《计算机图形学》12 月报告

徐国栋 171860633*

(南京大学 计算机科学与技术系，南京 210093)

摘 要: 这是为 2019 年秋学期计算机图形学课程的大作业编写的系统报告，包含进度报告、算法重述和应用设计三个部分。大作业中实现了图元的生成和编辑算法，实现文件交互和用户交互接口。应用设计上，我完成了满足文件接口要求的命令行程序，完成了基于鼠标点击的图元绘制、基于鼠标拖曳的图元移动，和基于可视化锚点及鼠标滚轮的图元旋转、图元缩放，并将控制逻辑集成到 GUI 应用中。最终使用静态编译对 Qt 应用程序进行打包.

关键词: 图元生成算法；图元编辑算法；图形视图框架；图元管理

1 进度报告

在算法模块方面，实现了直线和多边形的 DDA、Bresenham 算法；实现了中点圆和中点椭圆算法；实现了图元平移、缩放、旋转和两种裁剪算法；实现了 n 阶贝塞尔曲线和三次均匀 B 样条算法。

在文件输入接口方面，实现了一个命令行程序，支持解析固定格式的字符串命令。在用户交互接口方面，提供基于鼠标点击的直线、多边形、椭圆、曲线的绘制和实时渲染；实现了基于链表遍历的图元捕获，提供基于鼠标拖曳的图元移动操作；提供基于可视化锚点及鼠标滚轮的图元旋转、图元缩放操作.

2 算法重述

2.1 DDA算法

DDA 算法，即数值差分分析算法，直接利用直线 x 或 y 方向增量△x 或△y，在直线投影较长的坐标轴上，以单位增量对线段离散取样，确定另一个坐标轴上最靠近线段路径的对应整数值。实际实现时，采用增量法确定这个整数值，另一个坐标轴上的增量应满足的要求是，符号使起始点具备向结束点移动的趋势，模长等于当前坐标轴投影和较长坐标轴投影的比值。

代码实现如下：

```
void drawLineByDDA(int x1, int y1, int x2, int y2, QImage* _img, const QRgb _color){
    int stepx(1), stepy(1);                     // 设置增量的正负
    if (x1 > x2) stepx = -1;
    if (y1 > y2) stepy = -1;
    if (x1 == x2) {                             // 针对竖直线的优化
        while (y1 != y2) {
            setPix(QPoint(x1, y1), _color);
            y1 += stepy;
```

```
        }
        setPix(QPoint(x1, y1), _color);
        return;
    }
    else if (y1 == y2) {                                    //  针对水平线的优化
        …
    }                                                       //  下面实现数值差分分析算法
    double dx(x2 - x1), dy(y2 - y1);                        //  用于计算单位增量
    double x(fabs(dx)), y(fabs(dy));
    double length(x > y ? x : y);                           //  获得取样方向
    dx /= length;     dy /= length;                         //  获得单位增量
    x = x1; y = y1;                                         //  获得起始点位置
    int i(length);
    while ((i--) >= 0) {                                    //  循环控制
        setPix(QPoint(round(x), round(y)), _color);
        x += dx; y += dy;
    }
    }.
```

2.2 Bresenham算法

Bresenham 算法利用了光栅扫描时，线段离散取样位置的有限性，只有两个可能的位置符合采样要求，于是设计整型参量来表示两个侯选位置和理想位置的偏移量，通过检测这个整型参量的符号，在侯选位置里二选一。

算法推导如下：

对斜率 $0<m<1$ 的情况，$y_{k+1}=mx_{k+1}+b$，

比较 y_{k+1} 和 y_k、y_k+1 的偏移，$d1 = y - y_k = m(x_k + 1) + b - y_k$, $d2 = y_{k+1} - m(x_k + 1) - b$，

有 $\Delta x(d1 - d2) = 2m(x_k + 1) - 2y_k + 2b - 1$，

设置决策参数 $p_k= \Delta x(d1 - d2)$，

p_k 大于 0 意味着 y_k+1 比 y_k 更接近理想位置。

计算 p_{k+1} 和 p_k 的差，可知，

p_k 大于 0 取高像素 y_k+1 时的增量为 $2\Delta x-2\Delta y$，p_k 小于 0 取低像素 y_k 时的增量为 $2\Delta y$。

