《计算机图形学》12 月报告

171830526 赵文正

摘要：在 12 月的进度中，我实现了全部的图元绘制和图元变换功能。并且，现在的程序已经支持从控制台输入参数运行（这种方式只能读取指令文件，不会调用 GUI）。在图形界面下运行的功能也已经全部开发完毕，同时在图形界面做了支持指令输入和文件读取的接口功能。

一、需求分析：

本次作业要求实现一个有基础功能的画图软件，具体要实现的内容有：

核心算法模块（各种图元的生成、变换算法）

文件输入接口（读取包含了图元绘制指令序列的文本文件，依据指令调用①中的算法绘制图形以及保存图像）

用户交互接口（以鼠标交互的方式，通过鼠标事件获取所需参数并调用①中的算法将图元绘制到屏幕上）

二、作业要求：

本次作业，编程语言不限（C++，Python，Java 等），开发平台不限（Windows，Mac OS，Linux 等），GUI 开发框架不限（Qt，Tkinter，Web 等）。

三、开发环境：

程序开发语言：C++，MinGW 7.3.0 32-bit for C++

开发平台：Windows10

GUI 开发框架：Qt 5.13.1

四、程序框架设计：

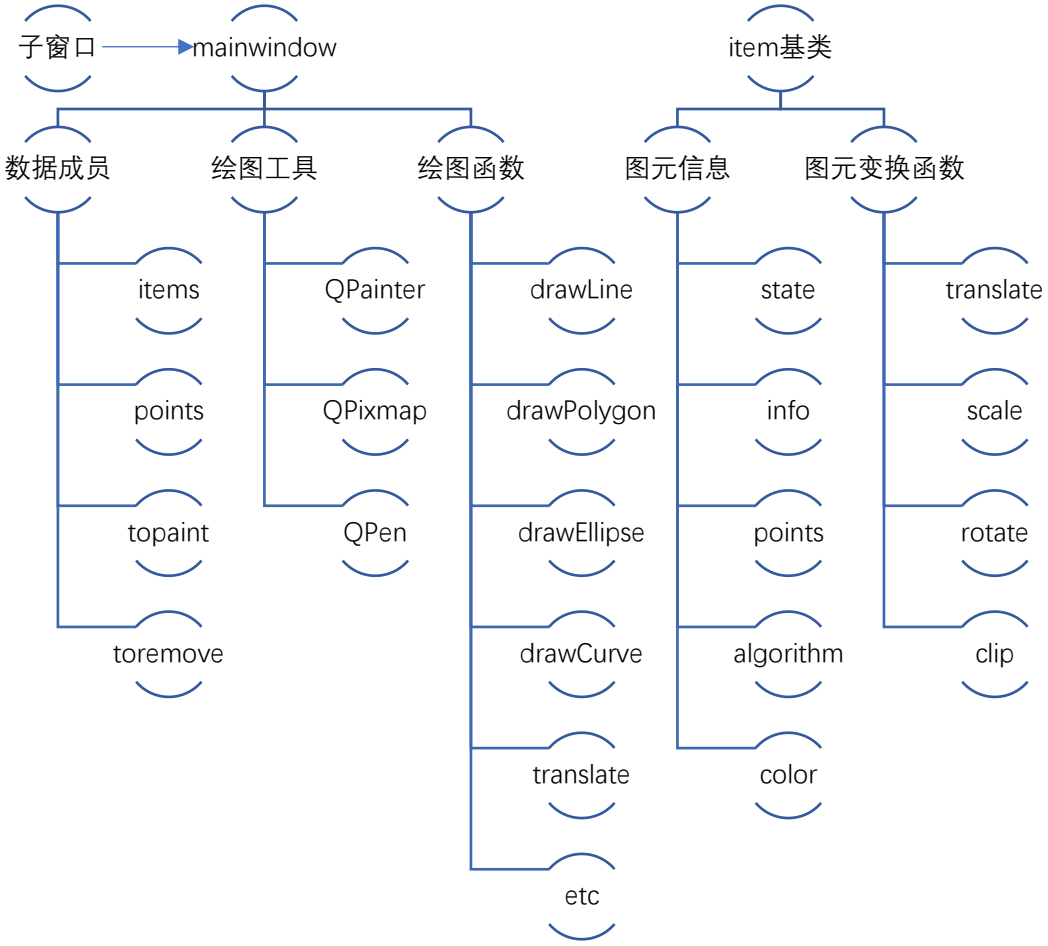
本次作业我选用 Qt 中的 MainWindow 类作为程序主窗口，ui 设计通过 Qt Designer 实现。采用 Qt 封装的 QPainter 来实现画笔功能，PaintDevice 则选用了绘制 2D 图形较方便的 QPixmap，通过 QPainter 在 QPixmap 上绘制，最后再调用 drawPixmap 将绘制的图形显示到主窗口上。

在 item.h 中有图元类 item 的定义，用来存储图元信息，实现对图元进行变换操作，以及支持一些来自上层需要调用的功能。

除此之外，其他.ui 和.h .cpp 文件均为 MainWindow 的子窗口，用于在鼠标选择相应菜

main.cpp	程序入口
mainwindow.h	主窗口头文件
mainwindow.cpp	主窗口源文件，实现大部分的绘图功能
mainwindow.ui	主窗口 ui 文件，用于图形界面设计
input_instruct.h .cpp .ui	输入指令的子窗口相关文件
invalid_instruct.h .cpp .ui	输出提示信息的子窗口相关文件
reset.h .cpp .ui	一个简单的弹出窗口
item.h	图元信息头文件
item.cpp	图元信息源文件，实现图元变换功能

单（如命令行输入）时，弹出子对话框。代码结构如下：



五、数据结构设计：

图元信息：

有关图元信息的内容定义在 item.h 文件中。首先我设计了一个 Point 结构体，用于存储点的信息：

```
struct Point{
    int x, y; //坐标
    QColor color; //颜色信息 0<=R, G, B<=255
    Point(int xx, int yy, QColor c) { x = xx; y = yy; color = c; }
};
```

同时定义了一个枚举类型 itemState，用来表示图元类型：

```
enum itemState {Line, Polygon, Ellipse, Curve};
```

然后是 item 类，在 item 类中定义了图元 id，图元类型，图元点阵信息等，同时实现了所有图元变换功能（包括平移，缩放，旋转和裁剪），以及一些对上层调用的反馈。

其中，旋转不支持椭圆，而裁剪只支持线段。

```
class item{
private:
    int id;
    itemState state;
    QVector<Point> points;//组成图元的所有点
    QVector<pair<int, int>> info;//关键点信息
    QColor clr;//图元颜色
    QString algorithm;//绘图时使用的算法，如果有的话
public:
    item(int i, itemState s) { id = i; state = s; }
    void set_id(int i) { id = i; }
    void clear_points() { points.clear(); }
    void save(const QVector<Point>& pts); //保存点阵
    void save_info(const QVector<pair<int, int>>& p);//保存关键点位置
    void save_color(QColor c) { clr = c; }//保存颜色信息
    void save_algorithm(QString str) { algorithm = str; }
    void translate(int dx, int dy); //平移变换
    void scale(int x, int y, double s);//缩放变换
    void rotate(int x, int y, int r);//旋转变换
    int encode(int x, int y, int xmin, int ymin, int xmax, int ymax);
    bool clip_Cohen_Sutherland(int x1, int y1, int x2, int y2);//裁剪
变换
    bool clip_Liang_Barsky(int x1, int y1, int x2, int y2);//裁剪变换
    bool find(int x, int y);//查找这个点是否存在
    ~item() { points.clear(); }
};
```

mainwindow:

在 mainwindow 类中，我实现了指令的解析和大部分的绘图函数，同时也定义了一些数据成员，用于存储图元信息等:

```
    Ui::MainWindow *ui;//ui
    //参数
    QString filename;//指令文件名
    QString savepath;//存储路径

    //画布
    QPixmap* board;
    //画笔
    QPainter* painter;//用于调用 drawPixmap
    QPainter* painterpix;//Pixmap 的画笔
    QPen* pen;//设置 painterpix

    //数据成员
    item* current;//正在编辑的图元
```

```
QVector<item*> items;//图元集合
QVector<Point> topaint;//要绘制的点
QVector<Point> toremove;//要移除的点
```

这些数据成员只是对功能实现比较关键的一部分。事实上，还有许多数据成员和程序实现的具体细节有关，这些细节会在后面有所说明。

六、算法实现：

1. 直线绘制算法：

1.1 DDA 算法：

函数原型: void **draw_line_DDA**(int x1, int y1, int x2, int y2);

函数功能:绘制一条从（x_1，y_1）到（x_2，y_2）的直线。

数值微分（DDA：Digital Differential Analyzer）画线算法，基本思想是高数中的微分算法。通常来说，根据直线 y=kx+b 的斜率 k 的绝对值，选择Δx=1 或Δy=1，来计算另一个坐标方向上的下一个像素的位置。这里运用图形学中常见的增量思想，以Δx=1 为例，y_{i+1}=y_i+k，这样就把原本乘法的式子变换成了加法，同时，根据y_i+k 的值，采用四舍五入取整的方式，确定y_{i+1}的位置。

一般来说，|k|≥1 时通常取Δy=1，一步一步去计算x_{i+1}的值；反过来也同样，|k| < 1 时，取Δx=1，计算y_{i+1}的值

1.2 Bresenham 算法：

函数原型: void **draw_line_Bresenham**(int x1, int y1, int x2, int y2);

函数功能:绘制一条从（x_1，y_1）到（x_2，y_2）的直线。

Bresenham 画线算法是一种精确而有效的光栅线段生成算法，它可用于圆和其他曲线显示的整数增量运算。在某个方向以单位间距取样，根据扫描转换原理，在每一个取样位置处，需确定哪个像素位置更接近于线段路径。

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实际线路径的像素。假设，现执行到第 k 步，选取的点为（x_k，y_k），那么只需要确定在列x_{k+1}上绘制哪个像素，是（x_{k+1}，y_k）还是（x_{k+1}，y_{k+1}）？这便是 Bresenham 算法主要讨论的问题。

我们设直线方程为 y=mx+b，显然第 k 步时，y=mx_{k+1}+b=m(x_k+1)+b，那么，两个候选像素与线段数学路径的垂直偏移为：

$$d_1=\text{y-}y_k=\text{m}(x_k+1)+\text{b-}y_k$$

$$d_2=y_{k+1}-y=y_k+1\text{- m}(x_k+1)+\text{b}$$

这两个点的距离差分为：

$$d_1\text{-}d_2=2\text{m}(x_k+1)\text{-}2y_k+2\text{b-}1$$

将 m=Δy/Δx 代入，得到：

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

上式中，p_k被称为第 k 步的决策参数，根据第 k 步的决策参数，可以求得第 k+1 步的决策参数，通过决策参数的正负，来判断选取像素的位置，便是 Bresenham 算法的基本步骤。在代码实现过程中，需要分两种情况进行讨论：斜率绝对值 > 1 和斜率绝对值 < 1，具体实现请参见源代码文件。

2. 多边形绘制算法：

函数原型: void **draw_polygon**(const QVector<pair<int, int>>& ctrlpoints, QString algorithm);

函数功能:以 ctrlpoints 中的点作为控制点，根据 algorithm 确定算法，绘制一个多边形。

多边形绘制算法同样有两种，分别是 DDA 算法和 Bresenham 算法，和直线绘制算法原理几乎相同。事实上，多边形的绘制就等同于多条首尾相接的直线的绘制，只需要将按顺序在相邻两点间绘制直线即可。故这里不再进行介绍，因为算法实质和绘制直线相同。

3. 椭圆绘制算法:

函数原型: void **draw_ellipse**(int xc, int yc, int rx, int ry);
函数功能:绘制一个以（x_c，y_c）为中心，r_x为长半轴，r_y为短半轴的椭圆。

椭圆绘制我采用了中点椭圆生成算法。首先椭圆的方程通常写作:

$$\frac{(x-x_0)^2}{a^2}+\frac{(y-y_0)^2}{b^2}=1$$

我们假设中心为（x_0,y_0），则椭圆方程简化为

$$\frac{x^2}{a^2}+\frac{y^2}{b^2}=1$$

将方程化为函数形式:

$$f(x,y)=r_y^2x^2+r_x^2y^2-r_x^2r_y^2$$

其中：$r_x=a,r_y=b$

算法思想可以解释为，从$(0,\ r_y)$开始，在 x 方向上取单位步长直到切线斜率=-1，然后转为在 y 方向上取单位步长，再覆盖第一象限剩下的曲线段。切线斜率=-1 的条件等价于:

$$2r_y^2x=2r_x^2y$$

算法同样通过中点对决策函数求值，来确定沿椭圆轨迹的下一位置，以区域 1 为例，若$p1_k<0$，则下一个点为(x_{k+1},y_k)，并且:

$$p1_{k+1}=p1_k+2r_y^2x_k+3r_y^2=p1_k+2r_y^2x_{k+1}+r_y^2$$

否则，下一个点为(x_{k+1},y_k-1)，并且:

$$p1_{k+1}=p1_k+2r_y^2x_k+3r_y^2-2r_x^2y_{k+1}=p1_k+2r_y^2x_{k+1}+r_y^2-2r_x^2y_{k+1}$$

概括起来，中点椭圆生成算法的算法步骤如下:

1）输入椭圆中心（x_c，y_c）和长短轴径r_x，r_y。

2）计算得到中心在原点的椭圆上的第一个点$(0,r_y)$。

3）计算区域 1（切线斜率绝对值 < 1）的初始决策参数$p1_0$

$$p1_0=r_y^2-r_x^2\ r_y+\frac{r_x^2}{4}$$

4）在区域 1 中每个x_k位置处，从$k=0$开始，反复按照上述公式确定候选像素，并进行决策参数增量计算，循环至$2r_y^2x\geq 2r_x^2y$为止。

