湖南科技大学课程教案
（章节、专题首页）

授课教师：王志喜　　职称：副教授　　单位：计算机科学与工程学院

课程名称	计算机图形图像技术
章节、专题	OpenGL中的图形变换
教学目标及基本要求	掌握OpenGL中图形变换的方法（含视图造型变换、投影变换和视口变换）。
教学重点	视图造型变换，投影变换
教学难点	视图造型变换，投影变换
教学内容与时间分配	（1）OpenGL顶点变换的步骤和常用的变换函数（0.4课时） （2）视图造型变换（0.2课时） （3）投影变换（0.4课时） （4）OpenGL中图形变换的例子（1课时） 共计2课时。
习　　题	见6.5.1节（程序设计题）。

第6章　OpenGL中的图形变换

6.1　顶点变换的步骤和常用的变换函数

6.1.1　顶点变换的步骤

如图6-1所示，OpenGL中指定顶点变换的步骤如下。

（1）指定视图和造型变换，两者统称为视图造型变换。该变换将输入的顶点坐标变换为观察坐标。

（2）指定投影变换，该变换实际上是投影变换和规范化变换的复合，将顶点从观察坐标变换到规范化坐标。该变换还定义了一个观察体，在观察体外的对象将被舍弃。

（3）指定视口变换，将规范化坐标变换为屏幕窗口坐标。

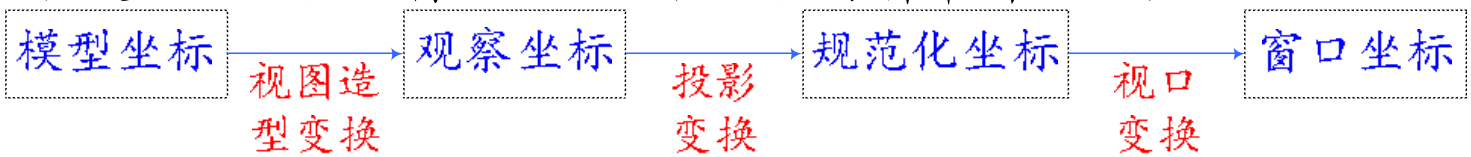


图6-1　顶点变换的步骤

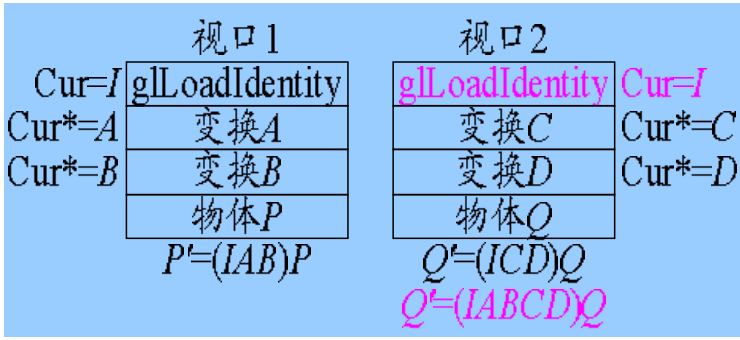
6.1.2 常用的变换函数

1. glMatrixMode()

【函数原型】void glMatrixMode(GLenum mode);

【功能】指定当前矩阵模式，随后指定的矩阵都属于这种模式。

【参数】可以选择GL_MODELVIEW（随后的矩阵是视图造型矩阵）、GL_PROJECTION（随后的矩阵是投影矩阵）或GL_TEXTURE（随后的矩阵是纹理变换矩阵）



【说明】

- 某一时刻只能处于一种模式。
- 每一种矩阵模式都有一个当前矩阵Cur。
- 默认为GL_MODELVIEW模式。

2. glLoadIdentity()

【函数原型】void glLoadIdentity(void);

【功能】将当前矩阵改为单位矩阵，即Cur=*I*。

【说明】每一种矩阵模式都有一个当前矩阵Cur。

3. glLoadMatrix*()

【函数原型】

- void glLoadMatrixd(const GLdouble *m);
- void glLoadMatrixf(const GLfloat *m);

【功能】将当前矩阵改为一个4×4矩阵M，即Cur=M。

【参数】用m指定该4×4矩阵的16个元素，列主向存储（如图6-2所示）。

$$\begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{pmatrix} \Rightarrow \begin{bmatrix} m_{11} & m_{21} & m_{31} & m_{41} \\ m_{12} & m_{22} & m_{32} & m_{42} \\ m_{13} & m_{23} & m_{33} & m_{43} \\ m_{14} & m_{24} & m_{34} & m_{44} \end{bmatrix}$$

图6-2　列主向存储

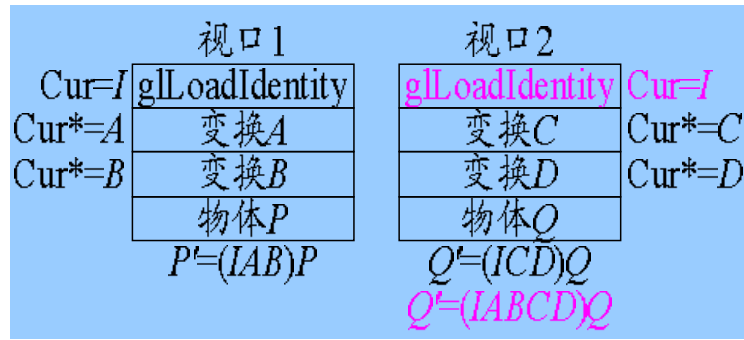
4. glMultMatrix*()

【函数原型】

- void glMultMatrixd(const GLdouble *m);
- void glMultMatrixf(const GLfloat *m);

