第一章 OpenGL 概述

1．1 OpenGL 入门

1．1．1 什么是 OpenGL

OpenGL 的前身是由 SGI 公司为其图形工作站开发的 IRIS GL，是一个工业标准的三维计算机图形软件接口。但是 SGI 公司向其他平台移植时，遇到了问题，为改进其移植性， 开发了 OpenGL，它有 GL 的功能，而且是开放的，适用于多种硬件平台及操作系统，用户可以方便地利用这个图形库，创建出接近光线跟踪的高质量静止或动画的三维彩色图像，而且要比光线跟踪算法快一个数量级。它的主要特点是：

①OpenGL 可以在网络上工作，即客户机/服务器型，显示图形的计算机(客户机)可以不是运行图形程序的计算机(服务器)，客户机与服务器可以是不同类型的计算机，只要两者服从相同的协议。

②OpenGL 是与硬件无关的软件接口，可以在多种硬件平台上运行，使得 OpenGL 的应用程序有较好的移植性。

1992 年 7 月，SGI 发布了 OpenGL 的 1．0 版本，后来 SGI 与微软共同开发了 Windows NT 下的新版本。1995 年 12 月，由 OpenGL ARB(Architecture Review Board)批准了 OpenGL 1．1 版本，这一版本的 OpenGL 性能得到了加强并引入了一些新功能，其中包括：在增强元文件中包含 OpenGL 调用，改进打印机支持，顶点数组的新特征，提高顶点位置，法线，颜色及色彩指数，纹理坐标，多边形边缘标识的传输速度，引入新的纹理特性。

Microsoft 开始把 OpenGL 集成到 Windows NT 中，后来又把它集成到新版本的 Windows 95 OEM Service Release 2(简称为 OSR2)及以后的版本中，用户既可以在 Windows 95/98、Windows NT/2000/XP 环境下开发 OpenGL 应用程序，又可以很方便地把已有的工作站上的程序移植过来。

1．1．2 OpenGL 的工作顺序

本节介绍从定义几何要素到把像素段写入帧缓冲区的过程，即 OpenGL 的工作顺序。在屏幕上显示图像的主要步骤是：

①构造几何要素(点、线、多边形、图像、位图)，创建对象的数学描述。

②在三维空间上放置对象，选择有利的场景观察点。

③计算对象的颜色，这些颜色可以直接定义，也可由光照条件及纹理间接给出。

④光栅化，把对象的数学描述和颜色信息转换到屏幕的像素。另外，也可执行消隐以及对像素的操作，如图 1—1 所示。

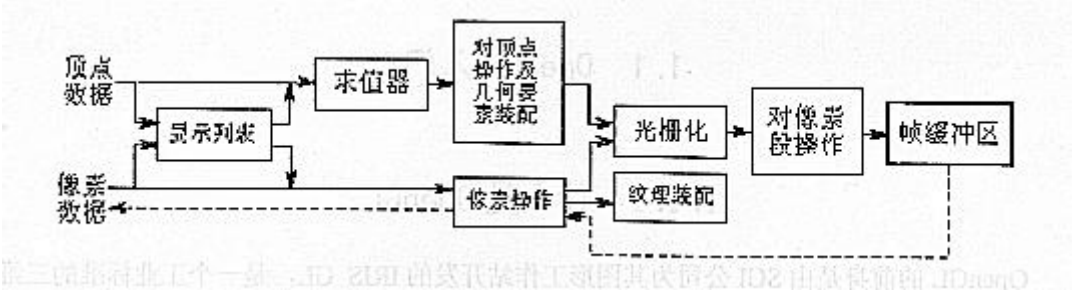

图 1—1 OpenGL 的操作顺序

(1)几何操作

1)针对每个顶点的操作

每个顶点的空间坐标经模型取景矩阵变换，法向矢量由逆矩阵变换，若允许纹理自动生成，则由变换后的顶点坐标生成新的纹理坐标，替代原有的纹理坐标，经过当前纹理矩阵变换，传递到几何要素装配步骤。

