实验二 多边形世界

一．实验目的

二．实验环境

三．实验内容和要求

四．实验原理

1. OpenGL 中的三维物体的显示
2. OpenGL 中的几种变换

五．实验指导

1. 坐标变换
2. 键盘、鼠标事件
3. 拾取

一．实验目的

1．掌握可交互的 OpenGL 应用程序的开发设计的方法。
2．掌握系统处理鼠标和键盘事件的编程方法。
3. 掌握 OpenGL 中的拾取机制。

二．实验环境

硬件环境：P4 CPU 2.0以上PC机，512M以上内存。

软件环境：Windows XP，Visual C++ 6.0，OpenGL 图形软件包，GLUT 开发包。

三．实验内容和要求：

编写一个可以交互的 OpenGL 应用程序，支持用户利用鼠标创建和删除二维多边形对象。基本功能包括：

对象创建：支持用户利用鼠标指定各个顶点位置，创建多边形。

对象删除：支持用户选择一个多边形（与你的多边形保存的数据结构有关）并删除。

对象存储：设计一种数据结构存储每个多边形的顶点与边，支持文件存盘，。

提示：由于删除操作需要多边形的选择机制，你可以在设计数据结构时，把可以帮助你完成选择的信息一起保存。

扩展功能可以包括（至少选择其一）：

1. 支持用户选择多边形的颜色
2. 支持用户移动多边形
3. 支持用户改变多边形单个顶点的位置
4. 支持 3D 多边形
5. 其他（需要通过指导老师认可）

完成一份实验报告，说明你的对象存储与对象选择方法，或者你所实现的一个扩展功能。

四．实验原理

OpenGL 通过相机模拟、可以实现计算机图形学中最基本的三维变换，即几何变换、投影变换、裁剪变换、视口变换等，同时，OpenGL 还实现了矩阵堆栈等。理解掌握了有关坐标变换的内容，就算真正走进了精彩地三维世界。

（一）OpenGL 中的三维物体的显示

（Ⅰ）坐标系统

在现实世界中，所有的物体都具有三维特征，但计算机本身只能处理数字，显示二维的图形，将三维物体及二维数据联系在一起的唯一纽带就是坐标。

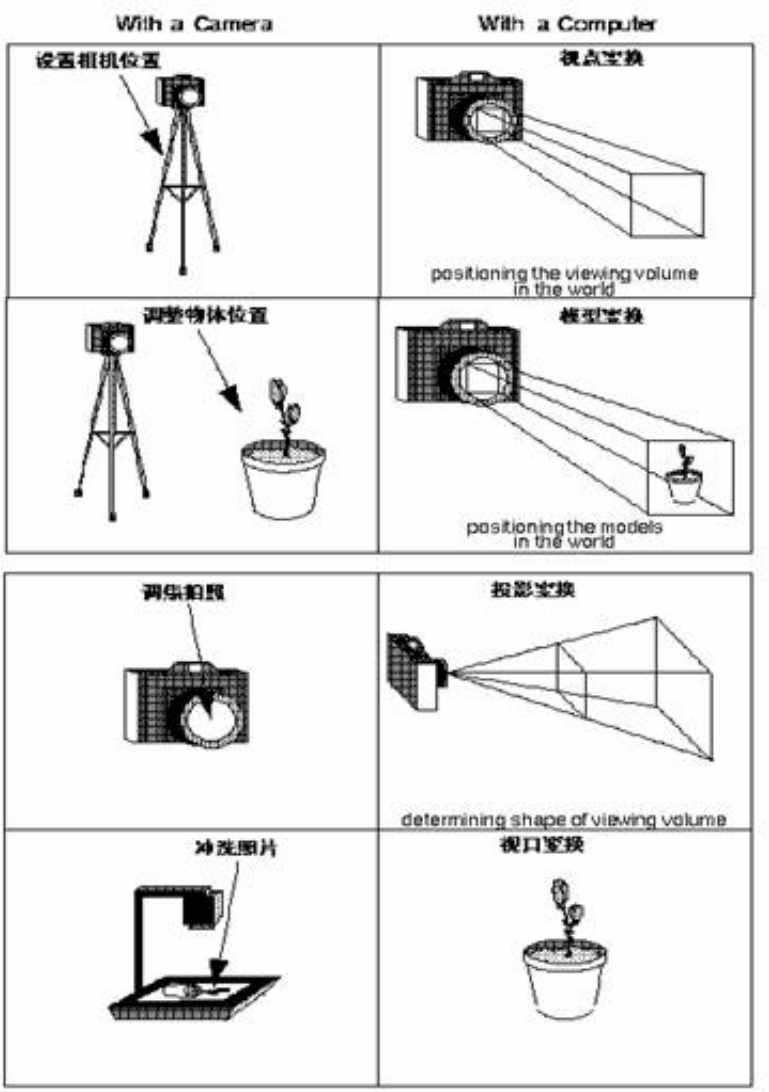
为了使被显示的三维物体数字化，要在被显示的物体所在的空间中定义一个坐标系。这个坐标系的长度单位和坐标轴的方向要适合对被显示物体的描述，这个坐标系称为世界坐标系。世界坐标系是始终固定不变的。

OpenGL 还定义了局部坐标系的概念，所谓局部坐标系，也就是坐标系以物体的中心为坐标原点，物体的旋转或平移等操作都是围绕局部坐标系进行的，这时，当物体模型进行旋转或平移等操作时，局部坐标系也执行相应的旋转或平移操作。需要注意的是，如果对物体模型进行缩放操作，则局部坐标系也要进行相应的缩放，如果缩放比例在案各坐标轴上不同，那么再经过旋转操作后，局部坐标轴之间可能不再相互垂直。无论是在世界坐标系中进行转换还是在局部坐标系中进行转换，程序代码是相同的，只是不同的坐标系考虑的转换方式不同罢了。

计算机对数字化的显示物体作了加工处理后，要在图形显示器上显示，这就要在图形显示器屏幕上定义一个二维直角坐标系，这个坐标系称为屏幕坐标系。这个坐标系坐标轴的方向通常取成平行于屏幕的边缘，坐标原点取在左下角，长度单位常取成一个象素。

（Ⅱ）三维物体的相机模拟

为了说明在三维物体到二维图象之间，需要经过什么样的变换，我们引入了相机（Camera）模拟的方式，假定用相机来拍摄这个世界，那么在相机的取景器中，就存在人眼和现实世界之间的一个变换过程。



从三维物体到二维图象，就如同用相机拍照一样，通常都要经历以下几个步骤：

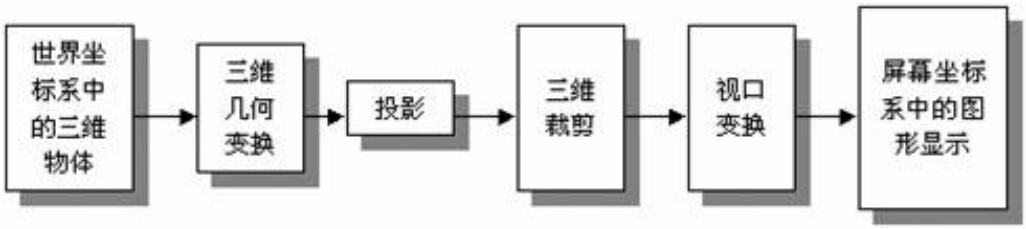
1、将相机置于三角架上，让它对准三维景物，它相当于 OpenGL 中调整视点的位置，即视点变换（Viewing Transformation）。

2、将三维物体放在场景中的适当位置，它相当于 OpenGL 中的模型变换（Modeling Transformation），即对模型进行旋转、平移和缩放。

3、选择相机镜头并调焦，使三维物体投影在二维胶片上，它相当于 OpenGL 中把三维模型投影到二维屏幕上的过程，即 OpenGL 的投影变换 （Projection Transformation），OpenGL 中投影的方法有两种，即正射投影和透视投影。为了使显示的物体能以合适的位置、大小和方向显示出来，必须要通过投影。有时为了突出图形的一部分，只把图形的某一部分显示出来，这时可以定义一个三维视景体（Viewing Volume）。正射投影时一般是一个长方体的视景体，透视投影时一般是一个棱台似的视景体。只有视景体内的物体能被投影在显示平面上，其他部分则不能。

