《计算机图形学》系统报告

201220086，景雅，201220086@smail.nju.edu.cn

2023 年 2 月 12 日

1 综述

1.1 实验内容

本次实验使用 Python3 语言编程，在已有的框架代码上实现一个简单的绘图系统。其主要功能包括对线段、多边形、椭圆、曲线的绘制、平移、旋转、裁剪，画布的重置、保存，设置画笔颜色等。

9 月进度：配置实验环境（python3.8+Anaconda+Pycharm），理解代码框架，实现线段绘制算法 DDA。

10 月进度：完成线段、椭圆绘制。

11 月进度：完成曲线绘制，图元平移、旋转、缩放，线段裁剪算法，完善命令行实现。

12 月–2 月进度：完成 GUI 图形界面代码，实现拓展功能。

1.2 代码框架

- 核心算法模块：cg_algorithms.py

  依赖 math 库，实现线段、多边形、椭圆、曲线的绘制和平移、旋转、缩放变换与线段裁剪。

- 命令行界面：cg_cli.py

  依赖第三方库 numpy 和 Pillow，读取包含了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。

- 用户交互界面：cg_gui.py

  依赖 PyQt5 库，以鼠标交互的方式，通过鼠标事件获取所需参数并调用核心算法模块中的算法将图元绘制到屏幕上，或对图元进行编辑。

2 算法介绍

2.1 绘制线段

2.1.1 Naive

算法原理 Naive 算法采用最朴素的线段绘制方法，利用两点坐标计算斜率，根据斜率与横坐标增量 Δx 计算 Δy，从而确定屏幕像素位置。具体实现时：

(1) 首先排除特殊情况，即横坐标不变、斜率不存在时，直接将纵坐标的值 +1 依次描点；
(2) 其余情况，选取横坐标大减横坐标小的点的坐标计算斜率 k，根据横坐标依次累加，纵坐标增量 $\Delta y = y_0 + k\Delta x$，描点。

算法评估 Naive 算法直接用数学公式计算，简单易懂，但每次计算都需要做乘法和加/减法运算，复杂度较高。

2.1.2 DDA 算法

算法原理 数值微分算法（Digital Differential Analyzer）使用增量思想绘制直线。

由于直线的一阶导数连续，即 Δx 与 Δy 成正比，可通过将当前位置的坐标在每个方向上增加一定的增量，来确定下一个点的位置。例如，在横坐标上以单位间隔对线段取样（取 $\Delta x = 1$），则对应纵坐标增量 $\Delta y = k\Delta x$，横纵坐标每次取点时增加对应增量。

此外，算法实现时，为使得生成的线段尽量连续均匀，相邻两个像素之间坐标最多相差一个像素。因此，考虑斜率大小选取横坐标或纵坐标作为单位间隔，即

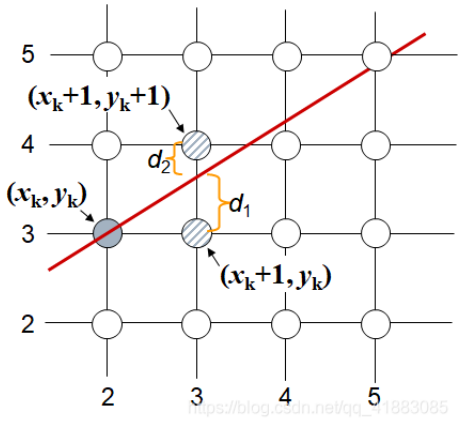
$$\begin{cases} x_{i+1} = x_i + 1\ ,\ y_{i+1} = y_i + k & k <= 1 \\ x_{i+1} = x_i + \frac{1}{k}\ ,\ y_{i+1} = y_i + 1 & k > 1 \end{cases}$$

算法评估 DDA 利用光栅特性消除了直线方程中的乘法，在 x 和 y 方向选用合适的增量来逐步推导各像素位置，计算速度比 Naive 方法快。但是，浮点增量迭加导致取整误差的长期累积，使计算的像素位置偏离实际线段，且程序运行时取整操作和浮点运算十分耗时。采用将增量 k 与 $\frac{1}{k}$ 分离成整数和小数部分，简化成整数操作，可以提升算法性能。

2.1.3 Bresenham 算法

算法原理 Bresenham 是一种光栅线段生成算法，通过引入整形参量定义来衡量两候选像素与线段上实际点之间在某个方向上的相对偏移，并利用整形参量的符号正负确定最接近实际路径的像素。

