OpenGL之矩阵变换

简述：

OpenGL通过矩阵变换来把三维物体转变为二维图象，进而在屏幕上显示出来。为了指定当前操作的是何种矩阵，使用了函数 **glMatrixMode** 。

可以移动、旋转观察点或者移动、旋转物体，使用的函数是**glTranslate***和 **glRotate*** 。

可以缩放物体，使用的函数是 **glScale***。

可以定义可视空间，这个空间可以是**“**正投影**”**的（使用 **glOrtho**或**gluOrtho2D**），也可以是**“**透视投影**”**的（使用 **glFrustum**或**gluPerspective** ）。

可以定义绘制到窗口的范围，使用的函数是 **glViewport** 。

矩阵有自己的**“**堆栈**”**，方便进行保存和恢复。这在绘制复杂图形时很有帮助。使用的函数是 **glPushMatrix** 和 **glPopMatrix** 。

前言：

我们生活在一个三维的世界——如果要观察一个物体，可以：

A. 从不同的位置去观察它。（视图变换）

B. 移动或者旋转它，还可以放大或缩小它。（模型变换）

C. 若把物体画下来，可以选择：是否需要一种**“**近大远小**”**的透视效果。另外，可能只希望看到物体的一部分，而不是全部（剪裁）。（投影变换）

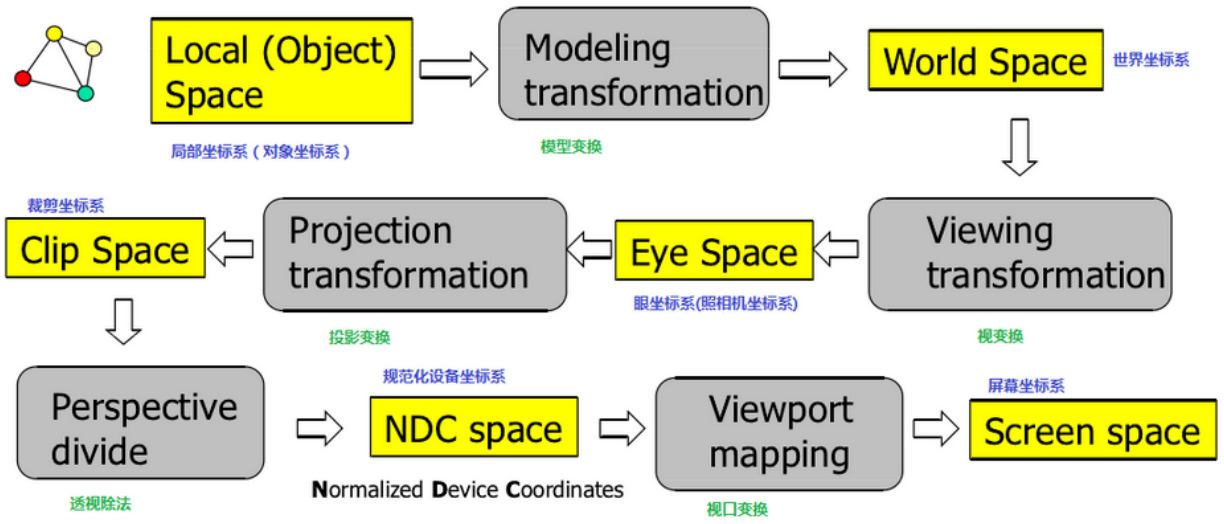
D. 可能希望把整个看到的图形画下来，但它只占据纸张的一部分，而不是全部。（视口变换）