2)几何要素装配

根据几何要素类型的不同，几何要素装配也不同。若使用平直明暗处理，线或多边形的所有顶点颜色相同，若使用裁剪平面，裁剪这些几何要素，此后每个顶点的空间坐标由投影矩阵变换，由标准取景平面裁剪 x=±w，y=±w，z=±w。若使用选择模式，没被裁剪掉的几何要素生成一个选中报表，否则投影矩阵除以 w，做视见区和深度范围操作，若几何要素是多边形，还要做剔除检验。最后根据点图案、线宽、点尺寸等生成像素段，并给其赋上颜色、深度值。

(2) 像素操作

由主机读入的像素首先解压缩成适当的组份数目，然后将数据放大、偏置并经过像素映射处理，根据数据类型限制在适当的取值范围内，最后写入纹理内存，在纹理映射中使用或光栅化成像素段。

若由帧缓冲区读入像素数据，则执行像素传输操作(放大、偏置、映射、调整)，结果以适当的格式压缩并返回给处理器内存。

像素拷贝操作相当于解压缩和传输操作的组合，只是压缩和解压缩不是必需的，数据写入帧缓冲区前的传输操作只有一次。

(3)像素段操作

若使用纹理化，每一个像素段由纹理内存产生纹素，如果还允许下面的操作，将做雾化效果计算、反走样处理。其后进行裁剪处理、α 检验(只在 RGBA 模式下使用)、模板检验、深度缓冲区检验、抖动处理，若在索引模式下，对指定的值进行逻辑操作，若在 RGBA 模式下则进行混合操作。

根据 OpenGL 所处的模式不同,由颜色或颜色索引屏蔽这个像素段,写入适当的帧缓冲区，若写入模板或深度缓冲区，在模板和深度检验后进行屏蔽，结果写入帧缓冲区而不做混合、抖动或逻辑操作。

1.2 一个简单的 OpenGL 程序

OpenGL 是一个功能强大的图形库，用户可以很方便地开发所需要的有多种特殊视觉效果的三维图形。作为简单的入门介绍，本节将结合一个简单程序，介绍与 OpenGL 相关的库函数、OpenGL 的语言规则、OpenGL 系统的状态。通过这个简单的程序，可以看到，OpenGL 程序的基本结构有两部分：初始化 OpenGL 绘图的状态和描述要绘制的物体。这样，读者可以初步掌握 OpenGL 程序设计的方法，设计功能相对简单、完整的应用程序。

1．2．1 一个简单的程序

下面给出一个绘制圆球线框图的完整程序(程序 1—1)，尽管这个程序比较简单，但是具有一个完整的程序结构，如果在这个程序的基础上，加以修改，可以实现一些简单的功能，如绘制二维、三维曲线等。

