《计算机图形学》6 月报告

李凯旭 171180589
（南京大学　　匡亚明学院计算机科学与技术方向）
联系方式：171180589@smail.nju.edu.cn

实验环境

依赖环境：

Anaconda 4.7.12
Python 3.7.4
Pillow 6.2.0
Numpy 1.16.5
Pyqt5 5.9.2

实验框架

所有文件及文件夹均在 source 文件夹下，按照以下方式排布：

Source/

cg_algorithm.py	核心算法模块，负责图元的生成与编辑
cg_clii.py	命令行处理程序（支持基础操作）
cg_gui.py	用户界面交互（支持部分额外功能）
icon/	用户界面图标的 png 文件，不建议修改
output/	输出文件夹

框架内容

核心算法模块（cg_algorithm.py）

1．线段绘制算法（draw_line（p_list, algorithm））

（1）参数和返回值

param p_list：　线段起始点与终点坐标$[\,[x_0,y_0]\,,[x_1,y_1]\,]$

param algorithm: 线段绘制算法，DDA 或 Bresenham

return ：绘制坐标的整数对序列 result: $[\,[x_0,y_0]\,,[x_1,y_1],\ldots\ldots,[x_n,y_n]\,]$

（2）算法描述

为了避免边界条件处理，当$x_0=x_1$时，直接按照$x=x_1$的方程绘制线段。其他线段采用 DDA 或 Bresenham 算法。此时，直线可以描述为$y=mx+b$

Ⅰ.DDA 算法【1】

根据$[x_0,y_0]$，$[x_1,y_1]$决定的直线方程$y=mx+b$，当$x\to x+1$时候，$y\to y+m$。因此，确定直线的斜率m之后，可以通过不断执行$x\to x+1$并计算对应y的方式求出直线上所有点的坐标。注意最后将得到的每个坐标都转化为y轴上最接近的整数坐标。

由于注意到当斜率m的绝对值$|m|>1$（即 $\mathrm{abs}(x_0-x_1)<\mathrm{abs}(y_0-y_1)$）时依靠 x 轴的步进得到的点数量较少。因此当$|m|>1$时依靠$y$轴的步进，执行$y\to y+1$计算$x$轴的步进 $\frac{1}{m}$ 求直线坐标。

注意到，当$|m| < 1$，$x_0 > x_1$或者$|m| > 1$，$y_0 > y_1$时，对应的步进需要设置为负数。

Ⅱ.Bresenham 算法【1】【2】

Bresenham 算法利用了输出设备坐标点均为整数的性质。

考虑$0 < m < 1$的情况。已知坐标(x_k, y_k)在直线上，考虑下一个在直线上的坐标(x_{k+1}, y_{k+1})。显然，$x_{k+1} = x_k + 1$。而y_{k+1}则只能选择y_k或$y_k + 1$。即判断$(x_k + 1, y_k)$与$(x_k + 1, y_k + 1)$哪一个更接近实际直线上点的坐标$(x_k + 1, y)$。

在【2】中，分别计算两个点与实际坐标的距离d_1与d_2：

$$d_1 = y - y_k = m(x_k + 1) + b - y_k$$

$$d_2 = y_{k+1} - y = y_k + 1 - m(x_k + 1) - b$$

得到$\triangle d = d_1 - d_2 = 2m(x_k + 1) - 2y_k + 2b - 1$。当$\triangle d > 0$时选择$(x_k + 1, y_k + 1)$，反之选择$(x_k + 1, y_k)$。

由于已知$m = (y_1 - y_0)/(x_1 - x_0) = \triangle y/\triangle x$，得到$p_k = \triangle x \triangle d = 2\triangle y x_k - 2\triangle x y_k + c$。其中，$p_k$的符号与$\triangle d$一致，可以根据$p_k$的符号决定第$k$个坐标后面一个坐标的选择，因此成为算法第$k$步的决策参数。

为了进一步简化计算，可以通过p_k计算p_{k+1}

$$p_{k+1} - p_k = 2\triangle y(x_{k+1} - x_k) - 2\triangle x(y_{k+1} - y_k) = 2\triangle y - 2\triangle x(y_{k+1} - y_k)$$

因此$p_k > 0$时，$p_{k+1} = p_k + 2\triangle y - 2\triangle x(y_{k+1} - y_k) = p_k + 2\triangle y - 2\triangle x$，$p_k < 0$时，$p_{k+1} = p_k + 2\triangle y - 2\triangle x(y_{k+1} - y_k) = p_k + 2\triangle y$

递推可以得到直线上所有点的坐标。

对于$m > 1$的情况，将x轴的步进 1 修改为y轴步进 1。对于 m 为负数的情况，先考虑$-1 < m < 0$。由于按照上面计算方法重新计算后，恰好发现$p_k = \triangle x \triangle d = -(2\triangle y x_k - 2\triangle x y_k) + c_2$。可以看到如果当$p_k > 0$时选择$(x_k + 1, y_k - 1)$，反之选择$(x_k + 1, y_k)$。之后考虑$m < -1$的情况。

因为按照上面计算过程比较复杂，实际代码采用【1】中描述的算法。

仍然考虑$0 < m < 1$。可以看到，除了直接计算$\triangle d = d_1 - d_2$，计算$y - \mathrm{y_k}$与 1/2 的大小关系同样可以判断点的选择。当$y - \mathrm{y_k} > \frac{1}{2}$时选择$y_k + 1$。反之选择$y_k$。可以看到，两种方式本质上没有差别，但是判断比原来要简便很多。

