《计算机图形学》系统报告

作者：陈攀岭 学号：171860516

(南京大学 计算机科学与技术系，南京 210093)

摘 要: 经过一个学期的学习，终于完成了图形学的大作业，实现了实验的系统功能基本要求，也完成了用户交互的实现。其中命令行基本功能，有线段、椭圆、多边形、曲线（Bezier 曲线和三次 B 样条曲线）的绘制生成、平移以及缩放功能，线段、多边形和曲线的旋转功能，线段的裁剪功能，以及画布的重置，画笔颜色设置，位图的保存功能。交互式的图形界面在此基础上有一定的拓展增加，更加用户友好并且完善。

关键词: 图形学，算法。

一、开发环境

编程语言	开发平台	GUI 开发框架
C++	Windows 10	QT 5.13.1

二、系统设计

1. 系统主窗口

系统主窗口采用 QT 框架，利用 QMainWindow 类对程序主窗口进行设计，便于使得系统程序主窗口简洁美观，并且后续操作较为方便。通过菜单实现文件的接口、图元生成的选择以及命令行界面/GUI 的转换。代码实现位于“mainwindow.h/cpp/ui”。

2. 画布设计

画布用于继承 QT 窗口类 QWidget，内含三个容器，分别用于存储图元、图层和 id，以及各种操作函数。代码实现位于“canvas.h/cpp”。

3. 命令行界面

创建类 Command。内含序列文件、图像保存目录以及画布，以及构造函数以及文件指令解析函数。代码实现位于“command.h/cpp/ui”。

4. 图像绘制

对于图像的绘制，每个图元都会涉及到生成和变换，对于图元存在的统一性，创建 Primitive 类表示图元的共性，保证不同图形的统一结构。后续每个图元可继承该类生成特定的图元类。代码实现位于“primitive.h/cpp”。

四个图元的子类分别为 Line、Polygon、Ellipse、Curve，代码实现分别位于“line.h/cpp”、

“polygon.h/cpp”、“ellipse.h/cpp”、“curve.h/cpp”。

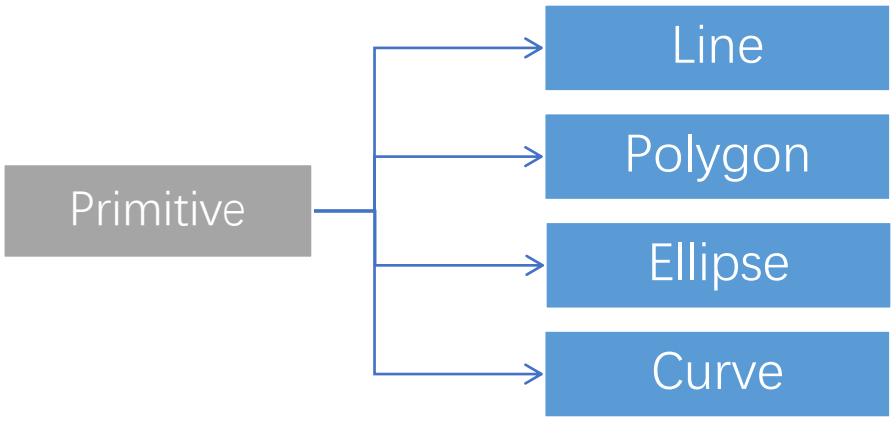

图 1 - 图元继承关系

三、算法实现

1. 图元生成

1. 1. 线段生成算法

1.1.1.数值差分分析——DDA(digital differential analyzer)

（1）算法原理：

利用光栅特性(屏幕单位网格表示像素列阵)，使用 x 或 y 方向单位增量(△x 或△y=±1)来离散取样，并逐步计算沿线路径各像素位置。即先在一个坐标轴上以单位间隔对线段离散取样，再确定另一个坐标轴上最靠近线段路径的对应整数值。根据斜率绝对值与 1 相比的大小，分别选取 x 或 y 作为间隔取样方向。

（2）实现思路：

①得到起点坐标(x1,y1)，终点坐标(x2,y2)，计算得到 x 和 y 方向起点到终点的绝对距离 dx，dy；

②设置变量 e 取|dx|和|dy|中的较大值，在 x、y 方向上的增量分别为 e/dx、e/dy；

③从起点开始绘制，循环绘点直到终点。每一次绘制对 x、y 坐标取整得到点坐标并绘制，然后坐标 x、y 分别加上相应方向的增量，进入下一次循环。

（3）关键代码：

```
void Line::drawDDA(QPainter *painter, int x1, int y1, int x2, int y2)
{
    double dx = x2 - x1;
    double dy = y2 - y1;
    double e = qAbs(dx) > qAbs(dy) ? qAbs(dx) : qAbs(dy);
    double de_x = dx/e;
    double de_y = dy/e;
    double tempx = x1;
    double tempy = y1;
    for(int i = 0; i <= e; i++)
    {
```

```
        QPoint tempP((int)(tempx+0.5),(int)(tempy+0.5));
        painter->drawPoint(tempP);
        tempx += de_x;
        tempy += de_y;
    }
}
```

1.1.2.Bresenham 画线算法

（1）算法原理：

采用整数增量运算，精确而有效的光栅设备线生成算法。根据光栅扫描原理(逐个像素和逐条扫描线显示图形)，在线段离散过程中的每一放样位置上只可能有两个像素更接近于线段路径。因此引入一个整型参量来衡量“两候选像素与实际线路径点间的偏移关系”，通过对整型参量值符号的检测，选择候选像素中离实际线路径近的像素作为线的一个离散点。根据斜率绝对值与 1 相比的大小，分别选取 x 或 y 作为间隔取样方向。

