《计算机图形学》报告

郑凯琳 205220025，205220025@smail.nju.edu.cn

---

1 综述

本实验将最终实现一个完整的图形学系统，能够绘制线段、曲线、多边形和椭圆等基本图形，并支持图元平移、缩放、旋转及线段裁剪等基本操作。

1.1 9 月进度概述

- 配置开发环境
- 完成图元绘制算法
    - 绘制线段
        - DDA 算法
        - Bresenham 算法
        - 对比分析
    - 绘制多边形
    - 绘制椭圆：中点圆生成算法
    - 绘制曲线
        - Bezier 算法
        - B-spline 算法
        - 对比分析

1.2 10 月进度概述

- 完成 CLI 程序：支持图元绘制所需的 API 调用
- DEBUG 图元绘制算法 * 在 CLI 测试时发现错误，重新修正
- 完成图元变换算法
    - 图元平移
- 图元绘制 —— CLI 测试成功！

1.3 11 月进度概述

- 完成图元变换算法

    - 图元旋转
    - 图元缩放
    - 对线段裁剪
        - Cohen-Sutherland 算法
        - Liang-Barsky 算法
- 完成 CLI 程序：支持图元变换所需的 API 调用
- 图元变换 —— CLI 测试成功！
- 理解 GUI 程序，初步设计 GUI 系统

1.4 12 月进度概述

- 对一些算法进行改进
    - 明显结构或逻辑改动：* 用蓝色标注
    - 主要以优化为主：简化代码，避免代码重复，提升可读性
- 搭建 GUI 系统
    - 基本功能
        - 初始功能：重置画布、保存画布、设置画笔颜色
        - 图元绘制：绘制线段、绘制多边形、绘制椭圆、绘制曲线
        - 图元变换：图元平移、图元旋转、图元缩放、对线段裁剪
    - 其他实现细节
        - 定义辅助数据（支持图形变换 & 裁剪操作
        - set_transform_status(self, status, selected_only=False)
        - 鼠标事件处理
        - 定义画布属性
        - 条件判断
    - MyItem 类
    - 拓展功能
        - 为常用的功能添加快捷键
        - 图元删除
        - 图元选择模式
        - 打开画布
- 总结

2 开发环境

Ananconda 安装及配置

Python 版本：Python 3.12.4

- numpy 1.26.4
- pillow 10.3.0
- PyQt5 5.15.10

3 图元绘制算法

3.1 绘制线段

3.1.1 DDA 算法

原理：

DDA（Digital Differential Analyzer）算法是一种增量法绘制直线的算法。该算法通过计算直线的斜率，从起点到终点逐步计算每个像素点的位置。DDA 算法的核心是通过计算 x 或 y 方向的增量，在每一步中使用增量来确定下一个像素点的坐标。

理解 & 代码分析：

1. 特殊情况处理：
   - 对于垂直线（$x_0 = x_1$），直接遍历 y 坐标绘制像素点。
2. 斜率计算：
   - 直线的斜率 k 计算公式为： $k = \frac{y_1 - y_0}{x_1 - x_0}$
3. 主变量选择：
   - 当斜率 $k < 1$ 时，以 x 为主变量进行计算。
   - 当斜率 $k \geq 1$ 时，以 y 为主变量进行计算。
4. 像素点绘制：
   - 通过增量 Δy 或 Δx 逐步更新 y 或 x 的值，绘制出直线上的像素点。
5. 实现细节：
   - 主轴判断：通过计算斜率来确定 x 或 y 作为主轴。
   - 坐标更新：根据斜率或其倒数来更新 y 或 x 的值。

- 方向调整：如果绘制方向不符合预期，则交换起点和终点确保一致性。

3.1.2 Bresenham 算法

原理：

Bresenham 算法是一种高效的基于整数运算的直线绘制算法。它通过判断每一步的误差积累来决定是在主轴方向（x 或 y 轴）上前进一个像素，还是同时在两个方向上前进。该算法仅使用整数加减法和比较操作，避免了 DDA 中的浮点运算。

理解 & 代码分析：

1. 初始化：
   - 计算线段的增量 dx 和 dy，即终点与起点在 x 轴和 y 轴上的距离。
   - 计算斜率 k，通过 $k = \frac{dy}{dx}$ 评估线段的倾斜度。
2. 特殊情况处理：
   - 对于垂直线（$x_0 = x_1$），直接遍历 y 坐标绘制像素点。
3. 主变量选择：
   - 当斜率 $k < 1$ 时，以 x 为主变量进行计算：
     - 通过 p_k进行决策，确定在 x 增加时 y 是否增加。
   - 当斜率 $k \geq 1$ 时，以 y 为主变量进行计算：
     - 使用相同的决策参数 p_k 来更新 x 值。
4. 更新和输出：
   - 在每一步中，将当前计算出的像素点坐标添加到结果列表中。

