

# 计算机图形学实验 进度报告/系统报告

学号：171860635 姓名：陈振宇

## 10月30日报告

### 目前已经完成的内容：

1. 使用java语言和swing/awt框架，搭建了图形化界面；
2. 完全完成了命令解析模块，完成了基本的错误报告机制；
3. 为后续可能添加的鼠标操作功能做了准备；
4. 完成了DDA直线绘制算法和Bersenham直线绘制算法，在此之上完成了对应的两种多边形绘制算法；
5. 完成了图形平移功能，若当前画布上的所有图形都使用已经完成的算法，则可以完成平移操作；
6. 完成了对于 `<Integer,Shape>` 键值对的保存功能，用于实现平移/旋转/缩放时的重绘。

### 框架/图形化界面相关内容：

目前的代码调用可以分为3个层次：Main->MyMouseListener->MyGraphics，“->”代表调用。其中，Main类中包含了main函数和层次调用信息，MyMouseListener继承了MouseListener，主要包含的功能是命令解析和鼠标操作解析。MyGraphics中包含了主界面中的JFrame，JPanel和对应JPanel的Graphics，主要完成了绘图逻辑。从下面这段main函数中的代码可以看出层次调用关系，其中g为 `p.getGraphics()`，p为 `JPanel`，f为 `JFrame`；

```
MyGraphics mg=new MyGraphics();
mg.init(g,p,f);
MouseListener mml=new MyMouseListener();
mml.init(mg);
p.addMouseListener(mml);
```

考虑到画板大小不会超过1000x1000，主界面的分辨率被设置为1280x1024，且不可更改。

### 命令解析模块相关内容：

命令解析模块完成的功能很容易理解，鉴于所需要实现的命令不多且后续不会更改，在这里没有使用表驱动而是使用了普通的 `if{}else{}`  进行解析。命令解析模块每次将文件中的一行读入 `String` 中，再将这个 `String` 通过 `split()` 分解为 `String[] token`。

```

BufferedReader in=new BufferedReader(new FileReader(path));
String str;
while((str=in.readLine())!=null)
{
    commands.add(str);
}

```

```

String[] token=commands.get(i).split(" ");

```

为了防止读取过程中代码的错误和自己完成的测试用例中产生的错误，在命令解析模块中实现了简单的报错机制。这个报错机制主要处理 `token` 中的数量错误，在词组数量不正确时打印错误信息和命令文件中产生错误的行号，同时抛出一个 `Exception`。

```

private void printError(String[] token,int len,int line)
{
    try {
        if (token.length != len) {
            System.out.println("Error @ MyMouseListener @ resolveCommands: invalid command @ lir
            throw (new Exception());
        }
    } catch (Exception e)
    {
        e.printStackTrace();
    }
}

```

## 算法模块相关内容：

算法部分在 `MyGraphics` 模块中完成。考虑到现有的框架不提供对于像素的操作，但是 `BufferedImage` 提供对于像素的操作，同时 `Graphics` 类中提供了显示 `BufferedImage` 的简洁接口，`ImageIO` 类中提供了保存 `BufferedImage` 为各种格式的接口，所以采用的思路是在 `BufferedImage` 中绘制需要的图片，在需要时与 `JPanel` 同步，同时保存时仅涉及对于 `BufferedImage` 的保存。

```

public void save(String name)
{
    print("save");
    //保存为bmp格式
    try{
        ImageIO.write(img,"bmp",new File(name+".bmp"));
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}

private void panelSync() {originalGraphics.drawImage(img,0,0,null);}

private void drawPixel(int x,int y)
{
    int rgb_int=originalGraphics.getColor().getRGB();
    img.setRGB(x,y,rgb_int);
    //panelSync();//效率太低。
}

```

项目中同时还包括了图形的保存功能。初步考虑，想要保存图形有两种方式：第一种方式是基类+派生类；第二种方式是基类+动态数组。这里采用了第二种方式。**Shape** 类中保存了各个图形的类型，颜色，ID，使用的算法信息。在需要的时候调用相应的函数进行保存。

```

public int id;
public int type;
public Point[] location;
public Color color;
public String algorithm;