这些，都可以在**OpenGL**中实现。

OpenGL变换实际上是通过矩阵乘法来实现。无论是移动、旋转还是缩放大小，都是通过在当前矩阵的基础上乘以一个新的矩阵来达到目的。

OpenGL可以在最底层直接操作矩阵。

综上，**OpenGL**中的坐标处理包括模型变换、视变换、投影变换、视口变换等内容，具体过程如下图所示：



1、模型变换和视图变换

从“相对移动”的观点来看，改变观察点的位置与方向和改变物体本身的位置与方向具有等效性。在**OpenGL** 中，实现这两种功能甚至使用的是同样的函数。

模型和视图的变换都通过矩阵运算来实现。

首先，设置当前操作的矩阵为“模型视图矩阵”，即：

```
glMatrixMode(GL_MODELVIEW);  // 选择模型观察矩阵
```

接着，在进行变换前把当前矩阵设置为单位矩阵。

```
glLoadIdentity();
```

然后，就可以进行模型变换和视图变换了。

模型变换通过对模型执行平移***(translation)***、缩放***(scale)***、旋转***(rotation)***、镜像***(reflection)***、错切***(shear)***等操作，来调整模型的过程。通过模型变换，可以按照合理方式指定场景中物体的位置等信息。

进行模型和视图变换，主要涉及到三个函数：

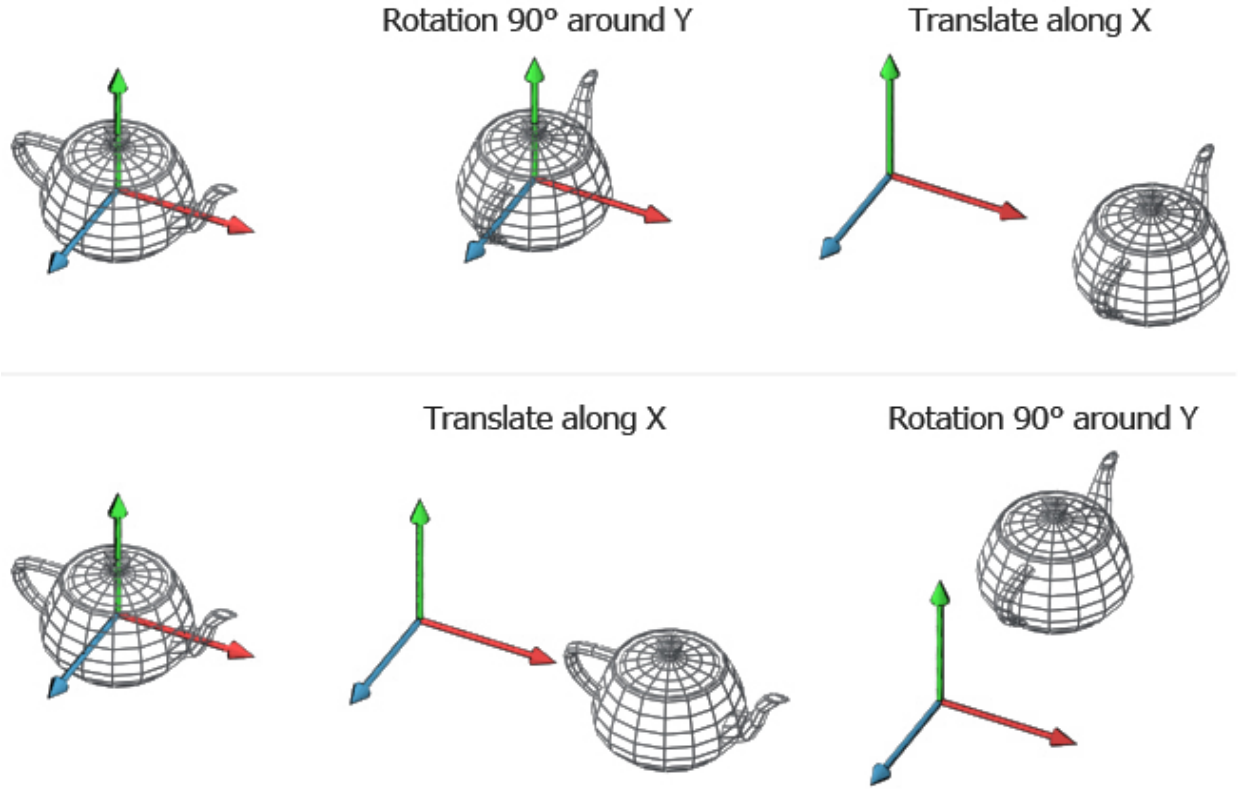
A. glTranslate* ，把当前矩阵和一个表示移动物体的矩阵相乘。三个参数分别表示了在三个坐标上的位移值。

