《计算机图形学》系统技术报告

作者姓名 胡思聪（161220046）*

1 综述

1.1 开发环境:2D：win10 + QT Creator 4.7.1。3D：vs2015+ OpenGL+Openmesh

1.2 编译器说明：编译器为MSVC2015_32bit

1.3 程序运行说明：2D和3D图形分开运行。

1.4 10月完成情况:

使用 QT 完成了初始程序的构建
使用中点画法，完成了直线，圆，椭圆的绘制
实现了保存功能
实现了绘制的撤销，前进操作。

11 月完成情况:

重构了整个系统，图元的绘制变换等完全由 graphics view/scene/items 实现。
优化了用户体验，图元绘制由鼠标实现，不再需要手动输入坐标，且中间过程可以实时绘制
添加了多边形的绘制。
添加了直线，圆，椭圆，多边形的编辑，旋转，缩放，平移。
（只针对当前图元，图元的鼠标选中功能将于 12 月完成；
由于旋转不闭合，圆与椭圆暂时只支持旋转 90 度的整数倍；
旋转，缩放功能可以自由指定操作的中心点）。
添加了区域填充。
暂时取消了保存功能和操作的回退功能。

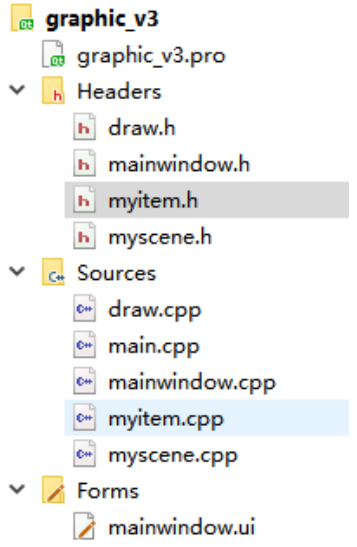
最终实验完成功能：（基于 11 月）
再次重构系统，不再继承 QgraphicsView；所有的继承，重写基于 QgraphicsScene 和 QgraphicsItem。
添加了图元选中功能。所有已绘制的图元可以再操作。
添加了图元删除功能。
恢复了保存功能。同时添加了文件选择对话框。
添加了曲线的绘制。
添加了图元的裁剪。
优化了用户体验。删除了平移，旋转等操作的选择按键。图元操作前无需点击按键。直接操作或结合快捷键即可。
注：图元选择的判定使用了默认的 boundingRect。所以判定范围为图元所在的矩形框。范围较大。实现了 shape 函数优化判定，但是此时判定需要非常精确，用户体验反而不如原始的 boundingrect。所以注释了 myitem.cpp 的 shape 函数。如需查看，删除注释重新编译即可。

---

* 作者简介:

2 算法介绍

2.1 总体框架



draw.h+draw.cpp 图元的绘制函数

myitem.h + myitem.cpp 继承 QGraphicsItem

myscene.h + myscene.cpp 继承 QgraphicsScene

mainwindow：主窗口。

2.2 鼠标事件详解（新增功能）

相较 11 月，此处为主要改变。

由于自定义了 QgraphicsScene，鼠标点击主界面，此时鼠标事件先传递给 Scene，再通过 Scene 的 `QGraphicsScene::mousePressEvent(event);`

将鼠标事件向下传递。若鼠标点击之处有 items，则该鼠标事件被 items 接收。Items 接收到鼠标事件并决定是由图元执行，还是忽略操作。

若忽略，该事件传回 Scene。并在 Scene 中执行对应操作。

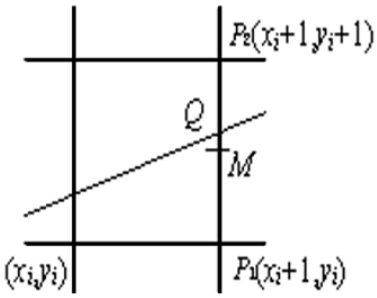
基于上述过程。图元的自由选中即可实现。同时由于划分 Scene 和 items 两个部分，代码框架更为清晰，简洁。

2.3 图元绘制算法

图元绘制利用 graphicsitem，自定义 graphicsitem 类并重写 painter 函数。通过鼠标事件获取点坐标并保存在数组中，利用 painter 函数，根据类型分别绘制即可。

