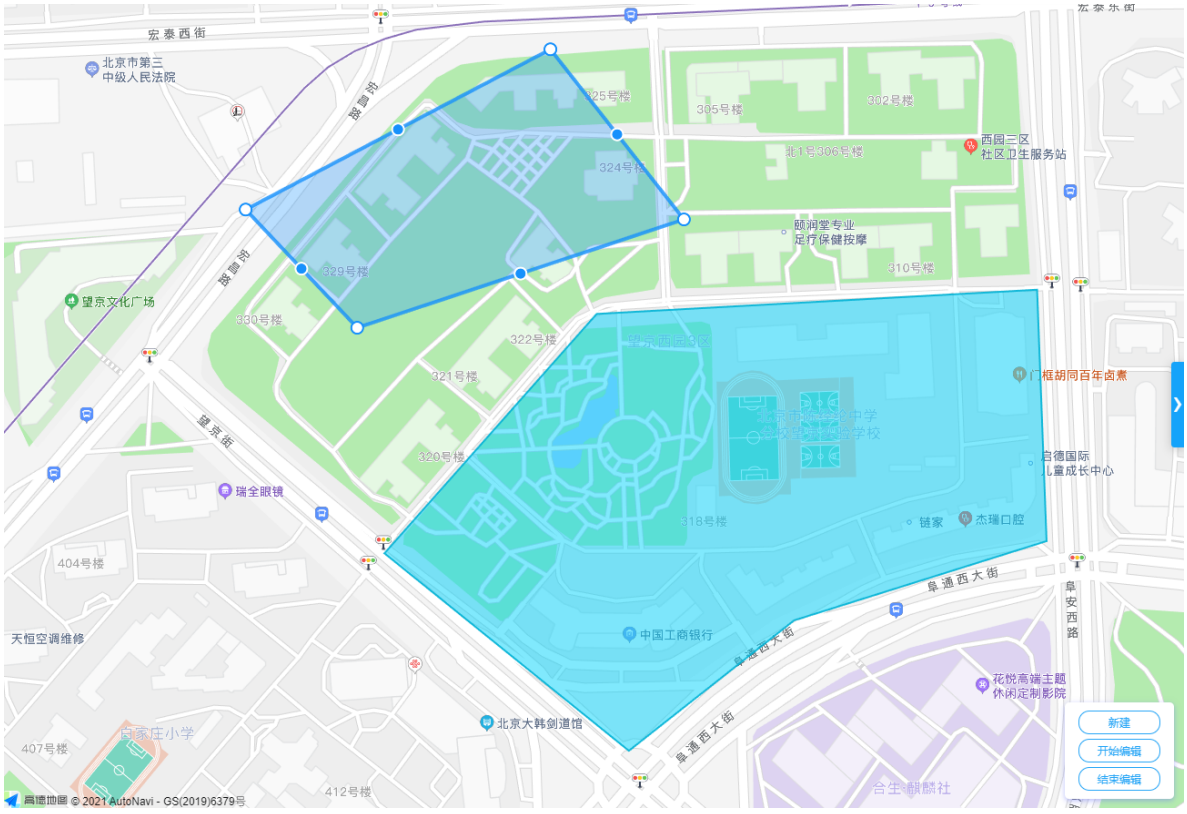
简介

多边形编辑器少数见于一些图片标注需求，常见于地图应用，用来绘制区域，比如高德地图：



示例地址：https://lbs.amap.com/api/jsapi-v2/example/overlay-editor/polygon-editor-avoidpolygon。请先试用一下，接下来实现它的所有功能。

基本准备

准备一个`canvas`元素，设置一下画布宽高，获取一下绘图上下文：

```
<div class="container" ref="container">
    <canvas ref="canvas"></canvas>
</div>
```

```
init () {
    let { width, height } = this.$refs.container.getBoundingClientRect()
    this.width = width
    this.height = height
    let canvas = this.$refs.canvas
    canvas.width = width
    canvas.height = height
    this.ctx = canvas.getContext('2d')
}
```

