

《计算机图形学》系统技术报告

黄彬寓 161220047

(南京大学 计算机科学与技术系, 南京 210093)

摘 要: 这是图形学大作业系统技术报告, 主要报告了大体上系统的框架和实现算法。

1 实验二维图形陈述

1.1 窗口设计---MainWindow类(由QMainWindow类派生)

该类利用 ui, 提供了系统窗口, 并为各种操作按钮提供了摆放的位置, 各按钮有各自对应的信号槽和信号函数, 右图与下图为具体实现的按钮:



该类的另一作用: 将 QScrollArea 的滚动区域类和自制的画板类作为成员对象, 这样滚动窗口和自制的画板就可以成功出现在窗口中了。

```
PaintBackground *area;  
QScrollArea *scrollArea;
```

在这些按钮中, 大部分按钮都以 MainWindow 作为中间桥梁给 PaintBackground 画板类传递信号、参数, 于是 PaintBackground 画板类接收到信号后调用相应的函数做出相应的操作。还有一些按钮用于对当前窗口的整体进行操作, 如 File 中的按钮: 打开、退出、创建、保存、另存为、打印等操作。

- 1) 打开: 在打开前判断当前图片是否被修改过, 若是, 则先跳出对话框询问当前图片是否保存, 若保存则跳转保存任务后跳转回来; 若未被修改, 则进行打开, 修改两个成员对象 area 与 scrollArea。
- 2) 退出: 在退出前判断当前图片是否被修改过, 若是, 则先跳出对话框询问当前图片是否保存, 若保存则跳转保存任务后跳转回来; 若未被修改或拒绝保存, 则退出程序。
- 3) 创建: 点击创建按钮后, 跳出新对话框 Dialog(见右图), 该对话框用于接收用户想要创建画布的大小及背景色, 点击 OK 按钮后修改两个成员对象 area 与 scrollArea, 生成用户想要的画布。



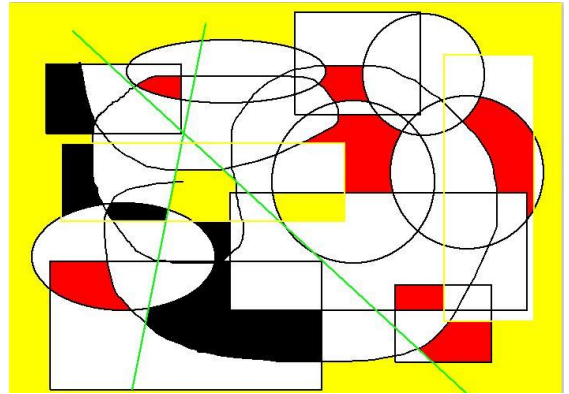
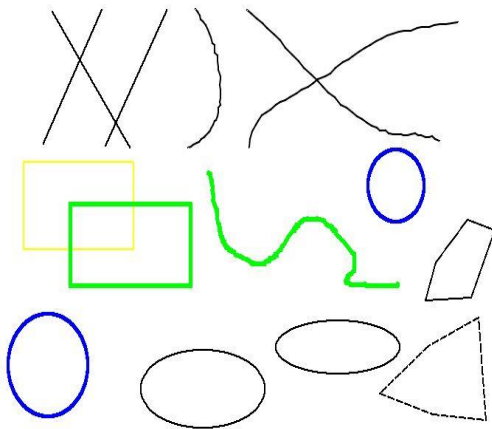
- 4) 保存：在保存前判断当前画布是否曾经保存过，若曾保存过，则直接调用 area 类中的保存图像 saveImage 函数，更新保存路径中的图像；若未曾保存过，则转至“另存为”任务。
- 5) 另存为：点击另存为按钮后，跳出 QFileDialog 类型的对话框选择保存路径，选择完成后保存至对应路径。
- 6) 打印：转至成员对象 Area 中的 doPrint 函数，跳转至 QprintDialog 的打印界面，当接收到信号 Accepted 时，执行打印。

