计算机图形学实验报告

报告时间	12月底
院系	计算机科学与技术系
学号	191220156
姓名	张桓
邮箱	2659428453@qq.com

项目进度

9月进度

- 配置相关环境，安装需要的`python`库
- 阅读`CG_demo\`的示例框架，理解代码逻辑以及后续工作内容
- 修改`cg_cli.py`文件的代码框架
- 预习部分绘图算法内容

10月进度

- 实现算法所有部分
  - 包括已学内容：两种线段绘制算法(DDA和$Bresenham$算法)、多边形绘制算法、椭圆绘制算法以及图形的平移算法
  - 预习并实现曲线绘制算法($Bezier$算法和$B-spline$算法)、缩放变换和线段裁剪算法(编码裁剪算法和$Liang–Barsky$算法)

11月进度

- 实现命令行部分并通过测试`input.txt`
- 基本实现图形界面的全部内容
- 对图形界面的易用性和鲁棒性进行改进

12月进度

- 添加一些新的功能：
  - 删除图元
  - 设置画笔粗细
  - 绘制填充多边形：扫描填充算法
  - 实现多边形裁剪：采用$Sutherland-Hodgman$算法

实验内容

配置环境

- 之前一直使用pycharm，主要添加了新的库pyQT5

理解框架代码

cg_algorithms.py

本文件中定义各种**绘制图形**和**图形变换**所需要的函数，包括绘制直线、多边形、曲线、椭圆，以及图形的平移变换、旋转变换、缩放和线段的裁剪。定义的所有函数供之后的`cg_cli.py`和`cg_gui.py`使用。

- 图形绘制

  绘制函数参数主要包括两部分内容：`p_list`以及`algorithm`，`p_list`一般为点(用一个二元列表表示)的坐标列表，来指出所绘图形需要的关键点坐标。`algorithm`一般为字符串类型，指明在绘制某些图形时所需要的算法，比如是`DDA`还是`Bresenham`。

  返回值`result`为绘制好的图元像素点的坐标列表。

- 图形变换

  变换函数的参数可分为三部分：`p_list`表示需要变换的图元的参数；`*argv`每个函数都不同，表示变换所需的一些参数，比如旋转中心、缩放中心等等；`algorithm`表示变换所使用的算法。

  返回值`result`为变换后图元像素点的坐标列表。

cg_cli.py

在命令行运行该文件需要接受两个参数，`input_file`和`output_dir`。运行时打开`input_file`循环读取**每一行指令**`line`并执行对应操作。指令有三种：控制指令、绘图指令、变换指令。

- 控制指令

  `line[0]`指示控制内容，有重置画布，保存画布和设置画笔颜色。设置画笔颜色`pen_color`是一个一维数组，有三个8位`int`元素代表RGB值，其实就代表了一个像素的颜色。

  而画布`canvas`则是一个三维数组，其原型为`([height, weight, 3], np.uint8)`，即为`height`×`weight`个3个8位`int`数字构成，也就是`height`×`weight`个RGB值。

- 绘图指令

  `line[0]`指示绘制操作，`line[1]`指示本次绘制的**图元编号**`item_id`，需要注意的是每个图元的**编号唯一**，因此可以作为`item_dict`字典的键，然后在对应值中记录**绘制该图元的详细信息。**

  > `item_dict`是用来存储画布上的**所有图元信息**的字典，键为`item_id`，值为[图元类型， 控制点坐标列表， 使用算法，颜色]。每执行一条绘制指令，图元会得到唯一的`item_id`并且在`item_dict`中记录该图元的详细信息。

- 变换指令

  `line[0]`指示变换操作，`line[1]`表示需要变换的图元编号，之后的参数为变换所需的参数，比如缩放中心、旋转中心等。

- 整体流程

  读取一条指令`line`后，如果是绘图指令则根据给定的图元编号`item_id`在`item_dict`中创建对应项，在值中记录该图元的详细信息，如果是变换指令则修改`item_dict`中`item_id`图元的对应信息。

重置画布时则清空`item_dict`。

保存画布(`saveCanvas`指令)时遍历`item_dict`中的所有项，根据每个值中记录的图元信息(图元类型、控制点坐标列表、使用算法、颜色)调用`cg_alogrithm.py`中定义的函数得到图元像素坐标列表，然后在画布的对应位置设置RGB即可，就绘制好了图像。

cg_gui.py

一共定义了三个类：`MyItem, MyCanvas, MainWindow`。

- `MainWindow`是自定义的窗口类，设置了GUI窗口的布局，以及菜单栏和状态栏和中心组件，包括画布等等。其中完成了部分菜单功能的**信号和槽函数的绑定**。

  > 比如当我们选择图元时，点击`list_widget`列表里的数字，数字内容发生改变就会产生信号，调用`MyCanvas`类里的`selection_changed`函数。还有点击线段-Native按钮就会调用`line_native_action`函数。

  窗口的右侧显示`list_widget`，记录已有图元的各个ID，通过点击ID即可选择不同的图元。

- `MyItem`是自定义的图元类，包含了图元ID、图元类型、图元参数、绘制算法、是否被选择等各种信息。并且**重载**了`paint()`函数来绘制该图元，当**场景更新**时会**自动调用**`paint()`。

  > 在`paint()`函数中使用的是cg_algorithm.py里定义的算法。

  `boundingRect()`函数返回被选择图元的矩形边框，方便标识。示例中只完成了线段的红色边框。

- `MyCanvas`是自定义的画布类，当点击窗口菜单栏的绘制算法时，`MainWindow`类里对应的槽函数会调用`MyCanvas`类里的某个函数，比如示例中的`line_native_action`调用了`start_draw_line`函数，这个中间函数确定了新图元的ID以及类型，所需算法等等。

  根据这些信息我们按下鼠标`MyCanvas`类中的`mousePressEvent`函数会被调用，创建了新的`item`并且**添加到场景中**，自动调用`paint`函数绘制出该线段；鼠标移动则调用`mouseMoveEvent`函数，修改场景中该线段的终点，并用`updateSence`函数来实时绘制线段；松开鼠标调用`mouseReleaseEvent`，将该图元添加到`item_dict`里并将其ID添加到`list_widget`里。

【参考文章】：

- pyQT doc
- [QGraphicsItem: when paint function is called]
- Qt绘图QGraphicsView、QGraphicsScene、QGraphicsItem简述

框架调整

封装函数

在`cg_cli.py`文件中，示例代码匹配命令行`line[0]`的方法时直接在if语句下编写了对应所有代码，封装性差不方便维护，自然而然可以将每个if语句块的代码抽象成一个函数，采用小写字母加下划线的方式定义函数体，如：

```
def reset_canvas(line):
    global width, height, item_dict
    width = int(line[1])
    height = int(line[2])
    item_dict = {}
def set_color(line):
    global pen_color
    pen_color[0] = int(line[1])
    pen_color[1] = int(line[2])
    pen_color[2] = int(line[3])
...
```

这样做的代价必须将`item_dict, pen_color, width, height`等变量设置为全局变量，要使各个函数可以修改全局变量，函数中使用全局变量时要增加`global`引用。

于是匹配`line[0]`时的代码就会简洁很多：

```
with open(input_file, 'r') as fp:
    line = fp.readline()
    while line:
        line = line.strip().split(' ')
        if line [0] == 'resetCanvas':
            reset_canvas ( line )
        elif line [0] == 'saveCanvas':
            save_canvas ( line )
        ...
        line = fp.readline()
```

【参考文章】：

- Python函数内修改全局变量

使用字典匹配函数

但即便做了这样的调整，匹配`line[0]`时仍用了大量的if-else语句，非常繁琐。python中没有switch这样的语句，也没有函数指针的概念，但是各种基本类的抽象性非常高。考虑用字典来做化简，设置字典`func_dict`，字典的项为`<方法名称: 对应函数>`，比如`<'resetCanvas : reset_canvas>'`，在匹配时直接用方法名称索引即可得到对应函数。字典设置如下：

```
func_dict = {
    'resetCanvas': reset_canvas,
    'saveCanvas': save_canvas,
    'setColor': set_color,
    'drawLine': draw_line,
    'drawPolygon': draw_polygon,
    ...
}
```

这样匹配时的代码会变得非常简洁，甚至可以添加一些**边界处理**，比如出现非法的命令时提示命令不存在，增加代码的鲁棒性，代码如下：

```
with open(input_file, 'r') as fp:
    line = fp.readline()
    while line:
        line = line.strip().split(' ')
        try:
            func_dict[line[0]](line)
        except KeyError:
            print(f'[ERROR] :You call the nonexistent func: {line[0]}!!!')
            exit()
        line = fp.readline()
```

算法部分

绘制线段

DDA算法

- 算法原理

  DDA算法是利用计算两个坐标方向的差分来确定线段显示的屏幕像素位置的线段扫描转换算法。算法过程如下：

  已知直线的起点和终点$(x0, y0), (x1, y1)$。计算出水平差和垂直差$\Delta x = x1 - x0, \Delta y = y1 - y0$可以确定直线的斜率$k = \frac{\Delta y}{\Delta x}$。

  - 如果$|k| < 1$即$\Delta x > \Delta y$，那么在x轴上以单位间隔$dx = 1$取样，计算从x_0到x_1(若$x_0 > x_1$则交换起点和终点)之间每个点的纵坐标y_i:

    $$dy = k$$
    $$y_{i+1} = y_i + k$$

    由于要求的是像素坐标，计算出的y值需要取整。

  - 如果$|k| > 1$即$\Delta x < \Delta y$，那么同理在y轴上以单位间隔$dy = 1$取样，计算从y_0到y_1(若$y_0 > y_1$则交换起点和终点)之间每个点的横坐标x_i:

    $$dx = \frac{1}{k}$$
    $$x_{i+1} = x_i + \frac{1}{k}$$

    计算出的x值需要取整。

- 代码实现：

```
elif algorithm == 'DDA':
    if x0 == x1:
        for y in range(min(y0, y1), max(y0, y1) + 1):
            result.append([x0, y])
    elif y0 == y1:
        for x in range(min(x0, x1), max(x0, x1) + 1):
            result.append([x, y0])
    else:
        k = (y1 - y0) / (x1 - x0)
        if abs(k) <= 1:
            if x0 > x1:  # make sure x is adding
                x0, y0, x1, y1 = x1, y1, x0, y0
            yi = y0
            for x in range(x0, x1 + 1):
                result.append([x, round(yi)])
                yi += k
        else:
            if y0 > y1: # make sure y is adding
                x0, y0, x1, y1 = x1, y1, x0, y0
            xi = x0
            for y in range(y0, y1 + 1):
```

```
22                  result.append([xi, round(y)])
23                  xi += 1/k
```