*** 改进 ***

避免重复逻辑：

- 使用 steep 判断是否为陡直线，通过交换 x 和 y 实现通用逻辑。

更少变量交换：

- 仅在必要时交换坐标和变量，减少冗余代码。

更简洁的 y_step 处理：

- 用 y_step 代替 flag，更直观地控制 y 的增长方向。

3.1.3 对比分析

特性	DDA算法	Bresenham算法
优点	简单直观，易于实现	效率高，适合实时绘图
	适用于任意斜率的直线绘制	通过整数运算实现更精确的直线绘制
缺点	使用浮点运算，速度较慢	理解和实现相对复杂
	可能导致不够精确的绘制	特殊情况下需额外处理
适用场景	简单图形应用，快速原型开发	性能要求高的实时图形系统
选择建议	对性能要求不高时选择	性能要求高时优先考虑

总的来说，Bresenham 算法比 DDA 算法更准确高效。

3.2 绘制多边形

通过逐条绘制多边形的边并将结果累加，利用线段绘制算法（DDA 或 Bresenham）生成各个顶点之间的线段，从而实现多边形的整体绘制。

3.3 绘制椭圆

3.3.1 中点圆生成算法

原理：

该算法通过决策参数的增量更新来选择像素点，进而逼近理想椭圆的边界。中点椭圆生成算法的核心思想类似于 Bresenham 算法，它在每一步根据决策参数的符号，判断如何更新坐标，以保证像素点尽量逼近椭圆轮廓。

椭圆的标准方程为：

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

其中，a 和 b 分别是椭圆的水平半径和垂直半径。

理解 & 代码分析：

第一象限的点生成：x 轴主导

1. 初始化决策参数：初始决策参数 $p_k = b^2 - a^2 \cdot b + \frac{a^2}{4}$，用于判断当前点位置是否偏离椭圆。

2. 循环条件：满足 $b^2 \cdot x < a^2 \cdot y$ 时，以 x 为主导，逐步增加 x 并根据 p_k 的值决定是否同时减少 y。
3. 决策参数更新：
   - 若 $p_k < 0$，表示下一个点仍在或接近椭圆边界，则仅增加 x，更新 p_k 。
   - 若 $p_k \geq 0$ ，则同时增加 x、减少 y，更新 p_k 以逼近椭圆边界。

第二象限的点生成：y 轴主导

1. 初始化决策参数：决策参数为：

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

2. 循环条件：满足 $y > 0$ 时，以 y 为主导，逐步减少 y，并根据决策参数决定是否同时增加 x。
3. 决策参数更新：
   - 若 $p_k > 0$ ，则仅减少 y，更新 p_k 。
   - 若 $p_k \leq 0$，则同时减少 y、增加 x，更新 p_k 以继续逼近椭圆轮廓。

*** 改进 ***

封装对称点处理：

将椭圆四个象限点的生成逻辑封装到 add_symmetric_points 函数中，避免代码重复，提升可读性。

3.4 绘制曲线

3.4.1 Bezier 算法

原理：

Bezier 曲线是一种用于通过多个控制点生成平滑曲线的技术。此曲线具有以下特性：

- 定义简洁：由一组控制点定义曲线形状。
- 平滑连续：通过控制点的权重插值构造，平滑过渡。
- 逐次递推计算：可以通过 De Casteljau 算法实现逐步生成。

Bezier 曲线的生成可以理解为在控制点之间逐次插值，并通过多个插值层逐步逼近最终的曲线上点。在这过程中，每一个 u 值都对应一个曲线点，曲线在 u 从 0 到 1 之间绘制完毕。

理解 & 代码分析：

De Casteljau 算法：

给定 n+1 个控制点 $P_0, P_1, \dots, P_n$，曲线上某个点 $P(u)$ 的位置可以通过如下递归关系式求得：

$$P_i^{(r)}(u) = (1-u) \cdot P_i^{(r-1)}(u) + u \cdot P_{i+1}^{r-1}(u)$$

这里，$P_i(0)$为原始控制点。

1. 初始化：
   - result 存储生成的贝塞尔曲线上各点。
   - n 为控制点个数减 1，用于循环递归插值。
   - u 控制曲线在 0 到 1 之间的分布步长。
2. 递归插值计算：
   - 对于每个步长 u，初始化 res 为控制点列表，执行 n 轮插值。
   - 在每一轮插值中，计算控制点之间的中间点 (1-u) * x1 + u * x2 以生成下一轮控制点列表。
   - 循环结束后，res[0] 即为当前 u 值下的 Bezier 曲线上点。
3. 生成曲线点：
   - 将每个得到的点四舍五入（round）以得到屏幕上的离散点，并加入 result。
   - 最后，返回整条曲线的点列表 result。