B. glRotate* ，把当前矩阵和一个表示旋转物体的矩阵相乘。物体将绕着 **(0 , 0 , 0)**到 **(x , y , z)** 的直线以逆时针旋转，参数**angle** 表示旋转的角度。

C. glScale* ，把当前矩阵和一个表示缩放物体的矩阵相乘。 **x , y , z**分别表示在该方向上的缩放比例。

为什么是相乘？假设当前矩阵为单位矩阵，然后先乘以一个表示旋转的矩阵 **R** ，再乘以一个表示移动的矩阵 **T** ，最后得到的矩阵再乘上每一个顶点的坐标矩阵**v**。所以，经过变换得到的顶点坐标就是 **((RT) v)** 。由于矩阵乘法的结合率，**((RT)v) = (R(Tv))**，换句话说，实际上是先进行移动，然后进行旋转。即：实际变换的顺序与代码中写的顺序是相反的。

注意，“先移动后旋转”和“先旋转后移动”得到的结果很可能不同。因为，矩阵乘法不满足交换律，即**AB≠BA**。



OpenGL之所以这样设计，是为了得到更高的效率。但在绘制复杂的三维图形时，如果每次都去考虑如何把变换倒过来，也是很痛苦的事情。这里介绍另一种思路，可以让代码看起来更自然（写出的代码其实完全一样，只是考虑问题时用的方法不同了）。

假如，坐标并不是固定不变的。旋转的时候，坐标系统随着物体旋转。移动的时候，坐标系统随着物体移动。如此一来，就不需要考虑代码的顺序反转的问题了。

以上都是针对改变物体的位置和方向来介绍的。如果要改变观察点的位置，除了配合使用 **glRotate*** 和 **glTranslate*** 函数以外，还可以使用这个数： **gluLookAt** 。它的参数比较多，前三个参数表示了观察点的位置，中间三个参数表示了观察目标的位置，最后三个参数代表从**(0,0,0)**到 **(x,y,z)**的直线，它表示了观察者认为的“上”方向。