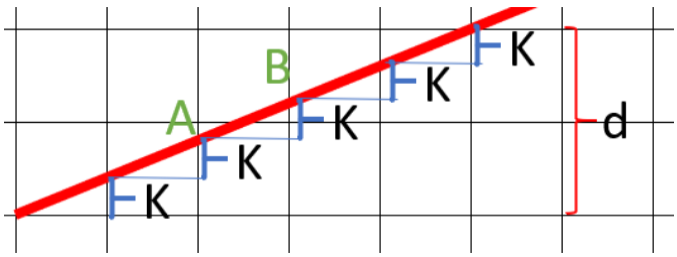
> $Q\&A$:
>
> - 为什么要求$\Delta x > \Delta y$时要在x轴方向以单位间隔取样，而在$\Delta x < \Delta y$时则在y轴方向取?
>
> 这是因为$\Delta x > \Delta y$时在x轴上单位取样可以比在y轴上单位取样取到更多的点，使得绘制的直线更加准确。考虑一种极端情况，比如$\Delta y == 0$直线平行于x轴，这时候如果在y轴方向取样，那么只能取到一个点，这显然不合理。
>
> 而$\Delta x < \Delta y$时在y轴方向取样可以同理解释。
>
> - DDA算法的优缺点有哪些?
>
> DDA算法计算像素位置要比直接用直线方程计算快，这是因为它利用光栅特性用递增的办法消除了直线方程中的乘法；但是浮点增量的连续跌价中取整误差的积累会使长线段所计算的像素位置偏离实际线段，而且取整和浮点运算仍然十分耗时。

Bresenham算法

- 算法原理

  Bresenham算法通过引入整型参量定义来衡量两候选像素与实际点在某方向上的相对偏移，并利用对整型参量符号的检测来确定最接近实际线段的像素。书上的推导已经非常详尽，这里描述一下我自己的理解。

  假设直线的起点和终点$(x0, y0), (x1, y1)$其中$x_0 < x_1, y_0 < y_1$。计算出水平差和垂直差$\Delta x = |x1 - x0|, \Delta y = |y1 - y0|$且$\Delta x > \Delta y$。这时在$x$轴方向上单位取样，当$x$每增加1时，$y$方向都会增加$k = \frac{\Delta y}{\Delta x} (0 < k < 1)$，我们记这种阶梯量为$d$。这几个变量间的关系如下图：



  当x增加1，y方向上相应的增加k值，d也就累加相应的k值：

  - 当$(d > 0.5)$时直线更靠近上者，即y上移取点$(x_k + 1, y_k + 1)$，上移同时需要对d减去一个1，这样可以将像素分布在实际线段两侧，避免了绘制的偏离。
  - 当$(d < 0.5)$时直线更靠近下者，即y不变取点$(x_k + 1, y_k)$，d继续使用不断积累直到$(d > 0.5)$。比如以起点(x_0, y_0)为例，如果$d = \frac{\Delta y}{\Delta x} > 0.5$，那么下一个点取$(x_1, y_1)$，否则取$(x_1, y_0)$。

  也就是说我们需要不断判断$d += \frac{\Delta y}{\Delta x} > 0.5$，如果成立则取候选点的上者，并将$d -= 1$，否则取候选点的下者。为了避免浮点运算，对式子进行处理得：$(2\Delta x d += 2\Delta y) - \Delta x > 0$，初始时$d = 0$，即第一个决策参数为$p_0 = 2\Delta y - \Delta x$，之后根据上述内容：

- 若$p_k > 0$，则取点(x_{k+1}, y_{k+1})，并且由于阶梯量d需要减1，因此增量为$(2\Delta y - 2\Delta x)$，下一步$p_{k+1} = p_k + 2\Delta y - 2\Delta x$
- 若$p_k < 0$，则取点(x_{k+1}, y_k)，这时阶梯量d可以继续使用，因此增量为$2\Delta y$，下一步$p_{k+1} = p_k + 2\Delta y$

这样通过另一种思路得到了和教材上的推导殊途同归的效果。

- 一些问题
  - 直线斜率为负数怎么办？斜率绝对值大于一怎么办？
    - 斜率绝对值大于一只需要将x轴和y轴坐标互换再计算，得到结果后将每个像素点的两个**坐标互换**后再返回;
    - 斜率为负数的话，决策参数还可以使用，只是候选点变成了(x_{k+1}, y_k)和(x_{k+1}, y_{k-1})，当$p_k > 0$时取(x_{k+1}, y_{k-1})，即y的坐标越来越小；当$p_k < 0$时取(x_{k+1}, y_k)。

- 关键代码部分实现

```
dy, dx, ex = abs(y1 - y0), abs(x1 - x0), 0
if dy > dx:  # the |k| > 1, need exchange x, y
    dx, dy = dy, dx
    ex = 1
    x0, y0, x1, y1 = y0, x0, y1, x1
if x0 > x1:  # make sure x is adding
    x0, y0, x1, y1 = x1, y1, x0, y0
s = (1 if y0 < y1 else -1)
p = 2 * dy - dx
y = y0
for x in range(x0, x1 + 1):
    if ex == 1: # exchange x, y
        result.append([y, x])
    else:
        result.append([x, y])
    if p > 0:
        p = p + 2 * dy - 2 * dx
        y += s
    else:
        p = p + 2 * dy
```

> $Q\&A$：
>
> - 当直线斜率恰好为$\frac{1}{2}$时取候选点中的哪一个？
>
> 斜率恰好等于$\frac{1}{2}$时，没有确定的候选标准，本算法将光栅点选在(x_{k+1}, y_k)，(根据《计算机图形学的算法基础》50P)
>
> - Bresenham如何减少浮点运算？
>
> 因为Bresenham是用决策参数的正负来决定取两个候选像素的哪一个的，只关注决策参数正负号，而不像DDA算法那样，需要不断求出精确值并且四舍五入；因此在$d += \frac{\Delta y}{\Delta x} > 0.5$时通过两边同乘以$2\Delta x$使得运算全部转换成整数运算。

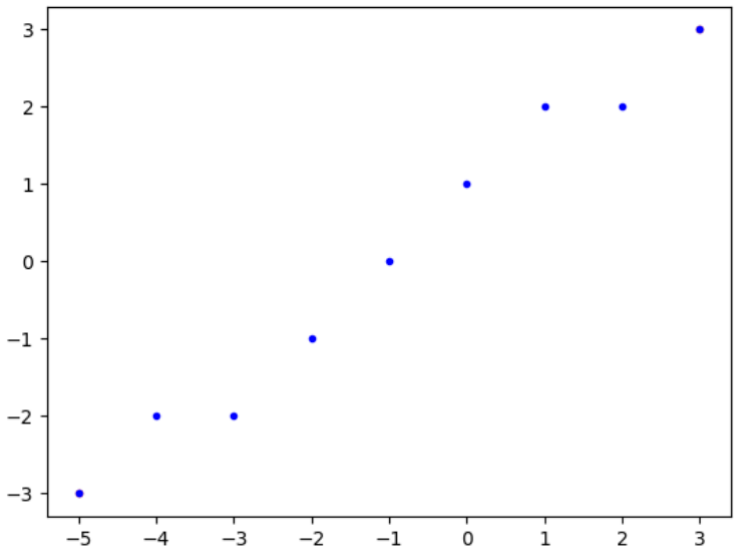
验证

通过`matplotlib.pyplot`库绘制直线， 观察算法给出的像素点位置情况，：

```
p_list = [[-5, -3], [3, 3]]
for i in range(len(p_list)): # 控制点为红色
    plt.plot(*p_list[i], 'r.')

res = draw_line(p_list, 'DDA')
# res = draw_line(p_list, 'Bresenham')
x = [p[0] for p in res]
y = [p[1] for p in res]
plt.plot(x, y, 'b.') # 线段点为蓝色
plt.show()
```

两种算法的结果均如下图所示：



绘制多边形

绘制多边形的操作就是根据多边形的顶点，按照一定的方向依次取相邻两个顶点在其之间绘制直线即可，这要求传入的顶点参数们满足相邻的顺序。但是算法本身非常简单，只需要不断调用上面实现的`draw_line`函数即可，代码如下：

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

绘制椭圆

算法原理

中点椭圆算法利用了椭圆的对称性，只计算一个象限的点再对称得到其余三个像素的点。考虑到中心在原点，轴对齐的椭圆方程：

$$\frac{x^2}{a^2}+\frac{y^2}{b^2}=1$$

进行适当变换可以得到方程：

$$f(x,y)=b^2x^2+a^2y^2-a^2b^2=0$$

这个函数相当于决策函数，因为对于一个点(a,b)，我们有

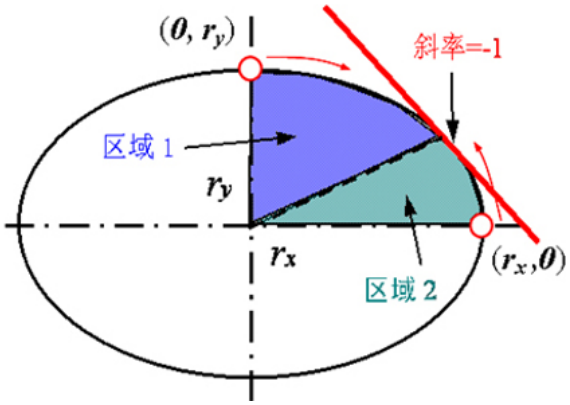
$$f(a,b)\begin{cases}=0,\text{该点在椭圆边界上}\\>0,\text{该点在椭圆边界外}\\<0,\text{该点在椭圆边界内}\end{cases}$$

之后将根据这个函数的正负性在两个候选点之间做出选择。

求出椭圆上的切线斜率:

$$k=\frac{dy}{dx}=-\frac{b^2x}{a^2y}$$

而斜率为-1的切线点将椭圆分成两部分，如图：



在区域1中，椭圆切线斜率$|k|<1$，选择在x轴方向上以单位间隔取样；在区域2中，切线斜率$|k|>1$，选择y轴方向上以单位间隔取样，至于原因已经在DDA算法中解释过了。

接下来就是从椭圆最上方的点开始顺时针计算第一象限的曲线上的像素坐标，每求出一个像素坐标也就求出了其余三个象限上的坐标。区域1和区域2上的点需要按照不同的方法：

- 区域1中($|k|<1$)

