《计算机图形学》12月报告

孙赫彬　201220187　邮箱:1747933957@qq.com

1 引言

本实验要完成一个图形学系统，在实现的过程中，需要应用各种图形学算法实现cg_algorithms.py中的各个函数，补全cg_cli.py中对命令行界面的实现，并且在cg_gui.py中实现用户交互界面的相关功能。

截至目前，我已经完成了：

算法部分 cg_algorithm.py

命令行界面 cg_cli.py

用户交互界面 cg_gui.py

2 算法

2.1 绘制线段

2.1.1 DDA算法

框架代码中给出了Naive算法的实现，但是在用该算法画图后，会发现越竖直的线越稀疏，完全竖直的线甚至会消失。这是因为Naive算法中，以x轴为单位间隔取样生成像素点，所以越竖直的线，生成的像素点越稀疏。

而DDA算法就是在Naive算法的基础上做了改进，它选取增长更快的坐标轴为单位间隔取样，这样就保证了另一条坐标轴上单位间隔之间的点大于等于1，确保了画出的图形不会过于稀疏。

具体的实现方法为：

先判断线段水平和垂直两种特殊情况，这样后面才能够计算斜率。

先求线段的斜率k，若斜率的绝对值小于等于1，则以x轴为单位间隔取样，x每增加1，y增加k；

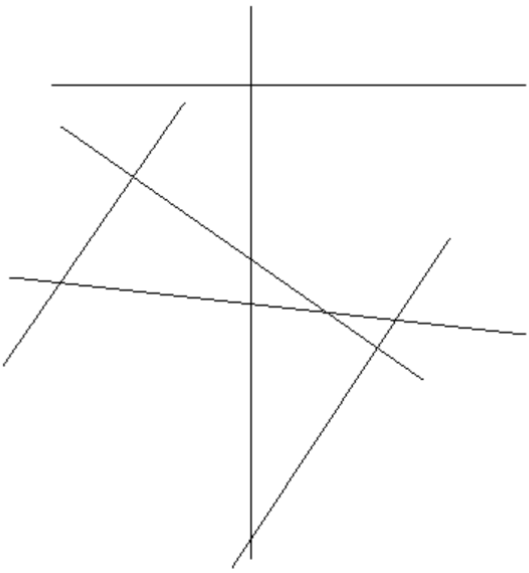
若斜率的绝对值大于1，则以y轴为单位间隔取样，y每增加1，y增加1/k

每次取样的结果都添加进列表result中即可。

```
        if x0 > x1:
            x0, y0, x1, y1 = x1, y1, x0, y0 # 确保x0在x1的左部
        if x0 == x1:
            if y0 > y1:
                y0, y1 = y1, y0
            for y in range(y0, y1 + 1):
                result.append((x0, y))
        elif y0 == y1:
            for x in range(x0, x1 + 1):
                result.append((x, y0))
        else:
            k = (y1 - y0) / (x1 - x0)
            if k <= 1 and k >= -1:
                temp = y0
                for x in range(x0, x1 + 1):
```

```
                result.append((x, int(temp)))
                temp += k
        else:
            if y0 > y1:
                x0, y0, x1, y1 = x1, y1, x0, y0 # 确保y0在y1的下部
            k = (x1 - x0) / (y1 - y0)
            temp = x0
            for y in range(y0, y1 + 1):
                result.append((int(temp), y))
                temp += k
```

用DDA算法进行测试的结果如下：



2.1.2 Bresenham算法

DDA算法中，连续的浮点数运算和取整操作耗时且容易积累误差，与之相对比，Breseham算法就很好的解决了这些问题，该算法通过整数增量运算，避免了复杂的运算。

为了方便理解，可以先考虑(x0,y0)在(x1,y1)的左侧，斜率0<m<1的情况。我们要做的是沿x轴单位距离取样，对于每个x值，选择离直线距离最近的整数y值并记录，直到取样到(x1,y1)为止。

假设第k次取样得到的像素点坐标为(x[k],y[k])，那么下一次取样的像素点有两种可选项：(x[k]+1,y[k])和(x[k]+1, y[k]+1)。为了确定到底要选取哪个像素点，可以将它们两个的y值与x坐标在直线上对应的准确y值进行比较。

假设直线方程为: y=mx+b，则：

x[k+1]对应的y值为：y=m*x[k+1]+b

(x[k]+1,y[k])与(x[k+1],y)的距离为：d1=y-y[k]=m*x[k+1]+b-y[k]

```
(x[k]+1,y[k]+1)与(x[k+1],y)的距离为：d2=y[k]+1-y=y[k]+1-m*x[k+1]-b
```

两式相减得：d1-d2=2*m*x[k+1]-2*y[k]*+2b-1

如果d1-d2为正数，那么 (x[k]+1,y[k])与(x[k+1],y)的距离比 (x[k]+1,y[k]+1)与(x[k+1],y)的距离大，应该选择 (x[k]+1, y[k]+1)，反之，应该选择 (x[k]+1,y[k])。

注意到m不是整数，如果不做优化，在计算d1-d2的时候仍然需要进行浮点数运算，这不是我们期望的。

考虑到m=dy/dx，我们可以把等式两边同时乘dx，得到:

dx(d1-d2)=2*dy*x[k]-2*dx*y[k]+2*dy+dx*(2b-1)

dx(d1-d2)与d1-d2的正负是相同的，因此可以判断dx(d1-d2)的大小。

设第k次的dx(d1-d2)为p[k]，则:

p[0]=2*dy*x0-2*dx*y0+2*dy+dx*(2b-1)=2*dy-dx

p[k]=2*dy*x[k]-2*dx*y[k]+2*dy+dx*(2b-1)，可以看出p[k]与p[k+1]之间并没有差很多，可以推到公式进行求解:

p[k+1]=2*dy*x[k+1]-2*dx*y[k+1]+2*dy+dx*(2b-1)

p[k+1]-p[k]=2*dy-2*dx*(y[k+1]-y[k])

