《计算机图形学》系统技术报告

作者姓名*　王明

(南京大学 计算机科学与技术系，南京　210093)

摘　要:　利用 Qt 搭建起实验的大框架，完成了直线、圆、椭圆、多边形、曲线的绘制、平移、旋转、编辑、裁剪等功能以及填充图元的生成和 3D 模型的显示。最后为实现用户友好性增加了调色板、清屏、保存图片等功能，并且实现了实时绘制的显示功能。

关键词:　图形绘制；图形显示；图形处理；用户友好；

1　引言

1.1　开发环境：

编程语言：C++　　开发平台：Window10　　GUI 开发框架：Qt

1.2　实验思路：

10 月份搭建起 Qt 实验框架，完成了直线（DDA 画线算法）和圆（Bresenham 画圆算法）的输入。11 月份首先是完成椭圆的绘制（Bresenham 画椭圆算法）和填充图元的生成（扫描线边界填充），然后重点放在了图元的变换和编辑上，变换利用的是对于当前的像素点进行矩阵变换，进行一个坐标的映射，编辑利用的是对于图元控制点的对应变换。除此之外，实现了颜色选择、清屏重绘等辅助功能。为实现用户友好性，以上功能均通过鼠标操作。12 月份完成剩下的功能，如直线裁剪和 3D 显示等等，最后完善好系统技术报告和系统实用说明。

2　算法介绍

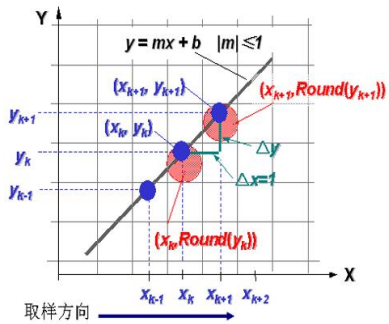
2.1　直线

2.1.1　直线的绘制

2.1.1.1　理论篇

DDA 画直线算法：

基本思想：直接利用直线的方程式，根据斜率，对于一个方向的坐标按照加 1 递增，另一个方向累加增量，计算精确值然后取整。



具体步骤：以直线的斜率在 0 到 1 之间为例，从直线的左端点到右端点，每次使得 x 坐标递增 1，然后每一步确定 y 的情况，通过在前一次 y 值的基础上加上增量精确地算出 y 的值，然后取整就得到了 y 坐标。

算法分析：该算法的想法比较符合正常的思维，就是控制一个方向的增长情况，而去关注另一个方向的坐标变化。比较巧妙的是该算法根据直线方程式，计算出了一个方向每次的增量（这个增量是固定的），用加法代替了乘法，大大地提高了算法的效率。

2.1.1.2　代码篇

计算斜率确定是延 x 还是 y 方向增长，然后每次进行增量的累加。（dx 和 dy 有一个为 1）

---

* 作者简介:南京大学计算机系 16 级本科生

```
void DrawWidget::ddaLine(QPainter &painter, int x1, int y1, int x2, int y2)
{
    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++)
    {
        painter.drawPoint((int)(x+0.5),(int)(y+0.5));                △ use of
        x+=dx;
        y+=dy;
    }
}
```

2.1.2 直线的平移

2.1.2.1 理论篇

基本思想：根据平移的定义，将直线的控制点（也就是起点和终点）平移到对于位置，剩下的点通过绘制直线来完成平移。

具体步骤：对于直线来讲，其控制点就是起点和终点，因此进入平移模式后，记录下鼠标的起点和终点，计算出终点和起点的差，这个差就是要求的平移量，将直线的起点和终点分别加上这个平移量，就可以得到平移后直线的控制点的坐标，然后利用直线的绘制算法将其绘制出来就完成了直线的平移。

算法分析:巧妙利用了直线绘制的算法，这样就只需要将控制点进行平移变换而提升了效率。

2.1.2.2 代码篇

首先计算出平移量(delta_x,delta_y)，然后用现在的坐标(p1,p2)加上平移量，得到平移后的坐标(p3,p4)，然后用绘制算法画出来。

```
int delta_x = a2.x() - a1.x();
int delta_y = a2.y() - a1.y();
QPoint delta = QPoint(delta_x,delta_y);
QPoint p3 = p1 + delta;
QPoint p4 = p2 + delta;

 if(choose == 2)
     ddaLine(qimage,current_color,p3,p4);
```

2.1.3 直线的旋转

2.1.3.1 理论篇

基本思想：确定参照点，确定旋转角度，将控制点先旋转变换，然后再通过控制点绘制出来。

$$x_1=x_r+(x-x_r)\cos\theta-(y-y_r)\sin\theta$$

$$y_1=y_r+(x-x_r)\sin\theta+(y-y_r)\cos\theta$$

具体步骤：直线的旋转首先要确定参照点，为了美观以及也是最正常的想法就是以直线的中点为参照点，然后要确定的就是旋转的角度，这个可以用鼠标位置和直线中点的连线的斜率表示。有了参照点和旋转角度之后，就和平移一样，同样是要确定直线的起点和终点 2 个控制点变换后的坐标，根据右边的坐标旋转的变换公式即可得到变换后的坐标，然后调用绘制算法就可以完成直线的旋转。

算法分析：和平移一样，很自然的想到不需要对所有点进行旋转变换，只需对控制点变换，剩下的通过绘制完成，提升了效率。

2.1.3.2 代码篇

求出直线的中点作为参照点，根据鼠标位置求出旋转角度，计算出直线起点和终点旋转后的位置，调用绘制算法根据新起点和新终点完成绘制。

```
int center_x = (p1.x()+p2.x())/2;
int center_y = (p1.y()+p2.y())/2;
double x = b2.x() - center_x;
double y = b2.y() - center_y;
angle = atan(y/x);
if(choose == 2)
{
    double len = sqrt(pow((p1.x()-p2.x()),2) + pow(p1.y()-p2.y(),2));
    double p1x = center_x + len/2;
    double p1y = center_y;
    int x1 = (int)(coordinate_x(p1x,p1y,center_x,center_y,angle));
    int y1 = (int)(coordinate_y(p1x,p1y,center_x,center_y,angle));
    int x2 = 2 * center_x - x1;
    int y2 = 2 * center_y - y1;
    QPoint pp1 = QPoint(x1,y1);
    QPoint pp2 = QPoint(x2,y2);
    ddaLine(qimage,current_color,pp1,pp2);
```

2.1.4 直线的编辑

2.1.4.1 理论篇

基本思想：鼠标捕获控制点，拖动改变控制点坐标，然后重新绘制。

具体步骤：直线的编辑就是实现直线的起点和终点可以移动，其思想比较简单：就是通过鼠标事件锁定某个端点，然后拖动鼠标，随鼠标的移动而改变自身的坐标，然后通过绘制算法绘制出来。其难点是如何捕获端点，因为一个像素点是很小的，你不能希望鼠标正好点击在某一个像素点，这显然是不行的，就算行，对于用户也是相当的不友好，因此比较好的做法是以该端点为圆心，画一个小的圆，点击在此区域内都算是选中了该端点。实现了端点的选中，直线的编辑很好实现了。

