

# 《CG2018》系统技术报告

郑来栋

(南京大学 计算机科学与技术系, 南京 210093)

**摘要:** 系统使用 C++语言, 基于 QT 和 OpenGL, 实现了一个交互式的绘图系统。实现了图形的绘制, 填充, 旋转, 平移, 缩放以及裁剪, 三维模型的显示, 实现了将图形保存为 BMP 格式的文件。

## 1 引言

引言

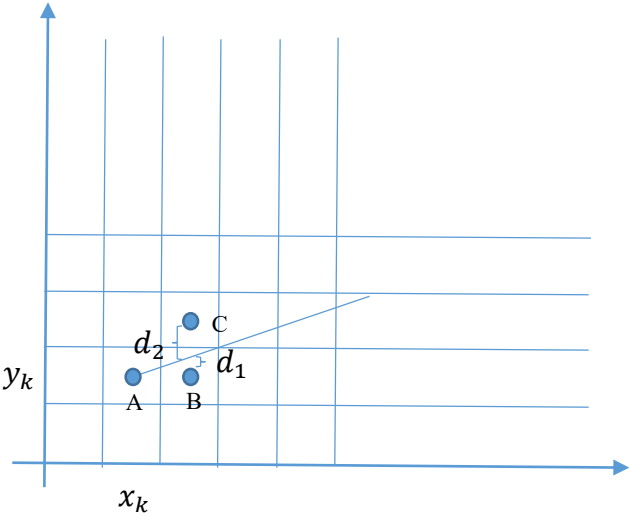
## 2 算法描述

### 2.1 图形绘制

#### 2.1.1 直线绘制-Bresenham 画线算法

假设线段  $l = \overrightarrow{P_1P_2}$ , 绘制从  $P_1(x_1, y_1)$ 到  $P_2(x_2, y_2)$ ,  $l$  直线方程记为  $y = mx + b$

先考虑  $0 < m < 1$ , 且  $x_1 < x_2$  的情况, 其余情况可根据对称性做一些提前处理得到。在  $\Delta x \geq \Delta y$  的情况下, 应在  $x$  方向进行取样, 并在  $y$  方向上最接近线段的像素上绘出一点。



如上图所示, 假设在绘制一系列点之后, 在  $A(x_k, y_k)$  绘出一点, 下一步需要确定在  $x = x_k + 1$  时绘制在  $B(x_k + 1, y_k)$  还是  $C(x_k + 1, y_k + 1)$ , 也就是决定 B 和 C 点哪一点离直线更近。

#### 2.1.1.1 推到决策参数 $p_k$ , 确定离线段最近的候选点

设线段  $l$  与  $BC$  连线交于  $D$ , 记  $d_1 = |BD|$ ,  $d_2 = |CD|$ , 下面将  $d_1, d_2$  用坐标表示出来。

D 点的 y 坐标可由直线方程得到  $y = m(x_k + 1) + b$

故,

$$\begin{cases} d1 = y - y_k = m(x_k + 1) + b - y_k \\ d2 = y_k + 1 - y = y_k + 1 - m(x_k + 1) - b \end{cases}$$

因此 B 和 C 离线段的竖直距离差  $|BD| - |DC|$

$$d_1 - d_2 = 2m(x_k + 1) - 2y_k + 2b - 1$$

记  $p_k = dx(d_1 - d_2)$

由于  $m = \frac{dy}{dx}$ , 化简可得

$$p_k = 2dyx_k - 2dxy_k + c$$

由于  $dx > 0$ , 所以  $p_k$  与  $d_1 - d_2$  同号, 也就是

$$\begin{cases} p_k > 0 \Leftrightarrow d_1 > d_2 & \text{绘制 C点} \\ p_k < 0 \Leftrightarrow d_1 < d_2 & \text{绘制 B点} \end{cases}$$

所以, 把上面的  $p_k$  记为决策参数, 用来决定每一步中离线段更近的候选点。

#### 2.1.1.2 增量式计算 $p_k$ , 消除代价高昂的乘法运算

当  $x = x_k + 2$  时, 同样可得

$$p_{k+1} = 2dyx_{k+1} - 2dxy_{k+1} + c$$

则

$$p_{k+1} - p_k = 2dy(x_{k+1} - x_k) - 2dx(y_{k+1} - y_k)$$

而  $x_{k+1} = x_k + 1$ ,  $y_{k+1} - y_k = \begin{cases} 1 & p_k \geq 0 \\ 0 & p_k < 0 \end{cases}$  所以

$$p_{k+1} = \begin{cases} p_k + 2dy & p_k \geq 0 \\ p_k + 2dy - 2dx & p_k < 0 \end{cases}$$

而计算开始时, 初始的决策参数  $p_0 = 2dy - dx$ , 这样可以只通过加减运算高效地计算决策参数

#### 2.1.1.3 处理所有情况

- (1)  $|m| \leq 1$  时, 在 x 方向取样,  $|m| > 1$  时, 在 y 方向取样
- (2) 为了使  $p_k$  的符号与  $d_1 - d_2$  相同, 需保证  $dx$  (或  $dy$ ) 为正值
- (3) 使用一个变量来记录取样方向是正方向还是负方向

```
dx=2*(x2-x1), dy=2*(y2-y1)
stepx=stepy=1 //采样方向的增量
if(dx<0){dx=-dx, stepx=-1}
if(dy<0){dy=-dy, stepy=-1}
plot(x0, y0)
if(dx>dy){
    p=dy-dx>>1
    while(x0!=x1){
        x0+=stepx
        if(p>=0){y0+=stepy; p-=dx}
        p+=dy
        plot(x0, y0)
    }
}
```

```

    }else{
        p = dx - dy >> 1
        while(y0 != y1){
            if(p>=0){x0 += stepx; p-=dy}
            y0+=stepy
            p+=dx
            plot(x0, y0)
        }
    }
}

```

### 2.1.2 圆绘制-中点圆算法

对于给定半径  $r$  和圆心  $(x_c, y_c)$ ，可以先计算出圆心在原点  $(0, 0)$  的圆的像素位置，然后通过平移得到最终的像素位置。

考虑到圆的对称性，我们只需计算出在  $x = 0$  和  $x = y$  两条直线之间的坐标(一个八分圆)，通过对称性就可以得到所有的像素位置。

类似于直线绘制过程中，我们使用决策参数来确定候选点  $(x_k + 1, y_k)$  和  $(x_k + 1, y_k - 1)$  中，哪一点离圆最近，决策参数  $p$  的递推公式为

$$p_{k+1} = \begin{cases} p_k + 2x_{k+1} + 1 & p_k < 0 \\ p_k + 2x_{k+1} + 1 - 2y_{k+1} & p_k > 0 \end{cases}$$

初始决策参数  $p_0 = 1 - r$

这样，可以根据  $p_k$  的值决定选择  $y_k$  还是  $y_k - 1$

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

绘制过程伪代码如下

```

(1)计算  $p_0 = 1 - r$ 
(2)plot( $x_0, y_0$ )
(3)在  $x$  方向进行取样，对每个  $x_k$ ，若  $p_k < 0$ ，则下一个点为  $(x_k + 1, y_k)$ 
    且  $p_{k+1} = p_k + 2x_{k+1} + 1$ 
    否则下一个点是  $(x_k + 1, y_k - 1)$ 
    且  $p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$ 
    plot( $x_k, y_k$ )

```

其中，子程序 `plot` 用于根据对称性，对每个八分圆上的点进行绘制，且进行平移得到最终像素坐标。

```

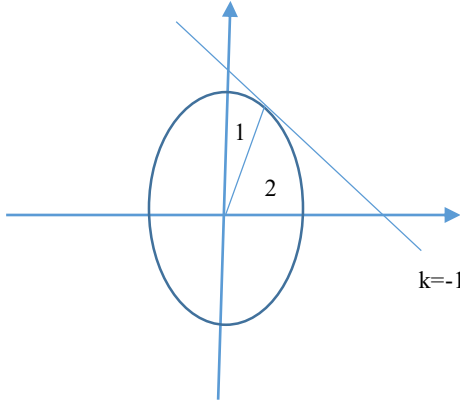
plot(x, y)
{
    draw(xc+x, yc+y)  draw(xc+x, yc-y)
    draw(xc-x, yc+y)  draw(xc-x, yc-y)
    draw(xc+y, yc+x)  draw(xc-y, yc+x)
    draw(xc+y, yc-x)  draw(xc-y, yc-x)
}

```

### 2.1.3 椭圆绘制-中点椭圆算法

类似于中点圆算法，将椭圆平移到原点(0, 0)，计算出像素坐标，然后再平移回去。且考虑对称性，只计算第一象限的像素即可。

依据椭圆切线斜率对第一象限进行划分，斜率绝对值小于 1 的区域在 x 方向取单位步长，在斜率绝对值大于 1 区域在 y 方向取单位步长。



在区域 1，决策参数的递推式如下

$$p1_{k+1} = \begin{cases} p1_k + 2r_y^2 x_{k+1} + r_y^2 & p1_k < 0 \\ p1_k + 2r_y^2 x_{k+1} + r_y^2 - 2r_x^2 y_{k+1} & p1_k \geq 0 \end{cases}$$

初始值

$$p1_0 = r_y^2 - r_x^2 r_y^2 + \frac{1}{4} r_x^2$$

且根据  $p1_k$  的符号，可以确定下一个 y 的值

$$y_{k+1} = \begin{cases} y_k & p1_k < 0 \\ y_k - 1 & p1_k \geq 0 \end{cases}$$

在区域 2，决策参数的递推式如下

$$p2_{k+1} = \begin{cases} p2_k - 2r_x^2 y_{k+1} + r_x^2 & p2_k > 0 \\ p2_k + 2r_y^2 x_{k+1} + r_x^2 - 2r_x^2 y_{k+1} & p2_k \geq 0 \end{cases}$$

