《计算机图形学》系统报告

张雨 171860029
mail：1254832148@qq.com
(南京大学 计算机科学与技术系, 南京 210093)

摘要：在计算机图形学的学习过程中，结合所学知识，实现了图形学系统。其中实现的核心算法模块，包含各种图元的生成：线段的 DDA、Bresenham 生成算法，多边形的 DDA、Bresenham 生成算法，椭圆的中点圆生成算法，曲线的 Bezier、B-spline 生成算法。其中实现的对图元的变换算法，包括：对图元的平移算法，对图元的旋转算法，对图元的缩放算法，对线段的裁剪算法。最终实现了文件输入的命令行系统与用户交互系统。

关键词：计算机图形学 图形学系统 图元生成算法 图元变换算法

◆ 综述

■ 开发环境说明

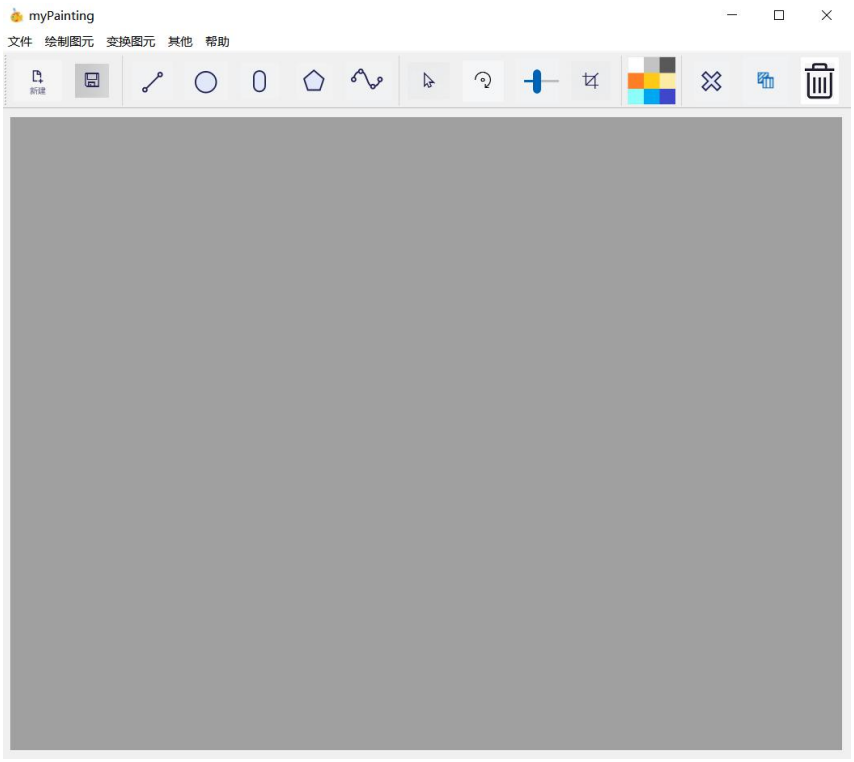
编程语言：C++
开发平台：Windows
GUI 开发框架：Qt
IDE/编译器：Qt Creator 4.10.0 (Community) + Desktop Qt 5.13.1 MinG 32bit/64bit

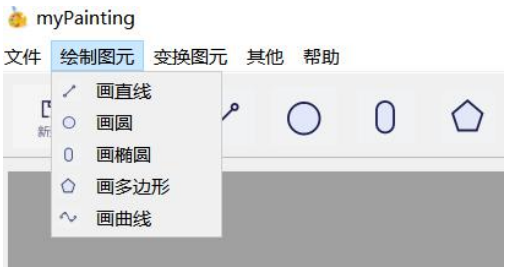
■ 实现的功能简介

图形界面部分

1. 主界面

• 实现：点击 exe 生成图形界面，初始如下。
• 点击菜单栏：文件→新建画布，保存画布。绘制图元→画直线，画圆，画椭圆，画多边形，画曲线。变换图元→平移，旋转，缩放，裁剪。其他→选择颜色，删除图元，改变图形，清空画布。帮助→版本
• 图标由左至右依次是：新建画布，保存画布，画直线，画圆，画椭圆，画多边形，画曲线，平移，旋转，缩放，裁剪，选择颜色，删除图元，改变图形，清空画布

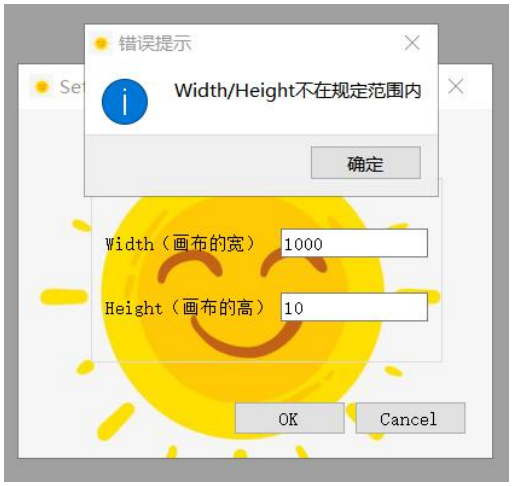




2. 新建画布

• 实现：点击工具栏从左到右第一个“新建”图标，开始新建画布

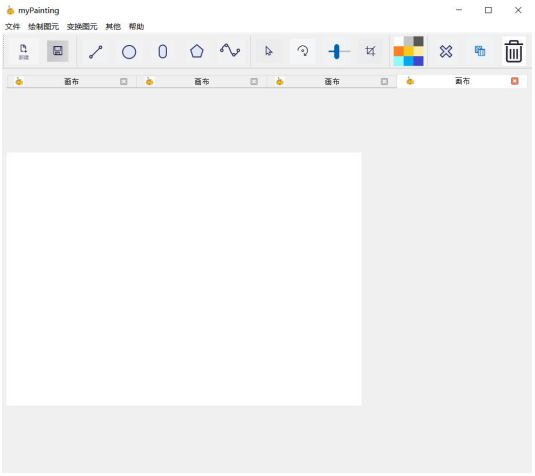
• 画布大小设置：在点击新建后出现询问框，输入画布的长宽设置，界面如下。输入的长宽在 100-1000 范围内，则新建相应大小的画布成功；否则弹出错误信息，新建画布失败。如下

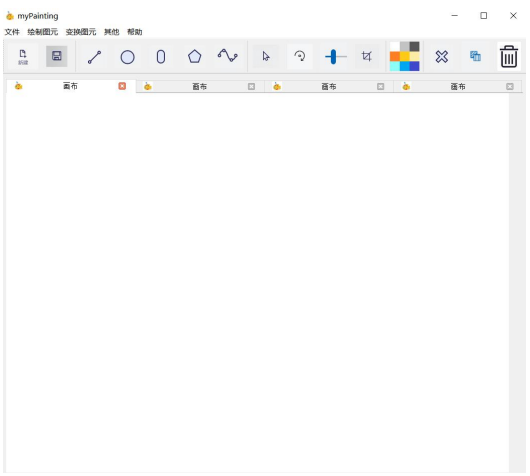


3. 多窗口新建画布

• 实现：可以同时多窗口设置画布，即可以新建多个画布，同时编辑，只需要点击“新建画布”即可。

• 每个画布之间的绘图互不干扰

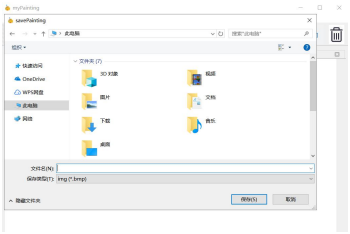




4. 保存画布

• 概述：点击工具栏左起第二个图标，保存当前图片为.bmp 格式

• 多画布，多窗口可以自己选择保存



5. 关闭窗口

点击每个窗口的右上角，即可关闭该窗口。为 Qt 中的自带函数实现。

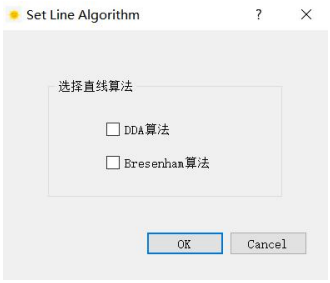
6. 图元的生成

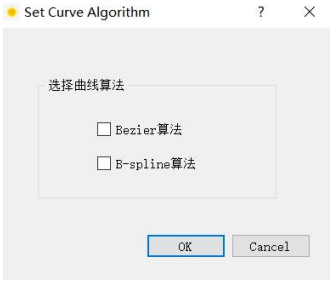
• 概述：点击相应的图标，生成相应的图元。从左起第三个开始，分别生成直线，圆，椭圆，多边形，曲线。

