《计算机图形学》十二月报告

作者姓名：谢靓静　学号：181860112　联系方式(qq): 641441363

(南京大学 计算机科学与技术系，南京　210093)

摘　要:　本学期课程实验的目标是跟随课程进度在项目中实现各种图形学算法，最终完成一个支持命令行和GUI的完整图形学系统。图形学系统主要由三大模块组成：核心算法模块(cg_algorithms.py)、命令行界面(CLI)程序(cg_cli.py)和用户交互界面(GUI)程序(cg_gui.py)。在本阶段，图形绘制系统已经全部完成。完成了算法模块的全部要求算法，在命令行下成功完成了测试，得到了有效且正确的输出结果。对图形交互界面进行了外观设计，设置了菜单栏、带图标的工具栏，提供了鼠标交互的操作，同时考虑了对非法操作的处理，代码具有良好的鲁棒性。

关键词:　计算机图形学;面向对象;GUI; 图元生成算法

1　引言

图形生成的基本构造单元称为输出图元，生成输出图元的子程序为构造图形提供了基本工具，而最基本的子程序是画点和画线。人类习惯以矢量表示和输入图形对象，而光栅扫描图形输出设备采用点阵表示方式输出图形，所以在交互式计算机图形系统中，需要进行线画图形生成的转换。[1]

2　开发和测试环境说明

开发使用 PyCharm 2019.1.2 x64：

- Windows10
- Python 3.8.3
  - numpy 1.18.5
  - pillow 7.2.0
  - pyqt 5.9.2

测试环境为 VirtualBox 虚拟机：

- Ubuntu 18.04 x86_64
- Python 3.7.4
  - numpy 1.18.1
  - pillow 7.0.0
  - pyqt 5.9.2

3　算法介绍

3.1 绘制线段

3.1.1 DDA 算法

数字差分分析(Digital Differential Analyzer, DDA)方法[1] 。是利用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。在一个坐标轴上取单位间隔($\Delta x = 1$或$\Delta y = 1$)，依据计算相应的Δy或Δx

决定另一个坐标轴上最靠近线段路径的对应数值。

在具体实现中，计算出横纵坐标的差值，分别记作 dx, dy；计算出直线斜率的值，记作 k:

$$dx = x_1 - x_0$$
$$dy = y_1 - y_0$$
$$k = dy / dx$$

对于以下三种情况，应先行讨论。此时不需要计算，可直接绘制：

- 斜率不存在 (x0 == x1)
- 斜率值为 0 (y0 == y1)
- 斜率绝对值为 1 (dx == ±dy)

否则按 DDA 算法执行运算，按斜率绝对值小于 1 和大于 1 两种情况讨论。

若斜率绝对值小于 1，对 x 坐标以 1 为步长递增计算，执行代码：

```
for i in range(abs(dx) + 1):
    x = x + 1
    y = y + k
    result.append((x, round(y)))
```

若斜率绝对值大于 1，执行对偶操作：

```
m = 1 / k
for i in range(abs(dy) + 1):
    x = x + m
    y = y + 1
    result.append((int(x), int(y)))
```

3.1.2 **Bresenham 算法**

Bresenham 画线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。为简化像素的选择，Bresenham 算法通过引入整形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，并利用对整形参量符号的检测来确定最接近实际线路径的像素[1] 。其特点是仅使用整数加法，减法和位移这类标准计算机体系结构中非常便宜的操作，是一种增量错误算法。

在具体实现中，与 DDA 算法执行具体算法前的讨论类似，先对三种可以直接绘图的情况进行讨论，之后分别计算直线斜率 k，横、纵坐标增量 dx、dy，其他常用数值，和决策参数 d 的初值：

$$k = dy / dx$$
$$dx = |x_1 - x_0|$$
$$dy = |y_1 - y_0|$$
$$d = dx - dy2$$

排除上述特殊情况之后，按 Bresenham 算法执行运算，按斜率绝对值与 1 的大小关系分 4 种情况讨论。

在直线斜率绝对值小于 1 的情况下，对 x 坐标以 1 为步长递增计算，决策参数更新方案为：

$$\begin{cases} d = d + dx^2 - dy^2 & ,d < 0 \\ d = d - dy^2 & ,d \geq 0 \end{cases}$$

在 d < 0 时，若 0 < k < 1，将 y 的值更新为 y + 1；若-1 < k < 0，将 y 的值更新为 y - 1。

在直线斜率绝对值大于 1 情况下，讨论过程及执行的操作为上文的对偶，对应的决策参数更新方案为：

$$\begin{cases} d = d + dy^2 - dx^2 & ,d < 0 \\ d = d - dx^2 & ,d \geq 0 \end{cases}$$

3.1.3 对比分析

DDA 算法利用光栅特性消除了直线方程中的乘法，计算像素位置速度比直接使用直线方程要快。但浮点增量的连续迭加中取整误差的积累会使长线段的像素位置偏离实际线段，而且程序中的取整操作和浮点运算仍然十分耗时。而 Bresenham 算法引入了决策参数，仅使用整数加法，减法和位移这类标准计算机体系结构中非常便宜的操作。其本质同 DDA 算法一样基于步进的思想，但避免了浮点运算，是 DDA 算法的改进。

3.2 绘制椭圆

3.2.1 中点椭圆生成算法

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，椭圆的曲线的生成可通过考虑椭圆长轴和短轴尺寸不同而修改画圆程序来实现。在任意方向利用两个焦点和一个椭圆边界上的点这三个坐标位置，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。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$\mathrm{r_x}$标识长半轴，参数$\mathrm{r_y}$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[1] 。因此讨论算法时，我们只考虑椭圆在第一象限的部分，其他象限的部分可作相应的对称变换而得。中心不在原点的椭圆可通过平移得到。

和 Bresenham 圆光栅化算法一样，中点椭圆算法利用了第一象限的椭圆是随 x 和 y 的单调递增或递减函数。在具体实现中，先分别计算长轴短轴长度 a、b，偏移量$(\mathrm{x_c}, y_c)$，其他常用数值和决策参数 d 的初值，并将 x 的初始值设置为 0， y 的初始值设置为 b：

$$\mathrm{a} = \frac{\mathrm{x_1} - \mathrm{x_0}}{2},\ \mathrm{b} = \frac{\mathrm{y_0} - \mathrm{y_1}}{2}$$

$$\mathrm{x_c} = \frac{\mathrm{x_0} + \mathrm{x_1}}{2},\ \mathrm{y_c} = \frac{\mathrm{y_0} + \mathrm{y_1}}{2}$$

$$\mathrm{d} = \mathrm{b}^2 - \mathrm{a}^2\mathrm{b} + \frac{\mathrm{a}^2}{4}$$

$$\mathrm{x} = 0,\ \mathrm{y} = \mathrm{b}$$

以椭圆上斜率为-1 的点为界将椭圆分成两个部分，如图 1 所示。通过在斜率绝对值小于 1 的区域的 x 方向取

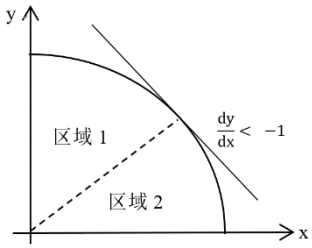


图 1 将椭圆分成两个部分

单位步长，在斜率绝对值大于 1 的区域的 y 方向取单位步长来处理这个象限。在区域 1 和区域 2 的交界区有 $\frac{\mathrm{dy}}{\mathrm{dx}} = -1$，可得区域 1 和区域 2 的分割条件为：

$$2\mathrm{r}_y^2 x = 2r_x^2 y$$

则移出区域 1 进入区域 2 的条件是：

$$2\mathrm{r}_y^2 x \geq 2r_x^2 y \qquad \text{(a)}$$

对第一象限内取样位置x_{k+1}处两个候选像素间中点位置对决策参数(即椭圆函数)进行计算分析。在区域 1 内部，假如 d < 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，决策参数的更新方案为：

$$\begin{cases} d = d + b^2(2x+3) & d < 0 \\ d = d + b^2(2x+3) + a^2(-2y+2) & d \geq 0 \end{cases}$$