2.3.1 直线绘制

直线在 graphicsitem 中保存形式为两个端点。绘制利用中点直线画法，完成斜率为【0，1】的直线的绘制，其余情况的直线通过对称变换获得。



如图所示，对于斜率小于 1 的直线，每次 x 坐标加 1 再利用中点与直线的关系决定 y 坐标不变还是加 1。具体算法如下

```
01.  void MainWindow::paintline_0_1(int x0,int y0,int x1,int y1,int z0,int z1,bool reverse)
02.  {
03.      QPainter painter(&pixmap);
04.      int a,b,delta1,delta2,d,x,y ;
05.      a = y0 - y1;
06.      b = x1 - x0;
07.      d = 2 * a + b ;
08.      delta1 = 2 * a ;
09.      delta2 = 2 *( a + b);
10.      x = x0 ;
11.      y = y0 ;
12.      if(reverse==false)
13.          painter.drawPoint(z0*x, z1*y);
14.      else
15.          painter.drawPoint(z0*y, z1*x);
16.      while( x<x1  )
17.      {
18.          if( d<0 )
19.          {
20.              x ++;
21.              y ++;
22.              d+= delta2;
23.          }
24.          else
25.          {
26.              x++;
27.              d+= delta1;
28.          }
29.          if(reverse==false)
30.              painter.drawPoint(z0*x, z1*y);
31.          else
32.              painter.drawPoint(z0*y, z1*x);
33.       }
34.  }
```

绘制过程中，通过传入的参数，实行坐标变换，得到最终直线。

void MainWindow::paintline_0_1(int x0,int y0,int x1,int y1,int z0,int z1,bool reverse);

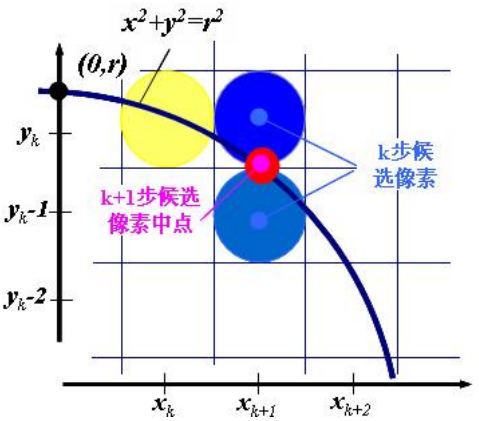
绘制了点（x0,y0）到点（x1,y1）的斜率在 0 到 1 之间的直线。参数 z0, z1, reverse 分别决定了直线沿着 y 轴，x 轴和 y=x 对称的相应直线。

```
double dy_dx=(double)(y1-y0)/(x1-x0);
if(dy_dx>=0&&dy_dx<=1)
{
    paintline_0_1(x0,y0,x1,y1,1,1,false);
}
else if(dy_dx>1)
{
    paintline_0_1(y0,x0,y1,x1,1,1,true);
}
else if(dy_dx<0&&dy_dx>=-1)
{
    paintline_0_1(x0,-y0,x1,-y1,1,-1,false);
}
else
{
    paintline_0_1(-y0,x0,-y1,x1,1,-1,true);
}
```

2.3.2 圆绘制

圆在 graphics item 中只保存鼠标事件起始点和终止点。通过计算得到圆心和半径

```
int x0=(start_x+end_x)/2;
int y0=(start_y+end_y)/2;
int R=min(start_x-end_x>0?start_x-end_x:end_x-start_x,start_y-end_y>0?start_y-end_y:end_y-start_y)/2;
```



圆绘制从（0，R）开始沿着 x 轴方向绘制出 1/8 个圆弧，再通过对称变换得到整个圆。

从（0，R）开始沿着 x 轴方向绘制，x 坐标每次加 1，y 坐标或不变或减 1，因此同样利用中点与圆的关系决定 y 坐标是否变化。具体代码省略。

2.3.3 椭圆绘制

椭圆的存储也是鼠标事件的起始点和终止点。通过计算得到椭圆的圆心和长短半轴。

```
int xc=(start_x + end_x)/2;
int yc=(start_y + end_y)/2;
int rx=(start_x - end_x>0?start_x - end_x:end_x-start_x)/2;
int ry=(start_y - end_y>0?start_y - end_y:end_y-start_y)/2;
```