令$e = y - \mathrm{y_k} - \frac{1}{2}$，因此只需要根据$e$的符号判断点的选择。理想情况是，对于每一个$(x_k, y_k)$，他都在直线上，这样$e$只需要取直线斜率$m$即可。然而事实是，大部分情况下$(x_k, y_k)$并不在直线上。因此每选择一个坐标，都需要对$e$进行更新。如果前一步选择了$(x_k + 1, y_k)$，那么$e$在当前值上加斜率$m$。否则，如果前一步选择了$(x_k + 1, y_k + 1)$，那么更新选择了$e$时候需要加上$m$后减 1。

同样的，按照整数化的思想，将e乘以$2\triangle x$可以进一步简化计算的消耗。

这里选择【1】中的整数化方式。

对于全情况的扩展与上面类似，不再赘述。

2. 多边形绘制算法（draw_polygon（p_list, algorithm））

（1）参数和返回值

param p_list： 多边形顶点序列$[\,[x_0, y_0]\,, [x_1, y_1], \ldots\ldots, [x_n, y_n]\,]$，需要沿边界按顺时针或逆时针顺序

param algorithm: 多边形绘制算法，DDA 或 Bresenham

return ： 绘制坐标的整数对序列 result: $[\,[x_0, y_0]\,, [x_1, y_1], \ldots\ldots, [x_m, y_m]\,]$

（2）算法描述:

对多边形每一条边执行 DDA 或 Bresenham 算法，不再赘述。

3. 椭圆绘制算法（draw_ellipse（p_list））

（1）参数和返回值

param p_list :　 椭圆的外切矩形左上角与右下角坐标[$[x_0, y_0]$, $[x_1, y_1]$],

return ： 绘制坐标的整数对序列 result: [$[x_0, y_0]$, $[x_1, y_1]$, …… , $[x_m, y_m]$]

（2）算法描述:

采用中点圆算法。中点原算法本质上是 Bresenham 直线算法的扩展。

首先，根据平移性质与椭圆的对称性，我们可以只考虑中心为原点的椭圆在第一象限的情况。此时，在椭圆上的每一个点切线的斜率m均为负数。假设椭圆方程为$\frac{x^2}{a^2}+\frac{y^2}{b^2}=1$。

已知坐标(x_k, y_k)在直线上的情况下，通过画图可以发现，下一个点(x_{k+1}, y_{k+1})为(x_k+1, y_k)或(x_k+1, y_k-1)。

由于这个时候计算两个候选点与椭圆上实际点的距离会比较麻烦，所以采用中点判定的方法。(x_k+1, y_k)与(x_k+1, y_k-1)的中点 P 为$(x_k+1, y_k-\frac{1}{2})$。若点 P 在椭圆内，选择(x_k+1, y_k)。反之选择(x_k+1, y_k-1)。因为判断 P 是否在椭圆内只需要判断点 P 处$\frac{x^2}{a^2}+\frac{y^2}{b^2}-1$与 0 的关系，所以计算被简化。

注意到，当$m<-1$时x轴的总距离较小，这时固定y的步进为 1。反之，若$-1<m<0$，固定x轴的步进为 1。

通过对称得到椭圆中心在原点时椭圆上所有点的坐标。通过平移得到实际以$[(x_0+x_1)/2, (y_0+y_1)/2]$为中心的椭圆上所有点的坐标。

4．圆绘制算法（draw_circle（p_list））

（1）参数和返回值

param p_list :　 圆的矩形框左上角与右下角坐标[$[x_0, y_0]$, $[x_1, y_1]$],

因为圆实际上是正方形包围框，所以在绘制圆的时候认为直径是x轴上的距离。也就是说允许$x_1-x_0 \neq y_0-y_1$，但是此时只有x_0, x_1, y_0三个点有效，y_1取值无效。

return ： 绘制坐标的整数对序列 result: [$[x_0, y_0]$, $[x_1, y_1]$, …… , $[x_m, y_m]$]

（2）算法描述:

采用上述的椭圆算法。不再赘述。

添加圆算法的目的主要是为了方便后面 Gui 程序中描述控制点坐标顺便加的功能，所以没有单独考虑圆的绘制算法。

5．曲线绘制算法（draw_curve（p_list，algorithm））

（1）参数和返回值

param p_list :　 曲线控制线坐标序列[$[x_0, y_0]$, $[x_1, y_1]$, …… , $[x_m, y_m]$]

param algorithm: 曲线绘制算法，Bezier 或 B-spline（仅限三次均匀 B 样条曲线）

return ： 绘制坐标的整数对序列 result: [$[x_0, y_0]$, $[x_1, y_1]$, …… , $[x_m, y_m]$]

（2）算法描述:

Ⅰ.Bezier 曲线【3】

贝塞尔曲线的思想在于使用参数方程。

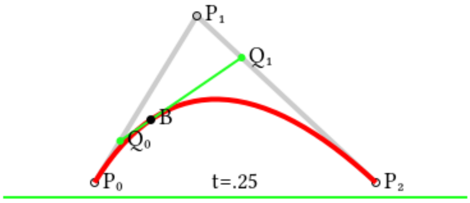
对于一条线段而言，除了$y=mx+b$的描述方式外，还有参数方程的描述。假设线段顶点为P_0, P_1，对于线段上的点P，如果确认$P_0P/P_0P_1 = t$，点P的坐标可以描述为$P = P_0+t(P_1-P_0)=(1-t)P_0+tP_1$。t ∈[0, 1]。可以看到，此时有两个控制点，t的最高阶为 1，因此直线称为一阶贝塞尔曲线。