当 x，y 的值不满足式(a)，说明进入区域 2，将决策参数的值更新为：

$$d = b^2\left(x + \frac{1}{2}\right)^2 + a^2(y-1)^2 - a^2b^2$$

假如 d > 0，中点位于椭圆边界之外，选择像素x_k；假如$d \leq 0$，中点位于椭圆边界之内或之上，选择像素x_{k+1}。决策参数的更新方案[3] 为：

$$\begin{cases} d = d + 2b^2x + 2b^2 - 2a^2y + 3a^2 & d < 0 \\ d = d - 2a^2y + 3a^2 & d \geq 0 \end{cases}$$

基于椭圆的对称性，只需要计算 1/4 椭圆，每当一个位置计算完成之后，向结果中添加对称的四个点的坐标：

```
result.append([x, y])
result.append([x, -y])
result.append([-x, y])
result.append([-x, -y])
```

全部点计算完成后，对结果中的点坐标执行平移和取整操作：

```
for p in result:
    p[0] += xc
    p[1] += yc
    p[0] = int(p[0])
    p[1] = int(p[1])
```

3.3 绘制曲线

3.3.1 Bezier 算法

Bezier 曲线是由法国工程师皮埃尔·贝济埃于 1962 年提出，用于进行汽车的主体设计。该方法能让设计者比较直观地意识到所给设计条件与最终生成曲线之间的关系，从而方便地通过修改输入参数来改变曲线的形状和阶次。曲线的形状趋向于的多边折线称为特征多边形，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。贝塞尔曲线完全由其控制点决定其形状， n 个控制点对应着 n-1 阶的贝塞尔曲线，并且可以通过递归的方式来绘制。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。[1]

一阶贝塞尔曲线是一条直线，有两个控制点。根据给定的介于[0, 1]之间的 t 值进行计算并线性插值得到。则有：[4]

$$B_1(t) = (1-t)P_0 + tP_1P_2 \quad , \quad t \in [0,1]$$

二阶贝塞尔曲线有三个控制顶点，其生成过程可以描述为：在平面内任选 3 个不共线的点，依次用线段连接。在第一条线段上任选一个点 D。计算该点到线段起点的距离 AD，与该线段总长 AB 的比例。之后根据上一步得到的比例，从第二条线段上找出对应的点 E，使得 AD:AB = BE:BC。此时 DE 再次成为一条直线，可以按照一阶的贝塞尔方程来进行线性插值了, t= AD:AE。得到二阶贝塞尔公式：

$$B_2(t) = (1-t)^2P_0 + 2t(1-t)P_1 + t^2P_2 \quad , \quad t \in [0,1]$$

二阶的贝塞尔通过在控制点之间再采点的方式实现降阶，每一次选点都是一次的降阶。同理，四个点对应是三次的贝塞尔曲线，高阶的贝塞尔曲线可以通过不停的递归直到一阶。

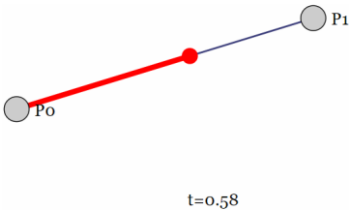


图 2 一阶贝塞尔曲线

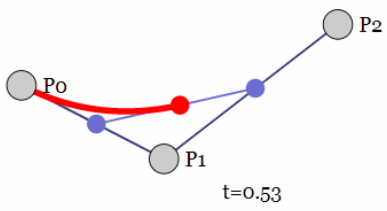


图 3 二阶贝塞尔曲线

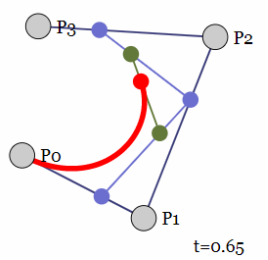


图 4 三阶贝塞尔曲线

贝塞尔曲线的公式表示如下：

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

$$\mathrm{B_{i,n}}(t) = C_n^i t^i (1-t)^{n-i} = \frac{n!}{i!\,(n-i)!} t^i (1-t)^{n-i} \quad , i = 0,1,\dots,n$$

基函数的递归性满足下式：

$$\mathrm{B_{i,n}}(t) = (1-t) B_{i,n-1}(t) + t B_{i-1,n-1}(t), i = 0,1,\dots,n$$

具体编程实现中，先设定步长 delta，按步长在[0,1]区间内直接套用上述递归性公式，为了提升效率，可以将递归实现更改为循环实现。

3.3.2 B-spline 算法

与 Bezier 曲线的定义类似，B 样条曲线方程表示为：

$$\mathrm{P(u)} = \sum_{i=0}^{n} d_i B_{i,k}(u)$$

其中，di(i=0,1...n)为控制顶点（坐标），Bi,k(i=0,1...n)为 k 次规范 B 样条基函数，最高次数是 k。基函数是由一个称为节点矢量的非递减参数 u 的序列 U：u0≤u1≤…≤un+k+1 所决定的 k 次分段多项式。

给定参数 u 轴上的节点分割：$U_{n,k} = \{u_i\}(i = 0,1,2\dots, n+k)$，称由下列 deBoox-Cox 递推关系所确定的 $B_{i,k}(u)$为$U_{n,k}$上的 k 阶(或 k-1 次)B 样条基函数。deBoor-Cox 算法描述了从给定参数区间[$\mathrm{u_j}$,u_{j+1}]计算 B 样条曲线型值点 P(u)的过程。利用计算出的型值点的折线来作为 B 样条的近似。

deBoor-Cox 递推公式：

$$B_{i,k}(u) = \left(\frac{u-u_i}{u_{i+k-1}-u_i}\right) B_{i,k-1}(u) + \left(\frac{u_{i+k}-u}{u_{i+k}-u_{i+1}}\right) B_{i+1,k-1}(u); (i = 0,1,2,\dots,n)$$

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

节点矢量中节点为沿参数轴均匀或等距分布称作均匀 B 样条曲线。

实际编程实现将绘制三次均匀 B 样条曲线。首先判断列表中点的个数，至少接受到四个点才开始调用算法进行运算。将节点矢量间距设置为 1，执行 deBoor-Cox 算法计算待绘制的点。

3.3.3 对比分析

Bezier 曲线是 B 样条曲线的特例，B 样条方法是 Bezier 方法的推广。Bezier 曲线特征多边形顶点的数量决定了 Bezier 曲线的阶次，即 n 个顶点的特征多边形必然产生 n-1 次的 Bezier 曲线，这是不够灵活的。同时，Bezier 曲线缺少局部性，修改某一控制顶点将影响整条曲线；控制多边形与曲线的逼近程度较差，次数越高，逼近程度越差；当表示复杂形状时，无论采用高次曲线还是多段拼起来的低次曲线，都相当复杂。以 B 样条基函数代替 Bernstein 基函数，可以改进 Bezier 特征多边形与 Bernstein 多项式次数有关，且整体逼近的缺点[1]。

