《计算机图形学》系统报告

181860154，朱倩，infinite0124@163.com

2020 年 12 月 14 日

目录

1 综述

本课程作业使用Python3语言编程，通过实现各种图形学算法，实现一个绘图系统。该绘图系统包含的功能和涉及的算法如下：

- 设置画笔，重置画布，保存画布
- 绘制线段（DDA算法、Bresenham算法）
- 绘制多边形（DDA算法、Bresenham算法）
- 绘制椭圆（中点椭圆生成算法）
- 绘制曲线（Bezier算法，B-spline算法）
- 图元平移，图元旋转，图元缩放
- 对线段裁剪（Cohen-Sutherland算法、Liang-Barsky算法）

完成进度描述：

- 9月：完成开发环境的配置；实现DDA算法，补充CLI和GUI对其的相关调用
- 10月：实现Bresenham算法，多边形绘制函数，中心椭圆生成算法，补充CLI和GUI对其的相关调用
- 11月：实现曲线绘制的Bezier算法，图形变换的平移、旋转、缩放，线段裁剪的Cohen-Sutherland算法，补充CLI对其的相关调用
- 12月：实现三次（四阶）均匀B样条曲线绘制算法、Liang-Barsky线段裁剪算法；实现GUI界面的设置画笔、重置画布、保存画布，曲线绘制，平移、旋转、缩放、线段裁剪操作；增加附加功能。

2 算法介绍

2.1 线段绘制算法

2.1.1 DDA算法

数字差分分析(DDA：Digital Differential Analyzer)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最靠近线段路径的对应整数值。

该算法主要是根据直线公式$y = kx + b$推导而来，其关键之处在于如何设定单位步进，即一个方向的单位步进为1，另一个方向的单位步进必然是小于1。

算法的具体过程描述如下：

1. 输入直线的起点坐标(x_0, y_0)，终点坐标(x_1, y_1)；初始化数组$result$为空，用于记录线段涉及的像素点坐标。

2. 计算X方向的间距:$\Delta x = x_1 - x_0$，Y方向的间距：$\Delta y = y_1 - y_0$.

3. 确定单位步进，取$maxDis = max(|\Delta x|, |\Delta y|)$。若$|\Delta x| \geq |\Delta y|$，则X方向的单位步进为1，Y方向的单位步进为$\frac{\Delta y}{maxDis}$；否则相反。

4. 令循环初始值为0，循环次数为$maxDis + 1$，定义变量$x = x_0$，$y = y_0$，执行如下操作：

   - 添加点$(int(x), int(y))$到结果$result$数组中。
   - x增加一个单位步进，y增加一个单位步进

该算法的优点在于：通过在X和Y方向使用合适的增量来逐步沿线的路径推出各像素位置，将坐标计算中的乘法转换成了加法，因此计算像素位置要比按照直线斜率计算坐标要快。

2.1.2 Bresenham算法

1. Bresenham画线算法的基本原理

Bresenham算法是一种精确而有效的光栅线段生成算法，可用于圆和其他曲线显示的整数增量运算。本质上也是采取了步进的思想，不过相比DDA算法有所优化，避免了步进时浮点数运算。

首先通过直线的斜率确定在 x 方向进行单位步进还是 y 方向进行单位步进：当斜率 k 的绝对值$|k| < 1$时，在 x 方向进行单位步进；当斜率 k 的绝对值$|k| > 1$时，在 y 方向进行单位步进。

下面以$|k| < 1$为例推导Bresenham算法的数学依据：

已知有一直线 y = kx+b ，$|k| < 1$。此时通过斜率确定了x方向为单位步进。从给定线段的左端点所在像素位置(x_0, y_0)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 值最接近线段的像素，并逐次绘出。

假设这个过程进行到第m步，即$x = x_m$，$y = y_m$时。那么当x执行一个单位步进时（即$x = x_{m+1}$时），y等于y_m还是等y_{m+1}更符合这个直线方程呢？此时可以通过比较y_m和y_{m+1}和真实的方程的 y 值的差是多少，看看哪一个更靠近真实的方程的y值，即像素列位置x_{m+1}处数学线段上的 y 坐标。

而像素列位置x_{m+1}处数学线段上的 y 坐标可计算为：

$$y_{real} = kx_{m+1} + b = k(x_m + 1) + b \tag{1}$$

用d_1和d_2来分别表示两个候选像素与线段数学路径的垂直偏移,可得：

$$\begin{cases} d_1 = y_{real} - y_m = k(x_m + 1) + b - y_m \\ d_2 = y_{m+1} - y_{real} = y_{m+1} - k(x_m + 1) - b \end{cases}$$

这两个分离点的距离差分为：

$$d_1 - d_2 = 2m(x_m + 1) - 2y_m + 2b - 1 \tag{2}$$

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，而整型参量符号的检测通过决策参数实现。

设Δx 为线段x方向的间距，Δy为线段y方向的间距，则$k = \Delta y/\Delta x$，可得第m步的决策参数：

$$p_m = \Delta x(d_1 - d_2) = 2\Delta y x_m - 2\Delta x y_m + c \tag{3}$$

其中$c = 2\Delta y + \Delta x(2b - 1)$是一常量。

由于x方向单位增量$\Delta x = 1 > 0$，p_m的符号与$d_1 - d_2$相同。假如y_m处的像素比y_{m+1}的像素更接近于线段(即$d_1 < d_2$)，那么参数p_m是负的，此时，选择绘制y_m处的像素；反之，y_{m+1}处的像素比y_m的像素更接近于线段(即$d_1 > d_2$)，那么，参数p_m是正的，此时，选择绘制y_{m+1}处的像素。也就是说，可以根据p_m的符号来决定第 m+1 步所需要选择的像素。

2. Bresenham线生成过程

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继的决策参数值。下面以$|k| < 1$为例推导决策参数的计算过程：

在 m+1 步，决策参数可从上述p_m方程中计算出为：

$$p_{m+1} = 2\Delta y \cdot x_{m+1} - 2\Delta x \cdot y_{m+1} + c \tag{4}$$

将上述方程减去方程(3)，可得到：

$$p_{m+1} - p_m = 2\Delta y(x_{m+1} - x_m) - 2\Delta x(y_{m+1} - y_m) \tag{5}$$

但$x_{m+1} = x_m + 1$，因而得到：

$$p_{m+1} = p_m + 2\Delta y - 2\Delta x(y_{m+1} - y_m) \tag{6}$$

其中：$y_{m+1} - y_m$的取值取决于参数p_m的符号，或者说，取决于像素的选择。规则如下：

$$\begin{cases} p_m > 0, y_{m+1} = y_m + 1 \\ p_m < 0, y_{m+1} = y_m \end{cases}$$

