《计算机图形学》系统技术报告

张书瀚*

(南京大学 计算机科学与技术系, 南京 210093)

摘 要: 计算机图形学课程要求实现一个图形绘制系统，满足包括直线、曲线、圆、多边形等多种图形的输入、编辑、裁剪、变换、显示和储存功能。该系统基于 C++与 OpenGL 实现，架构清晰，由良好的可扩展性。用户界面操作简单，易交互。

1 算法与原理

1.1 图形的生成算法

1.1.1 直线的 *Bresenham* 生成算法

Bresenham 算法是基于光栅扫描原理的直线生成算法。从直线的一端开始，通过逐步计算直线上的点与临近像素点的距离，选择最近的像素点作为直线的下一个点来生成直线。相较于 *DDA* 算法，*Bresenham* 算法的计算中不涉及到除法与浮点数，使得运算的速度大大增加，即算法的效率更高。

设直线的两端点为(x_{start}, y_{start})与(x_{end}, y_{end})。以$x_{start} < x_{end}$，$y < y_{end}$且直线斜率$m < 1$为例。

由于$m < 1$，所以以 x 轴为基准，当$x = x_{k+1}$ 时，我们需要确定 y 的位置。由于直线是上升的，所以只有$y = y_k$或者 $y = y_{k+1}$两种可能，问题就在于区分这两种可能的条件。

令$d_x = x_{end} - x_{start}$，$d_y = y_{end} - y_{start}$，$d_1$，$d_2$为两个候选像素与线段数学路径点的垂直偏移。则：

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

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

且$y_{k+1} = y_k + 1$

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

又$m = \frac{dy}{dx}$

$$\therefore p_k = d_x(d_1 - d_2) = 2d_x x_k - 2d_y y_k + c$$

其中 c 为常量

此时p_k即为 y 的位置的决策参数。当$p_k > 0$时，y_{k+1}更接近线段；当$p_k < 0$时，y_k更接近线段。为方便程序计算，可将p_k写成递归函数的形式：

$$p_k = \begin{cases} 2d_y - d_x, & k = 0 \\ p_{k-1} + 2d_y, & p_{k-1} < 0 \\ p_{k-1} + 2(d_y - d_x), & p_{k-1} \geq 0 \end{cases}$$

至此，我们便可以根据直线的两端点位置逐步确定直线上每个点所在的像素位置，从而画出直线。

上述的例子只是因起始点与结束点位置不同而产生的 8 种情况中的一种，其余的情况可以通过坐标变换以及 x, y 之间的坐标交换从而变成上述情况，其解决的原理相同。

* 作者简介:学号：151220161。联系方式：151220161@smail.nju.edu.cn

1.1.2 圆与椭圆的中点生成算法

（1） 圆的中点生成算法

圆的中点生成算法根据圆的半径计算，从某一个像素点开始，逐步计算下一个像素点的位置。其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周边界的相对位置关系(圆周边界的内或外)来选择像素。

定义一个半径为 r，圆心在(x_c, y_c)的圆，其圆方程为$f_{circle}(x,y) = (x-x_c)^2 + (y-y_c)^2 - r^2$。为了减少计算量，我们可以假设圆心在$(0,0)$，计算八分之一个圆，然后利用对称性通过坐标变换得到其余的像素点位置，最后将圆平移到(x_c, y_c)处。

从 y 轴上的圆周点$(0,r)$顺时针方向生成第一象限内八分之一圆周，由于切线斜率的绝对值小于 1，所以在 x 轴方向取样，x 的值递增时，y 的值递减。设第 k 步选取的像素为(x_k, y_k)，我们需要确定下一个像素点是(x_{k+1}, y_k)还是(x_k, y_{k-1})。将圆函数作为决策参数，第 k 个决策参数是圆函数在两候选像素中点处求值。即：

$$p_k = f_{circle}(x_{k+1}, y_k - \frac{1}{2})$$

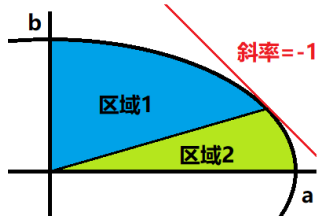
若$p_k < 0$，则候选像素点的中点在圆边界内，所以高像素(x_{k+1}, y_k)更接近圆边界的真实位置。若$p_k > 0$，低像素(x_k, y_{k-1})更接近边界。将p_k写成递归函数：

$$p_k = \begin{cases} \frac{5}{4} - r, & k = 0 \\ p_{k-1} + 2x_k + 1, & p_{k-1} < 0 \\ p_{k-1} + 2(x_k - y_k) + 1, & p_{k-1} \geq 0 \end{cases}$$

（2） 椭圆的中点生成算法

椭圆的中点生成算法与圆的类似，但是由于椭圆的长轴短轴长度不同，所以相较于圆的算法需要做一些修改。

定义一个长轴为 a，短轴为 b，中心在(x_c, y_c)的椭圆，其方程为$f_{ellipse}(x,y) = b^2(x-x_c)^2 + a^2(y-y_c)^2 - a^2b^2$。为了减少计算量，我们可以假设椭圆中心在$(0,0)$，计算第一象限内的椭圆弧，然后利用对称性通过坐标变换得到其余的像素点位置，最后将最后平移到(x_c, y_c)处。



