系统报告

《计算机图形学》12 月报告

姓名：唐金麟

联系方式：TangJinlin@smail.nju.edu.cn

目录

《计算机图形学》12 月报告

171860672 唐金麟 TangJinlin@smail.nju.edu.cn

一、初步分析

总体来看，需要实现的系统的主要功能为绘制图形，并保存为.bmp 格式的文件。而无论是哪种绘图算法，本质上来说，都是在“决定”哪些“位置”的像素点应该被置为当前画笔的颜色。也就是说，用到的框架需要提供以下功能：

①可以产生指定尺寸的“画布”；

②可以控制“画布“上具体某一像素点的颜色；

③可以保存“画布”为.bmp 格式文件。

经过广泛的信息搜集，以及对一些 GUI 框架帮助文档的查看，发现 QT 中的 Qlamge 类即可满足要求，且使用 QT 也便于后期实现用户交互接口（鼠标交互）。

二、算法介绍

2.1 直线的生成

1）DDA 算法

首先，给定的(x1,x2)与(x2,y2)两个点的相对位置总共会有 8 种情况（根据相对左右、上下、斜率的绝对值是否大于 1 来看）。所以，这里先对 x1,x2 的值做了一个判断，从而确定了两个点的左右相对位置，并且重新赋值为(xl, yl)和(xr, yr)，分别表示左点(left)和右点(right)。

这样一来，就剩下了四种情况，可根据(xl, yl)-> (xr, yr)直线斜率 k 分为：k>1、0<k<=1、-1<=k<=0、k>1。

而这四类情况可以先分成两类，即根据斜率(k)的绝对值来决定在 x、y 中哪一个方向上进行取样（即：以单位长度 1 来进行变化），另一个方向上则用斜率来计算每次循环的变化量(dx/dy)，且该方向上每次循环中确定点的坐标是时要对所计算得到的坐标进行四舍五入取整。

这里的对 dx、dy 的计算如下：

当斜率绝对值小于等于 1，沿着 x 方向取样，dx=1 dy=|k|

当斜率绝对值大于 1，沿着 y 方向取样，dx=|1/k| dy=1

由于初始点选择左点，所以 x 方向上 dx 一定为正。剩下的就是 y 方向上的正负问题，只需要再判断下 yl 和 yr 来决定 dy 的正负。

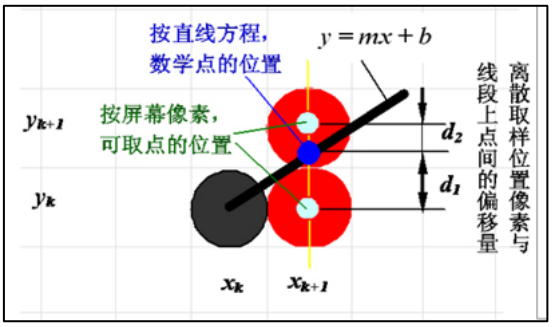
2）Bresenham 算法

Bresenham 算法与 DDA 算法相比，只用到了整数运算，是一种精确有效的光栅设备线

生成算法。在线段离散取样过程中，每一个放样位置上，只可能有两个像素点接近于线路径，而 Bresenham 算法的核心就在于如何根据决策参数的正负来决定应该取两点中的哪个点。

下面以生成方向从左到右，直线斜率在 0~1 之间（沿 x 正方向离散取样）的情况来进行推导说明。

如下图，假设第 k 步确定了像素点的位置为(x_k , y_k)，那么下一个取样位置 x_{k+1}有两个可能的位置：(x_{k+1} , y_k)和(x_{k+1} , y_k+1)。令(x , y)表示图中“按直线方程数学点的位置”，则 d_1和 d_2可计算如下：



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

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

距离的差分为：

$$d_1-d_2=2m(x_k+1)-2y_k+2b-1$$

而定义第 k 步的决策参数 p_k如下，且代入直线方程，以及 m=Δy/Δx（Δy 和Δx 为线段端点的垂直和水平偏移量）。

$$p_k=\triangle x(d_1-d_2)=2\triangle yx_k-2\triangle xy_k+c$$

此时，因为Δx>0，所以 p_k与 d_1-d_2符号相同，所以有以下结论：

①若 p_k>0，即 d_2<d_1，y_k+1 处的像素点更加接近线段，所以选择点(x_{k+1} , y_k+1)。

②若 p_k<0，即 d_2>d_1，y_k处的像素点更加接近线段，所以选择点(x_{k+1} , y_k)。

而关于 p_k的计算，可以利用如下第 k 步到第 k+1 步的增量来计算

$$k步：p_k=2\triangle y\cdot x_k-2\triangle x\cdot y_k+c$$
$$k+1步：p_{k+1}=2\triangle y\cdot x_{k+1}-2\triangle x\cdot y_{k+1}+c$$

相减，可得：

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

而 y_{k+1}-y_k的值有两种情况：0 或者 1，这取决于 p_k的正负(也就是 k+1 步选择了 y_k还是 y_k+1)。

所以，可得：

若 p_k>0，y_{k+1}-y_k=1

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

若 p_k>0，y_{k+1}-y_k=0

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

总结起来，在直线斜率在 0~1 之间情况下，Bresenham 算法的**处理流程**如下：

①从左端点开始，作为起始点(x_0 , y_0)，起始位置的决策参数 $p_0=2\Delta y-\Delta x$。

②从 k=0 开始，每个离散取样 x_k处，画点为(x_k , y_k)，则下一个要画的点为(x_{k+1} , y_{k+1})，其中 $x_{k+1}=x_k+1$；y_{k+1}以及 p_k的计算如下：

若 $p_k>0$，$y_{k+1}= y_k+1$，$p_{k+1}=p_k+2\Delta y-2\Delta x$;

若 $p_k<=0$，$y_{k+1}= y_k$，$p_{k+1}=p_k+2\Delta y$。

③循环此过程Δx 次，就完成了从左端点画线到右端点的过程。

以上处理适用于直线斜率 m 在 0~1 之间的过程。那么**其他情况**呢？

如同在 DDA 算法中分析的那样，给定的(x1,x2)与(x2,y2)两个点的相对位置总共会有 8 种情况。这里也是先把左端点、右端点判断出来，那么就只剩下 4 种情况（上述处理流程已解决其中 0<m<1 的情况）。

首先，当斜率 m>1 时，此时假设坐标轴的 x 与 y 互换（即整个坐标平面关于 y=x 对称变换一下），那么情况就会变得与 0<m<1 的情况一模一样。所以只需要将上述处理过程中，所有 x 有关的量与 y 有关的量互换，即可得到斜率 m>1 的情况下的处理流程。

剩下的两种情况：-1<m<0 和 m<-1。这里其实有两种选择：

方案一：先得出点(x2,y2)关于直线 y=y1 的对称点(x3,y3)，然后对(x1,y1)和(x3,y3)进行画线(此时可使用 m>0 的情况下的处理流程)；最后再将画出的线段再次关于 y=y1 对称即可。

