系统技术报告

作者姓名：刘潇远

学号 161220083

1、 综述

图形学大作业旨在实现简单的图形学绘制算法，主要涉及直线、曲线、多边形等二维图形的绘制、编辑、裁剪、变换和储存，还涉及到了三维模型(OFF 格式)的载入和显示。

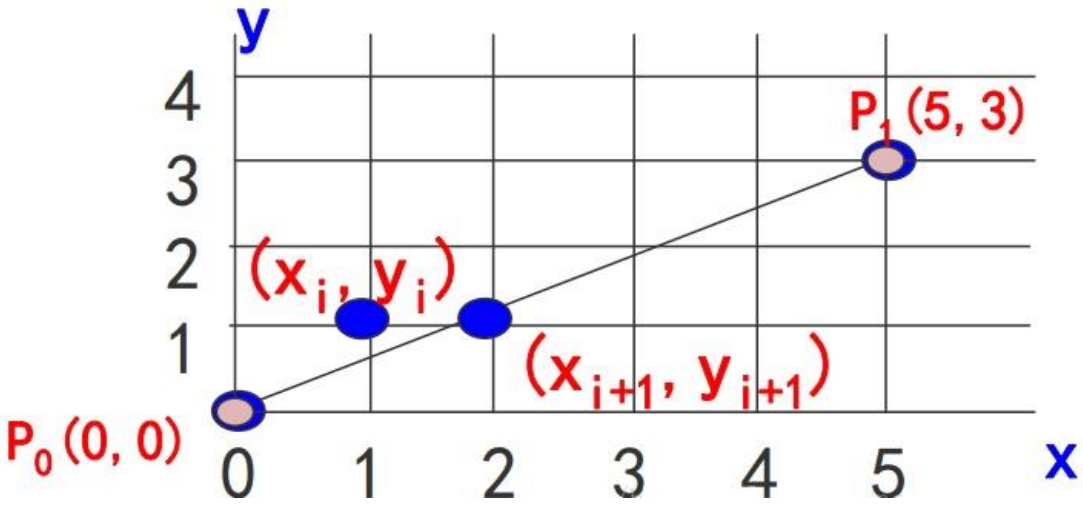
2、 系统介绍

系统提供了直线、曲线、多边形和填充区域的绘制和编辑功能，直线的裁剪功能，二维图像的平移、旋转、缩放功能和二维图像的储存功能，还提供过了 OFF 文件格式的三维模型的载入和显示。

3、 算法介绍

1） DDA 直线算法

DDA 算法的主要核心思想是在直线段的扫描转换算法的基础上，去掉效率比较低下的乘法操作。因此需要引入相关的增量思想。



接下来利用斜截式方程进行一个增量的推倒:

yi=kxi+byi=kxi+b (0<k<1)