添加顶点

创建一个多边形的基本操作是鼠标点击并添加顶点，所以需要监听点击事件，然后用线把点击的点都连接起来，鼠标点击事件对象的`clientX`好`clientY`是相对于浏览器窗口的，所以需要减去画布和浏览器窗口的偏移量来得到相对于画布的坐标：

```
toCanvasPos (e) {
    let { left, top } = this.$refs.canvas.getBoundingClientRect()
    return {
        x: e.clientX - left,
        y: e.clientY - top
    }
}
```

接下来绑定单击事件：

```
<canvas ref="canvas" @click="onClick"></canvas>
```

然后使用一个数组来保存我们每次单击新增的顶点：

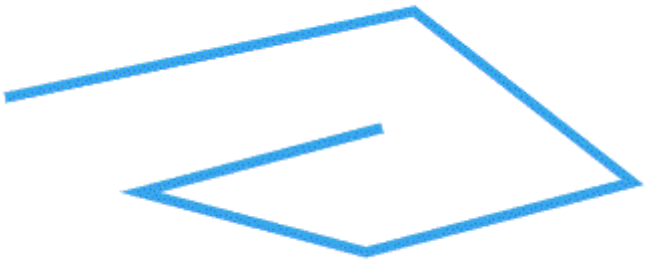
```
export default {
    data() {
        pointsArr: []
    },
    methods: {
        onClick (e) {
            let { x, y } = this.toCanvasPos(e)
            this.pointsArr.push({
                x,
                y
            })
        }
    }
}
```

顶点有了，我们遍历它连线画出来就行了：

```
render () {
    // 先清除画布
    this.ctx.clearRect(0, 0, this.width, this.height)
    // 顶点连线
```

```
    this.ctx.beginPath()
    this.pointsArr.forEach((item, index) => {
        if (index === 0) {
            this.ctx.moveTo(item.x, item.y)
        } else {
            this.ctx.lineTo(item.x, item.y)
        }
    })
    this.ctx.lineWidth = 5
    this.ctx.strokeStyle = '#38a4ec'
    this.ctx.lineJoin = 'round'// 线段连接处圆滑一点更好看
    this.ctx.stroke()
}
```

每次点击都需要调用这个方法重新绘制，效果如下：



但是这样还不是我们要的，我们想要一个从始至终都是闭合的区域，这很简单，把首尾两个点连起来就好了，但是这样不会跟着鼠标当前的位置变化，所以需要把鼠标当前的位置也作为一个顶点追加进去，不过在没有点击前它都只是一个临时的点，把它放进`pointsArr`不合适，我们用一个新变量来存储它。

监听鼠标移动事件来存储当前位置：

```
<canvas ref="canvas" @click="onClick" @mousemove="onMousemove"></canvas>
```

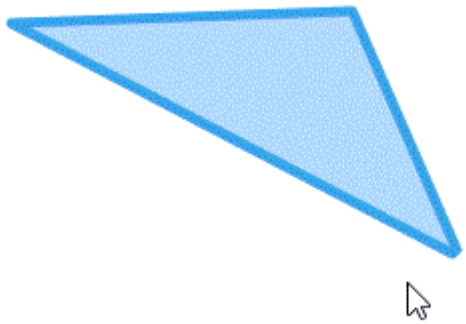
```
export default {
    data () {
        return {
            // ...
            tmpPoint: null
        }
    },
    methods: {
        onMousemove (e) {
            let { x, y } = this.toCanvasPos(e)
            if (this.tmpPoint) {
                this.tmpPoint.x = x
                this.tmpPoint.y = y
            } else {
                this.tmpPoint = {
                    x,
                    y
                }
            }
            this.render()// 鼠标移动时不断刷新重绘
        }
```

```
    }
}
```

