;
        objnew.spoint.py = objnew.spoint.py + 10;
        objnew.epoint.px = objnew.epoint.px + 10;
        objnew.epoint.py = objnew.epoint.py + 10;
        break;
    //虚线
    case "Brokenline":

        break;
    //多边形
    case "Polygon":
        for (var j = 0; j < objnew.vertices.length; j += 2) {
            objnew.vertices[j] = objnew.vertices[j] + 10;
            objnew.vertices[j+1] = objnew.vertices[j+1] + 10;
        }
        break;
    //曲线
    case "Curve":

        break;
    //椭圆
    case "Ellipse":
    //椭圆弧
    case "Elliparc":

        objnew.center.px = objnew.center.px + 10;
        objnew.center.py = objnew.center.py + 10;
        objnew.epoint.px = objnew.epoint.px + 10;
        objnew.epoint.py = objnew.epoint.py + 10;
        break;
    //圆
    case "Circle":
    //正圆弧
    case "Positivearc":
```

```
                objnew.center.px = objnew.center.px + 10;
                objnew.center.py = objnew.center.py + 10;
                break;
        }

        //在对象数组最后添加一个对象
        objs.push(objnew);
        rePaint();//画布重绘
    }
```

结果截图：

8 对齐、橡皮擦

8.1 对齐

上对齐：记录所有图元中的最上点 max，让其他图元的最上点等于 max。

下对齐：记录所有图元中的最下点 max，让其他图元的最下点等于 max。

左对齐：记录所有图元中的最左点 max，让其他图元的最右点等于 max。

右对齐：记录所有图元中的最右点 max，让其他图元的最右点等于 max。

水平居中：记录所有图元中的最左点 max 和最右点 min，计算 max 和 min 的中点 middle，让其他图元的水平中点等于 middle。

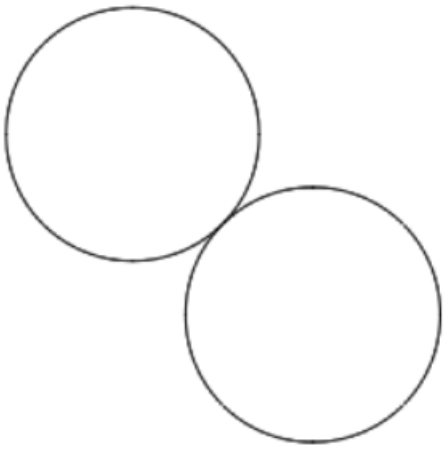
垂直居中：记录所有图元中的最上点 max 和最下点 min，计算 max 和 min 的中点 middle，让其他图元的垂直中点等于 middle。

主要代码：

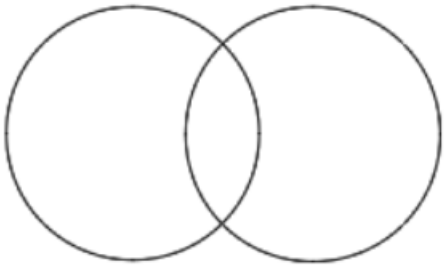
```
function align(direction) {
    var xuayaoyidongzuo=option.objs[0].limit_4[0]>option.objs[1].limit_4[0]?option.objs[0]:option.objs[1];
    var xuayaoyidongyou=option.objs[0].limit_4[1]>option.objs[1].limit_4[1]?option.objs[1]:option.objs[0];
    var xuayaoyidongshang=option.objs[0].limit_4[2]>option.objs[1].limit_4[2]?option.objs[0]:option.objs[1];
    var xuayaoyidongxia=option.objs[0].limit_4[3]>option.objs[1].limit_4[3]?option.objs[1]:option.objs[0];
    var zuizuo=option.objs[0].limit_4[0]>option.objs[1].limit_4[0]?option.objs[1].limit_4[0]:option.objs[0].limit_4[0];
    var zuiyou=option.objs[0].limit_4[1]>option.objs[1].limit_4[1]?option.objs[0].limit_4[1]:option.objs[1].limit_4[1];
    var zuishang=option.objs[0].limit_4[2]>option.objs[1].limit_4[2]?option.objs[1].limit_4[2]:option.objs[0].limit_4[2];
    var zuixia=option.objs[0].limit_4[3]>option.objs[1].limit_4[3]?option.objs[0].limit_4[3]:option.objs[1].limit_4[3];
    var zuocha=Math.abs(option.objs[0].limit_4[0]-option.objs[1].limit_4[0]);
    var youcha=Math.abs(option.objs[0].limit_4[1]-option.objs[1].limit_4[1]);
    var shangcha=Math.abs(option.objs[0].limit_4[2]-option.objs[1].limit_4[2]);
    var xiacha=Math.abs(option.objs[0].limit_4[3]-option.objs[1].limit_4[3]);
    switch (direction) {
        case 'top':
            if(option.objs[0] instanceof Circle)
                xuayaoyidongshang.center.py=xuayaoyidongshang.center.py-shangcha;
            rePaint();
            break;
        case 'bottom':
            if(option.objs[0] instanceof Circle)
                xuayaoyidongxia.center.py=xuayaoyidongxia.center.py+xiacha;
            rePaint();
```