其余按钮，均作为给 Area 画板对象传递信号的函数，将在下文介绍 Area 对象时或程序使用手册中详解。

1.2 画布设计---PaintBackground类(由QWidget派生)

这里是二维图形的功能实现的地方，主要实现在 mousePressEvent、mouseMoveEvent 及 mouseReleaseEvent 和 paint 函数中

1) 二维图形的输入：



1.1. 画笔：

在 mousePressEvent 中将 firstPoint 赋为鼠标当前坐标，在 mouseMoveEvent 和 mouseReleaseEvent 中将 secondPoint 赋为鼠标当前坐标，利用“化整体曲为部分直”的思想，在(firstPoint,secondPoint)段画直线，画完将 firstPoint 赋为 secondPoint 即可。

1.2. 直线：

在 mousePressEvent 中将 firstPoint 赋为鼠标当前坐标，在 mouseMoveEvent 和 mouseReleaseEvent 中将 secondPoint 赋为鼠标当前坐标。其中画直线使用的是“Bresenham 算法”，使用决策参数：

($0 < k < 1$ 部分)

利用 QPainter 类中的画点函数 DrawPoint 循环调用直至从一个点移动到另一个点为止。

若 $p_k > 0$ ，取高像素(x_{k+1}, y_{k+1}): $y_{k+1} - y_k = 1$,

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x;$$

若 $p_k < 0$ ，取低像素(x_{k+1}, y_k): $y_{k+1} - y_k = 0$,

$$p_{k+1} = p_k + 2\Delta y.$$

1.3. 矩形：

在 firstPoint 和 secondPoint 点与直线相同处理，得到这两个点后进行画矩形，根据

```
QPoint pp1(firstPoint.x(),secondPoint.y());
QPoint pp2(secondPoint.x(),firstPoint.y());
```

该两点找到矩形的另外两点：

于是根据这四个点，每相邻的两个点调用一次上述 Bresenham 画直线法，总共调用四次，生成了矩形。

1.4. 圆形：

在 firstPoint 和 secondPoint 点与直线相同处理，得到这两个点后进行画圆，其中令 firstPoint 为圆心，secondPoint 为圆上一点，故得到圆的半径 radius。现考虑“中点圆生成算法”，只要绘得中点圆第一象限上半部分的圆上点，便能通过圆的轴对称和中心对称得到其他 7 个部分上的对应点，再根据圆心进行集体的相对于原点(0,0)进行坐标偏移即可。在所求部分内，圆的切线的斜率永远是在-1 到 0 之间的，于是可以用类似于直线生成的方法，其中中点圆的决策参数使用的是：

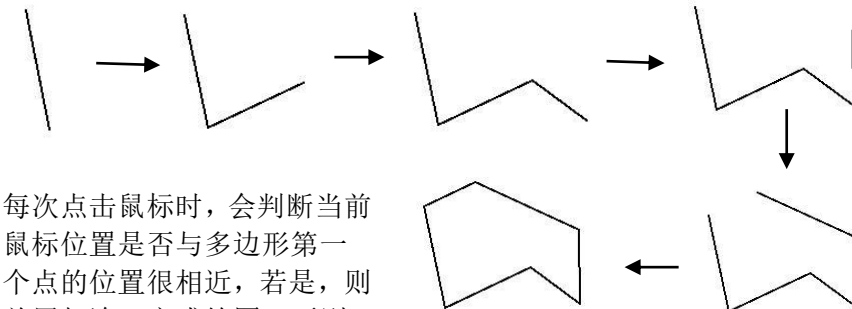
当 $p_k \geq 0$, $p_{k+1} = p_k + 4(x_i - y_i) + 10$; 当 $p_k < 0$, $p_{k+1} = p_k + 4x_i + 6$ 。

