

课程设计三

绘图程序

171860574 胡育玮 计算机科学与技术系

目录

一、项目概述

二、具体设计

三、遇到的问题 and 解决方案

一、项目概述

1.1 概述

本项目实现了一个简单的绘图程序，有 **鼠标交互绘图**和 **文件指令绘图**两个模块。其中鼠标交互绘图模块可以像几何画板等软件一样实现直线、椭圆等图元的绘制和操纵，而文件指令绘图模块则通过一套特定的指令来进行绘图。两个模块在功能上是等价的，可以进行完全相同的操作。

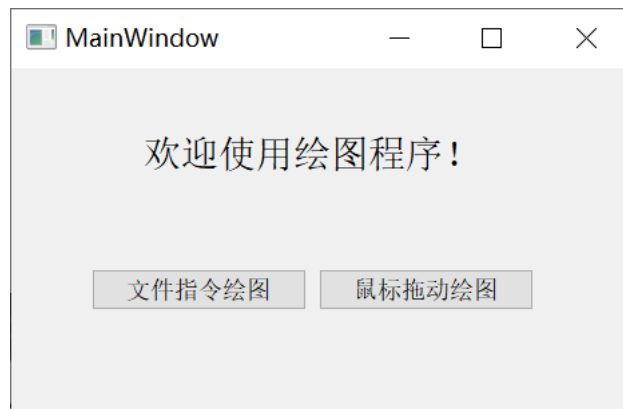


图 1：程序的开始界面。点击相应的按钮即可使用相应模块来绘图。

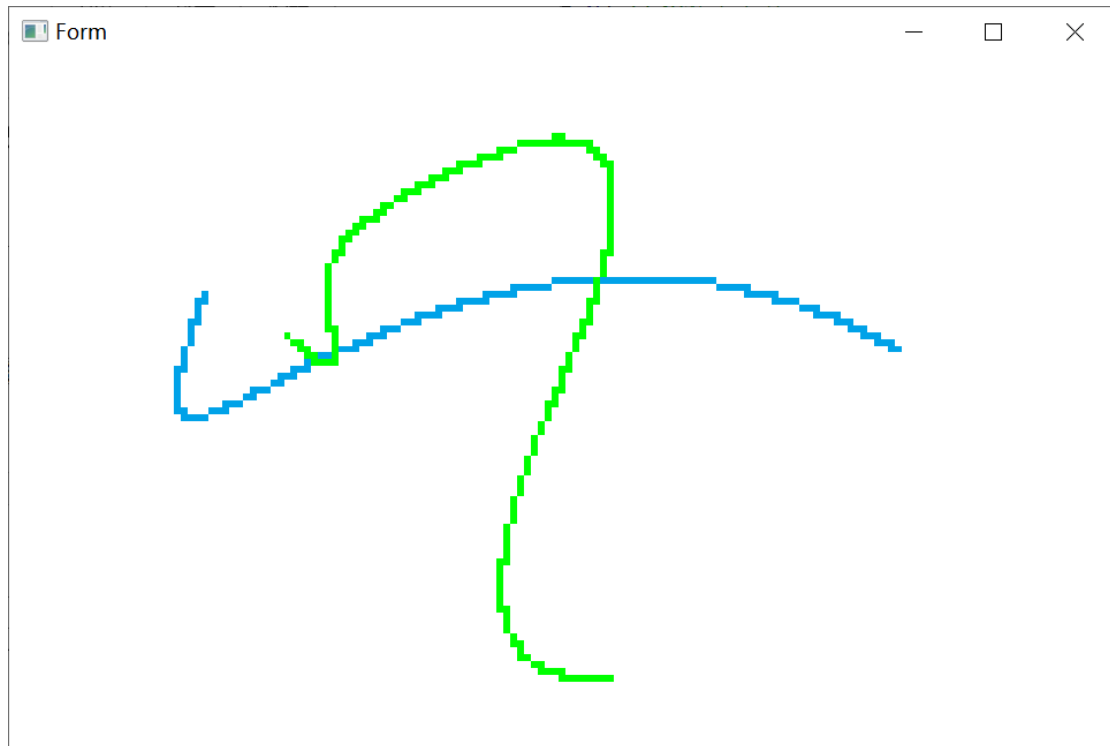


图 2：文件指令绘图模块的输出结果。该模块读取程序目录下的指令文件 `input.txt` 中的指令来进行绘制，并将绘图结果保存为 `bmp` 文件（如果有读取到 `save` 指令才会保存），并将最后一次保存的 `bmp` 文件的内容**放大**展示出来。本图就是放大展示后的结果。