如右图，在第一象限内的椭圆被分为两部分，在斜率绝对值小于 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。区域 1 和区域 2 的交替条件是$bx \geq ay$。

从$(0,b)$开始生成椭圆弧。在区域 1 中，假如第 k 步选择的像素为(x_k, y_k)，那么决定下一个像素点位置的决策参数为：

$$p1_k = f_{ellipse}(x_{k+1}, y_k - \frac{1}{2})$$

若$p1_k < 0$，则中点位于椭圆内部，所以高像素(x_{k+1}, y_k)更接近椭圆边界的真实位置。 否则，低像素(x_k, y_{k-1})更接近边界。将$p1_k$写成递归函数：

$$p1_k=\begin{cases} b^2-a^2y+\dfrac{a^2}{4}, & k=0 \\ p1_{k-1}+2b^2x_k+b^2, & p1_{k-1}<0 \\ p1_{k-1}+2(b^2x_k-a^2y_k)+b^2, & p1_{k-1}\geq 0 \end{cases}$$

在区域 2 中，其初始点应当取区域 1 中选择的最后位置(x_0,y_0)。若选择的某像素位置为(x_k,y_k)，那么决定下一个像素点位置的决策参数为：

$$p2_k=f_{ellipse}(x_k+\frac{1}{2},y_k-1)$$

若$p2_k>0$，则中点位于椭圆边界之外，所以选择像素(x_k,y_{k-1})。否则，像素(x_{k+1},y_{k-1})更接近椭圆边界的真实位置。将$p2_k$写成递归函数：

$$p2_k=\begin{cases} b^2(x_0+\dfrac{1}{2})^2+a^2(y_0-1)^2-a^2b^2, & k=0 \\ p2_{k-1}+2(b^2x_k-a^2y_k)+a^2, & p2_{k-1}<0 \\ p2_{k-1}-2a^2y_k+a^2, & p2_{k-1}\geq 0 \end{cases}$$

1.1.3 曲线生成算法

（1） *Bezier* 曲线算法

我们只考虑绘制三阶贝塞尔曲线，所以需要取四个点：起点，终点以及两个控制点。首先观察三阶贝塞尔曲线的方程：

$$B(t)=P_0(1-t)^3+3P_1t(1-t)^2+3P_2t^2(1-t)+P_3t^3, \quad t\in[0,1]$$

其中P_0为起点，P_3为终点，P_1,P_2为控制点。*t* 为曲线的追踪参数，绘制曲线的过程就是不断增加 *t* 并对函数求值的过程。理论上来说，*t* 的步进越小，绘制出的曲线就越精准，但实际操作中要合理选取Δt的值。

令$d_x=|P_0(x)-P_3(x)|$, $d_y=|P_0(y)-P_3(y)|$。经测试发现，当曲线上像素总个数$n=2\times max\{d_x,d_y\}$时，像素点的分布既不会太稀疏导致曲线出现明显的中断，也不会太密集使得曲线宽度增加，此时$\Delta t=\frac{1}{n-1}$。

根据上述过程，将贝塞尔曲线方程写成如下形式：

$$\begin{cases} x(k)=P_0(x)(1-\dfrac{k}{n-1})^3+3P_1(x)(\dfrac{k}{n-1})(1-\dfrac{k}{n-1})^2+3P_2(x)(\dfrac{k}{n-1})^2\left(1-\dfrac{k}{n-1}\right)+P_3(x)(\dfrac{k}{n-1})^3 \\ y(k)=P_0(y)(1-\dfrac{k}{n-1})^3+3P_1(y)(\dfrac{k}{n-1})(1-\dfrac{k}{n-1})^2+3P_2(y)(\dfrac{k}{n-1})^2\left(1-\dfrac{k}{n-1}\right)+P_3(y)(\dfrac{k}{n-1})^3 \end{cases}, \quad k\in N,k<n$$

即可计算出曲线的 *n* 个取样点，作出一条贝塞尔曲线。

1.1.4 多边形的生成算法

对于多边形的生成，采用依次取点，逐次连接的方法。

首先选取$n+1$个点，其中第 0 个与第 *n* 个点位置相同。当选取的某点P_n与P_0距离小于 10 时，认为多边形的顶点选择结束，将P_0的坐标赋给P_n，使多边形封闭。

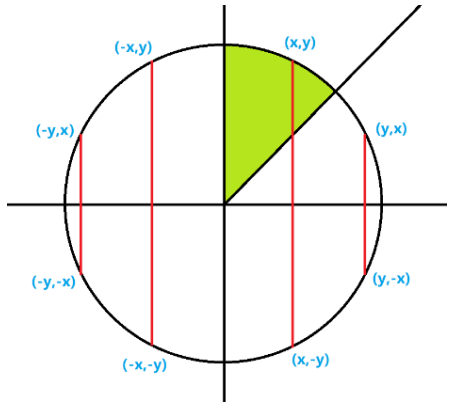
之后采用 *Bresenham* 算法构造直线P_kP_{k+1}，$k=0,1,2\cdots n-1$。所有的直线构造完成后，即生成了一个封闭的 *n* 边形。