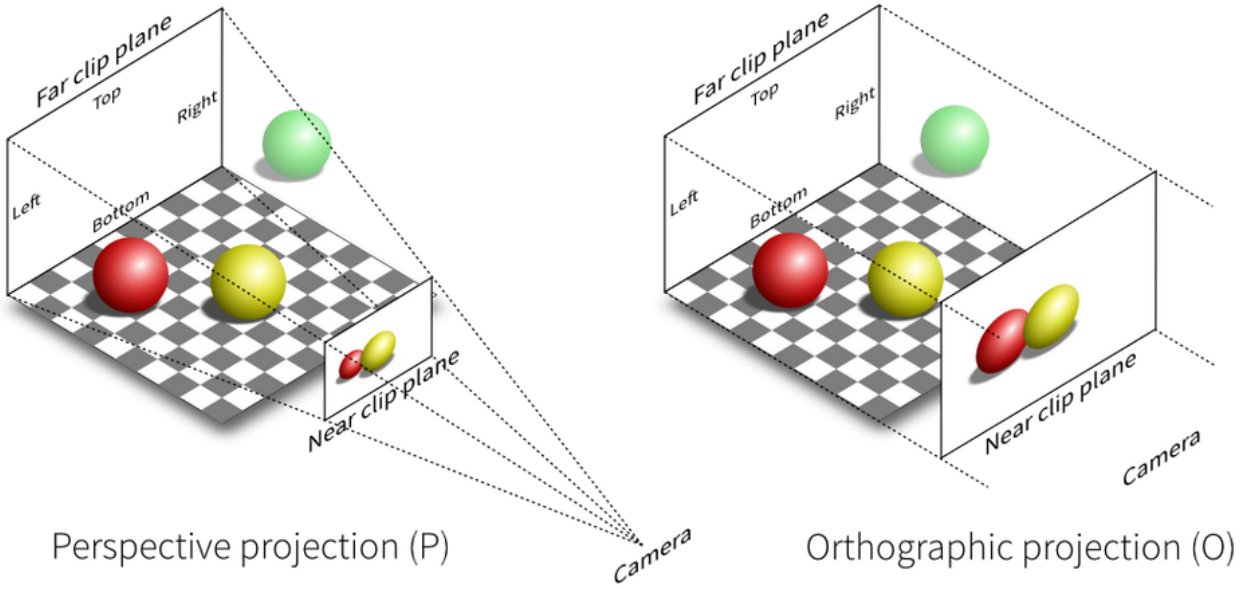
```
gluLookAt(GLdouble eyex,GLdouble eyey,GLdouble eyez,GLdouble centerx,GLdouble centery, \
  GLdouble centerz,GLdouble upx,GLdouble upy,GLdouble upz); //眼坐标，中间坐标（Z），上坐标（Y）
```

相当于在世界坐标系又建了个观察坐标系**gluLookAt(观察坐标系原点位置，观察坐标系Z轴，观察坐标系Y轴）**，**X**轴垂直于**Z,Y**，所以不用写。

2、投影变换

投影变换就是定义一个可视空间，可视空间以外的物体不会被绘制到屏幕上。（注意，坐标默认是**-1.0**到**1.0**，但可以更改）

OpenGL支持两种类型的投影变换，即透视投影**(perspective projection)**和正交投影**(orthographic projection)**。投影也是使用矩阵来实现的。



首先，设置当前操作矩阵模式为投影矩阵，即以**GL_PROJECTION**为参数调用 **glMatrixMode** 函数；

glMatrixMode(GL_PROJECTION); *//选择投影矩阵*

接着，在进行变换前把当前矩阵设置为单位矩阵。

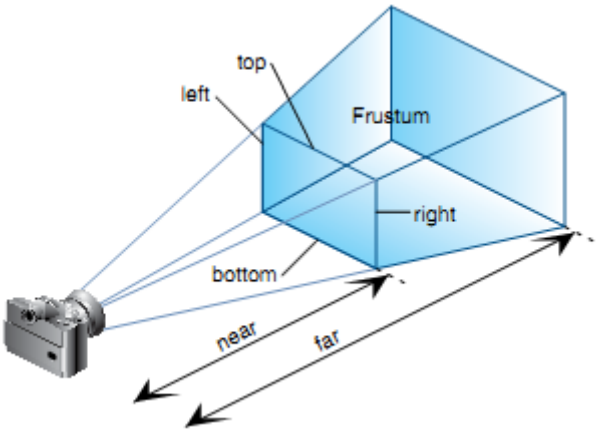
glLoadIdentity();

透视投影所产生的结果类似于照片，有近大远小的效果，如在火车头内向前照一个铁轨的照片，两条铁轨似乎在远处相交了；

使用 **glFrustum**函数可以将当前的可视空间设置为透视投影空间。

```
1 voidglFrustum(GLdouble left, GLdouble Right, GLdouble bottom, GL
 double top, GLdouble near, GLdouble far);
```