算法分析：这样做是很自然的想法，其中以面代点是一个很巧妙的操作。

2.1.4.2 代码篇

通过半径为 4 的圆作为点击区域捕获端点坐标，然后改变坐标到鼠标的位置，调用绘制算法就行了。

```
if(choose == 2)
{
    c1 = event->pos();
    if((c1.x()-p1.x())*(c1.x() - p1.x())+(c1.y()-p1.y())*(c1.y()-p1.y()) <= 16)
        valid_dot = 1;
    if((c1.x()-p2.x())*(c1.x() - p2.x())+(c1.y()-p2.y())*(c1.y()-p2.y()) <= 16)
        valid_dot = 2;
}
```

```
 if(choose == 2)
 {
     if(valid_dot == -1)
         ddaLine(qimage,current_color,p1,p2);
     else if(valid_dot == 1)
         ddaLine(qimage,current_color,c2,p2);
     else if(valid_dot == 2)
         ddaLine(qimage,current_color,p1,c2);
```

2.1.5　直线的裁剪

2.1.5.1　理论篇

直线的梁友栋-Barsky 参数裁剪算法：

$u \cdot pk \le qk$，　$k=1，2，3，4$

$p1=-\Delta x$，　$q1=x1-xmin$

$p2=\Delta x$，　$q2=xmax-x1$

$p3=-\Delta y$，　$q3=y1-ymin$

$p4=\Delta y$，　$q4=ymax-y1$

基本思想：将与视窗求交的方程式参数化，从而根据不同的情况判断直线与视窗的相对位置以及由内而外还是由外而内的直线走向，最后通过计算出的两个参数来裁剪出需要的部分。

具体步骤：首先通过右边的式子中的 q 的情况来判断直线是不是在窗口外（$p < 0$ 直线在窗口外，直接舍弃，　$q >= 0$ 直线在窗口内，　要做进一步判断）。再看 p 的情况来判断直线的走向（$p < 0$ 直线从外部到内部，　可计算出参数 r，取所有 r 和 0 中最大的值记为 u1，　$p > 0$ 时　直线从内部到外部，可计算出参数 r，取所有 r 和 1 中的最小值记为 u2），如果 u1 > u2，则说明直线完全在视窗外面也应该舍去，否则，u1 到 u2 就是所求的直线被裁剪后留下的部分。

算法分析：本质上还是用直线和视窗边界进行求交，但是巧妙的地方是，将求交的方程参数化后不仅形式得到了统一，同时通过 p 和 q 的几何意义大大简化了求交运算，效率大大提高。

2.1.5.2　代码篇

算法的核心就是通过判断 p 的大小以及计算出的 r 值去不断的更新 u1 和 u2　的值从而得到裁剪后的直线的控制点。

```
double r;
for(int i = 0; i < 4; i++)
{
    r = (double)q[i] / (double)p[i];
    if(p[i] < 0)
    {
        u1 = qMax(u1,r);
        if(u1 > u2)
            judge = true;
    }
    if(p[i] > 0)
    {
        u2 = qMin(u2,r);
        if(u1 > u2)
            judge = true;
    }
    if(p[i] == 0 && q[i] < 0)
    {
        judge = true;
    }
}
```

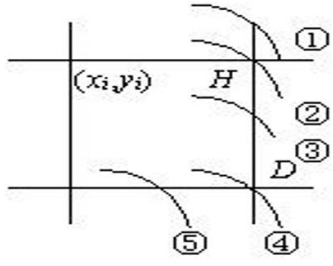
2.2　圆

2.2.1　圆的绘制

2.2.1.1　理论篇

Bresenham 画圆算法：

基本思想：根据切线的斜率来决定一个方向进行不断加一的延伸，而另一个方向采用逐步决策法，以 0<x<y 的 1/8 圆周为例，对于当前像素点(x,y)，它的下一个像素点只有可能是(x+1,y)或者(x+1,y-1)，所以通过比较在 x+1 时 y 的真实值与 y 还是 y-1 接近来选择其中一个像素点，不断重复这个过程。



具体步骤：由控制点可以计算出圆的圆心和半径，最核心的是对于当前点(x(i),y(i))去确定下一个点(x(i+1),y(i+1))，其中 x(i) = xi, y(i) = yi, x(i+1) = xi + 1,就要决策出 y(i+1) = yi 还是 yi – 1。定义 dH = (xi + 1)^2 + yi^2 – r^2, dD = r^2 - (xi + 1)^2 – (y1 - 1)^2，由此可以构造出决策参数 p(i) = dH – dD, 通过分析可得，p(i) < 0 应该选 H 点，而 p(i) > 0 应该选 D 点，也就是说每次只要知道 p(i)的正负就可以决策出选哪个像素点，进一步通过 p(i+ 1)与 p(i)的差可得他们之间的递推关系：p(i) < 0 时，p(i+1) = p(i) + 4*x(i) + 6；p(i) >= 0 时，p(i+1) = p(i) + 4(x(i) – y(i)) + 10；p(1) = 3 – 2 * R 可以单独计算，这样就形成一个完整的决策链，根据此决策链就可以绘制出 1/8 的圆周，然后通过对称的性质来补全。

算法分析：做法其实也是很自然的，也是保持一个方向的固定增长，去决策另一个方向的增长情况。有一个细节 dH 和 dD 其实不是精确的距离，而是近似过来的，不但不影响决策的正确性，反而会简化了计算，这是巧妙的一点，另一个巧妙的地方是对于决策参数也使用了递推，由 p(i)来求 p(i+1)，又减少了计算量。我觉得这两点是这个算法极其高效的原因。

2.2.1.2 代码篇

先初始化初始化参数（主要是 p1 = 3 – 2 * r），通过 p(i+1)和 p(i)的递推式不断计算出 p(i)，根据 p(i)的正负来决策出正确的像素点。

```
int x, y, p;
x = 0;
y = r;
p = 3 - 2 * r;
for(; x <= y; x++)
{
    qimage->setPixelColor(x + center_x, y + center_y,pen);
    qimage->setPixelColor(x + center_x, - y + center_y,pen);
    qimage->setPixelColor(- x + center_x, y + center_y,pen);
    qimage->setPixelColor(- x + center_x, - y + center_y,pen);
    qimage->setPixelColor(y + center_x, x + center_y,pen);
    qimage->setPixelColor(y + center_x, - x + center_y,pen);
    qimage->setPixelColor(- y + center_x, x + center_y,pen);
    qimage->setPixelColor(- y + center_x, - x + center_y,pen);
    if(p >= 0)
    {
        p += 4 * (x - y) + 10;
        y--;
    }
    else
    {
        p += 4 * x + 6;
    }
}
```

