《计算机图形学》12月报告

作者姓名 杨溢 学号 171860540

(南京大学 计算机科学与技术系, 南京 210093)

摘 要: 本次大作业已经完成。实现了一个完整的图形学系统，功能包括初始化/重置画布，保存画布，设置画笔颜色，绘制线段，绘制多边形，绘制椭圆，绘制曲线，对图元的平移、旋转、缩放和对线段的裁剪。系统实现了上述功能的算法、在CLI下使用文件输入调用上述功能的接口以及在GUI下使用鼠标事件进行绘图以及图元编辑的功能。

关键词: 图形学系统;图元算法;文件输入;鼠标交互；GUI

# 语言、开发环境以及框架

本程序使用python3.7编写，在windows运行；图形界面部分使用tkinter框架。

# 数据结构

本程序的主要涉及到三种数据结构，第一个是用于存放图元信息的Primitive类及其派生类Line,Circle,Polygon等，第二个是GUI类，用于实现GUI交互逻辑，以及画板和图元信息的存放。第三个是CLI类，它是精简的GUI类，删去了和GUI相关的部分，保留绘图部分。

## 图元

所有的图元都是对象，他们的基类是Primitive类，包括以下数据属性：

1. **def** \_\_init\_\_(self, vertex, pno, color):
2. self.vertex = vertex
3. self.pixels = []
4. self.pno = pno
5. self.color = color

**self.vertex** list类型，其中每个元素是一个二元组，表示该图元的顶点；

**self.pixels** list类型，元素类型同上，这个图元光栅化后的所有像素坐标都存在；

**self.pno** int类型，是图元的id

**self.color** 长度为3的list， 表示这个图元的RGB值。

还包括以下方法：

**self.rasterization(self)**: 使用图元的顶点等信息，来求出该图元在画布中所有像素的位置，并将其存放在self.pixels数据属性中；

**self.get\_pixels(self)**:返回self.pixels；若为空，则先调用上一个函数进行光栅化然后再返回。

**self.get\_color(self)**: 将self.color整理成一个24位的数字返回，供绘图使用。

·直线类：Line继承自Primitive类， 还有以下数据属性：

1. **def** \_\_init\_\_(self, vertex, pno, method, color):
2. super().\_\_init\_\_(vertex, pno, color)
3. self.vertical = 0  # 是否存在斜率
4. self.slope = 0
5. self.method = method

**self.vertical** int类型，用来表示是否垂直；

**self.slope** int类型，是线段的斜率；若垂直，则将斜率定为10000；

**self.method** int类型，用来表示画图的算法（DDA或Bresenham）

还有以下方法：

**self.DDA(self)**: DDA算法，用于在self.rasterization(self)中调用。

**self.Bresenham(self)**: Bresenham算法，同上。

·椭圆类：Circle, 继承自Primitives类，还具有以下的数据属性：

1. **def** \_\_init\_\_(self, vertex, pno, color):
2. super().\_\_init\_\_(vertex, pno, color)
3. self.rx = vertex[1][0]
4. self.ry = vertex[1][1]

**self.rx, self.ry** int类型，长半轴的短半轴（ps. 椭圆的中心存放在self.vertex的第一个元素中）

·多边形类：Polygon，继承自Primitive类，还具有以下数据属性：

1. **def** \_\_init\_\_(self, vertex, pno, method, is\_done, color):  # is\_done:命令行直接完成，图形界面要等待完成
2. super().\_\_init\_\_(vertex, pno, color)
3. self.method = method
4. self.is\_done = is\_done
5. self.lines = []
6. self.last\_point = vertex[0]
7. self.is\_updating = 0
8. self.new\_point = [0, 0]

**self.method** int类型，用来表示画图的算法（DDA或Bresenham）；

**self.is\_done** int类型，用于区分命令行绘制还是鼠标绘制；

**self.lines** list类型，元素为Line。用来表示构成多边形的直线；

**self.last\_point,** **self.new\_point**, list类型，**self.is\_updating** int类型，均用于鼠标绘制多边形过程中的操作。

还具有以下方法，均用于鼠标绘制多边形：

**self.updating(self, point)**: 鼠标拖动过程，用于动态显示当前绘制的边；

**self.update\_rasterization(self, point)**：鼠标松开后，用于添加刚刚绘制好的顶点和边；

**self.done(self)**：完成绘制后，连接第一个和最后一个点。

·曲线类：Curve，继承自Primitive类，还具有以下数据属性

**self.alg** 字符串，表示曲线绘制的算法，取值为‘Bezier’或‘B-spline’

|  |
| --- |
| 各接口的详细功能和具体实现将在后续算法和框架部分详细描述 |

## GUI框架

GUI框架是GUI类的一个对象，包括以下数据成员：

·窗口以及组件

**self.top**：主窗口

**self.paper**：tkinter的画布

以及各种按钮：**self.draw\_line, self.draw\_circle, self.clean, self.close, self.save, self.line\_DDA , self.line\_Bre, self.owl, self.polygon**, 等等，用来输入坐标以绘制图元或者改变鼠标绘制图元的类型。

·图元相关信息

**self.primitives** 由Primitive类的对象构成的list，存放着画布中所有图元的信息

·画布相关信息

**self.image** 图片文件，显示在tkinter的画布上

**self.color\_r, self.color\_g, self.color\_b** 画笔颜色

**self.size\_x, self.size\_y** 画布大小

**self.save\_name** 要保存的文件名

**self.draw** 画笔

等等。

以及各种运行时需要的函数。各函数的功能以及实现的过程将在框架设计部分详细描述。

## CLI类

CLI类是进行文件输入和绘制的对象，包括以下数据成员：

**self.image** 图片文件，显示在tkinter的画布上

**self.color\_r, self.color\_g, self.color\_b** 画笔颜色

**self.size\_x, self.size\_y** 画布大小

**self.primitives** 由Primitive类的对象构成的list，存放着画布中所有图元的信息

**self.save\_name** 要保存的文件名

**self.draw** 画笔

以及用来实现解析指令、绘制图元等功能的函数

# 图元绘制、变换算法原理、实现及分析

此部分基于Primitive类，图元的绘制使用图元的基本信息，如线段的顶点，椭圆的圆心的长短半轴，多边形的顶点等等，来获得图元的需要在画布上占据的所有像素点的坐标，并将其返回给框架。而图元的变换（待添加）在原图元的基础上，对图元进行修改并返回新的像素点坐标。

## 线段的DDA算法

·算法原理

DDA算法方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。也就是说，通过在一个坐标轴 上以单位间隔对线段取样(取△x=1 或△y=1)，计算△y 或△x 决定另一个坐标轴上靠近 线段路径的对应整数值。

先考虑具有正斜率，从左端点到右端点进行处理的线段。若斜率 m≤1，则在单位 x间隔(△x=1)取样并计算每个顺序的 y 值：

其中，下标k取整数值从第一个点 1 开始递增，直至后端点。由于 m 可以是0与1之间的任意实数，所以计算出的y值必须取整。 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

以此类推，可以得到所有的像素点。

·代码实现

线段绘制的两种算法涉及到公共部分：

首先，求出线段斜率的绝对值，根据绝对值确定两种算法是单位间隔取样还是x间隔取样，

1. **def** rasterization(self):  # method: 1 for DDA, 2 for Bresenham
2. self.slope = 10000
3. self.vertical = 0
4. **if** self.vertex[1][0] == self.vertex[0][0]:
5. self.vertical = 1  # 垂直 不存在斜率
6. **if** self.vertical == 0:
7. self.slope = (self.vertex[1][1]-self.vertex[0][1])/(self.vertex[1][0]-self.vertex[0][0])
8. self.pixels = []
9. x = 0
10. y = 1
11. tmp\_slope = self.slope

由于vertex[0][0]和vertex[0][1]分别代表着第一个点和x坐标和y坐标，为了接下来编码的统一性，令x=0， y=1，若斜率的绝对值大于1，则互换x和y的值。

1. x = 0
2. y = 1
3. tmp\_slope = self.slope
4. **if** abs(self.slope) >= 1:  # 斜率绝对值大于1， x y互换
5. x = 1
6. y = 0
7. **if** tmp\_slope != 0:
8. tmp\_slope = 1 / tmp\_slope

特殊地，如果线段为水平或者垂直，则不需要DDA或者Bresenham算法，直接绘制即可。

1. **if** (self.vertex[0][x] == self.vertex[1][x]) **or** (self.vertex[0][y] == self.vertex[1][y]):
2. **return** self.ver\_or\_hor(x, y)

其中， self.ver\_or\_hor(x, y)为垂直或者水平时的绘图的函数，就是直接按照垂直或者水平方向将像素点加入self.pixels 并返回。

1. **def** ver\_or\_hor(self, x, y):
2. **if** self.vertex[0][x] > self.vertex[1][x]:  # 第一个点是"横坐标"较小的
3. temp = self.vertex[0]
4. self.vertex[0] = self.vertex[1]
5. self.vertex[1] = temp
6. **for** i **in** range(self.vertex[0][x], self.vertex[1][x] + 1):
7. point = [0, 0]
8. point[y] = self.vertex[0][y]
9. point[x] = i
10. # print(point)
11. self.pixels.append(point)
12. **return** self.pixels

准备工作的最后一步是把“x坐标”较小的放在前面，这样就可以避免讨论“横坐标”是正向增长还是负向增长的问题了。

1. **if** self.vertex[0][x] > self.vertex[1][x]:  # 第一个点是"横坐标"较小的
2. temp = self.vertex[0]
3. self.vertex[0] = self.vertex[1]
4. self.vertex[1] = temp

最后就可以调用DDA算法了

1. **if** self.method == 1:
2. **return** self.DDA(x, y, tmp\_slope)

DDA算法的代码如下：

1. **def** DDA(self, x, y, tmp\_slope):
2. point = [self.vertex[0][0], self.vertex[0][1]]
3. self.pixels.append(point)
4. yk = self.vertex[0][y]
5. **for** i **in** range(self.vertex[0][x] + 1, self.vertex[1][x] + 1):
6. point = [0, 0]
7. point[x] = i
8. yk = yk + tmp\_slope
9. point[y] = (math.ceil(yk) **if** math.ceil(yk) - yk < yk - int(yk) **else** int(yk))
10. self.pixels.append(point)
11. **print**("DDA")
12. **return** self.pixels

首先将第一个点放入self.pixels列表中，然后根据“横坐标”（即跨度较大的坐标轴），逐个计算“纵坐标”的值并取证，然后放入self.pixels列表中，最后返回。

·性能评价

由于DDA算法使用光栅扫描的特性，消去了直线方程中的乘法，因此计算的速度得到了较大的提升，但是由于涉及到浮点运算和取整，并不利于硬件实现。

## 线段的Bresanham算法

·算法原理

Bresenham 画线算法是一种精确而有效的光栅线段生成算法，它可用于圆和其它曲线显示的整数增量运算。图 2-21(a)示出了直线段的局部显示，垂直轴线表示扫描线位置，水平轴线标识像素列数，在此例中，在某个方向以单位间距取样。根据扫描转换原理，在每一个取样位置处，需确定哪个像素位置更接近于线段路径。

先考虑具有小于 1 的斜率线的扫描转换过程。线路径上的离散像素位置可通过在 x 方向以单位间距对线路径取样来确定：从给定线段的左端点所在像素位(x0,y0)开始， 以单位间隔依次处理每个后继像素列 x 位置，在所处理像素列选择 y 值接近线段的像素，并逐次绘出.

