《计算机图形学》系统报告

学号：171860578 姓名：张梓悦 邮箱：2846728884@qq.com

1. 综述

在3月，我先对图形学相关算法进行了学习，并仔细研究了助教所提供的框架代码。最终完成了直线绘制、多边形绘制、椭圆绘制以及旋转、缩放、平移、裁剪的功能。并在Cli程序中添加了相应操作。

在4月，我继续补充了算法部分的代码，添加了曲线绘制算法，并测试了每一个指令的正确性，修改了一些代码中存在的bug，完成了Algorithm以及Cli中所有需要添加的功能。并开始进行qt的学习，为gui部分的书写做好准备。

在5月，充分了解qt的功能原理后开始对gui部分进行了书写，完成了直线绘制、曲线绘制、多边形绘制、椭圆绘制以及旋转、缩放、平移、裁剪对应的gui操作，同时也为gui添加了重置画布、保存画布的功能，并为gui设计了更为人性化的操作方式。已在5月份完成了助教所要求的全部基础功能，

在6月，对gui做出了更加人性化的设计，比如重置画布时的对话框，以及曲线控制点数的设置。同时添加了一些其他的拓展功能。其中拓展功能包括了拓展的算法：多边形填充以及多边形裁剪算法，较高级的交互方式：直接通过鼠标点击选择图元，较高级的程序功能：撤销、复制、粘贴

2. 算法介绍

2.1 直线的绘制算法

2.1.1 DDA算法

算法原理：

DDA算法是一种利用计算两个坐标方向差分确定线段每个像素点的线段扫描转换算法。下面结合我编写的该算法python代码来做具体的解释。

定义线段两端点坐标分别为(x0, y0), (x1, y1)。用直线方程$y = mx + b$来表示绘制线段所在直线。先考虑两端点具有正斜率，且是从左端点到右端点进行处理的线段。假设$m \leqslant 2$，则在x的单位间隔内（单位间隔为1，即一个个像素点的移动），每个点取样并计算y的值。

初始时，$x_k = x0$， $y_k = y0$。由于$y_k = mx_k + b$，且$y_{k+1} = mx_{k+1} + b$。

由于$x_{k+1} = x_k + 1$，故$y_{k+1} = mx_{k+1} + b = mx_k + m + 1$，因此$y_{k+1} = y_k + m$

如果考虑到起始点横坐标x0大于终止点横坐标x1，则让x和y的递增量1和m分别取负即可（即改为-1和-m）。

若 m>1，则将 x 和 y 的规则交换。即在单位 y 间隔(△y=1)取样，并计算每个连续的 x 值。因此有$x_{k+1} = x_k + (1/m)$

如果考虑到起始点横坐标y0大于终止点横坐标y1，则让x和y的递增量1/m和1分别取负即可（即改为-1/m和-1）。确定好x和y的递增量，即可循环绘制直线上的每个像素点。

综上可知$y = kx + b, 0 < k < 1$时，x每增加1，y增加k，可令$\Delta x = 1, \Delta y = k$。

则可以从x的起始点x_0每次增加Δx直到x_1，同时每次对y也增加Δy，

对于小数情况进行四舍五入即可，最终能够得到直线$y = kx + b$。其他情况同理。

算法实现：

```
    elif algorithm == 'DDA':
        if(abs(x1-x0)>=abs(y1-y0)):
            length=abs(x1-x0)
        else:
            length=abs(y1-y0)
        if (length==0):
            result.append((x0,y0))
            return result
        dx=(float)(x1-x0)/length
        dy=(float)(y1-y0)/length
        i=1
        x=x0
        y=y0
```

```
        while(i<=length):
            result.append((int(x+0.5),int(y+0.5)))
            x=x+dx
            y=y+dy
            i+=1
```

算法性能：

算法很好，但误差大致线性递增（每次累加的结果有常数级别的误差）。画布上并未见到明显误差。

2.1.2 Bresenham算法

算法原理：

DDA算法在每次增量操作时，x和y作为float值，每次增加一个增量Δx或者Δy，随着递增次数的增多，由于计算机表示浮点数的精确能力有限，x和y的值会逐渐偏离精确的x和y的值。因此，我们需要一个误差不会递增且很小的画线算法，即Bresenham算法。

首先，Bresenham算法只需要考虑直线斜率在[0, 1]内的情况。因为，若直线斜率在(1, +∞)内，则x关于y的直线斜率就在(0, 1]内，通过x和y的坐标交换，又可以正确地画出直线。若直线斜率小于0，则对增量取负即可。因此，只需展开对斜率在[0, 1]内情况的讨论。

Bresenham算法引入整型决策参数p_k决定下一点坐标，而非像DDA那样的浮点累加。如图所示，当直线斜率在0到1之间时，已经计算出像素点(x_k, y_k)，需要画出直线的下一个点(x_{k+1}, y_{k+1})，其可能等于(x_k, y_k)，也可能等于$(x_k, y_k + 1)$。用y_a表示下一像素点坐标的精确纵坐标值，由直线方程$y = mx + b$可得，$y_a = mx_{k+1} + b$。此时，$x_{k+1} = x_k + 1$，因此

$$y_a = m(x_k + 1) + b$$

其中y_{k+1}表示第k+1步应选点的纵坐标。所以，两个候选像素与线段数学路径的垂直偏移分别为：

$$\begin{cases} d1 = y_a - y_k = m(x_k + 1) + b - y_k \\ d2 = y_k + 1 - y_a = y_k + 1 - m(x_k + 1) - b \end{cases}$$

显然，y_{k+1}应该选择d1和d2相对较小者所对应的y_{k+1}。对d1和d2做差，并由其构造p_k，使得：

$$p_k = \Delta x(d1 - d2) = 2\Delta y x_k - 2\Delta x y_k + c$$

通过进一步计算我们可以知道，$p_{k+1} - p_k = 2\Delta y(x_{k+1} - x_k) - 2\Delta x(y_{k+1} - y_k)$，故可得：

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x(y_{k+1} - y_k)$$

根据p_k，我们可以知道第k步选择了什么像素，故可以推知$y_{k+1} - y_k$是1还是0。由此，即可获得p_{k+1}的值：

$$p_{k+1} = \begin{cases} p_k + 2\Delta y - 2\Delta x, & p_k > 0 \\ p_k + 2\Delta y, & p_k < 0 \end{cases}$$

结合上述原理，确定为以下几步：

1.画初始点

2.通过坐标变换将所有情况化为斜率在[0,1]内情况

3.确定p_k初值}

4.进入循环，每次根据p_k选择下一点，并更新p_k为p_{k+1}

5.循环结束

算法实现：

```
    elif algorithm == 'Bresenham':
        dx=abs(x1-x0)
        dy=abs(y1-y0)
        if(dx==0 and dy==0):
            result.append((x0,y0))
            return result
        gradient_flag=0
        if(dx<dy):
            gradient_flag=1
        if(gradient_flag==1):
            x0,y0=y0,x0
            x1,y1=y1,x1
            dx,dy=dy,dx
        xx=1
        if(x1-x0<0):
            xx=-1
        yy=1
        if(y1-y0<0):
            yy=-1
        p=2*dy-dx
        x=x0
        y=y0
        result.append((x,y))
        while(x!=x1):
            if(p>=0):
                p+=2*dy-2*dx
                y+=yy
            else:
                p+=2*dy
            x+=xx
            if(gradient_flag):
                result.append((y,x))
            else:
                result.append((x,y))
```

算法性能：

误差很小，速度很快。相对于DDA 算法有常数级别且不会递增的误差。

2.2 椭圆的绘制算法

2.2.1 中点椭圆生成算法

算法原理：

本部分介绍中点椭圆绘制算法，其思想实际上是Bresenham算法的通用思想。个人理解的该思想过程如下：

1.确定决策函数pk初始值

2.确定决策函数pk在不同情况下的增量

3.画出第一个点

4.循环，更新决策函数并绘制下一个点，直至画到终点

其中，决策函数表征的是$y_{k+1} or\ y_k+1$与y_k的距离差。

由于椭圆具有轴对称、中心对称的特征，因此我们可以画出椭圆的右上部分，然后根据对称性画出其余各部分。为了方便表述，令椭圆的圆心坐标为(x_c,y_c)，椭圆平行于$x_{轴}$所在直线的半径为r_x，平行于$y_{轴}$所在直线的半径为r_y，则假设画出了右上角的某个点$(x_c+x_0,y_c+y_0)(x_0,y_0\geqslant 0)$，则同时需要画出点$(x_c+x_0,y_c+y_0),(x_c-x_0,y_c+y_0),(x_c+x_0,y_c-y_0),(x_c-x_0,y_c-y_0)$。