代码实现如下：

```
void drawLineByBresenham(int x1, int y1, int x2, int y2, QImage* _img, const QRgb _color) {
    int stepx(1), stepy(1);
    if (x1 > x2) stepx = -1;
    if (y1 > y2) stepy = -1;
    if (x1 == x2) {                 //  针对竖直线的优化
        …
    }
    else if (y1 == y2) {            //  针对水平线的优化
        …
    }
    int x(x1), y(y1), dx(abs(x2 - x1)), dy(abs(y2 - y1));
```

```
if (dx == dy) {                          //  针对对角线的优化
    while (x != x2) {
        setPix(QPoint(x, y), _color);
        x += stepx; y += stepy;
    }
}                                        //  正式开始Bresenham算法
else if (dx > dy) {
    int p(2 * dy - dx), twody(2 * dy), twody_2dx(2 * (dy - dx)), i(dx);
    while ((i--) >= 0) {
        setPix(QPoint(x, y), _color);
        x += stepx;
        if (p < 0)
            p += twody;
        else {
            p += twody_2dx; y += stepy;
        }
    }
}
else {                                    //  所有变量反演
    …
}
}
```

从实现上看，Bresenham 算法不需要对浮点数取整，不存在 DDA 算法因取整造成的整体偏差。

在性能方面，因为现在的 CPU 性能挺好，很难看出 DDA 和 Bresenham 算法在用户体验方面的差异，在 Qt 应用的主线程中分别运行 DDA 和 Bresenham 算法来绘制直线和多边形，并且调用 update 函数立即渲染，肉眼无法察觉鼠标快速拖曳时，窗体画面的延时。

2.3 中点圆和中点椭圆算法

2.3.1 中点圆算法

决策参数和增量的推导类似 Bresenham 算法，推导如下：

定义圆函数：

$f_{circle}(x, y) = x^2 + y^2 – r^2$

圆边界上的点(x, y)满足 $f_{circle}(x, y) = 0$

任意点(x, y)与圆周的相对位置关系可由对圆函数符号的检测来决定：

① 若 $f_{circle}(x, y) < 0$，(x, y)位于圆边界内；

② 若 $f_{circle}(x, y) = 0$，(x, y)位于圆边界内；

③ 若 $f_{circle}(x, y) > 0$，(x, y)位于圆边界外。

第 k 个决策参数是圆函数在两候选像素中点处求值，

$p_k = f_{circle}(x_{k+1}, (y_{k+1} + y_k) / 2)$ 其中 $y_{k+1}= y_k-1$ 所以 $p_k = f_{circle}(x_{k+1}, y_k – 1/2)$

$p_k < 0$，中点在圆周边界内，选择像素位置(x_{k+1}, y_k)；

$p_k > 0$，中点位于圆周边界外，选择像素位置(x_{k+1}, y_{k-1});

p_k 符号决定两候选像素中点位置$(y_{k+2} + y_{k+1}) / 2$ 的取值，

若 $p_k < 0$，$(y_{k+2} + y_{k+1}) / 2 = y_k - 0.5$，即 $p_{k+1} = f_{circle}(x_{k+2}, y_k - 0.5)$;

若 $p_k > 0$，$(y_{k+2} + y_{k+1}) / 2 = y_k - 1.5$，即 $p_{k+1} = f_{circle}(x_{k+2}, y_k - 1.5)$。

只需要计算八分之一圆弧，另外七个圆弧通过对称、对映操作得到坐标。

代码实现：

```
if (rx == ry) {                                   //  标准圆算法
        int x(0), y(rx), p(3 - 2 * rx);   //  控制增量
        while (x <= y) {
            setPix(QPoint(x0 + x, y0 + y), _color);
            setPix(QPoint(x0 - x, y0 - y), _color);
            setPix(QPoint(x0 + x, y0 - y), _color);
            setPix(QPoint(x0 - x, y0 + y), _color);
            setPix(QPoint(x0 + y, y0 + x), _color);
            setPix(QPoint(x0 - y, y0 + x), _color);
            setPix(QPoint(x0 - y, y0 - x), _color);
            setPix(QPoint(x0 + y, y0 - x), _color);
            if (p >= 0) {
                p += 4 * (x - y) + 10; y--;
            }
            else
                p += 4 * x + 6;
            x++;
        }
    }
```