yi+1=kxi+1=b=k(xi+1)+b=kxi+k+b=yi+kyi+1=kxi+1=b=k(xi+1)+b=kxi+k+b=yi+k

所以

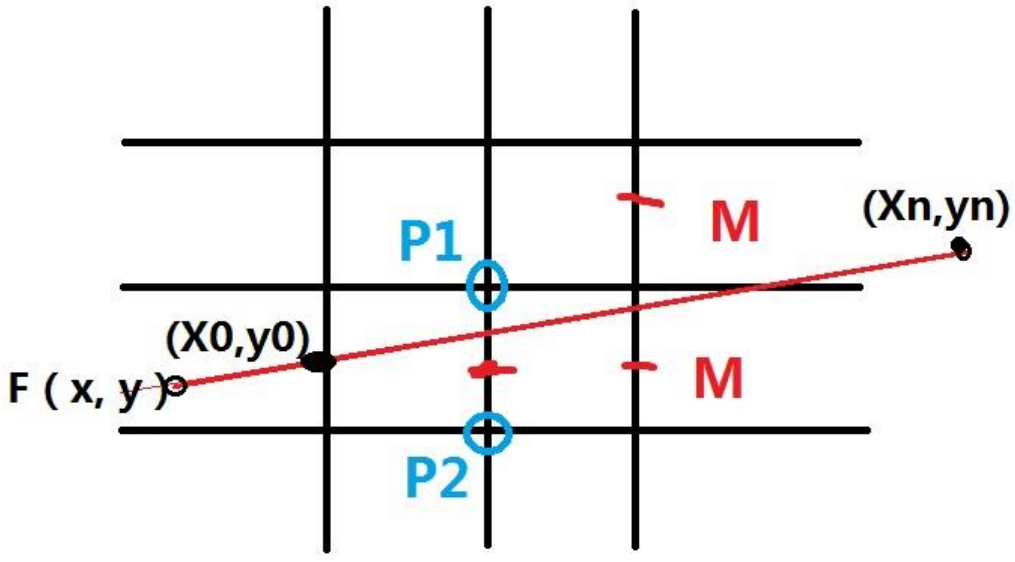
yi+1=yi+kyi+1=yi+k

这个式子的含义是：当前步的 y 值等于前一步的 y 值加上斜率 k，这样就把乘法变成了一个加法，提高了在计算机中的运行效率。

而对于斜率的其他情况，可以变换到斜率为 1 的情况进行计算，绘制的时候再变换回去即可

2） 中点直线算法

对于中点直线算法，也才用类似的思想，将乘法转化为加法计算



对于直线方程 F（x, y） = ax + by + c

直线经过两点 startPoint（x0,y0）， endPoint（xn,yn）

其中:

a = y0 -y1;

b = x1 - x0;

c = x0y1 - x1y0;

且对于直线上的点：F（x, y） == 0

取中点 M 带入方程 d = F(x+1,y + 0.5)判定 d 的符号：

如果 > 0,则 M 在直线上方，应该取正右边的点

如果 < 0，则 M 在直线下方，应盖取右上方的点，

采用增量计算后，可以提高运行效率。如果取正右方的 P1 点，则

d1 = F(xp+2, yp + 0.5) = d + a

增量为 deltD1 = a;

如果取右上方的 P2 点，则 d2 = F(xp+2, yp + 1.5) = d + a + b

为增量：deltD2 = a + b;

另外还需要计算 d 的初值，

d0 = F(x0+1, y0+0.5)= F(X0,Y0) + a + 0.5b ,且由于(x0, y0)在直线上，所以

f (x0,y0)= 0

所以 d0 = a + 0.5b。

到此为止大部分工作已经完成，但是仍有优化空间：由于我们每次只是考虑 d 的符号，所以为了减少浮点运算，同时将上述相关变量扩大 2 倍，优化的方案是 ：a + a 取代直接的 a*2，这样，同样以步进增量地推判定后续位置，便于硬件提高计算效率，继而提高算法实现效率。

deltD1 = a + a;

deltD2 = a + a + b + b;

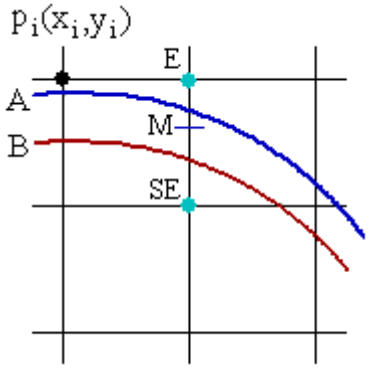
d0 = a + a + b;

或者：di+1 = di + deltD2 (di < 0，右上方点)

或者：di+1 = di + deltD1(di > 0，正右方点);

3） Bresenham 圆绘制算法

对于 Bresenham 画圆算法，采用八分法画圆，取Π/4 到Π/2 之间一段圆弧，判断可选点的中点与圆的关系，如果在圆内，则选择上面的待选点，否则选择下面的待选点，之后按照这八段圆弧的关系，将另外八段圆弧上与该点对应的点也绘制出来。



对于上图中的可选点 E 和 SE，判断他们的中点 M 是在圆内还是在圆外，如果 M 在圆内，则 E 离圆的边更近，选择 E，否则选择 SE

4） Bizier 曲线

设 P_0、P_0^2、P_2是一条抛物线上顺序三个不同的点。过 P_0和 P_2点的两切线交于 P_1点，在 P_0^2点的切线交 P_0P_1和 P_2P_1于 P_0^1和 P_1^1，则如下比例成立：

$$(P_0\ P_0^1)/(P_0^1P_1)=(P_1P_1^1)/(P_1^1P_2)=(P_0^1P_0^2)/(P_0^2P_1^1)$$

这是所谓抛物线的三切线定理。当 P0，P2 固定，引入参数 t，令上述比值为 t:(1-t)，即有：

$$P_0^1=(1-t)P_0+tP_1$$

$P_1^1=(1-t)P_1+tP_2$

$P_0^2=(1-t)P_0^1+tP_1^1$

t 从 0 变到 1，第一、二式就分别表示控制二边形的第一、二条边，它们是两条一次 Bezier 曲线。将一、二式代入第三式得:

$P_0^2=(1-t)^2P_0+2t(1-t)P_1+t^2P_2$

当 t 从 0 变到 1 时，它表示了由三顶点 P0、P1、P2 三点定义的一条二次 Bezier 曲线。并且表明这二次 Bezier 曲线 P02 可以定义为分别由前两个顶点(P0,P1)和后两个顶点(P1,P2)决定的一次 Bezier 曲线的线性组合。依次类推，由四个控制点定义的三次 Bezier 曲线 P03 可被定义为分别由(P0,P1,P2)和(P1,P2,P3)确定的二条二次 Bezier 曲线的线性组合，由(n+1)个控制点 Pi(i=0,1,...,n)定义的 n 次 Bezier 曲线 P0n 可被定义为分别由前、后 n 个控制点定义的两条(n-1)次 Bezier 曲线 P0n-1 与 P1n-1 的线性组合:

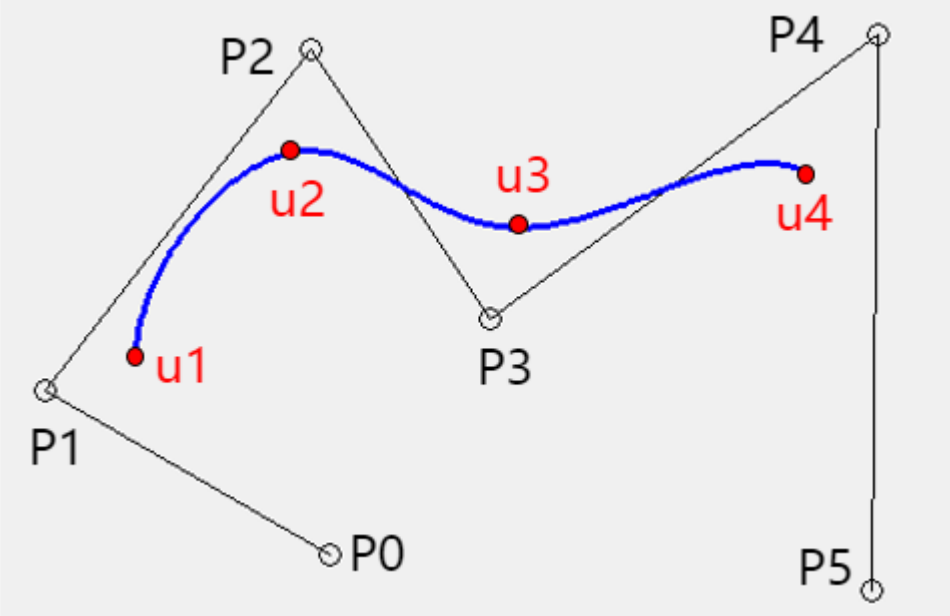
$P_n^0=(1-t)P_0^{n-1}+tP_1^{n-1}$ (0<=t<=1)

由此得到 Bizier 曲线的递推公式

$P_i^k=P_i$(k=0)或 $P_i^k=(1-t)P_i^{k-1}+tP_{i+1}^{k-1}$(1<=k<=n,0<=i<=n-k)

5） B 样条曲线

相比较 Beizer 曲线来说，B 样条有着两个优点：一个是 k 次 B 样条曲线具有良好的局部性，它只与 k+1 个控制点有关；另一个是 B 样条曲线拼接较为简单。



以三次 B 样条曲线为例。由于 k 次 B 样条曲线的控制点有 k+1 个，所以 $P_0P_1P_2P_3$ 控制 u_1u_2 段曲线，$P_1P_2P_3P_4$ 控制 u_2u_3 段曲线，$P_2P_3P_4P_5$ 控制 u_3u_4 段曲线。所以这样就不会像 beizer 曲线那样，改变任一控制点，都会对整个曲线产生影响。

