《计算机图形学》课程设计报告

题　　目： 基于 h5 的在线画布工具实现

姓　　名： 柳景兴 201622111920052

姓　　名： 邢文圣 201622111920103

姓　　名： 王晨旭 201622111920287

姓　　名： 刘俊州 201622111920121

姓　　名： 刘楚雄 201622111920026

姓　　名： 侯静怡 201622111920096

姓　　名： 陈雪　 201622111920289

姓　　名： 袁嘉豪 201622111920023

姓　　名： 常思攀 201622111920250

专业年级： 2016 软件工程产业计划

指导教师： 余敦辉

职　　称： 副教授

2019 年 5 月 29 日

基于 h5 的在线画布实现

摘要：计算机图形学是一种使用数学算法将二维或三维图形学转化为计算机显示器的栅格形式的科学，主要研究内容就是在计算机中表示图形、利用计算机进行图形的计算、处理和显示的相关原理与算法。随着 html 的不断完善和发展,它能够提供新的常用元素、表单功能、图形绘制、图像处理等功能。其中 html5 的 canvas 具有强大高效的绘图特性，被广泛应用于客户端的游戏开发及图像处理中，丰富了信息操作的方式，给客户带来了更好的用户体验。本文主要研究计算机图形学中图像显示和变换的原理和算法，并通过 html 实现。

关键字：计算机图形学；画图；html；canvas

目录

第一章 绪论

1.1 研究的背景

计算机图形学是随着计算机及其外围设备而产生和发展起来的。它是近代计算机科学与雷达电视及图象处理技术的发展汇合而产生的硕果。在造船、航空航天、汽车、电子、机械、土建工程、影视广告、地理信息、轻纺化工等领域中的广泛应用，推动了这门学科的不断发展，而不断解决应用中提出的各类新课题，又进一一步充实和丰富了这门学科的内容。计算机出现不久,为了在绘图仪和阴极射线管(CRT)屏幕上输出图形，计算机图形学随之诞生了。现在它已发展为对物体的模型和图象进行生成、存取和管理的新学科。[1]

canvas 是 HTML5 的一个自带的绘制图形图表、制作动画的标签,它本身没有绘图能力，只是一个放置在 Web 页面上的画布，但它提供了许多方法用来绘制路径、图表和字符等,实现图形绘制和动画制作可以使用 JavaScript 代码在 Web 页面上实现，不需要第三方插件。Canvas 可以直接在客户端渲染图形或动画，无需在服务器端加工,避免了服务器端存在运算速度不足、带宽有限等问题。

HTML5 之前，Web 实现绘制图形是使用 SVG、VML 等技术。SVG、 VML 是通过 XML 文档描绘图形来实现一个矢图形。矢量图形不能满足现代移动设备的位图级别图像的需求,而 HTML5 引入 Canvas 后，可以在 Web 页面直接生成的位图级别的图像,画布区域中的每一格像素都是可以控制的,可以生成复杂图形，甚至可以满足游戏界面等复杂的开发要求。[2]

1.2 本文研究内容与目的

本文结合计算机图形学中基本图形的扫描转换和图形变换原理与算法，如一维线框图形直线、圆、椭圆的扫描转换问题，二维多边形的填充，字符的表示，图形的裁剪，由简单图形生成复杂图形，二维三维图形的平移、旋转、变比、对称等，在 html 中的 canvas 实现出来。

第二章 需求分析

2.1 主要功能描述

用户可以在画布上画任意图形，可以保存、撤销、恢复等，且界面有良好操作提示等，具体如下：

(1) 选择画图工具：用户可以通过侧边工具栏选择想要使用的画图工具，且每种工具有相应的提示。

(2) 画图：根据用户选择的画图工具，在画布上通过鼠标点击移动，绘制点、直线、折线、直角线、多边形、圆、椭圆、圆弧、椭圆弧、任意曲线、圆角矩形、矩形及其字符；擦除；填充，且用户可以选择绘制的线宽及线型。

(3) 背景：用户可以更改画布的背景，且画布有网格线、标尺。

(4) 图形变换：用户可以选定在画布上的图元，在图元周围显示热点，用户根据热点拖动鼠标移动图元、缩放、旋转、对称图元。图形在移动过程中能够根据网格线自动对。

(5) 图元复制、剪切、粘贴：用户点击按钮或使用快捷键，对选择的图元进行复制、剪切与粘贴。

(6) 图形的组合与打散：用户选择多个图元，根据提示将图元组合和打散。
(7) 操作的撤销与恢复：用户点击上方菜单栏或使用快捷键撤销和恢复上一步操作。
(8) 保存：用户点击上方菜单栏或使用快捷键，对当前画布进行保存。

2.2 开发环境

Hbuilder

第三章 总体设计

3.1 软件结构设计

根据功能描述，界面结构描述如下：

1) 菜单栏的设计及实现功能：上方菜单栏包括保存、撤销、恢复、帮助等。
2) 画图区的设计及实现功能：中间画布包括可以调整背景颜色的画布、可以实现图元自动对齐的网格线。
3) 工具栏的设计及功能实现：左侧工具栏包括直线工具、矩形工具、圆形椭圆形工具、折线工具、裁剪工具、画多边形工具、铅笔工具、字符工具；选择线宽、线型；选择颜色。

