计算机图形学　12 月报告

夏宇　171180554

(南京大学 计算机科学与技术系, 南京　210093)

摘　要:　通过学习课本上所讲的图元生成算、图元变换以及曲线曲面的相关算法，并且利用 Qt Creator+ PyQt5 进行软件系统 GUI 的编写开发。手动实现课本中的相关画图算法，以及构建完整的图形界面系统，来实现作业所需要的画图、打开、保存以及图形显示的相关功能。通过将近一个学期的实验过程，全部完成了实验要求以及实验开发早期的所设定的相关拓展功能，命令行界面能够完成实验要求的基本功能，图形界面能够较好的完成了绘制图形的要求，用户交互体验良好。通过实验的进行，也将课堂上的算法以及自己课外补充到的内容运用到实验中来，加深了对图形学经典算法、思想的理解。

1　引言

计算机图形学大作业要求实现相关图元绘制算法，以及相关联的图元操作手段，并且要求能够：一、通过命令行读取命令文件绘制；二、通过图形界面获取鼠标信息绘制图片的两种方式进行输入操作。在作业要求的基础上，通过对一个基本的画图软件的需求进行分析，来确定所需要实现的功能以及采用的框架系统。并且，对已经实现和待实现的算法进行相关的解释，对已经实现和采用的系统软件框架做一个描述。根据我的对图形软件的需求分析和自己希望了解桌面应用开发的意向，我采用 QT Creator + PyQt5 进行系统框架的搭建。在实验的过程中，利用课程所学的图形学知识和自己补充的技术，完整地实现一整套图形绘制、编辑系统，包括以 DarkrainSp 命名的命令行程序以及 Darkrain 命名的图形化界面程序。

2　软件需求与功能分析

根据作业的要求以及软件的用途进行分析，来确定所需要的功能以及具体的实现方法：

2.1　基本绘图需求

图形学作业对与具体算法的实现有着具体的要求，基本要求的图元有 4 种：线段（DDA 和 Bresenham）、多边形（DDA 和 Bresenham）、椭圆（中间圆生成算法）、曲线（Bezier 和 B-spline）。还要求了基本的图形操作：平移、旋转、放缩。以及画布的设置、画笔颜色的设置及文件的保存。这些都通过实验说明的 PPT 有了具体的说明。所以这些操作需要在命令行和图形界面均需要实现。

2.2　算法及系统性能要求

对图形的绘制应该能够实时输出，所以需要绘图算法以及显示系统能够高效的进行相关操作。

2.3　扩展绘图需求

除了实验要求的功能之外，对于一个基本的绘图软件还应该有如下功能：

- 扩展图形的绘制，如矩形、三角形等。
- 对已绘制图元的基于 GUI 的可视化修改、编辑操作
- 类似铅笔的功能，在界面上拖动鼠标可以获得连续曲线
- 画笔的设置拓展，在设置颜色之外，还应该能够设置画笔的宽度

2.4 图形交互界面需求

为了方便用户的使用，对图形的交互界面有相关需求，首先需要能够便捷得进行用户所需要的操作。其次，还需要即时展示用户所需要的信息，以及扁平清爽的 UI 交互界面。

这方面可以参考比较成熟的画图软件，比如 Windows 中的“画图”软件，具体而言：

- 点击图形按钮，之后在画布上进行点击、拖动等操作实现图形绘制；点击图形旋转、平移按钮，可以通过点击图形来选中所需要的图元进行直接的操作；可以通过鼠标的点击进行画笔颜色、画笔宽度的选择；可以通过鼠标进行操作的撤销等等。
- 实时展示用户所绘制的图形，在进行图形绘制的时候，可以实时显示图形随鼠标所选中位置移动的变化；实时显示当前的画笔颜色、宽度；实时显示当前鼠标所在画布的坐标。
- 干净清爽的 UI 操作界面，控件的布局、颜色设置符合美观；对于较大的图形，能够设置滚动栏显示全部图形。

2.5 文件管理需求

实现文件的基本的打开、保存操作。考虑到：弹出窗口式的文件打开、保存操作；应该使画布尺寸随打开的图片变化。

综合以上五个方面的需求，我选择 PyQt5 以及 QT Creator 进行开发工作，通过 python 可以便于进行高层操作而无需过于关注底层内存管理系统的操作，能够提高开发的效率；使用 QT 可以方便得构建简介美观的图形界面，并且能方便的满足对颜色选择、文件操作的需求。

3 实验平台介绍

实验平台	Windows 10 Python 3.8
开发工具	PyQt 5.10.1
打包工具	PyInstaller 3.2.1
显示分辨率	1920x1080
缩放比例	150%

4 图元算法介绍

本实验是南京大学课程《计算机图形学》的一部分，深入掌握各种经典的图形学基本算法，以及通过大实验运用到实践当中取，是课程的主要目的，也是本次大实验的重点。课程中学习了基本图元绘制（包括线段、圆、椭圆）、图形的变换（平移、旋转、放缩）、图形的裁剪、曲线曲面，这当中有部分也在本次实验中加以实际运用。

下面就本门课程的相关算法分别进行介绍：

4.1 线段绘制算法

本门课程中，涉及到了 DDA 算法和 Bresenham 算法两种算法。下面分别介绍：

4.1.1 DDA 画线算法

计算机显示屏幕上是由一个个的像素组成的，而在数学中，一条直线是连续的不可分线条。所以，为了

显示出线条的轨迹线条，我们还需要决定屏幕上哪些像素应该被选中并且记录下来最终输出。因此，我们需要一个能够根据线段方程高效选择所需要输出的像素的算法，这便是 DDA 算法的作用。

DDA 算法是直线绘制的最简单算法。该算法主要是根据直线公式的斜率式 $y = kx + b$ 转化得到的。

对于一个已知的直线段，其有两个端点 $P_0(x_0,y_0)$和 $P_1(x_1,y_1)$，根据端点的信息，我们可以得到 k 和 b。在 k 和 b 已知的条件下，根据相应的 x 值，我们就能计算出一个 y 值。如果 x 每次递增 1，那么 y 的步进就为 k + b；同样知道一个 y 值也能计算出 x 值，此时 y 每次递增为 1，x 的步进为 (1-b)/k。根据计算出的 x 值和 y 值，向下取整，得到坐标(x', y')，并在(x', y')处绘制直线段上的一点。

为简化理论的计算，我们令 b 取 0，将起点看作(0,0)。设当前点为(x_i, y_i)，则用 DDA 算法求解$(x_{i+1}，y_{i+1})$的计算公式可以概括为：

$$xi + 1 = xi + xStep \tag{1}$$
$$yi + 1 = yi + yStep \tag{2}$$

我们一般通过计算 Δx 和 Δy 来确定 xStep 和 yStep：

- 如果 Δx > Δy ，说明 x 轴的最大差值大于 y 轴的最大差值，x 轴方向为步进的主方向，xStep = 1，yStep = k；
- 如果 Δy > Δx，说明 y 轴的最大差值大于 x 轴的最大差值，y 轴方向为步进的主方向，yStep = 1，xStep = 1 / k。

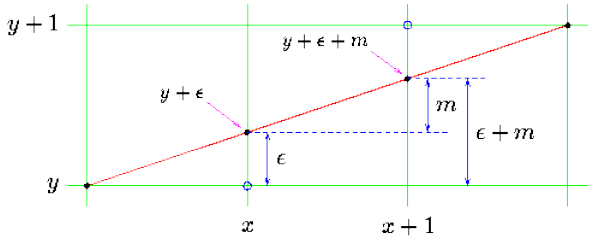
根据这个公式，就能通过(x_i, y_i)迭代计算出(x_{i+1}, y_{i+1})，然后在坐标系中绘制计算出的(x, y)坐标。

在学习的过程中我们发现，DDA 算法的形式以及内容不难理解，但是 DDA 算法同样有着严重的缺陷，DDA 需要进行浮点数计算，算法开销大，所以我们能否去使用计算更为方便的算法呢，带着个这个问题，我又学习了下面的 Bresenham 画线算法的实现。

4.1.2 Bresenham 画线算法

Bresenham 算法的基本思路与 DDA 算法类似，都是利用直线的显示性质来进行选点画线操作，Bresenham 算法的重点是如何通过利用直线方程的性质算法形式的优化。

由于显示直线的象素点只能取整数值坐标，可以假设直线上第 i 个象素点坐标为(xi，yi)，它是直线上点(xi，yi)的最佳近似，并且 xi=xi(假设 m<1)，如下图所示。那么，直线上下一个象素点的可能位置是(xi+1，yi)或(xi+1，yi+1)。



由图中可以知道，在 x=xi+1 处，直线上点的 y 值是 y=m(xi+1)+b，即：

$$-yi = m(xi + 1) + b \tag{3}$$

该点离象素点(xi+1，yi)和象素点(xi+1，yi+1)的距离分别是 d1 和 d2：

$$d1 = y - yi = m(xi + 1) + b - yi \tag{4}$$
$$\mathrm{d2} = (yi + 1) - \mathrm{y} = (yi + 1) - \mathrm{m}(xi + 1) - \mathrm{b} \tag{5}$$

这两个距离差是

$$d1 - d2 = 2m(xi + 1) - 2yi + 2b - 1 \tag{6}$$

我们来分析公式(7)：

(1)当此值为正时，d1>d2，说明直线上理论点离(xi+1，yi+1)象素较近，下一个象素点应取(xi+1，yi+1)。

(2)当此值为负时，d1<d2，说明直线上理论点离(xi+1，yi)象素较近，则下一个象素点应取(xi+1，yi)。

(3)当此值为零时，说明直线上理论点离上、下两个象素点的距离相等，取哪个点都行，假设算法规定这种情况下取(xi+1，yi+1)作为下一个象素点。

因此只要利用(d1-d2)的符号就可以决定下一个象素点的选择。为此，我们进一步定义一个新的判别式：

$$Pi = \triangle x \times (d1 - d2) = 2 \triangle y \cdot xi - 2 \triangle x \cdot yi + c \tag{7}$$