即，当$p_m > 0$时，y_{m+1}取高像素，$y_{m+1} = y_m + 1$，此时，决策参数为：

$$p_{m+1} = p_m + 2\Delta y - 2\Delta x \tag{7}$$

反之，当$p_m < 0$时，y_{m+1}取低像素，$y_{m+1} = y_m$，此时，决策参数为：

$$p_{m+1} = p_m + 2\Delta y \tag{8}$$

从线段某个坐标端点开始，在每个离散整数 x 位置，反复进行决策参数的这种递归运算，就可得到整个线段上的所有离散点集。而在起始像素位置(x_0, y_0)的第一个参数p_m可从方程(3)及$k = \Delta y/\Delta x$计算出：

$$p_0 = 2\Delta y - \Delta x \tag{9}$$

总的生成算法过程如下：

1) 输入线段的起点和终点，将左端点存储在(x_0, y_0)中。

2) 判断线段的斜率是否存在（即起点和终点的 x 坐标是否相同），若相同，即斜率不存在，只需在 y 方向进行单位步进$\Delta y + 1$次， x 方向的坐标保持不变即可绘制直线。

3) 计算线段的斜率k，分为下面几种情况处理:

   - k等于0，即线段平行于 x 轴，即程序只需在 x 方向进行单位步进$\Delta x + 1$次， y 方向的值不变
   - $|k|$等于1，即线段的 x 方向的单位步进和 y 方向的单位步进一样，皆为1。直接循环Δx次计算 x 和 y 坐标。

4) 根据输入的起点和终点的 x 、 y 坐标值的大小决定 x 方向和 y 方向的单位步进是1还是-1

5) 将(x_0, y_0)装入帧缓冲器，画出第一个点。

6) 若$|k| < 1$:

   a. 将m初始化为0，并计算决策参数的第一个值：$p_0 = 2\Delta y - \Delta x$.

   b. 如果$p_m < 0$，则下一个要绘制的点为（x_m+单位步进，y_m），$p_{m+1} = p_m + 2\Delta y$;否则要绘制的点为（$x_m$+单位步进，$y_m$+单位步进）,$p_{m+1} = p_m + 2\Delta y - 2\Delta x$；

   c. $m = m + 1$

   d. 回到步骤b，重复执行Δx次；

   若$|k| > 1$:

   a. 将m初始化为0，并计算决策参数的第一个值：$p_0 = 2\Delta x - \Delta y$.

   b. 如果$p_m < 0$，则下一个要绘制的点为（x_m，y_m+单位步进），$p_{m+1} = p_m + 2\Delta x$;否则要绘制的点为（$x_m$+单位步进，$y_m$+单位步进）,$p_{m+1} = p_m + 2\Delta x - 2\Delta y$；

   c. $m = m + 1$

   d. 回到步骤b，重复执行Δy次；

2.2 中点椭圆生成算法

1. 中心椭圆生成算法的基本原理

椭圆的曲线的生成可通过考虑椭圆沿长轴和短轴尺寸不同而修改画圆程序来实现。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x_r标识长半轴，参数y_r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。

标准椭圆方程可借助于椭圆中心坐标(x_c, y_c)和参数r_x和r_y写为：

$$(\frac{x-x_c}{r_x})^2 + (\frac{y-y_c}{r_y})^2 = 1 \tag{10}$$

为了简化，可以先考虑圆心在原点的椭圆的生成，对于中心不是原点的椭圆，可以通过坐标的平移变换获得相应位置的椭圆。

中点椭圆方法依据椭圆斜率$(r_y < r_x)$将第一象限的椭圆分成两部分，即区域 1 (切线斜率小于1）和区域 2 （切线斜率大于1），如图1所示:

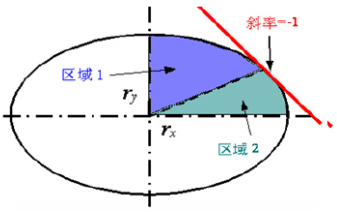


图 1: 第一象限椭圆区域划分

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：

$$d_y/d_x = -2r_y^2 x / r_x^2 y \tag{11}$$

在区域 1 和区域 2 的交界区：$d_y/dx = -1$，由此可得，区域 1 和区域 2 的分割条件为：$2r_y^2 x = 2r_x^2 y$，而移出区域 1 进入区域 2 的条件为$2r_y^2 x \geq 2r_x^2 y$。

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这个象限。区域 1 和区域 2 可以多种方式来处理：可以从位置$(0, r_y)$开始，在第一象限内沿椭圆路径顺时针步进，当斜率变为小于 -1 时从 x 方向的单位步长转化为 y 方向的单位步长；也可以从$(r_x, 0)$开始，以逆时针方式选取点，并当斜率为小于 -1 时将 y 方向的单位步长改为 x 方向单位步长。

给定参数r_y, r_x和(x_c, y_c)，先确定中心在原点的标准位置的椭圆点(x, y)，然后将点变换为中心在(x_c, y_c)的点。

假设$r_y \leq r_x$，取$(x_c, y_c) = (0, 0)$，定义椭圆函数为：

$$f_{ellipse}(x, y) = r_y^2 x^2 + r_x^2 y^2 - r_x^2 r_y^2 \tag{12}$$

该函数具有下列特性：

$$\begin{cases} f_{ellipse}(x, y) < 0, & \text{(x,y)位于椭圆周边界内;} \\ f_{ellipse}(x, y) = 0, & \text{(x,y)位于椭圆周边界上;} \\ f_{ellipse}(x, y) > 0, & \text{(x,y)位于椭圆周边界外.} \end{cases}$$

椭圆函数$f_{ellipse}(x,y)$作为中点椭圆生成算法的决策参数。在每个取样位置，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。

假设$r_y < r_x$，从$(0,r_y)$开始，在 x 方向取单位步长步进到区域 1 和区域 2 之间的边界，而后转换成 y 方向的单位步长通过第一象限中剩余的曲线段。

⑴. 区域 1 中(|切线斜率|≤1)

假如在前一步中选择了位置(x_k,y_k)，将第一象限内取样位置x_{k+1}处两个候选像素间中点对决策参数(即椭圆函数)求值：

$$p1_k = f_{ellipse}(x_{k+1}, y_k - \frac{1}{2} = r_y^2(x_k+1)^2 + r_x^2(y_k - \frac{1}{2})^2 - r_x^2 r_y^2 \tag{13}$$

假如$p1_k < 0$，中点位于椭圆内，扫描线y_k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$(y_k - 1)$上。

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

⑵. 区域 2 中(|切线斜率|>1)