方案二：重新推导这两种情况下的 y_{k+1} / x_{k+1}和 p_k计算公式。

这里方案一会产生不必要的开销，使程序的运行效率下降，所以这里选择了使用方案二，即重新推导得出了 y_{k+1}/x_{k+1}和 p_k的计算公式如下：

-1<m<0 的情况下：（沿 x 方向离散采样 $x_{k+1}=x_k+1$）

若 $p_k>0$，$y_{k+1}= y_k$，$p_{k+1}=p_k+2\Delta y$。

若 $p_k<=0$，$y_{k+1}= y_k-1$，$p_{k+1}=p_k+2\Delta y+2\Delta x$。

m<-1 的情况下：（沿 y 方向离散采样 $y_{k+1}=y_k-1$）

若 $p_k>0$，$x_{k+1}= x_k$，$p_{k+1}=p_k-2\Delta x$。

若 $p_k<=0$，$x_{k+1}= x_k+1$，$p_{k+1}=p_k-2\Delta y-2\Delta x$。

（注：Bresenham 算法中，直线垂直、水平、|m|=1 的这几种特殊情况无法使用以上所述的处理过程，需要单独处理，其生成过程也较为简单，dx 和 dy 直接按相应情形取 0 或 1 来完成线段的生成即可）

2.2 多边形的生成

多边形由多条直线组成，所以，一个多边形可以通过多次使用直线生成算法来完成。具体来说，就是对于按顺序给定的多边形的一系列顶点，依次在两点间使用直线生成的算法来生成多边形的某条边(根据生成多边形命令所指明的算法，选择调用 DDA 算法或 Bresenham 算法)。最终，第一个点和最后一个点之间再生成一条直线，即可完成整个多边形的生成。

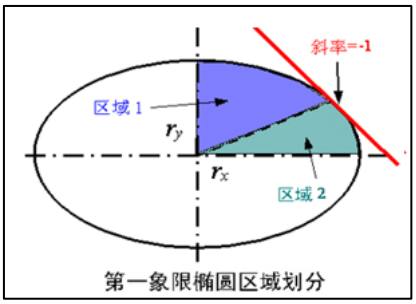
2.3 椭圆的生成 —— 中点椭圆算法

首先，若将椭圆先平移到中心在原点的位置，则椭圆在四个象限内的四个部分是对称的，且对称点可以十分方便地求出（只需要将 x、y 坐标取相反数）。所以，只需要用此算法得出第一象限内的点（1/4 圆弧），然后将点对称即可得出椭圆的全部圆弧。

定义椭圆函数如下：

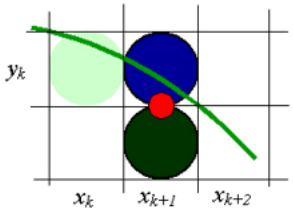
$$f_{ellipse}(x,y)=r_y^2x^2+r_x^2y^2-r_x^2r_y^2$$

第一象限的椭圆区域可以依据切线斜率分成两个部分（这里假设 $r_y<r_x$），如下图。



第一象限椭圆区域划分

这里从$(0,r_y)$开始顺时针生成椭圆，所以先是生成区域 1 中的圆弧。在区域 1 中，沿 x 正方向离散取样，设(x_k, y_k)为第 k 步确定的点，则下一个取样位置 x_{k+1}处有两个选择，如下图。



x_{k+1} 处候选像素及其中点

定义区域 1 的决策参数 $p1_k$，其值为两个候选点的中点代入 $f_{ellipse}$的值，即：

$$p1_k=f_{ellipse}(x_{k+1}, y_k-1/2)=r_y^2(x_k+1)^2+r_x^2(y_k-1/2)^2-r_x^2r_y^2$$

显然，对于 $p1_k$的正负，可以指示出哪个候选点离圆弧更近，从而做出选择：

若 $p1_k<0$，中点位于椭圆内，$y_{k+1}=y_k$；

若 $p1_k>=0$，中点位于椭圆外或者圆周上，$y_{k+1}=y_k+1$。

而 $p1_k$的值也不用每次都带入 $f_{ellipse}$来求，可以在 $p1_{k-1}$的基础上求（类似 Bresenham 算法中的迭代过程）。计算 $p1_{k+1}-p1_k$即可得出以下结论：

若 $p1_k<0$，

$$p1_{k+1}=p1_k+2r_y^2x_{k+1}+r_y^2$$

若 $p1_k>=0$，

$$p1_{k+1}=p1_k+2r_y^2x_{k+1}-2r_x^2y_{k+1}+r_y^2$$

如此循环，即可得到区域 1 中的圆弧。而循环直到 $2r_y^2x>=2r_x^2y$ 的时候，就进入了区域 2。在区域 2 中，过程类似，只是离散取样的方向变成了 y 负方向。定义区域 2 中的决策参数 $p2_k$（也是值为两个候选点的中点代入 $f_{ellipse}$的值），同样地推导，可得到(x_k, y_k)到下一个要画点 (x_{k+1}, y_{k+1})的决策过程如下：$(y_{k+1}=y_k-1)$

若 $p2_k>0$，中点位于椭圆内，$x_{k+1}=x_k$

$$p2_{k+1}=p2_k-2r_x^2y_{k+1}+r_x^2$$

若 $p2_k<=0$，中点位于椭圆外或者圆周上，$x_{k+1}=x_k+1$

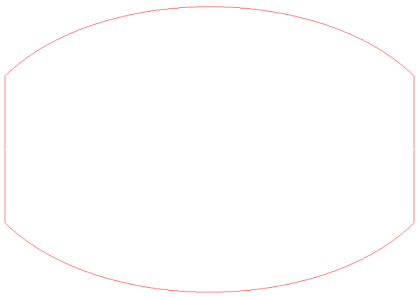
$$p2_{k+1}=p2_k+2r_y^2x_{k+1}-2r_x^2y_{k+1}+r_x^2$$

循环至(r_x, 0)即完成了第一象限圆弧的绘制。

以上处理流程是适用于在第一象限中，且假定$r_y \le r_x$的情况下来得出椭圆的 1/4 圆弧。那么如何处理$r_y > r_x$的情况呢？由于此时已平移到原点，所以若将椭圆旋转 90°（或者说关于 y=x 直线对称一下），就可以将$r_y > r_x$的情况转换为$r_y \le r_x$的情况。所以，在后者的情况看下，只需要先将r_y和r_x互换，然后执行适用$r_y \le r_x$的情况的处理流程，将得出的点的坐标(x, y)再变为(y, x)（关于 y=x 对称变换）就可以完成$r_y > r_x$的情况下的处理。

实现中点椭圆算法遇到的问题

这里在代码实现中点椭圆算法时，遇到了一个问题：画小椭圆时正常，但是画长短轴较大的椭圆时，椭圆形状奇怪，比如下图中画出的这个椭圆：



之后，仔细检查实现代码，也没有发现所用到的计算公式有错误。最终，经过一番网上查找，在一篇 博客 中找到了原因。原因是在使用 c++实现算法的时候，没有注意到使用 int 型整数变量相乘会有溢出的风险。具体情况如下图中的这句代码，“rx*rx*ry*ry”这个表达式涉及四个 int 型变量相乘，在画大椭圆时，rx 与 ry 均较大，所以这里有可能乘完的结果是超出了 int 型可以表示的范围。

```
/* 椭圆区域2：切线斜率>1 */
double p2= ry*ry*(xk+0.5)*(xk+0.5) + rx*rx*(yk-1)*(yk-1) - rx*rx*ry*ry;
```

所以，这里应该考虑利用 c++的计算语法将 int 类型提升转化为 double 类型，从而避免溢出。具体解决是将这句代码改成如下的形式：

```
/* 椭圆区域2：切线斜率>1 */
double p2 = ry * ry*(xk + 0.5)*(xk + 0.5) + 1.0*rx * rx*(yk - 1)*(yk - 1) - 1.0*rx*rx*ry*ry;
```

2.4 图元的平移

对一个点的平移操作较为简单，只需将点的坐标加上对应的平移量即可。

而这里实现图元平移操作，大体上有两种思路：①按图元的“参数信息“进行“平移“，然后重画（比如，直线平移两个端点、椭圆平移中心点；②直接平移构成图元的每一个像素点。

