《计算机图形学》系统技术报告

151220013 陈彧

(南京大学 计算机科学与技术系，南京 210023)

1 系统环境

本系统在 vs2013 最终版中进行开发，程序中使用了 GLU 工具包。

2 实验原理

2.1 直线

绘制直线利用的是线段生成的 Bresenham 算法。

2.1.1 线段生成的 Bresenham 算法概述

线段生成的 Bresenham 算法采用整数增量运算，它是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)，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。Bresenham 算法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”，它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。

2.1.2 问题定义

假如在第 k 步确定的像素位置为(x_k, y_k)，第 k＋1 步的问题是确定在取样位置 x_{k+1} 处选择哪个像素？位置(x_{k+1}, y_{k+1})或位置(x_{k+1}, y_k)？

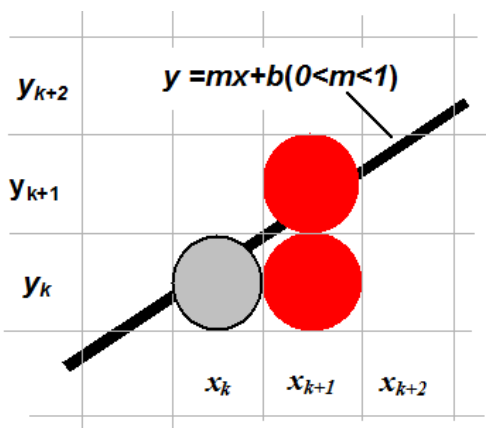


Fig. 1 从(x_k,y_k)像素画直线的屏幕网格

在取样位置 x_{k+1} 分别用 d_1 和 d_2 来标识两个候选像素与线段数学路径点的垂直偏移。

那么可求得：

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

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

两个距离点的差分为：

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

记△y 和△x 分别为线段端点的垂直和水平偏离量(整数)，令 m = △y / △x，代入等式(1)。

可以得到：

$$\triangle x(d_1 - d_2) = 2\triangle y x_k - 2\triangle x y_k + c \tag{2}$$

其中：

$$c = 2\triangle y + \triangle x(2b - 1)$$

2.1.3 决策参数

由等式(2)我们定义 p_k 如下：

$$p_k = \triangle x(d_1 - d_2) = 2\triangle y x_k - 2\triangle x y_k + c$$

其中，p_k 称为 Bresenham 画线算法中第 k 步的决策参数，并且它的符号与$(d_1 - d_2)$的符号相同($\triangle x > 0$)。根据第 k 步决策参数的符号，可判定两候选像素与线段的偏移关系，从而确定第 k+1 步选择的像素。

若 $p_k > 0$，即：$d_2 < d_1$，说明 y_{k+1} 处像素比 y_k 处像素更接近于线段点，选择较高像素(x_{k+1}, y_{k+1})；若 $p_k < 0$，即：$d_1 < d_2$，说明 y_k 处像素比 y_{k+1} 处像素更接近于线段点，选择较低像素(x_{k+1}, y_k)。

第 k 步的决策参数为：

$$p_k = 2\triangle y x_k - 2\triangle x y_k + c$$

第 k+1 步的决策参数为：

$$p_{k+1} = 2\triangle y x_{k+1} - 2\triangle x y_{k+1} + c$$

上述两方程相减，可得到：

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

又：

$$x_{k+1} = x_k + 1$$

故得到决策参数计算的增量公式：

$$p_{k+1} = p_k + 2\triangle y - 2\triangle x(y_{k+1} - y_k) \tag{3}$$

①若 $p_k > 0$，取高像素(x_{k+1}, y_{k+1})，即 $y_{k+1} - y_k = 1$。

则有：

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

②若 $p_k < 0$，取低像素(x_{k+1}, y_k)，即 $y_{k+1} - y_k = 0$。

则有：

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

2.1.4 生成过程

以下仅考虑当$|m| < 1$时，m 在其他取值范围中时情况类似。

①输入线的两个端点，并将左端点存贮在(x_0, y_0)中；

②将(x_0, y_0)装入帧缓冲器，画第一个点；

③计算常量：$\triangle x$、$\triangle y$、$2\triangle y$ 和 $2\triangle y - 2\triangle x$，起始位置$(x_0, y_0)$的决策参数 p_0 计算为：$p_0 = 2\triangle y - \triangle x$；

④从 k = 0 开始，在每个离散取样点 x_k 处，进行下列检测：

若 $p_k < 0$，画点(x_{k+1}, y_k)，且：$p_{k+1} = p_k + 2\triangle y$

若 $p_k > 0$，画点(x_{k+1}, y_{k+1})，且：$p_{k+1} = p_k + 2\triangle y - 2\triangle x$

⑤k = k+1；

⑥重复步骤 4，共$\triangle x$ 次。

2.2 曲线

绘制曲线利用的是 Bezier 曲线绘制。

2.2.1 Bezier 曲线定义

对一般 Bezier 曲线，最方便的是混合函数形式。

给定 $P_k = (x_k, y_k, z_k)$(其中 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 P_0 和 P_n 间的逼近 Bezier 多项式函数的路径(Bezier 曲线)：

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

上式中混合函数 $BEZ_{k,n}(u)$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：

$$BEZ_{k,k}(u) = u^k,\quad BEZ_{0,k}(u) = (1-u)^k$$

2.2.2 Bezier 曲线离散点生成

下图为三次 Bezier 曲线在某个 u 值下的计算过程：

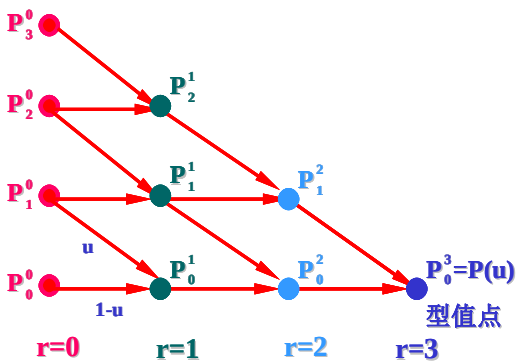


Fig. 2 三次 Bezier 曲线在某个 u 值下的计算过程

当 u 在 0 到 1 之间变化时，对应生成多个离散型值点。

2.3 多边形

绘制多边形涉及到的实验原理基本与绘制直线类似，故在此不再赘述。

2.4 圆与椭圆

绘制圆利用的是中点圆生成算法，绘制椭圆利用的是中点椭圆生成算法。由于绘制圆和椭圆的算法区别不是很大，故在此只介绍中点圆生成算法，中点椭圆生成算法不再赘述。

2.4.1 中点圆生成算法概述

中点圆生成算法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。

2.4.2 决策参数

定义圆函数：

$$f_{circle}(x, y) = x_2 + y_2 - r_2$$

圆边界上具有半径 r 的点(x, y)满足方程：

$$f_{circle}(x, y) = 0$$

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。

中点圆生成算法中圆函数的检测在每个离散取样位置上对两候选像素中点进行，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：

$$p_k = f_{circle}(x_{k+1}, (yk+1 + yk) / 2)$$

其中：

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

所以：

$$p_k = f_{circle}(x_{k+1}, y_k - 0.5)$$

假如 $p_k < 0$，中点在圆周边界内，扫描线 y_k 上的像素(高像素)接近于圆边界，选择像素位置(x_{k+1}, y_k)；

假如 $p_k > 0$，中点位于圆周边界外，扫描线 y_{k-1} 的像素(低像素)接近于圆边界，选择像素位置(x_{k+1}, y_{k-1});