5）使用区域 1 中最后点作为区域 2 的起始点（x_0，y_0）来计算区域 2 中参数初值为

$$p2_0=r_y^2(x_0+\frac{1}{2})^2+r_x^2(y_0-1)^2-r_x^2r_y^2$$

6）在区域 2 的每个y_k位置处，从$k=0$开始，反复按照下面公式确定候选像素，并进行决策参数增量计算，循环至$(r_x,0)$为止

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

7）确定其他三个象限中对称的点，然后将每个计算出来的点的位置平移到以（x_c，y_c）为中心的椭圆轨迹上。

事实上，上述算法在实现过程中，我没有完全按照步骤来实现，而是进行了一些变量上的调整和计算上的简化。例如我额外定义了两个变量 px 和 py，来存储$2r_y^2x_k$和$2r_x^2y_k$的值，

当在区域 1 时，x 每增加 1，就让 px 增加r_y^2，同理当 y 减少时，也让 py 减少r_x^2。这样避免了每次都对$2r_y^2x_k$和$2r_x^2y_k$进行求值比较，使得运算稍微进行了一定程度上的简化。具体实现内容请参见源代码文件。

4. 曲线绘制算法：

4.1 Bezier 曲线绘制算法：

函数原型: void **draw_curve_Bezier**(QVector<pair<int, int>> ctrlpoints, int precision);

函数功能:按照给定的控制点数组 ctrlpoints，按照规定精度 precision 绘制一条 Bezier 曲线。

Bezier 曲线算法的重点在于 Bernstein 基函数，通过基函数我们便可以求得位于曲线上的点，n 次 Bernstein 基函数的多项式形式为：

$$BEZ_{i,n}(u) = C_n^i u^i (1-u)^{n-i}$$

其中：$C_n^i = \frac{n!}{i!(n-i)!}$

Bezier 曲线是通过一组多边折线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曲线的形状有紧密的联系。因此，多边折线通常称为特征多边形，其顶点称为控制顶点。一般来说，Bezier 曲线段可拟合任何数目的控制顶点，且它们的相关位置决定 Bezier 多项式的次数。

假设给出 n+1 个控制顶点$P_i = (x_i, y_i)(i = 0, 1, 2, \dots, n)$。这些坐标点混合产生下面的位置向量$P(u)$，用来描述$P_0$和$P_n$间的逼近 Bezier 多项式函数的曲线。

$$P(u) = \sum_{i=0}^{n} P_i BEZ_{i,n}(u), u \in [0,1]$$

利用 Bernstein 基函数的降阶公式可知，得出 Bezier 曲线上的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 Bezier 混合函数为：

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

通过上述两个式子，便可以在给出 n+1 个控制顶点的情况下，求出 n 次的 Bezier 多项式函数，从而画出 Bezier 曲线，具体算法步骤请参见源代码。

4.2 B 样条曲线算法：

函数原型: void **draw_curve_Bspline**(QVector<pair<int, int>> ctrlpoints, int precision);

函数功能:按照给定的控制点数组 ctrlpoints，按照规定精度 precision 绘制一条 B 样条曲线。

Bezier 曲线具有许多优点，如凸包性、保型性等，但它也存在不足之处，例如特征多边形的顶点的数量决定了 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接起来的低次曲线，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。

B 样条基函数的定义如下：

给定参数 u 轴上的节点分割$U_{n,k} = \{u_i\}(i = 0, 1, 2, \cdots, n+k)$，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。

deBoox-Cox 的递推公式为：

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

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

那么，如果已知 n+1 个控制顶点$\{P_i\}(i = 0, 1, 2, \cdots, n)$及节点参数向量：

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

则称如下形式的参数曲线$P(u)$为 k 阶（k-1 次）B 样条曲线，

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

在本次程序设计中，我实现的 B 样条曲线为三次均匀 B 样条曲线，根据上述定义，可以得出三次均匀 B 样条曲线段的矩阵表示为：

$$P(u) = \frac{1}{6}[u^3 \quad u^2 \quad u \quad 1]\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}\begin{bmatrix} P_i \\ P_{i+1} \\ P_{i+2} \\ P_{i+3} \end{bmatrix}$$

在给定控制点数组的情况下，我们可以根据上面的公式，取窗口大小为 3，从$[P_0, P_3]$开始依次滑动窗口绘制 B 样条曲线并进行拼接，循环至$[P_{n-3}, P_n]$为止，然后将这些曲线段拼接起来，组成一条完整的 B 样条曲线。

事实上，B 样条曲线相较于 Bezier 曲线的优点就在于局部性和灵活性。挪动一个控制点只会影响部分曲线段的形状，这就使得 B 样条曲线更贴合他的控制多边形，同时也让 B 样条曲线的形状更容易控制。具体实现请参见源代码。

5. 图元平移算法：

函数原型: void **translate**(item* choose, int dx, int dy);

函数功能:将选中的图元 choose 按照平移向量（d_x，d_y）进行平移。

平移算法原理十分简单。根据输入的平移向量，将图元中的每个点进行平移，消除原来的点即可。

6. 图元旋转算法

函数原型: void **rotate**(item* choose, int x, int y, int r);

函数功能:将选中的图元 choose 以(x, y)为中心顺时针旋转 r 度（角度制）。

设旋转中心为(x, y)，要旋转的点为(x_0, y_0)，那么旋转后的点(x', y')的坐标应遵循如下公式：

$$\begin{cases} x' = (x_0 - x)cosr - (y_0 - y)sinr + x \\ y' = (y_0 - y)cosr + (x_0 - x)sinr + y \end{cases}$$

在实现过程中，我将图元的关键点（如直线的两个端点，多边形的顶点，曲线的控制点）按照上述公式进行旋转后，再对图元进行重绘，即可实现旋转功能。

7. 图元缩放算法

函数原型: void **scale**(item* choose, int x, int y, double s);

函数功能:将选中的图元 choose 以(x, y)为中心缩放 s 倍。

设缩放中心为(x, y)，要缩放的点为(x_0, y_0)，那么缩放后的点(x', y')的坐标应遵循如下公式：

$$\begin{cases} x' = (x_0 - x) * s + x_0 \\ y' = (y_0 - y) * s + y_0 \end{cases}$$

在实现过程中，将图元的关键点（如直线的两个端点，多边形的顶点，椭圆的中心等）按照上述公式进行计算，用得到的新的关键点对图元进行重绘，即可实现缩放功能。

注：在缩放过程中，对椭圆的长短半轴r_x和r_y应采取直接$*s$的做法，即：

$$\begin{cases} r_x' = r_x * s \\ r_y' = r_y * s \end{cases}$$

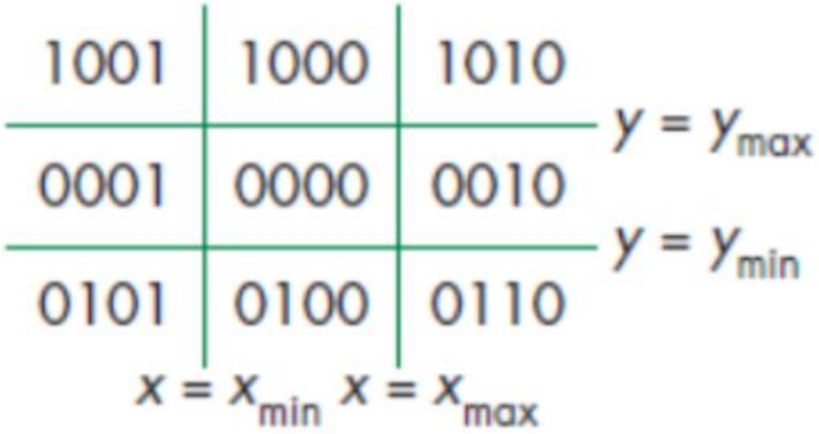
8. 图元裁剪算法

8.1 Cohen-Sutherland 算法：

函数原型: void **clip_Cohen_Sutherland**(item* choose, int x1, int y1, int x2, int y2);