B 样条曲线公式如下

$$S(t)=\sum_{j=i-k}^{i}P_j\cdot N_{j,k}(t)$$

S(t)表示的是 u_au_{a+1} 段曲线，k 表示的 k 次 B 样条曲线，所以 S(t)就是控制点与基函数的乘积之和，其中控制点是从 P_j 点到 P_i 点，一共有 i-j+1 个。图 1 中的 u_1u_2 段曲线，$S_{u1u2}(t) = P_0N_{0,3}(t) + P_1N_{1,3}(t) + P_2N_{2,3}(t) + P_3N_{3,3}(t)$。这个公式看起来非常抽象，可以写成下面的形式

$$P_{j,k}(t)=\sum_{i=0}^{K}P_{i+j}\cdot N_{i,k}(t)$$

$$N_{i,k}(t)=\frac{1}{k!}\cdot\sum_{r=0}^{k-i}(-1)^r C_{k+1}^{r}(t+k-i-r)^k,i=0,1,\ldots,k$$

在上图的公式中，j 表示的是起始的控制点，k 表示的是 k 次 B 样条曲线，i 表

示的是迭代参数，相当于控制点是从 Pj 到 Pj+k。基函数 N(t)中参数 i，k 跟其上面的公式保持同步。在三次 B 样条曲线中，四个基函数为

$$N_{0,3}(t)=\frac{1}{6}(-t^3+3t^2-3t+1)$$

$$N_{1,3}(t)=\frac{1}{6}(3t^3-6t^2+4)$$

$$N_{2,3}(t)=\frac{1}{6}(-3t^3+3t^2+3t+1)$$

$$N_{3,3}(t)=\frac{1}{6}t^3$$

跟据上面的基函数给出 P0,3(t)的公式及相关性质:

$$P_{0,3}(t)=\frac{1}{6}\begin{bmatrix}1 & t & t^2 & t^3\end{bmatrix}\begin{bmatrix}1 & 4 & 1 & 0\\ -3 & 0 & 3 & 0\\ 3 & -6 & 3 & 0\\ -1 & 3 & -3 & 1\end{bmatrix}\begin{bmatrix}P_0\\ P_1\\ P_2\\ P_3\end{bmatrix}$$

这条曲线的端点位置和端点切失如下:

$$P_{0,3}(0)=\frac{1}{6}(P_0+4P_1+P_2)=\frac{1}{3}(\frac{P_0+P_2}{2})+\frac{2}{3}P_1$$

$$P_{0,3}(1)=\frac{1}{6}(P_1+4P_2+P_3)=\frac{1}{3}(\frac{P_1+P_3}{2})+\frac{2}{3}P_2$$