假如 $p_k = 0$，中点位于圆周边界上，选低像素。

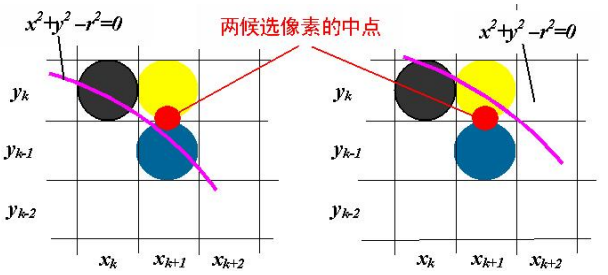


Fig. 3 沿圆路径取样位置 x_{k+1} 处候选像素间的中点

第 k 个决策参数是圆函数在其两候选像素中点处求值：

$$p_k = f_{circle}(x_{k+1}, y_k - 0.5)$$

类似地，第 k + 1 个决策参数也是圆函数在其两候选像素中点处求值：

$$p_{k+1} = f_{circle}(x_{k+2}, (y_{k+2} + y_{k+1}) / 2)$$

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 $p_{k+1} = f_{circle}(x_{k+2}, y_k - 1.5)$。

2.4.3 算法过程

①输入圆半径 r 和圆心(x_c, y_c)，并得到圆心在原点的圆周上的第一点为：

$$(x_0, y_0) = (0, r)$$

②计算圆周点(0, r)的初始决策参数值为：$p_0 = 5 / 4 - r$;

③从 k = 0 开始每个取样位置 x_k 位置处完成下列检测：

若 $p_k < 0$，选择像素位置(x_{k+1}, y_k)，且通过决策参数增量计算得：$p_{k+1} = p_k + 2x_{k+1} + 1$

若 $p_k > 0$，选择像素位置(x_{k+1}, y_{k-1})，且通过决策参数增量计算得：$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$

其中：$2x_{k+1} = 2x_k + 2$，$2y_{k+1} = 2y_k - 2$。

④确定在其它七个八分圆中的对称点；

⑤将计算出的像素位置(x, y)移动到中心在(x_c, y_c)的圆路径上，即对像素位置进行平移：

$$x = x + x_c，\ y = y + y_c$$

⑥重复步骤 3 到 5，直至 $x \geq y$。

2.5 填充区域

填充区域利用的是扫描填充图元生成方法。

2.5.1 基本原理

从多边形的顶点信息出发，求出位于其内部的各个像素(将多边形的顶点表示转换成点阵表示)，并将其

各个像素的颜色值写入帧缓冲器中相应的单元。

2.5.2　基本过程

对每条横越多边形的扫描线：

①求交点：确定其与多边形边的交点位置(一般，有偶数个交点)；

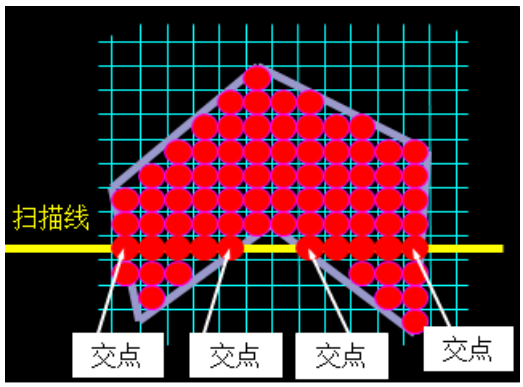


Fig. 4 填充基本过程—求交点

②按横坐标排序：将这些交点沿扫描线按横坐标值自左至右排列；

③配对存储：自左至右分对存贮；

④取整和填充：对交点坐标取整，并给每对交点间扫描线区段内的像素点的帧缓冲器位置设置指定填充颜色。

2.5.3　求交点的增量计算

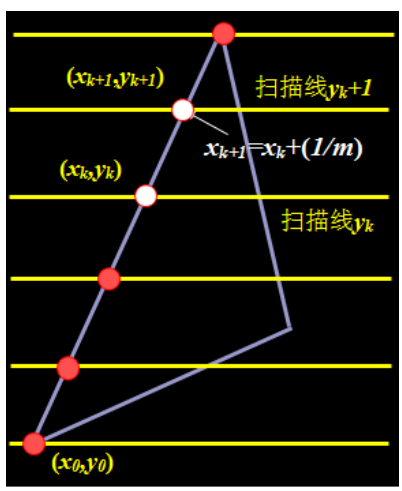


Fig. 5 求交点的增量计算

在图形算法中常利用待显示场景的各种连贯性，来减少计算处理。这里我们利用直线连贯性，在确定扫描线与边交点时可采用增量计算。

两相邻扫描线间 y 坐标变化为：

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

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：

$$x_{k+1} = x_k + (1 / m)$$

每个后继交点 x 值都可由前一条扫描线上交点值 x_k 加斜率的倒数并取整而增量地计算。

第 k 条扫描线相对于最初扫描线与一条边的交点 x_k 值计算为：

$$x_k = x_0 + (k / m)$$

2.5.4　求交过程的有序边表

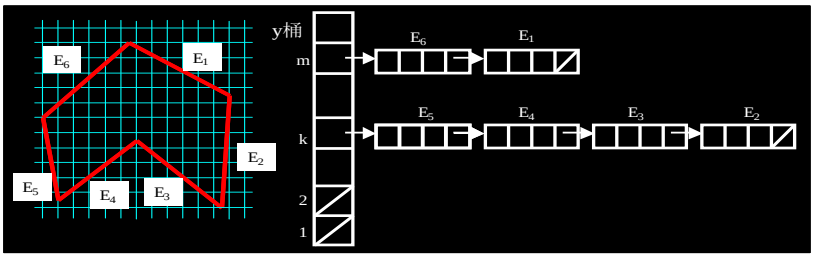


Fig. 6 有序边表

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，它为每条扫描线建立一个存储单元或“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 x 值或△x 值递增的顺序排列成链表。