【功能】将当前矩阵乘以一个4×4矩阵M，即Cur=Cur×M。

【参数】用m指定该4×4矩阵的16个元素，列主向存储。



5. glPushMatrix()，glPopMatrix()

【函数原型】

- void glPushMatrix(void);
- void glPopMatrix(void);

【功能】当前矩阵栈的压入和弹出。

【说明】如图6-3所示。

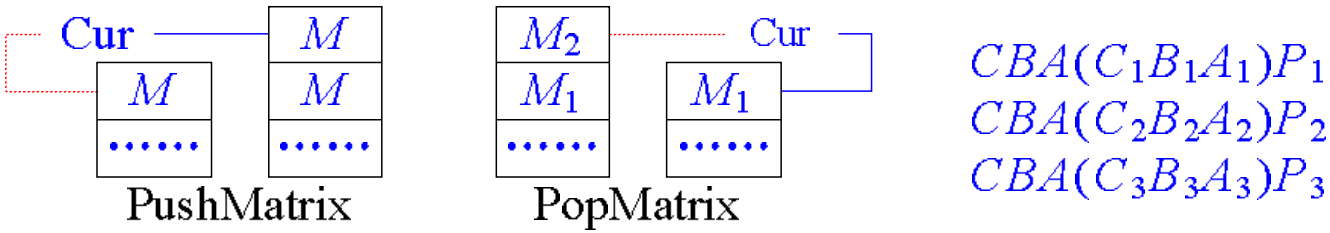


图6-3　当前矩阵栈

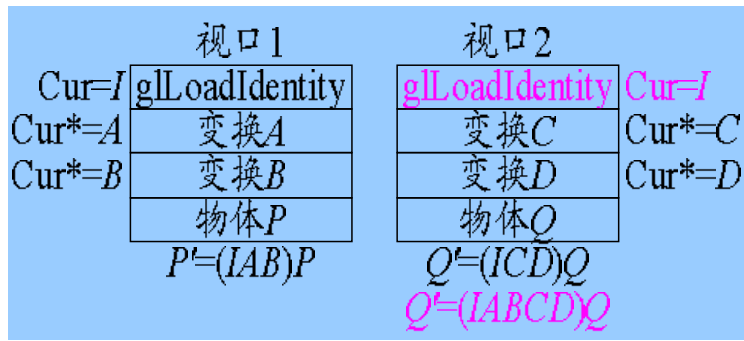
- 每一种矩阵模式都有一个矩阵栈。当前矩阵就是相应栈的栈顶矩阵。
- glPushMatrix()把当前矩阵的副本添加到栈中，使得栈顶矩阵与其下层矩阵相同。
- glPopMatrix()从栈中删除当前矩阵，使其下层矩阵成为当前矩阵。
- 初始时，每个栈都包含一个单位矩阵。

6.2 视图造型变换

6.2.1 变换的顺序

当执行变换A和B时，如果按不同顺序执行，结果往往会大不相同。下列代码在GL_MODELVIEW状态下，相继指定了单位矩阵和矩阵A、B、C。顶点v（使用齐次坐标）的变换顺序为$A(B(C(v)))$，即顶点的实际变换顺序正好与指定顺序相反。

```
glMatrixMode(GL_MODELVIEW); // 后继变换均为视图造型变换
glLoadIdentity(); // 当前矩阵改为单位矩阵
glMultMatrixf(A); // 乘以变换矩阵A
glMultMatrixf(B); // 乘以变换矩阵B
glMultMatrixf(C); // 乘以变换矩阵C
glBegin(GL_POINTS);
glVertex3f(v); // 指定待变换顶点v
glEnd();
```



6.2.2 基本的变换命令

OpenGL有三个基本的变换命令：glTranslate*()（平移）、glRotate*()（旋转）和glScale*()（缩放）。这些命令的作用等价于使用相应的变换矩阵调用glMultMatrix*()，但前者比后者计算要快。例如，如图6-4所示，如果使用glMultMatrix*()，则需要计算$\mathrm{Cur}=\mathrm{Cur}\times S$，共$4\times16$次实数乘法和$3\times16$次实数加法，而使用glScale*()，只需计算$\mathrm{Cur}=M$，共12次实数乘法。

$$\underbrace{\begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{pmatrix}}_{\mathrm{Cur}} \underbrace{\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{S} = \underbrace{\begin{pmatrix} m_{11}s_x & m_{12}s_y & m_{13}s_z & m_{14} \\ m_{21}s_x & m_{22}s_y & m_{23}s_z & m_{24} \\ m_{31}s_x & m_{32}s_y & m_{33}s_z & m_{34} \\ m_{41}s_x & m_{42}s_y & m_{43}s_z & m_{44} \end{pmatrix}}_{M}$$

图6-4 glScale*()的计算量

1. glTranslate*()

【函数原型】

- void glTranslated(GLdouble x, GLdouble y, GLdouble z);
- void glTranslatef(GLfloat x, GLfloat y, GLfloat z);

【功能】将当前矩阵乘以一个平移矩阵 T，即 $\mathrm{Cur} = \mathrm{Cur} \times T$。

【参数】用(x, y, z)指定沿x轴、y轴和z轴的平移量。

2. glRotate*()

【函数原型】

- void glRotated(GLdouble angle, GLdouble x, GLdouble y, GLdouble z);
- void glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z);

【功能】将当前矩阵乘以一个旋转矩阵 R，即 $\mathrm{Cur} = \mathrm{Cur} \times R$。

【参数】

- angle: 指定旋转角度（以度为单位）。
- (x, y, z): 指定旋转轴的方向分量。因为旋转轴通过原点，所以旋转轴就是$(0, 0, 0) \sim (x, y, z)$。