• 选择：由于许多图元的生成算法有多种，可以进行选择。

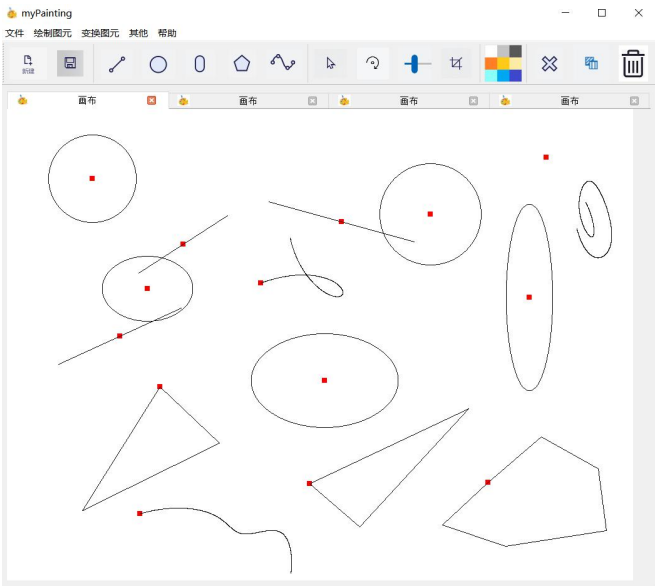
• 直线：选择 DDA 算法，Bresenham 算法。多边形：选择 DDA 算法，Bresenham 算法。曲线：贝塞尔曲线，B 样条曲线

• 每个图元都存在一个红色的控制点。





生成的图元如下。



7. 平移

• 概述：点击“平移图元”的图标

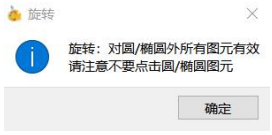
• 每个图元都有一个红点，表示的是该图元的控制点。选中相应的控制点，并且拖动，完成平移。

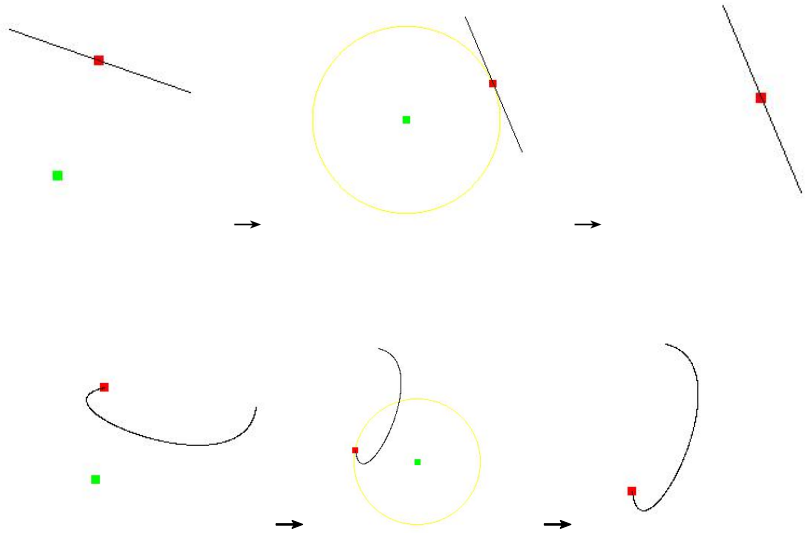
8. 旋转

• 概述：点击“旋转图元”的图标，会弹出旋转针对的图元范围的提示。

• 首先选中一点，作为旋转中心，该点为绿色的

• 然后选中某个图元控制点。拖拉控制，即可旋转。并且会有黄色的线画出旋转的轨迹（轨迹为一个圆形）。



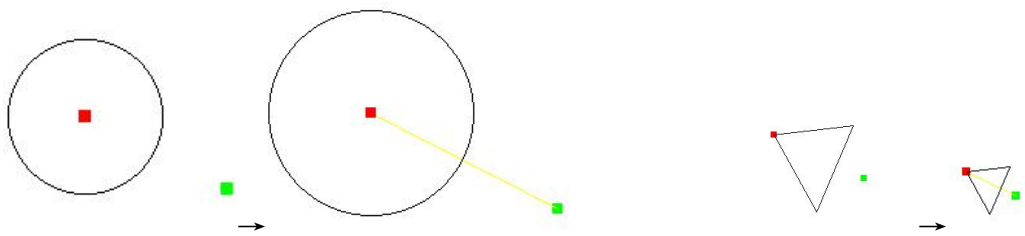


9. 缩放

• 概述：点击“缩放图元”的图标

• 首先选中一点，作为缩放中心，该点为绿色的

• 然后选中某个图元控制点。拖拉控制，即可缩放。并且会有黄色的线画出缩放的轨迹（轨迹为一条直线）。

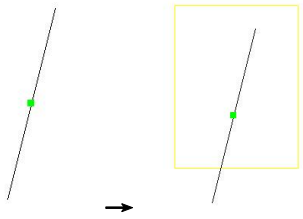


10. 直线裁剪

• 概述：点击工具栏第十一个图标，即裁剪直线。

• 裁剪算法有两种可选，分别是 Cohen-Sutherland 算法和 Liang-Barsky 算法

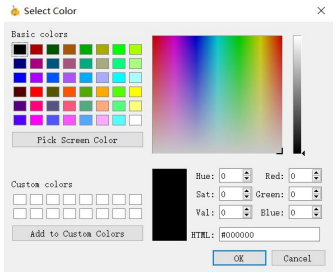
• 首先选择直线，通过选中控制点选中，选中直线的控制点变为绿色。然后拖出裁剪框，为黄色的矩形框。松开裁剪框。裁剪完成。



11. 设置画笔颜色

• 概述：点击工具栏中的彩色块的图标

• 可以选择画笔的颜色。接着画出的图元的颜色即为选择的颜色。



12. 删除图元

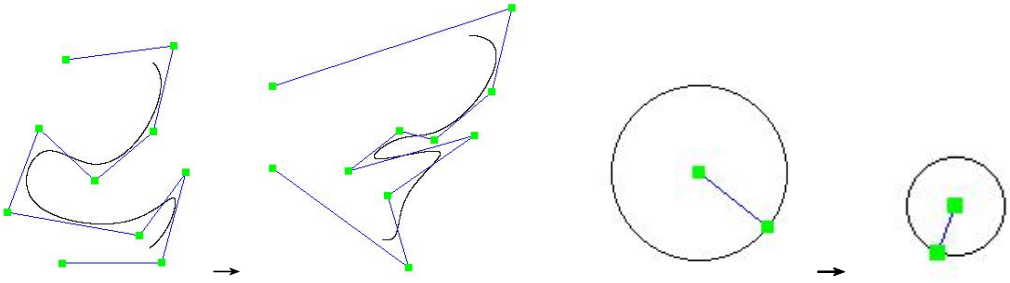
• 点击倒数第三个图标

• 首先选中图元，选中图元的控制点为绿色，此时点击右键。完成删除。

13. 修改图元

• 选中图元，然后就会显示改图元的所有控制点。曲线还会展示其控制多边形。此时可以拖动每个图元的控制点，改变图元的形状。

•注：因为实现原理的原因，在修改多边形时，修改最后一个控制点会导致形变。其他图元均没有问题。

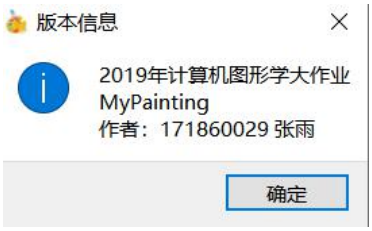


14. 清空画布

• 概述：点击工具栏最后一个图标，清空当前画布上的所有内容。此时画布上的所有图元均被清除。

15. 获取程序相关信息

• 概述：点击帮助→版本，弹出关于本程序的基本信息。



<u>命令行界面部分</u>

1. 画布操作

重置画布：resetCanvas width height
保存画布：saveCanvas name
设置画笔颜色：setColor R G B

2. 图元生成指令

绘制线段：drawLine id x1 y1 x2 y2 algorithm
绘制多边形：drawPolygon id n algorithm x1 y1 x2 y2 … xn yn
绘制椭圆：drawEllipse id x y rx ry
绘制曲线：drawCurve id n algorithm x1 y1 x2 y2 … xn yn

