《计算机图形学》系统技术报告

作者：戚赟 学号：161220097

(南京大学 计算机科学与技术系, 南京 210093)

摘 要:经过三个月的学习，我终于基本完善了我的图形学大作业，实现了基本要求的功能和一些附加的优化操作，使我的图形学大作业基本能够像一个比较好的绘图软件。其中我实现了直线、圆、椭圆、曲线(Bezier曲线）、多边形、填充区域的输入和编辑功能，还能够对于直线、圆、椭圆、曲线、多边形进行变换操作，还实现了对于图片的保存功能，包括平移、旋转、缩放。此外还有3D图形的显示，其中还包括了对于3D图形的视角转换的操作，从而可以从多个角度来进行对于3D图形的观察。

关键词: 算法原理，性能测试。

一、系统的环境配置

编程语言	系操作统	GUI框架	3D图形库
C++	WINDOWS 10	QT 5.11.2	OpenGL

在考虑如何来写图形学这个比较宏大的项目的之前，我和他人进行了沟通，终于最后决定使用QT来进行图形界面的编写，因为QT是也是C++的一个集成环境，和VS相差无几，同时开发的界面比较美观，也可以很好的使用OPENGL工具来显示3D模型。

二、整体的设计思路

1.主窗体的设计(MainWindow.cpp)

显示的界面主要是利用了QT提供的应用框架来做，这样的好处是可以利用QT自带的框架，利用它定义的一些可用的工具，来进行界面的设计，从而使得图形学的程序能够更加美观，实现也更加容易。

2.画板的设计(mywidget.cpp)

继承了QT自带的窗口类QWidget，实现了一个窗口，内含一个Pixmap，从而能够通过C++的聚合原则实现了一个画布，能够在上面进行绘画，实现了封装，保证了整体性，此外利用聚合原则能够为画布增添新的特征，以便更好的绘制。

3.对于图形的设计(Figure.cpp)

因为考虑到后来对于每一个2D图形，都有输入、编辑和变换三个主要的功能，于是我为了代码的简单和易维护构造了一个抽象类Figure，图形继承了Figure从而能利用C++的动态绑定机制实现了不同的图形的基本统一的结构，从而实现了多态性，更好的利用了复用。

对于每个图形，都有开始点，结束点和中心点，而且对于每个图形在第一次画完之后，便会进入编辑模式，因为我们是要要求是可编辑的点，所以有三个vector来进行对于可编辑的点的存储，所以我们在编辑选取点的时候，根据是否在这三个vector点集里面来判断是否有平移，旋转和编辑操作。并且根据操作来进行响应。