总体结构图如下：

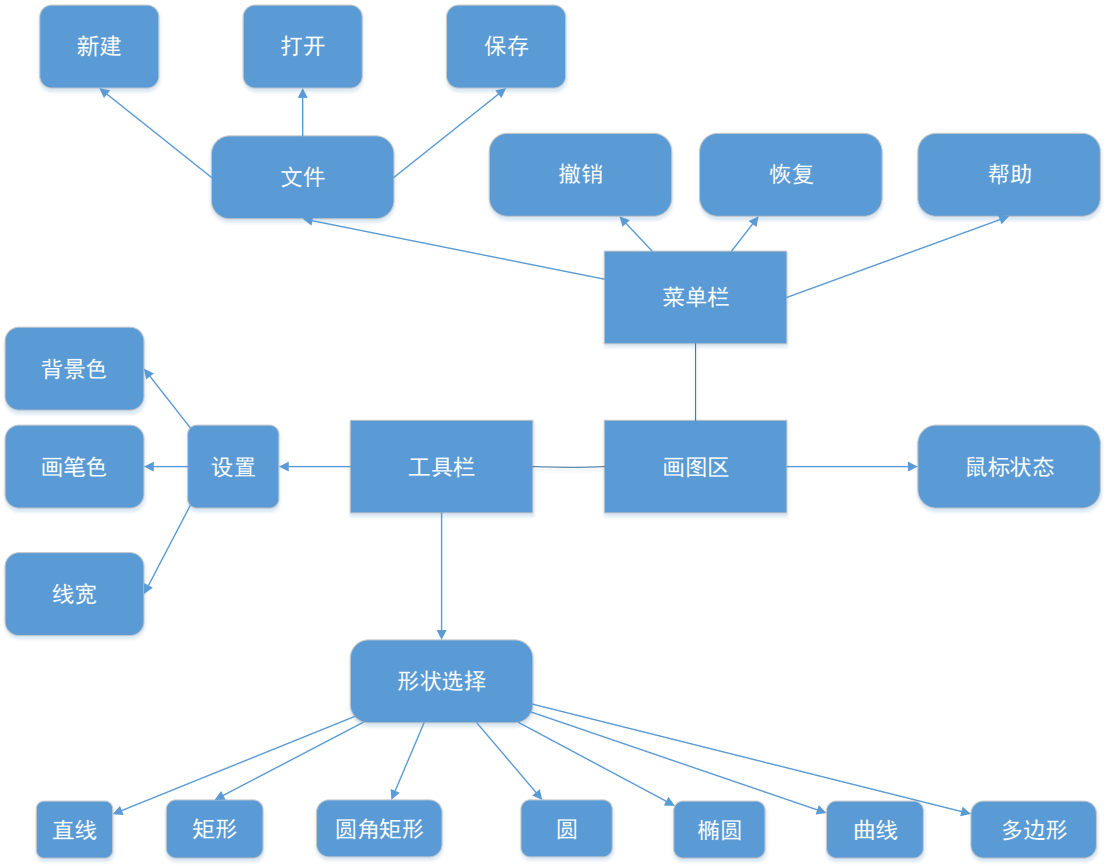

图 3.1

第四章 功能实现

4.1 主界面设计

主界面如图 4.1 所示：

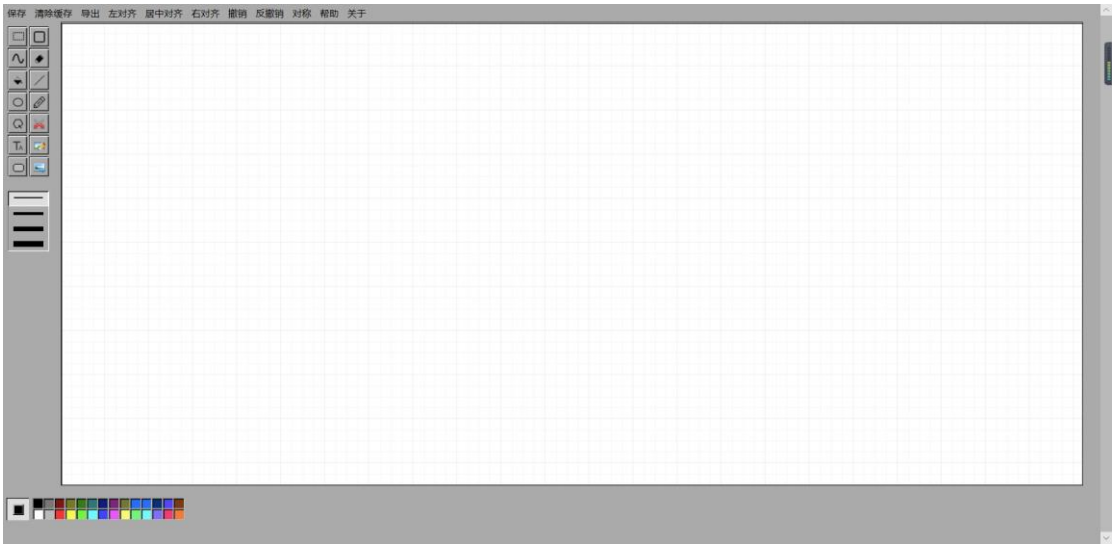

图 4.1

4.2 绘制基本图元

4.2.1 三种画直线方法

1、数值微分法 DDA:

基本思想：数值微分算法的本质是用数值方法解微分方程，通过同时对 x 和 y 各增加一个增量，计算下一步的 x、y 值。

先计算出直线的斜率，k=Δx/Δy，计算出y_{i+1}=y_i+kΔx

因此当Δx=1 时，有y_{i+1}=y_i+k，即每当 x 增加 1 时，y 增加 k，对 y 四舍五入来进行描点，由此得出 DDA 算法：

代码实现：

```
int Round(float x)  //做四舍五入
        {
            return (int)(x < 0 ? x - 0.5 : x + 0.5);
        }
        void DrawLine_DDA(int x1, int y1, int x2, int y2)
        {
            Bitmap image = new Bitmap(pictureBox1.Width, pictureBox1.Height);
            pictureBox1.Image = image;
            Graphics g = Graphics.FromImage(image);

            float x, y;      // 当前坐标点
            float dx, dy;    // x、y 方向上的增量
```

```
            int steps = Math.Abs(x2 - x1) > Math.Abs(y2 - y1) ? Math.Abs(x2 - x1) :
Math.Abs(y2 - y1);    //steps=x,y中变化量的最大值

            x = (float)x1;
            y = (float)y1;
            dx = (float)(x2 - x1) / steps;    //平均变化量
            dy = (float)(y2 - y1) / steps;    //平均变化量

            for (int i = 0; i < steps; i++)
            {
                g.FillEllipse(Brushes.Black, Round(x), Round(y), 2, 2);
                x += dx;
                y += dy;
            }
        }
```

2、中点画线算法：

基本思想：若实际直线经过两像素点中点的上方，则描右上方的点，否则描正右方的点。

直线方程可表示为：F(x,y)=a*x+b*y+c

其中：a=Y0-Y1, b=X1-X0, c=X0*Y1-X1*Y0。将中点（Xt，Yt）带入直线方程，

当：F(Xt,Yt) = 0 中点在直线上

F(Xt,Yt) < 0 中点在直线下方

F(Xt,Yt) > 0 中点在直线上方

可构造判别式：d = F(M)=F(Xi+1,Yi+0.5) = a(Xi+1)+b(Yi+0.5)+c

d 的初始值 d0=a+0.5b，由于只用于判断符号，可以取 d0=2a+b

当 d < 0 时， 取下方的像素点，此时再下一个象素的判别式为：

d’ = F(Xi+2,Yi+1.5) = d + a + b;

当 d>= 0 时，取上方的像素点，此时再下一个象素的判别式为：

d’ = F(Xi+2,Yi+0.5) = d + a;

代码实现：

```
    public void DrawLine_Midle(int x0, int y0, int x1, int y1)
        {
            Bitmap image = new Bitmap(pictureBox1.Width, pictureBox1.Height);
            pictureBox1.Image = image;
            Graphics g = Graphics.FromImage(image);

            Point point1 = new Point(0, 0);


            int a, b, d1, d2, d, x, y;
            float m;
            if (x1 < x0)
            {
```

```
    //交换初始点和结束点
    d = x0;
    x0 = x1;
    x1 = d;

    d = y0;
    y0 = y1;
    y1 = d;
}

a = y0 - y1;
b = x1 - x0;
if (b == 0)  //初始点和结束点在同一水平线上
    m = -1 * a * 100;
else
    m = (float)a / (x0 - x1);  //m=dy/dx
x = x0;
y = y0;
g.FillEllipse(Brushes.Black, x, y, 2, 2);   //填充第一个点
if (m >= 0 && m <= 1)  //斜率 0<m<1
{
    d = 2 * a + b;
    d1 = 2 * a;
    d2 = 2 * (a + b);
    while (x < x1)
    {
        if (d <= 0)
        {  //取右上方的点
            x++;
            y++;
            d += d2;
        }
        else
        {  //取正右方的点
            x++;
            d += d1;
        }
        g.FillEllipse(Brushes.Black, x, y, 2, 2);
    }
}
else if (m <= 0 && m >= -1)    //斜率-1<m<0
{
    d = 2 * a - b;
```

```
    d1 = 2 * a - 2 * b;
    d2 = 2 * a;
    while (x < x1)
    {
        if (d > 0)
        {       //取右下方的点（中点在直线的上方）
            x++;
            y--;
            d += d1;
        }
        else
        {
            x++;
            d += d2;
        }
        g.FillEllipse(Brushes.Black, x, y, 2, 2);
    }
}
else if (m > 1)//斜率 m>1
{
    d = a + 2 * b;
    d1 = 2 * (a + b);
    d2 = 2 * b;
    while (y < y1)
    {
        if (d > 0)
        {   //取右上方的点
            x++;
            y++;
            d += d1;
        }
        else
        {   //取正上方的点
            y++;
            d += d2;
        }
        g.FillEllipse(Brushes.Black, x, y, 2, 2);
    }
}
else//竖直的线
{
    d = a - 2 * b;
    d1 = -2 * b;
```

```
                d2 = 2 * (a - b);
                while (y > y1)
                {
                    if (d <= 0)
                    {          //取右下方的点
                        x++;
                        y--;
                        d += d2;
                    }
                    else
                    {      //取正下方的点
                        y--;
                        d += d1;
                    }
                    g.FillEllipse(Brushes.Black, x, y, 2, 2);
                }
            }
        }
```

