CGLAB 系统使用说明书

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 系统功能介绍

2.1 直线

2.1.1 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。

2.1.2 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。

2.2 曲线

2.2.1 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;
}
```

2.2.2 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。

2.3 多边形

2.3.1 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。

2.3.2　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。

2.4 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。

2.4.1 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。

2.4.2 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。

2.5 填充区域

2.5.1 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。

2.5.2 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。

2.6 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();
}
```

2.7 图形的变换

2.7.1 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')
{
```

```
    /* ... */
}
```

2.7.2 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')
{
    /* ... */
}
```

2.7.3 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）。

2.8 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;
}
```

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。

```
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);
}
```

3 系统操作说明及运行结果

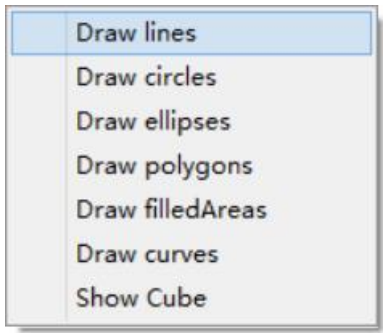
3.1 直线

3.1.1 绘制直线

此图形学系统可以绘制直线。

当进入此系统后，鼠标右键点击窗口界面中除标题栏区域的任一个位置，可以看到会弹出一个菜单，菜

单如下图：

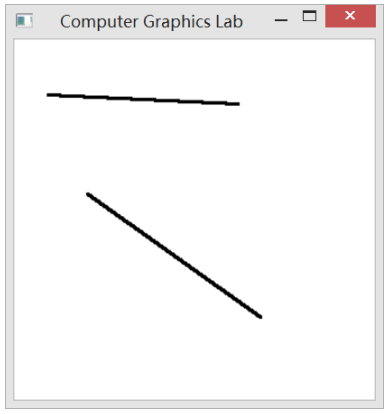


我们选择其中的一个菜单项 Draw lines。

此时，我们便进入了绘制直线的模式。

绘制直线的方式是这样的：鼠标左键点击窗口界面中除标题栏区域的任一个位置，并且不要松开，这时我们就选定了直线的一个端点。再强调一下，我们不要松开鼠标左键，然后移动鼠标，直到鼠标指针到达我们心仪的该直线另一个端点的位置，这时我们松开鼠标，然后再在界面空白的地方左键点击一下鼠标，一条直线就绘制完毕了，并且准备开始下一条直线的绘制。

可惜 PDF 不能插入动态图，只能贴一张静态图看看静态的效果：



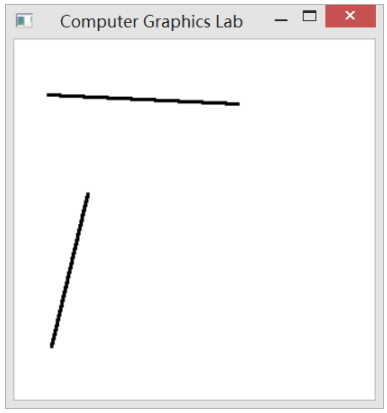
3.1.2 编辑直线

关于对直线的编辑，我简单借鉴了一下 Windows 操作系统里的画图软件，即只能对最后一次绘制的直线进行编辑。

编辑的方式，主要是可以改变直线两个端点的位置，操作方式对用户比较友好，只需用鼠标左键点击直线的某个端点，并且在鼠标左键不松开的情况移动鼠标，将该端点移动至我们心仪的其他地方。

并且此编辑功能可以连续多次使用，直至我们在界面空白的地方左键点击一下鼠标为止，并且准备开始下一条直线的绘制。

我们接下来看一下对刚才画的一条直线进行端点移动后的结果：



3.2 多边形