3. glScale*()

【函数原型】

- void glScaled(GLdouble x, GLdouble y, GLdouble z);
- void glScalef(GLfloat x, GLfloat y, GLfloat z);

【功能】将当前矩阵乘以一个缩放矩阵 S ，即 $\mathrm{Cur} = \mathrm{Cur} \times S$ 。

【参数】用(x, y, z)指定沿x轴、y轴和z轴的缩放系数。

6.3 投影变换

投影变换包括透视投影和正投影。

6.3.1 透视投影

1. 定义透视投影的函数

透视投影的观察体是一个棱台（如图6-5所示），在观察体内的物体投影到投影中心。

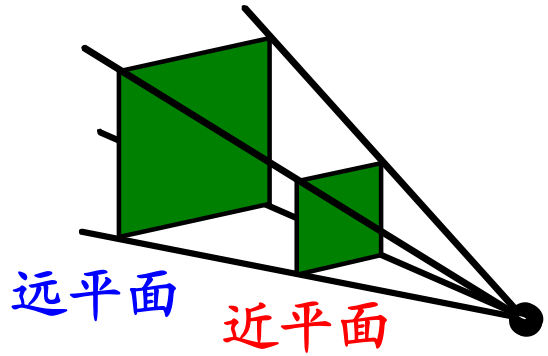


图6-5 观察体

可以使用函数glFrustum()或gluPerspective()定义棱台，计算透视投影矩阵 M，并将当前矩阵 Cur 乘以 M，使 $\mathrm{Cur}=\mathrm{Cur}\times M$。

2. glFrustum()

【函数原型】void glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);

【参数】

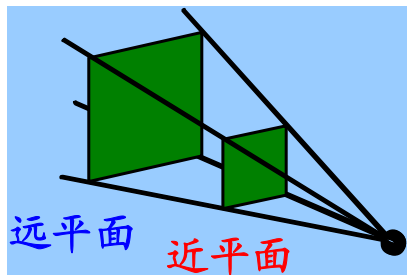
- left, right, bottom, top：裁剪窗口左、右、下、上4个边界的坐标。
- near, far：视点到近、远平面的距离，必须是正的。

【说明】

- 视点位于(0, 0, 0)，棱台范围为(left ~ right, bottom ~ top, -near ~ -far)。
- 只显示棱台范围内的景物（如图6-5所示）。

【变换矩阵】glFrustum()生成的变换矩阵为

$$\begin{pmatrix} \frac{2N}{R-L} & 0 & \frac{R+L}{R-L} & 0 \\ 0 & \frac{2N}{T-B} & \frac{T+B}{T-B} & 0 \\ 0 & 0 & -\frac{F+N}{F-N} & -\frac{2FN}{F-N} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$



3. gluPerspective()

【函数原型】void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble near, GLdouble far);

【功能】该函数定义一个以z轴为中线的四棱台。

【参数】

- fovy：y方向的视场角（0°~180°）。
- aspect：横纵比或宽高比，是长度的比值，不是角度的比值。
- near, far：视点到近、远平面的距离。

【说明】

- 调用gluPerspective()可以理解为使用下列代码段。

```
fovy *= 3.14159265358979323846 / 180;
double top = near * tan(fovy / 2), bottom = -top;
double right = aspect * top, left = -right;
glFrustum(left, right, bottom, top, near, far);
```

- 使用gluPerspective()定义观察体时，其纵横比应该与相应视口的纵横比一致，避免图像扭曲。

6.3.2 正投影

1. 定义正投影的函数

正投影的观察体是一个长方体，使用函数glOrtho()或gluOrtho2D()创建正投影的观察体，计算正投影矩阵M，并将当前矩阵Cur乘以M，使Cur=Cur×M。

2. glOrtho()

【函数原型】void glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);

【参数】

- left, right, bottom, top: 裁剪窗口左、右、下、上4个边界的坐标。
- near, far: 视点到近、远裁剪面的距离。视点之后的距离值为负数值（与观察方向相反）。

【变换矩阵】glOrtho()生成的变换矩阵为

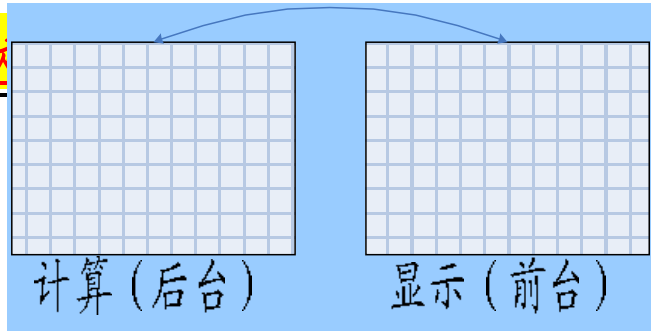
$$\begin{pmatrix} \frac{2}{R-L} & 0 & 0 & -\frac{R+L}{R-L} \\ 0 & \frac{2}{T-B} & 0 & -\frac{T+B}{T-B} \\ 0 & 0 & \frac{-2}{F-N} & -\frac{F+N}{F-N} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3. gluOrtho2D()

【函数原型】void gluOrtho2D(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top);

【功能】建立一个二维正投影观察区域（即二维裁剪窗口），可以理解为glOrtho(left, right, bottom, top, -1, 1)，当然，实际实现并不是这样的。