B 样条曲线和 Bezier 曲线的区别表现在以下几个方面：

- 基函数的次数：对于 Bezier 曲线，基函数的次数等于控制顶点数减一；对于 B 样条曲线，基函数的次数与控制顶点数无关。
- 积函数性质：Bezier 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：Bezier 曲线是一种特殊表示形式的参数多项式曲线；B 样条则是一种特殊表示形式的参数样条曲线。
- 局部控制能力：Bezier 曲线缺乏局部性质；B 样条曲线具有局部性质。[1]

3.4 平移变换

执行平移操作，将选中图元的端点值加上一个偏移量，之后重新绘制图元即可得到平移过后的曲线。

3.5 旋转变换

[1]当基准点为原点时，二维旋转变换方程为：

$$\begin{cases} x_2 = x_1 \cos\theta - y_1 \sin\theta \\ y_2 = x_1 \sin\theta + y_1 \cos\theta \end{cases}$$

则对于旋转中心不在原点的旋转变换，只需要先将旋转中心移动至坐标原点，进行旋转变换之后再给得到的坐标点增加一个偏移量即可。

实际代码实现中，旋转得到的x_2坐标为：

$$(x_0 + (x_1 - x_0) * \cos\theta - (y_1 - y_0) * \sin\ \theta$$

y_2坐标为：

$$y_0 + (x_1 - x_0) * \sin\theta + (y_1 - y_0) * \cos\theta$$

3.6 缩放变换

[1]二维缩放变换可通过将每个顶点的坐标值(x_1, y_1)乘以比例系数S_x和S_y来产生变换的坐标(x_2, y_2)而实现：

$$\begin{cases} x_2 = x_1 \cdot s_x \\ y_2 = y_1 \cdot s_y \end{cases}$$

缩放系数S_x为在 x 方向对物体的缩放，S_y在 y 方向对物体的缩放。缩放系数S_x和S_y可赋为任何正数。缩放系数小于 1 缩小物体，缩放系数大于 1 则放大物体。

- 当$S_x = S_y = 1$时，为恒等缩放变换，即图形变换后不变形。
- 当$S_x = S_y > 1$时，图形沿两个坐标轴等比例放大。
- 当$S_x = S_y < 1$时，图形沿两个坐标轴等比例缩小。
- 当$S_x \neq S_y$时，图形沿两个坐标轴方向作非均匀的比例变换。

实际代码实现中，缩放得到的x_2坐标为：

$$x_1 * s + x_0 * (1 - s)$$

y_2坐标为：

$$y_1 * s + y_0 * (1 - s)$$

3.7 线段裁剪

3.7.1 Cohen-Sutherland 算法

Cohen-Suthrland 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，从而大大减少交的计算量，提高线段裁剪算法的效率。编码算法以如图所示的 9 个区域为基础，根据每条线段的端点坐标所在的区域，给每个端点赋以四位二进制码(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。

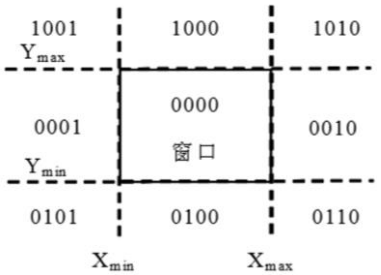


图 5 区域编码

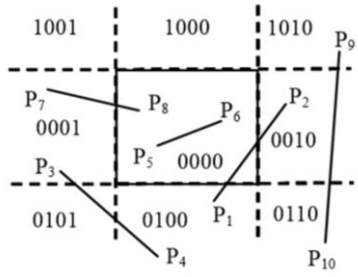


图 6 线段裁剪

四位区域码的 4 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。判断线段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的操作的方法，根据线段与裁剪窗口的关系可分三种情况处理：

- 线段完全在裁剪窗口之内。两个端点的区域码均为 0000，则该线段完全在裁剪窗口之内。
- 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。
- 其他，需要进行求交运算。

求交过程为：首先，对一条线段的外端点按上、下、右、左的顺序与每条裁剪边界比较来确定应该裁剪掉多少线段，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，则不需要进一步判断)。在实现算法时，不必把线段与每条窗口边界依次求交，只需要按顺序检测到端点的区域码的某位不为 0 时，才把线段与对应的窗口边界进行求交。[1]

3.7.2 Liang-Barsky 算法

Liang-Barsky 算法的解决思路是，将待裁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。

将二维裁剪问题转化为一维问题后，只要生成诱导窗口即可。如图 7 所示，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；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 计算如下：

$$Q_1Q_2 = lim \cap Window = LR \cap TB$$

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 Q1Q2 对应的参数区间，如图 8 所

示。

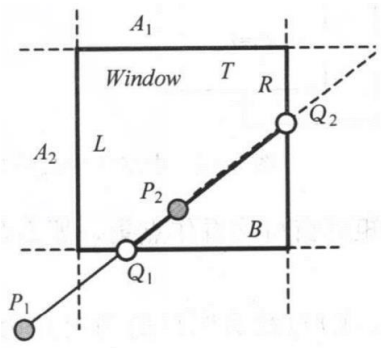


图 7 一维诱导窗口生成示意

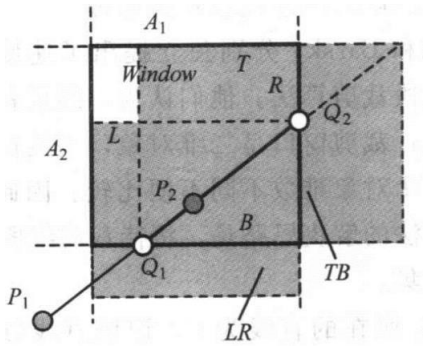


图 8 一维诱导窗口的计算

这样，P1P2 的可见部分 VW 可计算为

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

在实际代码实现中，首先将线段交点的参数初始化为

$$\text{Uone} = 0$$
$$\text{Utwo} = 1$$

计算p_k和q_k的值(其中 k = 0, 1, 2, 3)

$$\text{p} = [\text{x0} - \text{x1}, \text{x1} - \text{x0}, \text{y0} - \text{y1}, \text{y1} - \text{y0}]$$
$$\text{q} = [\text{x0} - \text{x\_min}, \text{x\_max} - \text{x0}, \text{y0} - \text{y\_min}, \text{y\_max} - \text{y0}]$$

对于每个 p 值，分三种情况讨论，分别表示进入端，离开端和空的情况。最后对存在性进行判断即可得到裁剪过后的线段端点。[1]

```
    for i in range(4):
        if p[i] < 0:
            Uone = max(Uone, q[i] / p[i])
        elif p[i] > 0:
            Utwo = min(Utwo, q[i] / p[i])
        elif p[i] == 0 and q[i] < 0:
            return ''
if Uone > Utwo:
    return ''
```

3.7.3 对比分析

通常，梁友栋-Barsky 算法比 Cohen-Sutherland 算法更有效，因为需要计算的焦点数量减少了，更新u_1, u_2仅需要一次除法；线段与窗口的交点仅计算一次就能得出u_1和u_2的最后值。相比之下，即使一条线段完全落在窗口之外，Cohen-Sutherland 也要对它反复求交点，而且每次求交计算都需要用到乘法和除法。[1]