式(4)中的△x=(x2-x1)>0，因此 pi 与(d1-d2)有相同的符号；这里△y=y2-y1，m=△y/△x；c=2△y+△x(2b-1)。

下面对式(4)作进一步处理，以便得出误差判别递推公式并消除常数 c。

将式(4)中的下标 i 改写成 i+1，得到：

$$pi + 1 = 2 \triangle y \cdot xi + 1 - 2 \triangle x \cdot yi + 1 + c \tag{8}$$

将式(5)减去(4)，并利用 xi+1=xi+1，可得：

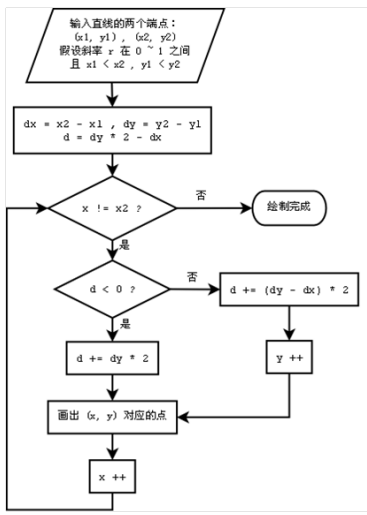
$$pi + 1 = pi + 2 \triangle y - 2 \triangle x \cdot (yi + 1 - yi) \tag{9}$$

再假设直线的初始端点恰好是其象素点的坐标，即满足：

$$y1 = mx1 + b \tag{10}$$

由式(4)和式(7)得到 p1 的初始值：

$$\mathrm{p1} = 2 \triangle y - \triangle x \tag{11}$$



Bresenham 流程图

这样，我们可利用误差判别变量，第 i+1 步的判别变量 pi+1 仅与第 i 步的判别变量 pi、直线的两个端点坐标分量差△x 和△y 有关，运算中只含有整数相加和乘 2 运算，而乘 2 可利用算术左移一位来完成，因此这个算法速度快并易于硬件实现。

通过对线段图元生成算法的学习，尤其是 Bresenham 算法对计算上的优化后，我逐渐加深了对课程知识的理解，了解了图元生成算法的不断优化之原理和意义所在。

4.2 圆、椭圆绘制算法

在数学中，圆的方程可以描述为 $(x-x_0)^2 + (y-y_0)^2 = R^2$，其中(x0, y0)是圆心坐标，R 是圆的半径，特别的，当(x_0, y_0)为坐标中心点时，圆方程可以简化为 $x^2 + y^2 = R^2$。在计算机图形学中，圆和直线一样，也

存在着在点阵输出设备上显示或输出的问题，因此也需要一套光栅扫描转换算法。为了简化，我们先考虑圆心在原点的圆的生成，对于中心不是原点的圆，可以通过坐标的平移变换获得相应位置的圆。下面我就分别来介绍中心圆算法、Bresenham 算法以及 Bresenham 椭圆算法。

4.2.1 中心圆算法

构造判别函数：

$$F(x,y) = x^2 + y^2 - R^2 \tag{12}$$

当 F(x, y）= 0，表示点在圆上，当 F(x, y）> 0，表示点在圆外，当 F(x, y）< 0，表示点在圆内。如果 M 是 P_1 和 P_2 的中点，则 M 的坐标是（$x_i + 1, y_i - 0.5$），当 F（$x_i + 1, y_i - 0.5$）< 0 时，M 点在圆内，说明 P_1 点离实际圆弧更近，应该取 P_1 作为圆的下一个点。同理分析，当 F（$x_i + 1, y_i - 0.5$）> 0 时，P_2 离实际圆弧更近，应取 P_2 作为下一个点。当 F（$x_i + 1, y_i - 0.5$）= 0 时，P_1 和 P_2 都可以作为圆的下一个点，算法约定取 P_2 作为下一个点。

现在将 M 点坐标（$x_i + 1, y_i - 0.5$）带入判别函数 F(x, y），得到判别式 d：

$$d = F(x_i + 1, y_i - 0.5) = (x_i + 1)^2 + (y_i - 0.5)^2 - R^2 \tag{13}$$

若 d < 0，则取 P_1 为下一个点，此时 P_1 的下一个点的判别式为：

$$d = F(x_i + 2, y_i - 0.5) = (x_i + 2)^2 + (y_i - 0.5)^2 - R^2 \tag{14}$$

展开后将 d 带入可得到判别式的递推关系：

$$d' = d + 2x_i + 3 \tag{15}$$

若 d > 0，则取 P_2 为下一个点，此时 P_2 的下一个点的判别式为：

$$d' = F(x_i + 2, y_i - 1.5) = (x_i + 2)^2 + (y_i - 1.5)^2 - R^2 \tag{16}$$

展开后将 d 带入可得到判别式的递推关系：

$$d' = d + 2(x_i - y_i) + 5 \tag{17}$$

特别的，在第一个象限的第一个点（0, R）时，可以推倒出判别式 d 的初始值 d_0：

$$d_0 = F(1, R - 0.5) = 1 + (R - 0.5)^2 - R^2 = 1.25 - R \tag{18}$$

根据上面的分析，可以写出中点画圆法的算法。观察中心原算法的整体过程我们可以发现，中心圆算法很好地利用了圆的基本性质，但是其存在着较多的浮点运算，使得画圆算法的开销较大。

4.2.2 Bresenham 画圆算法

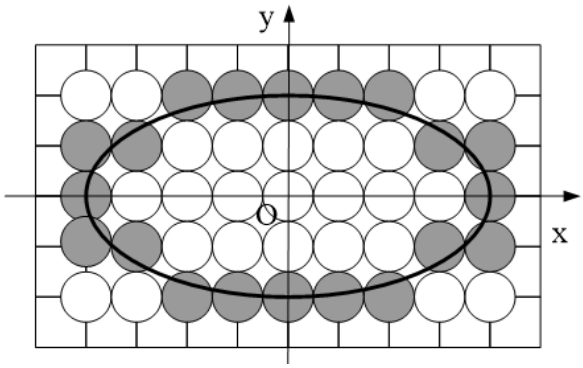
Bresenham 算法的主要思想继承于中心圆算法，仍然是利用判别式来确定点的位置，在 DDA 算法中，我们利用判别式(12)来计算下个点的位置，而计算判别式使用了浮点运算，影响了圆的生成效率。如果能将判别式规约到整数运算，则可以简化计算，提高效率。于是针对中点画圆法进行了多种改进，其中一种方式是将 d

的初始值由 1.25 – R 改成 1 – R，考虑到圆的半径 R 总是大于 2，因此这个修改不会响 d 的初始值的符号，同时可以避免浮点运算。还有一种方法是将 d 的计算放大两倍，同时将初始值改成 3 – 2R，这样避免了浮点运算，乘二运算也可以用移位快速代替，采用 3 – 2R 为初始值的改进算法。

4.2.3 Bresenham 画椭圆算法

利用我们之前提到的 Bresenham 画圆算法，我们可以轻松地得到 Bresenham 画椭圆算法，不过需要注意的是，椭圆中我们需要利用的是椭圆的四分对称性，与圆的有所不同。

椭圆的扫描转换是在屏幕像素点阵中选取最佳逼近于理想椭圆像素点集的过程。椭圆是长半轴和短半轴不相等的圆，椭圆的扫描转换与圆的扫描转换有类似之处。本节主要讲解顺时针绘制 1/4 椭圆的中点 Bresenham 算法原理，根据对称性可以绘制完整椭圆。

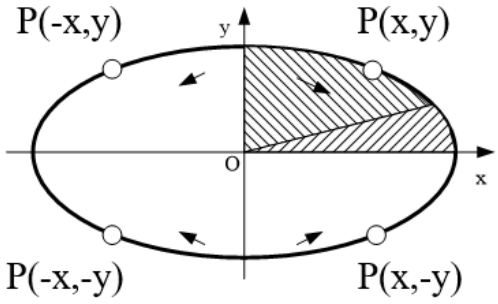


默认的椭圆是圆心位于坐标系原点，长半轴为 a、短半轴为 b 的椭圆。需要进行圆心平移或使用自定义坐标系可以绘制椭圆。

圆心在原点、长半轴为 a、短半轴为 b 的椭圆方程的隐函数表达式为 ：

$$F(x,y) = b2x2 + a2y2 - a2b2 = 0F(x,y) = b2x2 + a2y2 - a2b2 = 0 \tag{19}$$

椭圆将平面划分成三个区域：对于椭圆上的点，F(x，y)＝0；对于椭圆外的点，F（x，y）＞0；对于椭圆内的点，F（x，y）＜0。

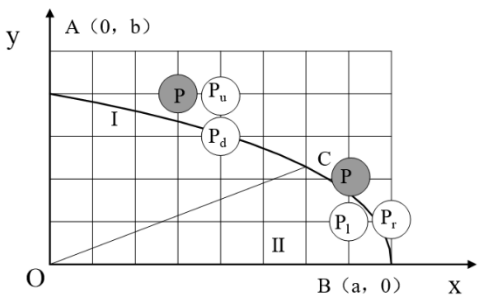


考虑到椭圆的对称性，可以用对称轴 x＝0，y＝0，把椭圆分成 4 等份。只要绘制出第一象限内的 1/4 椭圆弧，根据对称性就可绘制出整个椭圆，这称为四分法绘制椭圆算法。已知第一象限内的点 P(x,y)，可以顺时针得到另外 3 个对称点：P(x, -y)，P(-x, -y)和 P(-x, y)。

进而我们计算椭圆在点 P(x, y)处的法矢量：

$$N(x,y)=\frac{\partial F}{\partial x}i+\frac{\partial F}{\partial y}j=2b^2x_i+2a^2y_j \tag{20}$$

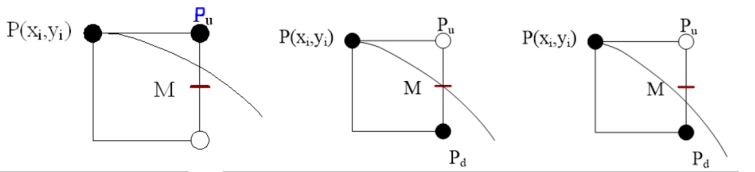
部分Ⅰ的 AC 椭圆弧段，法矢量的 x 向分量小于 y 向分量，斜率 k 处处满足|k|＜1，$|\Delta x|>|\Delta y|$，所以 x 方向为主位移方向；在 C 点，法矢量的 x 向分量等于 y 向分量，斜率 k 满足 k＝－1，$|\Delta x|=|\Delta y|$；在部分Ⅱ的 CB 椭圆弧段，法矢量的 x 向分量大于 y 向分量，斜率 k 处处满足|k|＞1，$|\Delta y|>|\Delta x|$，所以 y 方向为主位移方向。在部分Ⅰ椭圆的中点 Bresenham 的原理:每次在主位移 x 方向上走一步，y 方向上退不退步取决于中点误差项的值。在部分Ⅱ：每次在主位移方向 y 上退一步，x 方向上是否进步取决于中点误差项的值。