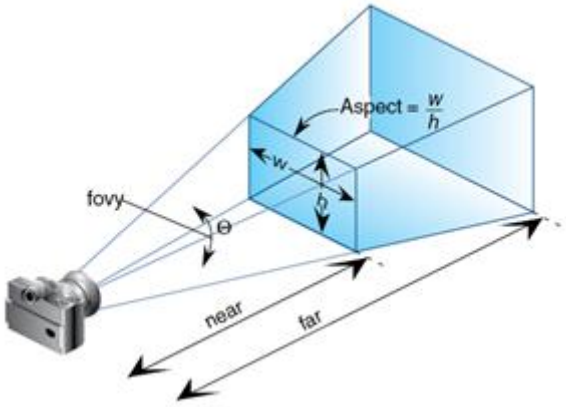
其参数的意义如下图：



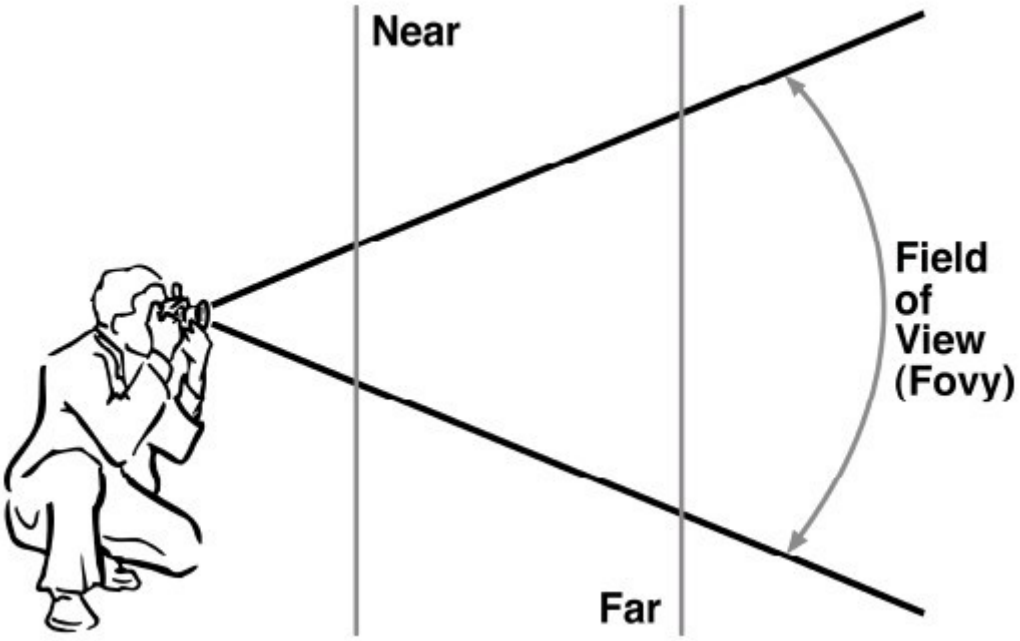
也可以使用更常用的 ***gluPerspective***函数。

API voidgluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar);