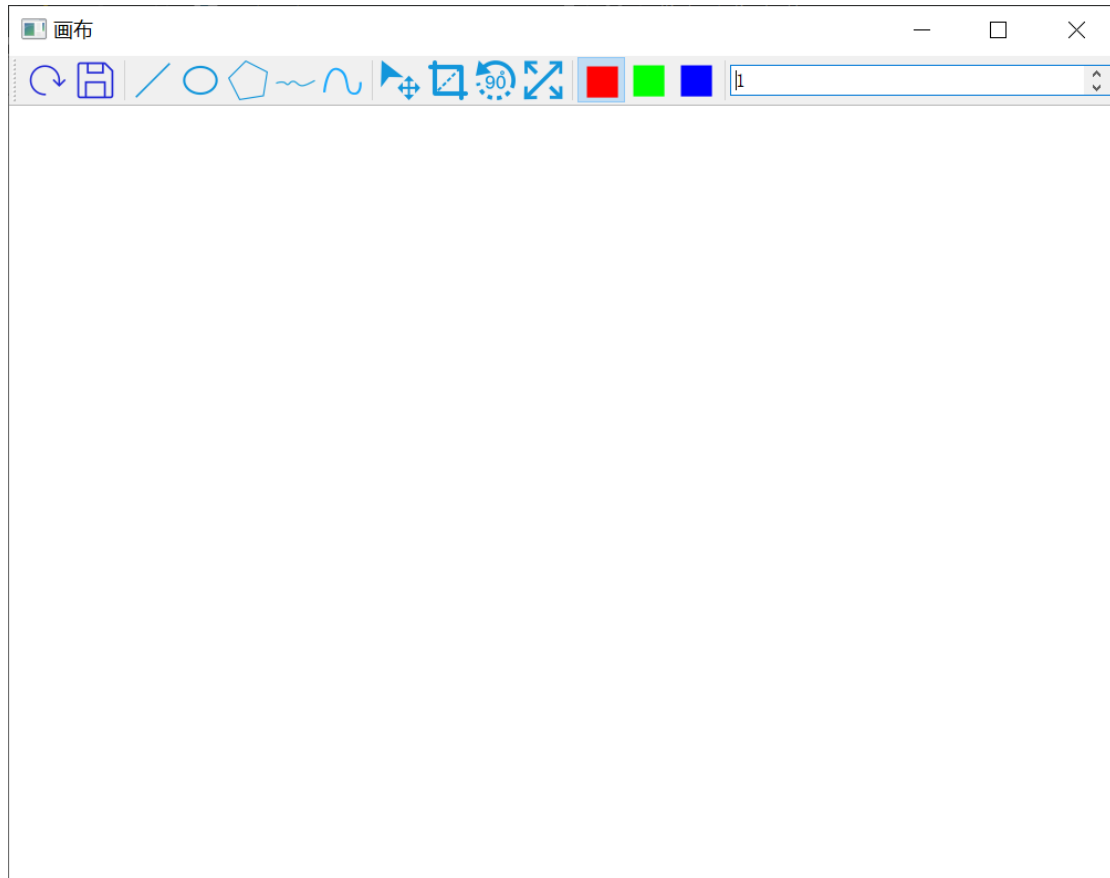


图 3：鼠标交互绘图界面。该界面的空白区域为画布，用户通过鼠标操作在这上面绘图。

本程序支持的操作如下：

图元的绘制：

- 直线（线段）绘制
- 椭圆绘制
- 多边形绘制
- 贝塞尔曲线绘制
- B 样条曲线绘制

图元的操纵：

- 平移
- 裁剪（仅线段）
- 旋转
- 缩放

鼠标交互和文件指令绘制模块都支持以上全部操作，两者使用的算法是通用的，因此可以在画布上绘制出完全一样的图形。



1.2 程序操作说明

1.2.1 文件指令绘图模块

如前述，在开始界面点击“文件指令绘图”按钮后，文件指令绘图功能*自动*完成，最后会用一个窗口来展示最后一次保存的 bmp 文件的内容。文件指令中若有保存指令，则会保存相应的文件到程序目录下。

1.2.2 鼠标交互绘图模块

进入图 3 所示界面后，可见标题栏下方的工具栏内有许多按钮，其功能分别如下：

-  刷新：清空当前画布上的所有图元
-  保存：保存当前画布的内容为 bmp 文件，文件名为

UI_output.bmp

-  直线绘图：绘制直线（线段）。
-  椭圆绘图：绘制椭圆。
-  多边形绘图：绘制多边形。
-  贝塞尔曲线绘图：绘制贝塞尔曲线。
-  B 样条曲线绘图：绘制 B 样条曲线。
-  平移：对画布上的图元进行平移。
-  裁剪：对画布上的所有**线段**进行裁剪。
-  旋转：对画布上的所有图元进行旋转。
-  缩放：对画布上的所有线段进行缩放。
-  红色：使用红色画笔来绘制图元。
-  绿色：使用绿色画笔来绘制图元。
-  蓝色：使用蓝色画笔来绘制图元。

点击相应按钮后，便可以以相应的模式进行绘图等操作。

（1）直线绘图

在画布上**左击**一个点，按住鼠标左键不放，拖动鼠标使光标在画布上移动，在想要的位置松开左键，这就是绘制一条线段的完整流程。在拖动鼠标的过程中，一开始的左击点（线段起点）与光标当前位置之间有线段连接，该线段会随着鼠标的拖动而“移动”，可以看作是“交互式动态效果”。

（2）椭圆绘图

操作同直线绘图，也是用按键+拖动的形式来绘图，也会动态显示椭圆形。

（3）多边形绘图

多边形绘图与直线和椭圆绘图不同。选择该模式后，在画布上不断用**左键单击**一些点，画布上会实时显示出这些点依次首尾相连形成的多边形。要结束绘制，需单击鼠标右键。没有按住拖动的操作。