同样的，考虑三个控制点的情况。假设控制点为P_0，P_1，P_2。首先确定参数t的数值。对于线段P_0P_1，按照参数t，按照一阶贝塞尔曲线绘制出Q_0点。同样的，对于P_1P_2，也使用参数t按照一阶贝塞尔曲线绘制出Q_1点。这样，三个点控制的二阶贝塞尔曲线转化为了Q_0，Q_1两个顶点控制，参数为t的一阶贝塞尔曲线。

计算得到贝塞尔曲线的方程：

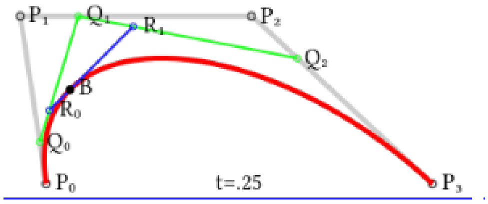
$P = (1-t)Q_0 + tQ_1 = (1-t)\big((1-t)P_0 + tP_1\big) + t\big((1-t)P_1 + tP_2\big) = (1-t)^2P_0 + 2t(1-t)P_1 + t^2P_2$

可以看到，参数t的最高阶数为 2，所以称为二阶贝塞尔曲线：



二次贝塞尔曲线的结构

同样的，四个控制线P_0，P_1，P_2,P_3可以实现三阶贝塞尔曲线：



三次贝塞尔曲线的结构

为了方便后面的计算，这次直接观察最后一步：由R_0，R_1决定曲线上的点B。可以看到，R_0点实际上是在P_0,P_1,P_2决定的二阶贝塞尔曲线上，而R_1点则在P_1,P_2,P_3决定的二阶贝塞尔曲线上。因此，对三阶贝塞尔曲线的计算转化为两个两个贝塞尔曲线的计算。

这可以一直扩展到 n 阶贝塞尔曲线。因此，n+1 个顶点控制的 n 阶贝塞尔曲线可以转化为 2 个由 n 个顶点控制的 n-1 阶贝塞尔曲线。假设由P_0，P_1，P_2,⋯P_n这 n+1 个顶点和参数 t 控制的贝塞尔曲线为$\mathrm{B}_{\mathrm{P_0P_1\ldots P_n}}$（t）。通过推导可以得到：

$\mathrm{B}_{\mathrm{P_0P_1\ldots P_n}}(\mathrm{t}) = t\big(\mathrm{B}_{\mathrm{P_0P_1\ldots P_{n-1}}}(\mathrm{t})\big) + (1-t)\big(\mathrm{B}_{\mathrm{P_1\ldots P_n}}(\mathrm{t}) - \mathrm{B}_{\mathrm{P_0P_1\ldots P_{n-1}}}(\mathrm{t})\big) = (1-t)\mathrm{B}_{\mathrm{P_0P_1\ldots P_{n-1}}}(\mathrm{t}) + t\mathrm{B}_{\mathrm{P_1\ldots P_n}}(\mathrm{t})$

根据这个表达式推导可以得到，$\mathrm{B}_{\mathrm{P_0P_1\ldots P_n}}(\mathrm{t})$的表达式为

$\mathrm{B}_{\mathrm{P_0P_1\ldots P_n}}(\mathrm{t}) = \sum_{i=0}^{n} B_{i,n}(t)\,Pi = \sum_{i=0}^{n} B_{i,n}(t)\,Pi = \sum_{i=0}^{n} C_n^i(1-t)^{n-i}t^i\,Pi \quad t \in [0,1]$

因此，给定控制点的坐标后，只需要按照方程计算就可以了。注意好对参数t设计合适的步进以防止出现线段断裂的情况。这里我选择了由控制点构成的多边形周长的倒数作为t的步进，因为在x不减的情况下曲线长度应该不会超过控制多边形的周长。但是如果控制点的x坐标突然大幅减小时还是会有一点瑕疵。

Ⅱ.B 样条曲线【4】

B 样条曲线是对于贝塞尔曲线的扩展。在贝塞尔曲线中，我们使用了$\mathrm{B}_{\mathrm{P_0P_1\ldots P_n}}(\mathrm{t}) = \sum_{i=0}^{n} B_{i,n}(t)\,Pi$来表示曲线。这个式子既可以理解为一个参数 t 的 n 阶多项式，但是也可以将多项式认为是为控制点的参数，这样可以更好体现控制点对于曲线的影响。按照同样的思想，B 样条曲线可以表示为$P = \mathrm{P}_{\mathrm{P_0P_1\ldots P_n}}(\mathrm{t}) = \sum_{i=0}^{n} N_{i,k}(t)\,Pi$。这样，只需要定义合适所谓的基函数$N_{i,k}(t)$就可以得到方程。（定义域为$[t_{k-1}, t_{n+1}]$，在贝塞尔中是[0, 1]。阶数 k 是额外定义，与 n 无关）。

根据 deBoor-Cox 的递推公式，基函数$N_{i,k}(t)$的定义如下：

$$N_{i,1}(t) = \begin{cases} 1 & if\ t \in [t_i, t_{i+1}) \\ 0 & otherwise \end{cases}$$