3.4.2 B-spline 算法

原理：

B-spline 曲线是一种分段定义的曲线，支持局部控制，常用于建模复杂曲线。此算法不要求曲线经过控制点，而是通过递推方法，逐步利用控制点及其对曲线上点的权重来构建最终的曲线。此曲线具有以下优点：

- 局部控制：对曲线的形状影响主要由附近的控制点决定，便于调整。
- 平滑性：B-Spline 曲线在参数空间上具有连续性，生成的曲线平滑。

- 高效计算：通过 De Boor-Cox 基函数的递归定义，计算效率高。

理解 & 代码分析：

De Boor-Cox 公式：

利用递归的方法计算基函数，并在给定的参数 u 处生成 B-Spline 曲线的点。

B-Spline 基函数 $N_{i,k}(u)$ 的计算公式如下：

- 当 $k = 1$ 时，

$$N_{i,1}(u) = \begin{cases} 1 & \text{if } u_i \leq u < u_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

- 当 $k > 1$ 时，

$$N_{i,k}(u) = \frac{u - u_i}{u_{i+k-1} - u_i} N_{i,k-1}(u) + \frac{u_{i+k} - u}{u_{i+k} - u_{i+1}} N_{i+1,k-1}(u)$$

这里，$N_{i,k}(u)$ 是在参数 u 下控制 i 生成的基函数，u_i 是控制点对应的节点值。

1. B-Spline 曲线生成：
   - 函数 bspline_curve 接收控制点列表 p_list，生成 B-Spline 曲线。
   - 曲线通过迭代计算不同 u 值下的控制点影响，以获取平滑的曲线点。
2. 基函数计算：
   - de_boor_cox 函数实现 De Boor-Cox 算法，采用递归方法计算 B-Spline 的基函数值。
   - 基函数值用于确定每个控制点在曲线生成中的权重，从而影响曲线的形状。

   实现细节 → 使用非递归的方式实现 de_boor_cox，将递归公式展开为动态规划的形式，从而避免递归带来的函数调用开销。这种方式通过一个二维数组保存计算的中间结果，提高了计算效率。
3. 局部控制与平滑性：
   - B-Spline 的局部控制特性使得只需调整相邻控制点即可影响曲线形状。
   - 通过细化 u 的步长（du），可以提高曲线的平滑度。

3.4.3 对比分析

1. **定义与构造方式**

Bezier 曲线：

- 由一组控制点直接定义。每条 Bezier 曲线对应一个特定的控制点集合。
- 生成方式主要依赖 De Casteljau 算法，通过逐次插值控制点得到曲线点。

B-spline 曲线：

- 由多个控制点和节点值定义，曲线的生成与这些节点密切相关。
- 不要求经过所有控制点，而是通过递推方法和基函数加权生成曲线，支持局部控制。

2. 控制特性

局部控制：

- Bezier：全局控制，调整任何一个控制点都会影响整个曲线的形状。
- B-spline：局部控制，调整某个控制点仅影响与之相邻的部分，便于局部调整。

曲线的光滑性：

- Bezier：在给定的控制点数目内，光滑性由曲线的阶数决定，但其形状较难微调。
- B-spline：可以通过增加控制点数目和细化节点值，确保曲线光滑性和连续性。

3. 计算效率

Bezier：

- 通常在控制点数较少的情况下表现良好，但随着控制点的增加，计算复杂度会显著上升。
- De Casteljau 算法的时间复杂度为 O(n^2)，其中 n 为控制点数量。

B-spline：

- 采用 De Boor-Cox 算法，具有更高的计算效率。基函数的递归定义能有效减少计算时间。
- 利用动态规划实现基函数的非递归计算，进一步提升性能。

4. 应用场景

Bezier 曲线：

- 广泛应用于计算机图形学、动画和工业设计中，适合用于简单的曲线设计和小范围内的变化。

- 由于其全局控制特性，更加适用于需要精确控制整体形状的场合。

B-spline 曲线：

- 常用于复杂形状建模、CAD/CAM 领域以及大规模曲线编辑。
- 适合需要频繁局部调整的场合，能够更好地处理复杂曲线和表面。

5. 视觉表现

Bezier 曲线：

- 可以通过较少的控制点快速生成曲线，简单直观，但在控制点较多时难以把握曲线的全局形状。

B-spline 曲线：

- 通过多控制点和节点的组合，可以生成更复杂的形状，曲线更平滑，但需要更多的参数设置。

总结：对于简单、少量控制点的应用，Bezier 算法更易于理解和实现。对于复杂的建模和需要局部控制的情况，B-spline 算法则表现得更加灵活和高效。

4 图元变换算法

4.1 图元平移

原理：

平移变换是一种基本的几何变换，用于在平面上移动图形或图元的每一个点。平移不改变图形的形状或大小，仅通过在水平和垂直方向上添加固定的偏移量（dx 和 dy）来改变其位置。平移变换的数学表示可以用以下公式描述：