当进入区域 2 时，其初始点取区域 1 中选择的最后位置。在区域 2 中，在负 y 方向以单位步长取样，在每一步中取水平像素间的中点对决策参数求值为：

$$p2_k = f_{ellipse}(x_k + \frac{1}{2}, y_k - 1) = r_y^2(x_k + \frac{1}{2})^2 + r_x^2(y_k - 1)^2 - r_x^2 r_y^2 \tag{14}$$

假如$p2_k > 0$，中点位于椭圆边界之外，选择像素x_k；假如$p2_k \leq 0$，中点位于椭圆边界之内或之上，选择像素x_{k+1}。

$$\begin{cases} p2_{k+1} = p2_k - 2r_x^2 y_k + 3r_x^2, if p_k \leq 0; \\ p2_{k+1} = p2_k + 2r_y^2 x_k - 2r_x^2 y_k + 2r_y^2 + 3r_x^2, if p_k > 0. \end{cases}$$

而

$$p_0 = r_y^2 - r_x^2 r_y + \frac{r_x^2}{4} \tag{15}$$

2. 中心椭圆生成算法的具体生成过程

1) 输入r_x、r_y和(x_c,y_c),得到中心在原点的椭圆的第一个点：$(x_0,y_0) = (0,r_y)$；

2) 计算区域 1 决策参数初值：

$$p1_0 = r_y^2 - r_x^2 r_y + r_x^2/4$$

3) 对于区域 1 每个x_k位置，$k=0$开始循环测试：

- $pl_k < 0$：选择像素(x_k+1, y_k)，计算$p1_{k+1} = p1_k + 2r_y^2 x_{k+1} + r_y^2$.
- $pl_k > 0$：选择像素(x_k+1, y_k-1)，计算$p1_{k+1} = p1_k + 2r_y^2 x_{k+1} - 2r_x^2 y_{k+1} + ry^2$.
- 循环至：$2r_y^2 x >= 2rx^2 y$

4) 设区域 1 最后一个点为(x_1,y_1)，计算区域 2 参数初值：

$$p2_0 = r_y^2(x_1 + \frac{1}{2})^2 + r_x^2(y_1 - 1)^2 - r_x^2 r_y^2$$

5) 对于区域 2 每个y_k位置处，$k=0$开始循环检测：

- $p2_k>0$：选择像素(x_k,y_k-1)，计算$p2_{k+1}=p2_k-2r_x^2y_{k+1}+r_x^2$.
- $p2_k<0$：选择像素(x_k+1,y_k-1)，计算$p2_{k+1}=p2_k+2r_y^2x_{k+1}-2r_x^2y_{k+1}+rx^2$.
- 循环至：$(r_x,0)$

6) 对称：根据第一象限生成的点坐标确定其他三个象限对称点

7) 平移：将每个像素位置(x,y)平移到中心在(x_c,y_c)的椭圆轨迹上，并按坐标值画点：$x=x+x_c,y=y+y_c$.

2.3 曲线绘制算法

2.3.1 Bezier算法

1. Bézier 曲线的定义和性质

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，Bézier 曲线段可拟合任何数目的控制顶点。Bézier 曲线段逼近这些控制顶点，且它们的相关位置决定了Bézier 多项式的次数。类似插值样条，Bézier 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Bézier 曲线，最方便的是混合函数形式。

假设给出 n+1 个控制顶点位置：$P_i=(x_i,y_i,z_i)(i=0,1,2,...n)$。这些坐标点混合产生下列位置向量$P(u)$，用来描述$P_0$ 和P_n 间的逼近Bézier 多项式函数的路径。

$$P(u)=\sum_{i=0}^{n}P_iBEZ_{i,n}(u)(0\le u\le 1)$$

其中，混合函数$BEZ_{i,n}(u)$是n 次Bernstein 多项式。图6-15(a)示出了Bézier曲线。

利用Bernstein 基函数的降(升)阶公式，得出Bézier 曲线上点的坐标位置的有效方法是使用递归计算。用递归计算定义的Bézier 混合函数为：

$$BEZ_{i,n}(u)=(1-u)BEZ_{i,n-1}(u)+uBEZ_{i-1,n-1}(u)$$

其中，$BEZ_{i,i}(u)=u^i$，$BEZ_{0,i}(u)=(1-u)^i$。

由Bézier 曲线定义可知：一条n 次Bézier 曲线被表示成它的n+1 个控制顶点的加权和，权是Bernstein 基函数。

2. Bézier 曲线生成的分割递推算法

给定任一参数u，计算设定阶次的曲线上对应点的坐标可以直接利用曲线方程或矩阵形式进行计算，但该方法不通用且计算工作量较大。而德卡斯特里奥(de Casteljau)递推算法产生曲线上的点相对而言要简单得多。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：

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

可以看出：当r=0 时，计算结果为控制顶点本身；而曲线上的型值点为：$P(u) = P_0^n$。图2为计算三次Bézier 曲线上点P(u)的计算过程：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。

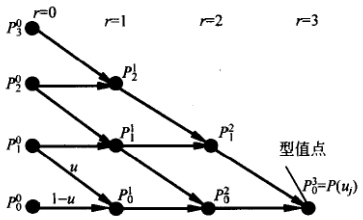


图 2: 三次Bézier 曲线de Casteljau离散生成过程

图3为三次Bézier 曲线在某个u 值(u 的变化生成多个离散型值点)下的几何意义。因此，可以通过控制误差用制控制多边形来近似逼近Bézier 曲线。

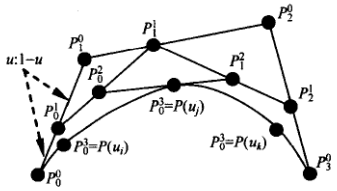


图 3: 三次Bézier 曲线作图过程

3. Bézier 曲线生成算法的伪代码描述

1）输入一系列控制点的坐标(x_i, y_i)存储在数组p_list中，用n表示控制点个数

2）设置精确度$precision$（本次作业设置为100，精确度越高曲线绘制越精确，同时计算耗时越高）

3）分别计算值点P_i^r的横坐标和纵坐标并存储在数组px和py中，用数组$points$存储最终需要连线的点：

先将px和py分别初始化为p_list中一系列控制点的横坐标和纵坐标

接着执行以下计算:

```
    for u 0 to precision do:
        u=u/precision
        for i 1 to n do:
            for j 0 to n-i do:
                px[j]=(1-u)*px[j]+u*px[j+1]
                py[j]=(1-u)*py[j]+u*py[j+1]
        points.append(int(px[0]),int(py[0]))
```