对直线方程 $y = mx + b$，考虑斜率 $m < 1$ 时，以横坐标单位间距取样。对已确定的像素位置 (x_k, y_k)，下一个像素位置必然在 (x_{k+1}, y_k) 或 (x_{k+1}, y_{k+1}) 之间。在 x_{k+1} 处，用 d_1 与 d_2 表示候选像素与线段上点的垂直偏移，如下图所示。



根据直线的方程，误差（距离差）为

$$\begin{cases} d1 = y - y_k = m(x_k + 1) + b - y_k \\ d2 = y_{k+1} - y = y_{k+1} - m(x_k + 1) + b \end{cases}$$

比较两者的大小，作差得到：$d_1 - d_2 = 2m(x_k + 1) - 2y_k + 2b - 1$

基于上述公式，代入 $m = \frac{\Delta y}{\Delta x}$，得到本算法的核心——决策参数：

$$p_k = \Delta x(d_1 - d_2) = 2\Delta y x_k - 2\Delta x y_k + c$$

其中 $\Delta x = 1$，$c = 2\Delta y + \Delta x(2b - 1)$ 是一个常量，此时方程简化成整数运算，且决策参数 p_k 的正负就标示着两个候选像素的误差大小。

显然，绘图时应该选择误差更小的，即 $p_k > 0$ 时选择 y_{k+1}，否则选择 y_k。

同时，利用横纵坐标整数递增，也可以计算后续的决策参数值。通过计算 $p_{k+1} - p_k$，并化简可以得到

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x(y_{k+1} - y_k)$$

因此，根据线段的起始像素位置 (x_0, y_0) 和第一个参数 $p_0 = 2\Delta y - \Delta x$，反复对决策参数进行递归运算，就可以得到线段上所有的离散点集，算法流程概括如下：

(1) 计算常量值 Δx, Δy, $2\Delta y$, $2\Delta y - 2\Delta x$；

(2) 循环扫描，对 p_k 的值进行判断；

(3) 若 p_k 的值为正，则 y+1 选点，同时 p_k += $2\Delta y - 2\Delta x$；

(4) 反之若 p_k 的值为负，则 y 不变选点，同时 p_k += $2\Delta y$。

算法评估 Bresenham 算法的运算全部为整数的加减法或判断决策参数的正负，复杂度较低，使得算法效率得到明显提高。此外，可以利用并行计算，加快图像生成速度。

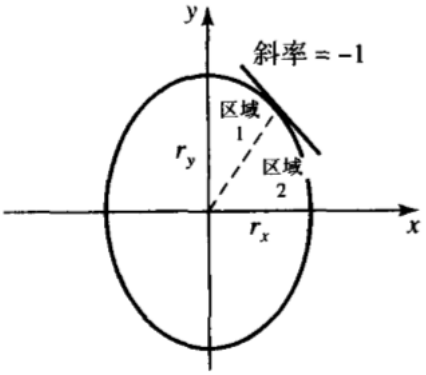
2.2 绘制多边形

多边形本质上是多条线段首尾相连而成，直接调用对应算法的线段绘制方法，并在命令行和 GUI 功能中添加相应代码接口即可。

2.3 绘制椭圆

中点圆生成算法原理与 Bresenham 算法相似，以决策参数的增量计算为基础，通过设定在每一步取样时寻找最接近圆周像素的决策参数来显示圆。为避免距离的平方根运算，通过检验两像素间的中间位置在椭圆内或边界外，来确定椭圆周上的像素。

针对先前已经实现的算法不难发现，扫描需要根据斜率考虑方向。由椭圆的性质可知，椭圆绘制需要分段。依据椭圆斜率将第一象限的椭圆分为两部分，即区域 1 和区域 2，如下图所示。两区域分割条件为 $dy/dx = -1$，可得到分段点：$2r_y^2 x >= 2r_x^2 y$，此时移出区域 1 进入区域 2。



定义椭圆函数作为算法的决策参数，在每个取样位置，对沿椭圆轨迹两个候选像素的中点求值。考虑区域 1，决策像素为

$$p_k = F(x_{k+1}, y - \frac{1}{2}) = (x_k + 1)^2 + (y_k - \frac{1}{2})^2 - r^2$$