若p[k]>0，则选择(x[k]+1, y[k]+1)作为下一个像素点，故p[k+1]=p[k]+2*dy-2*dx

反之，则选择 (x[k]+1,y[k])作为下一个像素点，故p[k+1]=p[k]+2*dy

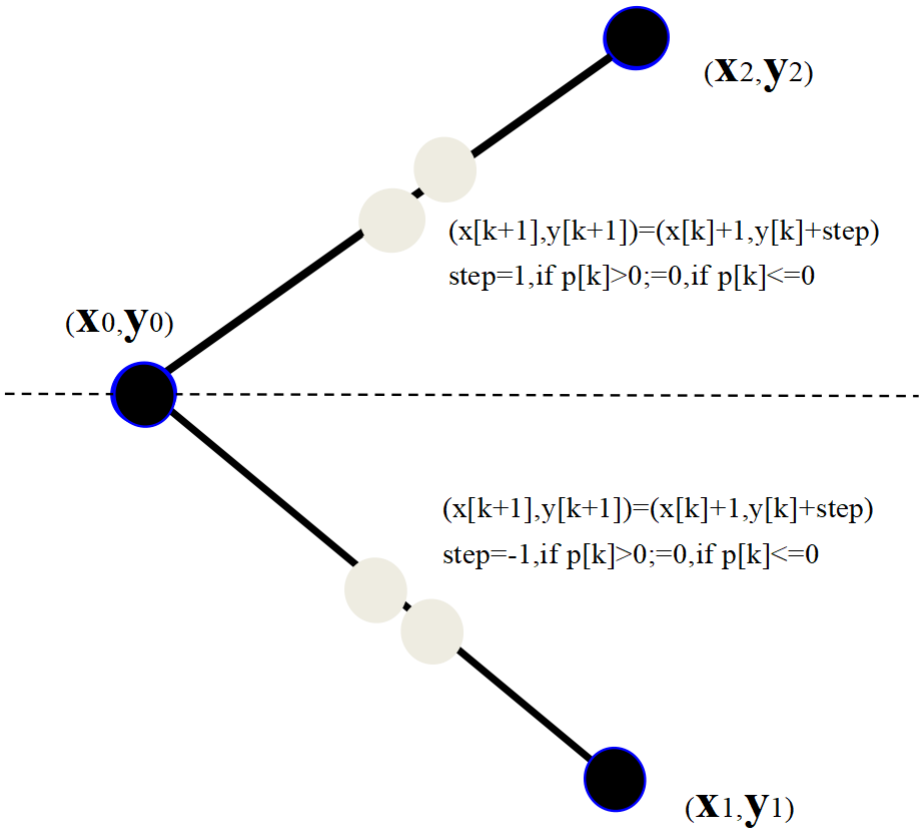
```
            step = int((y1-y0)/delta_y)
            step_1 = 2 * delta_y
            step_2 = 2 * delta_y - 2 * delta_x
            p = 2 * delta_y - delta_x
            result.append((x0, y0))
            cur_x = x0
            cur_y = y0
            for k in range(0,delta_x):
                if p < 0:
                    cur_x = cur_x + 1
                    result.append((cur_x,cur_y))
                    p = p + step_1
                else:
                    cur_x = cur_x + 1
                    cur_y = cur_y + step
                    result.append((cur_x,cur_y))
                    p = p + step_2
```

在具体实现的过程中，我们需要尽可能将其他情况转化为(x0,y0)在(x1,y1)的左侧，斜率0<m<1的情况，来利用上面推导出的公式。

首先考虑斜率m的绝对值小于1的情况:

若(x0,y0)在(x1,y1)的右侧，则把(x0,y0)和(x1,y1)调换位置，至此，(x1,y1)一定位于(x0,y0)的右上方或右下方。

```
            if x0 > x1:
                x0, y0, x1, y1 = x1, y1, x0, y0
```

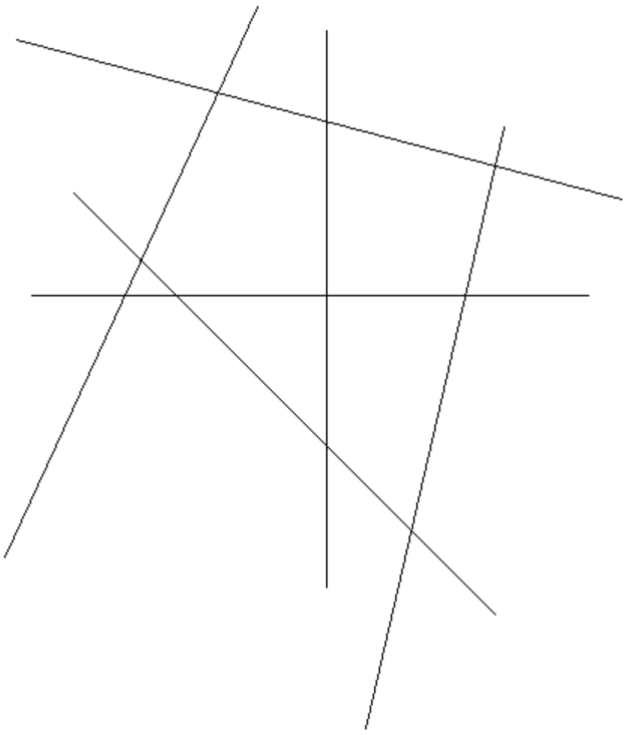


右上方的情况直接套用上面的公式即可。

右下方的情况下，可以计算(x1,y1)位于(x0,y0)右上方对称位置(x2,y2)时，需要的像素点，然后再对称到右下方即可。在代码实现中，也就是将step置为-1。

斜率m的绝对值大于等于1的情况下，将对x的处理和对y的处理互换即可。

测试结果如下：



2.2 绘制椭圆

这里运用的是中点圆生成算法，它的基本思路与Bresenham算法相似，都是判断候选像素点中离实际的点最近的，作为下一个像素点。其中，两个候选点与实际点距离的差，作为一个参数，可以找到规律进行计算。

首先，为了方便计算，将椭圆的中心平移到原点的位置，也就是x坐标统一减(x0+x1)/2，y坐标统一减(y0+y1)/2，求出此时椭圆的函数：(ry*x)**2+(rx*y)**2-(r1*r2)**2=0，其中，rx=(x1-x0)/2,ry=(y0-y1)/2