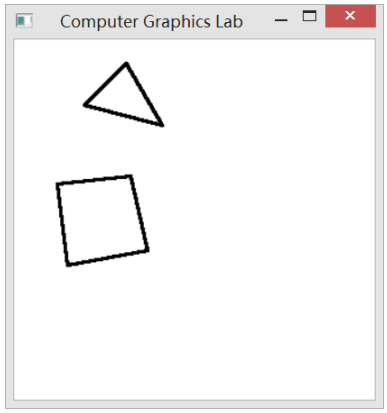
3.2.1 绘制多边形

和绘制直线一开始一样，鼠标右键弹出菜单，这时候我们选择菜单项 Draw polygons。

此时，我们便进入了绘制多边形的模式。

绘制多边形的方式是这样的：鼠标左键点击窗口界面中除标题栏区域的任一个位置，并且松开左键，这时我们就选定了多边形的一个端点。接下来我们发现随着鼠标指针的移动，从刚才选定的顶点到鼠标指针之间总是会产生一条直线，画面效果还是挺棒的。然后我们移动鼠标，直到鼠标指针到达多边形下一个端点的位置，这时我们点击鼠标左键，多边形的第 1 条边就绘制完毕了。我们按照以上的操作方式，绘制第 2 条边，绘制第 3 条边，……直到即将绘制最后 1 条边时，我们将到鼠标移动到最初开始的端点处，鼠标左键点击之。然后我们再在系统界面空白处左键点击一下鼠标，这样一个多边形就绘制完毕了，并且准备开始下一个多边形的绘制。

下面我们来看一下绘制效果：

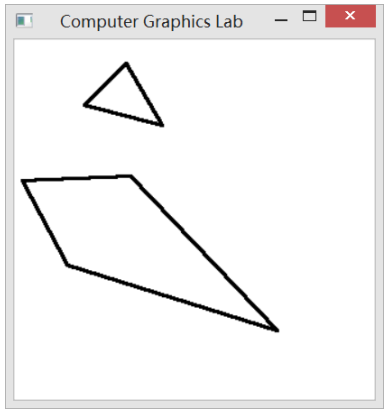


3.2.2 编辑多边形

编辑多边形的方式和编辑直线比较类似，主要是通过改变多边形某些顶点的位置来改变多边形的形状，操作方式对用户比较友好，只需用鼠标左键点击多边形的某个端点，并且在鼠标左键不松开的情况移动鼠标，将该端点移动至我们心仪的其他地方即可。

并且此编辑功能可以连续多次使用，直至我们在系统界面空白处左键点击一下鼠标为止，并且准备开始下一个多边形的绘制。

我们接下来看一下对刚才画的一个多边形进行端点移动后的结果：

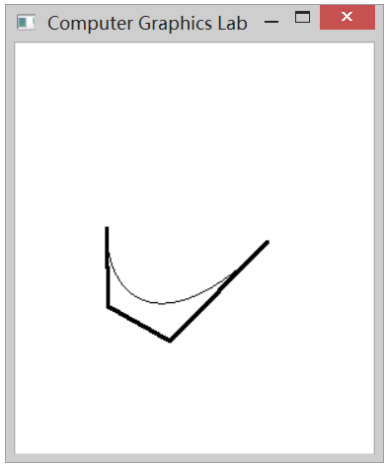


3.3 曲线

3.3.1 绘制曲线

首先在系统界面空白处右键点击鼠标，然后左键点击菜单中 Draw curves 的菜单项。

由于我们的曲线是利用三次 Bezier 曲线的函数表达式来实现绘制的，所以我们需要先画由三段连续的直线段构成的控制多边形，画的方式与画多边形类似，当我们画完第三段直线段时，曲线就出现了：

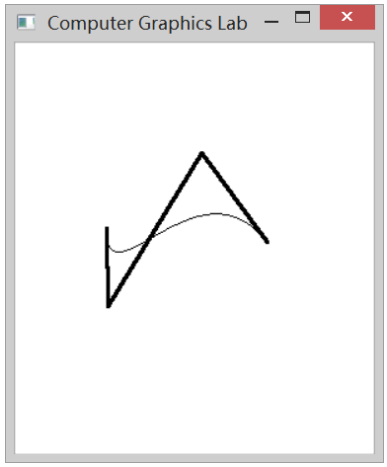


然后我们再在系统界面空白处左键单击一下鼠标，我们可以看到曲线的控制多边形消失，此时，此曲线绘制完毕，并且准备开始下一条曲线的绘制。

3.3.2 编辑曲线