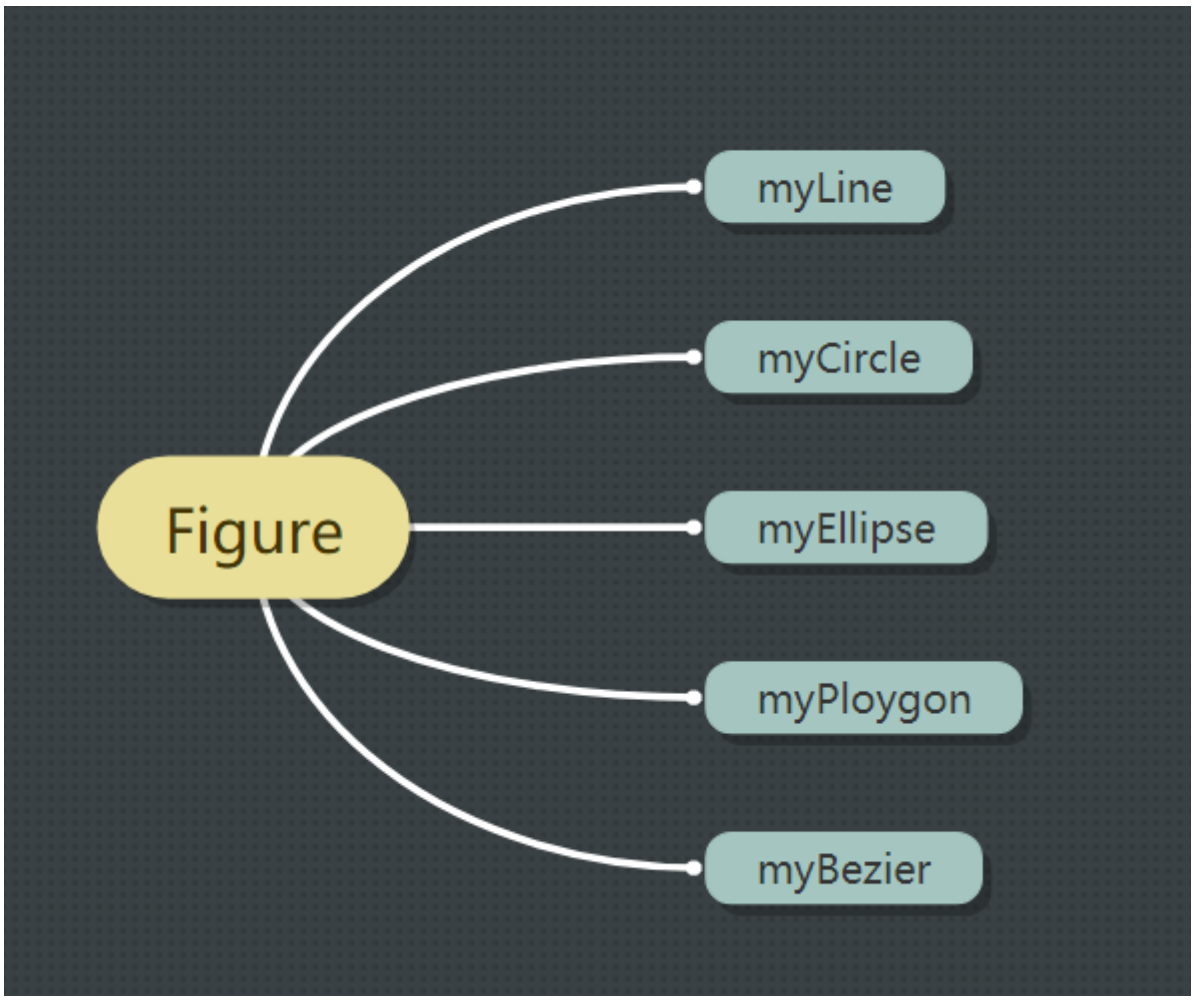
具体流程如下：

①一开始拖动绘制，到鼠标松开，保存最终的点，算出旋转，平移和编辑点并显示

②点击操作点，拖动进行响应的操作，松开继续执行②，否则进入③

③完成绘制

具体的类的继承关系，如下：



4.图片的绘制思路

图片的绘制：**双缓冲绘图。**先将显示图形放在缓冲区，再一起显示到屏幕上，以防止屏幕上的东西一个个出现或是发生闪烁。双缓冲是基于“空间换时间”和“功能分块”的思想实现，使得最后留在画板上的是你想要的东西，并且使绘图过程清晰。

三、基础算法、实现过程以及测试

图元生成类：

1.直线的绘制算法——DDA算法

算法原理：

DDA算法先通过鼠标相应得到初始点坐标和终点的坐标，然后取方向和方向的绝对距离，比较大小来判断以哪一个轴为增量1，这一步就相当于求直线的斜率是否大于1。然后对于某个方向的变化，和同时加上增量的变化，然后调用对于这个点进行绘制。

算法步骤即代码思路：

①获取起始坐标begin,end。

②计算x方向上和y方向上的绝对距离dx,dy，并且设置e为dx,dy中绝对值的较大者，分别求出方向上的增量e/dx,e/dy。

③从原点开始，每一步加上增量，并且绘制点

代码性能测试：

该算法只需根据线段的两端点， 计算线段上的中间点的坐标， 采用增量决策参数计算， 计算较快， 内存使用较小， 所有总的来说性能比较好。并且使用了 QPoint tempPoint ((int) (temp_x+0.5), (int) (temp_y)+0.5) 转型，使得直线的绘制图案最后能够变得更加细致一点，更加好看。

2.圆的绘制算法——中心圆生成算法

算法原理：

利用中心圆算法进行绘制，鼠标初始点为圆心，末尾点为圆上一点，则两个点之间的距离为半径，则利用算法的增量，对于1/8的圆上的点进行计算选择，P初始为3-2R，如果$p>=0$，$p+=4*(x-y)+10$，选择下面的y像素点，即y--，否则 $p+=4x+6$,保持不变。再分别计算其他7个圆弧上的点。于是可以进行圆的绘制。

算法步骤即代码思路：

①获取起始点和终点坐标，其中起始点为圆心，终点为圆上任意一点。

②计算出半径R

③设置参量$P=3-2R$;求在第一象限的斜率大于1的圆弧，将圆心移动到原点，从(0,R)开始，$x++$，根据p计算出画图的位置，并绘制它的所有对称点，并且根据(x,y)来更新p来为下一次找点做铺垫。