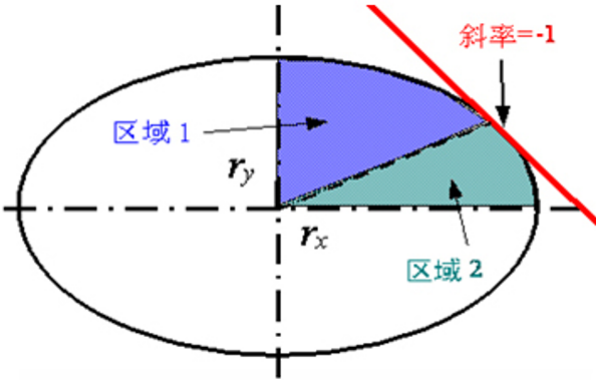
定义函数f(x,y)=(ry*x)**2+(rx*y)**2-(r1*r2)**2，对于任意点(x,y)：

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

为了方便计算，我们可以只画椭圆在第一象限的部分，再把它对称到其他三个象限即可。第一象限的部分又可以根据斜率与1的大小关系，分为两个区域：

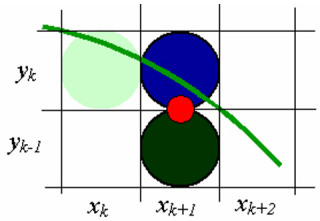


区域1为斜率大于-1的部分，要以x轴为单位取样。区域2为斜率小于-1的部分，要以y轴为单位取样。我们可以先算区域1的像素点，在斜率小于等于-1的时候，进入区域2计算。区域交替的条件为：

dy/dx=(-2*ry**2*x)/(2*rx**2*y)<=-1

即2*ry**2*x>=2*rx**2*y

在区域1中，若已经决定第k个像素点的坐标(x[k],y[k])，则第k+1个像素点的候选项有两个：(x[k]+1,y[k])和(x[k]+1, y[k]-1)，我们可以通过计算它们中点与椭圆的位置关系来判断应该选择哪个坐标。



将中点坐标代入椭圆函数中：p1[k]=f(x[k]+1,y[k]-0.5)=(ry*(x[k]+1))**2+(rx*(y[k]-0.5))**2-(r1*r2)**2

若p1[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p1[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p1[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

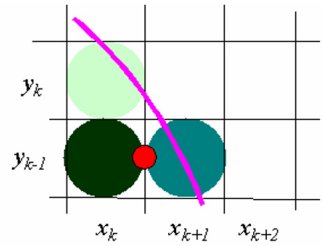
p1[k+1]=f(x[k+1]+1,y[k+1]-0.5)=(ry*(x[k+1]+1))**2+(rx*(y[k+1]-0.5))**2-(r1*r2)**2

=p[k]+2*ry**2*x[k+1]-2*rx**2*y[k+1]+ry**2

其中，p1的初始值为f(0+1,ry-0.5)=ry*ry-rx*rx*ry+rx*rx/4

```
p1 = ry*ry-rx*rx*ry+rx*rx/4
cur_x = 0
cur_y = ry
result.append((cur_x,cur_y))
while ry*ry*cur_x < rx*rx*cur_y:
    if p1 < 0:
        cur_x = cur_x + 1
        result.append((cur_x,cur_y))
        p1 = p1+2*ry*ry*cur_x+ry*ry
    else:
        cur_x = cur_x + 1
        cur_y = cur_y - 1
        result.append((cur_x,cur_y))
        p1 = p1+2*ry*ry*cur_x-2*rx*rx*cur_y+ry*ry
```

在区域2中，若已经决定第k个像素点的坐标(x[k],y[k])，则第k+1个像素点的候选项有两个：(x[k],y[k]-1)和(x[k]+1, y[k]-1)，我们可以通过计算它们中点与椭圆的位置关系来判断应该选择哪个坐标。



将中点坐标代入椭圆函数中：p2[k]=f(x[k]+0.5,y[k]-1)=(ry*(x[k]+0.5))**2+(rx*(y[k]-1))**2-(r1*r2)**2

若p2[k]<=0，表示中点位于椭圆的内部，应该选择中点右方的候选点(x[k]+1,y[k]-1)，此时

p2[k]=f(x[k]+0.5,y[k]-1)=(ry*(x[k+1]-0.5))**2+(rx*y[k+1])**2-(r1*r2)**2

p2[k+1]=f(x[k+1]+0.5,y[k+1]-1)=(ry*(x[k+1]+0.5))**2+(rx*(y[k]-1))**2-(r1*r2)**2

=p[k]+2*ry**2*x[k+1]-2*rx**2*y[k+1]+rx**2

若p2[k]>0，表示中点位于椭圆的外部，应该选择中点左方的候选点(x[k],y[k]-1)，此时

p2[k]=f(x[k]+0.5,y[k]-1)=(ry*(x[k+1]+0.5))**2+(rx*y[k+1])**2-(r1*r2)**2

p2[k+1]=f(x[k+1]+0.5,y[k+1]-1)=(ry*(x[k+1]+0.5))**2+(r*(y[k+1]-1))**2-(r1*r2)**2

=p[k]-2*rx**2*y[k+1]+rx**2

其中，p2的初始值为f(x[k]+0.5,y[k]-1)=(ry*(x[k]+0.5))**2+(rx*(y[k]-1))**2-(r1*r2)**2，(x[k],y[k])为进入区域2的第一个像素点。

```
    p2 = ry*ry*(cur_x+1/2)*(cur_x+1/2)+rx*rx*(cur_y-1)*(cur_y-1)-rx*rx*ry*ry
    while cur_y >= 0:
        if p2 > 0:
            cur_y = cur_y -1
            result.append((cur_x,cur_y))
            p2 = p2-2*rx*rx*cur_y+rx*rx
        else:
            cur_y = cur_y -1
            cur_x = cur_x +1
            result.append((cur_x,cur_y))
            p2 = p2+2*ry*ry*cur_x-2*rx*rx*cur_y+rx*rx
```