3. 图元变换指令

对图元平移：translate id dx dy
对图元旋转：rotate id x y r
对图元缩放：scale id x y s
对线段裁剪：clip id x1 y1 x2 y2 algorithm

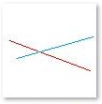
4. 功能使用

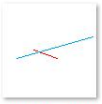
设置 input.txt 文件，执行./MyPainting.exe 指令序列文件 图像保存目录
实现结果如下。

input.txt

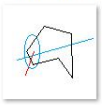
```
resetCanvas 100 100
setColor 255 0 0
drawLine 5 7 39 93 71 DDA
saveCanvas output_1
setColor 0 162 232
drawLine 233 96 35 15 58 Bresenham
saveCanvas output_2
clip 5 33 10 70 58 Cohen-Sutherland
saveCanvas output_3
rotate 5 33 49 90
saveCanvas output_4
drawEllipse 123 31 49 8 18
setColor 0 0 0
drawPolygon 666 6 DDA
44 22 73 30 74 77 59 56 32 63 25 49
saveCanvas output_5
rotate 666 59 56 -90
translate 666 -8 -18
saveCanvas output_6
resetCanvas 160 100
drawCurve 10 4 Bezier
28 34 9 86 61 4 129 42
setColor 0 255 0
drawCurve 11 4 Bezier
114 88 60 92 99 3 42 24
saveCanvas output_7
```

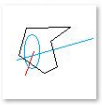

output_1.bmp


output_2.bmp


output_3.bmp


output_4.bmp


output_5.bmp


output_6.bmp


output_7.bmp

■ 性能测试

1. 系统性能测试

对于生成的系统，在一定程度上需要严格按照指定的操作顺序执行相应的功能，否则可能会导致异常退出。应该增加对错误输入情况的考虑，提供程序的鲁棒性。

2. 图元生成变换的性能测试

性能测试指对系统的各项性能指标进行测试，测试各个功能在各种情况下的正确性。各个图元的生成与变换都能够正确完成，具体的测试内容在使用说明书中可以看到，在此就不再详细说明。

◆ 系统功能的实现算法

■ 多窗口

1. 实现

主要通过 Qt 框架提供的系统调用，在每次新建画布的时候，将新建的画布加入子窗口中，并通过对新建窗口的初始化 initMdiArea 来实现多窗口的切换。这里将每个画布 Canvas 作为一个对象处理。

2. 多窗口的生成

```
Canvas *theCanvas = new Canvas(this,width,height);                    //新建一张画布
allCanvans.push_back(theCanvas);                                      //将新建的画布放入画布容器

QMdiSubWindow *w = ui->mdiArea->addSubWindow(theCanvas);              //将这张画布加入子窗口
allWindow.insert(w,WindowIndex);                                      //窗口和窗口序号加入Qmap中
WindowIndex++;
ui->mdiArea->setActiveSubWindow(w);

w->setWindowTitle(QStringLiteral("画布"));                             //设置每个画布上面的名字
w->show();
```

3. 确定当前所处窗口

寻址：对于应该在哪个子窗口进行操作，利用新建画布时的 Qmap 来寻找

代码实现如下。

```
unsigned int MainWindow::findWindow(){
    unsigned int index=allWindow.find(ui->mdiArea->activeSubWindow()).value();
    return index;
}
```

4. 实现想法

在绘图中很可能希望多图绘制，那么在一个 area 中设置多块画布同时绘制成了很自然的想法。在 qt 中又对相应的功能提供了便利的实现。同时只要将每块画布的指针放入 map 中，当点击相应画布的时候找到当前窗口画布并进行修改，即可完成。

■ 单个图元的绘制

1. 图元类的定义

自定义了一个类 AllFigure 类。它是所有图元的父类。直线定义了 Line 类，圆定

义了 Circle 类，椭圆定义了 Ellipse 类，曲线定义了 Curve 类。它们均继承了 AllFigure 类。

2. AllFigure 类

其中针对所有图元，定义了图元中的属性与功能。如下。

3. AllFigure 类的继承关系

■ 画布的使用

1. 概述

自定义了一个 Canvas 类，用作一个画布，其中包含用于实现功能的成员函数和成员变量。

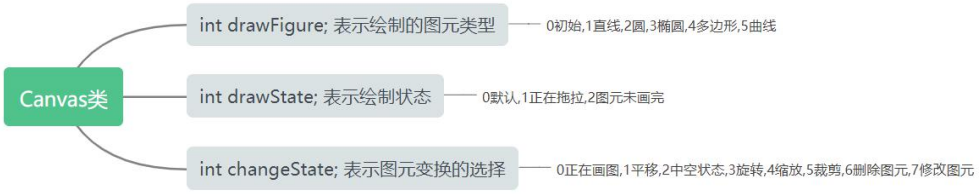
2. 图元存储

每张画布 Canvas 中有私有变量 vector<AllFigure*> figureBox 存储在当前画布上绘制的所有图元。每个图元的类型为自定义的 AllFigure 类型

3. QPixmap

设置了两个 QPixmap 类型的变量，TempPix 用于在拖动时的像素级界面缓冲，CanvasPix 用于在拖动完成后确定像素级绘制图形。

4. 状态变化



5. 选中图元

在图形界面下，选中某个图元，选中对应的红色控制点即可，具体实现是当像素偏差在 5 以内，表示选中，而每个图元有各自的 id。

6. 图元变换

每个图元的几个关键点被存储，在选中了某个图元后，根据变换类型改变存储的关键点信息，然后将所有该画布上的图元重新绘制，从而达到图元变换的效果。而父类 AllFigure 中有虚函数 drawFigure，通过指针调用会调用相应的子类的 override 的同名函数 drawFigure，从而达到图元绘制的效果。

◆ 命令行的实现

1. 判断是否为命令行

因为图形界面和命令行使用的同一程序，而命令行的大部分实现是在图形界面之后实现的，所以我在实现命令行的时候，仅仅将应该展示的窗口没有展示出来，其他大部分内容仍然是调用的原来程序中的函数。只要判断是否传入了参数，即判断 argc 的值，就可以判断是否要展示窗口。

2. 命令的存储

新建一个类 cmdcontrol，将文件中读出的命令一行一行的存入类中的私有变量 vector<string> cmdPacket 中。然后将这个 vector 传给处理的界面。对每个指令进行分词，然后判断对应的执行函数。其中如果发现是 drawPolygon 或者是 drawCurve，因为它们占两行命令行，那么也要读取相应的两行。

◆ 窗口间的跳转与窗口信息传递

利用 Qt 的信号与槽机制。
以直线算法弹出算法选择窗口为例。

1. 点击绘制直线图标
触发 on_LinePaint_triggered 函数。弹出 lineAlg 界面，接受来自 sendLineAlgorithm 传入的参数，并且传参数给 NewLine 函数

```
void MainWindow::on_LinePaint_triggered(){
    LineAlgorithm *lineAlg=new LineAlgorithm;
    lineAlg->setModal(true);
    lineAlg->show();
    connect(lineAlg,SIGNAL(sendLineAlgorithm(int)),this,SLOT(NewLine(int)));
}
```