4）将points中存储的点进行连线

2.3.2 B-spline算法

1. B 样条曲线的定义

已知n+1个控制顶点$P_i(i=0,1,2,...n)$及参数节点向量：

$$U_{n,k}=\{u_i\}(i=0,1,2...,n+k;u_i\leq u_{i+1})$$

称如下形式的参数曲线$P(u)$为k阶(k-1次)B样条曲线:

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

其中，P_i为控制顶点，$\{B_{i,k}(u)\}$为下列deBoox-Cox递推关系所确定的为$U_{n,k}$上的k阶(k-1次)B样条基函数：$\{B_{i,k}(u)\}$双下标中下标 k 表示次数、下标 i 表示序号。

deBoox-Cox 递推公式为：

$$B_{i,k}(u)=[\frac{u-u_i}{u_{i+k-1}-u_i}]B_{i,k-1}(u)+[\frac{u_{i+k}-u}{u_{i+k}-u_{i+1}}]B_{i+1,k-1}(u)(i=0,1,2...n)$$

$$\begin{cases}B_{i,1}(u)=1(u\in[u_i,u_{i+1}])\\B_{i,1}(u)=0(u\notin[u_i,u_{i+1}])\end{cases}$$

在上面的递推式中，若遇到0/0 则取值为0。常称u_i为节点，$U_{n,k}$为节点向量。若$u_{j-1}<u_j=u_{j+1}=...=u_{j+r-1}<u_{j+r}$，则称从$u_j$到$u_{j+r-1}$的每一个节点为r重节点。

从$\{B_{i,k}(u)\}$的递推公式说明：欲确定第i个k次B样条$\{B_{i,k}(u)\}$，需要用到$\{u_i,u_{i+1},...u_{i+k+1}\}$共 k+2 个参数节点。区间$[u_i,u_{i+k+1}]$称为$\{B_{i,k}(u)\}$的支承区间：$\{B_{i,k}(u)\}$的第一个下标等于其支承区间左端节点的下标，表示该B样条在参数轴上的位置。

曲线方程中相应 n+1 个控制顶点$\{P_i\}(i=0,1,2..n)$要用到 n+1 个 k 次 B 样条基函数$\{B_{i,k}(u)\}$，它们每个都是 k 次B样条。它们的支承区间所含节点的并集就是定义这一组B样条基的节点矢量$U_{n,k}=\{u_i\},(i=0,1,2,...,n+k;u_i\leq u_{i+1})$。

第 i 个 k 次B样条基函数$\{B_{i,k}(u)\}$具有支承区间$[u_i,u_{i+k+1}]$,其左端节点u_i下标与该B 样条的次数 k 无关，右端节点u_{i+1} 的下标与次数 k 有关。k 次B样条的支承区间包含k+1个节点区间：$[u_i,u_{i+1}],[u_{i+1},u_{i+2}],...,[u_{i+k},u_{i+k+1}]$，因此支承区间包含的节点区间数与次数k有关，所以在参数轴上任一点$u\in[u_i,u_{i+1}]$处，最多只有 k+1 个非零的k次B样条基函数，其它的 k 次B样条基函数在该处为零。

给定 n+1 个控制顶点$\{P_i\}$，定义一条 k 次B样条曲线要用到 n+1 个 k 次B样条基函数$\{B_{i,k}(u)\}$，这 n+1 个 k 次B样条基函数由节点矢量$U_{n,k}=\{u_i\},(i=0,1,2,...,n+k)$决定。一条 k 次B样条曲线段由 k+1 个控制顶点定义，在不含重节点情况下每增加一个顶点，曲线段数就加1，故 n+1 个控制顶点定义的 k 次B样条曲线共有 n-k+1 段，曲线定义所对应的节点区间由 n-k+2 个节点组成，共 n-k+1 个节点区间。

通常，可首先确定这 n-k+2 个节点，其首尾节点所限定的参数区间就是B样条曲线的定义域；然后，从首尾节点各向外延伸 k 个节点，共得到 n+k+2 个节点，构成节点矢量，以定义 n+1 个B 样条基函数。也可由该区间$[u_i,u_{i+1}]$的左右节点各向外扩展 k 个节点得到所要求的节点系列。