我们首先来构造上半部分Ⅰ的中点误差项。在上半部分Ⅰ，x 方向每次加 1，y 方向上减不减 1 取决于中点误差项的值。

从 $P_i(x_i, y_i)$ 点出发选取下一像素时，需将 $P_u(x_i + 1, y_i)$ 和 $P_d(x_i + 1, y_i - 1)$ 的中点 $M(x_i +1, y_i - 0.5)$代入隐函数，构造中点误差项：

$$d_{1i}=F(x_i+1,y_i-0.5)=b^2(x_i+1)^2+a^2(y_i-0.5)^2-a^2b^2 \tag{21}$$



当 d1i<0d1i<0 时，中点 M 在椭圆内，下一像素点应选取 P_u，即 y 方向上不退步；当 $d_{1i}>0$ 时，中点 M 在椭圆外，下一像素点应选取 P_d，即 y 方向上退一步；当 $d_{1i}=0$ 时，中点 M 在椭圆上，P_u、P_d 和椭圆的距离相等，选取 P_u 或 P_d 均可，约定取 P_d。

因此，

$$y_{i+1}=\begin{cases} y_i, & d_{li}<0 \\ y_i-1, & d_{li}\geq 0 \end{cases} \tag{22}$$

下面我们来求上半部分Ⅰ的递推公式：

如果考虑主位移方向再走一步，应该选择哪个中点代入中点误差项以决定下一步应该选取的像素，分两种情况讨论。

1) 当 d1i<0 时，下一步的中点坐标为 M(xi+2，yi−0.5):

$$\begin{aligned} d_{1(i+1)} &= F(x_i+2,y_i-0.5) \\ &= b^2(x_i+2)^2+a^2(y_i-0.5)^2-a^2b^2 \end{aligned}$$

$$= b^2(x_i + 1)^2 + a^2(y_i - 0.5)^2 - a^2b^2 + b^2(2x_i + 3)$$
$$= d_{1i}b^2 + b^2(2x_i + 3) \quad (23)$$

2) 当 d1i≥0 时，下一步的中点坐标为 M(xi+2，yi−1.5):

$$d_{1(i+1)} = F(x_i + 2, y_i - 1.5)$$
$$= b^2(x_i + 2)^2 + a^2(y_i - 1.5)^2 - a^2b^2$$
$$= b^2(x_i + 1)^2 + a^2(y_i - 0.5)^2 - a^2b^2 + b^2(2x_i + 3) + a^2(2y_i + 3)$$
$$= d_{1i}b^2 + b^2(2x_i + 3) + a^2(2y_i + 3) \quad (24)$$

接下来就可以计算中点误差项的初始值:

上半部分椭圆的起点为$A(0, b)$，因此，第一个中点是$(1, b - 0.5)$，对应的 d_{1i} 的初值为

$$d_{10} = F(1, b - 0.5) = b^2 + a^2(b - 1.5)^2 - a^2b^2 \quad (25)$$

下面我们来构造下半部分 II 的中点误差项，主要过程与上半部分的类似：在下半部分 II，主位移方向发生变化，中点 Bresenham 算法原理为：y 方向上每次减 1，x 方向上加 1 不加 1 取决于中点误差项的值。从上半部分 I 的终止点 $P_i(x_i, y_i)$出发选取下一像素时，需将 $P_l(x_i, y_i - 1)$和 $P_r(x_i + 1, y_i - 1)$的中点 $M(x_i + 0.5, y_i - 1)$代入隐函数，构造中点误差项

$$d_{2i} = F(x_i + 0.5, y_i - 1) = b^2(x_i + 0.5)^2 + a^2(y_i - 1)^2 - a^2b^2 \quad (26)$$

下半部分 II 的递推公式推导过程也与上半部分的类似：

1) 当 $d_{2i}<0$ 时，下一步的中点坐标为 $M(x_i + 1.5, y_i - 2)$下一步中点误差项为:

$$d_{2(i+1)} = F(x_i + 1.5, y_i - 2)$$
$$= b^2(x_i + 1.5)^2 + a^2(y_i - 2)^2 - a^2b^2$$
$$= b^2(x_i + 0.5)^2 + a^2(y_i - 1)^2 - a^2b^2 + b^2(2x_i + 2) + a^2(-2y_i + 3)$$
$$= d_{2i}b^2 + a^2(-2y_i + 3) + a^2(-2y_i + 3) \quad (27)$$

2) 当 $d_{2i}≥0$ 时，下一步的中点坐标为 $M(x_i + 0.5, y_i - 2)$，下一步中点误差项为:

$$d_{2(i+1)} = F(x_i + 0.5, y_i - 2)$$
$$= b^2(x_i + 0.5)^2 + a^2(y_i - 2)^2 - a^2b^2$$
$$= b^2(x_i + 0.5)^2 + a^2(y_i - 1)^2 - a^2b^2 + a^2(-2y_i + 3)$$
$$= d_{2i}b^2 + a^2(-2y_i + 3) \quad (28)$$

下面求下半部分的中点误差项的初始值：

在上半部分 I，法矢量的 x 向分量小于 y 向分量；在 C 点，法矢量的 x 向分量等于 y 向分量；在下半部分 II，法矢量的 x 向分量大于 y 向分量。则对于上半部分椭圆上一点 $P_i(x_i, y_i)$，如果其当前中点 $M(x_i + 0.5, y_i - 1)$，满足 x 向分量小于 y 向分量

$$b^2(x_i + 1)^2 < a^2(y_i - 0.5)^2 \quad (29)$$

而在下一个中点，不等号改变方向，则说明椭圆从上半部分Ⅰ转入了下半部分Ⅱ。
假定 $P_i(x_i, y_i)$点是椭圆上半部分Ⅰ的最后一个像素，$M(x_i+0.5, y_i-1)$是用于判断选取 P_u 和 P_d 像素的中点。由于下一像素转入了下半部分Ⅱ，其中点改为判断 P_l 和 P_r 的中点 $M_{II}(x_i+0.5, y_i-1)$，所以下半部分的初值 d_{20} 为

$$d_{20} = F(x + 0.5, y - 1) = b^2(x + 0.5)^2 + a^2(y - 1)^2 - a^2b^2 \tag{30}$$

至此，就完全得到了 Bresenham 椭圆算法的全过程，虽然过程比较繁琐，但是在理解了其整体过程后，能感受到，整体过程的合理性。在实际的操作中，我就采用的这种算法的优化版本，实际得出实验效果较好。

4.3 图元填充算法

填充图元是标准输出图元的一种，它通常是多边形填充图元。多边形有两种表示方法：一是顶点表示；另一种是点阵表示。因此，光栅系统中有两种基本的区域填充方法：一种是针对采用顶点表示的多边形区域，通过确定横越区域的扫描线的覆盖间隔来填充的多边形扫描转换区域填充方法，这种方法在一般的图形软件包中主要用来填充多边形、圆、椭圆和其它简单曲线；另一种是针对采用点阵表示的多边形区域，从给定的位置开始涂描着色直到达到指定的边界条件为止的区域填充方法，这种方法主要用在具有复杂形状边界的情况及交互式涂描系统中。
虽然在本系统中我没有考虑实现填充功能，但作为课程的主要内置，我还是就这两种算法分别做简单介绍。

4.3.1 扫描转换区域填充方法

主要方法为用水平扫描线从上到下扫描由多条首尾相连的线段构成的多边形，每根扫描线与多边形的某些边产生一系列交点。将这些交点按照坐标排序，将排序后的点两两成对，作为线段的两个端点，以所填的颜色画水平直线。
具体方法：

1) 初始化扫描线，然后更新活化边表，第一次更新的为最小 y 值为扫描线值的两条边；
2) 只要活化边表不为空时，执行下面的循环：
   a) 以活化边表中相邻两个边的 x 为填充范围的横坐标，以扫描线值为 y 坐标，填充这一个区域（其实就是画这根线）
   b) 填充完后，扫描线值加 1
   c) 根据扫描线值移除移已经处理完的活化边（扫描线值大于该边最大 y 值），将活化边表中相邻两个边的 x (之前用于确定填充横坐标区间的值）更新，也即加上各自斜率的倒数
   d) 更新一下活化边表，根据活化边表中每条边的 x 来进行排序
3) 循环直至边表为空，为了防止对多边形编辑后填充不正确，每次编辑完后，都要重新校正边表的值。

4.3.2 区域填充算法

区域填充是从区域的一个内点开始，赋予指定的颜色，然后将该颜色扩展到整个区域，它是对区域重新着色的过程。区域填充算法要求区域是连通的，只有在连通的区域内，才有可能将种子点的颜色扩展到区域内的其它点。在区域填充中，最常用的是四连通和八连通区域。值得注意的是，在采用 4 连通和 8 连通方式定义区域时，内部像素是 4 连通时，其边界的像素只要是 8 连通就可以了；而内部像素为 8 连通时，其边界的像素必须是 4 连通的，否则，就无法正确填充为 8 连通区域。

区域填充是从区域的一个内点开始，赋予指定的颜色，然后将该颜色扩展到整个区域。因此，必须要对区域内外点进行测试。用来鉴别物体内部种子点的方法主要有两种：奇偶规则或非零环绕数规则。对于标准

多边形和其它简单形状，非零环绕数规则和奇偶规则给出相同的结果；但对于复杂形状，两种方法会产生不同的内部和外部区域。

对边界表示的区域进行的区域填充称为边界填充；边界填充算法是从区域的一个内点开始，然后由内向外绘点直到边界。假如边界是以单一颜色指定的，则填充算法可逐个像素地向外处理，直到遇到边界颜色。下面以递归边界填充做简单介绍。边界填充程序接受输入内部点(x, y)的坐标、填充颜色和边界颜色，从(x, y)开始，程序按照连通性定义或要求检测相邻位置以确定它们是否是边界颜色，倘若不是，就用填充颜色涂色，并检测其相邻位置。这个过程延续到已经检测完区域边界颜色范围内的所有像素为止。

而后，对内点表示的区域进行的区域填充。

4.4 图元变换算法

目前，我们学习的是图形拓扑关系不变的几何变换，由于图形变换采用了齐次坐标表示，我们可以方便的用变换矩阵实现对图形的变换。课程与实验涉及到的变换算法有平移、旋转、放缩等。

4.4.1 图元的平移

平移变换不改变图形的大小和形状，物体上的每个点移动相同的偏移量，是一种不产生变形而移动物体的刚体变换。直线的平移是将平移向量施加到直线的每个端点；多边形的平移是将平移向量施加到多边形的每个顶点；曲线可用同样方法来平移：为了改变圆或椭圆的位置，可以平移其中心坐标并在新中心位置重画图形。