程序 1—1

```
/* Prog1_1.c */
```

```
#define rad 3.14159265/180.

#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
#include <math.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

/*  Clear the screen.  Set the current color to white.
 *  Draw the wire frame sphere.
 */
void CALLBACK display(void)
{
     int i,j;
     float x,y,z,r;
     glClearColor(0.5,0.5,0.5,1.0);
     glClear(GL_COLOR_BUFFER_BIT);
     glColor3f(1.0,  1.0,  0.0);
     glLoadIdentity();   /*  clear the matrix    */

     glTranslatef(0.0,  0.0,  -5.0);  /*  viewing transformation  */
     glRotatef(45.0, -45.0, 0.0, 1.0);
     glScalef(1.0,  1.0,  1.0);   /*  modeling transformation */
     for (i=0; i<180; i+=5)
     {
          glBegin(GL_LINE_LOOP);
               r=2.*sin(i*rad);
               z=2.*cos(i*rad);
               for (j=0;j<360; j+=5)
               {
                    x=r*cos(j*rad);
                    y=r*sin(j*rad);
                    glVertex3f(x,y,z);
               }
          glEnd();
     }
     for (j=0;j<360;j+=5)
     {
          glBegin(GL_LINE_LOOP);
               for  (i=0;i<=180;i+=5)
```

```
            {
                r=2.*sin(i*rad);
                z=2.*cos(i*rad);
                x=r*cos(j*rad);
                y=r*sin(j*rad);
                glVertex3f(x,y,z);
            }
        glEnd();
    }
    glFlush();
}

void myinit(void)
{
    glShadeModel (GL_FLAT);
}

/*  Called when the window is first opened and whenever
 *  the window is reconfigured (moved or resized).
 */
void CALLBACK myReshape(GLsizei w,  GLsizei h)
{

    glMatrixMode (GL_PROJECTION);   /*  prepare for and then  */
    glLoadIdentity ();  /* define the projection  */
    glFrustum (-1.0,  1.0,  -1.0, 1.0, 1.5, 20.0);    /*  transformation  */
    glMatrixMode(GL_MODELVIEW);     /*  back to modelview matrix    */
    glViewport  (0,  0, w, h);     /*  define the viewport */
}
/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, and handle input events.
 */
int main(int argc,  char** argv)
{
    auxInitDisplayMode(AUX_SINGLE | AUX_RGB | AUX_DEPTH) ;
    auxInitPosition(0,  0, 400, 400);
    auxInitWindow("wireframe of a sphere");
    myinit();
    auxReshapeFunc(myReshape);
    auxMainLoop(display);
    return(0);
}
```

其中主程序的功能是分别用三个前缀为“aux”的函数设置 OpenGL 窗口的显示模式，打

开窗口的左下角坐标为(0，0)，右上角坐标为(400，400)，auxInitWindow 函数给出窗口的标题，myinit 子程序设定 OpenGL 的明暗处理方式，auxReshapeFunc 函数在窗口移动、变形后，调用 myReshape 子程序重新计算模型、取景变换及投影变换，auxMainLoop 循环调用 display 子程序，设定颜色、投影变换、描述几何要素，display 子程序首先设置 OpenGL 窗口的背景及绘图颜色，然后对物体、场景进行旋转、缩放、平移操作。有关函数的使用请参见本书的第三章。glBegin 与 glEnd 之间的语句用于由给定的顶点绘制圆球的经线和纬线，其中 x，y，z 为顶点坐标，由 glVertex 函数定义所描述几何要素的顶点，在本书第 2．2 节将介绍几何要素的使用，上述前缀为“aux”的函数是 OpenGL 的辅助库函数，有关函数的使用及说明请参见本书的第 1.3.2 节。图 1-2 是这个程序运行的结果，由此不难看出，在 OpenGL 中绘制三维物体与绘制二维物体复杂程度基本相当，OpenGL 的图形显示功能是强大的。实现这些功能也很方便。另外，这个例子显示的圆球只是一个线框图，并不是三维实体，如果不对其做适当的旋转变换，看不出来是三维物体，即使在一个特定角度上进行观察，又因为没做消隐，立体感也不强，即需要解决的问题是做消隐操作和构造曲面，这些内容将在后面的章节详细介绍。值得注意的是，与 OpenGL 辅助函数库中的 auxWireSphere 函数相似，另一个函数 auxSolidSphere 可以绘制一个实体圆球。除此之外，辅助库中还有很多绘制其它三维物体线框图、实体图的子程序可供调用，具体内容请参见下一节。

运行 OpenGL 的软件及硬件需要是：

操作系统：Windows95(OEMServiceRelease2)/98 或 WindowNT4．0/2000 及以后版本；

软件开发环境：MicrosoftVisualC++4．0 及以上版本；

硬件：奔腾级微机，最好配有支持 OpenGL 硬件加速的图形卡。　　．

windows.h 支持微软 Windows 95/98 或 Windows NT/2000 环境下使用 openGL 的头文件，gl.h 是 openGL 核心函数的头文件，glaux.h 是辅助库函数的头文件，对于所有的 OpenGL 应用程序，这 3 个头文件是必需的。在创建执行文件时，要另外连接 opengl32.1ib，glu32.1ib，glaux.1ib 3 个函数库。运行已创建的执行文件时，在 windows 操作系统的 system 或 system32 目录下要有 opengl32.Dll 和 glu32.dll 两个动态连接库。