链表中的每个节点包含该边最大 y 值、边下端点 x 坐标值和边斜率倒数(即 1/m)。

有序边表的作用是：排除不必要的求交测试。

2.5.5　求交过程的活化边表

活化边表是记录多边形边沿扫描线的交点序列，简化后续计算过程，它由与当前扫描线相交的多边形边组成。

对第 k 条扫描线，活化边表可从有序边表中生成：

①y 桶对应于当前扫描线以下类 c(c = 1, 2, …, k - 1)所有边插入活化边表；

②删除满足 $y_{max} < y_k$ 的边；

③其它边的 x 域根据 $x_{k+1} = x_k + ((y_k - y_c) / m)$确定交点。

活化边表中的每个节点包含该边最大 y 值、与扫描线交点的 x 坐标值和边斜率倒数。

2.6 多边形裁剪

多边形裁剪利用的是 Sutherland-Hodgman 算法。

2.6.1　Sutherland-Hodgman 算法概述

Sutherland-Hodgman 算法是基于逐边裁剪的方式，它的基本裁剪过程如下：

①对多边形顶点集初始化(顺或逆时针排序)；

②依次用裁剪窗口左右上下边界对多边形裁剪；

③每次裁剪结果依次作为下次裁剪的输入，其中每次裁剪结果 = 新顶点序列 + 原顶点系列。

2.6.2　Sutherland-Hodgman 算法原理

相交方式决定了输出顶点的个数和类型，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：

①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。

②起始和终止点都在窗口边界内侧空间，则只有终止点输出。

③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。

④起始和终止点都在窗口外侧空间，不增加任何点。

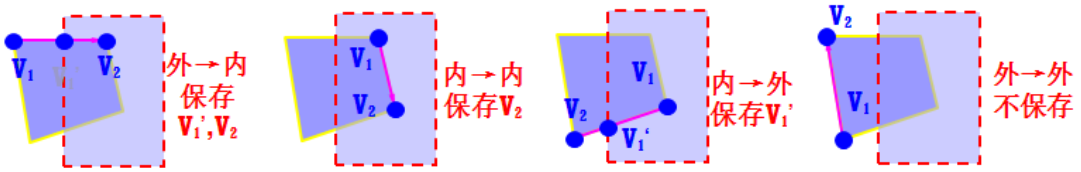


Fig. 7 多边形边界与裁剪窗口的各种相交方式

裁剪结果多边形顶点由两部分组成：

①落在裁剪窗口内侧空间的原多边形顶点；

②多边形边与裁剪窗口边界的交点。

只要将这两部分顶点按与处理过程相同的顺序(顺时针或逆时针)连接起来，就得到裁剪结果多边形。

2.7 图形的变换

图形的变换主要包括平移、旋转和缩放，可利用变换的齐次坐标矩阵表示。

2.7.1 平移变换

平移变换可表示为：

$$P_1 = T(t_x, t_y) \cdot P$$

其中，$T(t_x, t_y)$为平移矩阵。

$$T = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

Fig. 8 平移矩阵 $T(t_x, t_y)$

2.7.2 旋转变换

绕坐标原点旋转变换可写为：

$$P1 = R(\theta) \bullet P$$

其中，$R(\theta)$为旋转矩阵。

$$R = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Fig. 9 旋转矩阵 $R(\theta)$

2.7.3 缩放变换

相对于坐标原点的缩放变换表示为：

$$P_1 = S(s_x, s_y) \cdot P$$

其中，$S(s_x, s_y)$为缩放矩阵。

$$S = \begin{bmatrix} s_x & 0 & 1 \\ 0 & s_y & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Fig. 10 缩放矩阵 S(s_x, s_y)

2.8 绘制3D立方体的旋转

绘制 3D 立方体的旋转主要依赖于调用了实验环境中提供的 API。

2.9 图形的存储

实现图形的保存输出主要依赖于调用了实验环境中提供的 API。

3 实验设计

3.1 直线

3.1.1 直线的绘制

关于绘制直线，我运用了 Bresenham 算法，该算法在课上讲得比较易懂，并且在课件中也提供了当直线斜率 k 满足 0 < k < 1 时绘制直线的伪代码。

个人感觉主要就是需要按照直线斜率 k 的取值，进行分类讨论，写出各个情况时绘制图形的不同实现代码即可。以下是当直线斜率 k 满足 0 < k < 1 时绘制直线的 C++语言实现代码：

```
int ddx = x_max - x_min;
int ddy = y_max - y_min;
int x = x_min;
int y = y_min;
int p = 2 * ddy - ddx;
for (; x <= x_max; x++)
{
    glVertex2i(x, y);
    if (p >= 0)
    {
        y++;
        p += 2 * (ddy - ddx);
    }
    else
    {
        p += 2 * ddy;
    }
}
```

k 取其他值时较类似，在此不赘。

3.1.2 直线的编辑

编辑直线主要是通过使用回调函数 mouseButton(int button, int state, int x, int y)来响应鼠标操作产生的消息事件。每次改变直线某个端点的位置，就重新调用绘制直线的算法来实现直线的重绘：

```
if (button == GLUT_LEFT_BUTTON)
{
    if (state == GLUT_DOWN)
    {
        if ((abs(x - lines[lines.size() - 1].x_1) < 10) && (abs(CurrentHeight
```

```
- y - lines[lines.size() - 1].y_1) < 10))
        {   //start to edit one end point of the line
            lines[lines.size() - 1].x_1 = lines[lines.size() - 1].x_2;
             lines[lines.size() - 1].y_1 = lines[lines.size() - 1].y_2;
            left_button_down = 1;
            system_state = LINE_STATE2;
        }
        else
        {   //start to edit the other end point of the line
             /* ... */
         }
    }
}
```

由以上实现代码可以知道，我们是通过判断鼠标左键点击时，鼠标指针位置是否和直线的某个端点靠得较近来确定用户是否要开始编辑该直线。否则，我们开始绘制下一条直线。

3.2 曲线

3.2.1 曲线的绘制

曲线的绘制主要利用了三次 Bezier 曲线的函数表达式。

```
void drawBezierCurve(Curve curve)
{
    glColor3f(0, 0, 0);
    glBegin(GL_LINE_STRIP);
    for (int i = 1; i <= 1000; i++)
    {
        GLfloat t = i / 1000.0;
        GLfloat b0 = pow(1 - t, 3.0);
        GLfloat b1 = 3.0 * t * pow(1 - t, 2.0);
        GLfloat b2 = 3.0 * t * t * (1 - t);
        GLfloat b3 = t * t * t;

        GLfloat x = curve.p1.x_1 * b0 + curve.p2.x_1 * b1 + curve.p3.x_1 *
b2 + curve.p4.x_1 * b3;
        GLfloat y = curve.p1.y_1 * b0 + curve.p2.y_1 * b1 + curve.p3.y_1 *
b2 + curve.p4.y_1 * b3;
        glVertex2f(x, y);
    }
    glEnd();
    return;
}
```