算法伪代码：

```
x=0;

y=(int)R;

p=(int)(3-2*R);

for(;x<=y;x++){

draw_point_of_circle(painter,x_begin,y_begin,x,y);  //设置八个点

    if(p>=0){

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

```
    }else{
    p+=4*x+6;
    }
}
```

代码性能测试：

该算法采用了 Bresenham 算法中的决策参数的思想，拜托了每次计算 2 次方的计算束缚，所有计算效率较高，同样该算法的输入也是决定圆的顶点坐标，对内存要求不高，在实际运行时基本没有阻塞，即计算速度已经很快了。总的来说性能很好。

3.椭圆的绘制算法——Bresenham 椭圆生成算法

算法原理： 基本思想大致与中点圆生成算法相同。

(1）决策参数:定义的椭圆函数为： f(x,y)=ry^{2x}2+rx^{2y}2-rx^{2ry}2 决策参数的初始值和增量设计亦不同。

f(x,y)<0,(x,y)位于椭圆内,f(x,y)=0,(x,y)位于椭圆上,f(x,y)>0,(x,y)位于椭圆外。

每个取样位置按椭圆函数在椭圆轨迹两候选像素间中点求值的符号选择像素。

（2）不是分成八分圆来计算和生成，而是分成 4 个象限（4 个分圆）来生成，考虑第一象限的椭圆生 成，根据椭圆的切线斜率的绝对值是否大于 1 将椭圆区域分成两部分：斜率绝对值小于 1 的内沿 x 方向离散取样，斜率绝对值大于 1 的内沿 y 方向离散取样。

（3）利用平移：先确定中心在原点的便准位置的椭圆点（x,y）；然后将点变换为圆心在（Xc,Yc）的点。

（4）对称性：生成第一象限内的椭圆弧，再利用对称性求出其他三个象限的对应点

（5）分为两个部分来进行计算。

A．斜率绝对值小于 1 x 每步进一个单位，就需要在判断 y 保持不变还是也步进减 1， bresenham 算法定义判别式为：$D = d1-d$。 如果 $D < 0$，则取 $P1$ 为下一个点，否则，取 $P2$为下一个点。采用判别式 D，避免了中点算法因$y - 0.5$ 而引入的浮点运算，使得判别式规约为全整数运算，算法效率得到了很大的提升。根据椭圆方 程，可以计算出 d1 和 d2 分别是： $d1 = a2(yi2-y2)$，$d2 = a2(y2-yi+12)$ 以(0, b)作为椭圆上部区域的起点，将其代入判别式 D 可以得到如下递推关系： $D_{i+1} = D_i + 2b^2(2x_i + 3)(D_i < 0)$ **B． 斜率绝对值大于 1**

$$D_{i+1} = D_i + 2b^2(2x_i + 3)-4a^2(y_i - 1)(D_i >= 0)$$

$$D_0 = 2b^2-2a^2b + a^2$$

在生成椭圆下部区域时，以 y 轴为步进方向，如图（5-b）所示， y 每步进减一个单位，就需要在 判断 x 保持不变还是步进加一个单位，对于下部区域，计算出 d1 和 d2分别是：$d1 = b2(xi + 12-x2)$，$d2 = b2(x2-xi2)$ 以(xp, yp)作为椭圆下部区域的起点，将其代入判别式 D 可以得到如下递推关系：

$D_{i+1} = D_i - 2a^2y - a^2 + 2b^2x + 2b^2; (D_i < 0)$

$D_{i+1} = D_i - 2a^2y - a^2; (D_i >= 0)$

$D_0 = b^2(x^2 + x) + a^2(y^2 - y) - a^2b^2;$

算法关键代码：

```
int P_x = int((double)sqa / sqrt((double)(sqa + sqb)));
    while (x <= P_x){
        if (d < 0){
```

```
            d += 2 * sqb * (2 * x + 3);
        }else{
            d += 2 * sqb * (2 * x + 3) - 4 * sqa * (y - 1);
            if(y>0) y--;
        }
        x++;
        draw_point_of_ellipse(painter,x_centre,y_centre,x,y);//画四个象限的点
    }
    d = sqb * (x * x + x) + sqa * (y * y - y) - sqa * sqb;
    while (y >= 0){
    draw_point_of_ellipse(painter,x_centre,y_centre,x,y);//画四个象限的点
    y--;
    if (d < 0){
        x++;
        d = d - 2 * sqa * y - sqa + 2 * sqb * x + 2 * sqb;
    }else{
        d = d - 2 * sqa * y - sqa;
   }
```

代码性能测试：由于书上和PPT代码有误，自己尝试了许多，发现有问题，最后实现了两种方法，一种是修改了精度问题，一种是修改了一些计算方式，给出的是第二种，但同时属于中点椭圆生成算法，具体看源文件。性能测试不错，在精度修改之后看起来挺好。

4.多边形绘制算法——利用直线

算法原理：多边形的绘制算法很简单，利用直线的绘制算法，能够轻松将多边形画出来，对于之前存好的点来说，能够是将存的点，两两之间按照顺序利用直线连接起来的话，那么多边形就绘制完成了。

算法伪代码：