正如中点算法画直线的思想，我们也需要分切线斜率的情况。选择初始绘制点为椭圆最上方顶点(x_0,y_c+y_0)，则从其开始的一段，椭圆切线斜率恒在[-1,0]内。从椭圆方程可以求出，椭圆切线斜率为：

$$\frac{dy}{dx}=-2r_y^2x/r_x^2y$$

所以当椭圆斜率等于-1时，上式左值等于-1，可得：

$$r_y^2x=r_x^2y$$

这个等式就是分界条件，当上式左小于右时，处理的是切线斜率在[-1,0]的情况；左开始大于右时，处理切线斜率在$(-\infty,1)$的情况。

接下来考虑决策参数。决策参数也要根据区间分m为两种。记切线斜率在[-1,0]时决策参数为$p1_k$，在$(-\infty,-1)$时的决策参数为$p2_k$。先讨论$p1_k$，假设在前一步中选择了位置(x_k,y_k)，将第一象限内取样位置x_{k+1}处两个候选像素间中点对决策参数求得：

$$p1_k=r_y^2(x_k+1)^2+r_x^2\left(y_k-\frac{1}{2}\right)^2-r_x^2r_y^2$$

根据该公式，可以算出$p1_{k+1}$和$p1_k$的差得$p1_k$的增量：

$$p1_{k+1}=\begin{cases}p1_k+2r_y^2x_k+3r_y^2, & p_k<0\\ p1_k+2r_y^2x_k-2r_x^2y_k+2r_x^2+3r_y^2. & p_k\geqslant 0\end{cases}$$

同样，我们也可以推知，当过分界点之后的决策参数$p2_k$的初始值为和其增量，用坐标(x_{k-1},y_{k-1})表示分界点之前绘制的最后一个点坐标：

$$p2_k=\begin{cases}p2_k+2r_y^2x_k-2r_x^2y_k+2r_y^2+3r_x^2, & p_k\leqslant 0\\ p2_k-2r_x^2y_k+3r_x^2. & p_k>0\end{cases}$$

算法实现：

```
    x0, y0 = p_list[0]
    x1, y1 = p_list[1]
    result = []
    xx=int((x0+x1)/2)
    yy=int((y0+y1)/2)
    a=int((abs(x1-x0))/2)
    b=int((abs(y1-y0))/2)
    p=float(b**2+a**2*(0.25-b))
    x=0
    y=b
    result.append((xx+x,yy+y))
    result.append((xx-x,yy+y))
    result.append((xx+x,yy-y))
    result.append((xx-x,yy-y))
    while(b**2*x<a**2*y):
        if(p<0):
            p+=float(b**2*(2*x+3))
        else:
```

```
            p+=float(b**2*(2*x+3)-a**2*(2*y-2))
            y-=1
        x+=1
        result.append((xx+x,yy+y))
        result.append((xx-x,yy+y))
        result.append((xx+x,yy-y))
        result.append((xx-x,yy-y))
    p=float((b*(x+0.5))**2+(a*(y-1))**2-(a*b)**2)
    while(y>0):
        if(p<0):
            p+=float(b**2*(2*x+2)+a**2*(-2*y+3))
            x+=1
        else:
            p+=float(a**2*(-2*y+3))
        y-=1
        result.append((xx+x,yy+y))
        result.append((xx-x,yy+y))
        result.append((xx+x,yy-y))
        result.append((xx-x,yy-y))
    return result
```

2.3 多边形的绘制算法

算法原理：

多边形的绘制即多个点之间线段的连续绘制。每一段线段根据用户选择使用DDA算法或Bresenham算法即可。

算法实现：

```
    result = []
    for i in range(1,len(p_list)):
        line = draw_line([p_list[i - 1], p_list[i]], algorithm)
        result += line
    return result
```

2.4 曲线的绘制算法

2.4.1 Bezier算法

算法原理：

Bezier曲线是由多边形控制，逼近多边形的一种曲线。其做法是通过Bezier基函数来得到绘制点的坐标。Bezier基函数的原型比较复杂，但在我了解到Bezier曲线基函数已被证明可简化为伯恩斯坦基函数，形式为：

$$C_n^i t^i (1-t)^i$$

有了基函数，自然可以用参数t，取某一个步长，来近似绘制曲线。但本实验中采用的是用多边形去逼近Bezier曲线的算法。下面以4个点控制的曲线为例介绍算法执行步骤，其中算法接受参数为这4个点有序列（其中求中点是因为对u取$\frac{1}{2}$）：

1. 对两两相邻点求中点，得到三个中点
2. 对得到的三个中点求中点，得到两个中点

3. 对得到的两个中点求中点，得到一个中点
4. 将得到的最接近曲线的7个点根据最后那个中点对半分为4个4个。即

$P_0^0, P_0^1, P_0^2, P_0^3$以及$P_0^3, P_1^2, P_2^1, P_3^0$

5. 分别对这两组4个点调用该算法，递归到误差在可接受范围内为止

算法实现：

```
def Bezier_point(n, t, control_point):
    while(n!=1):
        for i in range(0, n-1):
            x0,y0=control_point[i]
            x1,y1=control_point[i+1]
            x=float(x0*(1-t))+float(x1*t)
            y=float(y0*(1-t))+float(y1*t)
            control_point[i]=x,y
        n-=1
    return control_point[0]

    result = []
    control_point = []
    if algorithm == 'Bezier':
        m=len(p_list)*1000
        for i in range(0, m):
            control_point=p_list.copy()
            t = float(i/m)
            x,y=Bezier_point(len(p_list), t, control_point)
            result.append((int(x+0.5),int(y+0.5)))
```

算法性能：

此算法效率会随着控制点数的增加降低运算的效率

2.4.2 B-spline算法

算法原理：

由于Bezier曲线调整的时候有“牵一发而动全身”的缺点，因此引入B样条曲线。B样条曲线也有对应的基函数，但为了保证变换一个控制点的时候只会影响到曲线的一小部分，引入了基函数的**支撑区间**的概念，即称函数值不为0的区间。比如三次Bezier曲线的四个Bezier曲线基函数的支撑区间都是[0,1]，所以对全局曲线的点都有影响。

所以，为了让B样条曲线有局部调整的优良特性，就要让每个基函数都只具有局部的支撑区间。

B样条基函数的产生使用递推形式，即de-BoorCox递推定义：

$$\begin{cases} N_{i,1}(u) = \begin{cases} 1, & u_i < u < u_{i+1} \\ 0, & o.t. \end{cases} \\ N_{i,k}(u) = \frac{u-u_i}{u_{i+k-1}-u_i} N_{i,k-1}(u) + \frac{u_{i+k}-u}{u_{i+k}-u_{i+1}} N_{i+1,k-1}(u) \end{cases}$$

其中的k是阶数不是次数，u是参数。

当节点向量为0,1,2,3.....n+k时，是均匀B样条基函数。当向量前k个数和后k个数都相等，中间均匀分布时，是准均匀B样条基函数。由于实验要求是均匀B样条基函数，为了优化性能，只认为节点向量是0,1,.....,n+k，代码可以写成如下形式：

```
def deboox_cox(i, k, u):
    if k == 1:
        if i<=u and u<i+1:
            return 1
        else:
            return 0
    else:
        return (u-i)/(k-1)*deboox_cox(i,k-1,u)+(i+k-u)/(k-1)*deboox_cox(i+1,k-
1,u)
```

绘制曲线时，就用每个点乘以基函数计算可得，最后将这些点相邻两点连线。

算法实现：

```
k = 3
        n=len(p_list)
        if(n<4):
            return result
        du=float(1/1000)
        u =float(k)

        while(u<n):
            x1,y1 = 0,0
            for i in range(0,n):
                x0,y0 = p_list[i]
                res=deboox_cox(i, k+1, u)
                x1 +=x0*res
                y1 +=y0*res
            result.append([round(x1), round(y1)])
            u+=du
```

算法性能：

此算法运行复杂度随着参数每次递增的du的增加而降低，即du越小，耗时越高。但测试时，UI操作十分连续，并未出现明显卡顿。

2.5 图元平移算法

算法原理：

不妨设dx为水平方向平移量，dy为垂直方向平移量。

x,y为原始坐标，x',y'为平移后坐标，则：

$$\begin{cases} x' = x + dx \\ y' = y + dy \end{cases}$$

算法实现：

```
def translate(p_list, dx, dy):
    """平移变换

    :param p_list: (list of list of int: [[x0, y0], [x1, y1], [x2, y2], ...]) 图元参数
    :param dx: (int) 水平方向平移量
    :param dy: (int) 垂直方向平移量
    :return: (list of list of int: [[x_0, y_0], [x_1, y_1], [x_2, y_2], ...]) 变换后的图元参数
    """
    result = []
    for x, y in p_list:
        result.append((x+dx,y+dy))
    return result
```