对于图元中的每一个点 (x, y)，经过平移变换后的新坐标为：

$$(x', y') = (x + dx, y + dy)$$

其中：

- dx 是水平平移量，决定了图元在水平方向上的移动。
- dy 是垂直平移量，决定了图元在垂直方向上的移动。

理解 & 代码分析：

1. 逐一遍历每一个点 p，包含点的 x 和 y 坐标。

2. result.append([p[0] + dx, p[1] + dy])：将当前点的 x 坐标加上水平平移量 dx，y 坐标加上垂直平移量 dy，计算出平移后的新坐标，将其添加到 result 列表中。

4.2 图元旋转

原理：

在二维平面中，图形的旋转变换可以通过矩阵运算实现。给定旋转中心 (x, y) 和顺时针旋转角度 r，可以将图元点集相对于该中心旋转 r 度。

1. **旋转矩阵：** 旋转矩阵用于表示点绕原点 (0, 0) 旋转的变换，公式如下：

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

其中：

- θ 是旋转角度。
- (x, y) 是待旋转点的初始坐标。
- (x', y') 是旋转后的坐标。

2. **平移与旋转组合：** 要绕任意点 (x, y) 旋转，首先将旋转中心平移至原点，再进行旋转变换，最后将其平移回原位置：

(1) **平移**：将待旋转的每个点相对于旋转中心转换为局部坐标。

(2) **旋转**：应用旋转矩阵。

(3) **反向平移**：将结果坐标平移回全局坐标系。

理解 & 代码分析：

输入参数：

- p_list：要旋转的点集，每个点以 [x, y] 表示。
- x、y：旋转中心的坐标。
- r：旋转角度（顺时针为正，逆时针为负）。
- unit：角度单位。True 表示度数，False 表示弧度。

实现步骤：

1. 判断角度单位：
    - 如果 unit=True，将角度 r 转换为弧度。
    - 如果 unit=False，直接使用 r。
2. 对每个点 [p[0], p[1]]：

- 平移到局部坐标系：计算相对坐标 x0 = p[0] - x 和 y0 = p[1] - y。
- 应用旋转公式：计算旋转后的坐标 x_new 和 y_new。
- 恢复全局坐标系：将 x_new 和 y_new 平移回原坐标系。

输出：

返回旋转后的点集，每个点为 [x', y']。

4.3 图元缩放

原理：

对于任意一个点 (x_0, y_0) 相对于缩放中心 (x, y) 的缩放变换公式为：

$$x' = (x_0 - x) \times s + x$$
$$y' = (y_0 - y) \times s + y$$

其中：

- (x_0, y_0) 是待变换点的坐标。
- (x, y) 是缩放中心。
- s 是缩放倍数。
- (x', y') 是缩放后的坐标。

理解 & 代码分析：

1. 首先将待变换点平移，使得缩放中心成为坐标原点，即 $(x_0 - x, y_0 - y)$ 。
2. 对平移后的坐标应用缩放倍数 s 。
3. 最后将缩放后的坐标平移回原坐标系。

4.4 对线段裁剪

4.4.1 Cohen-Sutherland 算法

原理：

一种基于编码的裁剪算法，通过为每个点分配一个“区域编码”来判断线段在裁剪窗口内的情况。

理解 & 代码分析：

1. **区域编码**：

- 为每个点分配一个 4 位的二进制编码，编码的每一位代表线段端点的位置关系（上下左右）：

- **左**：0001（最右一位为 1，表示点在窗口左侧之外）
- **右**：0010（倒数第二位为 1，表示点在窗口右侧之外）
- **下**：0100（倒数第三位为 1，表示点在窗口下侧之外）
- **上**：1000（最高位为 1，表示点在窗口上侧之外）

这样，窗口外的每个点都可以用 4 位编码来表示其在窗口外的具体方向。例如，如果点的编码为 1001，表示这个点在窗口左上方。

2. **判断规则**：

- **完全可见**：如果两个端点的编码按位 **或** 为 0，说明两个端点都在窗口内，无需裁剪。
- **完全不可见**：如果两个端点的编码按位 **与** 不为 0，说明线段在窗口外，无需绘制。
- **部分可见**：如果上述两种情况都不满足，则线段可能与窗口的边缘相交，需对其进行裁剪。

3. **裁剪过程**：

- 若线段部分可见，将位于窗口外的端点沿着对应方向的窗口边界向内裁剪，直到端点进入窗口内。该过程会不断更新端点的坐标和编码。
- 重复以上操作，直到线段符合完全可见或完全不可见的条件为止。