为简化像素的选择，Bresenham 算法通过引入整型参量定义来衡量两候选像素与线路径上实际(数学)点间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 近于实际线路径的像素。假如已经确定了这一步所 显示的像素在位置(xk,yk), 处，那么，下一步需要确定在列x(k+1)上绘制哪个像素：是在位置(x(k+1),yk)还是在位置(x(k+1),y(k)+1) ？在取样位置x(k+1)处，用d1和d2来标识下面和上面两个候选像素与线段数学路径的垂直偏移，在像素列位置x(k+1)处数学线段上的 y 坐标可计算为

则两个候选像素与线段数学路径的垂直偏移的差值为：

设：△y 和△x 分别为端点的垂直和水平偏离量，可知m=△y/△x，代入上述方程 可以得到：

pk成为画线算法中第k步的决策参数。由于 x 方向单位增量△x>0 ，因此， pk 的符号与d1-d2 的符号相同。假如yk处的像素比y(k+1)的像素更接近于线段(即d1<d2 )，那么，参数pk 是负的，此时，选择 绘制yk处的像素；反之，y(k+1)处的像素比yk的像素更接近于线段(即d1-d2>0)，那么， 参数 pk是正的，此时，选择绘制y(k+1)处的像素。也就是说，可以根据pk 的符号来决定 第 k+1 步所需要选择的像素.

每一单位步长都会引起沿线段 x 和 y 方向的坐标变化。因此，可利用递增整数运算得到后续pk的值：

而

由于x(k+1) = xk+1，而且pk>0时，y(k+1) = yk+1, pk<=0时，y(k+1) =yk.,因此：

pk>0时：

pk<=0时：

容易得知，pk的初始值：

因此，可由上述公式进行迭代，直到求出所有的点。

·代码实现

准备工作和DDA算法相同，此处不再赘述。

1. **elif** self.method == 2:
2. **return** self.Bresenham(x, y)

完成准备工作后，“横坐标”是跨度较大的坐标，以该坐标的单位间隔取样，迭代计算决策参数和对应“纵坐标”；由于已经将“横坐标”较小的排在了前面，因此只需考虑△x=1的情况即可。

函数详细内容如下:

1. **def** Bresenham(self, x, y):
2. point = [self.vertex[0][0], self.vertex[0][1]]
3. self.pixels.append(point)
4. # print(self.pixels)
5. deltax = self.vertex[1][x] - self.vertex[0][x]
6. deltay = self.vertex[1][y] - self.vertex[0][y]
7. **assert** deltax >= 0, "delta x < 0"
8. pk = 2 \* deltay - deltax
9. y\_negative = 0
10. **if** deltay < 0:  # 如果delta y 小于0， 则整体要向下, pk也要修改
11. y\_negative = 1
12. pk += 2 \* deltax
13. xk = self.vertex[0][x]
14. yk = self.vertex[0][y]
15. **for** k **in** range(deltax):
16. point\_a = [0, 0]
17. **if** pk < 0:
18. point\_a[x] = xk + 1  # self.vertex[k + 1][x]
19. point\_a[y] = yk - y\_negative
20. yk = yk - y\_negative
21. pk = pk + 2 \* deltay
22. **else**:
23. point\_a[x] = xk + 1  # self.vertex[k + 1][x]
24. point\_a[y] = yk + 1 - y\_negative
25. yk = yk + 1 - y\_negative
26. pk = pk + 2 \* deltay - 2 \* deltax
27. **if** y\_negative == 1:
28. pk = pk + 2 \* deltax
29. xk = xk + 1
30. self.pixels.append(point\_a)
31. **print**("Bresenham")
32. **return** self.pixels

前文的算法原理中，只描述了△y>0时的情况，而当△y<0时：

若pk>0，y(k+1)应该取上方的点，而此时上方的点是yk；pk<=0时，y(k+1)应该取下方的点，此时下方的点时yk-1。因此，当△y<0时，pk的初始值变为：

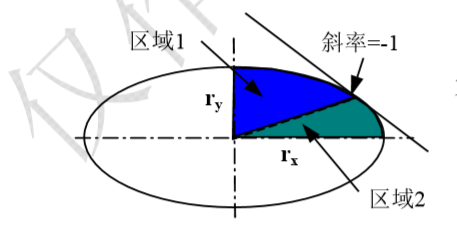
相应的，在迭代过程中，若pk<0, 则

若pk>=0，则

·性能评价

Bresenham算法简化了像素的选择，通过引入整型参量定义来衡量两个候选像素与线段路径上实际(数学)点间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于实际线路径的像素。由于决策参数的引入，整个计算过程中国不涉及到乘法及浮点运算，优化了在硬件上的计算速度。

## 中点椭圆算法

椭圆被定义为到两个定点(焦点)的距离之和等于常数的点的集合。在任意方向指定一个椭圆的交互方法是输入两个焦点和一个椭圆边界上的点，利用这三个坐标位置，就可求出显式方程中的常数，而后就可求出隐式方程中的系数，并用来生成沿椭圆路径的像素。假如短轴和长轴与坐标轴方向平行，那么椭圆方程就可大大简化。一个“标准位置”椭 圆是指其长轴和短轴平行于 x 和 y 轴，参数 rx 标出长半轴，参数ry标识短半轴。标准位 置的椭圆在四分象限中是对称的，利用对称性可减少计算量，只需计算一个四分象限中 椭圆曲线的像素位置，再由对称性得到其它三个象限中的像素位置。

中点椭圆方法依据椭圆斜率将第一象限的椭圆分成两部分，即区域 1 和区 域 2，如图所示。两个区域的分割条件可通过检测曲线的斜率值而得到。椭圆的斜率可从椭圆方程中计算出：

在区域 1 和区域 2 的交界处，,由此可得，区域 1 和区域 2 的分割条件为：。或者说，移出区域 1 进入区域 2 的条件是:

通过在斜率绝对值小于 1 的区域在 x 方向取单位步长，在斜率绝对值大于1的区域在 y 方向取单位步长来处理.

定于椭圆函数：

当函数值大于0时，点(x,y)位于椭圆周边界外；当函数值等于0时，点(x,y)位于椭圆周边界上；

当函数值小于0时，点(x,y)位于椭圆周边界内。因此，这个函数的值可以作为决策参数，按照椭圆函数在沿椭圆轨迹两个候选像素间中点求值的符号选择下一个像素。

在区域1中：

若前一步选择了(xk,yk)，则

假如<0，中点位于椭圆内，扫描线上的像素更接近于椭圆边界；否则，中点在椭圆之外，或在椭圆边界上，所选的像素应在扫描线上,即：

<0时，

>0时， p1(k+1) = p1k + 2ry^2xk – 2rx^2yk + 2rx^2 + 3ry^2

在区域2中：

若前一步选择了，则

假如 <0，中点位于椭圆内，扫描线 上的像素更接近于椭圆边界；否则，中点在椭圆之外，或在椭圆边界上，所选的像素应在扫描线上,即：

>0时，

<0时，

不难得知，的初始值为

可由此绘制出四分之一的椭圆；再由对称性绘制出其余部分，最后再根据圆心坐标平移到正确的位置即可。

·代码实现

首先，根据上述算法，先绘制以原点为圆心的四分之一椭圆

1. **def** rasterization(self):
2. self.rx = self.vertex[1][0]
3. self.ry = self.vertex[1][1]
4. **if** self.vertex[1][0] == 0 **or** self.vertex[1][1] == 0:
5. **return** self.pixels
6. self.pixels = []
7. rx = self.rx
8. ry = self.ry
9. # 区域1
10. self.pixels.append([0, ry])
11. x = 0
12. y = ry
13. p1k = ry \* ry - rx \* rx \* ry + rx \* rx /4   # P one K  not PLK
14. **while** 2 \* ry \* ry \* x <= 2 \* rx \* rx \* y:
15. **if** p1k < 0:
16. p1k = p1k + 2 \* ry \* ry \* x + 3 \* ry \* ry
17. x = x + 1
18. **else**:
19. p1k = p1k + 2\*ry\*ry\*x + 3\*ry\*ry - 2\*rx\*rx\*y + 2 \* rx\*rx
20. x = x + 1
21. y = y - 1
22. self.pixels.append([x, y])
23. # print([x, y])
24. p2k = ry\*ry\*(x+0.5)\*(x+0.5) + rx\*rx\*(y-1)\*(y-1) - rx\*rx\*ry\*ry
25. **while** y >= 0:
26. **if** p2k <= 0:
27. p2k = p2k + 2\*ry\*ry\*x + 2\*ry\*ry - 2\*rx\*rx\*y + 3\*rx\*rx
28. y = y - 1
29. x = x + 1
30. **else**:
31. p2k = p2k - 2\*rx\*rx\*y + 3\*rx\*rx
32. y = y - 1
33. self.pixels.append([x, y])

然后在根据x轴、y轴进行镜像，然后再根据圆心移动所有的像素点的坐标。

1. len\_1 = self.pixels.\_\_len\_\_()
2. **for** i **in** range(len\_1):  # y对称
3. temp = [-self.pixels[i][0], self.pixels[i][1]]
4. **if** temp[0] != 0:
5. self.pixels.append(temp)
7. len\_2 = self.pixels.\_\_len\_\_()
8. **for** i **in** range(len\_2):  # x对称
9. temp = [self.pixels[i][0], -self.pixels[i][1]]
10. **if** temp[1] != 0:
11. self.pixels.append(temp)
13. len\_3 = self.pixels.\_\_len\_\_()
14. **for** i **in** range(len\_3):  # 平移
15. self.pixels[i][0] += self.vertex[0][0]
16. self.pixels[i][1] += self.vertex[0][1]
17. **return** self.pixels

·性能评价

中点椭圆算法通过决策参数的引入，避免了椭圆的方程的平方根操作，同时利用对称性，也减少了计算量。

## 多边形的绘制方法

多边形的绘制可以转变为多个直线的绘制，因此不涉及到新的算法。但是，由于鼠标绘制多边形的需要，多边形的某条直线可能并不属于这个多边形，这就需要多边形来提供更多的接口，来实现鼠标拖动时的预览。

首先是构造函数。

1. **def** \_\_init\_\_(self, vertex, pno, method, is\_done, color):  # is\_done:命令行直接完成，图形界面要等待完成
2. super().\_\_init\_\_(vertex, pno, color)
3. self.method = method
4. self.is\_done = is\_done
5. self.lines = []
6. self.last\_point = vertex[0]
7. self.is\_updating = 0
8. self.new\_point = [0, 0]
9. **if** is\_done:
10. # 直接光栅化完成
11. **for** i **in** range(1, len(vertex)):
12. **print**(i)
13. self.update\_rasterization(vertex[i])
14. self.last\_point = vertex[i]
15. self.done()

若is\_done参数为1，则此时点的输入已经完成，可以直接生成像素点。若is\_done参数为0，则还需要后续的升级。

updating函数：用于鼠标正拖动时，最后一条直线的预览。与重写的get\_pixels配合使用。

1. **def** updating(self, point):
2. self.new\_point = point
3. self.is\_updating = 1

重写的get\_pixels函数如下，若当前鼠标正在拖动，则返回已经确定下来的像素点，和当前还没有确定的最后一条直线的像素点；否则直接返回所有像素点。

1. **def** get\_pixels(self):
2. **if** self.is\_updating == 1:
3. res = self.pixels
4. tmp\_line = Line([self.last\_point, self.new\_point], 0, self.method, self.color)
5. tmp\_pix = tmp\_line.get\_pixels()
6. **return** res + tmp\_pix
7. **else**:
8. **return** self.pixels

update\_rasterization函数对应松开鼠标的操作。此时，一个点确定被加入多边形，这条直线所对应的点也加入了多边形的像素中。

1. **def** update\_rasterization(self, point):
2. tmp\_line = Line([self.last\_point, point], 0, self.method, self.color)
3. self.last\_point = point
4. self.pixels.extend(tmp\_line.rasterization())
5. self.is\_updating = 0

done函数对应着多边形的绘制完成。连接最后一个点和第一个点。

1. **def** done(self):
2. tmp\_line = Line([self.last\_point, self.vertex[0]], 0, self.method, self.color)
3. tmp\_pix = tmp\_line.get\_pixels()
4. self.pixels.extend(tmp\_pix)