（4）贝塞尔曲线绘图

操作同多边形绘图，用鼠标左键单击画布上的位置，画布上会实时显示出以这些点为各个**控制点**而绘制出的贝塞尔曲线。右键单击以结束绘制。

（5）B 样条曲线绘图

操作与贝塞尔曲线绘图完全一致。

（6）平移

在画布上已经绘制完毕的每个图元都有其提示点（tip point），提示点会在界面中显示出来：

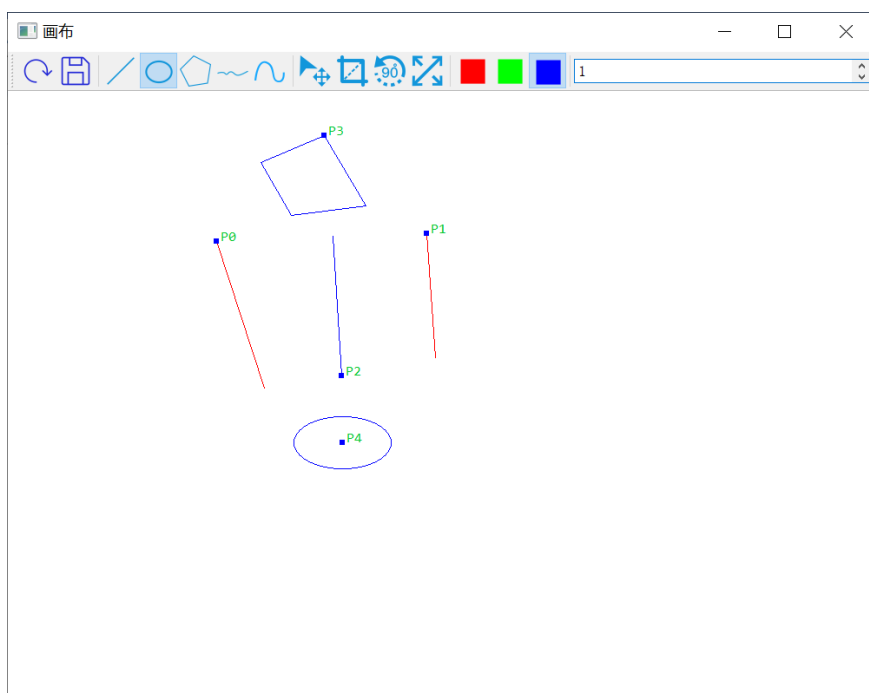



图 4：提示点

选择平移操作后，将光标移至提示点附近，若光标变成这样，则表示可以进行平移。按住鼠标左键不放，拖动鼠标，让光标在画布上移动到合适的位置后，松开左键，图元的平移就完成了。平移过程（鼠标拖动过程）中图元的图形不会随光标实时地移动，只有松开左键完成平移后，图元才会真正的“平移”并显示在界面上。下图为图 4 中多边形平移后的结果：

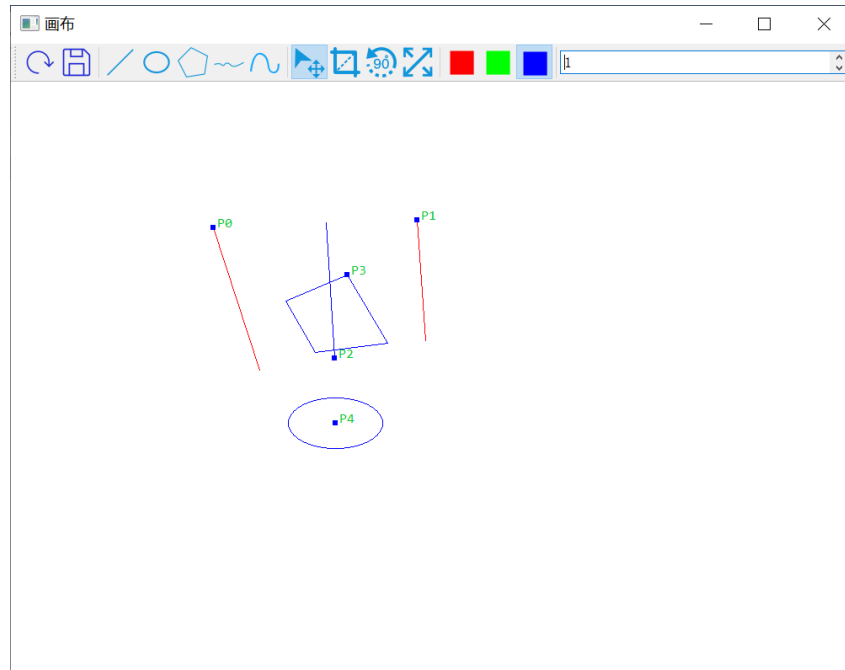


图 5：平移图 4 中的多边形后的结果

（7）裁剪

选择该模式后，在画布上选择起始点（设为 A）后，按下左键拖动鼠标，在某点（设为 B）松开左键。A，B 两点间形成一个矩形区域，为裁剪窗口。用这个窗口去裁剪画布上的所有线段，线段在窗口内的部分可以保留，不在窗口内的部分被删去。除线段以外的图元不受影响。

