《计算机图形学》11月进展报告

欧阳鸿荣 161220096

（南京大学 计算机科学与技术系 南京 210093）

1.综述

系统名	语言和框架	IDE	编译器
YoungPaint	C++和Qt 5.11.2	Qt Creator	MinGW 5.3.0

截止11月底，我项目目前能完成的功能是：

一、基本功能：

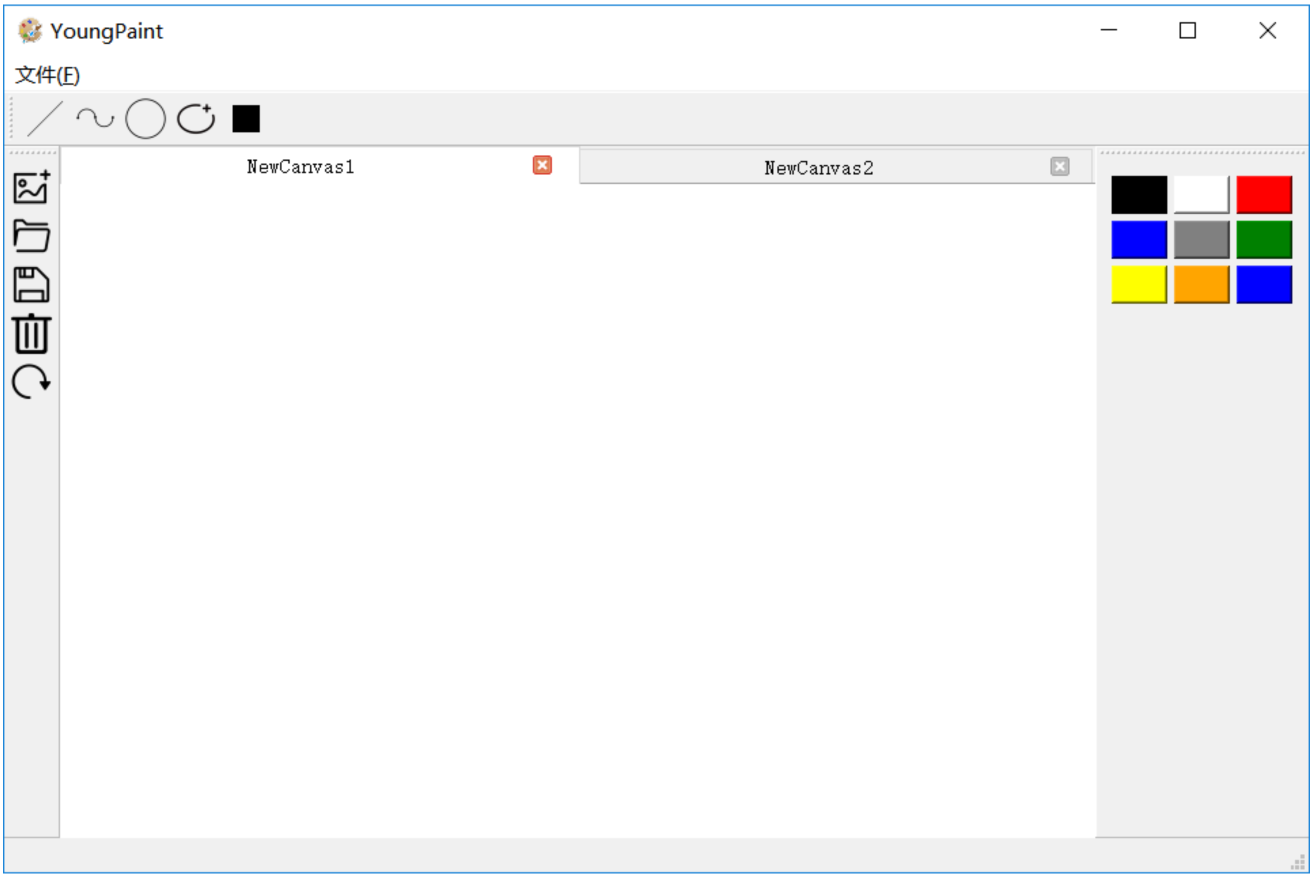
1. **二维图形的输入功能：**
    - 直线、圆，椭圆，多边形的输入实现
        - 类画图软件，用鼠标交互
    - 填充区域的输入
        - 实现了类似油漆桶的功能
        - 鼠标点击区域，洪泛填充与区域颜色相同的区域
2. **二维图形的编辑功能：**
    - 直线、圆，椭圆，多边形的编辑
        - 直线能编辑起点、终点
        - 圆能编辑半径
        - 椭圆能编辑长轴a和短轴b的长度
        - 多边形能编辑任意顶点

            - 鼠标点击拖动交互编辑
    - 二维图形的裁剪功能：
        - 直线的裁剪
            - 使用梁友栋算法
        - 裁剪窗口可用鼠标点击拖动输入
        - 裁剪后的图形仍然可以编辑
3. **二维图形的变换功能**
    - 直线、圆、椭圆、多边形的平移
    - 直线、圆、椭圆、多边形的旋转
        - 任意角度旋转
        - 直线、圆的旋转实现了精度控制
            - 旋转次数不多的情况下，长度/半径误差在1以内
    - 直线、圆、椭圆、多边形的缩放
    - 对变换后的图形仍然可以编辑
4. **二维图形的存储功能**
    - 讲绘制出来的图形保存为图像

二、扩展功能：

1. 画布的创建，多画布切换
2. 颜色的选择
3. 增加了画笔和笔刷的功能
4. 清屏和撤销的功能

将项目打包成了一个.exe文件，放在161220096_系统工程\Demo\YoungPaint里供助教测试，虽然在很多人电脑里测试都可以运行，但是程序依赖是否都有这个还不得而知，权当给助教一个参考。


YoungPaint
文件(F)
NewCanvas1
NewCanvas2

2.算法介绍

(1).直线算法 —— DDA直线算法

(a)基本原理

数值差分分析DDA(digital differential analyzer)：直接利用△x或△y的线段扫描转换算法，利用光栅特性(屏幕单位网格表示像素列阵)，使用x或y方向单位增量(△x或△y=±1)来离散取样，并逐步计算沿线路径各像素位置。在一个坐标轴上以单位间隔对线段离散取样，确定另一个坐标轴上最靠近线段路径的对应整数值。

(b)理论绘制过程

1.具有正斜率

1.1 从左到右生成线段。

若斜率 $m \leq 1$，在x方向以单位间隔($\triangle\ x = 1$)取样，以增量形式顺序计算每个y值：

$$y_{k+1} = y_k + m \quad (1)$$

下标k取整数值从k=1开始递增。m为0与1间的任意实数，计算出的坐标值必须取整。

1.2 从右端点到左端点生成线段

取$\triangle\ x = -1$

$$y_{k+1} = y_k - m \quad (2)$$

1.3 若斜率m>1，从左到右生成线段。

在y方向以单位间隔$\triangle\ y = 1$取样，顺序计算每个x值：

$$x_{k+1} = x_k + \frac{1}{m} \quad (3)$$

下标k取整数值从1开始递增，直至最后端点。计算出的坐标值必须取整。

1.4 若从右到左生成线段

取$\triangle\ y = -1$

$$x_{k+1} = x_k - \frac{1}{m} \quad (4)$$

2.具有负斜率

2.1 假如斜率的绝对值小于1时，

- 起始端点在左侧，设置$\triangle\ x = 1$并用方程(1)计算y；
- 起始端点在右侧，设置$\triangle\ x = -1$并用方程(2)得到y