进一步的描述请参见后文4.4节中多边形的绘制的部分。

## Bezier曲线的绘制

·算法原理

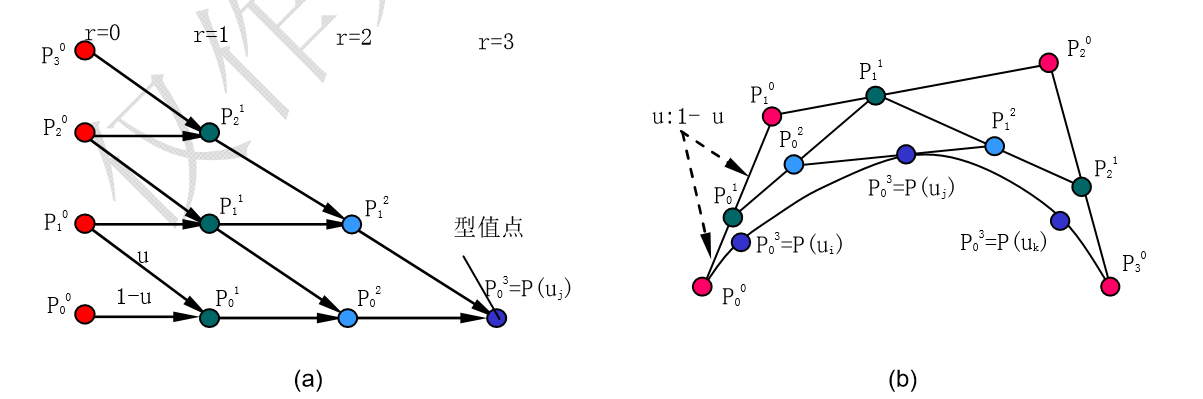
Bézier 曲线是通过一组多边折线的各顶点唯一定义出来的。曲线的形状趋向于多边折线的形状，改变多边折线的顶点坐标位置和改变曲线的形状有紧密的联系。因此，多边折线有常称为特征多边形，其顶点称为控制顶点。一般，Bezier 曲线段可拟合任何数目的控制顶点。Bezier 曲线段逼近这些控制顶点，且它们的相关位置决定了 Bezier 多项式的次数。类似插值样条，Bezier 曲线可以由给定边界条件、特征矩阵或混合函数决定，对一般 Bezier 曲线，方便的是混合函数形式

Bezier曲线是由Bernstein基函数线性组合而成的。Bernstein基函数的形式为：

给出n+1个控制点的位置，这些坐标点混合产生下列位置向量P(u),用来描述第一个和最后一个顶点之间逼近贝塞尔多项式函数的路径：

分割递推算法: 该算法描述了直接利用控制多边形顶点从参数u计算n次Bézier曲线型值点P(u)的过程。对于某一特定的参数u，其计算公式为:

下图以三次贝塞尔曲线为例，：r=0 时对应的顶点是曲线的 控制顶点本身；当 r 不断增加时，每两个顶点生成一个新的顶点，对应的顶点数递减；直到r=3 时只剩下一个顶点，即为所求的型值点。所有顶点构成一个直角三角形，在 r=1,2,3 各列中的每个点都有两个箭头指向它，代表该点是两箭头始点的线性组合，箭头上标注的代表权值。

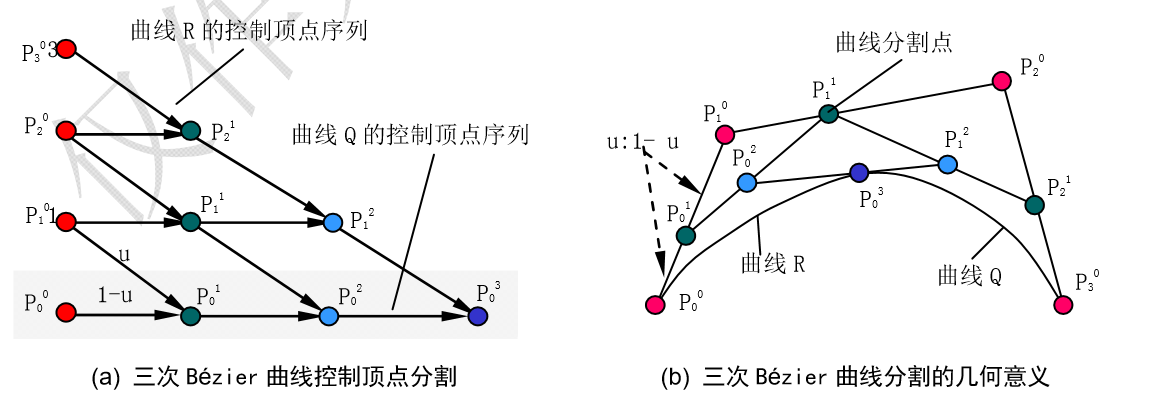


曲线逼近: 如果将 Bézier 曲线 P 从中间某点分成两段：Q 和 R。Q 和 R 仍是多项式曲线，可表示成 Bézier 曲线的形式，可用分割定理来求新的曲线 Q 和 R 的控制顶点。

分割定理：从参数u1处将贝塞尔曲线

分成两段曲线Q:P(u)(0<=u<=u1) 和R:P(u)(u1<=u<=1)可表示为

以三次贝塞尔曲线为例：



由于贝塞尔曲线的凸包性，在逼近到一定程度后，曲线会无限接近于由两端的控制点组成的直线。因此，可以用多i按行的各个顶点到底端直线的最大距离来表示逼近程度，进而判断什么时候用直线代替曲线。

·代码实现

使用分割递推和离散逼近，取参数u=0.5，先使用动态规划使用上述公式计算出P(i,r)。

1. **def** Bezier(self):
2. # print('Bezier')
3. p = np.zeros((len(self.vertex), len(self.vertex), 2), dtype=np.float)
4. **for** i **in** range(len(self.vertex)):
5. p[i][0] = self.vertex[i]
6. **for** i **in** range(len(self.vertex)-1, -1, -1):
7. **for** j **in** range(1, len(self.vertex)-i):
8. p[i][j][0] = 0.5 \* p[i][j - 1][0] + 0.5 \* p[i + 1][j - 1][0]
9. p[i][j][1] = 0.5 \* p[i][j - 1][1] + 0.5 \* p[i + 1][j - 1][1]

然后，根据离散逼近的方法，将曲线分为两个部分，取u1 = 0.5，分别判断两个部分控制点到直线的最大距离，若最大距离小于1则将该段曲线以直线代替，否则进行递归。

1. q\_vertex = p[0]
2. r\_vertex = [p[i][len(self.vertex)-i-1] **for** i **in** range(len(self.vertex))]
3. **if** max([self.distance(point, [q\_vertex[0], q\_vertex[len(self.vertex)-1]]) **for** point **in** q\_vertex]) <= 1:
4. # print(int(q\_vertex[0][0]))
5. vertex = [[int(q\_vertex[0][0]), int(q\_vertex[0][1])],
6. [int(q\_vertex[len(self.vertex)-1][0]), int(q\_vertex[len(self.vertex)-1][1])]]
7. temp = Line(vertex, 1, 1, 1)
8. self.pixels.extend(temp.get\_pixels())
9. **else**:
10. temp = Curve(q\_vertex, 1, 'Bezier', 1, 1)
11. self.pixels.extend(temp.get\_pixels())
12. **if** max([self.distance(point, [r\_vertex[0], r\_vertex[len(self.vertex)-1]]) **for** point **in** r\_vertex]) <= 1:
13. vertex = [[int(r\_vertex[0][0]), int(r\_vertex[0][1])],
14. [int(r\_vertex[len(self.vertex) - 1][0]), int(r\_vertex[len(self.vertex) - 1][1])]]
15. temp = Line(vertex, 1, 1, 1)
16. self.pixels.extend(temp.get\_pixels())
17. **else**:
18. temp = Curve(r\_vertex, 1, 'Bezier', 1, 1)
19. self.pixels.extend(temp.get\_pixels())

·评价：

Bezier曲线法能让设计者比较直观的意识到所给设计条件与终生成曲线之间的关系，能比较方便的通过修改输入参数来改变曲线的形状和阶次。Bézier 样条有很多性质使得在曲线和曲面设计中更有效且 更方便，也更容易使用。其离散生成相比固定步长生成，能够动态调整曲线中点的线密度，可以避免生成的曲线的点过于密集或者过于稀疏。

## B样条曲线的绘制

·算法原理

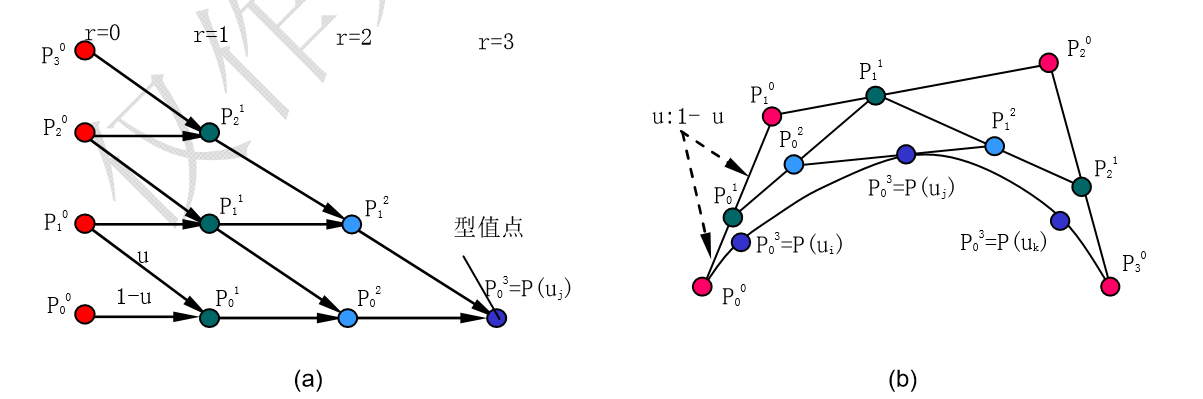
Bézier 曲线是通过一组多边折线的各顶点唯一定义出来的。曲线的形状趋向于多边折线的形状，改变多边折线的顶点坐标位置和改变曲线的形状有紧密的联系。因此，多边折线有常称为特征多边形，其顶点称为控制顶点。一般，Bezier 曲线段可拟合任何数目的控制顶点。Bezier 曲线段逼近这些控制顶点，且它们的相关位置决定了 Bezier 多项式的次数。类似插值样条，Bezier 曲线可以由给定边界条件、特征矩阵或混合函数决定，对一般 Bezier 曲线，方便的是混合函数形式

Bezier曲线是由Bernstein基函数线性组合而成的。Bernstein基函数的形式为：

给出n+1个控制点的位置，这些坐标点混合产生下列位置向量P(u),用来描述第一个和最后一个顶点之间逼近贝塞尔多项式函数的路径：

分割递推算法: 该算法描述了直接利用控制多边形顶点从参数u计算n次Bézier曲线型值点P(u)的过程。对于某一特定的参数u，其计算公式为:

下图以三次贝塞尔曲线为例，：r=0 时对应的顶点是曲线的 控制顶点本身；当 r 不断增加时，每两个顶点生成一个新的顶点，对应的顶点数递减；直到r=3 时只剩下一个顶点，即为所求的型值点。所有顶点构成一个直角三角形，在 r=1,2,3 各列中的每个点都有两个箭头指向它，代表该点是两箭头始点的线性组合，箭头上标注的代表权值。

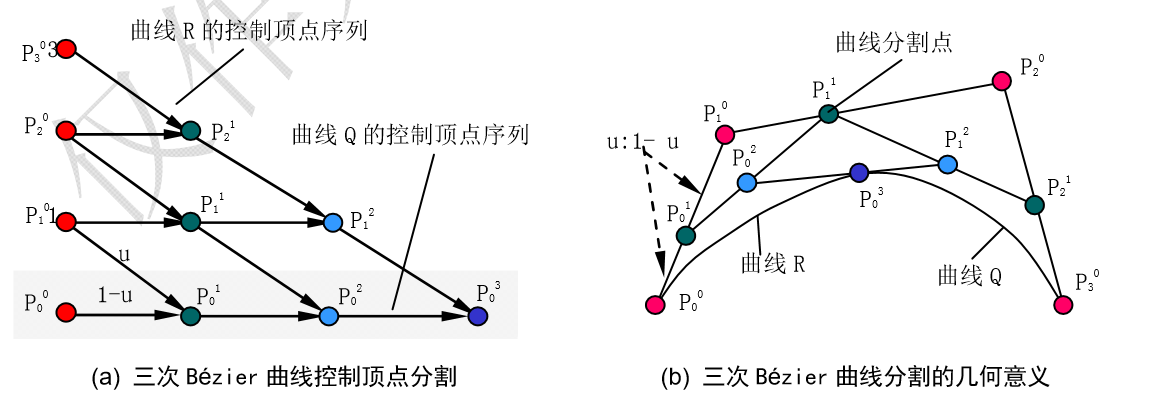


曲线逼近: 如果将 Bézier 曲线 P 从中间某点分成两段：Q 和 R。Q 和 R 仍是多项式曲线，可表示成 Bézier 曲线的形式，可用分割定理来求新的曲线 Q 和 R 的控制顶点。