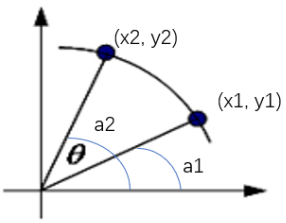
这里的实现方式选择了后者：直接平移构成图元的每一个像素点。下面叙述选择这样实现的考虑：

首先，对于平移操作，由于平移量都是整数，所以图元逐像素点平移不会产生非整数坐标，也就是说图元中的每个像素平移后，不需要再次面临“取哪一个像素“的决策问题。另外，从程序效率考虑，方式②只需要将图元所有像素点执行一次坐标的加法计算，而对于方式①，最后需要用新的参数信息，用对应的图元生成算法将所有像素点重新生成，显然无论是那种图元生成算法，本质上都是逐个算出了构成图元的每个像素点，而”在算法中计算出一个像素点“的代价，比”一次坐标加法“的代价，只可能更大，不可能更小（即根据参数信息重新生成图元的时间复杂度不会小于 O(n)），所以方式②在效率层面上更优。

2.5 图元的旋转

1）先实现将一个像素点绕基准点旋转的算法

先看最为基本的情况，基准点为原点的情况下，旋转θ角（逆时针为正），如右图。



这里通过极坐标变换，由θ=a2-a1，再将极坐标化为直角坐标，可以得出变换公式如下：

$$x2 = x1*\cos\theta - y1*\sin\theta$$

$$y2 = x1*\sin\theta + y1*\cos\theta$$

而对于基准点为任一点(rx, ry)的情况下，可以先将待平移点和基准点一起平移，将基准点平移到坐标原点，这样就化为了上面基准点为原点的情况，然后计算上述得出的变换公式，然后再按原来平移的路径反向平移回去即可。

所以，将平移中的加法/减法计算进去，最终可以得出一个按任意基准点平移的计算公式：（ 假设(x1, y1)绕(rx, ry)旋转θ角，得出(x2, y2) ）

$$x2 = xr + (x1-xr)*\cos\theta - (y1-yr)*\sin\theta$$

$$y2 = yr + (x1-xr)*\sin\theta + (y1-yr)*\cos\theta$$

2）利用点旋转的算法完成图元的旋转

①线段：旋转两个端点，重绘。

②多边形：旋转所有顶点，重绘。

③曲线：旋转所有控制点，重绘。

（注：本系统的命令行程序中椭圆不支持旋转，GUI 程序中椭圆支持旋转）

椭圆图元的旋转

这里额外说明一下，在 GUI 程序中是如何实现椭圆旋转的。

首先，这里以上过程为什么不能旋转椭圆？（但利用点的旋转可以完成椭圆上所有点绕中心点的旋转）因为这里用到的椭圆生成算法只能生成不“倾斜”的椭圆，所以一旦椭圆通过旋转发生了倾斜，就不能再次利用椭圆生成算法通过参数信息，重新生成对应的椭圆（缩放等等的其他操作完成后，会需要根据参数信息重绘椭圆）。

所以，在 GUI 程序中实现的椭圆图元，不仅仅要记录中心点坐标、长短轴长度，还会记录下一个角度 a，a 表示椭圆绕着自生的中心点顺时针转过的角度。显然，初始时 a=0。这样一来，其中椭圆的“重绘”过程就需要这样完成：用椭圆生成算法生成不倾斜的椭圆，再把椭圆上的所有点，绕自身中心点旋转角度 a。

之后，旋转椭圆的操作就可以这样完成：旋转中心点，改变 a 的值，重绘椭圆。

2.6 图元的缩放

1）先实现将一个像素点相对某固定点缩放的算法

先看最为基本的情况，固定点为原点的情况下，(x1, y1)缩放后为(x2, y2)，缩放系数为 s

$$x2 = x1*s$$

$$y2 = y1*s$$

而对于固定点为任一点(sx, sy)的情况下，仍然可以先将待缩放点和固定点一起平移，将固定点平移到坐标原点，这样就化为了上面固定点为原点的情况，然后计算上述得出的变换公式，然后再按原来平移的路径反向平移回去即可。

所以，将平移中的加法/减法计算进去，最终可以得出一个按任意固定点平移的计算公

式，
假设(x1, y1)按(sx, sy)按比例 s 缩放，得出(x2, y2)：

$$x2 = x1*s + sx*(1-s)$$
$$y2 = y1*s + sy*(1-s)$$

2）利用点缩放的算法完成图元的缩放

①线段：缩放两个端点，重绘。
②多边形：缩放所有顶点，重绘。
③椭圆：缩放中心点，同时按比例缩放长短轴：

$$rx=rx*s,$$
$$ry=ry*s,$$

然后重绘。
④曲线：缩放所有控制点，重绘。

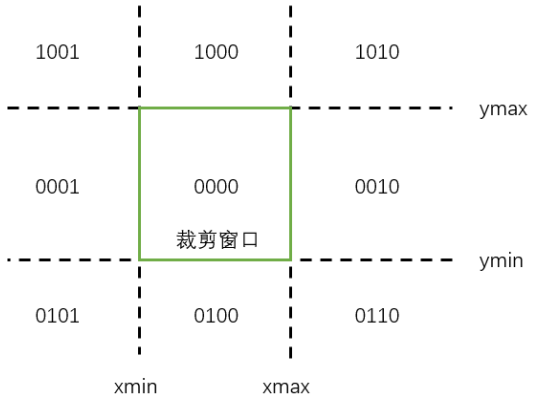
2.7 线段的裁剪

1）Cohen-Sutherland 算法

对算法的认识：Cohen-Sutherland 算法最为出色的一点是，通过二进制编码，可根据简单的直线端点间的“与”、“或”操作，直接排除大量无需剪裁的情况（即这些情况无需进行与边界间的交点计算以及相应的实、虚交点判别），从而大大提高了检测与计算效率。

下面结合自己的编程实现，来叙述对这个算法分析、理解与实现。

首先，裁剪窗口的四个边界所在直线可将平面分成 9 个区域，如下：



对这 9 个区域使用四个二进制位来进行编码，具体的编码规则如下：（从左往右依次是第 1、2、3、4 位）

若 x<xmin，第 4 位置 1，否则置 0；
若 x>xmax，第 3 位置 1，否则置 0；
若 y<ymin，第 2 位置 1，否则置 0；
若 y>ymax，第 1 位置 1，否则置 0。

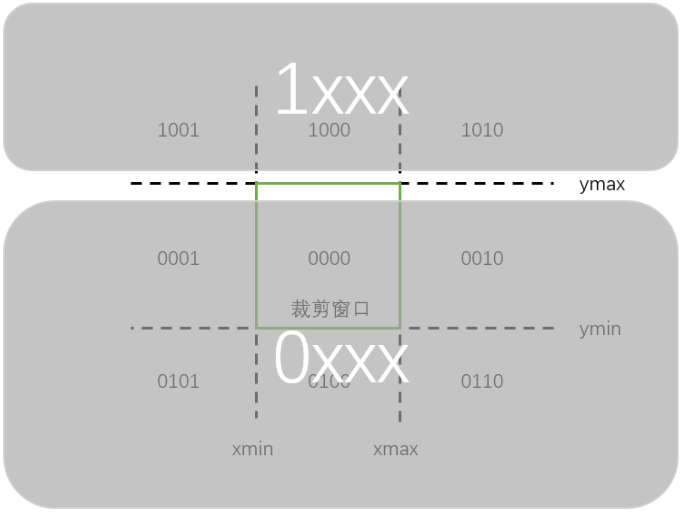
对待裁剪线段的两个端点按上述规则编码，假设得出它们两个对应的编码为 code1 和 code2。