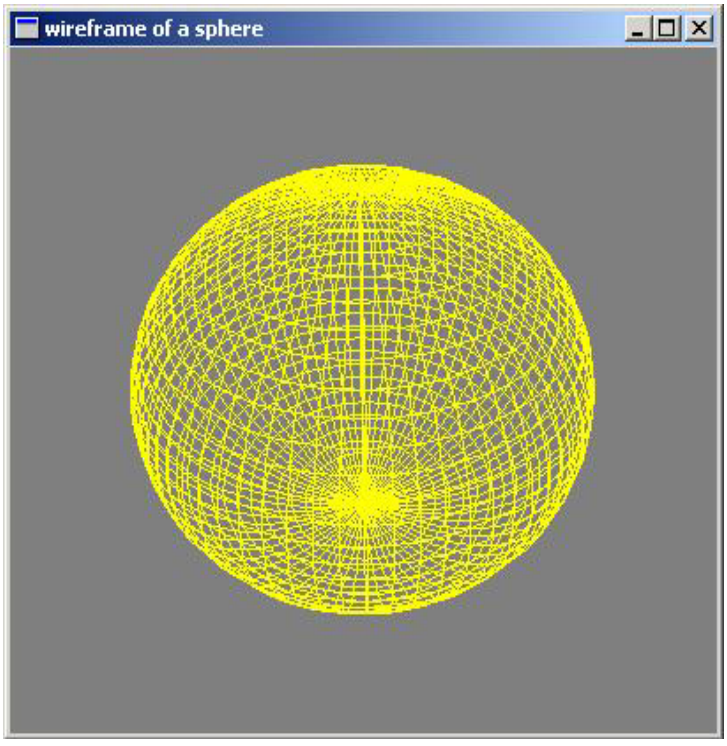

图 1—2　绘制一个线框图

1．2．2 OpenGL 的语法规则

由上一节的程序可以看出，OpenGL 核心函数命令的前缀为“gl”，组成命令的首字母大写，如 glColor3f，其常量是以 GL_开头，均用大写字母，用下划线把关键词分开，如 GL_LINE_LOOP。有些函数如 glVertex3f，为定义几何要素顶点的函数，“3”表示有 3 个参量，“f”说明类型为浮点，由于顶点的参量数不同(2 个或 3 个)，数据类型不同(整数、浮点或双精度)，参见表 1—1。

表 1—1　OpenGL 命令后缀及参量数据类型

后缀的数据类型	C 语言类型	OpenGL 类型定义
b　8 位整数	signed char	Glbyte
s　16 位整数	short	Glshort
l　32 位整数	long	GlinLt, Glsizei
f　32 位浮点数	float	GLfloat,Glclampf
d　64 位浮点数	Double	GLdouble，Glclampd
ub　8 位整数	Unsigned char	GLubyte，Glboolean
us　16 位整数	Unsigned short	Glushort
ui　32 位整数	Unsigned long	GLuint,GLenum,GLbitfield

glVertex 函数有多种形式，适用于具体的情况，有时会出现在上述命令后加上“v”的形式，如 glVertexfv3，其后缀“v”表示其参量是一个矢量或矩阵的指针，“3”表示含有 3 个浮点数的矢量的指针。对于以上提到的函数的多种形式，均可以 glVertex*表示，而 glVertex2{sifd}，则表示 glVextex2s，glVertex2i，glVertex2f，glVertex2d。对函数命名的这个约定在本书中是通用的。

1．2．3 OpenGL 的当前状态

OpenGL 绘图方式是由一系列状态决定的，例如当前绘图的颜色就是 OpenGL 的一个状态，当选定颜色后，OpenGL 就用这个颜色绘图，当然还有许多其他的状态，包括取景变换、投影变换、线及多边形的填充图案、绘制多边形的模式、光源的位置特性、物体的材料、纹理映射等状态，由 glEnable 和 glDisable 函数控制这些状态的打开和关闭。