```
                break;
            case 'left':
                if(option.objs[0] instanceof Circle)

                    xuayaoyidongzuo.center.px=xuayaoyidongzuo.center.px-
zuocha;
                rePaint();
                break;
            case 'right':
                if(option.objs[0] instanceof Circle)

xuayaoyidongyou.center.px=xuayaoyidongyou.center.px+youcha;
                rePaint();
                break;
            case 'center':
                if(option.objs[0] instanceof Circle)
                    var zhongdiancc=Math.abs(zuishang+zuixia)/2.0;
                    option.objs[0].center.py=zhongdiancc;
                    option.objs[1].center.py=zhongdiancc;
                rePaint();
                break;
            case 'pingfen':
                if(option.objs[0] instanceof Circle)
                    var chuzhongdiancc=Math.abs(zuizuo+zuiyou)/2.0;
                    option.objs[0].center.px=chuzhongdiancc;
                    option.objs[1].center.px=chuzhongdiancc;
                rePaint();
                break;
        }

}
```

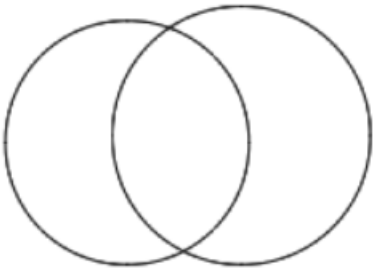
实现结果：



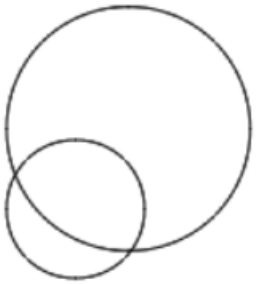
上对齐：



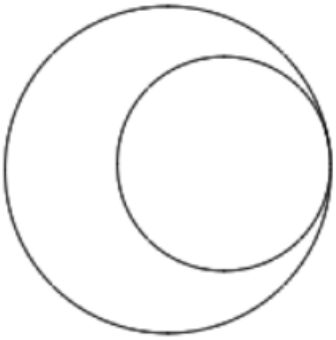
下对齐：



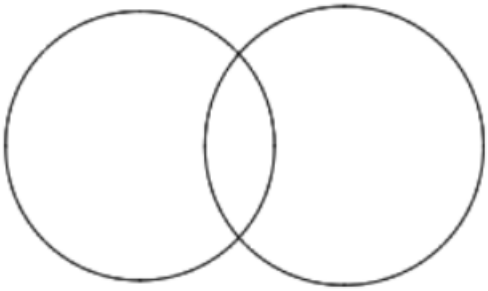
左对齐：



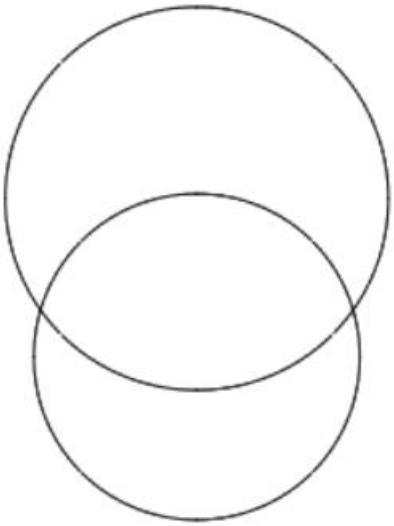
右对齐：



水平居中：



垂直居中：



8.2 橡皮擦

橡皮擦用填充背景色来实现，先获取背景色的值。然后当用户鼠标在画布上点击拖动时，获取点击拖动点的坐标，用背景颜色填充该像素点即可。

实现代码：

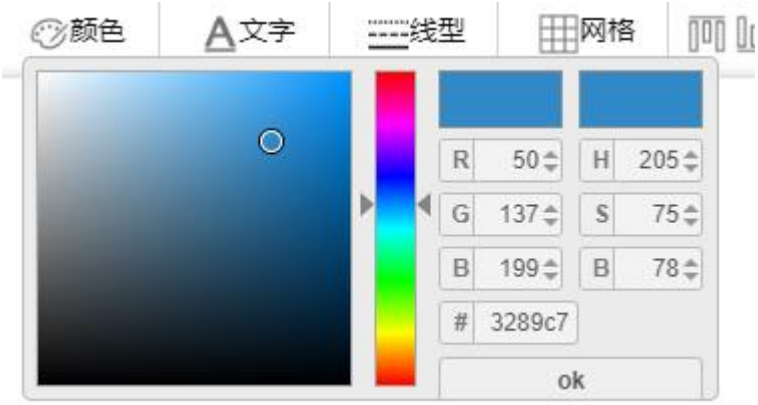
```
function Rubber(color,len) {
    rubberlength=5;
    Drawable.apply(this);//继承父类
    this.draw = function () {
        rubberlist(this.vertices,this.vertices.length-1,color,len);
    }
}

