

**《计算机图形学》实验报告**

**院 系：** 计算机学院

**任 课 教 师：** 杨晋吉

**学 生 姓 名：** 周炫

**学 号：** 20211131028

**学 院：** 计算机学院

**完 成 时 间：** 2024.12.31

## **一、实验目的**

本实验旨在深入理解计算机图形学的基本概念、原理与方法，强化常用算法设计的应用及分析能力，通过设计与实现一个具有图形生成、裁剪、几何变换、填充及命令操作等功能的交互式图形系统，提升解决实际问题与创新的能力，实现理论与实践的紧密结合。

## **二、实验环境**

实验采用 HTML、CSS 与 JavaScript 技术，在常见的现代浏览器环境下运行，利用浏览器的 Canvas 绘图 API 进行图形绘制与操作，无需额外特殊软件或硬件配置。

## **三、算法设计与实现**

### **（一）图形生成算法**

* ****点绘制：**在 drawPointAlgorithm 函数中，通过获取画布的 ImageData，使用 setPixel 函数在指定坐标设置像素颜色为黑色来绘制点，确保坐标在画布范围内。**
* ****直线绘制：**drawLineAlgorithm 基于 Bresenham 算法，计算直线的斜率与误差项，在循环中通过判断误差决定像素点的绘制位置，最后将处理后的 ImageData 放回画布完成直线绘制。**
* ****圆绘制：**drawCircleAlgorithm 运用中点圆算法，以给定圆心和半径，在循环中根据圆的对称性确定多个点的坐标并绘制，实现圆的绘制。**
* ****曲线绘制：**drawBezierCurveAlgorithm 依据三次贝塞尔曲线公式，在 0 到 1 的参数范围内以 0.001 步长计算曲线上的点坐标，使用 setPixel 绘制并显示曲线。**
* ****文字绘制：**在 drawText 函数中，先获取用户输入的字体大小和颜色，设置 ctx 的字体与填充样式，然后在指定坐标绘制输入的文字，并将文字信息存储到 shapes 数组。**

**具体代码如下：**

// Bresenham 算法绘制直线

let startX, startY;

function drawLineAlgorithm(x0, y0, x1, y1) {

const imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);

let dx = Math.abs(x1 - x0), dy = Math.abs(y1 - y0);

let sx = x0 < x1 ? 1 : -1, sy = y0 < y1 ? 1 : -1;

let err = dx - dy;

while (true) {

setPixel(imageData, x0, y0, [0, 0, 0, 255]); // 黑色像素

if (x0 === x1 && y0 === y1) break;

const e2 = 2 \* err;

if (e2 > -dy) { err -= dy; x0 += sx; }

if (e2 < dx) { err += dx; y0 += sy; }

}

ctx.putImageData(imageData, 0, 0);

}

// 中点圆算法绘制圆

let circleX, circleY;

function drawCircleAlgorithm(cx, cy, radius) {

const imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);

let x = radius, y = 0, err = 0;

while (x >= y) {

setPixel(imageData, cx + x, cy + y, [0, 0, 0, 255]);

setPixel(imageData, cx + y, cy + x, [0, 0, 0, 255]);

setPixel(imageData, cx - y, cy + x, [0, 0, 0, 255]);

setPixel(imageData, cx - x, cy + y, [0, 0, 0, 255]);

setPixel(imageData, cx - x, cy - y, [0, 0, 0, 255]);

setPixel(imageData, cx - y, cy - x, [0, 0, 0, 255]);

setPixel(imageData, cx + y, cy - x, [0, 0, 0, 255]);

setPixel(imageData, cx + x, cy - y, [0, 0, 0, 255]);

if (err <= 0) {

y++;

err += 2 \* y + 1;

}

if (err > 0) {

x--;

err -= 2 \* x + 1;

}

}

ctx.putImageData(imageData, 0, 0);

}

function drawBezierCurveAlgorithm(points) {

const [p0, p1, p2, p3] = points;

const imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);

for (let t = 0; t <= 1; t += 0.001) {

const x = Math.pow(1 - t, 3) \* p0.x +

3 \* Math.pow(1 - t, 2) \* t \* p1.x +

3 \* (1 - t) \* Math.pow(t, 2) \* p2.x +

Math.pow(t, 3) \* p3.x;

const y = Math.pow(1 - t, 3) \* p0.y +

3 \* Math.pow(1 - t, 2) \* t \* p1.y +

3 \* (1 - t) \* Math.pow(t, 2) \* p2.y +

Math.pow(t, 3) \* p3.y;

setPixel(imageData, Math.round(x), Math.round(y), [0, 0, 0, 255]);

}

ctx.putImageData(imageData, 0, 0);

}

### **（二）图形选择算法**

* **这一块的逻辑主要是为了后续对图形进行平移、旋转、缩放等操作进行前置准备。**
* **在 handleMouseDown 函数中，当处于选择模式时，遍历 shapes 数组，针对不同类型图形（点、直线、圆、文字）分别调用相应的判断函数（isNearPoint、isNearLine、isNearCircle、isNearText），根据设定的检测阈值（如 5 像素）确定是否选中图形，若选中则更新其 selected 属性并重绘画布显示选中状态。**

**具体代码如下：**

function isNearPoint(x, y, px, py) {

const threshold = 5; // 检测阈值

return Math.abs(x - px) <= threshold && Math.abs(y - py) <= threshold;

}

// 判断是否靠近直线

function isNearLine(x, y, line) {

const { startX, startY, endX, endY } = line;

const dist = pointToLineDistance(x, y, startX, startY, endX, endY);

return dist <= 5; // 允许一定误差

}

// 计算点到直线的距离

function pointToLineDistance(px, py, x1, y1, x2, y2) {

const A = px - x1, B = py - y1, C = x2 - x1, D = y2 - y1;

const dot = A \* C + B \* D;

const lenSq = C \* C + D \* D;

const param = lenSq !== 0 ? dot / lenSq : -1;

let xx, yy;

if (param < 0) {

xx = x1; yy = y1;

} else if (param > 1) {

xx = x2; yy = y2;

} else {

xx = x1 + param \* C;

yy = y1 + param \* D;

}

const dx = px - xx, dy = py - yy;

return Math.sqrt(dx \* dx + dy \* dy);

}

// 判断是否靠近圆

function isNearCircle(x, y, circle) {

const dx = x - circle.centerX;

const dy = y - circle.centerY;

const dist = Math.sqrt(dx \* dx + dy \* dy);

return Math.abs(dist - circle.radius) <= 5; // 检测是否在圆边附近

}

// 判断是否靠近文字（可根据文字宽高调整）

function isNearText(x, y, text) {

ctx.font = '16px Arial';

const textWidth = ctx.measureText(text.text).width;

const textHeight = 16; // 假设字体大小为 16px

return x >= text.x && x <= text.x + textWidth && y <= text.y && y >= text.y - textHeight;

}

### **（三）几何变换算法**

* ****平移****：applyTransform 函数中处理平移操作时，获取用户输入的平移距离 dx 和 dy，根据选中图形类型（点、直线、圆）分别对其坐标进行相应的加法运算，实现图形平移，并通过 redrawCanvas 函数更新显示。
* ****旋转****：对于旋转操作，获取用户输入角度并转换为弧度制，针对直线图形，先计算中点作为旋转中心，然后使用 rotatePoint 函数旋转起点和终点坐标，最后重绘画布。
* ****缩放****：缩放操作时，获取缩放比例因子，对于直线，以中点为基准缩放起点和终点；对于圆，直接缩放半径；对于多边形（代码中未完整实现但预留结构），计算中心后缩放各顶点坐标，完成后重绘画布。