其参数的意义如下图：

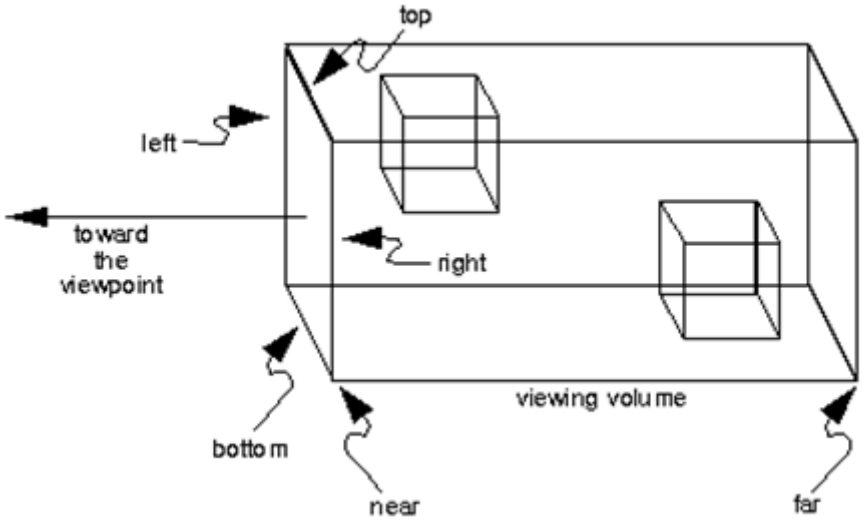


fovy指定视角，***aspect***指定宽高比，***zNear***和***zFar***指定剪裁平面



正投影相当于在无限远处观察得到的结果，只是一种理想状态。但对于计算机来说，使用正投影有可能获得更好的运行速度。

使用 **glOrtho** 函数可以将当前的可视空间设置为正投影空间。其参数的意义如下图：

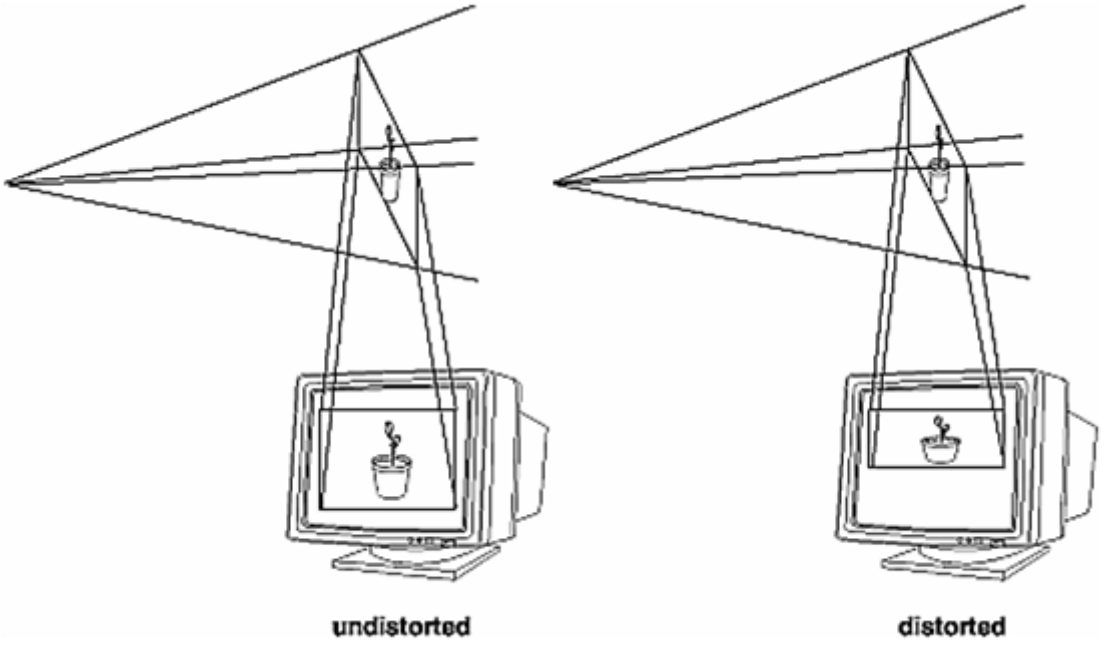


如果绘制的图形空间本身就是二维的，可以使用 ***gluOrtho2D*** 。他的使用类似于 ***glOrgho*** 。

```
glOrtho(GLdouble left,GLdouble right,GLdouble bottom,GLdouble top,GLdouble zNear,GLdouble zFar);
```

3、视口变换

当一切工作就绪，只需要把像素绘制到屏幕上了。那应该把像素绘制到窗口的哪个区域呢？通常情况下，默认是完整的填充整个窗口，但完全可以只填充一半。（即：把整个图象填充到一半的窗口内）