4、冲洗底片，决定二维相片的大小，它相当与 OpenGL 中的视口变换（Viewport Transformation）（在屏幕窗口内可以定义一个矩形，称为视口（Viewport），视景体投影后的图形就在视口内显示）规定屏幕上显示场景的范围和尺寸。

通过上面的几个步骤，一个三维空间里的物体就可以用相应的二维平面物体表示了，也就能在二维的电脑屏幕上正确显示了。总的来说，三维物体的显示过程如下：



（二）OpenGL 中的几种变换

OpenGL 中的各种转换是通过矩阵运算实现的，具体的说，就是当发出一个转换命令时，该命令会生成一个 4X4 阶的转换矩阵（OpenGL 中的物体坐标一律采用齐次坐标，即(x, y, z, w)，故所有变换矩阵都采用 4X4 矩阵），当前矩阵与这个转换矩阵相乘，从而生成新的当前矩阵。例如，对于顶点坐标 v ，转换命令通常在顶点坐标命令之前发出，若当前矩阵为 C，转换命令构成的矩阵为 M，则发出转换命令后，生成的新的当前矩阵为 CM，这个矩阵再乘以顶点坐标 v，从而构成新的顶点坐标 CMv。上述过程说明，程序中绘制顶点前的最后一个变换命令最先作用于顶点之上。这同时也说明，OpenGL 编程中，实际的变换顺序与指定的顺序是相反的。

（Ⅰ）视点变换

视点变换确定了场景中物体的视点位置和方向，就向上边提到的，它象是在场景中放置了一架照相机，让相机对准要拍摄的物体。缺省时，相机（即视点）定位在坐标系的原点（相机初始方向都指向 Z 负轴），它同物体模型的缺省位置是一致的，显然，如果不进行视点变换，相机和物体是重叠在一起的。

执行视点变换的命令和执行模型转换的命令是相同的，想一想，在用相机拍摄物体时，我们可以保持物体的位置不动，而将相机移离物体，这就相当于视点变换；另外，我们也可以保持相机的固定位置，将物体移离相机，这就相当于模型转换。这样，在 OpenGL 中，以逆时针旋转物体就相当于以顺时针旋转相机。因此，我们必须把视点转换和模型转换结合在一起考虑，而对这两种转换单独进行考虑是毫无意义的。

除了用模型转换命令执行视点转换之外，OpenGL 实用库还提供了 gluLookAt()函数，该函数有三个变量，分别定义了视点的位置、相机瞄准方向的参考点以及相机的向上方向。该

函数的原型为：

void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery, GLdouble upx, GLdouble upy, GLdouble upz);

该函数定义了视点矩阵，并用该矩阵乘以当前矩阵。eyex, eyey, eyez 定义了视点的位置；centerx, centery, centerz 变量指定了参考点的位置，该点通常为相机所瞄准的场景中心轴线上的点；upx, upy, upz 变量指定了向上向量的方向。

通常，视点转换操作在模型转换操作之前发出，以便模型转换先对物体发生作用。场景中物体的顶点经过模型转换之后移动到所希望的位置，然后再对场景进行视点定位等操作。模型转换和视点转换共同构成模型视景矩阵。

（Ⅱ）模型变换

模型变换是在世界坐标系中进行的。缺省时，物体模型的中心定位在坐标系的中心处。OpenGL 在这个坐标系中，有三个命令，可以模型变换。

1、模型平移

glTranslate{fd}(TYPE x,TYPE y,TYPE z);

该函数用指定的 x, y, z 值沿着 x 轴、y 轴、z 轴平移物体（或按照相同的量值移动局部坐标系）。

2、模型旋转

glRotate{fd}(TYPE angle, TYPE x, TYPE y, TYPE z);

该函数中第一个变量 angle 制定模型旋转的角度，单位为度，后三个变量表示以原点（0,0,0）到点(x,y,z)的连线为轴线逆时针旋转物体。例如，glRotatef(45.0, 0.0, 0.0, 1.0)的结果是绕 z 轴旋转 45 度。

3、模型缩放

glScale{fd}(TYPE x, TYPE y, TYPE z);

该函数可以对物体沿着 x,y,z 轴分别进行放大缩小。函数中的三个参数分别是 x、y、z 轴方向的比例变换因子。缺省时都为 1.0，即物体没变化。程序中物体 Y 轴比例为 2.0，其余都为 1.0，就是说将立方体变成长方体。

（Ⅲ）投影变换

经过模型视景的转换后，场景中的物体放在了所希望的位置上，但由于显示器只能用二维图象显示三维物体，因此就要靠投影来降低维数（投影变换类似于选择相机的镜头）。

事实上，投影变换的目的就是定义一个视景体，使得视景体外多余的部分裁剪掉，最终

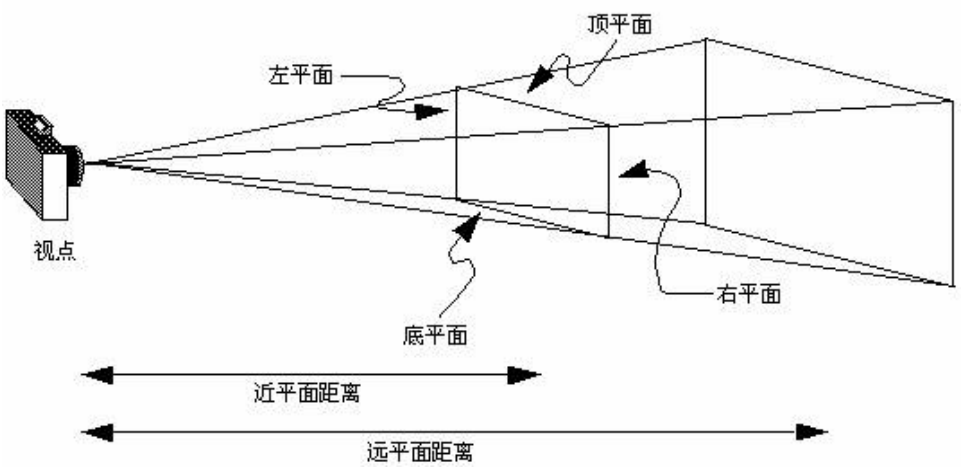
进入图像的只是视景体内的有关部分。投影包括透视投影（Perspective Projection）和正视投影（Orthographic Projection）两种。

透视投影，符合人们心理习惯，即离视点近的物体大，离视点远的物体小，远到极点即为消失，成为灭点。它的视景体类似于一个顶部和底部都被进行切割过的棱椎，也就是棱台。这个投影通常用于动画、视觉仿真以及其它许多具有真实性反映的方面。

OpenGL 透视投影函数有两个，其中函数 glFrustum()的原型为：

void glFrustum(GLdouble left,GLdouble Right,GLdouble bottom,GLdouble top,GLdouble near,GLdouble far);