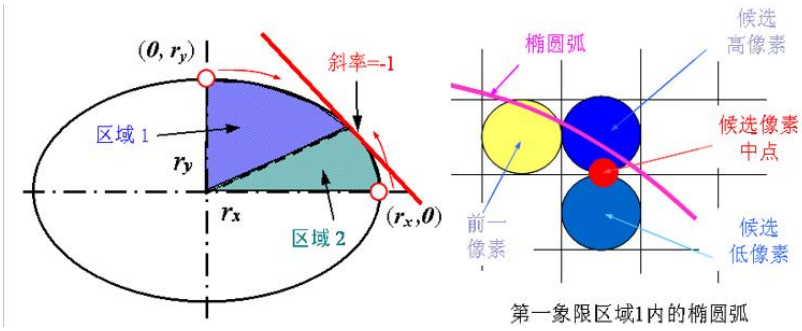
对于 rx>ry 的椭圆：

从（0，ry）开始绘制，沿着 x 轴方向采用中点法绘制，直至 dy/dx=-1。再沿着 y 轴方向绘制，直至（ry，0）处。

对于 rx<ry 的椭圆

将上述椭圆沿着 y=x 对称变换即可。

对于每个点（x,y），（x+x0,y+y0）即为要求所得的椭圆（x0,y0 为椭圆圆心）。



第一象限区域1内的椭圆弧

2.3.4 多边形绘制

多边形的实质为点的集合，保存每条线段的端点，并利用端点绘制多条线段即可得到所求多边形。在多边形绘制过程中，中间线段的起始点只能为上条线段的终止点。且当前线段终止点与多边形起始点距离相近时，自动将两点拟合为 1 点，多边形绘制完成。

附上 graphicsitem 的结构

```
1.  class Myitem : public QGraphicsItem
2.  {
3.  public:
4.      int type;
5.      QPoint points[100];
6.      int points_num;
7.  public:
8.      Myitem();
9.      QRectF boundingRect() const;
10.     void paint(QPainter *painter, const QStyleOptionGraphicsItem * option, QWi
    dget * widget);
11. };
```

2.4 图元的编辑，旋转，缩放，平移。

所有图元操作的思路都是通过鼠标事件改变图元的某点的坐标或者通过变换改变图元所有点的坐标，再利用 graphicsitem 的 painter 函数实时重绘。

2.4.1 图元的编辑(改变端点坐标)

图元编辑，当鼠标 pressevent 的坐标在该 item 的某点附近时，选取改点；后续 moveevent 时，将鼠标拖动过程中的点实时赋值给该 item 的选中坐标，并实时 update 即可。

2.4.2 图元的旋转

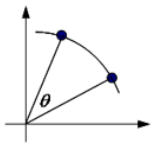
图元旋转先通过鼠标点击获得旋转的中心点，再点击并拖动实现旋转。旋转角度的求法如下，分别计算拖动点和拖动起始点与旋转中心的向量 v1,v2; 再利用反三角函数求得夹角。

```
1.  v1.rx()=from.x()-center_point.x();
2.  v1.ry()=from.y()-center_point.y();
3.  v2.rx()=to.x()-center_point.x();
```

```
4.  v2.ry()=to.y()-center_point.y();
5.  double Theta=atan2(v2.y(),v2.x())-atan2(v1.y(),v1.x());
6.  if(Theta>pi)
7.  {
8.      Theta=Theta-pi*2;
9.  }
10. if(Theta<-1*pi)
11. {
12.     Theta=Theta+pi*2;
13. }
```

的旋转轴旋转。

- 基准点为坐标原点：
  - $x_1=x\cos\theta-y\sin\theta$，
  - $y_1=x\sin\theta+y\cos\theta$。



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

- 任意基准位置：
  - $x_1=x_r+(x-x_r)\cos\theta-(y-y_r)\sin\theta$
  - $y_1=y_r+(x-x_r)\sin\theta+(y-y_r)\cos\theta$

直线和多边形的旋转即为旋转每个端点。再根据旋转后的端点依次绘制直线即可。

圆和椭圆由于算法的缘故目前只支持旋转 90 度的整数倍，此时也是旋转端点并依据旋转后的端点重绘即可。（圆和椭圆每个点都旋转一定角度虽然也能实现任意旋转的效果，但是此时结果不封闭。故目前不做任意角度的旋转）

2.4.3 图元的缩放

图元的缩放和旋转类似，也是先用鼠标指定基点，再滚顶鼠标滑轮，实现图元的缩放。

固定点缩放：选择变换后不改变物体位置的点(x_f, y_f)进行缩放。

- 多边形顶点、物体中点。

顶点(x,y)缩放后坐标(x_1,y_1)为：

$x_1=x\cdot s_x+x_f(1-s_x)$;

$y_1=y\cdot s_y+y_f(1-s_y)$;