1.5. 椭圆：

在 firstPoint 和 secondPoint 点与直线相同处理，得到这两个点后进行画椭圆，其中令 firstPoint 和 seconPoint 为希望生成椭圆的外切矩形的两个对角点，于是矩形的长和宽的 1/2 得到了椭圆的长短轴参数 r_x 、 r_y 。使用“中点椭圆生成算法”，只需要求得中点椭圆第一象限内的椭圆上点，再通过轴对称和中心对称就能获得另外三个象限上的对应点。在第一象限部分中，可以分为斜率在 0 到-1 的上半部分和-1 到无穷的下半部分，在上半部分中， $\Delta y = -1$, $\Delta x \geq 0$ ；在下半部分中， $\Delta x = -1$, $\Delta y \geq 0$ 。上半部分循环计算调用 drawPoint 至斜率为-1，下半部分循环计算调用 drawPoint 至 y 为 0。

1.6. 多边形：

由于我实现的是多次点击鼠标左键直至鼠标点击至多边形初始位置附近时停止的实现方式，因此每次鼠标信号只读入 firstPoint 的值即可。



每次点击鼠标时，会判断当前鼠标位置是否与多边形第一个点的位置很相近，若是，则首尾相连，完成绘图，否则，点击下一个点时继续进行多边形点扩充。

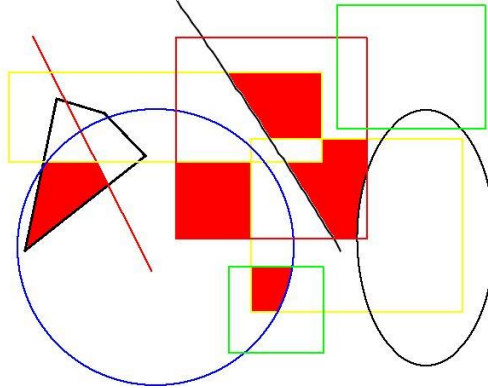
1.7. 曲线：

曲线的生成在鼠标点击方面与多边形类似但最后一次不同，多次点击鼠标左键，会安放曲线当前多个点，最后点击一次鼠标右键，就生成曲线，这里使用的是逼近样条，因此曲线会比插值样条相对来说平缓一些。采用 Bezier 曲线算法，加上反走样使曲线看上去尽可能平滑。

```
pic.setRenderHint(QPainter::HighQualityAntialiasing);
for(int i = 1; i <= 40; i++)
{
    t += 0.025;
    QPointF now(o, o);
    for(int j = 0; j < size; j++)
        now += points[j] * C(n, j) * qPow(t, j) * qPow(1-t, n-j);
    pic.draw(pre, now);
    pre = now;
}
```

1.8. 填充区域:

使用 4 连通洪泛填充算法，每次填充一个像素点的颜色，一开始使用的是递归洪泛填充算法，导致填充稍微大一点区域，程序就进入了崩溃状态，经调试发现是深度递归过度产生了爆栈的影响，于是把递归算法改成了迭代算法，每次判断一个点符合填充要求就让其进入队列，直至队列为空，循环结束。



1.9. 橡皮:

实现与画笔一模一样，唯一的区别是其颜色固定为背景色。

1.10.

画笔颜色、画笔线宽、画笔风格均由 MainWindow 提供信号槽并将信息转入 area 对象中以修改当前画笔风格。

2)图形生成过程可视化:

为了让比如画一个矩形的时候，当鼠标按下并拖动时会有一个临时矩形随着鼠标拖动而移动的过程得到体现，并且不会产生冗余的过程中的图形，我这里使用了两张画布，一张是主画布，一张是临时画布，临时画布主要用在绘制图形过程中将主画布遮盖。在 mouseMoveEvent 函数中，执行 tempImage=image;和 paint(tempImage);，于是主画布上这时没有多余图形出现，出现的图形只在临时画布上，为了让这一过程时显示的是临时画布，还需对 paintEvent 函数进行修改，

```
void PaintBackground::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    if(drawing==false ||
        (!in_operating&&(graphic_kind==FILLED ||graphic_kind==CURVE||graphic_kind==POLYGON))||
        graphic_kind==RUBBER|| graphic_kind == PEN )
        painter.drawImage(o,o,image);
    else
        painter.drawImage(o,o,tempImage);
}
```