4 CLI 程序逻辑

从 input_path.txt 文件中按行读取命令，并按空格分隔。提取信息后，将图元信息存储在 item_dict 中。当时别到编辑图元信息，调用算法模块的图元编辑函数，更改已经保存的图元信息。当识别到保存画布命令时，按照图元类型调用算法，得到绘制坐标点后保存画布。

5 代码鲁棒性和用户友好性

5.1 鼠标绘制多边形首尾连接方式不符合一般用户习惯

命令行状态下调用的 draw_polygon 函数：

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

直接应用于图形界面产生的问题：绘制得到的多边形首位保持连接，不符合一般用户的习惯。
解决方案：在 CLI 和 GUI 状态下调用不同的 draw_polygon 函数，定义 my_draw_polygon:

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

增加对鼠标双击信号的响应，在 mouseDoubleClickEvent 函数中将第 0 个点坐标再次加入 list:

```
elif self.status == 'polygon':
    if self.temp_item is not None:
        self.temp_item.p_list.append(self.temp_item.p_list[0])
        self.item_dict[self.temp_id] = self.temp_item
        self.list_widget.addItem(self.temp_id)
        self.finish_draw()
```

5.2 绘图完成之前切换菜单栏选项

初始版本存在的问题：绘制多边形过程中，双击之前切换菜单栏选项将导致图元绘制不完整。
解决方案：增加 judge_finish 函数，在收到菜单切换信号时调用，判断上一图元是否已经绘制完成，同时考虑图元编号的连续性问题，根据 judge_finish 的结果决定是否需要更新图元 id：

```
def start_draw_line(self, algorithm, item_id):
    if not self.judge_finish():
        item_id = str(int(item_id)+1)
```

若上一图元绘制尚未完成，在切换状态之前额外执行一次绘图结束动作：

```
def judge_finish(self) -> bool:
    if self.status == 'polygon':
        if self.temp_item is not None:
            self.temp_item.p_list.append(self.temp_item.p_list[0])
            self.item_dict[self.temp_id] = self.temp_item
            self.list_widget.addItem(self.temp_id)
            self.temp_id = self.main_window.get_id()
            self.temp_item = None
            self.updateScene([self.sceneRect()])
            return False
    return True
```

5.3 在没有选中图元的情况下选择图元编辑操作

针对图元编辑增加判断操作，selected_id 的值为空时返回 False，不执行操作，并输出提示信息。
以平移操作为例，输出提示信息：

```
def translate_action(self):
    if not self.canvas_widget.start_translate(str(self.item_cnt - 1)):
        self.statusBar().showMessage('您还没有选择要平移的图元！')
    else:
        self.statusBar().showMessage('图元平移')
```

判断是否选中图元：

```
def start_translate(self, item_id) -> bool:
    if not self.judge_finish():
        self.temp_id = str(int(item_id) + 1)
    if self.selected_id == '':
        self.status = ''
        self.basepoint = [-1, -1]
        return False
    self.status = 'translate'
    self.temp_id = self.selected_id
    self.temp_item = self.item_dict[self.temp_id]
    self.basepoint = [-1, -1]
    return True
```

5.4 对由单点组成的椭圆执行缩放操作

设置了椭圆的最小半径，即在 MousePressEvent 中添加不只一个点。工作状态为椭圆时，单击屏幕但不拖动，将生成一个直径为 1 像素的小圆，而不是仅在 list 中添加单个点。

```
elif self.status == 'ellipse':
    self.temp_item = MyItem(self.pen_width, self.color, self.temp_id, self.status,
                            [[x - 1, y + 1], [x + 1, y - 1]],self.temp_algorithm)
    self.scene().addItem(self.temp_item)
```

5.5 画布已经绘制了图元的情况下调整画布大小，图元将与画布等比例缩放

新画布的大小由用户在弹出的对话框中输入获得，之后遍历图元列表对已经加入图元列表的图元执行缩放操作。需要注意的是，正在绘制但还没有绘制完成的图元不会被缩放。

```
self.h = height
self.w = width
for item in self.canvas_widget.item_dict:
    self.canvas_widget.item_dict[item].p_list = alg.scale(self.canvas_widget.item_dict[item].p_list, 0,
                                                          0, s)
self.scene.setSceneRect(0, 0, width, height)
self.canvas_widget.setFixedSize(width, height)
```

6 遇到的问题和解决方案

6.1 双击后在图元列表中加入空值

原因：在 GUI 界面上执行双击操作时，将先产生一个单击信号，再产生一个双击信号，在初始版本的 mouseDoubleClickEvent 函数中没有对多边形以外的状态进行处理。

解决方案：在 mouseDoubleClickEvent 函数中增加对图元类型状态的响应，执行和 mousePressEvent 中相同的操作。

6.2 切换图元类型时标号不连续

原因：在原始版本中，每次切换菜单选项，都会调用 get_id 函数，例如：

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

每绘制完成一个图元，调用 finish_draw 函数中同样会调用 get_id 函数。在 get_id 函数中，执行了对 item_cnt 参数的递增操作。因此切换菜单项目时就会发生对 get_id 函数的重复调用，导致图元编号跳跃。

解决方案：在信号函数增加对 item_cnt 参数值的判断，仅当 item_cnt 的值为 0 时调用 get_id 函数。例如：

```
def line_naive_action(self):
    if self.item_cnt == 0:
        self.canvas_widget.start_draw_line('Naive', self.get_id()) # 这里
    else:
        self.canvas_widget.start_draw_line('Naive', str(self.item_cnt-1))
    self.statusBar().showMessage('Naive算法绘制线段')
```

6.3 递归执行Bezier曲线绘制算法计算耗时过长

原因：控制点数量过多时，将进行多层递归调用，其间将执行大量无用计算，占用大量空间。初始递归代码为：

```
def deCasteljau(p_list, n, i, t):
    if n == 0:
        return p_list[i]
    else:
        p1 = deCasteljau(p_list, n - 1, i, t)
        p2 = deCasteljau(p_list, n - 1, i + 1, t)
        return [(1 - t) * p1[0] + t * p2[0], (1 - t) * p1[1] + t * p2[1]]
```

解决方案：用循环(动态规划)算法代替原来的递归。更改后代码：

```
def bezier(p_list, n, t):
    result = [p for p in p_list]
    for i in range(1, n):
        for j in range(n - i):
            result[j] = [(1 - t) * result[j][0] + t * result[j + 1][0], (1 - t) * result[j][1] + t * result[j + 1][1]]
    return result[0]
```

6.4 在没有选中图元情况下切换到图元编辑状态，画布空白处双击鼠标导致程序崩溃

原因：未选中图元状态下进入图元编辑状态不会执行具体操作，但双击之后出发双击信号，其中包括了对图元状态的判断，并执行了部分操作。

解决方案：在返回 False 之前及时将 self.status 置为空。

6.5 清空画布时程序崩溃

原因：清空画布的同时清空 list_widget，将触发 selected_changed 函数，此时并没有选中图元，selected 为空导致了错误。

解决方案：在 selected_changed 函数中增加对当前画布上图元数量的判断操作，若当前画布上存在的图元数量为 0，则你不执行操作。

```
def selection_changed(self, selected):
    if self.main_window.item_cnt == 0:
        return
```