（2）实现思路：

①得到起点坐标(x1,y1)，终点坐标(x2,y2)，计算得到 x 和 y 方向起点到终点的绝对距离 dx，dy；

②设置变量 e 取|dx|和|dy|中的较大值，变量 e_取|dx|和|dy|中的较小值；变量 ddx 取 dx 和 dy 中绝对值较大者，变量 ddy 取 dx 和 dy 中绝对值较小者；变量 ex 和 ey 分别作为单位间隔，符号与 ddx 和 ddy 相同；令 tempx 为间隔取样方向起点坐标，tempy 为起点另一坐标，即若|dx|>|dy|，则从 x 方向取样，且(tempx,tempy)=(x1,y1)，否则，从 y 方向取样，(tempy,tempx)=(x1,y1)；

③计算初始决策参数 p=2*e_-e；

④从起点(tempx,tempy)开始逐点绘制，直到终点。每次循环时，首先判断坐标是否对应，若不对应更改为（tempy,tempx），绘点，根据 p 的正负选择下一点 p 值及 tempy 值：若 p>=0，则 p 更新为 p+2*(e_-e)，tempy=tempy+ey；否则，p 更新为 p+2*e_，tempy 不变。tempx 更新为 tempx+ex。

（3）关键代码：

```
void Line::drawBresenham(QPainter *painter, int x1, int y1, int x2, int y2)
{
    int dx = x2 - x1;
    int dy = y2 - y1;
    int e = qAbs(dx) > qAbs(dy) ? qAbs(dx) : qAbs(dy);
    int e_ = qAbs(dx) > qAbs(dy) ? qAbs(dy) : qAbs(dx);
    int ddx = qAbs(dx) > qAbs(dy) ? dx : dy;
    int ddy = qAbs(dx) > qAbs(dy) ? dy : dx;
    int ex = ddx > 0 ? 1 : -1;
    int ey = ddy > 0 ? 1 : -1;
    int p = 2*e_ - e;
    int tempx = qAbs(dx) > qAbs(dy) ? x1 : y1;
    int tempy = qAbs(dx) > qAbs(dy) ? y1 : x1;
    for(int i = 0; i <= e; i++)
```

```
    {
        QPoint tempP(tempx,tempy);
        if(qAbs(dx)<qAbs(dy))
        {
            tempP.setX(tempy);
            tempP.setY(tempx);
        }
        painter->drawPoint(tempP);
        if(p >= 0)
        {
            tempy += ey;
            p += 2*(e_ - e);
        }
        else
        {
            p += 2*e_;
        }
        tempx += ex;
    }
}
```

1. 2. 多边形生成算法

（1）算法原理：

利用线段生成算法，将多边形的各个顶点按照顺序连接起来，就构成了多边形。

（2）实现思路：

①得到顶点坐标(x1,y1)(x2,y2)...(xn,yn)；

②进行 n 次循环将顶点相连。每次循环，存储多边形顶点（xi,yi），连接(xi,yi)和(xi+1,yi+1)，当循环到(xn,yn)时，连接(xn,yn)和(x1,y1)；连接的线段算法，根据传入算法“DDA”或“Bresenham”分别选择相应的线段生成算法。

（3）关键代码：

```
void Polygon::drawPolygon(QPainter *painter, QString alg, int n, QStringList &list)
{
    int x2,y2;
    int x1 = list[0].toInt();
    int y1 = list[1].toInt();
    for(int i = 0; i < n; i++)
    {
        QPoint temp(x1,y1);
        keyP.push_back(temp);
        if(i == n - 1)
        {
            x2 = list[0].toInt();
```

```
            y2 = list[1].toInt();
            temp.setX(x2);
            temp.setY(y2);
            keyP.push_back(temp);
        }
        else
        {
            x2 = list[i*2+2].toInt();
            y2 = list[i*2+3].trimmed().toInt();
        }
        Line line;
        if(alg == "DDA")
        {
            useAlg = DDA;
            line.drawDDA(painter,x1,y1,x2,y2);
        }
        else if(alg == "Bresenham")
        {
            useAlg = Bresenham;
            line.drawBresenham(painter,x1,y1,x2,y2);
        }
        else
        {
            qDebug()<<"wrong algorithm!";
        }
        x1 = x2;
        y1 = y2;
    }
}
```

1.3. 中点椭圆生成算法

（1）算法原理：

避免平方根运算，直接采用像素与圆距离的平方作为判决依据，通过检验两候选像素中点与圆边界的相对位置关系(圆边界的内或外)来选择像素。标准位置椭圆在四分像限中是对称的，计算一个整四分像限中椭圆曲线的像素位置，再由对称性得到其它三个像限中的像素位置。先确定中心在原点的标准位置的椭圆点(x,y)，然后将点变换为圆心在(xc,yc)的点。

定义椭圆函数为：

$$f_{ellipse}(x,y) = \mathrm{r}_y^2\mathrm{x}^2 + \mathrm{r}_x^2\mathrm{y}^2 - \mathrm{r}_x^2\mathrm{r}_y^2$$

该函数具有下列特性：

$f_{ellipse}(x,y) < 0$，(x,y)位于椭圆边界内；

$f_{ellipse}(x,y) = 0$，(x,y)在椭圆边界上；

$f_{ellipse}(x,y)>0$，(x,y)位于椭圆边界外。

将椭圆函数$f_{ellipse}(x,y)$作为中点算法的决策参数。在每个取样位置，按照椭圆函数在沿椭圆轨迹两个候选像素间中点求值的符号选择下一个像素。

依据椭圆弧切线斜率，第一像限椭圆(ry≤rx)可分成两部分。在每一步中，需检测曲线的斜率值：椭圆的斜率可从椭圆方程中计算出：$dy/dx=-2r^2_yx/2r^2_xy$；在两区域交界处：$dy/dx=-1$，即：$2r2yx=2r2xy$。于是区域 1 和区域 2 的交替条件是：$2r^2_yx\geq 2r^2_xy$。然后根据斜率确定离散取样方向：斜率绝对值小于 1 的区域 1 内沿 x 方向离散取样；斜率绝对值大于 1 的区域 2 内沿 y 方向离散取样。