如右图，
当 drawing
为 false(没
在绘制图形
)，或没在
执行各种编

辑图形操作时(填充、曲线、多边形没有过程中生成因此排除)，画布上出现的是主画布，否则，比如 drawing==true(正在画图)或 in_operatin(正在执行图形操作)等情况时，需要呈现临时画布。这里我没有使用每次在 paintEvent 中遍历图形存放表格来重绘 image 的原因是想尽可能地提高运行速度，毕竟每次调用到 mouseMoveEvent 函数便重绘，则画任何一跳直线可能会遍历图形存放表并重绘几十次，是个不小的开销。

3) 二维图形编辑及平移、旋转、缩放、裁剪:

为了能让画板上的二维图形可编辑, 我将创建了一个基类 `Shape`, 将上述的直线 `Line` 类、矩形 `Rect` 类、圆形 `Circle` 类、椭圆 `Ellipse` 类、曲线 `Curve` 类、多边形 `Polygon` 类作为 `Shape` 的派生类。在 `Shape` 类中加入纯虚函数 `draw()` 等用以画出图形。

添加所有图形存放表 `QVector<Shape*> shapes`; 添加当前被选中图形 `Shape* cur_shape`; 这两者的并集是画布上的所有图形, 当某个图形被选中,

1. 若原本有其他图形被选中, 则将那个图形 `cur_shape` 放入 `shapes` 中, 并将被选中的图形从 `shapes` 中取出并存入 `cur_shape` 中, 进行操作;
2. 若原本该图形就是被选中的, 那这部操作忽略;
3. 若原本没有任何图形被选中, 则将被选中的图形从 `shapes` 中取出并存入 `cur_shape`。

注: 如何可选中在这里是最主要的问题, 首先在 `Shape` 类中加入纯虚函数 `beChosen(QPoint p)` 代表是否被选中, 返回 `bool` 型。具体 `beChosen(QPoint p)` 函数所需要做的就是重绘图形, 即几乎和 `draw()` 的框架相同, 然而不同的地方在于, `draw()` 中描点的部分 `drawPoint` 在 `beChosen(QPoint p)` 中被换成了判断, 即判断在 `draw()` 中描点的那个点 (x, y) 和我鼠标点击的位置坐标 p 是否相同, 若相同则返回 `true`, 说明鼠标点击的位置在直线上, 若整个循环结束仍没有 `return true`, 则在函数最后 `return false`, 表示没有选中。为了让选中的范围不是太小导致难以选中, 若鼠标点在图形中任一点的以 7 个像素点为半径的圆内, 则视为选中了图形。如圆形:

```
bool beChosen(QPoint point){
    int x=0,y=radius;
    int p=3-2*radius;
    for(;x<=y;x++)
    {
        if(near_8points(x,y,point.x(),point.y())==true)
            return true;
        if(p>=0)
        {
            p=p+4*(x-y)+10;
            y--;
        }
        else
            p=p+4*x+6;
    }
    return false;
}
```

```
void draw_circle(QPainter &pic){
    pen=pic.pen();
    if(is_chosen)
        pic.setPen(onepen);
    int x=0,y=radius;
    int p=3-2*radius;
    for(;x<=y;x++)
    {
        draw_8points(pic,x,y);
        if(p>=0)
        {
            p=p+4*(x-y)+10;
            y--;
        }
        else
            p=p+4*x+6;
    }
}
```