初始值

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

且根据  $p2_k$  的符号，可以确定下一个 x 的值

$$x_{k+1} = \begin{cases} x_k & p2_k > 0 \\ x_k - 1 & p2_k \leq 0 \end{cases}$$

且为了加快计算速度，应提前计算平方项，作为常数使用。绘制过程类似于中点圆算法，分开对区域 1 和区域 2 进行处理。

### 2.1.4 矩形和多边形

矩形和多边形的绘制是利用直线绘制完成的，系统存储其顶点，绘制时，在顶点之间按序绘制线段即可。不同点是它们的表示不同，矩形采用左上角和右下角表示(这就造成了系统只能表示标准矩形，旋转时只能旋转 90 度的倍数)，而多边形保存顶点表。

### 2.1.5 Bezier 曲线

给定  $n+1$  个控制点,  $p_k = (x_k, y_k)$ ,  $0 \leq k \leq n$ , 输出  $n$  阶贝塞尔曲线上的点。  
 曲线上点的坐标可以表示为

$$\begin{cases} x(u) = \sum_{k=0}^n x_k BEZ_{k,n}(u) \\ y(u) = \sum_{k=0}^n y_k BEZ_{k,n}(u) \end{cases} \quad 0 \leq u \leq 1$$

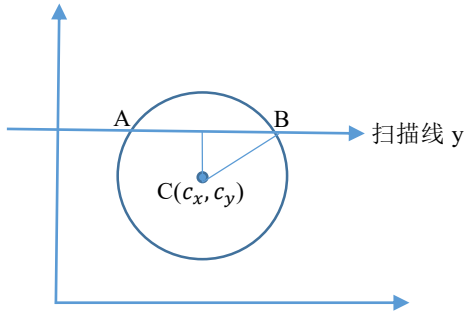
其中,  $BEZ_{k,n}(u) = \frac{n!}{k!(n-k)!} u^k (1-u)^{n-k}$ , 如果直接计算该式子, 计算量将相当大, 我们可以采用迭代式地计算

$$BEZ_{k,n}(u) = (1-u)BEZ_{k,n-1}(u) + uBEZ_{k-1,n-1}(u), n > k \geq 1$$

## 2.2 图形填充

### 2.2.1 圆的填充

采用扫描填充, 直接处理, 对每条平行  $x$  轴的扫描线, 计算其与圆两侧的交点, 填充交点之间的线段即可。扫描线  $y$  与圆两侧的交点可用勾股定理求出。



$$\begin{cases} x_A = c_x - \sqrt{r^2 - (y - c_y)^2} \\ x_B = c_x + \sqrt{r^2 - (y - c_y)^2} \end{cases}, \text{ 因此, 扫描线 } y \text{ 上地填充范围为 } [x_A, x_B]$$

```
center: P(cx, cy)
radius: r
for y from cy to cy + r - 1:
    dy = y - cy
    dx = sqrt(r^2 - dy^2)
    left = cx - dx
    right = cx + dx
    for x from left + 1 to right - 1:
        fillPoint(x, y)
        fillPoint(x, 2 * cy - y)
```

2.2.2 矩形填充

矩形填充比较简单，直接对四条边内部区域进行填充即可

```
for x from xmin to xmax:
    for y from ymin to ymax:
        fillPoint(x, y)
```

2.2.3 一般多边形的填充(扫描填充算法)

通过构建有序边表和活化边表，求出各条扫描线上的填充范围，实现了多边形的扫描填充算法。  
边表的数据结构如下：

```
struct Edge
{
    double x;           //起始点横坐标
    double dx;          //y 每增加 1，x 地增加量
    int y_upper;        //该边地最高点
}
```

主要步骤如下，

- (1) 生成有序边表  
扫描多边形中所有边，若是非水平边，计算相应的 x，dx 和 y\_upper 填入有序边表中
- (2) 用有序边表构造活化边表  
扫描当前扫描线及以下的有序边表，将 y\_upper 大于等于当前 y 值的 Edge 加入到本扫描线对应的活化边表中，其中起始点横坐标需要重新计算，然后根据 x 对活化边表从小到大排序
- (3) 根据活化边表计算各扫描线填充范围  
遍历活化边表中的 Edge 项，两两配对，对中间区域进行填充

2.3 几何变换

实现基本的几何变换，包括平移、旋转和缩放，只需实现对点的变换，其余几何图形可根据几何特征基于点进行变换。

设点 P(x, y)，

2.3.1 平移

设平移向量为  $t = (t_x, t_y)$ ，则 P 平移后的坐标为  $\begin{cases} x1 = x + t_x \\ y1 = y + t_y \end{cases}$

基于点的平移，实现基本几何形状的平移

几何图形	平移方式
直线	分别平移线段端点
圆	平移圆心
椭圆	平移椭圆外接矩形左上角和右下角端点
矩形	平移左上角和右下角
多边形	分别平移每个顶点
曲线	分别平移每个控制点

### 2.3.2 旋转

设旋转基准点为  $C = (c_x, c_y)$ ，旋转角为  $\alpha$ ，则  $P$  旋转后的坐标可表示为

$$\begin{cases} x1 = c_x + (x - c_x) \cos \alpha - (y - c_y) \sin \alpha \\ y1 = c_y + (x - c_x) \sin \alpha + (y - c_y) \cos \alpha \end{cases}$$

基于点的旋转，实现基本几何形状的旋转

几何图形	旋转中心	旋转方式
直线	线段中点	分别旋转线段端点
圆	--	do nothing
椭圆	椭圆中心	旋转椭圆外接矩形的顶点
矩形	矩形中心	旋转左上角和右下角的顶点
多边形	所有顶点的中心(取平均)	分别旋转每个顶点
曲线	所有控制点的中心(取平均)	分别旋转每个控制点

### 2.3.3 缩放

设缩放基准点为  $C = (c_x, c_y)$ ，缩放系数大小为  $s$ ，则  $P$  旋转后的坐标可表示为

$$\begin{cases} x1 = xs + c_x(1 - s) \\ y1 = ys + c_y(1 - s) \end{cases}$$

基于控制点的缩放也可以实现基本几何形状的缩放，其基准点的选取和旋转一样，都是选择“中心”

## 2.4 裁剪

设裁剪窗口是一个标准位置的矩形

$W = \{(x, y) \mid x_{\min} \leq x \leq x_{\max}, y_{\min} \leq y \leq y_{\max}\}$ ，且舍弃裁剪窗口外的图形部分。

### 2.4.1 线段裁剪（梁友栋-Barsky 算法）

设线段  $P = P_1P_2$ ，端点分别是  $P_1(x_1, y_1)$ ， $P_2(x_2, y_2)$ ，其参数方程可以写成如下形式：

$$\begin{cases} x = x1 + u\Delta x \\ y = y1 + u\Delta y \end{cases} \quad 0 \leq u \leq 1$$

其中

$$\begin{cases} \Delta x = x2 - x1 \\ \Delta y = y2 - y1 \end{cases}$$

如果点  $P(x, y)$  位于  $W$  内，则应满足

$$\begin{cases} x_{\min} \leq x + u\Delta x \leq x_{\max} \\ y_{\min} \leq y + u\Delta y \leq y_{\max} \end{cases}$$

将该公式变形做统一之后记为

$$up_k \leq q_k$$

可得

$$\begin{cases} p1 = -\Delta x & q1 = x1 - x_{\min} \\ p2 = \Delta x & q2 = x_{\max} - x1 \\ p3 = -\Delta y & q3 = y1 - y_{\min} \\ p4 = \Delta y & q4 = y_{\max} - y1 \end{cases}$$

我们可将这四种情况对应到裁剪窗口的四条边，对于  $p_k$  和  $q_k, k = 1, 2, 3, 4$  分别对应  $W$  的左、右、下、上

边界, 从上式可以有如下观察

- (1)  $p_k = 0$  线段平行于裁剪边界  $k$  (实际上是两条)
  - 若  $q_k < 0$ , 线段在裁剪窗口外
  - 若  $q_k \geq 0$ , 线段平行于裁剪边界且在窗口内
- (2)  $p_k < 0$ 
  - 线段从外到内穿过裁剪边界第  $k$  侧延长线
- (3)  $p_k > 0$ 
  - 线段从内到外穿过裁剪边界第  $k$  侧延长线

裁剪过程的伪代码如下

```

计算 pk, qk
u0 = 0, u1 = 1
for i from 1 to 4:
    if pi == 0:
        if qi < 0:
            return //在裁剪窗口外, 直接舍弃
        else
            r = qi / pi //r 表示直接与第 i 边的交点的参数值
            if pi < 0
                u1 = max(u1, r)
            else
                u2 = min(u2, r)
if u1 > u2:
    return
update:
//保留参数在[u1, u2]内的部分
P1 = (x1 + u1 * Δx, y1 + u1 * Δy)
P2 = (x1 + u2 * Δx, y1 + u2 * Δy)

```

算法 梁友栋-Barsky

#### 2.4.2 矩形裁剪

矩形裁剪较简单, 我们只要求出待裁剪矩形与裁剪窗口  $W$  的重叠部分即可

设待裁剪矩形左上角坐标为  $P1(x1, y1)$ , 右下角坐标为  $P2(x2, y2)$

先假设两个矩形相交, 相交后的矩形为  $C$ , 设  $C$  的左上角坐标为  $(xc1, yc1)$ , 右下角坐标为  $(xc2, yc2)$ , 则其顶点坐标可通过如下式子计算出来

$$\begin{cases} xc1 = \max(x1, xmin) \\ yc1 = \max(y1, ymin) \\ xc2 = \min(x2, xmax) \\ yc2 = \min(y2, ymax) \end{cases}$$

最后判断不等式  $\begin{cases} xc1 \leq xc2 \\ yc1 \leq yc2 \end{cases}$  是否满足即可判断出是否有交集, 如果相交, 则重设带裁剪矩形左上角和右下角的坐标。



### 2.4.3 一般多边形裁剪(Sutherland-Hodgeman 算法)

依次用裁剪窗口的四条边进行裁剪，每一次裁剪过程中，依次检查多边形的每条边，根据边与裁剪窗口对应当前用于裁剪的边界的位置关系，决定将哪些点加入到新顶点集中。

设当前用裁剪窗口的边  $E_i$  ( $0 \leq i \leq 3$  分别对应上、右、下和左边界)，当前检查边  $E = P_1P_2$ ,  $P_1$  为起点， $P_2$  为终点(因为检查过程是有序的，所以边必须有方向性)。设  $E_i$  和  $E$  的交点为  $V$ (若有)