分割定理：从参数u1处将贝塞尔曲线

分成两段曲线Q:P(u)(0<=u<=u1) 和R:P(u)(u1<=u<=1)可表示为

以三次贝塞尔曲线为例：



由于贝塞尔曲线的凸包性，在逼近到一定程度后，曲线会无限接近于由两端的控制点组成的直线。因此，可以用多i按行的各个顶点到底端直线的最大距离来表示逼近程度，进而判断什么时候用直线代替曲线。

·代码实现

使用分割递推和离散逼近，取参数u=0.5，先使用动态规划使用上述公式计算出P(i,r)。

1. **def** Bezier(self):
2. # print('Bezier')
3. p = np.zeros((len(self.vertex), len(self.vertex), 2), dtype=np.float)
4. **for** i **in** range(len(self.vertex)):
5. p[i][0] = self.vertex[i]
6. **for** i **in** range(len(self.vertex)-1, -1, -1):
7. **for** j **in** range(1, len(self.vertex)-i):
8. p[i][j][0] = 0.5 \* p[i][j - 1][0] + 0.5 \* p[i + 1][j - 1][0]
9. p[i][j][1] = 0.5 \* p[i][j - 1][1] + 0.5 \* p[i + 1][j - 1][1]

然后，根据离散逼近的方法，将曲线分为两个部分，取u1 = 0.5，分别判断两个部分控制点到直线的最大距离，若最大距离小于1则将该段曲线以直线代替，否则进行递归。

1. q\_vertex = p[0]
2. r\_vertex = [p[i][len(self.vertex)-i-1] **for** i **in** range(len(self.vertex))]
3. **if** max([self.distance(point, [q\_vertex[0], q\_vertex[len(self.vertex)-1]]) **for** point **in** q\_vertex]) <= 1:
4. # print(int(q\_vertex[0][0]))
5. vertex = [[int(q\_vertex[0][0]), int(q\_vertex[0][1])],
6. [int(q\_vertex[len(self.vertex)-1][0]), int(q\_vertex[len(self.vertex)-1][1])]]
7. temp = Line(vertex, 1, 1, 1)
8. self.pixels.extend(temp.get\_pixels())
9. **else**:
10. temp = Curve(q\_vertex, 1, 'Bezier', 1, 1)
11. self.pixels.extend(temp.get\_pixels())
12. **if** max([self.distance(point, [r\_vertex[0], r\_vertex[len(self.vertex)-1]]) **for** point **in** r\_vertex]) <= 1:
13. vertex = [[int(r\_vertex[0][0]), int(r\_vertex[0][1])],
14. [int(r\_vertex[len(self.vertex) - 1][0]), int(r\_vertex[len(self.vertex) - 1][1])]]
15. temp = Line(vertex, 1, 1, 1)
16. self.pixels.extend(temp.get\_pixels())
17. **else**:
18. temp = Curve(r\_vertex, 1, 'Bezier', 1, 1)
19. self.pixels.extend(temp.get\_pixels())

·评价：

Bezier曲线法能让设计者比较直观的意识到所给设计条件与终生成曲线之间的关系，能比较方便的通过修改输入参数来改变曲线的形状和阶次。Bézier 样条有很多性质使得在曲线和曲面设计中更有效且 更方便，也更容易使用。其离散生成相比固定步长生成，能够动态调整曲线中点的线密度，可以避免生成的曲线的点过于密集或者过于稀疏。

## 图元的平移

·算法原理

平移是指将物体沿直线路径从一个坐标位置移动到另一个坐标位置的变换，对于图元的直线、多边形的顶点、椭圆的圆心以及曲线的控制点，如下：

齐次表示为：

·代码实现

可以直接为端点加上平移的向量，然后重新生成图元的点。

1. **def** translate(self, x, y):
2. self.vertex = [[p[0] + x, p[1] + y] **for** p **in** self.vertex]
3. self.rasterization()

## 图元的旋转

·算法原理

二维旋转是将物体沿 xy 平面内的圆弧路径重定位。对于图元的直线、多边形的顶点和曲线的控制点，以原点为旋转中心的变换如下：

齐次表示为：

对于旋转中心不在原点的旋转变换，先将图元跟随旋转中心移动回原点，再进行旋转，然后再将旋转后的图元随旋转中心移动回原先的位置。

·代码实现

根据上述矩阵运算的结果，得到了以下代码。对于每一个顶点，先和旋转中心一起移动回原点，然后再旋转，然后再和旋转中心一起移动回原先的位置。

self.is\_changed表示当前是否正在被鼠标操作移动。若是，则对用浮点数记录的顶点进行操作，然后将其取整作为当前顶点，这样可以显著减小连续多次微小幅度的旋转所带来的误差。

1. **def** rotate(self, x0, y0, r):
2. # self.vertex = [[p[0], p[1]] for p in self.vertex]
3. **if** self.is\_changed:
4. **for** i **in** range(len(self.vertex\_float)):
5. x1 = self.vertex\_float[i][0]
6. y1 = self.vertex\_float[i][1]
7. self.vertex\_float[i][0] = x0 + (x1 - x0) \* math.cos(r/180\*math.pi) - (y1 - y0) \* math.sin(r/180\*math.pi)
8. self.vertex\_float[i][1] = y0 + (x1 - x0) \* math.sin(r/180\*math.pi) + (y1 - y0) \* math.cos(r/180\*math.pi)
9. # print(self.vertex)
10. # print("rotate ", r/180\*math.pi)
11. self.vertex = [[int(p[0]), int(p[1])] **for** p **in** self.vertex\_float]
12. **else**:
13. **for** i **in** range(len(self.vertex)):
14. x1 = self.vertex[i][0]
15. y1 = self.vertex[i][1]
16. self.vertex[i][0] = x0 + (x1 - x0) \* math.cos(r/180\*math.pi) - (y1 - y0) \* math.sin(r/180\*math.pi)
17. self.vertex[i][1] = y0 + (x1 - x0) \* math.sin(r/180\*math.pi) + (y1 - y0) \* math.cos(r/180\*math.pi)
18. self.rasterization()

## 图元的缩放

·算法原理

二维缩放变换改变物体的尺寸。对于图元的直线、多边形的顶点、曲线的控制点以及椭圆的外切矩形的四个顶点，以原点为旋转中心的变换如下

齐次变换如下：

对于缩放中心不在原点的缩放变换，先将图元跟随缩放中心移动回原点，再进行缩放，然后再将缩放后的图元随缩放中心移动回原先的位置。

·代码实现

基本同上节图元旋转，不在赘述。

1. **def** scale(self, x0, y0, s):
2. **if** self.is\_changed:
3. **for** i **in** range(len(self.vertex\_float)):
4. x1 = self.vertex\_float[i][0]
5. y1 = self.vertex\_float[i][1]
6. self.vertex\_float[i][0] = int((x1 - x0)\*s + x0)
7. self.vertex\_float[i][1] = int((y1 - y0)\*s + y0)
8. self.vertex = [[int(p[0]), int(p[1])] **for** p **in** self.vertex\_float]
9. **else**:
10. **for** i **in** range(len(self.vertex)):
11. x1 = self.vertex[i][0]
12. y1 = self.vertex[i][1]
13. self.vertex[i][0] = int((x1 - x0)\*s + x0)
14. self.vertex[i][1] = int((y1 - y0)\*s + y0)
15. self.rasterization()

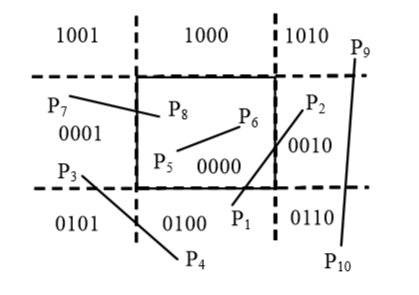
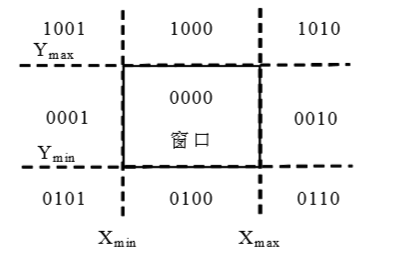
特殊地，椭圆缩放需要所放其外切矩形，代码如下：

1. **def** scale(self, x0, y0, s):
2. **if** self.rx == self.ry:
3. self.rx = int(s \* self.rx)
4. self.ry = int(s \* self.ry)
5. x1 = self.vertex[0][0]
6. y1 = self.vertex[0][1]
7. self.vertex[0][0] = int((x1 - x0) \* s + x0)
8. self.vertex[0][1] = int((y1 - y0) \* s + y0)
9. self.vertex[1] = [self.rx, self.ry]
10. self.rasterization()
11. **else**:
12. points = [[self.vertex[0][0] - self.rx, self.vertex[0][1] - self.ry],
13. [self.vertex[0][0] + self.rx, self.vertex[0][1] + self.ry]]
14. **for** i **in** range(len(points)):
15. x1 = points[i][0]
16. y1 = points[i][1]
17. points[i][0] = int((x1 - x0) \* s + x0)
18. points[i][1] = int((y1 - y0) \* s + y0)
19. self.vertex[0][0] = int((points[0][0] + points[1][0])/2)
20. self.vertex[0][1] = int((points[0][1] + points[1][1])/2)
21. self.vertex[1][0] = abs(int((points[0][0] - points[1][0])/2))
22. self.vertex[1][1] = abs(int((points[0][1] - points[1][1])/2))
23. self.rx = self.vertex[1][0]
24. self.ry = self.vertex[1][1]
25. self.rasterization()

## 线段的裁剪的Cohen-Sutherland算法

·算法原理

编码算法，即 Cohen-Sutherland 算法，是最早、最流行的线段裁剪算法。该算法采用区域检查的方法，能够快速有效地判断一条线段与裁剪窗口的位置关系，对完全接受或完全舍弃的线段无需求交，可以直接识别，大大减少求交的计算从而提高线段裁剪算法的速度。编码算法以下图所示的9个区域为基础，根据每条线段的端点坐标所在的区域，每个端点均赋以四位二进制码，称为区域码。区域码的各位表明线段端点对于 裁剪窗口的四个相对坐标位置.



线段完全在裁剪窗口之内时，两个端点的区域码均为 0000，则该线段完全在裁剪窗口之内。如上右图所示， 线段 P5P6 完全在裁剪窗口内。

线段完全在裁剪窗口之外时，两个端点的区域码相与后结果不为 0000，则该线段完全在裁剪窗口之外。如上右图所示，线段 P9P10 完全在裁剪窗口外，P9 端点的区域码为 1010，P10 端点的区域 码也为 0110，与操作后结果为 0010。在这种情况下，该线段可弃之。

其他情况上右图中 P1P2，P3P4，P7P8 都属于此类情况，需进行求交计算。虽然 P3P4 完全落在 窗口外，但由于没有简单的判断条件，也必须进行求交处理。求交过程为：首先，对一 条线段的外端点与一条裁剪边界比较来确定应裁剪掉多少线段。如上右图中 P7P8 线段相 对裁剪窗口左边界裁剪时，根据每个端点的区域码的第一位可以确定端点的位置性质。 P7 为外端点，即落在裁剪窗口左边界的左边，此点肯定不可见。P8 为内端点，即落在 裁剪窗口左边界的右边，为可能的可见点。 P7P8 线段与窗口左边界求交后，可以确定应 裁剪掉的线段为 P7 到 P7P8 线段与裁剪窗口 左边界的交点，而交点到 P8 线段为剩下部 分。然后，对线段的剩下部分与其他裁剪边界比较，直到该线段完全被舍弃或者找到位于窗口内的一段线段为止（即线段完全可见，则不需进一步判断）。算法可按上、下、右、左的顺序用裁剪边界检查线段的端点。在算法实现时，不必把线段与每条窗口边界 依次求交，只需按顺序检测到端点区域码的某位不为 0 时，才把线段与对应的窗口边界进行求交。

·代码实现

在Primitive类中的clip方法没有任何功能，仅提示“不可裁剪”

1. **def** clip(self, x1, y1, x2, y2, alg):
2. **print**('Primitive {} is not line. You can\'t clip it'.format(self.pno))

而该函数在Line类中进行了重写。

1. **def** clip(self, x1, y1, x2, y2, alg):
2. # print('clip line')
3. **if** alg == 'Cohen-Sutherland':...
4. **elif** alg == 'Liang-Barsky':...