2. 直线算法选择界面
其中先判断选择的是 DDA 还是 Bresenham 算法，并且要选择有且仅有一个算法。
然后 emit sendLineAlgorithm(参数)

```
void LineAlgorithm::on_buttonBox_clicked(QAbstractButton *button){
    bool dda=ui->DDA->isChecked();
    bool bres=ui->Bresenham->isChecked();
    if( (dda==false&&bres==false)||(dda==true&&bres==true) ){
        QString myTitle = QStringLiteral("错误提示");
        QString myInfo = QStringLiteral("请选择有且仅有一个算法选项！");
        QMessageBox::information(this,myTitle,myInfo,QStringLiteral("确定"));
    }
    else{
        if(dda==true){
            emit sendLineAlgorithm(0);
        }
        else{
            emit sendLineAlgorithm(1);
        }
    }
}
```

◆ 图元的生成算法/实现

◆ 直线的绘制

DDA 算法

1. 算法简介
数值差分分析 (digital differential analyzer)。直接利用直线 x 或 y 方向增量△x 或

△y 的线段扫描转换算法，利用光栅扫描显示特性：像素列阵为屏幕单位网格。

2. 算法原理

离散取样：使用 x 或 y 方向单位增量间隔(△x 或△y=±1)逐步计算沿线路径各像素位置。一个坐标轴上以单位增量对线段离散取样，确定另一个坐标轴上最靠近线段路径的对应整数值。

3. 取样方向

取决于直线斜率的绝对值大小。

斜率的绝对值小于 1 时，在 X 方向取样，计算 Y 方向位置坐标

斜率的绝对值大于 1 时，在 Y 方向取样，计算 X 方向位置坐标

4. 增量(△x 或△y)的取值

取决于线段生成方向与坐标轴方向的关系。

线段生成方向与坐标轴方向相同取＋1

线段生成方向与坐标轴方向相反取－1

5. C++代码实现

```
void Line::DrawLineDDA(QPainter &painter,QPoint begin,QPoint end){
    int x_b=begin.x();
    int y_b=begin.y();
    int x_e=end.x();
    int y_e=end.y();
    double delt_x=(x_e-x_b);
    double delt_y=(y_e-y_b);
    double x=x_b;
    double y=y_b;

    if(abs(delt_x)>abs(delt_y)){
        //斜率小于 1
        double m=delt_y/delt_x;
        if(delt_x>0){
            for(int i=0;i<=abs(delt_x);i++){
                painter.drawPoint((x+0.5),(y+0.5));
                x=x+1;
                y=y+m;
            }
        }
        else{
            for(int i=0;i<=abs(delt_x);i++){
```

```
                painter.drawPoint((x+0.5),(y+0.5));
                x=x-1;
                y=y-m;
            }
        }
    }
    else{
        //斜率大于 1
        double m=delt_x/delt_y;
        if(delt_y>0){
            for(int i=0;i<=abs(delt_y);i++){
                painter.drawPoint((x+0.5),(y+0.5));
                x=x+m;
                y=y+1;
            }
        }
        else{
            for(int i=0;i<=abs(delt_y);i++){
                painter.drawPoint((x+0.5),(y+0.5));
                x=x-m;
                y=y-1;
            }
        }
    }
}
```

6. 算法理解

以上是我的代码实现，因为对象限，斜率，直线的始末方向的思考不到位，在写代码过程中针对出现的错误结果不断发现代码中的错误并最后完成。

代码对每种情况一一做判断，具体过程是：首先判断了斜率的绝对值与 1 的关系，然后对直线绘制的方向，也就是判断是从上到下还是其他方向，最后绘制出直线。当然判断直线的绘制走向是针对图形界面进行的判断。

其中需要注意的是，qt 中的 xy 方向坐标轴布置方向，与平日的方向有所不同：qt 的 y 轴是自上向下的，而平时的惯性中 y 轴自下向上。

我的代码实现整体显得很长而且繁琐。

7. 代码学习

后来看到了如下的 DDA 算法的实现。来源于网上。

可以看到如下的代码代码量相比我写的远远小了很多。在核心思路一致的情况下，巧妙的是简化了对符号的判断，是值得学习的部分。所以，我也需要在已有算法的基础上多增加自己对实现的思考。

```
void CDraw::DDALine(CDC* pDC, int x1, int y1, int x2, int y2, COLORREF color){
```

```
    double dx, dy, e, x, y;
    dx = x2 - x1;
    dy = y2 - y1;
    e = (fabs(dx) > fabs(dy)) ? fabs(dx) : fabs(dy);
    dx /= e;
    dy /= e;
    x = x1;
    y = y1;
    for (int i = 1; i <= e; i++){
        pDC->SetPixel((int)(x + 0.5), (int)(y + 0.5), color);
        x += dx;
        y += dy;
    }
}
```

Bresenham 算法

1. 算法简介

Bresenham 提出的一种算法：采用整数增量运算，精确而有效的光栅设备线生成算法，它可用于其它曲线显示。

根据光栅扫描原理，线段离散过程中的每一放样位置上只可能有两个像素更接近于线段路径。Bresenham 算法引入一个整型参量来衡量“两候选像素与实际线路径点间的偏移关系”。Bresenham 算法通过对整型参量值符号的检测，选择候选像素中离实际线路径近的像素作为线的一个离散点

2. 决策参数

假设直线由 n+1 个点组成，且起点到终点的坐标分别为：(x_0,y_0)，(x_1,y_1)，…，(x_n,y_n)，第 k 步的决策参数为 p_k，则有纵坐标和决策参数的递推公式如下。

（因为对情况进行了处理，实际上只有四种情况，这主要利用了这样的判断 if(x_b>=x_e) { qSwap(x_b,x_e); qSwap(y_b,y_e);}）

① 情况一

$$p_0 = 2\triangle y - \triangle x$$

$$y_{k+1} = \begin{cases} y_k & p_k < 0 \\ y_k + 1 & p_k \geq 0 \end{cases}$$

$$p_{k+1} = \begin{cases} p_k + 2\triangle y & p_k < 0 \\ p_k + 2\triangle y - 2\triangle x & p_k \geq 0 \end{cases}$$

② 情况二

$$p_0 = 2\triangle y + \triangle x$$

$$y_{k+1} = \begin{cases} y_k - 1 & p_k < 0 \\ y_k & p_k \geq 0 \end{cases}$$

$$p_{k+1} = \begin{cases} p_k + 2\triangle y + 2\triangle x & p_k < 0 \\ p_k + 2\triangle y & p_k \geq 0 \end{cases}$$

③ 情况三

$$p_0 = 2\triangle x - \triangle y$$

$$x_{k+1} = \begin{cases} x_k & p_k < 0 \\ x_k + 1 & p_k \geq 0 \end{cases}$$

$$p_{k+1} = \begin{cases} p_k + 2\triangle x & p_k < 0 \\ p_k + 2\triangle x - 2\triangle y & p_k \geq 0 \end{cases}$$

④ 情况四

$$p_0 = 2\triangle x + \triangle y$$

$$x_{k+1} = \begin{cases} x_k - 1 & p_k < 0 \\ x_k & p_k \geq 0 \end{cases}$$

$$p_{k+1} = \begin{cases} p_k + 2\triangle x + 2\triangle y & p_k < 0 \\ p_k + 2\triangle x & p_k \geq 0 \end{cases}$$