编辑曲线的方式实质上就是编辑多边形的方式，就是左键点击曲线的控制多边形的顶点进行拖动，直到控制多边形的顶点到达心仪的位置时松开鼠标左键，此时曲线的这一次编辑就完毕了。我们可以多次对曲线进行编辑，直到我们在系统界面空白处左键单击一下鼠标为止，并且准备开始下一条曲线的绘制。

我们来看一下上面绘制的那条曲线经过编辑后的形状：



3.4 圆与椭圆

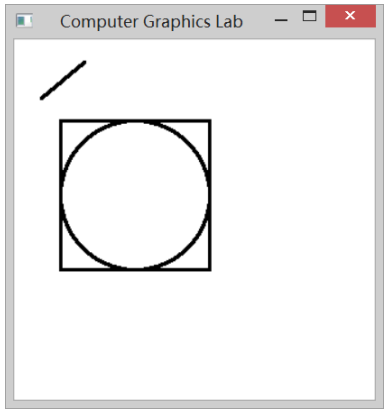
3.4.1 绘制圆

首先在系统界面空白处右键点击鼠标，然后左键点击菜单中 Draw circles 的菜单项。

此时，我们便进入了绘制圆的模式。

绘制圆的方式是这样的：鼠标左键点击窗口界面中除标题栏区域的任一个位置，并且不要松开，然后移动鼠标。这个时候我们可以看到随着鼠标的移动圆就画出来了，当我们觉得屏幕上画出的圆令自己比较满意的时候，就可以松开鼠标，这个时候带着外切正方形的圆就绘制出来了。最后我们在外切正方形外侧任意空白处点击一下鼠标左键，圆的外切正方形消失，此时我们的圆正式绘制完成，并且准备开始下一个圆的绘制。

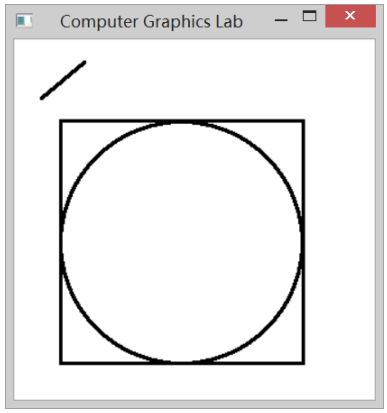
我们贴一张静态图看看效果：



3.4.2 编辑圆

对于圆的编辑，我们知道我们在画圆的过程中，当我们松开鼠标时，一个带着外切正方形的圆就绘制出来了。这个时候我们把鼠标指针移动到正方形的四个顶点处，左键点击并拖动顶点，我们可以发现圆的大小在鼠标拖动的同时发生了变化，当我们对自己编辑的结果比较满意时，便可松开鼠标。接下来，我们可以选择继续编辑，或者我们也可以选择在外切正方形外侧任意空白处点击一下鼠标左键，圆的外切正方形消失，此时我们对圆的编辑也完成了，并且准备开始下一个圆的绘制。

我们接下来看一下对刚才画的圆进行编辑后的结果：



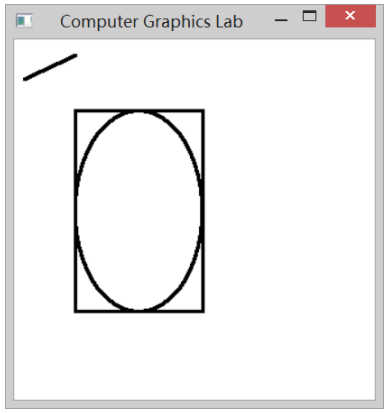
3.4.3 绘制椭圆

我们在菜单中选择菜单项 Draw Ellipses。

此时，我们便进入了绘制椭圆的模式。

绘制椭圆的方式是这样的：鼠标左键点击窗口界面中除标题栏区域的任一个位置，并且不要松开，然后移动鼠标。这个时候我们可以看到随着鼠标的移动椭圆就画出来了，当我们觉得屏幕上画出的椭圆令自己比较满意的时候，就可以松开鼠标，这个时候带着外切矩形的椭圆就绘制出来了。最后我们在外切矩形外侧任意空白处点击一下鼠标左键，椭圆的外切矩形消失，此时我们的椭圆正式绘制完成，并且准备开始下一个椭圆的绘制。