【注】在完成视图造型变换和投影变换以后，场景中位于观察体以外的部分会被舍弃。



6.4 OpenGL中图形变换的例子

6.4.1 一些需要说明的函数和调用

1. 深度测试的启用和关闭

当场景中出现了一个面片被另一个面片遮挡的情况时，为了确定到底是谁遮挡了谁，需要比较深度大小。在OpenGL中，可以通过启用深度测试完成该任务。

- glEnable(GL_DEPTH_TEST)。用于启用深度测试。
- glDisable(GL_DEPTH_TEST)。用于关闭深度测试。

2. 双缓存的设置与使用

在绘制需要经常变化的画面时，为了提高绘制性能，颜色缓存通常会使用双缓存，分别称为前台缓存和后台缓存。其中，前台缓存用于显示已经完成计算的画面，后台缓存用于计算新画面。新画面计算完成以后，交换两个缓存的地位。这样，显示和计算可以同时进行，提高了绘制性能。

- glutInitDisplayMode(GLUT_DOUBLE)。颜色缓存规定使用双缓存。
- glutSwapBuffers()。用于交换颜色缓存。

3. glutFullScreen()

【函数原型】void glutFullScreen(void);

【功能】使用全屏幕窗口。

4. glutPostRedisplay()

【函数原型】void glutPostRedisplay(void);

【功能】调用当前窗口中已注册的场景绘制函数。

5. glutMouseFunc()

【函数原型】void glutMouseFunc(void (*func)(int button, int state, int x, int y));

【功能】注册鼠标按键回调函数。

【参数】

- func为函数名。
- func的参数button表示哪个鼠标按键。
- func的参数state表示按键状态。
- func的参数(x, y)为鼠标位置（程序窗口坐标）。

6. glutKeyboardFunc()

【函数原型】void glutKeyboardFunc(void (*func)(unsigned char key, int x, int y));

【功能】注册键盘按键响应函数。

【参数】

- func为函数名。
- func的参数key表示哪个按键。
- func的参数(x, y)为鼠标位置（程序窗口坐标）。

7. glutTimerFunc()

【函数原型】void glutTimerFunc(GLuint millis, void (*func)(int value), int value);

【功能】注册定时器回调函数

【参数】

- millis为时间间隔。
- func为回调函数名。
- value为传递给func的参数。

【说明】在func中必须重新调用glutTimerFunc。

6.4.2# 斜投影的实现

1. 变换矩阵

不失一般性，这里只考虑形如$(a,b,1)$的投影向量。易知，当投影面为$z=z_0$时，斜投影变换的变换矩阵为

$$\begin{pmatrix} 1 & 0 & -a & az_0 \\ 0 & 1 & -b & bz_0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

变换方程为

$$\begin{cases} x' = x - az + az_0 \\ y' = y - bz + bz_0 \\ z' = z \end{cases}$$

该变换很难表示成三种基本变换的复合。

$$\begin{pmatrix} 1 & 0 & -\dfrac{p_x}{p_z} & \dfrac{p_x}{p_z}z_p \\ 0 & 1 & -\dfrac{p_y}{p_z} & \dfrac{p_y}{p_z}z_p \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

2. 变换的实现方法

```
void gllOrtho(double L, double R, double B, double T,
              double N, double F, double a, double b)
{   glOrtho(L, R, B, T, N, F); // 正投影变换, 参数仍然是(L, R, B, T, N, F)
    glMultMatrixd((const double[]) // 斜投影变换, 投影向量为(a, b, 1)
    {   1 , 0 , 0, 0,
        0 , 1 , 0, 0,
        -a, -b, 1, 0,
        a * -N, b * -N, 0, 1 // z0 = -N
    });
}
```

这里给出的程序需C99支持。其中，glOrtho()的参数仍然是(L, R, B, T, N, F)，这是因为斜投影保持z坐标不变，且裁剪窗口定义在平面$z=z_0$上，而当$z=z_0$时，有

$$\begin{cases} x' = x - az_0 + az_0 = x \\ y' = y - bz_0 + bz_0 = y \end{cases}$$

$$\begin{cases} x' = x - az + az_0 \\ y' = y - bz + bz_0 \\ z' = z \end{cases}$$

【注】为了方便，将在本章和后续章节中编写的一些通用函数组织在文件gll.h中。下面列出了gll.h的预处理部分。

```
// gll.h
#pragma once
#include<stdio.h> // sprintf等
#include<stdlib.h> // 随机数函数等
#include<string.h> // 字符串函数
#include<stdarg.h> // 可变参数
#include<iso646.h> // not, and, or, etc.
#include<complex.h> // 复数函数, 需C99支持
#include<math.h> // floor, ceil, etc.
#include<tgmath.h> // 通用数学函数, 需C99支持
#include<gl/freeglut.h> // GL, GLU和GLUT函数
```

```
#ifndef Complex // gll.h的复数类型, 需C99支持
typedef double _Complex Complex;
#endif
```

3. 程序示例

下列程序演示了正投影和斜投影的对比效果。在Paint()中，将程序窗口分成2个视口，左边视口显示立方体的正投影效果，右边视口显示同一立方体的斜投影效果。程序运行结果如图6-6所示。

