计算机图形学系统报告

计算机图形学系统报告

姓名：曹语桐 学号：201240069 邮箱：201240069@smail.nju.edu.cn

1引言

本实验使用`Python3`语言编程，实现一个绘图系统，文件结构如下:

```
学号_报告.pdf
|- 学号_说明书.pdf
|- 学号_演示.mp4
|- source
|   |- cg_algorithms.py
|   |- cg_cli.py
|   |- cg_gui.py
```

其中`cg_algorithms.py`中实现了各种图形学绘制算法，`cg_cli.py`实现了与命令行的交互，`cg_gui.py`实现了用户交互界面，使用户可以用鼠标在画布上绘制图形。

2算法

2.1绘制线段

2.1.1 DDA算法

使用Naive算法可以画出线段，但存在很大的缺陷：线段斜率越大时，绘制的线条上的像素点就越稀疏，当斜率不存在时，线段会消失。这是因为Naive算法对线段上每一个整数x值取样生成一个像素点，当线段在x轴上的投影较短时，取样就会变得稀疏。DDA算法则做出了改进，选取增长更快的坐标轴进行取样，就可以保证另一条坐标轴上1单位间隔中间的点数大于1，保证了线段的连续性。

具体算法流程为：

1.选取增长更快的坐标轴，如果这个坐标轴不是x轴，进行坐标轴的互换，保证x为步进方向

2.在x轴上进行取样

3.如果坐标轴进行过互换，再互换回来

```
reverse=math.fabs(y1-y0)-math.fabs(x1-x0)>0
if reverse:
    p_list=reverseAxis(p_list)
#保证x为步进方向
if p_list[0][0]<p_list[1][0]:
    start=p_list[0]
    end=p_list[1]
else:
    start=p_list[1]
    end=p_list[0]
x0, y0 = start
x1, y1 = end
if x1==x0:
    x=x0
    y=y0
    while y<=y1:
        result.append((x,y))
        y+=1
else:
    k=(y1-y0)/(x1-x0)
    x=x0
    y=y0
    while x<=x1:
        result.append((int(x),int(y)))
        x+=1
        y+=k
if reverse:
    result=reverseAxis(result)
```

2.1.2 Bresenham算法

DDA算法能绘制出比较紧密的线段，但是算法中每生成一个像素点，就需要进行一次浮点数运算和取整操作，费时而且随着线段长度的增加，误差会逐渐变大。Bresenham算法在此方面做了改进，不需要引入费时的浮点数运算，也能避免较大的偏差。

Bresenham算法的原理：

考虑以x轴增长更快的线段，满足这样的性质，x的值每次增长，y的值会在保持不变和增长1之间选择。因此Bresenham的画线算法可以这样简单描述:给定两点: 起点(x1, y1), 终点(x2, y2), 连接他们的直线可以用一元一次方程y=mx+b来描述, 规定m∈ (0,1)。应用Bresenham算法:定义一个参数P[i]用于判断下一个小方块的y值:y[i+1] 是保持不变还是增加1。p[i]的值是初始为`2*dy-dx`,若p[i]>=0,p[i]=`p[i]+2*dy-2*dx`,y[i+1] 增加1;若p[i]<0,p[i]=`p[i]+2*dy`,y[i+1] 保持不变。

算法流程：

1.需要的话互换xy轴

2.初始化p[0]

3.根据p[i]的值判断y[i+1] 是保持不变还是增加1，然后计算p[i+1]