具体代码如下：

// 几何变换：平移、旋转、缩放

function applyTransform(type) {

const selectedShape = shapes.find(shape => shape.selected);

if (!selectedShape) {

alert('请先选中一个对象！');

return;

}

if (type === 'translate') {

const dx = parseFloat(prompt('输入平移 x 距离:'));

const dy = parseFloat(prompt('输入平移 y 距离:'));

if (selectedShape.type === 'point') {

selectedShape.x += dx;

selectedShape.y += dy;

} else if (selectedShape.type === 'line') {

selectedShape.startX += dx;

selectedShape.startY += dy;

selectedShape.endX += dx;

selectedShape.endY += dy;

} else if (selectedShape.type === 'circle') {

selectedShape.centerX += dx;

selectedShape.centerY += dy;

}

redrawCanvas();

} else if (type === 'rotate') {

const angle = parseFloat(prompt('输入旋转角度（度数）:')) \* (Math.PI / 180);

if (selectedShape.type === 'line') {

const centerX = (selectedShape.startX + selectedShape.endX) / 2;

const centerY = (selectedShape.startY + selectedShape.endY) / 2;

// 旋转起点和终点

[selectedShape.startX, selectedShape.startY] = rotatePoint(

selectedShape.startX, selectedShape.startY, centerX, centerY, angle

);

[selectedShape.endX, selectedShape.endY] = rotatePoint(

selectedShape.endX, selectedShape.endY, centerX, centerY, angle

);

}

redrawCanvas();

} else if (type === 'scale') {

const factor = parseFloat(prompt('输入缩放比例:'));

if (selectedShape.type === 'line') {

const centerX = (selectedShape.startX + selectedShape.endX) / 2;

const centerY = (selectedShape.startY + selectedShape.endY) / 2;

// 缩放起点和终点

selectedShape.startX = centerX + (selectedShape.startX - centerX) \* factor;

selectedShape.startY = centerY + (selectedShape.startY - centerY) \* factor;

selectedShape.endX = centerX + (selectedShape.endX - centerX) \* factor;

selectedShape.endY = centerY + (selectedShape.endY - centerY) \* factor;

} else if (selectedShape.type === 'circle') {

selectedShape.radius \*= factor;

} else if (selectedShape.type === 'polygon') {

const centerX = selectedShape.points.reduce((sum, p) => sum + p.x, 0) / selectedShape.points.length;

const centerY = selectedShape.points.reduce((sum, p) => sum + p.y, 0) / selectedShape.points.length;

selectedShape.points = selectedShape.points.map(p => ({

x: centerX + (p.x - centerX) \* factor,

y: centerY + (p.y - centerY) \* factor,

}));

}

redrawCanvas();

}

}

### **（四）裁剪算法**

* ****直线裁剪（Cohen - Sutherland）****：cohenSutherland 函数实现直线裁剪，首先计算直线端点的裁剪编码，通过循环判断编码情况，依据不同条件进行直线与裁剪窗口边界的求交计算和端点更新，最终确定裁剪后的直线或判定直线完全在窗口外（返回 null）。
* ****圆形裁剪****：clipCircle 函数先检查圆心与裁剪窗口的位置关系，若圆完全在窗口外则返回 null，否则返回原圆信息，表示圆在窗口内无需裁剪。

具体代码如下：

// 裁剪

function clipShape() {

const selectedShape = shapes.find(shape => shape.selected);

if (!selectedShape) {

alert('请先选中一个对象！');

return;

}

const clipXMin = parseFloat(prompt('裁剪窗口 xMin:'));

const clipYMin = parseFloat(prompt('裁剪窗口 yMin:'));

const clipXMax = parseFloat(prompt('裁剪窗口 xMax:'));

const clipYMax = parseFloat(prompt('裁剪窗口 yMax:'));

if (selectedShape.type === 'line') {

return cohenSutherland(selectedShape, clipXMin, clipYMin, clipXMax, clipYMax);

} else if (selectedShape.type === 'circle') {

return clipCircle(selectedShape, clipXMin, clipYMin, clipXMax, clipYMax);

} else if (selectedShape.type === 'polygon') {

return clipPolygon(selectedShape, clipXMin, clipYMin, clipXMax, clipYMax);

}

return selectedShape;

redrawCanvas();

alert("裁剪完成！");

}

// Cohen-Sutherland 直线裁剪

function cohenSutherland(line, xmin, ymin, xmax, ymax) {

const { startX, startY, endX, endY } = line;

let [x0, y0, x1, y1] = [startX, startY, endX, endY];

function computeOutCode(x, y) {

let code = 0;

if (x < xmin) code |= 1;

if (x > xmax) code |= 2;

if (y < ymin) code |= 4;

if (y > ymax) code |= 8;

return code;

}

let outCode0 = computeOutCode(x0, y0);

let outCode1 = computeOutCode(x1, y1);

let accept = false;

while (true) {

if (!(outCode0 | outCode1)) {

accept = true;

break;

} else if (outCode0 & outCode1) {

return null;

} else {

let x, y;

const outCodeOut = outCode0 ? outCode0 : outCode1;

if (outCodeOut & 8) {

x = x0 + (x1 - x0) \* (ymax - y0) / (y1 - y0);

y = ymax;

} else if (outCodeOut & 4) {

x = x0 + (x1 - x0) \* (ymin - y0) / (y1 - y0);

y = ymin;

} else if (outCodeOut & 2) {

y = y0 + (y1 - y0) \* (xmax - x0) / (x1 - x0);

x = xmax;

} else if (outCodeOut & 1) {

y = y0 + (y1 - y0) \* (xmin - x0) / (x1 - x0);

x = xmin;

}

if (outCodeOut === outCode0) {

x0 = x;

y0 = y;

outCode0 = computeOutCode(x0, y0);

} else {

x1 = x;

y1 = y;

outCode1 = computeOutCode(x1, y1);

}

}

}

if (accept) {

return { type: 'line', startX: x0, startY: y0, endX: x1, endY: y1 };

}

return null;

}

// 圆形裁剪

function clipCircle(circle, xmin, ymin, xmax, ymax) {

const { centerX, centerY, radius } = circle;

// 检查圆心是否在裁剪窗口内

if (

centerX + radius < xmin ||

centerX - radius > xmax ||

centerY + radius < ymin ||

centerY - radius > ymax

) {

return null; // 圆完全在裁剪窗口外

}

// 返回裁剪后的圆

return { ...circle };

}

### **（五）填充算法**

* ****种子填充****：fillCircle 函数针对圆形种子填充，以圆心为起始点压入栈，获取圆心颜色作为目标颜色，在循环中弹出栈顶像素，判断其在圆内且颜色匹配时，设置为填充颜色，并将相邻像素压入栈，直至栈空完成填充。