2.2.2 圆的平移

2.2.2.1 理论篇

基本思想：和直线平移类似，只需对控制点进行平移变换，然后绘制出来。

具体步骤：和直线类似，通过鼠标的坐标计算出平移量，然后将控制点加上这个平移量，最后再调用绘制算法绘制出完整的圆。

算法分析：和直线的平移是一样的。

2.2.2.2 代码篇

先变换控制点，然后绘制。

```
int delta_x = a2.x() - a1.x();
int delta_y = a2.y() - a1.y();
QPoint delta = QPoint(delta_x,delta_y);
QPoint p3 = p1 + delta;
QPoint p4 = p2 + delta;
```

```
else if(choose == 3)
    BresenhamCircle(qimage,current_color,p3,p4);
```

2.2.3 圆的旋转

2.2.3.1 理论篇

基本思想：因为圆是中心对称的，因此从效果上旋转前和旋转后没有什么区别。

具体步骤：因为圆旋转前和旋转后并没有什么区别，因此直接画出来就行。

算法分析：为什么不采用对每个像素点乘上一个旋转矩阵得到相应的新位置？因为存在取整操作，变换后的图形很有可能不是圆的样子了，而且会很不封闭，因此不如直接画出来。

2.2.3.2 代码篇

直接调用圆的绘制算法即可。

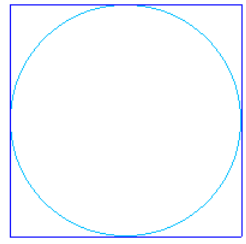
```
else if(choose == 3)
{
    BresenhamCircle(qimage,current_color,p1,p2);
}
```

2.2.4 圆的编辑

2.2.4.1 理论篇

基本思想：对于圆进行编辑的话，本质上是改变圆的半径大小，再进一步转化为，圆的半径大小是由控制点来控制的，这样对于圆的编辑就变成了对于控制点的改变。

具体步骤：当鼠标点击编辑按钮的时候，首先会自动生成一个圆的包围盒，也就是一个各边都和圆相切的正方形，这时候正方形的四个顶点就是圆的控制点，此时用鼠标选中其中一个控制点进行拖拉即可改变圆的大小，完成对圆的编辑。



算法分析：我觉得这样做是很自然的想法，要实现对圆的编辑，本质上就是改变圆的控制点，因为控制点决定了圆的一切属性，然后绘制出包围盒是一个很用户友好的操作，否则用户看不到控制点在哪里。

2.2.4.2 代码篇

和直线的编辑一样，同样是使用以点代面的做法，首先捕捉选中的是哪一个控制点。

```
else if(choose == 3)
{
    c1 = event->pos();
    int d;
    QPoint pp1,pp2,pp3,pp4;
    int x1 = p1.x();
    int x2 = p2.x();
    d = x2 - x1;
    pp1 = p1;
    pp2 = QPoint(p1.x()+d,p1.y());
    pp3 = QPoint(p1.x(),p1.y()+d);
    pp4 = QPoint(p1.x()+d,p1.y()+d);
    if((c1.x()-pp1.x())*(c1.x() - pp1.x())+(c1.y()-pp1.y())*(c1.y()-pp1.y()) <= 16) {...}
    if((c1.x()-pp2.x())*(c1.x() - pp2.x())+(c1.y()-pp2.y())*(c1.y()-pp2.y()) <= 16) {...}
    if((c1.x()-pp3.x())*(c1.x() - pp3.x())+(c1.y()-pp3.y())*(c1.y()-pp3.y()) <= 16) {...}
    if((c1.x()-pp4.x())*(c1.x() - pp4.x())+(c1.y()-pp4.y())*(c1.y()-pp4.y()) <= 16) {...}
}
```

然后根据变化后的控制点重新绘制出新的圆。

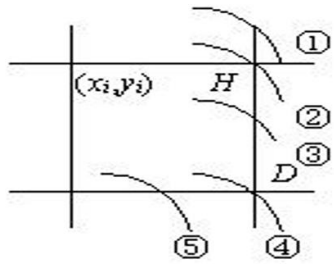
```
else if(choose == 3)
{
    int d = p2.x() - p1.x();
    if(valid_dot == -1)
        BresenhamCircle(qimage,current_color,p1,p2);
    else if(valid_dot == 1)
    {
        QPoint pp2 = QPoint(p1.x()+d,p1.y()+d);
        int delta = pp2.x() - c2.x();
        QPoint pp1 = QPoint(pp2.x()-delta,pp2.y()-delta);
        BresenhamCircle(qimage,current_color,pp1,pp2);
    }
    else if(valid_dot == 2) {...}
    else if(valid_dot == 3) {...}
    else if(valid_dot == 4) {...}
```

2.3 椭圆

2.3.1 椭圆的绘制

2.3.1.1 理论篇

Bresenham 画椭圆算法：



基本思想：根据切线的斜率来决定一个方向进行不断加一的延伸，而另一个方向采用逐步决策法，以 0<x<y 的椭圆为例，对于当前像素点(x,y)，它的下一个像素点只有可能是(x+1,y)或者(x+1,y-1)，所以通过比较在 x+1 时 y 的真实值与 y 还是 y-1 接近来选择其中一个像素点，不断重复这个过程。所以其核心思想还是要找出一个决策参数，用来决定当前点的下一个点的位置。

具体步骤：首先要明确只需绘制 1/4 的椭圆，然后这 1/4 的椭圆必须根据切线为 1 的点分成两段，这两段有一段是延着 x 轴加一延伸，有一段是沿着 y 轴加一延伸。由控制点可以计算出椭圆的长轴和短轴，最核心的是对于当前点(x(i),y(i))去确定下一个点(x(i+1),y(i+1))，其中 x(i) = xi, y(i) = yi, x(i+1) = xi + 1,就要决策出 y(i+1) = yi 还是 yi – 1。当沿 x 轴方向加一延伸的时候，定义 dH 和 dD 是当前点到 H 和 D 的距离，由此可以构造出决策参数 p(i) = dH – dD, 通过分析可得，p(i) < 0 应该选 H 点，而 p(i) > 0 应该选 D 点，也就是说每次只要知道 p(i)的正负就可以决策出选哪个像素点，进一步通过 p(i+1)与 p(i)的差可得他们之间的递推关系：p(i) < 0 时，p(i+1) = p(i) + 2 * b * b * (2 * x(i) + 3)；p(i) >= 0 时，p(i+1) = p(i) + 2 * b * b * (2 * x(i) + 3) - 4 * a * a * (y(i) - 1)；p(1) = 2 * b * b - 2 * b * a * a + a * a 可以单独计算。当沿 y 轴方向加一延伸的时候，通过分析可得，p(i) < 0 应该选 H 点，而 p(i) > 0 应该选 D 点，也就是说每次只要知道 p(i)的正负就可以决策出选哪个像素点，进一步通过 p(i+1)与 p(i)的差可得他们之间的递推关系：p(i) < 0 时，p(i+1) = p(i) - 2 * a * a * y(i) - a * a + 2 * b * b * x(i+1) + 2 * b * b；p(i) >= 0 时，p(i+1) = p(i) - 2 * a * a * y(i) - a * a。这样就形成一个完整的决策链，根据此决策链就可以绘制出 1/4 的椭圆，然后通过对称的性质来补全。