4.重复第三步直至所有x值取样完

5.需要的话互换xy轴

```
reverse=math.fabs(y1-y0)-math.fabs(x1-x0)>0
if reverse:
    p_list=reverseAxis(p_list)
#保证x为步进方向
if p_list[0][0]<p_list[1][0]:
    start=p_list[0]
    end=p_list[1]
else:
    start=p_list[1]
    end=p_list[0]
x0, y0 = start
x1, y1 = end
dx=math.fabs(x1-x0)
dy=math.fabs(y1-y0)
c=0
if y1-y0>0:
    c=1
else:
    c=-1
x=x0
y=y0
delta=2*dy-dx
while x<=x1:
    result.append((x,y))
    if delta>=0:
        x+=1
        y+=c
        delta=delta+2*dy-2*dx
    else:
        x+=1
        delta=delta+2*dy
if reverse:
    result=reverseAxis(result)
```

2.2绘制多边形

绘制多边形就相当于绘制顶点互相连接的线段，只需要按照要求的两个算法DDA算法和Bresenham算法依次绘制多边形边就可以了。

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

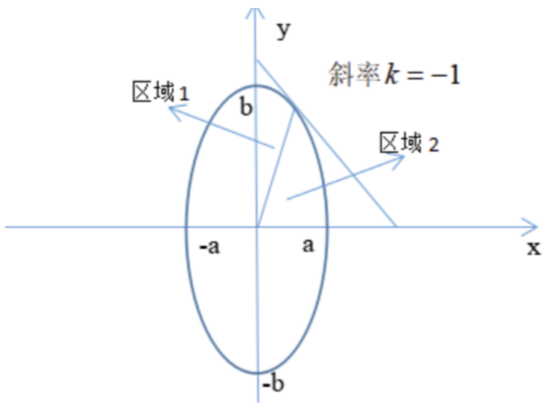
2.3绘制椭圆

本实验采用中点椭圆算法绘制椭圆

中点椭圆算法原理

中点椭圆算法的思路其实与Bresenham相似，也是在某一区域范围内，单位间隔取样，确定离指定椭圆最近的像素位置。这里只需要取样绘制四分之一的像素点（本实验选择绘制第一象限的椭圆），然后可以通过椭圆的对称性，绘制其他像素点。对于椭圆中心不在原点处的情况，只需要通过平移将椭圆中心平移到远点，绘制完成后，再将计算出的每个位置(x,y)往相反方向再平移回去就可以了。

如下图所示，在第一象限内，当过椭圆上一点的斜率k>-1时，在x方向取单位步长，即从位置(0,b)开始，在第一象限沿椭圆路径，当前点取(xk,yk)时，x步进一个单位，判断下一点取(x[k]+1,y[k])还是(x[k]+1,y[k]-1)；反之当k<-1时，在y方向取单位步长，当前点取(x[k],y[k])，判断下一点取(x[k],y[k]+1)还是取(x[k]-1,y[k]+1)。因此我们需要按照k>-1和k<-1将第一象限的椭圆划分成两个区域讨论。（易知椭圆在`ry**2*x==rx**2*y`时斜率k=-1，在此之前选取y轴为步进方向，之后选取x轴为步进方向）

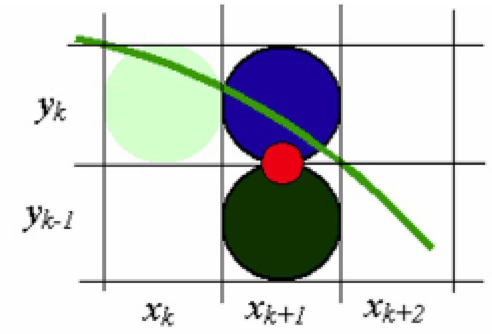


定义函数`f(x,y)=(ry*x)**2+(rx*y)**2-(r1*r2)**2`，对于任意点(x,y)： 若f(x,y)<0，则点位于圆内； 若f(x,y)=0，则点位于圆上； 若f(x,y)>0，则点位于圆外。如下图所示，将中点的坐标带入函数中：`p[k]=f(x[k]+1,y[k]-0.5)=(ry*(x[k]+1))**2+(rx*(y[k]-0.5))**2-(r1*r2)**2`。