4.4.2 Liang-Barsky 算法

原理：

一种基于参数化表示的裁剪算法，与 Cohen-Sutherland 不同，它通过数学计算直接得出裁剪后的端点。其特点是利用参数化表达式将线段转换为参数形式，然后通过裁剪窗口的边界来限制参数范围。

理解 & 代码分析：

1. **参数化表示**：

- 将线段表示为参数化方程：

$$x = x_0 + u \cdot (x_1 - x_0)$$
$$y = y_0 + u \cdot (y_1 - y_0)$$

其中，u 是一个参数，范围为 [0, 1]。u=0 对应线段起点，u=1 对应线段终点，0 < u < 1 则为线段上的某一点。

2. **计算边界条件**：

- 定义两个列表 p 和 q，其中 p 表示线段方向向量的各个分量，q 表示线段与窗口边界的距离关系：
    - 如果 p[i] < 0，则线段穿过窗口外边界，参数 u0 需要增大。
    - 如果 p[i] > 0，则线段在窗口内，参数 u1 需要减小。
- 对每个边界条件进行检查，依次更新参数 u0 和 u1。

3. **裁剪判定**：

- 如果最终得到的 u0 > u1，说明线段完全在窗口外，返回空结果。
- 否则，将 u0 和 u1 的值代入参数方程，计算得到新的裁剪后的线段起点和终点坐标。

4. **计算交点**：

- 使用更新后的 u0 和 u1 计算裁剪后的线段端点，即新裁剪窗口内的线段起点和终点。

*** 改进 ***

→ 使用 clip_param 函数抽象参数裁剪逻辑，避免重复代码。

→ 引入 dx 和 dy 缩短变量名，减少公式冗余。

→ 将四个裁剪边界逻辑合并，逻辑紧凑且可读性更高。

4.4.3 对比分析

算法	适用场景	优点	缺点
Cohen-Sutherland	适用于简单裁剪、快速判断	简单易实现	需要多次迭代，效率较低
Liang-Barsky	适用于复杂的裁剪、效率较高	参数化处理，效率高	逻辑复杂度较高，不如Cohen直观

Cohen-Sutherland 更适合简单的场景，Liang-Barsky 则更加适合追求效率的场景。

5 命令行界面（CLI）程序

实现原理：

1．通过读取和解析输入文件中的指令：

逐行读取输入文件（readline()），根据不同的指令类型执行相应的操作。每行指令都用空格分隔（line.strip().split(' ')），并存储在 line 列表中。

2．生成图形数据：

调用算法库（cg_algorithms.py）中的对应图元绘制/变换函数生成新的坐标集（p_list），并存入 item_dict。

3．通过 PIL 库保存为图像文件：待绘制点集（pixels）生成，保存为.bmp 文件。

实现细节：在处理图形绘制逻辑的部分，

- 添加了 print('XXX drawn')，XXX 为 LINE、POLYGON、ELLIPSE 和 CURVE，作为调试辅助。
- 根据 Pillow 版本而定，为确保最终输出的视觉结果以画布左上角为坐标原点，将原有的 canvas[height – 1 – y, x] 修改为 canvas[y, x]。

6 用户交互界面（GUI）程序

6.1 基本功能

6.1.1 初始功能

- **重置画布 reset_canvas_action()**

实现细节：

- 首先通过对话框让用户输入新的画布宽度和高度，确保输入值在 100 到 1000 之间。
- 清除图元列表控件 self.list_widget 的所有项。
- 清除画布 self.canvas_widget 上的所有图元对象，清空 self.canvas_widget.item_dict 字典。
- 调用 self.scene.clear() 清除场景中的所有图元。
- 重置图元计数器 self.item_cnt 和画布修改状态 self.isModified。
- 调整场景矩形大小 self.scene.setSceneRect(0, 0, self.width, self.height) 和画布尺寸 self.canvas_widget.setFixedSize(self.width, self.height)。

- **保存画布 save_canvas_action()**

实现细节：

- 构建一个包含所有图元信息的列表 save_list，每个图元的信息包括 ID、类型、参数、算法和颜色。

- 如果没有打开文件，弹出文件保存对话框让用户选择保存路径，然后使用 pickle.dump() 将 save_list 序列化并保存到文件。
- 如果已经打开文件，则直接将 save_list 序列化并保存到原文件。

- **设置画笔颜色 set_pen_action()**

实现细节：

- 弹出颜色选择对话框 QColorDialog.getColor()。
- 如果用户选择了颜色，将画布的临时画笔颜 self.canvas_widget.temp_color 更新为用户选择的颜色。

6.1.2 图元绘制

- **绘制线段 start_draw_line()**

实现原理：

绘制线段功能允许用户在画布上绘制线段。这通常涉及到设置画布状态为绘制线段，并使用特定的算法来绘制线段。

实现细节：

- 设置画布状态为 'line'，并存储算法名称和图元 ID。
- 在 mousePressEvent() 中，根据状态 'line' 创建线段图元 MyItem，并添加到场景中。
- 在 mouseMoveEvent() 中，更新线段图元的终点。
- 在 mouseReleaseEvent() 中，将临时线段图元添加到 self.item_dict 中，并更新图元列表控件。