函数功能:对给定的直线图元 choose（注：裁剪只支持直线），将其在左下角为(x_1, y_1)，右上角为(x_2, y_2)的矩形以外的点裁剪掉，只留下在矩形内部的点。

Cohen-Sutherland 算法是一种基于编码和位运算来进行运算的算法，他的思想是通过对于任一端点(x, y)，根据其坐标所在的区域，赋予一个 4 位的二进制码（如图），判断图形元素是否落在裁剪窗口之内并通过求交运算找出其位于内部的部分。



显然，编码为 0000 的点是我们需要保留的点，其他点都是要删除的。在裁剪一条线段时，先求出端点 p1 和 p2 的编码 code1 和 code2：

1）如果 code1 和 code2 均为 0，则说明 P1 和 P2 均在窗口内，那么线段全部位于窗口内部，应取之。

2）如果 code1 和 code2 经过按位与运算后的结果 code1&code2 不等于 0，说明 P1 和 P2 同时在窗口的上方、下方、左方或右方，那么线段全部位于窗口的外部，应弃之。

3）当（1）和（2）不满足时，确保 p1 在窗口外部：若 p1 在窗口内，则交换 p1 和 p2 的坐标值和编码。

4）求出直线段与窗口边界的交点，并用该交点的坐标值替换 p1 的坐标值。也即在交点 s 处把线段一分为二。考虑到 p1 是窗口外的一点，因此可以去掉 p1s 线段。然后回到（2）直到两个端点的编码均为 0，再用对应直线生成算法绘出直线。

8.2 Liang-Barsky 算法：

函数原型: void **clip_Liang_Barsky**(item* choose, int x1, int y1, int x2, int y2);

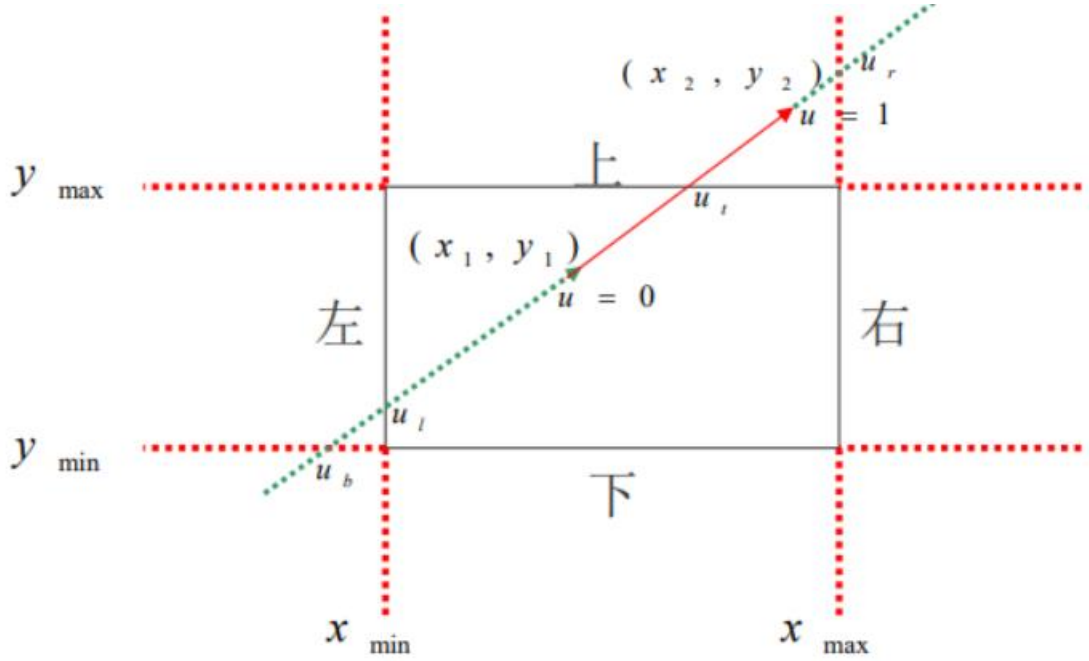
函数功能: 对给定的直线图元 choose，将其在左下角为(x_1, y_1)，右上角为(x_2, y_2)的矩形以外的点裁剪掉，只留下在矩形内部的点。

Liang-Barsky 算法的主要思想基于直线的参数方程，即：

$$\begin{cases} x = x_1 + u * (x_2 - x_1) = x_1 + u * \Delta x \\ y = y_1 + u * (y_2 - y_1) = y_1 + u * \Delta y \end{cases}$$

其中，(x_1, y_1)和(x_2, y_2)为直线的两个端点，$0 \le u \le 1$。

显然，如果我们能够确定一个u的范围，使得直线段在矩形区域内，就相当于完成了裁剪工作，如图：



如果用u_1和u_2来表示参数u的范围，从图中我们可以推导出：

$$u_1 = \max(0, u_b, u_l),\quad u_2 = \min(1, u_r, u_t)$$

所以我们只需要求出直线和矩形边界交点的参数 u 值，然后计算得出u_1和u_2，即可确定参数 u 的范围：$u_1 \le u \le u_2$，而后再根据参数方程对直线进行重绘，就完成了裁剪操作。