2.3.2　中点椭圆算法

椭圆的对称性比圆要弱一些，决策参数和增量在圆周斜率在过 1 时要进行调整，采用计算四分之一圆周，对称、对映出另外四分之三圆周的方案。另外，在每次步进之后，都要重新计算斜率，来判断是否更换决策参数和增量。

代码实现：

```
if (rx > ry) {                                      //  中点椭圆算法
        int x(0), y(ry);
        double pk(0);
        int ry2(ry * ry), rx2(rx * rx), rx2ry2(rx2 * ry2);
        setPix(QPoint(x0 + x, y0 + y), _color);
        setPix(QPoint(x0 - x, y0 - y), _color);
        setPix(QPoint(x0 + x, y0 - y), _color);
        setPix(QPoint(x0 - x, y0 + y), _color);
        pk = ry2 - rx2 * ry + rx2 / 4.0;
        while (ry2 * x <= rx2 * y) {
            x++;
            if (pk < 0)
                pk += (2 * ry2 * x + ry2);
            else {
                y--; pk += (2 * ry2 * x - 2 * rx2 * y + ry2);
            }
```

```
            setPix(QPoint(x0 + x, y0 + y), _color);
            ...
        }
        pk = ry2 * (x + 0.5) * (x + 0.5) + rx2 * (y - 1.0) * (y - 1.0) - rx2ry2;
        while (y > 0) {
            y--;
            if (pk > 0)
                pk += (-2 * rx2 * y + rx2);
            else {
                x++; pk += (2 * ry2 * x - 2 * rx2 * y + rx2);
            }
            setPix(QPoint(x0 + x, y0 + y), _color);
            ...
        }
    }
    else {
        swap(x0, y0); swap(rx, ry);                  //  先反演所有坐标
        int x(0), y(ry);                             //  再执行 rx > ry 的中点椭圆算法
        ...
    }
```

2.4 图元编辑算法

2.4.1　图元平移

二维平面上的图元平移可通过二维向量的加减运算来描述，对于控制点(x0,y0)，平移(x,y)即意味着平移到(x0+x,y0+y)。编程的时候需注意，椭圆的实轴和虚轴长度不是控制点，不能参与平移计算。

2.4.2　图元旋转

对于将控制点缓冲中的点逆时针绕(x,y)旋转角度制 r 的变化，可以通过以下函数描述：

```
        const double pi = 3.1415926;
        double cosr(cos(r * pi / 180.0)), sinr(sin(r * pi / 180.0));
        for (auto& i : ctrlbuffer) {
            int x0 = i.x(), y0 = i.y();
            i.setX(x + (x0 - x) * cosr - (y0 - y) * sinr);
            i.setY(y + (x0 - x) * sinr + (y0 - y) * cosr);
        }
```

推导的方式是设出两条射线和水平轴的夹角 r、r+x 和半径 h，dx=h*cos(r+x)，利用三角公式展开，利用原射线和水平轴夹角 x 的三角函数值，即坐标(x0,y0)，替换掉 h 和关于 x 的三角函数，即得到上面的函数表达式。

2.4.3　图元缩放

对于同一直线上的三个点 A(Xi,Yi)、B(X,Y)、C(a,b)，对于水平方向，设放缩比例为 S_x，做的是 A 以 B 为中心向 C 的缩放，有比例关系(Xi-X)*Sx=(a-X)，可以通过以下函数描述：

```
        for (auto& i : ctrlbuffer) {
            i.setX(x + (i.x() - x) * sx);
            i.setY(y + (i.y() - y) * sy);
```

```
}
```

2.4.4 CohenSutherland 裁剪算法

对目标点做四个方向九个区域的编码测试，用四个比特位表达目标点在九个区域中的哪一个，然后计算射线和目标点靠近的边框的交点，替换目标点，直到两端的目标点落在边框内，或都不可能落在边框内，结束算法。编码的策略如下：

```
short code(0b0000);
if (point.y() > y2)
    code |= 0b0001;
if (point.y() < y1)
    code |= 0b0010;
if (point.x() < x1)
    code |= 0b0100;
if (point.x() > x2)
    code |= 0b1000;
```