(a)  $P_1$  在  $E_i$  外侧空间， $P_2$  在  $E_i$  内侧空间

依次添加  $V$  和  $P_2$

(b)  $P_1$  和  $P_2$  都在内侧空间

添加  $P_2$

(c)  $P_1$  在  $E$  内侧空间， $P_2$  在  $E_i$  外侧空间

添加  $V$

(d)  $P_1$  和  $P_2$  都在外侧空间

不添加

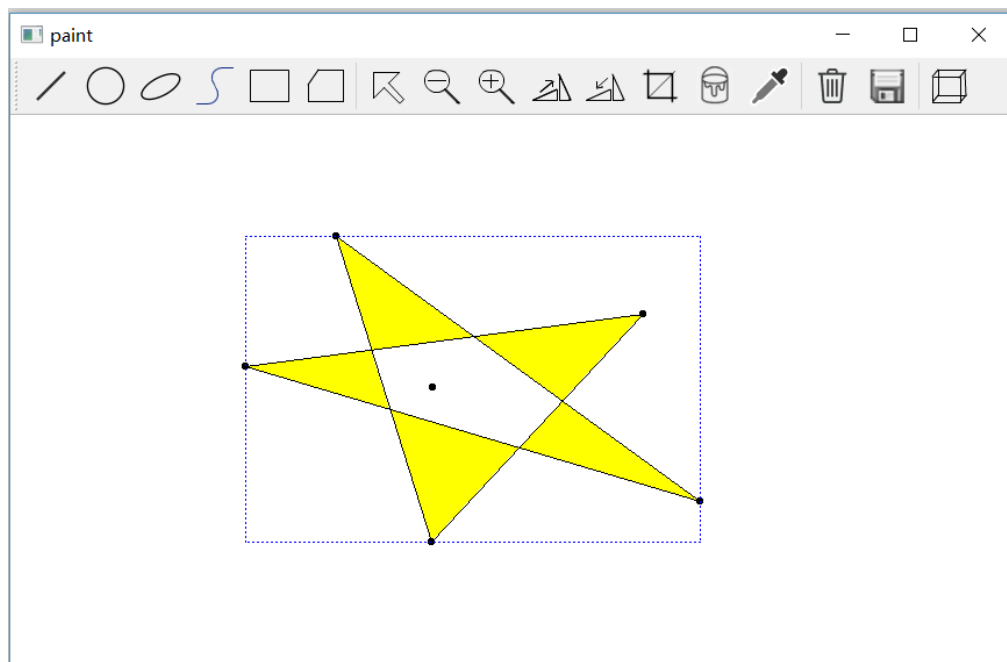
```

输入: vertex                //多边形顶点
    W(xmin, ymin, xmax, ymax)    // 裁剪窗口
    initialise cut_edges         // cut_edges 是裁剪窗口的上, 右, 下, 左四
条边
    for i in 0 to 3 do:
        vertex_temp = {}
         $E_i$  = cut_edges[i]
        for j in 0 to vertex.size - 2 do:
            start = vertex[j], end = vertex[j+1]
             $E$  = Line(start, end)
            if outside(start,  $E_i$ )
                if !outside(end,  $E_i$ )
                     $V$  = intersect( $E$ ,  $E_i$ ) //求交点坐标
                    vertex_temp.add( $V$ )
                    vertex_temp.add(end)
            else
                if outside(end,  $E_i$ )
                     $V$  = intersect( $E$ ,  $E_i$ ) //求交点坐标
                    vertex_temp.add( $V$ )
                else
                    vertex_temp.add(end)
        vertex = vertex_temp        //作为下一轮的输入
    vertex_temp = {}

```

### 3 系统框架

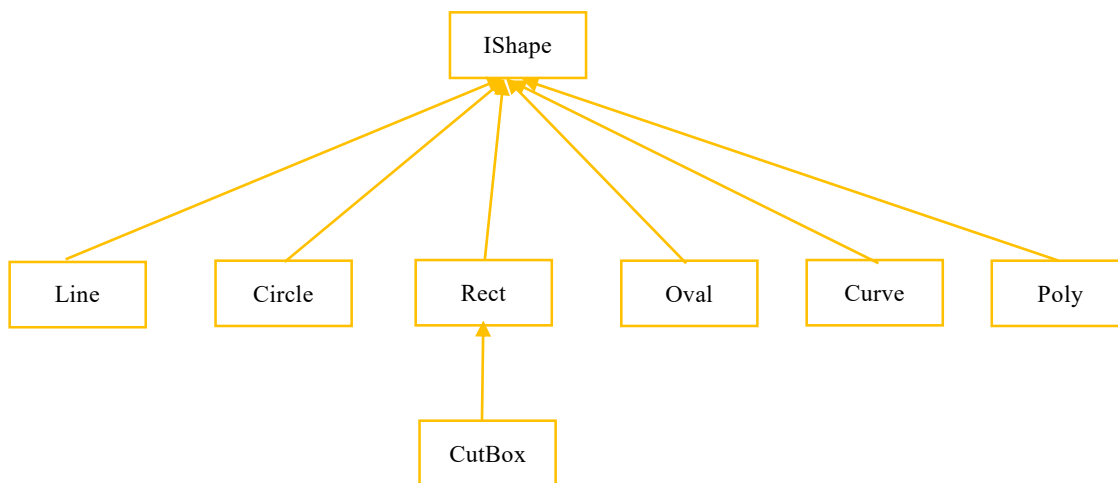
#### 3.1 用户界面



用户界面主要包括两大部分，一是顶部的 **ToolBar**，用于实现图形输入，编辑等菜单选项，改变系统状态。二是中间的绘图区，继承自 **QOpenGLWidget**，用于处理绘制图形，处理鼠标事件等。

#### 3.2 模块划分及类层次结构

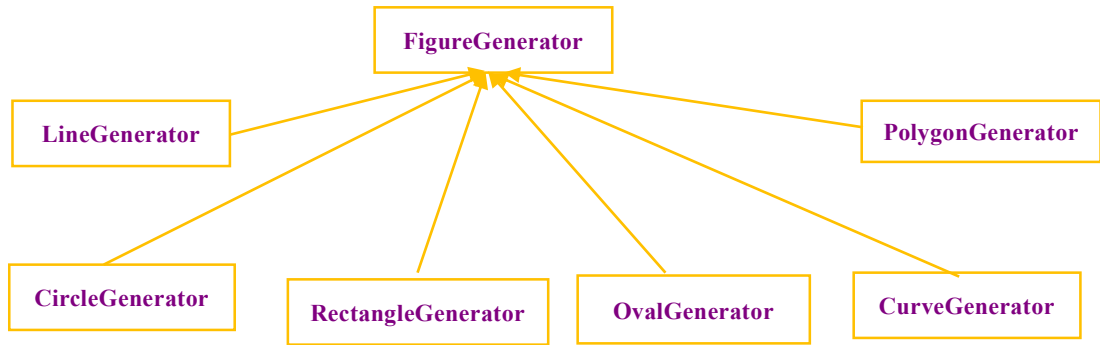
##### 3.2.1 图形类



- (1) 所有的图形类都继承自 **IShape**，**IShape** 实现了一些基本方法，且定义了一套接口，如 **draw()**，**translate()**，**rotate()**等。

- (2) 利用 C++ 的多态性，绘制系统不需要知道图形的种类然后分情况绘制。
- (3) CutBox 类继承自 Rect 类，用于裁剪窗口的绘制
- (4) Curve 类实现的是 Brazier 曲线

### 3.2.2 绘制类



- (1) 绘制类均继承自 FigureGenerator，用于实现图形的绘制功能
- (2) 图形输入功能来自于鼠标事件，依据当前状态和鼠标事件进行不同的动作，下表是对不同图形，不同的鼠标事件应当采取的动作

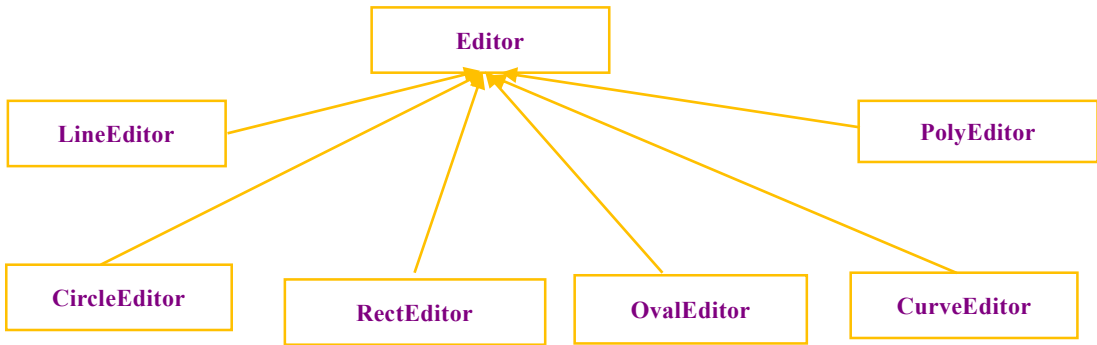
	MouseDown	MouseMove	MouseRelease
直线	以点击点为起点和终点创建一条直线	更新终点	绘制完成
圆	以点击点为圆心，半径为 1 创建一个圆	重新计算并设置半径	
椭圆	创建一个椭圆，以点击点的坐标初始化其外接矩形	重新设置外接矩形的一个顶点	
矩形	以点击点为起点和终点创建一个矩形	重新设置一个顶点	
多边形	以点击坐标添加一个顶点，如果多边形已封闭则结束绘制	更新最后一个顶点坐标	更新最后一个顶点坐标(跟踪鼠标)
曲线	添加一个控制点	不做操作	不做操作

其中，

- a. 对于多边形的绘制，由于不确定用户要输入多少顶点，因此当用户点击时就需要添加顶点，当多边形封闭时标志绘制结束(最后一个点和第一个点重合)
  - b. 对于曲线的绘制，同样需要点击时添加控制点，当用户再次点击 ToolBar 的曲线绘制按钮时，一条曲线绘制结束。
- (3) 如果这些功能在一个类里实现，将有很大的复杂度，难以控制，所以将每个图形的绘制抽象出一个类来实现，它们都能处理鼠标事件，这样设计每一个类结构简单容易控制。