具体代码如下：

// 种子填充【目前只支持圆形】

function fillCircle(circle, fillColor) {

const { centerX, centerY, radius } = circle;

const imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);

const stack = [[centerX, centerY]];

const targetColor = Array.from(ctx.getImageData(centerX, centerY, 1, 1).data);

while (stack.length) {

const [px, py] = stack.pop();

const index = (py \* canvas.width + px) \* 4;

if (isInsideCircle(px, py, centerX, centerY, radius) && matchColor(imageData.data, index, targetColor)) {

setColor(imageData.data, index, fillColor);

stack.push([px + 1, py], [px - 1, py], [px, py + 1], [px, py - 1]);

}

}

ctx.putImageData(imageData, 0, 0);

}

* ****扫描线填充****：fillCircleScan 函数用于圆形扫描线填充，遍历圆的扫描线范围（y 从 cy - r 到 cy + r），计算每条扫描线与圆的交点范围，在该范围内绘制线段实现填充。

具体代码如下：

// 扫描线填充【目前只支持圆形】

function fillCircleScan(circle, fillColor) {

const { centerX: cx, centerY: cy, radius: r } = circle;

ctx.fillStyle = fillColor;

// 遍历每条扫描线（即 y 从 cy-r 到 cy+r）

for (let y = Math.floor(cy - r); y <= Math.ceil(cy + r); y++) {

// 计算扫描线与圆的交点范围

const dy = y - cy;

if (Math.abs(dy) > r) continue; // 超出圆范围，跳过

const dx = Math.sqrt(r \* r - dy \* dy);

const xStart = Math.floor(cx - dx);

const xEnd = Math.ceil(cx + dx);

// 在该扫描线上填充从 xStart 到 xEnd 的像素

ctx.beginPath();

ctx.moveTo(xStart, y);

ctx.lineTo(xEnd, y);

ctx.stroke();

}

}

## **四、实验结果**

在浏览器中打开实验页面，可通过工具栏按钮或命令输入框执行各种操作。成功绘制出高质量的点、直线、圆、曲线和文字图形，选择功能准确识别所选图形，几何变换操作能精确改变图形位置、角度和大小，裁剪功能正确处理图形与裁剪窗口的关系，填充功能在圆形上实现预期的颜色填充效果，整体系统运行稳定，各功能交互流畅，达到了实验预期目标。