$$P'_{0,3}(0)=\frac{1}{2}(P_2-P_0),P'_{0,3}(1)=\frac{1}{2}(P_3-P_1)$$

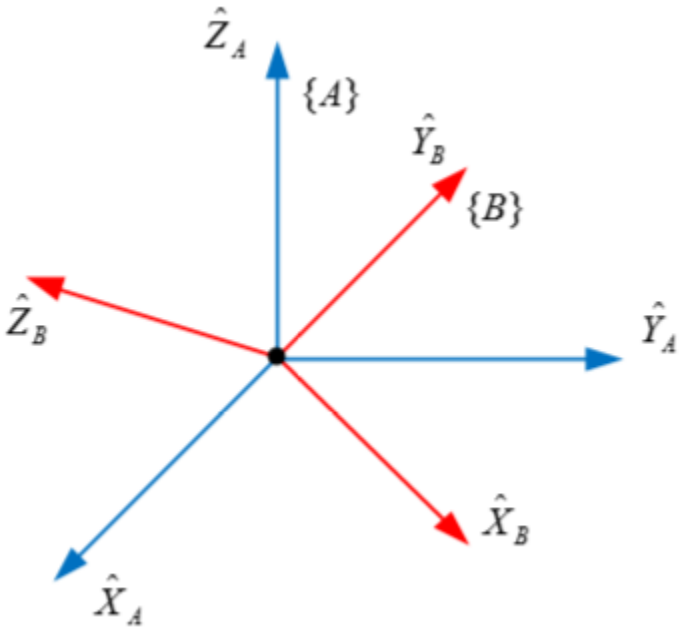
6） 图形平移

图形平移在所有的处理中算是最简单的，如果以能够围住图形的最小矩形左

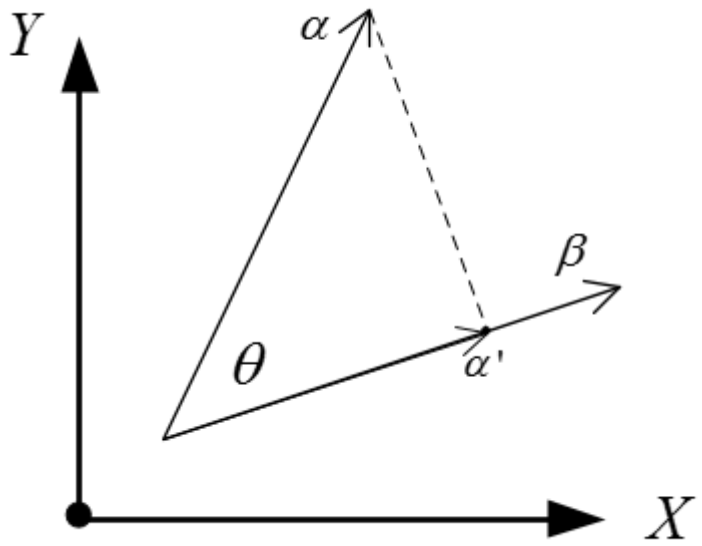
上角坐标作为标准，平移操作就是计算原左上角坐标与新左上角坐标差值（保留符号），将这个差值加到图形的每个像素坐标上即可

7） 图形旋转

图形旋转利用了旋转矩阵。假设最开始空间的坐标系 X_A，Y_A，Z_A 就是笛卡尔坐标系，这样我们得到空间 A 的矩阵 $V_A=\{X_A, Y_A, Z_A\}^T$，其实也可以看做是单位阵 **E**。进过旋转后，空间 A 的三个坐标系变成了图 1 中红色的三个坐标系 X_B, Y_B, Z_B, 得到空间 B 的矩阵 $V_B=\{X_B, Y_B, Z_B\}^T$。将两个空间联系起来可以得到 $V_B=R \cdot V_A$, **R** 就是旋转矩阵。



由于 $X_A=\{1,0,0\}^T$，$Y_A=\{0,1,0\}^T$，$Z_A=\{0,0,1\}^T$，旋转矩阵 **R** 就是由 X_B，Y_B，Z_B 三个向量组成的。旋转矩阵就是正交阵，因为单位向量无论怎么旋转长度肯定不会变而且向量之间的正交性质也不会变。

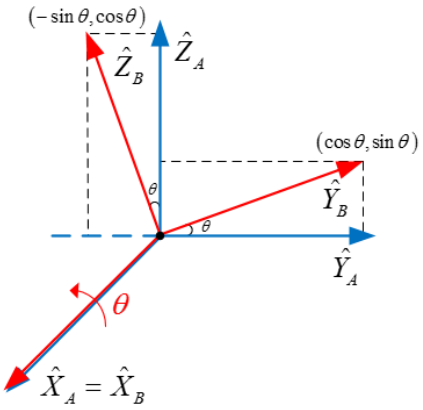


由于图中所有的向量均是单位向量，所以 X_B 与 X_A 点乘的结果可以看成 X_B 在 X_A 上的投影的模，也就是 X_B 在空间 A 中 x 轴的分量。图中中间的位置列出了 X_B 向量中的三个分量分别为 X_B 在 X_A 上的投影的模、X_B 在 Y_A 上的投影的模和 X_B 在 Z_A 上的投影的模。这从几何角度很好理解。以此类推，可以得出的旋转矩阵 R 的表达形式。矩阵 R 的第一行就是 X_A 在 X_B，Y_B，Z_B 上的投影的模，也就是 X_A^T。