2.2 假如斜率的绝对值大于1时，

- 起始端点在左侧，用$\triangle\ y = 1$和方程(3)，
- 当起始端点在右侧，$\triangle\ y = -1$和方程(4)。

(c)算法的C++实现

```
void LineController::MyDrawLineDDA(QPainter *painter, QPoint &start, QPoint &end)
{
    //首先先在这里实现我的直线算法
    qDebug()<<"MyDrawLine DDA"<<endl;

    int x1 = start.x();
    int y1 = start.y();
    int x2 = end.x();
    int y2 = end.y();

    double dx=x2-x1;
    double dy=y2-y1;
    double e=(fabs(dx)>fabs(dy))?fabs(dx):fabs(dy);
    double x=x1;
    double y=y1;

    dx/=e;
    dy/=e;

    for(int i=1;i<=e;i++){

        QPoint tempPt((int)(x+0.5), (int)(y+0.5));
        painter->drawPoint(tempPt);

        x+=dx;
        y+=dy;
    }
}
```

(2).圆算法 —— 中点圆算法

(a)基本原理

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。

(b)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。

(c)理论绘制过程

根据圆的对称性，只绘制了八分之一圆，其余部分通过对称性即可得到坐标。使用经过改良的中点圆算法，使用递推，减少了计算量，并且避免了浮点运算

算法过程

1. 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 $(x_0, y_0) = (0, r)$
2. 计算圆周点$(0, r)$的初始决策参数值为：$p_0 = \frac{5}{4} - r$
3. 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：$2x_{k+1} = 2x_k + 2, 且\ 2y_{k+1} = 2y_k - 2$
4. 确定在其它七个八分圆中的对称点
5. 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

    $x = x + x_c, y = y + y_c$

    重复步骤3到5，直到$x > y$

(d)算法的C++实现

```
//中点圆算法
void CycleController::MyDrawCycle(QPainter *painter, QPoint &start, QPoint &end)
{
    //首先先在这里实现我的画圆算法
    qDebug()<<"MyDrawCycle "<<endl;
    int x0 = start.x();
    int y0 = start.y();
    double R = this->getLength(start,end);
    int x,y,p;
    x=0;
    y=R;
    p=3-2*R;
    for(;x<=y;x++)
    {
        this->drawEighthCycle(painter,x0,y0,x,y);
        if(p>=0){
           p+=4*(x-y)+10;
           y--;
        }else{
            p+=4*x+6;
        }
    }
}
//由于中点圆算法只对八分之一圆进行绘制，因此用该函数将八分之一圆投射成完整的圆
void CycleController::drawEighthCycle(QPainter *painter, int x0, int y0, int x, int y)
{
    QPoint tempPt1(x0+x,y0+y);QPoint tempPt2(x0+y,y0+x);
    QPoint tempPt3(x0+x,y0-y);QPoint tempPt4(x0+y,y0-x);
    QPoint tempPt5(x0-x,y0-y);QPoint tempPt6(x0-y,y0-x);
    QPoint tempPt7(x0-x,y0+y);QPoint tempPt8(x0-y,y0+x);

    painter->drawPoint(tempPt1);painter->drawPoint(tempPt2);
    painter->drawPoint(tempPt3);painter->drawPoint(tempPt4);
    painter->drawPoint(tempPt5);painter->drawPoint(tempPt6);
    painter->drawPoint(tempPt7);painter->drawPoint(tempPt8);
}
```

(3).椭圆算法 —— 中点椭圆算法

(a)基本原理

考虑椭圆沿长轴和短轴不同而修改圆生成算法来实现椭圆生成。

给定长短轴参数r_x, r_y(假设$r_x < r_y$)和椭圆中心位置(x_c, y_c)，利用平移：先确定中心在原点的标准位置的椭圆点(x, y)；然后将点变换为圆心在(x_c, y_c)的点。再利用对称性：生成第一象限内的椭圆弧，再利用对称性求出其它三个象限的对应点。

(b)理论绘制过程

根据圆的对称性，只绘制了八分之一圆，其余部分通过对称性即可得到坐标。使用经过改良的中点圆算法，使用递推，减少了计算量，并且避免了浮点运算

算法过程

1. 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：$(x_0, y_0) = (0, r_y)$
2. 计算区域1中决策参数的初值为: $p1_0 = r_y^2 - r_x^2 r_y + r_x^2/4$
3. 在区域1中每个x_k位置处，从k=0开始，完成下列测试
    - 若p_{lk} < 0，椭圆的下一个离散点为(x_{k+1}, y_k)， 且：$pl_{k+1} = pl_k + 2r_y^2 x_k + 1 + r_y^2$。
    - 若$p_{lk} > 0$，椭圆的下一个离散点为$(x_{k+1}, y_k - 1)$， 且：$p1k + 1 = pl_k + 2r_y^2 x_k - 2r_x^2 y_{k+1} + r_y^2$
    - 其中有：$2r_y^2 x_{k+1} = 2r_y^2 x_k + 2r_y^2$；$2r_x^2 y_{k+1} = 2r_x^2 y_k - 2r_x^2$
    - 循环到：$2r_y^x \geq 2r_x^2 y$
4. 使用区域1中最后点(x_0, y_0)计算区域2参数初值为$p2_0 = r_y^2(x_0 + 1/2)^2 + r_x^2(y_0 - 1) - r_x^2 r_y^2$
5. 在区域2的每个y_k位置处，从k=0开始，完成下列检测
    - •假如$p2_k > 0$，椭圆下一点选为$(x_k, y_k - 1)$且：$p2_{k+1} = p2_k + 3r_x^2 - 2r_x^2 y$,
    - •否则，沿椭圆的下一个点为$(x_{k+1}, y_k - 1)$, 且：$p2_{k+1} = p2_k + 2r_y^2 x - 2r_x^y + 2r_y^2 + 3r_x^2$
    - 使用与区域1中相同的x和y增量计算
    - 循环直至$(r_x, 0)$
6. 确定其它三个像限中对称的点。
7. 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，并且按照坐标之绘点$x = x + x_c, y = y + y_c$

(c)算法的C++实现

```
//由于中点椭圆算法只对四分之一椭圆进行绘制，因此用该函数将四分之一椭圆投射成完整的椭圆
void EllipseController::drawQuarterEllipse(QPainter *painter, int x0, int y0, int x,
int y)
{
    QPoint tempPt1(x0+x,y0+y);
    QPoint tempPt2(x0+x,y0-y);
    QPoint tempPt3(x0-x,y0+y);
    QPoint tempPt4(x0-x,y0-y);

    painter->drawPoint(tempPt1);
    painter->drawPoint(tempPt2);
    painter->drawPoint(tempPt3);
    painter->drawPoint(tempPt4);
}

//中点椭圆算法
void EllipseController::MyDrawEllipse(QPainter *painter, QPoint &start, QPoint &end)
{
    //首先先在这里实现我的椭圆算法
    qDebug()<<"MyDrawEllipse "<<endl;
    int x0 = start.x(); //椭圆中心
    int y0 = start.y();
    int rx = abs(end.x()-x0); //椭圆长短轴
    int ry = abs(end.y()-y0);

    double rx_2 = rx*rx;
    double ry_2 = ry*ry;

    double pl = ry_2 - rx_2*ry + rx_2/4; //区域1中决策参数
    int x = 0;
    int y = ry;
    drawQuarterEllipse(painter,x0, y0, x, y);//第一个点

    //区域一 切线斜率k<=1
    while (ry_2*x <= rx_2*y){

        if (pl < 0){
            pl += 2*ry_2*x +3*ry_2;
        }else{
            pl += 2*ry_2*x - 2*rx_2*y + 2*rx_2+ 3*ry_2;
            y--;
        }
        x++;
        drawQuarterEllipse(painter,x0, y0, x, y);
    }
    //区域二 切线斜率k > 1
    //使用区域1中最后点(x0,y0)来计算区域2中参数初值
    pl = ry_2*(x+1/2)*(x+1/2)+rx_2*(y-1)*(y-1)-rx_2*ry_2;

    while (y > 0){
```

```
        if (p1 < 0){
            p1 += 2*ry_2*x - 2*rx_2*y + 2*ry_2 + 3*rx_2;
            x++;
        }
        else{
            p1 += 3*rx_2 - 2*rx_2*y;
        }
        y--;
        drawQuarterEllipse(painter,x0, y0, x, y);
    }
}
```