1.2 图形的变换算法

1.2.1 填充算法与原理

（1） 圆与椭圆的填充

由于圆与椭圆有固定的公式，所以填充可以利用公式判断某点是否在图形内，若在则将此点画出。但是这种方法需要进行大量的乘法运算与开平方运算，效率十分低，所以不采用。我们将之前的圆的中点生成算法加以改进，就可以对圆进行填充。



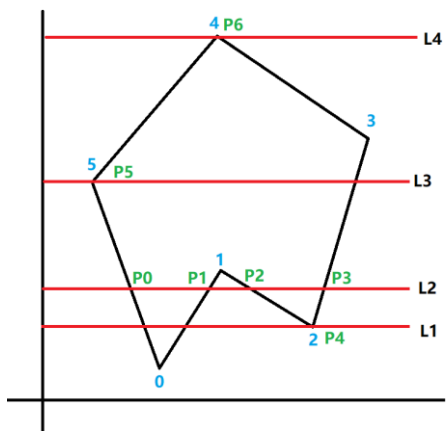
以原点在$(0,0)$的圆为例。如右图所示在中点生成算法中，我们只生成了绿色区域的八分之一圆弧。假设某时刻计算得到的点为(x,y)，那么根据对称性，可以得到其余七个坐标。将这些坐标按右图红线的方式连接即可绘制出四条线。在计算圆弧上坐标的同时，绘制这些线段，那么当八分之一圆弧生成完毕时，整个圆也就被填充完毕了。

对于椭圆，与圆的方式类似，在生成第一象限上的点的同时不断的绘制直线来填充椭圆。

（2） 多边形的填充——扫描线算法

由于多边形没有确定的函数，但每条边的方程时可以得到的，所以采用扫描线算法，逐行计算每条扫描线上应当被填充的点，以此完成整个多边形的填充。

扫描线算法的核心思想在于计算每条扫描线与多边形的交点，然后将交点按横坐标大小排序，两两分组，并填充每组坐标之间的点。



以右图为例。扫描线 *L2* 与多边形产生了 4 个交点，通过直线方程可以计算出P_0,P_1,P_2,P_3的坐标，然后绘制线段P_0P_1与P_2P_3，即完成了对多边形在扫描线 *L2* 上的填充。但仍有两个问题需要解决。

在计算扫描线与多边形交点时产生的计算量很大。对此，利用直线的连贯性，从多边形某边的下端点开始，在扫描线增加时采用增量的方式计算交点 *x* 坐标。相邻两扫线之间 *y* 坐标变化为：

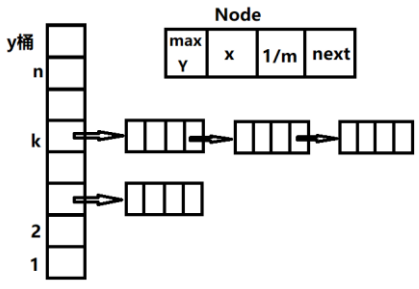
$$y_{k+1}=y_k+1$$

故当前扫描线交点 *x* 值x_{k+1}可从前一条扫描线上 *x* 交点值x_k来决定：

$$x_{k+1}=x_k+\frac{1}{m}$$

其中 m 为当前边的斜率。

在计算交点时，会遇到多边形的顶点在扫描线上的情况，如图中的P_4,P_5,P_6。对于这些情况，需要分类仔细处理，在之后会详细说明。



为了能很好的应用增量计算与处理定点问题，引入有序边表与活化边表。

有序边表是按边下端点 *y* 坐标对非水平边进行分类的指针数组。如图，为每条扫描线建立一个“桶”，对多边形所有边按下端点 *y* 坐标值进行 *y* 桶分类，下端点的 *y* 坐标值为 *i*

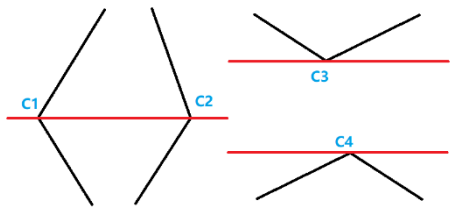
的边归入第 i 类。同一类中的边按交点 x 值递增的顺序排列成链表。链表中的每个节点包含该边上端点 y 坐标值、边下端点 x 坐标值和边斜率倒数$\frac{1}{m}$。

活化边表用来记录多边形边沿扫描线的交点序列，对第 k 条扫描线，其活化边表可从有序边表按如下方式生成：

y 桶由 1 到 k 的所有扫描线的结点加入 k 的活化边表，然后删除满足$y_{max} \le k$的边，剩下的每个节点 x 域都加上自己的$\frac{1}{m}$值来得到交点，最后按节点的 x 域排序。

当扫描线过多边形顶点时：

对于边在扫描线两侧的情形，加入活化边表后 *C*1 出现了两次，而在删除操作时会删除作为下方线段上端点的 *C1*，而保留作为上方线段下端点的 *C1*，所以恰好保留了一个，使得配对能够完成。