对每个图元的保存点依照公式矩阵变换后，按照变换后的点重绘即可得到缩放后的图元。

部分代码展示:

```
1.  void MainWindow::read_wheel_emit(int x)
2.  {
3.      double zoom_factor;
4.      if(x==120)
5.          zoom_factor=1.1;
6.      else
7.          zoom_factor=0.9;
8.      if(zoom_start==1)
```

```
    {
        for(int i=0; i<item[item_num]->points_num; i++)
        {
            int x0=zoom_center.points[0].x();
            int y0=zoom_center.points[0].y();
            item[item_num]->points[i].rx()=item[item_num]->points[i].x()*zoom_factor+x0*(1-zoom_factor);
            item[item_num]->points[i].ry()=item[item_num]->points[i].y()*zoom_factor+y0*(1-zoom_factor);
            Rect_select->points[i].rx()=Rect_select->points[i].x()*zoom_factor+x0*(1-zoom_factor);
            Rect_select->points[i].ry()=Rect_select->points[i].y()*zoom_factor+y0*(1-zoom_factor);
        }
        item[item_num]->update();
        Rect_select->update();
        scene->update();
        scene->setSceneRect(0,0,1920,1080);
        ui->graphicsView->setScene(scene);
        ui->graphicsView->show();
    }
}
```

2.4.4 图元的平移

图元平移较为简单，将每个图元的保存点加上相应的偏移值后得到新值，再将图元按新值重绘即可。偏移值通过鼠标事件的拖动操作获得。

2.5 区域的填充。

区域填充由于设计只在闭合状态下填充，所以采用了泛滥填充。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。由于直接递归，像素太多必然爆栈。所以用了 Qstack 栈，通过迭代，将当前像素点上下左右的合法点压入栈内，出栈染色并循环，直至栈空。

边界检测方法: 将 graphicsscene 渲染至 image 上，再检测 image 的像素颜色信息。填充也是填至另一张 image 上，最后 additem 至 scene 中即可。

部分代码展示:

```
while(!fillpoints.empty())
{
    QPoint point_pop;
    point_pop=fillpoints.pop();
    int a=point_pop.x();
    int b=point_pop.y();
    if(a<0||b<0||a>=1920||b>=1080)
    {
        over_flag=1;
        fillpoints.clear();
        return;
    }
```

```
13.     if(a-1<0||b-1<0||a+1>=1920||b+1>=1080)
14.     {
15.         over_flag=1;
16.         fillpoints.clear();
17.         return;
18.     }
19.     painter.setPen(pen);
20.     painter.drawPoint(a,b);
21.     if((QColor)fillmap.pixel(a,b-1)==Qt::white&&(QColor)scenemap.pixel(a,b-1)==Qt::white)  {…}
22.     if((QColor)fillmap.pixel(a-1,b)==Qt::white&&(QColor)scenemap.pixel(a-1,b)==Qt::white)  {…}
23.     if((QColor)fillmap.pixel(a,b+1)==Qt::white&&(QColor)scenemap.pixel(a,b+1)==Qt::white)  {…}
24.     if((QColor)fillmap.pixel(a+1,b)==Qt::white&&(QColor)scenemap.pixel(a+1,b)==Qt::white)  {…}
25. }
```

2.6 直线的裁剪算法

直线裁剪采用梁友栋算法。中心思想即为求裁剪窗口和直线的交点。

出发点：设要裁剪的直线段为P_0 P_1，P_i的坐标为(x_i,y_i)，i=0，1。 P_0P_1和窗口边界交于A、B、C和D四个点。

如果点P（x，y）位于由坐标（xmin，ymin）和（xmax，ymax）所确定的窗口内，那么下式成立：

$$xmin \le x1 + u \cdot \Delta x \le xmax$$

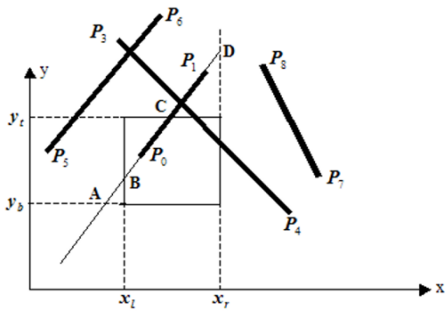
$$ymin \le y1 + u \cdot \Delta y \le ymax$$