2.6 图元旋转算法

算法原理：

不妨设x_r,y_r为旋转中心点坐标，θ为顺时针旋转角度。

x,y为原始坐标，x',y'为旋转后坐标，则：

$$\begin{cases} x' = x_r + (x - x_r)\cos\theta - (y - y_r)\sin\theta \\ y' = y_r + (x - x_r)\sin\theta + (y - y_r)\cos\theta \end{cases}$$

算法实现：

```
def rotate(p_list, x, y, r):
    """旋转变换（除椭圆外）

    :param p_list: (list of list of int: [[x0, y0], [x1, y1], [x2, y2], ...]) 图元参数
    :param x: (int) 旋转中心x坐标
    :param y: (int) 旋转中心y坐标
    :param r: (int) 顺时针旋转角度（°）
    :return: (list of list of int: [[x_0, y_0], [x_1, y_1], [x_2, y_2], ...]) 变换后的图元参数
    """
    angel=float(r*math.pi/180)
    cos=math.cos(angel)
    sin=-math.sin(angel)
    result = []
    for x0, y0 in p_list:
        x1=int(float(x)+float((x0-x)*cos)-float((y0-y)*sin)+0.5)
        y1=int(float(y)+float((x0-x)*sin)+float((y0-y)*cos)+0.5)
        result.append((x1,y1))
    return result
```

2.7 图元缩放算法

算法原理：

不妨设x_f,y_f为缩放中心点坐标，s为缩放倍数。

x,y为原始坐标，x',y'为缩放后坐标，则：

$$\begin{cases} x' = x \cdot s + x_f \cdot (1-s) \\ y' = y \cdot s + y_f \cdot (1-s) \end{cases}$$

算法实现：

```
def scale(p_list, x, y, s):
    """缩放变换

    :param p_list: (list of list of int: [[x0, y0], [x1, y1], [x2, y2], ...]) 图元参数
    :param x: (int) 缩放中心x坐标
    :param y: (int) 缩放中心y坐标
    :param s: (float) 缩放倍数
    :return: (list of list of int: [[x_0, y_0], [x_1, y_1], [x_2, y_2], ...]) 变换后的图元参数
    """
    result = []
    for x0, y0 in p_list:
        x1=int(float(x0*s)+float(x*(1-s))+0.5)
        y1=int(float(y0*s)+float(y*(1-s))+0.5)
        result.append((x1,y1))
    return result
```

2.8 线段裁剪算法

2.8.1 Cohen-Sutherland算法

算法原理：

Cohen-Sutherland线段裁剪算法是一种依靠编码的线段裁剪算法。依我的理解，其实也就是用编码的分类来映射到所有裁剪情况的分类。算法本身比较简单易于理解，也相对高效。该算法先将平面由裁剪框分成9个部分，对于任一端点(x,y)，根据其坐标所在的区域，

赋予一个4位的二进制码，判断图形元素是否落在裁剪窗口之内，如果有一部分在窗口之外再通过求交运算找出其位于内部的部分。

若$x < x_{min}$，对应code+1

若$x > x_{max}$，对应code+2

若$y < y_{min}$，对应code+4

若$y > y_{max}$，对应code+8

根据观察可以得出，线段两端点同在裁剪窗口上方，或同在下方，或同在左方，或同在右方时，线段一定要被舍弃；线段两端点同在裁剪窗口内，线段一定被保留；其余情况则有可能被舍弃或被裁剪。结合编码的特征可得，两端点编码按位与若不为零，则对应前一种情况，应该被舍弃；若均为0000，即按位或为0，则在裁剪窗口内，应该被保留。其他编码情况时，则需要根据窗口的边界所在直线，在循环中一点点对线段进行缩短。

记线段两端点为p1和p2，每次循环缩短前，若p1编码为0，则p2编码一定不为0，此时交换p1和p2两点。每次缩短后对编码进行判断，一旦满足第一种情况则立即裁剪，满足第二种情况时则立即舍弃，并退出循环。此算法还需额外注意平行于任意一条窗口边界的线段，做单独处理。

裁剪一条线段时，根据以上步骤，先求出端点a和b的编码code1和code2，接下来进行如下算法：

1. 如果code1和code2按位或的结果为0，则说明a和b均在窗口内，即线段完全位于窗口之内，直接返回原来的端点值即可。
2. 如果code1和code2按位与的结果不等于0。说明a和b同时在窗口的上、下、左或右方，即线段完全位于窗口的外部，返回空或者一对相同的端点即可。
3. 求出线段与窗口边界的交点，并用该交点的坐标值替换端点的坐标值。即在该交点处将线段分为两部分。
4. 再次计算code1和code2的值，如果code1和code2按位或的结果不为0，则裁剪失败，返回空或者一对相同的端点。
5. 保存修改后的端点坐标
6. 算法结束。

算法实现：

```
if algorithm == 'Cohen-Sutherland':
        x0,y0 = p_list[0]
        x1,y1 = p_list[1]
        code1=0
        code2=0
        if(x0<x_min):
            code1|=1
        if(x0>x_max):
            code1|=2
        if(y0<y_min):
            code1|=4
        if(y0>y_max):
            code1|=8
        if(x1<x_min):
            code2|=1
        if(x1>x_max):
            code2|=2
        if(y1<y_min):
            code2|=4
        if(y1>y_max):
            code2|=8

        if((code1|code2)==0):
            result=p_list
        elif((code1&code2)!=0):
            result=[[0,0],[0,0]]
        else:
            if(x0==x1):
                if (min(y0, y1) > y_max or max(y0, y1) < y_min):
                    result=[[0,0],[0,0]]
                    return result
                if (y1 >= y0):
                    if (y1 > y_max):
                        y1=y_max
                    if (y0 < y_min):
                        y0=y_min
                elif (y0 > y1):
                    if (y0 > y_max):
                        y0=y_max
                    if (y1 < y_min):
                        y1=y_min
                result=[[x0,y0], [x1,y1]]
```

```
                return result
            if(y0==y1):
                if (min(x0, x1) > x_max or max(x0, x1) < x_min):
                    result=[[0,0],[0,0]]
                    return result
                if (x1 >= x0):
                    if (x1 > x_max):
                        x1=x_max
                    if (x0 < x_min):
                        x0=x_min
                elif (x0 > x1):
                    if (x0 > x_max):
                        x0=x_max
                    if (x1 < x_min):
                        x1=x_min
                result=[[x0,y0], [x1,y1]]
                return result

            code=code1|code2
            if(code&1):
                yy=int(float((x_min-x1)*(y0-y1)/(x0-x1))+float(y1)+0.5)
                if(x0<x_min):
                    x0=x_min
                    y0=yy
                elif(x1<x_min):
                    x1=x_min
                    y1=yy
            if(code&2):
                yy=int(float((x_max-x1)*(y0-y1)/(x0-x1))+float(y1)+0.5)
                if(x0>x_max):
                    x0=x_max
                    y0=yy
                elif(x1>x_max):
                    x1=x_max
                    y1=yy
            if(code&4):
                xx=int(float((y_min-y1)*(x0-x1)/(y0-y1))+float(x1)+0.5)
                if(y0<y_min):
                    x0=xx
                    y0=y_min
                elif(y1<y_min):
                    x1=xx
                    y1=y_min
            if(code&8):
                xx=int(float((y_max-y1)*(x0-x1)/(y0-y1))+float(x1)+0.5)
                if(y0>y_max):
                    x0=xx
                    y0=y_max
                elif(y1>y_max):
                    x1=xx
                    y1=y_max
            if(x0 <= x_max and x0 >= x_min and x1 <= x_max and x1 >= x_min and
y0 <= y_max and y0 >= y_min and y1 <= y_max and y1 >= y_min):
                result=[[x0,y0], [x1,y1]]
            else:
                result=[[0,0],[0,0]]
```

2.8.2 Liang-Barsky算法

算法原理：

梁友栋-Barsky裁剪算法的核心是将线段所在的直线看成是有向的。假设线段两端点分别为p1和p2，则该直线的方向则被认为是p1->p2。这个带有方向的直线一定是从一个很远的地方接近裁剪窗口，然后再逐渐远离裁剪窗口。