（2）实现思路：

①得到中心坐标(x,y)，长短轴半径 rx，ry，起点(0,ry)；

②计算得到长短轴半径的平方 sqx，sqy，计算决策参数 p1=sqy-sqx*ry+0.25*sqx，绘图点(tempx,tempy)=(0,ry)；

③开始区域 1 的绘制，从起点开始循环绘点，循环至 sqy*tempx>=sqx*tempy。每次循环，通过 tempx，tempy 以及另外三个象限的对称点平移至(x,y)为中心的椭圆上，并绘点；根据 p1 的正负确定 tempy 及 p1 的更新：若 p1<0，p1 更新为 p1+2*sqy*tempx+3*sqy，tempy 不变；否则，p1 更新为 p1+2*sqy*tempx-2*sqx*tempy+3*sqy+2*sqx，tempy 减 1。Tempx 更新加 1；

④进行区域 2 的绘制，从区域 1 绘制结束点开始，循环至 tempy<0。每次循环，通过 tempx，tempy 以及另外三个象限的对称点平移至(x,y)为中心的椭圆上，并绘点；根据 p2 的正负确定 tempx 及 p2 的更新：若 p2>0，p2 更新为 p2- 2*sqx*tempy + 3*sqx，tempx 不变；否则，p2 更新为 p2-2*sqx*tempy+2*sqy*tempx+3*sqx+2*sqy，tempx 加 1。tempy 更新减 1。

（3）关键代码：

①对称绘点：

```
void Ellipse::drawPoint(QPainter *painter, int x, int y, int px, int py)
{
    QPoint temp1(x+px, y+py);
    QPoint temp2(x+px, y-py);
    QPoint temp3(x-px, y+py);
    QPoint temp4(x-px, y-py);
    painter->drawPoint(temp1);
    painter->drawPoint(temp2);
    painter->drawPoint(temp3);
    painter->drawPoint(temp4);
}
```

②椭圆绘制:

```
void Ellipse::drawEllipse(QPainter* painter, int x, int y, int rx, int ry)
{
    startP.setX(x);
    startP.setY(y);
```

```
    centerP.setX(x);
    centerP.setY(y);
    ra = rx;
    rb = ry;
    double sqx = rx * rx;
    double sqy = ry * ry;
    double p1 = sqy - sqx*ry + 0.25*sqx;
    int tempx = 0;
    int tempy = ry;
    while(sqy * tempx < sqx * tempy)
    {
        drawPoint(painter,x,y,tempx,tempy);
        if(p1 < 0)
        {
            p1 = p1 + 2*sqy*tempx + 3*sqy;
        }
        else
        {
            p1 = p1 + 2*sqy*tempx - 2*sqx*tempy + 3*sqy + 2*sqx;
            tempy--;
        }
        tempx++;
    }
    double p2 = sqy*(tempx + 0.5)*(tempx+0.5) + sqx*(tempy-1)*(tempy-1) - sqx*sqy;
    while(tempy >= 0)
    {
        drawPoint(painter,x,y,tempx,tempy);
        if(p2 > 0)
        {
            p2 = p2 - 2*sqx*tempy + 3*sqx;
        }
        else
        {
            p2 = p2 - 2*sqx*tempy + 2*sqy*tempx + 3*sqx + 2*sqy;
            tempx++;
        }
        tempy--;
    }
}
```

1. 4. 曲线生成算法

1.4.1. Bezier

（1）算法原理：

一般，Bezier 曲线段可拟合任何数目的控制点，控制点的相关位置决定了 Bezier 多项式的次数，一条 n 次 Bezier 曲线被表示成它的 n+1 个控制点的加权和，权是 Bernstein 基函数。

对一般 Bezier 曲线的混合函数形式，给定 Pk=(xk,yk,zk)(k=0,1,2,...,n)共 n+1 个控制点，这些点混合产生下列位置向量 P(u)，用来描述 P0 和 Pn 间的逼近 Bezier 多项式函数的路径(Bezier 曲线)。