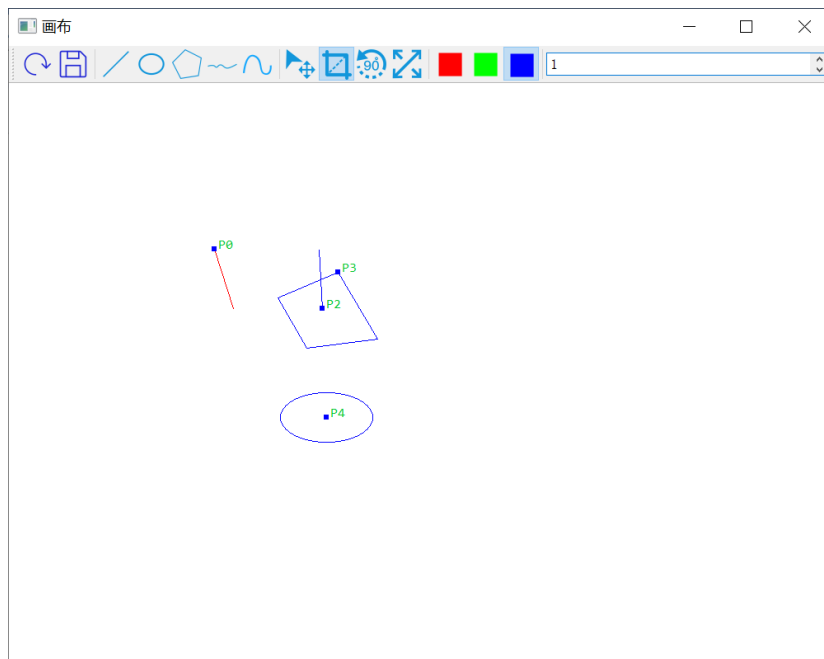


图 6：裁剪图 5 中的线段后的结果

（8）旋转

选择该模式后，*图中会出现一个参考点 C*：

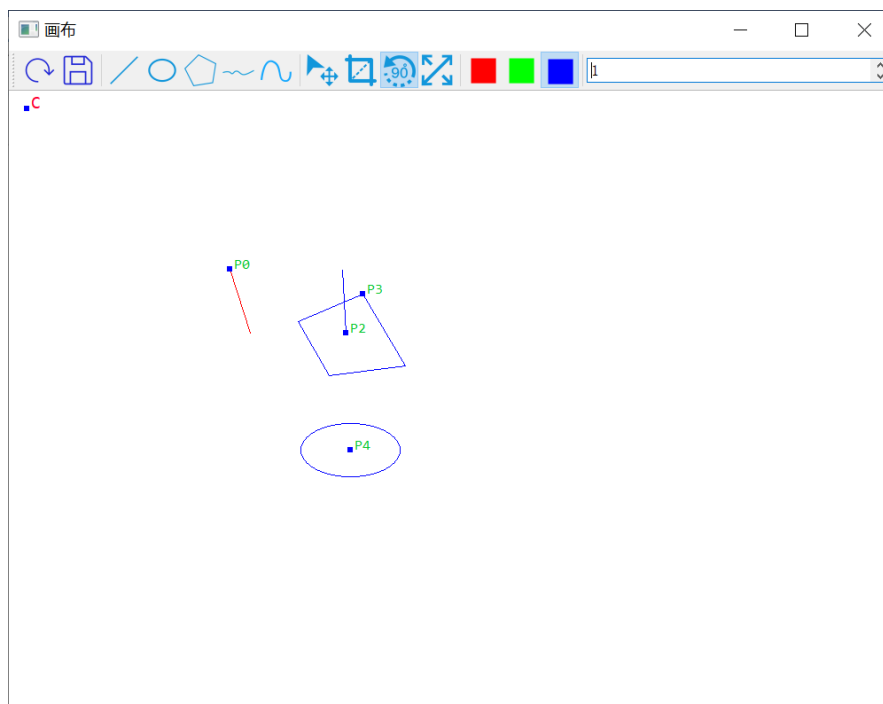



图 7：可见左上角的参考点 C

在该模式下，可像平移操作一样随意平移参考点 C 到合适的位置。然后，将光标移动到要旋转的图元的参考点上，当光标变为时表示可以进行旋转。此时单击左键，便会让这个图元完成顺时针旋转操作。旋转操作以参考点 C 为中心点，旋转的角度数（以整数的角度制表示）从工具栏最右侧的输入框中获取。用户可以用键盘改变输入框中的数值以设置不同的选择角度。默认为 1。退出旋转模式后，点 C 会消失。