计算交点的策略如下：

```
if ((code & 0b0100)) {
    p.setX(round(xmin));
    p.setY(round(a.y() + (p.x() - a.x()) * m));
}
else if ((code & (0b1000))) {
    p.setX(round(xmax));
    p.setY(round(a.y() + (p.x() - a.x()) * m));
}
else if ((code & (0b0001))) {
    p.setY(round(ymax));
    p.setX(round(a.x() + (p.y() - a.y()) / m));
}
else if ((code & (0b0010))) {
    p.setY(round(ymin));
    p.setX(round(a.x() + (p.y() - a.y()) / m));
}
```

这里的 m 表示两个端点构成的直线的斜率，这个斜率可能不存在。为了方便编写代码，我计算 m 的方法是：

```
double m = (q.y() - p.y()) / (q.x() - p.x() + 0.000000000001);
```

为了避免整形舍入的误差，计算交点时使用 round 函数来避免完全的向下舍入。

2.4.5 LiangBarsky 裁剪算法

梁友栋算法的中间目标是，假设线段可以被裁减，那么就去获得裁剪后线段端点关于原来端点的增量。设计参数 p1=-Δx，q1=x1-xmin， p2=Δx， q2=xmax-x1 ，p3=-Δy， q3=y1-ymin， p4=Δy， q4=ymax-y1，如果 qk<0，则线段完全在边界外，应舍弃该线段；如果 qk≥0，则线段平行于窗口某边界并在窗口内；如果 pk<0 时，线段从裁剪边界延长线的外部延伸到内部；如果 pk>0 时，线段从裁剪边界延长线的内部延伸到外部；根据参数 p、q 更新交点参数 u1，u2。u1、u2 表达线段端点关于原来端点的增量和线段投影的比值。代码实现如下：

```
int x(ctrlp[0].x()),y(ctrlp[0].y()),
```

```
    xend(ctrlp[1].x()),yend(ctrlp[1].y());
double dx(xend - x);
double dy(yend - y);
double p[4];
p[0] = -dx; p[1] = -p[0];
p[2] = -dy; p[3] = -p[2];
double q[4];
q[0] = x - x1; q[1] = x2 - x;
q[2] = y - y1; q[3] = y2 - y;
double u, u1(0), u2(1);
for (size_t k = 0; k < 4; k++) {
    u = q[k] / p[k];
    if (p[k] < 0) {
        if (u > u2)
            return;//  舍弃
        if (u > u1)
            u1 = u;
    }
    else if (p[k] > 0) {
        if (u < u1)
            return;
        if (u < u2)
            u2 = u;
    }
    else if (q[k] < 0)
        return;
}
ctrlp[0] = QPoint(x + u1 * dx, y + u1 * dy);
ctrlp[1] = QPoint(x + u2 * dx, y + u2 * dy);
```

2.5 曲线生成算法

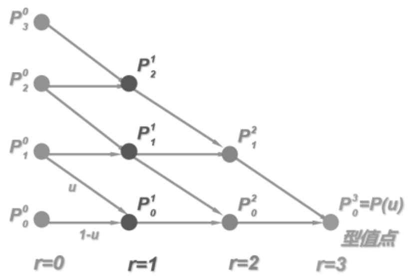
2.5.1 n 阶贝塞尔曲线

使用 de Casteljau 算法，用迭代的方式生成所有型值点。根据公式：

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

设有 n 个控制点，对于[0,1]中的每一个参数 t，需要做(n-1)次线段的 t 比例分割，第 i 次分割会产生(n-i)个中间型值点，第(n-1)次分割可以得到 1 个型值点，这个点就是需要的最终型值点。

算法举例如下：对于 4 个控制点，迭代 3 次获得一个最终型值点：



代码实现如下：

```
        vector<QPointF> p; p.assign(input.begin(), input.end());
        QPointF tmp = p[0]; //  为了避免误差累积，全程使用浮点数计算
        int div = sqrt(n); if (div < 1)div = 1;// 根据控制点个数调整步长
        for (double t = 0; t <= 1 + 0.000000001; t += 0.01 / div) {
            p.assign(input.begin(), input.end());
            for (int i = 1; i < n; i++) {          // 外层循环n-1次，即做n-1次t比分
                for (int j = 0; j < n - i; j++) { //每层循环计算出n-1,n-2,...,1个切分点
                    p[j] = (1.0 - t) * p[j] + t * p[j + 1];
                }
            }
            drawLineByBresenham( tmp.x(), tmp.y(), p[0].x(), p[0].y(), buffer, false);
            tmp = p[0];
    }
```