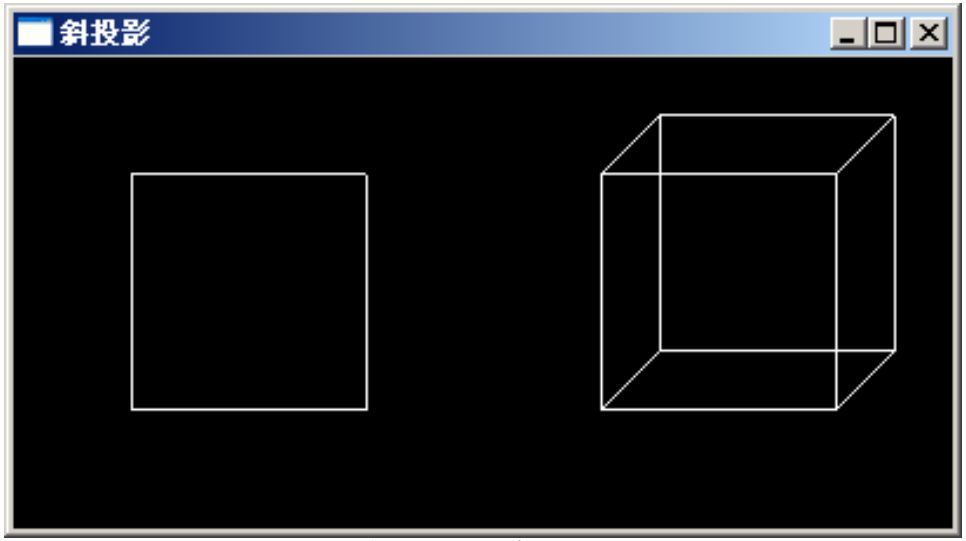


图6-6　斜投影

```
// gllOrtho.c
#include"gll.h"
void Paint() // 场景绘制函数
{   glClear(GL_COLOR_BUFFER_BIT); // 清除颜色缓存
    int w = glutGet(GLUT_WINDOW_WIDTH) / 2; // 视口宽度
    int h = glutGet(GLUT_WINDOW_HEIGHT); // 视口高度
    glViewport(0, 0, w, h); // 左侧视口，正投影
    glLoadIdentity(); // 将当前矩阵改为单位矩阵
    glOrtho(-2, 2, -2, 2, -1, 1); // 正投影
    glutWireCube(2); // 边长为2的立方体
    glViewport(w, 0, w, h); // 右侧视口，斜投影
    glLoadIdentity();
    gllOrtho(-2, 2, -2, 2, -1, 1, 0.25, 0.25); // 斜投影，投影向量(0.25, 0.25, 1)
    glutWireCube(2);
    glFlush();
}
```

```
int main(int argc, char *argv[])
{   glutInit(&argc, argv);
    glutInitWindowSize(400, 200); // 程序窗口大小(4 : 2)
    glutCreateWindow("斜投影");
    glutDisplayFunc(Paint);
    glutMainLoop();
}
```

6.4.3 平移和旋转的切换

下列程序用鼠标左键进行平移和旋转的切换。在Reshape()中调用glTranslatef()把物体平移到观察体中，在Paint()中调用平移和旋转变换。使用全屏窗口，按Esc键结束程序。运行时某一时刻的画面如图6-7所示。