3.2.3 图形编辑类

编辑类主要用于处理图形选中情况下的图形编辑，如修改顶点位置，平移，拖动矩形边长等等。



系统在鼠标点击时，或是在图形选择模式下，会首先检测点击点是否在已选中图形上(如果有)，如果没有，再次检测是否选中了新图形(点击了另一个图形上的点)。这些类重载了关于鼠标的一些事件函数，每个类又分别管理一类图形的编辑。这里的编辑主要事件如下表

图形	事件
直线	修改端点
圆	通过拖动圆上的点修改其半径
矩形	修改四个顶点的坐标
多边形	修改顶点的位置
椭圆	通过修改其外接矩形的四个顶点修改其大小
曲线	修改控制点的位置

3.2.4 工具栏

工具栏包含系统的所有功能，包括绘图，编辑，选中，旋转，填充，保存，打开三维模型等



ToolBar 是一个自定义的工具栏类，它继承自 `QToolBar` 类，其中每个工具对应一个 `QAction` 成员：

```
QAction      *m_Line;
QAction      *m_Circle;
QAction      *m_Oval;
QAction      *m_Curve;
```

每个 `QAction` 通过信号槽机制连接到一个响应函数：

```
connect(m_Line, SIGNAL(triggered()), this, SLOT(onLineClicked()));
connect(m_Circle, SIGNAL(triggered()), this, SLOT(onCircleClicked()));
connect(m_Oval, SIGNAL(triggered()), this, SLOT(onOvalClicked()));
connect(m_Curve, SIGNAL(triggered()), this, SLOT(onCurveClicked()));
```

这些响应函数是在 `ToolBar` 类中定义的槽函数：

```
public slots:
    void onLineClicked();
    void onCircleClicked();
    void onOvalClicked();
    void onCurveClicked();
```

在这些响应函数中做相应动作即可。

### 3.2.5 PaintWidget 和 GL3Dwidget

我们的绘图任务是在这两个类中完成的，分别负责 2D 和 3D 绘图，它们均继承自 `QOpenGLWidget` 和 `QOpenGLFunctions`，重载了 `initializeGL`、`paintGL` 和 `resizeGL` 三个函数用来实现绘制，每次需要重绘的时候，调用它们的 `update` 函数即可重新绘制。

(1) `PaintWidget` 用来绘制 2D 图形，所有的绘制最终都调用点的绘制函数来实现

```
glBegin(GL_POINTS);
glColor3f(r, g, b);
glVertex2i(x, y);
glEnd();
```

(2) `PaintWidget` 重载了一些鼠标事件函数，并负责将这些事件分发到对应的处理者，比如负责绘图的，编辑的等。

```
void mousePressEvent(QMouseEvent *event);
void mouseMoveEvent(QMouseEvent *event);
void mouseReleaseEvent(QMouseEvent * event);
```

(3) `GL3Dwidget` 中绘制三维模型时，各个面分别绘制，每个面由一个多边形围成

```
for(auto f: faces)
{
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);           //非填充
    glBegin(GL_POLYGON);                                   //绘制多边形
    for(size_t v: f)                                       //绘制该面上的所有顶点
        glVertex3f(vertices[v].x, vertices[v].y, vertices[v].z);
    glEnd();
}
```

### 3.2.6 图形保存功能

系统实现了将图形保存为 BMP 文件格式，其中有两个主要任务，一是填充文件头部，二是将图形像素填充到文件对应的位置。如果当前有图形处于选中状态，只保存被选中的图形，否则，保存画板上所有的图形。