下面对所有可能的情况分类说明。

①两个特殊情况。

第一种情况最为特殊，线段两个端点都在裁剪窗口内，此时 code1 和 code2 都为全零，也就是当"(code1 | code2)==0"的情况下，无需裁剪，保留全部线段。

第二种情况的考虑，需要从上述的编码规则出发，这四个比特位中，其实分别代表左、右、下、上四个边界所在的直线，而 1/0 表示在这条直线的哪一侧。其中 1 代表的那一侧的共同点，就是它们都是"远离"裁剪窗口的那一侧（即裁剪窗口这个封闭矩形落在的是 0 代表的那一侧）。示意图如下：（以 ymax 为例）



通过以上结论，我们可以发现，当 code1 和 code2 有某一位都是 1 时，表示两个端点都在某个边界直线"远离"裁剪窗口的那一侧，这就说明待裁剪的直线完全落在窗口，所以此时可以直接丢弃掉线段。也就是说，当"(code1&code2)!=0"的时候（某一位都是 1），直接丢弃线段。

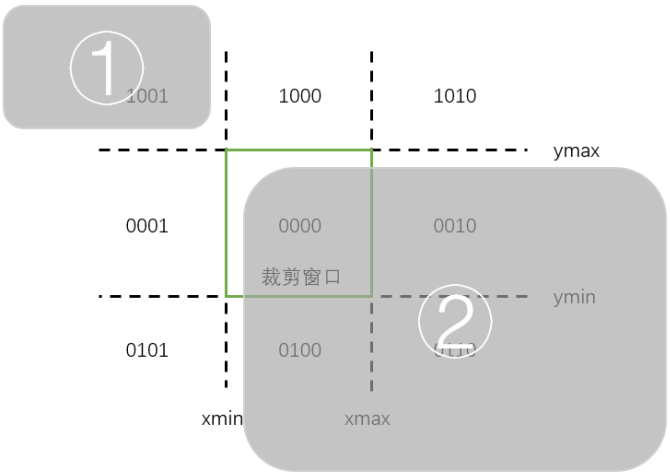
②其他情况

除了以上两种特殊情况，其余情况都无法避免求交点。但是，求出交点后，可以以交点为界，将直线分成两个部分，对其中靠近裁剪窗口的一部分进行递归处理，另一部分直接丢弃。

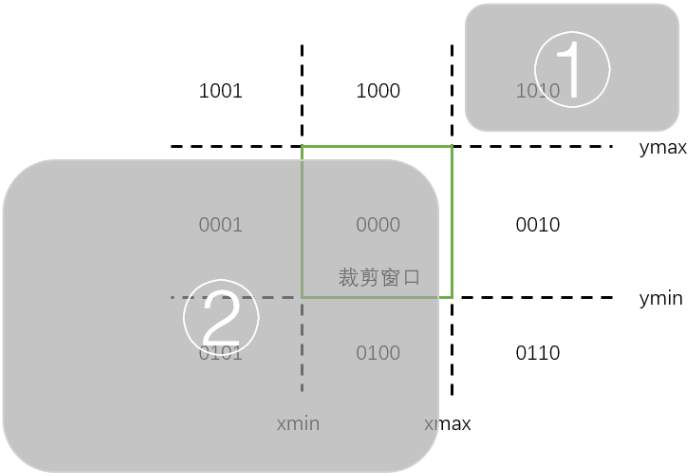
剩下的情况都是 code1 和 code2 中有某一个位不全为 0，我们可以从左往右检查（四个位分别代表四个边界），code1 和 code2 的某一位不全为 0，那么就求出线段所在直线与这个位代表的边界直线的交点。同时考虑，保留以交点为界的哪一部分。

四条边界其实是等价的，只是方向不同，所以，下面以 ymax 这条边界来说明这部分情况下的处理方式。

左侧第一位代表着 ymax 这一边界。所以，若编码的左侧第 1 位不全为 0，但 code1 和 code2 也没有某一位都是 1，说可能的情况是两个端点分别落在下图中的区域①和区域②：



或者



此时，从示意图不难看出，线段与 y=ymax 直线必有一个交点（假设落在区域①的点为 P1，区域②中的点为 P2，交点为 P3），且此时求出 P3 后，只需要对 P3P2 这一线段递归调用 Cohen-Sutherland 算法进行处理即可，而 P1P3 这一段可以直接舍弃（因为可以确定不可能在裁剪窗口中）。

所以，对应的其他三个边界上的情况(xmin、xmax、ymin)，也是类似的处理。

综上，即考虑完了所有可能的情况。

2）Liang-Barsky 算法

对算法的认识：Liang-Barsky 算法是比 Cohen-Sutherland 算法更快的参数化线段裁剪算法，其基本思路是：将待裁剪线段和裁剪窗口均看作一维点集，裁剪结果即为两点集的交集。

下面叙述，对 Liang-Barsky 算法的分析、理解与实现。

首先，这里先将 P1(x1, y1)、P2(x2,y2)两个点构成的线段所在的直线，写成参数方程的形

式：

$$x= x1+ u*(x2-x1) = x1+ u*\Delta x$$

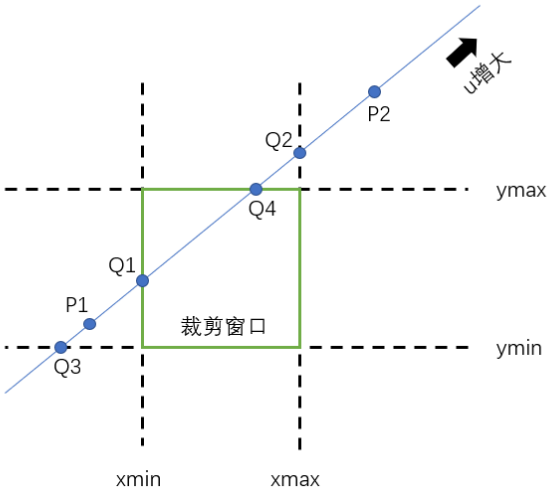
$$y= y1+ u*(y2-y1) = y1+ u*\Delta y$$

由参数方程的性质，可知，P1 点对应的参数 $u_{P1}=0$，P2 点对应的参数 $u_{P2}=1$。u 越小，表示在直线上越往 P2->P1 点的方向上偏，u 越大，在直线上表示越往 P1->P2 点的方向上偏。

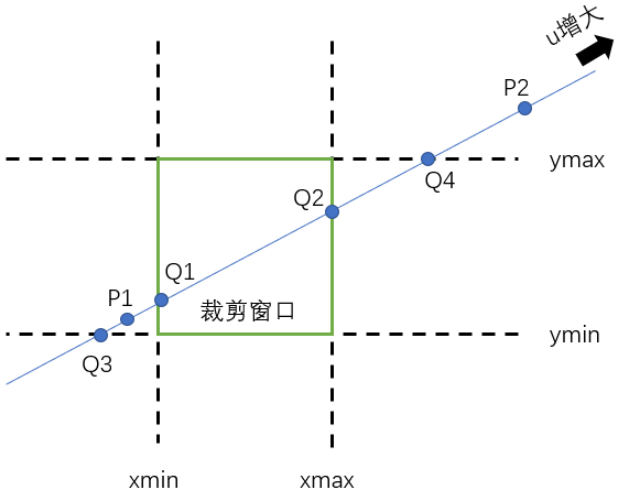
利用参数方程，我们可以将二维上的参量 x 和 y 转化为一维参量 u，用 u 的值表示在直线上这一维度的位置。所以，Liang-Barsky 算法的核心就在于如何将二维的裁剪窗口通过参数方程化为直线上的一维参量，这里将二维裁剪窗口化为一维后的点集称为“诱导窗口”。