这样，就只有线段才可以进行裁剪操作了。

首先调用Line.code()方法：

1. **if** alg == 'Cohen-Sutherland':
2. code = self.encode(x1, y1, x2, y2)

使用输入的两个裁剪边界，为直线的两个边界进行编码，函数如下

1. **def** encode(self, x1, y1, x2, y2):
2. xmax = max(x1, x2)
3. xmin = min(x1, x2)
4. ymax = max(y1, y2)
5. ymin = min(y1, y2)
6. code = [0, 0]
7. **if** self.vertex[0][0] < xmin:
8. code[0] = code[0] | 8
9. **elif** self.vertex[0][0] > xmax:
10. code[0] = code[0] | 4
11. **if** self.vertex[0][1] < ymin:
12. code[0] = code[0] | 2
13. **elif** self.vertex[0][1] > ymax:
14. code[0] = code[0] | 1
15. **if** self.vertex[1][0] < xmin:
16. code[1] = code[1] | 8
17. **elif** self.vertex[1][0] > xmax:
18. code[1] = code[1] | 4
19. **if** self.vertex[1][1] < ymin:
20. code[1] = code[1] | 2
21. **elif** self.vertex[1][1] > ymax:
22. code[1] = code[1] | 1
23. **return** code

编码完成之后，若两个顶点都在边界内，则不需要进行任何操作。若有一个顶点在裁剪边界内，则求唯一的顶点的交点作为裁剪过后的另一个顶点。根据编码，选择方向然后使用self.node()函数求交点即可。

1. **if** code[0]==0 **and** code[1] == 0:
2. **return**  # 啥也不做
3. **elif** code[0]&code[1] != 0:
4. self.is\_deleted = 1
5. **else**:
6. **if** code[1]==0:
7. code[1] = code[0]
8. code[0] = 0
9. temp = self.vertex[0]
10. self.vertex[0] = self.vertex[1]
11. self.vertex[1] = temp
12. **if** code[0]==0:
13. cross\_points = [self.vertex[0]]
14. res = []
15. **if** code[1]&8==8:
16. **print**("x min")
17. res = self.node(x1, y1, x2, y2, 1)
18. **elif** code[1]&4==4:
19. **print**("x max")
20. res = self.node(x1, y1, x2, y2, 2)
21. **if** code[1]&2==2:
22. **print**("y min")
23. res = self.node(x1, y1, x2, y2, 3)
24. **elif** code[1]&1==1:
25. **print**("y max")
26. res = self.node(x1, y1, x2, y2, 4)
27. **if** len(res) != 0:
28. cross\_points.append(res)
29. self.vertex = cross\_points
30. self.rasterization()

若两个顶点都在边界之外，则需要考虑两个或以上的边界。先将两个顶点的编码按为取或，然后根据编码的各个位来求交点。若交点符合要求就会放进新的顶点组中。最后必会得到两个顶点作为新的顶点，或者得到1个顶点，作为和裁剪边界的顶点的交点，或者得不到任何顶点，说明该线段已经被完全裁剪掉。

1. **else**:
2. cross\_points = []
3. cross = code[0] | code[1]
4. res = []
5. **if** cross & 8 == 8:
6. **print**("x min")
7. res = self.node(x1, y1, x2, y2, 1)
8. **if** len(res) != 0:
9. cross\_points.append(res)
10. **if** cross & 4 == 4:
11. **print**("x max")
12. res = self.node(x1, y1, x2, y2, 2)
13. **if** len(res) != 0:
14. cross\_points.append(res)
15. **if** cross & 2 == 2:
16. **print**("y min")
17. res = self.node(x1, y1, x2, y2, 3)
18. **if** len(res) != 0 **and** res **not** **in** cross\_points:
19. cross\_points.append(res)
20. **if** cross & 1 == 1:
21. **print**("y max")
22. res = self.node(x1, y1, x2, y2, 4)
23. **if** len(res) != 0 **and** res **not** **in** cross\_points:
24. cross\_points.append(res)
25. **if** len(cross\_points)==0:
26. self.is\_deleted = 1
27. **else**:
28. self.vertex = cross\_points
29. self.rasterization()

而self.node()函数的实现如下，首先求出原线段和给定边界的交点，然后根据裁剪边界，判断交点是否位于边界上，若是则返回，若不是则返回空值。

1. **def** node(self, x1, y1, x2, y2, pos):
2. xmax = max(x1, x2)
3. xmin = min(x1, x2)
4. ymax = max(y1, y2)
5. ymin = min(y1, y2)
6. x1 = self.vertex[0][0]
7. x2 = self.vertex[1][0]
8. y1 = self.vertex[0][1]
9. y2 = self.vertex[1][1]
10. res = [0, 0]
11. **if** pos==1:  # xmin
12. y = (y2-y1)/(x2-x1)\*(xmin-x1) + y1
13. y = int(y)
14. res = [xmin, y]
15. **if** pos == 2:  # xmax
16. y = (y2 - y1) / (x2 - x1) \* (xmax - x1) + y1
17. y = int(y)
18. res = [xmax, y]
19. **if** pos==3:  # ymin
20. x = (x2-x1)/(y2-y1)\*(ymin-y1) + x1
21. x = int(x)
22. res = [x, ymin]
23. **if** pos==4:  # ymax
24. x = (x2-x1)/(y2-y1)\*(ymax-y1) + x1
25. x = int(x)
26. res = [x, ymax]
27. **if** (res[0]>=xmin) **and** (res[0]<=xmax) **and** (res[1]>=ymin) **and** (res[1]<=ymax):
28. **return** res
29. **else**:
30. **return** []

·评价：

此算法对完全接受或完全舍弃的线段无需求交，可以直接识别，大大减少求交的计算从而提高线段裁剪算法的速度。

## 线段的裁剪的梁友栋-Barsky算法

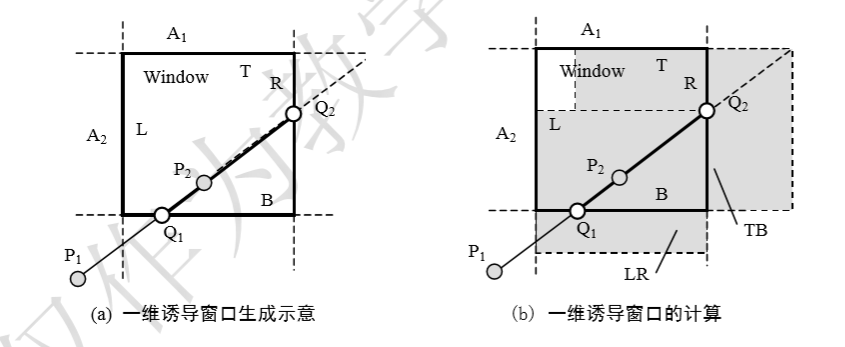
·算法原理

线段裁剪的基本问题是：裁剪窗口是二维对象；而线段是一维对象，两个对象维数不同不便比较。因此，他们所提出的算法的解决思路是：将待裁剪线段及裁剪矩形窗口均看作点集，那么裁剪结果即为两点集的交集。

设 P1P2所在直线为 L，记该直线(或其延长线)与裁剪窗口的两交点为 Q1Q2，称 Q1Q2为诱导窗口，它是一维的。这样，P1P2关于矩形窗口的裁剪结果与 P1P2关于诱导窗口 Q1Q2的裁剪结果是一致的，就将二维裁剪问题化简为一维裁剪问题。

转化为一维问题后，为解决二维裁剪问题，只要生成诱导窗口。图 3-14(a)中，P1P2 所在直线 Line 与窗口左、右、上、下四边界所在直线交点分别为 L、R、T 和 B；Q1Q2 为诱导窗口；记窗口左右边界所在直线夹成的带形区域为 A1；窗口上下边界所在直线夹 成的带形区域为 A2；窗口区域为 Window。那么，诱导窗口Q1Q2计算可如下：

其中：LR 和 TB 分别为在水平方向和垂直方向的参数区间。上式给出了 Q1Q2对应的参数区间，如图所示：



这样，P1P2 的可见部分 VW 可计算为

具体实现详见下面的代码。

·代码实现

首先初始化p和q的值：

1. **elif** alg == 'Liang-Barsky':
2. **print**('Liang-Barsky')
3. u1 = 0
4. u2 = 1
5. xmax = max(x1, x2)
6. xmin = min(x1, x2)
7. ymax = max(y1, y2)
8. ymin = min(y1, y2)
9. p = [0, 0, 0, 0]
10. q = [0, 0, 0, 0]
11. p[0] = -self.vertex[1][0] + self.vertex[0][0]
12. p[1] = -p[0]
13. p[2] = -self.vertex[1][1] + self.vertex[0][1]
14. p[3] = -p[2]
15. q[0] = self.vertex[0][0] - xmin
16. q[1] = xmax - self.vertex[0][0]
17. q[2] = self.vertex[0][1] - ymin
18. q[3] = ymax - self.vertex[0][1]
19. r = 0.0
20. flag = 0

然后，分别对四个方向计算u1和u2，若出现p=0&&q<0的情况，则说明要裁剪的线段再区域外，结束循环。

1. **for** i **in** range(4):
2. **if** p[i]<0:
3. r = q[i] / p[i]
4. u1 = max(u1, r)
5. **if** u1 > u2:
6. flag = 1
7. **break**
8. **if** p[i] > 0:
9. r = q[i] / p[i]
10. u2 = min(u2, r)
11. **if** u1 > u2:
12. flag = 1
13. **break**
14. **if** p[i]==0 **and** q[i] < 0:
15. flag = 1
16. **break**
17. **if** flag == 1:
18. self.is\_deleted = 1
19. **return**

最后，根据u1和u2确定裁剪过后的线段顶点。

1. **if** flag == 1:
2. self.is\_deleted = 1
3. **return**
4. **else**:
5. point1\_x = int(self.vertex[0][0] + u1\*(self.vertex[1][0] - self.vertex[0][0]))
6. point1\_y = int(self.vertex[0][1] + u1 \* (self.vertex[1][1] - self.vertex[0][1]))
7. point2\_x = int(self.vertex[0][0] + u2 \* (self.vertex[1][0] - self.vertex[0][0]))
8. point2\_y = int(self.vertex[0][1] + u2 \* (self.vertex[1][1] - self.vertex[0][1]))
9. self.vertex = [[point1\_x, point1\_y], [point2\_x, point2\_y]]
10. self.rasterization()

·评价

通常，梁友栋-Barsky 算法比 Cohen-Sutherland 算法更有效，因为需要计算的交点数目减少了，更新参数 u1、u2 仅仅需要一次除法；线段与窗口的交点仅需计算一次就能得出 u1、u2 的后值；相比之下，即使一条线段完全落在裁剪窗口之外，Cohen-Sutherland 算法也要对它反复求交点，而且每次求交计算都需要除和乘。

# 框架设计

本系统包括两个程序，分别使用文件输入进行绘图和使用鼠标交互进行绘图等操作，分别对应着CLI类和GUI类。在生成程序时，分别创建两个类中的一个类的一个对象，即可运行系统。本部分将详细描述各个部分的功能及其实现，并进行一些正确性、鲁棒性检测（待添加）以及效率分析。