对于 k > 1 时，有如下的递推式：

$$N_{i,k}(t)=\left(\frac{t-t_i}{t_{i+k-1}-t_i}\right)N_{i,k-1}(t)+\left(\frac{t_{i+k}-t}{t_{i+k}-t_{i+1}}\right)N_{i+1,k-1}(t)$$

对于一阶样条可以看到，对于每一个 i，基函数$N_{i,1}(t)$只在$[t_i,t_{i+1}]$上有定义。而且特殊的是，此时每个顶点对应的基函数有数值的区间是互相不重叠的。

对于二阶样条则有：

$$N_{i,2}(t)=\frac{t-t_i}{t_{i+1}-t_i}N_{i,1}(t)+\frac{t_{i+2}-t}{t_{i+2}-t_{i+1}}N_{i+1,1}(t)=\begin{cases}\frac{t-t_i}{t_{i+1}-t_i} & t_i\le t<t_{i+1}\\ \frac{t_{i+2}-t}{t_{i+2}-t_{i+1}} & t_{i+1}\le t<t_{i+2}\end{cases}$$

因为后面的太复杂就不推导了。在视频中当初取点时候有些分散，后来修改后连续性就比较好了。

可以看到，对于 k 阶的 B 样条，他的每一个基函数就需要 k 个区间（从t_i到t_{i+k}）来决定。而总共有 n+1 个控制点，考虑到重合的控制区间，总共需要 n+1+k 个控制区间，并且是一个 k-1 次的函数。在实际写代码时，还是选择采用递归的方式。因为递归思想简单代码容易写。并且在实际中，因为只要解决 3 次曲线，递归次数有限。而控制区间 ti 为了简便，直接选择了固定 1 到 10 范围内的 n+k+1 个区间。

可以看到，贝塞尔曲线和 B 样条曲线的定义其实本质一致，都是$\mathrm{B}_{\mathrm{P_0P_1\ldots P_n}}(\mathrm{t})=\sum_{i=0}^{n}B_{i,n}(t)\,Pi$。他们的区别就在于选择了不同的基函数。贝塞尔曲线的优势在于数学原理容易理解，构造也比较方便。但是缺陷在于曲线的整体性太强，一个控制点的移动会对整条曲线都会产生影响，所以在实际使用的时候很难对局部的误差进行调整。使用分段贝塞尔曲线又容易出现接口不连续。而 B 样条曲线的弥补了贝塞尔整体性过强的弱点。对于每一个控制点，他只能修改 k+1 个区间内 t 多项式的数值。但是，他的基函数构造却很麻烦，在数学上不像贝塞尔曲线直观，并且在参数步进不好设计，容易出现曲线过于分散的情况，所以对于设计困难，但是对于使用者确实方便了很多。

6. 平移算法（translate（p_list, dx, dy））【5】

（1）参数和返回值

param p_list： 图元参数坐标序列[$[x_0,y_0]$, $[x_1,y_1]$, ……, $[x_n,y_n]$]

dx：水平方向平移量

dy：竖直方向平移量

return ： 平移后图元坐标的整数对序列 result: [$[x_0,y_0]$, $[x_1,y_1]$, ……, $[x_m,y_m]$]

（2）算法描述：

将图元每一个坐标$[x_k,y_k]$修改为$[x_k+dx,y_k+dy]$，不作说明。

7. 旋转算法（rotate（p_list, x, y, r））【5】

（1）参数和返回值

param p_list： 图元参数坐标序列[$[x_0,y_0]$, $[x_1,y_1]$, ……, $[x_n,y_n]$]

x：旋转中心 x 坐标

y：旋转中心 y 坐标

r：顺时针旋转角

return ： 平移后图元坐标的整数对序列 result: [$[x_0,y_0]$, $[x_1,y_1]$, ……, $[x_m,y_m]$]

（2）算法描述：

首先考虑旋转中心在原点的情况。因为一般的习惯是以逆时针旋转为正，这里解释的时候采用的是逆时针。

解释时需要使用极坐标。假设原坐标$[x,y]$，旋转中心$[0,0]$，向量$[x,y]$与x轴正半轴方向单位向量$[1,0]$夹角为Θ，线段长度为d。那么坐标$[x,y]$可以描述为$[dcos\Theta,dsin\Theta]$。逆时针旋转角度r后坐标转化为

$[dcos(\Theta+r), dsin(\Theta+r)]$

$$dcos(\Theta+r) = dcos\Theta cosr - dsin\Theta sinr = xcosr - ysinr$$
$$dsin(\Theta+r) = dsin\Theta cosr + dcos\Theta sinr = xsinr + ycosr$$

所以转化后坐标为$[xcosr - ysinr, xsinr + ycosr]$。

如果旋转中心$[x_0, y_0]$不为$[0,0]$。平移后实际坐标为:

$x' = x_0 + (x - x_0)cosr - (y - y_0)sinr$

$y' = y_0 + (x - x_0)sinr + (y - y_0)cosr$

由于代码中是顺时针旋转角为r，实际上就是逆时针旋转角为$-r$。即:

$x' = x_0 + (x - x_0)cosr + (y - y_0)sinr$

$y' = y_0 - (x - x_0)sinr + (y - y_0)cosr$

注意计算三角函数值时转化为弧度制。

8. 缩放算法（scale（p_list, x, y, s））【5】

（1）参数和返回值

param p_list : 图元参数坐标序列[$[x_0, y_0]$, $[x_1, y_1]$, …… , $[x_n, y_n]$]

x：缩放中心 x 坐标

y：缩放中心 y 坐标

s：缩放中心

return ： 平移后图元坐标的整数对序列 result: [$[x_0, y_0]$, $[x_1, y_1]$, …… , $[x_m, y_m]$]