对于边在扫描线上侧的情形，*C3* 出现两次但都不会被删除，所以两个排序后两个 *C3* 能够直接配对，不会对其他点造成影响。

对于边在扫描线下侧的情形， *C4* 出现两次但都被删除，所以 *C4* 不会被输出。由于是填充算法，所以顶点不被输出也无妨，不会产生影响。

1.2.2 平移算法

对图形所有的控制点(x_k, y_k)执行

$$\begin{cases} x_k' = x_k + t_x \\ y_k' = y_k + t_y \end{cases}$$

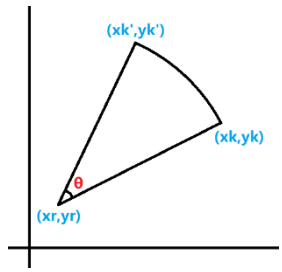
其中(t_x, t_y)为平移向量。

1.2.3 旋转算法

对图形所有的控制点(x_k, y_k)执行

$$\begin{cases} x_k' = x_r + (x_k - x_r)\cos\theta - (y_k - y_r)\sin\theta \\ y_k' = y_r + (x_k - x_r)\sin\theta + (y_k - y_r)\cos\theta \end{cases}$$

其中(x_r, y_r)为旋转中心，θ为旋转角。



为了能够实时显示图形旋转的结果，我们需要在每一次 displayFunc（显示屏幕）的循环中都进行旋转的计算，即根据当前图形位置与当前鼠标位置计算旋转角。但由于 displayFunc 的循环速度非常快，所以如果用上述方法实时计算，那么每一次的旋转角都非常小，所以运算产生的误差很大。因此添加了方法 setRotate，在旋转操作开始时（即鼠标点下时）记录当前图形位置，之后的计算都使用记录值来计算，可以有效减小误差。

1.2.4 缩放算法

对图形所有的控制点(x_k, y_k)执行

$$\begin{cases} x_k' = x_k \cdot s_x + X_0(1 - s_x) \\ y_k' = y_k \cdot s_y + Y_0(1 - s_y) \end{cases}$$

其中(s_x, s_y)为在 x,y 方向上的缩放比例，即缩放向量。(X_0, Y_0)为缩放中的不动点（基准点）。

与旋转操作相同，缩放时实时计算导致缩放参数十分小。为了减小误差，在操作开始时记录图形位置，在之后的缩放中都使用记录位置而不是实时位置计算。

1.2.5 裁剪算法与原理

（1） 直线裁剪——梁友栋-Barsky 裁剪算法

对于一条端点在$P_1(x_1,y_1)$，$P_2(x_2,y_2)$的线段可以用参数方程表示：

$$\begin{cases} x = x_1 + u(x_2 - x_1) = x_1 + u \times \Delta x \\ y = y_1 + u(y_2 - y_1) = y_1 + u \times \Delta y \end{cases}, \qquad 0 \le u \le 1$$

所以如果点$P(x,y)$位于由坐标(x_{min},y_{min})和(x_{max},y_{max})所确定的窗口内，那么就有：

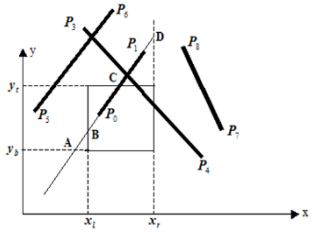
$$u \times p_k \le q_k, \qquad k = 1,2,3,4$$

其中：

$$\begin{cases} p_1 = -\Delta x, & q_1 = x_1 - x_{min} \\ p_2 = \Delta x, & q_2 = x_{max} - x_1 \\ p_3 = -\Delta y, & q_3 = y_1 - y_{min} \\ p_4 = \Delta y, & q_4 = y_{max} - y_1 \end{cases}$$

从上式可以知道：

- 当$p_k = 0$时，线段平行于窗口某边界的直线；
- 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$q_k < 0$时，则线段完全在边界外，应舍弃该线段。
- 当$p_k = 0$且$q_k \ge 0$时，则线段平行于窗口某边界并在窗口内。



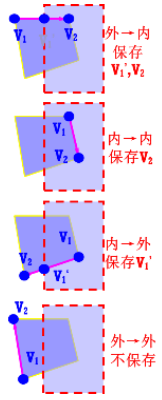
当$p_k \ne 0$时，对于每条直线，可以计算出参数 u1 和 u2，该值定义了位于窗口内的线段部分：

1. u1 的值由线段从外到内遇到的矩形边界所决定（pk<0），对这些边界计算 rk=qk/pk，u1 取 0 和各个 r 值之中的最大值。
2. u2 的值由线段从内到外遇到的矩形边界所决定（pk>0），对这些边界计算 rk=qk/pk，u2 取 1 和各个 r 值之中的最小值。
3. 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。

（2） 多边形裁剪——Sutherland-Hodgman 算法

Sutherland-Hodgman 算法的思想很简单，即遍历多边形的所有边与裁剪窗口的所有边，根据构成多边形边的每对顶点在裁剪窗口半空间中的位置关系确定多边形边界与裁剪窗口的相交方式，以此生成新的多边形顶点。