2. 三次（四阶）均匀B样条曲线生成算法代码

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

        def B_spline(p_list):
            k = 4
            n=len(p_list)
            if n<4:
                return p_list
            du=1/100
            u =k-1
            while u<=n:
                x,y = 0,0
                for i in range(n):
                    x0,y0 = p_list[i]
                    temp=B(i, k, u)
                    x +=x0*temp
                    y +=y0*temp
                result.append([int(x), int(y)])
                u+=du
```

2.3.3 Bezier算法和B样条曲线比较

Bézier 曲线和B样条曲线的主要区别表现在以下四个方面：

- 基函数的次数：对于Bézier 曲线，基函数的次数等于控制顶点数减1；对于B 样条曲线，基函数的次数与控制顶点数无关。
- 基函数性质：Bézier 曲线的基函数即Bernstein 基函数是多项式函数；B 样条曲线的基函数即B 样条基函数是多项式样条。
- 曲线性质：Bézier 曲线是一种特殊表示形式的参数多项式曲线；B 样条曲线则是一种特殊表示形式的参数样条曲线。
- 局部控制能力：Bézier 曲线缺乏局部性质；B 样条曲线具有局部性质。

2.4 图形变换

2.4.1 平移

原始位置(x,y)按平移距离dx和dy到新位置(x_1,y_1)的移动计算公式为：

$$\begin{cases} x_1 = x + dx \\ y_1 = y + dy \end{cases}$$

对于不产生变形的刚体变换，物体上的每个点移动相同的坐标，具体为：

- 直线：平移线的每个端点
- 多边形：平移每个顶点坐标
- 圆或椭圆：平移其中心坐标并在新的中心位置重画图形
- 曲线：平移定义曲线的控制点位置，用新的控制点坐标重构曲线

2.4.2 旋转

假设旋转中心为(x_r,y_r)，旋转角度为θ，原坐标为(x,y)，则旋转变换后的坐标(x_1,y_1)计算公式为：

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

对于不产生变形的刚体变换，物体上的每个点旋转相同的角度，具体为：

- 直线：旋转每个线端点
- 多边形：每个顶点旋转指定旋转角
- 曲线：旋转控制点

2.4.3 缩放

缩放变换改变物体的尺寸——多边形。

假设缩放中心为(x_f,y_f)，缩放系数为s，原坐标为(x,y),则旋转变换后的坐标(x_1,y_1)计算公式为：

$$\begin{cases} x_1 = x * s + x_f(1 - s) \\ y_1 = y * s + y_f(1 - s) \end{cases}$$

2.5 二维线段裁剪算法

线段裁剪算法的目标就是减少线段求交、判断等计算量，并有效识别给定线段与裁剪窗口相对位置关系。对那些不是完全可见或是完全不可见的线段的裁剪，需计算线段与裁剪窗口边界的交点，然后通过对线段的端点进行“内—外检测”来处理线段。线段裁剪处理过程主要包括两个方面：

(1) 预处理：位置关系判断

预处理主要是为了减少不必要的求交计算，即求交前先进行给定线段与裁剪窗口的相互位置关系的测试，分为三种情况：

- 线段完全在裁剪窗口之内；
- 线段完全在裁剪窗口之外；
- 其它。

(2) 交点计算：给定线段与一个或多个裁剪边界的交点计算

为便于求交计算，一般线段采用参数形式表示。设线段端点为(x_1, y_1)和(x_2, y_2)，其参数表达式为：

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

如果该线段其中一个或两个端点都在裁剪矩形外，该线段与裁剪边界求交时，可得出线段（或延长线）与裁剪窗口边界的交点参数u 值。由于在计算两条线段之间的交点时，在计算时实际上求出的交点为两条直线之间的交点，因此需判断得到交点的有效性，即如果与矩形边界交点的u 值不在0 和1 之间，则该交点无效，表示求得的交点为两线段延长线的交点；如果u 值在0 和1 之间，该交点有效，然后根据有效的交点可以确定线段的哪一段是可见的，哪一段是不可见的。此方法可用于各个裁剪边界，以便确定该线段的显示部分。平行于窗口边界的线段可作为特殊情况处理。

2.5.1 Cohen-Sutherland算法

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。

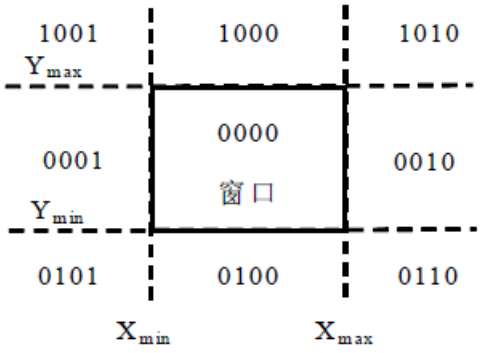


图 4: 区域编码

编码算法以图4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。四位区域码中各位从左到右依次表示上、下、右、左。区域码的任何位赋值

为1代表端点落在相应的区域中，否则该位为0。若端点在裁剪窗口内，区域码为0000；如果端点在裁剪窗口的右上角，则区域码为1010。设最左边的位为第1 位，则区域码的生成可采用比较法或差值法：

1．比较法

根据区域编码规则，如图4所示，在确定区域码每位的值时，区域码各位的值可通过将端点坐标值(x,y)与裁剪边界比较来确定：

(1)如果$x < x_{min}$，表示该点在裁剪窗口左边界的左边，则第1位置1，否则置0；

(2)如果$x > x_{max}$，表示该点在裁剪窗口右边界的右边，则第2位置1，否则置0；

(3)如果$y < y_{min}$，表示该点在裁剪窗口下边界的下边，则第3位置1，否则置0；

(4)如果$x > y_{max}$，表示该点在裁剪窗口上边界的上边，则第4位置1，否则置0。

2．差值法

对可进行位操作的语言，区域码各位的值可按下列两步确定：

⑴计算端点坐标和裁剪边界之间的差值；

⑵用各差值符号来设置区域码各位的值：第1位为$x - x_{min}$ 的符号位；第2位为$x_{max} - x$的符号位；第3位为$y - y_{min}$的符号位；第4位为$y_{max} - y$的符号位。

判断线段是否完全在裁剪窗口内或外可采用对两个端点的区域码进行逻辑与操作的方法，根据线段与裁剪窗口的关系可分三种情况处理：

1．线段完全在裁剪窗口之内

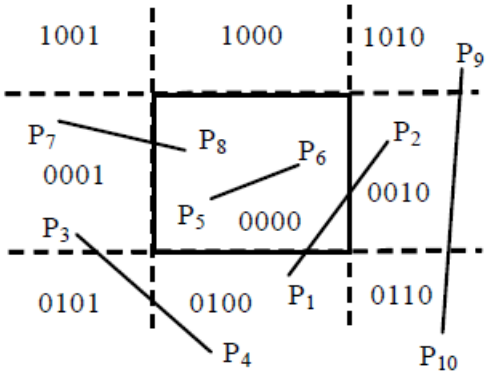


图 5: 线段裁剪

两个端点的区域码均为0000，则该线段完全在裁剪窗口之内。如图5所示，线段P_5P_6完全在裁剪窗口内。

2．线段完全在裁剪窗口之外

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。如图5所示，线段P_9P_{10}完全在裁剪窗口外，P_9端点的区域码为1010，P_{10}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。

3．其它

图5中P_1P_2，P_3P_4，P_7P_8都属于此类情况，需进行求交计算。虽然P_3P_4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。如图5中P_7P_8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。P_7为外端点，即落在裁剪窗口左边界的左边，此点肯定不可见。P_8为内端点，即落在裁剪窗口左边界的右边，为可能的可见点。P_7P_8线段与窗口左边界求交后，可以确定应裁剪掉的线段为P_7到P_7P_8线段与裁剪窗口左边界的交点，而交点到P_8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。

Cohen-Sutherland算法的程序流程:

```
def Cohen_Sutherland(p_list):
    x1,y1=p_list[0]
    x2,y2=p_list[1]
    result=[]
    flag=1
    i=0
    while flag:
        i+=1
        if i>4:
            break
        c1=encode(x1,y1,x_min,y_min,x_max,y_max)
        c2=encode(x2,y2,x_min,y_min,x_max,y_max)
        if (c1 & c2)==0:#probably part of line in window
            if (c1 or c2)!=0:#at least one point out of window
                if c1==0:# p1 in window,exchange p1 and p2 to ensure p1 
                    temp_x=x1
                    temp_y=y1
                    x1=x2
                    y1=y2
                    x2=temp_x
                    y2=temp_y
                if c1&1:#left
                    x1,y1=inter_point(x_min,x1,y1,x2,y2)
                elif c1&2:#right
                    x1,y1=inter_point(x_max,x1,y1,x2,y2)
                elif c1&4:#down
                    y1,x1=inter_point(y_min,y1,x1,y2,x2)
                elif c1&8:#up
                    y1,x1=inter_point(y_max,y1,x1,y2,x2)
            else:#p1 and p2 both in window
```