七、系统介绍：

目前设计的主窗口界面如图：

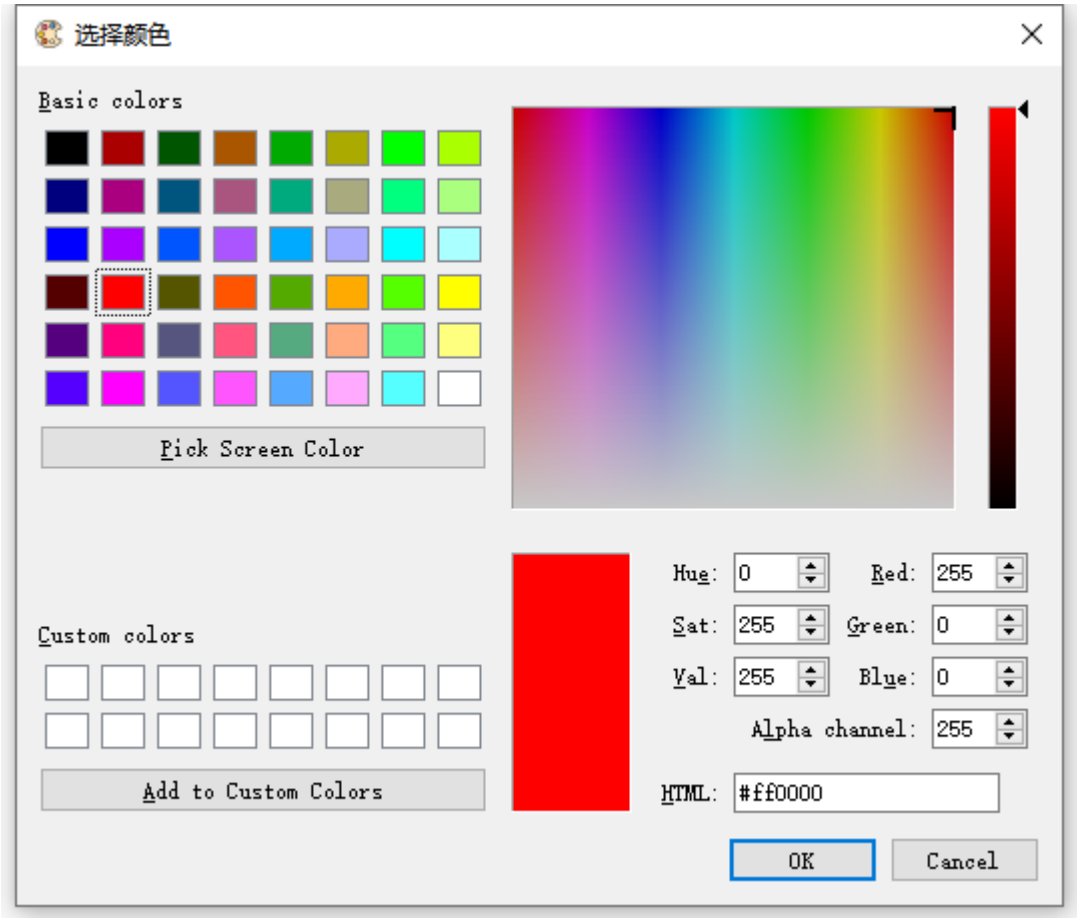
从图上可以清晰地看出，程序主窗口分为四部分，最上面两个部分为菜单栏和工具栏，设计思路参考了我们经常使用的一些绘图软件，包含了程序的几乎全部功能（如图元绘制，变换等操作）。所占比例最大的是绘图窗口部分，用来显示所绘图元，同时支持鼠标交互选取功能。左下角是状态栏，用来显示软件当前状态（如绘制直线，绘制椭圆等）。