3.2.2 曲线的编辑

曲线的编辑主要是基于对 Bezier 曲线控制多边形的编辑，每次控制多边形发生改变，其对应的 Bezier 曲线也会发生变动，即重绘。

3.3 多边形

3.3.1 多边形的绘制

绘制多边形其实是绘制直线的升级版，相当于绘制很多直线，所以主要涉及了状态机设计的一些知识。

```
case POLYGON_STATE1:
    if (button == GLUT_LEFT_BUTTON)
    {
        if (state == GLUT_DOWN)
```

```
        {   //start to draw the polygon
            system_state = POLYGON_STATE2;
            /* ... */
            newLine.x_1 = x;
            newLine.y_1 = CurrentHeight - y;
            newLine.x_2 = x;
            newLine.y_2 = CurrentHeight - y;
            /* ... */
        }
    }
    break;
case POLYGON_STATE2:
    /* ... */
    break;
case POLYGON_STATE3:
    if (button == GLUT_LEFT_BUTTON)
    {
        if (state == GLUT_DOWN)
        {
            if ((abs(x - polygons[polygons.size() - 1][0].x_1) < 5) &&
(abs(CurrentHeight - y - polygons[polygons.size() - 1][0].y_1) < 5))
            {   //draw the last line of the polygon
                system_state = POLYGON_STATE4;
                /* ... */
            }
            else
            {   //draw the next line of the polygon
                system_state = POLYGON_STATE2;
                /* ... */
                newLine.x_1 = polygons[polygons.size() -
1][polygons[polygons.size() - 1].size() - 1].x_2;
                newLine.y_1 = polygons[polygons.size() -
1][polygons[polygons.size() - 1].size() - 1].y_2;
                newLine.x_2 = polygons[polygons.size() -
1][polygons[polygons.size() - 1].size() - 1].x_2;
                newLine.y_2 = polygons[polygons.size() -
1][polygons[polygons.size() - 1].size() - 1].y_2;
                /* ... */
            }
        }
    }
    break;
```

由以上实现代码可以知道绘制多边形过程中状态 POLYGON_STATE1 表示开始绘制多边形的第一条边，状态 POLYGON_STATE3 表示开始绘制多边形的下一条边或最后一条边。

3.3.2 多边形的编辑

编辑多边形也和编辑直线比较类似，主要要也是通过使用回调函数 mouseButton(int button, int state, int x, int y)来响应鼠标操作产生消息事件。每次改变多边形某个顶点的位置，就重新调用绘制直线的算法来实现与该顶点相邻的两条直线的重绘：

```
case POLYGON_STATE4:
    /* ... */
    break;
case POLYGON_STATE5: //edit polygon state
```

```
    if (button == GLUT_LEFT_BUTTON)
    {
        if (state == GLUT_DOWN)
        {
            int i = 0;
            for (; i < polygons[polygons.size() - 1].size(); i++)
            {    //judge whether one of vertices of the polygon needs to edit
                if ((abs(x - polygons[polygons.size() - 1][i].x_1) < 10) &&
(abs(CurrentHeight - y - polygons[polygons.size() - 1][i].y_1) < 10))
                    break;
            }
            if (i < polygons[polygons.size() - 1].size())
            {
                left_button_down = 1;
                //(i - 1)th and ith edge need to be edited;
                edit_polygon_point = i;
                system_state = POLYGON_STATE5;
            }
            else
            {   //draw the next polygon
                /* ... */
            }
        }
        /* ... */
    }
    break;
```

由以上实现代码可以知道编辑多边形过程中状态 POLYGON_STATE5 主要表示正在编辑多边形的状态。我们是通过判断鼠标左键点击时，鼠标指针位置是否和多边形某个顶点靠得较近来确定用户是否要开始编辑该多边形。否则，我们开始绘制下一个多边形。

3.4 圆与椭圆

同样地，圆与椭圆的设计思路比较相似，故在这里只介绍圆的部分，椭圆不再赘述。

3.4.1 圆的绘制

关于绘制圆，我运用了圆的中点生成算法，该算法在上课时讲得比较清楚易懂。

```
void drawCircles(Circle c)
{
    glColor3f(0, 0, 0);
    glBegin(GL_POINTS);
    //consider x = 0, y = 0; afterwards we move all the points
    int p0 = 5 / 4 - c.r;
    int x = 0;
    int y = c.r;
    int p = p0;
    while (x <= y)
    {
        glVertex2i(x + c.x, y + c.y);
        glVertex2i(y + c.x, x + c.y);
        glVertex2i((-1) * x + c.x, (-1) * y + c.y);
        glVertex2i((-1) * y + c.x, (-1) * x + c.y);
        glVertex2i((-1) * x + c.x, y + c.y);
        glVertex2i(y + c.x, (-1) * x + c.y);
        glVertex2i(x + c.x, (-1) * y + c.y);
```

```
        glVertex2i((-1) * y + c.x, x + c.y);
        if (p < 0)
        {
            p = p + 2 * x + 3;
            x = x + 1;
            y = y;
        }
        else if (p >= 0)
        {
            p = p + 2 * x + 5 - 2 * y;
            x = x + 1;
            y = y - 1;
        }
    }
    glEnd();
    return;
}
```

由以上代码可知，我们先假设以(0,0)为圆心画圆，之后再平移到指定圆心处。

并且我们只需画整个圆的 1/8 部分，然后将剩下的部分通过横坐标 x 和纵坐标 y 的映射变化就能顺利画出来了。

3.4.2 圆的编辑

编辑圆主要是通过使用回调函数 mouseButton(int button, int state, int x, int y)来响应鼠标操作产生的消息事件。每次改变圆外切正方形某个端点的位置，就重新调用绘制圆的算法来实现圆的重绘：

```
if (button == GLUT_LEFT_BUTTON)
{
    if (state == GLUT_DOWN)
    {
        if ((abs(x - circleBounds[0].x_1) < 10) && (abs(CurrentHeight - y
- circleBounds[0].y_1) < 10))
        {
            left_button_down = 1;
            system_state = CIRCLE_STATE2;
        }
        else
        {
            //other cases
            /* ... */
        }
    }
}
```

由以上实现代码可以知道，我们是通过判断鼠标左键点击时，鼠标指针位置是否和圆外切正方形的某个端点靠得较近来确定用户是否要开始编辑该圆。

3.5 填充区域

3.5.1 填充区域的绘制

绘制填充区域实现算法有一部分和绘制多边形实现算法重复，在此不赘。与绘制多边形不同的是，我们添加了 int isFilledAreaEdit 和 int isFilledAreaEnd 两个变量来共同标识是否应该给多边形内部区域填充颜色。