```
                    flag=0
            else:# line out of window
                flag=0
        result=[[int(x1),int(y1)],[int(x2),int(y2)]]
        return result
```

2.5.2 Liang-Barsky线段裁剪算法

1. Liang-Barsky线段裁剪原理

给定需要裁减的线段P_1P_2,设P_1P_2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Q_1Q_2，称Q_1Q_2为诱导窗口，它是一维的。这样，P_1P_2关于矩形窗口的裁剪结果与P_1P_2关于诱导窗口Q_1Q_2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。

在一维数轴上，数轴坐标的参数表达式为：$P = P_1 + u(P_2 - P_1)$。假设P_1为数轴原点，则$u_{P_1} = 0, u_{P_2} = 1$；Q_1Q_2的参数分别为u_1和u_2，且$u_1 < u_2$；而P_1P_2和Q_1Q_2间有四种位置关系:

1) $Q_1P_1P_2Q_2(u_1 < 0, u_2 > 1)$：完全在窗口内

2) $P_1Q_1Q_2P_2(0 \le u_1 \le 1, 0 \le u_2 \le 1)$

3) $P_1P_2Q_1Q_2$或$Q_1Q_2P_1P_2(u_1 < 0, u_2 < 0$或$u_1 > 1, u_2 > 1)$：完全在窗口外

4) $P_1Q_1P_2Q_2$或$Q_1P_1Q_2P_2(u_1 < 0, 0 \le u_2 \le 1$或$0 \le u_1 \le 1, u_2 > 1)$

根据上述关系，可以得出：P_1P_2至少部分可见的充要条件为：

$$max(u_{P_1}, u_1) \le min(u_{P_2}, u_2)$$

且其可见部分的区间为：$[max(u_{P_1}, u_1), min(u_{P_2}, u_2)]$。

2. 诱导窗口的生成计算

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6中，P_1P_2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；Q_1Q_2为诱导窗口；记窗口左右边界所在直线夹成的带形区域为A_1；窗口上下边界所在直线夹成的带形区域为A_2；窗口区域为Window。那么，诱导窗口Q_1Q_2计算可如下：

$$Q_1Q_2 = Line \cap Window = Line \cap (A_1 \cap A_2) = (Line \cap A_1) \cap (Line \cap A_2) = LR \cap TB$$

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Q_1Q_2对应的参数区间。

这样，P_1P_2的可见部分VW可计算为：

$$VW = P_1P_2 \cap Q_1Q_2 = P_1P_2 \cap LR \cap TB$$

3. Liang-Barsky线段裁剪算法

参数化形式的裁剪条件：

$$\begin{cases} xw_{min} \le x_1 + u\Delta x \le xw_{max} \\ yw_{min} \le y_1 + u\Delta y \le yw_{max} \end{cases}$$

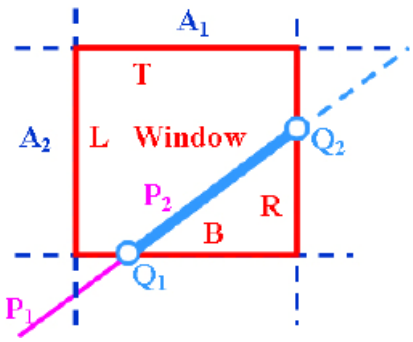


图 6: 一维诱导窗口生成示意

可统一表示为：$u * p_k \leq q_k$。其中，k=1,2,3,4，对应裁剪窗口的左、右、上、下边界；参数p、q 定义为：

$$\begin{cases} p_1 = \Delta x, q_1 = x_1 - xw_{min} \\ p_2 = \Delta x, q_2 = xw_{max} - x_1 \\ p_3 = \Delta y, q_3 = y_1 - yw_{min} \\ p_4 = \Delta y, q_4 = yw_{max} - y_1 \end{cases}$$

根据p_k可以确定线段与裁剪窗口的相互位置关系：

①若$p_k = 0$，直线平行于裁剪边界之一。此时，如果同时满足$q_k < 0$，则线段完全在边界外；如果同时满足$q_k \geq 0$，线段平行于裁剪边界，并且在窗口内；

②当$p_k < 0$, 线段从裁剪边界延长线的外部延伸到内部；

③当$p_k > 0$，线段从裁剪边界延长线的内部延伸到外部。

因而，当p_k非零时，可计算出线段与边界k或延长线交点的u值：$u = q_k/p_k$；对每条线段，可计算出参数u_1和u_2，它们定义了在裁剪矩形内的线段部分：如果$u_1 > u_2$，则线段完全落在裁剪窗口外，被舍弃；否则，被裁剪线段的端点由参数u的两个值计算出来：u_1的值由线段从外到内遇到的矩形边界所决定(p¡0)，对这些边界计算参数： $r_k = q_k/p_k$，u_1取0和各个r_k值之中的最大值；u_2的值由线段从内到外遇到的矩形边界所决定(p¿0)，对这些边界计算参数：$r_k = q_k/p_k$，u_2取1和各个r_k值之中的最小值。

在具体实现时，算法的实现过程如下：

(1) 将线段交点的参数初始化为 $u_1 = 0, u_2 = 1$；