接下来连线的时候加上这个点，另外也设置一下填充样式：

```
render () {
    this.ctx.clearRect(0, 0, this.width, this.height)
    this.ctx.beginPath()
    let pointsArr = this.pointsArr.concat(this.tmpPoint ? [this.tmpPoint] :
[])// ++ 把鼠标当前位置追加到最后
    pointsArr.forEach((item, index) => {
        if (index === 0) {
            this.ctx.moveTo(item.x, item.y)
        } else {
            this.ctx.lineTo(item.x, item.y)
        }
    })
    this.ctx.closePath()// ++ 闭合路径
    this.ctx.lineWidth = 5
    this.ctx.strokeStyle = '#38a4ec'
    this.ctx.lineJoin = 'round'
    this.ctx.fillStyle = 'rgba(0, 136, 255, 0.3)'// ++
    this.ctx.fill()// ++
    this.ctx.stroke()
}
```

效果如下：



最后添加一下双击事件来完成顶点的添加：

```
<canvas ref="canvas" @click="onClick" @mousemove="onMousemove"
@dblclick="onDbClick"></canvas>
```

```
{
    onDbClick () {
        this.isClosePath = true// 添加一个变量来标志是否闭合形状
        this.tmpPoint = null// 清空临时点
        this.render()
    },
    onClick (e) {
        if (this.isClosePath) {
            return
        }
```

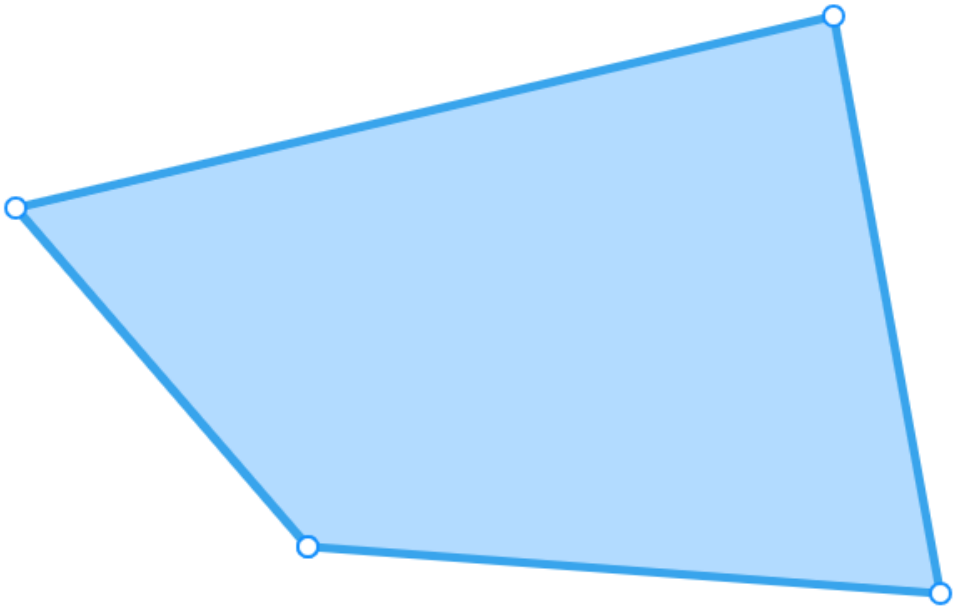
```
        // ...
    },
    onMousemove (e) {
        if (this.isClosePath) {
            return
        }
        // ...
    }
}
```

需要注意的是`dbClick`事件触发的时候也同时会触发两次`click`事件，这样就导致最后双击的位置也被添加进去了，而且添加了两次，可以手动把最后两个点去掉或者自己使用`click`事件来模拟双击事件，本文方便起见就不处理了。

拖动顶点

多边形闭合后，允许拖动各个顶点来修改位置，为了直观，像高德的示例一样给每个顶点都绘制一个圆形：

```
render() {
    // ...
    // 绘制顶点的圆形
    if (this.isClosePath) {
        this.ctx.save()// 因为要重新设置绘图样式，为了不影响线段，所以需要保存一下绘图状态
        this.ctx.lineWidth = 2
        this.ctx.strokeStyle = '#1791fc'
        this.ctx.fillStyle = '#fff'
        this.pointsArr.forEach((item, index) => {
            this.ctx.beginPath()
            this.ctx.arc(item.x, item.y, 6, 0, 2 * Math.PI)
            this.ctx.fill()
            this.ctx.stroke()
        })
        this.ctx.restore()// 恢复绘图状态
    }
}
```