- **绘制多边形 start_draw_polygon()**

实现原理：

绘制多边形功能允许用户在画布上绘制多边形。这通常涉及到设置画布状态为绘制多边形，并使用特定的算法来绘制多边形。

实现细节：

- 设置画布状态为 'polygon'，并存储算法名称和图元 ID。
- 在 mousePressEvent() 和 mouseMoveEvent() 中，添加多边形的顶点。
- 在 mouseReleaseEvent() 中，完成多边形的绘制，将其添加到 self.item_dict 和图元列表控件。

- **绘制椭圆 start_draw_ellipse()**

实现原理：

绘制椭圆功能允许用户在画布上绘制椭圆。中点圆生成算法是一种基于中点的算法，用于确定椭圆上的点。

实现细节：

- 设置画布状态为 'ellipse'，并存储图元 ID。
- 在 mousePressEvent() 和 mouseMoveEvent() 中，根据鼠标位置确定椭圆的中心和半径。
- 在 mouseReleaseEvent() 中，完成椭圆的绘制，将其添加到 self.item_dict 和图元列表控件。

- **绘制曲线 start_draw_curve()**

实现原理：

绘制曲线功能允许用户在画布上绘制曲线。这通常涉及到设置画布状态为绘制曲线，并使用特定的算法（如 Bezier 或 B-spline）来绘制曲线。

实现细节：

- 设置画布状态为 'curve'，并存储算法名称和图元 ID。
- 在 mousePressEvent() 中，创建曲线图元 MyItem，并添加到场景中。
- 在 mouseMoveEvent() 中，更新曲线的控制点。
- 在 mouseReleaseEvent() 中，完成曲线的绘制，将其添加到 self.item_dict 和图元列表控件。

6.1.3 图元变换

- **图元平移 start_translate()**

实现细节：

- 设置画布状态为 'translate'，并处理平移操作。
- 在 mousePressEvent() 中，记录平移的起始位置。
- 在 mouseMoveEvent() 中，计算平移的距离，并更新图元的位置。

- **图元旋转 start_rotate()**

实现细节：

- 设置画布状态为 'rotate'，并处理旋转操作。
- 在 mousePressEvent() 中，记录旋转的起始位置和旋转中心。

- 在 mouseMoveEvent() 中，计算旋转角度，并更新图元的旋转。在 MainWindow 类中连接信号和槽函数，实现这些动作。

- **图元缩放 start_scale()**

实现细节：

- 设置画布状态为 'scale'，并处理缩放操作。
- 在 mousePressEvent() 中，记录缩放的起始位置和缩放中心。
- 在 mouseMoveEvent() 中，计算缩放比例，并更新图元的大小。

- **对线段裁剪 start_clip()**

实现细节：

- 设置画布状态为 'clip'，并存储裁剪算法名称。
- 在 mousePressEvent() 和 mouseMoveEvent() 中，确定裁剪区域。
- 在 mouseReleaseEvent() 中，调用裁剪算法 alg.clip() 对线段进行裁剪，并更新线段图元。

最后，在 MainWindow 类中连接信号和槽函数，实现这些动作。

6.2 其他实现细节

1. 定义辅助数据，支持后续的图形变换和裁剪操作

- self.origin_p_list = None
- 用于存储当前选中图元的原始点列表（即图元的坐标点）。
- 在进行图形变换（如平移、旋转、缩放）时，程序需要知道图元的原始位置，以便计算变换后的新位置。

- self.origin_pos = None
- 用于存储用户在进行变换操作（如平移、旋转、缩放）时的起始位置。
- 这个位置通常是用户在画布上点击的点，作为变换的参考点。
- 在平移操作中，origin_pos 表示用户点击的起始位置；在旋转和缩放操作中，它用于计算相对于该点的变换效果。

- self.trans_center = None
- 用于存储变换的中心点，通常是用户希望围绕其进行旋转或缩放的点。
- 在进行旋转时，trans_center 是旋转的中心点；在缩放时，它是缩放的中心点。

- 这个变量的存在使得变换操作更加灵活，用户可以选择任意点作为变换的中心，而不仅仅是图元的中心。

- self.border = None
  - 用于存储在裁剪操作中显示的裁剪框的图形对象。
  - 当用户进行裁剪操作时，border 可以用来可视化裁剪区域，帮助用户直观地理解裁剪效果。
  - border 是一个 QGraphicsRectItem 对象，表示裁剪框的矩形区域。它的大小和位置会根据用户的鼠标操作动态更新。

2. set_transform_status(self, status, selected_only=False)