$$P(u)=\sum_{k=0}^{n}P_kBEZ_{k,n}(u)\,,\ 0\le u\le 1$$

混合函数$BEZ_{k,n}(u)$是 Bernstein 多项式。

利用 Bernstein 基函数的降(升)阶公式，可使用递归计算得出 Bezier 曲线上点的坐标位置。用递归计算定义 Bezier 混合函数：

$$BEZ_{k,n}(u)=(1-u)BEZ_{k,n-1}(u)+uBEZ_{k-1,n-1}(u)$$

其中，$BEZ_{k,n}(u)=U^k$，$BEZ_{0,k}(u)=(1-u)^k$。

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

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

曲线上的型值点为：$P(u)=P_0^n$。

通过将参数 u 从 0 开始取值，间隔为 dt，直到 u 取值为 1，得到 1/dt 个型值点，将曲线上的型值点两两之间通过线段连接，即可完成曲线的绘制。

例：三次 Bezier 曲线：

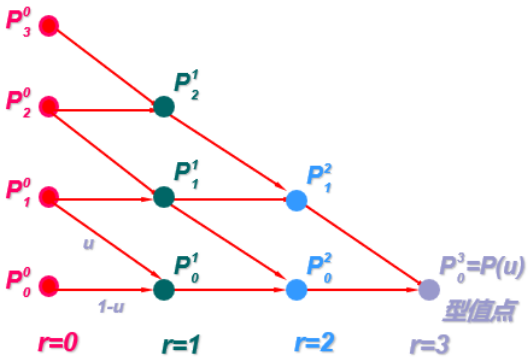

图 2 – 参数 u 下的型值点 P(u)计算过程

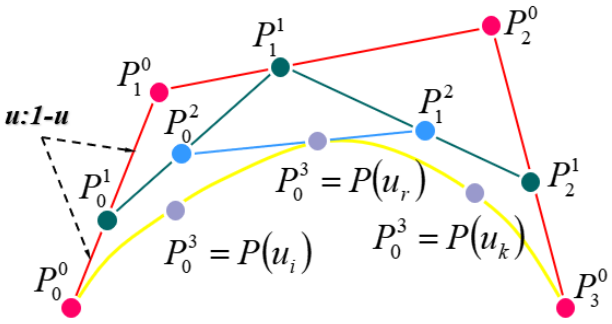

图 3 – Bezier 曲线生成过程

（2）实现思路：

①存入控制点(x1,y1)...(xn,yn)；

②绘制点(tempx,tempy)，初始化为起点(x1,y1)，设置参数为 t，以 0.01 为间隔开始循环计算型值点。每一次循环，存储所有控制点的 x 和 y 坐标值 qx[n]和 qy[n]，使用 de Casteljau 算法计算型值点。将 i 从 0 至 n 进行循环，每次循环中，将 k 从 0 至 n-i 进行循环，每次循环，更新 qx[k]为 qx[k]+t*(qx[k+1]-qx[k])，qy[k]为 qy[k]+t*(qy[k+1]-qy[k])。每次参数 t 得到型值点为(qx[0],qy[0])，画直线连接(tempx,tempy)到(qx[0],qy[0])，然后更新(tempx,tempy)为(qx[0],qy[0])。循环结束后，最后画直线连接点(tempx,tempy)和(xn,yn)。

（3）关键代码：

①存入控制点：

```
void Curve::set(QPainter *painter, int n, QStringList &list)
{
    color = painter->pen().color();
    int x1;
    int y1;
    for(int i = 0; i < n; i++)
    {
        if(i == n - 1)
        {
            x1 = list[i*2].trimmed().toInt();
            y1 = list[i*2+1].trimmed().toInt();
        }
        else
        {
            x1 = list[i*2].toInt();
            y1 = list[i*2+1].toInt();
        }
        QPoint temp(x1,y1);
        keyP.push_back(temp);
        keyTemp.push_back(temp);
    }
}
```

②曲线绘制：

```
void Curve::drawBezier(QPainter *painter)
{
    useAlg = Bezier;

    int tempx = keyP[0].rx();
    int tempy = keyP[0].ry();
    for(double t = 0; t < 1; t += 0.01)
```

```
    {
        double *qx = new double[keyP.size()];
        double *qy = new double[keyP.size()];
        double x,y;
        for(int i = 0; i < keyP.size(); i++)
        {
            qx[i]=keyP[i].rx();
            qy[i]=keyP[i].ry();
        }
        for(int i = 0; i < keyP.size(); i++)
        {
            for(int k = 0; k < keyP.size() - i - 1; k++)
            {
                x = qx[k]+t*(qx[k+1]-qx[k]);
                y = qy[k]+t*(qy[k+1]-qy[k]);
                qx[k]=x;
                qy[k]=y;
            }
        }
        Line line;
        line.drawDDA(painter,tempx,tempy,(int)(qx[0]+0.5),(int)(qy[0]+0.5));
        tempx = (int)(qx[0]+0.5);
        tempy = (int)(qy[0]+0.5);
    }
    Line line_;
    line_.drawDDA(painter,tempx,tempy,keyP[keyP.size()-1].rx(),keyP[keyP.size()-
1].ry());
}
```

1.4.2. B-spline

（1）算法原理：

B 样条曲线是一个分段曲线，一条 k 次 B 样条曲线由 n+1 条 k 次 B 样条分段曲线组成，由节点矢量$U_{n,k+1}=\{u_i\}\ (\mathrm{i}=0,1,2,\ldots,\mathrm{n}+\mathrm{k}+1)$所决定。

B 样条基函数：给定参数 u 轴上的节点分割为$U_{n,k+1}=\{u_i\}\ (\mathrm{i}=0,1,2,\ldots,\mathrm{n}+\mathrm{k}+1)$，称由下列递推关系所确定的$B_{i,k+1}(t)$为$U_{n,k+1}$上的 k+1 阶(或 k 次)B 样条基函数：

deBoox-Cox 递推公式：

$$B_{i,k+1}(u)=\frac{u-u_i}{u_{i+k}-u_i}B_{i,k}(u)+\frac{u_{i+k+1}-u}{u_{i+k+1}-u_{i+1}}B_{i+1,k}(u)\,,(i=0,1,2,\ldots,n)$$

1) $u\in[u_i,u_{i+1})$时，$B_{i,1}(u)=1$；
2) $u\in$其他时，$B_{i,1}(u)=0$；
3) 在上面的递推式中，若遇到 0/0 则取值为 0。

常称u_i为节点，$U_{n,k+1}$为节点向量。若$u_{j-1} < u_j = u_{j+1} = \cdots = u_{j+r-1} < u_{j+r}$，则称从$u_j$到$u_{j+r-1}$的每一个节点为 r 重节点。一条 k 次 B 样条曲线段由$k+1$个控制顶点定义，在不含重节点情况下，每增加一个顶点，曲线的段数就加 1，即$n+1$个顶点定义的 k 次 B 样条曲线有$n-k+1$段，在由顺序$n-k+2$个节点构成的$n-k+1$个节点的区间上，每一曲线段的两端各 k 个节点区间不能作为该曲线段的定义区间，故一条 k 次 B 样条曲线定义域应为：$u \in [u_k, u_{n+1}]$。

对于三次 B 样条曲线的生成，跟 Bezier 曲线生成相似，利用型值点连接的折线作为曲线的近似。

令：

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

$$P_i^r(u) = \begin{cases} P_i ; (r=0) \\ \lambda_i^r(u)P_i^{r-1}(u) + [1-\lambda_i^r(u)]P_{i-1}^{r-1}(u); (r=1,2,\dots k; i=j-k+r+1,\dots j) \end{cases}$$