(4).平移变换算法

(a)基本原理

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

$$
\begin{aligned}
x1 &= x + d_x \\
y1 &= y + d_y
\end{aligned}
$$

(b)算法实现

给每一个图形都定义一个中心点（一般是对称中心或外接矩形的中心），绘制时加粗显示辅助，用户通过鼠标拖动中心点得到偏移量dx和dy。对于不同图形的平移，采取不同的实现方法：

1. 直线：对线上的起始点和顶点平移dx和dy，然后重新绘制
2. 圆：平移圆心和圆周上的点（决定半径），然后重新绘制
3. 椭圆：平移中心和外界矩形的一个顶点（决定a、b），然后重新绘制
4. 多边形：平移所有的顶点，然后重新绘制

####

(5).旋转变换算法

(a)基本原理

旋转是沿xy平面内圆弧路径重定位。指定旋转基准点位置(x_r,y_r)旋转θ角，对任意基准位置的旋转

$$
\begin{aligned}
x1 &= x_r + (x-x_r)cos\theta - (y-y_r)sin\theta \\
y1 &= y_r + (x-x_r)sin\theta + (y-y_r)cos\theta
\end{aligned}
$$

(b)算法实现

给每一个图形都定义一个旋转点，绘制时加粗显示辅助，用户通过鼠标拖动中心点得到旋转角θ

对于不同图形的旋转，采取不同的实现方法：

1. 直线：直线的旋转点定义为四等分点。
   - 用户操作旋转点，得到旋转角，然后对直线的两端点旋转，然后重新绘制
2. 圆：圆的旋转点定义为在$r/2$的位置。（圆的旋转没必要就是了）
   - 用户操作旋转点，得到旋转角，然后对圆周上的点旋转，然后重新绘制
3. 椭圆：椭圆的旋转点相对于其他图形略有不同。
   - 由于椭圆的中点椭圆算法只能实现对称轴垂直或者平行坐标轴的椭圆，因此在不引入其他算法的情况下，椭圆的旋转我定义了一个“偏向角α”来付诸实现。
     - α的初始值为0，为了方便叙述，假设椭圆中心为$(0,0)$，对旋转点(x_m,y_m)，为向量$(0,1)$与$(x_m,y_m,)$的夹角，而实际处理时为了方便，当夹角超过$\pi/2$时，对α取负
     - 从而每次旋转时，只是对椭圆的旋转角做出了改变，但是实际上椭圆Ellipse类的对象中椭圆的关键信息并没有改变，只是每次绘制时，对椭圆的每个顶点都旋转α角，这样就实现了旋转。与此同时，编辑的时候对编辑点也通过α转化为未旋转前椭圆的编辑，这样让旋转后的椭圆也能够编辑

- 但是此方法有一个缺陷，就是椭圆的旋转由于是对点的旋转实现的，因此椭圆会变得稀疏，或者说变成不连续的椭圆。这个目前还在想办法解决。

4. 多边形：多边形的旋转点定义与椭圆类似，但是也有区别
    - 由于多边形的绘制是对各个端点连线，因此每次都对图形旋转α的差值，但是旋转时保持旋转中心不动

(C) C++实现实现

这里主要介绍旋转时用到的一些辅助函数

(1) 角度函数：对于点A绕着点CENTER旋转到点B的情况，用余弦定理得到角β

```
double FigureController::getRotaryAngle(Point center, Point a, Point b)
{
    double ab = a.distanceToPoint(b.getQPoint());
    double ac = a.distanceToPoint(center.getQPoint());
    double bc = b.distanceToPoint(center.getQPoint());
    qreal cosC = (bc*bc+ac*ac-ab*ab)/(2*bc*ac);
    double theta = qAcos(cosC);
    return theta; //弧度制
}
```

(2) 顺逆时针判断函数：β这并不是真正的旋转角，故通过判断顺逆时针来对β进行修正得到θ角（通过线性规划）

```
bool FigureController::clockWise(Point center, Point a, Point b)
{
    //k=0
    if(center.getY()==a.getY()){
        if(a.getX()>center.getX())// -------->型向量
        {
            if(b.getY() > a.getY()){
                return true;
            }else{
                return false;
            }
        }
        else// <--------型向量
        {
            if(b.getY() < a.getY()){
                return true;
            }else{
                return false;
            }
        }
    }
    //k不存在
    if(center.getX()==a.getX()){
        if(a.getY()<center.getY())//竖直向上
        {
            if(b.getX()>a.getX()){
                return true;
            }else{
                return  false;
```

```
            }
        }
        else //竖直向下
        {
            if(b.getX()<a.getX()){
                return true;
            }else{
                return  false;
            }
        }
    }
    //斜率存在切不为零
    double x0 = 0;
    double y0 = 0;
    double x1 = a.getQPoint().x() - center.getQPoint().x(); //把中点当原点
    double y1 = a.getQPoint().y() - center.getQPoint().y(); //把中点当原点
    double k = (y1-y0)/(x1-x0);
    double x2 = b.getX() - center.getX();   //把中点当原点（要用坐标直接比较还得标准化）
    double y2 = b.getY() - center.getY();   //把中点当原点
    if(a.getX()>center.getX())//方向向右
    {
        if(y2>(k*x2)){ //y > kx
            return true;
        }else{
            return false;
        }
    }
    else //方向朝左
    {
        if(y2<(k*x2)){ //y > kx
            return true;
        }else{
            return false;
        }
    }

}
```

根据旋转方向对β处理得到旋转角θ，便可以对点进行旋转

(3) 旋转误差修正函数：

这是在实现圆的旋转后（虽然圆的旋转并无意义，但是起码放大了精度损失这个问题）发现的，由于旋转具有一定的精度损失（像素点都是整数的坐标），因此当精度累积后，可能会导致旋转失真甚至图形不断放大或者缩小。因此我采取了3个措施来实现旋转误差的修正：

（1）旋转时，在double转int时加上修正参数0.5

（2）其中ROTATE_ACCURACY是误差的容许值，ridus是旋转前圆的半径，x，y分别是旋转后的点。由于绕着圆心旋转，因此就寻找以(x, y)的附近在点集$(x \pm ROTATEA_CCURACY, y \pm ROTATEA_CCURACY)$的范围内到圆心距离与原半径误差最小的点。这样确保了误差不会太大

```
Point CycleController::getTheAccurayRotatePoint(qreal ridus, int x, int y)
{
```

```
    int resX = x-ROTATE_ACCURACY;
    int resY = y-ROTATE_ACCURACY;
    double minDiff =100;
    for(int i=x-ROTATE_ACCURACY;i<=x+ROTATE_ACCURACY;i++){
        for(int j=y-ROTATE_ACCURACY;j<=y+ROTATE_ACCURACY;j++){
            Point tmp(i,j);
            double diffTmp = fabs(this->cycle-
>centerPoint.distanceToPoint(tmp.getQPoint())-ridus);
            if(diffTmp<minDiff){
                resX=i;
                resY=j;
                minDiff = diffTmp;
            }

        }
    }
    return Point(resX,resY);
}
```