此函数用于设置画布的当前变换状态。

- status 参数指定变换的类型，可以是 'translate'、'scale'、'rotate'、 'clip'。
- selected_only 参数决定是否只对当前选中的图元进行变换，默认为 False，即对所有图元进行变换。
- 函数内部首先将画布的 status 设置为传入的 status。
- temp_item 用于临时存储当前正在变换的图元对象，初始设为 None。
  - 如果 selected_only 为 True 且有图元被选中（self.selected_id 不为空），则 temp_item 会被设置为选中图元的对象。
  - 如果没有图元被选中，或者 selected_only 为 False，则 temp_item 会被设置为字典 self.item_dict 中的第一个图元对象。

当启动各个图元变换函数时，将 status 设置为其变换类型，同时设置 selected_only 为 True，表示平移操作只针对当前选中的图元。

3. 鼠标事件处理

- **mousePressEvent(self, event: QMouseEvent) -> None**

功能：

处理鼠标按下事件，根据当前的画布状态（self.status）执行不同的操作。

实现细节：

- 坐标转换： 获取鼠标点击位置，并将其从窗口坐标转换为场景坐标。
- 图元创建： 如果状态是绘制线段、多边形或椭圆，创建一个新的图元对象 MyItem，并添加到场景中。

- 图元更新： 如果状态是绘制曲线，检查是否已有临时图元对象，如果没有则创建，然后添加新的点到曲线的点列表中。
- 选择模式： 如果状态是选择模式，检查点击的位置是否有图元，如果有，则更新选中图元的状态，并在列表控件中高亮显示。
- 变换操作： 如果状态是平移、旋转、缩放或裁剪，记录选中图元的原始点列表，并根据需要记录变换的起始位置或中心点。

- **mouseMoveEvent(self, event: QMouseEvent) -> None**

功能：处理鼠标移动事件，动态更新图元的位置或变换状态。

实现细节：

- 坐标转换： 获取鼠标移动位置，并将其从窗口坐标转换为场景坐标。
- 图元更新： 根据当前状态（线段、椭圆、多边形、曲线），更新图元的点列表。
- 平移： 如果状态是平移，计算位移并应用平移算法。
- 旋转： 如果状态是旋转，计算旋转角度并应用旋转算法。
- 缩放： 如果状态是缩放，计算缩放比例并应用缩放算法。
- 裁剪： 如果状态是裁剪，动态更新裁剪框的位置和大小。

其中，

➔ calculate_rotation(self, current_x, current_y)

向量计算： 计算起始点和当前点相对于旋转中心的向量。

角度计算： 利用向量的正弦和余弦值计算旋转角度。

➔ calculate_scale_factor(self, current_x, current_y)

距离计算： 计算起始点和当前点到缩放中心的距离。

比例计算： 计算并返回缩放比例。

- **mouseReleaseEvent(self, event: QMouseEvent) -> None**

功能：处理鼠标释放事件，完成图元的绘制或变换操作。

实现细节：

- 图元完成： 根据状态完成图元的绘制，并将图元添加到图元字典和列表控件中。
- 裁剪完成： 如果状态是裁剪，处理裁剪事件并更新图元。

其中，

➔ handle_clip_event(self, x, y)

坐标计算： 计算裁剪区域的最小和最大坐标。

裁剪算法： 应用裁剪算法，获取裁剪后有效的点。

图元更新： 如果裁剪后没有有效的点，隐藏图元并清空点列表；否则，更新图元为裁剪后的点列表，并确保其可见。

裁剪框移除： 移除裁剪框。

4. 定义画布属性

- self.width = 600 && self.height = 600
  - 设置画布的初始宽度及高度为 600 像素。
- self.isModified = False
  - 用于跟踪画布是否被修改过。
  - 当用户进行任何修改（如绘制图形、变换操作等）时，这个值会被设置为 True。
  - 这个标志通常用于决定是否需要在用户关闭程序或打开新文件时提示保存更改。
- self.opened_filename = ' '
  - 用于存储当前打开的文件的路径或名称。
  - 当用户打开一个文件时，这个值会被更新为文件的路径或名称。
  - 这个属性有助于程序跟踪当前操作的文件，并在保存或另存为时提供上下文。

5. 条件判断

```
if self.canvas_widget.status == 'polygon' or self.canvas_widget.status == 'curve':
    self.canvas_widget.finish_draw()
```

为了防止在用户已经开始绘制一个多边形或曲线但尚未完成时，又尝试开始一个新的操作（如绘制另一个曲线、进行变换或裁剪等）。

这样的设计可以确保用户在切换操作之前完成当前的绘制，避免出现**不完整的图形**和**不一致的状态**。

6.3 MyItem 类

绘制方法 def paint

功能：定义如何绘制图元。

实现细节：

- 根据图元类型，使用不同的绘制算法 `alg.draw_line`、`alg.draw_polygon`、`alg.draw_ellipse` 或 `alg.draw_curve` 来获取图元的像素点。
- 遍历像素点列表，使用 `painter.drawPoint(*p)` 绘制每个点。
- 如果图元被选中，绘制一个红色的边界框，使用 `painter.setPen(QColor(255, 0, 0))` 设置画笔颜色，并调用 `painter.drawRect(self.boundingRect())` 绘制矩形。