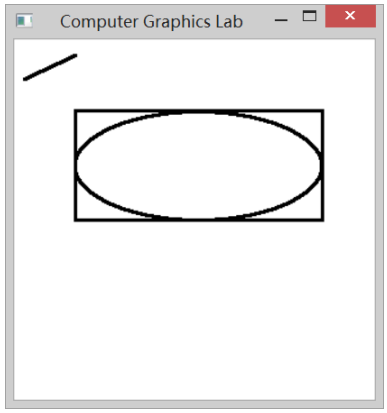
我们贴一张静态图看看效果：



3.4.4 编辑椭圆

对于椭圆的编辑，我们知道我们在画椭圆的过程中，当我们松开鼠标时，一个带着外切矩形的椭圆就绘制出来了。这个时候我们把鼠标指针移动到矩形的四个顶点处，左键点击并拖动顶点，我们可以发现椭圆的大小在鼠标拖动的同时发生了变化，当我们对自己编辑的结果比较满意时，便可松开鼠标。接下来，我们可以选择继续编辑，或者我们也可以选择在外切矩形外侧任意空白处点击一下鼠标左键，椭圆的外切矩形消失，此时我们对椭圆的编辑也完成了，并且准备开始下一个椭圆的绘制。

我们接下来看一下对刚才画的椭圆进行编辑后的结果：



3.5 填充区域

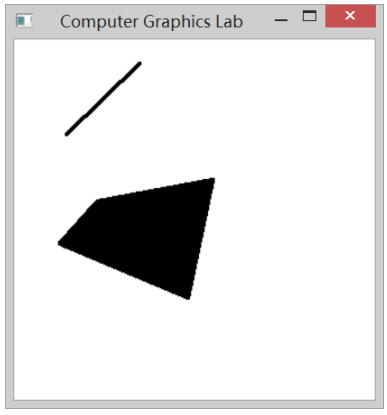
3.5.1 绘制填充区域

我们在菜单中选择菜单项 Draw filledAreas。

此时，我们便进入了绘制填充区域的模式。

对于用户来说，绘制填充区域的方式和绘制多边形的方式是一样的：鼠标左键点击窗口界面中除标题栏区域的任一个位置，并且松开左键，这时我们就选定了填充区域的一个端点。接下来我们发现随着鼠标指针的移动，从刚才选定的顶点到鼠标指针之间总是会产生一条直线，画面效果还是挺棒的。然后我们移动鼠标，直到鼠标指针到达填充区域下一个端点的位置，这时我们点击鼠标左键，多边形的第 1 条边就绘制完毕了。我们按照以上的操作方式，绘制第 2 条边，绘制第 3 条边，……直到即将绘制最后 1 条边时，我们将到鼠标移动到最初开始的端点处，鼠标左键点击之。然后我们再在系统界面空白处左键点击一下鼠标，这样一个填充区域就绘制完毕了，并且准备开始下一个填充区域的绘制。

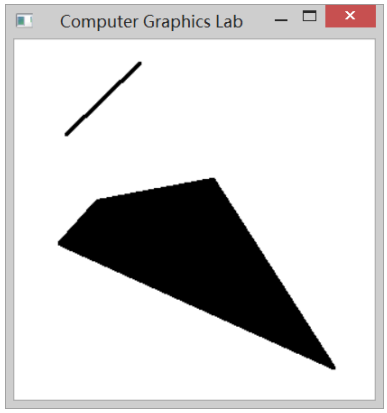
下面我们来看一下绘制效果：



3.5.2 编辑填充区域

对于用户来说，编辑填充区域的方式和编辑多边形的方式是一样的：主要是通过改变填充区域某些顶点的位置来改变填充区域的形状，操作方式对用户比较友好，只需用鼠标左键点击填充区域的某个端点，并且在鼠标左键不松开的情况移动鼠标，将该端点移动至我们心仪的其他地方即可。并且此编辑功能可以连续多次使用，直至我们在系统界面空白处左键点击一下鼠标为止，并且准备开始下一个填充区域的绘制。