直线与裁剪窗口的四个边界共有四个交点，记为 Q_k(k=1, 2, 3, 4 分别对应 xmin, xmax, ymin, ymax，即左、右、下、上边界的交点)。那么，这四个交点中，有两个交点即构成“诱导窗口”，关键在于确定是哪两个点。

比如下图中，Q1Q4 构成诱导窗口。



而下图中，Q1Q2 为诱导窗口。

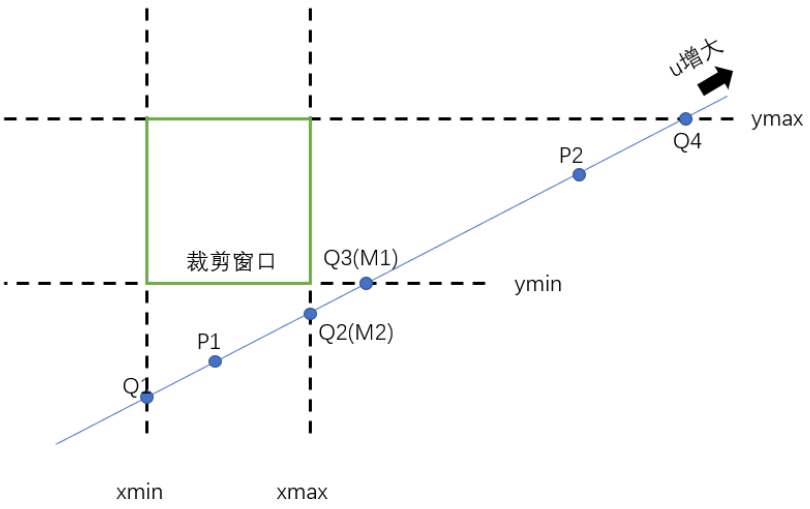


通过上面两幅图，我们可以大致感觉出诱导窗口给人的感觉是一种“两边夹”的感觉，即从直线无限远的两端，从中间（裁剪窗口的位置）行进，四个交点中，最“靠里面”的两个交点即为诱导窗口。下面，将这种“感觉阐述”得更加有条理。

首先，一个窗口边界直线可以将平面分成两部分，这里将不含裁剪窗口一侧称为直线外侧，含有裁剪窗口的一侧称为内侧。那么，从 u 增大的方向行进，根据穿越直线是从外侧到内侧，还是内侧到外侧，可以将四个边界交点可以分成两类。而从外侧穿到内侧的两个交点中，取 u 值较大的那个，记为 M1；从内侧穿到外侧的两个交点中，取 u 值较小的那个，记为 M2。这里记 M1、M2 对应的参数为 u1、u2。

当 u1<u2 的时候，对应上图两种情况，诱导窗口即为 M1M2，按此窗口裁剪即可。

而当 u1>=u2 的时候，可以发现，一种情况如下图（其他类似）。此时，按上述规则确定得到的 M1M2，可以发现，直线完全在窗口外，此时无需裁剪。



Liang-Barsky 算法的核心思想正是以上所叙述的这种思想。只不过，它使用了一些较为巧妙的技巧，来表达了上面提到的“内侧”、“外侧”的概念，同时也使得对四条边界的处理计算变得统一，下面叙述。

这里先定义变量的值，然后再解释其意义。

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

对比参数方程可看出，若按以上的值定义变量，交点 Q_k 对应的参数 u 的值就是 $u_k=qk/pk$。

而且，可以得出以下结论：（pk 中 k 可取 1，2，3，4）

当 pk<0 时，表示此交点属于从边界直线的外侧穿越到内侧；当 pk>0 时，表示此交点属于从边界直线的内侧穿越到外侧。因为，例如，当$\Delta x \geqslant 0$ 时，对于左边界 p1<0（p1=-Δx），线段从左边界的外部到内部；对于右边界 p2>0（p2=Δx），线段从右边界的内部到外部。当 Δy<0 时，对于下边界 p3>0（p3=-Δy），线段从下边界的内部到外部；对于上边界 p4<0（p4=Δy），线段从上边界的外部到内部。

所以，确定 M1，只需要在 pk<0 时算出的 qk/pk 中取最大的值；确定 M2，在 pk>0 时算出的 qk/pk 中取最小的值。这样即可得出 u1，u2。当然，若 u_{P1}>u1，则 u1 要取 u_{P1}；若 u_{P2}<u2，则 u2 要取 u_{P2}（这两种情况说明，线段端点在裁剪窗口内，即 P1P2 对应的点集是诱导窗口 M1M2 点集的子集）。

综上，Liang-Barsky 算法的编程实现可以按照以下过程进行：

1. 初始化 u1=0，u2=1（分别对应 P1、P2）；
2. 按上面给出的表达式计算 pk、qk（k=1，2，3，4）；

3. 依次判断 pk，并考虑以下情况：
（1）若 pk<0，u1=max(u1, qk/pk)；（从外侧到内侧情况，更新 u1）
（2）若 pk>0，u2=min(u2, qk/pk)；（从内侧到外侧情况，更新 u2）
（3）若 pk==0&&qk<0，说明此线段平行于当前边界且位于边界外侧。（pk==0 说明线段垂直或平行，再根据 qk 的取值，可以推出这一条）
4. 每次计算完 u1、u2，判断是否“u1>u2”，如果是，则弃用线段，立即返回。最后 p1~p4 处理完毕，还未弃用，则将 u1、u2 带入参数方程，得出新的线段的两个端点。

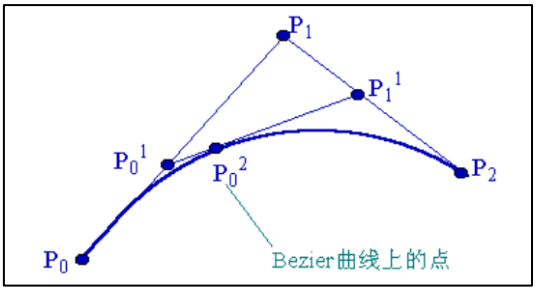
2.8 曲线的生成

1）Bezier 曲线

对 Bezier 曲线的认识：贝塞尔曲线几何意义明确，形状容易构造，形状控制起来也较为简单；缺点是它的局部形状控制能力弱且拼接起来较复杂。

下面重点讲述本系统实现中使用的用于 Bezier 曲线离散点生成的“德卡斯特里奥（de Casteljau）算法”。

从 Bezier 曲线的动态作图过程中，可以直观地感受到，这个过程其实就是在控制点的连接折线段上，两两控制点之间，按照一定的比例（即参数 u 的值），取出一个分割点，然后再将这组分割点的连接折线段，再次按照这个比例，再取出一组分割点……按照此过程进行下去，直到得出的点只有一个了，最终的这个点就是 Bezier 曲线上参数 u 对应的点。



这里以上图作为一个具体的例子来说明。P_0、P_1、P_2是三个控制点，第一次“分割”得出了 P_0^1、P_1^1两个点，且其中比例关系满足：

$$P_0P_0^1 : P_0^1P_1 = P_1P_1^1 : P_1^1P_2 = u : 1-u$$

第二次分割，得到了 P_0^2，也就是最终曲线上的一个离散点，且其比例关系满足：

$$P_0^1P_0^2 : P_0^2P_1^1 = u : 1-u$$

而 de Casteljau 算法则是给出了上述“直观”过程的递推计算公式：

$$P_i^k = \begin{cases} P_i & k=0 \\ (1-t)P_i^{k-1} + tP_{i+1}^{k-1} & k=1,2,\cdots,n, i=0,1,\cdots,n-k \end{cases}$$