决策变量的符号表示着两候选像素的中点是否在圆内，从而选择距离更近的像素。推导得到决策变量的增量计算公式如下：

区域一：

$$\begin{cases} p1_{k+1} = p1_k + 2r_y^2 x_k + 3r_y^2 & p1_k < 0 \\ p1_{k+1} = p1_k + 2r_y^2 x_k + 3r_y^2 - 2r_x^2 y_k + 2r_x^2 & p1_k >= 0 \end{cases}$$

区域二：

$$\begin{cases} p2_{k+1} = p2_k - 2r_x^2 y_k + 3r_x^2 & p2_k <= 0 \\ p2_{k+1} = p2_k + 2r_y^2 x_k + 3r_x^2 - 2r_x^2 y_k + 2r_y^2 & p2_k > 0 \end{cases}$$

同时，根据公式计算决策变量的初始值 $p_0 = r_y^2 - r_x^2 r_y + \frac{r_x^2}{4}$。

因此，对于中心在 (x_c, y_c)、长短轴径分别为 r_x 和 $r_y (r_y < r_x)$ 的椭圆，生成算法的整体流程如下：

(1) 计算决策变量的初始值，并选择椭圆最上方的点；

(2) 沿着 x 轴循环扫描，根据公式更新决策变量和 y 值，直到结束区域 1；

(3) 记忆区域 1 的最后一个位置，根据公式计算新的决策变量初始值；

(4) 沿着 y 轴循环扫描，根据对应公式计算决策变量和 x 值，直到结束区域 2；

(5) 由得到的第一象限全部图像，对称得到其他三个象限中的点；

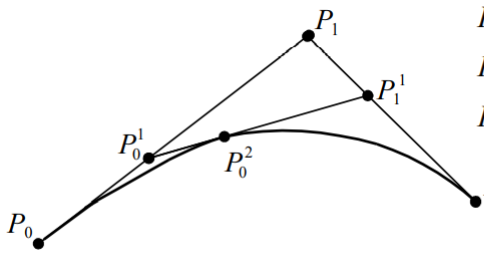
(6) 根据椭圆中心平移图像坐标。

2.4 绘制曲线

2.4.1 Bezier 曲线绘制算法

一阶 Bezier 曲线是一条直线，相当于在两个控制点之间做插值，即一个点在两个点之间移动。它的轨迹是一条直线，也就是两个控制点之间的曲线。用公式表达为：$B_1(t) = P_0 + (P_1 - P_0)t$，这里的 t 是曲线的参数化表示中联结坐标 x 与 y 的参数。

对于有三个控制点的情况，此时生成二阶 Bezier 曲线，可以看作是两层的一阶 Bezier 叠加，如下图所示。



$$P_0^1 = (1-t)P_0 + tP_1$$
$$P_1^1 = (1-t)P_1 + tP_2$$
$$P_0^2 = (1-t)P_0^1 + tP_1^1$$

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

因此，更高阶的 Bezier 曲线也能一步步地降阶，最终化为一阶处理。利用德卡斯特里奥递推算法产生曲线上的点，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 Bezier 曲线型值点 P(u) 的过程。对于某一特定参数 u，其计算公式为：

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

其中，r=0 时计算结果为控制顶点本身，而曲线上的型值点为 $P(u) = P_0^n$。

2.4.2 B-spline 曲线绘制算法

Bezier 曲线存在不足之处，例如特征多边形顶点的数量决定了 Bezier 曲线的阶次，缺少灵活性与局部性。因此，引入 B 样条基函数代替 Bernstein 基函数，改进多项式次数和整体逼近的缺点。