### 3.2.7 图形删除

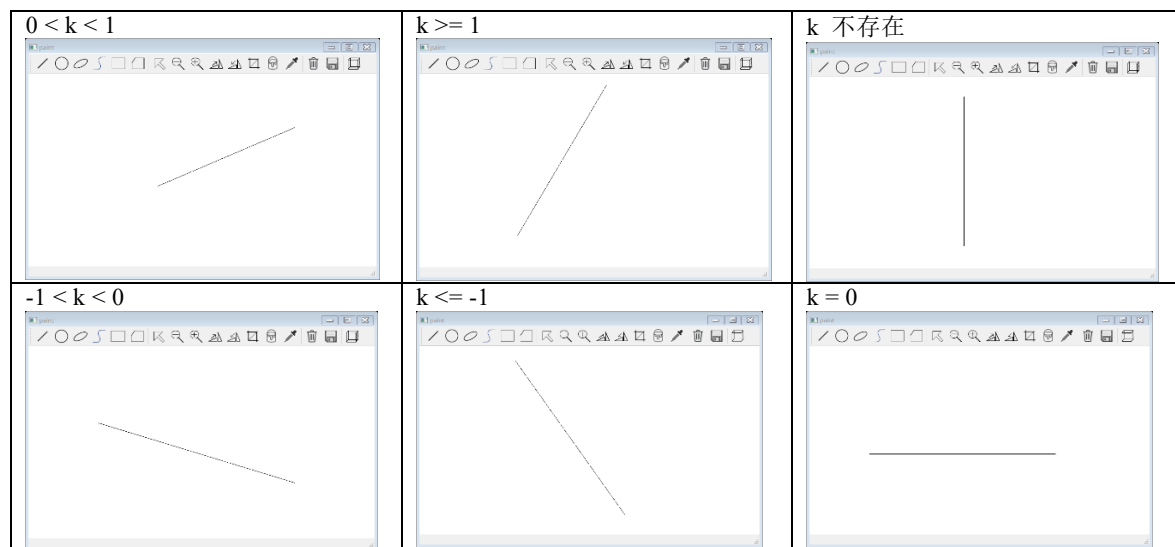
删除当前处于选中状态的图形。

## 4 软件测试

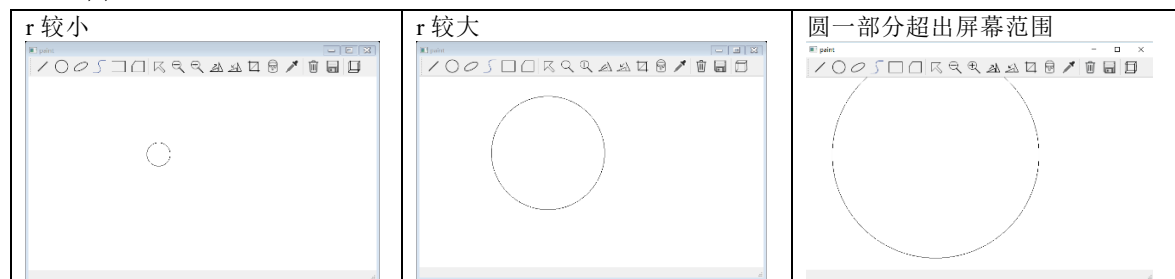
### 4.1 绘制

#### 4.1.1 直线

在各种斜率情况下，都能正确绘制直线，效果如下



#### 4.1.2 圆

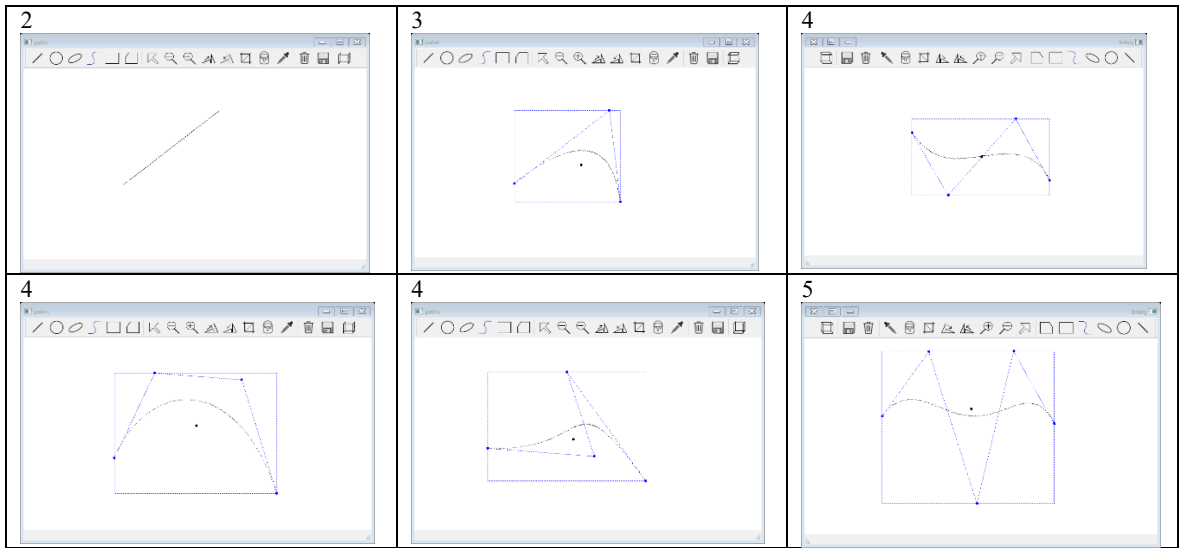


#### 4.1.3 椭圆



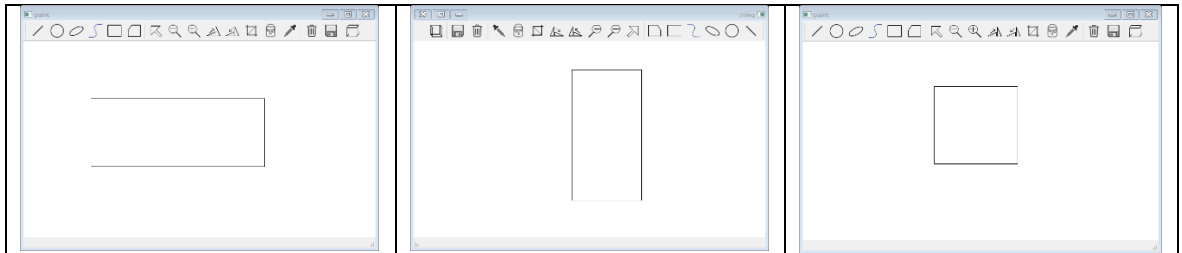
#### 4.1.4 曲线

对不同的控制点个数



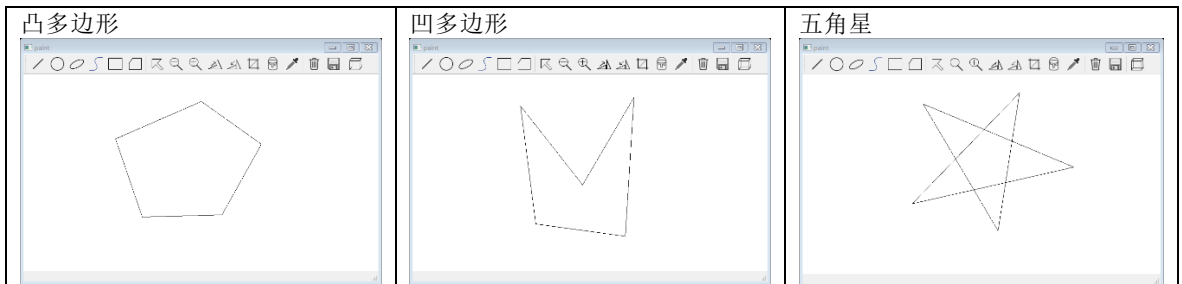
#### 4.1.5 矩形

各种形状的矩形



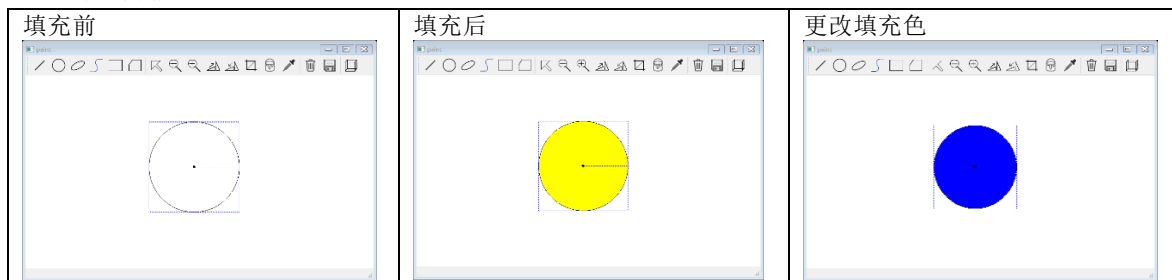
#### 4.1.6 多边形

绘制三个顶点及以上的各种多边形

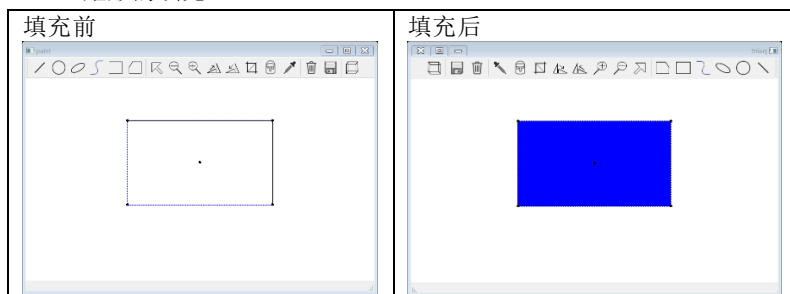


## 4.2 填充

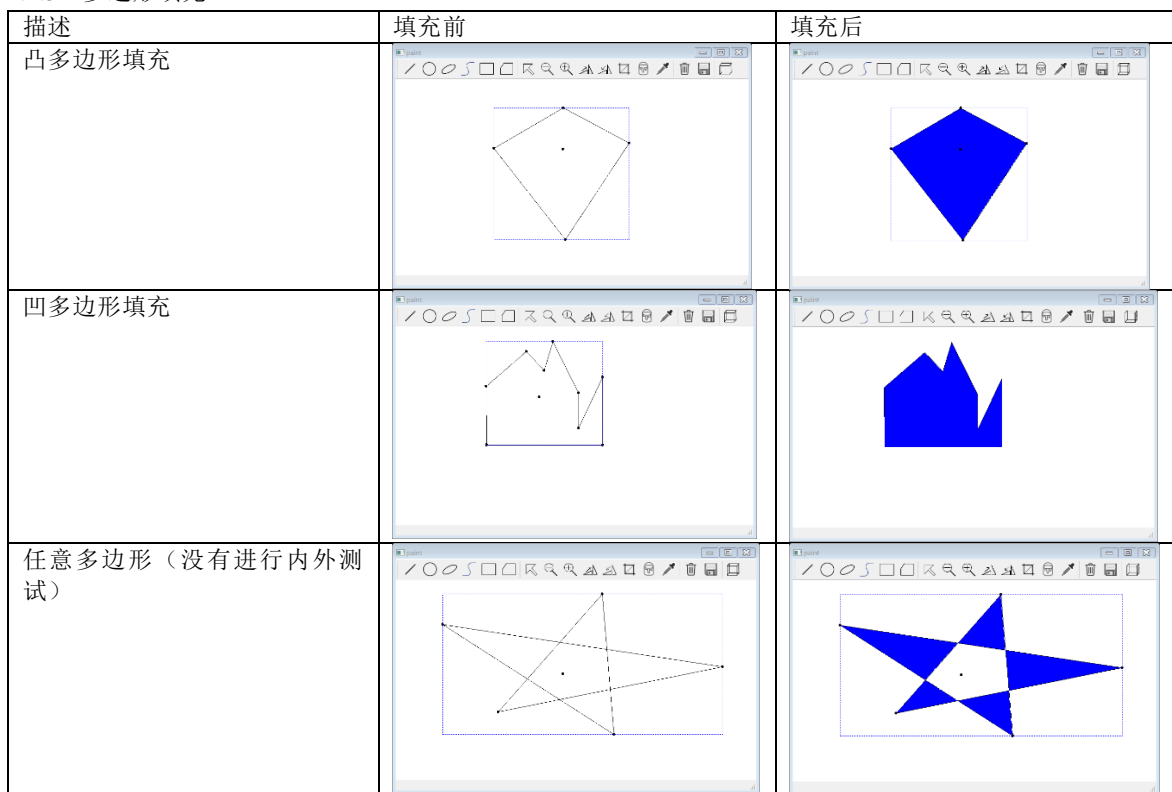
### 4.2.1 圆的填充



### 4.2.2 矩形的填充



### 4.2.3 多边形填充

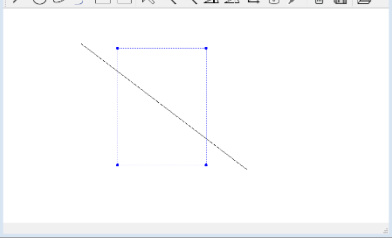
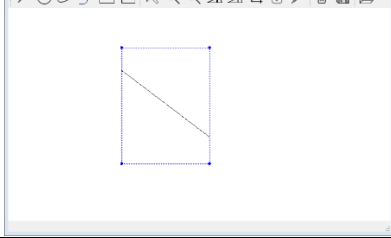
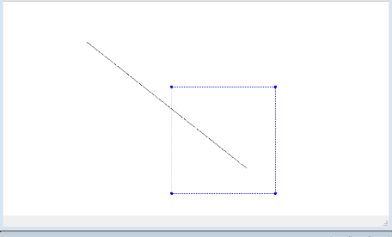
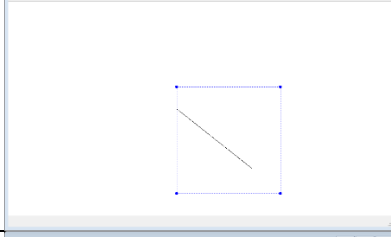
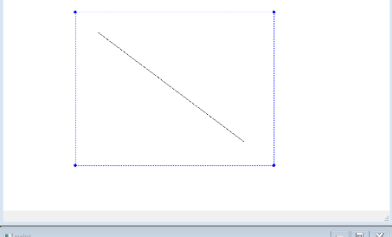
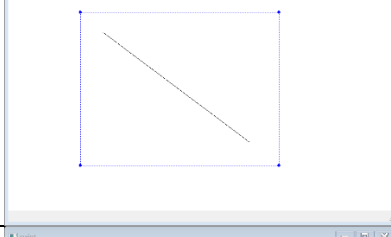
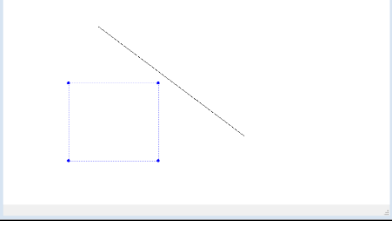
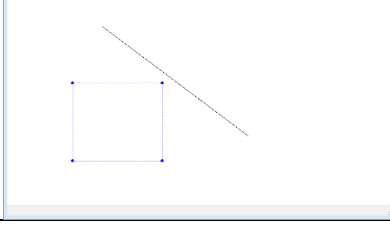




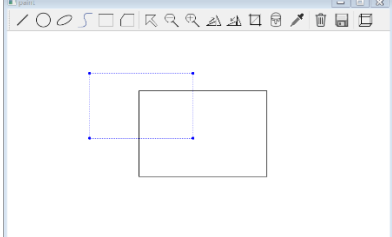
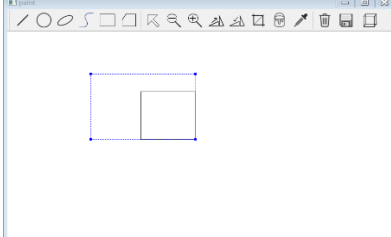
### 4.3 裁剪(内裁剪)