function rubberlist(pp,n,color,len)
{
    var p = [];
    for(var i = 0;i<=pp.length-1;i++){
        p.push(pp[i]);
    }
    if (n<= 1)
        return null;
    if((p[n-1]<p[0]+1)&&(p[n-1]>p[0]-1)&&(p[n]<p[1]+1)&&(p[n]>p[1]-1))
    {
        ctx.fillStyle=color;
        ctx.fillRect(parseInt(p[0]), parseInt(p[1]), len, len);
        return null;
    }
    p1 = [];
    var i, j;
    p1.push(p[0],p[1]);
    for(i=2; i<=n; i+=2)
    {
        for(j=0; j<=n-i;j+=2)
        {
            p[j] = (p[j] + p[j+2])/2;
            p[j+1] = (p[j+1] + p[j+3])/2;
        }
        p1.push(p[0],p[1]);
    }
    rubberlist(p1,p1.length-1,color,len);
    rubberlist(p,p.length-1,color,len);
}
```

创建一个 Rubber 对象，监听用户的点击事件和拖动事件，记录点击点和拖动点，当事件发生时，用背景色填充点击点和拖动点。

操作方法：先用其他笔画图，再点击橡皮擦，拖动鼠标经过想要擦除的点，就能实现背景色的填充，即橡皮擦。

9 画布背景、导向线、标尺

9.1 背景颜色设置



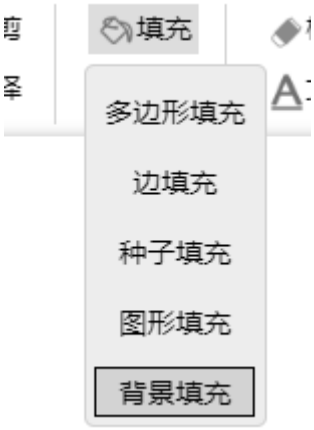
调用颜色拾取器：

```
setSelector = function (hsx, cal) {
    $(cal).data('colpick').selector.css('backgroundColor', '#' +
```