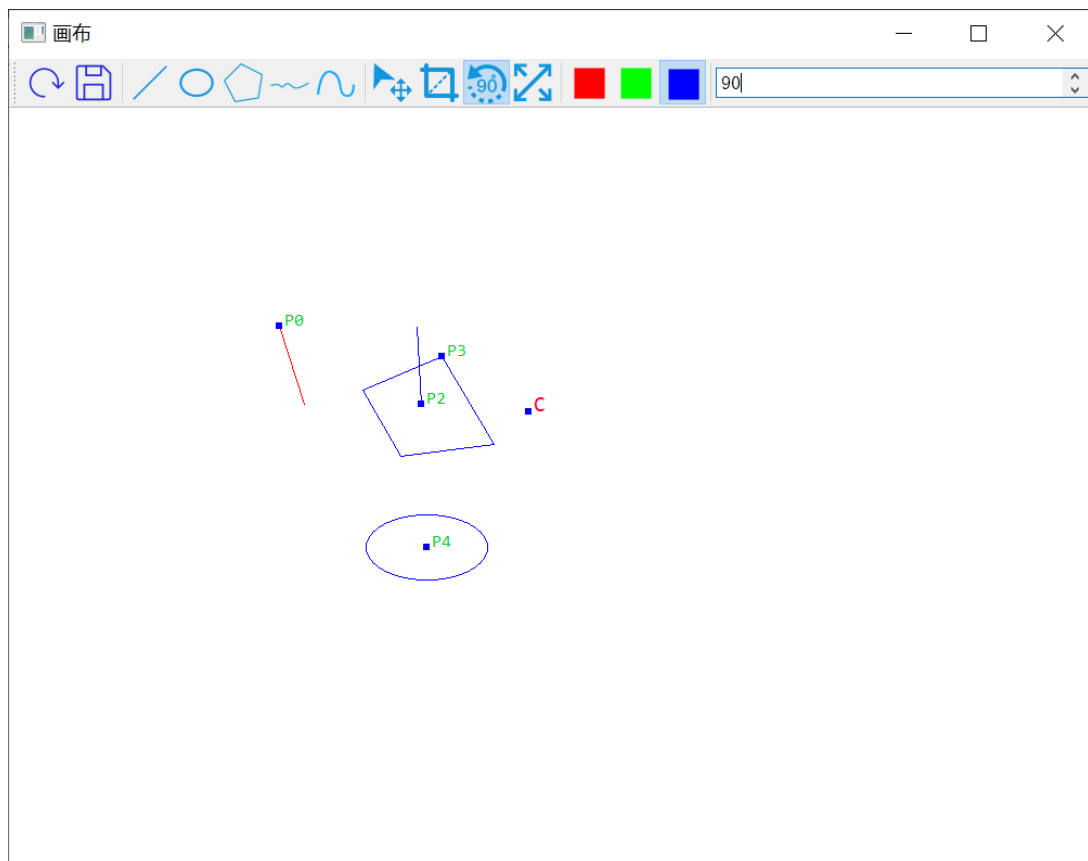


图 8：将参考点 C 移动到合适位置

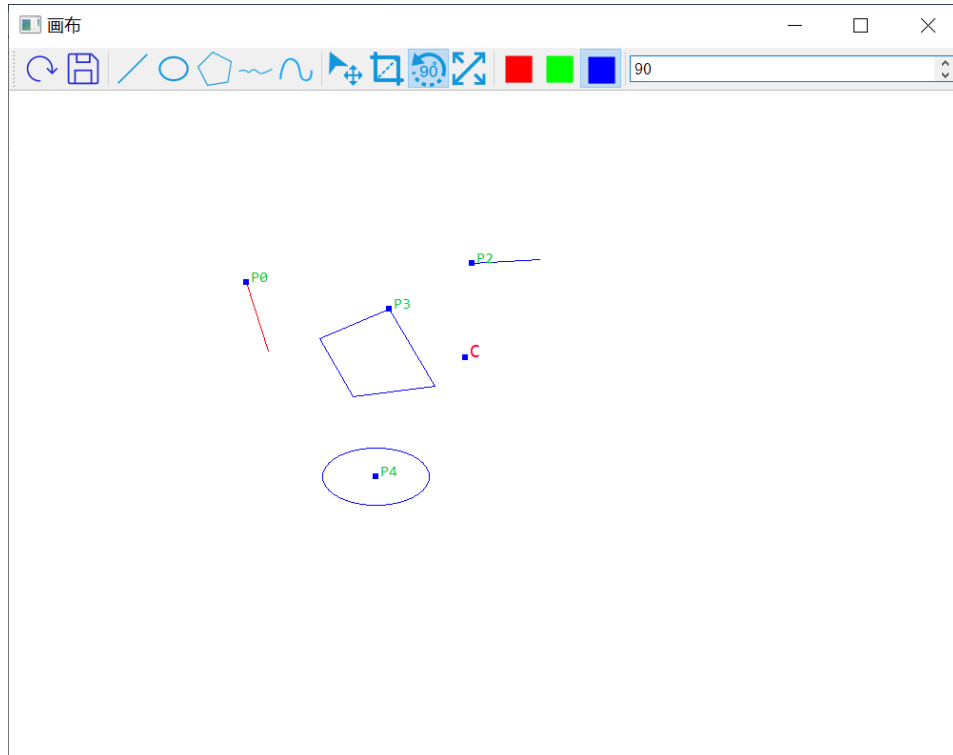


图 9：90 度旋转提示点 P2 对应的直线后

（9）缩放

操作与旋转操作完全一致。以参考点 C 为中心对相应图元进行缩放，缩放倍数（为大于 0 的正数，可以为小数）从输入框中获取。退出旋转模式后，点 C 会消失。

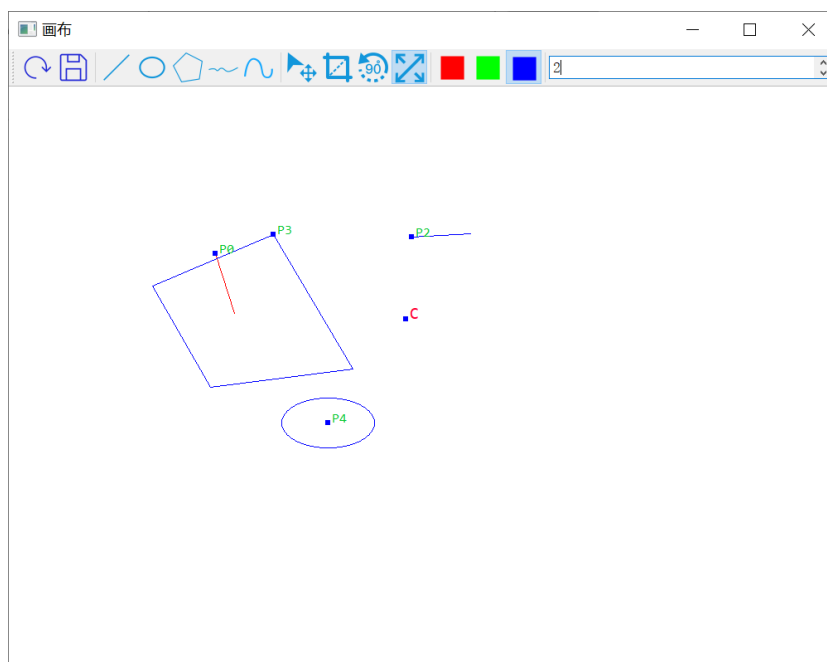


图 10: 在图 9 的基础上, 相对于点 C 2 倍缩放提示点 P3 对应的多边形图元

(10) 红色, 绿色, 蓝色

点击后, 新绘制的图元会以相应的颜色显示。

(11) 刷新

点击后, 画布会清空, 所有已绘制的图元被删除。

(12) 保存

点击后, 将当前画布的内容保存至 UI_output.bmp 文件中。注意, 提示点和参考点不会出现在保存的文件中, 也即 UI_output.bmp 中只有各个图元本身。

二、具体设计

2.1 设计综述

面向对象是一种以数据为中心，关心数据及对其的操作的表示、封装的程序设计范式。从这个角度出发，我们应首先重点分析出在本程序中，处于中心位置的“数据”是什么。

可以看出，本程序中的操作可以大致分为两类：**图元的生成**和**图元的操纵**。其中图元的生成包括直线、椭圆、多边形、两种曲线的生成；图元的操纵包括旋转、平移、缩放、裁剪。由此可以看出，若把单个图元的点集作为“核心数据”，则旋转、平移、缩放、裁剪可以看做是对数据的操作。因此，可以将图元的点集和这 4 个操作对应的算法封装到一个类中。而图元的生成操作可以单独封装到另一个类中，里面包含各种图元的点集的生成算法。这就是本程序的核心设计思路。

2.2 具体设计

下面介绍各个主要类的具体设计。

2.2.1 Alg 类

Alg 类封装了各种类型的图元的生成算法，包括直线的两种生成算法（DDA 和 Bresenham 算法），中点椭圆生成算法，贝塞尔曲线生成算法，B 样条曲线生成算法。算法细节在此不赘述。

2.2.2 Item 类（图元基类）

Item 类是各种类型的图元（直线、椭圆、多边形、曲线）的基类，是抽象类。该类封装了图元的点集数据以及以下方法：

public 方法及属性：