算法分析：做法其实也是很自然的，也是保持一个方向的固定增长，去决策另一个方向的增长情况，只是要注意通过斜率来区分沿哪个方向增长。有一个细节 dH 和 dD 其实不是精确的距离，而是近似过来的，不但不影响决策的正确性，反而会简化了计算，这是巧妙的一点，另一个巧妙的地方是对于决策参数也使用了递推，由 p(i)来求 p(i+1)，又减少了计算量。我觉得这两点是这个算法极其高效的原因。

2.3.1.2 代码篇

核心的代码就是计算出 p，并根据递推计算下一个 p，根据 p 的正负来决策像素点的位置。

```
p = 2 * b * b - 2 * b * a * a + a * a;
int key_p = round((double)(a * a) / sqrt((double)(a * a + b * b)));
for(; x <= key_p; x++)
{
    qimage->setPixelColor(x + center_x, y + center_y,pen);
    qimage->setPixelColor(x + center_x, - y + center_y,pen);
    qimage->setPixelColor(- x + center_x, y + center_y,pen);
    qimage->setPixelColor(- x + center_x, - y + center_y,pen);
    if(p >= 0)
    {
        p += 2 * b * b * (2 * x + 3) - 4 * a * a * (y - 1);
        y--;
    }
    else
    {
        p += 2 * b * b * (2 * x + 3);
    }
}
p = b * b * (x * x + x) + a * a * (y * y - y) - a * a * b * b;
for(; y >= 0; y--) {...}
```

2.3.2 椭圆的平移

2.3.2.1 理论篇

基本思想：和直线平移类似，只需对控制点进行平移变换，然后绘制出来。

具体步骤：和直线类似，通过鼠标的坐标计算出平移量，然后将控制点（椭圆的控制点其实就是其包围盒的顶点）加上这个平移量，最后再调用绘制算法绘制出完整的圆。

算法分析：和直线的平移是一样的。

2.3.2.2 代码篇

首先根据鼠标的偏移量计算出平移后控制点的坐标，然后调用绘制算法绘制即可。

```
int delta_x = a2.x() - a1.x();
int delta_y = a2.y() - a1.y();
QPoint delta = QPoint(delta_x,delta_y);
QPoint p3 = p1 + delta;
QPoint p4 = p2 + delta;

else if(choose == 4)
    DirectOval(qimage,current_color,p3,p4,-angle);
```

2.3.3 椭圆的旋转

2.3.3.1 理论篇

基本思想：一开始的想法是对于当前绘制好的椭圆，遍历每一个像素点，乘上旋转矩阵，得到新的像素点的位置，即可完成旋转，尝试了之后发现画出来的根本不是椭圆，究其原因，是对于变换后的像素点进行取整，因此旋转前后旋转后根本不是一一对应的。最终采用的是密集采样绘制椭圆的算法，就是将取点的间隔缩短（我使用的参数是 0.5），然后密集取点，这样即使有重合点，但由于取点的数量很大，还是可以形成一个椭圆的。

具体步骤：重构椭圆的绘制算法，采用密集取点的方法，结合旋转角，绘制出旋转后的斜椭圆。

算法分析：首先是不可以在原来的椭圆绘制算法里乘以旋转矩阵的方法来绘制，原因之前已经提到了，这里采用密集取点的方法相对来说效率会低一点，但是其功能还是实现得比较好。

2.3.3.2 代码篇

i 每次增加 0.5，通过 0.5 的步长实现密集取点，注意到在当前的点的上下左右四个点也被绘制了出来，这样绘制出来的椭圆可能会比较粗，但是可以尽可能保证其封闭性。

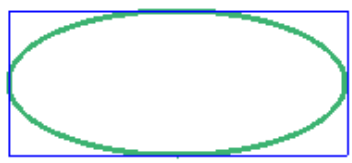
```
double th = acos(-1) / 180;
double i;
double x,y,tx,ty;
for(i=-180;i<=180;i=i+0.5)
{
    x=a*cos(i*th);
    y=b*sin(i*th);
    tx=x;
    ty=y;
    x=tx*cos(angle)-ty*sin(angle)+center_x;  /*坐标旋转*/
    y=center_y-(ty*cos(angle)+tx*sin(angle));
    qimage->setPixelColor((int)x, (int)y,pen);
    qimage->setPixelColor((int)(x+1), (int)y,pen);
    qimage->setPixelColor((int)x, (int)(y+1),pen);
    qimage->setPixelColor((int)(x-1), (int)y,pen);
    qimage->setPixelColor((int)x, (int)(y-1),pen);
}
```

2.3.4 椭圆的编辑

2.3.4.1 理论篇

基本思想：对于椭圆进行编辑的话，本质上是改变椭圆的长轴和短轴的大小，再进一步转化为，椭圆的长轴和短轴大小是由控制点来控制的，这样对于椭圆的编辑就变成了对于控制点的改变。

具体步骤：当鼠标点击编辑按钮的时候，首先会自动生成一个椭圆圆的包围盒，也就是一个各边都和椭圆相切的矩形，这时候矩形的四个顶点就是椭圆的控制点，此时用鼠标选中其中一个控制点进行拖拉即可改变椭圆的大小，完成对椭圆的编辑。



算法分析：我觉得这样做是很自然的想法，要实现对椭圆的编辑，本质上就是改变椭圆的控制点，因为控制点决定了椭圆的一切属性，然后绘制出包围盒是一个很用户友好的操作，不仅利于用户寻找控制点进行编辑，而且可以对于编辑后的椭圆的大小有一个直观的感受。

2.3.4.2 代码篇

同样是先根据鼠标点击的位置去捕获控制点的位置。

```
else if(choose == 4)
{
    c1 = event->pos();
    QPoint pp1,pp2,pp3,pp4;
    pp1 = p1;
    pp2 = QPoint(p2.x(),p1.y());
    pp3 = QPoint(p1.x(),p2.y());
    pp4 = p2;
    if((c1.x()-pp1.x())*(c1.x() - pp1.x())+(c1.y()-pp1.y())*(c1.y()-pp1.y()) <= 16)
    {
        valid_dot = 1;
        c1 = pp1;
    }
    if((c1.x()-pp2.x())*(c1.x() - pp2.x())+(c1.y()-pp2.y())*(c1.y()-pp2.y()) <= 16) {...}
    if((c1.x()-pp3.x())*(c1.x() - pp3.x())+(c1.y()-pp3.y())*(c1.y()-pp3.y()) <= 16) {...}
    if((c1.x()-pp4.x())*(c1.x() - pp4.x())+(c1.y()-pp4.y())*(c1.y()-pp4.y()) <= 16) {...}
}
```

然后根据改变后的控制点位置调用绘制算法绘制出相应的椭圆。

```
else if(choose == 4)
{
    if(valid_dot == -1)
        DirectOval(qimage,current_color,p1,p2,-angle);
    else if(valid_dot == 1)
        DirectOval(qimage,current_color,c2,p2,-angle);
    else if(valid_dot == 2) {...}
    else if(valid_dot == 3) {...}
    else if(valid_dot == 4)
        DirectOval(qimage,current_color,p1,c2,-angle);
```