以下是扫描填充图元生成算法：

```
vector<list<Node>> ET;
list<Node> AET;
void fillArea(vector<Line> p)
```

```
{
    glColor3f(0, 0, 0);
    glBegin(GL_POINTS);

    vector<floatPoint> floatPoints;

    //find the y_min and y_max
    /* ... */

    //y_min <=> 0, y_max <=> y_max - y_min
    for (int i = y_min; i <= y_max; i++)
    {
        list<Node> nodeList;
        ET.push_back(nodeList);
    }

    for (int i = 0; i <= y_max - y_min; i++)
    {
        for (int j = 0; j < floatPoints.size(); j++)
        {
            if (floatPoints[j].y_1 == y_min + i)
            {
                if (floatPoints[(j - 1 + floatPoints.size()) %
(floatPoints.size())].y_1 > floatPoints[j].y_1)
                {
                    Node newNode;
                    newNode.x = floatPoints[j].x_1;
                    newNode.y_max = floatPoints[(j - 1 + floatPoints.size()) %
(floatPoints.size())].y_1;
                    newNode.dx = (floatPoints[(j - 1 + floatPoints.size()) %
(floatPoints.size())].x_1 - floatPoints[j].x_1) / (floatPoints[(j - 1 +
floatPoints.size()) % (floatPoints.size())].y_1 - floatPoints[j].y_1);
                    ET[i].push_front(newNode);
                }

                //the other case
                /* ... */
            }
        }
    }

    for (int i = 0; i <= y_max - y_min; i++)
    {
        //add x;
        for (list<Node>::iterator it = AET.begin(); it != AET.end(); it++)
        {
            (*it).x += (*it).dx;
        }

        //sort AET
        AET.sort(compareFun);

        //clear some node whose y_max no more than (i + y_min) in AET
        for (list<Node>::iterator it = AET.begin(); it != AET.end(); )
```

```
        {
            if ((*it).y_max == (i + y_min))
            {
                list<Node>::iterator it2 = it;
                it2++;
                AET.erase(it);
                it = it2;
            }
            else
            {
                it++;
            }
        }

        //push new node into AET and sort
        for (list<Node>::iterator it = ET[i].begin(); it != ET[i].end();
it++)
        {
            AET.push_front(*it);
        }
        AET.sort(compareFun);

        //draw between pairs of points
        for (list<Node>::iterator it = AET.begin(); it != AET.end();)
        {
            list<Node>::iterator it2 = it;
            it2++;
            for (int j = (*it).x; j <= (*(it2)).x; j++)
            {
                glVertex2i(j, i + y_min);
            }
            it++;
            it++;
        }
    }

    //clear the tables
    /* ... */

    glEnd();
    return;
}
```

此算法中主要涉及 ET 和 AET 两张表，算法主要是对 AET 进行不断地修改，然后最终画出指定区域的点集。每一轮对 AET 的修改过程主要是这样的：

1. 将 AET 中每一个元素的 x 值增加 dx
2. 对 AET 中的元素基于 x 值进行排序
3. 删除 AET 表中 y_max 值小于等于扫描线高度的元素
4. 将 ET 表中最低点与当前扫描线高度一致的直线对应的元素压入 AET 中，并对 AET 中的元素基于 x 值进行排序
5. 在当前扫描线高度上，在 AET 中保存的成对的点之间画点。

3.5.2 填充区域的编辑

编辑填充区域实现算法有一部分和编辑多边形实现算法重复，在此不赘。与绘制多边形不同的是，我们添加了 int isFilledAreaEdit 和 int isFilledAreaEnd 两个变量来共同标识是否应该给多边形内部区域填充颜色。

主要也是通过使用回调函数 mouseButton(int button, int state, int x, int y)来响应鼠标操作产生消息事件。每次改变填充区域某个顶点的位置，就重新调用绘制直线、区域填充的算法来实现对填充区域的重绘。我们是通过判断鼠标左键点击时，鼠标指针位置是否和填充区域某个顶点靠得较近来确定用户是否要开始编辑该填充区域。

3.6 多边形的裁剪

多边形的裁剪主要是基于裁剪的基本过程：

①对多边形顶点集初始化(顺或逆时针排序)；

②依次用裁剪窗口左右上下边界对多边形裁剪；

③每次裁剪结果依次作为下次裁剪的输入，其中每次裁剪结果 = 新顶点序列 + 原顶点系列。

```
for (int i = 0; i < 4; i++)
{
    //initialization
    /* ... */
    if (tempPolygon.size() == 0)
    {
        //corner case
        /* ... */
    }
    else if (tempPolygon.size() == 1)
    {
        //corner case, too
        /* ... */
    }
    else
    {
        //initialization, too
        /* ... */
        for (int j = 0; j < tempPolygon.size(); j++)
        {
            //initialization, too, too
            /* ... */
            if (!(isOutside(lineTemp, peStop)))
            {
                if (!lpii)
                {
                    lpii = 1;
                    //calculate the new point
                    floatPoint newPoint;
                    if (i == 0)
                    {
                        newPoint.y_1 = yMin;
                        newPoint.x_1 = peStart.x_1 + (newPoint.y_1 -
peStart.y_1) * (peStop.x_1 - peStart.x_1) / (peStop.y_1 - peStart.y_1);
                    }
                    else if (i == 1)
                    {
                        newPoint.x_1 = xMin;
```

```
                    newPoint.y_1 = (newPoint.x_1 - peStart.x_1) *
(peStop.y_1 - peStart.y_1) / (peStop.x_1 - peStart.x_1) + peStart.y_1;
                }
                else if (i == 2)
                {
                    newPoint.y_1 = yMax;
                    newPoint.x_1 = peStart.x_1 + (newPoint.y_1 -
peStart.y_1) * (peStop.x_1 - peStart.x_1) / (peStop.y_1 - peStart.y_1);

                }
                else if (i == 3)
                {
                    newPoint.x_1 = xMax;
                    newPoint.y_1 = (newPoint.x_1 - peStart.x_1) *
(peStop.y_1 - peStart.y_1) / (peStop.x_1 - peStart.x_1) + peStart.y_1;
                }
                newPoints.push_back(newPoint);
            }
            newPoints.push_back(peStop);
        }
        else
        {   //the other case
            /* ... */
        }
        peStart.x_1 = peStop.x_1;
        peStart.y_1 = peStop.y_1;
    }
  }
  cweStart.x_1 = cweStop.x_1;
  cweStart.y_1 = cweStop.y_1;
  tempPolygon.swap(newPoints);
  newPoints.clear();
}
```

3.7 图形的变换

3.7.1 平移变换

实现平移的算法比较简单，以填充区域为例，我们只需将填充区域的每个顶点的横坐标、纵坐标的值进行一定增加或减小即可。

实现算法如下：

```
if (key == 'a')
{
    for (int i = 0; i < filledAreas[filledAreas.size() - 1].size(); i++)
    {
        filledAreas[filledAreas.size() - 1][i].x_1 -= 1;
        filledAreas[filledAreas.size() - 1][i].x_2 -= 1;
    }
}
else if (key == 'd')
{
    /* ... */
}
else if (key == 'w')
{
```

```
    /* ... */
}
else if (key == 's')
{
    /* ... */
}
```

3.7.2 旋转变换

实现旋转的算法比较简单，以填充区域为例，我们将填充区域的每个顶点绕着填充区域第一个选定顶点进行旋转，旋转之后得到的点坐标表示方法可以参照教材中旋转矩阵相关内容而得。