  由于这部分切线斜率$|k|<=1$，因此如果前一步选择了(x_k,y_k)，候选点有$(x_k+1,y_k),(x_k+1,y_k-1)$，我们取**两个候选点的中点**对决策参数(即椭圆函数求值):

  $$p1_k=f(x_k+1,y_k+\frac{1}{2})=b^2(x_k+1)^2+a^2(y_k-\frac{1}{2})^2-a^2b^2$$

  - 若$p1_k<=0$，中点位于椭圆内，因此椭圆曲线更靠近候选点(x_k+1,y_k)，因此下一步决策中候选点变成了(x_k+2,y_k)和(x_k+2,y_k-1)，中点为$(x_k+2,y_k-\frac{1}{2})$，决策参数为:

$$p1_{k+1} = f(x_k + 2, y_k - \frac{1}{2}) = p1_k + 2b^2x_k + 3b^2$$

- 若$p1_k > 0$，中点位于椭圆外，实际曲线更靠近候选点$(x_k + 1, y_k - 1)$，因此下一步决策中候选点为$(x_k + 2, y_k - 1)$和$(x_k + 2, y_k - 2)$，中点为$(x_k + 2, y_k - \frac{3}{2})$，决策参数为:

$$p1_{k+1} = f(x_k + 2, y_k - \frac{1}{2}) = p1_k + 2b^2x_k - 2a^2y_k + 2a^2 + 3b^2$$

求出初始值$p1_0$:

$$p1_0 = f(1, b - \frac{1}{2}) = b^2 - a^2b + \frac{a^2}{4}$$

- 区域2中($|k| > 1$)

这部分在$-y$方向上以单位步长取样，同区域1一样每一步中取两个候选点的中点对决策参数求值:

$$p2_k = f(x_k + \frac{1}{2}, y_k - 1) = b^2(x_k + \frac{1}{2})^2 + a^2(y_k - 1)^2 - a^2b^2$$

若$p2_k > 0$则中点位于椭圆之外，选择像素点$(x_k, y_k - 1)$，否则中点位于椭圆内，选择像素点$(x_k + 1, y_k - 1)$，下一步的决策值同区域1计算方法:

$$p2_{k+1} = \begin{cases} p2_k - 2a^2y_k + 3a^2 & ,\text{p2k} >= 0 \\ p2_k + 2b^2x_k - 2a^2y_k + 2b^2 + 3a^2 & ,\text{p2k} < 0 \end{cases}$$

根据区域1中最后点作为区域2的起始点(x_0, y_0)求初始值$p2_0$:

$$p2_0 = b^2(x_0 + \frac{1}{2})^2 + a^2(y_0 - 1)^2 - a^2b^2$$

以上分析都是在椭圆**中心位于原点**考虑的，如果椭圆中心不在原点，只需要按照原点计算后再加上**偏移向量**即可。

代码实现

函数原型中参数为矩形包围框左上角和右下角的坐标`p_list=[[x0, y0], [x1, y1]]`，因此先按照这两个点计算出椭圆的`a, b`参数以及中心偏移向量`(xc, yc)`，后续代码按上述内容即可:

```python
def draw_ellipse(p_list):
    result = []
    x0, y0 = p_list[0]
    x1, y1 = p_list[1]
    a, b = abs(x1 - x0) / 2, abs(y0 - y1) / 2
    x, y = 0, b
    xc, yc = (x0 + x1) / 2, (y0 + y1) / 2 # vector need to be added

    p1k = b*b*(x+1)**2 + a*a*(y-1/2)**2 - a*a*b*b
    while b*b*x <= a*a*y:  # start in region 1
        if xc+x and yc+y:
            result += [[xc+x, yc+y], [xc+x, yc-y],
                       [xc-x, yc+y], [xc-x, yc-y]]
        else:
```

```
            result += [[xc+x, yc+y]]
        if p1k >= 0:
            p1k += 2*b*b*x - 2*a*a* y + 2*a*a + 3*b*b
            x, y = x + 1, y - 1
        else:
            p1k += 2*b*b*x + 3*b*b
            x, y = x + 1, y

    p2k = b*b*(x+1/2)**2 + a*a*(y-1)**2 - a*a*b*b
    while y >= 0:  # start in region 2
        if xc+x and yc+y:
            result += [[xc+x, yc+y], [xc+x, yc-y],
                       [xc-x, yc+y], [xc-x, yc-y]]
        else:
            result += [[xc+x, yc+y]]
        if p2k >= 0:
            p2k += -2*a*a*y + 3*a*a
            x, y = x, y - 1
        else:
            p2k += 2*b*b*x - 2*a*a*y + 2*b*b + 3*a*a
            x, y = x + 1, y - 1
    return result
```

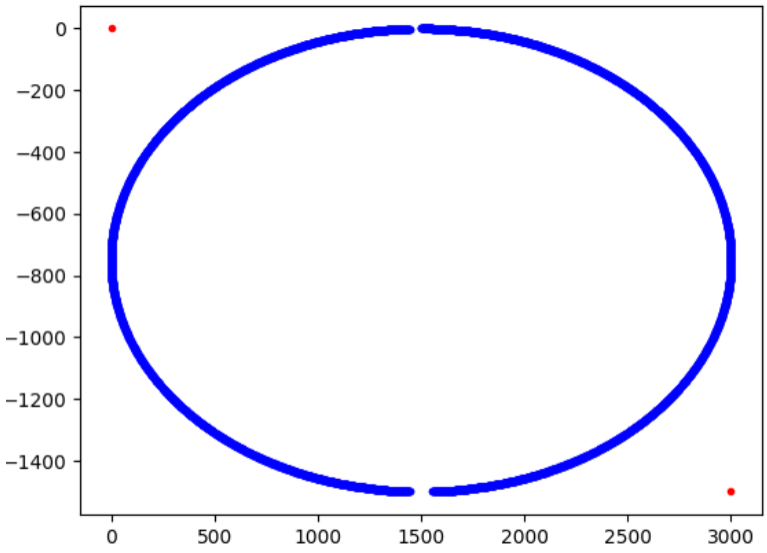
验证

由于该算法得到的结果是像素点的集合，是离散的点，为了让结果更加直观，将矩形包围框**跨度大一些**，这样看起来就近似为曲线，代码如下：

```
p_list = [[0, 0], [3000, -1500]]
for i in range(len(p_list)): # 控制点为红色
    plt.plot(*p_list[i], 'r.')

res = draw_ellipse(p_list)
x = [p[0] for p in res]
y = [p[1] for p in res]
plt.plot(x, y, 'b.') #椭圆点为蓝色
plt.show()
```

绘制结果如下：


0
−200
−400
−600
−800
−1000
−1200
−1400
0
500
1000
1500
2000
2500
3000

绘制曲线

Bezier算法

- 算法原理

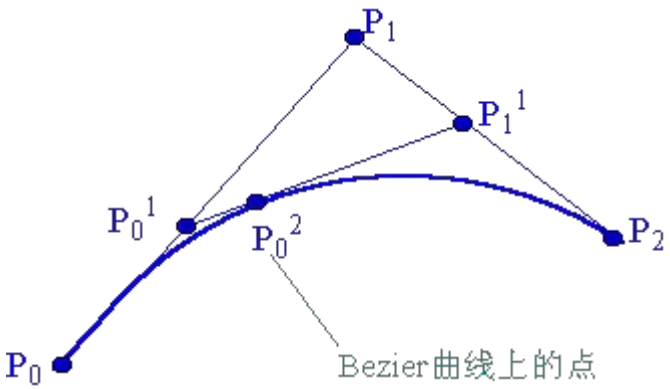
> $Bezier$曲线是通过一组多边折线得各个顶点唯一确定出来的，多边折线称为特征多边形其顶点称为控制顶点。假设给出$(n+1)$个控制顶点位置$P_i(i=1,2,\dots n)$，可以得到位置向量

$$P(u)=\sum_{i=0}^{n}P_iB_{i,n}(u)\quad(0<u<1)$$

> 其中$B_{i,n}(u)=C(n,i)u^i(1-u)^{n-i}$，为$n$次$Bernstein$基函数。$P(u)$即描述$P_0$到$P_1$逼近$Bezier$多项式函数的曲线。

但是在实际的绘制中，我们一般不会直接用上述曲线方程来计算，因为计算工作量太大。而是使用$de\ Casteljau$递推算法来产生曲线上的点，

原理和简单推导（以三个控制点为例）：



设P_0、P_1、P_2是三个控制顶点。P_0^2是$Bezier$曲线上的点，有如下比例成立：

$$\frac{P_0P_1}{P_0^1P_1}=\frac{P_1P_1^1}{P_1^1P_2}=\frac{P_0^1P_0^2}{P_0^2P_1^1}$$

引入参数t，令上述比值为$t:(1-t)$，即：

$$\begin{aligned}P_0^1&=(1-t)P_0+tP_1\\P_1^1&=(1-t)P_1+tP_2\\P_0^2&=(1-t)P_0^1+tP_1^1\end{aligned}$$

于是有：

$$P_0^2=(1-t)^2P_0+2t(1-t)P_1+t^2P_2$$

对其进行推广可以对于某一特定的参数u计算公式为：

$$P_i^r=\begin{cases}P_i & ,r=0\\(1-u)P_i^{r-1}+uP_{i+1}^{r-1} & ,r=1,2,\dots n;\ i=0,1,2,\dots n-r\end{cases}$$

其中$0<u<1$，可以看到多阶的曲线上需求每一个r阶点都可以通过降阶变成0阶点(即**控制点**)。

也就是说我们的工作就是，对于给出的$n+1$个控制点，我们需要求各个参数u下的$Bezier$曲线上的点，即n阶点P_0^n，它的递推式上面已经给出；当阶数为0时P_i就是已知的控制点中的第i个；而参数$0<u<1$，我们可以按照一个非常小的增量比如$\frac{1}{1000}$让u从0开始递增即可。

- 代码实现

  直接按照递推式写成递归函数的形式即可，由于画布像素的限制，曲线上坐标都进行了`round`操作：

```python
def draw_curve_Bezier(p_list):
    def P(r, i, u):
        if r == 0:
            return p_list[i]
        else:
            left = P(r - 1, i, u)
            right = P(r - 1, i + 1, u)
            return [(1-u)*left[0]+u*right[0], (1-u)*left[1]+u*right[1]]

    res = []
    n = len(p_list) - 1  # 最终点的阶数
    for i in range(0, 1001):
        u = i / 1000 # u取值为(0, 1)
        be_point = P(0, n, u) # 参数u下曲线上的点为P(0, n, u)
        res.append([round(be_point[0]), round(be_point[1])])
    return res
```

- 上面这种做法形式优雅，但是效率非常低因为递归调用会涉及到大量重复计算，因此采用**动态规划**的方式更好，存储矩阵中`loc[n][0]`就是$Bezier$曲线上的点：