```
($(cal).data('colpick').hsl ? hslToHex({h:hsx.h,s:100,x:50}) :
hsbToHex({h:hsx.h,s:100,x:100})) );
                $(cal).data('colpick').selectorIndic.css({
                    left: parseInt($(cal).data('colpick').height *
hsx.s/100, 10),
                    top: parseInt($(cal).data('colpick').height * (100-
hsx.x)/100, 10)
                });
            },
            //Set the hue selector position
            setHue = function (hsx, cal) {
                $(cal).data('colpick').hue.css('top',
parseInt($(cal).data('colpick').height - $(cal).data('colpick').height *
hsx.h/360, 10));
            },
            //Set current and new colors
            setCurrentColor = function (hsx, cal) {
                $(cal).data('colpick').currentColor.css('backgroundColor',
'#' + ($(cal).data('colpick').hsl ? hslToHex(hsx) : hsbToHex(hsx)) );
            },
            setNewColor = function (hsx, cal) {
                $(cal).data('colpick').newColor.css('backgroundColor', '#' +
($(cal).data('colpick').hsl ? hslToHex(hsx) : hsbToHex(hsx)) );
            },
            //Called when the new color is changed
    //Color space convertions
    var hexToRgb = function (hex) {
        var hex = parseInt(((hex.indexOf('#') > -1) ? hex.substring(1) :
hex), 16);
        return {r: hex >> 16, g: (hex & 0x00FF00) >> 8, b: (hex &
0x0000FF)};
    };
    var hexToHsb = function (hex) {
        return rgbToHsb(hexToRgb(hex));
    };
    var hexToHsl = function (hex) {
        return rgbToHsl(hexToRgb(hex));
    };
    var rgbToHsb = function (rgb) {
        var hsb = {h: 0, s: 0, x: 0};
        var min = Math.min(rgb.r, rgb.g, rgb.b);
        var max = Math.max(rgb.r, rgb.g, rgb.b);
        var delta = max - min;
        hsb.x = max;
```

```
    hsb.s = max != 0 ? 255 * delta / max : 0;
    if (hsb.s != 0) {
        if (rgb.r == max) hsb.h = (rgb.g - rgb.b) / delta;
        else if (rgb.g == max) hsb.h = 2 + (rgb.b - rgb.r) / delta;
        else hsb.h = 4 + (rgb.r - rgb.g) / delta;
    } else hsb.h = -1;
    hsb.h *= 60;
    if (hsb.h < 0) hsb.h += 360;
    hsb.s *= 100/255;
    hsb.x *= 100/255;
    return hsb;
};
var rgbToHsl = function (rgb) {
    return hsbToHsl(rgbToHsb(rgb));
};
var hsbToHsl = function(hsb) {
    var hsl = {h: 0, s: 0, x: 0};
    hsl.h = hsb.h;
    hsl.x = hsb.x*(200-hsb.s)/200;
    hsl.s = hsb.x*hsb.s/(100-Math.abs(2*hsl.x-100));
    return hsl;
};
var hslToHsb = function(hsl) {
    var hsb = {h: 0, s: 0, x: 0};
    hsb.h = hsl.h;
    hsb.x = (200*hsl.x + hsl.s*(100-Math.abs(2*hsl.x-100)))/200
    hsb.s = 200*(hsb.x-hsl.x)/hsb.x;
    return hsb;
};
var hsbToRgb = function (hsb) {
    var rgb = {};
    var h = hsb.h;
    var s = hsb.s*255/100;
    var v = hsb.x*255/100;
    if(s == 0) {
        rgb.r = rgb.g = rgb.b = v;
    } else {
        var t1 = v;
        var t2 = (255-s)*v/255;
        var t3 = (t1-t2)*(h%60)/60;
        if(h==360) h = 0;
        if(h<60) {rgb.r=t1;rgb.b=t2; rgb.g=t2+t3}
        else if(h<120) {rgb.g=t1; rgb.b=t2;    rgb.r=t1-t3}
        else if(h<180) {rgb.g=t1; rgb.r=t2;    rgb.b=t2+t3}
```