```

其中**Point**类中保存两个 **int** 。

## 算法相关内容：

DDA直线算法相比Bresenham直线算法较为简单。对于特殊情况的直线，如垂直于**x**轴或**y**轴的直线，要单独做特殊处理。对于其他所有情况下的直线，考虑直线的斜截式：

$$y = kx + b$$

先考虑 $|k| < 1$ 的情况。在这种情况下，以**x**作为算法的循环体。直线的斜率可以用另一种形式表达：

$$k = \frac{\Delta y}{\Delta x}$$

在 $\Delta x = 1$ 的情况下，也就是一般化DDA算法中循环体的步长， $\Delta y = k$ ，所以在每次循环中要将 $y$ （作为一个double变量）加上 $k$ 。同理，在 $|k| \geq 1$ 的情况下，以 $y$ 作为循环体，每次循环中 $x$ （作为一个double变量）加上 $\frac{1}{k}$ 即可。每次取点时，利用 `Math.round()` 操作，将 $x$ 或者 $y$ 四舍五入取整即可完成画点操作。

Bresenham算法相较而言更为复杂。除去两种特殊情况之外，算法整体分为8种情况，最终可以归类为2种情况，方便代码完成。两种特殊情况同DDA直线算法中的特殊情况，分别是直线垂直于x轴和直线垂直于y轴，这两种情况都很好完成。

下面给出一种泛用式Bresenham算法的推导过程，只需要考虑两种情况即可。先考虑第一种情况，直线的斜率绝对值在 $(0, 1]$ 中。

设需要画线的两个点分别是 $(x_b, y_b)$ 和 $(x_e, y_e)$ 。其中b和e分别代表着begin和end。设通过算法画出的第*i*个点为 $(x_i, y_i)$ ，其中*i*从0开始计数。

定义 $\Delta x = x_e - x_b$ ， $\Delta y = y_e - y_b$ 。定义 $x_s$ 和 $y_s$ ，对应代码中的 `sx` 和 `sy`，代表x和y的步长，取值为1或-1。当 $x_e > x_b$ 时， $x_s = 1$ ，否则为-1。 $y_s$ 同理。

设决策参数为 $T$ ， $T$ 的定义将会在之后完成。定义所需要解决的问题A：已知 $(x_i, y_i) = (x_t, y_t)$ ，以及 $T_i$ ，其中 $T_i$ 是用来决策第*i* + 1个点的决策参数。现在要求根据Bresenham直线算法求 $(x_{i+1}, y_{i+1})$ 和 $T_{i+1}$ 。

决策参数应该如何规定？假设现在我们要在两个整数点之间做决策，两个整数点的y坐标分别是 $y_{bias}$ 和 $y_{same}$ ，定义 $d_1 = y_{bias} - y$ ， $d_2 = y - y_{same}$ ，其中 $y$ 为在这个横坐标处真正的y值。一个简单易懂的定义是 $T = |\Delta x|(|d_2| - |d_1|)$ ，这样当 $T > 0$ 时，显然要选择 $d_1$ 对应的点，也就是 $y_{bias}$ 。当 $T \leq 0$ 时，选择另一个点。这是对本程序中采用算法的决策参数的一个直观的理解。

下面给出 $T$ 的严格定义： $T_{i+1}$ 为算法中预测 $(x_{i+2}, y_{i+2})$ 的决策参数。 $T_{i+1} = |\Delta x|(|d_2| - |d_1|)$ ，其中 $d_1 = y_{bias} - y_{real}$ ， $d_2 = y_{real} - y_{same}$ ， $y_{bias} = y_{i+1} + y_s$ ， $y_{same} = y_{i+1}$ ， $y_{real} = k(x_{i+1} + x_s) + b$ 。