接下来将第一象限的像素点对称到第二象限，再将一二象限的像素点对称到三四象限，最后整体平移到原来的位置即可。

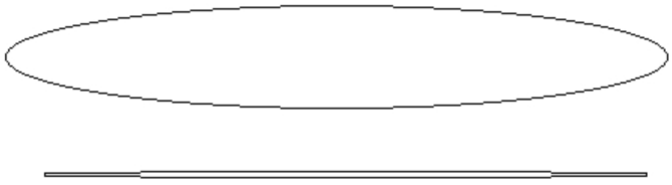
```
    for (x,y) in list(result):
        result.append((-x,y))
    for (x,y) in list(result):
        result.append((x,-y))
    for i in range(0,len(result)):
        result[i]=(round(result[i][0]+delta_x),round(result[i][1]+delta_y))
```

遇到的bug

在实现这个算法的过程中，因为计算p1初始值的时候，将p1 = ry*ry-rx*rx*ry+rx*rx/4中的-写成+，也就是p1 = ry*ry+rx*rx*ry+rx*rx*/4，这不仅导致了椭圆的弧度不正确，更是导致了进入区域2的时机不对，有的时候甚至没有进入区域2就直接结束了算法，如下图所示。

甚至在画的椭圆很扁的时候，只能返回如下图所示的一个小点。

改正了这个错误后，画出的椭圆明显正常了：



2.3 绘制多边形

多边形绘制的算法，只需按序调用相应的直线绘制算法即可。

重要的是修改cg_gui.py文件，使得我们能够在图形化界面绘制多边形。

首先需要规定绘制多边形的方法，这里规定左键单击确定多边形的顶点，右键单击表示绘制完成。

```
            if self.temp_item is None:
                self.start_draw()
                self.temp_item = MyItem(self.temp_id, self.status, [[x, y], [x,
y]], self.temp_algorithm, self.pen_color)
                self.scene().addItem(self.temp_item)
            else:
                if event.button() == Qt.RightButton:
                    self.finish_draw()
                    self.temp_item = None
                else:
                    self.temp_item.p_list.append([x, y])
```

另外需要注意的是boundingRect函数中的处理，我们要获取的是整个图形用矩形框住后的左下角的坐标和宽、高，因此需要进行循环，将x设置为p_list中x的最小值，y设置为p_list中y的最小值，w为x的最大值减去最小值，h为y的最大值减去最小值。其余处理同直线的情况。

```
            x, y = self.p_list[0]
            w, h = self.p_list[0]
            for [x0,y0] in self.p_list:
                if x0 < x:
                    x = x0
                if y0 < y:
                    y = y0
                if x0 > w:
                    w = x0
                if y0 > h:
                    h = y0
            w = w - x
            h = h - y
            return QRectF(x - 1, y - 1, w + 2, h + 2)
```

2.4 平移变换

平移的算法很简单，只需要将p_list中所有的坐标都加上平移量即可

```
def translate(p_list, dx, dy):
    for i in range(0,len(p_list)):
        p_list[i]=[p_list[i][0]+dx,p_list[i][1]+dy]
    return p_list
```

如果已经选中了已有图形，并且在平移状态进行拖动，则鼠标按下时要将目前操作的id和图形读出来记下，并且记下当前鼠标的位置

```
        elif self.status == 'translate' and self.selected_id in self.item_dict:
            self.temp_id = self.selected_id
            self.temp_item = self.item_dict[self.selected_id]
            self.pre_x, self.pre_y = x, y
```

在鼠标拖动的过程中，要随时改变图形在屏幕上的位置，并记录当前的鼠标的位置

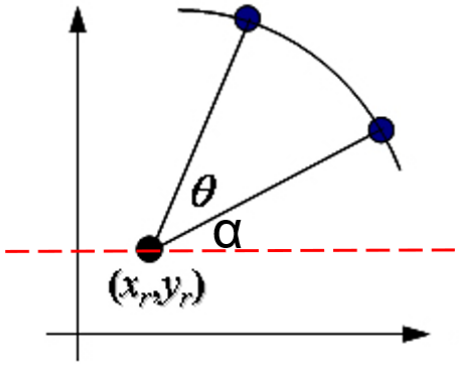
```
        elif self.status == 'translate':
            self.temp_item.item_translate(x-self.pre_x,y-self.pre_y)
            self.pre_x,self.pre_y = x, y
```

在MyItem类中定义函数item_translate来对图形做平移。

```
    def item_translate(self, dx, dy):
        self.p_list = alg.translate(self.p_list, dx, dy)
```

2.5 旋转变换

如下图所示，要把坐标为(x,y)的点，以(xr,yr)为旋转中心，顺时针旋转θ度，得到(x1,y1)



设(x,y)与旋转中心的距离为r，旋转后的(x1,y1)与x轴的夹角为α

则x-xr=rcos(θ+α), y-yr=rsin(θ+α)

x1-xr=rcosα=rcos(θ+α-θ)=r[cos(θ+α)cosθ+sin(θ+α)sinθ]=(x-xr)cosθ+(y-yr)sinθ

y1-yr=rsinα=rsin(θ+α-θ)=r[sin(θ+α)cosθ-cos(θ+α)sinθ]=-(x-xr)sinθ+(y-yr)cosθ

故推出公式：x1=xr+(x-xr)cosθ-(y-yr)sinθ

y1=yr+(x-xr)sinθ+(y-yr)cosθ

具体实现时，要先把角度转化为弧度，然后才能运用公式进行计算，每次计算出的结果要变成整数，否则会影响后续使用。

```
    angle = r * math.pi / 180
    for i in range(0,len(p_list)):
        x, y = p_list[i]
        x1 = xr + (x-xr) * math.cos(angle) + (y-yr) * math.sin(angle)
        y1 = yr - (x-xr) * math.sin(angle) + (y-yr) * math.cos(angle)
        p_list[i] = [x1, y1]
    return p_list
```

如果已经选中了已有图形，并且在旋转状态进行操作，则鼠标按下时要将目前操作的id和图形读出来记下，并且记下当前鼠标的位置

```
        elif self.status == 'rotate' and self.selected_id in self.item_dict and
self.item_dict[self.selected_id].item_type != 'ellipse':
            self.temp_id = self.selected_id
            self.temp_item = self.item_dict[self.selected_id]
            self.pre_x, self.pre_y = x, y
```