型值点：

$$P(u) = \sum_{i=j-k+r+2}^{j} P_i^r(u)B_{j,k+1-r}(u) ; (r=0,1,2,\dots,k)$$

于是三次 B 样条曲线函数格式为：

$$P(u) = \sum_{i=0}^{n} P_i(u)B_{i,3}(u) ; u \in [u_3, u_{n+1}]$$

其中，$B_{0,3}(u) = \frac{1}{6}(-t^3+3t^2-3t+1)$；$B_{1,3}(u) = \frac{1}{6}(3t^3-6t^2+4)$；$B_{2,3}(u) = \frac{1}{6}(-3t^3+3t^2+3t+1)$；$B_{3,3}(u) = \frac{1}{6}t^3$。

因此，对于 n 个顶点曲线的绘制，首先分为$n-3$段，逐段绘制。在每一段中，设定参数为 u，根据曲线函数求得型值点逐个连接最终得到曲线。

（2）实现思路：

①存入控制点(x1,y1)...(xn,yn)；

②绘制点(tempx1,tempy1)(tempx2,tempy2)。令 m=n-3，将 i=0 至 3 循环，每一次循环中，将参数 t 从 0 至 1 以 dt=0.01 递增循环，每次循环中通过 ni~ni+3 计算三次 B 样条曲线函数得到型值点(x,y)，第一次循环时令(tempx1,tempy1)为(x,y)，另外则令(tempx2,tempy2)为（x,y），并且连接(tempx1,tempy1)和(tempx2,tempy2)。两层虚幻结束，三次 B 样条绘制完成。

（3）关键代码：

①存入控制点，同 Bezier 曲线

②绘制曲线

```
void Curve::drawBspline(QPainter *painter)
{
    useAlg = Bspline;
    int m = keyP.size()-3;
    double x,y;
```

```
    int tempx1,tempy1,tempx2,tempy2;
    for(int i = 0; i < m; i++)
    {
        for(double t = 0; t < 1; t += 0.01)
        {
            x = ((-pow(t,3)+3*pow(t,2)-3*t+1)/6)*keyP[i].rx() + ((3*pow(t,3)-
6*pow(t,2)+4)/6)*keyP[i+1].rx() + ((-
3*pow(t,3)+3*pow(t,2)+3*t+1)/6)*keyP[i+2].rx() + (pow(t,3)/6)*keyP[i+3].rx();
            y = ((-pow(t,3)+3*pow(t,2)-3*t+1)/6)*keyP[i].ry() + ((3*pow(t,3)-
6*pow(t,2)+4)/6)*keyP[i+1].ry() + ((-
3*pow(t,3)+3*pow(t,2)+3*t+1)/6)*keyP[i+2].ry() + (pow(t,3)/6)*keyP[i+3].ry();
            if(t == 0.0)
            {
                tempx1 = (int)(x+0.5);
                tempy1 = (int)(y+0.5);
            }
            else
            {
                tempx2 = (int)(x+0.5);
                tempy2 = (int)(y+0.5);
                Line line;
                line.drawDDA(painter,tempx1,tempy1,tempx2,tempy2);
                tempx1 = tempx2;
                tempy1 = tempy2;
            }
        }
    }
}
```

2. 图像变换

2. 1. 图元的平移

（1）算法原理

将物体沿直线路径从一个坐标位置到另一个坐标位置的重定位：原始位置为(x,y)，平移距离为 dx 和 dy，平移到新位置(x+dx,y+dy)。

对于不产生形变的刚性物体（即图元），物体上的每一个点都移动相同的坐标。对于线段，平移线段的两个端点；对于多边形，平移每一个顶点；对于椭圆，平移中心点；对于曲线，平移曲线的控制点坐标。再进行图形的重绘。

（2）实现思路：

①对于各种图元，分别移动相应的关键点(x,y)，更新关键点坐标为(x+dx,y+dy)；

②根据关键点，在新坐标对图形进行重绘。

（3）关键代码：

```
    centerP.setX(centerP.rx()+dx);
    centerP.setY(centerP.ry()+dy);
    startP.setX(startP.rx()+dx);
    startP.setY(startP.ry()+dy);
    endP.setX(endP.rx()+dx);
    endP.setY(endP.ry()+dy);            //直线/椭圆
    for(int i = 0; i < keyP.size(); i++)
    {
        keyP[i].setX(keyP[i].rx()+dx);
        keyP[i].setY(keyP[i].ry()+dy);
    }                                   //多边形/曲线
    QPen t;
    t.setWidth(penWidth);
    t.setColor(color);
    painter->setPen(t);
    draw_(painter,startP,endP);         //图形重绘
```

2. 2. 图元的旋转

（1）算法原理

将物体沿 xy 平面内圆弧路径从一个坐标位置到另一个坐标位置的重定位：原视位置为(x,y)，旋转基准点位置(xr,yr)，旋转角 θ (逆时针旋转为正)。

若旋转点为坐标原点，则旋转后(x1,y1)为$x_1 = xcos\theta - ysin\theta$，$y_1 = xsin\theta + ycos\theta$，若为任意基准位置，则类似于以原点旋转后的平移，即$x_1 = x_r + (x - x_r)cos\theta - (y - y_r sin\theta)$，$y_1 = y_r + (x - x_r)sin\theta + (y - y_r)cos\theta$。