菜单栏展开后如下图所示：

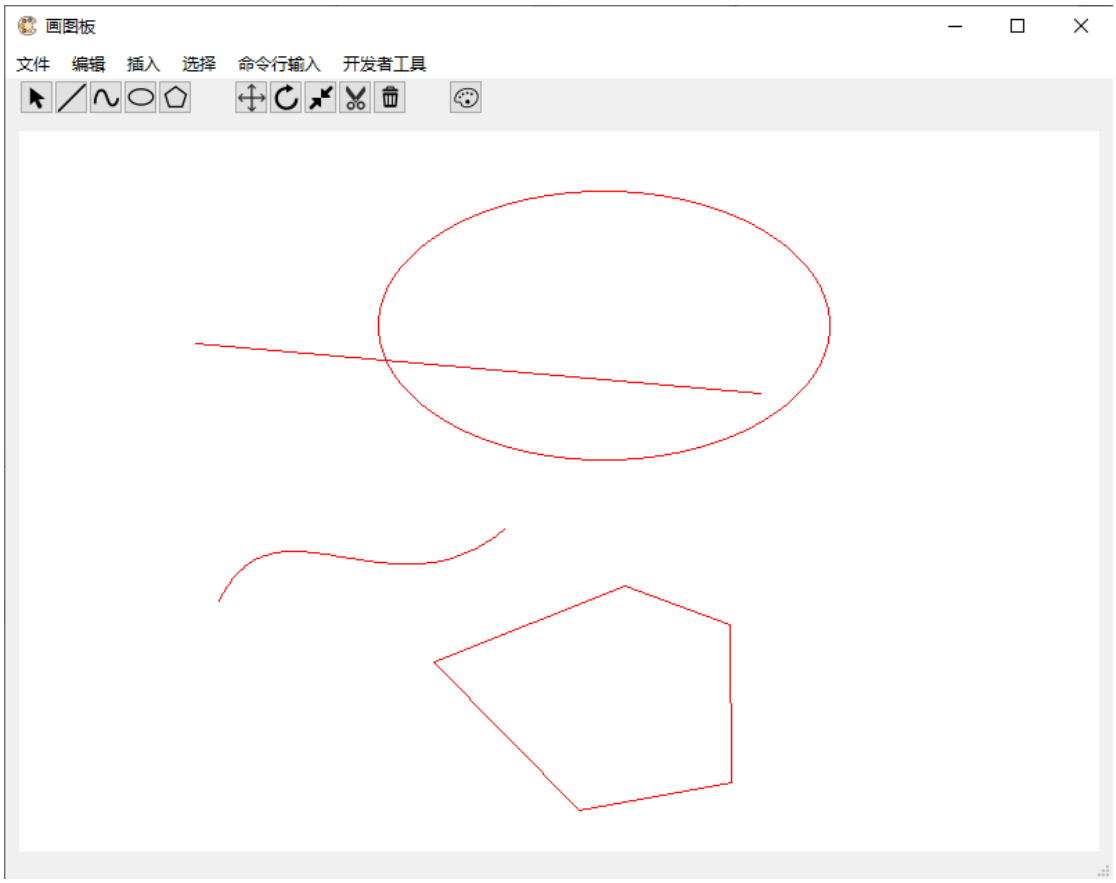
工具栏：

从左到右依次为：选择图元，绘制直线，绘制曲线，绘制椭圆，绘制多边形；移动，旋转，缩放，裁剪，删除；更改画笔颜色。（具体使用方法请参见使用说明书）

例如，我们首先修改一下画笔颜色为红色：

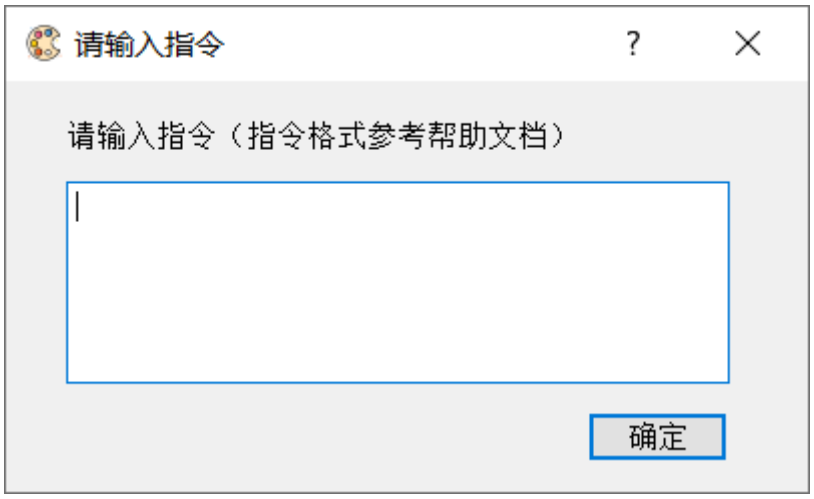


然后再用鼠标画几个图元：

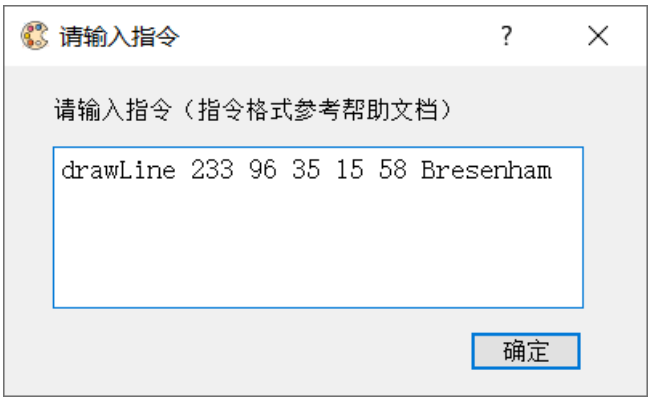


（注：由于鼠标交互功能无法通过截图来展现，故这里不进行详细展示）

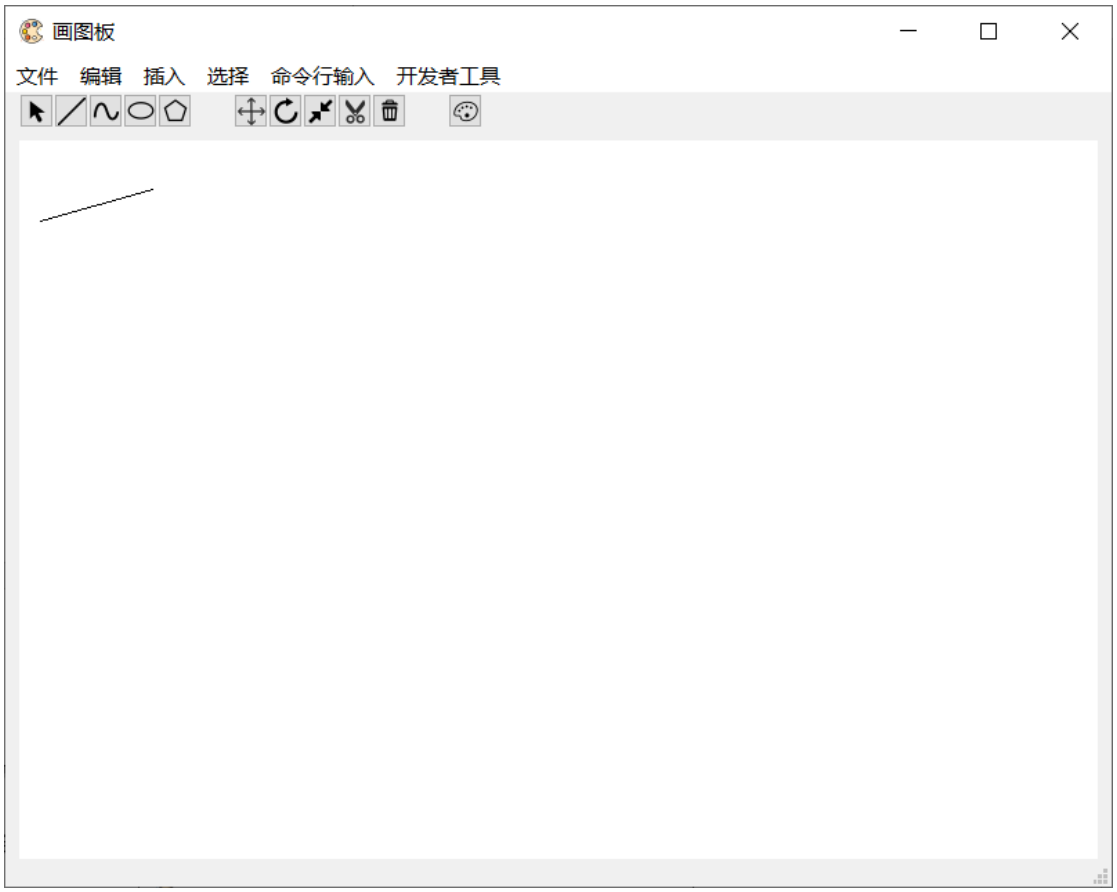
除了使用鼠标进行操作，在 GUI 界面下我也实现了指令绘图的接口，便于更加精准的绘制图元。首先打开菜单栏命令行输入，点击命令行输入->从指令输入，即可看到如下界面：