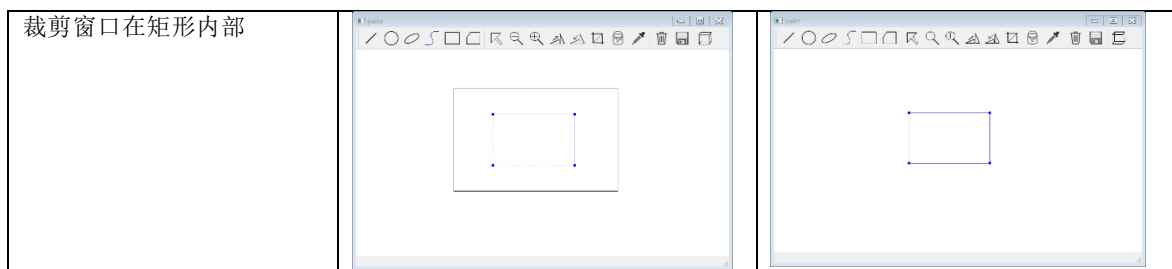
#### 4.3.1 线段裁剪

根据线段与裁剪窗口的位置关系进行测试

描述	裁剪前	裁剪后
线段穿过裁剪窗口，两个顶点均不在裁剪窗口内部		
线段一点在裁剪窗口内		
线段完全在裁剪窗口内		
线段在裁剪窗口外，这种情况下，虽为外裁剪，但作为交互性的应用，可以保留不与裁剪窗口相交的图形		

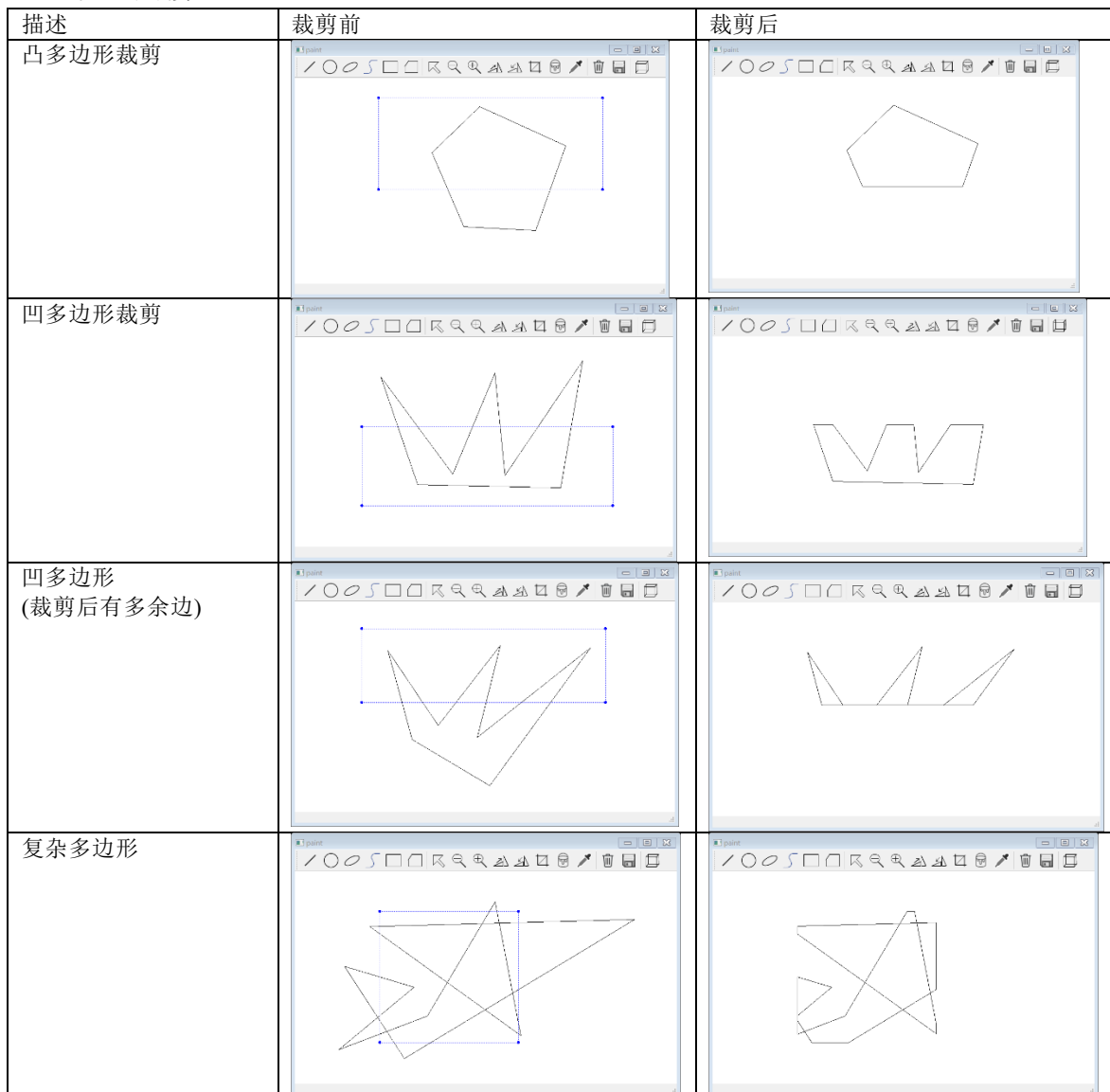
#### 4.3.2 矩形裁剪

描述	裁剪前	裁剪后
矩形与裁剪窗口部分相交		



矩形完全在裁剪窗口内部以及外部的情况，和线段裁剪类似

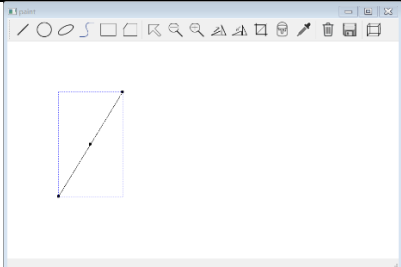
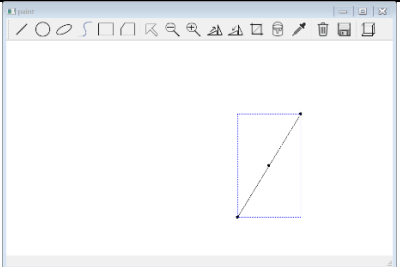
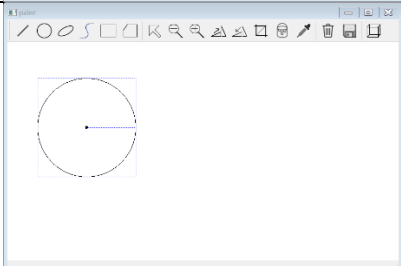
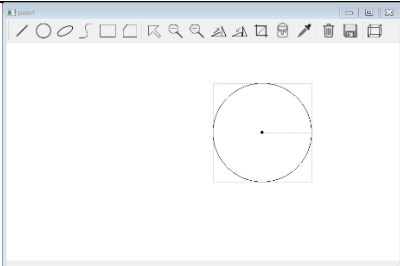
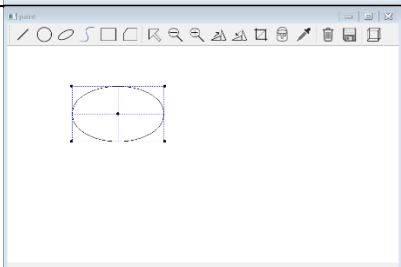
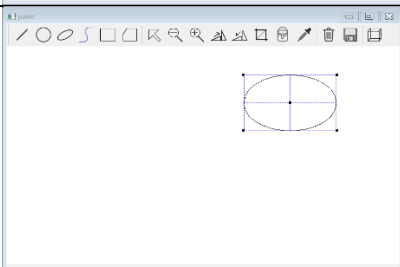
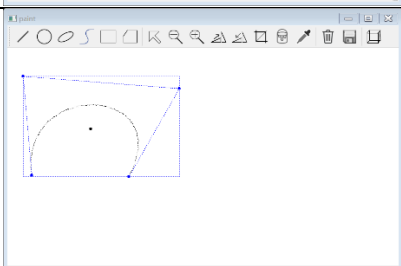
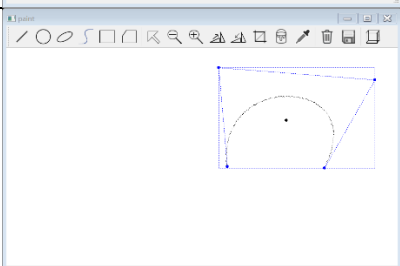
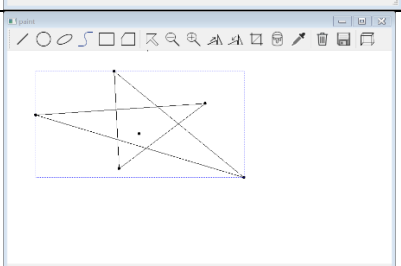
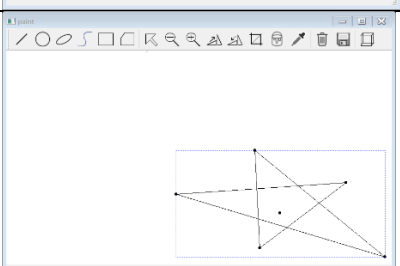
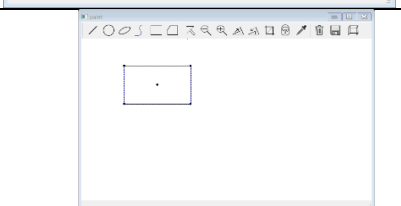
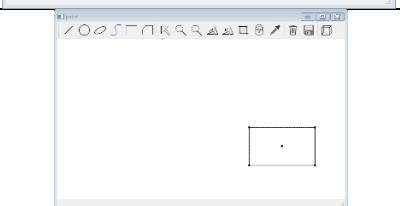
#### 4.3.3 多边形裁剪