通过 div 参数来控制参数 t 的步长，避免曲线过长（控制点过多）时，步长太小导致的出现折线的问题。

编程的过程中需要注意，必须使用浮点数做中间运算，否则迭代的过程中，整型变量会发生连续舍入，使得部分曲线呈现阶梯状的特点。

2.5.2　三次均匀 B 样条

使用 de Boor-Cox 算法，对于 k 次的 B 样条基函数，构造一个递推的公式，由 0 次多项式的递推构造 1 次的，1 次的递推构造 2 次……递推公式如下：

$$B_{i,1}(u)=\begin{cases}1 & u_i<x<u_{i+1}\\ 0 & Otherwise\end{cases}$$

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

一阶的多项式涉及一个区间两个节点，K 阶的 $B_{i,k}$ 涉及 k 个区间 k+1 个节点。

代码实现如下：

1、递归函数

```
    double Proc::bspline(double* U, double u, int i, int k) {
        double result;
```

```
        if (k == 1) {
            if (U[i] < u && u < U[i + 1]) result = 1;
            else result = 0;
        }
        else {//  用条件语句体现约定： 0/0=0
            result = 0;
            if (i + k - 1 != i)// 要求 U[i + k - 1] - U[i] ! = 0
                result += (u - U[i]) / (U[i + k - 1] - U[i]) * bspline(U, u, i, k - 1);
            if (i + k != i + 1)// 要求 U[i + k] - U[i + 1] ! = 0
                result += (U[i + k] - u) / (U[i + k] - U[i + 1]) * bspline(U, u, i + 1, k - 1);
        }
        return result;
    }
```

2、参数的步长迭代

```
        for (int i = 0; i < n + k + 1; i++)
            U[i] = i;
        ……
        for (double u = U[k - 1]; u < U[n + 1]; u += 0.01 / div) {
            QPointF curP(0, 0);
            for (int i = 0; i < n + 1; i++)
                curP += input[i] * bspline(U, u, i, k);
            if (fabs(curP.x()) > 0.0001 || fabs(curP.y()) > 0.0001)
                tmpBuf.push_back(curP);
        }
```

对于公式中 U 的取值，只要保证基函数系数的分子分母数量级一致即可，所以这里直接用区间段的索引给 U 赋值。迭代中进行额外的判断，避免两端处，(0,0)被加入型值点序列。

3 应用设计

以 Qt 为编程框架，C++为编程语言，程序分为三个模块：图形学算法、命令行交互和手绘板交互。

图形学算法方面，将所有图元生成算法以静态成员函数的形式封装在 Proc 类中，在这些函数里实现上述算法，采用面向过程的风格，公共接口设计如下：

```
        /*向buffer填充构成直线(x1,y1)-(x2,y2)的点，clear变量控制是否清空帧缓存*/
        static void drawLineByDDA(
            int x1, int y1, int x2, int y2, std::vector<QPoint>& buffer, bool clear = true
        );
        static void drawLineByBresenham(
            int x1, int y1, int x2, int y2, std::vector<QPoint>& buffer, bool clear = true
        );
        /*向buffer填充构成多边形{xi,yi}的点*/
        static void drawPolygonByDDA(
            const std::vector<int>& xs, const std::vector<int>& ys, std::vector<QPoint>& buffer
        );
        static void drawPolygonByBresenham(
```

```
    const std::vector<int>& xs, const std::vector<int>& ys, std::vector<QPoint>& buffer
);
/*向buffer填充构成椭圆{xi,yi}的点*/
static void drawEllipse(int x0, int y0, int rx, int ry, std::vector<QPoint>& buffer);
/*向buffer填充构成贝塞尔曲线{xi,yi}的点*/
static void drawCurveByBezier(
    const std::vector<int>& xs, const std::vector<int>& ys, std::vector<QPoint>& buffer
);
/*向buffer填充构成B样条曲线{xi,yi}的点*/
static void drawCurveByBSpline(
    const std::vector<int>& xs, const std::vector<int>& ys, std::vector<QPoint>& buffer
);
/*修改ctrlp为包含在矩形(x1,y1)(x2,y2)中的线段端点*/
static void clipByCohenSutherland(int x1, int y1, int x2, int y2, std::vector<QPoint>& ctrlp);
static void clipByLiangBarsky(int x1, int y1, int x2, int y2, std::vector<QPoint>& ctrlp);
/*将ctrlbuffer中的点平移(x,y)，这里的ctrlbuffer是控制点，例如直线的端点，椭圆的中心等*/
static void translate(int x, int y, std::vector<QPoint>& ctrlbuffer);
/*将ctrlbuffer中的点以(x,y)为中心顺时针旋转角度r，这里的ctrlbuffer是控制点*/
static void rotate(int x, int y, int r, std::vector<QPoint>& ctrlbuffer);
/*将ctrlbuffer中的点以(x,y)为中心放缩s，这里的ctrlbuffer是控制点，例如直线的端点，椭圆的中心等*/
static void scale(int x, int y, float sx, float sy, std::vector<QPoint>& ctrlbuffer);
```