6.6 对图元进行旋转、缩放、平移操作后鼠标点选功能失效

原因：最初一般对算法模块的计算结果使用了浅拷贝，结果导致 ItemAt 函数无法对图元当前位置进行正确的判断。

解决方案：对更新后的图元信息进行深拷贝。

7 额外的图形学功能

7.1 调整画布大小

添加菜单项和消息响应函数。在收到修改画布大小信号时，弹出对话框，要求输入要修改到的画布高度和宽度。

```
def reset_canvas_action(self):
    dialog = QDialog()
    layout = QFormLayout(dialog)
    heightEdit = QLineEdit(dialog)
    widthEdit = QLineEdit(dialog)
    buttonBox = QDialogButtonBox(QDialogButtonBox.Ok | QDialogButtonBox.Cancel, dialog)
    buttonBox.accepted.connect(dialog.accept)
    buttonBox.rejected.connect(dialog.reject)
    layout.addRow("画布高度：", heightEdit)
    layout.addRow("画布宽度：", widthEdit)
    layout.addWidget(buttonBox)
    dialog.setWindowTitle("请输入两个整数")
```

用户按下提交键后，进行过所提交的数据类型的正确性判断之后，修改窗口的大小。

```
if dialog.exec():
    if self.is_number(heightEdit.text()) and self.is_number(widthEdit.text()):
        height = int(heightEdit.text())
        width = int(widthEdit.text())
        s1 = height / self.h
        s2 = width / self.w
        s = 1
        if s1 <= 1 and s2 <= 1:
            s = max(s1, s2)
        else:
            s = min(s1, s2)
        self.h = height
        self.w = width
        for item in self.canvas_widget.item_dict:
            self.canvas_widget.item_dict[item].p_list = alg.scale(self.canvas_widget.item_dict[item].p_list, 0,
                                                                  0, s)
        self.scene.setSceneRect(0, 0, width, height)
        self.canvas_widget.setFixedSize(width, height)
```

7.2 清空画布

先将 item_cnt 的值置为 0，下一步清空当前的 Selection。之后清空 list_widget，最后清空 canvas_widget 并刷新界面即得到与初始状态相同的空白画布。

7.3 保存画布

点击保存按钮后，弹出文件对话框。设置文件格式，并获得文件路径和名称。

```
def save_action(self):
    dialog = QFileDialog()
    filename = dialog.getSaveFileName(filter="Image Files(*.jpg *.png *.bmp)")
```

在确定正确设置文件名之后存储图片。

```
if not filename[0] == '':
    pixmap = QPixmap()
    pixmap = self.canvas_widget.grab(self.canvas_widget.sceneRect().toRect())
    pixmap.save(filename[0])
```

7.4 设置画笔颜色

在 MyItem 中添加 Qpen 类型的成员变量，存储每个图元的颜色。在 MyCanvas 中添加 color 成员变量，存储绘制下一图元需要使用的颜色。在 Paint 函数中，每次绘制图元前调用 setPen 函数，即可将图元色彩显示在画布上。

```
for p in item_pixels:
    painter.setPen(self.pen)
    painter.drawPoint(*p)
```

当单击更改颜色菜单项时，弹出颜色对话框，选择并存储颜色，改变 MyCanvas 对象中的 color 变量。若此时有已经选中的图元，根据所选中图元 id，更改对应的 pen。

```
def set_color(self, item_id):
    color_chosen = QColorDialog()
    self.color = color_chosen.getColor()
    if not self.judge_finish():
        self.temp_id = str(int(item_id) + 1)
    if self.selected_id != '':
        self.temp_item.pen.setColor(self.color)
```

7.5 设置画笔宽度

在 MyCanvas 中添加 width 成员变量，具体逻辑与颜色设置类似。

7.6 鼠标点选图元

设置自定义状态'selection'，selection 状态下执行鼠标单击将调用 itemAt 函数，若发现当前鼠标点选位置上存在图元，则调用 selection_changed 函数，进行选中图元的切换。

```
elif self.status == 'selection':
    item = self.scene().itemAt(pos, QtGui.QTransform())
    if item is not None:
        self.selection_changed(item.id)
self.updateScene([self.sceneRect()])
super().mousePressEvent(event)
```

7.7 快捷工具栏

在 MainWindow 中添加工具栏，并添加图形按钮和相应的信号响应函数。

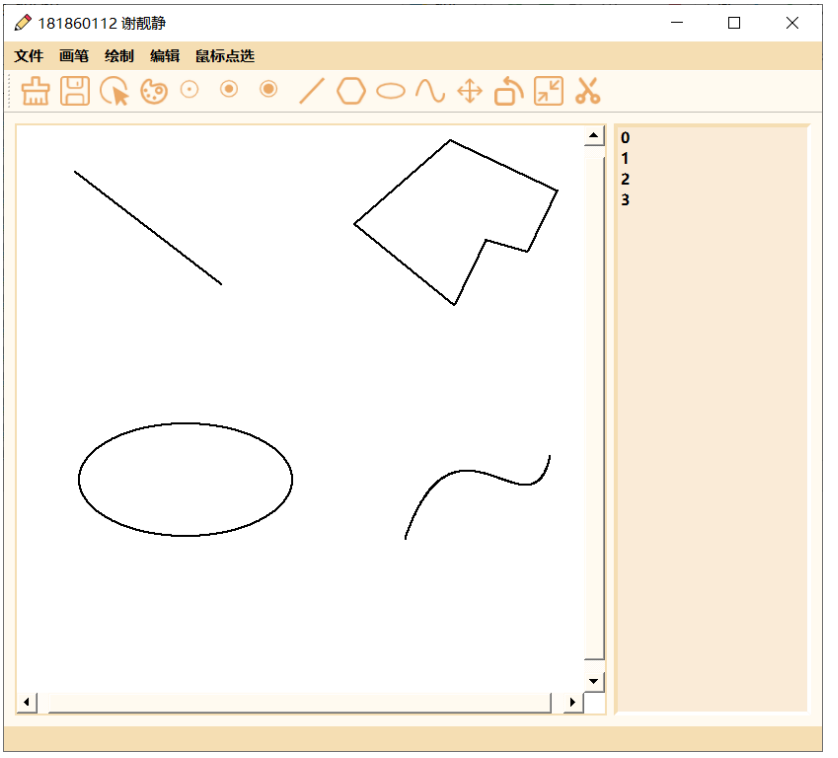
```
# 设置工具栏
toolBar = QToolBar()
self.addToolBar(toolBar)
# 添加图形按钮
clear = QAction(QIcon('images\\清空.png'), "清空画布", toolBar)
clear.setStatusTip("清空画布")
clear.triggered.connect(self.clear_canvas_action)
toolBar.addAction(clear)
```