(1) virtual void **translate**(int dx, int dy) = 0;

平移操作的公共接口。平移操作和旋转、缩放操作一样，其具体实现由各个图元子类决定，没有“公共”的实现方案，因此在 Item 图元基类中设为纯虚函数。

(2) void **trans_point**(QPoint &po, int dx, int dy);

平移单个点。平移单个点和旋转单个点、缩放单个点一样，是平移、旋转、缩放操作的基础。这三个基础函数对于每种图元的对应操作都是一样的，因此可以在基类中给出实现。

(3) virtual void **rotate**(int x, int y, int r) = 0;

旋转操作的公共接口。

(4) void **rotate_point**(QPoint &po, int x, int y, double theta);

旋转单个点。

(5) virtual void **scale**(int x, int y, double s) = 0;

缩放操作的公共接口。

(6) void **scale_point**(QPoint &po, double s, double xtmp, double ytmp);

缩放单个点。

(7) void **draw**(QPainter& p);

画图元操作。该操作仅使用类中的图元的点集这一属性，对于各个子类是共通的，因此可以放到基类中实现。

(8) Type type; //图元的类型

int id; //图元的 ID

QPen pen; //画笔信息，用来存储绘制当前图元的画笔的颜色

protected 方法及属性：

(9) QList<QPoint> *plist: 图元的点集

(10) Alg algo: 绘图算法类对象。在进行旋转、缩放操作时，一般都需要重新生成图元的点集，故包含此成员对象。

2.2.3 图元子类

图元子类一共有 4 种：**Line 类、Elli 类、Poly 类、Curve 类**，分别代表直线、椭圆、多边形、曲线类。

这 4 个子类中，除了各自覆盖（**override**）Item 基类中的 3 个纯虚方法，给出自己的实现以外，还有自己的一些独特的属性以及对应的一些方法。**这些独特的属性是为了方便对图元进行操纵而设立的，在 3 种图元操纵中需要用到这些属性。**下面逐一介绍。