对于不变形的刚体变换，物体上所有点均旋转相同的角度。对于直线，旋转两个线端点；对于多边形，旋转每一个顶点；对于曲线，旋转每一个控制点。

（2）实现思路：

①对于各种图元，分别旋转相应的关键点(x,y)，更新关键点坐标。

②根据关键点，在新坐标对图形进行重绘。

（3）关键代码：

```
    double x1 = startP.rx();
    double y1 = startP.ry();
    double x2 = endP.rx();
    double y2 = endP.ry();
    double tempx1;
    double tempx2;
    double tempy1;
    double tempy2;
    tempx1 = x + (x1 - x)*qCos(rad) - (y1 - y)*qSin(rad);
    tempy1 = y + (x1 - x)*qSin(rad) + (y1 - y)*qCos(rad);
```

```
    tempx2 = x + (x2 - x)*qCos(rad) - (y2 - y)*qSin(rad);
    tempy2 = y + (x2 - x)*qSin(rad) + (y2 - y)*qCos(rad);
    startP.setX((int)(tempx1+0.5));
    startP.setY((int)(tempy1+0.5));
    endP.setX((int)(tempx2+0.5));
    endP.setY((int)(tempy2+0.5));                          //直线
    for(int i = 0; i < keyP.size(); i++)
    {
        x1 = keyP[i].rx();
        y1 = keyP[i].ry();
        tempx1 = x + (x1 - x)*qCos(rad) - (y1 - y)*qSin(rad);
        tempy1 = y + (x1 - x)*qSin(rad) + (y1 - y)*qCos(rad);
        keyP[i].setX(tempx1+0.5);
        keyP[i].setY(tempy1+0.5);
    }                                                      //多边形/曲线
    painter->setPen(color);
    draw_(painter,startP,endP);                            //重绘
```

2. 3. 图元的缩放

（1）算法原理

缩放变换改变物体的尺寸，物体上的点移动对缩放点的距离进行相应的等比例改变。

相对于原点的缩放，每个坐标乘缩放系数 s，得到新坐标(x1,y1)；若是固定点的缩放，若固定点为(xf,yf)，则缩放后为$x_1 = x \cdot s + x_f(1-s)$，$y_1 = y \cdot s + y_f(1-s)$。

对于不变形的刚体变换，物体上所有点相对于不动点进行相同的比例 s 缩放。对于直线，缩放两个线端点；对于椭圆，缩放中心点并缩放长短半轴；对于多边形，缩放每一个顶点；对于曲线，缩放每一个控制点。

（2）实现思路：

①对于各种图元，分别缩放相应的关键点(x,y)，更新关键点坐标。

②根据关键点，在新坐标对图形进行重绘。

（3）关键代码：

```
    float x1 = startP.rx();
    float y1 = startP.ry();
    float x2 = endP.rx();
    float y2 = endP.ry();
    float tempx1,tempx2,tempy1,tempy2;
    tempx1 = x1*s + x*(1-s);
    tempy1 = y1*s + y*(1-s);
    tempx2 = x2*s + x*(1-s);
    tempy2 = y2*s + y*(1-s);
    startP.setX(tempx1+0.5);
    startP.setY(tempy1+0.5);
```

```
    endP.setX(tempx2+0.5);
    endP.setY(tempy2+0.5);          //直线
    float x1 = centerP.rx();
    float y1 = centerP.ry();
    float tempx1,tempy1,tempx2,tempy2;
    tempx1 = x1*s + x*(1-s);
    tempy1 = y1*s + y*(1-s);
    tempx2 = ra*s;
    tempy2 = rb*s;
    centerP.setX(tempx1+0.5);
    centerP.setY(tempy1+0.5);
    ra = tempx2+0.5;
    rb = tempy2+0.5;
    endP.setX(centerP.rx() + ra);
    endP.setY(centerP.ry() + rb);
    startP = centerP;                   //椭圆
    float x1,y1,tempx1,tempy1;
    for(int i = 0; i < keyP.size(); i++)
    {
        x1 = keyP[i].rx();
        y1 = keyP[i].ry();
        tempx1 = x1*s + x*(1-s);
        tempy1 = y1*s + y*(1-s);
        keyP[i].setX(tempx1+0.5);
        keyP[i].setY(tempy1+0.5);
    }                                   //多边形/曲线
    painter->setPen(color);
    draw_(painter,startP,endP);         //重绘
```

2. 4. 线段的裁剪

2.4.1. Cohen-Sutherland 算法

（1）算法原理

Cohen-Sutherland 算法是最早、最流行的线段裁剪算法，其核心思想是通过编码测试来减少要计算交点的次数。

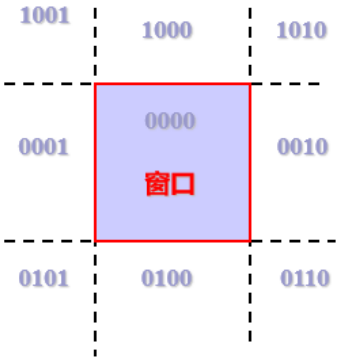

图 4 - Cohen-Sutherland 算法区域码

将线段端点按区域赋以四位二进制码(区域码)，区域码从左至右分别编号为上、下、右、左区域，若落在相应位置则置为 1，否则置 0。

区域码位值通过将端点坐标值(x,y)与裁剪窗口边界坐标比较来确定：$y > yw_{max}$：第 1 位置 1，否则，置 0；$y < yw_{min}$：第 2 位置 1，否则，置 0；$x < xw_{min}$：第 3 位置 1，否则，置 0；$x > xw_{max}$：第 4 位置 1，否则，置 0。

对于可进行位操作语言，区域码可根据两步确定：

①计算端点坐标和裁剪边界之间的坐标差值；

②用各差值符号来设置区域码各位的值：第 1 位为$yw_{max} - y$ 的符号位；第 2 位为$y - yw_{min}$ 的符号位；第 3 位为$x - xw_{min}$ 的符号位；第 4 位为$xw_{max} - x$的符号位。