(2) 定义一个函数，用 p、q 来判断是舍弃线段还是改变交点的参数r：当$p < 0$时，参数r 用于更新u_1；当$p > 0$时，参数r 用于更新u_2：

如果更新u_1或u_2后使$u_1 > u_2$，则舍弃该线段；否则，更新适当的u 值仅仅求出了交点，缩短线段。

(3) p、q 的四个值经过测试后，当p=0且$q < 0$时，说明该线段平行于边界且位于边界之外，舍弃该线段。假如该线段未被舍弃，则裁剪线段的端点由u_1、u_2值决定。

(4) 反复调用该函数，计算出各个裁剪边界的p、q 值，进行判断。

2.5.3 Liang-Barsky算法和Cohen-Sutherland算法的比较

通常，Liang-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u_1、u_2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u_1、u_2的最后值；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。Liang-Barsky 和Cohen-Sutherland算法都可以扩展为三维裁剪算法。

3 系统介绍

3.1 系统框架

- 核心算法模块（各种图元的生成、编辑算法）：cg_algorithms.py
- 命令行界面（CLI）程序：cg_cli.py
- 用户交互界面（GUI）程序：cg_gui.py

3.2 交互逻辑

3.2.1 命令行交互

交互方式：读取包含图元绘制指令序列的文本文件，依据指令调用核心算法模块中的算法绘制图形以及保存图像。

程序接受两个外部参数：指令文件的路径（input_path)和图像保存目录(output_dir)

测试程序时的指令格式：python cg_cli.py input_path output_dir

文本文件中的指令格式：

- 重置画布(清空当前画布，并重新设置宽高): resetCanvas width height （width, height: int； $100 \leq width, height \leq 1000$）
- 保存画布(将当前画布保存为位图name.bmp): saveCanvas name （name: string）
- 设置画笔颜色: setColor R G B
- 绘制线段: drawLine id x0 y0 x1 y1 algorithm (id: string, 图元编号，每个图元的编号是唯一的; x0, y0, x1, y1: int, 起点、终点坐标; algorithm: string, 绘制使用的算法，包括"DDA"和"Bresenham")
- 绘制多边形: drawPolygon id x0 y0 x1 y1 x2 y2 ... algorithm (id: string, 图元编号，每个图元的编号是唯一的; x0, y0, x1, y1, x2, y2 ... : int, 顶点坐标; algorithm: string, 绘制使用的算法，包括"DDA"和"Bresenham")
- 绘制椭圆（中点圆生成算法）: drawEllipse id x0 y0 x1 x1 (id: string, 图元编号，每个图元的编号是唯一的; x0, y0, x1, y1: int, 椭圆矩形包围框的左上角和右下角对角顶点坐标)
- 绘制曲线: drawCurve id x0 y0 x1 y1 x2 y2 ... algorithm (id: string, 图元编号，每个图元的编号是唯一的; x0, y0, x1, y1, x2, y2 ... : int, 控制点坐标; algorithm: string, 绘制使用的算法，包括"Bezier"和"B-spline", 其中"B-spline"要求为三次（四阶）均匀B样条曲线，曲线不必经过首末控制点)

- 图元平移: translate id dx dy (id: string, 要平移的图元编号; dx, dy: int, 平移向量)
- 图元旋转: rotate id x y r (id: string, 要旋转的图元编号; x, y: int, 旋转中心; r: int, 顺时针旋转角度（°）)
- 图元缩放: scale id x y s (id: string, 要缩放的图元编号 x, y: int, 缩放中心 s: float, 缩放倍数)
- 对线段裁剪： clip id x0 y0 x1 y1 algorithm （id: string, 要裁剪的线段编号； x0, y0, x1, y1: int, 裁剪窗口的左上角和右下角对角顶点坐标； algorithm: string, 裁剪使用的算法，包括"Cohen-Sutherland"和"Liang-Barsky"）

3.2.2 用户交互

交互方式：以鼠标、键盘交互，通过鼠标、键盘事件获取所需参数和指令，并调用核心算法模块中的算法将图元绘制到屏幕上。

测试程序时的指令格式如下：python cg_gui.py

3.3 设计思路

3.3.1 命令行交互模式

通过解析输入文件中的一行行命令获得指令及相应参数，调用cg_algorithm.py中对应的绘制函数并传入参数。

3.3.2 用户交互模式

使用GUI库PyQt5，主要设计了三个类：

- 自定义图元类MyItem，继承自QGraphicsItem。主要属性有图元ID，图元类型，图元参数，绘制算法等。定义paint函数，用于根据图元的属性调用cg_algorithm.py中对应的绘制函数。定义boundingRect函数，用于计算包围图元的正矩形框（QRectF类型）