考察直线的参数方程。设p1坐标为(x1, y1)，p2坐标为(x2, y2)，则该直线的参数方程为：

$$\begin{cases} x &= x1 + u(x2 - x1), \\ y &= y1 + u(y2 - y1). \end{cases}$$

其中，0<=u<=1时，表示的是线段；而$-\infty < u < +\infty$时，表示的是一整条直线。

假设裁剪线段不平行于裁剪窗口任意一条边，裁剪窗口四条边所在的四条直线是一定会被裁剪线段所在直线穿过的。由于我们给直线定了方向，所以穿过分入边和出边，入边就是最先触碰到的裁剪窗口的两条直线，出边就是远离裁剪窗口穿过的两条直线。经过观察可以确定，裁剪线段所在直线与两条入边各有一个交点，假设分别对应参数值u1, u2；裁剪线段所在直线与两条出边也各有一个交点，假设分别对应两个参数值u3, u4。则只需要选出u1、u2和0三者的最大值记为t_1，并选出u3、u4和1三者的最大值记为t_2，比较t_1和t_2，若$t_1 > t_2$，则表示线段在裁剪窗口外应被舍弃；若$t_1 < t_2$，则以t_1、t_2两者为参数的两点就是被裁剪后线段的起点和终点。

因此，需要解决的问题就是哪两天半是入边，哪两条边是出边。说判断方法之前，需要了解到直线上点在裁剪窗口内的条件。假定裁剪窗口左、右边界x值分别为x_{min}和x_{max}，裁剪窗口下、上边界y值分别为y_{min}和y_{max}，则直线上的点落在窗口内，条件是：

$$\begin{cases} xmin < x1 + u(x2 - x1) < xmax, \\ ymin < y1 + u(y2 - y1) < ymax. \end{cases}$$

可以转化为书上提到的四个用"≤"表达的不等式，也就是$u_k * p_k <= q_k, u = 1, 2, 3, 4$的形式。当各不等式取等时，则$p_k$小于0时，$u_k$表达的是入边与直线交点的参数值；$p_k$大于0时，$u_k$表达的是出边与直线交点的参数值，出边和入边就是这样确定的。

综上该算法的大概流程如下：

先计算p数组：p=$[x0 - x1, x1 - x0, y0 - y1, y1 - y0]$

再计算q数组：q=$[x0 - x\_min, x\_max - x0, y0 - y\_min, y\_max - y0]$

接下来可分情况讨论，考虑p和q数组中每一对p、q：

1. 若$p = 0, q < 0$，表明直线与裁剪框平行，但是在裁剪框外面。因此需要直接舍弃该直线，即返回空或者一对相同的端点。
2. 若$p = 0, q \geq 0$，表明直线与裁剪框平行，且其延长线必然经过裁剪框的内部，接下来需要计算该线是在框的内部、外部还是与其相交
3. 若$p < 0$，表明直线从裁剪边界的外部延伸到内部
4. 若$p > 0$，表明直线从裁剪边界的内部延伸到外部
5. 对于(3)和(4)的情况，找到直线与边界的交点，并计算出在该窗口内线段对应的参数u_1以及u_2。u_1由使直线是从外部延伸到窗口内部的边界决定。u_2由使直线是从内部延伸到窗口外部的边界决定。 先计算$r = q/p$，则$u_1 = max(r, 0)$，$u_2 = min(r, 1)$，如果$u_1 > u_2$，则这条直线完全在窗口外面，需要舍弃。 否则，可根据u_1以及u_2这两个值，计算出裁剪后的线段的端点，最后返回这两个新端点即可。

算法实现：

```
elif algorithm == 'Liang-Barsky':
        x0,y0 = p_list[0]
        x1,y1 = p_list[1]
```

```
        p=[x0-x1,x1-x0,y0-y1,y1-y0]
        q=[x0-x_min,x_max-x0,y0-y_min,y_max-y0]
        u1=float(0)
        u2=float(1)
        flag=False
        for i in range(0,4):
            if(p[i]==0 and q[i]<0):
                flag=True
            else:
                if(p[i]==0):
                    continue
                r=float(q[i]/p[i])
                if(p[i]<0):
                    u1=max(u1,r)
                else:
                    u2=min(u2,r)
                if(u1>u2):
                    flag=True
        if(flag==False):
            xx0=int(x0+u1*(x1-x0)+0.5)
            yy0=int(y0+u1*(y1-y0)+0.5)
            xx1=int(x0+u2*(x1-x0)+0.5)
            yy1=int(y0+u2*(y1-y0)+0.5)
            result=[[xx0,yy0],[xx1,yy1]]
        else:
            result=[[0,0],[0,0]]
```

2.9 拓展算法

2.9.1 多边形裁剪Sutherland-Hodgeman算法

算法原理：

1. Sutherland-Hodgeman算法是一个相对简单且易于理解的算法，它算法的核心思路就是让裁剪窗口的四个边界分别切这个多边形一刀，切完四刀，裁剪完成。过程中，切完一刀的输出多边形作为准备切下一刀的输入多边形。

   以窗口**左边界**切割来叙述一下这个算法的细节。首先置输出多边形为空，按某一时针顺序遍历多边形的每条边P_iP_{i+1}：

   1. 如果这条边从左到右穿过左边界(P_i在左侧而P_{i+1}在边界上或右侧)，则输出多边形加入这条边与左边界交点和P_{i+1}；
   2. 如果这条边两顶点都在边界上或右侧，输出多边形加入P_{i+1}
   3. 如果这条边从右到左穿过左边界，则输出多边形加入这条边与左边界的交点。

   其他边切割以此类推。故综上，代码如下，

算法实现：

```
def polygon_clip(p_list, x_min, y_min, x_max, y_max):

    result = []
    if(x_min==x_max or y_min==y_max):
        result=[[0,0],[0,0]]
        return result

    if x_min > x_max:
        x_min, x_max = x_max, x_min
```

```
    if y_min > y_max:
        y_min, y_max = y_max, y_min


    tmp = p_list
    n = len(tmp)
    for j in range(0,len(tmp)):
        if (tmp[j][0] < x_min and tmp[(j + 1) % n][0] >= x_min):
            y = (float)(tmp[(j+1)%n][1] - tmp[j][1])/(tmp[(j+1)%n][0]-tmp[j]
[0])*(x_min-tmp[j][0])+tmp[j][1]
            y=int(y)
            result.append([x_min, y])
            result.append(tmp[(j + 1)%n])
        elif (tmp[j][0] >= x_min and tmp[(j+1)%n][0] >= x_min):
            result.append(tmp[(j+1)%n])
        elif (tmp[j][0] >= x_min and tmp[(j + 1)%n][0] < x_min):
            y = (float)(tmp[(j+1)%n][1] - tmp[j][1])/(tmp[(j+1)%n][0]-tmp[j]
[0])*(x_min-tmp[j][0])+tmp[j][1]
            y=int(y)
            result.append([x_min, y])

    tmp=result
    n = len(tmp)
    result=[]
    for j in range(0,len(tmp)):
        if (tmp[j][1] < y_min and tmp[(j + 1) % n][1] >= y_min):
            x = (float)(tmp[(j+1)%n][0] - tmp[j][0])/(tmp[(j+1)%n][1]-tmp[j]
[1])*(y_min-tmp[j][1])+tmp[j][0]
            x=int(x)
            result.append([x, y_min])
            result.append(tmp[(j + 1)%n])
        elif (tmp[j][1] >= y_min and tmp[(j+1)%n][1] >= y_min):
            result.append(tmp[(j+1)%n])
        elif (tmp[j][1] >= y_min and tmp[(j + 1)%n][1] < y_min):
            x = (float)(tmp[(j+1)%n][0] - tmp[j][0])/(tmp[(j+1)%n][1]-tmp[j]
[1])*(y_min-tmp[j][1])+tmp[j][0]
            x=int(x)
            result.append([x, y_min])

    tmp=result
    n = len(tmp)
    result=[]
    for j in range(0,len(tmp)):
        if (tmp[j][0] > x_max and tmp[(j + 1) % n][0] <= x_max):
            y = (float)(tmp[(j+1)%n][1] - tmp[j][1])/(tmp[(j+1)%n][0]-tmp[j]
[0])*(x_max-tmp[j][0])+tmp[j][1]
            y=int(y)
            result.append([x_max, y])
            result.append(tmp[(j + 1)%n])
        elif (tmp[j][0] <= x_max and tmp[(j+1)%n][0] <= x_max):
            result.append(tmp[(j+1)%n])
        elif (tmp[j][0] <= x_max and tmp[(j + 1)%n][0] > x_max):
            y = (float)(tmp[(j+1)%n][1] - tmp[j][1])/(tmp[(j+1)%n][0]-tmp[j]
[0])*(x_max-tmp[j][0])+tmp[j][1]
            y=int(y)
            result.append([x_max, y])

    tmp=result
```

```
    n = len(tmp)
    result=[]
    for j in range(0,len(tmp)):
        if (tmp[j][1] > y_max and tmp[(j + 1) % n][1] <= y_max):
            x = (float)(tmp[(j+1)%n][0] - tmp[j][0])/(tmp[(j+1)%n][1]-tmp[j]
[1])*(y_max-tmp[j][1])+tmp[j][0]
            x=int(x)
            result.append([x, y_max])
            result.append(tmp[(j + 1)%n])
        elif (tmp[j][1] <= y_max and tmp[(j+1)%n][1] <= y_max):
            result.append(tmp[(j+1)%n])
        elif (tmp[j][1] <= y_max and tmp[(j + 1)%n][1] > y_max):
            x = (float)(tmp[(j+1)%n][0] - tmp[j][0])/(tmp[(j+1)%n][1]-tmp[j]
[1])*(y_max-tmp[j][1])+tmp[j][0]
            x=int(x)
            result.append([x, y_max])

    if(len(result)==0):
        result=[[0,0],[0,0]]
    return result
```