$y_{bias}$ 和 $y_{same}$ 可以理解为与前一个点（也就是 $(x_{i+1}, y_{i+1})$ ，这个点在算法中要计算 $T_{i+1}$ 的时候是已知的）在同一行的坐标和“偏差一行”的坐标。 $y_{real}$ 是所预测点的准确y坐标。根据上一段，在 $T_{i+1} > 0$ 时，我们选择 $(x_{i+1} + x_s, y_{bias})$ 这个点，否则选择 $(x_{i+1} + x_s, y_{same})$ 。

$T_{i+1}$ 之所以用*i* + 1作为下标而不是用*i* + 2作为下标，是因为 $T_{i+1}$ 使用了 $(x_{i+1}, y_{i+1})$ 的信息作为判断依据。在不使用迭代计算的时候，使用上一个点的信息和直线方程信息就可以判断出下一个点的位置。根据定义很容易完成对于 $T_{i+1}$ 的计算。绝对值是一个难以处理的地方，但是好在我们已经定义了 $x_s$ 和 $y_s$ ，可以将这两个值与许多值相乘得到绝对值。 $d_1$ 和 $d_2$ 的符号（可以理解为“走向”）和 $\Delta y$ 是相同的（考虑 $y_s$ 的定义），同时 $\Delta x$ 和 $x_s$ 的符号也是相同的。这样可以将 $T_{i+1}$ 表示为：

$$T_{i+1} = (\Delta x x_s)(d_2 y_s - d_1 y_s) = (\Delta x x_s)(d_2 - d_1) y_s$$

考虑到 $k = \frac{\Delta y}{\Delta x}$ ，上式可以被化简为

$$T_{i+1} = 2x_sy_s\Delta yx_{i+1} - 2x_sy_s\Delta xy_{i+1} + C$$

其中 $C$ 是一个常数，这个常数将会在接下来的迭代运算中消除。

考虑到 $\Delta xx_s = |\Delta x|$ ， $\Delta yy_s = |\Delta y|$ ：

$$T_{i+1} = 2y_s|\Delta x|x_{i+1} - 2y_s|\Delta x|y_{i+1} + C$$

进行迭代运算：

$$T_{i+1} - T_i = 2|\Delta y| - 2y_s|\Delta x|(y_{i+1} - y_i)$$

这样就得出了迭代表达 $T$ 的公式。

若 $y_{i+1} = y_i$ ，即 $T_i < 0$ 时（ $T_i$ 和 $y_{i+1}$ 可以互相进行判断）：

$$T_{i+1} = T_i + 2|\Delta y|$$

否则

$$T_{i+1} = T_i + 2|\Delta y| - 2y_s^2|\Delta x| = T_i + 2|\Delta y| - 2|\Delta x|$$

现在迭代公式已经完成了，只剩下一个初始值 $T_0$ 。鉴于之前的公式中常数都被 $C$ 隐藏了，现在从原始公式开始计算。将 $(x_0, y_0)$ 代入原始公式即可，其中 $b$ 利用 $(x_0, y_0)$ 消去。

$$T_0 = \Delta xx_sy_s(2y_{real} - y_{same} - y_{bias})$$

代入计算之后：

$$T_0 = 2x_s|\Delta y| - |\Delta x|$$

此时整个算法完成。

对于剩下一种情况，将上述公式中所有的 $x$ 和 $y$ 互相替换即可。

## 11月30日报告

目前已完成的内容：

1. 这次提交的重心在于图形化界面方向，预计下次提交完成所有算法；
2. 基本完成了鼠标交互的图形化界面，并按照规定与之前的命令行交互程序做了区分。现在整个项目中同时存在两个 `main` 函数，分别表示命令行交互（`main`）和图形界面交互（`main_GUI`）；
3. 算法方面，完成了椭圆的绘制算法；
4. 对于图形化界面，完成了：
  1. 图片读取功能，目前仅支持png格式，但是得益于java强大的框架，想要支持其他格式只需要添加几行代码；
  2. 铅笔功能，实现这功能需要一些小技巧；
  3. 拖动绘制直线功能。类似于windows画图，不是在松开鼠标按键的时候绘制而是在拖动鼠标的时候实时地执行擦除-绘制操作，剩下的功能也都可以进行相同的操作；
  4. 拖动绘制矩形功能，相当于是拖动直线绘制功能的封装；
  5. 拖动绘制椭圆功能，调用椭圆绘制算法；
  6. 橡皮擦功能，提供一个固定大小的正方形作为画板的橡皮擦；
  7. 颜色更改功能，随意设置所需的颜色。目前只有黑白和RGB纯色可选，后续将添加选取任意颜色的方法，输入RGB值或在RGB色图上选取点即可选取新的颜色；
  8. 动态调整画板大小功能，与10月份提交的版本不同，现在画板大小可以任意调整，且在调整时不会发生画板被清零的情况，但是由于拖动从画板上消失的像素点不会重新出现；
  9. 鼠标拖动的边界控制功能，使得鼠标无论怎么拖动画板上的图形都不会出界。