```
for(int i = 0; i < this->set_of_point.size() - 1; i++)
{
       draw_line(painter,this->set_of_point[i],this->set_of_point[i+1]); //两两之间画直线
}
```

代码性能测试：由于处理器的快速，多边形的性能还是很好的，不会很慢

5.曲线的绘制——贝塞尔曲线

算法原理：

1） Bezier 曲线是通过一组多边折现的各顶点唯一定义出来的。曲线的形状趋向于多边折线的形状，改变多边折线的顶点坐标位置和改变曲线的形状有紧密的联系。 多边折线常称为特征多边形，其顶点称为控制顶点。 n 次 Bernstein 基函数的多项式：

$$BEZ_{I,N}(u) = C(n,i)u_i(1-u)_{n-i}, C(n,i) = \frac{n!}{i!(n-i)!}, i = 0,1,\ldots,n$$

2） Bezier 曲线绘制： 假设给出 n+1 个控制顶点位置： $P_i = (x_i, y_i)(i = 0,1,2,\ldots,n)$。这些坐标点混合产生下列位置向量用来描述 P0 和 Pn 间的逼近 Bezier 多项式函数的路径。德卡斯特里奥递推算法直接利用控制多边形顶点从参数 u 计算 n 次 Bezier 曲线型值点的过程，其计算公式为：

$$P_i^r = \begin{cases} P_i \\ (1-u)P_i^{r-1} + uP_{i+1}^{r-1} \end{cases} (i = 0,1,2,\cdots,n-r),(r = 1,2,\cdots n)$$

算法伪代码： u 取某个 0 到 1 之间的值时： p 为控制点数组， q 为 Bezier 曲线上的点