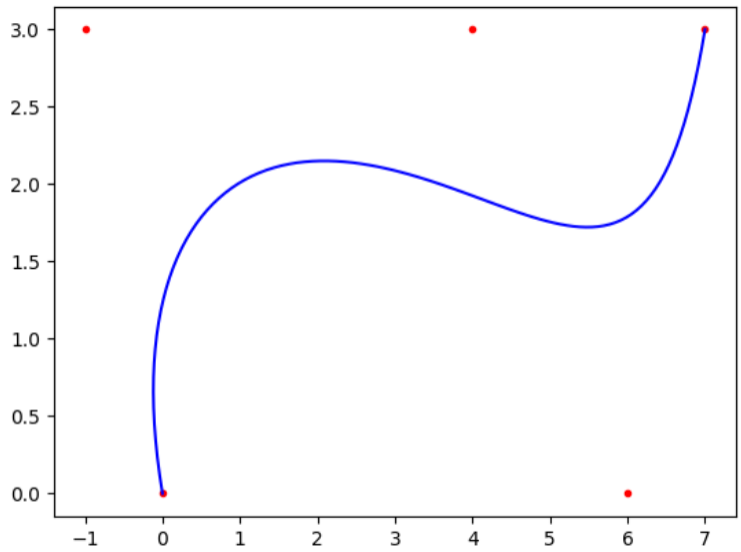
```python
def draw_curve_Bezier(p_list):
    n = len(p_list)-1
    loc = [[[0,0]*(n+1)]*(n+1)] #阶数r从0~n共(n+1)阶，每一阶的点数为(n+1-r)
    loc[0] = p_list # 初始化0阶点为控制点
    for v in range(0, 1001):
        u = v/1000 #u取值(0, 1)
        for r in range(1, n+1): #从上到下，阶数从1到n
            for i in range(0, n-r): #从左到右
                loc[r][i] = [(1-u)*loc[r-1][i][0] + u*loc[r-1][i+1][0],
                             (1-u)*loc[r-1][i][1] + u*loc[r-1][i+1][1]]
        res.append([round(loc[n][0][0]), round(loc[n][0][1])])
    return res
```

- 验证，通过`matplotlib.pyplot`库绘制所画的$Bezier$曲线是否正确，代码为

```python
p_list = [[0, 0], [-1, 3], [4, 3], [6, 0], [7, 3]]
for i in range(len(p_list)): # 控制点为红色
    plt.plot(*p_list[i], 'r.')

res = draw_curve_Beier(p_list)
x = [p[0] for p in res]
y = [p[1] for p in res]
plt.plot(x, y, 'b-') #Bezier曲线为蓝色
plt.show()
```

暂时删除了源程序中的取整操作，绘制出的$Bezier$曲线如图：



【参考文章】：

Finding a Point on a Bézier Curve: De Casteljau's Algorithm

Bezier曲线原理

B-spline算法(B样条曲线)

- 算法原理

  $B-spline$曲线的定义为：

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

其中P_i为控制多边形的顶点；$B_{i,k}$为k阶$(k-1$次$)$基函数，和$Bezier$算法不同，$Bezier$算法中阶数等于次数。为了解释定义域，需要先给出基函数的算法，即$deBoor-cox$递推算法：

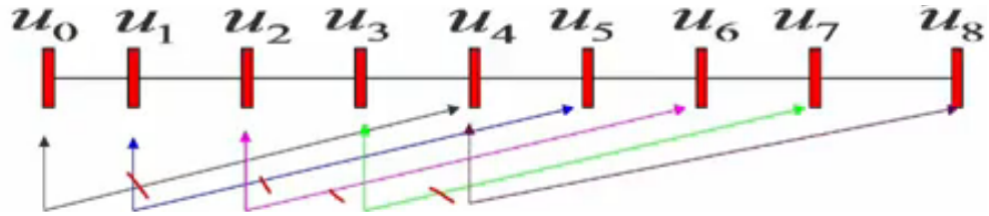
$$B_{i,k}(u)=\frac{u-u_i}{u_{i+k-1}-u_i}*B_{i,k-1}+\frac{u_{i+k}-u}{u_{i+k}-u_{i-1}}*B_{i+1,k+1}(u)$$

$$B_{i,1}(u)=\begin{cases}1 & u_i<u<u_{i+1};\\ 0 & Otherwise\end{cases}$$

我们规定： $\frac{0}{0}=0$

可以看到第i个k阶B样条$B_{i,k}(u)$涉及到的节点向量有$u_i, u_{i+1}, u_{i+2}\dots u_{i+k}$，支撑区间为$[u_i, u_{i+k}]$，因此$B$样条曲线$P(u)$涉及到的节点向量有$u_0, u_1, u_2\dots u_{n+k}$共$n+k+1$个节点。

我们以3次(4阶)B样条曲线为例，假设有5个控制点，则$n=k=4$，则$U=\{u_0, u_1\dots u_8\}$，曲线的第一项为$P_0B_{0,4}$，基函数涉及节点矢量为$\{u_0, u_1\dots u_4\}$；第二项为$P_1B_{1,5}$，基函数涉及点为$\{u_1, u_2\dots u_5\}$，之后类推



因为要让足够多的基函数和顶点对应，要找能够支撑最多基函数的区间，上图中为$u\in(u_3, u_5)$，推广得到一般情形为$u\in(u_{k-1}, u_{n+1})$。

同理可得，定义在(u_4, u_5)的曲线段涉及到的控制点有(P_1, P_2, P_3, P_4)，推广到一般情形，(u_i, u_{i+1})上的曲线段涉及到的控制点为$(P_{i-k+1}, P_{i-k+2}\dots P_i)$

- 3次均匀B样条曲线的绘制

以上为B样条曲线的算法原理，可以通过递推式暴力求解曲线上的坐标，但是这样开销太大。由于我们只需要求3次4阶均匀B样条曲线，设控制点为$(P_0, P_1, \dots P_n)$，其中每相邻4个点可以构造出一条三次曲线，其中第i段三次(4阶)均匀B样条曲线$P_{i,4}(u)$的矩阵可以表示为：

$$P_{i,4}(u)=\frac{1}{6}*\begin{bmatrix}u^3 & u^2 & u & 1\end{bmatrix}\begin{bmatrix}-1 & 3 & -3 & 1\\ 3 & -6 & 3 & 0\\ -3 & 0 & 3 & 0\\ 1 & 4 & 1 & 0\end{bmatrix}\begin{bmatrix}P_i\\ P_{i+1}\\ P_{i+2}\\ P_{i+3}\end{bmatrix}\quad i=0,1\dots n-3; 0<u<1$$

因此我们直接利用这个矩阵来求出B样条曲线上的$n-k+2=n-4+2=(n-2)$条三次曲线的方程，然后代入$u(0<u<1)$，求出曲线上的坐标即可。

- 代码实现：

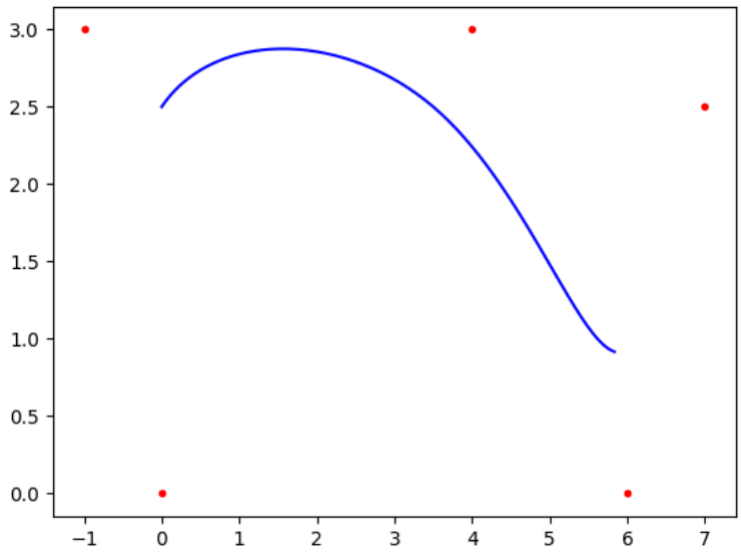
```
def draw_curve_B(p_list):
    def curve_point(i, u): #第i条曲线上参数为u的点坐标
        t = []
        point = [0, 0]
        t += [-u**3+3*u**2-3*u+1, 3*u**3-6*u**2+4,
             -3*u**3+3*u**2+3*u+1, u**3]
        for j in range(4): #每条曲线涉及4个控制点
            point[0] += t[j]*p_list[i+j][0]
            point[1] += t[j]*p_list[i+j][1]
        point[0], point[1] = point[0]/6, point[1]/6
        return point

    res = []
    n = len(p_list) - 1 #控制点从0开始编号
    for i in range(n-2): # k=4为阶数，一共n+1-(k-1)=n-2条三次曲线
        for v in range(1001):
            u = v/1000 #每条曲线上u从(0, 1)
```

```
            p = curve_point(i, u)
            res.append([round(p[0]), round(p[1])])
    return res
```

- 验证

  同样使用验证$Bezier$曲线时的代码，暂时删除了源程序中的取整操作，绘制出的B样条曲线如图：



【参考文章】：

B样曲线的矩阵表示

均匀B样条和准均匀B样条

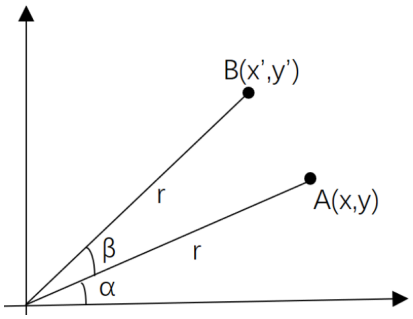
平移变换

只需要对图元的每个点像素坐标都进行(dx, dy)向量移动即可，代码如下：

```python
def translate(p_list, dx, dy):
    return [[x+dx, y+dy] for [x, y] in p_list]
```

旋转变换

点(x, y)绕原点旋转角度θ°（逆时针旋转为正）后得到的点$(x1, y1)$，假设线段$OA = r$



显然有：

$$\begin{cases} x1 = rcos(\theta + \alpha) = rcos\theta cos\alpha - rsin\theta sin\alpha \\ y1 = rsin(\theta + \alpha) = rsin\theta cos\alpha + rcos\alpha sin\theta \end{cases}$$

又因为：

$$\begin{cases} x = rcos\alpha \\ y = rsin\alpha \end{cases}$$

代入上式得：

$$\begin{cases} x1 = xcos\theta - ysin\theta \\ y1 = xsin\theta + ycos\theta \end{cases}$$

如果指定旋转基准点(x_r, y_r)和旋转角度θ°，其实就相当于先按照原点旋转，再将点沿(x_r, y_r)向量平移，因此(x_1, y_1)为：

$$\begin{cases} x_1 = x_r + (x - x_r)cos\theta - (y - y_r)sin\theta \\ y_1 = y_r + (x - x_r)sin\theta + (y - y_r)cos\theta \end{cases}$$