若p[k]<0，表示中点位于椭圆的内部，应该选择中点上方的候选点(x[k]+1,y[k])，此时`p[k+1]=f(x[k+1]+1,y[k]-0.5)=(ry*(x[k+1]+1))**2+(rx*(y[k]-0.5))**2-(r1*r2)**2=p[k]+ry**2* (2*x[k+1]+1)`；

若p[k]>=0，表示中点位于椭圆的外部，应该选择中点下方的候选点(x[k]+1,y[k]-1)，此时`p[k]=f(x[k]+1,y[k+1]+0.5)=(ry*x[k+1])**2+(rx*(y[k+1]+0.5))**2-(r1*r2)**2`

`p1[k+1]=f(x[k+1]+1,y[k+1]-0.5)=(ry*(x[k+1]+1))**2+(rx*(y[k+1]-0.5))**2-(r1*r2)**2=p[k]+2*ry**2*x[k+1]-2*rx**2*y[k+1]+ry**2`



算法流程：

1.将椭圆中心平移到坐标原点

2.从坐标(0,ry)开始初始化p[0]，y轴为步进方向

3.根据p[i]的值判断y[i+1] 是保持不变还是增加1，然后计算p[i+1]

4.重复第三步直至所有斜率为-1

5.更换x为步进方向，计算区域2的像素点

6.对称的将第一象限的像素点对称到其他象限

7.平移椭圆到中心与原中心重合

```
result=[]
center=[int((x0+x1)/2),int((y0+y1)/2)]
rx=math.fabs((x1-x0)/2)
rx=int(rx)
ry=math.fabs((y0-y1)/2)
ry=int(ry)
#part 1
x,y=0,ry
d=ry**2-rx**2*ry+rx**2/4
while ry**2*x<rx**2*y:
    result.append([x,y])
    x+=1
    if d>=0:
        y-=1
        d+=2*ry**2*x+ry**2-2*rx**2*y
    else:
        d+=2*ry**2*x+ry**2

#part 2
d=ry**2*(x+0.5)**2+rx**2*(y-1)**2-rx**2*ry**2
while y>=0:
    result.append([x,y])
    y-=1
    if d<=0:
        x+=1
        d+=rx**2-2*rx**2*y+2*ry**2*x
    else:
        d+=rx**2-2*rx**2*y


result=copy_ellipse(result)
```

```
result=translate(result,center[0],center[1])
```

2.4绘制曲线

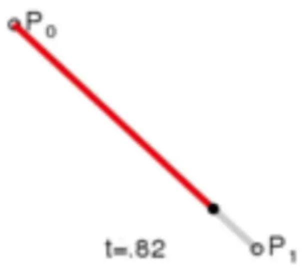
2.4.1 Bezier曲线

原理：

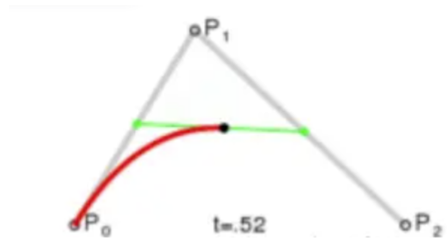
贝塞尔曲线是线性插值的结果。对参数方程P(t)定义域中的每一个t值生成一个插值点。为了生成这个点，我们要根据序列中每两个相邻的点，生成位于这两点间(1-t)/t位置的新顶点，并将其添加进下一个阶数的新点集中。对新的点集重复这一操作，直到新点集中只剩一个顶点，这个顶点即为这个t值所求的顶点，将其添加到结果中。t的取值范围为[0,1]，t取值间隔越小，曲线越紧密，反之越稀疏。