B 样条基函数的 deBook-Cox 递推关系公式为：

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

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

给定 n+1 个控制顶点，定义一条 k 次 B 样条曲线需要 n+1 个 k 次 B 样条基函数，由此可以得到参数曲线 $P(u)$ 公式为：

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

实验中绘制三次四阶 B 样条曲线，绘制时利用一个大小为 4 的移动窗口在控制点数组上滚动，带入上述公式，生成曲线的每个点自然衔接，得到最终曲线。

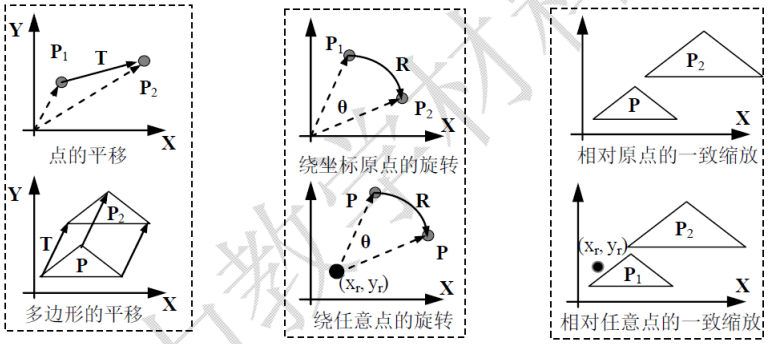
2.5 二维几何变换

2.5.1 基本原理

图形变换的实质是改变图形的坐标位置。一个图形的基本要素是点，点构成线，线构成面（图形），因此只要改变图形上的各点坐标位置，整个图形就完成了变换。对应上述绘制

的各种图元，变换时绘制规则没有变化，只需变换与绘制该图形有关的“控制点”，例如直线的端点、多边形的顶点、椭圆的中心坐标等，即可完成图形的变换。

二维图形的几何变换包括平移、旋转和缩放等，如下图所示。



(a) 二维平移变换　　(b) 旋转变换　　(c) 比例变换

2.5.2 图元平移

平移变换是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化，即物体上的每个点移动相同的偏移量。平移图形的“控制点”，然后再绘制出图形，即完成了整体图元的平移。

2.5.3 图元旋转

旋转变换是将物体绕着指定点旋转一定角度，规定逆时针方向为正。将所有旋转都先平移到基准点为坐标原点，旋转后再移动到基准点位置。因此，旋转坐标方程为

$$\begin{cases} x = (x_0 - x_r)cos\theta - (y_0 - y_r)sin\theta + x_r \\ y = (x_0 - x_r)sin\theta + (y_0 - y_r)cos\theta + y_r \end{cases}$$

其中基准点坐标为 (x_r, y_r)，旋转角为 θ。

2.5.4 图元缩放

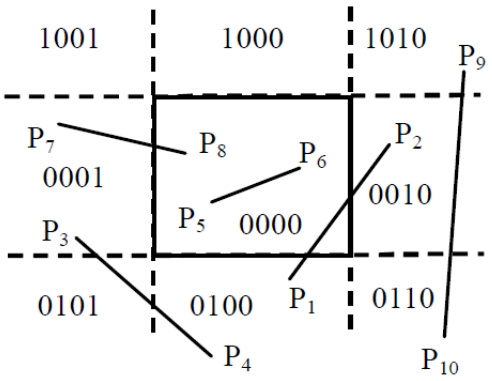
缩放变换相对任意点改变图形的尺寸。依然将所有缩放操作先平移到基准点为坐标原点，将“控制点”乘相应的缩放系数后再移动到基准点位置，即可实现图元的缩放。

2.6 线段裁剪

识别图形在指定区域内、外部分的过程称为裁剪，一般在给定的视区中显示窗口内的部分图形，用来裁剪对象的区域称为裁剪窗口。裁剪处理主要包括两部分内容：判断点在区域内还是区域外，图形元素与区域边界的交点计算。

2.6.1 编码算法（Cohen-Sutherland 算法）

Cohen-Sutherland 算法基于裁剪窗口将整个坐标分成 9 个区域，并对每个落在区域内的端点赋四位二进制码（称为区域码）。其中从左到右依次表示上、下、右、左，赋值为 1 表示端点落在该区域，如下图所示。



线段端点直接与裁剪窗口边界坐标比较，可以得到对应的区域码。然后，基于得到的区域码，判断直线和裁剪窗口的关系。例如二者的区域码都是 0000，则说明整个线段都在裁剪窗口内。若二者相与后结果不为 0000，则说明整个线段都在区域外。剩余情况均为部分在区域内，需要求交点，舍弃部分线段，然后在剩余部分中继续求解。

2.6.2 梁友栋-Barsky 线段裁剪算法

算法将待裁剪线段及裁剪矩形窗口均看作点集，裁剪结果就是两个点集的交集。用参数方程 $P = P_1 + u(P_2 - P_1)$ 表示直线，即可将直线转换成点集表示，u 表示裁剪线段的多少。

参数化表示裁剪条件，发现直线上的某点 (x, y) 在裁剪窗口内，满足如下条件：

$$\begin{cases} x_{wmin} \le x_1 + u\Delta x \le x_{wmax} \\ y_{wmin} \le y_1 + u\Delta y \le y_{wmax} \end{cases}$$