8 最终效果演示

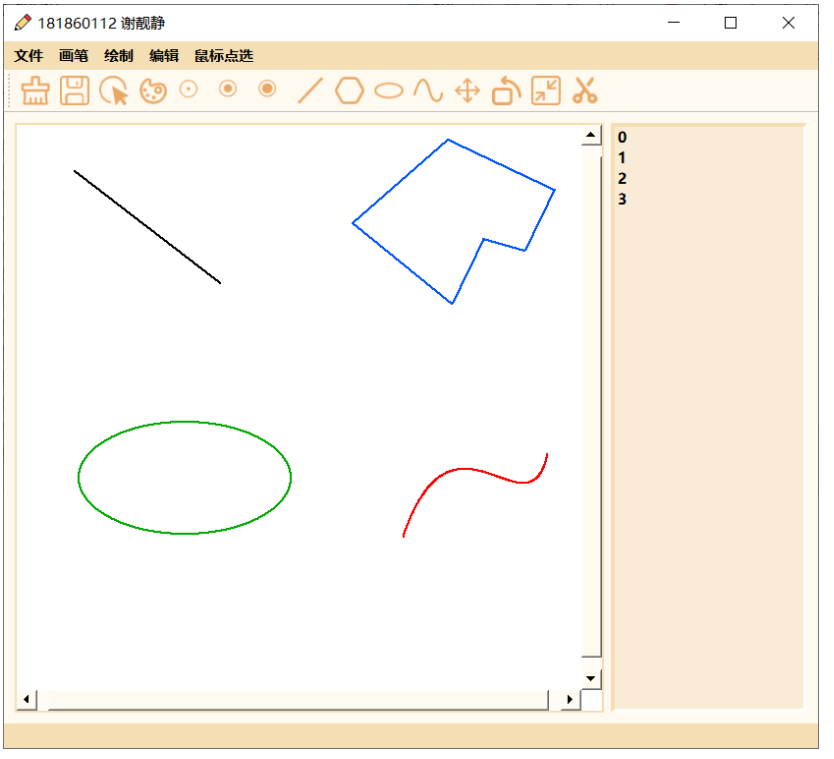
8.1 GUI部分

8.1.1 程序外观

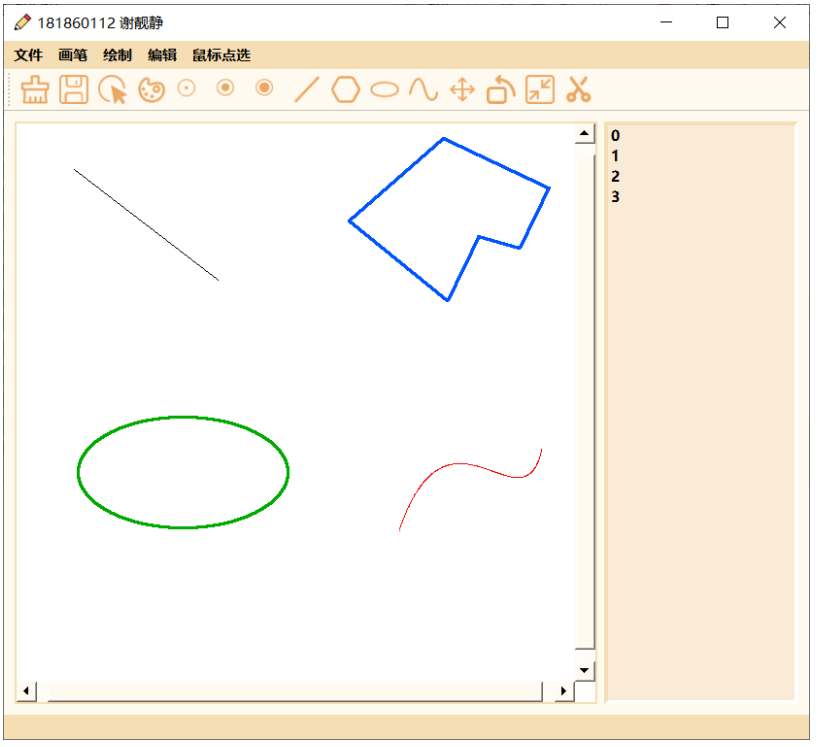
8.1.2 直线、多边形、椭圆、曲线绘制



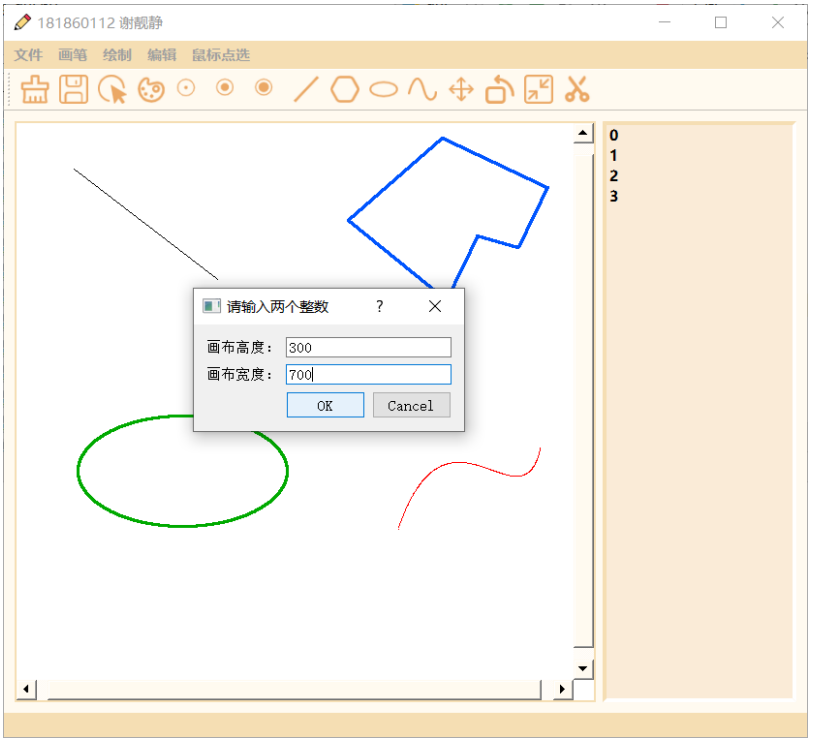
8.1.3 设置画笔颜色

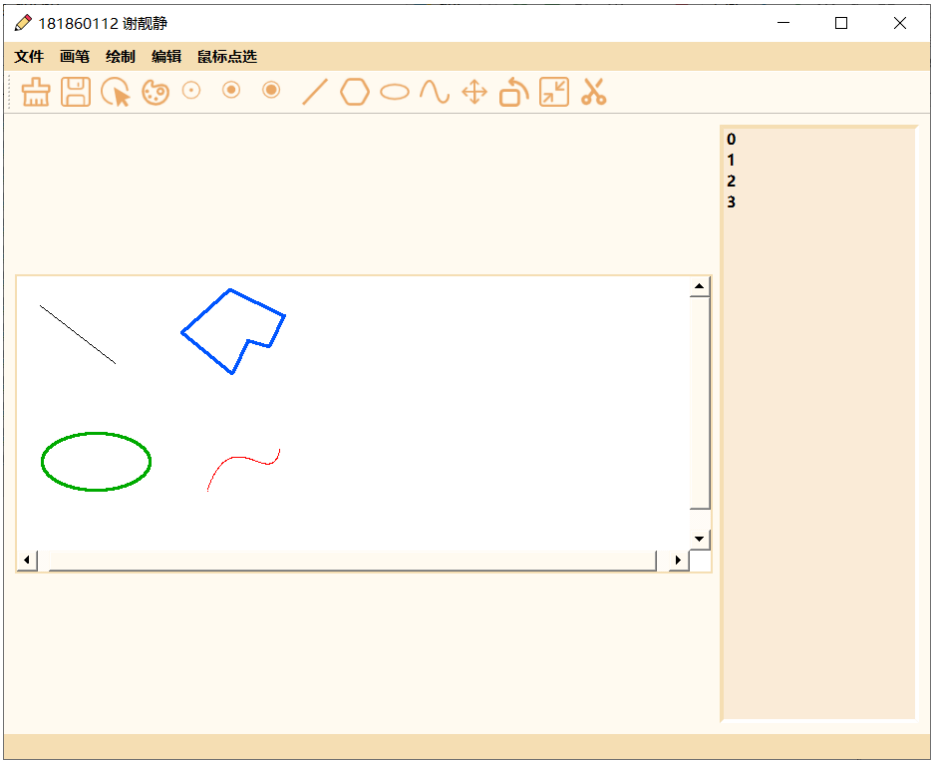


8.1.4 设置画笔粗细

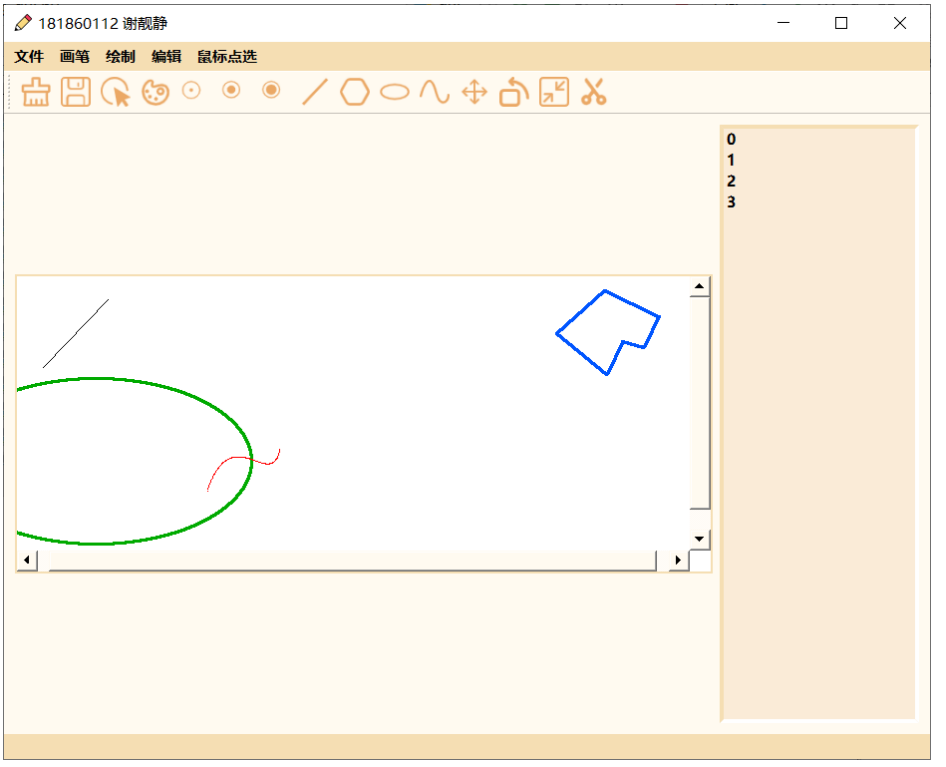


8.1.5 调整画布大小（图元等比例缩放）

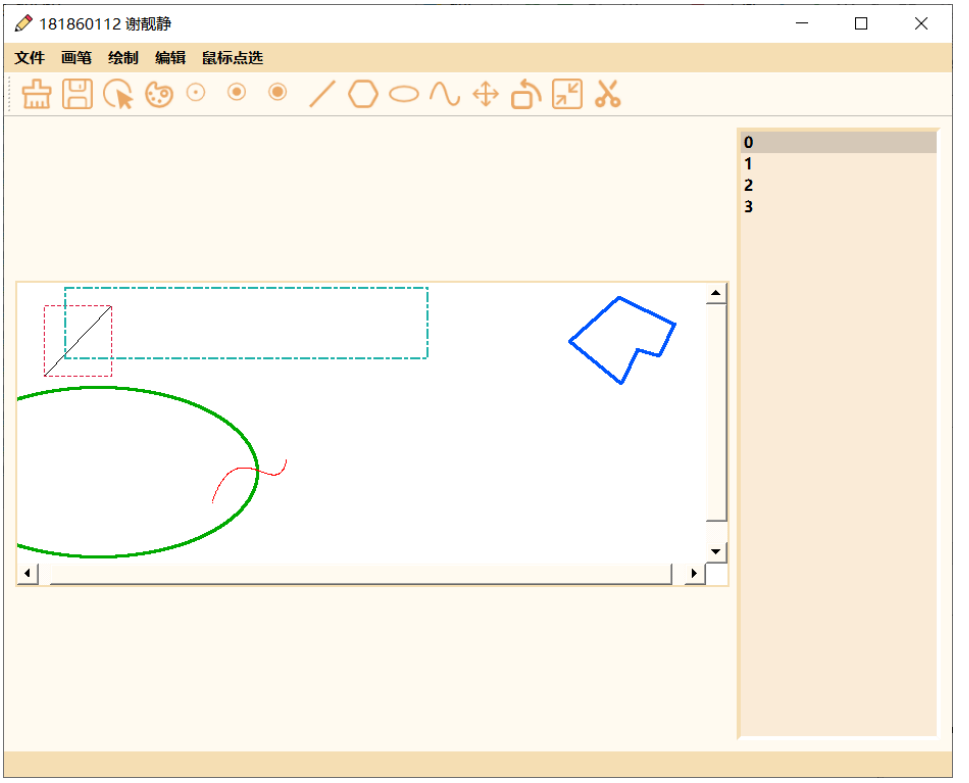


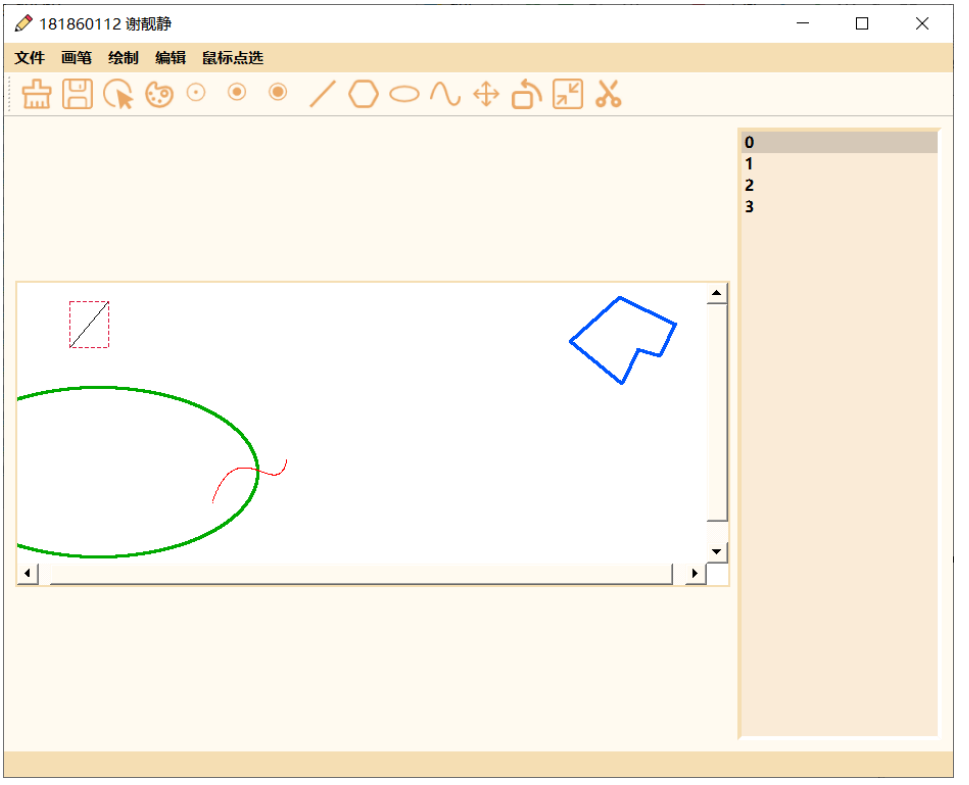


8.1.6 图元平移、旋转、缩放

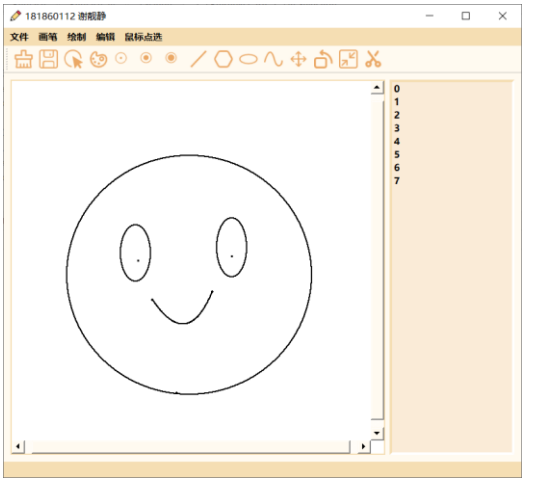


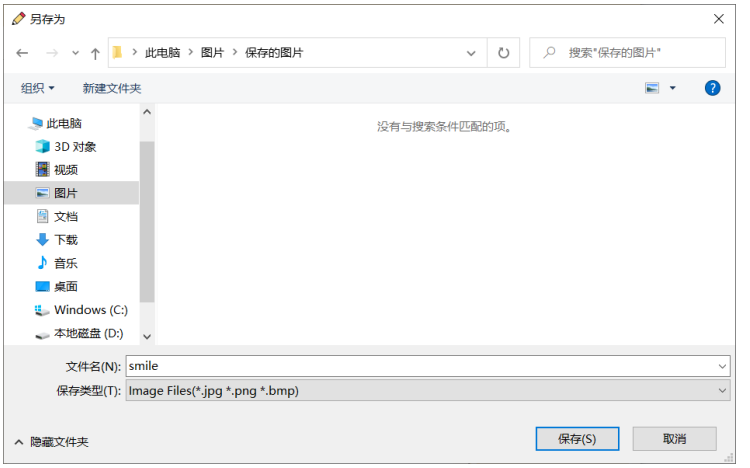
8.1.7 图元裁剪

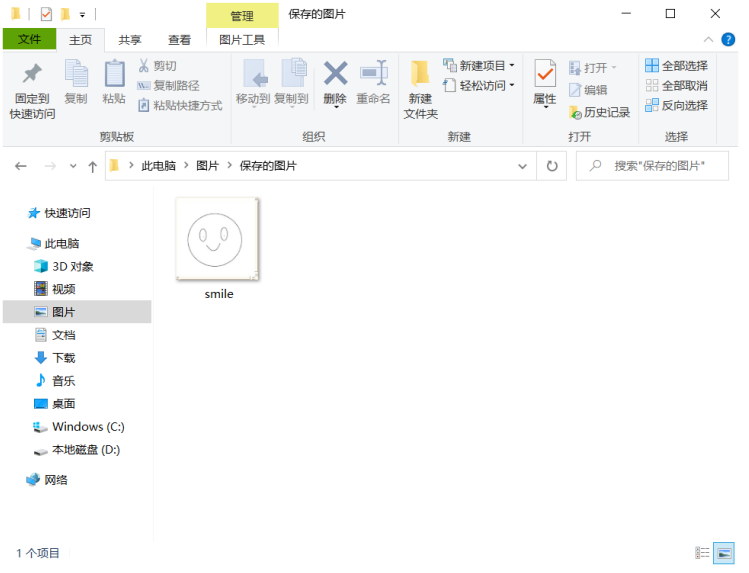

181860112 谢靓静
文件 画笔 绘制 编辑 鼠标点选
0
1
2
3


181860112 谢靓静