（2）算法描述:

首先考虑缩放中心在原点的情况。

假设原坐标$[x, y]$，缩放中心$[0,0]$，s_x为x方向缩放倍数，s_y为y轴方向缩放倍数。

$$x' = s_x x$$
$$y' = s_y y$$

所以转化后坐标为$[s_x x, s_y y]$。

如果旋转中心$[x_0, y_0]$不为$[0,0]$，先将缩放中心平移至$[0,0]$进行缩放，之后再将缩放后的图像至原处。

$x' = s_x(x - x_0) + x_0 = xs_x + x_0(1 - s_x)$

$y' = s_y(y - y_0) + y_0 = ys_y + y_0(1 - s_y)$

9. 线段裁剪算法（clip（p_list, x_min, y_min, x_max, y_max, algorithm））

（1）参数和返回值

param p_list : 线段端点坐标[$[x_0, y_0]$, $[x_1, y_1]$]

x_min：裁剪窗口左下角 x 坐标

y_min：裁剪窗口左下角 y 坐标

x_max：裁剪窗口右上角 x 坐标

y_max：裁剪窗口右上角 y 坐标

algorithm：裁剪算法，包括 Conhen-Sutherland 与 Liang-Barsky

return ： 裁剪后线段端点坐标 result: [$[x_0', y_0']$, $[x_1', y_1']$]

（2）算法描述:

为了方便起见，$x_0 = x_1$或者$y_0 = y_1$都是单独处理的。(主要是防止除 0，后来发现实际上 Liang-Barsky 本来应该是没有这个问题的，但是起初实现的时候在为 0 的情况没有及时返回，导致后面默认系数为正的条件不成立，出现了除 0 的情况)

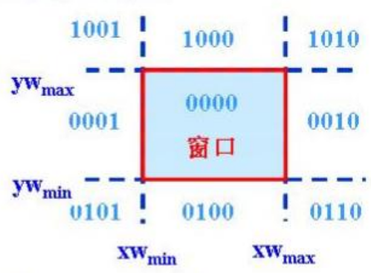
若$x_0 = x_1$，只有当$x_{min} \le x_0 \le x_{max}$时有可能需要裁剪，否则直接丢弃。这里定义$y_0 < y_1$。如果有$y_0 > y_{max}$或者$y_1 < y_{min}$时依然直接丢弃。否则，$y_0 = \max(y_0, y_{min})$，$y_1 = \min(y_1, y_{max})$（分情况画图就

懂了，不做说明）

同样，若$y_0 = y_1$，只有当$y_{min} \le y_0 \le y_{max}$时有可能需要裁剪，否则直接丢弃。这里定义$x_0 < x_1$。如果有$x_0 > x_{max}$或者$x_1 < x_{min}$时依然直接丢弃。否则，$x_0 = \max(x_0, x_{min})$，$x_1 = \min(x_1, x_{max})$

Ⅰ.Cohen-Sutherland：【6】【7】

以窗口四条边所在直线为分割线，整个界面分为 9 个不同的区域：



对 9 个不同区域进行编码。编码的设计思想是，从最低位至最高位依次表示左，右，下，上。在窗口最左侧的左侧，最低位置 0，反之置 1。意义在于，如果某点最低位为 1，该点在左边界的左侧，必定会在某次切割时被最左侧的边界切割掉。同样的方法可以得到每个区域其他三位的编码。

对于所有的直线，都可以按照左边界、右边界、下边界、上边界的顺序使用四条直线进行切割。

经过尝试可以看到，以下两种情况是容易判断的：

（1） 两个端点对应的区域值都是 0。

如果线段两个端点都在窗口内，那么就完全不需要裁剪直接选取原线段。这时候两个端点对应的区域值都是 0。反之，如果整个线段最后都能在窗口内输出，两个端点对应的区域值一定得都是 0 才行。

（2） 在某一位上，两个端点均为 1。

以最低位为例，这说明两个端点都在窗口左边界的左侧。无论怎样这条直线必定会在被左边界切割时完全舍弃。

注意到，任意一条直线，裁剪后线段如果在窗口中可以输出，此时必定满足（1）。否则，必定在某一步中满足（2），然后被直接丢弃。

那么对于普通直线的算法就清晰了：按照左边界、右边界、下边界、上边界的顺序使用四条直线进行切割。每次切割后判断线段是否满足（1）或（2），满足则直接简取或简弃，否则继续切割。

在实际代码中没有按照这种顺序判断四遍。因为如果按顺序写的话相当于一份代码写了四遍。所以这里注意到。如果一条线段不能判断简取简弃，那么至少有一个顶点是在窗口 0000 外且两个顶点的区域码不同。由于区域码较大的那个一定是在区域外（因为肯定不为 0），针对区域码大的进行判断，分为四种情况（左右下上）裁剪并更新端点的区域码，这样子只要做一个 while 循环，并且每次循环中只要根据区域码更新一个端点及对应区域码。因为选中顶点区域码只要不为 0，每次必定会变小；为 0 则直接退出，最后不会出现死循环的情况。只需要定一个 flag 变量，当 flag 确认最后一步是简取而不是简弃的时候将两个点坐标输出就行。