## CLI设计

本部分以CLI类为基础，实现了通过文件输入进行绘图的功能。类的构造函数初始化了各个必要的数据成员，并将输入文件和保存路径作为参数传递给cmd\_line\_act()函数（下main详述），后者将解析文件中的指令并进行操作。

## 命令行文件的读取和执行

**CLI.refresh()**函数：

此函数作为刷新函数，在添加了新的图元或者对图元进行了修改过后被调用，功能是清除画布并重新逐个绘制图元。

1. **def** refresh(self):
2. self.image = Image.new("RGB", (self.size\_x, self.size\_y), (255, 255, 255))
3. self.draw = ImageDraw.Draw(self.image)
4. i = 0
5. **for** primitive **in** self.primitives:
6. pixels = primitive.get\_pixels()
7. **for** point **in** pixels:
8. **if** (point[0] >= 0) **and** (point[1] >= 0) **and** (point[0] <= self.size\_x-1) **and** (point[1] <= self.size\_y-1):
9. self.draw.point((point[0], point[1]), fill=primitive.get\_color())
10. i = i + 1

**CLI．cmd\_line\_act()**函数：

在此函数中，逐行读取文件中的指令，并将指令进行分割，

1. **for** cmd **in** cmd\_file.readlines():
2. color = [self.color\_r, self.color\_g, self.color\_b]
3. words = cmd.split()

并根据指令，对画布进行相应的操作：

·resetCanvas指令：直接修改**self.size\_x**和**self.size\_y,** 然后调用清空画布函数**self.clean\_pic(self)**;

·saveCanvas指令：将保存文件名与保存路径拼接，然后调用**self.save\_canvas(self)**函数；

·setColor指令：直接修改**self.color\_r, self.color\_g,self.color\_b**三个数据属性；

·drawLine,drawEllipse指令：利用得到顶点等信息，创建一个Line/Circle类的对象，并将其放入**self.primitives**列表中，然后调用**self.refresh()**对画布进行刷新操作；

1. **elif** words[0] == "drawLine":
2. vertex = [[int(words[2]), int(words[3])], [int(words[4]), int(words[5])]]
3. pid = int(words[1])
4. alg = 1 **if** words[6]=="DDA" **else** 2
5. line\_2b\_drawn = Line(vertex, pid, alg, color)
6. self.primitives.append(line\_2b\_drawn)
7. self.refresh()

·drawPolygon,drawCurve指令：这两个指令相对特殊，由两行组成，因此在循环中，当前行为这两个操作的时候，要修改一个临时变量lines，以方便下次循环读取多边形和曲线的点。此外，还需要几个临时变量来存储点的数量和图元id以及绘图算法：

1. **elif** words[0] == "drawPolygon":
2. lines = 1
3. line2\_id = int(words[1])
4. line2\_n = int(words[2])
5. line2\_alg = 1 **if** words[3]=="DDA" **else** 2

在下一次循环时的操作就和画直线和椭圆的操作相同了：

1. **elif** lines == 1:  # polygon
2. **print**("Polygon, 2nd line")
3. vertex = []
4. **for** i **in** range(line2\_n):
5. point = [int(words[2\*i]), int(words[2\*i+1])]
6. vertex.append(point)
7. polygon\_2b\_drawn = Polygon(vertex, line2\_id, line2\_alg, 1, color)
8. self.primitives.append(polygon\_2b\_drawn)
9. self.refresh()
10. lines = 0

·drawCurve指令操作类似，略去具体代码。

·translate指令：得到指令中的信息后，调用Primitive类的translate方法即可。

1. **elif** words[0] == "translate":
2. **print**("translate")
3. **for** i **in** range(len(self.primitives)):
4. **if** int(words[1]) == self.primitives[i].get\_id():
5. # num = i
6. self.primitives[i].translate(int(words[2]), int(words[3]))
7. **break**
8. self.refresh()

·rotate指令：得到指令中的信息后，调用Primitive类的rotate方法即可。

1. **elif** words[0] == "rotate":
2. **print**("rotate")
3. **for** i **in** range(len(self.primitives)):
4. **if** int(words[1]) == self.primitives[i].get\_id():
5. # num = i
6. self.primitives[i].rotate(int(words[2]), int(words[3]), int(words[4]))
7. **break**
8. self.refresh()

·scale指令：得到指令中的信息后，调用Primitive类的scale方法即可。

1. **elif** words[0] == "scale":
2. **print**("scale")
3. **for** i **in** range(len(self.primitives)):
4. **if** int(words[1]) == self.primitives[i].get\_id():
5. self.primitives[i].scale(int(words[2]), int(words[3]), int(words[4]))
6. **break**
7. self.refresh()

·clip指令：得到指令中的信息后，调用Primitive类的clip方法即可。

1. **elif** words[0] == "clip":
2. **print**("clip")
3. **for** i **in** range(len(self.primitives)):
4. **if** int(words[1]) == self.primitives[i].get\_id():
5. self.primitives[i].clip(int(words[2]), int(words[3]), int(words[4]), int(words[5]), words[6])
6. **break**
7. self.refresh()

## GUI设计

GUI分为菜单栏、按钮和画布。菜单栏中的“文件”包括清除画布、关闭程序以及调用上面部分的文件输入功能。“绘图”中可以选择直线、多边形、曲线的绘制算法以及线段的裁剪算法。

按钮包括了绘制各种图元、进行图元变换、打开命令行文件、清除画布以及退出程序的按钮。按下大部分按钮会调用GUI.set\_type()函数（下文详述）以改变当前鼠标的功能，而调色板会调用windows提供的调色版以改变画笔颜色，打开文件则会弹出窗口以选择文件和输出路径，清空画布和退出程序分别调用相关的函数。

画布是主体部分，鼠标交互在此中进行，白色部分为当前的画布，拖动白色部分的边缘可以调节画布大小，最大至填充满灰色区域。整个灰色部分为一个tkinter.canvas画布，而白色部分为一个PIL.Image对象。

GUI如下图：

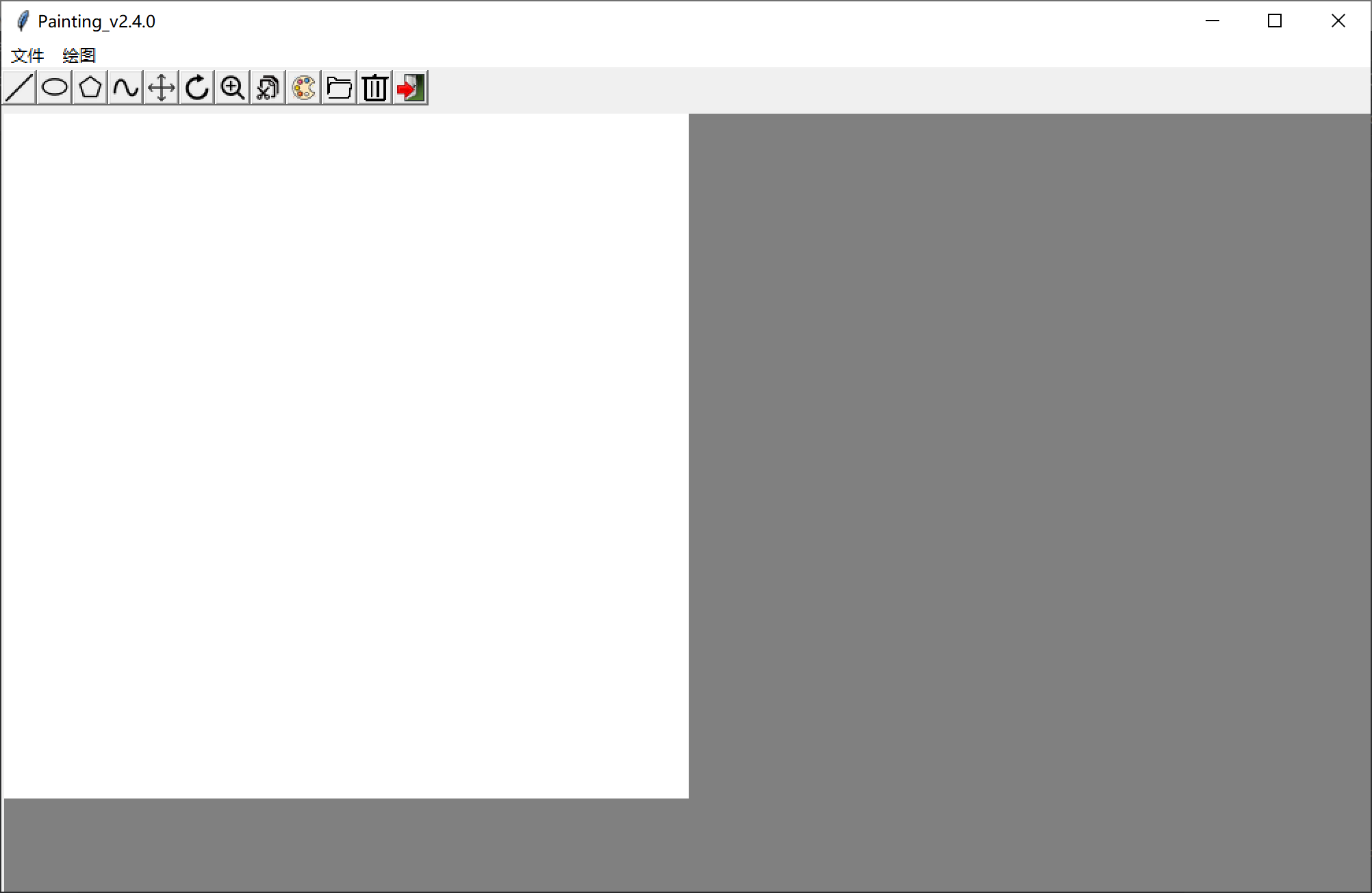


Figure 1

## 鼠标交互进行图元绘制及变换

首先是一些绘图共用接口：

·GUI.refresh(self): 用于在图中的部分图元有变动的时候，刷新画布。代码如下。

1. **def** refresh(self):
2. self.paper.delete(ALL)
3. self.image = Image.new("RGB", (self.size\_x, self.size\_y), (255, 255, 255))
4. self.draw = ImageDraw.Draw(self.image)
5. self.map = np.full((self.size\_x, self.size\_y), -1)
6. i = 0
7. **for** primitive **in** self.primitives:
8. pixels = primitive.get\_pixels()
9. **for** point **in** pixels:
10. **if** (point[0] >= 0) **and** (point[1] >= 0) **and** (point[0] <= self.size\_x-1) **and** (point[1] <= self.size\_y-1):
11. self.draw.point((point[0], point[1]), fill=primitive.get\_color())
12. **if** self.map[point[0]][point[1]] == -1:
13. self.map[point[0]][point[1]] = i
14. **else**:
15. self.map[point[0]][point[1]] = -2
16. i = i + 1
17. self.photo = ImageTk.PhotoImage(self.image)
18. self.paper.create\_image(2, 2, image=self.photo, anchor=NW)
19. **if** self.rotate\_point != [-1, -1]:
20. self.paper.create\_oval(self.rotate\_point[0]-2, self.rotate\_point[1]-2,
21. self.rotate\_point[0]+2, self.rotate\_point[1]+2,
22. fill='red')
23. **if** self.scale\_point != [-1, -1]:
24. self.paper.create\_oval(self.scale\_point[0]-2, self.scale\_point[1]-2,
25. self.scale\_point[0]+2, self.scale\_point[1]+2,
26. fill='green')

略去了部分后文中在特殊情况会使用的代码。首先抛弃旧的图片，创建新的图片。然后将self.primitives里的图元按照顺序，逐个获取其所有像素及其颜色，绘制在新的图片上。最后将新的图片粘贴在canvas上。

使用此方法刷新的效率理论上不高，尤其是在使用鼠标绘制图形的时候看似会有很大的延迟，不过经过测试，在画布中的线画图元数量小于80个时，能够保持每秒以上30帧的刷新率；而在接受效率损失之后，在对图元进行修改时，重叠部分的表现会更加准确：如，一个覆盖着其他图元的图元被裁剪掉覆盖着其他图元的一部分后，原先被覆盖的图元的这部分能够正确的显示；而在鼠标绘制图元时，在鼠标拖动的时候也能够方便的实现图元的预览。