2.9.2 多边形扫描填充算法

算法原理：

在扫描转换算法被提出之前，也曾有人提出过多边形填充算法，即对每个像素点判断其在不在某个多边形内，如果在则填充该像素点。显然，遍历一张画面的全部像素并做判断复杂度过高。又有人提出根据多边形的顶点坐标，给多边形框出一个正方形区域，使得多边形在其之内，缩减需要判断的像素点的数目，但复杂度也没有明显改观（代价主要在判断上）。因此引入了高效的扫描转换填充。

多边形的扫描转换填充在我的理解是一个十分巧妙且高效的算法。本部分算法利用有序边表实现。其中比较重要的原理是，一条直线穿过一个多边形时，大部分情况下有偶数个交点。对这些交点顺序编号，1和2、3和4...之间的部分应该被填充。

假设交点中有多边形的顶点：

1. 如果顶点是局部极值点，那么需要将该局部极值点**看做是两个重合的点**，否则，会误填充多边形外的区域
2. 如果顶点不是局部极值点，那么正常的填充会误认为其是两个顶点，会在某一时刻导致**活动边表中记录的直线与多边形交点成为奇数**，也会误填充多边形之外的区域

针对局部极值点，我们需要做的处理是将其看做是两个点，即在有序边表中该交点两条直线应该在同一个桶中。而针对非局部极值点的点，我们需要进行缩边操作，即**将上面一条边下端点去除或将下面一条边的上端点去除。**我在代码中采取的是**将下面一条边的上端点去除。**另外，还有一种情况，即**多边形的一条边平行于扫描线。**根据算法原理，这条边**不会进入边表**，不影响填充。综上，可以得出有序边表填充步骤为：

1. **根据用户输入建立边表NET**
2. **从图形最下部像素横坐标开始向上，将NET表内容加入AET，绘制，并对AET进行更新**
3. **每次更新，对x坐标加斜率倒数，直到扫描线纵坐标大于等于边表中直线最高点y_{max}为止，删除改边记录**
4. **更新达到图形所占用的最高扫描线，算法结束**

其中扫描操作的具体伪代码为：

for (各条扫描线i)

{

1. 把新边表NET[i]中的边结点用插入排序法插入AET表，使之按x坐标递增顺序排列；
2. 遍历AET表，把y max= i 的结点从AET表中删除，并把y max > i结点的x值递增D x；
3. 对AET表重新排序；
4. 遍历AET表，把配对交点区间(左闭右开)上的象素(x, y)，用drawpixel (x, y, color) 改写象素颜色值；

}

算法实现：

```
ddef polygon_fill(p_list):

    result = []
    n = len(p_list)
    y_max = 0
    y_min = 10000
    for i in range(n):
        x0, y0 = p_list[i]
        if y0 > y_max:
            y_max = y0
        if y0 < y_min:
            y_min = y0

    NET = []
    AET = Node()

    for i in range(0, y_max + 1):
        node = Node()
        NET.append(node)

    for i in range(y_min, y_max + 1):
        for j in range(0, n):
            if p_list[j][1] == i:
                x0, y0 = p_list[j]
                x1, y1 = p_list[(j - 1 + n) % n]
                if y1 > y0:
                    node = Node(x0,float((x1-x0)/(y1-y0)),y1,NET[i].next)
                    NET[i].set_next(node)
                x1, y1 = p_list[(j + 1 + n) % n]
                if y1 > y0:
                    node = Node(x0,float((x1-x0)/(y1-y0)),y1,NET[i].next)
                    NET[i].set_next(node)

    for i in range(y_min, y_max + 1):

        node1 = NET[i].next
        node2 = AET
        while node1 != None:
            while node2.next != None and node1.x >= node2.next.x:
                node2 = node2.next
            temp = node1.next
            node1.set_next(node2.next)
            node2.set_next(node1)
            node1 = temp
            node2 = AET
```

```
        node1 = AET
        node2 = node1.next
        while node2 != None:
            if node2.y_max == i:
                node1.set_next(node2.next)
                node2 = node1.next
            else:
                node1 = node1.next
                node2 = node2.next

        node = AET.next

        while node != None:
            node.set_x(node.x + node.dx)
            node = node.next

        node1 = AET
        node2 = AET.next
        node1.set_next(None)
        while node2 != None:
            while node1.next != None and node2.x >= node1.next.x:
                node1 = node1.next
            temp = node2.next
            node2.set_next(node1.next)
            node1.set_next(node2)
            node2 = temp
            node1 = AET

        node = AET.next
        while node != None and node.next != None:
            x = int(node.x)
            while x <= node.next.x:
                result.append([x, i])
                x = x + 1
            node = node.next.next

    return result
```

3. 系统介绍

3.1 系统框架

cg_algorithms.py：核心算法模块（各种图元的生成、编辑算法）

cg_cli.py：命令行界面（CLI）程序

cg_gui.py：用户交互界面（GUI）程序

3.2 交互逻辑

CLI程序

每次读取一整行指令，由split 函数分解该行指令，通过每一条指令的第一个单词，区别每一条指令的操作。同时解析剩下的单词，构造出函数调用时的参数，从而正确执行出每条指令对应的操作。

GUI程序

通过qt提供的槽函数机制，将主窗口类MainWindow中菜单栏的选项与画布窗体类MyCanvas建立连接，从而能向画布中添加图元类MyItem，并调用相应的paint函数进行绘制。

其中槽函数的格式如下：

```
        exit_act.triggered.connect(qApp.quit)
        set_pen_act.triggered.connect(self.set_pen_action)
        reset_canvas_act.triggered.connect(self.reset_canvas_action)
        save_canvas_act.triggered.connect(self.save_canvas_action)
        line_naive_act.triggered.connect(self.line_naive_action)
```

3.3 设计思路

画布窗体类MyCanvas

继承自QGraphicsView，采用QGraphicsView、QGraphicsScene、QGraphicsItem的绘图框架

捕捉用户的鼠标点击以及释放的坐标，从而通过坐标点对应的p_list，建立图元信息。

自定义图元类MyItem

继承自QGraphicsItem

通过MyCanvas类创建MyItem，保存了图元的各种信息，包括：

```
    :param item_id: 图元ID
    :param item_type: 图元类型，'line'、'polygon'、'ellipse'、'curve'等
    :param p_list: 图元参数
    :param algorithm: 绘制算法，'DDA'、'Bresenham'、'Bezier'、'B-spline'等
```