由于$python$的`math`库中三角函数的参数为**弧度**，因此需要先用`math.radians()`函数将角度转换成弧度。

代码如下：

```python
def rotate(p_list, x, y, r):
    res = []
    theta = math.radians(r)
    cos = math.cos(theta)
    sin = math.sin(theta)
    for p in p_list:
        x1 = x + (p[0] - x) * cos - (p[1] - y) * sin
        y1 = y + (p[0] - x) * sin + (p[1] - y) * cos
        res.append([round(x1), round(y1)])
    return res
```

缩放变换

点(x,y)相对于原点缩放系数为(S_x,S_y)，则缩放后的点为

$$\begin{cases}x_1=xS_x\\y_1=yS_y\end{cases}$$

给定缩放中心(x_r,y_r)就相当于先相对于原点缩放，再按照向量(x_r,y_r)平移，因此结果为：

$$\begin{cases}x_1=x_r+(x-x_r)S_x\\y_1=y_r+(y-y_r)S_y\end{cases}$$

代码如下：

```python
def scale(p_list, x, y, s):
    res = []
    for p in p_list:
        res.append([round(x+(p[0]-x)*s), round(y+(p[1]-y)*s)])
    return res
```

线段裁剪

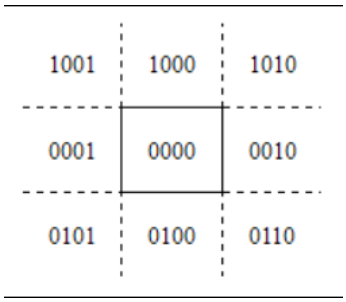
Cohen-Sutherland算法/编码裁剪算法

- 算法原理

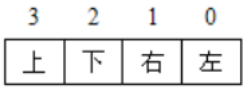
  该算法主要分为三步：

  - 第一步：对线段的端点区位编码

    给定矩形窗口，左上角为(x_{min}, y_{max})右下角为(x_{max}, y_{min})。将四条边界延长，则整个被平面分成九个区域，窗口内编码为0000，如图：

    每个区域内的点都对应着一个四位二进制区位码：

    任何位赋值1，代表端点落在相应的位置上，否则该位置为0。例如，如果端点在裁剪窗口内，则区位码为0000，如果端点在矩形裁剪窗口的左下角，则区位码为0101。

    根据要裁剪线段P_1P_2的端点坐标求出相应的编码值C_1和C_2。

  - 第二步：判断P_1、P_2的位置
    - 若$C1 = C2 = 0$，即P_1、P_2的编码全为零，线段P_1P_2在窗口内，保留线段P_1P_2，过程结束。
    - 否则，若$C1 \wedge C2 \neq 0$，即作P_1、P_2编码的逻辑与，结果非零时，表示P_1、P_2在窗口的**同侧**，将P_1、P_2丢弃，过程结束。
    - 否则，线段必有一端点在窗口外，令该点为P_1，进行下一步。
  - 第三步：裁剪窗口外的线段部分

    根据P_1点的编码值，确定其在哪条边界线之外，求线段与该边界线的交点P。交点把线段分成两段，舍去P_1P段，把交点P作为剩余线段的P_1端点重新进行第二步。

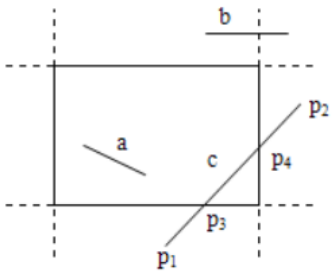
    求直线和边界的交点也很简单，比如要求直线和上边界的交点，先表示出直线方程：

    $$y - y_1 = \frac{y_1 - y_2}{x_1 - x_2}(x - x_1)$$

    代入已知的纵坐标$y = y_{max}$得交点的横坐标：

    $$x = x_1 + \frac{(y_{max} - y_1)(x_2 - y_1)}{y_2 - y_1}$$

比如下面的例子：



- 线段a经第二步测试为窗口内线段（$C_1 = C_2 = 0$），保留整条线段后结束算法。
- 线段b经第二步测试为窗口外同侧线段（$C1 \wedge C2 \neq 0$），舍弃整个线段后结束算法。
- 线段c需在第三步求出与窗口边界的交点P_3，舍去P_1P_3段；再把P_3作为新的P_1再进行第二步测试，又到第三步求出与窗口边界的交点P_4，舍去P_4P_2段，再把P_4作为新的P_2，经第二步测试为窗口内线段（$C_1 = C_2 = 0$），保留P_3P_4后结束算法。

- 代码实现

通过比较端点的坐标和边界的关系，通过或操作即可得到点的编码，上下右左依次对应或$8, 4, 2, 1$，可以编写出编码函数`encode(x, y)`；

之后的操作按照上述算法流程实现即可，在第三步裁剪时找边界外的点时，若`c1==0`则交换两点，`c1`则表示界外点。从高位到低位检查编码，不为0则求点和边界的交点作为新的端点，一直到两端点都在边界内结束。完整代码如下：

```
x1, y1 = p_list[0]
x2, y2 = p_list[1]
if y_min > y_max:
    y_max, y_min = y_min, y_max

def encode(x, y):  # 给定点[x, y]返回编码
    code = 0
    if y > y_max:
        code |= 8
    elif y < y_min:
        code |= 4
    if x > x_max:
        code |= 2
    elif x < x_min:
        code |= 1
    return code

if algorithm == 'Cohen-Sutherland':
    c1, c2 = encode(x1, y1), encode(x2, y2)
    if c1 == 0 and c2 == 0:  # 全部保留
        return p_list

    while True:  # 开始不断裁剪直到保留或舍弃
        if c1 & c2 != 0:  # 全部舍弃
            return []
        elif c1 == 0 and c2 == 0:  # 全部保留
            return [[round(x1), round(y1)], [round(x2), round(y2)]]

        if c1 == 0:  # 找到边界外的那个点
```

```
            x1, x2, y1, y2 = x2, x1, y2, y1
            c1, c2 = c2, c1

        if c1 & 8 == 8:  # 上边界外
            x1 = x1 + (y_max - y1) * (x2 - x1) / (y2 - y1)
            y1 = y_max
        elif c1 & 4 == 4:  # 下边界外
            x1 = x1 + (y_min - y1) * (x2 - x1) / (y2 - y1)
            y1 = y_min
        elif c1 & 2 == 2:  # 右边界外
            y1 = y1 + (y2 - y1) * (x_max - x1) / (x2 - x1)
            x1 = x_max
        elif c1 & 1 == 1:  # 左边界外
            y1 = y1 + (y2 - y1) * (x_min - x1) / (x2 - x1)
            x1 = x_min
        c1 = encode(x1, y1)  # 更新c1和c2的编码
        c2 = encode(x2, y2)
```

Liang-Barsky算法

- 算法原理

  Liang-Barsky算法的主要思想有两部分：

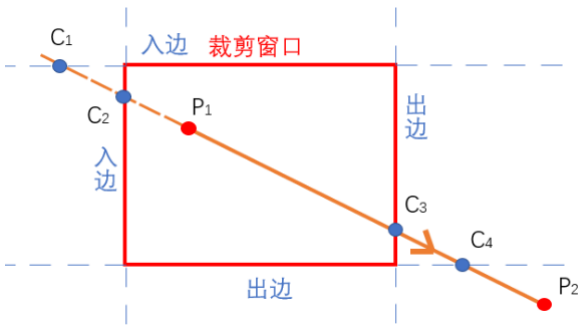
  - 用**参数方程**表示直线，对于端点为$(x_1,y_1),(x_2y_2)$的线段：

$$\begin{cases} x = x_1 + u(x_2 - x_1) = x_1 + u\Delta x \\ y = y_1 + u(y_2 - y_1) = y_1 + u\Delta y \end{cases}, 0 \leqslant u \leqslant 1$$

    显然当$u = 0$时表示起点(x_1,y_1)，当$u = 1$时表示终点(x_2,y_2)。

  - 将被裁剪线段看作**有方向**的线段，并将窗口四条边分成两类：

    将线段看作一条**有方向**的直线，则参数$-\infty < u < +\infty$。当u从$-\infty$到$+\infty$遍　历直线时，首先对裁剪窗口的两条边界直线（下边和左边）从外面向里面移动，再对裁剪窗口两 条边界直线（上边和右边）从里面向外面移动。因此把边界分成入边和出边两种：入边包括左边和下边，出边包括上边和右边。



    接下来就是判断直线上的点和窗口的位置关系，如果点在窗口内，有：

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

    这可以用4个不等式来表示：$up_k \leqslant q_k$，$k = 1,2,3,4$，其中：

$$\begin{cases} p_1 = -\Delta x, & q_1 = x_1 - x_{min}(\text{左边界}) \\ p_2 = \Delta x, & q_2 = x_{max} - x_1(\text{右边界}) \\ p_3 = -\Delta y, & q_3 = y_1 - y_{min}(\text{下边界}) \\ p_4 = \Delta y, & q_4 = y_{max} - y_1(\text{上边界}) \end{cases}$$

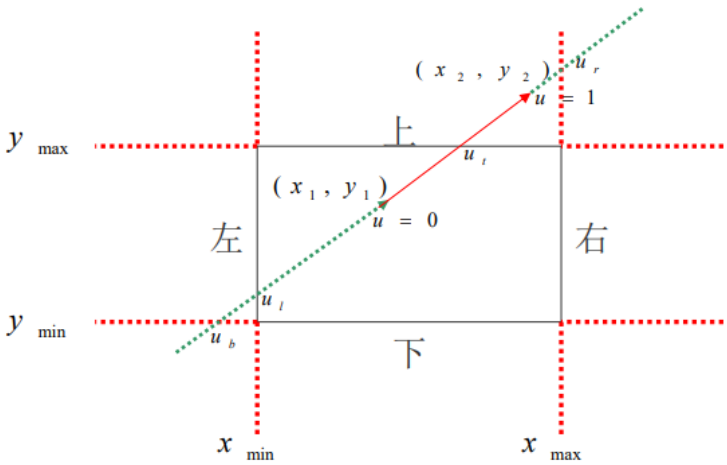
有以下结论：

- $p_k = 0$时，线段和某边界线平行，若再有$q_k < 0$则可以保证该线段完全在窗口外部，全部舍弃
- $p_k < 0$时，线段从裁剪边界延长线的外部延伸到内部，即对应**入边**
- $p_k > 0$时，线段从裁剪边界延长线的内部延伸到外部，即对应**出边**