## 图形化界面与鼠标交互相关内容：

1. 考虑到项目需要同时接收鼠标按下操作，抬起操作，与拖动操作，所以 `MyMouseListener` 类同时继承了 `MouseMotionListener` 和 `MouseListener` 的接口；

```
public class MyMouseListener implements MouseMotionListener,MouseListener
```

2. 使用了一个新的类 `LayoutInitializer` 用于处理按钮与逻辑的绑定。按钮的生成在画布显示之前，而鼠标侦听器的生成在画布显示之后，所以按钮的绑定在画布的生成之后。这个顺序是唯一需要注意的点；
3. 使用了java自带的 `JFileChooser` 生成窗口进行文件选择。使用这个窗口有利于跨平台。

```
JFileChooser jf=new JFileChooser();  
jf.setFileSelectionMode(JFileChooser.FILES_ONLY);  
jf.setMultiSelectionEnabled(false);  
jf.showOpenDialog(mml.mg.frame);  
File f=jf.getSelectedFile();
```

4. 对于铅笔的操作，直接在拖动侦测中画一个点是不行的，因为主循环的速度肯定要慢于鼠标移动一个像素的速度。解决方法就是在拖动侦测中绘制上一次记录的点A和鼠标所在点之间的直线，并把鼠标所在点赋给A；

```

@Override
public void mouseDragged(MouseEvent mouseEvent){
    //省略
    else if (penMode == MODE_PENCIL)
    {
        mg.drawPixelWrapper(mouseEvent.getX(),mouseEvent.getY());
        mg.drawPixelWrapper(x_begin,y_begin);
        mg.drawLineDDA(MyGraphics.ELEMENT,x_begin,y_begin,mouseEvent.getX(),mouseEvent.getY());
        x_begin=mouseEvent.getX();
        y_begin=mouseEvent.getY();
    }
    //省略
}

```

5. 如何实现在鼠标拖动过程中动态实现擦除-绘制功能是一个难点。在11月的提交中暂且采用最笨的方法，在按下鼠标的时候保存现在的图片，在拖动时将保存的图片写入画板并重新绘制新的图元。

```

if (penMode == MODE_LINE) {
    mg.setImage(backGround);
    mg.drawLineBresenham(MyGraphics.ELEMENT, x_begin, y_begin, mouseEvent.getX(), mouseEvent.getY());
    //已经同步
}

```

## 算法相关内容

本次完成了椭圆绘制算法。剩下的所有算法将会在下一次更新中实现，届时可以进行测试。

考虑一个中心点在原点的椭圆。将椭圆的标准方程化为一个参数方程：

$$f(x, y) = r_y^2 x^2 + r_x^2 y^2 - r_x^2 r_y^2$$

显然，当 $(x, y)$ 在椭圆上时， $f(x, y) = 0$ 。所以通过特殊点代入可知，当 $f(x, y) < 0$ 时，某点位于椭圆内部；当 $f(x, y) > 0$ 时，某点位于椭圆外部。 $f(x, y)$ 将作为算法的判断依据。