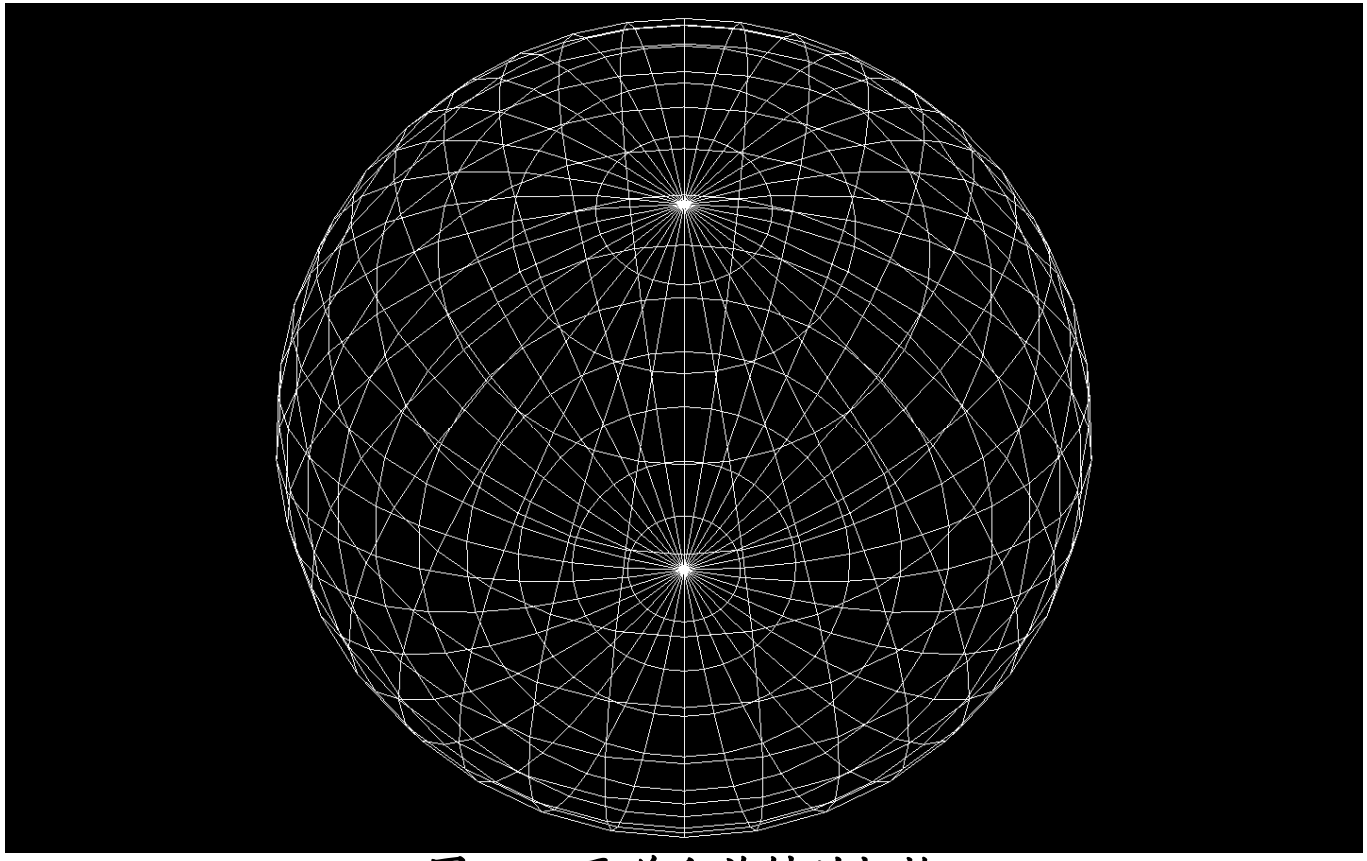


图6-7 平移和旋转的切换

（1）预处理与共享变量。

```
// Transform.c
#include<gl/freeglut.h>
#include<iso646.h> // not, and, or, etc.
```

```
int sign = 1; // 旋转还是平移的标识
int rot = 0; // 旋转角度或平移距离
```

（2）键盘、鼠标与窗口变化的响应。

```
void Mouse(int but, int state, int x, int y) // 鼠标响应函数
{   // 鼠标按键, 按键状态, 光标位置(像素)
    if(but == GLUT_LEFT_BUTTON and state == GLUT_DOWN)
        sign = not sign, rot = 0; // 切换旋转和平移, 恢复旋转角度
}
```

```
void Keyboard(unsigned char key, int x, int y) // 键盘响应函数
{   if(key == 27) exit(0); // 按下Esc键退出
}
```

```
void Reshape(int w, int h) // 窗口变化响应函数
{   glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION); // 当前变换类型为投影变换
    glLoadIdentity(); // 将当前矩阵改为单位矩阵
    gluPerspective(30, (float) w / h, 1, 1000); // 透视投影变换
    glTranslatef(0, 0, -180); // 远移180
    glMatrixMode(GL_MODELVIEW); // 当前变换类型为视图造型变换
    glLoadIdentity();
}
```

（3）场景绘制。

```
void Paint() // 场景绘制函数
{   glClear(GL_COLOR_BUFFER_BIT); // 清除颜色缓存
    glLoadIdentity(); // 消除以前变换的影响
    if(sign) glRotatef(rot, 1, 0, 0); // 绕x轴旋转rot
    else glTranslatef(0, 0, -rot); // 远移rot
    glRotatef(-90, 1, 0, 0); // 将高度方向从z方向调整为y方向
    glutWireSphere(45, 36, 18); // 线框球(半径, 经线数, 纬线数)
    glutSwapBuffers(); // 交换颜色缓存
}
```

（4）场景刷新。

```
void Timer(int millis) // 定时器响应函数
{   rot = (rot + 2) % 360; // 修改旋转角度或平移距离
    glutPostRedisplay(); // 调用场景绘制函数
    glutTimerFunc(millis, Timer, millis); /* 指定定时器响应函数(间隔毫秒数,
函数名, 函数参数值) */
}
```

（5）窗口初始化与回调函数注册。

```
int main(int argc, char *argv[])
{   glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE); // 双缓存
    glutCreateWindow("平移和旋转的切换");
    // 当前窗口初始化
    glutFullScreen(); // 使用全屏幕窗口
    // 注册回调函数
    glutDisplayFunc(Paint); // 指定场景绘制函数
    glutReshapeFunc(Reshape); // 指定窗口变化响应函数
    glutMouseFunc(Mouse); // 指定鼠标响应函数
    glutKeyboardFunc(Keyboard); // 指定键盘响应函数
    glutTimerFunc(30, Timer, 30); /* 指定定时器响应函数(间隔毫秒数, 函数
名, 函数参数值) */
    glutMainLoop();
}
```

6.4.4 一个简单的日地月系统※

下列程序演示了日地月的公转和自转等效果。为了增强演示效果，还绘制了日地月的经纬线，并在赤道上放置了一个小球。运行时某一时刻的画面如图6-8所示。

在Paint()中，首先定义太阳，然后规定地球的公转角度和放置位置，考虑地球的黄赤角以保证地轴方向，规定自转角度并定义地球，最后规定月球的公转角度（自转角度不需要指定）和放置位置，并定义月球。