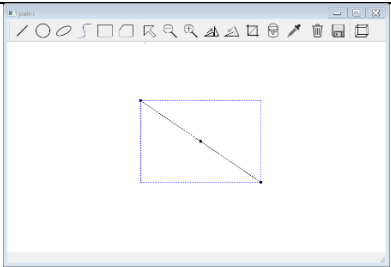
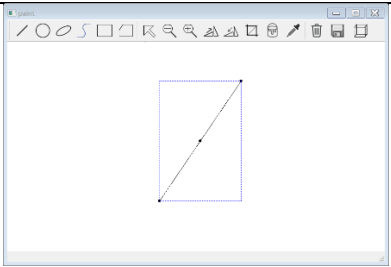
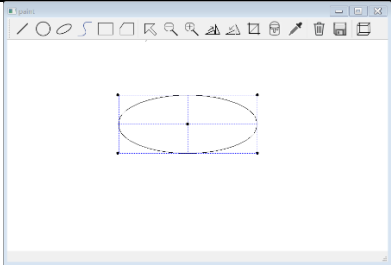
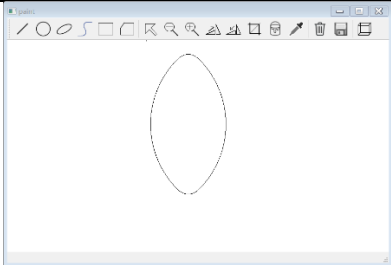
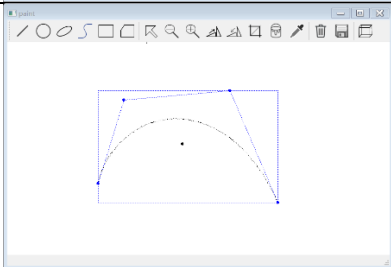
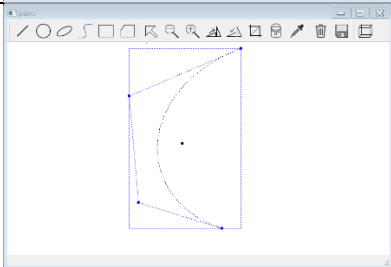
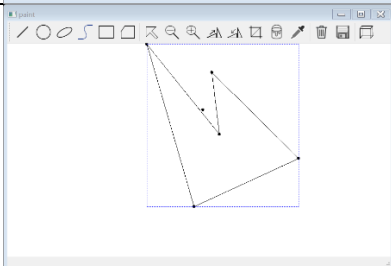
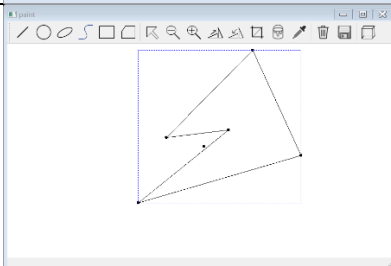
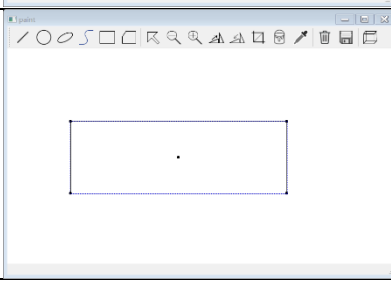
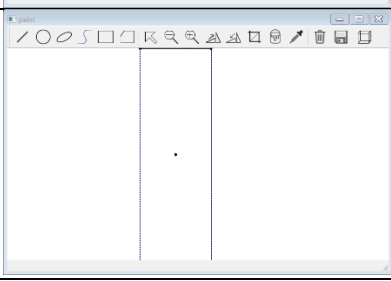
#### 4.3.4 填充区域

填充区域只支持多边形区域以及矩形区域的裁剪，是在对多边形裁剪后重新填充实现的。

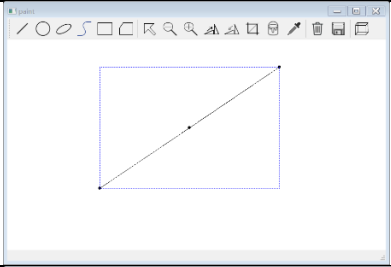
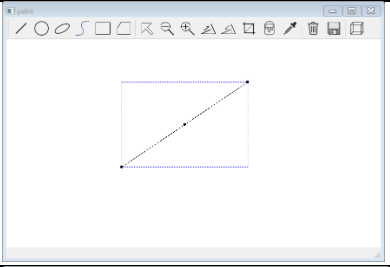
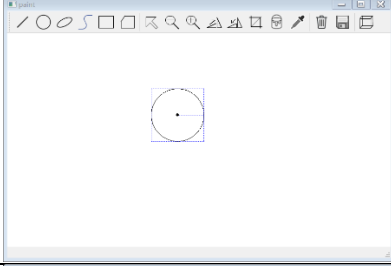
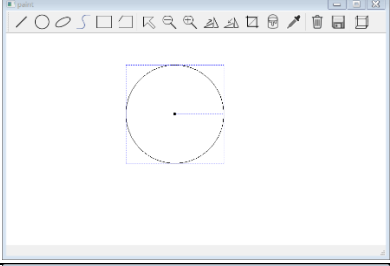
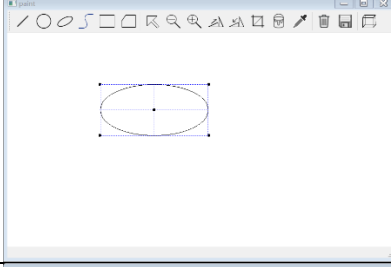
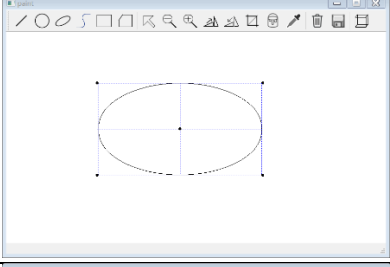
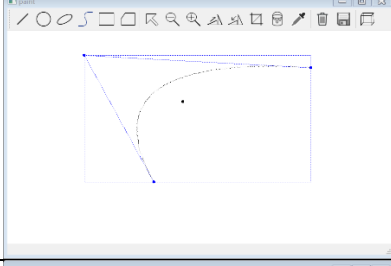
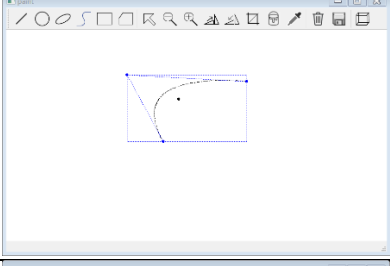
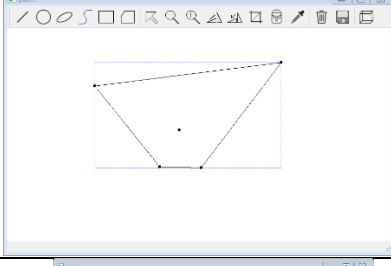
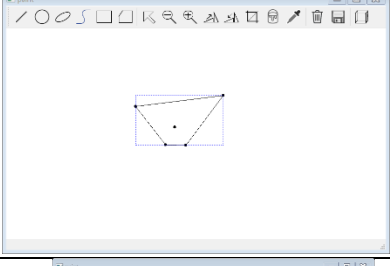
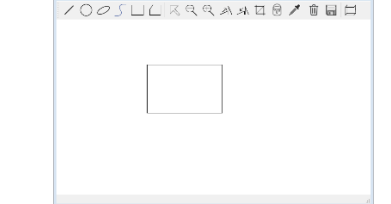
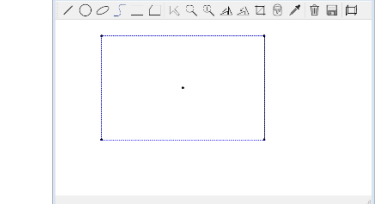
#### 4.4 平移（填充区域平移之后重新填充）

直线		
圆		
椭圆		
曲线		
多边形		
矩形		

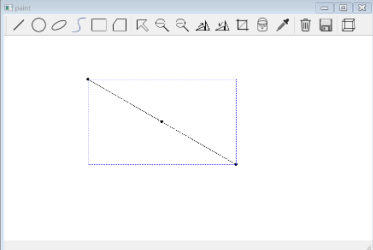
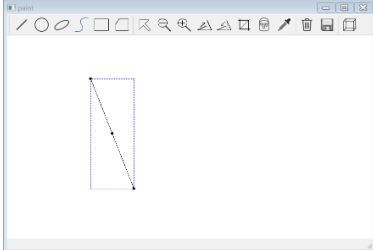
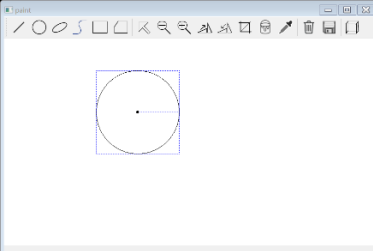
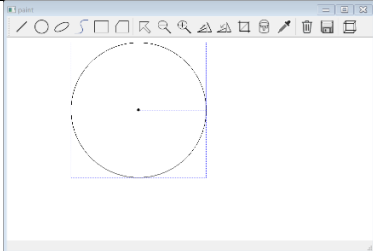
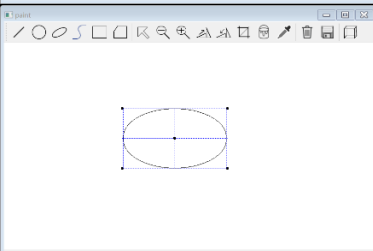
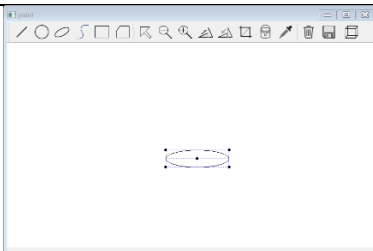
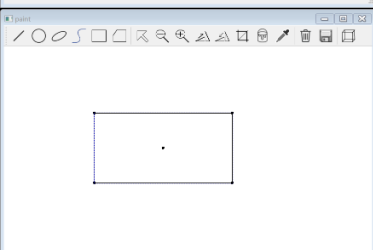
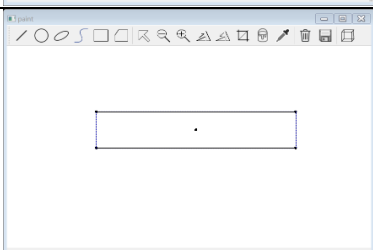
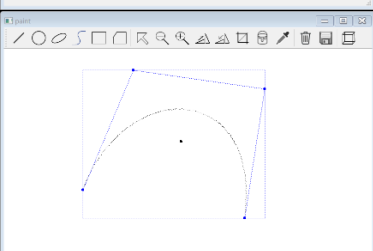
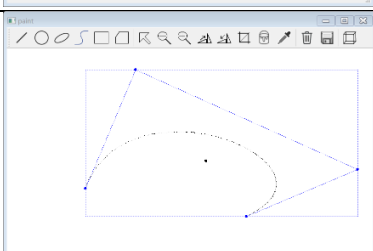
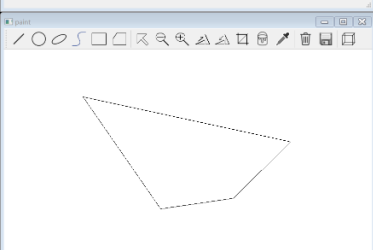
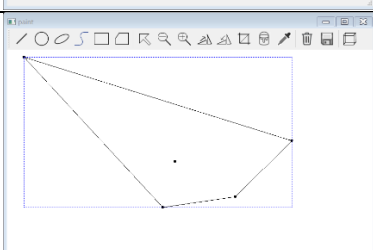
4.5 旋转（以图形中心为旋转中心）