平移变化可表示为：

$$\begin{cases} x_2 = x_1 + t_x \\ y_2 = y_1 + t_y \end{cases} \tag{31}$$

其齐次形式为：

$$[x_2 \quad y_2 \quad 1] = [x_1 \quad y_1 \quad 1]\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix} \tag{32}$$

4.4.2 图元的旋转

旋转变换仍保持图形各部分间的线性关系和角度关系，变换后直线的长度不变，是一种不变形地移动物体的刚体变换，物体上的所有点旋转相同的角度。直线段旋转是将直线段的每个端点旋转指定的角度；多边形的旋转则是将每个顶点旋转指定的旋转角；曲线的旋转则是旋转控制取样点。

旋转变化可表示为：

$$\begin{cases} x_2 = x_1 \cos\theta - y_1 \sin\theta \\ y_2 = x_1 \sin\theta + y_1 \cos\theta \end{cases} \tag{33}$$

其齐次形式为：

$$[x_2 \quad y_2 \quad 1] = [x_1 \quad y_1 \quad 1]\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{34}$$

4.4.3 图元的放缩

二维比例变换改变物体的尺寸,多边形的缩放可将变换应用于每个顶点。而对以参数方程形式定义的其它物体的变换则只需对方程的参数进行缩放。

平移变化可表示为：

$$\begin{cases} x_2 = x_1 \cdot S_x \\ y_2 = y_1 \cdot S_y \end{cases} \tag{35}$$

其齐次形式为：

$$[x_2 \quad y_2 \quad 1] = [x_1 \quad y_1 \quad 1]\begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{36}$$

4.5 图元剪裁算法

课程和实验中，剪裁算法也是重要的一部分。识别图形在指定区域内、外部分的过程称为裁剪，用来裁剪对象的区域称为裁剪窗口。一般而言，当指定裁剪窗口后，总是希望把落在窗口内的部分图形显示在给定的视区中，而把窗口外的图形采用相应方法裁剪掉，不予显示。剪裁的对象有点、线段、多边形等等。

4.5.1 点的裁剪

点的剪裁是图元剪裁技术的基础，所以在开始介绍其他剪裁技术前，我首先介绍点的建材算法。设裁剪窗口是矩形，如果点 P(x, y) 满足下列不等式，则点 P 显示，否则点 P 不显示。

$$\begin{cases} x_{\text{wmin}} \le x \le x_{\text{wmax}} \\ x_{\text{wmin}} \le \text{y} \le \text{y}_{\text{wmax}} \end{cases} \tag{37}$$

其中：x_{wmin} $x_{\text{wm}ax}$ x_{wmin} y_{wmax} 是是视窗边界。

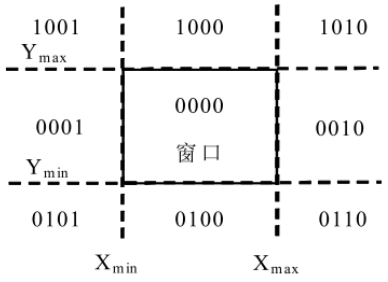
4.5.2 线段的裁剪

在认识了点的裁剪之后，我们又在课堂上学习了线段的裁剪。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。下面来介绍几种课堂上的重要算法。

4.5.2.1 Cohen-Sutherland 算法

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。

算法采用的编码方式是按照下图的九个区域划分



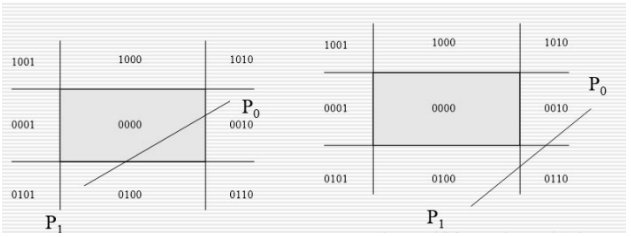
四位区域码中各位从左到右依次表示 上、下、右、左。区域码的任何位赋值为 1 代表端点落在相应的区域中，否则该位为 0。若端点在裁剪窗口内，区域码为 0000。

剪裁步骤为：

1) 若直线段的两个端点的区域编码都为 0，有 RC0|RC1=0（二者按位相或的结果为零，即 RC0=0 且 RC1=0），说明直线段的两个端点都在窗口内，应“简取”之。
2) 若直线段的两个端点的区域编码都不为 0，且 RC0&RC1≠0（二者按位相与的结果不为零，即 RC0≠0 且

RC1≠0），说明直线段位于窗外的同一侧，或左方、或右方、或上方、或下方，应“简弃”之。

3) 若直线段既不满足“简取”也不满足“简弃”的条件，则需要与窗口进行“求交”判断。这时，直线段必然与窗口边界或窗口边界的延长线相交，分两种情况处理。



4.5.2.2 梁友栋-Barsky 算法

Cyrus 和 Beck 用参数化方法提出了比 Cohen-Sutherland 更有效的算法。后来梁友栋和 Barsky 独立地提出了更快的参数化线段裁剪算法，也称为 Liany-Barsky（LB）算法。算法使用直线的参数方程和不等式组来描述线段和裁剪窗口的交集。求解出的交集将被用于获知线的哪些部分是应当绘制在屏幕上的。

算法的具体过程如下：

1) 初始化线段交点的参数：$u_1=0$，$u_2=1$；
2) 计算出各个裁剪边界的 p、q 值；
3) 根据 p、q 来判断：是舍弃线段还是改变交点的参数。
   a) 当 p<0 时，参数 r 用于更新 u_1；（$u_1=\max\{u_1, \ldots, r_k\}$）
   b) 当 p>0 时，参数 r 用于更新 u_2。（$u_2=\min\{u_2, \ldots, r_k\}$）
   c) 如果更新了 u_1 或 u_2 后，使 $u_1>u_2$，则舍弃该线段。
   d) 当 p=0 且 q<0 时，因为线段平行于边界并且位于边界之外，则舍弃该线段。见下图所示。
4) p、q 的四个值经判断后，如果该线段未被舍弃，则裁剪线段的端点坐标由参数 u_1 和 u_2 的值决定。

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。

学习后，我的主要感觉是，梁友栋-Barsky 算法的主要思想是利用直线的参数表示，从而将二维的问题一维化，并且利用相关的性质，达到简化算法的作用。

4.5.3 多边形的裁剪

多边形作为实区域考虑时，封闭的多边形裁剪后仍应是封闭的多边形，以便于填充。因此在多边形裁剪时必须考虑多边形线段间的关系。许多算法对多边形的裁剪是基于矩形裁剪窗口的，如果需要用任意形状的多边形对线段裁剪，需对原有算法进行扩充。在这里，我简单介绍几种对于凸多边形的剪裁算法，凹多边形可以通过转化为凸多边形来实现。

4.5.3.1 Sutherland-Hodgman 算法

该算法每次用裁剪窗口的一条边界对要裁剪的多边形进行裁剪，算法将多边形裁剪分解为多边形关于裁剪窗口每条边界所在直线的裁剪。它通过依次按照裁剪矩形的各边界来处理所有多边形的顶点来实现。

算法的每一步考虑以窗口的一条边以及延长线构成裁剪线。该线将平面划分成两分：一部分包含裁剪窗口，称为可见一侧；另一部分称为不可见一侧。根据多边形每一边与窗口边所形成的位置关系，输出 0 个、1 个或 2 个顶点到结果多边形顶点表中，这些顶点系列构成了裁剪后的多边形。裁剪得到的结果多边形的顶点

由两部分组成：一部分为落在可见一侧的原多边形顶点；另一部分为多边形的边与裁剪窗口边界的交点。然后将这两部分顶点按一定顺序连接起来，得到裁剪结果多边形，作为下一条裁剪窗口边界处理过程的输入。多边形的裁剪可以认为是构成多边形的线段的裁减。而线段的裁剪可通过检测线段与裁剪窗口边界的位置关系决定是否输出该线段的顶点。线段与窗口边界的位置关系主要包括四种：① 如果第一点在窗口边界外侧，而第二点在窗口边界内侧，则多边形的该边与窗口边界的交点和第二点都被加到输出顶点表中；② 如果两顶点都在窗口边界内侧，则只有第二点加入输出顶点表中；③ 如果第一点在窗口边界内侧而第二点在外侧， 则只有与窗口边界的交点加到输出顶点表中；④ 如果两个点都在窗口边界外侧，输出表中不增加任何点。

由于该算法裁剪所得的结果应是，也只能是一个多边形，即：只有一个输出顶点表，表中最后一个顶点总是连着第一个顶点，因此当裁剪后的多边形有两个或者多个分离部分时会出现多余的线。

4.5.3.2 Weiler-Atherton 算法

为了适应剪裁凹多边形的需求，根据多边形处理方向(顺时针/逆时针)和当前处理的多边形顶点对是由外到内，还是由内到外来确定裁剪后多边形的顶点连接方式：沿着多边形边界方向连接；还是沿着窗口边界方向连接，来确定结果的正确性。

本算法的基本实现步骤包括：

1) 建立多边形和裁剪窗口的顶点表；
2) 求出多边形与窗口的交点，并将这些交点按顺序插入两多边形的顶点表中。在两多边形顶点表中的相交点间建立双向指针。
3) 建立空的裁剪结果多边形的顶点表。
4) 选取任一没有被跟踪过的交点为起点，将其输出到结果多边形顶点表中：如果该交点为进点，跟踪多边形边界；否则，跟踪多边形边界（顶点表）。
5) 跟踪多边形边界，每遇到多边形顶点，将其输出到结果多边形顶点表中，直至遇到新的交点。
6) 将该交点输出到结果多边形顶点表中，并通过连接该交点的双向指针改变跟踪方向：如果上一步跟踪的是多边形边界，改为跟踪窗口边界；反之，改为跟踪窗口边界)。
7) 重复 5)、6)，直到回到起点。

但是，需要注意的是，当多边形顶点和边与窗口边界重合时，交点计算须视情况而定，处理方法如下：与裁剪窗口边界重合的多边形的边不参与求交点。对于顶点落在裁剪窗口边界上的多边形边：如果它落在该裁剪边的内侧，将该顶点算作交点；否则，不将该顶点算作交点。