(1) Line 类

QPoint sp: 直线的起点

QPoint ep: 直线的终点

(2) Elli 类

QPoint mid: 椭圆的中心点

int a: 椭圆的 x 轴半径

int b: 椭圆的 y 轴半径

(3) Poly 类

QList<QPoint> *nodelist: 多边形或曲线的节点数组

(4) Curve 类

QList<QPoint> *nodelist : 曲线的节点数组

int curvetype: 曲线的类型: 0 = bezier, 1 = B 样条

各个子类对平移、旋转、缩放操作的具体实现细节在此不赘述。

2.2.4 工厂类及其子类

本程序使用了工厂模式这一设计模式。

在简单工厂模式中，仅有一个总的工厂类用来创建产品；它根据接收到的参数来决定创建哪一种产品。例如，现在有宝马车和奔驰车两种车需要生产。在简单工厂模式中，仅有一个工厂类负责生成这两种车。其 UML 类图如下：

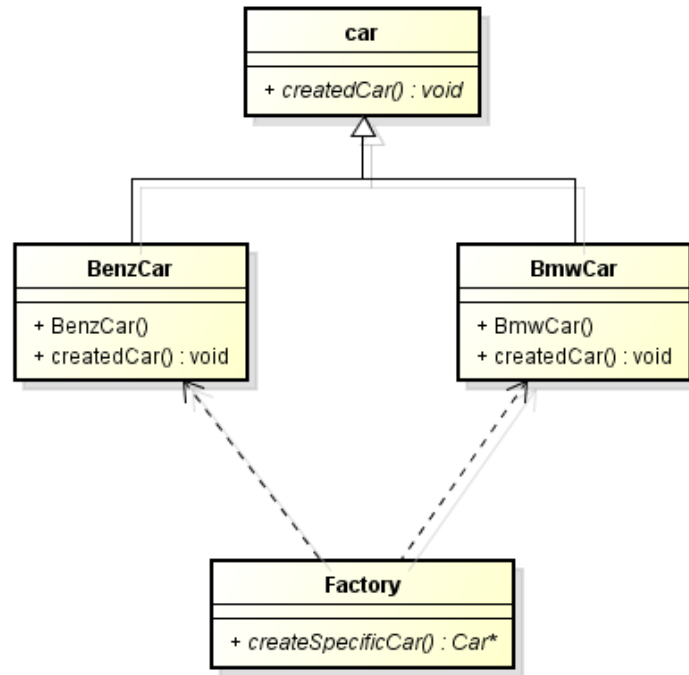


图 11：简单工厂模式的 UML 类图

其工厂类的可能的代码如下：

```
class CarFactory { //车厂
public:
    Car* createSpecificCar(CarType type) {
        switch(type) {
            case BENZ: return (new BenzCar()); //生产奔驰车
            case BMW: return (new BmwCar()); //生产宝马车
            default: return NULL; break;
        }
    }
};
```

可见，工厂类的生成函数需要一个参数来确定到底生成哪种产品。

然而，**这样的简单工厂模式下，每次增加新的车型时都需要修改工厂类的生产函数**，这就违反了开放封闭原则：软件实体（类、模块、函数）可以扩展，但是不可修改。

因此，有了改进版的**工厂方法模式**。工厂方法模式下由**多个继承同一工厂基类的工厂子类来负责生成具体的产品**，要生产哪种产品就调用哪个工厂子类的生产函数来生产。当有新的产品（**继承统一的产品基类**）出现时，只需新创建一个对应的工厂子类即可实现新产品的生成，这样无需修改原有的代码。

本程序使用**工厂方法模式**来进行新图元的生成。**Item** 类就是统一的产品基类，各个图元子类是各种类型的产品。每种产品都需要一个对应的工厂子类来完成生成工作。

（1）工厂基类

```
class Factory { //工厂基类

public:

    virtual Item* createItem(int idpa, QList<QPoint> *ppa, const QPen& p)
    = 0;

};
```

工厂基类提供一个**纯虚生产方法接口**，其返回值应为抽象的产品基类类型（或其指针）。

（2）工厂子类

工厂子类覆盖基类中的 `createItem()` 抽象接口，给出具体的实现。

例如，在直线工厂 `LineFty` 子类中，`createItem()` 的定义为：

```
Line* LineFty::createItem(int idpa, QList<QPoint> *ppa, const QPen& p)
{
    return new Line(idpa, ppa, p);
}
```

返回的是子类的对象指针。

另外 3 个工厂子类的生产函数的实现不再赘述。

2.2.5 display 类

该类用于 **文件指令绘图**，继承 Qt 内置的 `QWidget` 类。其主要接口和属性的介绍如下：

- (1) `QPixmap *hehe`：画布，用于绘图
- (2) `void draw()`：读取指令文件中的指令，调用相应的处理函数进行绘制等操作。
- (3) `QPen pen`：画笔
- (4) `QList<Item*> item`：图元列表对象
- (5) `LineFty linefty` 等：各个图元工厂对象。共 4 个工厂，对应 4 种图元
- (6) `Alg algo`：绘图算法对象
- (7) `void drawLine(int id, int x1, int y1, int x2, int y2, char alg[])` 等：各个图元的绘制函数，由 `draw()` 方法在处理好相应的数据后调用。这些

绘制函数会使用绘图算法对象 **algo** 来执行各种图元的点集的生成，并使用相应的工厂对象来最终生成相应的图元对象指针并添加到图元列表对象 **item** 中。一共有 4 个这种绘制函数，对应 4 种类型的图元。