文件 画笔 绘制 编辑 鼠标点选
0
1
2
3

8.1.8 保存画布

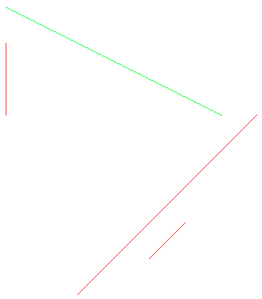

181860112 谢靓静
文件 画笔 绘制 编辑 鼠标点选
0
1
2
3
4
5
6
7

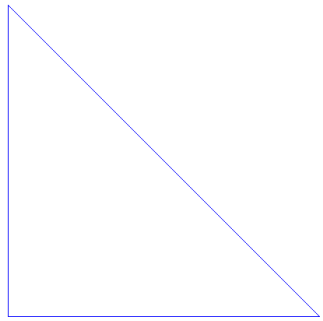

另存为
此电脑 › 图片 › 保存的图片
搜索"保存的图片"
组织
新建文件夹
此电脑
3D 对象
视频
图片
文档
下载
音乐
桌面
Windows (C:)
本地磁盘 (D:)
没有与搜索条件匹配的项。
文件名(N): smile
保存类型(T): Image Files(*.jpg *.png *.bmp)
隐藏文件夹
保存(S)
取消

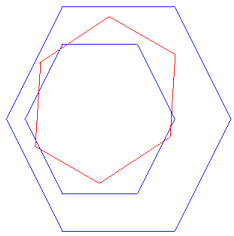

管理
保存的图片
文件
主页
共享
查看
图片工具
固定到快速访问
复制
粘贴
剪切
复制路径
粘贴快捷方式
剪贴板
移动到
复制到
删除
重命名
组织
新建项目
轻松访问
新建文件夹
新建
属性
打开
编辑
历史记录
打开
全部选择
全部取消
反向选择
选择
此电脑 › 图片 › 保存的图片
搜索"保存的图片"
快速访问
此电脑
3D 对象
视频
图片
文档
下载
音乐
桌面
Windows (C:)
本地磁盘 (D:)
网络
smile
1 个项目