根据线段端点的区域码可以判断：

①完全在窗口边界内的线段：两端点区域码均为 0000；

②完全在裁剪矩形外的线段：两端点区域码同样位置都为 1；对两个端点区域码进行逻辑与操作，结果不为 0000。

③不能确定完全在窗口内外的线段：进行求交运算，按“左-右-上-下”顺序用裁剪边界检查线段端点。将线段的外端点与裁剪边界进行比较和求交，确定应裁剪掉的线段部分；反复对线段的剩下部分与其它裁剪边界进行比较和求交，直到该线段完全被舍弃或找到位于窗口内的一段线段为止。

（2）实现思路：

①定义 uint8_t 类型变量 b1、b2，作为线段两端点的区域码，初始化为 0；

②通过比较与或运算确定 b1、b2 的值；

③判断 b1&b2 是否为 0，若为 0，则完全在矩形外，两端点都设置为(-1,-1)；

④判断 b1 != 0 || b2 != 0，即部分在矩形内，通过交运算按“左-右-上-下”检查两个端点，并裁剪掉多余线段，即将端点置于矩形边界上；

⑤对于其余情况，即完全在矩形内，端点不变。

⑥新端点确定，对线段重绘。

（3）关键代码：

```
bool Line::clipCS(QPainter *painter, int x1, int y1, int x2, int y2)
{
    int sx = startP.rx();
    int sy = startP.ry();
    int ex = endP.rx();
    int ey = endP.ry();
    double dx = ex - sx;
    double dy = ey - sy;
    uint8_t b1,b2;
    b1 = b2 = 0;
    b1 |= sy > y2 ? 8 : 0;
```

```
    b1 |= sy < y1 ? 4 : 0;
    b1 |= sx > x2 ? 2 : 0;
    b1 |= sx < x1 ? 1 : 0;
    b2 |= ey > y2 ? 8 : 0;
    b2 |= ey < y1 ? 4 : 0;
    b2 |= ex > x2 ? 2 : 0;
    b2 |= ex < x1 ? 1 : 0;
    if((b1&b2) != 0)
    {
        sx = -1;
        sy = -1;
        ex = -1;
        ey = -1;
    }
    else if(b1 != 0 || b2 != 0)
    {
        if((b1&1) == 1)
        {
            sy = (dy/dx)*(x1 - sx) + sy + 0.5;
            sx = x1;
            b1 |= sy > y2 ? 8 : 0;
            b1 |= sy < y1 ? 4 : 0;
        }
        else if((b2&1) == 1)
        {
            ey = (dy/dx)*(x1 - ex) + ey + 0.5;
            ex = x1;
            b2 |= ey > y2 ? 8 : 0;
            b2 |= ey < y1 ? 4 : 0;
        }
        if((b1&2) == 2)
        {
            sy = (dy/dx)*(x2 - sx) + sy + 0.5;
            sx = x2;
            b1 |= sy > y2 ? 8 : 0;
            b1 |= sy < y1 ? 4 : 0;
        }
        else if((b2&2) == 2)
        {
            ey = (dy/dx)*(x2 - ex) + ey + 0.5;
            ex = x2;
            b2 |= ey > y2 ? 8 : 0;
            b2 |= ey < y1 ? 4 : 0;
        }
```

```
        if((b1&4) == 4)
        {
            sx = (dx/dy)*(y1 - sy) + sx + 0.5;
            sy = y1;
        }
        else if((b2&4) == 4)
        {
            ex = (dx/dy)*(y1 - ey) + ex + 0.5;
            ey = y1;
        }
        if((b1&8) == 8)
        {
            sx = (dx/dy)*(y2 - sy) + sx + 0.5;
            sy = y2;
        }
        else if((b2&8) == 8)
        {
            ex = (dx/dy)*(y2 - ey) + ex + 0.5;
            ey = y2;
        }
    }
    startP.setX(sx);
    startP.setY(sy);
    endP.setX(ex);
    endP.setY(ey);
    painter->setPen(color);
    draw_(painter,startP,endP);
    return true;
}
```

2.4.1. Liang-Barsky 算法

（1）算法原理：

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：

$$x_{min} \le x + u\Delta x \le x_{max}$$

$$y_{min} \le y + u\Delta y \le y_{max}$$

这四个不等式可以表示为：$u \cdot p_k \le q_k, (k = 1,2,3,4)$

其中，p、q 的定义为：$p_1 = -\Delta x, q_1 = x_1 - x_{min}$；$p_2 = \Delta x, q_2 = x_{max} - x_1$；$p_3 = -\Delta y, q_3 = y_1 - y_{min}$；$p_4 = \Delta y, q_4 = y_{max} - y_1$。

因此，从上式可以知道，任何平行于窗口某边界的直线，其$p_k = 0$，k 值对应于相应的边界（k=1,2,3,4）对应于左、右、下、上边界）：如果$q_k < 0$，则线段完全在边界外，应舍弃该线段；如果$q_k \ge 0$则线段平行于窗口某边界并在窗口内。当$p_k < 0$时，线段从裁剪边界延长线的外部延伸到内部；当$p_k > 0$时，

线段从裁剪边界延长线的内部延伸到外部。

当$p_k \neq 0$时，可以计算出参数 u 的值，它对应于无限延伸的直线与延伸的窗口边界 k 的交点：$u = q_k/p_k$。对于每条直线，可以计算出参数 u1 和 u2，该值定义了位于窗口内的线段部分。u1 的值由线段从外到内遇到的矩形边界所决定（$p_k < 0$），对这些边界计算$r_k = q_k/p_k$，u1 取 0 和各个 r 值之中的最大值；u2 的值由线段从内到外遇到的矩形边界所决定（$p_k > 0$），对这些边界计算$r_k = q_k/p_k$，u2 取 1 和各个 r 值之中的最小值；如果 u1>u2，则线段完全落在裁剪窗口之外，应当被舍弃；否则，被裁剪线段的端点可以由 u1 和 u2 计算出来。