$$R=\begin{bmatrix}\hat{X}_B & \hat{Y}_B & \hat{Z}_B\end{bmatrix}\longrightarrow \hat{X}_B=\begin{bmatrix}\hat{X}_B\bullet\hat{X}_A\\ \hat{X}_B\bullet\hat{Y}_A\\ \hat{X}_B\bullet\hat{Z}_A\end{bmatrix}\longrightarrow R=\begin{bmatrix}\hat{X}_B\bullet\hat{X}_A & \hat{Y}_B\bullet\hat{X}_A & \hat{Z}_B\bullet\hat{X}_A\\ \hat{X}_B\bullet\hat{Y}_A & \hat{Y}_B\bullet\hat{Y}_A & \hat{Z}_B\bullet\hat{Y}_A\\ \hat{X}_B\bullet\hat{Z}_A & \hat{Y}_B\bullet\hat{Z}_A & \hat{Z}_B\bullet\hat{Z}_A\end{bmatrix}\hat{X}_A^T$$

根据上面公式可以推出 A 到 B 的旋转矩阵等于 B 到 A 的旋转矩阵的转置。根据前面所说的 A 到 B 的旋转矩阵的逆就是等于 B 到 A 的旋转矩阵，因此很容易推出 R^{-1} 等于 R^T。

现在以三个欧拉角中的 RotX 为例首先计算出 RotX 的旋转矩阵 R_{rotx}。由于 X 轴是垂直于 YoZ 平面的，所以 X_A 和 Y_B，Z_B 的点乘结果为 0，同时 X_B 和 Y_A，Z_A 的点乘结果也为 0。由于 X_A，X_B 都是单位向量，所以 X_A 和 X_B 的点乘结果为 1。由于绕 x 轴旋转，观察 Y_B 和 Z_B 分别在 Y_A 和 Z_A 上的投影情况如下



$$R_{rotx}(\theta)=\begin{bmatrix}1 & 0 & 0\\ 0 & \cos\theta & -\sin\theta\\ 0 & \sin\theta & \cos\theta\end{bmatrix}$$

这样就完成旋转矩阵 **R**rotx

在实际程序实现中，直接利用库函数计算旋转矩阵，之后将旋转矩阵送入 Graphics 类，这样在显示时，原先的图像就被自动旋转了。

8） 图形填充的扫描线算法

对于一个给定的多边形，用一组水平(垂直)的扫描线进行扫描，对每一条扫描线均可求出与多边形边的交点，这些交点将扫描线分割成落在多边形内部的线段和落在多边形外部的线段；并且二者相间排列。于是，将落在多边形内部的线段上的所有象素点赋以给定的色彩值。 算法中不需要检验每一个象素点，而只考虑与多边形边相交的交点分割后的扫描线段。对于每一条扫描线的处理 ：

求交点：首先求出扫描线与多边形各边的交点；

交点排序：将这些交点按 X 坐标递增顺序排序；

交点匹配：即从左到右确定落在多边形内部的那些线段；

区间填充：填充落在多边形内部的线段。

求交点时最简单的办法时将多边形的所有边放在一个表中，在处理每条扫描线时，从表中顺序取出所有的边，分别求这些边与扫描线的交点。 但是这样将做一些无益的求交点动作，因为扫描线并不一定与多边形的边相交，扫描线只与部分甚至较少的边相交；因此，在进行扫描线与多边形边求交点时， 应只

求那些与扫描线相交的边的交点，用边表来确定哪些边是下一条扫描线求交计算时应该加入运算的。

建立边的分类表 ET(Edge Table)，每个结点结构如下：(Ymax ,ΔX ,XYmin，)

Ymax:边的最大 Y 值；

ΔX:从当前扫描线到下一条扫描线之间的 X 增量(dX/dY)；

XYmin:边的下端点的 X 坐标；

Next:指针，指向下一条边。

边的分类表建立时，先按下端点的纵坐标(y 值)对所有边作桶分类，再将同一组中的边按下端点 X 坐标递增的顺序进行排序，X 坐标还相同的按 ΔX 递增的顺序进行排序。