实现算法如下：

```
else if (key == 'q')
{
    vector<vector<Line>> temp = filledAreas;
    int x0 = filledAreas[filledAreas.size() - 1][0].x_1;
    int y0 = filledAreas[filledAreas.size() - 1][0].y_1;

    for (int i = 0; i < filledAreas[filledAreas.size() - 1].size(); i++)
    {
        filledAreas[filledAreas.size() - 1][i].x_1 = (temp[temp.size() -
1][i].x_1 - x0) * 0 - (temp[temp.size() - 1][i].y_1 - y0) * 1 + x0;
        filledAreas[filledAreas.size() - 1][i].y_1 = (temp[temp.size() -
1][i].x_1 - x0) * 1 + (temp[temp.size() - 1][i].y_1 - y0) * 0 + y0;
        filledAreas[filledAreas.size() - 1][i].x_2 = (temp[temp.size() -
1][i].x_2 - x0) * 0 - (temp[temp.size() - 1][i].y_2 - y0) * 1 + x0;
        filledAreas[filledAreas.size() - 1][i].y_2 = (temp[temp.size() -
1][i].x_2 - x0) * 1 + (temp[temp.size() - 1][i].y_2 - y0) * 0 + y0;
    }

    for (int i = 0; i < temp.size(); i++)
    {
        temp[i].clear();
    }
    temp.clear();
}
else if (key == 'e')
{
    /* ... */
}
```

3.7.3 缩放变换

实现缩放的算法比较简单，以填充区域为例，我们将填充区域以绘图窗口左下角为基点进行缩放，缩放之后得到的点坐标表示方法可以参照教材中缩放矩阵相关内容而得。

实现算法如下：

```
else if (key == 'z')
{
    if (resizeFilledArea > 0)
    {
        for (int i = 0; i < filledAreas[filledAreas.size() - 1].size(); i++)
        {
            filledAreas[filledAreas.size() - 1][i].x_1 =
filledAreas[filledAreas.size() - 1][i].x_1 / 2;
            filledAreas[filledAreas.size() - 1][i].y_1 =
```

```
filledAreas[filledAreas.size() - 1][i].y_1 / 2;
            filledAreas[filledAreas.size() - 1][i].x_2 =
filledAreas[filledAreas.size() - 1][i].x_2 / 2;
            filledAreas[filledAreas.size() - 1][i].y_2 =
filledAreas[filledAreas.size() - 1][i].y_2 / 2;
        }
        resizeFilledArea--;
    }
}
else if (key == 'x')
{
    /* ... */
}
```

其中变量 resizeFilledArea 的作用主要关系到我们的规定，即直线、多边形、填充图形、圆和椭圆进行缩放的操作过程中，图形的大小不能小于最初时的大小（即要想实现缩小，必须首先放大，规定这一点的目的是防止图形在一开始直接缩小后再放大会造成图形相关长度信息与最初相比发生改变，这是由数据类型为整型导致的）。

3.8 绘制三维六面体

绘制 3D 立方体的旋转主要依赖于调用了实验环境中提供的 API。

```
void rotate3D()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glTranslatef(0.0f, 0.0f, -10.0f);
    glRotatef(angle, -1.0f , 1.0f, 1.0f);
    glBegin(GL_QUADS);

    glColor3f(0.0f, 1.0f, 0.0f);
    glVertex3f(1.0f , 1.0f, -1.0f);
    glVertex3f(-1.0f , 1.0f, -1.0f);
    glVertex3f(-1.0f , 1.0f, 1.0f);
    glVertex3f(1.0f , 1.0f, 1.0f);

    /* ... */

    glEnd();
    angle -= 1.0f;
    return;
}
```

3.9 图形的存储

实现图形的保存输出主要依赖于调用了实验环境中提供的 API。

```
void grab()
{
    //  FILE*    pDummyFile;  //指向另一 bmp 文件，用于复制它的文件头和信息头数据
    FILE*    pWritingFile;  //指向要保存截图的 bmp 文件
    GLubyte* pPixelData;    //指向新的空的内存，用于保存截图 bmp 文件数据
    //  GLubyte  BMP_Header[BMP_Header_Length];
    GLint    i, j;
    GLint    PixelDataLength;   //BMP 文件数据总长度
```

```
    // 计算像素数据的实际长度
    i = CurrentWidth * 3;   // 得到每一行的像素数据长度
    while (i % 4 != 0)      // 补充数据，直到 i 是的倍数
        ++i;
    PixelDataLength = i * CurrentHeight;  //补齐后的总位数

    // 分配内存和打开文件
    pPixelData = (GLubyte*)malloc(PixelDataLength);
    if (pPixelData == 0)
        exit(0);

    //  pDummyFile = fopen("bitmap1.bmp", "rb");//只读形式打开
    //  if (pDummyFile == 0)
    //      exit(0);

    pWritingFile = fopen("grab.bmp", "wb"); //只写形式打开
    if (pWritingFile == 0)
        exit(0);

    //把读入的 bmp 文件的文件头和信息头数据复制，并修改宽高数据
    //  fread(BMP_Header, sizeof(BMP_Header), 1, pDummyFile);  //读取文件头
和信息头，占据 54 字节
    fwrite(BMP_Header, sizeof(BMP_Header), 1, pWritingFile);
    fseek(pWritingFile, 0x0012, SEEK_SET); //移动到 0X0012 处，指向图像宽度所
在内存
    i = CurrentWidth;
    j = CurrentHeight;
    fwrite(&i, sizeof(i), 1, pWritingFile);
    fwrite(&j, sizeof(j), 1, pWritingFile);

    // 读取当前画板上图像的像素数据
    glPixelStorei(GL_UNPACK_ALIGNMENT, 4);  //设置 4 位对齐方式
    glReadPixels(0, 0, CurrentWidth, CurrentHeight,
        GL_BGR_EXT, GL_UNSIGNED_BYTE, pPixelData);

    // 写入像素数据
    fseek(pWritingFile, 0, SEEK_END);
    //把完整的 BMP 文件数据写入 pWritingFile
    fwrite(pPixelData, PixelDataLength, 1, pWritingFile);

    // 释放内存和关闭文件
    //  fclose(pDummyFile);
    fclose(pWritingFile);
    free(pPixelData);
}
```