观察算法我们发现本算法在逻辑上会比先前的算法复杂很多，但是其更大范围的可用性确实十分有利。

4.6 曲线绘制算法

参数是指曲线方程中使用的自变量，当它在某个范围内改变时，对应坐标点在曲线上移动。“参数曲线”是指用参数作为自变量的函数曲线，有时使用参数曲线可简化矢量表示形式。本课程和实验中，参数样条是我们关注的主要内容，样条曲线的相关内容比较丰富，由于版面问题，下面就两种重要的样条曲线做介绍。

4.6.1 Bezier 曲线

n 次 Bernstein 基函数的多项式形式为：

$$\mathrm{BEZ}_{i,n}(u) = C(n,i)\mathrm{u}^{i}1\text{-}\mathrm{u}^{n-i} \tag{38}$$

Bezier 曲线的基本方程很简单，假设给出 n+1 个控制顶点位置，这些坐标点混合产生下列位置向量 *P(u)*：

$$P(u)=\sum_{i=0}^{n} P_i\, BEZ_{i,n}(u) \quad (0<u<1) \tag{39}$$

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。
其公式为：

$$P_i^r=\begin{cases} P_i, & (r=0) \\ (1-r)P_i^{r-1}+uP_{i+1}^{r-1}, & (r=1\ldots n) \end{cases} \tag{40}$$

4.6.2 B 样条

以 B 样条基函数代替 Bernstein 基函数，从而改进了 Bézier 特征多边形与 Bernstein 多项式次数有关，且是整体逼近的弱点。
B 样条基函数为：

$$B_{i,k}(u)=\left[\frac{u-u_i}{u_{i+k-1}-u_i}\right]B_{i,k-1}(u)+\left[\frac{u_{i+k}-u}{u_{i+k-1}-u_i}\right]B_{i+1,k-1}(u) \tag{41}$$

由此可知：

1) $B_{i,0}(u)$是一个阶梯函数，它在半开区间 $u\in[u_i,u_{i+1})$外都为零；
2) 当 p>0 时，$B_{i,k}(u)$是两个 k-1 次基函数的线性组合；
3) 计算一组基函数时需要事先制定节点矢量 U 和次数 k；
4) 定义式中可能出现 0/0，我们规定 0/0=0；
5) $B_{i,k}(u)$是定义在整个实数轴上的分段多项式函数，但我们一般只对它在区间$[u_0,u_m]$上的部分感兴趣；
6) 半开区间$[u_i,u_{i+1})$称为第 i 个节点区间（knot span），它的长度可以为零，因为相邻节点可以是相同的.

B 样条的定义为：

$$P(u)=\sum_{i=0}^{n} P_i\, B_{i,k}(u) \quad (u_{k-1}<u<u_{n+1}) \tag{39}$$

B 样条曲线有很多与贝塞尔曲线一样的重要性质，因为前者是后者的推广。而且，B-样条曲线有比贝塞尔曲线更有利于实际应用的性质。本实验中，就利用了三阶 B 样条的来进行曲线的输出。

5 程序结构、功能及代码

根据实验的要求，即命令行和 UI 界面的双重界面。所以，本实验的代码结果一共由两个项目组成分别是 DarkrainSp 和 Darkrain。DarkrainSp 用于读取在命令行下的命令并操作，所有的功能要求都按照具体的实验要求来进行；Darkrain 则是本项目的主要版本，具有 UI 界面和拓展功能。DarkrainSp 实际上是 Darkrain 的简化版本，所以接下来的介绍主要内容会以 Darkrain 为中心，关于 DarkrainSp 只会介绍其代码框架及运行主要流程。

下面我将会展示代码的主要框架和运行主要流程，而后我会 Darkrain 的相关核心代码以及图元算法进行展示。

5.1 整体代码框架

5.1.1 Darkrain 代码框架

下面为 Darkrain 的代码主要框架

```
Darkrain
│   main.py          //程序入口
│   main.spec        //打包工具 Pyinstaller 配置文件
├──icons            //程序图标储存位置
│       clip.png
│       colors.png
│       curve.png
│       ellipse.png
│       line.png
│       move.png
│       pencil2.png
│       polygon.png
│       rect.png
│       rotate.png
│       scale.png
│       size.png
│       touch.png
│       trash.png
│       triangle.png
│       width.png
│       window.ico
│       window.png
│
├──src   //源代码主目录
│   │   addpolygon.py    //矩形、三角形
│   │   config.py        //设置和工具函数
│   │   curve.py         //曲线图元
│   │   ellipse.py       //椭圆图元
│   │   line.py          //线段图元
│   │   mainwindow.py    //主窗口
│   │   myQLabel.py      //主画板
│   │   pencil.py        //铅笔图元
│   │   polygon.py       //多边形图元
│   │   primitive.py     //图元类
│   └──toolDialog.py     //工具窗口
│
└──ui   //UI 界面文件
    │   mainwindowUI.py
    │   mainwindowUI.ui      //Qt designer 文件
    └──style.qss       //界面 QSS
```

5.1.2 DarkrainSp 代码框架

DarkrainSp 的代码框架和 Darkrain/src 下的类似，是简化的版本

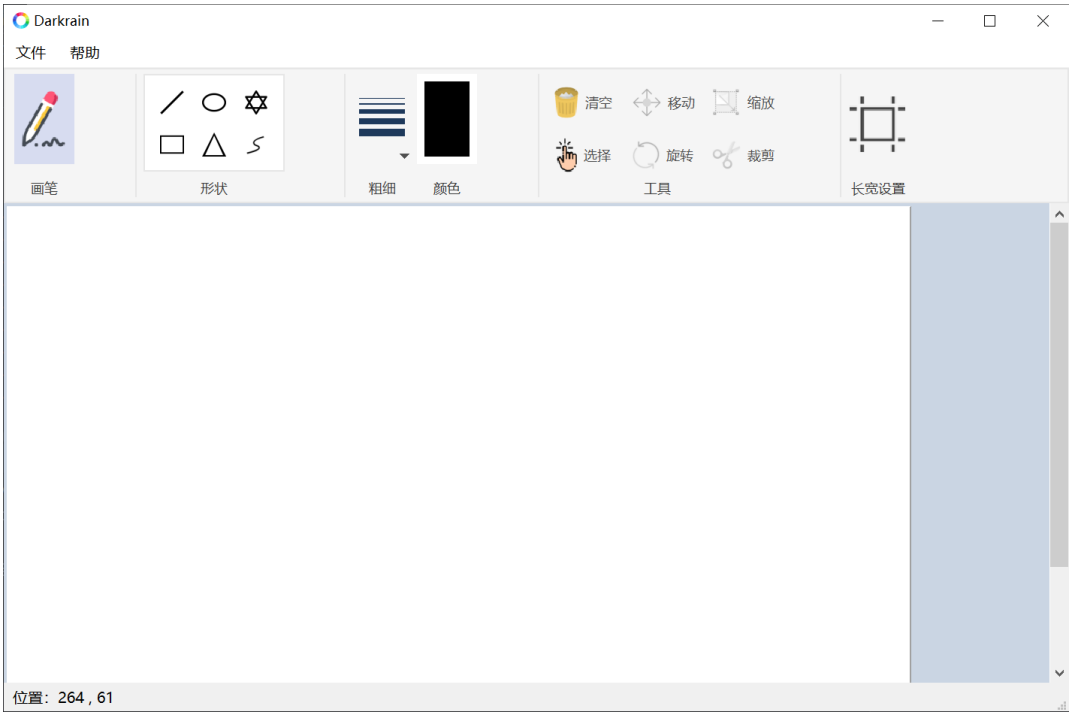
```
DarkrainSp
│   config.py
│   curve.py
│   ellipse.py
│   line.py
```

```
    main.py
    main.spec
    polygon.py
    primitive.py
```

5.2 运行主要流程

5.2.1 Darkrain 运行主要流程

在介绍 Darkrain 之前，首先展示一下主界面结构，以对其功能有初步了解：



软件打开后，通过点按相关形状按钮，在画布上按压、拖动即可绘制图元；通过点按相关功能按钮，即可进行相应的功能操作。

简单的运行流程是这样的：程序运行后，首先通过预先写好的界面设置，设置主界面，导入设置数据、图标，主窗口生成成功后，开始监听事件，事件包括按钮操作以及鼠标在画板上的鼠标操作，将相应的操作对应到不同的处理函数中去，进行相应的处理操作。并且通过关闭操作关闭软件。

5.2.2 DarkrainSp 运行主要流程

DarkrainSp 的主要运行流程比较简单，在通过命令行打开之后，逐行读入数据，将数据传入到指令解析函数中，根据不同的指令执行不同的操作，当全部指令读入完毕后，关闭程序。不同的操作中，包括图元绘制、图元编辑、画布调整、文件保存等相关功能。

5.3 核心数据结构

5.3.1 MyQLabel 自定义画板类

MyQLabel 类的初始化函数中主要的内容如下：

Pixmap 主要用来储存当前绘画内容，可以直接输出到 Qlabel 上，可以看到有三个 Pixmap，其作用会在后面提到。

```
# init PixMap
self.mainPixMap = QPixmap(qsize)
self.mainPixMap.fill(QColor("white"))
self.setPixmap(self.mainPixMap)
self.tempPixMap = self.mainPixMap.copy()
self.backPixMap = self.mainPixMap.copy()
self.setPixmap(self.tempPixMap)
```

初始化画笔类：

```
# init painter
self.mainPen = QPen()
self.mainPen.setColor(QColor("black"))
self.mainPen.setWidth(2)
self.painter = QPainter()
self.painter.begin(self.mainPixMap)
self.painter.setPen(self.mainPen)
```

初始化模式 mode、图元列表 primList，以及编辑状态下的相关内容。

```
# init mode
self.mode = PencilMode

# init primList to save primitives
self.primList = []
self.nowPrim = -1
self.nowChoose = -1
self.chooseTool = 0

# init tool
self.tempPosiSet = []
self.tempPolygon = None
self.tempStart = QPoint(0, 0)
self.tempEnd = QPoint(0, 0)
self.ToolPoint1 = QPoint(0, 0)
self.ToolPoint2 = QPoint(0, 0)
```

5.3.2 图元类（以 Line 为例）

定义了图元的开始位置、结束位置以及宽度、颜色、使用算法等参数。

```
def __init__(self, sPos: QtCore.QPoint = QtCore.QPoint(0, 0),
             ePos: QtCore.QPoint = QtCore.QPoint(0, 0),
             width: int = 2,
             color: QtGui.QColor = QtGui.QColor("black"),
             algorithm: str = "Bresenham"):
    super().__init__()
    self.startPos = sPos
    self.endPos = ePos
    self.width = width
    self.color = color
    self.algorithm = algorithm
    if algorithm != "Bresenham" and algorithm != "DDA":
        ErrorExit("Wrong Line Algorithm", -1)
    self.pointList = []
    if sPos != ePos:
        self.rewrite()
```