边界框计算方法 def boundingRect

功能：计算并返回图元的边界框，即包围图元的最小矩形区域。

实现细节：

- 定义了一个内部辅助函数 `calculate_bounding_box(p_list)` 来计算点列表的最小和最大坐标值。
- 根据图元类型，使用不同的方法计算边界框：
    - 对于线段和椭圆，直接使用两个端点坐标计算边界框。
    - 对于多边形和曲线，使用 `calculate_bounding_box` 函数计算边界框。
- 返回一个 `QRectF` 对象，表示图元的边界框，坐标为 `(x_min - 1, y_min - 1, w + 2, h + 2)`，其中 `w` 和 `h` 分别为边界框的宽度和高度。

6.4 拓展功能

6.4.1 为常用的功能添加快捷键

通过过 QKeySequence 类设置快捷键，让用户可以通过键盘快速触发相应的动作。

例："Ctrl+P"（设置画笔）、"Ctrl+R"（重置画布）、"Ctrl+O"（打开画布）、"Ctrl+S"（保存画布）、"Ctrl+X"（退出）

6.4.2 图元删除

start_delete 的功能是删除当前在画布上选中的图元。

1. 检查选中状态：如果没有图元被选中，则直接退出方法。
2. 标记修改：设置画布状态为已修改。
3. 移除图元：从画布的场景中移除选中的图元，并更新内部数据结构。

- 清除场景中的图形对象和内部引用：
    - 调用 self.clear_selection 方法清除当前的选中状态。

    - 调用 self.scene().removeItem(self.temp_item) 从场景中移除图元对象。
    - 从 self.item_dict 字典中删除与 self.selected_id 对应的条目。
    - 将 self.temp_item 设置为 None，清除对该图元的引用。
- 从列表控件中移除对应项：
    - 使用 self.list_widget.findItems(temp_id, Qt.MatchContains) 在列表控件中查找与被删除图元 ID 匹配的项。
    - 如果找到了匹配项，使用 self.list_widget.takeItem 方法从列表控件中移除该项。

4. 更新界面：从列表控件中移除对应的图元项，并刷新画布显示。

然后，在连接 MainWindow 类中的信号和槽函数，启动功能。

6.4.3 图元选择模式

start_select

功能： 设置画布状态为“选择模式”。

实现： 将画布的当前状态（self.status）设置为 'selecting'，表示用户可以开始选择图元。

同时，在**鼠标点击事件处理**中，也实现了相关功能。

代码会检查鼠标点击的位置下是否有图元。

- 如果有选中的图元，并且之前有其他图元被选中，那么先取消之前图元的选中状态。
- 然后更新当前被点击图元的选中状态，并在图元列表控件中高亮显示对应的项。

选择模式提供了一种**直观的方式**来**交互和操作图形界面中的图元对象**。

6.4.4 打开画布 open_canvas_action()

用于打开并加载已保存的画布文件，允许用户继续编辑之前的工作。

实现细节：

- 如果画布当前状态是绘制多边形或曲线，调用 finish_draw 方法结束当前的绘制操作。
- 提示保存更改：
    - 检查画布是否已被修改且没有打开文件，或画布已被修改。

    - 如果需要，弹出消息框询问用户是否保存更改。
    - 如果用户选择保存，则调用 `save_canvas_action` 方法保存画布。
    - 如果用户选择取消，则终止打开画布的操作。
- 调用 reset_canvas_action 方法重置画布，但不调整尺寸。
- 弹出文件选择对话框，让用户选择要打开的画布文件（仅限 .bmp 文件）。
- 加载文件：如果用户选择了文件，尝试打开并读取文件内容。
    - 使用 pickle 反序列化文件内容，恢复图元列表。
- 恢复图元：
    - 遍历反序列化得到的图元列表，为每个图元创建 MyItem 对象。
    - 将每个图元添加到画布的场景中，并更新图元列表控件和字典。
- 更新界面：
    - 更新项计数器 item_cnt。
    - 更新当前打开的文件路径 opened_filename。
    - 更新窗口标题，显示打开的文件名。
- 如果在打开文件或反序列化过程中出现异常，弹出错误消息框提示用户。

7 总结

成功实现了一个完整的计算机图形学系统，支持基本图形的绘制以及图元的平移、缩放、旋转和裁剪等操作。也通过本项目深入理解了计算机图形学的基本概念和算法。