椭圆是轴对称和中心对称的，所以只需要画四分之一即可。算法中选取第一象限绘制。

通过画直线的经验，可以知道当线条在 $x$ 方向较为平缓时，应使用 $x$ 作为循环单位，否则使用 $y$ 作为循环单位。这在椭圆中形成了一个分割点，在分割点的两边分别要以 $x$ 和 $y$ 作为循环元。椭圆上顶点的 $|k| = 0$ ，右顶点的 $|k| = +\infty$ ，其中 $k$ 是椭圆切线的斜率。考虑到斜率变化是连续的，所以分割点左边（靠近上顶点）一定使用 $x$ 作为循环元，右边（靠近右顶点）一定使用 $y$ 作为循环元。同时在切割点处 $|k| = 1$ （ $k = -1$ ）。

现在考虑以 $x$ 为循环元的情况。

令决策参数为 $p$ 。考虑已经选取点 $(x_0, y_0)$ ，现有两个候选点 $(x_0 + 1, y_0)$ 和 $(x_0 + 1, y_0 - 1)$ 。这两个候选点的中点若在椭圆外部，则选取更接近椭圆外部的点 $(x_0 + 1, y_0)$ ，否则选择 $(x_0 + 1, y_0 - 1)$ 。所以选取 $p = f(x_0 + 1, y_0 - \frac{1}{2})$ 。当 $p > 0$ 时，中点更靠近椭圆外侧，所以选取候选点 $(x_0 + 1, y_0)$

，否则选取候选点 $(x_0 + 1, y_0 - 1)$ 。

考虑算法中的情况。当绘制某个点 $(x_0, y_0)$ 时， $p = f(x_0 + 1, y_0 - \frac{1}{2})$ ，因为在上一轮循环中，当前的点已经被确定，同时用于处理下一个点的 $p$ 也已经被计算得出。所以先根据 $p$ 选择一个点。当选择外侧的点时：

$$p_{after} - p_{before} = f(x_0 + 2, y_0 - \frac{1}{2}) - f(x_0 + 1, y_0 - \frac{1}{2})$$

经过化简之后：

$$p_{after} - p_{before} = 2r_y^2 x_0 + 3r_y^2$$

当选择内侧的点时：

$$p_{after} - p_{before} = f(x_0 + 2, y_0 - \frac{3}{2}) - f(x_0 + 1, y_0 - \frac{1}{2})$$

经过化简之后：

$$p_{after} - p_{before} = 2r_y^2 x_0 - 2r_x^2 y_0 + 2r_x^2 + 3r_y^2$$

这就得出了椭圆第一部分的递推公式。

现考虑以 $y$ 为循环元的情况。首先要确定一个新的 $p$ 。新的 $p$ 通过第一阶段最后的点得出，代入原公式 $p = f(x_0 + 1, y_0 - \frac{1}{2})$ 即可。

同理，考虑算法中的情况。当绘制点 $(x_0, y_0)$ 时， $p = f(x_0 + \frac{1}{2}, y_0 - 1)$ 。当 $p > 0$ 时，选取更接近外侧的点 $(x_0 + 1, y_0 - 1)$ ，否则选取更靠近内侧的点 $(x_0, y_0 - 1)$ 。

当需要选取 $(x_0 + 1, y_0 - 1)$ 时：

$$p_{after} - p_{before} = f(x_0 + \frac{3}{2}, y_0 - 2) - f(x_0 + \frac{1}{2}, y_0)$$

经过化简之后：

$$p_{after} - p_{before} = -2r_x^2 y_0 + 3r_x^2$$

当需要选取 $(x_0, y_0 - 1)$ 时：

$$p_{after} - p_{before} = f(x_0 + \frac{1}{2}, y_0 - 2) - f(x_0 + \frac{1}{2}, y_0)$$



经过化简之后：

$$p_{after} - p_{before} = -2r_x^2 y_0 + 3r_x^2 + 2r_y^2 x_0 + 2r_y^2$$

这样就完成了椭圆的绘制算法。

相比于直线，椭圆的算法反而更加简单一些，因为椭圆的算法只需要考虑一种情况，甚至对于 $r_x > r_y$ 情况和 $r_x < r_y$ 的情况是相同的。直线的算法反而要考虑8种情况及它们的合并化简。