因此当$p_k \neq 0$时，我们需要求线段和边界延长线的交点，来确定裁剪窗口内的部分线段。根据上面的推导，边界上的点满足$up_k = q_k$，即：

$$u = \frac{q_k}{p_k}, k = 1, 2, 3, 4$$

k取$1, 2, 3, 4$依次对应线段和左、右、下、上边界及延长线的交点参数u的值，如下图所示：



显然，裁剪结果的端点的参数(u_1, u_2)为:

$$\begin{cases} u1 = max(0, u_b, u_l) \\ u2 = min(1, u_t, u_r) \end{cases}$$

其中$u1$就是0和线段与**入边**两交点的**最大值**；$u2$就是1和线段与**出边**两交点的**最小值**。

最后对计算出的结果进行判断：若$u_1 > u_2$，则说明线段完全在窗口外，直接舍弃；否则保留结果代入直线方程中求出起点终点就为最终答案。

- 代码实现

```
elif algorithm == 'Liang-Barsky':
dx, dy = x2 - x1, y2 - y1
p = [-dx, dx, -dy, dy]
q = [x1 - x_min, x_max - x1, y1 - y_min, y_max - y1]
u1, u2 = 0, 1
for k in range(4):
    if p[k] == 0 and q[k] < 0:  # 全部舍弃
        return []
```

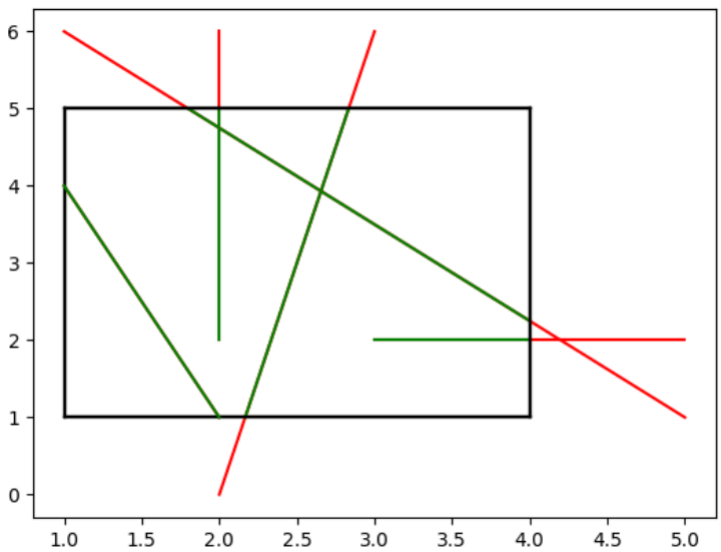
```
    elif p[k] < 0:  # 入边
        u1 = max(u1, q[k] / p[k])
    elif p[k] > 0:  # 出边
        u2 = min(u2, q[k] / p[k])
return [[round(x1 + u1 * (x2 - x1)), round(y1 + u1 * (y2 - y1))],
        [round(x1 + u2 * (x2 - x1)), round(y1 + u2 * (y2 - y1))]]
```

验证

同样用`matplotlib.pyplot`库，设置裁剪窗口为黑色，原始线段为红色，裁剪结果线段为绿色，代码如下：

```
x_min, y_min, x_max, y_max = 1, 1, 4, 5 # 设定边框
all_lines = [[[2,2],[2,6]], [[3,2],[5,2]], [[1,6],[5,1]],
             [[2,0],[3,6]], [[2,1],[1,4]]] # 检测5条线段
for p_list in all_lines:
    plt.plot([p_list[0][0], p_list[1][0]],
             [p_list[0][1], p_list[1][1]], 'r-')
    res = clip(p_list, x_min, y_min, x_max, y_max, "Cohen-Sutherland")
    # res = clip(p_list, x_min, y_min, x_max, y_max, "Liang-Barsky")
    plt.plot([x_min, x_min], [y_min, y_max], 'k-')
    plt.plot([x_max, x_max], [y_min, y_max], 'k-')
    plt.plot([x_min, x_max], [y_min, y_min], 'k-')
    plt.plot([x_min, x_max], [y_max, y_max], 'k-')
    x = [p[0] for p in res]
    y = [p[1] for p in res]
    plt.plot(x, y, 'g-')  # 裁剪结果为绿色
plt.show()
```

暂时省略了裁剪算法最终的`round`操作，两个算法的绘制结果都如下图所示：



【参考文章】：

WiKi_Cohn-sutherland algorithm

WiKi_Liang–Barsky algorithm

计算机小白也看得懂的Liang-Barsky算法

Cohen-Sutherland算法(编码裁剪算法)

实现命令行部分

对于`cg_cli.py`工作原理的解释在之前的报告中已经展示，详情见：理解框架代码:cg_cli.py，我们就按照理解中分类的三种指令来实现：

控制指令

只用实现`save_canvas()`函数，该函数遍历`item_dict`中的每个图元详细信息，根据信息设置相关算法的参数得到点位置序列，最后设置画布上相关位置像素的颜色即可，关键代码如下：

```
if item_type == 'polygon':
    pixels = alg.draw_polygon(p_list, algorithm)
    for x, y in pixels:
        canvas[y, x] = color
elif item_type == 'ellipse':
    pixels = alg.draw_ellipse(p_list)
    for x, y in pixels:
        canvas[int(y), int(x)] = color
elif item_type == 'curve':
    pixels = alg.draw_curve(p_list, algorithm)
    for x, y in pixels:
        canvas[y, x] = color
```

绘图指令

这部分指令的工作就是按照命令`line`向`item_dict`字典中添加相关图元的详细参数，包括图元编号`item_id`，图元类型，使用算法，控制点列表，颜色等。

实现`draw_polygon()`函数，控制点需要多边形顶点序列`[[x0, y0],[x1, y1]..]`，而命令`line`中依次给出每个点的横纵坐标，用`for`循环两两配对生成顶点序列即可，其余信息按`line`的给出顺序配置即可，代码如下：

```
def draw_polygon(line):
    global item_dict
    item_id = line[1]
    p_list = []
    for i in range(2, len(line) - 2, 2):
        p_list.append([int(line[i]), int(line[i + 1])])
    algorithm = line[-1]
    item_dict[item_id] = ['polygon', p_list, algorithm, np.array(pen_color)]
```

剩下的`draw_ellipse()`和`draw_curve()`大同小异，不再赘述。

变换指令

这部分指令的工作是按照命令`line`调用相关算法修改`item_dict`本身，即其中指定图元的详细信息，特别是控制点列表`p_list`。

实现`translate()`函数时，只需要将原本的图元信息中控制点列表经过平移算法变化即可，即修改`item_dict[item_id][1]`，代码如下：

```
def translate(line):
    item_id = line[1]
    p_list = item_dict[item_id][1]  # 图元原本的控制点
    dx, dy = int(line[2]), int(line[3])
    item_dict[item_id][1] = alg.translate(p_list, dx, dy)  # 修改成平移后的控制点
```

`rotate()`和`scale()`两个函数实现大同小异：

```
def rotate(line):
    item_dict[item_id][1] = alg.rotate(item_dict[line[0]][1], int(line[2]),
                                       int(line[3]), float(line[4]))
def scale(line):
    item_dict[item_id][1] = alg.rotate(item_dict[line[0]][1], int(line[2]),
                                       int(line[3]), float(line[4]))
```

`clip()`函数同样，只是需要的参数多一些：

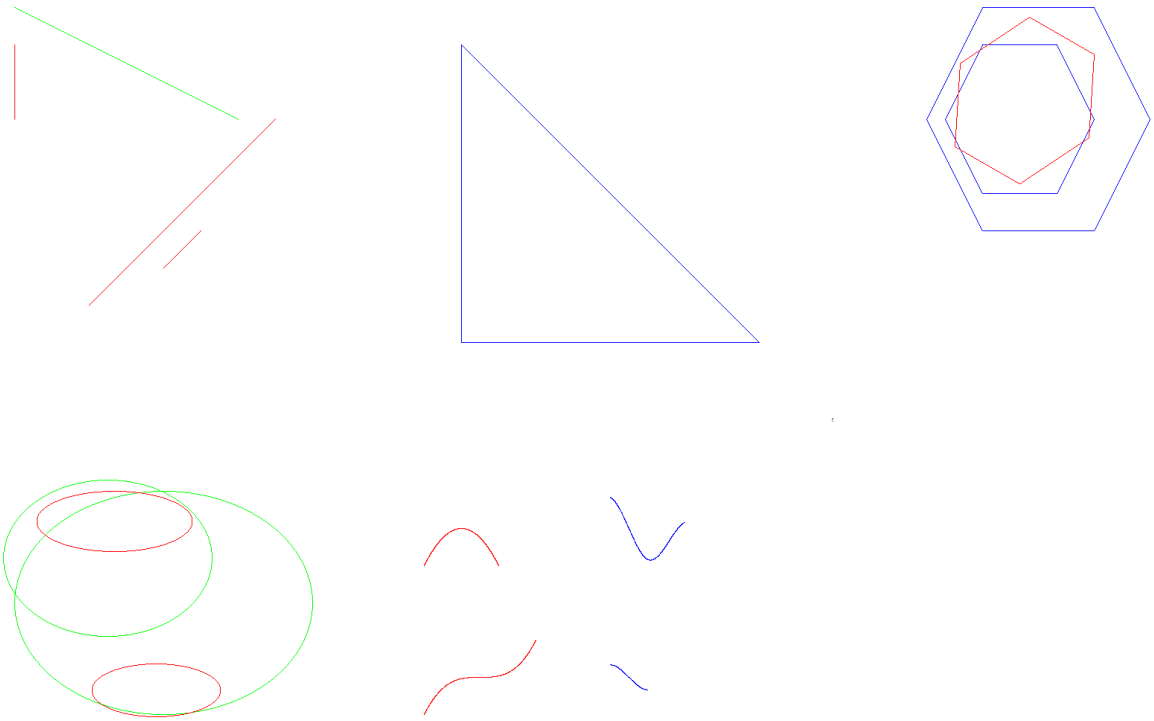
```
def clip(line):
    global item_dict
    item_id = line[1]
    p_list = [item_dict[item_id][1][0], item_dict[item_id][1][1]]  # 图元原本的参数
    x_min, y_min, x_max, y_max= int(line[2]), int(line[3]),
                                int(line[4]), int(line[5])
    algorithm = line[6]
    item_dict[item_id][1] = alg.clip(p_list, x_min, y_min,
                                     x_max, y_max, algorithm) # 修改成裁剪后的参数
```

测试

使用`input.txt`进行测试，在命令行输入指令：

```
python cg_cli.py input.txt output_dir
```

输出图像如下，可以看到符合预期。



实现图形界面

对于`cg_gui.py`工作原理的解释在之前的报告中已经展示，详情见：理解框架代码:cg_gui.py，下面就对于不同功能槽函数以及后续操作的实现进行介绍。

设置功能

- 设置画笔颜色

  使用QT的颜色会话框`QColorDialog`即可实现，在窗口类中新增`pen_color`属性，之后`MyItem`类中初始化时增加颜色参数即可实现彩色绘制。

```
def set_pen_action(self):
    self.pen_color = QColorDialog.getColor()
    self.statusBar().showMessage('设置画笔')
    self.list_widget.clearSelection()
    self.canvas_widget.clear_selection()
```