这四个不等式可以表示为：

$$u \cdot pk \le qk, \quad k=1,2,3,4$$

其中，p、q定义为：

$$p1=-\Delta x, \quad q1=x1-xmin$$
$$p2=\Delta x, \quad q2=xmax-x1$$
$$p3=-\Delta y, \quad q3=y1-ymin$$
$$p4=\Delta y, \quad q4=ymax-y1$$



梁友栋算法的过程：如下

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

代码解析：本部分功能通过求鼠标拖动的裁剪框和直线的交点。根据交点决定是重绘还是舍弃。

```
if(u1>u2) //直线完全在裁剪窗口外，移除直线
{
    removeItem(cut_item);
}
else   //直线在裁剪窗口内存在，根据交点重绘直线即可
{
    cut_item->points[0].rx()=x1+u1*deltax;
    cut_item->points[0].ry()=y1+u1*deltay;
    cut_item->points[1].rx()=x1+u2*deltax;
    cut_item->points[1].ry()=y1+u2*deltay;
    cut_item->update();
}
```

2.7 曲线的绘制算法

n次Bernstein基函数多项式形式：

$BEZ_{i,n}(u)=C(n,i)u^i(1-u)^{n-i}$,

其中：$C(n,i)=n!/[i!(n-i)!]$ $(i=0,1,\ldots,n)$

□ 非负性：

- $BEZ_{i,n}(u)\geq 0$ $(i=0,1,\ldots,n)$；

  $u\in(0,1)$时，$0< BEZ_{i,n}(u) <1$

□ 权性：

$$\sum_{i=0}^{n} BEZ_{i,n}(u)\equiv 1,\quad u\in(0,1)$$

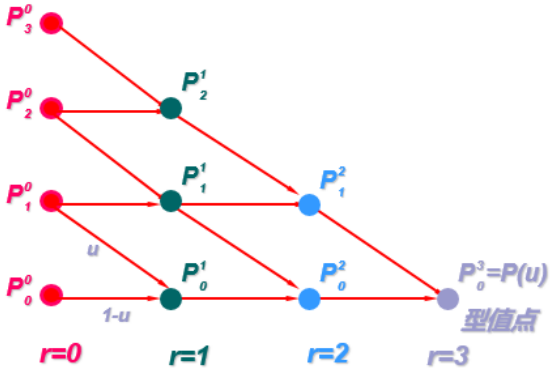
□ 对称性：

- $BEZ_{i,n}(u)=BEZ_{n-i,n}(1-u),(i=0,1,\ldots,n)$

德卡斯特里奥(de Casteljau)算法描述了从参数u计算n次Bézier曲线型值点$P(u)$的过程：

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

- 可以证明曲线上的型值点为：$P(u)=P_0^n$




```
1.  QPointF Casteljau(double u, int i, int r, QPointF*points, int point_num)
2.  {
3.      if(r==0)
4.      {
5.          return points[i];
6.      }
7.      else
8.      {
9.          return (1-u)*Casteljau(u,i,r-1,points,point_num)+u*Casteljau(u,i+1,r-1,points,point_num);
10.     }
11. }
12.
```

```
void paintbezier(QPainter *painter, QPointF*points, int point_num)//曲线
{
    if(point_num==1)
        return;
    else if(point_num==2)
    {
        paintline(painter,points[0].x(),points[0].y(),points[1].x(),points[1].y());
    }
    else if(point_num>2)
    {
        for(double u=0;u<=1;u=u+0.001)
        {
            painter->drawPoint(Casteljau(u, 0, point_num-1, points, point_num));
        }
    }
}
```

2.8 图片的保存

```
QString fileName;
fileName = QFileDialog::getSaveFileName(this, tr("保存文件
"), "", tr("Image Files (*.png *.jpg *.bmp)"));
if (!fileName.isNull()) {
    qDebug()<<"START";
    QImage Save=QImage(1260,540, QImage::Format_RGB32);
    Save.fill(Qt::white);
    QPainter painter;
    painter.begin(&Save);
    scene.render(&painter);
    painter.end();
    Save.save(fileName);
    qDebug()<<"END";
}
```

文件保存主要利用 render 函数，将 Scene 场景的图元全部渲染到 Image 里，并将 Image 保存至选定位置。

2.9 三维

OFF 文件格式：

顶点数 面片数 边数

x y z

x y z

…

顶点个数 N v1 v2 v3 … vn

顶点个数 M v1 v2 v3 … vm

所以利用 openmesh 读取 OFF 文件放在 mesh 中，在根据文件内容，用 opengl 绘制出来即可。