5.4 程序核心功能及运行过程

5.4.1 程序及界面初始化

首先，程序打开，通过 main.py 中的 app()函数，进入到 mainwindow.py 中。在 mainwindow.py 中 首先将 MyMainWindow 类(继承于 QMainWindow)实例化，实例化主要分为以下几个过程：

1) 通过读取 UI 界面(在 mainwindowUI.py 中，在这个文件中只有整体界面的框架)以及 QSS 设置文件（主要是设置主界面的菜单栏和按键样式），设置基本图形界面；
2) 实例化并初始化 MyQLabel(继承于 QLabel 类，在这里作为自定义的画布类)，并进行界面参数设置；

```
# set myQLabel
self.picLabel = myQLabel.MyLabel(self.scrollAreaWidgetContents,
                                 self, QtCore.QSize(START_WIDTH, START_HEIGHT))
sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Preferred, QtWidgets.QSizePolicy.Preferred)
sizePolicy.setHorizontalStretch(0)
sizePolicy.setVerticalStretch(0)
sizePolicy.setHeightForWidth(self.picLabel.sizePolicy().hasHeightForWidth())
self.picLabel.setSizePolicy(sizePolicy)
self.picLabel.setCursor(QtGui.QCursor(QtCore.Qt.CrossCursor))
self.picLabel.setAutoFillBackground(True)
self.picLabel.setFrameShape(QtWidgets.QFrame.Panel)
self.picLabel.setFrameShadow(QtWidgets.QFrame.Raised)
self.picLabel.setText("")
self.picLabel.setObjectName("picLabel")
self.formLayout.setWidget(0, QtWidgets.QFormLayout.LabelRole, self.picLabel)
self.picLabel.setMouseTracking(True)
```

3) 设置“宽度设置”按钮的下拉菜单；
4) 绑定按钮 clicked 信号、动作 triggered 信号到相应的处理函数上；
5) 设置按钮的图标
6) 设置按钮的按键状态(setCheckable)，并预先将图元编辑相关操作设置为 setdisabled(Ture).

5.4.2 图元绘制过程

主界面实例化完成之后，通过 show()显示界面，而后通过 exec()监听事件。鼠标点击不同按钮会触发不同的处理函数。我们首先来看一下与图元相关联的响应函数。

```
# --pushButton
# -1.
self.pushButtonLine.clicked.connect(self.LineMode)
self.pushButtonPolygon.clicked.connect(self.PolygonMode)
self.pushButtonEllipse.clicked.connect(self.EllipseMode)
self.pushButtonPencil.clicked.connect(self.PencilMode)
self.pushButtonRect.clicked.connect(self.RectMode)
self.pushButtonTriangle.clicked.connect(self.TriangleMode)
self.pushButtonCurve.clicked.connect(self.CurveMode)
```

pushButtonxxxx 代表着相应的功能按钮，不同的图元按钮在被点击后，相应的事件响应函数就会被触发。

下面以 Line（线段图元）为例，介绍整个图元绘制过程，以期待介绍本程序的主要内容。pushButtonLine 的 clicked 信号与 LineMode 函数相关联，点击 Line 的按键后，LineMode 函数会被执行：

LineMode 函数如下：

```
def LineMode(self):
    self.resetChecked(self.pushButtonLine)
    self.picLabel.changeMode(LineMode)
    LOG("Line Mode")
```

在函数中执行了两步操作，resetChecked 是封装的函数，它会设置当前图形界面上图元相关按键的状态，会将其他的图元 checked 状态设置为 False，而将本按键的状态设置为 True。而后执行 picLabel(即之前实例化的 myQlabel)的 changeMode 函数。

在介绍 changeMode 之前，我首先介绍一下本程序的模式切换功能，在 config.py 中设置了多种模式：

```
# define ModeType
LineMode = 1
PolygonMode = 2
EllipseMode = 3
TriangleMode = 4
RectMode = 5
CurveMode = 6
PencilMode = 10
ClearMode = 11
ChooseMode = 12
```

其中大部分是不同图元绘制的模式，还有 Clear、以及 Choose 模式，分别对应着图元清空、图元选择。

接下来看一下图元模式转换函数（部分操作已折叠）：

```
def changeMode(self, newMode: int) -> None:
    # before change
    LOG("change mode: " + str(self.mode) + '  ' + str(newMode) + ' nowPrim' + str(self.nowPrim))

    if self.mode == PolygonMode and (newMode != PolygonMode and newMode != ClearMode):...

    if self.mode == CurveMode and (newMode != CurveMode and newMode != ClearMode):...

    if self.mode == ChooseMode:...

    # start change
    self.mode = newMode

    # after change
    if newMode == ClearMode:...

    # reshow
    self.reShowPicAll()
```

当前的模式为 self.mode，待转换的模式为 newmode。函数操作分为了转换前、转换和转换后三个部分。对于正常的转换操作，函数执行比较简单，只会简单的将模式转换，即改变 mode 为 newmode。当前 mode 为 Polygon 或者是 Curve 时，会结束图元的绘制操作，即代表输入以及结束。ChooseMode 和 ClearMode 的相关操作会在后面介绍。

在转换 mode 之后，就可以进行在画布上点按鼠标进行图元操作了。

鼠标在画布上会有三个事件被捕捉，分别是 Press, Move, Release，分别对应着 mousePressEvent，mouseMoveEvent，mouseReleaseEvent 三个事件处理函数。

```
def mousePressEvent(self, ev: QMouseEvent) -> None:...

def mouseMoveEvent(self, a0: QMouseEvent) -> None:...

def mouseReleaseEvent(self, ev: QMouseEvent) -> None:...
```

现在我们来看一下一次鼠标在画布上点击、拖动及释放的完全过程。

首先来看 mousePressEvent：

```
def mousePressEvent(self, ev: QMouseEvent) -> None:
    # get position
    LOG("Button push start")
    # why the position?
    xPos = ev.x() - self.pos().x() + self.geometry().x()
    yPos = ev.y() - self.pos().y() + self.geometry().y()
    LOG("pos: ", xPos, yPos, self.geometry().x(), self.geometry().y())
    # LOG("Pos: "+str(xPos)+str(yPos))
    nowPos = QPoint(xPos, yPos)
    # self.tempPixMap = self.mainPixMap.copy()
```

在本函数的初始部分，会获取当前鼠标在画布上的相对位置。
而后，根据不同的模式会进入到不同的操作：

```
# line
if self.mode == LineMode:...

# polygon
elif self.mode == PolygonMode:...

# additional polygon: Triangle rect
elif self.mode == RectMode or self.mode == TriangleMode:...

# ellipse
elif self.mode == EllipseMode:...

# curve
elif self.mode == CurveMode:...

# pencil
elif self.mode == PencilMode:...

# choose mode
elif self.mode == ChooseMode:...

# reprint
self.reShowPic()
```

在 LineMode 下：

```
if self.mode == LineMode:
    newLine = line.Line(nowPos, nowPos, self.mainPen.width(), self.mainPen.color())
    self.primList.append(newLine)
    self.nowPrim = len(self.primList) - 1
```

首先通过建立一个新的 Line 类实例，而后把 line 加入到 self.primList 就是当前程序的图元列表中去，修改当前图元，使其指向 newLine。
拖动鼠标时，mouseMoveEvent 触发，同样是获取当前的坐标位置，而后：

```
if a0.buttons() and Qt.LeftButton:
    LOG("Button Moving")
    # line
    if self.mode == LineMode:
        self.primList[self.nowPrim].setEnd(nowPos)
```

根据当前坐标，设置线段的终点。
这时，通过 reshowPic 函数刷新 MyQLabel 就可以显示线段了，且会随着鼠标拖动而移动。

```
def reShowPic(self) -> None:
    LOG("reshow start")
    self.painter.drawPixmap(0, 0, self.mainPixMap.width(),
                            self.mainPixMap.height(), self.tempPixMap)
    if self.mode is ChooseMode:...

    else:  # normal
        if 0 <= self.nowPrim < len(self.primList):
            tempColor = self.mainPen.color()
            tempWidth = self.mainPen.width()
            self.mainPen.setColor(self.primList[self.nowPrim].color)
            self.mainPen.setWidth(self.primList[self.nowPrim].width)
            self.painter.setPen(self.mainPen)
            for point in self.primList[self.nowPrim].pointList:
                self.painter.drawPoint(point)
            self.mainPen.setColor(tempColor)
            self.mainPen.setWidth(tempWidth)
            self.painter.setPen(self.mainPen)
```

在 reshowPic 中，当前模式不为 ChooseMode 时（选择模式下只重新输出选择图元），只会重新输出当前图元，所以在图元数量较大的时候，也不会因为需要刷新的图元过多而降低系统速度。

在 mouseReleaseEvent 中的操作比较简单

```
def mouseReleaseEvent(self, ev: QMouseEvent) -> None:
    LOG("Button Release")
    if self.mode != ChooseMode:
        self.tempPixMap = self.mainPixMap.copy()
    else:
        if self.chooseTool == Clip:...
```

上面就是绘制线段的全过程。

其他图元的绘制过程与其相类似，需要注意的是对于多边形还有一些其他的相关由多点确定的图形，再点击时的操作差距比较大，以 Polygon 为例：

```
# polygon
elif self.mode == PolygonMode:
    if 0 <= self.nowPrim < len(self.primList) and self.primList[self.nowPrim].isEnd is False:
        self.primList[self.nowPrim].appendPos(nowPos)
    else:
        LOG("new Polygon!")
        newPolygon = polygon.Polygon(self.mainPen.width(), self.mainPen.color())
        newPolygon.setStart(nowPos)
        newPolygon.appendPos(QPoint(xPos + 1, yPos + 1))
        self.primList.append(newPolygon)
        self.nowPrim = len(self.primList) - 1
```