下面以一阶、二阶、三阶曲线为例描述Bezier曲线的生成过程：

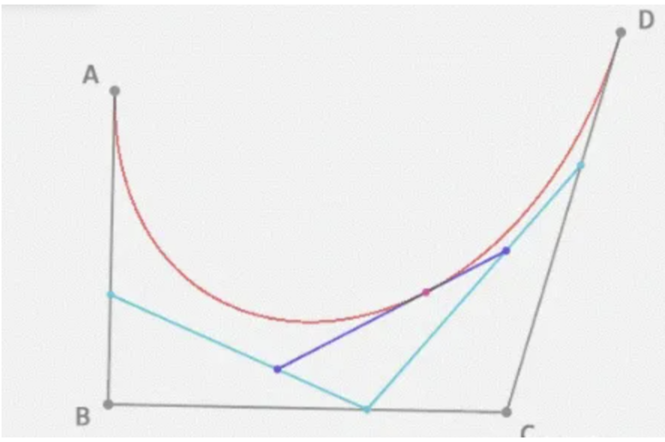
一阶曲线：



二阶曲线：



三阶曲线：



Bezier曲线生成公式：

$$P(t) = \sum^{n} P_i B_{i,n}(t), t \in [0,1]$$

$$B_{i,n}(t) = C_n^i t^i (1-t)^{n-i} = \frac{n!}{i!(n-i)!} t^i (1-t)^{n-i} \text{【}i = 0, 1, \cdots, n\text{】}$$

其中`B[i][n][t]`可以通过递归的de_Casteljau算法求出以提高效率，公式如下：

$$B_{i,n}(t) = (1-t)B_{i,n-1}(t) + tB_{i-1,n-1}(t) \text{【}i = 0, 1, \cdots, n\text{】}$$

```
def de_Casteljau(num,p_list,t):
    p=[]
    for r in range(num):
        if r==0:
            for i in range(num):
                p.append(p_list[i])
        else:
            for i in range(num-r):
                x=(1-t)*p[i][0]+t*p[i+1][0]
                y=(1-t)*p[i][1]+t*p[i+1][1]
                p[i]=[x,y]
    return p[0]

result=[]
num=len(p_list)
tnum=num*1000
if algorithm=="Bezier":
    for i in range(tnum):
        t=i/tnum
        x,y=de_Casteljau(num,p_list,t)
        result.append([int(x),int(y)])
```

2.4.2 B-spline（三次四阶，均匀）

原理：

B样条曲线与贝塞尔曲线的不同是Bezier算法每个点都是由整条曲线生成的，而本算法每个点仅由附近的几个数据点生成，因此每个数据 点仅影响附近的位置。B样条有三大要素：节点，控制点，阶次。控制点和贝塞尔的一样，就是空间上决定曲线形状的点。设*U* 是*m* + 1个非递减数的集合，*u*0 <= *u*2 <= *u*3 <= ... <= *um*。*ui*称为节点（knots），集合U 称为节点向量（knot vector），半开区间[*ui*, *ui*+1) 是第*i*个节点区间（*knot span*）。注意某些*ui*可能相等，某些节点区间会不存在。如果节点等间距(即，*ui*+1 - *ui* 是一个常数，对 0 <= *i* <= *m* - 1)，节点向量或节点序列称为均匀的；否则它是非均匀的。节点可认为是分隔点，将区间[*u*0, *um*]细分为节点区间。所有B-样条基函数被假设定义域在[*u*0, *um*]上。在本文中，我们经常使用*u*0 = 0和*um* = 1，所以定义域是闭区间[0,1]。

为了定义B-样条基函数，我们还需要一个参数，基函数的次数（degree）*p*，第*i*个*p*次B-样条基函数，写为*Ni,p*(*u*)。

递归定义如下：

$$b_{j,0}(t) := \begin{cases} 1 & t_j < t < t_{j+1} \\ 0 & \dots \end{cases}$$

$$b_{j,n}(t) := \frac{t - t_j}{t_{j+n} - t_j} b_{j,n-1}(t) + \frac{t_{j+n+1} - t}{t_{j+n+1} - t_{j+1}} b_{j+1,n-1}(t).$$