（3）由于半径的计算也有误差，因此在一次旋转中，对每次求得的半径取平均，以求达到稳定。

基于上述算法，**后期对于直线的旋转也进行了相应的校正处理**

(6).放缩变换算法

(a)基本原理

我实现但是相对于原点的缩放。

将每个顶点坐标乘以缩放系数s_x, s_y到得到新坐标 (x_1, y_1)的移动满足

$$x1 = x \cdot s_x$$
$$y1 = y \cdot s_y$$

(b)算法实现

1. 直线：对线上的起始点和顶点放缩，然后重新绘制
2. 圆：放缩圆心和圆周上的点（决定半径），然后重新绘制
3. 椭圆：放缩中心和外界矩形的一个顶点（决定a、b），然后重新绘制
4. 多边形：放缩所有的顶点，然后重新绘制

(7).区域填充算法 ——泛滥填充

(a)基本定义

泛滥填充算法：区域内部用单一颜色定义的区域填充。通过替换指定的内部颜色来对这个区域着色(填充)。

(b)填充算法

从种子点开始，按像素连通定义，递归检测和扩展区域内部像素，并将填充颜色赋给这些像素，直到所有内部点均被着色。

(c)C++代码实现

泛滥填充算法实际上思路很简单，我目前实现是类似windows画图程序的油漆桶功能，即鼠标点击一个点(x, y)，则按照**4连通**定义，递归检测，对其连通区域中与(x, y)颜色相同的像素点赋予用户选择颜色。

但是泛滥填充算法的效率和对于栈的要求比较高，因此我对其进行了一些优化

(1）定义`bool processed[]`数组，大小为整个区域，表示是否处理过，这样对于处理过的点可以不在判断

(2) 改递归为循环，用一个栈stack存储待处理的点，处理后将未处理过的且四连通的的颜色与底色相同的点入栈，每次对出栈的点处理，一直填充直到栈为空

(3) 将栈stack定义在堆区，减少对系统栈的使用负担

```
stack->clear();
stack->push(QPoint(cx,cy));
while(!stack->empty()){
    QPoint p = stack->pop();
    int x = p.x();
    int y = p.y();

    if(x<=0 || x>=pix->width()-2) continue;     //超限
    if(y<=0 || y>=pix->height()-2) continue;    //超限
    if(img->pixelColor(x,y) != backcolor) continue;   //边界
    if(processed[x][y] == true) continue;      //上色过了
    processed[x][y] = true;
    painter->drawPoint(x,y);
    if(!processed[x][y+1]){
        stack->push(QPoint(x,y+1));
    }
    if(!processed[x][y-1]){
        stack->push(QPoint(x,y-1));
    }
    if(!processed[x-1][y]){
        stack->push(QPoint(x-1,y));
    }
    if(!processed[x+1][y]){
        stack->push(QPoint(x+1,y));
    }
}
stack->clear();
```

(8).直线裁剪算法 ——梁友栋-Barsky参数裁剪算法

(a)基本原理

对于直线段$P_1(x_1, y1), P_2(x_2, y2)$，可以用参数方程形式表示

$$x = x_1 + u(x_2 - x_1) = x_1 + \Delta x$$
$$y = y_1 + u(y_2 - y_1) = y_1 + \Delta y$$

若有$P(x, y)$位于由$(xmin, ymin), (xmax, ymax)$所构成的裁剪窗口内，则有下式成立

$$xmin \le x1 + u \cdot \Delta x \le xmax$$
$$ymin \le y1 + u \cdot \Delta y \le ymax$$

这四个不等式可以表达为

$$u \cdot p_k \le_k, k = 1, 2, 3, 4$$

其中，p, q被定义为

$$p_1 = -\Delta x, q_1 = x1 - xmin$$
$$p_2 = \Delta x, \quad q_2 = xmax - x1$$
$$p_3 = -\Delta y, q_3 = y1 - ymin$$
$$p_4 = \Delta y, \quad q_4 = ymax - x1$$

则有以下结论:

平行于窗口某边界的直线，其$p_k = 0$，k值对应于相应的边界（$k = 1$，2，3，4对应于左、右、下、上边界）。

1. 如果 $q_k < 0$，则线段完全在边界外，应舍弃该线段。
2. 如果 $q_k \ge 0$，则线段平行于窗口某边界并在窗口内。
3. 如果 $p_k < 0$，则线段从裁剪边界延长线的外部延伸到内部;
4. 如果 $p_k > 0$，则线段从裁剪边界延长线的内部延伸到外部

当$\Delta x \ge 0$时

1. 对于左边界$p_1 < 0$（$p_1 = -\Delta x$），线段从左边界的外部到内部;
2. 对于右边界$p_2 > 0$（$p_2 = \Delta x$），线段从右边界的内部到外部

当$\Delta y < 0$时

1. 对于下边界$p_3 > 0$（$p_3 = -\Delta y$），线段从下边界的内部到外部;
2. 对于上边界$p_4 < 0$（$p_4 = \Delta y$），线段从上边界的外部到内部

当$p_K \ne 0$时，可以计算出参数u的值，它对应于无限延伸的直线与延伸的窗口边界k的交点

$$u = q_k / p_k$$

对于每条直线，可以计算出参数u_1和u_2，该值定义了位于窗口内的线段部分:

1. u_1的值由线段从外到内遇到的矩形边界所决定（$p_k < 0$），对这些边界计算$r_k = q_k / p_k$，u_1取0和各个r值之中的最大值。
2. u_2的值由线段从内到外遇到的矩形边界所决定（$p_k > 0$），对这些边界计算$r_k = q_k / p_k$，u_2取1和各个r值之中的最小值。
3. 如果$u1 > u2$，则线段完全落在裁剪窗口之外，应当被舍弃；否则，被裁剪线段的端点可以由$u1$和$u2$计算出来。

(b)优点——与Cohen-Sutherland算法的比较

与Cohen-Sutherland算法相比，梁友栋-Barsky算法减少了交点计算次数:

（1）梁友栋-Barsky

- 更新参数u_1、u_2仅需一次除法;
- 线段与窗口的交点仅计算一次就计算出u_1、u_2的最后值。

（2）Cohen-Sutherland算法

- 即使对完全落在裁剪窗口之外的一条线段，也要对它反复求交点，而且每次求交计算都需要除法和乘法运算。

(c)算法过程

1. 参数初始化：线段交点初始参数分别为：$u_1=0$，$u_2=1$。
2. 定义判断函数
   - 用p、q来判断：是舍弃线段？还是改变交点参数r
     - $p<0$，参数r用于更新u_1
     - $p>0$，参数r用于更新u_2
   - 更新后的判断
     - 若更新u_1或u_2后，使$u_1>u_2$，则舍弃该线段
     - 否则，更新u值仅仅是求出交点、缩短线段
3. 求解交点参数
   - 测试四个p、q值后，若该线段被保留，则裁剪线段的端点由u_1、u_2值决定
   - $p=0$且$q<0$时，舍弃该线段
4. 反复执行上述过程，计算各裁剪边界的p，q值进行判断。

(d)算法的C++实现

```
bool LineController::cutLineLiangBsrsky(QPoint cutStart, QPoint cutEnd, QPainter
*painter, QPen pen)
{
    //在这里进行判断
    //对裁剪窗口预处理
    double xmin = std::min(cutStart.x(),cutEnd.x());
    double ymin = std::min(cutStart.y(),cutEnd.y());
    double xmax = std::max(cutStart.x(),cutEnd.x());
    double ymax = std::max(cutStart.y(),cutEnd.y());
    //得到待裁剪直线的各个端点
    double x1 = curLine->startPoint.getX();
    double y1 = curLine->startPoint.getY();
    double x2 = curLine->endPoint.getX();
    double y2 = curLine->endPoint.getY();
    //各个参数的定义
    double dx = x2-x1;  //△x
    double dy = y2-y1;  //△y
    double p[4] = {-dx,dx,-dy,dy};
    double q[4] = {x1-xmin,xmax-x1,y1-ymin,ymax-y1};
    double u1 = 0;
    double u2 = 1;

    //梁友栋裁剪算法，对p和q进行判断
    for(int i=0;i<4;i++){

        if(fabs(p[i])<1e-6 && q[i]<0){  //p=0且q<0时，舍弃该线段
            this->clearState();
            return false;
        }
```

```
        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
        }

        if(u1>u2){  //如果u1>u2，则线段完全落在裁剪窗口之外，应当被舍弃
            this->clearState();
            return false;
        }
    }

    curLine->setStartPoint(Point(x1+int(u1*dx+0.5), y1+int(u1*dy+0.5)));
    curLine->setEndPoint(Point(x1+int(u2*dx+0.5), y1+int(u2*dy+0.5)));
    MyDrawLineDDA(painter,curLine->startPoint.point,curLine->endPoint.point);
    drawHandle(painter,pen);

    return true;

}
```