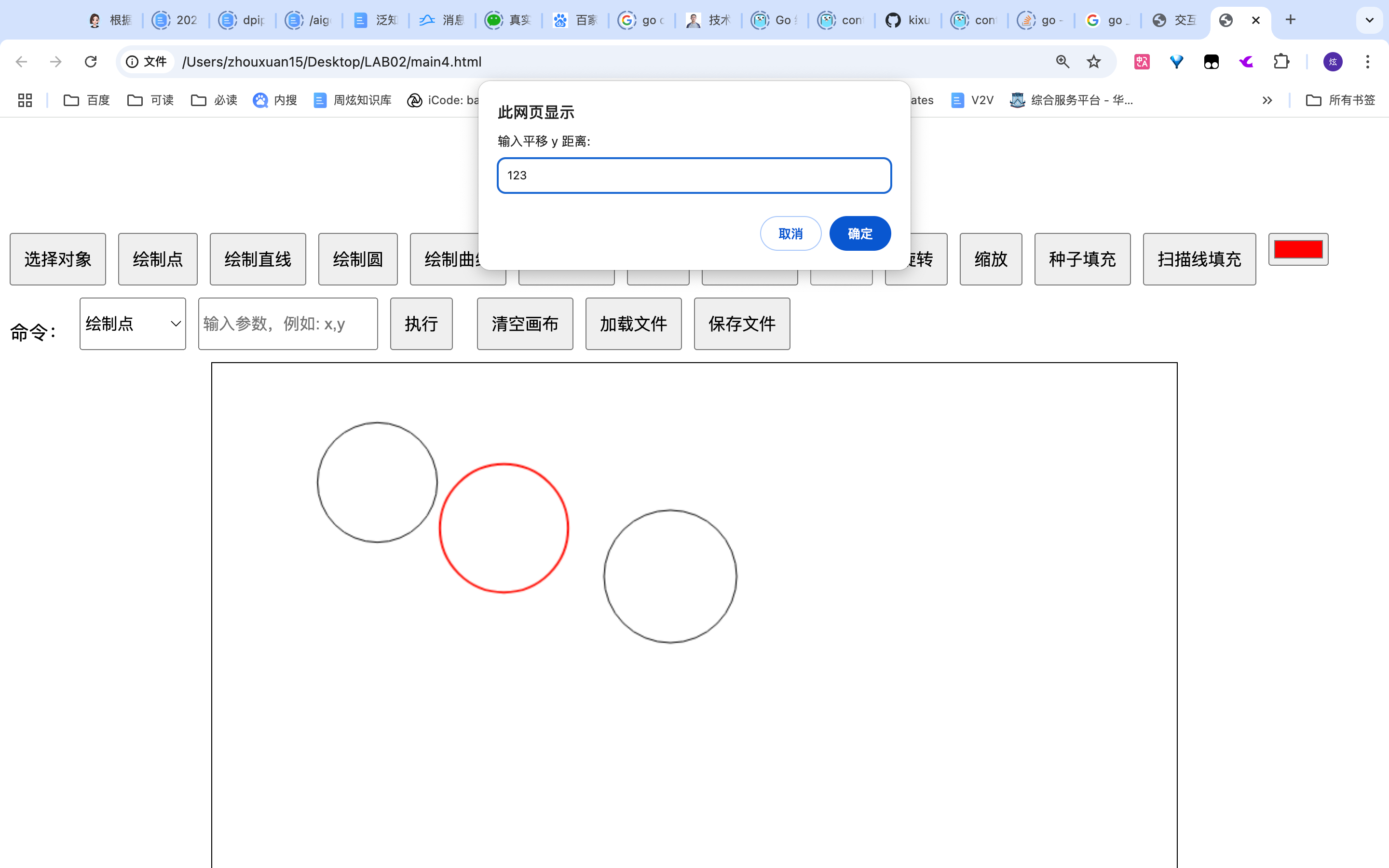
1. 绘制功能：包括点、线、圆、曲线和文字的绘制



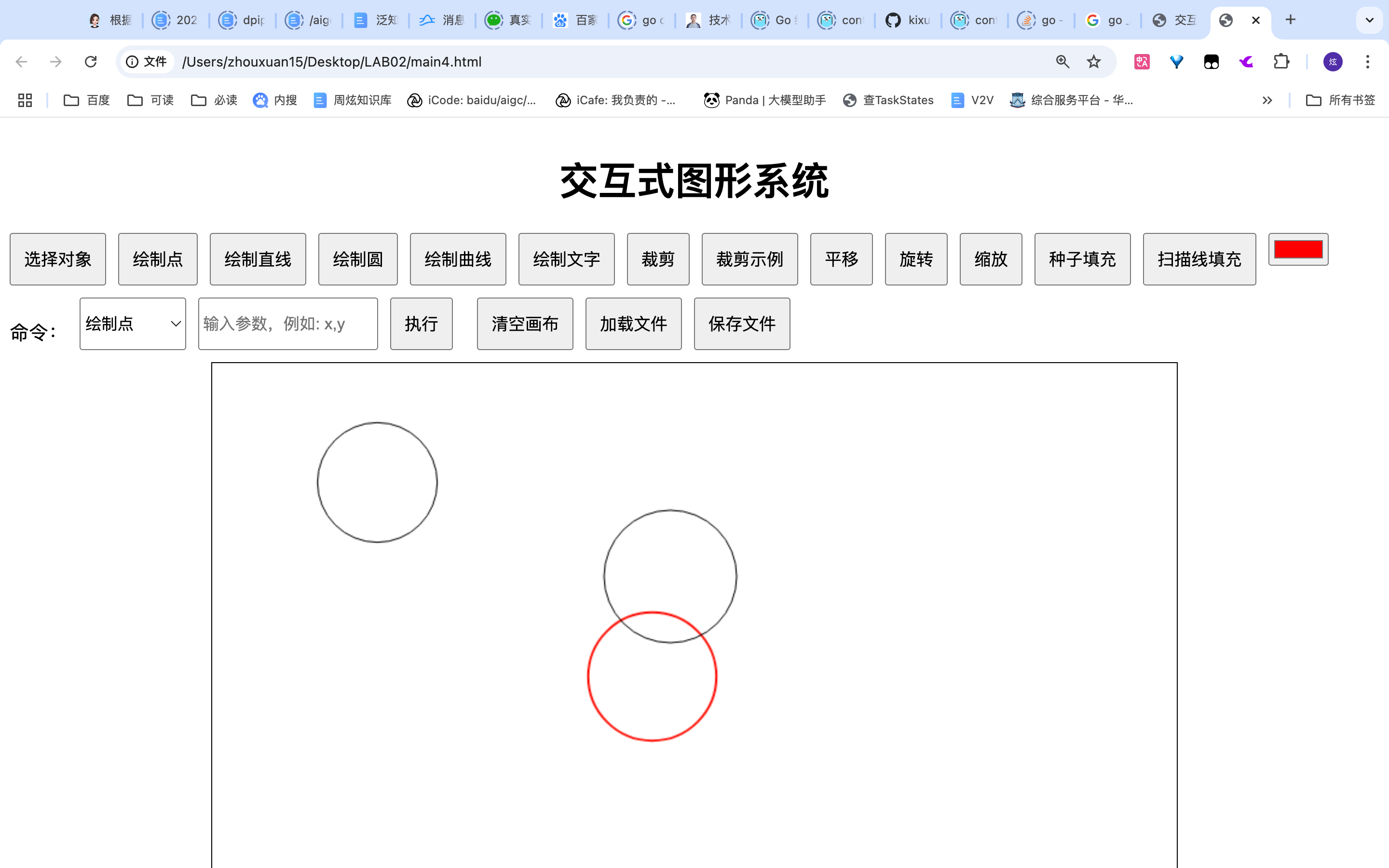
1. 进行操作前必须选中对象，否则会产生提示，选中的对象为红色（字体除外）



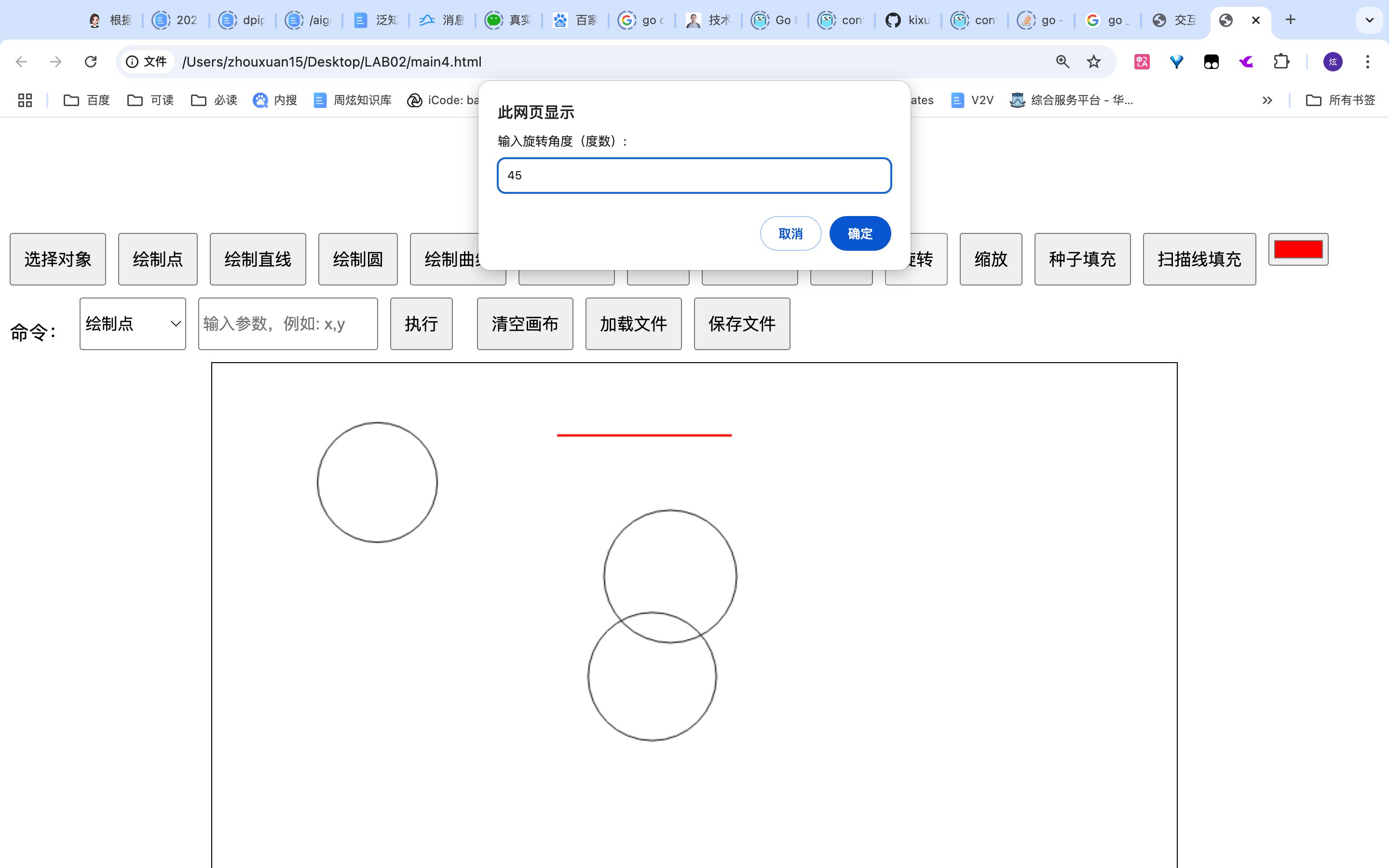
1. 测试平移功能：对圆进行平移



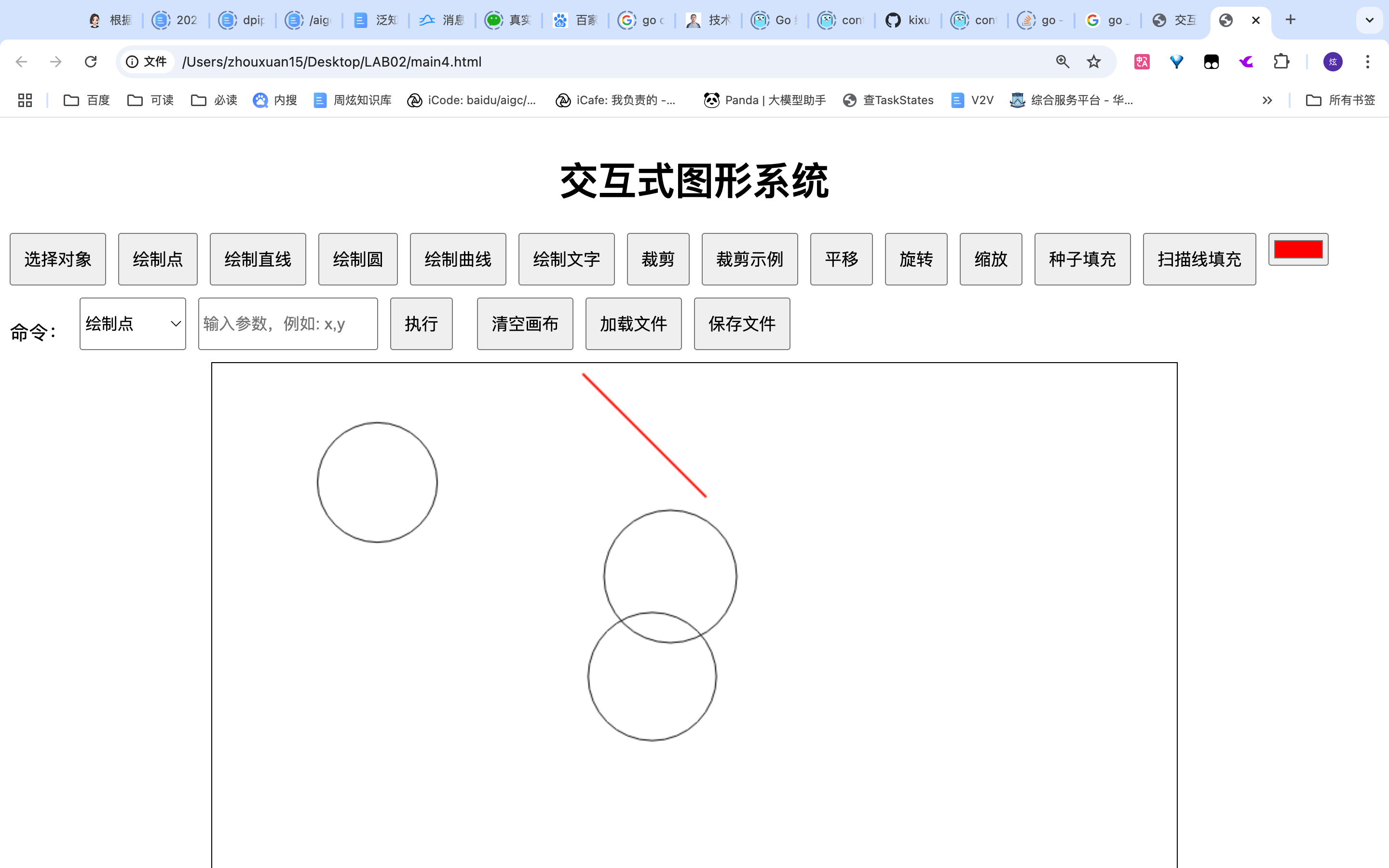
平移后的圆位置：



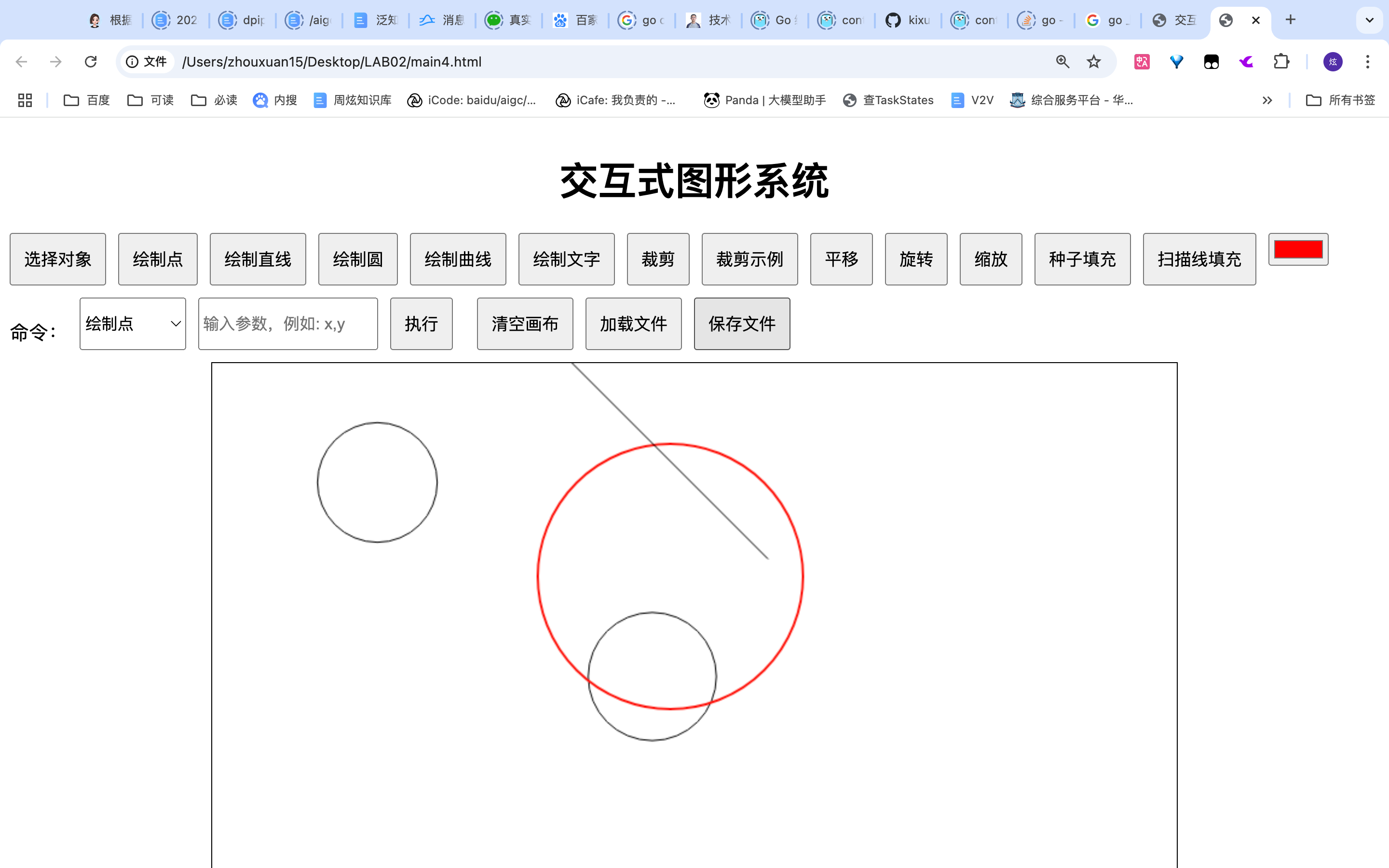