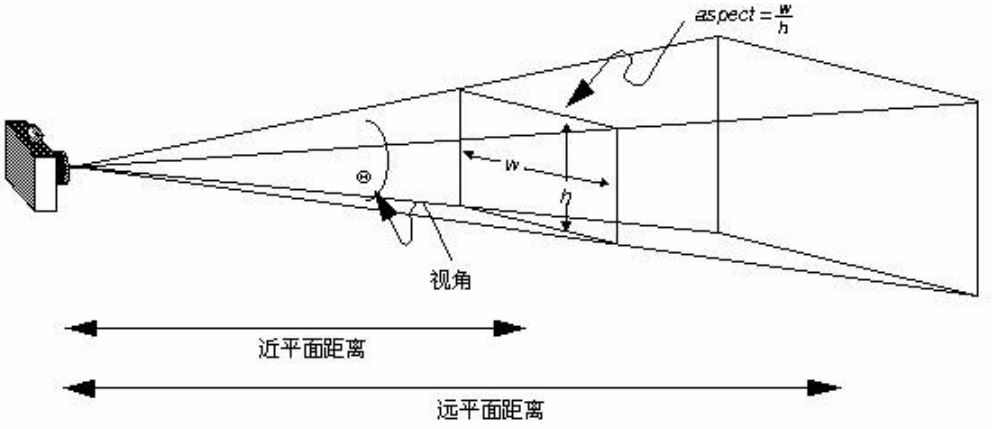
它创建一个透视视景体。其操作是创建一个透视投影矩阵，并且用这个矩阵乘以当前矩阵。这个函数的参数只定义近裁剪平面的左下角点和右上角点的三维空间坐标，即（left, bottom, -near）和（right, top, near）；最后一个参数 far 是远裁剪平面的 Z 负值，其左下角点和右上角点空间坐标由函数根据透视投影原理自动生成。near 和 far 表示离视点的远近，它们总为正值。该函数形成的视景体如图所示。



另一个透视函数是：

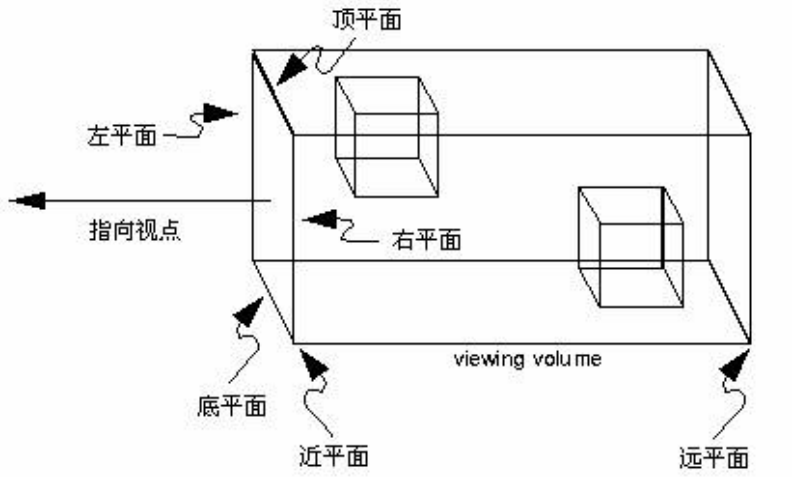
void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar);

它也创建一个对称透视视景体，但它的参数定义于前面的不同，参数 fovy 定义视野在 X-Z 平面的角度，范围是[0.0, 180.0]；参数 aspect 是投影平面宽度与高度的比率；参数 zNear 和 Far 分别是远近裁剪面沿 Z 负轴到视点的距离，它们总为正值。



以上两个函数缺省时，视点都在原点，视线沿 Z 轴指向负方向。

正射投影，又叫平行投影。这种投影的视景体是一个矩形的平行管道，也就是一个长方体，如图所示。正射投影的最大一个特点是无论物体距离相机多远，投影后的物体大小尺寸不变。这种投影通常用在建筑蓝图绘制和计算机辅助设计等方面，这些行业要求投影后的物体尺寸及相互间的角度不变，以便施工或制造时物体比例大小正确。



OpenGL 正射投影函数也有两个，一个函数是：

void glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far)

它创建一个平行视景体。实际上这个函数的操作是创建一个正射投影矩阵，并且用这个矩阵乘以当前矩阵。其中近裁剪平面是一个矩形，矩形左下角点三维空间坐标是（left, bottom, -near），右上角点是（right, top, -near）；远裁剪平面也是一个矩形，左下角点空间坐标是（left, bottom, -far），右上角点是（right, top, -far）。所有的 near 和 far 值同时为正或同时为负。如果没有其他变换，正射投影的方向平行于 Z 轴，且视点朝向 Z 负轴。这意味着物体在视点前面时 far 和 near 都为负值，物体在视点后面时 far 和 near 都为正值。

另一个函数是：

void gluOrtho2D(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top)

它是一个特殊的正射投影函数，主要用于二维图像到二维屏幕上的投影。它的 near 和 far 缺省值分别为-1.0 和 1.0，所有二维物体的 Z 坐标都为 0.0。因此它的裁剪面是一个左下角点为（left, bottom）、右上角点为（right, top）的矩形。

（Ⅳ）视口变换。

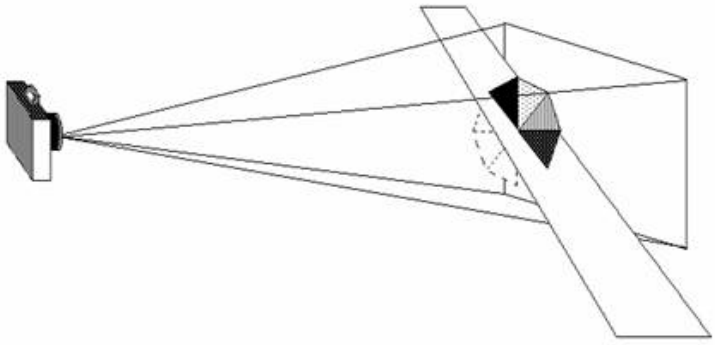
视口变换就是将视景体内投影的物体显示在二维的视口平面上。运用相机模拟方式，我们很容易理解视口变换就是类似于照片的放大与缩小。在计算机图形学中，它的定义是将经过几何变换、投影变换和裁剪变换后的物体显示于屏幕窗口内指定的区域内，这个区域通常为矩形，称为视口。OpenGL 中相关函数是：

glViewport(GLint x, GLint y, GLsizei width, GLsizei height);

这个函数定义一个视口。函数参数(x, y)是视口在屏幕窗口坐标系中的左下角点坐标，参数 width 和 height 分别是视口的宽度和高度。缺省时，参数值即(0, 0, winWidth, winHeight)指的是屏幕窗口的实际尺寸大小。所有这些值都是以象素为单位，全为整型数。

（Ⅴ）裁剪变换

在 OpenGL 中，除了视景体定义的六个裁剪平面（上、下、左、右、前、后）外，用户还可自己再定义一个或多个附加裁剪平面，以去掉场景中无关的目标，如图所示。



附加平面裁剪函数为：

void glClipPlane(GLenum plane, Const GLdouble *equation);

函数参数 equation 指向一个拥有四个系数值的数组，这四个系数分别是裁剪平面 Ax+By+Cz+D=0 的 A、B、C、D 值。因此，由这四个系数就能确定一个裁剪平面。参数 plane 是 GL_CLIP_PLANEi(i=0,1,...)，指定裁剪面号。

在调用附加裁剪函数之前，必须先启动 glEnable(GL_CLIP_PLANEi)，使得当前所定义的裁剪平面有效；当不再调用某个附加裁剪平面时，可用 glDisable(GL_CLIP_PLANEi)关闭相

应的附加裁剪功能。

（Ⅵ）矩阵栈的操作

在讲述矩阵栈之前，首先介绍两个基本 OpenGL 矩阵操作函数：

1、void glLoadMatrix{fd}(const TYPE *m)

设置当前矩阵中的元素值。函数参数*m 是一个指向 16 个元素(m0, m1, ..., m15)的指针，这 16 个元素就是当前矩阵 M 中的元素，其排列方式如下：

M=| m0 m4 m8 m12 |
| m1 m5 m9 m13 |
| m2 m6 m10 m14 |
| m3 m7 m11 m15 |

2、void glMultMatrix{fd}(const TYPE *m)

用当前矩阵去乘*m 所指定的矩阵，并将结果存放于*m 中。当前矩阵可以是用 glLoadMatrix() 指定的矩阵，也可以是其它矩阵变换函数的综合结果。

OpenGL 的矩阵堆栈指的就是内存中专门用来存放矩阵数据的某块特殊区域。一般说来，矩阵堆栈常用于构造具有继承性的模型，即由一些简单目标构成的复杂模型。矩阵堆栈对复杂模型运动过程中的多个变换操作之间的联系与独立十分有利。因为所有矩阵操作函数如 glLoadMatrix()、 glMultMatrix()、glLoadIdentity()等只处理当前矩阵或堆栈顶部矩阵，这样堆栈中下面的其它矩阵就不受影响。堆栈操作函数有以下两个：