其内部操作 near_8points()和 draw_8points()为:

```
bool near_8points(int x,int y,int tx,int ty){
    int pos_x,pos_y;
    pos_x=center_x+x; pos_y=center_y+y;
    if((pos_x-tx)*(pos_x-tx)+(pos_y-ty)*(pos_y-ty)<=49)
        return true;
    pos_x=center_x+y; pos_y=center_y+x;
    if((pos_x-tx)*(pos_x-tx)+(pos_y-ty)*(pos_y-ty)<=49)
        return true;
    pos_x=center_x-x; pos_y=center_y+y;
    if((pos_x-tx)*(pos_x-tx)+(pos_y-ty)*(pos_y-ty)<=49){
        return true;
    }
    pos_x=center_x-y; pos_y=center_y+x;
    if((pos_x-tx)*(pos_x-tx)+(pos_y-ty)*(pos_y-ty)<=49)
        return true;
    pos_x=center_x+x; pos_y=center_y-y;
    if((pos_x-tx)*(pos_x-tx)+(pos_y-ty)*(pos_y-ty)<=49)
        return true;
    pos_x=center_x+y; pos_y=center_y-y;
    if((pos_x-tx)*(pos_x-tx)+(pos_y-ty)*(pos_y-ty)<=49)
        return true;
    pos_x=center_x-x; pos_y=center_y-y;
    if((pos_x-tx)*(pos_x-tx)+(pos_y-ty)*(pos_y-ty)<=49)
        return true;
    pos_x=center_x-y; pos_y=center_y-y;
    if((pos_x-tx)*(pos_x-tx)+(pos_y-ty)*(pos_y-ty)<=49)
        return true;
    return false;
}
```

```
void draw_8points(QPainter &pic,int x,int y){
    pic.drawPoint(center_x+x,center_y+y);
    pic.drawPoint(center_x+y,center_y+x);
    pic.drawPoint(center_x-x,center_y+y);
    pic.drawPoint(center_x-y,center_y+x);
    pic.drawPoint(center_x+x,center_y-y);
    pic.drawPoint(center_x+y,center_y-y);
    pic.drawPoint(center_x-x,center_y-y);
    pic.drawPoint(center_x-y,center_y-y);
}
```

于是下列各操作都是基于某图形被选中的,若当前没有选中图形,则相当于操作没有实施。

1. 平移:

首先,在 Shape 基类中加入纯虚函数 void translation(QPoint p1,QPoint p2)=0;,然后在每个派生类图形对象中加入该函数,p1 和 p2 为鼠标点击拖动起始点和终止松开点,故距离 p2-p1 即为平移的距离,对于各自的图形,将其的关键点平移(p2-p1)个距离后画出新图形,即调用 draw 函数即可。其中需要注意的问题是:为了让过程可视化,每次 mouseMoveEvent 就会调用 translation 和 draw,故 translation 的两个参数就不能填为 firstPoint 和 secondPoint 了,于是加入 lastPoint,代表上一次调用 mouseMoveEvent 函数时的 secondPoint,于是 translation 的两个实参就可以写为 translation(lastPoint,secondPoint),否则若用 firstPoint 代替 lastPoint 会导致小小的平移一下就带来图形的一大段移动。

2. 旋转:

这里旋转其实就是基于某个点将所有关键点坐标进行变换,得到新的坐标点后画出形,即调用 draw 函数即可。首先需要在 Shape 基类中加入纯虚函数 void rotation(int angle)=0,;

我们可以先把旋转基点认定为原点(0, 0),各个关键点相应的得到基点在原点时的相应坐标(x,y)。

然后就可以使用基点为原点时的旋转矩阵:

$$x_1 = \cos(\text{angle}) * x - \sin(\text{angle}) * y;$$

$$y_1 = \sin(\text{angle}) * x + \cos(\text{angle}) * y;$$

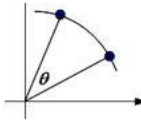
再把基点原来的坐标 p 加上(x₁,y₁),就得到了某个关键点旋转后的坐标。

的旋转和旋转。

基准点为坐标原点:

$$\begin{cases} x_1 = x \cos \theta - y \sin \theta \\ y_1 = x \sin \theta + y \cos \theta \end{cases}$$