在鼠标拖动的过程中，要随时改变图形在屏幕上的位置，并记录当前的鼠标的位置

```
        elif self.status == 'rotate':
            self.temp_item.item_rotate(x,y,self.pre_x,self.pre_y)
            self.pre_x,self.pre_y = x, y
```

在MyItem类中定义函数item_rotate来对图形做平移。

```
    def item_rotate(self,x,y,pre_x,pre_y):
        min_x, min_y = self.p_list[0]
        max_x, max_y = self.p_list[0]
        for [x0,y0] in self.p_list:
            if x0 < min_x:
                min_x = x0
            if y0 < min_y:
                min_y = y0
            if x0 > max_x:
                max_x = x0
            if y0 > max_y:
                max_y = y0
        center_x=(max_x+min_x)/2
        center_y=(max_y+min_y)/2
        cur_r=math.atan2(y-center_y,x-center_x)/math.pi*180
        pre_r=math.atan2(pre_y-center_y,pre_x-center_x)/math.pi*180
        r=cur_r-pre_r
        self.p_list=alg.rotate(self.p_list,center_x,center_y,r)
```

遇到的问题

图形多次旋转后变形严重，经过多处测试，发现是由于每次旋转后都会对结果进行取整操作，因此旋转的次数越多，图形的相对关系越差。于是取消对结果的取整操作，使得p_list中保存的结果可以是浮点数，在绘制图形时再进行取整。

2.6 缩放变换

缩放倍数为s，若缩放中心为(0,0)，则`x2=x1*s`，`y2=y1*s`。

当缩放中心为(x,y)时，可以先将图形整体平移，将(x,y)平移到原点的位置，这时(x1,y1)的坐标变为(x1-x0,y1-y0)，再将它按缩放中心为(0,0)时缩放，得到结果((x1-x0)*s,(y1-y0)*s)，最后平移回(x,y)，得到结果(x+(x1-x0)*s,y+(y1-y0)*s).

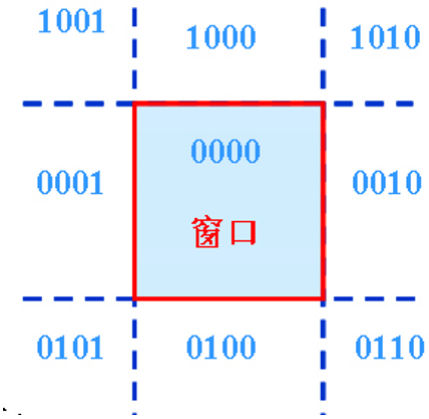
```
    for i in range(0,len(p_list)):
        x1=x+(p_list[i][0]-x)*s
        y1=y+(p_list[i][1]-y)*s
        p_list[i]=[round(x1),round(y1)]
    return p_list
```

2.7 线段裁剪

2.7.1 Cohen-Sutherland算法

这一算法的基本思路是先将窗口和窗口外的空间划分为9部分，并编码，然后分情况讨论，分别将每个窗口外区域的部分剪掉。

首先对空间进行分块，若在左边界左边，则第一位为1；在右边界右边，则第二位为1；在上边界的上边，则第三位为1；在下边界的下边，则第四位为1。具体的编码分布如下：



接下来编写一个函数，来判断点(x,y)对应的编码。初始值设置为0，若某一位需要设置为1，则与该位为1，其余位为0的数进行或操作。

```
def get_code(x,y,x_min,y_min,x_max,y_max):
    res=0
    if x < x_min:
        res=res|0b0001
    if x > x_max:
        res=res|0b0010
    if y < y_min:
        res=res|0b0100
    if y > y_max:
        res=res|0b1000
    return res
```

在具体实现时，首先分别获取(x0,y0)和(x1,y1)的编码，然后进行循环，只有当线段的两个端点的编码都变为0时，就代表裁剪完成，可以返回当前两个端点的坐标了。

若两个端点的编码的交操作结果不是0，代表两个端点都在窗口外部的同一端，线段在窗口外面，直接返回空值即可。

其余情况下，我们将一个端点收缩到一个裁剪边界上，并且将该端点在该边界上的编码变为0。在算直线与裁剪边界的交点时，我用的是点斜式：`y=k1(x-x0)+y0`，`x=k2(y-y0)+x0`，其中k1为(y1-y0)/(x1-x0)，k2为(x1-x0)/(y1-y0)，直接代入坐标即可算出交点坐标。

```
        code0=get_code(x0,y0,x_min,y_min,x_max,y_max)
        code1=get_code(x1,y1,x_min,y_min,x_max,y_max)
        k1,k2=0,0
        if x1 != x0:
            k1=(y1-y0)/(x1-x0)
        if y1 != y0:
            k2=(x1-x0)/(y1-y0)
        while True:
            if code0==0 and code1==0:
```

```
                return [[round(x0),round(y0)],[round(x1),round(y1)]]
            elif code0&code1!=0:
                return [[0,0],[0,0]]
            elif code0&0b0001==0b0001:  #剪[x0,y0]在框左边的部分
                y0=k1*(x_min-x0)+y0
                x0=x_min
                code0=code0&0b1110
            elif code1&0b0001==0b0001:  #剪[x1,y1]在框左边的部分
                y1=k1*(x_min-x1)+y1
                x1=x_min
                code1=code1&0b1110
            elif code0&0b0010==0b0010:  #剪[x0,y0]在框右边的部分
                y0=k1*(x_max-x0)+y0
                x0=x_max
                code0=code0&0b1101
            elif code1&0b0010==0b0010:  #剪[x1,y1]在框右边的部分
                y1=k1*(x_max-x1)+y1
                x1=x_max
                code1=code1&0b1101
            elif code0&0b0100==0b0100:  #剪[x0,y0]在框下边的部分
                x0=k2*(y_min-y0)+x0
                y0=y_min
                code0=code0&0b1011
            elif code1&0b0100==0b0100:  #剪[x1,y1]在框下边的部分
                x1=k2*(y_min-y1)+x1
                y1=y_min
                code1=code1&0b1011
            elif code0&0b1000==0b1000:  #剪[x0,y0]在框上边的部分
                x0=k2*(y_max-y0)+x0
                y0=y_max
                code0=code0&0b0111
            elif code1&0b1000==0b1000:  #剪[x1,y1]在框上边的部分
                x1=k2*(y_max-y1)+x1
                y1=y_max
                code1=code1&0b0111
```