3、Bersenham 画线算法

基本思想：若实际直线经过的点与正右方的点的差值 d 大于 0.5，则取右上方的点，否则，取正右方的点。

令 dx=x2-x1，dy=y2-y1

递推公式 ：d_{i+1}=d_i+2dy-2dx(y_i-y_{i-1})

di 的初值： d_1=2dy-dx

当 di≥0 时，选右上方的点，d_{i+1}=d_i+2(dy-dx)

当 di<0 时，选正右方的点，d_{i+1}=d_i+2dy

代码实现：

```
public void DrawLine_Bresenham(int x0, int y0, int x1, int y1)
        {
            Bitmap image = new Bitmap(pictureBox1.Width, pictureBox1.Height);
            pictureBox1.Image = image;
            Graphics g = Graphics.FromImage(image);
            Point point1 = new Point(0, 0);
            int dx = Math.Abs(x1 - x0);
            int dy = Math.Abs(y1 - y0);
            int x = x0;
            int y = y0;
            int stepX = 1;
            int stepY = 1;
            if (x0 > x1)  //从右向左画
                stepX = -1;
            if (y0 > y1)  //从上向下画
                stepY = -1;
```

```
        if (dx > dy)
        //沿着最长的那个轴前进      （x 的变化值比 y 大）
        {
            //int e = dy * 2 - dx;              //（e 的初始值 e=-dx）
            int e = -dx;
            for (int i = 0; i <= dx; i++)
            {
                g.FillEllipse(Brushes.Black, x, y, 2, 2);
                x += stepX;
                e = e + 2*dy;                   //ei=e+dy+dy
                if (e >= 0)  //画右上方的点
                {
                    y += stepY;
                    e = e - 2 * dx;             //ei=e-2dx
                }
            }
        }
        else             //x 的变化率比 y 小
        {
            int e = - dy;
            for (int i = 0; i <= dy; i++)
            {
                g.FillEllipse(Brushes.Black, x, y, 2, 2);
                y += stepY;
                e = e + 2 * dx;
                if (e >= 0)
                {
                    x += stepX;
                    e = e - 2 * dy;
                }
            }
        }
    }
```

4.2.2 圆与椭圆

中点画圆：

基本思想：

构造圆的函数：

F（X，Y）=$X^2 + Y^2 - R^2$

F（X，Y）= 0 点在圆上；F（X，Y）< 0 点在圆内；F（X，Y）> 0 点在圆外。

M 为 P1、P2 间的中点，M=(Xp+1,Yp-0.5)有如下结论：F（M）< 0 取 P1，否则取 P2

构造判别式： d = F(M)= F(xp + 1, yp - 0.5)=(xp + 1)2 + (yp - 0.5) 2 - R2

若 d<0, 则右上方为下一个象素，那么再下一个象素的判别式为： d= d + 2xp +3；

若 d>=0,正右方为下一个象素，那么再下一个象素的判别式为：d= d+2(xp - yp) + 5

代码实现：

```
MidpointCircle(r, color)
 int r, color;
{
    int x,y;
    float d;
    x=0; y=r; d=1.25-r;
   drawpixel(x,y,color);
   while(x<y)
   {
        if(d<0)
         {
            d+ = 2*x+3;
            x++;
         }
         else
          {
            d+ = 2*(x-y) + 5;
             x++;y--;
          }
    }
}
```

4.2.3 矩形

由四条直线组成，记录矩形左上角的坐标和矩形的长和宽等并保存下来。

4.2.4 多边形

画法同矩形

4.2.5 折线

4.2.6 曲线

贝塞尔曲线：

1) 贝塞尔曲线点的数量决定了曲线的阶数，一般 N 个点构成的 N-1 阶贝塞尔曲线，即 3 个点为二阶，至少由 3 个点组成，第一个点为起点，最后一个点为终点，其余点都为控制点；
2) 各个点按顺序连接起来，形成直线，将整个曲线的绘制量化为从 0~1 的过程，progress 为当前过程的进度，progress 的区间即 0~1。每一条线都需要根据 progress 生成一个点，一个点从 P0 移动到 P1,这是这条线从 0~1 的过程；
3) 绘制一个二阶贝塞尔曲线过程为：由 A、B、C 这 3 个点组成 2 条线 AB 和 BC，2 条线根据 progress 分别生成 2 个移动的点 D 和 E，而 D 和 E 又连成一条线，始终保持 AD:DB=BE:EC。DE 再根据 progress 生成点 F，只剩一个点，无法构成线，即为最终构成贝塞尔曲线的点；
4) 经过上面 点生线，线生点 的过程 ，我们拿到了点 F 在移动中所有点的，将这些点集合连接起来，即形成了贝塞尔曲线；
5) 点生线，线生点 是一个递归的过程，通过底层的点，一步步推算出最高阶的点。整个推导过程像一个金字塔，底部点的数量最多，每高一阶点的数量就减 1，直至