- 重置画布

  利用弹出的`QDialog`会话框来重置后的画布尺寸，高和宽都使用`QSlider`组件来确定。布局如下：

点击OK后，会清除所有的图元信息：

```
self.item_cnt = 0  # 清空图元
self.canvas_widget.clearCanvas()  # 清空画布图元
self.list_widget.clearSelection()  # 清除图元列表的选择
self.canvas_widget.clear_selection()  # 清除画布的选择
self.list_widget.clear()  # 清除图元列表
```

- 保存画布

  首先弹出文件对话框`QFileDialog`确定保存文件名：

```
dialog = QFileDialog()
filename = dialog.getSaveFileName(filter="Image Files(*.jpg *.png *.bmp)")
if filename[0]:
    self.canvas_widget.save_canvas(filename[0], self.weight, self.height)
```

  然后调用画布类中的函数，创建一个`QPixmap`图片对象，在上面绘制所有图元的像素点，然后用`save`函数将这个图片对象保存为文件即可。

```
def save_canvas(self, filename, weight, height):
    painter = QPainter()
    pixmap = QPixmap(weight, height)
    pixmap.fill(QColor(255, 255, 255))  # 涂满白色
    painter.begin(pixmap)
    for item in self.item_dict:
        self.item_dict[item].paint(painter, QStyleOptionGraphicsItem)
    painter.end()
    pixmap.save(filename, "bmp", 100)
```

【参考文章】：

Qt基础-标准对话框（QColorDialog、QFileDialog

将 Qt 中 自画图片保存为 图片文件

Python QPixmap.save方法代碼示例

QPixmap 类

绘图功能

鼠标点击某个绘图功能之后，产生信号调用对应的槽函数，比如`line_bresenham_action()`等，进行初步的操作，再调用画布类`MyCanvas`中相应操作，并且在界面状态栏展示信息即可。在窗口类中的这部分函数都比较简单，比如：

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

而画布类中的`start_draw_xxxx`函数则确定当前的状态以及图元编号，使用算法等等，比如：

```
def start_draw_line(self, algorithm, item_id):
    self.status = 'line'
    self.temp_algorithm = algorithm
    self.temp_id = 'line' + item_id
```

其余绘图功能这部分内容大同小异，不再赘述。

重点在于画布类中的后续操作的实现，即监控鼠标的各种动作：点击、移动及释放来确定图元绘制的方式。也就是三个函数`mousePressEvent`，`mouseMoveEvent`，`mouseReleaseEvent`。

- 绘制直线

  比较简单，当鼠标点击后直接创建新的图元添加到场景即可，并且将该点作为绘制起始点：

  ```
  if self.status == 'line':
          self.temp_item = MyItem(self.temp_id, self.status, [[x, y], [x, y]],
  self.temp_algorithm, self.main_window.pen_color)
          self.scene().addItem(self.temp_item)
  ```

  鼠标移动时，只需要将直线终点不断刷新成鼠标的当前位置即可：

  ```
  if self.status == 'line':
      self.temp_item.p_list[1] = [x, y]
  ```

  释放鼠标则代表绘制结束，只需要将该线段加入到`item_dict`和侧边栏结束绘制即可：

  ```
  if self.status == 'line':
      self.add_item()
  ```

- 绘制椭圆和绘制直线的过程几乎完全一致，不再赘述。
- 绘制多边形

  绘制多边形的逻辑比绘制直线复杂，因为不知道将会有多少个顶点以及何时结束绘制。为了达到更好的用户体验，我的设计是**点击鼠标左键则在对应位置加入新的顶点，点击鼠标右键结束绘制**。因此实际上只关注鼠标的点击操作：

  - 如果是绘制一个新的多边形，则创建新图元添加到场景

- 如果是在绘制过程之中，点击左键向图元参数中添加点击位置；点击右键代表结束绘制，相当于绘制直线中的释放鼠标：

```
if self.temp_item is None: # 新的图元
    self.temp_item = MyItem(self.temp_id, self.status, [[x, y], [x, y]], self.temp_algorithm, self.main_window.pen_color)
    self.scene().addItem(self.temp_item)
    self.setMouseTracking(True)  # 实时追踪鼠标位置
else:
    if event.button() == QtCore.Qt.RightButton:  # 停止绘制多边形
        self.add_item()
        self.setMouseTracking(False)
    else:
        self.temp_item.p_list.append([x, y])  # 按左键表示继续增加参数点
```

为了在绘制时对形状有更好的把握，我添加了`setMouseTracking(True)`实时显示当前鼠标位置可能出现的结果，这样绘制体验会更好。

- 绘制曲线沿用了和多边形一样的思路：点击左键加入新的控制点，右键结束绘制。代码几乎完全一致，不再赘述。

【参考文章】：How to return mouse coordinates in realtime?

变换功能

鼠标点击某个变换功能后，根据槽函数`xxxx_action`调用画布类中的`start_xxxx`函数，同时在状态栏展示信息，比如：

```
def translate_action(self):
    self.canvas_widget.start_translate()
    self.statusBar().showMessage('平移操作')
```

画布类中的`start_xxxx`函数用于确定当前状态，以及需要变换的图元和其他相关信息。比如：

```
def start_translate(self):
    self.status = 'translate'
    self.temp_item = self.item_dict[self.selected_id]  # 所要操作的是被选中图元
    self.rawList = self.temp_item.p_list
```

其余变换的`start_xxxx`函数大同小异，代码细节见源文件，不再赘述。

之后如何根据鼠标或者其他操作实现变换才是关键，我根据不同变换操作的特点分别进行了处理：

- 平移操作

  显然直接拖动鼠标移动图元是最理想的操作，平移算法中需要平移的水平分量和垂直分量，因此点击鼠标则确定了平移的起始点，在`mousePressEvent`中：

```
elif self.status == 'translate':
    self.begin = [x, y]
```

根据鼠标的移动位置，计算它和平移起始点之间的水平和垂直分量作为算法参数即可，在`mouseMoveEvent`中：

```
elif self.status == 'translate':
    self.temp_item.p_list = alg.translate(self.rawList, x - self.begin[0], y - self.begin[1])
```

释放鼠标后平移结束，保存结果到`rawList`用于后续使用，在`mouseReleaseEvent`中：

```
elif self.status == 'translate':
    self.rawList = self.temp_item.p_list
```

- 旋转操作

直接输入中心点和旋转角度貌似体验并不好，因此我决定通过滚轮来实现，下滑滚轮顺时针旋转，上划滚轮逆时针旋转；变换中心则通过点击鼠标确定，默认中心为`(0, 0)`。

在`mousePressEvent`中，点击更新旋转中心`begin`：

```
elif self.status == 'rotate':
    self.begin = [x, y]
    self.rotate_angle = 0  # 每次选择新的旋转中心角度清零
```

之后监控鼠标滚轮事件，通过滚轮上下来改变`rotate_angle`，作为旋转算法的参数：

```
def wheelEvent(self, event: QtGui.QWheelEvent) -> None:  # 鼠标滚轮
    if self.begin == []:  # 还没有选择参考点
        return
    if self.status == 'rotate':
        if event.angleDelta().y() > 0:
            self.rotate_angle -= 1
        elif event.angleDelta().y() < 0:
            self.rotate_angle += 1
        self.temp_item.p_list = alg.rotate(self.rawList, self.begin[0], self.begin[1], self.rotate_angle)
    self.updateScene([self.sceneRect()])
```

- 缩放操作

沿用旋转的思路，同样使用滚轮实现缩放，下滑缩小上滑放大，在滚轮事件中加入新的分支即可：

```
elif self.status == 'scale':
    if event.angleDelta().y() > 0:
        self.scale_factor += 0.1
    elif event.angleDelta().y() < 0:
        self.scale_factor -= 0.1
    self.temp_item.p_list = alg.scale(self.rawList, self.begin[0], self.begin[1], self.scale_factor)
```

其余和旋转操作大同小异，不再赘述。

- 裁剪操作

重点在于如何确定裁剪窗口的左上角和右下角坐标，这里采取的方法像绘制线段那样，点击鼠标确定裁剪窗口的一个顶点，移动鼠标并释放决定该顶点的对角顶点，这样就确定了裁剪窗口的参数。

在`mousePressEvent`中：

```
elif self.status == 'clip':
    self.begin = [x, y]
```

在`mouseReleaseEvent`中：

```
elif self.status == 'clip':
    theline = self.item_dict[self.selected_id]
    theline.p_list = alg.clip(theline.p_list, self.begin[0], self.begin[1], x, y,
self.temp_algorithm)
```

【参考文章】：

PyQt - QMessageBox

Python PyQt5.QtGui.QWheelEvent() Examples

一些改进

交互易用性

添加旋转缩放控制窗口

旋转和缩放在滚轮实现的基础上，添加了控制窗口，可以直接输入旋转角度和缩放比例来操作，也可以直接输入变换中心点的坐标。

首先在窗口上添加新的组件，变换中心点的横纵坐标，旋转角度和缩放比例都使用`QSpinBox`组件，方便微调，其中缩放比例的跨度为`0.1`。达到效果如下：

框内的值发生变化会调用绑定好的函数，比如：

```
self.angle_box.valueChanged.connect(self.change_angle)
```

旋转角度改变后，调用`change_angle`函数，即修改画布类中的值并调用旋转算法旋转：

```
def change_angle(self):
    if self.canvas_widget.status == 'rotate':
        ...
        self.canvas_widget.rotate_angle = self.angle_box.value()
        self.canvas_widget.temp_item.p_list = alg.rotate(self.canvas_widget.rawList,
self.beginx, self.beginy, self.angle_box.value())
        self.canvas_widget.updateScene([self.canvas_widget.sceneRect()])
```

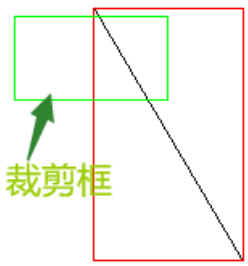
为了使滚轮和控制窗口信息同步，需要修改画布类的`wheelEvent`函数：

```
if self.status == 'rotate':
    if event.angleDelta().y() > 0:
        self.rotate_angle -= 1
        self.main_window.angle = self.rotate_angle
        self.main_window.angle_box.setValue(self.rotate_angle)
    elif event.angleDelta().y() < 0:
        self.rotate_angle += 1
        self.main_window.angle = self.rotate_angle
        self.main_window.angle_box.setValue(self.rotate_angle)
    self.temp_item.p_list = alg.rotate(self.rawList, self.begin[0], self.begin[1],
self.rotate_angle)
```