2.7.2 Liang-Barsky算法

这一算法的基本思想是把直线上的点表示为`x=x0+t*(x1-x0)`，`y=y0+t*(y1-y0)`

对于上下左右四条窗口的边，分别算出两个变量q和d。q/d就表示为该边与直线交点的参数t，d的绝对值为该方向上交点与(x0,y0)的距离，q的绝对值为该方向上(x0,y0)与(x1,y1)的距离。其中，q为正数代表该点为直线与始边的交点，为负数代表该点为直线与终边的交点。

在计算时，只需要上下左右分别截取一次即可。每次先看q的符号，也就是看这个交点是直线与始边的交点还是与终边的交点。

若大于0，则是与始边的交点，首先计算该交点的参数`r = d[i]/q[i]`，若它大于t1，则表示两个点都在直线之外，直接返回[[0,0],[0,0]]。若它大于t0，则表示线段与直线有交点，需要截取到交点位置，也就是将t0置为r。

若小于0，则是与终边的交点，首先计算该交点的参数`r = d[i]/q[i]`，若它小于t0，则表示两个点都在直线之外，直接返回[[0,0],[0,0]]。若它小于t1，则表示线段与直线有交点，需要截取到交点位置，也就是将t1置为r。

最后，根据t0和t1算出两个新的端点，并返回。

```
        t0,t1=0.0,1.0
```

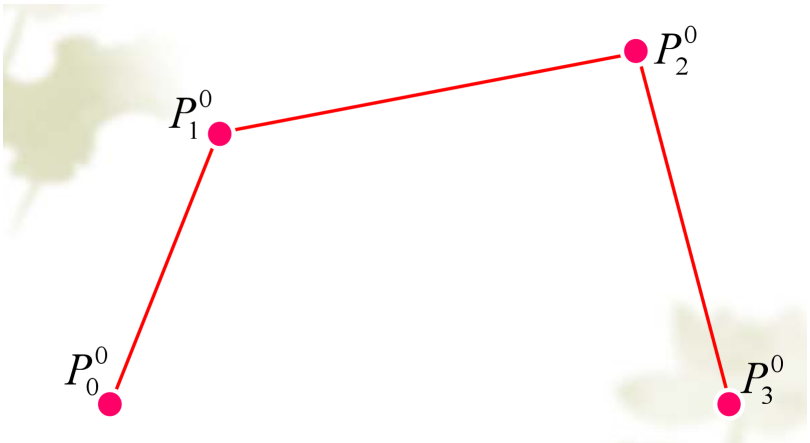
```
        deltax=x1-x0
        deltay=y1-y0
        q=[-deltax,deltax,-deltay,deltay]   #左，右，下，上
        d=[x0-x_min,x_max-x0,y0-y_min,y_max-y0]
        for i in range(0,4):
            if q[i] < 0:
                r = d[i]/q[i]
                if r > t1:
                    return [[0,0],[0,0]]
                elif r > t0:
                    t0 = r
            elif q[i] > 0:
                r = d[i]/q[i]
                if r < t0:
                    return [[0,0],[0,0]]
                elif r < t1:
                    t1 = r
            elif d[i] < 0:
                return [[0,0],[0,0]]
        x1=x0+t1*deltax
        y1=y0+t1*deltay
        x0=x0+t0*deltax
        y0=y0+t0*deltay
        return [[round(x0),round(y0)],[round(x1),round(y1)]]
```

2.8 绘制曲线

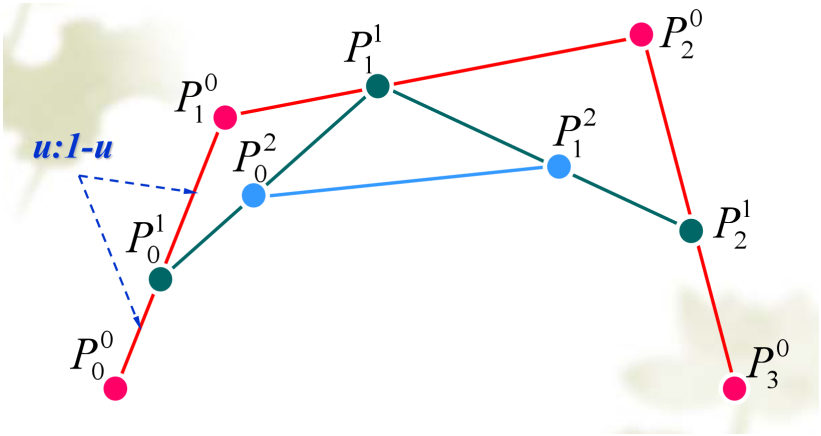
2.8.1 Bezier算法

本算法的目的是根据某一比例(1-u):u，用所有数据点生成一个该比例得到的点作为结果。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-u):u位置的新顶点，并将其添加进新序列中。对新序列重复这一操作，直到序列中只剩一个顶点，这个顶点即为所求，将其添加到结果中。

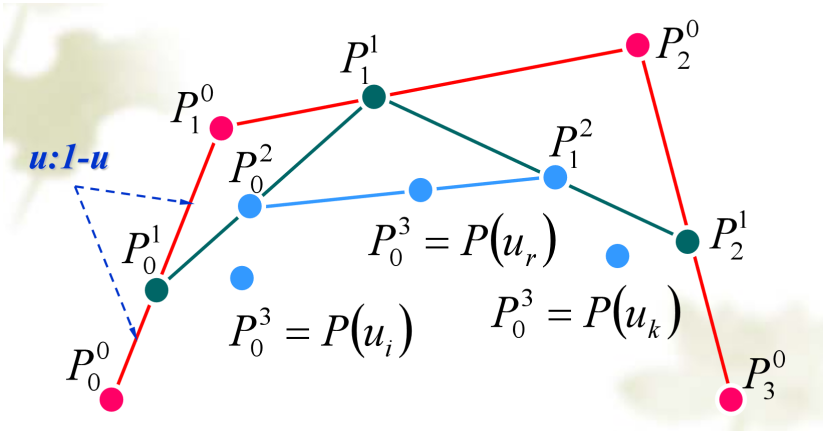
以四个数据点为例，起始时序列中有四个数据点，如下图所示。



第一次循环时，将每两个数据点以u:1-u的比例进行划分，得到的顶点加入新的序列中，得到上标为1的三个顶点。再将这三个顶点分别划分，得到上标为2的两个顶点。



最后将序列中的两个顶点按比例划分，就得到了唯一一个上标为3的顶点，这个顶点就是我们要求的。可以看到下图有三个上标为3的顶点，它们的计算方式相同，不同的就是每次计算时比例u的值。不断调整u的值，就可以得到所求曲线上所有点的坐标了。