我们接下来看一下对刚才画的一个填充区域进行端点移动后的结果：



3.6 多边形裁剪

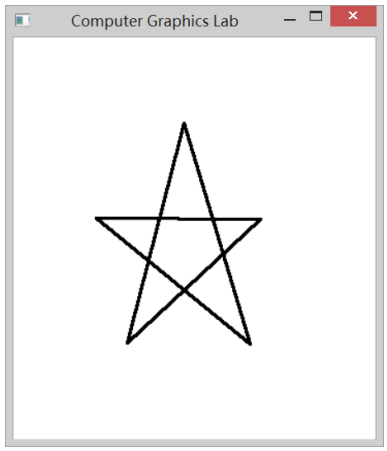
裁剪这个功能我们只针对多边形，并且是当前正处于编辑状态下的多边形。

当我们的多边形处于待编辑状态下（可以简单地理解为刚绘制出来）时，我们按下键盘上的 c 键，这时我们就进入了裁剪多边形的状态，我们可以利用鼠标在系统界面上绘制一个矩形的裁剪窗口。绘制矩形裁剪窗口方式是这样的，我们用鼠标左键单击系统绘图界面，然后不要松开鼠标左键，我们在鼠标左键不松开的状态下拖动鼠标，这时我们会发现一个裁剪窗口在屏幕上就出现了，这个时候当我们松开鼠标左键时，裁剪窗口就不动了。这个时候我们有两个选择：

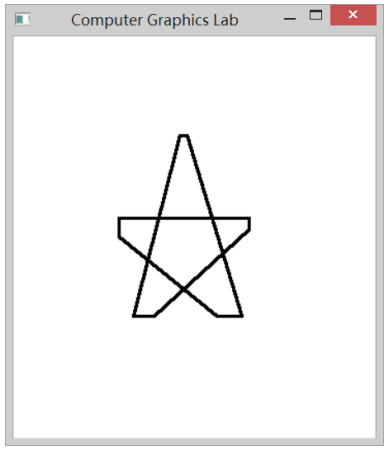
①继续编辑裁剪窗口。我们可以用鼠标左键去拖动裁剪窗口的四个顶点直至我们认为裁剪窗口的大小和位置令自己比较满意，并且编辑裁剪窗口的次数也是没有限制的；

②不再编辑裁剪窗口，再一次按下键盘上的 c 键对当前正处于编辑状态的多边形进行裁剪，裁剪完毕后我们会发现当前正处于编辑状态的多边形处于裁剪窗口内部的部分留了下来，而在外部的部分就消失了。

这是我们裁剪前截图：



这是我们裁剪后截图：



并且我们需要了解的是，当裁剪操作刚刚完成时，多边形依旧进入了编辑状态，所以我们还是可以继续裁剪的，不过我们也可以选择在系统界面空白处单击鼠标左键，表示确认多边形绘制、编辑结束，并且准备开始下一个多边形的绘制。

3.7 图形的变换

我们拿填充图形来说明图形变换的功能。

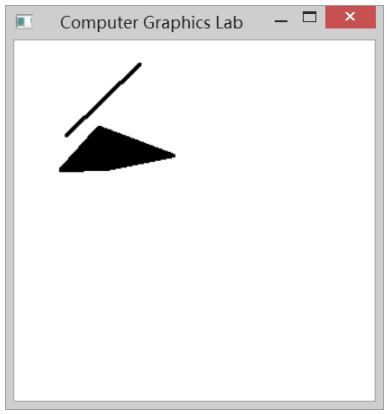
3.7.1 平移变换

首先必须强调的一点是，平移功能针对的是当前刚画好的图形，即正处于编辑状态的图形。

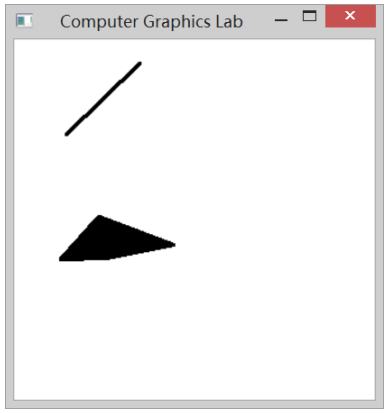
对于用户来说，平移功能的操作方式比较简易。键盘上 w 键表示将图形向上移动，s 键表示将图形向下移动，a 键表示将图形向左移动，d 键表示将图形向右移动。

我们接下来看一下平移的效果。

平移填充图形前：



平移填充图形后：