最高阶只有 1 个点。

4.2.8 线型线宽

线刷子：

原理：设直线斜率在【－1，1】之间，此时可把刷子置成垂直方向，刷子的中点对准直线某一端点，然后让刷子中心往直线的另一端移动，即可刷出具有一定宽度的线。

方刷子：

原理：把边宽为指定线宽的正方形的中心对准单象素宽的线条上的各个象素，并沿直线作平行移动，即可获得具有线宽的线条。

线型的改变：首先由基本的画直线方法，设置不同线型的二进制数组，根据选择的线型，循环输出该二进制数组即实现不同线型的画法。

4.3 填充

4.3.1 有序边表填充

1) 判断凸多边形和凹多边形：凸多边形：对于多边形内任意两点，连接他们的直线上的所有点均在该多边形内部，该多边形即为凸多边形；凹多边形：不满足上述条件的多边形即为凹多边形。
2) 内外点的判定：从无穷远处向该点引一条射线，如果这条射线与多边形的交点为奇数时，此点为内点，否则为外点。
3) 边界的处理：左闭右开，上闭下开
4) 确定扫描线，求扫描线与各边的交点
5) 进行区间填充

```
//多边形填充算法，点击按钮后调用此函数
        public void fill_polygon(Bitmap image)
        {
            y_min = pictureBox1.Height - point_list[0].Y;
            y_max = pictureBox1.Height - point_list[0].Y;
            //先将所有的边存入 Edge_list 中
            for (int i = 0; i < point_list.Count - 1; i++)
            {
                //更新最大最小纵坐标
                y_min = Math.Min(y_min, pictureBox1.Height - point_list[i].Y);
                y_max = Math.Max(y_max, pictureBox1.Height - point_list[i].Y);

                Edge one_edge = new Edge();
                //x 为下端点的横坐标，需要先判断那个点是下端点
                if (pictureBox1.Height - point_list[i].Y > pictureBox1.Height - point_list[i + 1].Y)
                {
                    one_edge.x = point_list[i + 1].X;
                }
                else if (pictureBox1.Height - point_list[i].Y < pictureBox1.Height - point_list[i + 1].Y)
                {
                    one_edge.x = point_list[i].X;
                }
```

```
                    //如果这条线是水平的则不存入 EdgeTable 中
                    else if (point_list[i].Y == point_list[i + 1].Y)
                    {
                        continue;
                    }
                    //y_max 为上端点的 y 坐标
                    one_edge.y_max = Math.Max(pictureBox1.Height - point_list[i].Y, pictureBox1.Height -
point_list[i + 1].Y);
                    //y_min 为下端点的 y 坐标
                    one_edge.y_min = Math.Min(pictureBox1.Height - point_list[i].Y, pictureBox1.Height -
point_list[i + 1].Y);

                    //delta_x
                    if (pictureBox1.Height - point_list[i].Y > pictureBox1.Height - point_list[i + 1].Y)
                    {
                        one_edge.delta_x = (double)(point_list[i].X - point_list[i + 1].X) /
(double)(pictureBox1.Height - point_list[i].Y - pictureBox1.Height +point_list[i + 1].Y);
                    }
                    else
                    {
                        one_edge.delta_x = (double)(point_list[i + 1].X - point_list[i].X) /
(double)(pictureBox1.Height - point_list[i + 1].Y - pictureBox1.Height +point_list[i].Y);
                    }
                    //加入列表中
                    Edge_list.Add(one_edge);
                }

                //最后首位相连的边
                //更新最大最小纵坐标
                y_min = Math.Min(y_min, pictureBox1.Height - point_list[point_list.Count - 1].Y);
                y_max = Math.Max(y_max, pictureBox1.Height - point_list[point_list.Count - 1].Y);
                Edge last = new Edge();
                if (point_list[0].Y == point_list[point_list.Count - 1].Y)
                {
                    //水平的线不要
                }

                //按照上端点的不同分两种情况
                else if (pictureBox1.Height - point_list[0].Y > pictureBox1.Height - point_list[point_list.Count -
1].Y)
                {
                    last.x = point_list[point_list.Count - 1].X;
```

```
                    //y_max 为上端点的 y 坐标
                    last.y_max = Math.Max(pictureBox1.Height - point_list[0].Y, pictureBox1.Height -
point_list[point_list.Count - 1].Y);
                    //y_min 为下端点的 y 坐标
                    last.y_min = Math.Min(pictureBox1.Height - point_list[0].Y, pictureBox1.Height -
point_list[point_list.Count - 1].Y);

                    //delta_x
                    last.delta_x = (double)(point_list[0].X - point_list[point_list.Count - 1].X) /