4 实验测试

4.1 直线

4.1.1 绘制直线

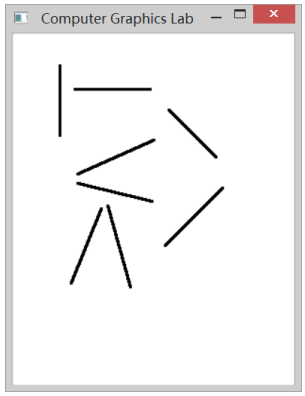


Fig. 11 直线绘制

4.1.2 编辑直线

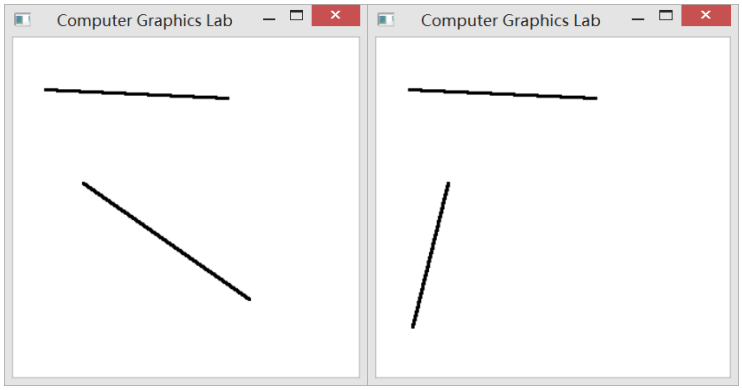


Fig. 12 直线编辑前后

4.2 曲线

4.2.1 绘制曲线

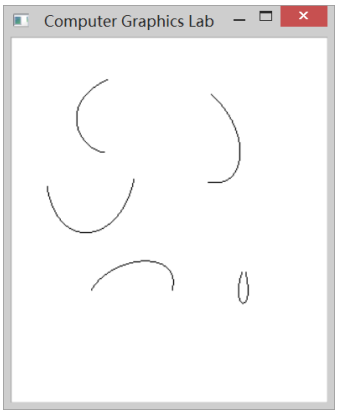


Fig. 13 曲线绘制

4.2.2 编辑曲线

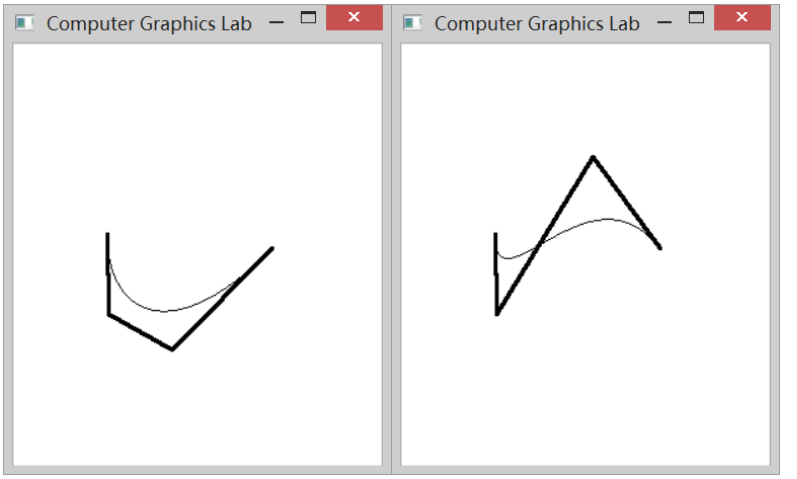


Fig. 14 曲线编辑前后

4.3 多边形

4.3.1 绘制多边形

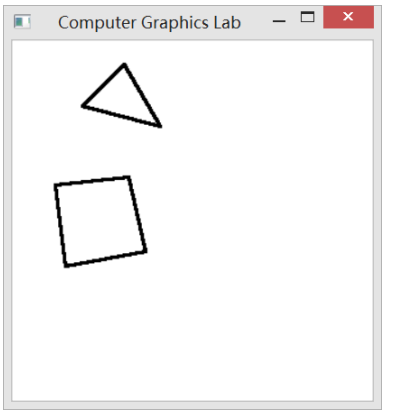


Fig. 15 多边形绘制

4.3.2 编辑多边形

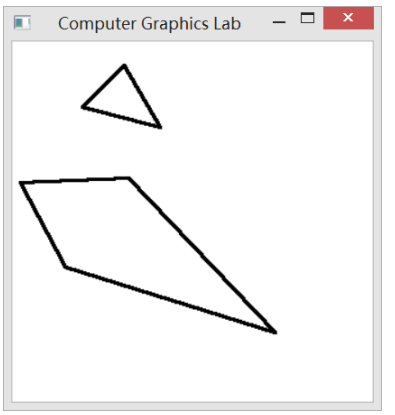


Fig. 16 多边形编辑

4.4 圆与椭圆

4.4.1 绘制圆

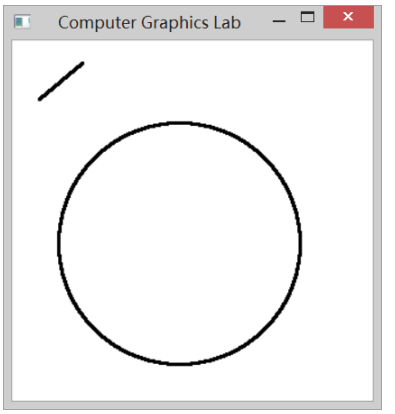


Fig. 17 圆的绘制

4.4.2 绘制椭圆

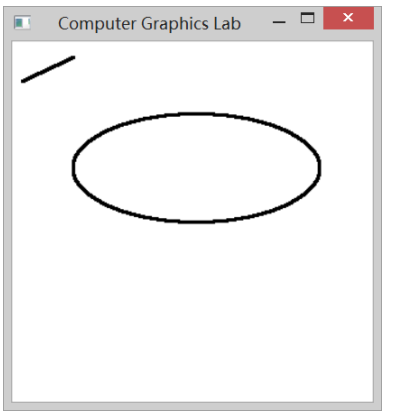


Fig. 18 椭圆的绘制

4.5 填充区域

4.5.1 绘制填充区域

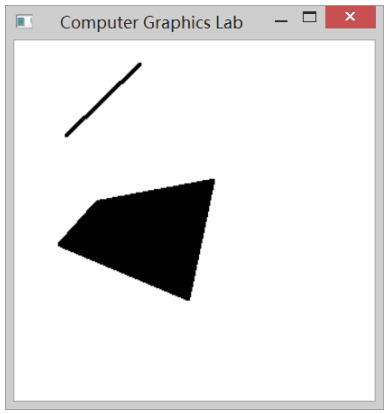


Fig. 19 填充区域的绘制

4.5.2 编辑填充区域

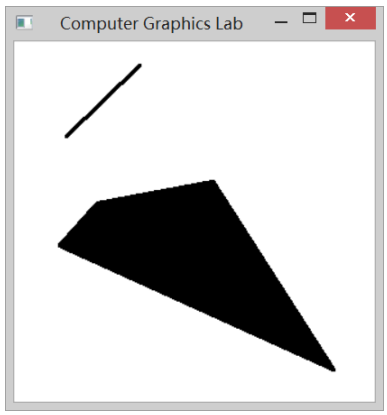
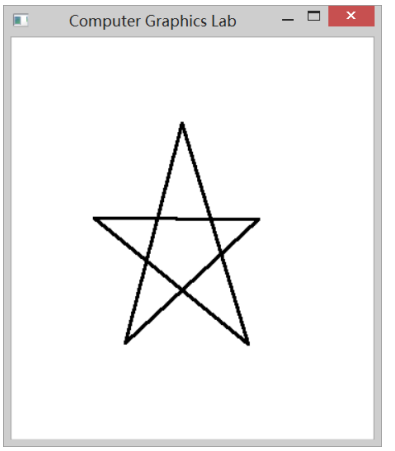