在绘制图元的过程中，通过调用alg类中的相关算法函数，获得绘制结果的像素点坐标列表，最终将获得的像素坐标绘制到画布上即可得到最终结果。

同时提供了一个boundingRect()函数，用以绘制每一个图元的边框，便于在选择图元时获得更直观的效果。

主窗口类MainWindow

其中：使用QListWidget来记录已有的图元

使用QGraphicsView作为画布

通过menuBar()函数建立菜单栏

通过triggered.connect连接信号和槽函数

每一个菜单中的操作对应一个连接信号，该信号对应一个槽函数，槽函数通过调用MyCanvas类中的对应操作函数，进行图元的绘制以及编辑操作。

4. 其他介绍

4.1 巧妙的设计

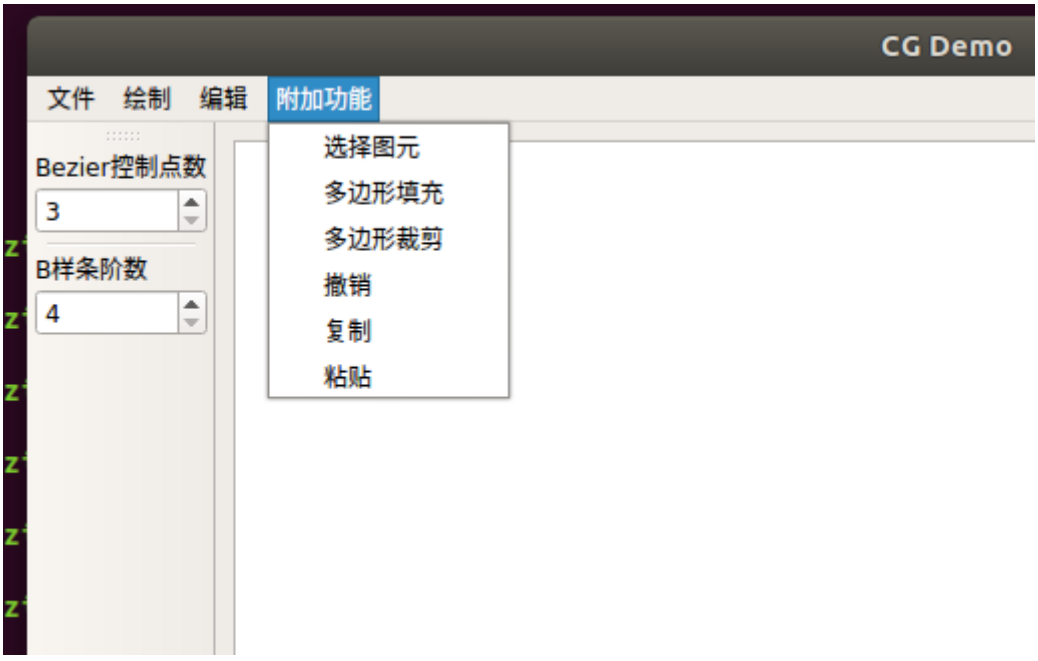
4.1.1 GUI曲线控制点数目设定

通过在MainWindow中添加QSpinBox进行控制点数目设定

```
        self.bezier_box = QSpinBox()
        self.bezier_box.setRange(3, 20)
        self.bezier_box.setSingleStep(1)
        self.bezier_box.setValue(self.bezier_num)
        self.setbar.addWidget(QLabel("Bezier控制点数"))
        self.setbar.addWidget(self.bezier_box)
        self.setbar.addSeparator()
        self.bspline_box = QSpinBox()
        self.bspline_box.setRange(4, 20)
        self.bspline_box.setSingleStep(1)
        self.bspline_box.setValue(self.bspline_num)
        self.setbar.addWidget(QLabel("B样条阶数"))
        self.setbar.addWidget(self.bspline_box)
        self.bezier_box.valueChanged.connect(self.set_bezier_num)
        self.bspline_box.valueChanged.connect(self.set_bspline_num)
```