```
            else if(h<240) {rgb.b=t1; rgb.r=t2;    rgb.g=t1-t3}
            else if(h<300) {rgb.b=t1; rgb.g=t2;    rgb.r=t2+t3}
            else if(h<360) {rgb.r=t1; rgb.g=t2;    rgb.b=t1-t3}
            else {rgb.r=0; rgb.g=0;    rgb.b=0}
        }
        return {r:Math.round(rgb.r), g:Math.round(rgb.g),
b:Math.round(rgb.b)};
    };
    var hslToRgb = function(hsl) {
        return hsbToRgb(hslToHsb(hsl));
    };
    var rgbToHex = function (rgb) {
        var hex = [
            rgb.r.toString(16),
            rgb.g.toString(16),
            rgb.b.toString(16)
        ];
        $.each(hex, function (nr, val) {
            if (val.length == 1) {
                hex[nr] = '0' + val;
            }
        });
        return hex.join('');
    };
    var hsbToHex = function (hsb) {
        return rgbToHex(hsbToRgb(hsb));
    };
    var hslToHex = function (hsl) {
        return hsbToHex(hslToHsb(hsl));
    };
```

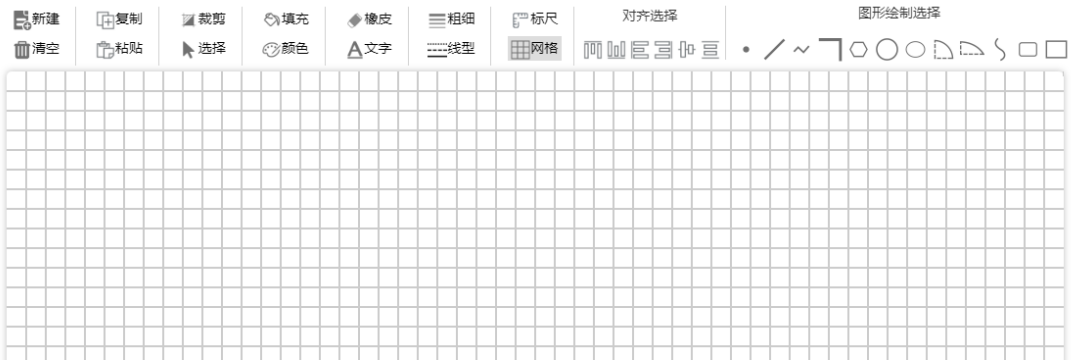


在填充按钮下设置背景按钮：

```
<button id="fill5" class="button2" style="width:80px;height:25px;margin-top: 10px;border-
```

style:none;background-color:#ededed;">背景填充</button>

9.2 网格




```
//横向网格
function drawVerticalAxisTicks1() {
    var deltaY;
    for (var i = 1; i < AXIS_HEIGHT/4; ++i) {
          if (i % 5 === 0) {
             deltaX = TICK_WIDTH;
             ctx2.moveTo(0, 4*i);
             ctx2.lineTo(1080, 4*i);
//             ctx1.textAlign = 'left';
//             ctx1.fillText(i * HORIZONTAL_TICK_SPACING, 10, 0 +
HORIZONTAL_TICK_SPACING * i);
        }else {
             deltaX = TICK_WIDTH/2;
        }
        ctx2.moveTo(AXIS_ORIGIN.x, i * VERTICAL_TICK_SPACING);
        ctx2.lineTo(AXIS_ORIGIN.x + deltaX, i * VERTICAL_TICK_SPACING);
        ctx2.stroke();
    }
}


//纵向网格
function drawHorizontalAxisTicks1() {
```

```
    var deltaY;
    for (var i=1; i < AXIS_WIDTH/4; ++i) {
        if (i % 5 === 0) {
            deltaY = TICK_WIDTH;

//设置网格宽度
//一条一条画网格
           ctx2.moveTo(4* i, 0);
           ctx2.lineTo(4 * i, 540);
//          ctx2.textAlign = 'left';
//          ctx2.fillText(i * VERTICAL_TICK_SPACING, 0 +
VERTICAL_TICK_SPACING * i, 20);
         }else {
           deltaY = TICK_WIDTH/2;
         }
        ctx2.moveTo(AXIS_ORIGIN.x + i * HORIZONTAL_TICK_SPACING,
AXIS_MARGIN);
        ctx2.lineTo(AXIS_ORIGIN.x + i * HORIZONTAL_TICK_SPACING,
AXIS_MARGIN + deltaY);
        ctx2.stroke();
    }
}
```

9.3 标尺


```
//横向刻度
function drawVerticalAxisTicks() {
    var deltaY;
    for (var i = 1; i < NUM_VERTICAL_TICKS; ++i) {
          if (i % 5 === 0) {
            deltaX = TICK_WIDTH;
            ctx1.moveTo(0, 0 + HORIZONTAL_TICK_SPACING * i);
            ctx1.lineTo(10, 0 + HORIZONTAL_TICK_SPACING * i);
            ctx1.textAlign = 'left';
            ctx1.fillText(i * HORIZONTAL_TICK_SPACING, 10, 0 +
HORIZONTAL_TICK_SPACING * i);
```

```
        }else {
            deltaX = TICK_WIDTH/2;
        }
        ctx1.moveTo(AXIS_ORIGIN.x, i * VERTICAL_TICK_SPACING);
        ctx1.lineTo(AXIS_ORIGIN.x + deltaX, i * VERTICAL_TICK_SPACING);
        ctx1.stroke();
    }
}