```
Input: array P[0:n] of n+1 points and real number u in [0,1]
Output: point oncurve,
C(u) Working:
point array Q[0:n]for i := 0 to n do
Q[i] := P[i]; // save input
for k := 1 to n do
for i := 0 to n - k do
Q[i] := (1 - u)Q[i] + u Q[i + 1];
return Q[0];
```

4)代码的具体实现见附上的源文件$Bezier.cpp$。

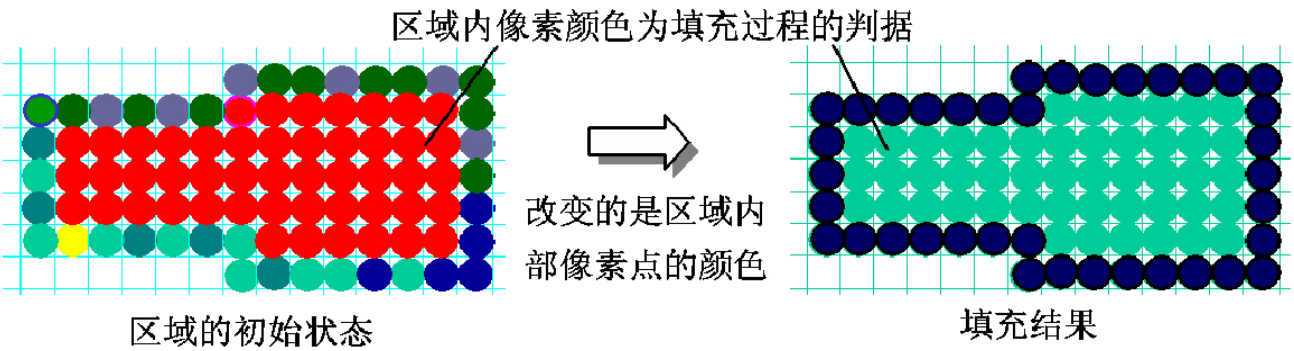
算法性能测试：

在画到10几个点的时候，本来会觉得性能下降的会比较快速，但是性能并没有下降到掉帧，还是给人感觉就是直接画出来的，所以代码的性能还是比较好的。

6.填充算法

算法原理：

填充算法利用了泛滥填充算法：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。实现的是**连通度为四**的填充算法。



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)。先将PIXMAP转化为QImage类，从而可以获取每一个像素的点的颜色，定义一个栈，将起始点入栈，当栈不为空的时候，pop出一个点，将其没有进入过栈的点pop进去，然后进行边界判断，最后如果是连通的，就将其颜色改为相同的颜色，最后再转化回来从而达到目的 。

算法步骤：

①选取的点作为种子点，压进入栈中，这个点被记为已读

②从栈中pop出一个点，若碰到边界，则停止，若不是已读过的点，就压入栈中，重新执行②；否则如果连通，将颜色改为与最终颜色，并且将这些点压入栈中，重新执行②。

③栈为空的时候，停止算法。

算法性能测试：

通过对代码的改进，改变了填充速度，使得我的填充速度变的比较快速，利用了栈代替了原来的递归实现了递归操作，从而大大降低了速度，使得填充速度变得可以接受。

图元裁剪类：

①直线的裁剪算法——梁友栋裁剪算法

算法原理：

假设点P1P2W1W2的横坐标分别是x1,x2,w1,w2，线段P1P2与蓝色裁剪窗口W1W2（蓝色的线之间）的存在公共部分（可见部分）的充要条件是：

$max(min(x1,x2),min(w1,w2)) \leq min(max(x1,x2),max(w1,w2))$

根据一维裁剪窗口与线段的左右两端点的坐标值大小，就能确定线段和裁剪窗口的可见部分。

设Δx=x2-x1，Δy=y2-y1，参数u在0～1之间取值,任何线段上的点P(x,y)都可以满足下面的方程：

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

当点P位于由$(xmin, ymin)$和$(xmax, ymax)$所确定的裁剪窗口内时满足下面不等式:

$x_{min} \leq x \leq x_{max}$，转化得到四个不等式
$y_{min} \leq y \leq y_{max}$

$$
\begin{aligned}
&u \times r_k \leq q_k, k = 1,2,3,4 \\
&r_1 = -\Delta x, q_1 = x_1 - x_{min}, u_1 = \frac{q_1}{r_1} \\
&r_2 = \Delta x, q_2 = x_{max} - x_1, u_2 = \frac{q_2}{r_2} \\
&r_3 = -\Delta y, q_3 = y_1 - y_{min}, u_3 = \frac{q_3}{r_3} \\
&r_4 = \Delta y, q_4 = y_{max} - y_1, u_4 = \frac{q_4}{r_4}
\end{aligned}
$$

所以根据坐标可以求出u_1, u_2, u_3, u_4。下面讨论线段在前面图中的PAPB与裁剪窗口的关系，显然r1 < 0 , r2 > 0 , r3 < 0 , r4 > 0,若左端点大者$(max(xmin, xL))$小于右端点小者$(min(xmax, xR))$时，可以看出L1与裁剪窗口一定有公共部分，反之，PAPB在裁剪窗口之外，但是通过u_k只能求出直线L1与裁剪窗口可见部分，根据梁友栋-Barsky裁剪算法$r_k < 0$求出的u_k最大值是左端点的u值，按照$r_k > 0$求出的u_k的最小值是右端点的u值，当线段与裁剪窗口有可见部分时u在范围[0，1]。

算法伪代码：

```
for(int i = 0; i < 4; i ++){
        double r = q[i] / p[i];
        if(p[i] < 0){
            u_1 = u_1 > r ? u_1 : r;
            if(u_1 > u_2){
                flag = true;
            }
```

```
        }else if(p[i] > 0) {
            u_2 = u_2 < r ? u_2 : r;
            if(u_1 > u_2){
                flag = true;
            }
        }else if(p[i] == 0 && q[i] < 0){
            flag = true; //
        }
    }
```

算法性能测试：

裁剪下来能够继续编辑，而且因为代码量不多，所以算法性能很快。

图元编辑类：

1.图形的编辑

算法原理：

图形的编辑点其实很简单，因为确定图形的点都被存放了起来，所以只要选取了编辑的点，那么就进入编辑模式，而且对于选取的编辑的点，保持不动，然后将该点的位置改变成当前的位置，并且调用函数重新进绘制一次图，就实现了编辑。

代码：

```
    this->set_of_point[num] = end_pos;
    this->draw_(painter,this->point_begin,this->point_end);//画画
    return true;
```

算法性能测试：

由于是重新绘图，那么算法的性能和画图相差无几，性能很好。

图元变换类：

①图元的平移：

算法原理：

图形的平移算法简单易懂，我们每个图形都会被设置一个点为图形的平移点，则按住这个点进行移动的时候，会算出之前的中心点和鼠标的最后位置的点的变化dx,dy，然后对于每一个图形的控制点，加上或者减去这个点，在执行一次绘画函数，就能够实现图形的平移。

代码：