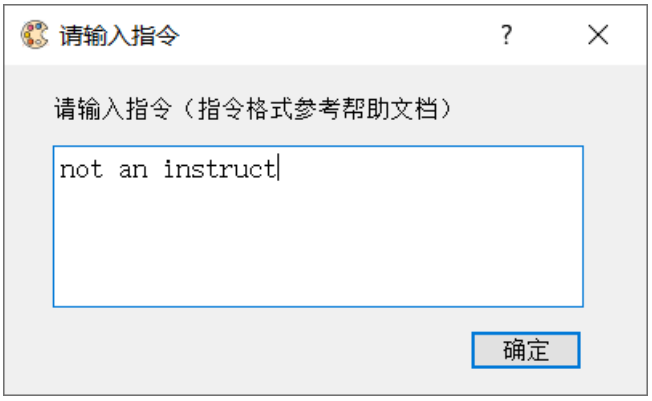
在这里可以输入指令来绘图，例如我们输入指令来绘制一条直线：



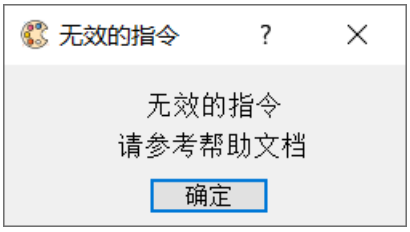
执行结果如图：



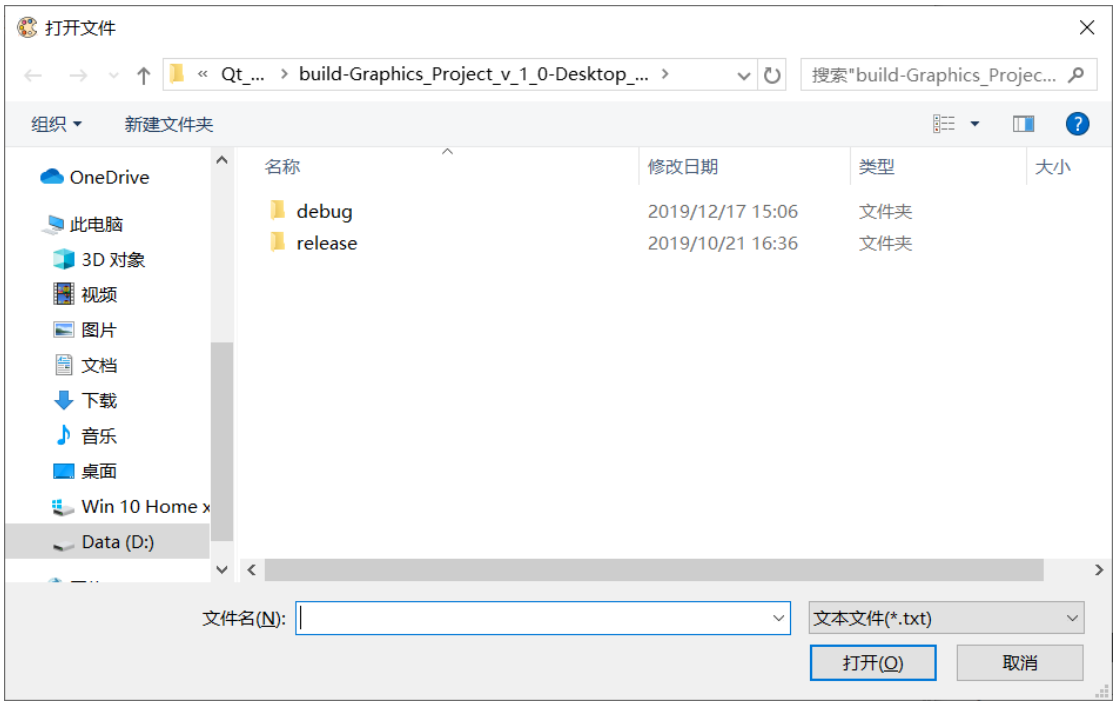
当然也进行了指令无效检查，如果输入一个无效指令：



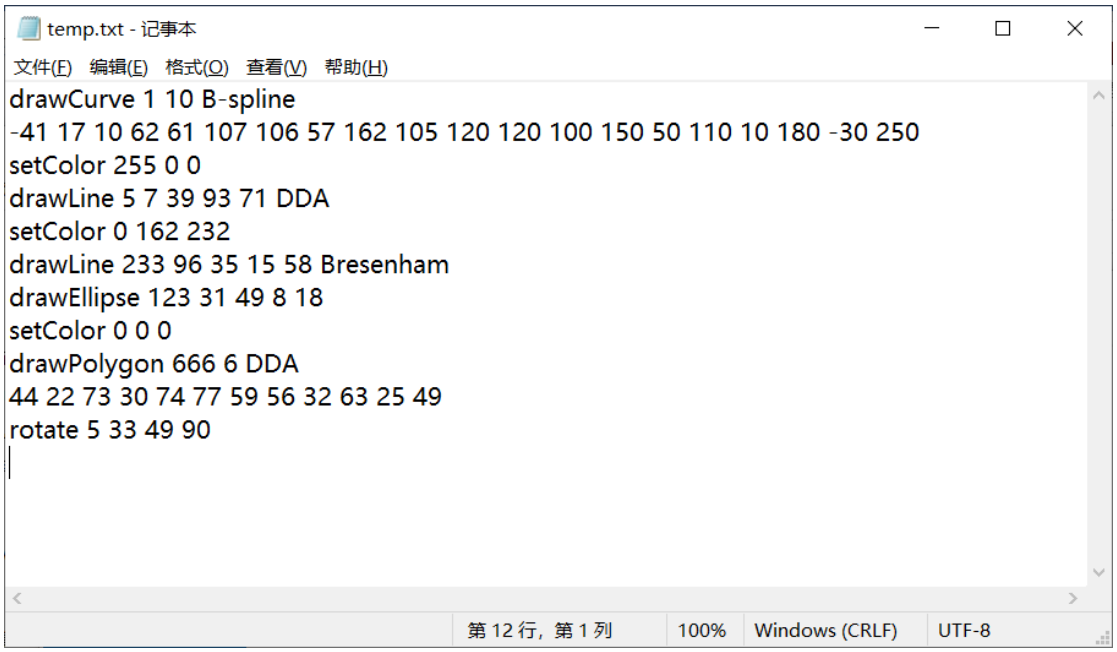
会弹出对应的提示框，提示该指令无效



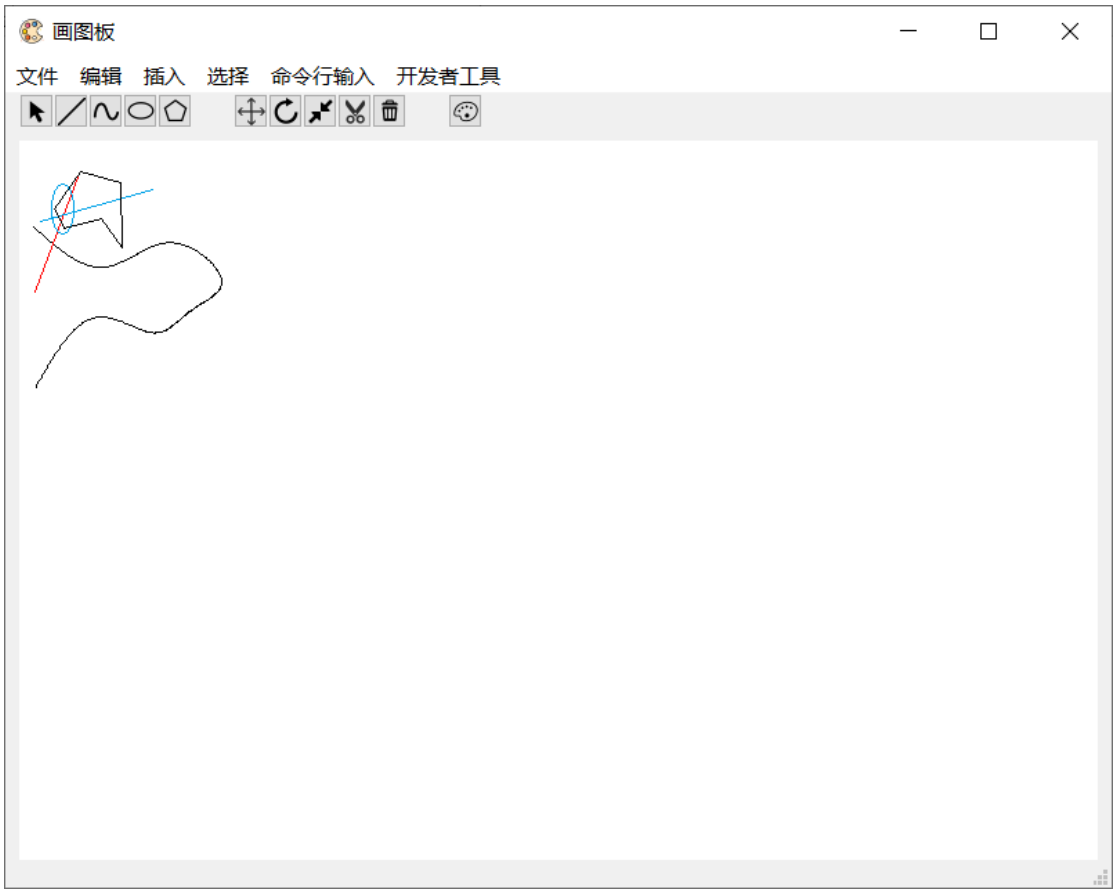
具体指令格式请参见使用说明书。同时除了键盘输入，我还实现了读取指令文件进行绘图的功能，打开命令行输入->从文件输入后，可以看到如下窗口：