要拖动首先要知道当前鼠标在哪个顶点内，可以在`mousedown`事件里使用`isPointPath`方法来检测：

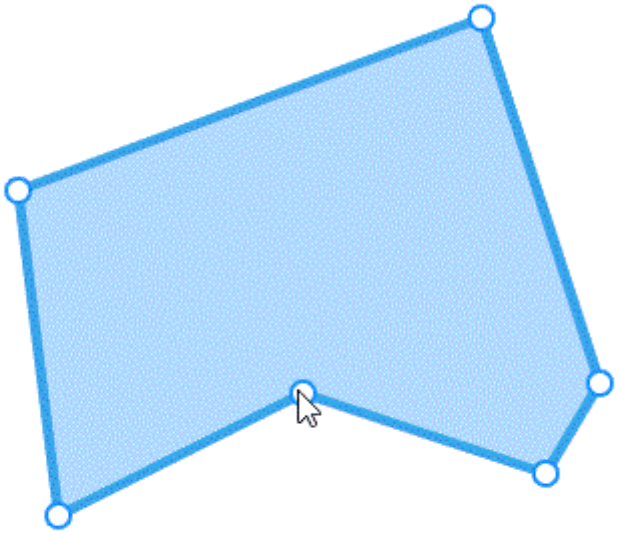
```
<canvas
    ref="canvas"
    @click="onClick"
    @mousemove="onMousemove"
    @dblclick="onDbClick"
    @mousedown="onMousedown"
></canvas>
```

```
export default {
    onMousedown (e) {
        if (!this.isClosePath) {
            return
        }
        this.isMousedown = true
        let { x, y } = this.toCanvasPos(e)
        this.dragPointIndex = this.checkPointIndex(x, y)
    },
    // 检测是否在某个顶点内
    checkPointIndex (x, y) {
        let result = -1
        // 遍历顶点绘制圆形路径，和上面的绘制顶点圆形的区别是这里不需要实际描边和填充，只需要
路径
        this.pointsArr.forEach((item, index) => {
            this.ctx.beginPath()
            this.ctx.arc(item.x, item.y, 6, 0, 2 * Math.PI)
            // 检测是否在当前路径内
            if (this.ctx.isPointInPath(x, y)) {
                result = index
            }
        })
        return result
    }
}
```

知道当前拖动的是哪个顶点后就可以在`mousemove`事件里面实时更新该顶点的位置了：

```
onMousemove (e) {
    // 实时更新当前拖动的顶点位置
    if (this.isClosePath && this.isMousedown && this.dragPointIndex !== -1) {
        let { x, y } = this.toCanvasPos(e)
        // 删除原来的点插入新的点
        this.pointsArr.splice(this.dragPointIndex, 1, {
            x,
            y
        })
        this.render()
    }
    // ...
}
```

效果如下：



拖动整体

高德的示例并没有拖动整体的功能，但是不影响我们支持，整体拖动的逻辑和拖动单个顶点差不多，先判断鼠标按下时是否在多边形内，然后在移动过程中更新所有顶点的位置，和拖动单个的区别是记录和应用的是移动的偏移量，这就需要先缓存一下鼠标按下的位置和此刻的顶点数据。