3. 具体算法

|m|＜1 的 Bresenham 画线算法如下。

⑴. 输入线的两个端点，并将左端点存贮在(x0,y0)中；

⑵. 将(x0,y0)装入帧缓冲器，画第一个点；

⑶. 计算常量：△x、△y、2△y 和 2△y-2△x，起始位置(x0,y0)的决策参数 p0 计算为：p0=2△y - △x

⑷. 从 k=0 开始，在每个离散取样点 xk 处，进行下列检测：

若 pk< 0，画点(xk+1,yk)，且：pk+1= pk+2△y；

若 pk ＞0， 画点(xk+1,yk+1)，且：pk+1=pk+2△y-2△x。

⑸. k=k+1；

⑹. 重复步骤 4，共△x 次。

4. C++代码实现

```
void Line::DrawLineBresenham(QPainter &painter,QPoint begin,QPoint end){
    int x_b = begin.x();
    int y_b = begin.y();
    int x_e= end.x();
    int y_e= end.y();
    double delt_x=x_e-x_b;
    double delt_y=y_e-y_b;

    if(abs(delt_x)>abs(delt_y)){
        //斜率小于 1
        if(x_b>=x_e){
            qSwap(x_b,x_e);
            qSwap(y_b,y_e);
        }
        delt_x=x_e-x_b;
        delt_y=y_e-y_b;
        double x=x_b;
        double y=y_b;

        if(y_b<=y_e){
            double p=2*delt_y-delt_x;
            for(int i=0;i<=delt_x;i++){
                painter.drawPoint(x,y);
                x++;
                if(p>=0){
                    y++;
                    p=p+2*(delt_y-delt_x);
                }
                else{
                    p=p+2*delt_y;
                }
            }
        }
        else{
            //此处代码省略
        }
    }
    else{
        //斜率大于 1，此处代码省略
    }
}
```

5. 算法理解
上述代码省略了很多。
根据斜率和直线绘制的走向，一共有八种情况。在写代码时，可以通过 qSwap 交换点的位置，将八种情况转换为四种情况。而这四种情况并不是简单地简单地改变加减符号，是需要耐心计算的。
DDA 算法还会需要除法计算斜率并每次在参数上增加斜率的值，我觉得这会无形中增加误差，因为计算机在做除法时会有误差，而累加则放大了误差。
而 Bresenham 算法的比较不涉及除法，整数的增量可以减少误差。

◆ 圆形的绘制

中点圆算法

1. 思想
避免平方根运算，直接采用像素与圆距离的平方作为判决依据。通过检验两候选像素中点与圆周边界的相对位置关系(圆周边界的内或外)来选择像素

2. 优点
适应性强：易应用于其它圆锥曲线。
误差可控：对于整数圆半径，生成与 Bresenham 算法相同的像素位置。且所确定像素位置误差限制在半个像素以内。

3. 算法过程
- 输入圆半径 r 和圆心(x_c,y_c)，并得到圆心在原点的圆周上的第一点为$(x_0,y_0)=(0,r)$。
- 计算圆周点(0,r)的初始决策参数值为：$p_0=5/4-r$；
- 从 k=0 开始每个取样位置 x_k 位置处完成下列检测：
  若 $p_k<0$，选择像素位置：(x_k+1,y_k)，且：$p_{k+1}=p_k+2x_{k+1}+1$
  若 $p_k>0$，选择像素位置：(x_k+1,y_k-1)，且：$p_k+1=p_k+2x_{k+1}+1-2y_{k+1}$
  其中有 $2x_{k+1}=2x_k+2$，且 $2y_{k+1}=2y_k-2$
- 确定在其它七个八分圆中的对称点。
- 将计算出的像素位置(x,y)移动到中心在(x_c,y_c)的圆路径上，即：对像素位置进行平移：$x=x+x_c$，$y=y+y_c$；
- 重复上面的第 3 步骤到第 5 步骤，直至 $x\geq y$

4. C++实现
```
void Circle::DrawCircleCenter(QPainter &painter,QPoint begin,QPoint end){
    double tmp=qPow(begin.x()-end.x(),2)+qPow(begin.y()-end.y(),2);
    double radius = qFloor(qSqrt(tmp));          //算出半径
```

```
        int dx = begin.x();         //圆心与原点在 x 轴的偏移
        int dy = begin.y();         //圆心与原点在 y 轴的偏移
        int x = 0,y= radius;         //第一个点
        double p =3-2*radius;    //变量 d 初始值
        while(x<=y){
            //绘制每个 1/8 个圆
            painter.drawPoint(x+dx,y+dy);
            painter.drawPoint(y+dx,x+dy);
            painter.drawPoint(-x+dx,y+dy);
            painter.drawPoint(y+dx,-x+dy);
            painter.drawPoint(x+dx,-y+dy);
            painter.drawPoint(-y+dx,x+dy);
            painter.drawPoint(-x+dx,-y+dy);
            painter.drawPoint(-y+dx,-x+dy);
            if(p<0){
                //参考自 PPT
                p+=4*x+6;
                x++;
            }
            else{
                p+=4*(x-y)+10;
                y--;
                x++;
            }
        }
}
```

Bresenham 算法

1. 算法原理

在 0≤x≤y 的 1/8 圆周上，像素坐标 x 值单调增加，y 值单调减少。设第 i 步已确定（xi,yi）是要画圆上的象素点，看第 i+1 步象素点（xi+1,yi+1）应如何确定。下一个像素点只能是(xi+1,yi)或(xi+1,yi-1)中的一个

2. C++实现（部分核心如下）

```
        while(x<=y){
            //绘制每个 1/8 个圆，省略了 7 个部分
            painter.drawPoint(x+dx,y+dy);
            if(p<0){
                //参考自 PPT
                p=p+4*x+6;
                x++;
```

```
        }
        else{
            p=p+4*(x-y)+10;
            x++;
            y--;
        }
    }
```

◆ 椭圆的绘制

中点椭圆生成算法

1. 与圆的生成的比较

画椭圆的算法也采用了中点画椭圆法，与上面的画圆算法比较类似，不同之处主要在于决策参数计算方法不同以及区域分割方式不同。因为椭圆有长短轴之分，所以在绘制时分成四部分。以中心为原点，画完第一象限再对称变换到其他象限。第一象限中又分为两个区域，以斜率绝对值等于一为分界点。

中点圆的生成算法算是中点椭圆的特殊化。

2. 实现流程

- 输入 r_x、r_y 和中心(x_c,y_c)，得到中心在原点的椭圆上的第一个点：$(x_0,y_0)=(0,r_y)$；
- 区域 1 中决策参数的初值为：$p1_0=r_y^2-r_x^2r_y+ r_x^2/4$
- 在区域 1 中每个 x_k 位置处，从 k=0 开始循环测试：

  若 $p1_k<0$，椭圆的下一个离散点为(x_{k+1},y_k)，且 $p1_{k+1}=p1_k+2r_y^2x_{k+1}+r_y^2$。

  若 $p1_k>0$ ，椭圆的下一个离散点为(x_{k+1},y_k-1)

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

  其中：$2r_y^2x_{k+1}=2r_y^2x_k+2r_y^2$；$2r_x^2 y_{k+1}=2r_x^2y_k-2r_x^2$，该过程循环到：$2r_y^2x\geqslant 2r_x^2y$
- 区域 1 中最后点(x_1,y_1)计算区域 2 参数初值：$p^2_0=r_y^2(x_0+1/2)2+r_x^2 (y_0-1)-r_x^2r_y^2$
- 在区域 2 的每个 y_k 位置处，从 k=0 开始，完成下列检测：

  若 $p2_k>0$，椭圆下一点选为(x_k,y_k-1)，且 $p2_{k+1}=p2_k-2r_x^2y_{k+1}+r_x^2$,

  否则，沿椭圆的下一个点为(x_{k+1},y_k-1)，且 $p2_{k+1}=p2_k+2r_y^2 x_{k+1}-2r_x^2 y_{k+1}+r_x^2$,

  与区域 1 中相同的 x 和 y 增量计算

  循环至$(r_x,0)$
- 对称：确定其它三个象限中对称的点。
- 平移：将每个计算出的像素位置(x,y)平移到中心在(x_c,y_c)的椭圆轨迹上，并按坐标值画点