首先会先进行判断如果判断当前还有一个多边形未输入完，当前点会被加入到先前多边形的点集中去。

5.4.3 图元编辑过程

图元编辑过程的核心部分就是根据所选择的图元以及选定的编辑工具，通过鼠标获取调整变量，对图元进行编辑的过程。

核心是编辑工具的切换，以及不同编辑状态下的操作不同。

首先就切换编辑工具作介绍：

在选择模式下，选择一个图元后，会进入到 switchChoose 中，设置能够使用的功能按钮状态：

```
if isinstance(self.primList[num], line.Line):
    self.parentWindow.pushButtonScale.setDisabled(False)
    self.parentWindow.pushButtonMove.setDisabled(False)
    self.parentWindow.pushButtonRotate.setDisabled(False)
    self.parentWindow.pushButtonClip.setDisabled(False)
elif isinstance(self.primList[num], ellipse.Ellipse):
    self.parentWindow.pushButtonScale.setDisabled(False)
    self.parentWindow.pushButtonMove.setDisabled(False)
elif isinstance(self.primList[num], polygon.Polygon):
    self.parentWindow.pushButtonScale.setDisabled(False)
    self.parentWindow.pushButtonMove.setDisabled(False)
    self.parentWindow.pushButtonRotate.setDisabled(False)
elif isinstance(self.primList[num], curve.Curve):
    self.parentWindow.pushButtonScale.setDisabled(False)
    self.parentWindow.pushButtonMove.setDisabled(False)
    self.parentWindow.pushButtonRotate.setDisabled(False)
elif isinstance(self.primList[num], addpolygon.AddPolygon):
    self.parentWindow.pushButtonScale.setDisabled(False)
    self.parentWindow.pushButtonMove.setDisabled(False)
    self.parentWindow.pushButtonRotate.setDisabled(False)
elif isinstance(self.primList[num], pencil.Pencil):
    self.parentWindow.pushButtonMove.setDisabled(False)
```

之后比较重要的就是在点按时：

按照不同的选择的编辑工具分别操作：

mousePressEvent：

```
elif self.mode == ChooseMode:
    LOG("choose push down")
    if self.chooseTool != 0:
        self.ToolPoint1 = nowPos
        self.ToolPoint2 = nowPos
        self.tempStart = self.primList[self.nowChoose].startPos
        if isinstance(self.primList[self.nowChoose], line.Line) \
                or isinstance(self.primList[self.nowChoose], ellipse.Ellipse):
            self.tempEnd = self.primList[self.nowChoose].endPos
        elif isinstance(self.primList[self.nowChoose], addpolygon.AddPolygon):
            self.tempEnd = self.primList[self.nowChoose].endPos
            self.tempPosiSet = self.primList[self.nowChoose].PosiSet.copy()
        else:
            self.tempPosiSet = self.primList[self.nowChoose].PosiSet.copy()
    if self.chooseTool == Clip:
        self.tempPolygon = addpolygon.AddPolygon(Rect, 2, QColor("red"))
        self.tempPolygon.setStart(nowPos)
```

mouseMoveEvent：

```
if self.chooseTool == Move:
    nowPrim.translate(self.ToolPoint2.x() - self.ToolPoint1.x(),
                      self.ToolPoint2.y() - self.ToolPoint1.y())
elif self.chooseTool == Rotate:
    LOG("rotate start")
    nowPrim.rotate(nowPrim.startPos, self.ToolPoint1, self.ToolPoint2)
elif self.chooseTool == Scale:
    LOG("scale")
    standard = sqrt((self.ToolPoint1.x() - nowPrim.startPos.x())
                    * (self.ToolPoint1.x() - nowPrim.startPos.x()) +
                    (self.ToolPoint1.y() - nowPrim.startPos.y())
                    * (self.ToolPoint1.y() - nowPrim.startPos.y()))
    scaled = sqrt((self.ToolPoint2.x() - nowPrim.startPos.x())
                  * (self.ToolPoint2.x() - nowPrim.startPos.x()) +
                  (self.ToolPoint2.y() - nowPrim.startPos.y())
                  * (self.ToolPoint2.y() - nowPrim.startPos.y()))
    scale_n = scaled / standard
    nowPrim.scale(nowPrim.startPos.x(), nowPrim.startPos.y(), scale_n)
elif self.chooseTool == Clip:
    self.tempPolygon.setEnd(nowPos)
```

mouseReleaseEvent：

```
if self.chooseTool == Clip:
    startPos = self.tempPolygon.startPos
    endPos = self.tempPolygon.endPos
    LOG("ready for clip")
    LOG(startPos.x(), startPos.y(), endPos.x(), endPos.y())
    self.primList[self.nowChoose].clip(startPos.x(), startPos.y(),
                                       endPos.x(), endPos.y())
    self.tempPolygon = None
    self.reShowPic()
```

5.4.4 相关其他功能

相关的辅助功能包括：文件的打开、图片的保存，以及画笔的颜色、宽度设置，画布的大小调整、清空等等。

在这里就不加以展示了。

5.5 图元算法实现

这里我将介绍实验中所实现的图元相关的绘制、编辑算法，由于整体内容较多，图元绘制部分我选择最为重要的线段绘制、椭圆绘制、曲线绘制，图形编辑部份则以线段的编辑为例，来介绍四种图形编辑与变化（平移、旋转、放缩、裁剪）。

由于相关算法的理论部分已经在前面加以介绍了，这里只做代码核心部分的简单展示。

5.5.1 图元绘制算法

5.5.1.1 线段绘制

1) DDA 算法绘制

```
dX = abs(self.startPos.x() - self.endPos.x())
dY = abs(self.startPos.y() - self.endPos.y())
# 2 way:
if -1 <= k <= 1:
    # get start end
    if startX > endX:
        startX = self.endPos.x()
        startY = self.endPos.y()
        endX = self.startPos.x()
        endY = self.startPos.y()
    y = float(startY)
    for x in range(startX, endX):
        y += k
        self.pointList.append(QtCore.QPoint(x, int(y)))
else:  # |k|>1
    # get start end
    if startY > endY:
        startX = self.endPos.x()
        startY = self.endPos.y()
        endX = self.startPos.x()
        endY = self.startPos.y()
    x = float(startX)
    for y in range(startY, endY):
        x += 1 / k
        self.pointList.append(QtCore.QPoint(int(x), y))
```

2) Bresenham 算法绘制(其中一种情况)

```
if -1 <= k <= 1:
    d1 = 2 * dY
    d2 = 2 * (dY - dX)
    p = 2 * dY - dX
    # choose step
    if k > 0:
        step = 1
    else:
        step = -1
    # get start end
    if startX > endX:
        startX = self.endPos.x()
        startY = self.endPos.y()
        endX = self.startPos.x()
        endY = self.startPos.y()
    y = startY
    self.pointList.append(QtCore.QPoint(startX, startY))
    self.pointList.append(QtCore.QPoint(endX, endY))
    for x in range(startX, endX):
        if p < 0:
            p += d1
        else:
            y += step
            p += d2
        self.pointList.append(QtCore.QPoint(x, y))
```

5.5.1.2 椭圆绘制

1) Bresenham 椭圆算法

```
# paint first half
p1 = SquareShortR - SquareLongR * shortR + SquareLongR / 4.0
while (tempX+1.0) * SquareShortR < (tempY+0.5) * SquareLongR:
    self.addPoint(centerX, centerY, tempX, tempY, horizontalFocus)
    if p1 < 1:
        p1 += 2 * SquareShortR * tempX + 3 * SquareShortR
        tempX += 1
    else:
        p1 += 2 * SquareShortR * tempX + 3 * SquareShortR - 2 * SquareLongR * tempY + 2 * SquareLongR
        tempX += 1
        tempY -= 1

LOG("pointList3 size: ", len(self.pointList))
# paint last half
p2 = SquareShortR * (tempX + 0.5) * (tempX + 0.5) \
    + SquareLongR * (tempY - 1) * (tempY - 1) \
    - SquareShortR * SquareLongR
while tempY >= 0:
    self.addPoint(centerX, centerY, tempX, tempY, horizontalFocus)
    if p2 >= 0:
        p2 += -2 * SquareLongR * tempY + 3 * SquareLongR
        tempY -= 1
    else:
        p2 += 2 * SquareShortR * tempX + 2 * SquareShortR - 2 * SquareLongR * tempY + 2 * SquareLongR
        tempX += 1
        tempY -= 1
```

5.5.1.3 曲线绘制

1) Bezier 曲线绘制

```
while t <= 1:
    for k in range(pos_len):
        x_array[k] = self.PosiSet[k].x()
        y_array[k] = self.PosiSet[k].y()
    for i in range(1, pos_len):
        for j in range(pos_len - i):
            x_array[j] = (1 - t) * x_array[j] + t * x_array[j + 1]
            y_array[j] = (1 - t) * y_array[j] + t * y_array[j + 1]
    if abs(toInt(x_array[0]) - toInt(x)) <= 1 and abs(toInt(y_array[0]) - toInt(y)) <= 1:
        self.pointList.append(QtCore.QPoint(toInt(x_array[0]), toInt(y_array[0])))
    else:
        tempLine = line.Line(QtCore.QPoint(toInt(x), toInt(y)),
                             QtCore.QPoint(toInt(x_array[0]), toInt(y_array[0])),
                             self.color, "Bresenham")
        self.pointList.extend(tempLine.getDrawPoint())
    x = x_array[0]
    y = y_array[0]
    t += 1 / 64 * pos_len
```

2) B 样条

```
while u <= pos_len:
    x = 0
    y = 0
    for i in range(pos_len):
        ratio = self.bspline_tool(u, i, k)
        # (ratio)
        x += self.PosiSet[i].x() * ratio
        y += self.PosiSet[i].y() * ratio
    # LOG(x, y)
    if x == 0 and y == 0:...
    if begin is True:...
    else:
        if abs(toInt(x_s) - toInt(x)) == 0 and abs(toInt(y_s) - toInt(y)) == 0:
            pass
        elif abs(toInt(x_s) - toInt(x)) <= 1 and abs(toInt(y_s) - toInt(y)) <= 1:
            self.pointList.append(QtCore.QPoint(toInt(x), toInt(y)))
        else:...
        x_s = x
        y_s = y
    if 1e-2 > u - self.lastSave > -1e-5:
        LOG("copy save")
        self.tempPointList = self.pointList.copy()
    u += 1.0 / 128
```

5.5.2 图元编辑、变换算法

5.5.2.1 平移

```
def translate(self, dx: int, dy: int):
    self.startPos = QtCore.QPoint(self.startPos.x() + dx, self.startPos.y() + dy)
    self.endPos = QtCore.QPoint(self.endPos.x() + dx, self.endPos.y() + dy)
    self.rewrite()
```