检测是否在多边形内：

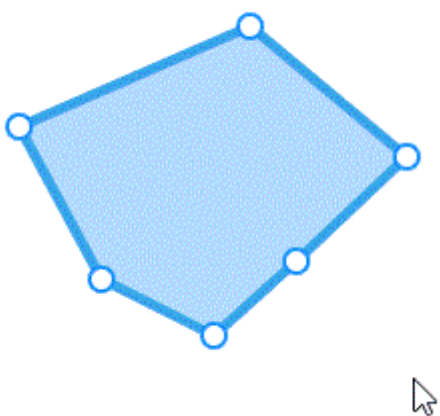
```
export default{
    onMousedown (e) {
        // ...
        // 记录按下的起始位置
        this.startPos.x = x
        this.startPos.y = y
        // 记录当前顶点数据
        this.cachePointsArr = this.pointsArr.map((item) => {
```

```
            return {
                ...item
            }
        })
        this.isInPolygon = this.checkInPolygon(x, y)
    },
    // 检查是否在多边形内
    checkInPolygon (x, y) {
        // 绘制并闭合路径，不实际描边
        this.ctx.beginPath()
        this.pointsArr.forEach((item, index) => {
            if (index === 0) {
                this.ctx.moveTo(item.x, item.y)
            } else {
                this.ctx.lineTo(item.x, item.y)
            }
        })
        this.ctx.closePath()
        return this.ctx.isPointInPath(x, y)
    }
}
```

更新所有顶点位置：

```
onMousemove (e) {
    // 更新所有顶点位置
    if (this.isClosePath && this.isMousedown && this.dragPointIndex === -1 &&
this.isInPolygon) {
        let diffX = x - this.startPos.x
        let diffY = y - this.startPos.y
        this.pointsArr = this.cachePointsArr.map((item) => {
            return {
                x: item.x + diffX,
                y: item.y + diffY
            }
        })
        this.render()
    }
    // ...
}
```

效果如下：



吸附功能

吸附功能能提升使用体验，首先吸附到顶点是比较简单的，遍历一下所有顶点，计算与当前顶点的距离，小于某个值就把当前顶点的位置突变过去就可以了。

以拖动更新单个顶点的位置时添加一下吸附判断：

```
onMousemove (e) {
    if (this.isClosePath && this.isMousedown && this.dragPointIndex !== -1) {
        let { x, y } = this.toCanvasPos(e)
        let adsorbentPos = this.checkAdsorbent(x, y)// ++ 判断是否需要进行吸附
        this.pointsArr.splice(this.dragPointIndex, 1, {
            x: adsorbentPos[0],// ++ 修改为吸附的值
            y: adsorbentPos[1]// ++ 修改为吸附的值
        })
        this.render()
    }
    // ...
}
```

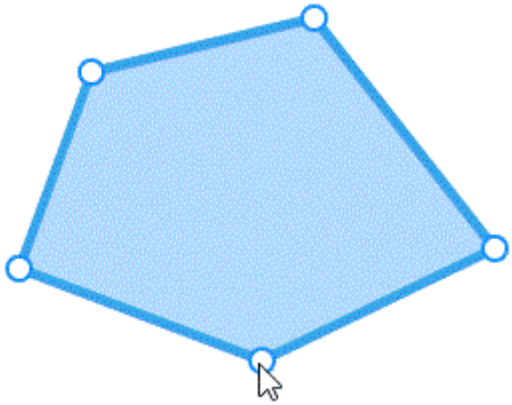
判断吸附的方法：

```
checkAdsorbent (x, y) {
    let result = [x, y]
    // 吸附到顶点
    let minDistance = Infinity
    this.pointsArr.forEach((item, index) => {
        // 跳过和自身的比较
        if (this.dragPointIndex === index) {
            return
        }
        // 获取两点距离
        let distance = this.getTwoPointDistance(item.x, item.y, x, y)
        // 如果小于10的话则使用该顶点的位置来替换当前鼠标的位置
        if (distance <= 10 && distance < minDistance) {
            minDistance = distance
            result = [item.x, item.y]
        }
    })
    return result
}
```

`getTwoPointDistance`方法用来计算两个点的距离：

```
getTwoPointDistance (x1, y1, x2, y2) {
    return Math.sqrt(Math.pow(x1 - x2, 2) + Math.pow(y1 - y2, 2))
}
```

效果如下：



除了在拖动的时候吸附，添加顶点的时候也可以添加吸附功能，这里就不做了。另外除了吸附到顶点，还需要吸附到线段，也就是线段上离当前点最近的一个点上，也以拖动单个顶点为例来看一下。