//纵向刻度
function drawHorizontalAxisTicks() {
    var deltaY;
    for (var i=1; i < NUM_HORIZONTAL_TICKS; ++i) {
        if (i % 5 === 0) {
            deltaY = TICK_WIDTH;
           ctx1.moveTo(0 + VERTICAL_TICK_SPACING * i, 0);
           ctx1.lineTo(VERTICAL_TICK_SPACING * i, 10);
//          ctx1.textAlign = 'left';
           ctx1.fillText(i * VERTICAL_TICK_SPACING, 0 +
VERTICAL_TICK_SPACING * i, 20);
         }else {
           deltaY = TICK_WIDTH/2;
         }
        ctx1.moveTo(AXIS_ORIGIN.x + i * HORIZONTAL_TICK_SPACING,
AXIS_MARGIN);
        ctx1.lineTo(AXIS_ORIGIN.x + i * HORIZONTAL_TICK_SPACING,
AXIS_MARGIN + deltaY);
        ctx1.stroke();
    }
}
```

10 图元复制、剪切、粘贴

10.1 复制

图元复制首先需要获得需要复制的图元，通过图元选择功能获取待复制的图元，进而 js 中全局变量作为剪切板，用于保存复制的图元。对于单个图元或多个图元均使用图元对象数组进行保存，在复制的时候加以判断，将图元数据保存在剪切板。

实现代码:

```
/**
 * 拷贝图元,单个图元/多个图元
```

```
 */
var selected_item = null;

function copy_select() {
    // console.log("zxs111");
    selected_item = option.objs;
}
```

结果截图：

10.2 剪切

图元剪切首先需要获得需要复制的图元，通过图元选择功能获取待剪切的图元，进而 js 中全局变量作为剪切板，用于保存剪切的图元。剪切的图元从图元对象数组中移除，在剪切的时候加，将待剪切的图元加入到对象数组中，展现到画板上。

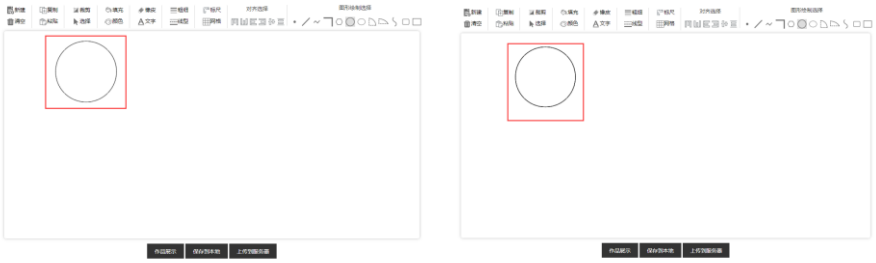
实现代码:

```
/**
 * 固定重复功能
 * 重复最后一个图元
 * 向右下方固定移动 10 哥
 */



function shear() {
    //获取对象数组中的最后一个对象
    var objnew = objs[objs.length - 1];
    //对象向右下方移动 10 个像素
    //获取数组中最后一个元素的类型
    var name = objnew.__proto__.constructor.name;
    switch(name){
        //矩形
        case "Rectangle":
        //圆角矩形
```

```
        case "Roundedrectangle":
        //直角矩形
        case "Rightangle":
        //直线
        case "Line":
            objnew.spoint.px = objnew.spoint.px + 10;
            objnew.spoint.py = objnew.spoint.py + 10;
            objnew.epoint.px = objnew.epoint.px + 10;
            objnew.epoint.py = objnew.epoint.py + 10;
            break;
        //虚线
        case "Brokenline":

            break;
        //多边形
        case "Polygon":
            for (var j = 0; j < objnew.vertices.length; j += 2) {
                objnew.vertices[j] = objnew.vertices[j] + 10;
                objnew.vertices[j+1] = objnew.vertices[j+1] + 10;
            }
            break;
        //曲线
        case "Curve":

            break;
        //椭圆
        case "Ellipse":
        //椭圆弧
        case "Elliparc":

            objnew.center.px = objnew.center.px + 10;
            objnew.center.py = objnew.center.py + 10;
            objnew.epoint.px = objnew.epoint.px + 10;
            objnew.epoint.py = objnew.epoint.py + 10;
            break;
        //圆
        case "Circle":
        //正圆弧
        case "Positivearc":
            objnew.center.px = objnew.center.px + 10;
            objnew.center.py = objnew.center.py + 10;
            break;
    }
```