建立新的顶点表，用以下判断方法想表中添加顶点：



•起始点在窗口边界外侧空间，终止点在内侧空间，则该边与窗口边界的交点和终止点都输出。

• 起始和终止点都在窗口边界内侧空间，则只有终止点输出。

•起始点在窗口边界内侧空间，终止点在外侧空间，则只有该边界与窗口边界的交点输出。

• 起始和终止点都在窗口外侧空间，不增加任何点。

当遍历结束后，我们可以认为多边形已经被裁剪完毕，新生成的顶点表即为多边形裁剪结果。

（3） 其他裁剪

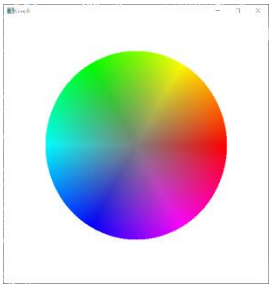
对于曲线，圆与椭圆，没有采用真正意义上的裁减算法，而是通过特殊方式展示裁剪效果。

每一次在屏幕上画出裁剪框时，都会在所有已绘制图形中记录裁剪框的范围，在裁剪结束后，图形

在裁剪范围之外的点仍会被计算，但不会被显示出来。这样以来，就有了图形被裁剪过的效果，但实际上图形并没有真正实现“裁剪”。

1.3 其他算法

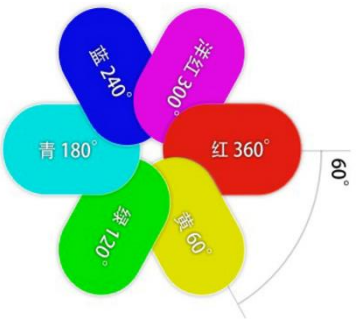
1.3.1 选色盘的实现



为了能够更加方便的选取多种颜色，实现了一个如右上图所示的简单的选色盘，可以选取多种颜色。

选色器是根据 HSL 模型转换为 RGB 模型来实现颜色的选择。固定 HSL 模型中 L 即亮度值为 0.5，此时 H（色彩）与 S（饱和度）的值可以对应到圆上的某点，其中 H 取决于该点与圆心连线与 x 轴的夹角，其对应值如右下图所示；而 S 取决于该点到圆心距离 d。在圆上任意点击一个点，计算出夹角θ与距离 d。记 r 为圆的半径，则有：

$$\begin{cases} h = \dfrac{\theta}{2\pi}, & h \in [0,1] \\ s = \dfrac{d}{r}, & s \in [0,1] \end{cases}$$



以h,s,l为参数计算参数p,q以及t_R,t_G,t_B：

$$q = \begin{cases} l \times (1+s), & l < 0.5 \\ l + s - (l \times s), & l \geq 0.5 \end{cases}$$

$$p = 2 \times l - q$$

$$\begin{cases} t_R = h + \dfrac{1}{3} \\ t_G = h \\ t_B = h - \dfrac{1}{3} \end{cases}$$

$$if\ t_C < 0 \rightarrow t_C = t_C + 1.0, for\ each\ C \in \{R,G,B\}$$

$$if\ t_C > 1 \rightarrow t_C = t_C - 1.0, for\ each\ C \in \{R,G,B\}$$

根据上述参数，，计算R,G,B的值：

$$for\ each\ C \in \{R < G < B\}, \qquad Color_C \begin{cases} p + ((q-p) \times 6 \times t_C, & t_c < \dfrac{1}{6} \\ q, & \dfrac{1}{6} \leq t_C < \dfrac{1}{2} \\ p + \left((q-p) \times 6 \times \left(\dfrac{2}{3} - t_c\right)\right), & \dfrac{1}{2} \leq t_c < \dfrac{2}{3} \\ p, & t_c \geq \dfrac{2}{3} \end{cases}$$

这样一来就可以通过在选色器上点击任意一点来选择颜色了。

1.3.2 图形的储存

图形的储存考虑将图形存储为 24 位 bmp 格式的图片。用 OpenGL 的函数 glReadPixels 可以读出当前所有在窗口内的像素点的信息，所以问题在于如何将信息存为 bmp 格式的图片。

已知 bmp 文件有长为 54 字节的头部，之后为像素信息，所以只需要找到一个已有的 bmp 文件，拷贝其头部，然后修改其头部中有关图像宽高的信息，再将之前读取的像素信息附在头部之后即可。

2 系统框架设计与功能介绍

2.1 系统框架设计

2.1.1 数据结构

（1） 图形类 Shape 及其子类

在画图程序中，所有的图形都有一定的共性，如数据成员“颜色”，“坐标集”等与成员函数“draw”，“fill”等。为了保持数据的一致性，提高程序的可拓展性，以 Shape 为父类构造一系列图形子类。Shape 类包含了所有图形类必需的数据与方法，其中一部分方法以虚函数的方式实现，以方便不同的子类对其进行覆盖重写。

```
class Shape {
protected:
    vector<pixel> points;        //坐标集
    Color lineColor;             //图形线颜色
    Color fillColor;             //图形填充颜色
    bool isCutted = false;       //是否被裁剪
    pixel cut1 = { 0, 0 }, cut2 = { WINX,WINY };    //裁剪窗口
    bool isFill = false;         //是否被填充
    vector<pixel> scalelist;     //缩放记录
    vector<pixel> rotatelist;    //旋转记录
    vector<pixel> editlist;      //编辑记录

public:
    virtual void draw(){}        //绘图
    virtual void fill(){}        //填充
    virtual void rotate(){}      //旋转
    virtual bool isSelect(pixel p){ return false; } //选中
    virtual void showEdit(){}    //编辑框
    virtual bool isEdit(pixel p) { return false; }  //编辑
    virtual void setEdit() {}        //设置编辑记录
    //裁剪
    virtual bool cut(pixel c1, pixel c2){ ... }

    void addPixel(pixel p);      //向坐标集中添加新坐标
    void setColor(Color color); //设定颜色
    void scale();            //缩放
    void translate();        //平移
    void setScale();         //设置缩放记录
    void setRotate();        //设置旋转记录
    void setFill();          //设置图形是否被填充过
    void myglVertex2i(int x, int y);    //显示像素点
};
```