• void glPushMatrix(void);

该函数表示将所有矩阵依次压入堆栈中，顶部矩阵是第二个矩阵的备份；压入的矩阵数不能太多，否则出错。

• void glPopMatrix(void);

该函数表示弹出堆栈顶部的矩阵，令原第二个矩阵成为顶部矩阵，接受当前操作，故原顶部矩阵被破坏；当堆栈中仅存一个矩阵时，不能进行弹出操作，否则出错。

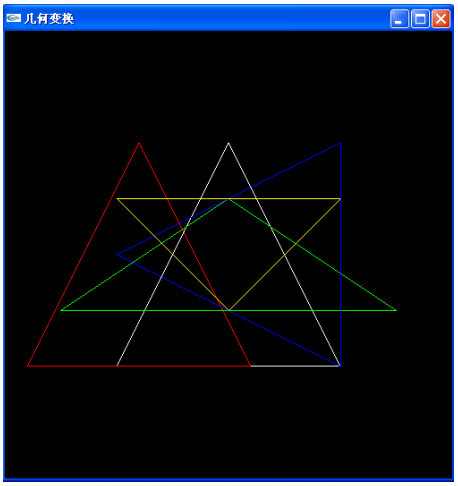
五．实验指导：

一．坐标变换

1、几何变换：

例如：我们要绘制一系列的三角形。先绘制一个白色三角形，后面要绘制的三角形都要经过一定的几何变换。第二个红色三角形是白三角形沿 X 负轴平移后的三角形，第三个绿色三角形是白三角形分别沿 X 轴和 Y 轴比例变换后的三角形，第四个蓝色三角形是白三角形绕 Z 正轴逆时针转 90 度后

的三角形，第五个黄色三角形是白三角形沿 Y 轴方向缩小一倍且相对于 X 轴作反射后形成的三角形。



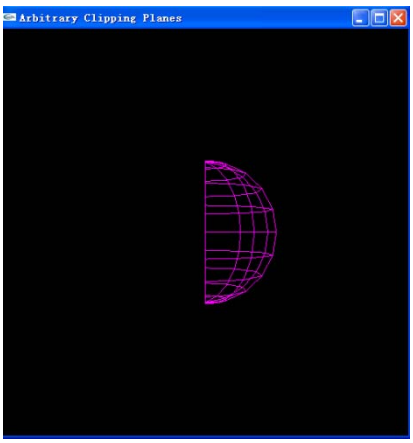
程序如下：

```
#include<gl/glut.h>
void draw_triangle(void)
{
	glBegin(GL_LINE_LOOP);
	glVertex2f(0.0, 25.0);
	glVertex2f(25.0, -25.0);
	glVertex2f(-25.0, -25.0);
	glEnd();
}
void display(void)
{
	glClearColor (0.0, 0.0, 0.0, 1.0);
	glClear (GL_COLOR_BUFFER_BIT);
	glLoadIdentity ();
	glColor3f (1.0, 1.0, 1.0);   /* 白色 */
	draw_triangle ();
	glLoadIdentity ();
	glTranslatef (-20.0, 0.0, 0.0);
	glColor3f(1.0,0.0,0.0);          /* 红色 */
	draw_triangle ();
	glLoadIdentity();
	glScalef (1.5, 0.5, 1.0);
	glColor3f(0.0,1.0,0.0);          /* 绿色 */
	draw_triangle ();
	glLoadIdentity ();
	glRotatef (90.0, 0.0, 0.0, 1.0);
	glColor3f(0.0,0.0,1.0);          /* 蓝色 */
	draw_triangle ();
	glLoadIdentity();
```

```
    glScalef (1.0, -0.5, 1.0);
    glColor3f(1.0,1.0,0.0);          /* 黄色 */
    draw_triangle ();
    glFlush();
}
void myinit (void)
{
    glShadeModel (GL_FLAT);
}
void myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho(-50.0, 50.0, -50.0*(GLfloat)h/(GLfloat)w, 50.0*(GLfloat)h/(GLfloat)w,-1.0,1.0);
    else
        glOrtho(-50.0*(GLfloat)w/(GLfloat)h, 50.0*(GLfloat)w/(GLfloat)h, -50.0, 50.0,-1.0,1.0);
    glMatrixMode(GL_MODELVIEW);
}
void main(void)
{
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
    glutInitWindowPosition(0,0);
    glutInitWindowSize(500,500);
    glutCreateWindow ("几何变换");
    myinit ();
    glutReshapeFunc (myReshape);
    glutDisplayFunc(display);
    glutMainLoop();
}
```

2、裁剪变换

下面这个例子不仅说明了附加裁剪函数的用法，而且调用了 gluPerspective()透视投影函数，大家可以细细体会其中的用法。



程序如下：

```
#include <gl/glut.h>
void display(void)
{
    GLdouble eqn[4] = {1.0, 0.0, 0.0, 0.0};
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 0.0, 1.0);
    glPushMatrix();
    glTranslatef (0.0, 0.0, -5.0);
    /* clip the left part of wire_sphere : x<0 */
    glClipPlane (GL_CLIP_PLANE0, eqn);
    glEnable (GL_CLIP_PLANE0);
    glRotatef (-90.0, 1.0, 0.0, 0.0);
    glutWireSphere(1.0,10,10);
    glPopMatrix();
    glFlush();
}
void myinit (void)
{
    glShadeModel (GL_FLAT);
}
void myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
    glMatrixMode(GL_MODELVIEW);
}
void main(int argc, char ** argv)
{
    glutInit(&argc,argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
```

```
    glutInitWindowPosition(0,0);
    glutInitWindowSize(500, 500);
    glutCreateWindow ("Arbitrary Clipping Planes");
    myinit ();
    glutReshapeFunc(myReshape);
    glutDisplayFunc(display);
    glutMainLoop();
}
```

二．键盘、鼠标事件

（一）键盘输入

GLUT 允许我们编写程序，在里面加入键盘输入控制，包括了普通键，和其他特殊键（如F1,UP）。在这一节里我们将学习如何去检测哪个键被按下，可以从 GLUT 里得到些什么信息，和如何处理键盘输入。

到现在，你应该注意到了，只要你想控制一个事件的处理，你就必须提前告诉 GLUT，哪个函数将完成这个任务。到现在为止，我们已经使用 GLUT 告诉窗口系统，当窗口重绘时我们想调用哪个渲染函数；当系统空闲时，哪个函数被调用；当窗口大小改变时，哪个函数又将被调用。

相似的，我们必须做同样的事来处理按键消息。我们必须使用 GLUT 通知窗口系统，当某个键被按下时，哪个函数将完成所要求的操作。我们同样是调用一个函数注册相关的回调函数。

当按下一个键后，GLUT 提供了两个函数为这个键盘消息注册回调。第一个是glutKeyboardFunc。这个函数是告诉窗口系统，哪一个函数将会被调用来处理普通按键消息。

普通键是指字母，数字，和其他可以用 ASCII 代码表示的键。

函数原型如下：

```
void glutKeyboardFunc(void(*func)(unsigned char key,int x,int y));
```

指定当任何键被按下时，函数 func()将被调用。所按下的键连同光标的位置一并传入func()。注意，光标位置的单位为像素，是从窗口的左上角开始度量的。

参数：

func: 处理普通按键消息的函数的名称。如果传递 NULL，则表示 GLUT 忽略普通按键消息。

这个作为 glutKeyboardFunc 函数参数的函数需要有三个形参。第一个表示按下的键的ASCII 码，其余两个提供了，当键按下时当前的鼠标位置。鼠标位置是相对于当前客户窗口的左上角而言的。