B样条基函数定义如下：

$$\mathbf{S}(t) = \sum_{i=0}^{m} \mathbf{P}_i b_{i,n}(t)\ ,\ t \in [0,1].$$

```
def deBoor_Cox(u, k, i):
if k==1:
    if u>=i and u<i+1:
        return 1
    else:
        return 0
else:
    b1=deBoor_Cox(u,k-1,i)
    b2=deBoor_Cox(u,k-1,i+1)
    x=b1*(u-i)/(k-1)+b2*(i+k-u)/(k-1)
    return x


result=[]
num=len(p_list)
k=4#k阶B样条
u=k-1
while u<num:
    x,y=0,0
    for i in range(num):
        #为每个点计算 B[i,k]
        b=deBoor_Cox(u,k,i)
        x+=(b*p_list[i][0])
        y+=(b*p_list[i][1])
    result.append([round(x),round(y)])
    u+=(1/1000)
```

2.5 图片平移

图元平移较为简单，只需要把选中图形的像素点的(x,y)值分别加上(dx,dy)就可以了，具体代码如下：

```
new_list=[]
for point in p_list:
    nx,ny=point
    nx+=dx
    ny+=dy
    new_list.append([nx,ny])
```

2.6 图元旋转

如下图所示，要绕(0,0)旋转(nx,ny)点`r°`到(rx,ry)，需要经过以下的变换：`rx=round(nx*math.cos(r)-ny*math.sin(r)); ry=round(ny*math.cos(r)+nx*math.sin(r))`

图元旋转流程：

1.对图形进行整体平移，将旋转中心平移到（0，0）

2.将选中图形的每个点都进行上述旋转变换

3.如果经过了平移，再平移回原位置

其中在进行旋转前，需要把角度转化为弧度，并且对每一个像素点的坐标值进行取整

```
new_list=translate(p_list,-x,-y)
r=(r/180)*math.pi
res_list=[]
for point in new_list:
    nx,ny=point
    rx=round(nx*math.cos(r)-ny*math.sin(r))
    ry=round(ny*math.cos(r)+nx*math.sin(r))
    res_list.append([rx,ry])
res_list=translate(res_list,x,y)
```

2.7 图元缩放

缩放倍数为s，若缩放中心为(0,0)，缩放前像素位置为(x1,y1)，缩放后像素位置为(x2,y2)，则 `x2=x1*s ，y2=y1*s` 。

图元缩放流程：

1.对图形进行整体平移，将缩放中心平移到(0,0)

2.将选中图形的每个点都进行上述缩放变换

3.如果经过了平移，再平移回原位置

```
new_list=[]
    for point in p_list:
        nx,ny=point
        nx=round(x+(nx-x)*s)
        ny=round(y+(ny-y)*s)
        new_list.append([nx,ny])
```

2.8 对线段裁剪

2.8.1 Cohen-Sutherland 算法

Cohen-Sutherland算法原理：

Cohen-Sutherland算法对需要裁剪的线段的端点进行编码，每段直线的端点都被赋予一组四位二进制代码，称为区域编码，用来标识直线段端点相对于窗口边界及其延长线的位置。
本实验的窗口为矩形窗口，由上、下、左、右4条边界组成，延长窗口的4条边界形成9个区域。这样根据直线段的任一端点P(x，y)所处的窗口区域位置，可以赋予一组4位二进制编码，称为区域码 `RC=C3C2C1C0`。C0代表左边界， C1代表右边界， C2代表下边界， C3代表上边界。

为了保证窗口内及窗口边界上直线段端点的编码为零，定义规则如下：
C0：若端点的 x < wxl，则C0=1，否则C0=0。
C1：若端点的x > wxr，则C1=1，否则C1=0。
C2：若端点的y < wyb，则C2=1，否则C2=0。
C3：若端点的y > wyt，则C3=1，否则C3=0。