2.4 多边形

2.4.1 多边形的绘制

2.4.1.1 理论篇

基本思想：多边形本质上是由多段直线连接而成，因此绘制的部分应该调用直线的绘制算法来完成，那么就需要确定控制点的位置，有多少控制点就要绘制多少条直线，因此控制点采用 vector 来存放，然后调用直线绘制算法就可以绘制出多边形。

具体步骤：首先进入绘制模式后，用 vector 存下来鼠标点击的点的位置，然后直接一个大循环，对于 vector 每相邻 2 个点，当成直线的控制点，将其绘制出来即可。

算法分析：这是很自然的想法，多边形由直线构成，当然要利用已经实现的直线绘制算法，这时只需将控制点有序存储起来即可，算法的执行效率是很高的。

2.4.1.2 代码篇

就是调用直线绘制算法将所有的直线有序绘制出来。

```
void DrawWidget::Polygon(QImage *qimage, QColor pen, vector<QPoint> list)
{
    for(int i = 1; i <= list.size(); i++)    △ comparison of integers of d
    {
        ddaLine(qimage,pen,list[i-1],list[i]);    △ implicit conversion c
    }
    ddaLine(qimage,pen,list[list.size()-1],list[0]);
}
```

2.4.2 多边形的平移

2.4.2.1 理论篇

基本思想：和直线平移类似，只需对控制点进行平移变换，然后绘制出来。

具体步骤：和直线类似，通过鼠标的坐标计算出平移量，然后将控制点（多边形的控制点就是 vector 里存储的连续的点）加上这个平移量，最后再调用绘制算法绘制出完整的圆。

算法分析：和直线的平移是一样的。

2.4.2.2 代码篇

首先根据鼠标得到平移量，然后更新 vector 里的点的坐标。

```
vector<QPoint> tmp;
tmp.clear();
for(int i = 0; i <= list.size(); i++)
    tmp.push_back((list[i] + delta));
```

然后再调用多边形的绘制算法绘制出来即可。

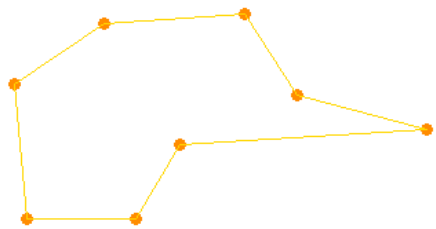
```
else if(choose == 5)
{
    for(int i = 0; i < tmp.size(); i++)      comparisc
        FlagPoint(qimage,QColor(255,140,0),tmp[i]);
    if(tmp.size() >= 2)
        Polygon(qimage,current_color,tmp);
}
```

2.4.3 多边形的编辑

2.4.3.1 理论篇

基本思想：对于多边形进行编辑，就是改变多边行的控制点的位置，利用鼠标去拖动控制点即可。

具体步骤：当鼠标点击编辑按钮的时候，为了操作的方便，多边形的顶点均已通过一个小圆来标识出来，此时用鼠标选中其中一个顶点进行拖拉即可改变多边形的形状，完成对多边形的编辑。



算法分析：我觉得这样做是很自然的想法，要实现对多边形的编辑，本质上就是改变多边形的顶点的位置，因为多边形的顶点决定了多边形的形状。最后我认为表示出多边形的顶点有利于操作的方便，提升了用户友好性。

2.4.3.2 代码篇

首先判断鼠标选中的是哪一个顶点。

```
else if(choose == 5)
{
    c1 = event->pos();
    for(int i = 0 ; i < list.size(); i++)    comparison of integers of different signs: 'int' and 's
    {
        if((c1.x()-list[i].x())*(c1.x()-list[i].x())+(c1.y()-list[i].y())*(c1.y()-list[i].y()) <= 16)
        {
            dot_index = i;
            c1 = list[i];    implicit conversion changes signedness: 'int' to 'std::vector::size_ty
            break;
        }
    }
}
```

然后更新 vector 里相应顶点的坐标，最后调用多边形绘制算法画出即可。

```
else if(choose == 5)
{
    if(dot_index == -1)
    {
        for(int i = 0; i < list.size(); i++)     compa
            FlagPoint(qimage,QColor(255,0,0),list[i]);
        if(list.size() >= 2)
            Polygon(qimage,current_color,list);
    }
    else
    {
        list.erase(list.begin()+dot_index);
        list.insert(list.begin()+dot_index,c2);
        for(int i = 0; i < list.size(); i++)     compa
            FlagPoint(qimage,QColor(255,0,0),list[i]);
        if(list.size() >= 2)
            Polygon(qimage,current_color,list);
    }
```

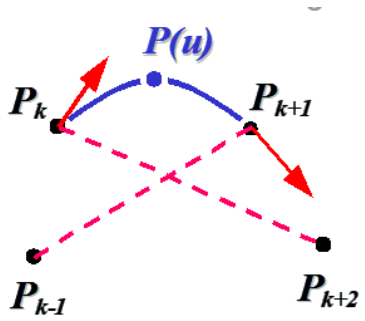
2.5 曲线

2.5.1 曲线的绘制

2.5.1.1 理论篇

Cardinal 样条曲线算法：

基本思想：Cardinal 样条是插值分段三次曲线，如下图所示，Cardinal 样条由四个连续控制点给出，中间两个控制点是曲线段端点，另二个点用来计算终点斜率。Cardinal 样条是局部控制很强的样条，其每一段曲线（比如 p(k) 和 p(k+1) 这一段曲线是由 p(k-1),p(k),p(k+1),p(k+2)这四个控制点完全控制）就可以被周围的 4 个控制点完全控制。下面就可以得到 Cardinal 曲线的边界条件（如右图所示）。其中的参数 t 称作张量，是用来的控制曲线的松紧程度。这样就可以对控制点，每 4 个控制点来绘制一条曲线，其中 p0 左边补一个控制点，就是 p0，pn 的右边补一个控制点，就是 pn。这样就有 n+2 个控制点，就可以绘制出 n+2-3 = n – 1 条曲线，就完成了曲线的绘制。



$$\mathbf{P}(0) = \mathbf{p}_k$$
$$\mathbf{P}(1) = \mathbf{p}_{k+1}$$
$$\mathbf{P}'(0) = \frac{1}{2}(1-t)(\mathbf{p}_{k+1} - \mathbf{p}_{k-1})$$
$$\mathbf{P}'(1) = \frac{1}{2}(1-t)(\mathbf{p}_{k+2} - \mathbf{p}_k)$$

具体步骤：首先最为核心的就是根据 Cardinal 样条的边界条件生成一个决策矩阵，如右图的（MC 矩阵），这个矩阵，然后可以通过参数化 p(k) 到 p(k+1) 之间的曲线，取名为 u，然后根据 p(k-1),p(k),p(k+1),p(k+2)，u 和 M 矩阵计算出曲线的坐标，这样这一段曲线的坐标就可以完全确定（当然这中间需要指定密度，也就是曲线需要采样多少个点），最后就是对于每一段曲线都使用这样的算法，就可以得到完整的曲线。

$$\mathbf{M}_C = \begin{bmatrix} -s & 2-s & s-2 & s \\ 2s & s-3 & 3-2s & -s \\ -s & 0 & s & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