一个经常的用法是当按下 ESCAPE 键时退出应用程序。注意，glutMainLoop 函数产生的是一个永无止境的循环。唯一的跳出循环的方法就是调用系统 exit 函数。这就是我们函数要做的，当按下 ESCAPE 键调用 exit 函数终止应用程序（同时要记住在源代码包含头文件 stdlib.h）。下面就是这个函数的代码：

```
void processNormalKeys(unsigned char key,int x,int y)
{
        if(key==27)
                Exit(0);
}
```

下面让我们控制特殊键的按键消息。GLUT 提供函数 glutSpecialFunc 以便当有特殊键按下的消息时，你能注册你的函数。

函数原型如下：

void glutSpecialFunc(void (*func)(int key,int x,int y));

参数：

func: 处理特殊键按下消息的函数的名称。传递 NULL 则表示 GLUT 忽略特殊键消息。

下面我们编写一个函数，当一些特殊键按下的时候，改变我们的三角形的颜色。这个函数使在按下 F1 键时三角形为红色，按下 F2 键时为绿色，按下 F3 键时为蓝色。

```
void processSpecialKeys(int key, int x, int y)
{   switch(key)
{     case GLUT_KEY_F1 :
            red = 1.0;
            green = 0.0;
            blue = 0.0;
break;
            case GLUT_KEY_F2 :
                  red = 0.0;
                  green = 1.0;
                  blue = 0.0;
 break;
          case GLUT_KEY_F3 :
                red = 0.0;
                green = 0.0;
                blue = 1.0;
break;
}
}
```

上面的 GLUT_KEY_*在 glut.h 里已经被预定义为常量。这组常量如下：

GLUT_KEY_F1	F1 function key
GLUT_KEY_F2	F2 function key
GLUT_KEY_F3	F3 function key
GLUT_KEY_F4	F4 function key
GLUT_KEY_F5	F5 function key
GLUT_KEY_F6	F6 function key
GLUT_KEY_F7	F7 function key
GLUT_KEY_F8	F8 function key
GLUT_KEY_F9	F9 function key
GLUT_KEY_F10	F10 function key
GLUT_KEY_F11	F11 function key
GLUT_KEY_F12	F12 function key
GLUT_KEY_LEFT	Left function key
GLUT_KEY_RIGHT	Up function key
GLUT_KEY_UP	Right function key
GLUT_KEY_DOWN	Down function key
GLUT_KEY_PAGE_UP	Page Up function key
GLUT_KEY_PAGE_DOWN	Page Down function key
GLUT_KEY_HOME	Home function key
GLUT_KEY_END	End function key
GLUT_KEY_INSERT	Insert function key

为了让上面 processSpecialKeys 函数能编译通过，我们还必须定义，red，green，blue 三个变量。此外为了得到我们想要的结果，我们必须修改 renderScene 函数。

```
.....
//所有的变量被初始化为 1，表明三角形最开始是白色的。
float red=1.0, blue=1.0, green=1.0;
void renderScene(void) {
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    glRotatef(angle,0.0,1.0,0.0);
    // glColor3f 设置绘制三角形的颜色。
    glColor3f(red,green,blue);
    glBegin(GL_TRIANGLES);
        glVertex3f(-0.5,-0.5,0.0);
        glVertex3f(0.5,0.0,0.0);
        glVertex3f(0.0,0.5,0.0);
    glEnd();
    glPopMatrix();
    angle++;
    glutSwapBuffers();
}
```

下面我们就该告诉 GLUT，我们刚刚定义的函数用来处理按键消息，也就是要调用 glutKeyboardFunc 和 glutSpecialFunc 函数，我们在 main 函数里调用它们。

CTRL，ALT 和 SHIFT 键组合

一些时候我们想知道要是一个组合键（modifier key）也就是 CTRL，ALT 或者 SHIFT 键被按下该如何处理。GLUT 提供了一个函数来检测什么时候有组合键被按下。这个函数仅仅只能在处理按键消息或者鼠标消息函数里被调用。

函数原型如下：

int glutGetModifiers(void);

这个函数的返回值是 glut.h 里预定义的三个常量里的一个，或它们的“或”组合。这三个常量是：

1：GLUT_ACTIVE_SHIFT: 返回它，当按下 SHIFT 键，或者按下 CAPS LOCK 键，注意两者同时按下时，不会返回这个值。

2：GLUT_ACTIVE_CTRL: 返回它，当按下 CTRL 键。

3：GLUT_ACTIVE_ALT：返回它,当按下 ALT 键。

注意，窗口系统可能会截取一些组合键（modifiers），这时就没有回调发生。现在让我们扩充 processNormalKeys，处理组合键。按下 r 键时 red 变量被设置为 0.0，当按下 ALT+r 时 red 被设置为 1.0。代码如下：

```
void processNormalKeys(unsigned char key, int x, int y)
{
    if (key == 27)
        exit(0);
    else
        if (key=='r') {
            int mod = glutGetModifiers();
            if (mod == GLUT_ACTIVE_ALT)
                red = 0.0;
            else
                red = 1.0;
        }
}
```

注意如果我们按下 R 键，将不会有什么发生，因为 R 与 r 键的 ASCII 码不同。即这是两个不同的键。最后就是如何检测按键 CTRL+ALT+F1。这种情况下，我们必须同时检测两个组合键，为了完成操作我们需要使用“或”操作符。下面的代码段，使你按下 CTRL+ALT+F1 时颜色改变为红色。

```
void processSpecialKeys(int key, int x, int y)
{
    int mod;
```

```
    switch(key) {
        case GLUT_KEY_F1:
            mod = glutGetModifiers();
            if (mod==(GLUT_ACTIVE_CTRL|GLUT_ACTIVE_ALT)) {
                red = 1.0; green = 0.0; blue = 0.0;
            }
            break;
        case GLUT_KEY_F2:
            red = 0.0;
            green = 1.0;
            blue = 0.0;
            break;
        case GLUT_KEY_F3:
            red = 0.0;
            green = 0.0;
            blue = 1.0;
            break;
    }
}
```

（二）鼠标

在前一小节，我们看了怎么使用 GLUT 的 keyboard 函数来增加 OpenGL 程序的交互性，下面我们来研究一下鼠标。GLUT 的鼠标接口提供一系列的选项来增加鼠标的交互性。这就是检测鼠标单击和鼠标移动。

检测鼠标 Clicks

和键盘处理一样，GLUT 为你的注册函数（也就是处理鼠标 clicks 事件的函数）提供了一个方法。函数 glutMouseFunc，这个函数一般在程序初始化阶段被调用。

函数原型如下：

```
void glutMouseFunc(void(*func)(int button, int state, int x, int y));
```

参数：

func：处理鼠标 click 事件的函数的函数名。

从上面可以看到到，处理鼠标 click 事件的函数，一定有 4 个参数。第一个参数表明哪个鼠标键被按下或松开，这个变量可以是下面的三个值中的一个：

GLUT_LEFT_BUTTON
GLUT_MIDDLE_BUTTON
GLUT_RIGHT_BUTTON

第二个参数表明，函数被调用发生时，鼠标的状态，是被按下，还是松开，可能取值如

下：

GLUT_DOWN

GLUT_UP

当函数被调用时，state 的值是 GLUT_DOWN，那么程序可能会假定将会有个 GLUT_UP 事件，甚至鼠标移动到窗口外面，也如此。然而，如果程序调用 glutMouseFunc 传递 NULL 作为参数，那么 GLUT 将不会改变鼠标的状态。

剩下的两个参数（x,y）提供了鼠标当前的窗口坐标（以左上角为原点）。

检测动作（motion）

GLUT 提供鼠标 motion 检测能力。有两种 GLUT 处理的 motion：主动移动（active motion）和被动移动（passive motion）。Active motion 是指鼠标移动并且有一个鼠标键被按下。Passive motion 是指当鼠标移动时，并没有鼠标键按下。如果一个程序正在追踪鼠标，那么鼠标移动期间，每一帧将产生一个结果。