缩放操作的改进和旋转操作完全一致，不再赘述。

裁剪操作显示裁剪框

将裁剪框显示出来更加方便，由于窗口为矩形，相当于添加了一个新的多边形图元，裁剪过程中会显示，当结束后自动删除该图元即可。达到效果如下：



因此鼠标点击时，加入新多边形图元作为裁剪窗口，规定为绿色：

```
elif self.status == 'clip':
    self.temp_item = MyItem(None, 'polygon', [(x, y), (x, y), (x, y), (x, y)], 'DDA',
                                    QColor(0, 255, 0))  # 裁剪时画一个矩形框
    self.scene().addItem(self.temp_item)
```

鼠标移动时，更新裁剪矩形的参数，显示裁剪框的实时位置：

```
elif self.status == 'clip':
    x0, y0 = self.temp_item.p_list[0]
    self.temp_item.p_list = [[x0, y0], [x0, y], [x, y], [x, y0]]
    self.updateScene([self.sceneRect()])
```

鼠标释放时，这才根据裁剪矩形的位置调用算法进行裁剪操作，删除裁剪框图元，并且当线段被裁剪没时删除该线段的全部信息：

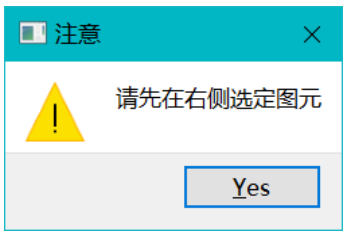
```
elif self.status == 'clip':
    x_min, y_min = self.temp_item.p_list[0]
    x_max, y_max = self.temp_item.p_list[2]
    if x_min > x_max:
        x_max, x_min = x_min, x_max
    if y_min > y_max:
        y_max, y_min = y_min, y_max
    theline = self.item_dict[self.selected_id]
    theline.p_list = alg.clip(theline.p_list, x_min, y_min, x_max, y_max,
self.temp_algorithm)
    self.scene().removeItem(self.temp_item)
    if len(theline.p_list) == 0:
        self.delete_item()
```

系统鲁棒性

画布类中的`start_xxxx`函数，如果事先没有选中图元，那么不应该进行任何操作，加入代码：

```
if self.selected_id == '':
    QMessageBox.warning(self, '注意', '请先在右侧选定图元', QMessageBox.Yes,
QMessageBox.Yes)
    return
```

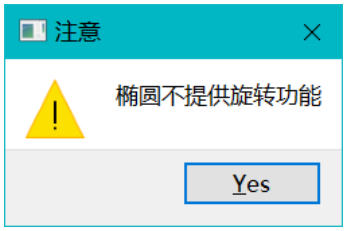
那么会跳出提示框；



由于椭圆不支持旋转操作，在`start_rotate`函数中加入：

```
if self.item_dict[self.selected_id].item_type == 'ellipse':
    QMessageBox.warning(self, '注意', '椭圆不提供旋转功能', QMessageBox.Yes,
QMessageBox.Yes)
    return
```

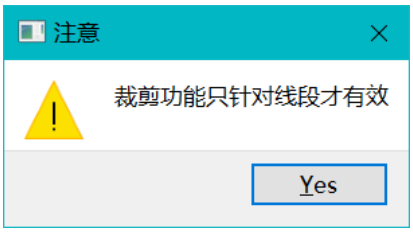
如果对椭圆旋转则跳出提示框：



同样裁剪算法只适用于线段，在`start_clip`函数中加入：

```
if self.item_dict[self.selected_id].item_type != 'line':
    QMessageBox.warning(self, '注意', '裁剪功能只针对线段才有效', QMessageBox.Yes, QMessageBox.Yes)
    return
```

如果对非线段图元进行裁剪则跳出提示框：



新功能

删除图元

选中图元后，可以将该图元完全删除，包括`item`的全部信息：图元列表，画布信息，`item_dict`等等，关键代码如下：

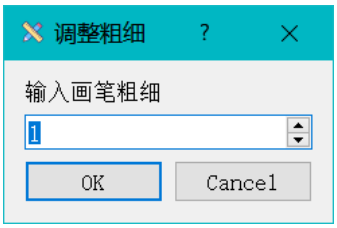
```
self.scene().removeItem(self.item_dict.pop(self.selected_id))
self.selected_id = ''
self.temp_id = ''
self.list_widget.takeItem(self.list_widget.row(self.list_widget.selectedItems()[0]))
self.list_widget.selectionModel().clear()
self.clear_selection()
self.main_window.statusBar().showMessage('空闲')
self.status = ''
```

设置画笔粗细

方法和设置画笔颜色完全一致，在窗口类中增加`self.pen_width`属性，增加`MyItem`类时添加宽度参数即可。对应槽函数为：

```python
def set_pen_width_action(self):
    width, ok = QInputDialog.getInt(self, '调整粗细', '输入画笔粗细',
                                    value=1, min=1, max=10)
    if ok and width > 0:
        self.pen_width = width
```

点击时将弹出一个输入对话框：



绘制时将`paint`函数中的`painter`用一个规定颜色和宽度的`QPen`初始化，即可实现不同粗细的绘制：

```python
painter.setPen(QPen(self.color, self.width))
painter.drawPoint(*p)
```

绘制填充多边形

操作逻辑和绘制普通多边形完全一致，即点击鼠标左键添加图元顶点参数，右键结束绘制。关键在于如何通过顶点参数得到多边形的内部点，这就用到了扫描填充算法。

- 算法原理
  - 用水平扫描线从下到上扫描由点线段构成的多段构成的多边形。
  - 每根扫描线与多边形各边产生一系列交点。将这些交点按照x坐标排序，将排序后的交点成对取出，每一对点之间的一系列点就是多边形的内部点，需要填充颜色。
  - 多边形被扫描完毕后，填色也就完成。

  为此，需要引入新的数据结构，即有序边表`NET`和活化边表`AET`。

  有序边表：是按照边的下端点y坐标对非水平边进行分类的指针数组，同类链表中的边按照x值的递增顺序排列而成，其中每个结点包含：下端点x坐标，上端点y坐标y_{max}，边斜率的倒数$\frac{1}{m}$。显然根据这四个信息可以完全确定一条边。

  活化边表：记录多边形沿扫描线的交点序列，对于第k条扫描线，首先根据有序边表，将在k以下的边插入活化边表，再删除其中满足$y_{max} < y_k$的边，其余的边一定和扫描线有交点，根据$x_{k+1} = x_k + \frac{(y_k - y_c)}{m}$计算交点横坐标。活化边表每个结点记录：改变的最大y值y_{max}，与扫描线交点横坐标值，边斜率倒数$\frac{1}{m}$。

- 代码实现

  首先需要创建有序边表，可以用列表代替，按照y坐标由小到大不断`append`可以达到桶的效果，每个桶内是个链表，因此创建链表结点类`Node`：

```
class Node:
    def __init__(self, x=0, dx=0.0, y_max=0, nxt=None):
        self.x = x
        self.dx = dx
        self.y_max = y_max
        self.next = nxt

    def set_next(self, nxt):
        self.next = nxt
```

之后先将活化边表初始化为空链表，从下到上处理每一条扫描线：首先将 `NET` 中登记项 y对应的各“吊桶”合并到表 `AET` 中，将 `AET` 中各吊桶按x坐标递增排序：

```
while node1 is not None:  # 将NET对应边插入AET中,并在插入时对x进行排序
    while node2.next is not 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
```

再将 `AET` 表中有 `y == y_max` 的各项清除出表：

```
node1 = AET  # 删除y_max == y_k的边否则保留
node2 = node1.next
while node2 is not None:
    if node2.y_max == i:
        node1.set_next(node2.next)
        node2 = node1.next
    else:
        node1 = node1.next
        node2 = node2.next
```

留下的各项必定和扫描线有交点，通过 `x=x+1/m` 求得 `AET` 中各边与下一条扫描线交点的x 坐标：

```
node = AET.next
while node is not None:  # 计算AET中扫描线和边交点的横坐标x = x + 1/m
    node.x = node.x + node.dx
    node = node.next
```

由于前一步可能破坏AET表中各项x坐标的递增次序，故按x 坐标重新排序

```
node1 = AET  # 对AET表按照x重新排序
node2 = AET.next
node1.set_next(None)
while node2 is not None:
    while node1.next is not 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
```

两两取交点，将两者之间的坐标加入到结果中即可：

```
node = AET.next
while node is not None and node.next is not None:
    x = int(node.x)
    while x <= node.next.x:
        result.append([x, i])
        x = x + 1
    node = node.next.next  # 将一对点之间的像素点加入到结果中
```

【参考资料】： 孙正兴, CG-2017-03-Primitives , 75页

多边形裁剪

$Sutherland-Hodgman$算法也称为逐边裁剪算法，即多边形的裁剪是关于裁剪窗口每条边界的裁剪。

- 算法原理
  - 裁剪窗口边界所在的**直线**将平面分成两个**半空间**：内测空间和外侧空间
  - 根据多边形的每条边和裁剪窗口边界所形成的半空间的相交关系，计算出交点，得到的新的多边形控制点序列作为下一次迭代的输入。

因此重点就在于求出多边形的边和裁剪窗口边界的位置关系，对于裁剪窗口的每一条边界，都对多边形所有边进行如下判断：

- 终点在窗口内：
  - 如果起点在窗口外，则计算该边和裁剪窗口的交点加入到结果中
  - 将终点加入到结果中
- 终点在窗口外且起点在窗口内：
  - 计算该边和裁剪窗口的交点加入到结果中

不断迭代，直到处理完裁剪窗口的所有边界即可。

- 代码实现

下面展示了代码的关键部分，即迭代裁剪的部分，实现细节详见源码

```
for i in range(len(clip_list)): # 一条边一条边迭代裁剪
    next_polygon = result.copy()
    result = []
```

```
    # 下面两个点确定裁剪窗口的一条边
    c_edge_start = clip_list[i - 1]
    c_edge_end = clip_list[i]
    for j in range(len(next_polygon)):
        # 下面两个点确定被裁减多边形的一条边
        s_edge_start = next_polygon[j - 1]
        s_edge_end = next_polygon[j]

        if is_inside(c_edge_start, c_edge_end, s_edge_end): # 终点在内起点在外
            if not is_inside(c_edge_start, c_edge_end, s_edge_start):
                intersection = ...
                result.append(intersection)
            result.append(s_edge_end)  # 结果为交点和终点
        elif is_inside(c_edge_start, c_edge_end, s_edge_start): # 终点在外起点在内
            intersection = ...
            result.append(intersection)
```

注意到起点在内终点在外的情况下，结果中只加入了一个交点，我们知道实际上起点和交点都是结果的一部分，但是这个起点在处理上一条边的时候被加入了，为了防止重复只加入交点。

【参考资料】： 孙正兴, CG-2017-05-Trimming , 35，36页