算法分析：采用 Cardinal 样条曲线算法极大利用了其超强的局部控制性，不仅对于曲线的绘制有较高的效率，在对曲线进行编辑的时候，也只需做局部的调整，相对于全局控制的曲线算法，极大减少了计算量，效率上讲是很高的。

2.5.1.2 代码篇

首先将 MC 矩阵保存为一个 vector。

```
void DrawWidget::MatrixGet()
{
    m.push_back(-tension);m.push_back(2-tension);m.push_back(tension-2);m.push_back(tension);
    m.push_back(2*tension);m.push_back(tension-3);m.push_back(3-2*tension);m.push_back(-tension);
    m.push_back(-tension);m.push_back(0);m.push_back(tension);m.push_back(0);
    m.push_back(0);m.push_back(1);m.push_back(0);m.push_back(0);
}
```

然后结合参数 u 以及 p(k-1),p(k),p(k+1),p(k+2)四个控制点以及 MC 矩阵得到曲线上一个点的坐标。

```
double DrawWidget::Matrix(int p0, int p1, int p2, int p3, double u)
{
    double a, b, c, d;
    a = m[0] * p0 + m[1] * p1 + m[2] * p2 + m[3] * p3;
    b = m[4] * p0 + m[5] * p1 + m[6] * p2 + m[7] * p3;
    c = m[8] * p0 + m[9] * p1 + m[10] * p2 + m[11] * p3;
    d = m[12] * p0 + m[13] * p1 + m[14] * p2 + m[15] * p3;
    double ret = ((a*u+b)*u+c)*u+d;
    return ret;
}
```

最后套一个大的循环，对于每四个控制点都生成一段曲线，就可以得到完整的曲线了。（其中的 grain 是粒度，也就是一段曲线由几个点组成，我是根据控制点的距离生成的，这样尽量保持曲线的均匀分布）。

```
void DrawWidget::CardinalCurve(QImage *qimage, QColor pen, vector<QPoint> list)
{
    int len = list.size();                △ implicit conversion changes signedness: 'std::vector::size_type' (aka
    QPoint start = list[0];
    QPoint end = list[list.size()-1];
    list.insert(list.begin(),start);
    list.insert(list.end(),end);
    int dotex = 0, dot0 = 1, dot1 =2, dot2 = 3;
    for(int i = 1; i < len; i++)
    {
        int grain = (int)sqrt(pow(list[dot1].x() - list[dot0].x(),2) + pow(list[dot1].y() - list[dot0].y(),2));
        for(int j = 0; j < grain; j++)
        {
            double x = Matrix(list[dotex].x(),list[dot0].x(),list[dot1].x(),list[dot2].x(),(j * (1.0 / grain)));
            double y = Matrix(list[dotex].y(),list[dot0].y(),list[dot1].y(),list[dot2].y(),(j * (1.0 / grain)));
            qimage->setPixelColor((int)x,(int)y,pen);                    △ use of old-style cast
        }
        dotex++;
        dot0++;
        dot1++;
        dot2++;
    }
}
```

2.5.2　曲线的平移

2.5.2.1　理论篇

基本思想：和直线平移类似，只需对控制点进行平移变换，然后绘制出来。

具体步骤：和直线类似，通过鼠标的坐标计算出平移量，然后将控制点（曲线的控制点就是 vector 里存储的连续的点）加上这个平移量，最后再调用绘制算法绘制出完整的圆。

算法分析：和直线的平移是一样的。

2.5.2.2　代码篇

首先根据鼠标得到平移量，然后更新 vector 里的点的坐标。

```
vector<QPoint> tmp;
tmp.clear();
for(int i = 0; i < list.size(); i++)
    tmp.push_back((list[i] + delta));
```

然后再调用曲线的绘制算法绘制出来即可。

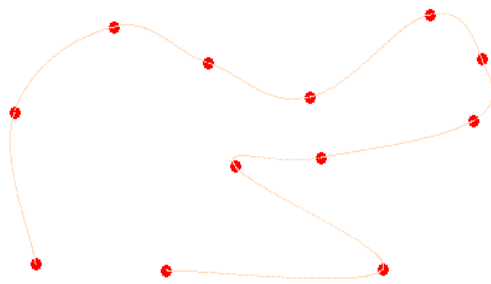
```
else if(choose == 6)
{
    for(int i = 0; i < tmp.size(); i++)      ⚠ compa
        FlagPoint(qimage,QColor(255,0,0),tmp[i]);
    if(tmp.size() >= 2)
        CardinalCurve(qimage,current_color,tmp);
}
```

2.5.3 曲线的编辑

2.5.3.1 理论篇

基本思想：对于曲线进行编辑，就是改变曲线的控制点的位置，利用鼠标去拖动控制点即可。

具体步骤：当鼠标点击编辑按钮的时候，为了操作的方便，曲线的控制均已通过一个小圆来标识出来，此时用鼠标选中其中一个控制点进行拖拉即可改曲线的形状（局部控制），完成对曲线的编辑。



算法分析：我觉得这样做是很自然的想法，要实现对曲线的编辑，本质上就是改变曲线的控制点的位置，因为曲线的顶点决定了曲线的形状。最后我认为表示出曲线的顶点有利于操作的方便，可以清晰地看出曲线的变化，提升了用户友好性。

2.5.3.2 代码篇

首先判断鼠标选中的是哪一个顶点。

```
else if(choose == 6)
{
    c1 = event->pos();
    for(int i = 0 ; i < list.size(); i++)     ⚠ comparison of integers of different signs: 'int' and '
    {
        if((c1.x()-list[i].x())*(c1.x()-list[i].x())+(c1.y()-list[i].y())*(c1.y()-list[i].y()) <= 16)
        {
            dot_index = i;
            c1 = list[i];     ⚠ implicit conversion changes signedness: 'int' to 'std::vector::size_t
            break;
        }
    }
}
```

然后更新 vector 里相应顶点的坐标，最后调用多边形绘制算法画出即可。

```
else if(choose == 6)
{
    if(dot_index == -1)
    {
        for(int i = 0; i < list.size(); i++)      ⚠ compar
            FlagPoint(qimage,QColor(255,0,0),list[i]);
        if(list.size() >= 2)
            CardinalCurve(qimage,current_color,list);
    }
    else
    {
        list.erase(list.begin()+dot_index);
        list.insert(list.begin()+dot_index,c2);
        for(int i = 0; i < list.size(); i++)      ⚠ compar
            FlagPoint(qimage,QColor(255,0,0),list[i]);
        if(list.size() >= 2)
            CardinalCurve(qimage,current_color,list);
    }
```

2.6 填充

2.6.1 填充的实现

2.6.1.1 理论篇

填充图元的扫描线边界填充算法：

基本思想：使用递归种子填充的话经常会爆栈，因为递归的层数太多，栈空间不够用，所以采用扫描线边界填充算法。其从起始点开始首先填充该行，然后对于当前行的上一行和下一行进行遍历，找出新的标志点放入栈中，然后利用栈的特性递归下去，最后栈为空的时候填充就结束了。