直线		
圆	无需操作	
椭圆		
曲线		
多边形		
矩形		

#### 4.6 缩放(填充区域的缩放：缩放之后重新填充)

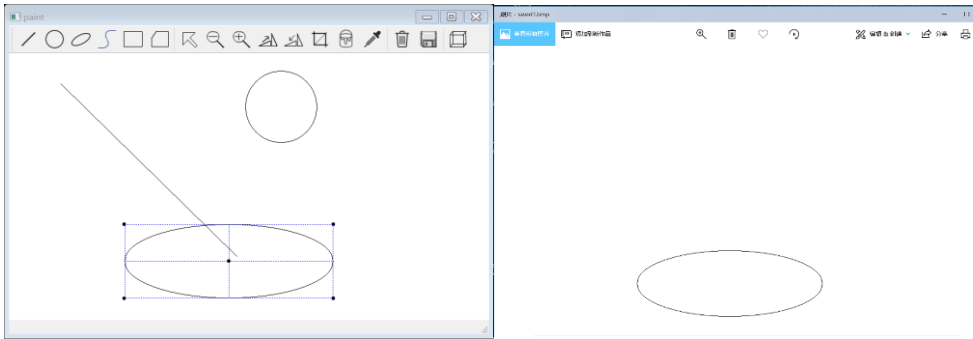
直线（缩小）		
圆		
椭圆		
曲线		
多边形		
矩形		

## 4.7 图形编辑

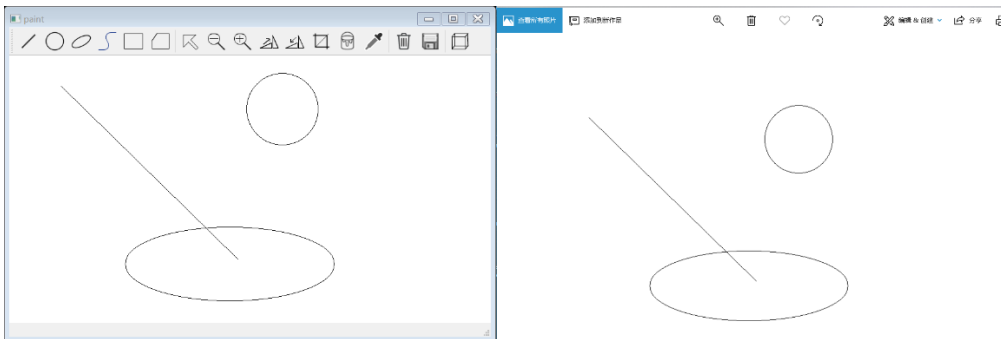
图形	编辑前	操作	编辑后
直线		拖动右下端点	
圆		拖动圆上的点	
椭圆		拖动编辑框左上角（四个角可编辑）	
矩形		拖动右下角（可编辑四个顶点）	
曲线		拖动其中一个控制点	
多边形		拖动其中一个顶点	

## 4.8 图形保存

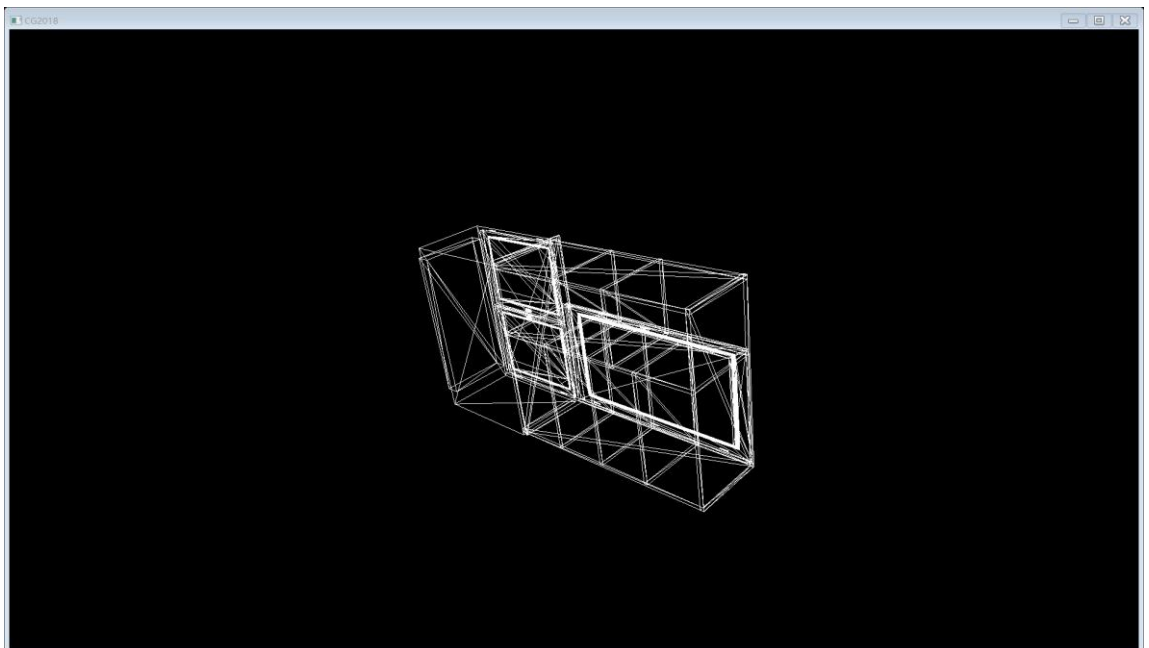
### 4.8.1 有图形选中的情况下，只保存选中的图形

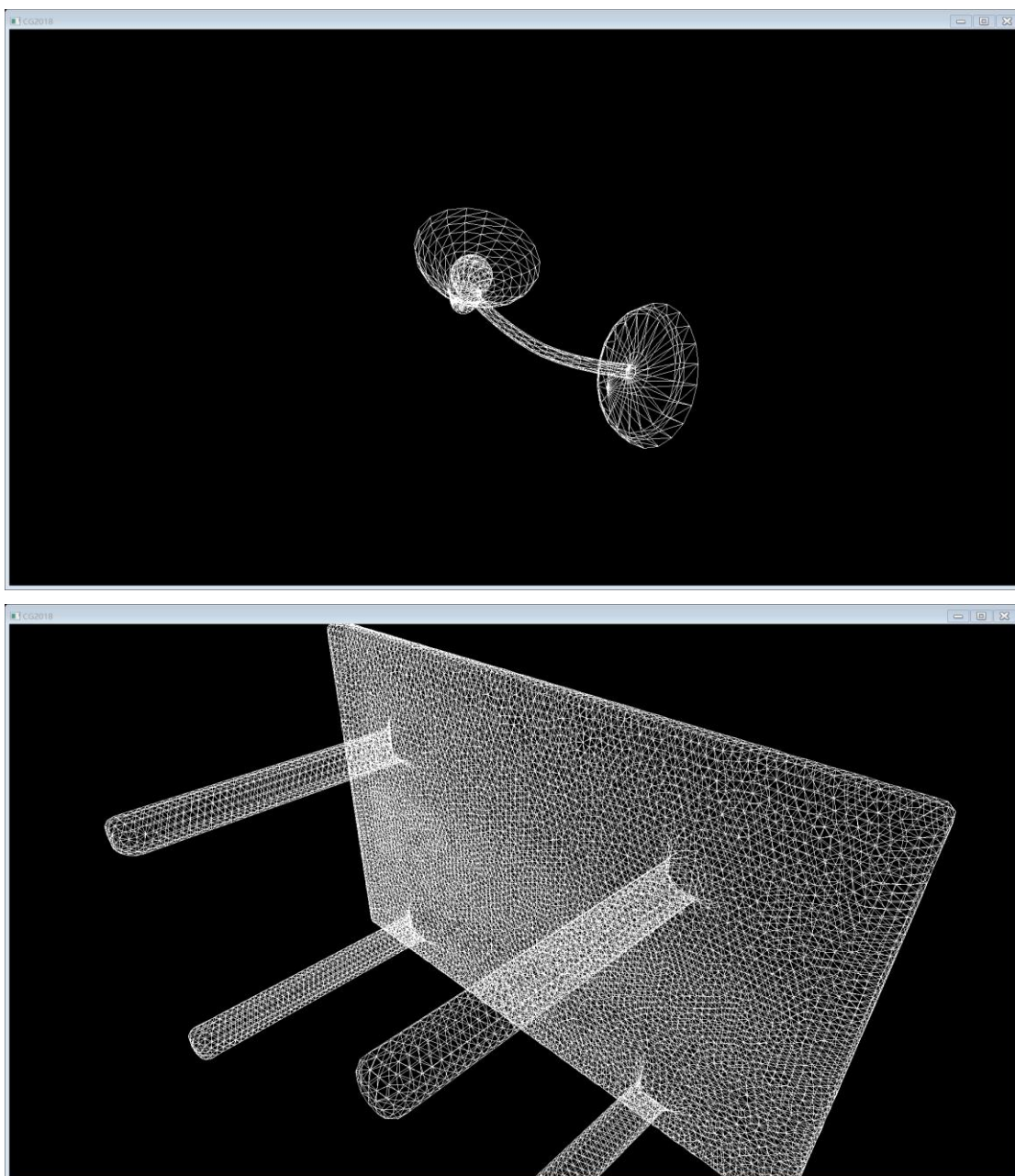


### 4.8.2 若没有图形选中，保存所有图形



## 4.9 三维模型显示



**References:**

- [1] Donald Hearn, M.Pauline Baker 《Computer Graphics》
- [2] QT documentation <https://doc.qt.io/>