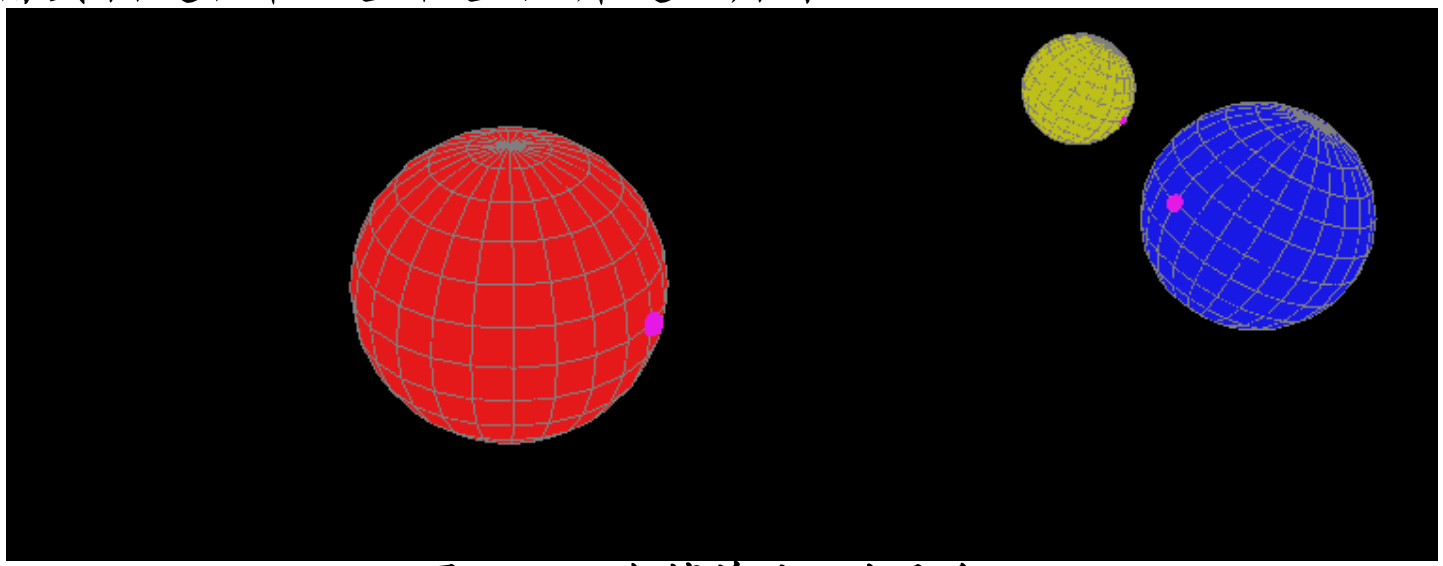


图6-8 一个简单的日地月系统

（1）预处理与共享变量。

```
// PlanetarySimple.c
#include<math.h> // fmod()等
#include<gl/freeglut.h>
```

```
float day = 0; // 地球自转角度
float month = 0; // 月亮公转角度(或自转角度, 两者相同)
float year = 0; // 地球公转角度
float sun = 0; // 太阳自转角度
```

（2）按键与窗口变化的响应。

```
void Keyboard(unsigned char key, int x, int y) // 键盘按键响应函数
{   if(key == 27) exit(0); // 按下Esc键退出
}
```

```
void Reshape(int w, int h) // 窗口变化回调函数
{   glViewport(0, 0, w, h); // 视口的位置和大小
    glMatrixMode(GL_PROJECTION); // 当前变换类型为投影变换
    glLoadIdentity(); // 当前矩阵为单位矩阵，不再有其他投影变换
    gluPerspective(30, (float)w / h, 1, 1000); /* 透视变换(y向张角，宽高比，近平面，远平面) */
    glTranslatef(0, 0, -8); // 适当远移
    glRotatef(30, 1, 0, 0); // 调整观察角度
    glMatrixMode(GL_MODELVIEW); // 当前变换类型为视图造型变换
    glLoadIdentity(); // 当前矩阵为单位矩阵，不再有其他造型变换
}
```

（3）日地月定义。

```
void Wire(float r) // 经纬线
{   glColor3f(0.5, 0.5, 0.5); // 经纬线是灰色的
    glutWireSphere(r * 1.005, 24, 12); // 经纬线将星球包裹在内
}
```

```
void Ball(float r) // 在赤道上放置一个小球验证自转
{   glPushMatrix(); // 保存当前矩阵
    glTranslatef(-r, 0, 0); // 在赤道与负x轴交点处放置一个小球
    glColor3f(0.9, 0.1, 0.9); // 小球是紫色的
    glutSolidSphere(r * 0.08, 12, 6); // 小球
    glPopMatrix(); // 恢复当前矩阵，避免上述变换影响到其他物体
}
```

```
void Sun(float r) // 太阳
{   glPushMatrix(); // 保存当前矩阵
    glRotatef(90, -1, 0, 0); // 将物体的高度方向从z调整到y
    glColor3f(0.9, 0.1, 0.1); // 太阳是红色的
    glutSolidSphere(r, 24, 12); // 太阳
    Wire(r); // 经纬线
    Ball(r); // 在赤道上放置一个小球验证自转
    glPopMatrix(); // 恢复当前矩阵, 避免上述变换影响到其他物体
}
```

```
void Planet(float r) // 地球
{   glPushMatrix(); // 保存当前矩阵
    glRotatef(90, -1, 0, 0); // 将物体的高度方向从z调整到y
    glColor3f(0.1, 0.1, 0.9); // 地球是蓝色的
    glutSolidSphere(r, 24, 12); // 地球
    Wire(r); // 经纬线
    Ball(r); // 在赤道上放置一个小球验证自转
    glPopMatrix(); // 恢复当前矩阵, 避免上述变换影响到其他物体
}
```

```
void Moon(float r) // 月亮
{   glPushMatrix(); // 保存当前矩阵
    glRotatef(90, -1, 0, 0); // 将物体的高度方向从z调整到y
    glColor3f(0.75, 0.75, 0.1); // 月亮是浅黄色的
    glutSolidSphere(r, 24, 12); // 月亮
    Wire(r); // 经纬线
    Ball(r); // 在赤道上放置一个小球验证自转
    glPopMatrix(); // 恢复当前矩阵, 避免上述变换影响到其他物体
}
```