每个状态变量或模式都有缺省值，根据数据类型的不同，可用 glGetBooleanv，glGetDouble，glGetFloatv，glGetIntegerv 四个函数之一查询状态的当前值，可以查询的状态变量值见本书的第三部分中 glGet 函数的说明。

GLboolean glIsEnable(GLenum cap) 函数的功能是：

若 cap 是允许使用的模式，则函数返回值是 GL_TRUE，否则为 GL_ FALSE，也有一些变量用特定的命令查询，如 glGetLight*，glGetError，glGetPolygonStipple 等。

若 OpenGL 出现错误，则调用 glGetError 函数返回当前错误代码，忽略引起错误的命令，而不影响 OpenGL 状态或帧缓冲区内容，表 1-2 列出了 OpenGL 出错代码。glGetError 函数的具体使用方法，请查阅本书第三部分。

表 1—2 OpenGL 出错代码

出错代码	解　　释
GL_INVALID_ENUM	GLenum 参量超出范围
GL_INVALID_VALUE	数值参量超出范围
GL_INVALID OPERATION	当前状态的操作非法
GL_INVALID_OVERFLOW	命令使堆栈上溢
GL_INVALID_UNDERFLOW	命令使堆栈下溢

Void glPushAttrib(GLboolean mask)函数把由 mask 标识的属性压入属性堆栈，具体值见表 1-3。这些值可以用逻辑“或”组合起来使用，调用 glPopAttrib 函数则恢复所有状态变量。

表 1—3　属性组

mask 字位	属性组
GL_ACCM_BUFFER_BIT	累加缓冲区
GL_ALL_ATIRIB_BITS	——
GL_COLOR_BUFFER_BIT	颜色缓冲区
GL_CURRENT_BIT	当前值
GL_DEPTH_BUFFER_BIT	深度缓冲区
GL_ENABLE_BIT	允许使用值
GL_EVAL_BIT	求值
GL_FOG_BIT	雾
GL_HINT_BIT	隐含值
GL_LIGHTING_BIT	光照
GL_LINE_BIT	线
GL_LIST_BIT	列表
GL_PIXEL_MODE	像素
GL_POINT_BIT	点
GL_POLYGON_BIT	多边形
GL_POLYGONSTIPPLE_BIT	多边形填充
GL_SCISSOR_BIT	裁剪
GL_STENCIL_BUFFER_BIT	模板缓冲区
GL_TEXTURE_BIT	纹理
GL_TRANSFORM_BIT	变换
GL_VIEWPORT_BIT	视见区

1．3　OpenGL 程序设计的预备知识

1．3．1　与 OponGL 相关的库函数

尽管 OpenGL 的核心函数(前缀为“gl”的函数)功能很强大，但只提供了最基本绘图的命令。需要把这些基本命令组织起来才能完成高层次的图形，因此需要建立在 OpenGL 核心函数

基础上的库函数，简化编程工作，提高工作效率。另外 OpenGL 是一个图形软件接口，由于OpenGL 是与硬件、窗口系统无关的，但是绘图工作最终是要由具体的硬件、窗口系统来完成的，需要有专门的子程序完成窗口管理，处理输入事件等任务。如果这些工作均由用户完成，将是十分复杂、无法想象的，为此，系统提供了与 OpenGL 相关的库函数，完成上面提到的工作，这些库函数是：

①OpenGL 应用程序库，前缀为“glu”，其功能为提供设置特定的取景方向，投影矩阵，多边形镶嵌，绘制曲面等，具体内容请参见本书第三部分，查阅 GLU 各函数的功能、用法。

②OpenGL 对 Windows 95/98，WindowsNT/2000 系统的扩展，包括 WGL 函数(前缀为“wgl”)和 Win32 函数，WGL 函数把 OpenGL 与 Windows 95/98，WindowsNT/2000 窗口系统联接起来，管理绘图描述表，显示列表，执行函数和字体位图，Win32 函数支持窗口的像素格式和双缓冲，仅适用于 OpenGL 的图形窗口。具体内容请参见本书第三部分有关 Windows95/98，WindowsNT/2000 库函数的说明。