```
    for(int i = 0; i < this->set_of_point.size();i++)
    {
        set_of_point[i].rx() += change_rx;
        set_of_point[i].ry() += change_ry;
    }
    this->draw_(painter,this->point_begin,this->point_end);
```

算法性能测试：

与图元的平移一样，算法的性能很好，宛如第一次绘图。

②图元的旋转：

2.1 直线的旋转：

算法原理：

旋转必须知道直线的旋转角，根据当前的旋转点，中心点和最后的鼠标移动到的**end_pos**，我们利用三角形的余弦定理算出旋转角A的**cosa**值，在根据正负旋转，确定**sina**的正负号，最后根据坐标变换公式。

坐标旋转变换的公式为： 设对图片上任意点(x,y)，绕一个坐标点(rx0,ry0)逆时针旋转a角度后的新的坐标设为(x0, y0)，有公式：

$$x0 = (x - rx0) * cos(a) - (y - ry0) * sin(a) + rx0;$$
$$y0 = (x - rx0) * sin(a) + (y - ry0) * cos(a) + ry0;$$

所以利用算出旋转角度后， 算出起点和终点围绕中点坐标改变后的值，再调用**draw_**函数进行绘制，就能够完成图片的旋转。

算法代码简介：

角度的计算需要一些技巧，下面也要用到。所以我将这份**完整贴出**：

旋转角的获取：

```
    //获取旋转角
    QPoint temp_center = this->point_center;
    QPoint temp_begin = this->point_of_rotate[0]; //开始旋转的点
    //根据三角型公式来求a的值
    double AB2 = get_2_distance(temp_center,temp_begin);
    double AC2 = get_2_distance(temp_center,end_pos);
    double BC2 = get_2_distance(temp_begin,end_pos); //
    double sina,cosa; //a的值为0到pi
    cosa = (AB2 + AC2 - BC2)/(2 * sqrt(AB2*AC2)); //得到cosa的值
    sina = sqrt(1 - cosa * cosa); //得到sina的值
    //判断是否是正旋转还是负旋转
    if(temp_begin.rx() == temp_center.rx()) //可以知道两个点是一条直线上的，则我们可以知道斜率
不存在
    {
        if(temp_begin.ry() > temp_center.ry()){
            if(end_pos.rx() < temp_begin.rx())
                sina = -sina; //为负
        }else{
```

```
            if(end_pos.rx() > temp_begin.rx())
                sina = -sina; //为负
        }
    }
    else
    {
        double k = double(temp_begin.ry()-temp_center.ry())/(temp_begin.rx()-
temp_center.rx());//算出斜率
        if(end_pos.ry()- k * (end_pos.rx() - temp_center.rx()) - temp_center.ry() > 0){
            if(temp_begin.rx() < temp_center.rx())
                sina = - sina;
        }
        else if(end_pos.ry()- k * (end_pos.rx() - temp_center.rx()) + temp_center.ry()
 > 0) {
            if(temp_begin.rx() > temp_center.rx())
                sina = - sina;
        }
    }
```

性能测试：

计算旋转角后，算出坐标，然后重新绘制，算法算的很快速，因此不会比较缓慢，性能良好。

2.2 多边形，曲线的旋转

算法原理：

多边形和曲线的旋转是类似的，他们共同享有一个变量$rotate_angle$来对于当前的旋转角度进行描述，为什么要加这个角度变量的原因是，多边形不同的是，它的中心点和旋转点是由多变形的外接矩形来进行改变的，如果在一次变化中就改变的话，那么中心点的位置就不会在一个地方旋转，不像旋转。

改进方法：设置变量是否正在旋转is_rotating，利用变量来实现对于点的更新操作。

1. 旋转操作的时候：is_is_rotating = false;
2. 计算角度，利用2.1直线算出sina和cosa;
3. 利用sina和cos更新出了中心坐标之外的点的所有顶点，并且重新绘图
4. 停止绘图
5. 执行其他操作，更新中心点和旋转点

这样就保持了旋转的是按一个中心进行旋转的。

算法代码简介：

和2.1中代码类似，主要是旋转角的计算。

性能测试：

测试性能良好，计算速度快速。

2.3椭圆的旋转

算法原理：

椭圆的旋转是最繁琐的，因为我们的绘图是中点椭圆生成算法，椭圆的绘制只有水平的，而我实现的是任意角度的旋转操作，所以我定义一个变量$rotate_angle$，这次所有的固定点的坐标不会更改，每次绘图时，通过$rotate_angle$和一开始的控制点，来算出应该在的点，进行绘制，所以这时候，点的更新是一个问题。必须重新操作，设置一个is_rotating变量

算法过程：

1. is_rotating = true
2. 计算出旋转角，然后rotate_angle加上或减去旋转角，其中，所有点都不变化。
3. 绘图，根据rotate_angle和起始坐标算出旋转后的坐标，然后进行绘制。
4. is_rotating = false

算法代码简介：

和2.1中代码类似，主要是旋转角的计算。

性能测试：

在测试椭圆旋转的时候遇到了一些问题，主要是如果太细的话，点的个数是确定的，但是旋转之后，精度损失，点之间的会有空隙，造成填充算法的失败。如果两两之间用直线之间相连，得出的图案太丑了，最后的解决办法是，将线画粗一点，比较简陋的解决了这个问题。

测试性能良好，计算速度快速。

③图元的缩放：

算法原理：

对于图元的缩放功能，我实现的是利用的相对于原点的缩放功能。

（1）缩小，所有控制点进行将(x,y)坐标都变为原来的$\frac{1}{2}$倍来进行操作，进行重新绘制图片

（2）放大，所有控制点进行将(x,y)坐标都变为原来的2倍来进行操作，进行重新绘制图片

算法的代码：