Ⅰ.Liang-Barsky：【8】

该算法使用了参数方程。因为个人觉得图像解释有够复杂化，而且对写代码意义不大就不解释了。

$(x, y) = (x_0 + t * (x_1 - x_0), y_0 + t * (y_1 - y_0)) = (x_0 + t * \triangle x, y_0 + t * \triangle y) \quad t \in [0,1]$

如果直线上某一点(x, y)在裁剪窗口内，则有：

$x_{min} \le x \le x_{max}$，$y_{min} \le y \le y_{max} \rightarrow x_{min} \le x_0 + t * \triangle x \le x_{max}$，$y_{min} \le y_0 + t * \triangle y \le y_{max}$

所以整个裁剪过程在代数上可以被描述为六个不等式：

$$-\triangle x * t \le x_0 - x_{min}$$
$$\triangle x * t \le x_{max} - x_0$$

$$-\triangle y * t \le y_0 - y_{min}$$
$$\triangle y * t \le y_{max} - y_0$$

$$t \ge 0$$
$$t \le 1$$

这六个不等式中只有t一个变量，只需要根据最后t的范围确定裁剪后的两个端点位置就可以。相对于 Cohen-Sutherland 的方法要好写很多，只需要注意分母不要为 0 就可以。

10. 多边形填充算法（scan_fill_polygon（p_list））【9】

（1）参数和返回值

param p_list : 多边形图元参数坐标序列$[\,[x_0, y_0]\,, [x_1, y_1], \ldots\ldots, [x_n, y_n]\,]$

return ： 填充区域所有点坐标的整数对序列 result: $[\,[x_0, y_0]\,, [x_1, y_1], \ldots\ldots, [x_m, y_m]\,]$

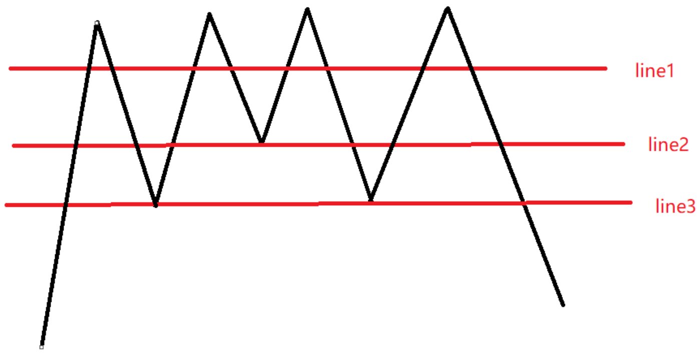
（2）算法描述:

使用较为原始的扫描填充算法，原因是实现比较容易。个人认为区域种子填充是更好的算法，但是需要额外实现非矩形的裁剪框点的区域判断。

首先，根据多边形图元参数坐标序列$[\,[x_0, y_0]\,, [x_1, y_1], \ldots\ldots, [x_n, y_n]\,]$选取出$y_{min}$和$y_{max}$。将$y_y$从$y_{min}$增大到$y_{max}$，对每一条直线$y = y_y$判断直线上上哪些点在多边形的内部。

对于凸多变形而言，直线与多边形至多有两个交点，只要多边形每条线段都与直线求交就可以了。因为对 python 语言不熟悉就没有使用更快速的活性列表方法。

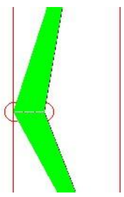
然后考虑凹多边形。，随便画个图片简化理解:



可以看到，如果认为每一个直线与多边形端点相交时交点两个坐标一样的点。将直线与多边形边界的所有交点的坐标按照x轴坐标的由小到大排序，那么每相邻两顶点作一组，可以看到，每组组内的两个点相连得到的线段上所有店都在多边形内部，最后得到的就是多边形内所有顶点的坐标。

实际在 gui 程序里面跑的时候偶尔会有一两条小线段的颜色填充出错，这里目前还不是很清楚具体的原因，应该有少部分与端点相关的 corner case 没有考虑到。

后来发现大概是出现下面情况时候会有问题:



也就是说两个端点相邻时上面的描述不成立导致出现白线，在内部额外增加特判之后解决会出现直线空白的情况。虽然没有完全消除误差，但是之后偶尔有一到两条小线段，误差暂时可以接受。

11. 椭圆（包括圆）填充算法（seed_fill_ellipse（p_list））【9】

（1）参数和返回值

param p_list： 外切矩形坐标序列左上角和右下角坐标[$[x_0,y_0]$, $[x_1,y_1]$]（实际只要对角）

return ：填充区域所有点坐标的整数对序列 result: [$[x_0,y_0]$, $[x_1,y_1]$, ……, $[x_m,y_m]$]

（2）算法描述：

因为椭圆的中心一定是在椭圆内部的，所以采用种子填充十分方便。这里只需要考虑中心在原点的时候第一象限的情况，其他的通过平移对称可以全部找到。

采用相当于 4 联通的方式，对于每一个点$[x,y]$，如果$[x,y]$在椭圆内，那么下一步判断相邻两点$[x+1,y]$与$[x,y+1]$是否在椭圆内。如果在，将当前节点标记为 1 并加入输出，对该点继续进行上述操作，否则返回。如果遇到标记为 1 的节点说明已经判断过，也直接返回。