算法中还用到了活化边的概念，把与当前扫描线相交的边称活化边 AEL(Active Edge List) 。组成的表称为活性表 AET，其数据域组成如下：

Ymax :存放边的上端点 Y 坐标；

X ：边与当前扫描线交点的 X 坐标；

ΔX ，next 指针 :同边表。

将一条扫描线与某条边的交点连接起来，就可以直接得到要求的所有交点。在填充过程中，为每一条扫描线建立相应的活化链表，它表示了该扫描线要求交点的那些边，在实用中每一 条边的活化链表的信息与上一条边的活化链表的信息有继承性，再结合 EL 表使得建立十分方便。

算法处理扫描线步骤为：

1、对于扫描线 Y=yc，若对应的 ET 中非空，则将其所有的边从 ET 中取出并且插入到边的活化半表中，并对 AET 中各边按 X 递增排序；

2、若相对于当前扫描线的活化边表 AET 非空，则将 AET 中的边两两依次配对，即第 1，2 边为一对，第 3，4 边为一对，依次类推，每一对边与当前扫描线的交点所构成的区段位于多边形内，依次将这些区段上的点进行着色；

3、将当前的扫描线的纵坐标 Y 累加 1，即 Y=Y+1；

4、将边的活化链表 AET 中满足 Y=ymax 的边删去；

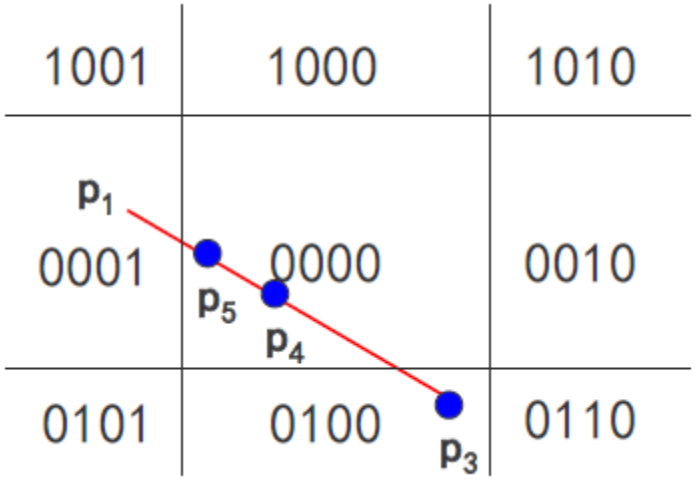
5、将边的活化链表 AET 中剩下的每一条边的 X 域累加 ΔX，即 X=X+ΔX；

6、重复执行 1

重复执行(1)。

8）直线的中点分割算法

线段和窗口的关系分成完全在窗口内、完全在窗口外和与窗口有交点三种情况。直线裁剪的中点分割算法类似二分法，通过二分逼近来确定直线段与窗口（也就是框）的交点。



窗口的四边所在的四条直线可将整个屏幕平面分成 3*3=9 个区域，如果直线段的两个端点都在除了中心区域某个区域内，那么这条直线段与窗口没有相交的部分，无需裁剪；如果直线段的两个端点都在中心区域，那么这条直线段完全落在窗口内，只需将其变换颜色重新绘制一遍即可。

而其余的情况中，如果直线的两个端点落在两侧列的区域（1001，0001，0101 和 1010，0010，0110）或者两侧行的区域（1001，1000，1010 和 0101，0100，0110）那么也是不会被裁剪的，其余情况都要被裁剪。

对于需要裁剪的情况，首先求两端点的中点，判断其是否在裁剪区域内，如果在的话，则以该点将线段分为两部分，对这两部分别求其于窗体框的交点；如果中点不在窗体框内，则比较两端点与窗体框中心的距离，将距离远的那个端点与中点构成的一段丢弃，将距离近的端点与中点之间的一段作为新的直线，重复上述过程，直到中点与窗口边界的坐标值在一定误差范围内相等，就找到了交点。如果找到了交点，那么就将交点与离窗体框中心近的一段裁剪出来。如果没有找到交点，那么根据以上各种情况的排除，这一段完全在窗体框中。也将其裁剪出来即可。