1. 测试旋转功能：选中直线进行旋转45度



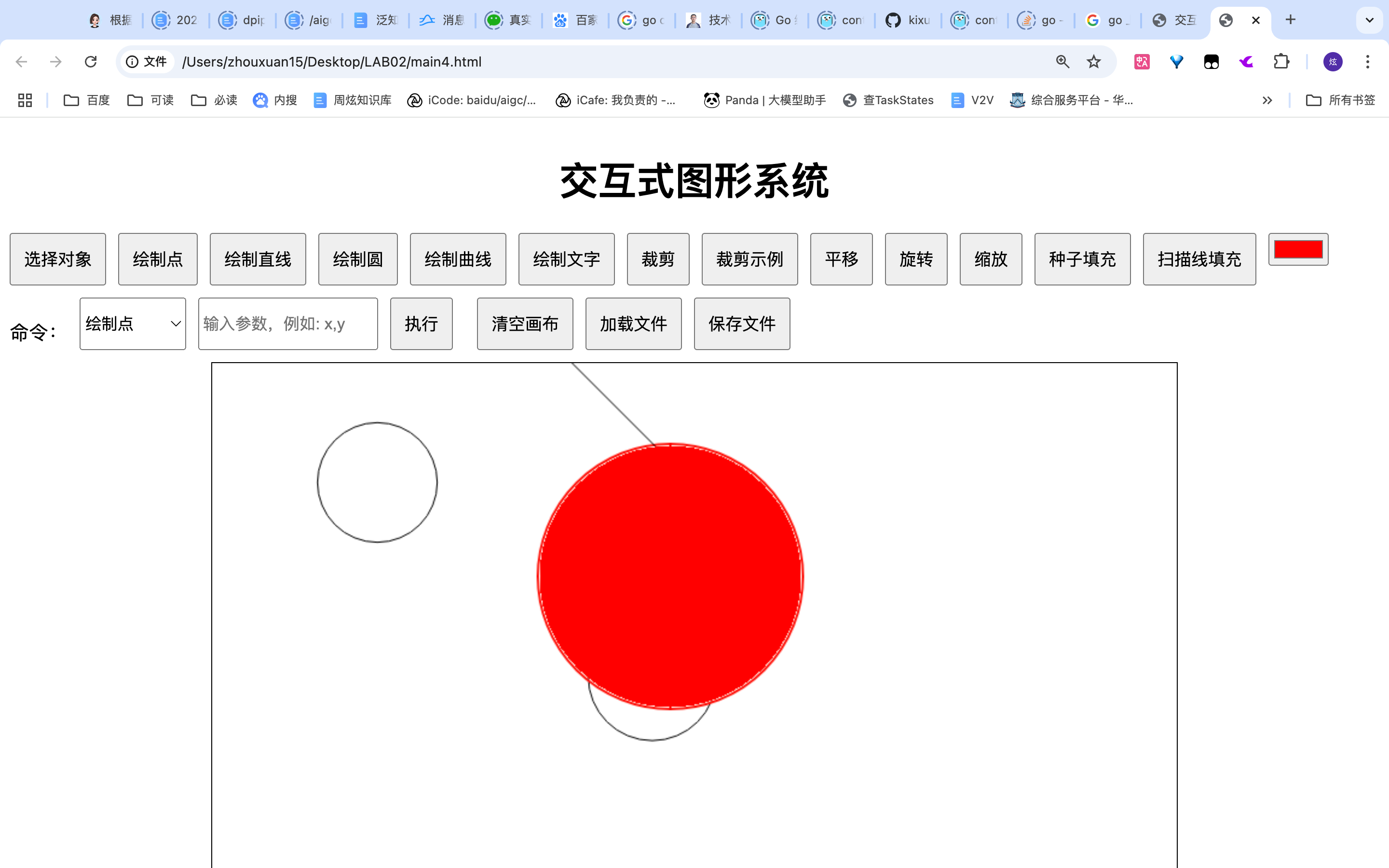
旋转后的直线为45度



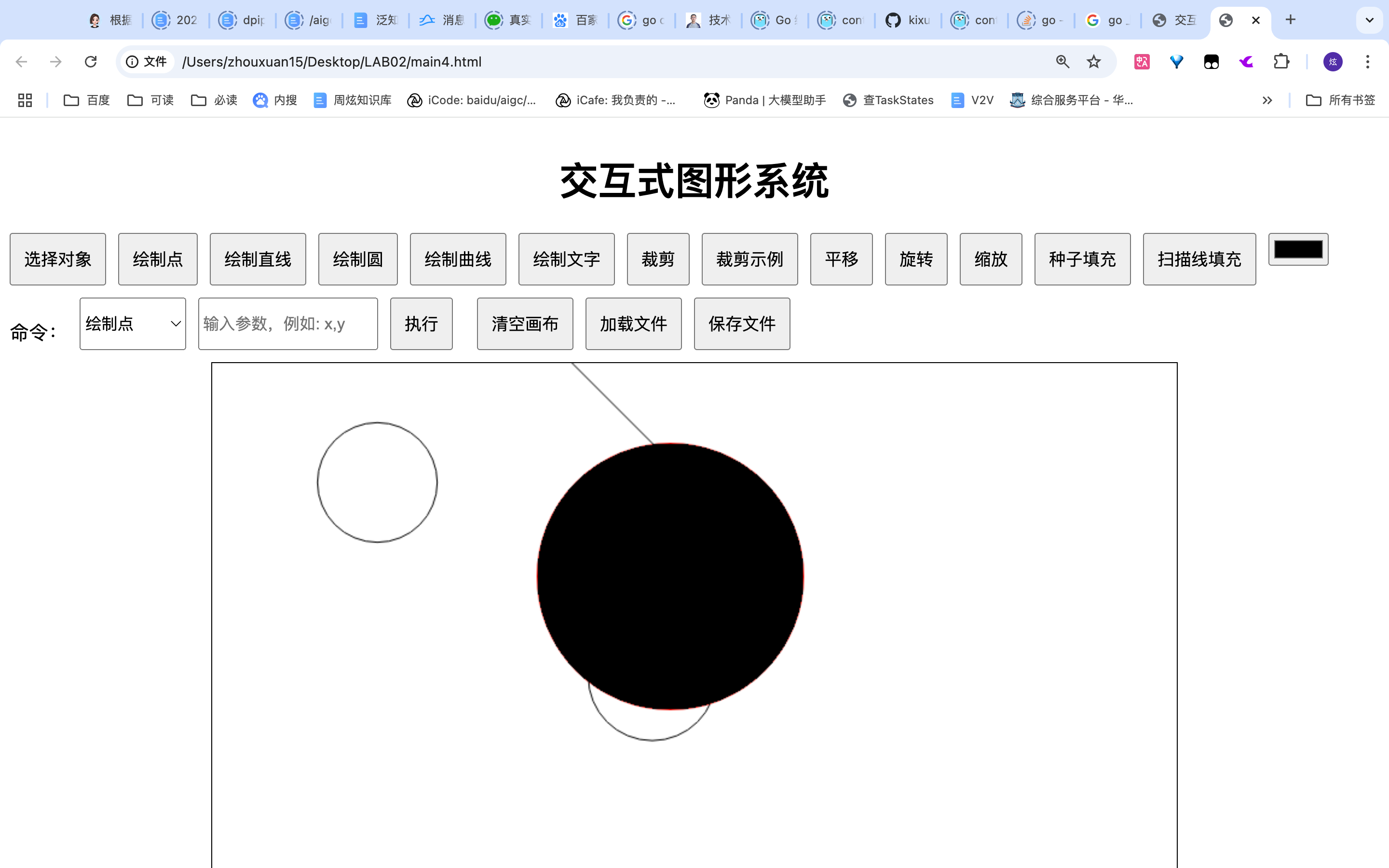
1. 测试缩放功能：在原图基础上对圆进行2倍缩放



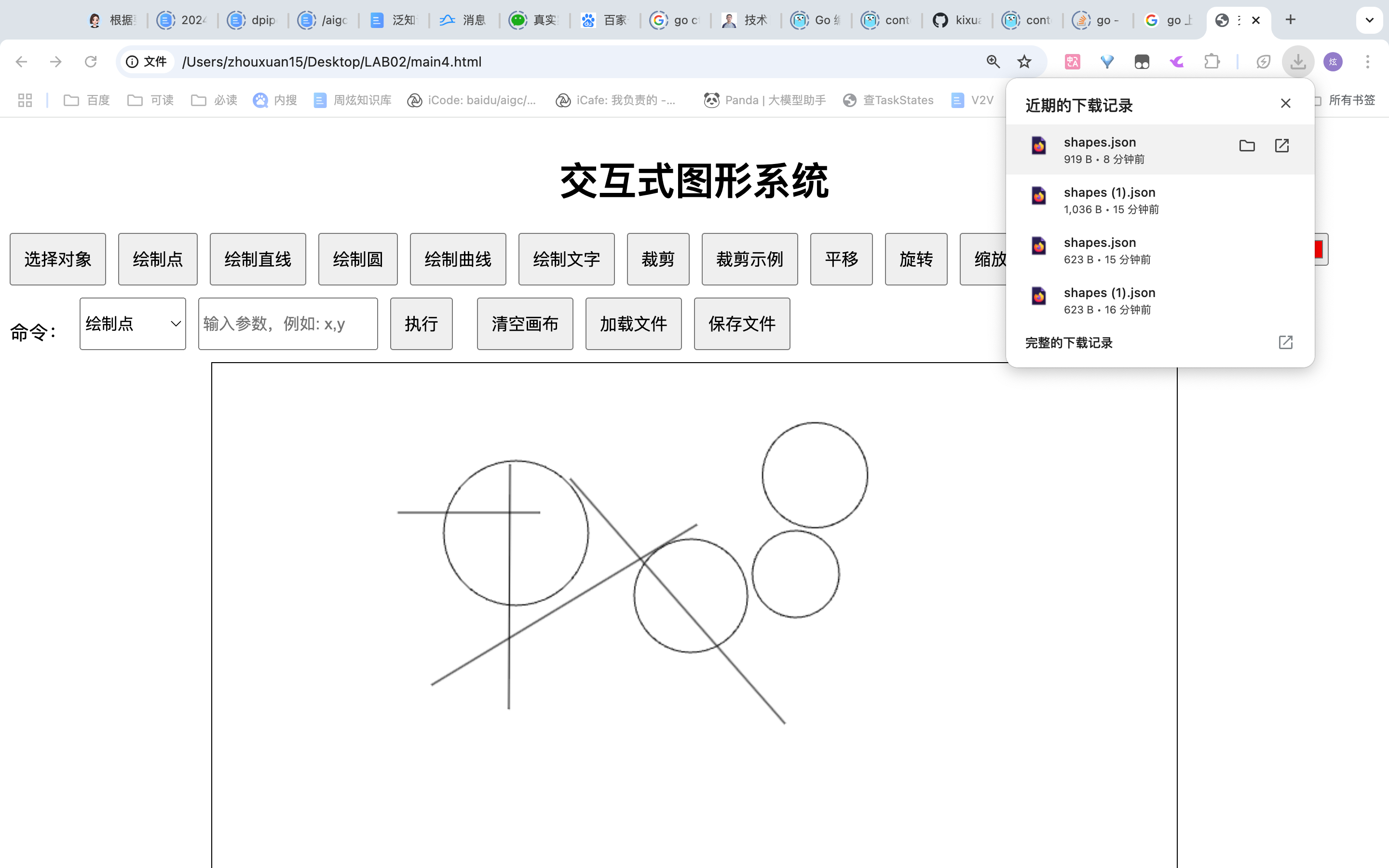
1. 种子填充功能：放大可以发现细节符合种子填充特点



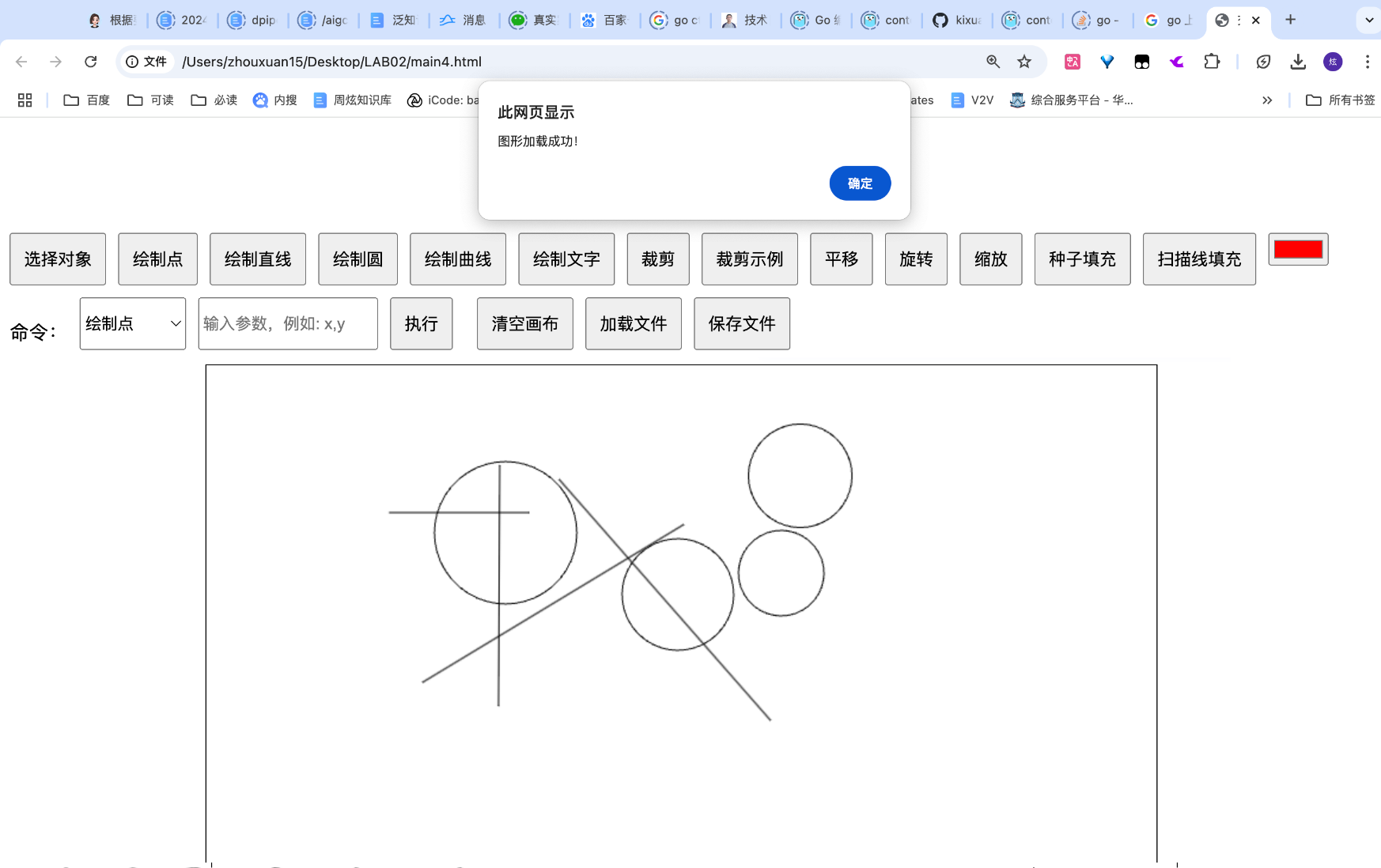
1. 扫描线填充功能：放大可以发现细节符合扫描线填充特点



1. 保存图形文件，这里选择了使用json格式进行保存，存储图形的位置及大小