其中的 k 表示阶数，0 阶的点就是控制点本身。下面的公式可以理解为求两个点之间按比例 u 分割的点的坐标，也就是计算得出上述所说的“分割点”的坐标。

所以，有了这个公式，在编写程序实现的时候，就非常简单方便地可以写出程序。可以直接使用递归来实现。也可以使用类似动态规划中的技巧，使用一个二维数组来实现，i 、k 是其两个维度，先计算 k=0，i=0，1，2……的值，然后计算 k=1，i=0，1，2……的值，依次类推。

实现 Bezier 曲线绘制算法中遇到的问题

最初实现完成后，发现画出的 Bezier 曲线大致形状符合，但是不够光滑，边缘很粗糙。所以分析认为这里算法整体上没有大的问题，但是肯定有着计算上的小问题。最为可疑的就是 double 和 int 之间转换的精度问题，所以在完整地检查了一遍代码后，发现了问题所在。使用递推公式计算出的点的坐标是使用 int 型整数来保存的，而这里递推有着好几层，每一层损失小数部分的精度，那么累加起来，最后一层算出来的曲线上的离散点，就会存在着较大的误差。所以，这里递归计算出来的 P_i^k 的坐标值需要使用 double 来存储，最终算进曲线上的点的时候，再四舍五入保留小数。

2）B-spline **曲线**

对 B-spline 曲线的认识：Bezier 曲线局部修改的能力较差，且拼接起来较复杂。B 样条曲线则是保留了 Bezier 曲线的优点，但又克服了它的这两个缺点，有着良好的局部修改特性。

本系统中使用了 de Boor-Cox 算法完成了 3 次均匀 B-spline 曲线的绘制，下面对此进行叙述。

首先，B-spline 曲线的参数曲线形式定义如下：（此处 k 代表的是阶数不是次数）

$$P(u)=\sum_{i=0}^{n}P_iB_{i,k}(u),\ u\in[u_{k-1},u_{n+1}]$$

而在 de Boor-Cox 算法中，有以下递推定义：

$$B_{i,1}(u)=\begin{cases}1 & u_i<x<u_{i+1}\\ 0 & Otherwise\end{cases}\qquad \text{并约定：}\ \frac{0}{0}=0$$

$$B_{i,k}(u)=\frac{u-u_i}{u_{i+k-1}-u_i}B_{i,k-1}(u)+\frac{u_{i+k}-u}{u_{i+k}-u_{i+1}}B_{i+1,k-1}(u)$$

将此递归定义式带入 B-spine 曲线的定义式中，可以得出 B-spline 曲线上离散点的一个递推计算公式：（k-1 <= j < n-1；u_j <= u < u_{j+1}；）

$$P_i^r(u)=\begin{cases}P_i\ ,r=0;\\ \lambda_i^r(u)P_i^{r-1}(u)+\left(1-\lambda_i^r(u)\right)P_{i-1}^{r-1}(u)\\ \qquad\qquad i=j-k+r+1,\ldots,j\end{cases}$$

其中：

$$\lambda_i^r(u)=\frac{u-u_i}{u_{i+k-r}-u_i}\quad(r=0,1,2,\cdots\cdots,k-1)$$

此时，可以明显地看出来 B-spline 曲线与 Bezier 曲线有着非常类似的递推计算公式：在 Bezier 曲线中的参数 u，现在变了$\lambda_i^r(u)$。且从上面的递推式可以看出，通过 de Boor-Cox 算法这样定义基函数，计算一个 3 次 B 样条曲线（代入 k=4）上的离散点 P_j^3 只会用到四个控制点：P_j^0、P_{j-1}^0、P_{j-2}^0、P_{j-3}^0，（而在 Bezier 曲线中，每一个离散点的生成都会用到所有的控制点）。所以 j 在 k-1 ~ n-1 之间递增取值，相当于就是依次利用**每 k 个控制点作为一组**，来生成一段曲线上的离散点。**这也就解释了为什么 B 样条曲线有着形状的局部控制能力。**

有了以上这个公式，计算离散就变得与 Bezier 曲线类似了。只是在原来的基础上，在

$[u_{k-1}, u_{n+1}]$上对每一段 $u_j <= u < u_{j+1}$（k-1 <= j < n-1），按以上公式进行计算（u 按一定的步长增加）。

但这里还未涉及分割节点 u_i的取值到底是多少，所以还需要额外说明一下分割节点 u_0、u_1、……u_{n+k}的取值问题。大部分关于 de Boor-Cox 算法的资料对分割节点如何取值都没有一个详细的说明。在这里，说一点自己对这个的理解：

首先，这里的参数 u 的取值范围不再是 Bezier 曲线中的[0, 1]，因为这里的 u 不再是一种"比例"系数了，而这里的λ才是，所以λ才是[0,1]之间的一个数。此外，从λ的计算公式中，可以发现，u 的值的大小无关紧要，因为这里算的是一个比值，两个参数 u 相减的值再比上了两个参数 u 相减的值。而按照均匀 B 样条曲线的定义，这里的分割节点之间的差值要相等，所以这里的分割节点取值可以简单地以这样方式来取：$u_i = i$ （i=0, 1, 2, … n+k）。

有了以上分析过程，就可以类似 Bezier 曲线的生成算法来编写程序来完成 B-spline 离散点的生成过程了。

此外，以下是自己在实现 de Boor-Cox 算法时额外的一些思考。

① 为什么 u 的取值会在$[u_{k-1}, u_{n+1}]$这个处在"中间"的区间，而不是整个分割节点所在的区间$[u_0, u_{n+k}]$呢？

分析：

a）首先是左端点：上面分析中提到离散点P_j^3的生成会用到四个控制点：P_j^0、P_{j-1}^0、P_{j-2}^0、P_{j-3}^0，所以，如果 j 从 0 开始取，那么 j-3 就会小于 0 越界了，所以自然地，要满足 j-(k-1)>=0，那么 j 的第一个取值就是 k-1；

b）然后是右端点：控制点的只有 n+1 个，下标最大的就是 n，所以自然地就有 j<n+1。

② 为什么分割节点需要 u_0、u_1、……u_{n+k}这么多个？

分析：

在 de Boor-Cox 算法规定的递推定义中，可以发现，j 取 n 时（即轮到最后一组的 k 个控制点时），计算 λ 时，会用到 u_{i+k}，而 i 最大可以取到 j，即最大会用的分割节点为 u_{n+k}。同理，j 取 k-1 的时候，i 最小可为 j-k+1，所以最小下标的 u 为 $u_i = u_0$。

三、系统介绍

3.1 命令行程序

1）实现概述

系统信息

开发工具	Qt 5.9.8
构建套件（编译）	Desktop Qt 5.9.8 MinGW 32bit
运行环境	Windows 10 控制台
打包工具	Enigma Virtual Box 9.40 Build 20191010
推荐分辨率	1920×1080

注：如需编译压缩包内的控制台程序源代码，请注意项目所在路径不能含有中文字符。

代码框架设计

- **头文件**

algorithm.h	算法模块对外接口
canvas.h	画布类定义及对外接口
pixelset.h	图元类（图元基类、直线类、多边形类等）的定义及对外接口

- **源文件**

main.cpp	程序入口
algorithm.cpp	核心算法模块的实现
canvas.cpp	画布类的实现
pixelset.cpp	图元类的实现

这里的命令行程序，主要完成读取指令文件，输出.bmp 文件的功能，也包含着算法的核心模块。这里使用了 QT 中 QImage 类来完成对像素点的控制，主要使用了其中的 setPixelColor()函数来设定某一坐标像素点的 RGB 颜色，以及使用 save()函数保存为.bmp 文件。