（2） 存储类 Graph

为了储存之前画过的图像并在屏幕上重写，构造了 Graph 类。该类的作用在于储存所有以画过的图像，并提供了一系列方法来选择某个图像，并调用该图像的填充，编辑，变换等方法来对图像进行修改。

```
class Graph {
private:
    //容器
    vector<Shape*> container;
public:
    //向容器中添加
    void addShape(Shape* shape);
    //画出容器中所有的图形
    void draw();
    //设定颜色
    void setColor(Color color);
    //选择图像并根据当前系统状态来对被选中的图像进行修改
    void graphSelect(pixel p);
    //裁剪
    void graphCut(pixel c1, pixel c2);
    //清空容器
    void clear();
};
```

（3） 枚举类以及全局变量

为了能够方便的控制程序的进程与状态，设置了三个枚举类与一些全局变量。

COLOR 类用来表示颜色，TYPE 类用来表示图形，STATE 类用来表示程序状态。

COLOR CurColor 用来记录当前选择的颜色，方便在画图时进行赋值。

TYPE CurType 用来记录当前选择要画的图形，方便进一步调用不同的对象。

STATE CurState 用来记录当前程序的状态，比如处于画图状态，平移图像状态，旋转图像状态等。这样才能根据状态的不同调用对象中不同的成员函数。

COLORSTATE ColorState 用来记录当前的颜色选取状态，即是否正在选色器中选取颜色。与 CurState 记录的状态分开，这样就可以在不用重新选择操作的情况下改变颜色。

Shape CurShape 用来记录当前被选中的图形，方便对其进行操作。

pixel Begin, Current 负责记录鼠标的起始点与终止点，方便将点的坐标赋值给对象的数据成员。

2.1.2 框架设计与运行流程

程序主要是由鼠标状态函数 mouseFunc，显示函数 displayFunc 与 3.1.1（3）中所述的全局变量搭配来控制的。

程序运行的流程如下：

在 main 函数中进行一系列的初始化后，程序通过 glutDisplayFunc(displayFunc)显示界面，同时 displayFunc 函数进入循环工作，以完成对屏幕的重写。鼠标右键进入菜单并选择后，setValue 函数会给 CurColor，CurType 与 CurState 赋值，之后进入不同的系统状态。

在不同的状态下，mouseFunc 函数接受鼠标点击事件，然后根据当前的状态调用不同的函数来完成操作。

画图状态下，鼠标左键点击与鼠标移动配合来确定图形的控制点，然后根据当前图形种类构造新的图形对象，赋值，并存入存储器中。在此期间，display 函数会保持下面的循环：通过调用 Graph.draw 地画出存储

器中存储的图形，根据全局变量构造新图形 Shape，调用 Shape.draw 画出新图形。这样的循环使得图形能够不断在屏幕上重写而不消失

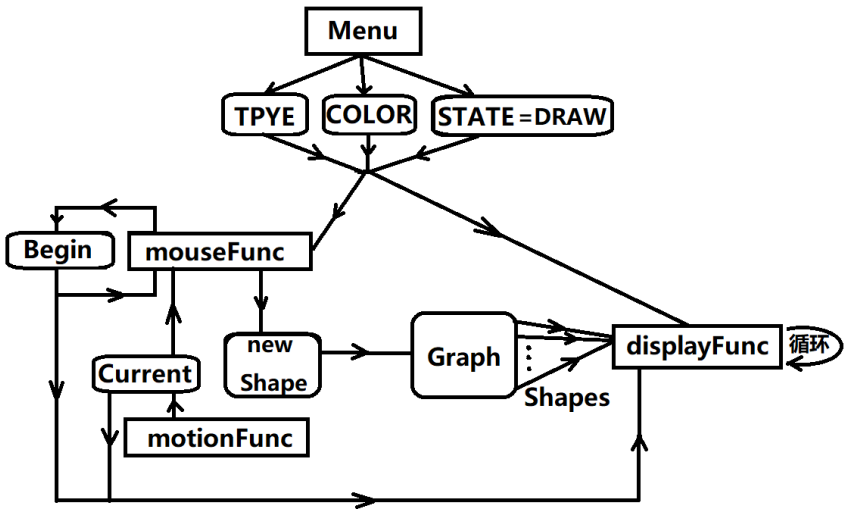
图形编辑状态下，选择图形会出现图形的编辑框与编辑点，点击拖动编辑点可以对图形的控制点进行修改，从而完成对图形的编辑。