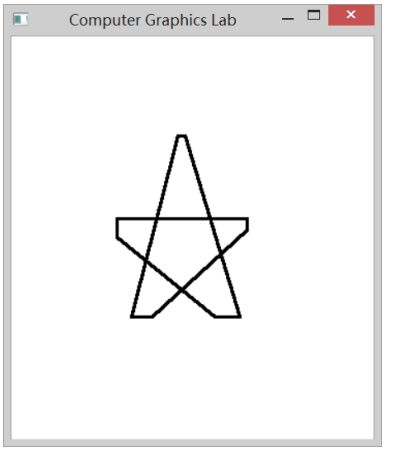
Fig. 20 填充区域的编辑

4.6 多边形裁剪

4.6.1 裁剪前



4.6.2 裁剪后



4.7 图形的变换

4.7.1 平移变换

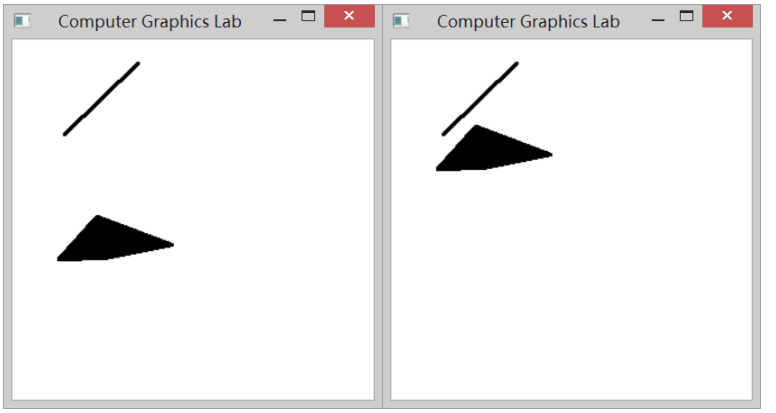


Fig. 21 平移变换前后

4.7.2 旋转变换

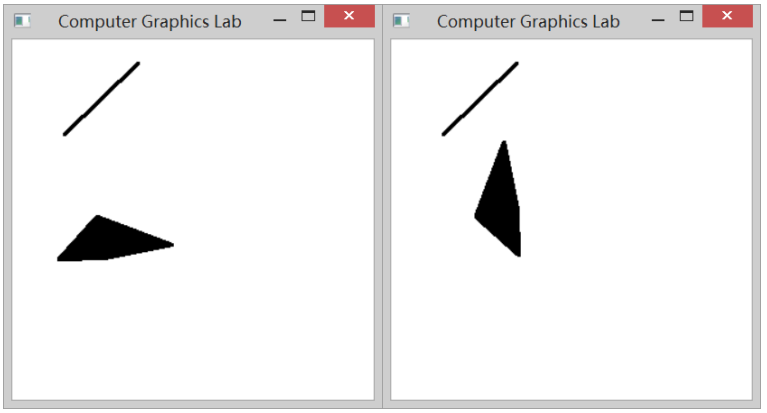


Fig. 22 旋转变换前后

4.7.3 缩放变换

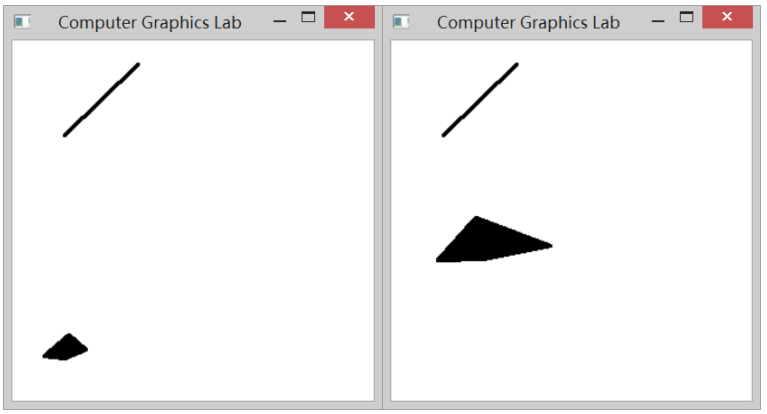


Fig. 23 缩放变换前后

4.8 绘制三维六面体的旋转

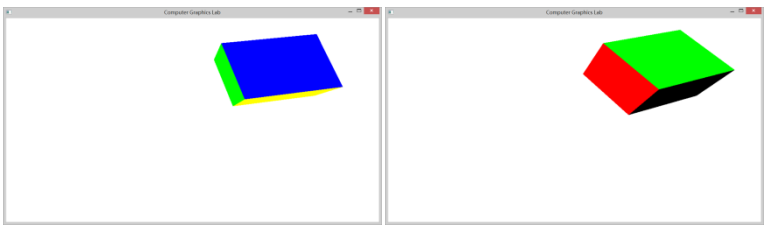


Fig. 24 三维六面体的旋转

4.9 图形的存储

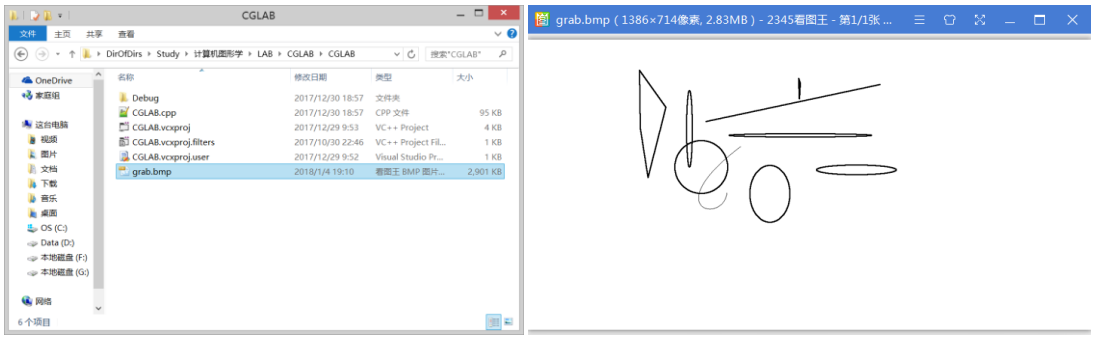


Fig. 25 图形的存储

References:

[1] 孙正兴.计算机图形学教程.北京:机械工业出版社,2006.8

[2] 改进 OpenGL 抓图功能
http://blog.csdn.net/dreamcs/article/details/6052984

[3] 旋转立方体(opengl)
https://www.cnblogs.com/djcsch2001/archive/2011/03/05/1971818.html