8）梁友栋算法

一条两端点为 P_1（x_1，y_1）、P_2（x_2，y_2）的线段可以用参数方程形式表示：

$$x= x_1+ u\cdot（x_2-x_1）= x_1+ u\cdot\Delta x\ 0\leqslant u\leqslant 1$$

$$y= y_1+ u\cdot（y_2-y_1）= y_1+ u\cdot\Delta y$$

式中，$\Delta x=x_2-x_1$，$\Delta y=y_2-y_1$，参数 u 在 0～1 之间取值，P（x，y）代表了该线段上的一个点，其值由参数 u 确定，由公式可知，当 u=0 时，该点为 P_1（x_1，y_1），当 u=1 时，该点为 P_2（x_2，y_2）。如果点 P（x，y）位于由坐标（x_{min}，y_{min}）和（x_{max}，y_{max}）所确定的窗口内，那么下式成立：

$$x_{min}\leqslant x_1+ u\cdot\Delta x\leqslant x_{max}$$

$$y_{min}\leqslant y_1+ u\cdot\Delta y\leqslant y_{max}$$

这四个不等式可以表示为：

$$u\cdot p_k\leqslant q_k\ ,\quad k=1，2，3，4$$

其中，p、q 定义为：

$$p_1=-\Delta x,\quad q_1=x_1-x_{min}\quad p_2=\Delta x,\quad q_2=x_{max}-x_1\quad p_3=-\Delta y,\quad q_3=y_1-y_{min}\quad p_4=\Delta y,$$

$$q_4=y_{max}-y_1$$

从上式得知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）。如果还满足 $q_k<0$，则线段完全在边界外，应舍弃该线段。如果 $p_k=0$ 并且 $q_k\geqslant 0$，则线段平行于窗口某边界并在窗口内，可以得出下面结论：

1、当 $p_k<0$ 时，线段从裁剪边界延长线的外部延伸到内部；

2、当 $p_k>0$ 时，线段从裁剪边界延长线的内部延伸到外部；

当 $\Delta x\geqslant 0$ 时，对于左边界 $p_1<0$（$p_1=-\Delta x$），线段从左边界的外部到内部；

对于右边界 $p_2>0$（$p_2=\Delta x$），线段从右边界的内部到外部。

当 $\Delta y<0$ 时，对于下边界 $p_3>0$（$p_3=-\Delta y$），线段从下边界的内部到外部；

对于上边界 $p_4<0$（$p_4=\Delta y$），线段从上边界的外部到内部。

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，即：

$$u = q_k/p_k$$

对于每条直线，可以计算出参数 u_1 和 u_2，该值定义了位于窗口内的线段部分：

1、u_1 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 $r_k=q_k/p_k$，u_1 取 0 和各个 r 值之中的最大值。

2、u_2 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 $r_k=q_k/p_k$，u_2 取 0 和各个 r 值之中的最小值。

3、如果 $u_1>u_2$，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 u_1 和 u_2 计算出来。

9）多边形裁剪

多边形裁剪主要介绍一下思路，将多边形超出裁剪区域的部分，按照裁剪矩形的四条边收缩到裁剪矩形的边上，运行的我的 QT 程序即可看到收缩到矩形的每条边都有不同的颜色进行显示

10）3D 图像显示（Qt 程序）

实验要求解析 OFF 文件，OFF 文件格式如下：

列表。详见下面的例子。

OFF

顶点数 面片数 边数

x y z

x y z

…

顶点个数 N v1 v2 v3 … vn

顶点个数 M v1 v2 v3 … vm

…

由上可以看出，OFF 文件从第二行开始才是有用信息，从第二行中读出顶点数、面数和边数，然后根据顶点数，将所有顶点坐标读出，之后再按照面数，每行首先读出该面的点数，然后将所有点读进来。

在读完 OFF 文件后，openGL 开源程序提供了相应的接口，只需将这些信息输入即可进行显示。

4、 总结

这三个月主要用在 C#GUI 框架、QtGUI 框架的学习和 C#语言的学习上，然后对几个基本算法进行了初步而又简单的实现，逐渐掌握了 C#GUI 编程方法。在算法的实现过程中，我发现原本有些不太容易理解的地方，通过自己的动手实践，变得非常容易理解，因为这些算法都是面向实现的，为了加快运行速度和减少计算量，算法的发明者们都进行了大量优化，只有对其进行实现，才能完全理解和掌握它。

5、 参考文献

《计算机图形学教程》 机械工业出版社 孙正兴 主编 周良 郑洪源 谢强 编著
《计算机图形学》课堂 PPT