和以前一样，你必须注册用来处理鼠标事件的函数（定义函数）。GLUT 让我们可以指定两个不同的函数，一个追踪 passive motion，另一个追踪 active motion。

它们的函数原型，如下：

void glutMotionFunc(void(*func)(int x, int y));

void glutPassiveMotionFunc(void(*func)(int x, int y));

指定移动和被动移动回调函数。鼠标的位置(x,y)将返回给这两类回调函数。

参数：

Func：处理各自类型 motion 的函数名。

处理 motion 的参数函数的参数(x,y)是鼠标在窗口的坐标，以左上角为原点。

检测鼠标进入或离开窗口

GLUT 还能检测鼠标离开或进入窗口区域。可以定义一个回调函数去处理这两个事件。GLUT 里，调用这个函数的是 glutEntryFunc 函数。

函数原型如下：

void glutEntryFunc(void(*func)(int state);

参数：

Func：处理这些事件的函数名。

上面函数的参数中，state 有两个值：

GLUT_LEFT

GLUT_ENTERED

表明，是离开，还是进入窗口。

把它们放一起

首先我们要做的是在 GLUT 里定义哪些函数将负责处理鼠标事件。因此我们将重写 main 函数，让它包含所有必须的回调注册函数。

```
void main(int argc, char **argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DEPTH|GLUT_DOUBLE|GLUT_RGBA);
    glutInitWindowPosition(100,100);
    glutInitWindowSize(320,320);
    glutCreateWindow("Sample");
    glutDisplayFunc(renderScene);
    glutIdleFunc(renderScene);
    glutReshapeFunc(changeSize);
    // 这里添加鼠标处理回调函数
    glutMouseFunc(processMouse);
    glutMotionFunc(processMouseActiveMotion);
    glutPassiveMotionFunc(processMousePassiveMotion);
    glutEntryFunc(processMouseEntry);
    glutMainLoop();
}
```

好了，现在做点有趣的。我们将定义那些将做一些不可思议事件的回调函数。当一个鼠标键和 alt 键都被按下，我们将改变三角形的颜色。鼠标左键使三角形变成红色，中间的将三角形变成绿色，鼠标右键将三角形变成蓝色。函数如下：

```
void processMouse(int button, int state, int x, int y) {
    specialKey = glutGetModifiers();
    // 当鼠标键和 alt 键都被按下
    if ((state == GLUT_DOWN) &&(specialKey == GLUT_ACTIVE_ALT))
    {
        // set the color to pure red for the left button
        if (button == GLUT_LEFT_BUTTON) {
            red = 1.0; green = 0.0; blue = 0.0;
            }
        // set the color to pure green for the middle button
        else if (button == GLUT_MIDDLE_BUTTON) {
            red = 0.0; green = 1.0; blue = 0.0;
            }
        // set the color to pure blue for the right button
        else { red = 0.0; green = 0.0; blue = 1.0;
        }
```

```
    }
}
```

接下来有一个精细的颜色拾取方法。当一个鼠标键被按下，且 alt 键被按下。我们把 blue 设为 0.0，并且让 red 和 green 分量的值取决于鼠标在窗口中的位置。

函数如下：

```
void processMouseActiveMotion(int x, int y) {
    // the ALT key was used in the previous function
    if (specialKey != GLUT_ACTIVE_ALT) {
    // setting red to be relative to the mouse position inside the window
        if (x < 0) red = 0.0;
        else if (x > width) red = 1.0;
            else red = ((float) x)/height;
            // setting green to be relative to the mouse position inside the window
        if (y < 0) green = 0.0;
        else if (y > width) green = 1.0;
            else    green = ((float) y)/height;
        // removing the blue component
        blue = 0.0;
    }
}
```

下面给 passive motion 添加一些动作。当 shift 键被按下，鼠标将在 x 轴上有一个旋转。我们要修改 renderScene 函数。

函数如下：

```
float angleX = 0.0;
...
void renderScene(void) {
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    glRotatef(angle, 0.0, 1.0, 0.0);
    // This is the line we added for the rotation on the X axis
    glRotatef(angleX, 1.0, 0.0, 0.0);
    glColor3f(red, green, blue);
    glBegin(GL_TRIANGLES);
        glVertex3f(-0.5, -0.5, 0.0);
        glVertex3f(0.5, 0.0, 0.0);
        glVertex3f(0.0, 0.5, 0.0);
    glEnd();
    glPopMatrix();
    angle++;
    glutSwapBuffers();
}
```

现在我们添加一个函数处理 passive motion 事件，该函数将改变 angleX 的值。

```
void processMousePassiveMotion(int x, int y) {
    // User must press the SHIFT key to change the rotation in the X axis
    if (specialKey != GLUT_ACTIVE_SHIFT) {
        // setting the angle to be relative to the mouse position inside the window
        if (x < 0)
            angleX = 0.0;
        else if (x > width)
                angleX = 180.0;
            else
                angleX = 180.0 * ((float) x)/height;
    }
}
```

最后鼠标离开窗口将使动画停止，为了做到这样，我们也需要改变函数 renderScene。

```
// initially define the increase of the angle by 1.0
float deltaAngle=1.0;
...
void renderScene(void) {
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    glRotatef(angle, 0.0, 1.0, 0.0);
    glRotatef(angleX, 1.0, 0.0, 0.0);
    glColor3f(red, green, blue);
    glBegin(GL_TRIANGLES);
        glVertex3f(-0.5, -0.5, 0.0);
        glVertex3f(0.5, 0.0, 0.0);
        glVertex3f(0.0, 0.5, 0.0);
    glEnd();
    glPopMatrix();
    // this is the new line previously it was: angle++
    Angle += deltaAngle;
    glutSwapBuffers();
}
```

processMouseEntry 是最后一个函数。注意，这个函数在微软操作系统下可能工作的不是很好。

```
void processMouseEntry(int state) {
    if (state == GLUT_LEFT)
        deltaAngle = 0.0;
    else
        deltaAngle = 1.0;
}
```

三. 拾取

拾取是交互式图形系统的重要环节，比如可以通过在屏幕上使用鼠标点击，击中画面中的物体，或者使用鼠标拖动画面上的物体，这都需要拾取，即首先需要确定点击中的物体。拾取为图形操作提供了直观手段；而反馈则是指将图形系统在屏幕上的输出截获为图形数据，常被用于图形数据交换、图形后处理等应用。比如可以将屏幕上的画面（图像）保存为 Postscript 文件（矢量图形文件）、可以把 OpenGL 的绘制结果输出到绘图仪（HP GL2 格式）、甚至使用反馈模式，将 OpenGL 的绘制结果输出为 XML 文件，在服务器和移动设备间传递图形数据，这样在移动设备上只需要进行 2D 的图形绘制。本节将介绍拾取和反馈功能。

（一）OpenGL 中的拾取机制

OpenGL API 提供了对 3D 场景中的物体进行拾取的机制。下面将介绍如何检测鼠标点中或落在 OpenGL 窗口方形区域内的物体。检测鼠标点中的物体的基本步骤如下:

1. 得到鼠标点击位置的窗口坐标；
2. 进入选择模式；
3. 重定义观察体，使得只有光标覆盖的小窗口内的物体被绘制；
4. 绘制场景，可以是所有的物体或只是可拾取的物体；
5. 退出选择模式，并识别出在拾取窗口中被绘制的物体。

为了能够识别绘制的物体，必须对场景中的相关物体进行命名。0penGL API 提供了对单个物体或物体集合进行命名的机制。当进入了选择模式（OpenGL API 提供的特别绘制模式），物体并不被绘制到帧缓存，而是物体的名字（加上深度信息）被收集到一个数组中。对于未命名的物体，则只有深度信息被收集。

使用 0penGL 的术语，当一个元素以选择模式被绘制时则触发一次“击中”（hit）。击中的记录都保存在选择缓存中。当退出选择模式时，0penGL 返回记录了击中信息的选择缓存。由于 0penGL 也提供了每次击中时的深度信息，应用程序可以很容易地识别出哪一个物体最靠近用户。