```
    //在对象数组最后添加一个对象
    objs.push(objnew);
    rePaint();//画布重绘
}
```

结果截图：



10.3 粘贴

首先通过全局变量获得剪切板的图元对象数组，并依次复制图元对象数组中每个图元对象，通过粘贴，指定位置实现图元的复制。

实现代码:

```
/**
 * 黏贴功能
 */
//是否剪贴标识
var paste = false;
var paste_position_x = 0;
var paste_position_y = 0;

//点击黏贴按键，此时 mousedown 监听点击获取点击坐标黏贴功能
function paste_function() {
    paste = !paste;
}

/**
 *
 * @param paste_sx  黏贴左上角的 x 坐标
 * @param paste_sy  黏贴左上角的 x 坐标
 */
function paste_items( paste_sx , paste_sy){
    //判断是单个图元选中或多个图元选中
    //如果是多个图元选中

    if (selected_item instanceof Array){
        for ( var i = 0; i <selected_item.length; i++){
            //取得当前数组中的对象
            var tempitem = selected_item[i];
```

```
                item_repeat_by_position(tempitem , paste_sx , paste_sy);
            }
        }else {
            //选中单个图元
            item_repeat_by_position(selected_item , paste_sx , paste_sy);
        }

        // if(!obj_copy.empty()) {
        //         for (var i = 0; i < objs.length; i++) {
        //             obj_copy = objs[i];
        //             repeat(obj_copy, p);
        //         }
        // }
        // if(!objs_cut.empty()){
        //         for (var i = 0; i < objs.length; i++) {
        //             obj_copy = objs[i];
        //             repeat(obj_copy, p);
        //         }
        // }
        //
        // obj_copy.length=0;    //选择复制的对象数组清空
        // objs_cut.length=0;    //剪切板清空

        console.log("paste");
}

/**
 * 根据指定位置复制单个图元
 */
function item_repeat_by_position(item , position_x , position_y) {
        var objnew = null;
        var name = item.__proto__.constructor.name;
        var distans_x = 0;
        var distans_y = 0;

        switch(name) {
            //矩形
            case "Rectangle":
                objnew = new Rectangle();
                objnew = new Line();
                distans_x = position_x - item.spoint.px;
                distans_y = position_y - item.spoint.py;

                objnew.spoint.px = position_x;
```

```
            objnew.spoint.py = position_y;
            objnew.epoint.px = item.epoint.px + distans_x;
            objnew.epoint.py = item.epoint.py + distans_y;
            break;
        //圆角矩形
        case "Roundedrectangle":
            objnew = new Roundedrectangle();
            objnew = new Line();
            distans_x = position_x - item.spoint.px;
            distans_y = position_y - item.spoint.py;

            objnew.spoint.px = position_x;
            objnew.spoint.py = position_y;
            objnew.epoint.px = item.epoint.px + distans_x;
            objnew.epoint.py = item.epoint.py + distans_y;
            break;
        //直角矩形
        case "Rightangle":
            objnew = new Rightangle();
            objnew = new Line();
            distans_x = position_x - item.spoint.px;
            distans_y = position_y - item.spoint.py;

            objnew.spoint.px = position_x;
            objnew.spoint.py = position_y;
            objnew.epoint.px = item.epoint.px + distans_x;
            objnew.epoint.py = item.epoint.py + distans_y;
            break;
        //直线
        case "DianHeng":
            objnew = new DianHeng();
            objnew = new Line();
            distans_x = position_x - item.spoint.px;
            distans_y = position_y - item.spoint.py;

            objnew.spoint.px = position_x;
            objnew.spoint.py = position_y;
            objnew.epoint.px = item.epoint.px + distans_x;
            objnew.epoint.py = item.epoint.py + distans_y;
            break;
        case "Doublepoint":
            objnew = new Doublepoint();
            objnew = new Line();
            distans_x = position_x - item.spoint.px;
```