首先需要根据顶点创建一下线段：

```
createLineSegment () {
    let result = []
    // 创建线段
    let arr = this.pointsArr
    let len = arr.length
    for (let i = 0; i < len - 1; i++) {
        result.push([
            arr[i],
            arr[i + 1]
        ])
    }
    // 加上起点和终点组成的线段
    result.push([
        arr[len - 1],
        arr[0]
    ])
    // 去掉包含当前拖动点的线段
    if (this.dragPointIndex !== -1) {
        // 如果拖动的是起点，那么去掉第一条和最后一条线段
        if (this.dragPointIndex === 0) {
            result.splice(0, 1)
            result.splice(-1, 1)
        } else { // 其余中间的点则去掉前一根和后一根
            result.splice(this.dragPointIndex - 1, 2)
        }
    }
    return result
}
```

创建线段最好把两个端点相同的线段过滤掉，然后在`checkAdsorbent`方法添加吸附线段的逻辑，注意要添加到吸附到顶点的代码之前，这样会优先吸附到顶点。

```
checkAdsorbent (x, y) {
    let result = [x, y]
    // 吸附到线段
    let segments = this.createLineSegment()// ++
    // 吸附到顶点
    // ...
}
```

有了线段就可以遍历线段计算和当前点距离最近的线段，使用点到直线的距离公式：

$$d = \left| \frac{Ax_0 + By_0 + C}{\sqrt{A^2 + B^2}} \right|$$

标准的直线方程为：`Ax+By+C=0`，有三个未知变量，我们只有两个点，显然计算不出三个变量，所以我们使用斜截式：`y=kx+b`，即不垂直于x轴的直线，计算出`k`和`b`，这样：`Ax+By+C = kx-y+b = 0`，得出`A = k`，`B = -1`，`C = b`，这样只要计算出`A`和`C`即可：

```
getLinePointDistance (x1, y1, x2, y2, x, y) {
    // 垂直于x轴的直线特殊处理，横坐标相减就是距离
    if (x1 === x2) {
        return Math.abs(x - x1)
    } else {
        let B = -1
        let A, C
        A = (y2 - y1) / (x2 - x1)
        C = 0 - B * y1 - A * x1
        return Math.abs((A * x + B * y + C) / Math.sqrt(A * A + B * B))
    }
}
```

知道最近的线段之后问题又来了，得知道线段上离该点最近的一个点，假设线段`s`的两个端点为：`(x1,y1)`、`(x2,y2)`，点`p`为：`(x0,y0)`，那么有如下推导：

```
// 线段s的斜率
let k = (y2 - y1) / (x2 - x1)
// 端点1代入斜截式公式y=kx+b
let y1 = k * x1 + b
// 得出b
let b = y1 - k * x1
// k和b都知道了，直线公式也就知道了
let y = k * x + b = k * x + y1 - k * x1 = k * (x - x1) + y1
// 线段上离点p最近的点和p组成的直线一定是垂直于线段s的，即垂线，垂线的斜率k1和线段的斜率k乘积为-1，那么
let k1 = -1 / k
// 点p代入斜截式公式y=kx+b，求出垂线的直线方程
let y0 = k1 * x0 + b
let b = y0 - k1 * x0
let y = k1 * x + y0 - k1 * x0 = k1 * (x - x0) + y0 = (-1 / k) * (x - x0) + y0
// 最后这两条线相交的点即为距离最近的点，也就是联立这两个直线方程，求出x和y
let y = k * (x - x1) + y1
let x = (k * k * x1 + k * (y0 - y1) + x0) / (k * k + 1)
```