12. 多边形裁剪算法（clip_polygon（p_list, x_min, y_min, x_max, y_max））【10】【11】

（1）参数和返回值

param p_list： 图元控制点坐标序列[$[x_0,y_0]$, $[x_1,y_1]$, ……, $[x_n,y_n]$]

x_min：裁剪窗口左下角 x 坐标

y_min：裁剪窗口左下角 y 坐标

x_max：裁剪窗口右上角 x 坐标

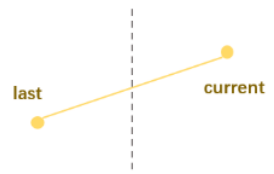
y_max：裁剪窗口右上角 y 坐标

return ：裁剪后图元控制点整数对序列 result: [$[x_0,y_0]$, $[x_1,y_1]$, ……, $[x_m,y_m]$]

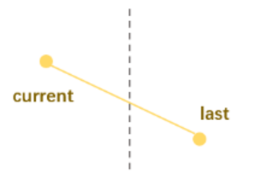
（2）算法描述：

使用 Sutherland-Hodgman 算法。类似 Conhen-Sutherland 算法，按照左右下上的顺序依次裁剪多边形。对于特定的直线边界，按照给定多边形的端点顺序遍历每一条边。对于每一条边而言，必然是下面四种情况的一种。

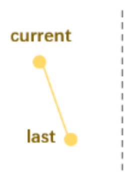
- **上一个顶点在裁剪区域外，当前顶点在裁剪区域内**
- **上一个顶点在裁剪区域内，当前顶点在裁剪区域外**
- **上一个和当前顶点都在顶点在裁剪区域外**
- **上一个和当前顶点都在顶点在裁剪区域内**



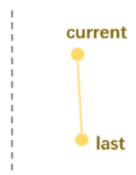
情况1:
上一个顶点在裁剪区域外，当前顶点在裁剪区域内



情况2:
上一个顶点在裁剪区域内，当前顶点在裁剪区域外



情况3:
上一个和当前顶点都在顶点在裁剪区域外



情况4:
上一个和当前顶点都在顶点在裁剪区域内

而处理的方法是：

情况 1：保留 current 与连线交点

情况 2：保留连线交点

情况 3：不保留点

情况 4：保留 current

判断点是否在区域内可以复用 Conhen-Sutherland 算法中的区域码。由于这里支持了矩形框裁剪多边形，所以交点的计算还是比较容易的。（但是算法本身对于任意凸多边形选框应该都是成立的，这里为了简化处理仅支持了矩形）

命令行界面模块（cg_cli.py）

1. 启动方式：

命令行界面输入 python cg_cli.py input_file output_file

2. 指令格式

启动后逐行读取指令，允许出现如下指令：

（1） resetCanvas x y

创建一个宽 x 像素，高 y 像素的画布

（2） saveCanvas name

将当前图像保存到名字为 name.bmp 的文件中

（3） setColor r g b

修改画笔的颜色

（4） drawLine name x0 y0 x1 y1 algorithm

直线，图元 id 为 name，端点(x0, y0), (x1, y1),算法有 DDA, Bresenham 算法两种

（5） drawPolygon name x0 y0 x1 y1 …… xn yn algorithm

多边形绘制，图元 id 为 name，输入为顶点序列，算法 DDA 或 Bresenham

（6） drawEillipse name x0 y0 x1 y1

椭圆，图元 id 为 name，椭圆外切矩形框左上顶点坐标(x0, y0)，右下顶点坐标(x1, y1)

（7） drawCircle name name x0 y0 x1 y1

圆，图元 id 为 name，圆外切矩形框左上顶点坐标(x0, y0)，右下顶点坐标(x1, y1)，实际在绘制的时候，如果按照 x 轴和按照 y 轴的半径不一致，以 x 轴半径为准

（8） drawCurve name x0 y0 x1 y1 …. xn yn algorithm

曲线，图元 id 为 name，控制点序列，算法 Bezier 或 B-spline，B-spline 默认三次均匀

（9） translate name dx dy

平移，图元 id 为 name，dx 为 x 轴偏移，dy 为 y 轴偏移

（10） rotate name x y r

旋转，图元 id 为 name，旋转中心为(x, y)，r 为顺时针旋转角

（11） scale name x y s

缩放，图元 id 为 name，缩放中心为(x, y)，缩放倍数为 r

（12） clip name x_min y_min x_max y_max algorithm

窗口裁剪，name 是需要进行裁剪的图元 id，(x_min, y_min)为窗口左下角坐标，(x_max, y_max)为窗口右上角坐标，算法 algorithm 有 Cohen-Sutherland 和 Liang-Barsky 两种