3.系统介绍

3.1 实验环境

操作系统（运行平台）	Windows 10
开发语言	C++
开发环境	Qt Creator
编译环境	MinGW 5.3.0

3.2 系统组织

本系统使用Qt提供图形界面，通过Qt集成的QOpenGLWidget类来提供画布，实验主要分为以下类

类名	继承于	类功能
MainWindow	QMainWindow	提供整个系统的框架，并提供基础功能，如选择绘图类型（目前是直线，圆，椭圆），撤销，清屏，新建图像，保存图像等选项，是交互的窗口。
Canvas_GL	QOpenGLWidget	每个Canvas_GL类都是一个绘图的画布，用户在上面绘图
ColorPanel	QWidget	尚未完全实现，用于提供更为便捷的功能选择
FigureController		图形控制的接口，用于定义绘图行为（目前只有绘制，后期预计在此接口上实现更多功能）
LineController	FigureController	控制直线的输入和编辑，实现了DDA直线算法
CycleController	FigureController	控制圆的输入和编辑，实现了中点圆和Bresenham画圆法
EllipseController	FigureController	控制椭圆的输入和编辑，实现了中点点椭圆绘制算法
PolygonController	FigureController	控制了多边形输入和编辑
Figure		图形类的基类，用于记录画过的图形，便于后期操作
Line	Figure	直线，记录了直线的起点和终点
Cycle	Figure	圆，记录了圆心和半径（实际上是圆周的任意一点）
Ellipse	Figure	椭圆，记录了代表长短轴的矩形（其实是中心和矩形顶点）
Polygon	Figure	多边形，记录了多边形的各个顶点坐标
Point		点，集成了QPoint，对于关于点的一些常用操作进行抽象集成

3.2.1 系统结构和类关系图

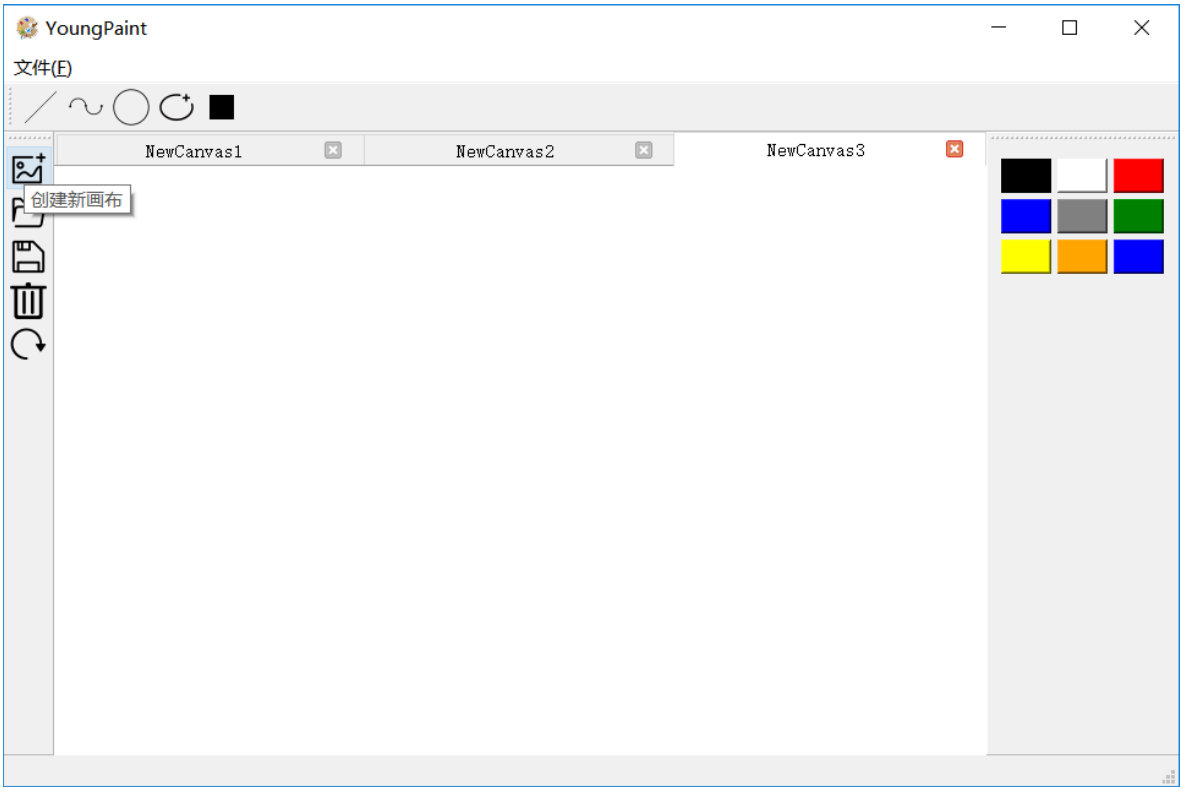
3.2.2程序基本流程

1. 系统主题框架类：MainWindow，内集成一个画布类Canvas_GL的数组
2. 每次创建新画布，就显示新创建的画布
    - 画布可以自由切换
    - 选择功能时，对所有的画布都生效
    - 每个画布中都集成了各种图形的Controller
    - 撤销、清空、保存功能都对当前活动的画布（活动窗口）处理，不影响其他窗口
3. 画布接受事件输入
    - 通过接受鼠标事件，对基本图形输入，编辑
        - 共性通过对应图形的Controller的对应函数来处理，这里用到了动态绑定来精简代码
            - mousePressEvent
            - mouseMoveEvent
            - mouseReleaseEvent
        - 对图形的特殊性质特别处理
    - 对于画笔和笔刷，将每次move的点通过直线连起来
    - 填充功能直接对鼠标点击区域填充
    - 裁剪目前只对直线有效，鼠标输入裁剪框，点击裁剪按钮即可裁剪