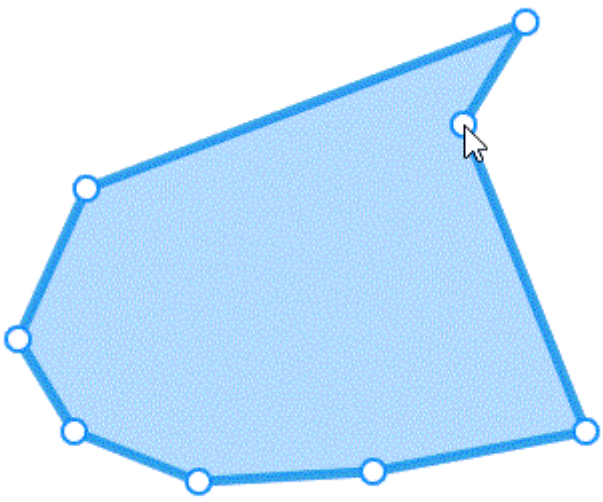
根据以上推导，可以计算出最近的点，不过最后还需要判断一下这个点是否在线段上，也许是在直线的其他位置：

```
getNearestPoint (x1, y1, x2, y2, x0, y0) {
    let k = (y2 - y1) / (x2 - x1)
    let x = (k * k * x1 + k * (y0 - y1) + x0) / (k * k + 1)
    let y = k * (x - x1) + y1
    // 判断该点的x坐标是否在线段的两个端点之间
    let min = Math.min(x1, x2)
    let max = Math.max(x1, x2)
    // 如果在线段内就是我们要的点
    if (x >= min && x <= max) {
        return {
            x,
            y
        }
    } else { // 否则返回null
        return null
    }
}
```

接下来跟吸附到顶点一样，突变到这个位置：

```
checkAdsorbent (x, y) {
    let result = [x, y]
    // 吸附到线段
    let segments = this.createLineSegment()// 创建线段
    let nearestLineResult = this.getPintNearestLine(x, y, segments)// 找到最近的一
条线段
    if (nearestLineResult[0] <= 10) {// 距离小于10进行吸附
        let segment = nearestLineResult[1]// 线段的两个端点
        let nearestPoint = this.getNearestPoint(segment[0].x, segment[0].y,
segment[1].x, segment[1].y, x, y)// 找到线段上最近的点
        if (nearestPoint) {
            result = [nearestPoint.x, nearestPoint.y]
        }
    }
    // 吸附到顶点
    // ...
}
```

效果如下：



删除及新增顶点

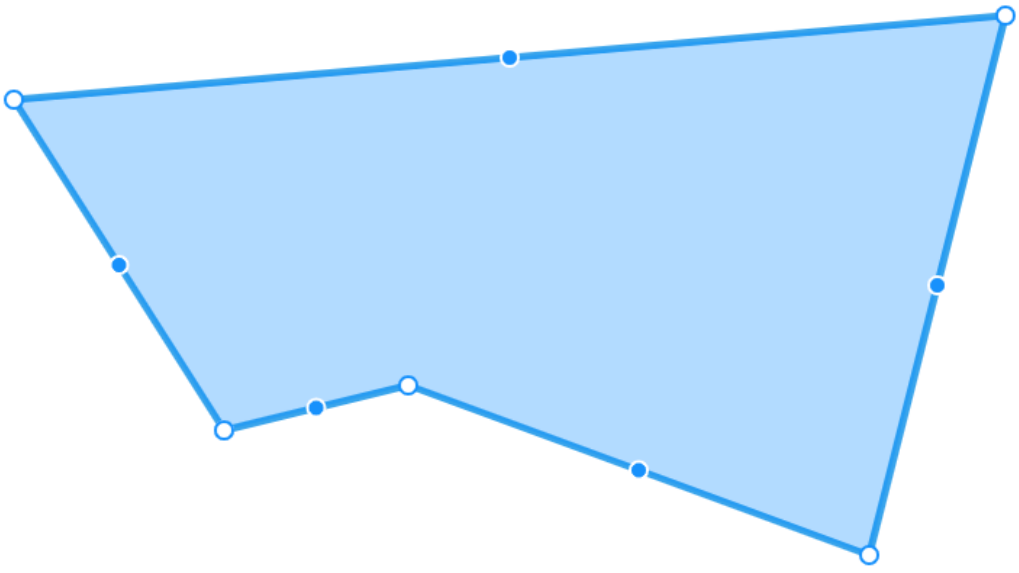
高德的多边形编辑器在没有拖动的时候会在每条线段的中心都显示一个实心的小圆点，你不点它它昙花一现，当你去拖动它时它就会变成真实的顶点，也就完成了顶点的新增。