(一) 名字堆栈

如同这个名字暗示的，名字堆栈是将物体指定的名字以堆栈形式保存。但事实上，并不使用字符串为对象命名，而是使用数字标记物体。然而，由于 OpenGL 使用了名字这个术语，本节也将使用名字而不是数字。当一个物体被绘制时，如果它和新的观察体相交，则一个击中记录被创建。击中记录保存了在名字堆栈上的当前名字、该物体的最大和最小深度值。

注意：即使是名字堆栈为空，一个击中记录也将被创建，这时它也只包含深度信息。如果更多的物体在名字堆栈被修改前绘制或者程序退出了选择模式，则击中记录中的深度值将被相应地修改。只有当名字堆栈的当前内容被修改后或程序退出了选择模式，一个击中记录才被保存在选择缓存上。因此选择模式下的绘制函数需要对名字堆栈的内容以及元素的绘制负责。

OpenGL 提供了以下函数来操纵名字堆栈：

(1) glInitNames()函数

void glInitNames();

函数 glInitNames()创建一个空的名字堆栈，在压入名字前需要先对名字堆栈进行初始化。

void glPushName(GLuint name);

在栈顶压入名字。名字堆栈的最大尺寸依赖于特定实现，然而根据 OpenGL 规范，它必须至少容纳 64 个名字，这对于绝大多数应用程序来说足够了。如果希望能确保安全，可以用状态变量 GL_NAME_STACK_DEPTH 进行查询。

glGetIntegerv(GL_NAME_STACK_DEPTH);

当压入的值超出了名字堆栈的容量，则会产生一个溢出错误：GL_STACK_OVERFLOW。

void glPopName();

从栈顶去掉名字，从空栈中去掉名字将会产生一个下溢错误：GL_STACK_UNDER_FLOW。

(2) glLoadName()函数

void glLoadName(GLuint name);

函数 glLoadName()将用 name 替换栈顶的名字。它和下面的调用等价：

glPopName();

glPushName(name);

这个函数是上面代码片断的缩写，在一个空的堆栈上转载名字将触发一个错误：GL_INVALID_OPERATION。

注意：对以上函数的调用在非选择模式时将被忽略。这意味着可以使用单个的绘制函数，它包含了所有的名字堆栈函数。在正常绘制时，这些函数将被忽略，而进入选择模式时，则击中记录将被收集。

注意：不能在 glBegin 和 glEnd 间放置这些函数，这会造成一些困难，使得某些情况下必须有一个新的绘制函数在选择模式下使用。例如，在 glBegin 和 glEnd 间绘制了许多点，

如果希望对这些点分组命名，就不得不对每个名字都创建一个 glBeign 和 glEnd 块。

下面的代码演示了选择模式下的绘制过程：

```
#define BODY
#define HEAD
.....
void renderInSelectionMode()
{
    glInitNames();
    glPushName(BODY) ;
    drawBody();
    glPopName();
    glPushName(HEAD) ;
    drawHead();
    drawEyes();
    glPopName();
drawGround();
}
```

下面逐行解释：

1) glInitNames();

该函数创建一个空的名字堆栈，这需要在任何其他的堆栈操纵前被调用，如 Load、Push 或 Pop。

2) glPushName(BODY);

一个名字被压入堆栈，该堆栈中现在只有一个名字。

3) drawBody();

该函数调用了 OpenGL 元素来进行绘制。如果任何被绘制的元素和观察体相交，则一条击中记录将被生成。击中记录的信息包含了当前名字堆栈上的名字——BODY 以及和观察体相交的元素的最大和最小深度值。

4) glPopName();

该函数用来去掉栈顶的名字。由于当前栈中只有一个名字，现在则为空了。由于名字堆栈被修改了，如果在行 3 中创建了击中记录，则该纪录将被保存到选择缓存中。

5) glPushName(HEAD);

该函数的作用是：又压入一个名字到栈中。现在名字堆栈中还是只有一个名字，堆栈被修改了，但是由于没有击中记录，因此没有内容添加到选择缓存中。

6) drawHead();

drawHead()函数是又一个绘制 OpenGL 元素的函数。同样，如果任何被绘制的元素和

观察体相交，则一条击中记录将被生成。

7) drawEyes();

drawEyes()函数仍然是一个绘制 OpenGL 元素的函数。同样，如果任何被绘制的元素和观察体相交，并且在行 6 中已经生成了击中记录，则该击中记录将被更新。击中记录中的名字不变，但是如果 drawEyes()中有其他的元素，有着更小的最小值或更大的最大值，则这些新值将被保存在击中记录中。如果 drawEyes()中的元素和观察体相交，而 drawHead()中并没有产生击中记录，则一条新的击中记录将会生成。

8) glPopName();

该函数用来去掉栈顶的名字。由于当前栈中只有一个名字，现在则为空了。同样，由于名字堆栈被修改了，如果有击中记录生成，则该纪录将被保存到选择缓存中。

9) drawGround();

如果在此处被绘制的 OpenGL 元素和观察体相交，则将会有一条击中记录生成。由于名字堆栈为空，击中记录中将没有名字，只有深度信息。如果在随后的代码中不再修改名字堆栈，则创建的击中记录将只会在应用程序退出选择模式时被加入到选择缓存中。

注意：行 4 和行 5 可以替换为

glLoadName(HEAD);

注意：可以在开始时就压入一个空白名字（一个不会被使用的值），随后使用 glLoadName()，而不是 glPushName()和 glPopName()。直接扔掉无名对象比识别对象是否是空白的名字要快，但另一方面，使用 glLoadName()可能比 glPopName()加上 glPushName()要快。

1．对一个对象使用多个名字

并没有规则指明一个对象只能有一个名字。可以对一个对象指定多个名字。假定在一张网格上分布着许多雪人。除了使用 l，2，3，...对它们进行命名外，也可以用它们所在的行数和列数进行命名。这就需要每个雪人有两个名字来描述它的位置，即网格的行数和列数。

下面的两个函数给出了两种不同的方法，第一个使用了单一命名：

```
for(int i=-3; i<3; i++)
    for(int j=-3; j<3; j++){
        glPushMatrix();
        glPushName(i*6+j);
        glTranslatef(i*10.0, 0, 0, j*10.0);
        glCallList(snowManDL);
        glPopName();
```

```
            glPopMatrix();
}
```

下面的函数对每个雪人指定两个名字，在这种情况下，如果一条击中记录生成时，它将包含两个名字：

```
for(int i=-3; i<3; i++){
    glPushName(i);
    for(int j=-3; j<3; j++){
        glPushMatrix();
        glPushName(j);
        glTranslatef(i*10.0, 0, 0, j*10.0);
        glCallList(snowManDL);
        glPopName();
        glPopMatrix();
}
    glPopName();
}
```

一个对于多重命名的自然扩展是层次命名，这可以用来识别点中的特定部位。比如，不仅希望知道是哪个雪人被点中，还想了解是头部还是身体被点中。下面的函数提供了这些信息：

```
For(int i=-3; i<3; i++)
{
    glPushName(i);
    for(int j=-3; j<3; j++){
        glPushMatrix();
        glPushName(j);
        glTranslatef(i*10.0, 0.0, j*10.0);
        glPushName(HEAD);
        glCallList(snowManHeadDL);
        glLoadName(BODY);
        glCallList(snowManBodyDL);
        glPopName();
        glPopName();
        glPopMatrix();
    }
    glPopName();;
}
```

这种情况下，当点中雪人的头部或身体时，将会得到三个名字：行数、列数和 BODY 或 HEAD。

2．选择模式进行绘制

前面提到过，在非选择模式时（绘制到帧缓存），所有的堆栈函数调用将被忽略。这意

味着，可以在两种模式下使用同一个绘制函数。然而，这可能造成时间上的很大浪费。一个应用程序可能只有很少可以选择的物体。而在两种模式下使用同一个绘制函数将会造成在选择模式下大量绘制不可拾取的对象。不仅绘制时间加长、处理击中记录的时间也被加长。