使用到的各个算法的实现，以全局函数的形式放在了 Algorithm.cpp 文件中，在 Algorithm.h 中提供接口，共各个图元类使用。而各类图元则以类的形式，使用了继承、多态的思想来实现。

```
12      //画一条直线-DDA算法
13      void drawLine_DDA(int x1,int y1,int x2,int y2, PixelSet& myset);
14      //画一条直线-Bresenham算法
15      void drawLine_Bresenham(int x1,int y1,int x2,int y2, PixelSet& myset);
16
17      //画多边形-DDA算法
18      void drawPolygon_DDA(const vector<Point>& vertexs, PixelSet& myset);
19      //画多边形-Bresenham算法
20      void drawPolygon_Bresenham(const vector<Point> &vertexs, PixelSet& myset);
21
22      //画椭圆-中点椭圆生成算法
23      void drawEllipse(int x,int y,int rx,int ry, PixelSet& myset);

26      //将某个点(x,y)绕(xr,yr)旋转r°角
27      void rotatePoint(int &x,int &y,int xr,int xy,int r);
28
29      //将某个点(x,y)以(sx,sy)为中心按s的比例缩放
30      void scalePoint(int &x,int &y,int sx,int sy,float s);
31
32      //裁剪
33      //Cohen-Sutherland算法
34      void Cohen_Sutherland(int &x1, int &y1, int &x2, int &y2, int xmin, int ymin, int xmax, int ymax);
35      //Liang-Barsky算法
36      bool Liang_Barsky(int &x1, int &y1, int &x2, int &y2, int xmin, int ymin, int xmax, int ymax);
```

… …

2）在命令行程序的设计中，重要的类有以下几个。

①PixelSet 类

这个类是所有图元的基类，体现着所有类型图元的共通性。其中存储着一个图元的所有像素点的坐标（使用 vector 存储）、ID 和颜色。其方法包含着一个图元可能的操作（或者只是待实现的接口，c++中的虚函数），如绘制、平移、旋转、缩放等。

```cpp
//图元基类
class PixelSet     △ 'PixelSet' has virtual functions but non-virtual destruct
{
protected:
    int id;//图元编号
    QColor color;//图元颜色
    vector<Point> points;//构成图元的所有像素点的坐标
    friend class Canvas;
public:
    PixelSet(int i=0,QColor icolor = Qt::black){id=i;color=icolor;}
    //设定图元ID
    void setID(int i){id=i;}
    //设定图元颜色
    void setColor(const QColor &icolor){color=icolor;}
    //增加一个指定坐标的像素点
    void add(int x,int y){ points.push_back(Point(x,y));}
    //将本图元绘制于画布上
    void paint(QImage *image) const;
    //平移:(dx,dy)
    void translate(int dx,int dy);
    //根据参数绘制相应的图元
    virtual void refresh()=0;
    //旋转
    virtual void rotate(int x,int y,int r)=0;
    //缩放
    virtual void scale(int x,int y,float s)=0;
    //裁剪
    virtual void clip(int x1, int y1, int x2, int y2, string algorithm){}
};
```

（注：表示点坐标的类定义如下。）

```cpp
//表示一个坐标点的类
class Point
{
public:
    int x,y;
    Point(int ix=0,int iy=0) {x=ix;y=iy;}
    //设定坐标
    void set(int ix,int iy){x=ix;y=iy;}
};
```

②Line 类

直线类，继承 PixelSet 类，表示直线图元。在基类的基础上，额外记录了直线的参数信息。同时，也根据直线的特性，重写实现了基类中的旋转、缩放、剪裁等的成员虚函数。

```cpp
//线段
class Line:public PixelSet
{
    //图元的参数
    int x1,y1,x2,y2;
    string algorithm;
public:
    Line(){}
    Line(int ix1,int iy1,int ix2,int iy2,string ialgorithm){ ... }
    //参数设定
    void set(int ix1,int iy1,int ix2,int iy2,string ialgorithm){ ... }
    //根据参数绘制图元
    void refresh();
    //旋转
    void rotate(int x,int y,int r);
    //缩放
    void scale(int x,int y,float s);
    //裁剪
    void clip(int x1, int y1, int x2, int y2, string algorithm);
};
```

③Polygon 类

多边形类，继承 PixelSet 类，表示多边形图元。在基类的基础上，额外记录了多边形的顶点信息。同时，也根据多边形的特性，重写实现了基类中的旋转、缩放等的成员虚函数。

```
76  //多边形
77  class Polygon:public PixelSet
78  {
79      //图元的参数
80      vector<Point> vertexs;
81      string algorithm;
82  public:
83      Polygon(){}
84      Polygon(const vector<Point>& ivertexs, string ialgorithm){ ... }
87      //参数设定
88      void set(const vector<Point>& ivertexs, string ialgorithm){ ... }
92      //根据参数绘制图元
93      void refresh();
94      //旋转
95      void rotate(int x,int y,int r);
96      //缩放
97      void scale(int x,int y,float s);
98  };
```

④Ellipse 类

椭圆类，继承 PixelSet 类，表示椭圆图元。在基类的基础上，额外记录了椭圆的参数信息。同时，也根据椭圆的特性，重写实现了基类中的缩放等的成员虚函数。

```
101 class Ellipse:public PixelSet
102 {
103     //图元的参数
104     int x,y;
105     int rx,ry;
106 public:
107     Ellipse(){}
108     Ellipse(int ix,int iy,int irx,int iry){ ... }
111     //参数设定
112     void set(int ix,int iy,int irx,int iry){ ... }
116     //根据参数绘制图元
117     void refresh();
118     //缩放
119     void scale(int x,int y,float s);
120 };
```

⑤Curve

曲线类，继承 PixelSet 类，表示曲线图元。在基类的基础上，额外记录了曲线的控制点及算法信息。同时，也根据曲线的特性，重写实现了基类中的缩放、旋转等的成员虚函数。

```cpp
//曲线
class Curve:public PixelSet
{
    //图元的参数
    vector<Point> vertexs;
    string algorithm;
public:
    Curve(){}
    Curve(const vector<Point>& ivertexs, string ialgorithm){ ... }
    //参数设定
    void set(const vector<Point>& ivertexs, string ialgorithm){ ... }
    //根据参数绘制图元
    void refresh();
    //旋转
    void rotate(int x,int y,int r);
    //缩放
    void scale(int x,int y,float s);
};
```