(8) **void translate_unit(int id, int dx, int dy)**等：图元操纵函数。对 item 图元列表中的某个 id 的图元进行平移、旋转、缩放、裁剪操作。对于前 3 种操作，直接调用 **Item** 类的相应公共接口，利用虚函数的动态绑定特性即可实现对不同类型图元的处理：

```
void display::translate_unit(int id, int dx, int dy) {  
  
    for (Item* it: item) {  
  
        if (it->id == id) {  
  
            it->translate(dx, dy);  
  
            break;  
  
        }  
  
    }  
  
}
```

而对于裁剪操作，则需要在确定图元列表中的相应图元是直线后，通过对图元对象指针继进行类型转换来调用直线 **Line** 类独有的裁剪方法。

2.2.6 drawwindow 类

该类用于鼠标交互绘图。其主要方法和属性如下：

```
enum Dr_State //drag state, 拖动的状态
{
    still,    //
    topoint,  //画曲线，多边形时的待点击状态，旋转等操作时待点击状态
    todrag    //画直线、椭圆时的待拖动或拖动中状态
};

enum Prim    //当前选择的绘图模式
{
    ellipse,  //椭圆绘图
    line,     //直线绘图
    poly,     //多边形绘图
    curve,    //曲线绘图
    trans,    //平移
    rota,     //旋转
    sca,      //缩放
    clip,     //裁剪
    unchecked //未选择
};

struct TP //提示点
{
    int id;    //对应的图元ID
    QPoint p;  //图元的提示点
    TP(int idp, QPoint pp) {id = idp; p = pp;}
    TP() {}
};
```

```

int w;           //width, 宽度, x轴
int h;           //height, 高度, y轴
int unit_id;     //要添加到图元列表中的已画好的图元的ID
int curvetype;   //当前正在绘制的曲线的类型: 0 = Bezier 1 = Bspline
QPixmap hehe;    //主画布
QPixmap hehedis; //用于显示到屏幕上的画布, 在主画布的基础上多画了提示点
QPen pen;
QPainter painter;
QList<Item*> item; //图元列表
Dr_State state;   //鼠标当前的拖动状态
Prim prim;        //当前选择的绘图模式
QPoint start_pos; //“起点”
QList<QPoint*> cur; //当前正在绘制的图元的点列表
QList<TP> tippoint; //图元的提示点集
QList<QPoint> ctrlpoint; //图元的控制点集
int trindtip;     //所选择的图元在tippoint数组中的索引
int trind;        //所选择的图元在item数组中的索引
bool totrans;     //是否已进入平移状态
QPoint rp;        //旋转和缩放的参考点
bool showrp;      //是否显示参考点

QString savename = "UI_output.bmp";

//各个工厂
LineFty linefty;
ElliFty ellipty;
PolyFty polyfty;
CurveFty curvefty;

//绘图算法
Alg algo;

```

2.3 绘图的实现

在负责鼠标交互绘图的 `drawwindow` 类中, 有专门负责响应鼠标事件的, 继承自 `QWidget` 基类的槽函数。鼠标事件一共有 3 种: 鼠标点击事件、鼠标移动事件、鼠标松开事件。当用户在窗口内用鼠标做出点击、移动、松开等事情时, Qt 的事件系统会接收到相应的信号并调用这 3 个槽函数进行处理。事件系统在调用槽函数时, 会封装一个 `QMouseEvent` 类的指针作为参数传给槽函数槽函数, 槽函数可以通过

这个参数来获得诸如光标当前的位置坐标、鼠标按下的是哪个键等信息，以便进一步处理。

本程序就是利用这样的机制来进行鼠标交互绘图的。在 `drawwindow` 类中维护了许多程序状态信息，**在槽函数中根据状态信息以及通过函数参数获取的光标位置坐标、按键种类等信息来判断应当进行的操作**，如对特定的图元进行旋转、直线绘图时随光标移动实时显示出直线等。

在生成图元的点集时，会使用成员对象 `algo` 来使用图元生成算法。在最终生成图元对象时，会使用各个工厂对象来生产。

直线、椭圆图元最终生成于鼠标松开事件的处理中；多边形、贝塞尔及 B 样条曲线图元最终生成于鼠标右键点击事件的处理中。**进行这些处理时会使用工厂对象来创建图元对象。**

在对图元进行平移、选择、缩放处理时，**同样利用了虚函数的动态绑定特性**，使得要进行相应的处理只需在图元列表对象 `item` 中根据要处理的图元的 `id` 来找到对应图元后，调用图元基类 `Item` 类的平移、旋转、缩放接口即可针对不同图元子类进行不同的处理。

三、遇到的困难和解决方案

1. 面向对象式设计

一开始我认为本程序完全无需使用面向对象范式来设计，用过程

式范式完全可行且程序也一样可以被组织得很美观。而使用面向对象范式进行思考，我一开始发现无从下手。后来在复习了面向对象的核心思想，也即以数据为中心（而不是以功能为中心），将数据及其操作封装成对象之后，我成功分析出了该程序的“核心数据”是图元的点集，对图元的各种操作可以跟点集一起被封装起来，且不同类型的图元可以继承同一个图元基类。这样一来设计思路便开始清晰。同时，本程序还使用了工厂模式。虽然不使用也完全可行，但使用这种设计模式更能体现出良好的设计风格。

2. 对 Qt 图形窗口的不熟悉

在设计时遇到了从事件系统获取光标坐标与画布位置有偏移的情况，后通过对获取的光标坐标减去一个 `offset` 偏移值的方式来解决。本质上是由对 Qt 的图形窗口机制不够了解而产生问题。