任意基准位置:

$$R = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$


通过旋转角度 `angle` 和旋转按钮的传入，就可以成功进行旋转。但是系统封装的 `sin` 和 `cos` 函数中，参数是弧度制的，故得到 `angle` 后还需要将其/ $180 \times \text{PI}$ ，由角度值转为弧度值，具体步骤为：

```
int x = point.x()-point_st.x();
int y = point.y()-point_st.y();
int x1 = cos(angle/180.0*PI)*x-sin(angle/180.0*PI)*y;
int y1 = sin(angle/180.0*PI)*x+cos(angle/180.0*PI)*y;
point.setX(point_st.x()+x1);
point.setY(point_st.y()+y1);
```

3. 缩放：

首先，在 `Shape` 基类中加入纯虚函数 `void changeSize(int index)=0;`，再各个派生类中对其进行重写。这其实也是对各图形的关键点进行坐标转换，得到新坐标后，调用各个图形的 `draw` 函数进行重绘即可。为了让缩放变得尽量省力，我设置了三处信号，分别为 `Edit` 中的 `Magnify` 按钮、`Narrow` 按钮，和鼠标滚轮(向上滚放大，向下滚缩小)，和双击图形(左键双击图形放大，右键双击图形缩小)。将他们按比例进行缩小或者放大。

4. 裁剪：

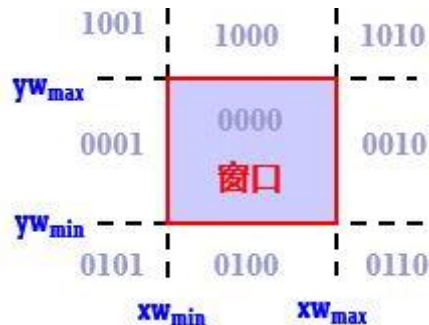
这里在 `Shape` 基类中首先加入纯虚函数 `cut()`；，在直线中对其进行重写，完成矩形框裁剪直线的过程。首先需要有个过程可视化的矩形，这里和输入图形的矩形大同小异，主要是到 `mouseReleaseEvent` 函数接收到时，将矩形的两个对角 `p1`, `p2` 传入 `cut` 函数中，显然这里实参即为 `firstPoint` 和 `secondPoint`。

这里我采用 Cohen-Sutherland 算法，

设置 `leftBitCode=1`, `rightBitCode=2`,

`BottomBitCode=4`, `topBitCode=8`，对任意一个点，通过 `encode` 函数和裁剪窗口，得到对应的 `code` 值。

```
inline bool inside(int code){return (code==0);}
inline bool accept(int code1,int code2){
    return ((code1==0)&&(code2==0));
}
inline bool reject(int code1,int code2){
    return ((code1&code2)!=0);
}
int encode(QPoint p,int left,int right,int bottom,int top){
    int code=0;
    if(p.x()<left)
        code=code|leftBitCode;
    if(p.x()>right)
        code=code|rightBitCode;
    if(p.y()<bottom)
        code=code|bottomBitCode;
    if(p.y()>top)
        code=code|topBitCode;
    return code;
}
```



`inside` 函数用来判断某个点是否在窗口内；

`accept` 函数用来判断两个点是否存在窗口内，若是，则这段线段可以全部保存；

`reject` 函数用来判断两个点是否是在同一边，若是，则说明这段线段与裁剪窗口一定没有相交部分。

然后按照“左-右-下-上”的顺序循环对直线的剩下部分和裁剪边界进行比较和求交，直到 `accept`(找到一段符合要求的线段)或 `reject`(完全被舍弃)中有一个值为 `true` 时，裁剪停止。