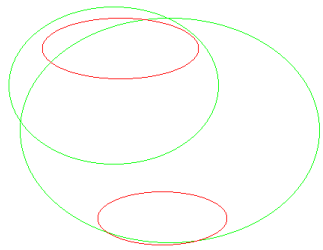
8.2 CLI部分

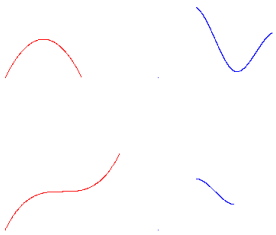
按照给出的测试文件绘制的到的图片如下：











9 参考文献:

[1] 孙正兴. 计算机图形学教程[M]. 机械工业出版社, 2006.

[2] Rogers D , 罗杰斯, Rogers. Procedural elements for computer graphics[M]// Procedural elements for computer graphics =. China Machine Press, 2002.

[3] 中点圆、椭圆生成算法：https://blog.csdn.net/u012866328/article/details/52607439

[4] 贝塞尔曲线：https://www.icourse163.org/learn/CAU-45006?tid=1450413503#/learn/announce

[5] B 样条法绘制曲线：https://www.icourse163.org/learn/CAU-45006?tid=1450413503#/learn/announce

[6] B 样条法绘制曲线递归算法：https://en.wikipedia.org/wiki/De_Boor%27s_algorithm