为了实现这一算法，不断取不同的比例u，取du为0.001/n，u的初始值为du，不停的加du来取遍从0到1的所有值，对不同的u。根据所有数据点算出，对于这个u得到的顶点坐标，并将这个顶点加入结果的曲线中。如此反复，直到从0到1中取了足够多且均匀的u，将结果返回即可。

```
    result=[]
    n = len(p_list)
    if algorithm == 'Bezier':
        du=0.001/n
        result.append(p_list[0])
        u=du
        while u<1:
            temp=p_list.copy()
            for i in range(1,n):    #n-1次
                for j in range(0,n-i):
                    x=(1-u)*temp[j][0]+u*temp[j+1][0]
                    y=(1-u)*temp[j][1]+u*temp[j+1][1]
                    temp[j]=[x,y]
            result.append([round(temp[0][0]),round(temp[0][1])])
            u=u+du
        return result
```

2.8.2 B-spline算法

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。为了实现这一功能，定义该算法的递推公式如下：

$$B_{i,k+1}(u)=\frac{u-u_i}{u_{i+k}-u_i}B_{i,k}(u)+\frac{u_{i+k+1}-u}{u_{i+k+1}-u_{i+1}}B_{i+1,k}(u)$$

其中，$u\in[u_i,u_{i+1})$时，$B_{i,1}(u)=1$；

$u\in$其它时，$B_{i,1}(u)=0$。

在具体实现时，首先需要设置n+k+2个节点，节点的数值分别为0到n+k+1。在代码中没有直接体现，因为直接根据节点的下标就可以得到对应的数值。然后定义一层循环，从k到n均匀地取足够多的u的值进行计算。接着对于给定的u，用下述公式分别计算横纵坐标，其中B的相关值可以用上面提到的递推公式计算。每个u计算得到的结果都是曲线上的点，可以加入结果中返回。

$$P(u)=\sum_{i=0}^{n}P_iB_{i,k+1}(u),u\in[u_k,u_{n+1}]$$

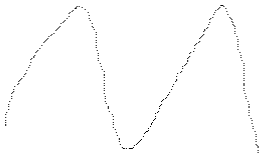
具体代码如下：

```
        result=[]
        n = len(p_list)
        dk=0.001/n
        k = 4
        for i in range(k-1,n):
            u = i+dk
            while u <= i+1:
                x = 0
                y = 0
                for j in range(i-k+1,i+1):
                    temp=B_spline(u,k,j)
                    x=x+p_list[j][0]*temp
                    y=y+p_list[j][1]*temp
                result.append([round(x),round(y)])
                u = u + dk
        return result
```

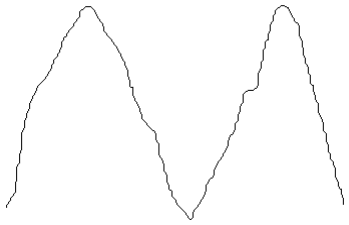
2.8 自由绘制

这一功能只需要记录鼠标移动过程中的所有点即可实现，也就是在mouseMoveEvent函数中，每次向temp_item. p_list加入当前鼠标所在的坐标。

需要注意的是，电脑对于坐标点的检测有一定的时间间隔，如果在绘制时只绘制保存下来的数据点，线条是不够清晰的，如下图所示：



因此在绘制时，需要调用绘制直线的函数，补齐每两个相邻数据点之间的部分，这样线条才能足够清晰。



3 系统框架

3.1 命令行界面（CLI）程序

3.1.1 框架代码理解

在代码的最外层，设置各个变量的初始值，包括item_dict, pen_color, width, height，然后打开input_file文件，也就是命令行界面的输入，进行逐行读取。对每行的输入，按空格分隔，然后分情况讨论。

如果读到的指令是resetCanvas，那么就需要清空画布，根据后面输入的两个参数，设置画布的宽和高。因为测试时所有指令都是合法的，所以不需要特殊判断，可以直接将输入的字符串转化为int型，然后存入width和height中，完成重新设置画布大小的功能。接着将item_dict设置为空字典，这样就可以清空画布了。

如果读到的指令是saveCanvas，那么就需要保存画布。先使用np.zeros函数生成一个空数组canvas，用来存储各个像素点的颜色值。然后调用fill函数将所有像素点的颜色都修改为[255, 255, 255]。接下来遍历字典item_dict里面的值，它们都是之前的指令中画过的内容，我们要把它们依次存入canvas数组中，这部分的框架代码只给出了线段的实现过程：调用我们在其他模块写好的draw_line函数，然后将返回的像素点数组对应的颜色修改为color。多边形、椭圆和曲线的保存需要我们自己实现。最后，调用save函数保存画布所有像素点的颜色值。

如果读到的指令是setColor，那么就需要设置画笔颜色，将后面输入的三个参数分别转化为int型，存入pen_color数组中即可。

如果读到的指令是drawLine，表示要绘制线段，我们要从输入中获取参数，将图元的编号、横纵坐标、绘制使用的算法、当前的画笔颜色，分别存入字典item_dict中。

给出的框架代码只实现了上述功能，我们还需要实现saveCanvas指令中除了线段之外的情况的数据处理，绘制多边形、椭圆、曲线，图元平移、旋转、缩放，对线段裁剪，这些命令的识别和实现。