(double)(pictureBox1.Height - point_list[0].Y - pictureBox1.Height +point_list[point_list.Count - 1].Y);

                    //加入列表中
                    Edge_list.Add(last);

                }
                else if (pictureBox1.Height - point_list[0].Y < pictureBox1.Height - point_list[point_list.Count -
1].Y)
                {
                    last.x = point_list[0].X;

                    //y_max 为上端点的 y 坐标
                    last.y_max = Math.Max(pictureBox1.Height - point_list[0].Y, pictureBox1.Height -
point_list[point_list.Count - 1].Y);
                    //y_min 为下端点的 y 坐标
                    last.y_min = Math.Min(pictureBox1.Height - point_list[0].Y, pictureBox1.Height -
point_list[point_list.Count - 1].Y);


                    //delta_x
                    last.delta_x = (double)(point_list[point_list.Count - 1].X - point_list[0].X) /
(double)(pictureBox1.Height - point_list[point_list.Count - 1].Y - pictureBox1.Height + point_list[0].Y);

                    //加入列表中
                    Edge_list.Add(last);
                }

                //所有的边已经生成，需要对其按照下端点坐标进行分类，将下端点坐标相同的分到同一类
                //先对下端点的 y 坐标进行排序，这样同一类的点会被排在相邻的位置
                Edge_list.Sort(compare_ymin);

                //遍历列表，将下端点 y 坐标相同的点加入同一个数组里，并将这个数组压入 ET 中
                for (int i = 0; i < Edge_list.Count; i++)
                {
```

```
            List<Edge> one_list = new List<Edge>();
            one_list.Add(Edge_list[i]);

            //如果下一个和这一个 y_min 相同的话则不断加入新的边，直到不相同
            if (i < Edge_list.Count - 1) //后面还有一条边就判断是否属于一个类，没有的话直接加
入 ET 就行
            {
                if (Edge_list[i + 1].y_min == Edge_list[i].y_min)
                {
                    while (Edge_list[i + 1].y_min == Edge_list[i].y_min)
                    {
                        one_list.Add(Edge_list[i + 1]);
                        i++;
                        //如果超出了数组范围就跳出
                        if (i == Edge_list.Count - 1)
                        {
                            break;
                        }
                    }
                }
            }
            //将这一类的边加入 ET 中
            ET.Add(one_list);
        }


        //开始填充算法
        for (int i = y_min; i <= y_max; i++)
        {
            //将 ET 表中该类边都提取出来，添加到 AEL 中
            for (int j = 0; j < ET.Count; j++)
            {
                if (ET[j][0].y_min == i)
                {
                    //将边添加到活化链表里
                    AEL = AEL.Union(ET[j]).ToList<Edge>();

                    //将下端点的坐标加入列表中作为初始值
                    for (int k = 0; k < ET[j].Count; k++)
                    {
                        x_list.Add(ET[j][k].x);
                    }
                    break; //跳出
```

```
            }
        }

        //对交点进行排序后开始画图,进行着色
        if (x_list.Count > 0)
        {
            x_list_sort.Clear();
            //因为引用赋值的话会有问题，所以采用这种循环的方式赋值
            for (int m = 0; m < x_list.Count; m++)
            {
                double x = x_list[m];
                x_list_sort.Add(x);
            }
            x_list_sort.Sort();
            draw(i, x_list_sort, image);
        }

        //删除边,遍历 AEL，如果 i+1 等于某条边的 y_max 则删去这条边，同时删去对应的 x
        int num = AEL.Count;
        int remove_num = 0;
        for (int j = 0; j < num; j++)
        {
            if (i + 1 == AEL[j - remove_num].y_max)
            {
                AEL.RemoveAt(j - remove_num);
                x_list.RemoveAt(j - remove_num);
                remove_num++;
            }
        }

        //更新交点坐标
        if (x_list.Count > 0)
        {
            for (int j = 0; j < x_list.Count; j++)
            {
                x_list[j] = x_list[j] + AEL[j].delta_x;
            }
        }
    }
}
```