```
void myWidget::zoom_in()
{
    if(this->is_editing == true && this->cur_figure != nullptr)
    {
        this->cur_figure->zoom_in_point();
        *temp_draw_area = *cur_draw_area;
        my_paint(temp_draw_area);
    }
}

void myWidget::zoom_out()
{
    if(this->is_editing == true && this->cur_figure != nullptr)
    {
        this->cur_figure->zoom_out_point();
        *temp_draw_area = *cur_draw_area;
        my_paint(temp_draw_area);
    }
```

```
}
```

算法的性能测试：

由于只是将坐标计算进行重新绘制，算法的性能还是挺优越的。

3D图形类：

算法原理：

（1）绘制：实现了类，继承了$QOpenGLWidget()$类，从而实现了一个3D画板，3D的图形的绘制是点线面之间的关系的集合，我们利用OPENGL库来进行绘制，首先读取off文件，并且将点，线和面存了起来，然后利用OpenGL自带的库存，进行了绘制。从而能够显示3D图形。

下面是具体代码实现的点线面的绘制。

```
void my_3DWidget::drawPoints(){
    glColor3f(0.0, 1.0, 0.0);
    glPointSize(2);
    glBegin(GL_POINTS);
    for(int i = 0 ;i < numbers_of_Vertices; i++)
        glVertex3f(vecs[i].x(),vecs[i].y(),vecs[i].z());
    glEnd();
}
void my_3DWidget::drawLines()
{
    for(int i=0;i<faces.size();i++){
        QVector<int> face = faces[i];
        glColor3f(0.0, 0, 1.0); //青色
        glBegin(GL_LINES); //开始划线
        for(int j = 2 ; j < face.length();j++){
            glVertex3f(vecs[face[j-1]].x(), vecs[face[j-1]].y(), vecs[face[j-1]].z());
            glVertex3f(vecs[face[j]].x(), vecs[face[j]].y(), vecs[face[j]].z());
        }
        glVertex3f(vecs[face[face.length()-1]].x(),
vecs[face[face.length()-1]].y(),vecs[face[face.length()-1]].z());
        glVertex3f(vecs[face[1]].x(), vecs[face[1]].y(), vecs[face[1]].z());
        glEnd();
    }
}
void my_3DWidget::drawFaces()
{
    for(int i = 0 ;i < faces.size(); i++){
        QVector<int> face = faces[i];
        glColor3f(1.0, 1.0, 1.0); //白色
        if(face[0] == 3){
            glBegin(GL_TRIANGLES);
        }else if(face[0] == 4){
           glBegin(GL_TRIANGLES);
        }else{
               glBegin(GL_POLYGON);
        }
```

```
        for(int j = 1 ; j< face.length();j++){
            glVertex3f(vecs[face[j]].x(), vecs[face[j]].y(), vecs[face[j]].z());
        }
        glEnd();
    }
}
```

（2）操作：

算法原理：

定义了数据结构

```
    float angle; //旋转角度
    float offsetX;  //z轴观察距离
    float offsetY;  //z轴观察距离
    float offsetZ;  //z轴观察距离
```

从而实现了旋转等操作。通过捕捉键盘事件，来进行相应的改变X,Y,Z和让3D图形按某个方式进行绕点旋转。