⑧OpenGL 编程辅助库函数，前缀为“aux”，使编程更简明、更完整，具体内容在下一节讲述。

④OpenInventor 是基于 OpenGL 面向对象的工具包，提供创建交互式三维图形应用程序的对象和方法，提供了预定义的对象和用于交互的事件处理模块，创建和编辑三维场景的高级应用程序单元，有打印对象和与其他图形格式交换数据的能力。

⑤OpenGL 对 X-window 系统的扩展，前缀为“glx”，提供创建 OpenGL 命令描述表并使之与 X-window 系统连接起来的能力，是针对 X-window 的库函数，与 WGL 函数相似，不在本书的讨论范围内。

1．3．2　使用 aux 库

因为 OpenGL 是与窗口系统或操作系统无关的，也自然不会包含打开窗口和从键盘或鼠标读入事件的命令，然而一个完整的图形程序缺少上面提到的基本功能是难以想象的，Windows95/98，WindowsNT/2000 的 OpenGL 实现提供了一个辅助函数库 aux，用于解决了开窗口、处理输入事件等问题。

另外，OpenGL 的绘图命令只能生成简单的几何形状(点、线、多边形)，辅助函数库中有创建复杂三维几何形状(球、环面、茶壶等)的子程序，大大简化了程序设计。

下面介绍辅助函数库 aux 的主要功能。

(1)管理窗口

打开窗口前，要为其指定特性：是单缓区，还是双缓冲，是用 RGBA 值还是色彩索引存贮颜色等，这时要首先调用 auxInitDisplayMode 函数。

1)void auxInitDisplayMode(GLbitfield mask)函数除了定义颜色及缓冲区特性外，也可以指定有深度、模板、累加缓冲区的窗口，mask 是 AUX_RGBA 或 AUX_INDEX 与 AUX_SINGLE 或 AUX_DOUBLE 及与缓冲区允许标识 AUX_DEPTH，AUX_STENCIL 或 AUX_ACCUM 的逻辑或组合，缺省值是 AUX_SINGLE｜AUX_INDEX。即色彩索引模式、单缓冲区的窗口，若定义一个双缓冲，RGBA 模式，有深度和模板缓冲区的窗口，则组合为 AUX_DOUBLE｜AUX_RGBA｜AUX_DEPTH｜AUX_STENCIL。把定义了特性的窗口放在屏幕哪个位置，窗口的大小如何，则由 auxInitPosition 定义。

2) void auxInitPosition(GLint x，GLint y，Glsizei width，Glsizei height) 函数： x，y 为窗口左下角的坐标，width，height 分别为窗口的宽和高， 缺省值为(x，y)=(0，0)，width×height=100×100。

完成上述两个函数的调用，则用 auxInitWindow 函数开窗口，窗口的各种特性均由上面的两个函数给定。

3) void auxInitWindow(GLbyte *) 函数：窗口的标题为字符串 titleString，窗口把 ESC 键与退出函数联系起来，可以用来关闭窗口，退出程序，窗口的缺省背景值是黑色(RGBA 模式)或色彩索引模式下的索引零的颜色。

(2)处理输入事件

创建窗口后，在进入主循环前，用下列三个函数定义回调函数：

1) void CALLBACK auxReshapeFunc(Void(*function)(GLsizei，GLsizei)) 函数：当改变窗口尺寸、移动窗口、重新显示窗口时，调用函数 function 重新定义窗口属性的值，通常 function 调用 glViewPort 函数，对当前图形进行裁剪，重新定义投影矩阵，改变长宽比，避免图像变形，缺省时，即使不调用这个函数，也会调用一个二维正交投影的变换函数。