使用***glViewport***来定义视口。其中前两个参数定义了视口的左下脚（***0,0***表示最左下方），后两个参数分别是宽度和高度。

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

4、操作矩阵堆栈

在进行矩阵操作时，可能需要先保存某个矩阵，过一段时间再恢复它。当需要保存时，调用 **glPushMatrix** 函数，它相当于把矩阵放到堆栈上。当需要恢复最近一次的保存时，调用 **glPopMatrix** 函数，它相当于把矩阵从堆栈上取下。**OpenGL** 规定堆栈的容量至少可以容纳**32**个矩阵，某些**OpenGL**实现中，堆栈的容量实际上超过了**32**个。因此不必过于担心矩阵的容量问题。

通常，用这种先保存后恢复的措施，比先变换再逆变换要更方便，更快速。

注意：模型视图矩阵和投影矩阵都有相应的堆栈。使用 **glMatrixMode** 来指定当前操作的究竟是模型视图矩阵还是投影矩阵。

5、综合举例

制作一个三维场景，包括了太阳、地球和月亮。假定一年有**12**个月，每个月**30**天。每年，地球绕着太阳转一圈。每个月，月亮围着地球转一圈。即一年有**360**天。现在给出日期的编号（**0~359**），要求绘制出太阳、地球、月亮的相对位置示意图。（这些数据是为了编程方便才这样设计的）

A. 首先，认定这三个天体都是球形，且它们的运动轨迹处于同一水平面，建立以下坐标系：太阳的中心为原点，天体轨迹所在的平面表示了**X**轴与**Y**轴决定的平面，且每年第一天，地球在**X**轴正方向上，月亮在地球的正**X**轴方向。

B. 下一步是确立可视空间。注意：太阳的半径要比太阳到地球的距离短得多。如果直接使用天文观测得到的长度比例，则当整个窗口表示地球轨道大小时，太阳的大小将被忽略。因此，只能成倍的放大几个天体的半径，以适应观察的需要。（百度一下，得到太阳、地球、月亮的大致半径分别是：**696000km**， **6378km**，**1738km**。地球到太阳的距离约为**1.5**亿**km=150000000km**，月亮到地球的距离约为**380000km**。）

C. 将三个天体的半径分别“修改”为：**69600000**（放大**100**倍），**15945000**（放大**2500**倍），**4345000**（放大**5000**倍）。将地球到月亮的距离“修改”为**38000000**（放大**100**倍）。地球到太阳的距离保持不变。

D. 为了让地球和月亮在离我们很近时，仍然不需要变换观察点和观察方向就可以观察它们，把观察点放在这个位置：**(0,-200000000, 0)**——因为地球轨道半径为**150000000**，凑个整，取**-200000000**就可以了。观察目标设置为原点 **0 0 0**（即太阳中心），选择**Z**轴正方向作为 “上”方。当然还可以把观察点往“上”方移动一些，得到**(0, -200000000, 200000000)**，这样可以得到**45**度角的俯视效果。

E. 为了得到透视效果，使用 **gluPerspective** 来设置可视空间。假定可视角为**60**度（如果调试时发现该角度不合适，可修改之。这里选择的数值是**75**。），高宽比为**1.0**。最近可视距离为**1.0**，最远可视距离为**200000000*2=400000000**。即：

gluPerspective(60, 1, 1, 400000000);

现在来看看如何绘制这三个天体。

A. 为了简单起见，把三个天体都想象成规则的球体。而所使用的 **glut** 实用工具中，正好就有一个绘制球体的现成函数：**glutSolidSphere**，这个函数在“原点”绘制出一个球体。由于坐标是可以通过 **glTranslate*** 和 **glRotate*** 两个函数进行随意变换的，所以就可以在任意位置绘制球体了。函数有三个参数：第一个参数表示球体的半