- 画布窗体类MyCanvas：继承自QGraphicsView，采用QGraphicsView、QGraphicsScene、QGraphicsItem的绘图框架。负责响应鼠标和键盘事件，在画布场景中添加图元，并在item_dict中记录所有的图元。
- 主窗口类MainWindow，继承自QMainWindow。设置菜单栏实现用户对系统功能的选择，使用QGraphicsView作为画布，连接菜单栏点击信号和槽函数来调用画布类的对应函数实现相应操作。

类与类之间的关系：

主窗口类主要用于用户对系统功能的选择；其包含的画布窗体类主要用于响应鼠标键盘事件来获取图元参数；画布窗体类中记录的图元类以画布窗体类获取的参数作为自身属性，根据这些属性调用相应的绘制算法来计算需要绘制的点集，最后用QPainter完成最终的绘制。

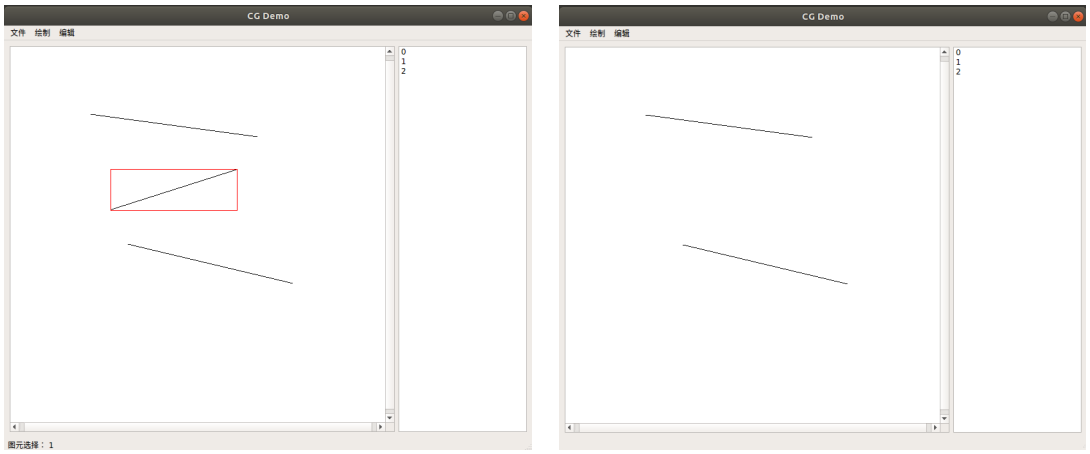
4 其他

4.1 额外的功能

在“编辑”菜单栏下的“图元选择”、“平移”、“旋转”、“缩放”状态下可以进行图元选择并实行如下操作：

1）图元的删除

选中图元后按“Backspace”键可以将选中的图元删除：

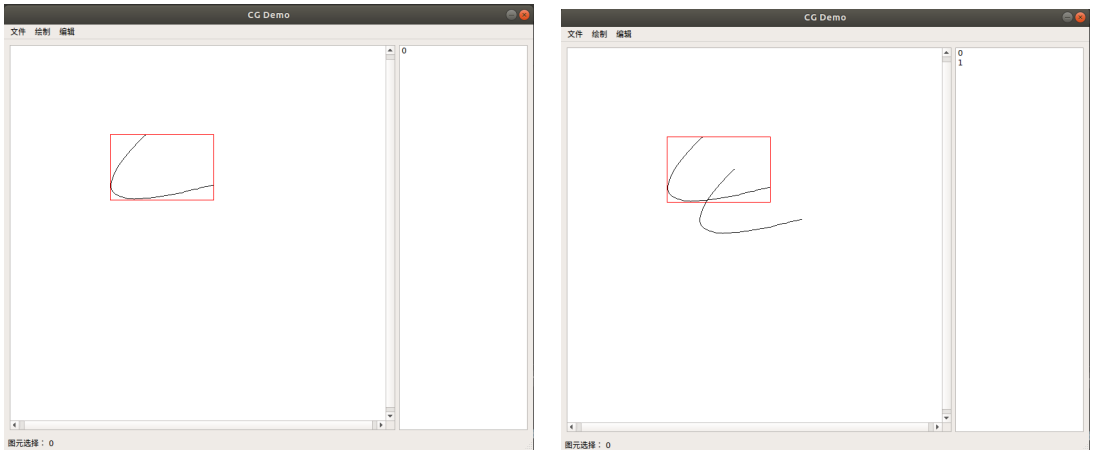


(a) 选择图元　　(b) 删除图元

图 7: 图元的删除

2）图元的复制粘贴

选中图元后，按“Ctrl+C”可以复制选中图元，按“Ctrl+V”粘贴：



(a) 选择图元　　(b) 复制图元

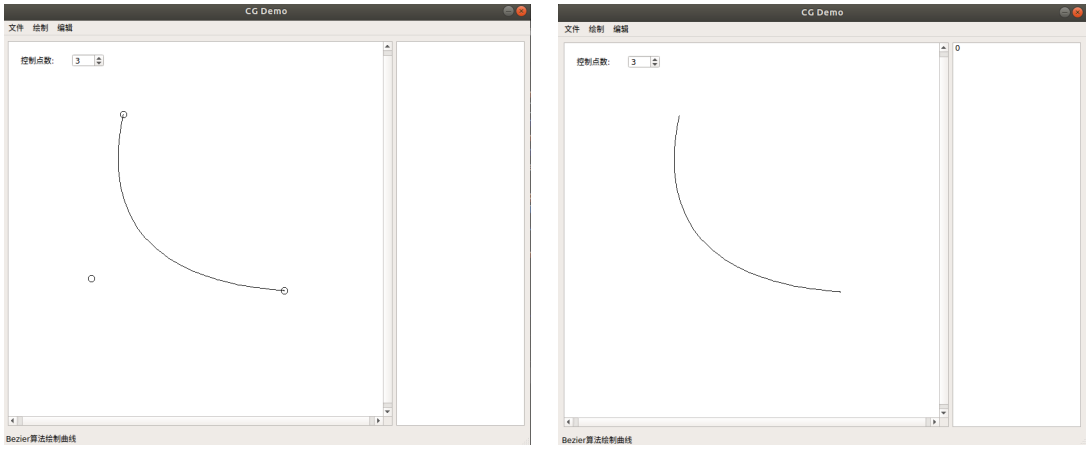
图 8: 图元的复制粘贴

4.2 易用的交互

1）曲线的绘制

曲线的控制点数由用户输入，同时为方便用户自由调节曲线的形状，将曲线控制点用小圆圈标出。可通过拖动小圆圈调节曲线控制点的位置，进而完成对曲线形状及位置的调整。

单击鼠标右键结束调整过程，标志控制点的小圆圈消失，完成该曲线的绘制。



(a) 调节曲线形状　　(b) 完成曲线绘制

图 9: 曲线的绘制

2）鼠标、键盘的利用

旨在减少用户切换菜单栏的频率，使用户操作更方便快捷。

- 在“编辑”菜单栏下的图元选择、平移、旋转、缩放操作中增加图元选择功能，方便用户在其中某一状态下先后对多个图元进行操作。在上述可选择图元的功能下，鼠标左键点击图元所在矩形框范围内的某一位置即可选中该图元。
- 除鼠标左键外，利用鼠标滚轮进行缩放操作，鼠标右键结束曲线和多边形的绘制、取消图元的选中。
- 引入键盘操作进行图元的快速删除和复制粘贴。

4.3 巧妙的设计

1）绘制模式的设置

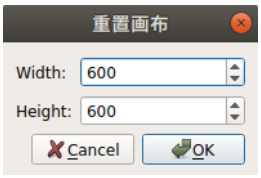
以曲线绘制为例：在调节曲线形状时，除曲线本身外，需要同时绘制标记控制点的小圆圈；在结束曲线调节后，不用再绘制标记控制点的小圆圈。针对两种不同的绘制需求，可以通过在cg_algorithm.py中的draw_curve函数中增设flag参数（默认为0），需要绘制标记控制点的小圆圈时在调用draw_curve函数时传入flag=1，不需时可以在调用时设flag=0或者不传入flag参数。

2）绘制状态的设置

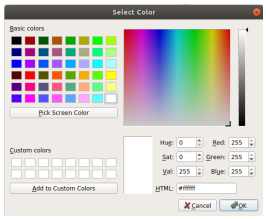
以曲线绘制为例：在拖拽鼠标左键确定曲线大致位置和拖拽控制点微调曲线形状时，系统对鼠标事件的响应措施是不同的。为正确响应鼠标事件，利用有限状态机的思想，即设置stage参数，将“确定大致位置”，“调节控制点”，“结束调节”三个阶段的stage值分别设为0，1，2。这样响应鼠标事件时可以先判断曲线绘制目前处于哪一阶段，再实行相应操作。

4.4 好看的界面

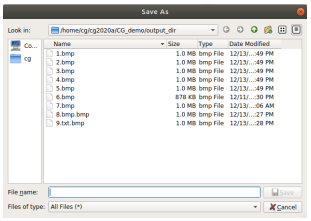
利用PyQt下的对话框类QDialog及其派生类QColorDialog和QFileDialog和布局类QFormLayout，实现较好的重置画布，颜色选择和画布保存界面。



(a) 重置画布



(b) 设置画笔颜色



(c) 保存画布

5 总结

本课程作业使用Python3语言编程，实现了几种常用的图形绘制算法，提供命令行和图形界面两种交互方式，满足一个简洁绘图系统的基本需求。

参考文献

[1] 孙正兴. 计算机图形学教程. 机械工业出版社, 南京, 2006.

[2] https://blog.csdn.net/mmogega/article/details/53055596

[3] https://blog.csdn.net/orbit/article/details/7496008

[4] https://blog.csdn.net/qq_32583189/article/details/53018981

[5] https://blog.csdn.net/a3631568/article/details/53637473

[6] https://blog.csdn.net/keneyr/article/details/83871170

[7] https://blog.csdn.net/fgh1991/article/details/89851327

[8] http://zetcode.com/gui/pyqt5

[9] https://doc.qt.io/archives/qt-4.8/qrectf.html

[10] https://blog.csdn.net/fgh1991/article/details/89851327