```
        distans_y = position_y - item.spoint.py;

        objnew.spoint.px = position_x;
        objnew.spoint.py = position_y;
        objnew.epoint.px = item.epoint.px + distans_x;
        objnew.epoint.py = item.epoint.py + distans_y;
        break;
    case "LongDotted":
        objnew = new LongDotted();
        objnew = new Line();
        distans_x = position_x - item.spoint.px;
        distans_y = position_y - item.spoint.py;

        objnew.spoint.px = position_x;
        objnew.spoint.py = position_y;
        objnew.epoint.px = item.epoint.px + distans_x;
        objnew.epoint.py = item.epoint.py + distans_y;
        break;
    case "Dotted":
        objnew = new Dotted();
        objnew = new Line();
        distans_x = position_x - item.spoint.px;
        distans_y = position_y - item.spoint.py;

        objnew.spoint.px = position_x;
        objnew.spoint.py = position_y;
        objnew.epoint.px = item.epoint.px + distans_x;
        objnew.epoint.py = item.epoint.py + distans_y;
        break;
    case "Pointline":
        objnew = new Pointline();
        objnew = new Line();
        distans_x = position_x - item.spoint.px;
        distans_y = position_y - item.spoint.py;

        objnew.spoint.px = position_x;
        objnew.spoint.py = position_y;
        objnew.epoint.px = item.epoint.px + distans_x;
        objnew.epoint.py = item.epoint.py + distans_y;
        break;
    case "Line":
        objnew = new Line();
        distans_x = position_x - item.spoint.px;
        distans_y = position_y - item.spoint.py;
```

```
        objnew.spoint.px = position_x;
        objnew.spoint.py = position_y;
        objnew.epoint.px = item.epoint.px + distans_x;
        objnew.epoint.py = item.epoint.py + distans_y;
        break;
    //虚线
    case "Brokenline":
        objnew = new Brokenline();
        break;
    //多边形
    case "Polygon":
        objnew = new Polygon();
        distans_x = position_x - item.vertices[0];
        distans_y = position_y - item.vertices[1];
        for (var j = 0; j < item.vertices.length; j += 2) {
            objnew.vertices[j] = item.vertices[j] + distans_x;
            objnew.vertices[j + 1] = item.vertices[j + 1] + distans_y;
        }
        break;
    //曲线
    case "Curve":
        objnew = new Curve();
        distans_x = position_x - item.center.px;
        distans_y = position_y - item.center.py;

        objnew.center.px = item.center.px + distans_x;
        objnew.center.py = item.center.py + distans_y;
        objnew.epoint.px = item.epoint.px + distans_x;
        objnew.epoint.py = item.epoint.py + distans_y;
        break;
        break;
    //椭圆
    case "Ellipse":
        objnew = new Ellipse();
        distans_x = position_x - item.center.px;
        distans_y = position_y - item.center.py;

        objnew.center.px = item.center.px + distans_x;
        objnew.center.py = item.center.py + distans_y;
        objnew.epoint.px = item.epoint.px + distans_x;
        objnew.epoint.py = item.epoint.py + distans_y;
        break;
    //椭圆弧
```

```
            case "Elliparc":
                objnew = new Elliparc();
                distans_x = position_x - item.center.px;
                distans_y = position_y - item.center.py;

                objnew.center.px = item.center.px + distans_x;
                objnew.center.py = item.center.py + distans_y;
                objnew.epoint.px = item.epoint.px + distans_x;
                objnew.epoint.py = item.epoint.py + distans_y;
                break;
            //圆
            case "Circle":
                objnew = new Circle();
                distans_x = position_x - item.center.px;
                distans_y = position_y - item.center.py;
                objnew.center.px = item.center.px + distans_x;
                objnew.center.py = item.center.py + distans_y;
                objnew.radius = item.radius;
                break;
            //正圆弧
            case "Positivearc":
                objnew = new Positivearc();
                distans_x = position_x - item.center.px;
                distans_y = position_y - item.center.py;
                objnew.center.px = item.center.px + distans_x;
                objnew.center.py = item.center.py + distans_y;
                objnew.radius = item.radius;
                break;
        }
        console.log("zxs111111111111111");
        //在对象数组最后添加一个对象
        objs.push(objnew);
        rePaint();//画布重绘
}
```

结果截图：


新建
清空
复制
粘贴
裁剪
选择
填充
颜色
橡皮
文字
粗细
线型
标尺
网格
对齐选择
图形绘制选择
作品展示
保存到本地
上传到服务器