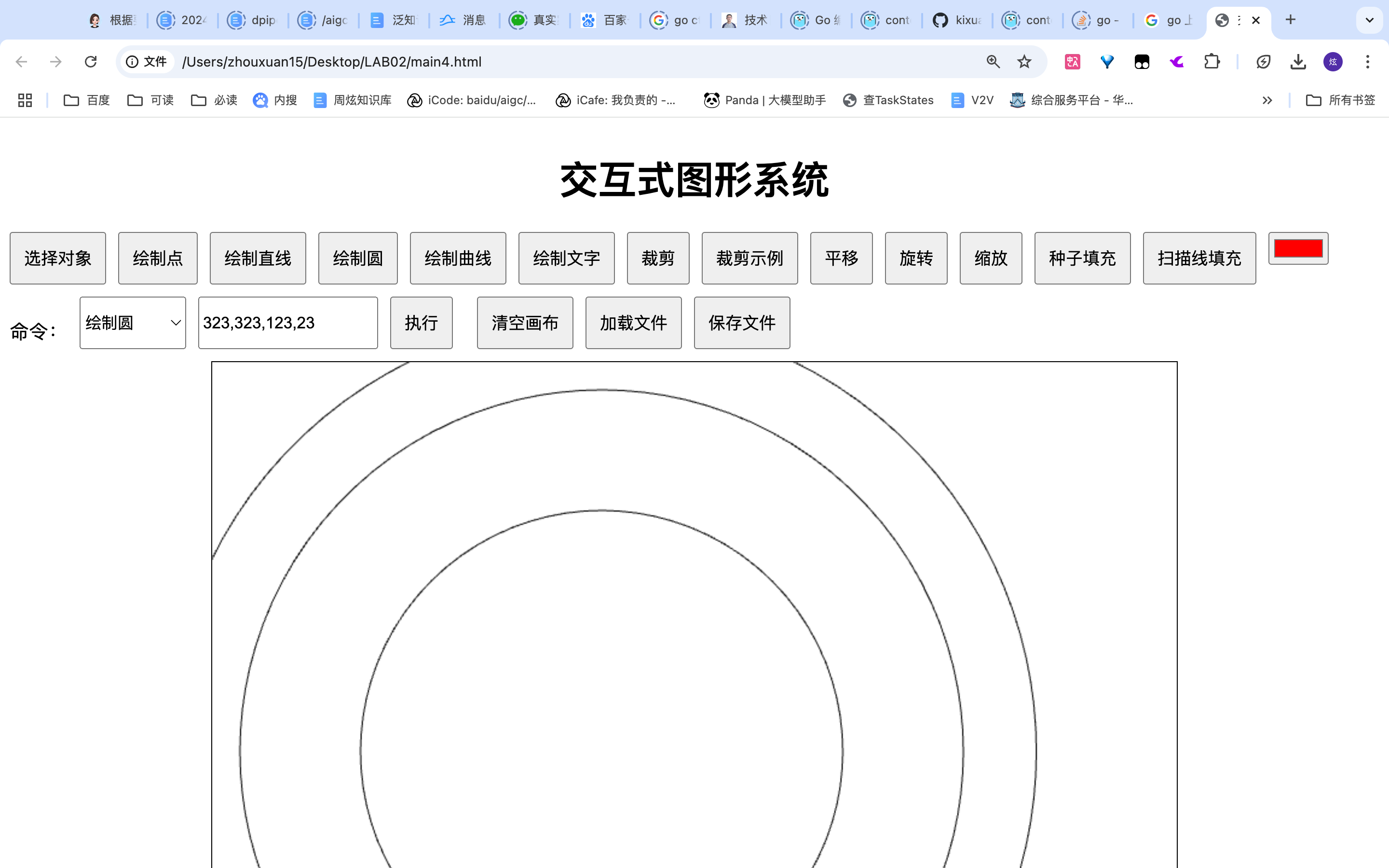
1. 加载文件功能，发现复原之前的图形



1. 清空画布功能



1. 命令行功能：为了简便操作进行了按钮简化，也可以通过输入命令进行相关操作



## **五、问题分析与解决**

### **（一）图形绘制精度问题**

在绘制复杂图形或进行多次变换后，图形边缘出现轻微锯齿和模糊现象。这是由于在算法实现中，像素绘制的离散性和坐标计算的舍入误差累积导致。通过优化算法中的坐标计算精度，如在贝塞尔曲线绘制时增加参数步长的细分程度，以及在绘制直线和圆时对边缘像素的抗锯齿处理（如采用半透明像素绘制边缘），在一定程度上缓解了该问题，使图形显示更加平滑。