额外功能在 cg_cli.py 中不支持。

用户界面模块（cg_gui.py）

1. 启动方式：

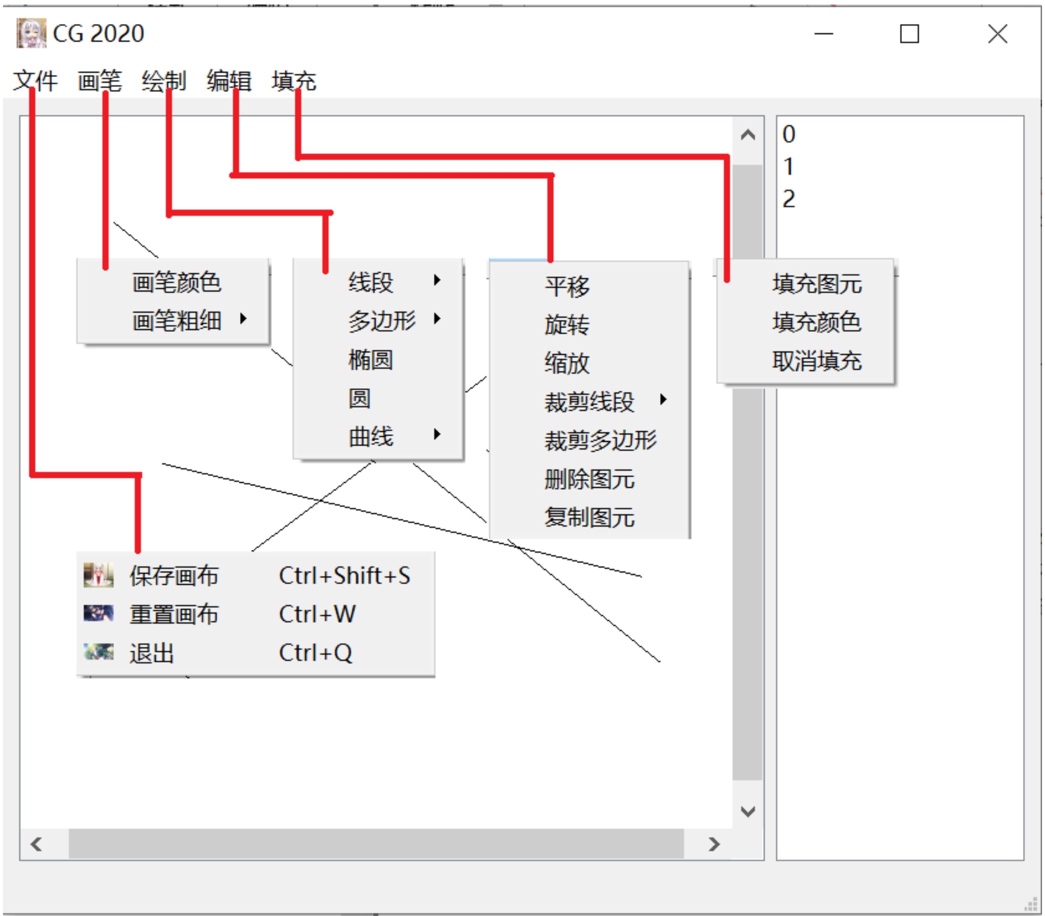
命令行界面输入 python cg_gui.py

2. 基本思想

按照描述，采用了 QGraphicsView、QGraphicsScene、QGraphicsItem 的绘图框架。理念是用三个大变量描述整个视图环境，分别是场景，视图和图元。简单的理解大概是，整个环境称为场景，每次绘制都是在场景中添加一个新的图元（例如直线，椭圆，多边形等等）。这些图元放置在场景中。而用户界面看到的称为视图，实际上就是从某一个角度去观察场景环境。由于这里是二维，视图 QGraphicsView 存在感不强，所以只需要处理好 QGraphicsScene 和 QGraphicsItem。

3. 界面描述：

对于用户而言，视图界面如下：



4. 操作大致描述

详细内容请参考使用说明书，这里只会提一些关键操作。

每一个最尾部的按钮对应一个 action。

进行平移，旋转，缩放，裁剪，删除，复制，填充操作时，需要点击右侧的 list_widget 选中具体图元。保存画布时输入要保存的文件名，在 output 文件夹下回保存当前画布的 bmp 文件。

由于在 gui 中设定 setMouseTracking 为 True，划线等操作时不需要一直摁住鼠标。

在绘制多边形时，如果在某次按下顶点时发现他与初始点的距离在 10 像素以内，默认多边形绘制完成。另一种结束方式是双击鼠标左键，直接结束图元绘制。

曲线双击结束绘制。

被红框包围表示图元被选中。可以进行平移、旋转、缩放以及针对线段的裁剪。

平移需要第一次点击确认初始位置，之后图元跟随鼠标移动，第二次点击后确定位置。

旋转需要第一次点击确认旋转中心，默认的初始位置为 x 坐标对应的 y 轴平行线。因此第一次点击后可能会出现突然大幅转动的情况。旋转中心不会显示。

缩放需要第一次点击确认缩放中心，默认鼠标位置与缩放中心距离的 1/400 为缩放倍数。由于缩放中心不显示，并且对大型图元的填充会导致计算机计算压力较大，极其不建议对已经填充好的图元进行缩放，否则会有 gui 卡死的错觉。

裁剪仅针对线段与多边形，只需要两次点击确认边框即可。

参考资料

【1】计算图形学算法基础第 2 版 David F.Rogers 著，石教英等译 p47，P50，P66

【2】计算图形学教程 ，孙正兴主编 周良，郑洪源，谢强编著 p76

【3】https://en.wikipedia.org/wiki/B%C3%A9zier_curve

【4】https://blog.csdn.net/Hachi_Lin/article/details/89812126

【5】计算图形学课件 4 transformation

【6】https://en.wikipedia.org/wiki/Cohen%E2%80%93Sutherland_algorithm

【7】https://www.geeksforgeeks.org/line-clipping-set-1-cohen-sutherland-algorithm/

【8】https://en.wikipedia.org/wiki/Liang%E2%80%93Barsky_algorithm

【9】https://blog.csdn.net/DUGUjing/article/details/83049407

【10】https://blog.csdn.net/damotiansheng/article/details/43274183

【11】https://github.com/phenomLi/Blog/issues/30