⑥Canvas 类

这里使用 QImage 作为画布，但是直接使用不太方便，所以在此基础上封装了一个 Canvas 类来作为画布，它使用 vector 存储了当前画布所有图元的指针，可以读取指令文件，设置画布大小、画笔颜色、保存画布为.bmp 文件等，也提供了生成各种图元、执行各类变换操作的接口。

```cpp
//表示画布的类
class Canvas
{
    //当前画布大小
    int width,height;
    //当前画笔颜色
    QColor color;
    //当前所有的图元
    vector<PixelSet *> pixelsets;
    //保存文件的目录
    QString outputDir;
public:
    Canvas(QString dir="..//picture//"):color(0,0,0){outputDir=dir;reset(100,100);}
    ~Canvas();
    //重置画布
    void reset(int w,int h);
    //保存画布
    void save(const QString& name);
    //设置当前画笔颜色
    void setColor(int r, int g, int b);
    //刷新画布显示（将当前所有图元输出到画布，且跳过超出画布边界的点）
    void refresh(QImage *image);
    //读取文件中的命令
    void drawFromFile(const char* inputFile);
    //画一条直线
    void drawLine(int id,int x1,int y1,int x2,int y2, string algorithm);
    //画多边形
    void drawPolygon(int id, const vector<Point>& vertexs, string algorithm);
    //画椭圆-中点椭圆生成算法
    void drawEllipse(int id,int x,int y,int rx,int ry);
    //画曲线
    void drawCurve(int id, const vector<Point>& vertexs, string algorithm);
    //指定id的图元平移
    void translate(int id,int dx, int dy);
    //指定id的图元旋转
    void rotate(int id,int x,int y,int r);
    //指定id的图元缩放
    void scale(int id,int x,int y,float s);
    //指定id的图元裁剪
    void clip(int id,int x1, int y1, int x2, int y2, string algorithm);
};
```

3.2 GUI 程序

1）概述

系统信息

开发工具	Qt 5.9.8 + Visual Studio 2017
构建套件（编译）	Desktop Qt 5.9.8 MSVC-2017 64bit（在 VS2017 中构建项目）
运行环境	Windows 10
打包工具	Enigma Virtual Box 9.40 Build 20191010
推荐分辨率	1920×1080

代码框架设计

- **头文件**

algorithm.h	算法模块对外接口
canvas.h	画布类定义及对外接口
pixelset.h	图元类（图元基类、直线类、多边形类等）的定义及对外接口
CG_Painter.h	程序主窗口类

- **源文件**

CG_Painter.cpp	程序主窗口类的实现（核心：包含鼠标事件响应逻辑）
main.cpp	程序入口
algorithm.cpp	核心算法模块的实现
canvas.cpp	画布类的实现
pixelset.cpp	图元类的实现

- **其他文件**

CG_Painter.ui	主窗口的界面设计文件（使用 Qt Designer 进行可视化编辑）
CG_Painter.qrc	Qt 资源文件（用于添加图片资源）
CG_Painter.rc	VS2017 的资源文件（用于设置.exe 文件的图标）

总体上来说，这里使用了一个 Qt 中的主窗口，在主窗口中通过捕获鼠标事件，获取所需绘图参数，然后调用上述控制台程序中的 Canvas 类（画布类）提供的接口完成绘图，并且将绘图结果展示到主窗口的绘图区域。

布局如下：（中间为绘图区域）

具体的功能与操作方式，请见于“**系统使用说明书**”，这里不再赘述。

2）实现中值得一提的部分

- **状态标识**

这里的 GUI 程序，主要考虑如何以较为友好的交互方式来确定绘制图元的参数信息，所以，这里对 Qt 中的鼠标事件进行了较为细致的处理。

这里总共用到了四种鼠标事件：按下、释放、移动、双击。而这四种鼠标事件对应了四个事件响应函数，那么如何在同一个事件响应函数中实现不同状态下不同的响应逻辑呢（比如绘制直线状态和绘制多边形状态对鼠标事件的响应是不同的）？

这里，使用了 C++枚举类型变量来标识当前绘图状态，然后根据不同状态来设计了对应的鼠标事件响应逻辑。比如，如下图中所示，这里的第一层是大的状态：画直线、多边形、椭圆等。而每一个大的状态下还有小的状态，比如画直线时有几个阶段：按下第一个点之前/之后等。

```
24      //表示当前状态：画直线、多边形、椭圆
25      enum PAINTER_STATE {
26          NOT_DRAWING, DRAW_LINE, DRAW_POLYGON, DRAW_ELLIPSE
27      };
28      PAINTER_STATE state = NOT_DRAWING;
```

这种方式下，避免了各个不同状态下响应逻辑的重叠，结构更为清晰，也便于可以逐步开发实现更为复杂的功能。

- **双缓冲机制**

GUI 程序中一个重要的核心机制就是使用了“双缓冲机制”的思路来避免了动态更新画布（比如直线绘制完成前直线随鼠标移动而改变、拖动图元进行平移、椭圆的放大缩小...）时出现闪烁、拖影等的问题。

主要的思路：这里使用了两块画布（即两个 Canvas 类对象），一个是 myCanvas（作为主要的显示画布），一个是 bufCanvas（临时缓冲）。每次绘制还未结束之前，都是在 bufCanvas

上进行修改（每次修改前，bufCanvas 要从 myCanvas 中复制全部内容），并且显示出的是 bufCanvas 上的内容。最终绘制完成后，将 bufCanvas 中的内容复制到 myCanvas 中，并且切换回显示 myCanvas 的内容。

在这种双缓冲机制下，相当于每一次的修改都是基于刚进入绘制状态时的画布内容进行修改。而且由于 myCanvas 中保存着刚进入绘制状态时的画布内容，所以这里对 bufCanvas 一次修改完成后，再次修改时，不用对上次修改的内容进行擦除，而是直接从 myCanvas 中复制内容过来即可，所以避免了闪烁、拖影等等的问题。

- **通过鼠标事件获取绘制参数**

这里可通过鼠标事件获取到的参数大体上有两个：①点的坐标信息；②鼠标的位移信息。在实现中就是通过这两个信息的值，结合一些数学上平面几何的知识来计算出所以需要的绘制参数。

最为典型的，比如，图元旋转的实现，最为重要的就是计算绕旋转点的旋转角度，而这里可以通过三个点（旋转点、鼠标按下时的点、拖动后的点）构成的三角形来计算这个旋转角度，也就是转化为了一个“通过三角形的三个顶点坐标来计算其中一个角的角度”的几何问题。

四、总结

本系统完成的核心算法模块，完成了直线生成（DDA 算法、Bresenham 算法），多边形生成，椭圆生成（中点椭圆算法），曲线生成（Bezier 和 B-spline），图元平移、旋转、缩放，线段剪裁（Cohen-Sutherland 算法、Liang-Barsky 算法）的实现。GUI 程序实现了较为友好的用户交互 UI，可以较为方便地画直线、多边形、椭圆、曲线，且可以保存画布、设定颜色、对所有类型的图元进行平移、旋转、缩放、删除。

参考资料

1. 课堂讲授 PPT（报告中使用了 PPT 中的部分示意图、公式）

2. QT 帮助文档（Qt Assistant 5.9.8 (MinGW 5.3.0 32-bit) 查阅相关库函数用法）

3. 计算机图形学实验（四）—— 中点椭圆算法源码（解决了画大椭圆错误的情况） - junewgl - CSDN 博客
（https://blog.csdn.net/qq_40755094/article/details/84258043）

4. [计算机图形学经典算法] Liang-Barsky（梁友栋-Barsky） 算法 （附 Matlab 代码） - Holeung blog - CSDN 博客
（https://blog.csdn.net/soulmeetliang/article/details/79185603）

5. 梁友栋-Barsky 裁剪算法 - bigwhiteshark(云飞扬) - 博客园
（https://www.cnblogs.com/jenry/archive/2012/02/12/2347983.html）

6. [计算机图形学经典算法] Cohen – Sutherland 算法 （附 Matlab 代码） - Holeung blog - CSDN 博客
（https://blog.csdn.net/soulmeetliang/article/details/79179350）

7. Bezier 曲线(1)：Introducion - lyrich 的博客
（https://blog.csdn.net/u013213111/article/details/94067849）

8. B 样条曲线——de Boor 递推算法实现 - HachiLin 的博客
（https://blog.csdn.net/Hachi_Lin/article/details/89812126）