统一用 $up_k \le q_k$ 形式表示，参数 p、q 定义为：

$$\begin{cases} p1 = -\Delta x \quad q1 = x_1 - x_{wmin} \\ p2 = \Delta x \quad q2 = x_{wmax} - x_1 \\ p3 = -\Delta y \quad q3 = y_1 - y_{wmin} \\ p4 = \Delta y \quad q4 = y_{wmax} - y_1 \end{cases}$$

根据 p_k 可以确定线段和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 $q_k \ge 0$，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，是入边。

(3) 当 $pk > 0$，线段从裁剪边界延长线的内部延伸到外部，是出边。

因此，当 p_k 非 0 时，可以计算出线段与边界 k 或延长线交点的 u 值：$u = q_k/p_k$。对每条线段计算裁剪矩形内的线段参数 u_1、u_2，根据两者大小关系与值确定最终的裁剪线段。

2.6.3 算法比较

Cohen-Sutherland 算法直观方便，速度较快，但是需要更多的加法、乘法和位逻辑乘运算，且对于跨越 3 个区域的线段，需要两次判别反复求交点，才能完成裁剪。

梁友栋-Barsky 算法所需计算量较小，更新参数 u_1 和 u_2 仅需要一次除法，确定后直线与窗口边界的交点也只需计算一次，比编码算法更加简洁。

3 命令行界面 CLI 介绍

读取包含了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，主要包含三种指令：

- 图元绘制指令

  图元绘制指令有 drawLine, drawPolygon, drawEllipse, drawCurve.

  遇到相应指令，将图元的编号、控制点坐标、绘制使用的算法等信息存储在字典 item_dict 中，等待保存画布时统一调用核心算法模块程序中的函数进行绘制。
- 图元编辑指令

  图元编辑指令有 translate, rotate, scale, clip.

  核心算法模块程序中，对图元的编辑都是在修改图元的控制点 p_list 坐标，因此遇到相应指令时，找到 item_dict 中存储的对应图元信息，直接调用编辑函数修改图元的控制点坐标，并储存在 item_dict 中。
- 画布操作指令

  画布操作指令有 resetCanvas, saveCanvas, setColor.

  程序储存了画布的长宽和画笔颜色信息，重置画布即修改画布长宽，并将图元字典 item_dict 清空。保存画布操作，遍历图元字典，依次调用核心算法模块程序中的函数，并将画布保存在目标文件夹下。

4 GUI 图形用户界面介绍

4.1 代码框架

- 画布窗体类 MyCanvas

  当前窗口的画布，管理所有图元并监管鼠标操作，根据鼠标的具体操作增添或修改自定义图元对象的信息。
- 自定义图元类 MyItem

  各个图元对象的模板，保存图元的基本信息，绘制图元和它的选中边框。
- 主窗口类 MainWindow

最上层主窗口，负责设计整个窗口的布局和管理菜单栏的选择操作，根据鼠标的选择相应触发画布的操作。

4.2 交互实现

主窗口类中使用 QListWidget 记录已有的图元编号，QGraphicsView 作为画布，维护全局信息。菜单栏中的各个选项命名为 XXX_canvas_act，其执行函数命名为 XXX_action()，通过 XXX_act.triggered.connect(self.XXX_action) 将鼠标所按选项与执行函数连接起来。执行函数再调用画布窗体的不同接口传入参数。

画布窗体类主要成员包括 item_dict（字典，记录所有图元对象），status（当前执行操作状态），selected_id（当前选择图元编号），temp_XXX（处理鼠标事件产生的临时信息），主要函数包括 start_XXX()（接收执行的操作、算法、编号信息），mouseXXXEvent()（鼠标事件），selection_XXX()（处理当前选择情况）。主窗口中的 self.XXX_action 函数调用 start_XXX() 函数，将当前执行操作、算法、编号等信息传入。随后，用户执行鼠标操作，触发相应函数，函数根据当前状态做出相应处理。

绘制线段与椭圆时，由于两个控制点即可确定图像，因此通过鼠标按下确定第一个控制点并创建图元对象，按住拖动鼠标以确定第二个控制点。由于每次拖动都会调用函数更新屏幕，因此可以在拖动过程中实时显示图像。

绘制多边形和曲线需要多个控制点，因此不断按下左键以加入控制点，按下右键结束绘制，每次加入过程都会调用函数显示当前图像。

平移图元和裁剪线段需要鼠标使用时的起始和终止位置坐标，因此实现方法与绘制线段类似，不断记录拖动的第二个点，调用算法修改图元对象的控制点坐标。

旋转和缩放图元需要中心坐标和角度/倍数信息，因此左键首先按下确定中心位置，然后拖动一段距离以确定角度/倍数信息，使用布尔类型表示当前状态。

自定义图元类储存每个图元的具体信息，包括编号、类型、控制点坐标、算法、是否被选择信息。每次更新屏幕时都会调用 paint() 函数，根据图元信息调用 cg_algorithms.py 中的绘制算法和是否绘制选中框（QRectF 类型），即可将每个图元的像素显示在画布上。