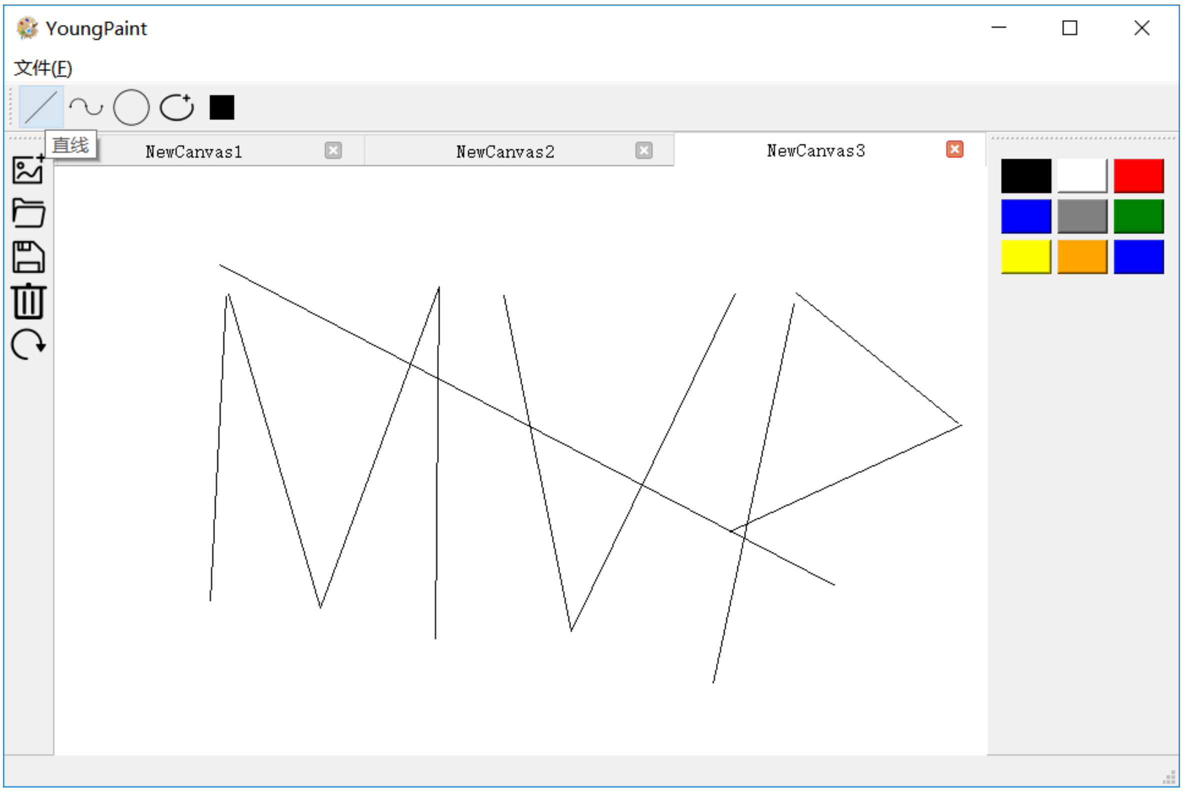
3.3 功能演示

(1) 打开画面

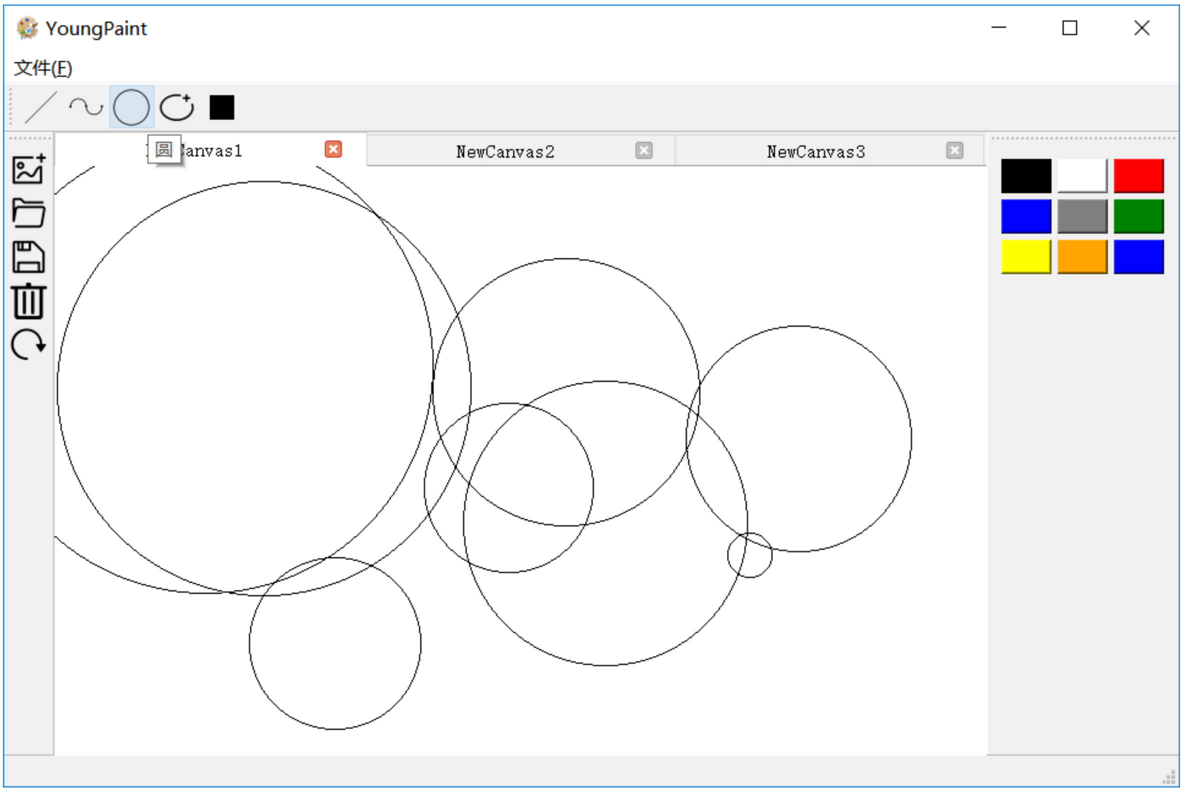
(2) 创建新画布：点击左侧工具栏的创建新画布，则可以创建画布，并且可以创建多个画布



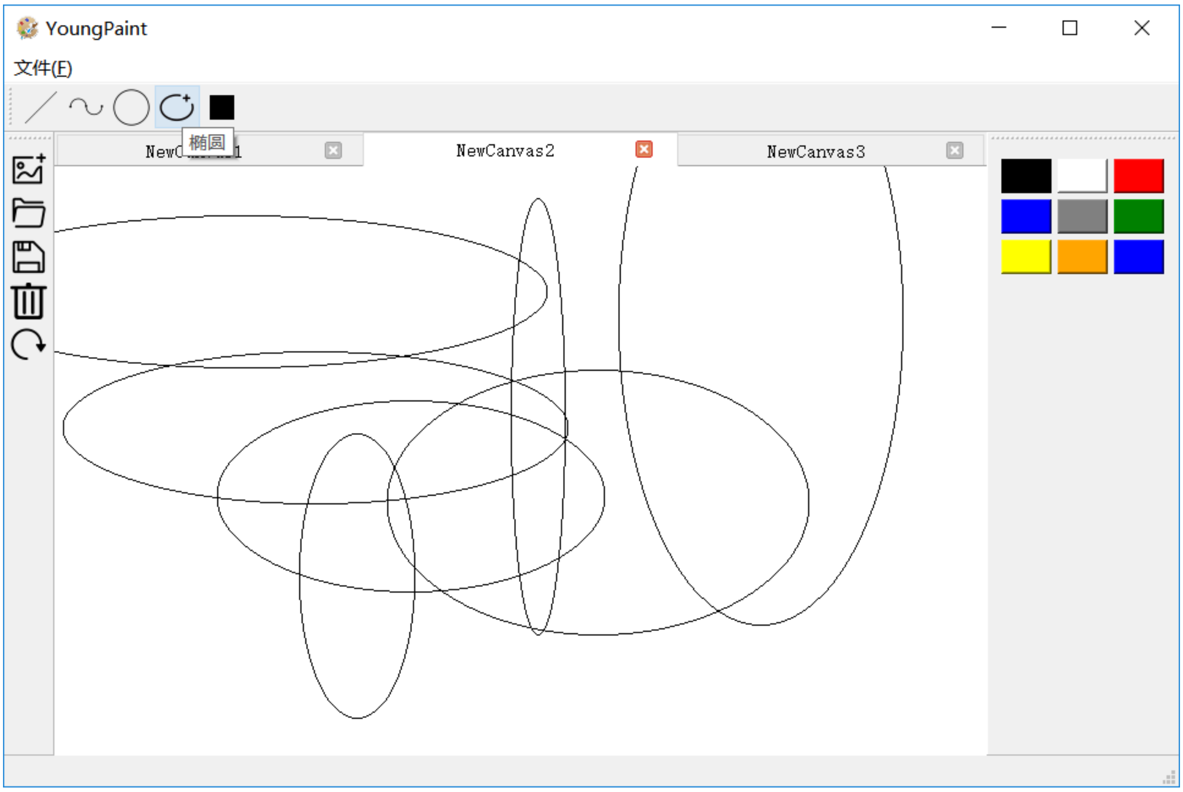
(3) 画直线：点击上方工具栏的直线，则可绘制直线，鼠标点击确定起点，释放确定终点