首先在非拖动的情况下插入虚拟顶点并渲染，然后拖动前再把它去掉，因为加入了虚拟顶点，所以在计算`dragPointIndex`时需要转换成没有虚拟顶点的真实索引，当检测到拖动的是虚拟节点时把它转换成真实顶点就可以了。

先插入虚拟顶点，给顶点增加一个`fictitious`字段来代表是否是虚拟顶点：

```
render () {
    // 先去掉之前插入的虚拟顶点
    this.pointsArr = this.pointsArr.filter((item) => {
        return !item.fictitious
    })
    if (this.isClosePath && !this.isMousedown) {// 插入虚拟顶点
        this.insertFictitiousPoints()
    }
    // ...
    // 先清除画布
}
```

插入虚拟顶点就是在每两个顶点之间插入这两个顶点的中点坐标，这个很简单，就不附代码了，另外，绘制顶点的时候如果是虚拟顶点，那么把描边颜色和填充颜色反一下，用来作区分，效果如下：

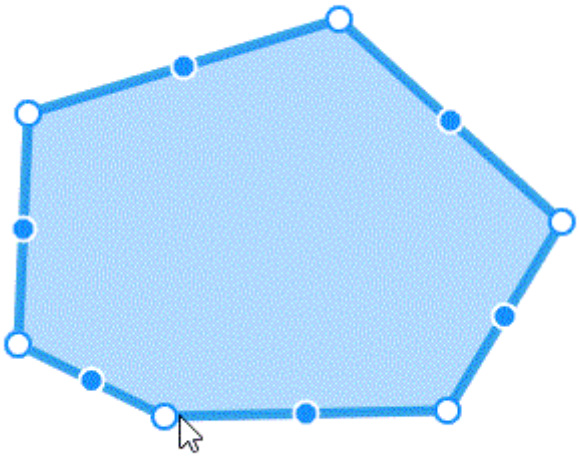


接下来修改一下`mousemove`方法，如果拖动的是虚拟顶点，那就把它转换成真实顶点，也就是把`fictitious`字段给删了：

```
onMousemove (e) {
    if (this.isClosePath && this.isMousedown && this.dragPointIndex !== -1) {
        // 如果是虚拟顶点，转换成真实顶点
        if (this.pointsArr[this.dragPointIndex].fictitious) {
            delete this.pointsArr[this.dragPointIndex].fictitious
```

```
            }
            // 转换成没有虚拟顶点时的真实索引
            let prevFictitiousCount = 0
            for (let i = 0; i < this.dragPointIndex; i++) {
                if (this.pointsArr[i].fictitious) {
                    prevFictitiousCount++
                }
            }
            this.dragPointIndex -= prevFictitiousCount
            // 移除虚拟顶点
            this.pointsArr = this.pointsArr.filter((item) => {
                return !item.fictitious
            })
            // 之前的拖动逻辑...
        }
        // ...
    }
```

效果如下：



最后修复一下整体拖动时的bug：

```
this.pointsArr = this.cachePointsArr.map((item) => {
    return {
        ...item,// ++，不要把fictitious状态给丢了
        x: item.x + diffX,
        y: item.y + diffY
    }
})
```

删除顶点的话很容易，直接从数组里移除即可，详见源码。

支持多个多边形并存

以上只是完成了一个多边形的创建和编辑，如果需要同时存在多个多边形，每个都可以选中进行编辑，那么上面的代码是无法实现的，需要调整代码组织方式，每个多边形都要维护各自的状态，那么可以创建一个多边形的类，把上面的一些状态和方法都移到这个类里，然后选中那个就操作哪个类即可。

总结

本文实现了一个基本的图形标注功能，可根据以上思路实现业务中的相关需求及提炼成一个通用的插件。