注：具体实现效果见“演示.*mp4*”！

4.3 拓展功能

4.3.1 选择图元

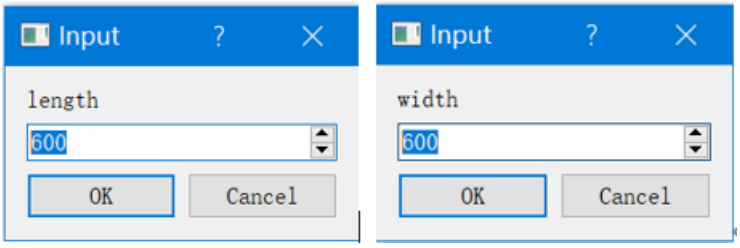
鼠标点击选择图元是一般绘图系统中更常用的选择方法，相较于用编号选择更具实用性。利用框架代码中 boundingRect 函数返回得到的矩形框，使用 item.boundingRect().contains(x, y) 判断坐标是否在该矩形框内，从而可以得到选中的图元编号。最后，使用 self.list_widget.setCurrentItem() 函数关联右侧状态栏的选项。

4.3.2 删除图元

绘制时经常可能出现操作失误，带来冗余图形。选中图元后点击删除，在画布的 item_dict 删去，在状态栏即 QListWidget 中移出 Item，此处注意要将所有该图元信息在代码中全部删除，否则后续可能带来未知错误。

4.3.3 重置画布大小

每次重置画布时，可以设置画布大小，使用 Pyqt5 自带的 QInputDialog 对话框可以获得所需参数，从而重新设置画布大小，显示效果如下：



4.3.4 鲁棒性

开发和测试的过程中，遇到诸多问题致使程序崩溃，这时需要在程序中全面地考虑所有可能遇到的情况。例如：

(1) 多边形、曲线没有按右键结束绘制就去执行其他操作，若不进行任何处理，该图元则会一直保持在画布上，且没有办法被选中。因此，在每个执行操作前，都需判断是否绘制完成，保存图元信息。

(2) 裁剪线段时，裁剪框未选上线段，等效于将整个线段删除，但是图元还在字典中保存，更新屏幕时依然会调用该图元的 paint() 函数，出现访问下标越界等问题，造成程序崩溃。需要将该图元的全部信息在画布、状态栏中删除，以免出现错误。

(3) 鼠标的坐标信息和绘制所需的控制点坐标存在 float 和 int 类型不对等的问题，因此在每次获取坐标时都需统一。

(4) 由于框架代码中由 currentTextChanged 触发改变选择函数，该函数在当前选择的内容发生改变时触发信号，因此有时状态栏选择或删除后，难以清除当前选择。将信号槽改为 itemClicked() 或 itemSelectionChanged() 可能是更好的选择，但该修改会导致代码框架发生较大改变，因此，此处选择简单的加入 clear 状态，等效实现清除选择。

5 总结

图形学大实验耗时一整个学期，实现了一个功能较全面的绘图系统，可以绘制多种图形并对其编辑。在课堂上，虽然学习了每种图形的绘制算法，但落实在代码上依然是一头雾水，通过查阅资料编写代码，我对这些算法有了更深刻的理解。此外，在框架代码上，使用 Pyqt5 实现图形界面很有挑战性，很好地锻炼了我阅读框架代码、查询官方 API、全面思考增强程序鲁棒性的能力，收获满满。

参考文献

[1] 孙正兴，周良，郑洪源等. 计算机图形学教程 [M]. 北京：机械工业出版社，2006.

[2] 计算机图形学——3 种直线绘制算法原理及代码实现
https://blog.csdn.net/qq_45720855/article/details/123763991

[3] 直线裁剪算法研究 (cohen-sutherland 算法和 liang-barsky 算法)
https://www.docin.com/p-2381905483.html

[4] Qt 图形视图框架：QGraphicsView 详解
https://blog.csdn.net/kenfan1647/article/details/117383803

[5] Qt QListWidget 详解
https://blog.csdn.net/wzz953200463/article/details/109648247