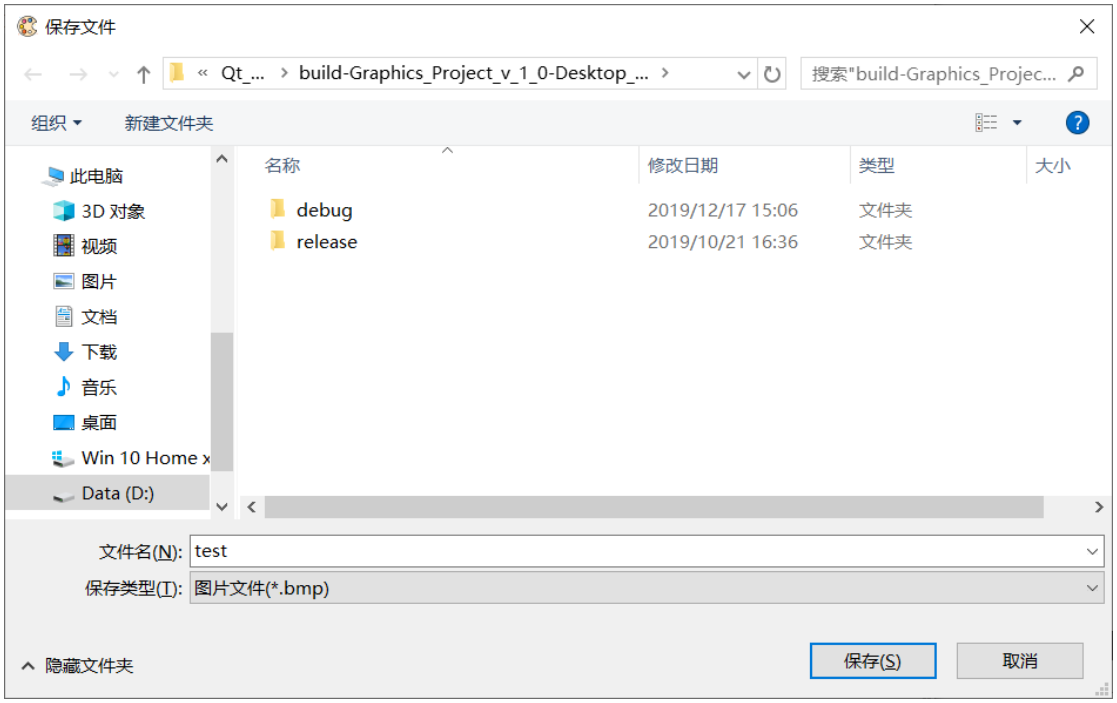
在此窗口下选择一个.txt 文本文件，程序会读取文本文件中的指令，来进行绘图。例如，我写了一个指令文件，指令序列如下：



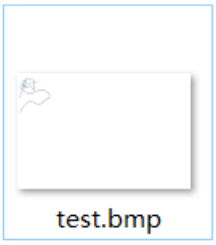
在程序中打开该文件后，可以看到指令得到正确执行，执行结果如图：



点击文件->保存可以将当前画板保存为.bmp 图片文件，这里我们将该画板命名为 test.bmp，保存在程序运行目录下：

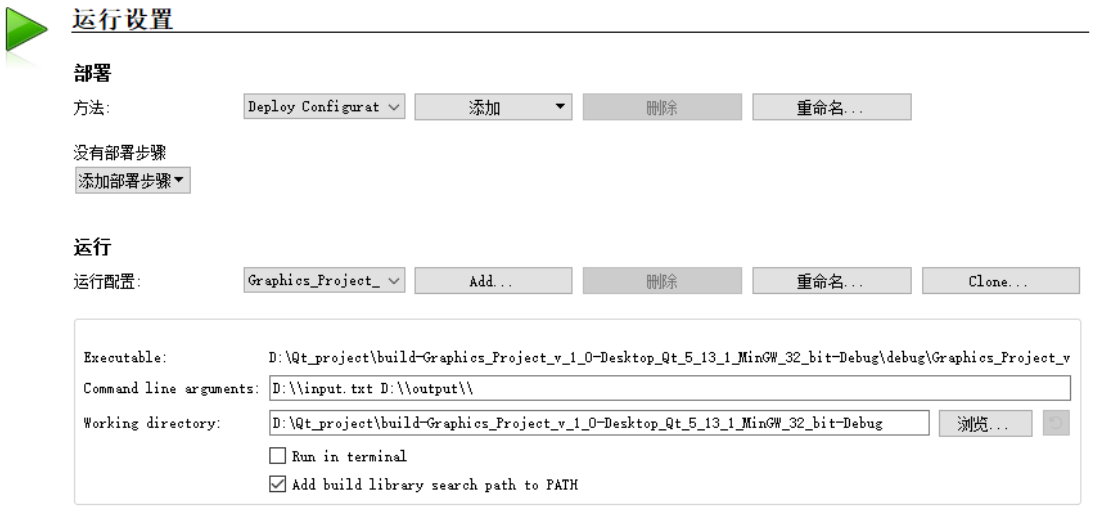


打开程序运行目录，可以看到 test.bmp 被成功保存：

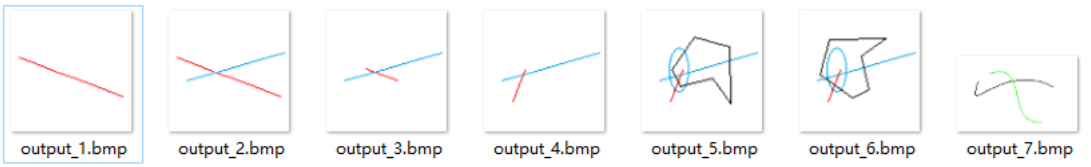


八、程序运行测试：

利用 Qt 自带的命令行添加参数功能，使用提供的样例 input.txt 进行测试



测试结果为：



九、参考资料：

《计算机图形学教程》 孙正兴主编
Qt 官方手册 QVector QString QPainter QPixmap 等内容
数篇 CSDN 博客或博客园博客：
https://www.cnblogs.com/lifexy/p/9203929.html
https://www.cnblogs.com/clairvoyant/p/5540023.html
https://blog.csdn.net/orbit/article/details/7496008
https://blog.csdn.net/qq_32583189/article/details/53018981
https://www.cnblogs.com/jenry/archive/2012/02/12/2347983.html
https://www.cnblogs.com/cnblog-wuran/p/9813841.html
https://blog.csdn.net/xyisv/article/details/83514472

https://www.cnblogs.com/cnblog-wuran/p/9813788.html
其中一篇关于 Bezier 曲线讲述非常形象：https://www.jianshu.com/p/55099e3a2899
其他网页资源：
http://c.biancheng.net/qt/