·GUI.clean\_pic(self): 用于清除画布并重新设置画布大小。首先清空self.primitives,并清空self.paper（即图片所挂靠的canvas），然后调用self.refresh()函数，根据尺寸设置新的画布。以及重置旋转中心、缩放中心等等。

1. **def** clean\_pic(self):
2. self.paper.delete(ALL)
3. self.primitives = []
4. self.rotate\_point = [-1, -1]
5. self.scale\_point = [-1, -1]
6. self.primitive\_changing = -1
7. self.refresh()
8. self.is\_curve\_painting = 0
9. self.is\_polygon\_painting = 0
10. self.is\_rotating = 0

·GUI.save\_canvas(self):使用PIL.Image.save()函数来保存当前绘制的图片。

下面是绘图部分。

首先，定义三个鼠标的函数

1. **def** leftdown(self, event):
2. **def** leftmove(self, event):
3. **def** leftrelease(self, event):

并在\_\_init\_\_()函数中将其与画板（self.paper）绑定:

1. self.paper.bind('<Button-1>', self.leftdown)
2. self.paper.bind('<B1-Motion>', self.leftmove)
3. self.paper.bind('<ButtonRelease-1>', self.leftrelease)

另外，GUI类中还有几个转为鼠标绘制而准备的数据属性，其中，self.cur为正在绘制的图元的id，其数值等于self.primitives的长度；self.start\_x, self.start\_y为按下鼠标时记录的坐标，self.type为正在绘制的类型。

·绘图种类设置

首先，数据属性中添加几个按钮，并挂在主窗口中。其作用是选择要使用鼠标绘制的种类。self.line\_DDA = Button(self.top, command=**lambda**: self.set\_type(0), text="DDA直线")

1. self.line\_Bre = Button(self.top, command=**lambda**: self.set\_type(1), text="Bresenham直线")
2. self.owl = Button(self.top, command=**lambda**: self.set\_type(2), text="椭圆")
3. self.polygon = Button(self.top, command=**lambda**: self.set\_type(3), text="多边形")

其中，set\_type函数如下

1. **def** set\_type(self, type\_t):  # 设置鼠标画图的类型
2. self.type = type\_t
3. **if** type\_t>=4 **and** type\_t !=8:
4. self.pack\_dis\_ctrl\_point(1)
5. **else**:
6. self.pack\_dis\_ctrl\_point(0)
7. **if** self.is\_polygon\_painting == 1:  # 完成多边形的绘制
8. self.is\_polygon\_painting = 0
9. self.primitives[self.cur].done()
10. self.refresh()
11. **if** self.is\_curve\_painting == 1:
12. self.is\_curve\_painting = 0
13. self.refresh()
14. **if** self.is\_rotating:
15. self.is\_rotating = 0
16. self.rotate\_point = [-1, -1]
17. self.refresh()
18. **if** self.is\_scaling:
19. self.is\_scaling = 0
20. self.scale\_point = [-1, -1]
21. self.refresh()
22. **if** self.is\_clipping:
23. self.is\_clipping = 0
24. self.last\_point = [-1, -1]
25. self.clip\_point = [-1, -1]
26. self.refresh()

更改self.type数据属性，0和1分别为DDA算法和Bresenham算法的直线，2为椭圆，3为多边形，4为曲线，5为平移变换，6为旋转变换， 7为缩放变换， 8为裁剪。特殊的，如果正在绘制多边形、曲线或者正在进行旋转或者缩放变换的话，点击选择种类的按钮会使多边形和曲线绘制停止、删除当前的旋转或缩放中心并连接多边形第一个和最后一个点。

·直线的绘制

按下鼠标左键，会开始记录第一个坐标， 并以此坐标为起点和终点，在画布中创建一条直线，最后刷新画布。

1. **def** leftdown(self, event):
2. self.cur = self.primitives.\_\_len\_\_()  # 为即将要绘制的图元分配图元id
3. self.start\_x = event.x  # 记录第一个点的坐标
4. self.start\_y = event.y
5. color = [self.color\_r, self.color\_g, self.color\_b]
6. **if** self.type == 0 **or** self.type == 1:
7. temp\_list = [[self.start\_x, self.start\_y], [self.start\_x, self.start\_y]]
8. line\_being\_drawn = Line(temp\_list, self.primitives.\_\_len\_\_(), self.type+1, color)
9. self.primitives.append(line\_being\_drawn)

拖动鼠标左键，更新当前正在绘制的直线的两个点，第一个点是鼠标左键点击的时候记录下的点，第二个点是当前的点，重新计算图元的像素，并刷新画布，以实现拖动的过程中能够预览的效果。

1. **def** leftmove(self, event):
2. x = event.x
3. y = event.y
4. **if** self.type == 0 **or** self.type == 1:
5. temp\_list = [[self.start\_x, self.start\_y], [x, y]]  # 更新当前正在绘制的图元的点
6. self.primitives[self.cur].vertex = temp\_list
7. self.primitives[self.cur].rasterization()

松开鼠标，结束绘制，无需进行操作。

·椭圆的绘制：基本和直线类似

按下鼠标左键，开始画图；拖动鼠标时更新圆心坐标以及长半轴短半轴，并重新生成图元的像素；松开鼠标左键，结束绘制。

·多边形的绘制[[1]](#footnote-1)\*

多边形的绘制相对复杂，本程序多边形的绘制逻辑与windows画图程序类似，即，按下并拖动形成第一条直线，接下来点击或拖动形成后续的直线，最后按选择画图类型的按钮结束绘制并形成封闭多边形。

首先，需要GUI类的两个数据属性，self.is\_polygon\_painting和self.polygon\_last\_point，分别记录当前是否正在绘制多边形以及多边形上一个画过的点。

按下鼠标左键时，分为两种情况：若当前没有正在绘制多边形，则创建一个新的多边形，其中is\_done参数为0，然后self.is\_polygon\_painting设置为1。

若正在绘制，由于函数开头的语句将self.cur增加了1，因此要先将其减一，然后用当前点对多边形进行update。

1. **elif** self.type ==3:
2. **if** self.is\_polygon\_painting == 0:
3. polygon\_being\_drawn = Polygon([[self.start\_x, self.start\_y]], self.primitives.\_\_len\_\_(), 1, 0, color)
4. self.primitives.append(polygon\_being\_drawn)
5. self.is\_polygon\_painting = 1
6. **else**:
7. self.cur -= 1
8. self.primitives[self.cur].updating([event.x, event.y])

鼠标拖动时，只需要对使用当前点对图像进行update，此时多边形的一条边尚未绘制完成，拖动时经过的点不能加入多边形，只是显示出来以供预览，因此要使用updating()函数而非update\_rasterization()函数。刷新画布。

1. **elif** self.type == 3:
2. self.primitives[self.cur].updating([x, y])

松开鼠标左键，鼠标所在位置将被添加进多边形中，多边形的一条边绘制完毕。

1. **elif** self.type == 3:
2. self.primitives[self.cur].update\_rasterization([event.x, event.y])
3. self.polygon\_last\_point = [event.x, event.y]
4. self.refresh()

结束绘制多边形：任意点击一个选择类型即可。此功能在GUI.set\_type(self, type)函数中。

1. **if** self.is\_polygon\_painting == 1:  # 完成多边形的绘制
2. self.is\_polygon\_painting = 0
3. self.primitives[self.cur].done()
4. self.refresh()

·曲线的绘制

和多边形的绘制有些类似，

点击鼠标左键，若当前没有进行绘制，则创建一个以当前点为两个控制点的曲线；若当前曲线正在绘制，则调用Curve.begin\_update()方法将当前的点添加进曲线的控制点中。

1. **elif** self.type==4:  # curve
2. **if** self.is\_curve\_painting == 0:
3. curve\_being\_drawn = Curve([[self.start\_x, self.start\_y],[self.start\_x, self.start\_y]],
4. self.primitives.\_\_len\_\_(),
5. ('Bezier' **if** self.curve\_type==1 **else** 'B-spline'), 0, color)
6. self.primitives.append(curve\_being\_drawn)
7. self.is\_curve\_painting = 1
8. **else**:
9. self.cur -= 1
10. self.primitives[self.cur].begin\_update([event.x, event.y])
11. self.refresh()

拖动鼠标左键，调用Curve.updating()方法，将控制点中的的最后一个，也就是刚刚拖动经过的点修改为现在的点，并对图元进行刷新。

1. **elif** self.type == 4:
2. self.primitives[self.cur].updating([x, y])

松开鼠标左键，同拖动，无需进行操作，当前点会永久添加进入当前正在绘制的曲线的控制点中。

1. **elif** self.type == 4:
2. self.primitives[self.cur].updating([event.x, event.y])
3. self.refresh()

·图元的平移

对于使用鼠标事件实现图元的变换，这里需要在GUI类中添加新的数据属性self.map，是一个二维矩阵，大小和画布大小相同，矩阵中的元素存放着占用当前点的图元的下标，若没有被占用则为-1，若被多个图元占用则为-2，self.map的更新在self.refresh()函数中，即每次刷新都要更新self.map矩阵。这样的矩阵可以使得鼠标事件来实现图元的变换成为可能。

另外，在鼠标左键点击的函数中定义一个find()函数，用来确定当前的鼠标位置对应着哪一个图元。由于很难实现精准的点击，因此将误差设置为四周的五个像素。函数实现如下：

1. **def** find(point):
2. res = -1
3. **for** i **in** range(point[0] - 5, point[0] + 5):
4. **for** j **in** range(point[1] - 5, point[1] + 5):
5. **if** (i >= 0) **and** (i <= 499) **and** (j > 0) **and** (j <= 499):
6. **if** self.map[i][j] >= 0:
7. **if** res == -1:
8. res = self.map[i][j]
9. **elif** self.map[i][j] != res:
10. res = -1
11. **return** res
12. **print**("select ", res)
13. **return** res

鼠标事件实现图元的平移本身比较简单，鼠标左键点击时，选择图元，若未找到则无响应，若找到了则将正在移动的图元设置为此图元。

1. **elif** self.type ==5:
2. self.primitive\_changing = find([event.x, event.y])
3. **if** self.primitive\_changing >= 0:
4. self.is\_translating = 1
5. self.last\_point = [event.x, event.y]

鼠标拖动时，调用Primitives类的translate函数，计算当前坐标与上一个坐标的差值并进行移动。

1. **elif** self.type == 5 **and** self.is\_translating == 1:
2. self.primitives[self.primitive\_changing].translate(x - self.last\_point[0], y - self.last\_point[1])
3. self.last\_point = [x, y]

鼠标松开时，结束平移。

1. **elif** self.type == 5:
2. self.is\_translating = 0
3. self.primitive\_changing = -1

·图元的旋转

首先，第一次点击时，点击一个位置来选择旋转中心，当再次选择了绘图或者变换类型时，旋转中心会被删除。然后后续操作与平移类似。鼠标左键点击时， 选择图元并记录当前角度。需要注意的时arctan函数的定义域和值域，所以会函数中考虑了一些特殊情况。另外，在选择图元之后，还调用了Primitive.change()函数，此函数将图元的顶点以浮点数记录，可以显著提高连续旋转以及缩放时的精度。

1. **elif** self.type == 6:
2. **print**("rotate")
3. **if** self.is\_rotating == 0:
4. self.rotate\_point = [event.x, event.y]
5. self.is\_rotating = 1
6. self.refresh()
7. **else**:
8. self.primitive\_changing = find([event.x, event.y])
9. **if** self.primitive\_changing >= 0:
10. # self.is\_rotating = 1
11. self.primitives[self.primitive\_changing].change(1)
12. **if** event.x == self.rotate\_point[0]:
13. self.start\_angle = math.pi/2 **if** event.y - self.rotate\_point[1] > 0 **else** -math.pi/2
14. **else**:
15. self.start\_angle = math.atan((event.y - self.rotate\_point[1])/(event.x - self.rotate\_point[0]))
16. **if** event.x - self.rotate\_point[0] < 0:
17. self.start\_angle = self.start\_angle + math.pi