效果图：

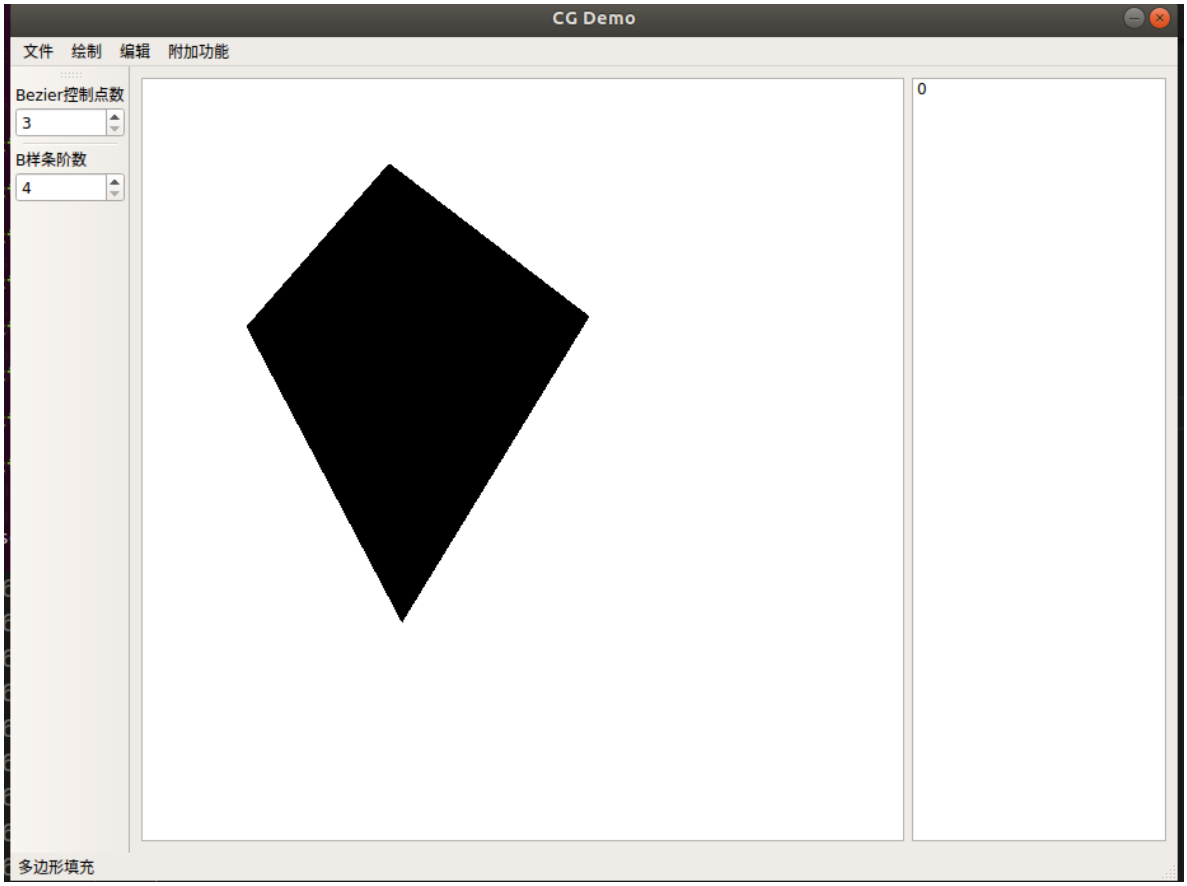
4.2 额外的功能



4.2.1 多边形填充

将绘制的多变形内部使用颜色进行填充

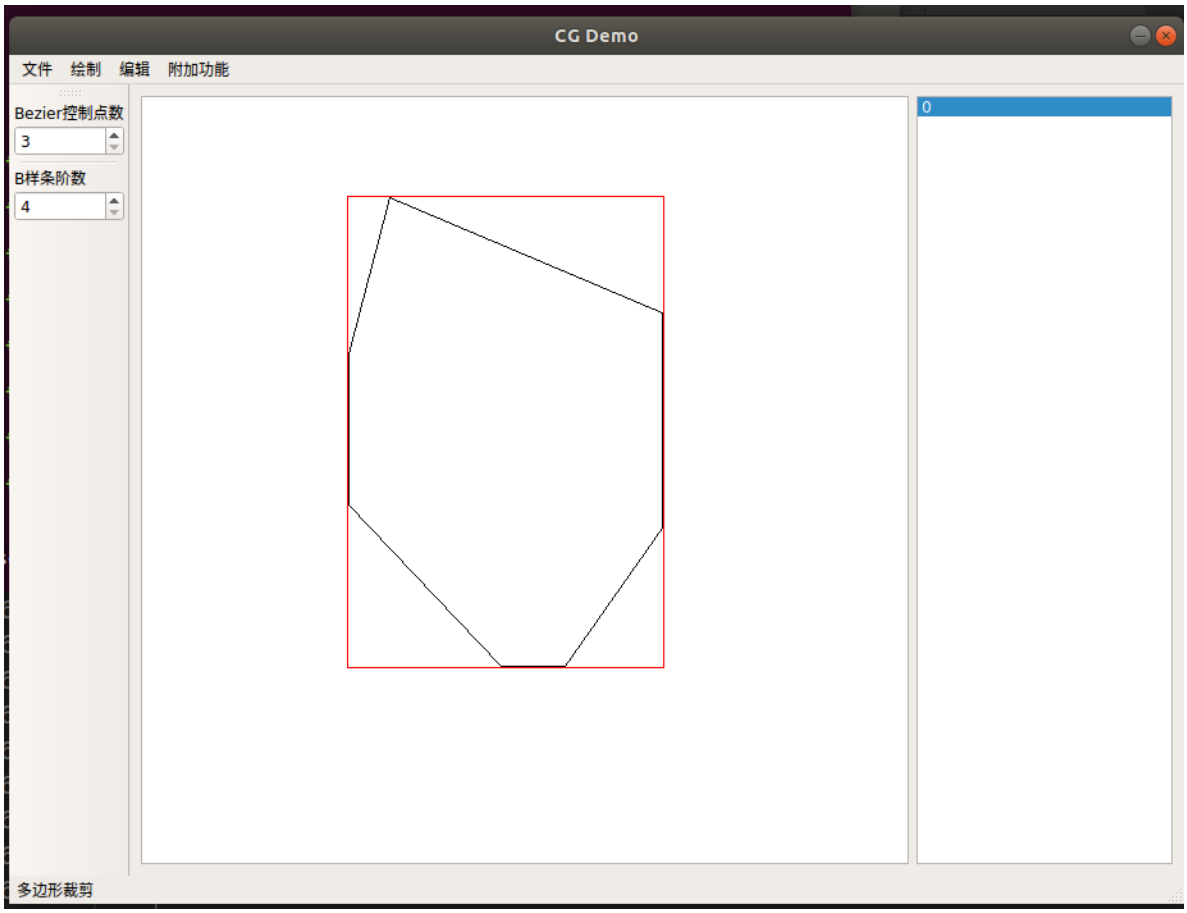
效果图：



4.2.2 多边形裁剪

将绘制的多变形进行裁剪操作

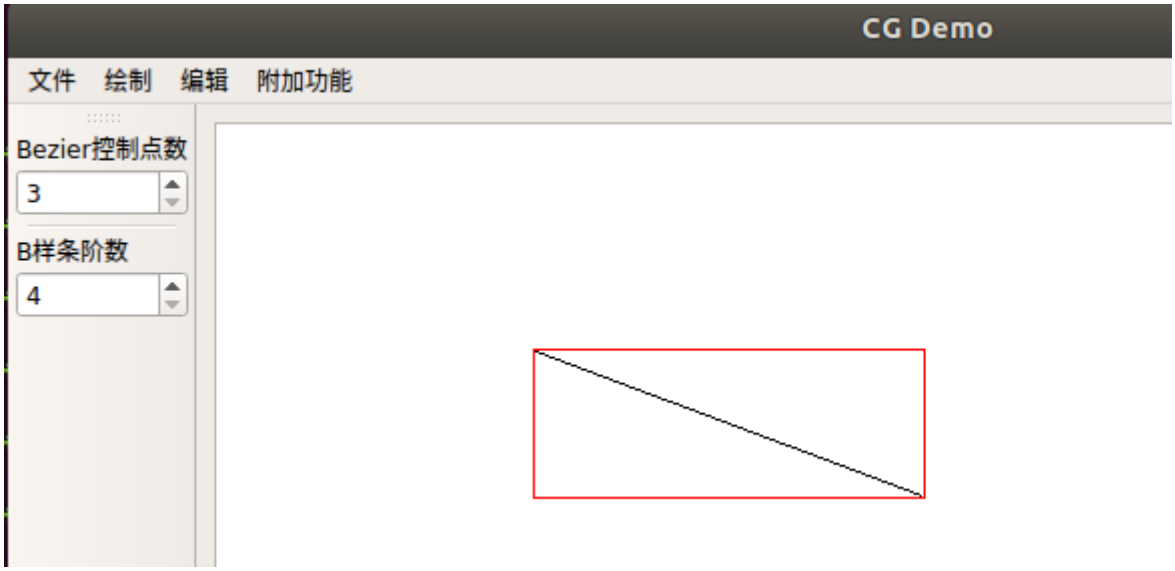
效果图:



4.2.3 选择图元

直接通过鼠标点击选择图元

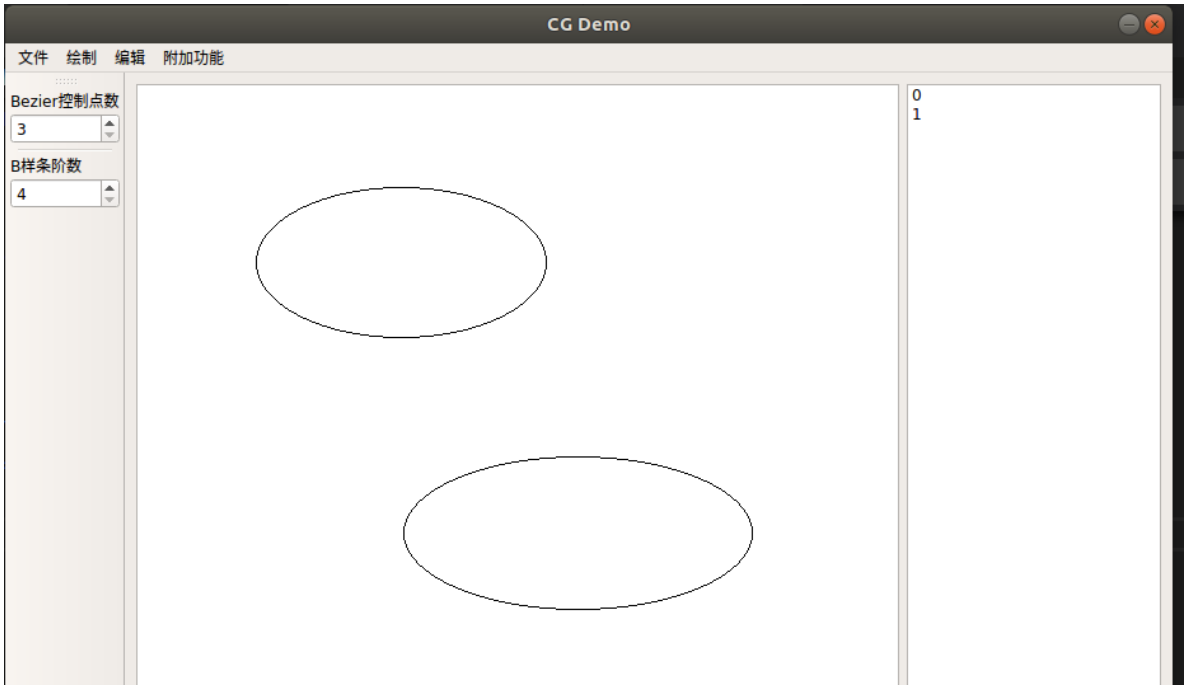
效果图:

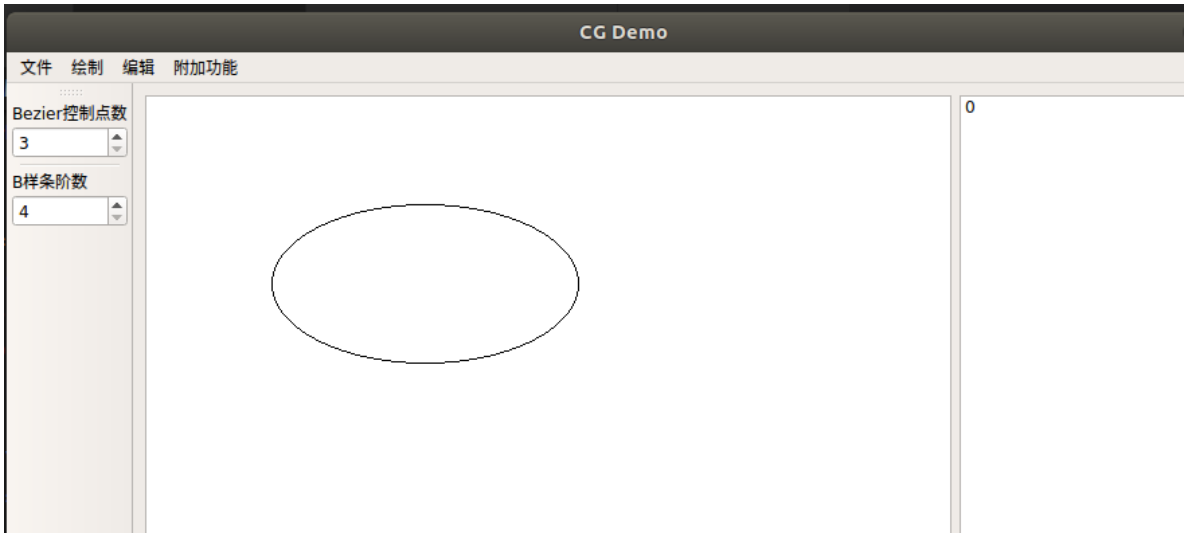


4.2.4 撤销

撤销上一步操作

效果图:

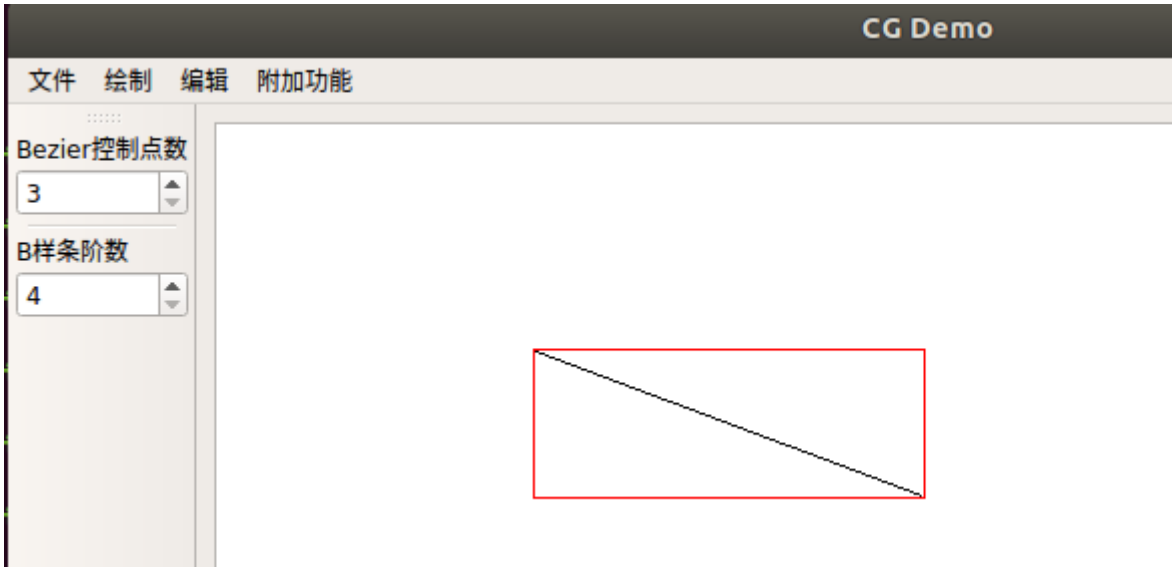




4.2.5 复制

先选中图元，再在菜单中的附加功能中选取复制，即可将图元保存在剪切板中，用以接下来的粘贴操作。

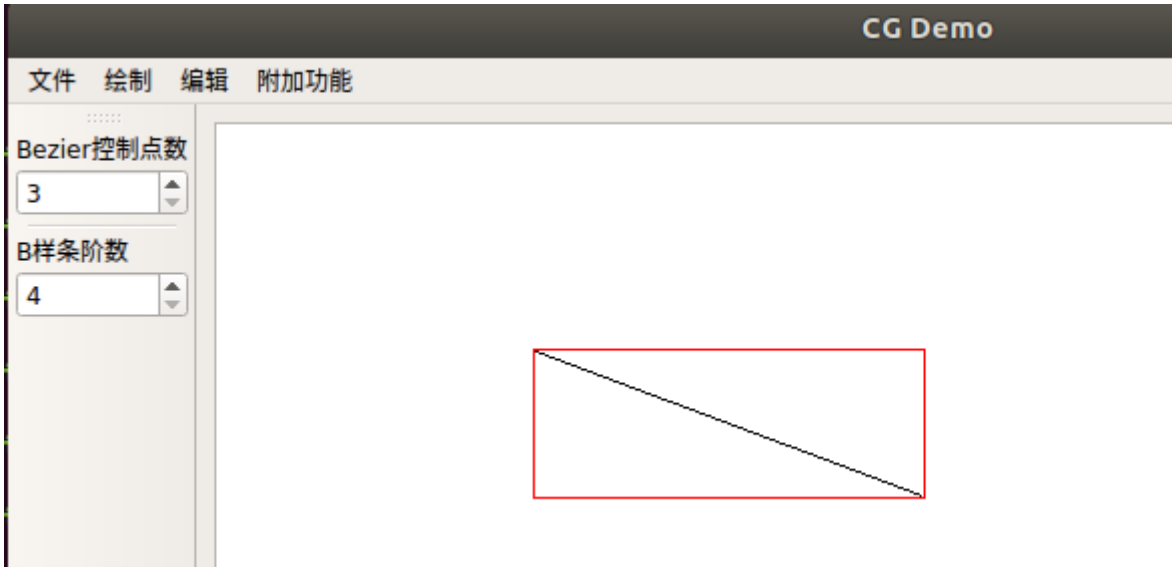
效果图：



4.2.6 粘贴

在菜单中的附加功能中选取粘贴操作，只要之前复制过图元（剪切板中有保存的图元），即可通过在屏幕上直接点击，实现图元的粘贴。

效果图：

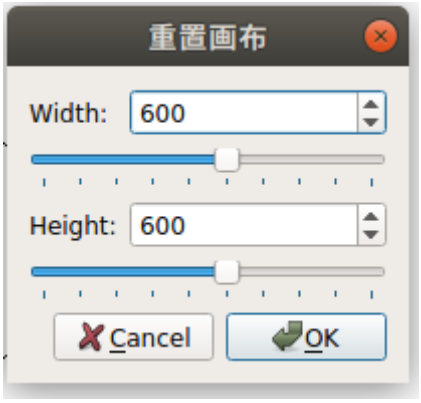


4.3 易用的交互

4.3.1 重置画布

使用qt提供的QSpinBox以及QSlider控件，将重置画布时改大小的过程变得简单易用。

效果图：

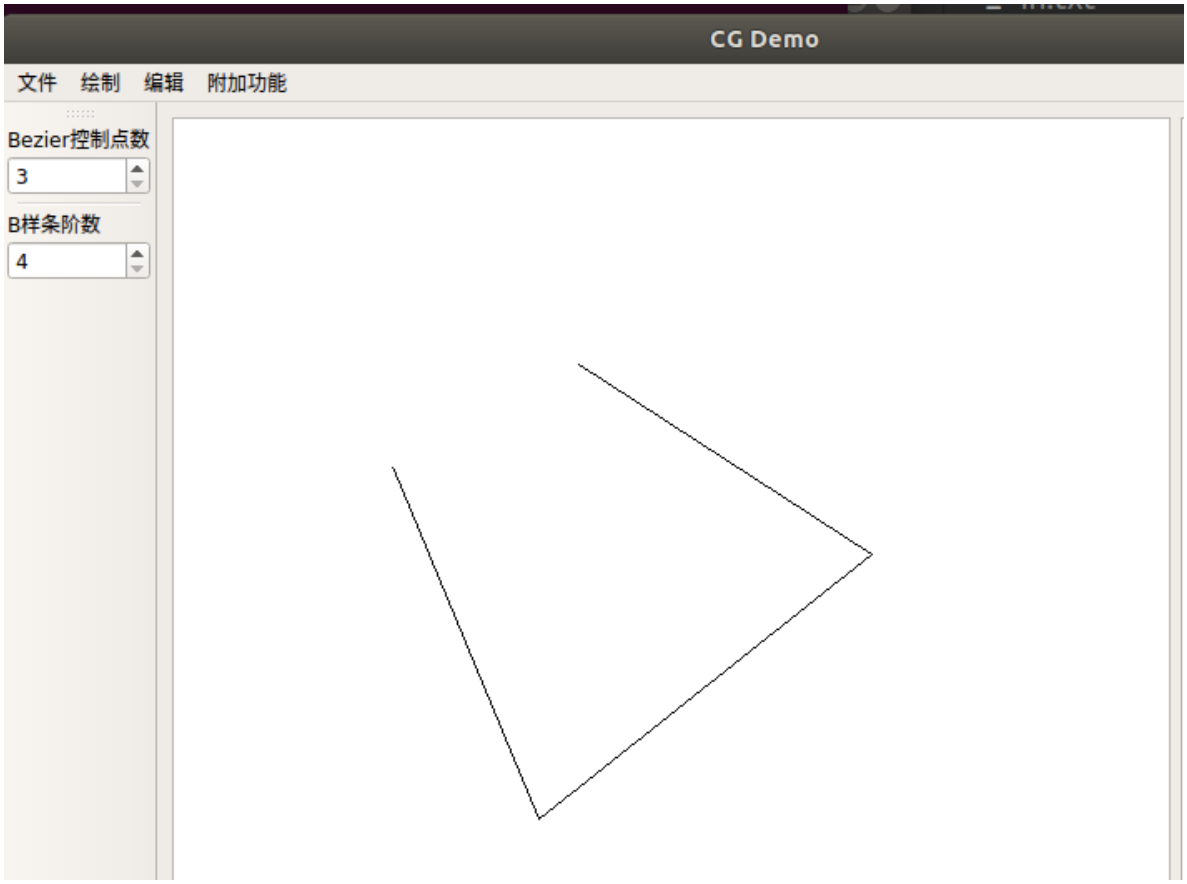


4.3.2 长方形绘制

绘制过程：

选点过程参考了WINDOWS 绘图工具的实现，先同绘制直线的方式一样获取两个初始点，再按下鼠标确定新的点，如果该点的位置距离第一个初始点的距离平方小于50，则结束多边形选点同时不再保存这一个结束点。最后调用多变形绘制算法进行多边形的绘制。因此绘制过程非常方便简单，不用提前选点。同时，在绘制过程中将会禁用菜单栏、选择栏、以及控制点数调整，因此长方形绘制具有很好的鲁棒性。

效果图（绘制过程）：



5. 参考资料

1] 孙正兴, 计算机图形学课程PPT, 2020