5.5.2.2 旋转

```
def rotate(self, basic: QtCore.QPoint,
           start: QtCore.QPoint, end: QtCore.QPoint):
    LOG("rotate p1")
    if basic == start and basic == end:
        return
    cos_val = cos_by_point(basic, start, end)
    sin_val = sin_by_point(basic, start, end)
    LOG(cos_val, ', ', sin_val)
    LOG(cos_val * cos_val + sin_val * sin_val)
    x = basic.x()
    y = basic.y()
    LOG(self.startPos.x(), self.startPos.y(), self.endPos.x(), self.endPos.y())
    LOG("rotate p2")
    temp_x = self.startPos.x()
    temp_y = self.startPos.y()
    self.startPos = QtCore.QPoint(x + (temp_x - x) * cos_val - (temp_y - y) * sin_val,
                                  y + (temp_x - x) * sin_val + (temp_y - y) * cos_val)
    temp_x = self.endPos.x()
    temp_y = self.endPos.y()
    self.endPos = QtCore.QPoint(x + (temp_x - x) * cos_val - (temp_y - y) * sin_val,
                                y + (temp_x - x) * sin_val + (temp_y - y) * cos_val)

    LOG(self.startPos.x(), self.startPos.y(), self.endPos.x(), self.endPos.y())
    self.rewrite()
```

5.5.2.3 放缩

```
def scale(self, x: int, y: int, r: float):
    temp_x = self.startPos.x()
    temp_y = self.startPos.y()
    self.startPos = QtCore.QPoint(toInt(temp_x * r + x * (1 - r)),
                                  toInt(temp_y * r + y * (1 - r)))
    temp_x = self.endPos.x()
    temp_y = self.endPos.y()
    self.endPos = QtCore.QPoint(toInt(temp_x * r + x * (1 - r)),
                                toInt(temp_y * r + y * (1 - r)))
    self.rewrite()
```

5.5.2.4 裁剪

1) Cohen-Sutherland 算法

```
while True:
    if (code1 | code2) == 0:
        acc = True
        break
    elif code1 & code2:
        break
    else:
        code = code1
        if code1 == 0:
            code = code2
        if code & TOP:
            x = X1 + (X2 - X1) * (yMax - Y1) / (Y2 - Y1)
            y = yMax
        elif code & BOTTOM:
            x = X1 + (X2 - X1) * (yMin - Y1) / (Y2 - Y1)
            y = yMin
        elif code & RIGHT:
            y = Y1 + (Y2 - Y1) * (xMax - X1) / (X2 - X1)
            x = xMax
        else:
            y = Y1 + (Y2 - Y1) * (xMin - X1) / (X2 - X1)
            x = xMin

        if code == code1:
            X1 = x
            Y1 = y
            code1 = getOutCode(X1, Y1, xMin, xMax, yMin, yMax)
        else:
            X2 = x
            Y2 = y
            code2 = getOutCode(X2, Y2, xMin, xMax, yMin, yMax)
```

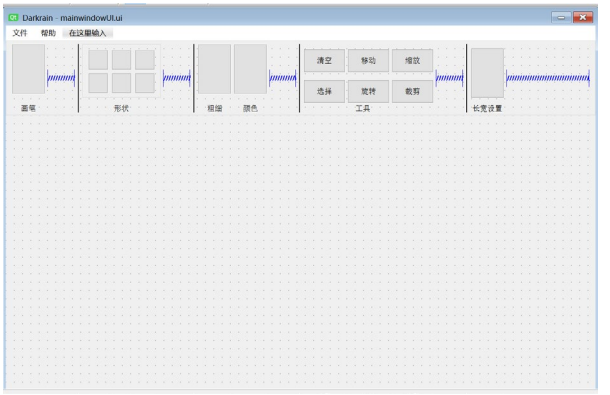
2) Liang-Barsky 算法

```
for i in range(4):
    if p[i] == 0:
        continue
    LOG(u1, ' ', u2)
    r = q[i] / p[i]
    if p[i] < 0:
        if r > u2:
            exist = 0
            break
        if r > u1:
            u1 = r
    elif p[i] > 0:
        if r < u1:
            exist = 0
            break
        if r < u2:
            u2 = r
```

5.6 UI界面设计

5.6.1 QT Designer 设计界面

利用可视化界面设计整体框架：



5.6.2 QSS 部分

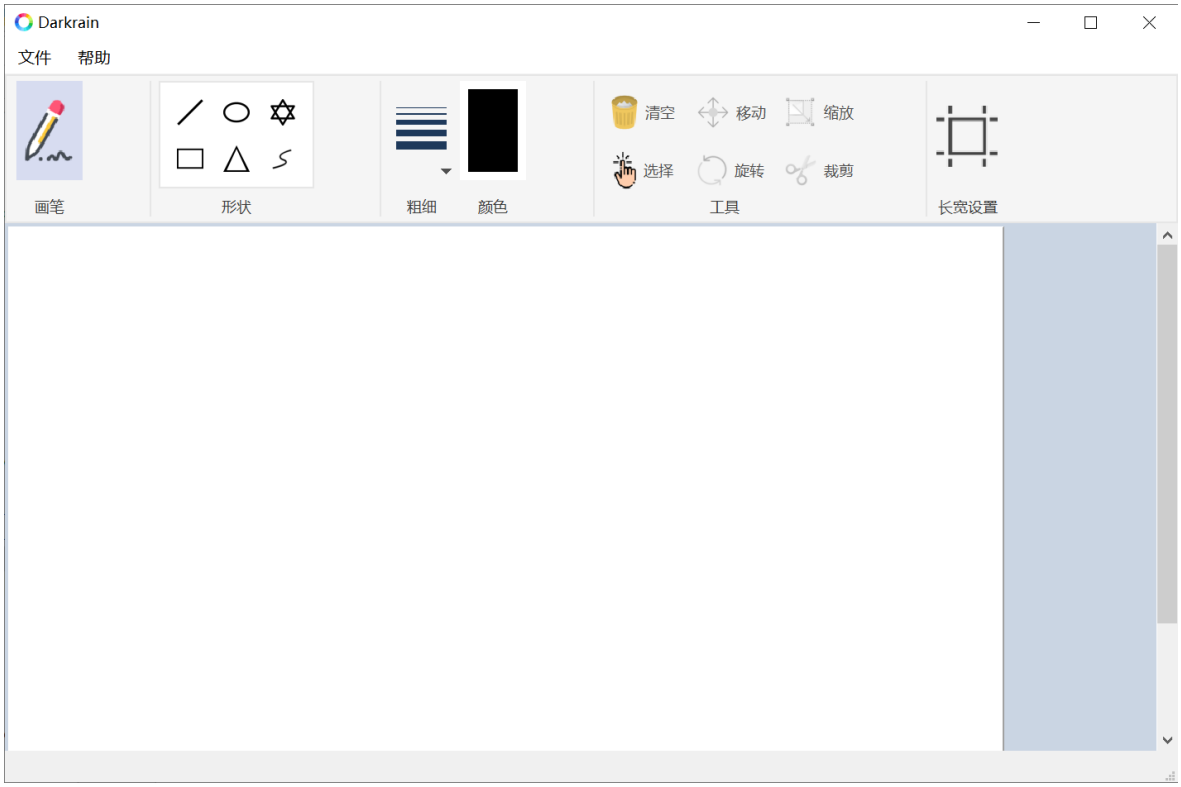
主要对按键以及相关控件进行重新设置：

```
QGroupBox{
    background-color: rgb(245, 245, 245);
    font:10px "Microsoft YaHei";
    color: white;
    border:0.5 solid rgb(230,230, 230);
    /*border-style :none*/
}
QGroupBox#groupBox_2{
    font:10px "Microsoft YaHei";
    background-color: rgb(255, 255, 255);
}
QLabel{
    font: 9pt "Microsoft YaHei UI";
    color: rgb(60,60,60)
}
QFrame[frameShape="5"]
{
    color:  rgb(230,230, 230);
}
QPushButton
{
    font:9px "Microsoft YaHei";
    background-color:transparent;
    color: rgb(80,80,80)
}
```

5.6.3 代码部分

代码部分在前面的初始化部分已经介绍，未介绍的主要就是的就是设置了滑动窗口区域，便于浏览。

几个部分结合的整体效果即为：



5.7 优化与性能改善

在多个地方我都对代码进行了一定程度优化和改善，这里选则一部分做简单说明。

1) 多个 Pixmap

利用多个 Pixmap，实现了对导入的背景图片的保存，使得能够在调整大小后也不损失背景图片的信息，同时又保证了原来的画布能在图片导入时随着图片大小改变

2) 多个刷新函数

刷新函数就是重新在 PixMap 上绘制的点，在不需要全部重新绘制的时候（即大部分情况，包括正常的图元绘制、图元编辑），只有在切换模式等少量场景全部刷新，大大提高了系统的性能。

3) 补充的画笔功能

利用补充的画笔功能，可以更加便捷的画出希望画出的图形，更加符合绘图软件的需求。

4) 曲线绘制优化

利用 B 样条的性质，绘图时，在重新绘制时只绘制最后一部分点的内容，因为前面的部分没有改变；而在图元编辑时则能够全部重新绘制。

6 结束语

通过学习课本上所讲的图元绘制、变换、剪裁算法，并且利用 Qt Creator+ PyQt5 进行软件系统 GUI 的编写开发，完整地实现了一整套图形绘制、编辑系统，包括以 DarkrainSp 命名的命令行程序以及 Darkrain 命名的图形化界面程序。通过实现课本中的相关画图算法，以及构建完整的图形界面系统，来实现作业所需要的画图、打开、保存以及图形显示的相关功能。通过自己的测试，验证了程序能够完成其预先功能。

但是，随着实验的不断深入，我还知道在我的程序中还有这许多可以改善的功能。比如说图元的选择操作也可以改用 GUI 方式进行、绘制曲线时能提升更多的速度、操作管理上可以提供撤销恢复操作、文件管理上可以进一步完善、图形化界面也可以学习更好的 UI 设计方法。除此之外，还可以尝试着 3D 图形的显示、编辑。

References:

[1] Digital differential analyzer - Wikipedia: https://en.wikipedia.org/wiki/Digital_differential_analyzer_(graphics_algorithm)

[2] Bresenham 算法原理: https://blog.csdn.net/yzh1994414/article/details/82860187

[3] "The Bresenham Line-Drawing Algorithm" , by Colin Flanagan

[4] Cohen-Sutherland 直线裁剪算法 https://blog.csdn.net/jxch___/article/details/80726853

[5] 梁友栋-Barsky 裁剪算法 https://www.cnblogs.com/jenry/archive/2012/02/12/2347983.html

[6] **A Primer on Bézier Curves** https://pomax.github.io/bezierinfo/zh-CN/#decasteljau

[7] Les Piegl and Wayne Tiller: The NURBS Book, Springer-Verlag 1995-1997 (2nd ed).

[8] 《计算机图形学教程》孙正兴