算法流程：

1.对直线段两个端点进行编码

2.若直线段的两个端点的编码都为0，说明直线段的两个端点都在窗口内，保留整条线段；

若直线段的两个端点的编码都不为0，且RC0&RC1≠0，说明直线段位于窗外的同一侧，或左方、或右方、或上方、或下方，整条线段可以舍弃；

若直线段既不满足“简取”也不满足“简弃”的条件，则需要与窗口进行“求交”判断。这时，直线段必然与窗口边界或窗口边界的延长线相交，重复对窗口外的线段端点求其与窗口边界及延长线的交点，进行剪裁，直到直线段两个端点的编码都为0或两个端点的编码RC0&RC1≠0

```
if algorithm=="Cohen-Sutherland":
code0=0
code1=0
while 1:
    code0,code1=0,0
    if x0<x_min:code0+=1
    if x0>x_max:code0+=2
    if y0<y_min:code0+=4
    if y0>y_max:code0+=8
    if x1<x_min:code1+=1
    if x1>x_max:code1+=2
    if y1<y_min:code1+=4
    if y1>y_max:code1+=8

    if (code0|code1)==0:
        return [[x0,y0],[x1,y1]]
    if (code0&code1)!=0:
        return [[0,0],[0,0]]
```

```
    #对code0操作
    if code0==0:
        code0,code1=code1,code0
        x0,y0,x1,y1=x1,y1,x0,y0
    if code0&1:
        y0=round(y0+(x_min-x0)*(y1-y0)/(x1-x0))
        x0=x_min
    if code0&2:
        y0=round(y0+(x_max-x0)*(y1-y0)/(x1-x0))
        x0=x_max
    if code0&4:
        x0=round(x0+(y_min-y0)*(x1-x0)/(y1-y0))
        y0=y_min
    if code0&8:
        x0=round(x0+(y_max-y0)*(x1-x0)/(y1-y0))
        y0=y_max
```

2.8.2 Liang-Barsky 算法

算法原理：

Liang-Barsky算法用参数方程表示直线，将待剪裁直线看作一个有方向的线。将裁剪线段和裁剪窗口都视为点集，裁剪的结果是两个点集的交集。

直线参数方程：

$x=x1+u*(x2-x1)=x1+u*\Delta x, 0\le u\le 1$

$y=y1+u*(y2-y1)=y1+u*\Delta y, 0\le u\le 1$

对于上下左右四条窗口的边，需要算出它们与线段交点的u值，不等式如下：

$$\begin{cases} x_{min} \le x_1 + u * \Delta x \le x_{max} \\ y_{min} \le y_1 + u * \Delta y \le y_{max} \end{cases}$$

化简后：

$$\begin{cases} u * (-\Delta x) \le x_1 - x_{min} \\ u * \Delta x \le x_{max} - x_1 \\ u * (-\Delta y) \le y_1 - y_{min} \\ u * \Delta y \le y_{max} - y_1 \end{cases}$$

可归纳为：

$$u * p_k \le q_k , k = 1, 2, 3, 4$$

当p[k]<0时，线段从裁剪边界延长线的外部延伸到内部，也就是入边

当p[k]>0时，线段从裁剪边界延长线的内部延伸到外部，也就是出边

当p[k]= 0时，且q[k]<0则线段完全在边界外；若q[k]<=0则线段完全在边界内

```
elif algorithm=="Liang-Barsky":
    p=[x0-x1,x1-x0,y0-y1,y1-y0]
    q=[x0-x_min,x_max-x0,y0-y_min,y_max-y0]
    u0,u1=0,1
    for i in range(4):
        if p[i]<0:
            #入边
            u0=max(u0,q[i]/p[i])
        elif p[i]>0:
            #out
            u1=min(u1,q[i]/p[i])
        elif p[i]==0:
            if q[i]<0:
                return [[0,0],[0,0]]
        if u0>u1:
            return [[0,0],[0,0]]

    r_x0=round(x0+u0*(x1-x0))
    r_y0=round(y0+u0*(y1-y0))
    r_x1=round(x0+u1*(x1-x0))
    r_y1=round(y0+u1*(y1-y0))
    result=[[r_x0,r_y0],[r_x1,r_y1]]
```