3.1.2 saveCanvas指令的补充

与线段的处理类似，在类型为多边形、椭圆、曲线的情况下，我们也需要先调用cg_algorithms中对应的draw系列的函数，然后遍历返回的像素点数组，将carvas每个对应坐标处像素点的颜色修改为color。

3.1.3 draw系列命令

drawEllipse为绘制椭圆命令，因为算法只有一种，所以不需要读取算法，只要读取id和坐标，然后按顺序把它们和其他信息存入item_dict中。

drawPolygon和drawCurve指令，分别是绘制多边形和曲线，这两个指令与前面的指令不同，是不定长的，输入的坐标个数可以有多个。在具体实现的时候，需要将每个点的横纵坐标分别转化成int型存入一个列表中，再将这些列表统一存入一个列表，作为参数存入item_dict。我使用while循环来实现这一功能，每次将读取到的横纵坐标生成列表，使用append函数追加到总列表的尾部。

3.1.4 图元变换指令

translate指令需要平移图元，首先仍然需要读取输入的id和平移的横纵距离dx、dy，然后调用cg_algorithms中实现的translate函数来获取变换后的图元参数，并且用返回结果覆盖item_dict中原来的图元参数。

如果是rotate指令，则需要获取旋转后的坐标，先读取输入的id、x、y、r，然后将这些作为参数和p_list一起传递给rotate函数，接着把返回的图元参数存入item_dict中。

scale指令的实现与translate类似，不同的是多了一个s参数，它是float型的变量，需要作为最后一个参数传递给scale函数。

clip指令要裁剪线段，也是先获取输入，然后调用clip函数，将结果存入item_dict中。

3.2 用户交互界面（GUI）程序

3.2.1 框架代码理解

首先看主模块的代码，在cg_gui.py的最后面，总共只有5行，主要目标是创建一个MainWindow类的窗口，并且显示窗口进行交互。

接下来看主模块中调用的MainWindow类。首先看__init__函数，它首先使用QListWidget来记录已有的图元，并用于选择图元。然后设置画布和菜单栏。可以看出，菜单栏中的功能并不完整，还需要我们根据2.2中的功能进行补充，添加更多的菜单项。接下来连接信号和槽函数的相关代码也明显需要我们补充，框架代码只给出了线段的naive画法的实现，其他功能都需要我们自行书写函数并创建连接。接下来的框架代码设置了主窗口的布局。

在MainWindow中，调用了画布窗体类MyCanvas，并创建了对象canvas_widget，对窗口进行操作后，就会连接到对应的函数来对画布canvas_widget进行修改。MyCanvas类的init函数中，只给出了绘制线段的函数，其他操作的相关函数需要自行补充。为了存储选中的图元，设置变量selected_id用于存选中图元的id，并且在item_dict的项selected中，可以存储该图元是否已经被选中。对于选中图元的处理，有两个函数，clear_selection函数用于清空选择，也就是取消对selected_id代表的图元的选择；selection_changed函数可以先清空选择，再选择新的图元。对鼠标行为的处理分为三个函数，mousePressEvent、mouseMoveEvent和mouseReleaseEvent，它们分别在鼠标按键、移动和松开的时候进行处理，其中仍然只有线段的相关处理，其他功能需要添加。

MyItem是自定义图元类，里面存储了图元的id、类型、参数、绘制算法等信息，并且包含了函数来对图元进行处理。

3.2.2 框架代码修改

关于get_id()函数的调用，观察到在菜单栏中重新选择要绘制的图形后，图形的计数会多加一。观察框架代码发现，框架中在菜单栏中要绘制的图形被选择时，和在每个图形绘制结束时，会调用get_id()函数，这就导致了最后一个图形绘制结束后加的1和重新在菜单中选择后加的1重复。因此我对get_id()函数的调用位置进行了更改，变为统一在图形绘制开始时调用，也就是在mousePressEvent中检测到绘制新的图形时，将id加一。

3.2.3 MainWindow类补充

对比发现，菜单栏中缺少保存画布的功能，因此在文件菜单中添加该项。

接下来分别为各信号设置槽函数。

line_naive_act的槽函数line_naive_action已经给出，参照它可以完成绘制图形的相关槽函数，只需要修改下图所示红色方框中的内容即可。这里调用的函数，应该在MyCanvas类中加以定义。

```
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()
```

图元变换类的函数，也可以参照上图实现，更改红框内的内容，然后删去最后两行，因为不需要取消选择。

设置画笔对应的函数，需要先调用getColor函数来获取画笔的颜色，然后调用canvas_widget对象中的函数来设置画布上画笔的颜色。

save_canvas_action函数是保存画布的函数，它需要先调用getText函数来让用户输入保存的路径，然后调用函数来保存画布内容。

reset_canvas_action函数要重置画布，需要重置的变量包括item_cnt，list_widget，scene，canvas_widget。其中，item_cnt需要置零，list_widget和scene需要调用clear函数清空，canvas_widget则需要调用MyCanvas类中的函数实现重置功能。在实际使用时，发现如果选中图形后直接重置画布，会发生错误直接退出，这是因为清空list_widget时没有切断连接，故在清空前要切断list_widget的连接，清空后再重新连接。

3.2.4 MyCanvas类补充

为了设置画笔颜色，增加一个变量pen_color用于记录画笔颜色，并且新建一个函数set_pen来更改pen_color的值。

start系列函数包含绘制图形和图元变换两种，参考框架代码给出的start_draw_line，将status进行修改即可。

重置画布函数reset_canvas，只需要将各变量重新置为初始值即可。

对鼠标动作的处理，由于与图形的具体实现有关，因此在前面与图形绘制算法一起进行了说明。

4 结束语

命令行界面程序已经给出了框架代码，但是很多指令并没有实现，需要添加部分指令的识别，并且在合适的位置调用cg_algorithms.py中的函数来处理数据。

图形化界面虽然菜单栏已经完成，但是信号和槽函数的设置、相关功能的实现和对于算法的调用还没有全部实现，需要自行补充。

目前算法部分已经全部完成，需要补充的部分也已经补充完毕。

致谢:

在此,我要向计算机图形学的授课教师孙正兴老师和张岩老师，以及本课程的全体助教表示感谢.

References:

[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] Python Qt教程https://www.byhy.net/tut/py/gui/qt_01/