鼠标拖动时，计算当前角度和前一个点的差值调用Primitive.rotate()函数来进行旋转，并记录当前角度。

1. **elif** self.type == 6 **and** self.primitive\_changing != -1:
2. **if** x == self.rotate\_point[0]:
3. angle = math.pi/2 **if** y - self.rotate\_point[1] > 0 **else** -math.pi/2
4. **else**:
5. angle = math.atan((y - self.rotate\_point[1])/(x - self.rotate\_point[0]))
6. **if** x - self.rotate\_point[0] < 0:
7. angle = angle + math.pi
9. self.primitives[self.primitive\_changing].rotate(self.rotate\_point[0], self.rotate\_point[1],
10. (angle - self.start\_angle)\*180 / math.pi)
11. self.start\_angle = angle

鼠标松开时，结束旋转，并调用Primitive.change()函数删除图元中以浮点数存放的顶点。

1. **elif** self.type == 6:
2. self.primitive\_changing = -1
3. self.primitives[self.primitive\_changing].change(0)

·图元的缩放

图元缩放的实现与旋转基本相同，鼠标左键点击时，先选择一个点作为缩放中心，然后再选择图元进行缩放。

1. **elif** self.type == 7:
2. **if** self.is\_scaling == 0:
3. self.scale\_point = [event.x, event.y]
4. self.is\_scaling = 1
5. self.refresh()
6. **else**:
7. self.primitive\_changing = find([event.x, event.y])
8. self.primitives[self.primitive\_changing].change(1)
9. **if** self.primitive\_changing >= 0:
10. **print**('a')
11. self.start\_distance = math.sqrt(pow(event.x - self.scale\_point[0], 2) +
12. pow(event.y - self.scale\_point[1], 2))

鼠标移动时，计算差值、缩放并记录当前的缩放倍数。

1. **elif** self.type == 7 **and** self.primitive\_changing != -1:
2. cur\_dis = math.sqrt(pow(event.x - self.scale\_point[0], 2) +
3. pow(event.y - self.scale\_point[1], 2))
4. self.primitives[self.primitive\_changing].scale(self.scale\_point[0], self.scale\_point[1],
5. cur\_dis/self.start\_distance)
6. self.start\_distance = cur\_dis

鼠标松开时，结束缩放并调用Primitive.change()函数。

1. **elif** self.type == 7:
2. self.primitive\_changing = -1
3. self.primitives[self.primitive\_changing].change(0)

·图元的裁剪

第一次点击时，选择一个图元。若被选择的图元是直线，则在图元两端进行标记。（GUI。refresh()函数中）：

1. **if** self.type == 8 **and** self.primitive\_changing != -1 **and** self.primitives[self.primitive\_changing].\_\_class\_\_.\_\_name\_\_=='Line':
2. vertex = self.primitives[self.primitive\_changing].get\_vertexes()
3. self.paper.create\_oval(vertex[0][0] - 2, vertex[0][1] - 2,
4. vertex[0][0] + 2, vertex[0][1] + 2,
5. fill='blue')
6. self.paper.create\_oval(vertex[1][0] - 2, vertex[1][1] - 2,
7. vertex[1][0] + 2, vertex[1][1] + 2,
8. fill='blue')

第二次点击下时，记录下裁剪区域的第一个顶点。拖动时，得到裁剪区域的另一个顶点。创建一个与需要被裁剪的直线相同的临时直线，对其进行裁剪。

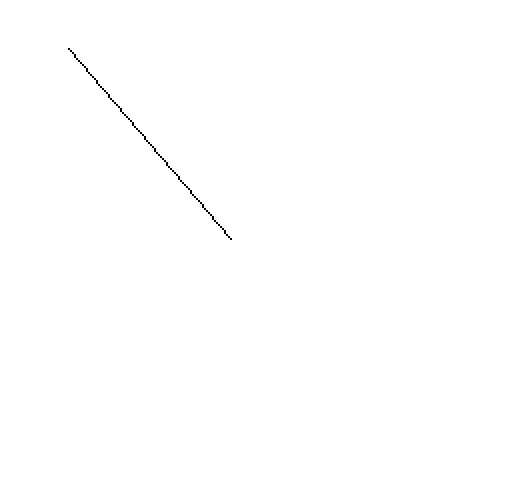
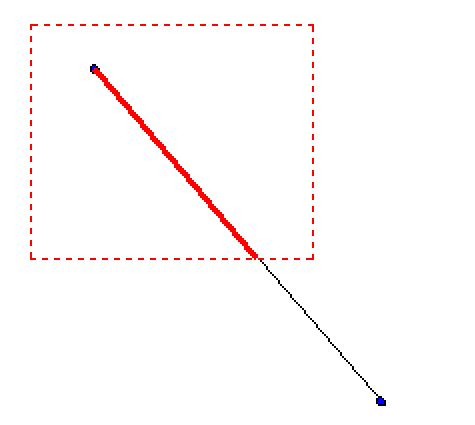
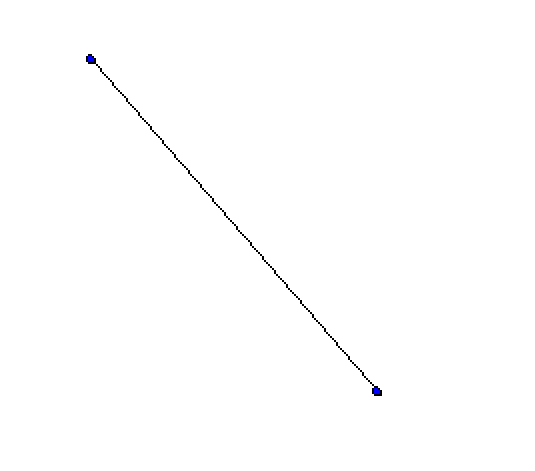
1. **elif** self.type == 8 **and** self.primitive\_changing != -1 **and** self.is\_clipping==1:
2. self.clip\_point = [x, y]
3. self.tmp\_cut\_line = Line(self.primitives[self.primitive\_changing].get\_vertexes(), -1,
4. self.primitives[self.primitive\_changing].get\_method(), 0)
5. **print**(self.primitives[self.primitive\_changing].get\_vertexes())
6. self.tmp\_cut\_line.clip(self.last\_point[0], self.last\_point[1],
7. self.clip\_point[0], self.clip\_point[1], self.clip\_alg)
8. self.clipped = 1

在GUI.refresh()函数中刷新时，高亮显示临时直线的裁剪效果，并用虚线绘制当前的裁剪区域。

1. **if** self.type == 8 **and** self.primitive\_changing != -1 **and** self.is\_clipping==1:
2. vertex = self.tmp\_cut\_line.get\_vertexes()
3. **if** self.tmp\_cut\_line.is\_deleted != 1:
4. self.paper.create\_line(vertex[0][0], vertex[0][1], vertex[1][0], vertex[1][1], fill='red', width=3)
5. vertex = [self.last\_point, self.clip\_point]
6. xmax = max(vertex[0][0], vertex[1][0])
7. xmin = min(vertex[0][0], vertex[1][0])
8. ymax = max(vertex[0][1], vertex[1][1])
9. ymin = min(vertex[0][1], vertex[1][1])
10. x = [xmin, xmin, xmax, xmax]
11. y = [ymin, ymax, ymax, ymin]
12. **for** i **in** range(4):
13. self.paper.create\_line(x[i], y[i], x[(i+1)%4], y[(i+1)%4], fill='red', dash=(4, 4))

松开鼠标时，即即对线段本身进行裁剪。

·线段裁剪实现效果：



总体实现效果：

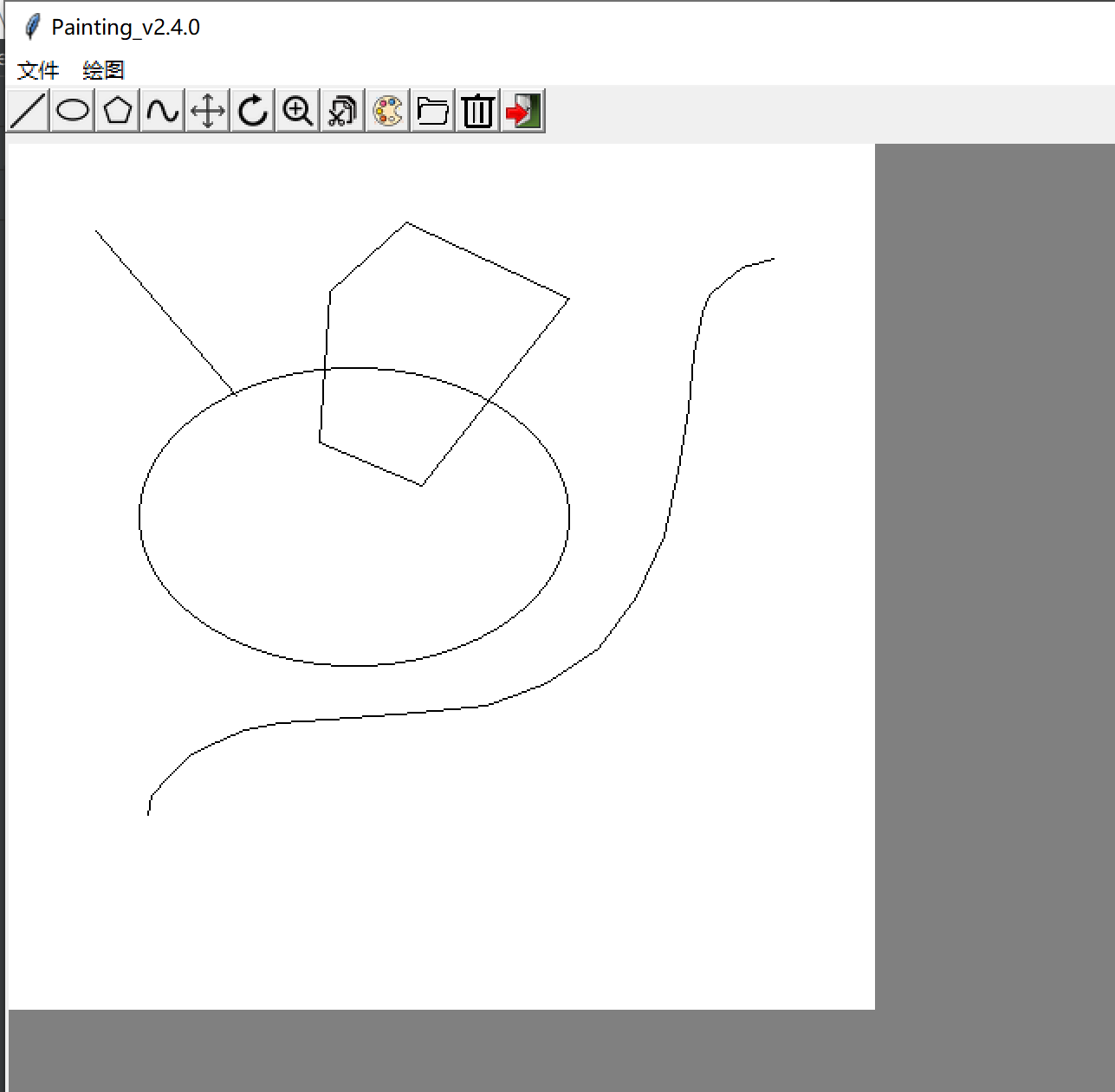


Figure 3

# 其他功能

（待添加）

References:

1. [Python Tkinter Button控件command传参问题解决方案 - weixin\_41098458的博客 - CSDN博客](https://blog.csdn.net/weixin_41098458/article/details/81781930)
2. [Tkinter选择路径功能的实现 - Clew123的博客 - CSDN博客https://blog.csdn.net/zjiang1994/article/details/53513377/](https://blog.csdn.net/zjiang1994/article/details/53513377/)
3. <https://stackoverflow.com/questions/11529170/askdirectory-changes-focus-to-different-window>

1. \* :其中，Polygon类的部分接口在3.4节详细描述，下段Curve类的接口在3.5节详细描述。 [↑](#footnote-ref-1)