3 系统框架

3.1 命令行交互框架

以下针对主要部分的改动做出阐释。

1.saveCanvas指令

遍历item_dict中每一个图元，调用相关的绘制算法，得到像素点数组，将每个像素点位置颜色改为当前的画笔颜色。

2.图元绘制指令

包括线段、多边形、椭圆、曲线，负责创建一个新的item，用命令行指令中的图元信息定义item，然后存入item_dict，以备保存画布时调用。

3.图元变换指令

调用对应算法，对需要变换的图元item的p_list重新赋值，以备保存画布时调用。

3.2 用户交互逻辑

3.2.1MainWindow类

在文件菜单中添加缺少的保存画布选项，然后为信号设置槽函数。

图元绘制类的槽函数与线段naive算法绘制相似，主要调用MyCanvas类，然后对选择信息进行清空，如下：

```
def line_naive_action(self):
    self.canvas_widget.start_draw_line('Naive', self.get_id())
    self.statusBar().showMessage('Naive算法绘制线段')
    self.list_widget.clearSelection()
    self.canvas_widget.clear_selection()
```

图元变换类主要调用MyCanvas类，如下:

```
def translate_action(self):
    self.canvas_widget.start_translate()
    self.statusBar().showMessage('平移图元')
```

保存画布功能，调用`QFileDialog.getSaveFileName`,用户输入文件名称，就可以将当前画布上的内容保存到指定文件中。

重置画布功能，对`item_cnt list_widget scene  canvas_widget`进行重置。

3.2.2 MyCanvas类

增加了如下变量:

```
self.pen_color=QColor(255,0,0)#画笔颜色
self.x0=0,self.y0=0#平移坐标1，裁剪坐标1，旋转中心，缩放中心
self.x1=0,self.y1=0#平移坐标2，裁剪坐标2，缩放坐标1
self.x2=0,self.y2=0#缩放坐标2
self.degree=0#旋转角
```

鼠标动作:

1.线段绘制

按下时为线段起点，线段中点随鼠标运动，直至鼠标松开为最终线段终点。

2.多边形绘制

依次按下左键的鼠标位置为多边形顶点，顶点设置完毕后，按下右键，终止此多边形的绘制。

3.椭圆

按下时为椭圆包围框左上角，包围框右下角随鼠标运动，直至鼠标松开为最终右下角。

4.曲线

依次按下左键的鼠标位置为曲线控制点，控制点设置完毕后，按下右键，终止此曲线的绘制。

5.平移

按下时为(x0,y0)，(x1,y1)随鼠标运动，直至鼠标松开为最终(x1,y1)。dx=x1-x0,dy=y1-y0为平移过程中的移动量。

6.旋转

按下左键设定旋转中心。右键按下时为(x1,y1)，(x2,y2)随鼠标运动，直至鼠标松开为最终(x2,y2)。旋转中心分别与(x1,y1)，(x2,y2)连线的夹角为旋转角度。

7.缩放

按下左键设定缩放中心(x0,y0)。右键按下时为(x1,y1)，(x2,y2)随鼠标运动，直至鼠标松开为最终(x2,y2)。s=(x2-x0)/(x1-x0)为缩放倍数