4.3.2 边填充

正负相消算法：

基本思想：对于每一条扫描线和每条多边形的交点，将该扫描线上交点右方所有的象素

取补。对多边形的每一条边作此处理，多边形各边的处理顺序随意。

```
public void fill_edge(Bitmap image)
        {
         //     Bitmap picBackGround = new Bitmap(this.pictureBox1.Width, this.pictureBox1.Height);

            Bitmap imageBox = new Bitmap(pictureBox1.Width, pictureBox1.Height);
            pictureBox1.Image = image;
            Graphics g = Graphics.FromImage(image);



            //画布的宽和高
            int pbx = pictureBox1.Width;
            int pby = pictureBox1.Height;

            //获取（x,y）的颜色
            //Color currentColor = picBackGround.GetPixel(x, y);

            //将边加入边列表
            //先将所有的边存入 Edge_list 中
            for (int i = 0; i < point_list.Count - 1; i++)
            {

                Edge one_edge = new Edge();
                //x 为下端点的横坐标，需要先判断那个点是下端点
                if ( point_list[i].Y >    point_list[i + 1].Y)
                {
                    one_edge.x = point_list[i + 1].X;
                }
                else if ( point_list[i].Y <    point_list[i + 1].Y)
                {
                    one_edge.x = point_list[i].X;
                }
                //如果这条线是水平的则不存入 EdgeTable 中
                else if (point_list[i].Y == point_list[i + 1].Y)
                {
                    continue;
                }
                //y_max 为上端点的 y 坐标
                one_edge.y_max = Math.Max( point_list[i].Y,    point_list[i + 1].Y);
                //y_min 为下端点的 y 坐标
                one_edge.y_min = Math.Min( point_list[i].Y,    point_list[i + 1].Y);
```

```
                    //delta_x
                    if ( point_list[i].Y >  point_list[i + 1].Y)
                    {
                        one_edge.delta_x = (double)(point_list[i].X - point_list[i + 1].X) /
(double)( point_list[i].Y -  point_list[i + 1].Y);
                    }
                    else
                    {
                        one_edge.delta_x = (double)(point_list[i + 1].X - point_list[i].X) /
(double)( point_list[i + 1].Y -  point_list[i].Y);
                    }
                    //加入列表中
                    Edge_list.Add(one_edge);
                }

                //最后首位相连的边
                Edge last = new Edge();
                if (point_list[0].Y == point_list[point_list.Count - 1].Y)
                {
                    //水平的线不要
                }

                //按照上端点的不同分两种情况
                else if ( point_list[0].Y >  point_list[point_list.Count - 1].Y)
                {
                    last.x = point_list[point_list.Count - 1].X;

                    //y_max 为上端点的 y 坐标
                    last.y_max = Math.Max( point_list[0].Y,  point_list[point_list.Count - 1].Y);
                    //y_min 为下端点的 y 坐标
                    last.y_min = Math.Min( point_list[0].Y,  point_list[point_list.Count - 1].Y);

                    //delta_x
                    last.delta_x = (double)(point_list[0].X - point_list[point_list.Count - 1].X) /
(double)( point_list[0].Y -  point_list[point_list.Count - 1].Y);

                    //加入列表中
                    Edge_list.Add(last);

                }
                else if (pictureBox1.Height - point_list[0].Y < point_list[point_list.Count - 1].Y)
                {
                    last.x = point_list[0].X;
```

```
                //y_max 为上端点的 y 坐标
                last.y_max = Math.Max( point_list[0].Y,    point_list[point_list.Count - 1].Y);
                //y_min 为下端点的 y 坐标
                last.y_min = Math.Min( point_list[0].Y,    point_list[point_list.Count - 1].Y);


                //delta_x
                last.delta_x    =    (double)(point_list[point_list.Count    -    1].X    -    point_list[0].X)    /
(double)( point_list[point_list.Count - 1].Y -    point_list[0].Y);

                //加入列表中
                Edge_list.Add(last);
            }

            //所有的边已经生成

            //这条边的第一个点开始向右，画与背景相反的颜色的线
            //遍历列表
            for (int i = 0; i < Edge_list.Count; i++)
            {
                int x, y;     //下端点
                x = Edge_list[i].x;
                y = Edge_list[i].y_min;

                Color color;
                Point point1 = new Point();
                point1.X = x;
                point1.Y = y;
                Point point2 = new Point();
                Point point12 = new Point();
        //        Point point22 = new Point();
                for (int j = 0; j < Edge_list[i].y_max - Edge_list[i].y_min; j++)
                {
                    int xget = point1.X;
                    int yget = point1.Y;
                    //获取右边点的颜色
                    Color currentColor = imageBox.GetPixel(xget+2, yget);
                    if (currentColor == Color.Black)
                    {
                        color = Color.White;
                    }
                    else
```

```
                {
                    color = Color.Black;
                }
                //颜色取反
                Pen mypen = new Pen(color, 2);
                point12.X = point1.X + 2;
                point12.Y = point1.Y ;
                point2.X = pbx;
                point2.Y = point1.Y;
                g.DrawLine(mypen, point12, point2);

                point1.Y = point1.Y + 1;
                point1.X = (int)Edge_list[i].delta_x + point1.X;

            }
        }


    }
```

4.3.3 种子填充

原理：假设在多边形区域内部右一象素已知，由此出发找到区域内的所有象素。

边界表示的 4 连通区域的递归填充算法:

```
void BoundaryFill4(int x,int y,int boundarycolor,int newcolor)
{   int color;
    if(color!=newcolor && color!=boundarycolor)
    {   drawpixel(x,y,newcolor);
        BoundaryFill4 (x,y+1, boundarycolor,newcolor);
        BoundaryFill4 (x,y-1, boundarycolor,newcolor);
        BoundaryFill4 (x-1,y, boundarycolor,newcolor);
        BoundaryFill4 (x+1,y, boundarycolor,newcolor);
    }
}
```

4.4 裁剪

Cohen-SutherLand 算法：

基本思想：对每条直线段 p1(x1,y1)p2(x2,y2)分三种情况处理：

(1) 直线段完全可见，“简取”之。

(2) 直线段完全不可见，“简弃”之。

(3) 直线段既不满足“简取”的条件，也不满足“简弃”的条件，需要对直线段按交点进行分段，分段后重复上述处理。

裁剪一条线段时，先求出端点 p1 和 p2 的编码 code1 和 code2，然后：

(1)若 code1|code2=0，对直线段应简取之。

(2)若 code1&code2≠0，对直线段可简弃之。

(3)若上述两条件均不成立。则需求出直线段与窗口边界的交点。在交点处把线段一分为二，其中必有一段完全在窗口外，可以弃之。再对另一段重复进行上述处理，直到该线段完全被舍弃或者找到位于窗口内的一段线段为止。