  $x=x+x_c$， $y=y+y_c$

3. C++实现

```
void Ellipse::DrawEllipse(QPainter *painter, QPoint &begin, QPoint &end){
        int x_b=begin.x();                                          //椭圆中心点
        int y_b=begin.y();
        int rx=abs(end.x()-x_b);                                    //椭圆长短轴
        int ry=abs(end.y()-y_b);
        double rx_2=rx*rx;    double ry_2=ry*ry;
        int x=0;    int y=ry;
        DrawEllipsePoints(painter,x_b,y_b,x,y);                      //画第一个点
        double p=ry_2- rx_2*ry + rx_2/4;                             //决策点
        while (ry_2*x <= rx_2*y){
              // 切线斜率 k<=1 的区域
              if (p < 0){
                    p += 2*ry_2*x +3*ry_2;
              }
              else{
                    p += 2*ry_2*x - 2*rx_2*y + 2*rx_2+ 3*ry_2;
                    y--;
              }
              x++;
              DrawEllipsePoints(painter,x_b, y_b, x, y);
        }
        p= ry_2*(x+1/2)*(x+1/2)+rx_2*(y-1)*(y-1)-rx_2*ry_2;    //决策点
        while (y > 0){
              // 切线斜率 k>1 的区域，在此省略
        }
}
```

4. 算法理解

Bresenham 的直线算法感觉和中点椭圆算法的出发思想很相似，都是在可能的两个点之间选择更近的一个，并且通过决策函数，减少复杂的计算量，完成增量计算。

而圆和椭圆均可以用到象限对称的思想很大程度的减少工作量。

◆ 多边形的绘制

多边形由直线构成，其基本算法同直线的算法，包括 DDA 和 Bresenham 算法

◆ 曲线的绘制

Bezier 曲线

1. 基函数

- n 次 Bernstein 基函数多项式形式：$BEZ_{i,n}(u)=C(n,i)u^i(1-u)^{n-i}$，
  其中：$C(n,i)=n!/[i!(n-i)!]$ (i=0,1,…,n)
- 递归定义：$BEZ_{i,n}(u)=(1-u)BEZ_{i,n-1}(u)+uBEZ_{i-1,n-1}(u)$
  (i=0,1,…,n)
  一个 n 次 Bernstein 基函数能表示成两个 n-1 次基函数的线性和

2. Bezier 曲线定义

$$P(u)=\sum_{k=0}^{n}P_k BEZ_{k,n}(u),\quad 0\le u\le 1$$

3. *Bézier* 曲线的拟合特性

Bézier 曲线段可拟合任何数目的控制点，而 *Bézier* 曲线段逼近这些控制点，通过控制多边形大致勾画 *Bézier* 曲线的形状。并且 *Bézier* 曲线以 P_0 为起点，以 P_n 为终点，过起始点。当修改某一控制顶点时，曲面上距它近的点受影响大，距它远的受影响小。这是它的拟局部性，但是它不具有局部性。

4. 优缺点

优点：几何意义明确；形状构造容易；形状控制简单。
最大缺点：局部形状控制能力差；曲线曲面拟合误差大。

5. C++代码实现

```
void Curve::CurveBezier(QPainter &painter,vector<QPointF>p){
    int pointsnum=p.size();
    if(pointsnum==1){
        painter.drawPoint(p[0]);
    }
    else if(pointsnum>1){
        // 如果将每个部分这么表示 a * (1 - t)^b * t^c * Pn;
        // 首先杨辉三角形计算 a 的值
        int *a=new int[pointsnum];
        a[0]=1;
        a[1]=1;
        for(int i=3;i<=pointsnum;i++){
```

```
                int *t=new int[i-1];
                for(int j=0;j<i-1;j++){
                    t[j]=a[j];
                }
                a[0]=1;
                a[i-1]=1;
                for(int j=0;j<i-2;j++){
                    a[j+1]=t[j]+t[j+1];
                }
            }
            //然后画出点  prec 精度表示画多少个点
            int prec=pointsnum*200;
            for(int i=0;i<prec;i++){
                //  画 prec 个点
                double u=(double)i/(double)prec;
                double x=0;
                double y=0;
                for(int k=0;k<pointsnum;k++){
                    //  每个点是 p0+p1+p2 按权重来的
                    double pre=pow(1-u,pointsnum-k-1) * pow(u,k) * a[k];
                    x+= pre* (p[k].x());
                    y+= pre* (p[k].y());
                }
                painter.drawPoint(QPoint(x+0.5,y+0.5));
            }
        }
}
```

6. 代码思考

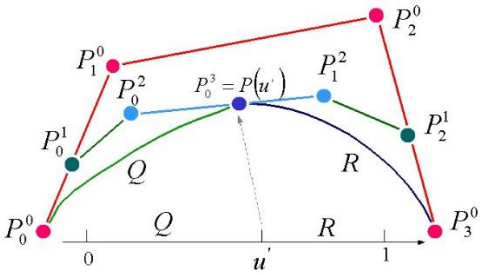
在写代码时，计算 P_i 可以通过递归定义不断重复计算，但是那样计算的话会开销很大。就是计算如下公式。

$$B(t)=\sum_{i=0}^{n}\binom{n}{i}Pi(1-t)^{n-i}t^{i}=\binom{n}{0}P0(1-t)^{n}t^{0}+\binom{n}{1}P1(1-t)^{n-1}t^{1}+\ldots+\binom{n}{n-1}Pn-1(1-t)^{1}t^{n-1}+\binom{n}{n}Pn(1-t)^{0}t^{n},t\in[0,1]$$

一种可能的解决方法是，对于存在的 $a * (1 - t)^b * t^c * P_n$ 形式，利用杨辉三角形递归计算，可以减少很大的开销。

7. 代码阅读

在网上看到了另外一种 Bezier 曲线的实现方式。与套用定义定义公式不同，该代码，对 Bezier 曲线不断二分，即 u 取 1/2，然后每次切割曲线，直至近乎完全连续为止。



如上图，第一次求出点p_0^3，u=1/2，然后如此依次画出$p_0^0p_0^3$和$p_0^3p_3^0$之间的曲线。

具体的网上的代码如下。

```
void CBezierView::DrawBzier(DPoint * p){
    if (n<= 0)
        return;
    if((p[n].x<p[0].x+1)&&(p[n].x>p[0].x-1)&&(p[n].y<p[0].y+1)&&(p[n].y>p[0].y-1)){
        pDC->SetPixel(p[0].x, p[0].y, RGB(0,0,255));
        return;
    }
    DPoint *p1;
    p1 = new DPoint[n+1];
    int i, j;
    p1[0] = p[0];
    for(i=1; i<=n; i++){
        for(j=0; j<=n-i;j++){
            p[j].x = (p[j].x + p[j+1].x)/2;
            p[j].y = (p[j].y + p[j+1].y)/2;
        }
        p1[i] = p[0];
    }
    DrawBzier(p);
    DrawBzier(p1);
    delete p1;
}
```

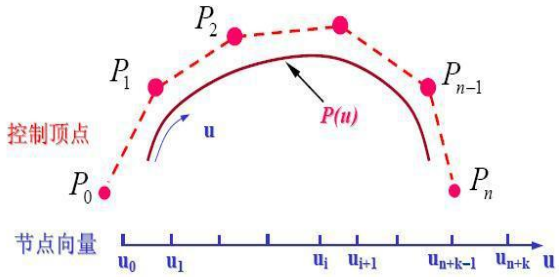
B-spline

1. B 样条曲线定义

给定 n+1 个控制顶点{P_i}(i=0,1,…,n)，$P_0P_1…P_n$ 为控制多边形，共 n+k+1 个参数节点向量：$U_{n,k}=\{u_i | i=0,1,…,n+k，u_i \le u_{i+1}\}$。称如下形式的参数曲线 P(u)为 k 阶(k-1 次)B 样条曲线：

$$P(u)=\sum_{i=0}^{n} P_i B_{i,k}(u), u \in [u_k, u_n]$$

其中，$B_{i,k}(u)$为 k 阶(k-1 次)B 样条基函数。$B_{i,k}(u)$双下标中下标 k 表示 k 阶(k-1 次)数，而下标 i 表示序号。