因此，在这些情况下，对于选择模式编写特别的绘制函数是有意义的。但是，这样做也需要花费更多的心血，比如，在一个应用程序中，绘制了几个房间，每个房间中都有几件可供拾取的物体，而必须避免用户穿墙拾取物体的情况（即物体在其他的房间，应该不可见）。如果使用了同样的绘制函数，则墙体会造成击中，于是可以使用深度信息扔掉墙后的物体。但是，当决定在选择模式下只绘制可拾取物体时，则无法分辨物体是否在当前房间或在墙后。

因此，当为选择模式构造特别的绘制函数时，不仅需要考虑可拾取的物体，还需要考虑可能造成遮挡的其他物体，比如墙。对于高交互性的实时应用，一个可能的办法是对可能造成遮挡的物体进行简化表示，例如使用一个多边形替换复杂的墙体、一个盒子替换一张桌子等。

3．选择模式

上面已经介绍了 OpenGL 命名机制，下面将介绍如何进入为拾取设计的选择模式。

首先，告诉 OpenGL 击中记录的保存地点，通过以下函数来完成：

void glSelectBuffer(GLsizei size, GLuint *buffer);

参数：

buffer：GLuint 的数组，用于给 OpenGL 保存击中记录

size：该 buffer 的尺寸

在进入选择模式前需要调用该函数，然后使用下面函数进入选择模式：

void glRenderMode(GLenum mode);

参数：

Mode：使用 GL_SELECT 进入选择模式，GL_RENDER 返回正常绘制模式，后者是缺省值

下面是精妙的部分。应用程序需要重新定义观察体，使得只用鼠标点中的临近区域被绘制。为了实现这点，需要将矩阵模式设置为 GL_PROJECTION，应用程序将当前矩阵压入堆栈保存，然后初始化矩阵。下面的代码演示了如何来做：

```
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIndentity();
```

现在投影矩阵变成了单位矩阵，需要定义观察体，使得只用光标附件的小区域被绘制，

可以用下面的函数来实现：

void gluPickMatrix(GLdouble x, GLdouble y, GLdouble width, GLdouble height, GLint viewport[4]);

参数：

x，y：代表了光标的位置，它们定义了拾取区域的中心，用窗口坐标指定，然后需要注意 0penGL 中的窗口坐标原点在视区的左下角，而操作系统通常在左上角

width，height：拾取区域的尺寸，太小，则用户很难选择小的物体；太大，则有可能选择太多的物体

viewport：当前视区

在调用上面函数前，需要获得当前视区，这可以通过使用 OpenGL 状态变量 GL_VIEWPORT 来查询得到（使用函数 glGetIntegerv()），然后调用 gluPickMatrix，最后设置投影（透视或正交），这和正常绘制模式下一样。接着切换到 GL_MODELVIEW 矩阵模式，并初始化名字堆栈开始绘制。下面的代码演示了使用透视投影模式的函数调用序列。

```
glGetIntegerv(GL_VIEWPORT, viewport);
gluPickMatrix(cursorX, viewport[3]-cursorY, 5, 5, viewport);
gluPerspective(45, ratio, 0.1, l000);
glMatrixMode(GL_MODELVIEW);
glInitNames();
```

注意：gluPickMatrix 的第二个参数，前面提到过 0penGL 的窗口坐标原点与操作系统的不一致，第二个参数提供了两者间的转换，即将原点从左上角变换到左下角。

例子中的拾取区域是一个 5×5 的窗口。大家可能会发现这并不适用于自己的应用程序，如果很难拾取到物体，可以进行调整。

下面的函数给出了进入选择模式和开始拾取的所有操作。假定 cursorX 和 cursorY 是鼠标点击处的操作系统窗口坐标：

```
#define BUFSIZE 512
GLuint selectBuf[BUFSIZE];
…
void startPicking(int cursorX, int cursorY)
{
    GLint viewport[4];
    glSelectBuffer(BUFSIZE, selectBuf);
    glRenderMode(GL_SELECT);
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
```

```
    glGetIntegerv(GL_VIEWPORT, viewport);
    gluPickMatrix(cursorX, viewport[3]-cursorY, 5, 5, viewport);
    gluPerspective(60, 1, 0.1, 1000);
    glMatrixMode(GL_MODELVIEW);
    glInitNames();
}
```

大家可以将上面的代码用于自己的应用程序，并参考自己的投影模式进行适当修改，但只在进入绘制时，并在任何图形元素被绘制前。

4．处理击中记录

为了处理击中记录，应用程序需要首先返回正常绘制模式，这通过下面的调用实现：

glRenderMode(GL_RENDER);

这个函数将返回在选择模式下进行绘制时创建的击中记录的个数。在这以后应用程序可以开始处理选择缓存。注意，在这之前，并不能保证击中记录都被保存在选择缓存中。进一步来讲，需要将投影矩阵重置。由于在进入选择模式前使用了函数 glPushMatrix()进行保存，只需要调用函数 glPopMatrix()进行恢复：

```
void stopPicking(){
    int hits;
    //重新载入原来的投影矩阵
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();
    glMatrixMode(GL_MODELVIEW);
    glFlush();
    //返回法线绘制模型
    hits=glRenderMode(GL_RENDER);
    //如果有 hits 执行它们
    if(hits!=0)
        processHits(hits, selectBuf);
}
```

最后的任务是解析选择缓存。选择缓存按照击中记录的生成次序(即绘制次序)对它们进行了序列化保存。注意深度检测没有画出的元素也会产生击中记录。由于包含名字的个数不同，击中记录的尺寸是变长的。

击中记录的第一个字段保存了它包含的名字的个数。第二和第三个字段保存了击中的最小和最大的深度值。只有和视线体相交的元素的顶点被统计。而被裁减的多边形 OpenGL 将构造出新的顶点。因此，得到的是和视线体相交的元素的最大和最小深度值，而不是所有被绘制的元素。深度值从 Z 缓存中取出（在其中值域为[0,1]），并和 2^{32}-1 相乘，然后取整。注意：由于 Z 缓存的非线性特性，得到的深度值并不和顶点到视点的距离成线性比例。然后

是名字的序列,这些名字都来自击中记录产生时的名字堆栈。注意:由于名字的个数可能为 0,所以这个序列也可能为空。

下表是一个有三个击中记录的选择缓存的示例。

击中记录的内容	描 述
0	第一击中记录中没有名字
4.2822e+009	第一击中记录的最小深度值
4.28436e+009	第一击中记录的最大深度值
1	第二击中记录中的名字数目
4.2732e+009	第二击中记录的最小深度值
4.27334e+009	第二击中记录的最大深度值
6	第二击中记录的唯一名字
2	第三击中记录中的名字数目
4.27138e+009	第三击中记录的最小深度值
4.27155e+009	第三击中记录的最大深度值
2	第三击中记录的第一名字
5	第三击中记录的第二名字

为了识别出最靠近视点的物体，可以使用深度信息，例如可以选择具有最小深度值的物体，作为用户击中的对象。以上表的例子来说，是第三个击中。下面的函数打印出最近对象的名字。

```
void processHits(GLint hits, GLuint buffer[])
{
    unsigned int i,j;
    GLuint names, *ptr, minZ, *ptrNames, numberOfNames;
    printf("hits=%d\n", hits);
    ptr=(GLuint *) buffer;
    minZ=0xffffffff;
    for(i=0; i<hits; i++){
        names= *ptr;
        ptr++;
        if(*ptr<minZ){
            numberOfNames=names;
            minZ=*ptr;
            ptrNames=ptr+2;
        )
        ptr+=names+2;
    }
    printf("The closest hit names are");
    ptr=ptrNames;
    for(j=0; j<numberOfNames; j++, ptr++){
```

```
            printf("%d", *ptr);
        }
        Printf("\n");
}
```

通过鼠标点击可以选中鼠标点击处最前面的雪人，并能分辨头部或身体被点中，将被点中的部位以线框方式突出显示。