### **（二）裁剪算法效率问题**

对于复杂图形和大量图形的裁剪操作，Cohen - Sutherland 直线裁剪算法在某些极端情况下计算量较大，影响系统响应速度。分析发现是因为在求交计算和编码判断过程中存在重复计算。通过引入空间划分和提前判断机制，对图形进行预分区，减少不必要的求交计算，显著提高了裁剪算法的效率，提升了系统在处理复杂裁剪场景时的性能。

### **（三）填充算法局限性问题**

种子填充算法在处理复杂连通区域或非圆形图形时存在缺陷，容易出现填充泄漏或不完全填充情况。这是由于其简单的基于四邻域或八邻域的搜索方式在复杂图形结构下不适用。考虑后续研究采用更先进的扫描线种子填充算法或区域生长算法替代，以增强填充功能的通用性和稳定性，适应更多类型图形的填充需求。

## **六、总结与展望**

通过本次实验，我成功构建了一个功能较为完善的交互式图形系统，深入掌握了计算机图形学的核心算法与技术。在实验过程中，对图形生成、变换、裁剪和填充等操作的原理与实现细节有了透彻理解，提升了算法设计、问题解决和编程实践能力。

未来可进一步拓展图形类型支持，如多边形的复杂操作（布尔运算等）；优化算法性能，引入硬件加速技术提升绘制和处理速度；增强用户交互体验，添加撤销 / 重做功能、图形分层管理等；探索与其他图形库或技术的融合，如 WebGL，实现更高级的 3D 图形效果和交互功能，不断完善图形系统的功能与性能。