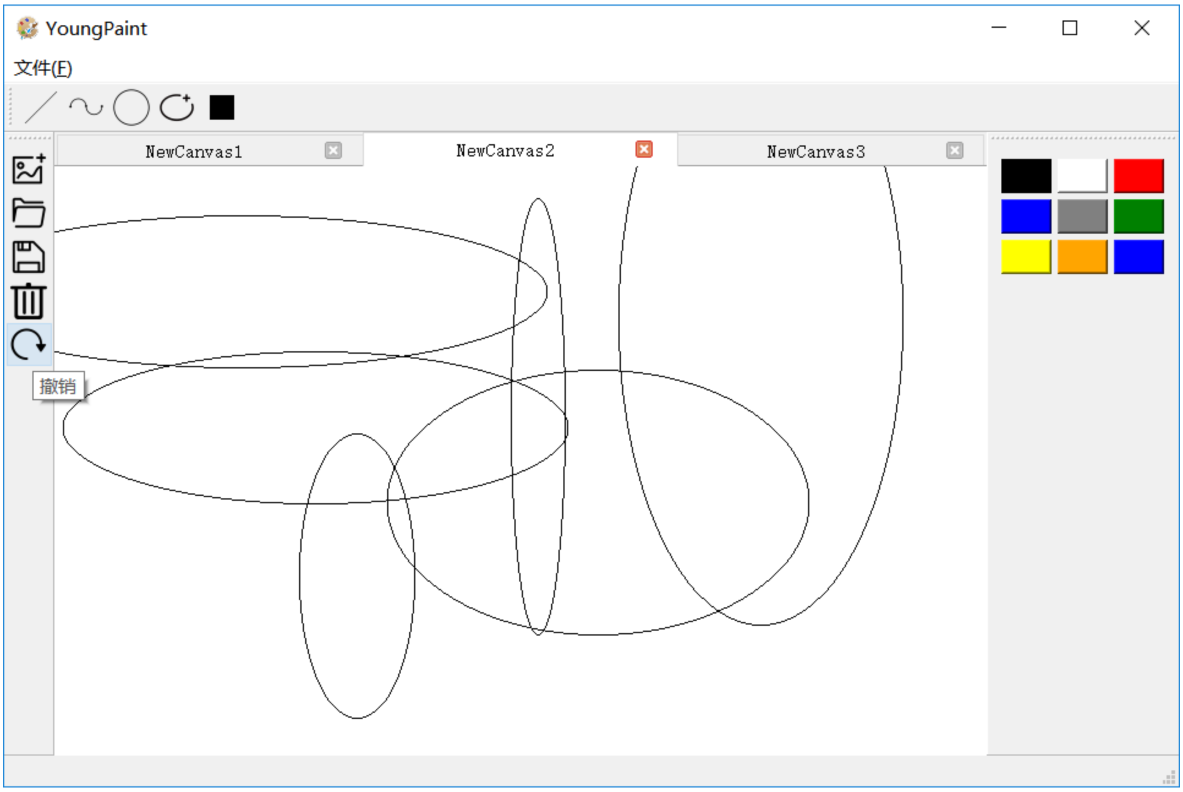
(4) 画圆：点击上方工具栏的圆，则可绘制圆。鼠标点击确定圆心，释放确定半径



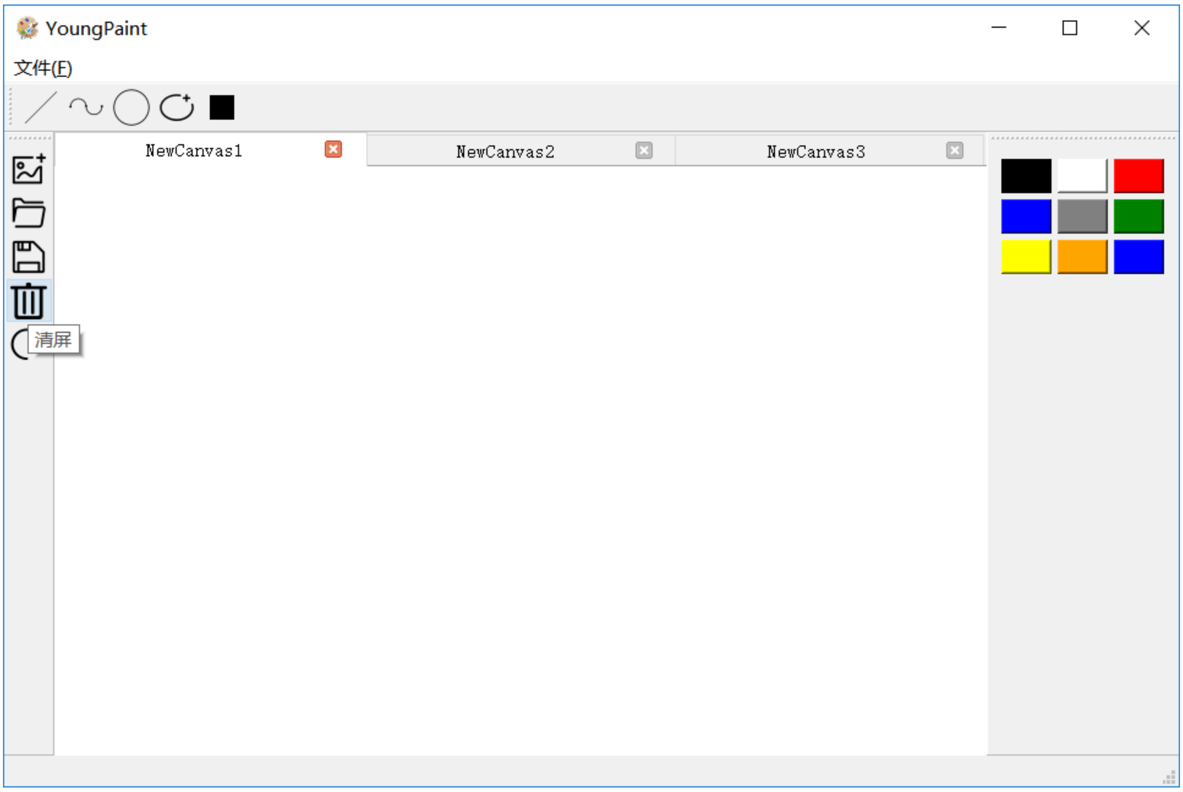
(5) 画椭圆：点击上方工具栏的椭圆，则可绘制椭圆。鼠标点击确定中心，释放确定长轴和短轴