（4）场景绘制。

```
void Paint() // 场景绘制函数
{   glClear(GL_COLOR_BUFFER_BIT); // 清除颜色缓存
    glClear(GL_DEPTH_BUFFER_BIT); // 清除深度缓存
    glLoadIdentity(); // 不再有其他造型变换
    // 太阳: 自转
    {   glPushMatrix();
        glRotatef(sun, 0, 1, 0); // 太阳自转
        Sun(0.4); // 太阳
        glPopMatrix(); // 取消太阳自转的影响
    }
```

```
    // 地球: 自转-->黄赤角-->保证地轴方向-->位置-->公转
    glRotatef(year, 0, 1, 0); // (5) 地球公转
    glTranslatef(2, 0, 0); // (4) 地球相对太阳的初始位置
    glRotatef(-year, 0, 1, 0); // (3) 取消公转的影响, 保证地轴方向不变
    glRotatef(-23.5, 0, 0, 1); // (2) 黄赤交角, 公转平面与自转平面的夹角
    {   glPushMatrix();
        glRotatef(day, 0, 1, 0); // (1) 地球自转
        Planet(0.3); // 地球
        glPopMatrix(); // 取消地球自转的影响
    }
    // 月亮: 自转-->位置-->公转-->地球黄赤角--> ...
    glRotatef(month, 0, 1, 0); // (2) 月亮公转
    glTranslatef(0.6, 0, 0); // (1) 月亮相对地球的初始位置
    // 月亮自转周期与公转周期相同, 不需要指定
    Moon(0.15); // 月亮
    glutSwapBuffers(); // 交换颜色缓存
}
```

（5）场景刷新。

```
void Timer(int millis) // 定时器回调函数(场景刷新)
{   float base = 36; // 基准的角度增量(对应0.1天/帧)
    float dday = base * 0.1; // 地球自转角度的增量(较慢, 0.1倍正常速度)
    float dmonth = base / 27.3217; // 月亮公转角度的增量(恒星月)
    float dyear = base / 365.2564; // 地球公转角度的增量(恒星年)
    float dsun = base / 25.38; // 太阳自转角度的增量(恒星周期)
    day = fmod(day + dday, 360); // 计算地球自转角度(较慢)
    month = fmod(month + dmonth, 360); // 计算月亮公转角度
    year = fmod(year + dyear, 360); // 计算地球公转角度
    sun = fmod(sun + dsun, 360); // 计算太阳自转角度
    glutPostRedisplay(); // 调用场景绘制函数
    glutTimerFunc(millis, Timer, millis); /* 指定定时器函数(间隔毫秒数, 函数
名, 函数参数值) */
}
```

（6）窗口初始化与回调函数注册。

```
int main(int argc, char *argv[])
{   glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE); // 双缓存
    glutCreateWindow("一个简单的日地月系统");
    // 当前窗口初始化
    glutFullScreen(); // 使用全屏幕窗口
    glEnable(GL_DEPTH_TEST); // 打开深度测试, 用于比较远近
    // 注册回调函数
    glutDisplayFunc(Paint); // 指定场景绘制函数
    glutReshapeFunc(Reshape); // 指定窗口变化响应函数
    glutKeyboardFunc(Keyboard); // 指定键盘按键响应函数
    glutTimerFunc(30, Timer, 30); /* 指定定时器函数(间隔毫秒数, 函数名,
函数参数值) */
    glutMainLoop();
}
```

6.5 练习题

6.5.1 程序设计题

1. 请使用OpenGL、GLU和GLUT编写一个显示线框立方体的程序。其中立方体的半径为1.5单位，并首先绕(0, 0, 0)~(1, 1, 0)旋转30度，然后远移6.5单位；观察体规定为：视场角=30度，宽高比=1，近=1，远=100；程序窗口的大小为(200, 200)，标题为“线框立方体”。

2. 请使用OpenGL和GLUT编写一个显示线框球体的简单图形程序。其中球体的半径为0.8，经线数为24，纬线数为12，并绕 x 轴旋转30度，程序窗口的大小为(200, 200)，标题为“线框球”。

3. 请使用OpenGL和GLUT编写一个显示线框椭球体的简单图形程序。其中椭球体的两极方向为上下方向，左右方向的半径为0.98，上下方向的半径为0.49，前后方向的半径为0.6，经线数为48，纬线数为24，使用正投影，裁剪窗口为(-1, -0.5)~(1, 0.5)，程序窗口的大小为(400, 200)，标题为“线框椭球”。

4. 请使用OpenGL、GLU和GLUT编写一个三维犹他茶壶程序。其中茶壶的半径为1单位，并远移6.5单位；观察体规定为：视场角=30度，宽高比=1，近=1，远=100；程序窗口的大小为(200, 200)，标题为“旋转的尤他茶壶”。茶壶绕z方向中轴不断旋转，旋转的时间间隔为25毫秒，角度间隔为2度。注意旋转角度必须限定在0~360度以内。

5. 请使用OpenGL、GLU和GLUT编写一个简单的多视口演示程序。要求：(1)在屏幕窗口左下角的1/4部分显示一个红色的填充正三角形；(2)在屏幕窗口右上角的1/4部分显示一个绿色的填充正方形；(3)三角形和正方形的左下角顶点坐标值均为(0, 0)，右下角顶点坐标值均为(1, 0)；(4)裁剪窗口均为(-0.1, -0.1)~(1.1, 1.1)；(5)程序窗口的大小为(200, 200)，标题为“多视口演示”。

6. 请使用OpenGL、GLU和GLUT编写一个多视口演示程序。要求：(1)在屏幕窗口左下角的1/4部分显示一个红色的填充矩形，该矩形的一对对角顶点是(0, 0)和(1, 1)；(2)在屏幕窗口右下角的1/4部分显示一个绿色的填充犹他茶壶，茶壶半径为0.4，并向右向上各移0.5；(3)在屏幕窗口上部居中的1/4部分显示一个蓝色的填充正三角形，该正三角形的左下角顶点是(0, 0)，右下角顶点是(1, 0)；(4)裁剪窗口均为(-0.1, -0.1)~(1.1, 1.1)，程序窗口的大小为(200, 200)，背景为黑色，标题为“多视口演示”。

6.5.2 阶段实习题

演示一个不断旋转、缩放和移动的正三棱锥。要求正三棱锥4个面的颜色各不相同。可以分成三个小题完成。