2. de Boor-Cox 递推定义：

$$B_{i,1}(u)=\begin{cases}1 & u_i < x < u_{i+1}\\ 0 & Otherwise\end{cases}$$

其中，$B_{i,1}(u)$是 0 次多项式，0 次多项式是常数。

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

de Boor-Cox 的原理是，只要是 k 次的 B 样条基函数，构造一个递推的公式，由 0 次多项式（常数）的递推构造 1 次的，1 次的递推构造 2 次的等等。B 样条基函数 $B_{i,k}(u)$是定义在一组节点向量上的基函数，B 样条基函数不像伯恩斯坦基函数那样简单地只和 u 有关，在一个区间里面，而是和一组节点向量有关，节点向量不一样，基函数也不一样。

3. 优缺点

主要优点：

局部控制能力强；

拟合精度高。

最大缺点：

形状构造过程复杂；

初等函数形状拟合能力差。

4. 与 Bezier 的比较

Bézier 曲线曲面具有许多优点，如凸包性、保型性等，但也存在不足之处：其一是缺少局部性，修改某一控制顶点将影响整条曲线；其二是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；其三是当表示复杂形状时，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 基函数而获得的 B 样条(Basic-spline)曲线曲面克服了上述缺点。

具体从如下方面进行比较。

基函数的次数：对于 Bézier 曲线，基函数的次数等于控制顶点数减 1；对于 B 样条曲线，基函数的次数与控制顶点数无关。

基函数性质：Bézier 曲线的基函数，即 Bernstein 基函数，是多项式函数；B 样条曲线的基函数，即 B 样条基函数，是多项式样条。

曲线性质：Bézier 曲线是一种特殊表示形式的参数多项式曲线；B 样条曲线则是一种特殊表示形式的参数样条曲线。

局部控制能力：Bézier 曲线缺乏局部性质；B 样条曲线具有局部性质。

5. 实现细节

实验中要求是三次 B 样条曲线。

我的代码实现中是通过递归计算，开销较大。

6. C++代码实现

```
double polynomial(int k,int d,double u){
    if(d==1){
        if( u>=(double)(k) && u<=(double)(k+1) ){
            return 1.0;
        }
        else{
            return 0.0;
        }
    }
    else{
        return
(u-k)/(d-1)*polynomial(k,d-1,u)+(k+d-u)/(d-1)*polynomial(k+1,d-1,u);
    }
}
void Curve::CurveBspline(QPainter &painter,vector<QPoint> points){
    int degree=4;                    // degree 为 4，那么次数为 d-1 即 3
    int pointnum=points.size();      // 控制点的数目 s
    double u; double du=0.001;       // u 和 u 的变化精度
    for(u=degree-1;u<=pointnum;u=u+du){   // u 范围是 [d-1,n+1]
        QPointF toDraw(0.0,0.0);
        double bkd;
        for(int k=0;k<pointnum;k++){
            bkd=polynomial(k,degree,u);
            toDraw.setX(toDraw.x()+double(points[k].x())*bkd);
            toDraw.setY(toDraw.y()+double(points[k].y())*bkd);
        }
        painter.drawPoint(toDraw.x()+0.5,toDraw.y()+0.5);
    }
}
```

图元的变换算法/实现

◆ 平移

1. 基本概念

平移即将图形改变位置，形状不变的图元变换算法

2. 实现思路

translate id dx dy，对于命令行界面，只要将每个图形的关键点的位置修改dx/dy，那么就可以得到修改以后的图的关键点，然后依据新的关键点重新绘制图元

对于鼠标点击，只要获得平移结束关键点的位置，通过简单的做差运算，就可以自主算得 dx/dy，那么就同样可以修改以后的图的关键点，然后依据新的关键点重新绘制图元

原位置(x,y)，结束的位置(x',y')，那么有 x'=x+dx，y'=y+dy

3. 算法原理

平移是将物体沿着直线路径从一个坐标位置到另一个坐标位置重定位。对于原始位置 P(x,y)平移 d_x 和 d_y 到新位置 P(x_1,y_1)的移动满足

$x_1 = x+d_x$

$y_1 = y+d_y$

4. C++实现

```
void AllFigure::translate(int xc, int yc){
    int PointNum=infoPoint.size();
    for(int i=0;i<PointNum;i++){
        infoPoint[i].setX(infoPoint[i].x()+xc);
        infoPoint[i].setY(infoPoint[i].y()+yc);
    }
}
```

◆ 旋转

1. 算法原理

对于点（x，y）绕（xc，yc）旋转θ得到（x’，y’），基本方法是：

$x'=x_c+(x-x_c)*\cos\theta-(y-y_c)*\sin\theta$
$y'=y_c+(x-x_c)*\sin\theta+(y-y_c)*\cos\theta$

2. 实现过程

旋转是沿定旋转基准点位置旋转某个角度，将所有图元上的点的位置按数学公式进行一定的变化，只要计算出每个关键点的位置，然后按数学关系修改关键点位置，那么就可以根据新的关键点重新画出旋转以后的图

3. C++代码

```
void AllFigure::rotate(int angle, double xc, double yc){
        double radian = qDegreesToRadians(double(angle-180));
        int PointNum=infoPoint.size();
        for(int i=0;i<PointNum;i++){
                double x=infoPoint[i].x();
                double y=infoPoint[i].y();
                infoPoint[i].setX(xc+(x-xc)*qCos(radian)-(y-yc)*qSin(radian)+0.5);
                infoPoint[i].setY(yc+(x-xc)*qSin(radian)+(y-yc)*qCos(radian)+0.5);
        }
}
```

◆ 缩放

1. 算法原理

对于点（x，y），绕中点(x_c,y_c)按比例 multi 变化为(x',y')：

$x'=x_c+(x-x_c)*$ multi

$y'=y_c+(y-y_c)*$ multi

2. 实现思路

缩放其实是将图形乘以一个缩放系数，那么只要将每个图形的关键点距离缩放中心的比例计算出来，然后修改关键点位置，即乘以缩放系数，那么就可以根据新的关键点重新画出缩放以后的图

3. C++实现

```
void AllFigure::scale(double multi,double xc,double yc){
        int PointNum=infoPoint.size();
        for(int i=0;i<PointNum;i++){
                double x=infoPoint[i].x();
                double y=infoPoint[i].y();
                x=x*multi+xc*(1-multi);
```

```
            y=y*multi+yc*(1-multi);
            infoPoint[i].setX(x+0.5);
            infoPoint[i].setY(y+0.5);
        }
}
```

◆ 裁剪

Cohen-Suther land 算法

1. 算法原理
将窗口区域分为 9 个部分，每个部分给一个区域码，然后计算线段两端端点的区域码，根据区域码来选择抛弃线段。如下。

• 完全在窗口边界内的线段：两端点区域码均为 0000；
• 完全在裁剪矩形外的线段：两端点区域码同样位置都为 1。对两个端点区域码进行逻辑与操作，结果不为 0000。
• 不能确定完全在窗口内外的线段，进行求交运算： 按“左-右-上-下”顺序用裁剪边界检查线段端点。将线段的外端点与裁剪边界进行比较和求交，确定应裁剪掉的线段部分；反复对线段的剩下部分与其它裁剪边界进行比较和求交，直到该线段完全被舍弃或找到位于窗口内的一段线段为止。

2. 核心思想
通过编码测试来减少要计算交点的次数。

3. C++实现
代码具体参考自该网站
https://www.jianshu.com/p/d512116bbbf3

Liang-Barsky 裁剪算法

1. 算法原理
• 设待裁剪的线段为 AB，其中 A(x_1，y_1)，B（x_2，y_2）
• 设变量 p_i 表示第 i 侧 AB 向量从内到外/从外到内。i∈0~3，分别表示左、右、