```
void my_3DWidget::keyPressEvent(QKeyEvent *event){
    switch(event->key()){
        case Qt::Key_Left:  angle+=5;break;
        case Qt::Key_Right: angle-=5;break;
        case Qt::Key_Up:    offsetZ+=0.5;break;
        case Qt::Key_Down:  offsetZ-=0.5;break;
        case Qt::Key_A:     offsetX+=0.5;break;
        case Qt::Key_D:     offsetX-=0.5;break;
        case Qt::Key_W:     offsetY+=0.5;break;
        case Qt::Key_S:     offsetY-=0.5;break;
    default:break;
    }
    update();
}
void my_3DWidget::paintGL(){
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW); //模型视图
    glLoadIdentity();   //恢复初始坐标系
    glTranslatef(offsetX, offsetY, offsetZ); //更改坐标系
    glRotatef(angle, 0,0.5,0); //按角度在旋转
    gluLookAt(1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
    this->draw3D(); //绘制3D图形
}
```

四、遇到的一些问题和解决及感悟

在这个比较庞大的项目中，我在从一开始的使用QT的探索，到后来重新架构整个项目的3个月之间，经历了比较痛苦的过程，但又是十分开心的过程，因为我收获了很多，下面我说明我的遇到的问题，解决方案和感悟。

①QT框架的使用：一开始对于QT的框架和它的构建都十分的陌生，但是经过后来的使用，我能够通过问同学，阅读技术博客和官方文档示例，慢慢的一步步掌握一些组件的用法，这些细细小小的组件最后汇聚成一个大的项目，我从而明白了学习框架的好处和弊端，并更加知道如何学习一个陌生的知识。

②整体架构：在我最初的实验报告中，当时的直线，圆和椭圆还是只能够进行简单的绘制，于是只用了函数来写，但是后期操作比较繁琐，于是我就进行了重新的架构，利用了C++的多态机制，我通过设计了一个抽象类Figure，来实现对于图形的整体操控，从而能够更好的实现了，也使代码的耦合度低，让我懂得了前期的设计是很重要的，做好前期的设计工作，就能事半功倍的完成这个设计。

③对于一些算法的理解和绘制：有些时候，一些原理很难搞懂，而且计算十分繁琐，但是在坚持不懈的努力之后，我终于能够正常的理解算法，并且进行绘制，感谢技术博客以及同学对于我的帮助，让我受益匪浅。

④对于旋转的问题：旋转的编写部分是我遇到的调过最多的BUG部分，这主要依靠了设计，也要区分出每个子类图元的不同点，从而能够正确实现，从小功能完善，到最后的整体实现。

⑤对于程序的鲁棒性：一个好的应用程序，操作是友善的，而且BUG不会太多，对于这些良好的用户体验，让我费劲心思去完善小小的细节（虽然仍会有不足的地方），但是抠细节才能更好的做出棒的软件。

五、结束语：

整个系统的开发过程中几乎涵盖了课堂上所教授的大部分图形生成算法， 整个实现的过程使我充分地认识到了理论学习与上机实现的差距， 在上机实现实现算法的过程中遇到的种种困难也大大加深了我对相应的算法的理解。 总的来说， 这些基础而又重要的图形学算法使我对计算机图形学有了新的认识和看法。感谢图形学，领我进入了一个新的知识领域，并且让我学会将知识转化成现实的应用，让我受益匪浅！

致谢 在此,我向图形学老师的严谨教学及助教们的工作表示感谢！

引用文献参考：

[1] 中点画圆算法：http://blog.csdn.net/u013044116/article/details/49305017

[2]双缓冲绘图：https://blog.csdn.net/u012891055/article/details/41727391?utm_source=blogxgwz1

[3]画板的实现：https://blog.csdn.net/cutter_point/article/details/43087497

[4]信号：https://blog.csdn.net/qq_31821675/article/details/69951562?locationNum=8&fps=1

[5]画笔画刷类：http://www.cnblogs.com/Jace-Lee/p/5946342.html

[6] 椭圆的生成算法:http://blog.csdn.net/orbit/article/details/7496008

[7] Bezier 曲线绘制原理:http://blog.csdn.net/joogle/article/details/7975118

[8]点的旋转:https://blog.csdn.net/yuliying/article/details/73104801

[9]梁友栋直线裁剪算法：https://blog.csdn.net/daisy_ben/article/details/51941608

[10]3D显示:http://www.cnblogs.com/aminxu/p/4705604.html?tdsourcetag=s_pctim_aiomsg

[11]off文件读取:https://blog.csdn.net/OOFFrankDura/article/details/80203664?tdsourcetag=s_pctim_aiomsg

[12]Bezier 曲线绘制原理:https://aaaaaaaty.github.io/bezierMaker.js/playground/playground.html.

附中文参考文献:

[1] 孙正兴——计算机图形学教程 [2] 课堂教学 PPT 内容