4.5 图形变换

利用二维变换矩阵实现图形的平移，缩放，旋转等变换

$T_{2D}=\begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}$，其中$\begin{bmatrix} a & d \\ b & e \end{bmatrix}$用于缩放，旋转，对称，错切等；$[c \quad f]$用于平移图形；[g] 用于在 x 轴的 1/g 处产生灭点；[h] 用于在 y 轴的 1/h 处产生灭点；[i] 用于对图形整体作伸缩变换。

4.5.1 平移

平移是指将 p 点沿直线路径从一个坐标位置移到另一个坐标位置的重定位过程。

$$[\ x^* \quad y^* \ 1\ ]=[\ x \quad y \quad 1\ ]\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{bmatrix}=[\ x+T_x \quad y+T_y \quad 1\ ]$$

4.5.2 缩放

比例变换是指对 p 点相对于坐标原点沿 x 方向放缩 Sx 倍，沿 y 方向放缩 Sy 倍。其中 Sx 和 Sy 称为比例系数。

$$[\ x^* \quad y^* \ 1\ ]=[\ x \quad y \quad 1\ ]\begin{bmatrix} S_x & 0 & 0 \\ 0 & S_x & 0 \\ 0 & 0 & 1 \end{bmatrix}=[\ xS_x \quad yS_y \quad 1\ ]$$

(1) Sx＝Sy＝1 时　　图形不变

(2) Sx＝Sy＞1 时　　图形沿两轴方向等比例放大

(3) Sx＝Sy＜1 时　　图形沿两轴方向等比例缩小

(4) Sx＜＞Sy 时　　图形沿两轴方向作非均匀比例变换

4.5.3 旋转

二维旋转是指将 p 点绕坐标原点转动某个角度（逆时针为正，顺时针为负）得到新的点 p’的重定位过程。

逆时针：

$$[\ x^* \quad y^* \ 1\ ]=[\ x \quad y \quad 1\ ]\begin{bmatrix} cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}=[\ x\cos\theta-y\sin\theta \quad x\sin\theta+y\cos\theta \quad 1\ ]$$

顺时针：

$$[\ x^* \quad y^* \ 1\ ]=[\ x \quad y \quad 1\ ]\begin{bmatrix} cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}=[\ x\cos\theta+y\sin\theta \quad -x\sin\theta+y\cos\theta \quad 1\ ]$$

4.5.4 对称

$$[\ x^* \quad y^* \ 1\ ]=[\ x \quad y \quad 1\ ]\begin{bmatrix} a & d & 0 \\ b & e & 0 \\ 0 & 0 & 1 \end{bmatrix}=[\ ax+by \quad dx+ey \quad 1\ ]$$

(1) b＝d＝0, a＝-1, e＝ 1 时　　x*＝ -x, y*＝ y,　以 y 轴对称

(2) b＝d＝0, a＝1, e＝ -1 时　　x*＝ x, y*＝ -y,　以 x 轴对称
(3) b＝d＝0, a＝ e＝ -1 时　　x*＝ -x, y*＝ -y, 以原点对称
(4) b＝d＝1,　a＝e＝ 0 时　　x*＝ y, y*＝ x, 以 y = x 直线对称
(5) b＝d＝-1, a＝ e＝ 0 时　　x*＝ -y, y*＝ -x, 以 y = -x 直线对称

4.6 选中

保存每个图元的所在的矩形，判断鼠标是否在矩形内，如果在矩形内变换鼠标样式，点击图元，显示热点。

4.7 组合与打散

鼠标拖动选择多个图元，判断在鼠标框选范围内的图元并显示热点。

4.8 保存

1) 保存当前画布的 json 内容，保存在 local storage 中；
2) 将当前画布导出，保存为图片。

4.9 撤销与恢复

将每个图元以 json 格式保存在画布栈__canvas 中，设置画布元素栈__canvasStack 和历史纪录栈__hisoryStack，保存当前画布的 json。

当需要撤销的时候，将“画布元素栈”栈顶元素出栈，并压入“历史记录栈”中，将“画布元素栈”新的栈顶元素重新加载（如果什么都没有就化为白板）；

当需要恢复的时候，将“历史纪录栈”栈顶元素出栈，并压入“画布元素栈”中，将“画布元素栈”新的栈顶元素重新加载（这个不会发生什么都没有的情况）。

第五章 结语

由于计算机图形学设备的不断更新和图形软件功能的不断扩充，也由于计算机硬件功能的不断增强和系统软件的不断完善，计算机图形学的应用极大地提高了人们理解数据、观察或想象图形的能力。[3]通过 html 实现计算机图形学中基本图形的扫描转换和图形变换的算法，更加深刻地理解了算法地原理与含义，优化了目前画图软件的功能。

参考文献

[1]陈敏雅, 金旭东.浅谈计算机图形学与图形图像处理技术[J].长春理工大学学报, 2011, 6 (01) :138-139+146.

[2]朱文.基于 HTML5 Canvas 技术的在线图像处理方法的研究[D].广州:华南理工大学, 2013.

[3]张浩.简析计算机图形学中数据结构的应用[J].南方农机, 2018, 49 (5) :134.