裁剪过程：

①参数初始化：线段交点初始参数分别为：u1=0，u2=1；

②定义判断函数，用 p、q 来判断：是舍弃线段？还是改变交点参数 r：p＜0，参数 r 用于更新 u1；p＞0，参数 r 用于更新 u2。若更新 u1 或 u2 后，使 u1＞u2，则舍弃该线段；否则，更新 u 值仅仅是求出交点、缩短线段。

③求解交点参数：测试四个 p、q 值后，若该线段被保留，则裁剪线段的端点由 u1、u2 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。

（2）实现思路：

①初始化 u1,u2,p[0~4],q[0~4]，flag 标识是否完全被舍弃。

②四次循环测试 p，q。每次循环，首先计算 r=q[i]/p[i]，判断 p[i]<0，更新 u1 为 u1 和 r 的大值，若 u1>u2，舍弃线段；否则，判断 p[i]>0，更新 u2 为 u2 和 r 的小值，若 u1>u2，舍弃线段；否则，判断 p[i]==0 && q[i]<0，若为真，舍弃线段。

③四次循环后，若线段未舍弃，根据 u1 和 u2 计算得到两个端点的坐标，进行重绘。

（3）关键代码：

```
bool Line::clipLB(QPainter *painter, int x1, int y1, int x2, int y2)
{
    int sx = startP.rx();
    int sy = startP.ry();
    int ex = endP.rx();
    int ey = endP.ry();
    double u1 = 0;
    double u2 = 1;
    double p[4],q[4];
    p[0] = sx - ex;
    p[1] = -p[0];
    p[2] = sy - ey;
    p[3] = -p[2];
    q[0] = sx - x1;
    q[1] = x2 - sx;
    q[2] = sy - y1;
```

```
    q[3] = y2 - sy;
    double r;
    bool flag = false;
    for(int i = 0; i < 4; i++)
    {
        r = q[i]/p[i];
        if(p[i] < 0)
        {
            u1 = u1 > r ? u1 : r;
            if(u1 > u2)
            {
                flag = true;
                break;
            }
        }
        else if(p[i] > 0)
        {
            u2 = u2 < r ? u2 : r;
            if(u1 > u2)
            {
                flag = true;
                break;
            }
        }
        else if(p[i] == 0 && q[i] < 0)
        {
            flag = true;
            break;
        }
    }
    if(flag)
    {
        startP.setX(-1);
        startP.setY(-1);
        endP.setX(-1);
        endP.setY(-1);
    }
    else
    {
        startP.setX(sx - (sx - ex)*u1 + 0.5);
        startP.setY(sy - (sy - ey)*u1 + 0.5);
        endP.setX(sx - (sx - ex)*u2 + 0.5);
        endP.setY(sy - (sy - ey)*u2 + 0.5);
    }
```

```
    painter->setPen(color);
    draw_(painter,startP,endP);
    return true;
}
```

3. 程序接口

（1）实现思路：

在“main.cpp”中，通过判断命令行参数个数确定进入的程序功能。若参数大于 1，则进入命令行模式；否则，直接进入 GUI 模式。

（2）关键代码：

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    if(argc > 1)                        //命令行模式
    {
        Command cmd(argv[1], argv[2]);
        cmd.parsing();
    }
    else                                //GUI 交互模式
    {
        MainWindow w;
        w.show();
        return a.exec();
    }
}
```

四、实验问题及收获

整个实验从最开始到结束几乎是一整个学期的时间，其中也是不停的修修改改甚至是推倒重来。其中的过程肯定不能说是很如意的，其中确实遇到了不少的问题，通过了解并解决这些问题，也更好的理解了实验内容以及课堂的学习内容，也有了一定的收获。

①对于算法的理解：有好些算法都不是一下子就能搞懂的，算法的原理和函数的计算是较为晦涩的。要将其理解并且通过代码实现，单单靠书本或者课件内容很难搞懂。因此就需要查询资料及技术博客或者是询问同学们对疑问进行解决。通过一次次的查阅及询问，得以对算法原理有了更加深入的理解。

②对于 QT 的使用：有一说一，这次实验是我第一次使用 QT 完成项目。因为对于 QT 的陌生，实验刚开始也走了一些弯路，但是也是一次次的失败让我逐渐了解了 QT，并最后完成了此次实验。这一个从无到有的过程，了解了新的生产工具，学习了新的知识。

③对构建项目的理解：整个项目逐步实现的过程中，随着项目逐渐完善，代码量逐渐增长，程序的测试变得繁琐，因此更加注意程序语言的规范性，对于代码以及数据结构进行更多的思考。开始追求代码更为简洁高效，程序设计应当具有更远的前瞻性，考虑周全，因此事先足够的设计准备才会有事半功倍的效果。否则，像我自己这次实验粗略的开始导致了事倍功半的效果。

五、结束语

整个实验的中心内容为课堂教授的图形算法，通过自己的实验，巩固了自己所学习的理论知识，也通过这一次实验，了解到实践与理论的区别。对于知识的掌握，所需要的就是理论与实践的相互结合，实验让我对于所学习内容有了更多更新更深刻的理解。同时，也感谢老师以及助教们教学答疑的辛苦工作，以及实验过程中同学和技术博客们的施以援手。

参考文献：

[1]孙正兴.计算机图形学教程

[2]课堂 PPT 内容

[3] 【Qt5 开发及实例】19、一个简单的画图程序：
https://blog.csdn.net/cutter_point/article/details/43087497

[4] 算法系列之十三：椭圆的生成算法：
https://blog.csdn.net/orbit/article/details/7496008

[5] Bezier 曲线原理：
https://blog.csdn.net/joogle/article/details/7975118

[6] 二次与三次 B 样条曲线 c++实现：
https://blog.csdn.net/jiangjjp2812/article/details/100176547