具体步骤：每次从栈中取出某行可填充段的第一个像素点，然后将此段区域填充，然后对于该行的上下两行进行检索，取出其可填充段的首像素点放入栈中，这样就完成了一个循环。算法的实质就是首先填充改行，然后将上下两行检索到的可填充的像素点，放入栈中，为下一次循环做准备。算法结束的标志是栈为空，递归结束的条件，是对于当前行的上一行或下一行进行检索没有可填充段，这样就说明碰到边界了。

算法分析：显然该算法的效率会比直接的边界种子递归效率高很多，因为边界种子递归算法每次都是给一个像素点上色，而扫描线边界填充每一次都是给一行像素点上色，所以在效率上至少相差一个次方量级。所以该算法更加实用。

2.6.1.2 代码篇

```
void DrawWidget::SeadStuff(QImage *qimage, QColor pen, QColor stuffcolor, QPoint p2)
{
    int x1 = p2.x();
    int y1 = p2.y();
    if(pen == stuffcolor)
        return;
    QStack<QPoint> *qstack = new QStack<QPoint>();
    int i = x1;
    for(; i >= 0; i--) {...}
    i = i + 1;
    qstack->push(QPoint(i,y1));
    while(!qstack->empty())
    {
        QPoint p = qstack->pop();
        int len = StuffLine(qimage,pen,stuffcolor,p);
        PushStack(qimage,pen,stuffcolor,p,qstack,len);
    }
}
```

2.7 3D图像

2.7.1 3D 图像的显示

2.7.1.1 理论篇：

基本思想：采用的是 OpenGL 的第三方库来做的，其绘制的机制是，每次绘制一个切面（由三个顶点组成的三角形切面），因此其实要做的就是解析 OFF 文件，将其表示成切面的数组，每次传一个切面进去绘制出来即可。

具体步骤：解析 OFF 文件，有两个 vector 来存，第一个是用来存所有用到的控制点的坐标，第二个是用来存每个切面用到的控制点的序号，就解析完毕了。然后调用 OpenGL 的绘制算法，每次传一个切面进去，然后再根据顶点序号从另一个 vector 获取顶点三维坐标即可。

算法分析：还是比较轻松的，看懂了绘制机制以及 OFF 文件的格式就基本知道了该存哪些数据，以及该传哪些参数进去。绘制出来的效果也还是可以的。

2.7.1.2 代码篇

解析 OFF 文件：

```
void PaintThreeD::Parseoff(QString filepath)
{
    QFile f(filepath);
    if(!f.open(QIODevice::ReadOnly | QIODevice::Text))
    {
        qDebug() << "Open failed." << endl;
        return;
    }
    QTextStream input(&f);
    QString lineStr;
    input >> lineStr;
    int v_num, s_num, nothing;
    input >> v_num >> s_num >> nothing;
    for(int i = 0; i < v_num; i++)
    {
        VertexValue vv;
        input >> vv.v1 >> vv.v2 >> vv.v3;
        vertex.push_back(vv);
    }
    for(int j = 0; j < s_num; j++)
    {
        int t;
        input >> t;
        VertexID vd;
        input >> vd.id1 >> vd.id2 >> vd.id3;
        slide.push_back(vd);
    }
    f.close();
}
```

绘制函数中传参数：

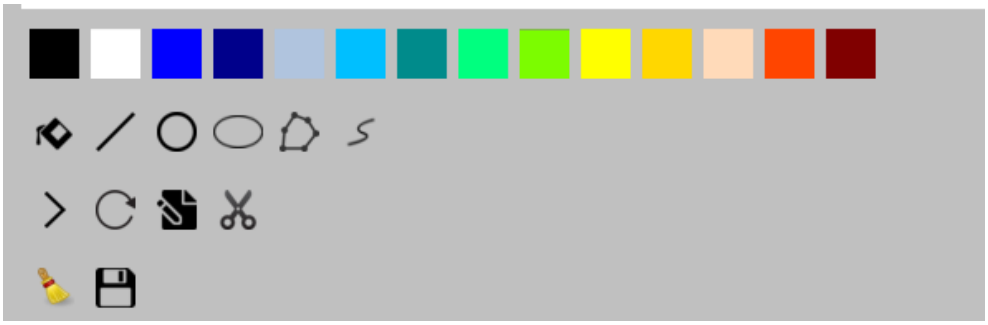
```
void PaintThreeD::paintGL()
{

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();
        glRotatef((GLfloat)angle1, 0.0, 1.0, 0.0);
        glRotatef((GLfloat)angle2, 1.0, 0.0, 0.0);
        glScalef((GLfloat)x,(GLfloat)y,(GLfloat)z);            △use of old-style cast  △use of
        glBegin(GL_TRIANGLES);
        //glColor3f((GLfloat)0.69, (GLfloat)0.77, (GLfloat)0.87);
        for(int j = 0; j < (int)slide.size(); j++)
        {
            glColor3f((GLfloat)0.2f, (GLfloat)0.2f, (GLfloat)0.2f);     △use of old-style cast  △u
            glVertex3f(vertex[slide[j].id1].v1, vertex[slide[j].id1].v2, vertex[slide[j].id1].v3);
            glColor3f((GLfloat)0.4f, (GLfloat)0.4f, (GLfloat)0.4f);     △use of old-style cast  △u
            glVertex3f(vertex[slide[j].id2].v1, vertex[slide[j].id2].v2, vertex[slide[j].id2].v3);
            glColor3f((GLfloat)0.8f, (GLfloat)0.8f, (GLfloat)0.8f);     △use of old-style cast  △u
            glVertex3f(vertex[slide[j].id3].v1, vertex[slide[j].id3].v2, vertex[slide[j].id3].v3);
        }
        glEnd();
}
```

3 系统介绍

3.1 用户界面：

分为了 2 个区，一个是 QWidget 类的画板，一个是由 QToolBar 工具栏类组成了功能选择区。

3.1.1 功能选择区界面截图：



3.1.2 功能选择区简介：

第一行是色盘，点击选取画笔和填充的颜色。

第二行是绘制图元，分别是填充、直线、圆、椭圆、多边形、曲线。

第三行是变换选项，分别是平移、旋转、编辑（包含了缩放）、裁剪。

第四行是辅助选项，清屏和保存。

3.2 代码框架设计：

文件	代码行数	功能
main.cpp	11	程序的入口，启动界面
drawwidget.h	81	画板的声明
drawwidget.cpp	1415	画板各类函数的实现
paint.h	41	主窗口的声明
paint.cpp	111	主窗口控件的生产和管理

3.2.1 Paint 类：

私有成员：

```
private:
    Ui::Paint *ui;          //主UI
    QToolBar *toolbar1;     //图元工具栏
    QToolBar *toolbar2;     //辅助功能工具栏
    QToolBar *toolbar3;     //色盘工具栏
    QToolBar *toolbar4;     //图元变换工具栏
    DrawWidget *drawwidget1;    //画板类
```