8.裁剪

按下时为裁剪窗口左上角，右下角随鼠标运动，直至鼠标松开为最终裁剪窗口右下角。

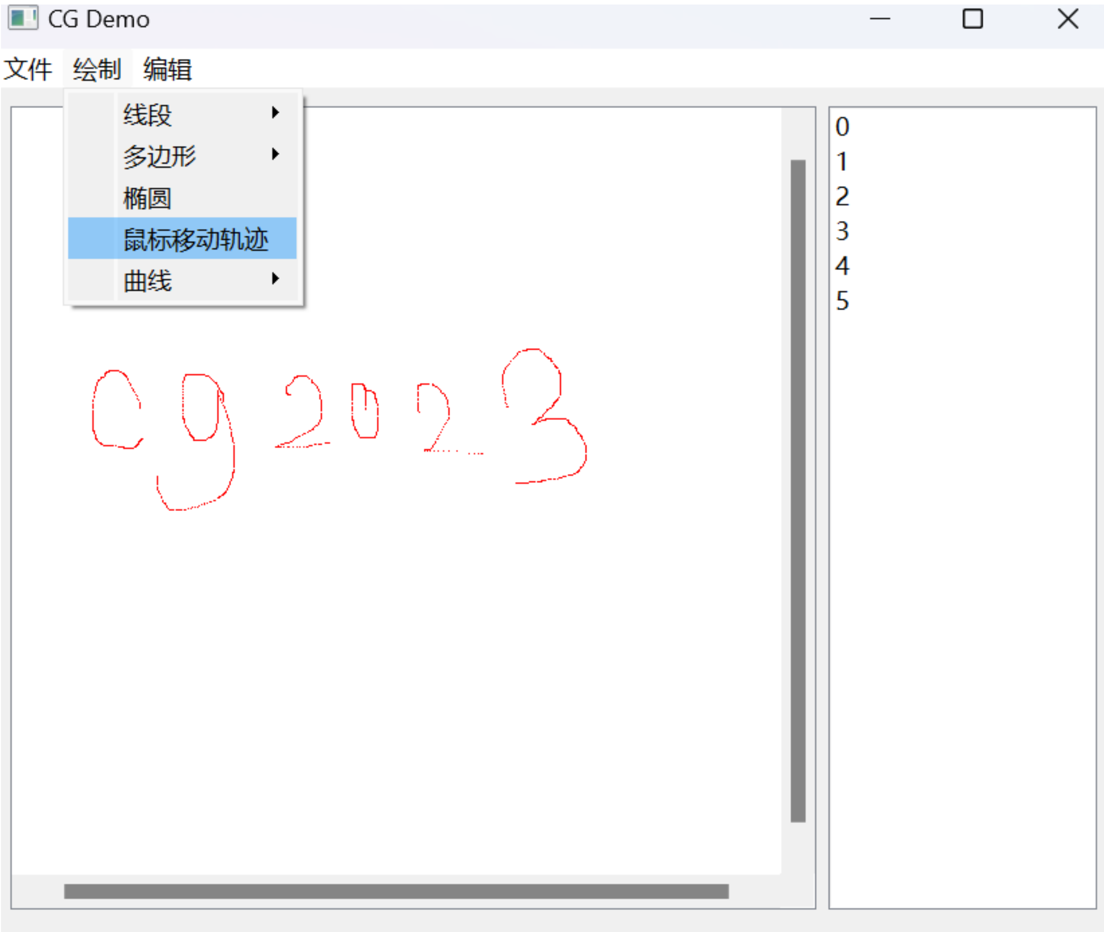
3.2.3 MyItem

补充完成`painter`函数中对不同图元类型的绘制，调用alg中的算法完成。

补充完成`boundingRect`中对除线段外其他图元类型的边框确定。

4 额外功能

添加了绘制鼠标运动轨迹的功能，按下鼠标在画布上拖动就可以记录下鼠标轨迹，效果图如下：



5 参考资料

【1】《计算机图形学教程》，孙正兴、周良等编，机械工业出版社，2006；

【2】Liang-Barsky 算法https://blog.csdn.net/weixin_44397852/article/details/109081908

【3】维基百科 B-样条https://zh.wikipedia.org/wiki/B%E6%A0%B7%E6%9D%A1

【4】Cohen-Sutherland 直接剪裁算法 https://blog.csdn.net/jxch/article/details/80726853