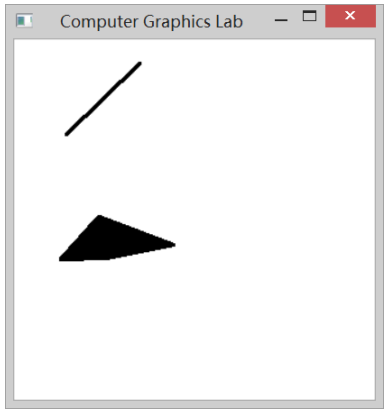
3.7.2 旋转变换

首先必须强调的一点是，旋转功能针对的是当前刚画好的图形，即正处于编辑状态的图形。

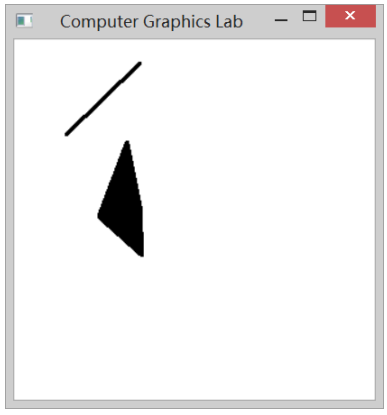
对于用户来说，旋转功能的操作方式比较简易。键盘上 q 键表示将图形逆时针旋转 90 度，e 键表示将图形顺时旋转 90 度。其中圆和椭圆是绕着圆心和椭圆心旋转的，直线、多边形、填充区域是绕着第一个确定的顶点旋转的。

我们接下来看一下旋转的效果。

旋转填充图形前：



旋转填充图形后：



3.7.3 缩放变换

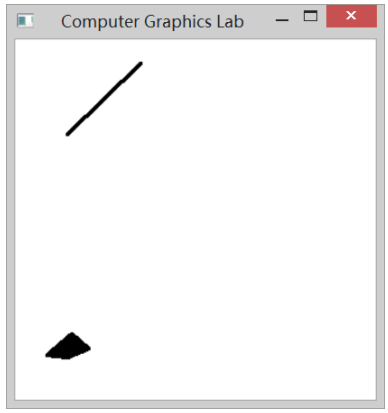
首先必须强调的一点是，缩放功能针对的是当前刚画好的图形，即正处于编辑状态的图形。

对于用户来说，缩放功能的操作方式比较简易。键盘上 x 键表示将图形放大，z 键表示将图形缩小。

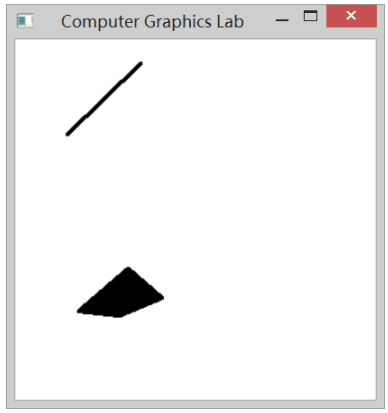
其中，直线、多边形和填充图形是基于绘图窗口左下角进行缩放的，圆和椭圆是基于圆心和椭圆心进行缩放，并且我们规定，直线、多边形、填充图形、圆和椭圆进行缩放的操作过程中，图形的大小不能小于最初时的大小（即要想实现缩小，必须首先放大，规定这一点的目的是防止图形在一开始直接缩小后再放大会造成图形相关长度信息与最初相比发生改变，这是由数据类型为整型导致的），另外圆和椭圆最多只能放大一次（防止在过度放大过程中，图形比例会发生变化）。

我们接下来看一下缩放的效果。

放大填充图形前：

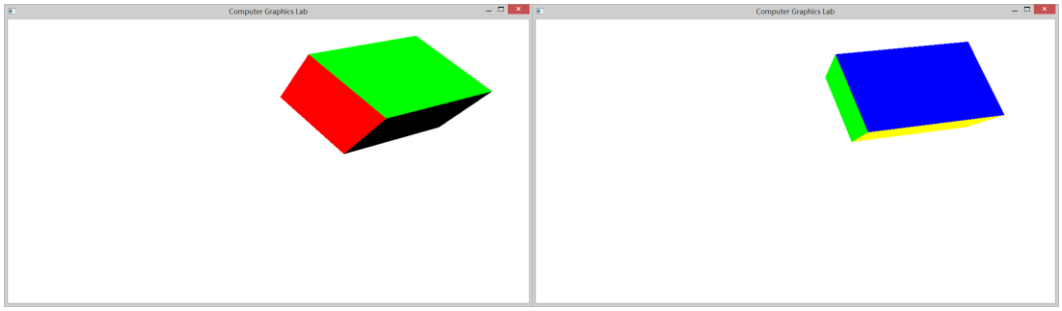


放大填充图形后：



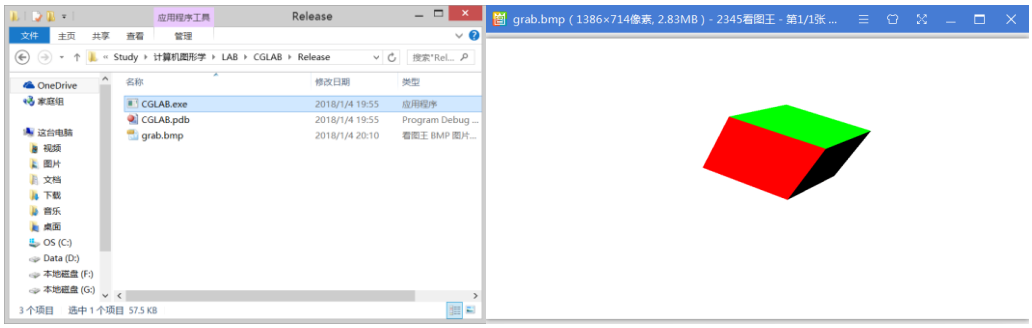
3.8 绘制3D立方体的旋转

显示旋转的 3D 立方体的操作很简单，只需在系统界面空白处右键点击鼠标，然后左键点击菜单中 Show Cube 的菜单项即可。



3.9 图形的存储

图形的存储的操作方式，即当我们关闭此图形系统时，系统会自动保存关闭那一瞬间界面显示的内容。保存的图片格式为 bmp 格式，保存的路径为此应用程序同目录下。



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