构造函数：

```
    QVBoxLayout *vboxlayout1 = new QVBoxLayout(this);
    vboxlayout1->addWidget(drawwidget1);
    vboxlayout1->addWidget(toolbar3);
    vboxlayout1->addWidget(toolbar1);
    vboxlayout1->addWidget(toolbar4);
    vboxlayout1->addWidget(toolbar2);
    this->setLayout(vboxlayout1);

    connect(toolbar1_actiongroup1,SIGNAL(triggered(QAction *)),this->drawwidget1,SLOT(draw(QAction *)));
    connect(toolbar2_actiongroup1,SIGNAL(triggered(QAction *)),this->drawwidget1,SLOT(draw(QAction *)));
    connect(this->drawwidget1,SIGNAL(draw_shape(int)),this->drawwidget1,SLOT(setChoose(int)));
    connect(toolbar3_actiongroup1,SIGNAL(triggered(QAction *)),this->drawwidget1,SLOT(pick(QAction *)));
    connect(this->drawwidget1,SIGNAL(pick_color(QColor)),this->drawwidget1,SLOT(setColor(QColor)));
    connect(toolbar4_actiongroup1,SIGNAL(triggered(QAction *)),this->drawwidget1,SLOT(transform(QAction *)));
    connect(this->drawwidget1,SIGNAL(transform_shape(int)),this->drawwidget1,SLOT(setMode(int)));
```

主要的功能是进行了垂直布局，以及用 connect 函数建立了各子控件信号和槽的联系。

3.2.2 DrawWidget 类：

私有成员：

```
private:
    QPoint p1, p2;  //绘图时鼠标的起点和终点
    QPoint a1, a2;  //平移时鼠标的起点和终点
    QPoint b1, b2;  //旋转时鼠标的起点和终点
    QPoint c1, c2;  //编辑时鼠标的起点和终点
    QPoint d1, d2;  //裁剪时鼠标的起点和终点
    int mode;   //当前模式的标志位 -1无模式 1绘画  2平移  3旋转  4编辑  5裁剪
    int choose; //当前图元的标志位 -1清屏 0保存图片 1填充 2直线 3圆 4椭圆 5多边形 6曲线
    int mouse_state; //当前鼠标状态的标志位 -1无状态 1点击  2拖动  3释放
    QColor current_color;  //当前画笔颜色
    QImage *qimage;  //当前的画布
    QImage *tmp_qimage;  //为实现动态作画而创建的画布
    QImage *editframe;  //画包围盒的画布
    QImage *cutframe;  //画裁剪窗口
    int valid_dot;  //编辑时选中的点的标志位 -1没选中 1选中p1  2选中p2  3选中p3  4选中p4
    double angle; //旋转时的角度 0~2Pi
    vector<QPoint> list; //存储每一次画曲线的控制点
    vector<double> m; //曲线变换矩阵
    double tension; //张量
    bool finish; //绘制曲线的标志
    int dot_index; //选中的曲线的控制点的标志位
```

公有函数：

```
public:
    void paintEvent(QPaintEvent *event);  //绘图事件
    void mousePressEvent(QMouseEvent *event);  //鼠标点击事件
    void mouseMoveEvent(QMouseEvent *event);  //鼠标拖动事件
    void mouseReleaseEvent(QMouseEvent *event);  //鼠标释放事件
    double coordinate_x(double x, double y, int a, int b, double angle);  //计算旋转后的坐标
    double coordinate_y(double x, double y, int a, int b, double angle);
    void ddaLine(QImage *qimage, QColor pen, QPoint p1, QPoint p2);  //画直线
    void BresenhamCircle(QImage *qimage, QColor pen, QPoint p1, QPoint p2);  //画圆
    void BresenhamOval(QImage *qimage, QColor pen, QPoint p1, QPoint p2);  //画椭圆
    void DirectOval(QImage *qimage, QColor pen, QPoint p1, QPoint p2, double angle);  //画斜椭圆
    void Rectangle(QImage *qimage, QColor pen, QPoint p1, QPoint p2);  //画矩形
    void Polygon(QImage *qimage, QColor pen, vector<QPoint> plist); //画多边形
    void SeadStuff(QImage *qimage, QColor pen, QColor stuffcolor, QPoint p2);  //填充
    int StuffLine(QImage *qimage, QColor pen, QColor stuffcolor, QPoint p);
    void PushStack(QImage *qimage, QColor pen, QColor stuffcolor, QPoint p, QStack<QPoint> *qstack, int len)
    void Translation(QImage *qimage, QPoint a1, QPoint a2);  //平移
    void Rotation(QImage *qimage, QPoint b1, QPoint b2);  //旋转
    void Editing(QImage *qimage, QPoint c1, QPoint c2);  //编辑
    void EditFrame(QImage *qimage, QPoint p1, QPoint p2);  //画包围盒
    void SaveImage();  //保存图片
    void LiangBarskyLineCut(QImage *qimage, QPoint d1, QPoint d2); //线段的裁剪
    void FlagPoint(QImage *qimage, QColor pen, QPoint p);  //标记曲线的控制点
    void CardinalCurve(QImage *qimage, QColor pen, vector<QPoint> list); //画曲线
    void MatrixGet(); //得到参数矩阵
    double Matrix(int p0, int p1, int p2, int p3, double u);  //计算坐标
```

这个类里面实现了所有要求的算法。各个模式的切换、各种图元的切换、各种功能选项的切换都是通过改变全局变量标志位的值来实现的。

另外还有一些用于与主界面中控件交互的信号量和槽函数;

```
signals:
    void draw_shape(int choose);
    void pick_color(QColor color);
    void transform_shape(int mode);

public slots:
    void draw(QAction *q);
    void setChoose(int choose);
    void pick(QAction *q);
    void setColor(QColor color);
    void transform(QAction *q);
    void setMode(int mode);
```

4 结束语

感觉整个图形学大作业上手较难，主要是 Qt 是首次使用，在入门之后，框架搭起来之后就好写多了。主要还是要上课认真听讲，将算法合理实现到代码中去。真的学到了很多东西，不仅是会用 Qt 写图像界面，而且对于图形学的理解也是更深一层。

致谢 在此，我向帮助过我的同学、老师、助教们表示诚挚的感谢，谢谢你们的耐心指导和热情帮助。以及向网络上写技术博客的博主们表示感谢，谢谢你们提供的很多问题的分析。

附中文参考文献:

[1] https://blog.csdn.net/K54387/article/details/77926313/

[2] http://www.cnblogs.com/wangbin-heng/p/9484272.html

[3] https://blog.csdn.net/rl529014/article/details/51589096

[4] https://www.cnblogs.com/tornadomeet/archive/2012/08/22/2651574.html

[5] https://www.cnblogs.com/tornadomeet/archive/2012/08/24/2654327.html

[6] http://www.cnblogs.com/sparkmorry/p/3535936.html

[7] https://blog.csdn.net/iw1210/article/details/51253458