2 实验三维图形陈述

2.1 了解OFF文件格式

1. 第一行为读入三维图形的顶点数、面数、边数。

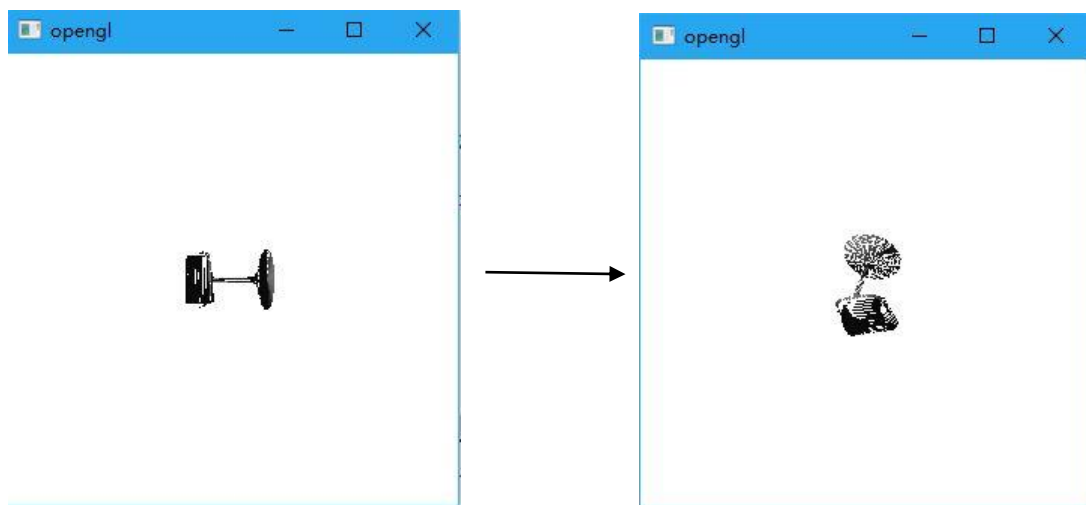
一个三维图形可以通过顶点数和面数已经每个面上的顶点来确定图形模样，故边数这里没有要求。

2. 接下来的第 2 行到第 `vert_num+1` 行为各个顶点的三维坐标。

3. 接下来的 `face_num` 行为各个面上所拥有的顶点。

2.2 用OpenGL实现三维图形导入

程序运行时首先跳出文件对话框选择 OFF 格式文件导入，之后根据 OFF 文件格式导入信息至结构体 `myMesh` 中，再开始 `draw` 操作，于是三维图形就可以生成在窗口中。其中设置了鼠标点击信号 `mousePressEvent`，当接受到鼠标左键点击时，将生成的，三维图形进行旋转 5 度。多次点击后就能看到三维图形的多个面。



3 结束语

这次实验是进大学以来为数不多的大型实验，在这次实验中我感觉到自己的自学能力和负重能力得到了较大的提升，对计算机世界、像素世界里的图形生成和编辑有了很深刻的认识，感谢老师和助教一个学期辛勤的上课指导和督促。

References:

- [1] 课件 PPT.CG02,CG03,CG04,CG05
- [2] 窗口部件的使用: <https://blog.csdn.net/kilotwo/article/details/79238545>;
- [3] 三角函数(sin,cos)的参数: <https://zhidao.baidu.com/question/33163812.html>;
- [4] Qt 画图软件框架入门:
https://max.book118.com/html/2017/0608/112558330.shtm?tdsourcetag=s_pctim_aiomsg;
- [5] 椭圆生成算法的学习: <https://blog.csdn.net/orbit/article/details/7496008>;
- [6] 洪泛区域填充算法的学习: <https://blog.csdn.net/orbit/article/details/7323090>;
- [7] 洪泛区域填充算法的学习: <https://www.cnblogs.com/songr/p/5773133.html>;
- [8] Qt 画图框架学习: <https://blog.csdn.net/lq1259156776/article/details/52454865>;
- [9] 旋转变换矩阵: <https://blog.csdn.net/csxiaoshui/article/details/65446125>;
- [10] 裁剪直线算法思路: <https://blog.csdn.net/pleasecallmewhy/article/details/8393445>;
- [11] 反走样技术: <https://bbs.csdn.net/topics/330186675>;