命令行交互方面，在 class Cli 中解析命令，调用 Proc 提供的算法，公共 API 设计如下：

```
bool handleCmd(std::string _cmd = std::string("resetcanvas 100 100"));
bool handleScript(const char* filename = "");
```

手绘板交互方面，通过在手绘面板 class ScribbleArea 中拦截四种鼠标事件，完成用户输入的获取，调用 Proc 类提供的图元生成算法，将结果实时渲染到窗体上。

GUI 的涂鸦功能的实现细节在此不再赘述，下面介绍图元编辑的实现。

对于图元编辑的 GUI 交互，采用捕捉被点击图元的方法，为当前所有可见图元构造矩形框，存储在一个链表中，在 mouseMoveEvent 中捕捉满足 QRect::contains(QPoint)的鼠标点，对符合要求的图元的矩形框做特殊标注，意味着鼠标捕获了目标图元。

考察 Qt 使用的图形视图框架，内部通过 BSP 树实现鼠标和图元的快速对应。事实上，当二维空间中的图元数量达到一定数量级，像我目前这样遍历链表而用 Rect.contains(Point)的做法捕获图元是极为缓慢的。GUI 框架大多通过树形结构比如 BSP 树、4 叉树来从坐标索引图元。对于画板，考虑到可能的交互强度，使用链表遍历来查找图元，延时是完全可以接受的。

在拥有了从鼠标点击索引图元的实现以后，对于图元平移，记录图元的原始位置和当前鼠标位置，每一次鼠标移动，先把图元移回初始位置，再渲染到当前鼠标位置，从用户界面观察，相当于自己在拖曳图元。

对于缩放和旋转，大作业的要求和 word 文档、ppt 等软件提供的交互逻辑有所出路，要求围绕固定点做缩放和旋转，所以我设计了基于可视化锚点的交互逻辑，点击功能按钮后，会要求用户放下一个图钉形状的锚点，接下来用户点击图元，实现图元指定，转动鼠标滚轮，通过滚轮的前进和后退，映射到缩放比例(>1 或<1)和旋转角度(顺时针或逆时针)。

提供基于鼠标滚轮的缩放和旋转，需要解决的问题有精度问题，因为图元控制点是用整型数记录的，连续对一个图元做几十上百次浮点精度的变形，控制点的相对位置会发生扭曲，累计误差不可接受。为了解决这个缺陷（根据大作业要求，控制点坐标用整数表示），我采用先把图元恢复到初始位置，再重新渲染的方式，来消除对同一个图元连续操作时的误差累计。

致谢 在此，我向教授计算机图形学的张岩老师、孙正兴老师，以及致力于提高 GUI 编程体验的 Qt 开源社区表示感谢.

References:

[1] 计算几何-求线段交点算法和代码 https://blog.csdn.net/tengchongwei/article/details/72922056 .

[2] [Qt]状态栏 QStatusBar 使用 https://blog.csdn.net/humanking7/article/details/88065425 .

[3] [简书] Qt 之图形视图框架 https://www.jianshu.com/p/3f33fae46f96 .

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

[5] [简书] 必须要理解掌握的贝塞尔曲线 https://www.jianshu.com/p/0c9b4b681724 .