平移状态下，鼠标左键按下选择图形，拖动时调用图形对象的 translate 函数平移图形，鼠标松开释放图形。

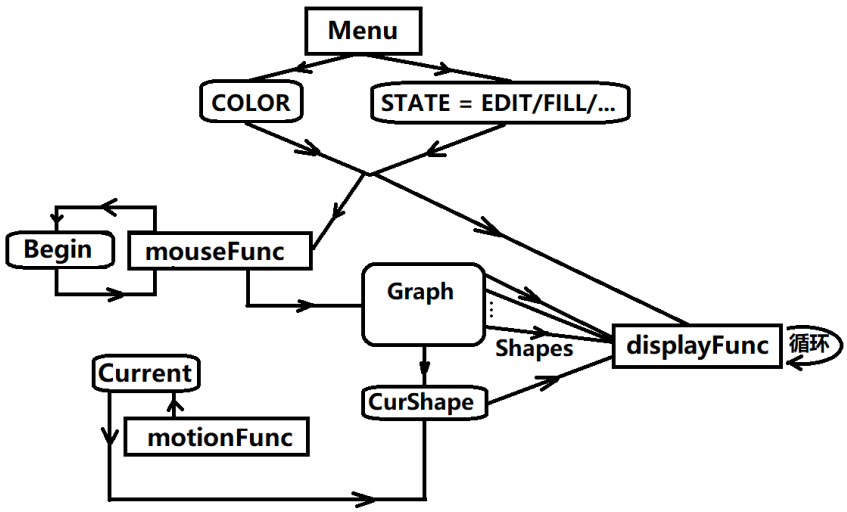
旋转、填充、缩放等类似。

2.1.3 流程图

在画图状态下，程序流程图如下



在编辑和变换状态下，流程图有一些变化



2.2 系统功能

说明： √表示该图形的对应功能已完全实现

○表示该图形对应功能实现不完全，具体说明见 2.2.5（3）其他裁剪

空表示该图形对应功能未实现或不需要实现

		图形					
		直线	曲线	圆	椭圆	多边形	立方体
操作	绘制	√	√	√	√	√	√
	编辑	√	√	√	√	√	
	填充			√	√	√	
	裁剪	√	○	○	○	√	
	平移	√	√	√	√	√	√
	旋转	√	√	√		√	
	缩放	√	√	√	√	√	√
	选择颜色	√	√	√	√	√	√
	保存	√	√	√	√	√	√

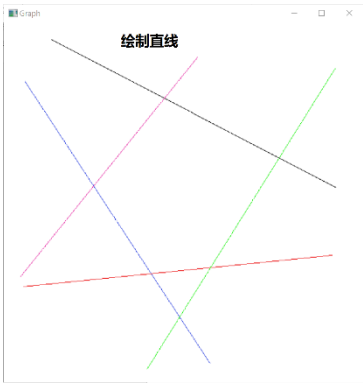
3 系统测试

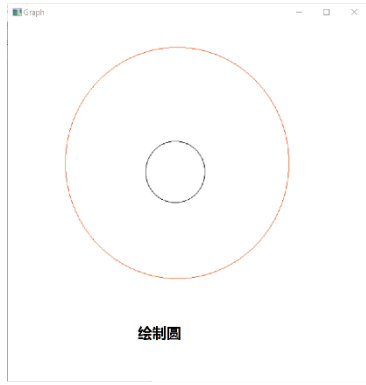
3.1 画图测试

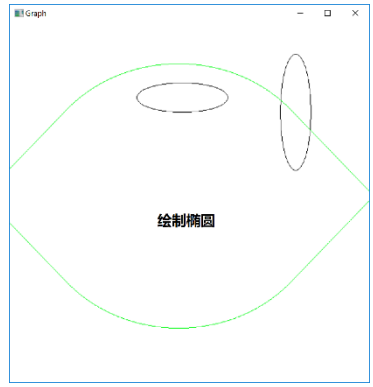
绘制直线：斜率为正，绝对值大于 1；斜率为正，绝对值小于 1；斜率为负，绝对值大于 1；斜率为负，绝对值小于 1 共四条.测试结果正常。

绘制圆：半径较大一个；较小一个。测试结果正常。

! 绘制椭圆：x 轴为长轴，较小一个；y 轴为长轴，较小一个；x 轴为长轴，较大一个。经测试发现，当 x 轴长略大于 y 轴长且都较大时，椭圆会出现变形。经过分析，确定时椭圆的中点生成算法在这种情况下会出现问题，属于算法层面问题，暂无较好解决方法。