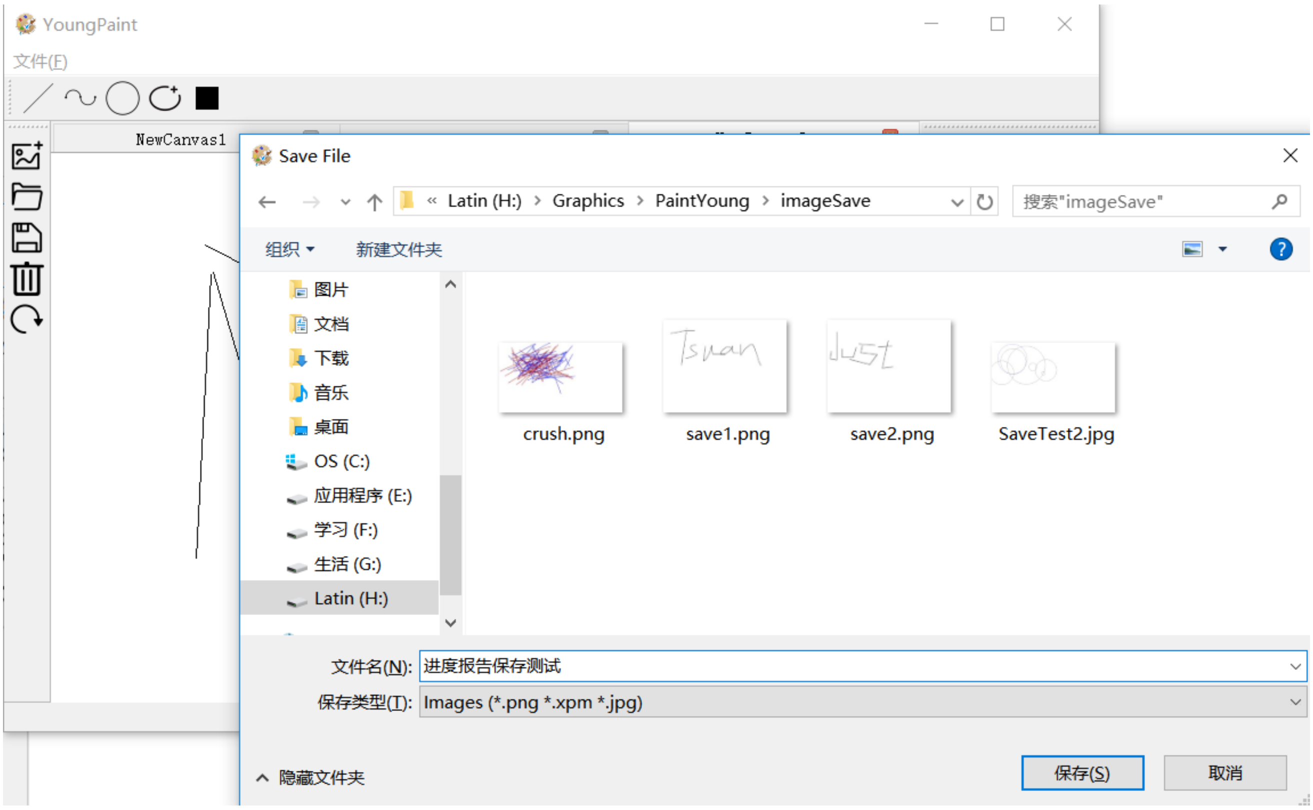
(6) 撤销：点击左侧工具栏撤销按钮，即可撤销



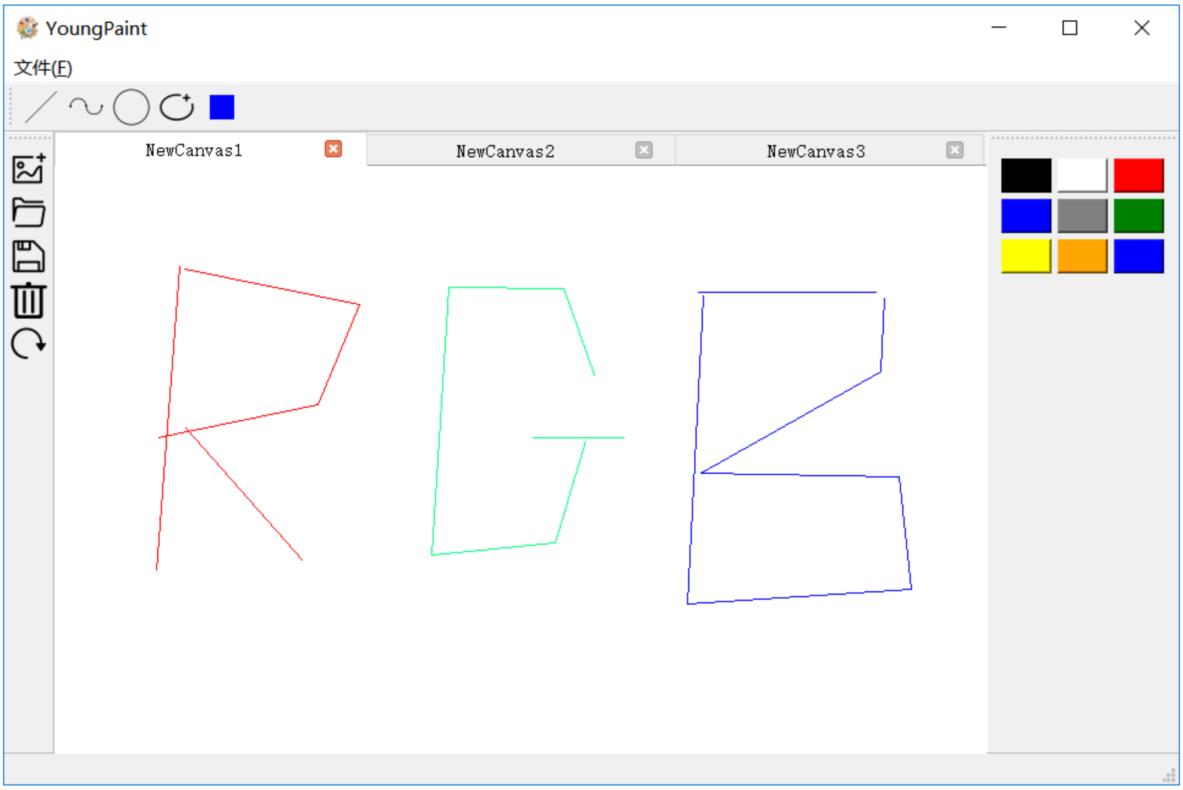
(7) 清屏：点击左侧工具栏清屏按钮，即可清屏



(8) 保存：点击左侧工具栏保存按钮，即可保存



(9) 颜色选择：点击上方工具栏颜色选择按钮，即可选择颜色



4.总结

本绘图系统名为YoungPaint，基于C++和Qt 5.11.2，于Qt Creator上开发，编译环境为MinGW 5.3.0

在9-10月关于图形学的学习中，基于我在课上所学的理论知识，以及课外对于Qt的交互、界面设计的学习，在截止10月底的系统中，我实现了二维图形中直线，圆以及椭圆的输入，并且实现创建多个窗口，画笔颜色的选择，绘画的撤销以及图像的保存功能。

这次实验是我第一次写具有图形交互的的实验，感觉十分有趣。把图形学课上的理论同实践相结合并且不断探索，不断阅读各种文档资料学习新知识的感觉也不错。尤其是双缓冲绘图的实现，起初我为了实现类似画图程序的动态效果而自己实现了一个，后来听同学说这就是双缓冲技术，独立探索出了这样的技巧让我感觉我的确是有在学习东西的，这也让我对于该实验有着更大的兴趣。尽管由于其他原因，10月份的程序不能说尽善尽美，但是基于我对于程序的理解，一遍上着高级程序设计课学习C++各种高级性质，我尽可能把我的知识和设计体现在代码上。

5.参考文献

[1] 孙正兴,周良,郑洪源,谢强.计算机图形学教程.北京:机械工业出版社,2006.

[2] 陈家骏,郑滔.程序设计教程用 C++语言编程(第 3 版).北京:机械工业出版社,2015.

（附其他参考资料）

[3] Qt学习社区上的《Qt基础教程之Qt学习之路》

[4] Qt官方文档

[5] Qt5.9.4利用QOpenGLWidget类进行opengl绘图 https://blog.csdn.net/cpwwhsu/article/details/79773235

[6] Qt学习之路-简易画板3(双缓冲绘图) https://blog.csdn.net/u012891055/article/details/41727391?utm_source=blogxgwz0