径，后两个参数代表了“面”的数目，简单点说就是球体的精确程度，数值越大越精确，当然代价就是速度越缓慢。这里只是简单的设置后两个参数为**20**。

B. 太阳在坐标原点，所以不需要经过任何变换，直接绘制就可以了。

C. 地球则要复杂一点，需要变换坐标。由于今年已经经过的天数已知为 **day** ，则地球转过的角度为 **day / 一年的天数 * 360度**。前面假定每年都是**360**天，因此地球转过的角度恰好为 **day** 。所以可以通过下面的代码来解决：

```
glRotatef(day, 0, 0, -1);
/* 注意地球公转是“自西向东”的，因此是饶着Z轴负方向进行逆时针旋转 */
glTranslatef(地球轨道半径, 0, 0);
glutSolidSphere(地球半径, 20, 20);
```

D. 月亮是最复杂的。因为它不仅要绕地球转，还要随着地球绕太阳转。但如果选择地球作为参考，则月亮进行的运动就是一个简单的圆周运动了。如果先绘制地球，再绘制月亮，则只需要进行与地球类似的变换。

```
glRotatef(月亮旋转的角度, 0, 0, -1);
glTranslatef(月亮轨道半径, 0, 0);
glutSolidSphere(月亮半径, 20, 20);
```

E. 但这个“月亮旋转的角度”，并不能简单的理解为 **day** / 一个月的天数**30*360**度。因为在绘制地球时，这个坐标已经是旋转过的。现在的旋转是在以前的基础上进行旋转，因此还需要处理这个“差值”。可以写成：**day / 30 * 360 - day**，即减去原来已经转过的角度。这只是一种简单的处理，当然也可以在绘制地球前用 **glPushMatrix** 保存矩阵，绘制地球后用 **glPopMatrix**恢复矩阵。再设计一个跟地球位置无关的月亮位置公式，来绘制月亮。通常后一种方法比前一种要好，因为浮点的运算是不精确的，即是说计算地球本身的位置就是不精确的。拿这个不精确的数去计算月亮的位置，会导致 “不精确”的成分累积，过多的“不精确”会造成错误。这个小程序没有去考虑这个，但并不是说这个问题不重要。

F. 还有一个需要注意的细节： **OpenGL**把三维坐标中的物体绘制到二维屏幕，绘制的顺序是按照代码的顺序来进行的。因此后绘制的物体会遮住先绘制的物体，即使后绘制的物体在先绘制的物体的“后面”也是如此。使用深度测试可以解决这一问题。使用的方法是：

a. 以**GL_DEPTH_TEST**为参数调用**glEnable**函数，启动深度测试。

b. 在必要时（通常是每次绘制画面开始时），清空深度缓冲，即：**glClear(GL_DEPTH_BUFFER_BIT);** 其中，**glClear(GL_COLOR_BUFFER_BIT)**与**glClear(GL_DEPTH_BUFFER_BIT)** 可

以合并写

为： **glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);** 且后者的运行速度可能比前者快。

```
// 太阳、地球和月亮
// 假设每个月都是30天
// 一年12个月，共是360天
staticint day = 200; // day的变化：从0到359
voidmyDisplay(void)
{
 glEnable(GL_DEPTH_TEST);
 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
 glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(75, 1, 1, 400000000);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(0, -200000000, 200000000, 0, 0, 0, 0, 0, 1);
 // 绘制红色的“太阳”
  glColor3f(1.0f, 0.0f, 0.0f);
  glutSolidSphere(69600000, 20, 20);
 // 绘制蓝色的“地球”
  glColor3f(0.0f, 0.0f, 1.0f);
  glRotatef(day/360.0*360.0, 0.0f, 0.0f, -1.0f);
  glTranslatef(150000000, 0.0f, 0.0f);
  glutSolidSphere(15945000, 20, 20);
 // 绘制黄色的“月亮”
  glColor3f(1.0f, 1.0f, 0.0f);
  glRotatef(day/30.0*360.0 - day/360.0*360.0, 0.0f, 0.0f,
-1.0f);
  glTranslatef(38000000, 0.0f, 0.0f);
  glutSolidSphere(4345000, 20, 20);
  glFlush();
 }
```