上、下边界。则：$p_0=-\Delta x$　$p_1=\Delta x$　$p_2=-\Delta y$　$p_3=\Delta y$

• 定义变量 q_i 如下：

$q_0=x_1-xw_{min}$

$q_1=xw_{max}-x_1$

$q_2=y_1-yw_{min}$

$q_3=yw_{max}-y_1$

• 设 r=qi/pi，则 r 为直线与第 i 个边界交点对应的参数值。

• 设直线参数为 u，线段裁剪后的端点对应参数为 u1 与 u2，其中 u1 更小。

• 遍历四个方向的 p 和 q 的值

若 $p_i=0$，说明线段与该边界平行。

若 qi<0，说明线段整个都在外部，可以直接 return false，否则不做处理，continue。

若 $p_i<0$，说明 AB 从外到内穿过边界，u1 取 u1 和 r 中的较大值

若 $p_i>0$，说明 AB 从内到外穿过边界，u2 取 u2 和 r 中的较小值

• 遍历结束后若 u1>u2，则说明整个线段都不在窗口内。可以直接舍弃。否则保留该区间之内的线段。

2. C++代码实现

```
for(int i=0;i<4;i++){
        //p=0 且 q＜0 时，线段被裁掉
      if(fabs(p[i])<1e-6){
            if(q[i]<0){
                infoPoint.clear();
                break;
            }
      }
      else{
            double r = q[i]/p[i];
            if(p[i]<0){
                u1 = r>u1?r:u1; //u1 取 0 和各个 r 值之中的最大值
            }else{
                u2 = r<u2?r:u2; //u2 取 1 和各个 r 值之中的最小值
            }           //如果 u1>u2，则线段完全落在裁剪窗口之外，应当被舍弃
            if(u1>u2){
                id=-1;
                infoPoint.clear();
            }
      }
      QPoint newp1(x1+int(u1*dx+0.5), y1+int(u1*dy+0.5));
      infoPoint[0]=newp1;
      QPoint newp2(x1+int(u2*dx+0.5), y1+int(u2*dy+0.5));
      infoPoint[1]=newp2;
}
```

◆ 系统框架

◆ 主窗口

MainWindow 主窗口

- → MdiArea 多文档区域
- → QMdiSubWindow 多个子窗口区域
  - → Canvas 画布

◆ 画布

Canvas 画布

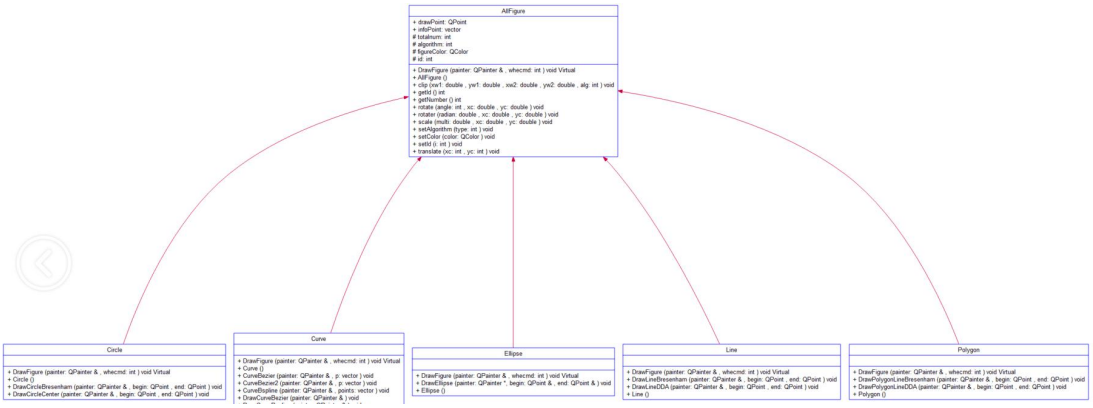
- → QPixmap *CanvasPix 实画布
- → QPixmap *TempPix 虚画布（用于缓存）
  - → AllFigure 各个图元

◆ 图元控制

AllFigure 父类控制各个图元，设置图元颜色/绘制图元/平移操作

- → Line 直线绘图
- → Circle 圆形绘图
- → Ellipse 椭圆绘图
- → Polygon 多边形
- → Curve 曲线

UML 图如下



文件安排

文件名称	文件内容
main.cpp	主函数
Mainwindow.h/cpp	主窗口
Canvas.h/cpp	画布类
Allfigure.h/cpp	图形-所有图元父类
Line.h/cpp	图形-直线类
Circle.h/cpp	图形-圆类
Ellipse.h/cpp	图形-椭圆类
Polygon.h/cpp	图形-多变形类
Curve.h/cpp	图形-曲线类
CanvasSet.h/cpp	窗口-设置画布
Clipalgorithm.h/cpp	窗口-设置裁剪算法
Curvealgorithm.h/cpp	窗口-设置曲线算法
Linealgorithm.h/cpp	窗口-设置直线算法
Polygonalgorithm.h/cpp	窗口-设置多边形算法
Cmdcontrol.h/cpp	控制-解析命令行指令

◆ 总结

◆ 10 月 31 日小结

1. 熟悉了 Qt 的使用流程，能够使用 Qt 新建 UI 项目。
2. 能使用当前已经学习的基本知识绘制图元，对计算机图形的图元生成的理解有了更多的理解
3. 在代码整体的构建中，用到了 C++许多关于类的知识，对类的继承/虚函数等使用有了进一步的理解
4. 总结第一阶段：将整个代码的框架基本有所构建，实现了画布/窗口画图/保存清空画布/基本图元构建/选中平移操作等功能

◆ 11 月 30 日小结

1. 将图形的旋转缩放，线段的裁剪进行了代码的书写。
2. 完成了读入文件，对文件中命令行进行解析的功能。
3. 修改了很多原来存在的 bug。
4. 主要在原来的基础上进行修改。所以很多结构性的东西都进行了修改。所以在最开始设定代码框架的时候，就应该有规划要怎么写。
5. 对于图形界面的旋转缩放等功能还需要完成。

◆ 12 月 31 日小结

1. 将图形的旋转、缩放、裁剪完成了图形界面的书写，并且优化了界面，提供了赋值线条便于交互。
2. 完成了 Bezier 曲线和三次 B 样条曲线的代码，并提供了控制多边形的可视化。
3. 增加了删除图元，修改图元控制点从而修改图元的功能。
4. 全面优化了界面和细节。处理了 bug。

◆ 小结

1. 对图形学中的图元生成算法和变换算法有了更深的学习与掌握。
2. 在整个代码的书写中，掌握了很多方法，进一步锻炼了写代码的能力。

◆ 参考资料

- https://www.bilibili.com/video/av9722792（Qt 入门）
- https://blog.csdn.net/qq_41453285/article/details/91628383（QMessageBox 的使用）
- https://blog.csdn.net/naibozhuan3744/article/details/79166653（设置画布/背景）
- https://www.cnblogs.com/Jace-Lee/p/6055078.html（主窗口/子窗口）
- https://www.cnblogs.com/azbane/p/8656427.html（多文档界面）
- https://blog.csdn.net/qq_35263780/article/details/7781595（initial/resize/paint）
- http://www.qiliang.net/old/nehe_qt/lesson01.html（initial/resize/paint）
- https://www.cnblogs.com/lomper/p/3954660.html（点击界面跳转）
- https://blog.csdn.net/weixin_34244102/article/details/91992432（Bresenham 直线参考）
- https://www.cnblogs.com/keguniang/p/9688126.html（直线裁剪算法）
- https://www.jianshu.com/p/d512116bbbf3（Cohen–Sutherland 算法）
- https://blog.csdn.net/syx1065001748/article/details/70313523（Bezier 贝塞尔曲线）
- https://blog.csdn.net/mylovestart/article/details/8434310（Bezier 贝塞尔曲线）
- https://blog.csdn.net/Mahabharata_/article/details/71856907（B 样条曲线）
- https://blog.csdn.net/liumangmao1314/article/details/54588155（B 样条曲线）