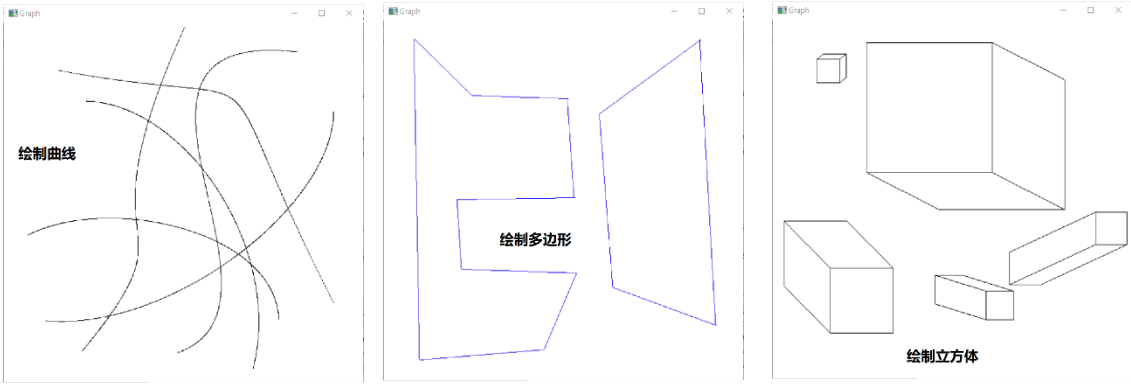




绘制曲线：尝试绘制多样曲线。测试结果正常。

绘制多边形：凹多边形一个，凸多边形一个。测试结果正常。

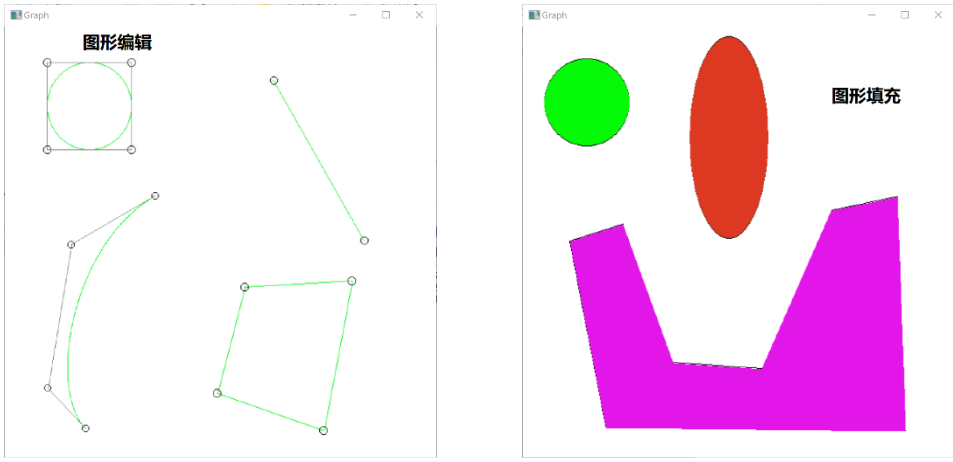
绘制 3D 图形：尝试多种立方体。测试结果正常。



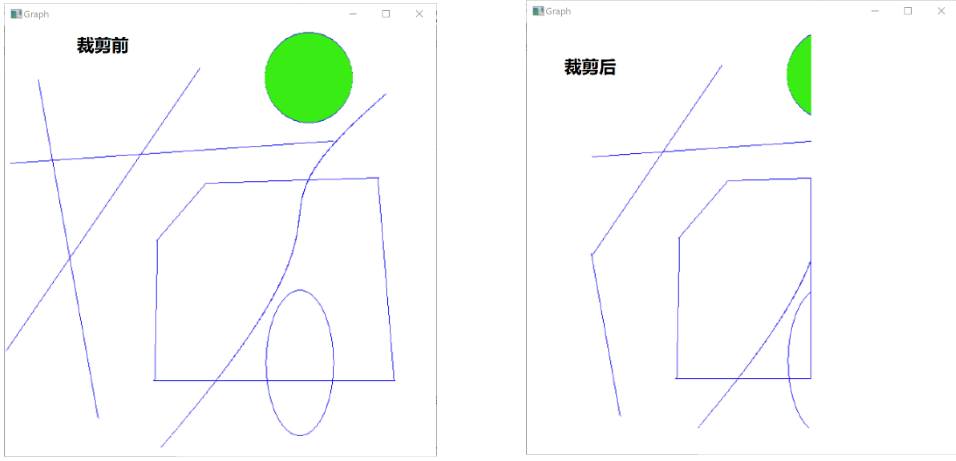
3.2 编辑与变换测试

编辑图形：由于图形的编辑无法用图片展示过程，所以图片主要展示编辑框。另外由于不能同时编辑多个图形，所以将多个图形 PS 到同一张图上显示。经测试编辑功能正常。

填充图形：对圆形，椭圆形与多边形进行不同颜色的填充。测试结果正常。



裁剪图形：对屏幕上的各种图形进行裁剪。测试结果正常。



平移，旋转与缩放图形：无法用图片展示过程。对多种图形的测试结果正常。

3.3 其他测试

颜色选择测试：在之前的测试中已采用了多种颜色。测试结果正常。

保存测试：画面可以完整保存。测试结果正常。

致谢 在此,对提供算法支持的张岩老师表示感谢.

References:

[1] 《Bresenham 直线算法与画圆算法》 http://www.360doc.com/content/13/1220/09/11400509_338596444.shtml

[2] 《OpenGL 入门学习》 http://www.cppblog.com/doing5552/archive/2009/01/08/71532.html

[3] 《OpenGL 基础：glut 处理鼠标事件(含滚轮输入)》 http://blog.csdn.net/jacky_chenjp/article/details/69396540

[4] 《glutDisplayFunc》 https://baike.baidu.com/item/glutDisplayFunc/3044282?fr=aladdin

[5] 《OPENGL 绘制贝塞尔曲线》 http://blog.csdn.net/qq_28057541/article/details/51305292

[6] 《菜鸟玩 HSL 转 RGB》 http://www.zcool.com.cn/article/ZMTQzMjA0.html