2) void CALLBACK auxKeyFunc(GLint key，void(*function)(void)) 函数： 按下 key 键时，调用 function 函数，key=AUX_A～AUX_Z，AUX_a～AUX_z，AUX_0～AUX_9，AUX_LEFT，AUX_ RIGHT，AUX_UP，AUX_DOWN，AUX_ESCAPE，AUX_SPACE，AUX_ RETRUN，处理完键入事件，窗口会自动重画。

3) void CALLBACK auxMouseFunc(GLint button， GLint mode, void(*function)(AUX_EVENTREC*))函数:

当鼠标键 button 进入 mode 状态时，调用函数 function，button 为 AUX_LEFTBUTION，AUX_MIDDLEBUTTON， AUX_RIGHTBUTTON， mode 为 AUX_MOUSEDOWN，AUX _MOUSEUP，function 的变量是结构类型。AUX_EVENTREC 的指针由 auxMouseFunc 函数为其分配内存，用如下函数可以得到鼠标的位置,用法如下：

```
void Function(AUX_EVENTREC *event)
{ GLint x，y;
  x=event->data[AUX_MOUSEX];
  y=event->data[AUX_MOUSEY];
  ……}
```

(3)载入颜色映射

若使用色彩映射模式，把一个颜色调入颜色查找表的过程是依赖于窗口系统，因此在 OpenGL 核心函数中也不会存在这样一个程序，辅助函数库的 auxSetOneColor 函数可以用 RGBA 值载入一个色彩索引，其函数原型为：

```
void auxSetOneColor(GLint index，GLfloat red，GLfloat green，GLfloat blue);
```

（4）初始化并绘制三维物体

辅助函数库提供了绘制三维物体的子程序，这些三维物体是球、立方体、盒子、环面、圆柱、二十面体、八面体、四面体、十二面体、圆锥，对每个物体有两个子程序分别绘制线框图和实体图。线框图无表面法线；实体图有表面法线和明暗处理，可用于光照，这些函数是：

```
void auxWireSphere(GLdouble radius);
void auxSolidSphere(GLdouble radius);
void auxWireCube(GLdouble size);
void auxSolidCube(GLdouble size);
void auxWireBox(GLdouble width，GLdouble height,GLdouble depth);
void auxSolidBox(GLdouble width，GLdouble height，GLdouble depth);
void WireTorus(GLdouble innerRadius，GLdouble outerRadius);
void SolidTorus(GLdouble innerRadius，GLdouble outerRadius);
void auxWireCylinder(GLdouble radius，GLdouble height);
```

```
void auxSolidCylinder(GLdouble radius, GLdouble height);
void auxWireIcosahedron(GLdouble radius);
void auxSolidIcosahedron(GLdouble radius);
void auxWireOctahedron(GLdouble radius);
void auxSolidOctahedron(GLdouble radius);
void auxWireDodecahedron(GLdouble radius);
void auxSolidDodecahedron(GLdouble radius);
void auxWireCone(GLdouble radius, GLdouble height);
void auxSolidCone(GLdouble radius, GLdouble height);
void auxWireTeapot(GLdouble size);
void auxSolidTeapot(GLdouble size);
```

(5)管理后台进程

可以用 auxIdleFunc 指定一个函数，当没有其他事件需要处理时就执行它，这个子程序的唯一参量是函数的指针，若传递值是零，则不执行这个函数。函数原型如下：

```
void CALLBACK auxIdleFunc(void *func);
```

(6)运行程序

当创建窗口、移动、改变窗口大小或有输入事件时，auxMainLoop 指定一个重新绘制屏幕显示的函数即 displayFunc。

```
void CALLBACK auxMainLoop(void(*displayFunc)(void));
```

程序 1—2 演示如何调用上述函数绘制的一个 OpenGL 动画，如图 1—3 所示。

程序 1—2

```
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

GLvoid initialize(GLvoid);
GLvoid CALLBACK drawScene(GLvoid);
GLvoid CALLBACK resize(GLsizei, GLsizei);
GLvoid drawLight(GLvoid);
void polarView(GLdouble, GLdouble, GLdouble, GLdouble);

GLfloat latitude, longitude, radius;

void _CRTAPI1 main(void)
{
  initialize();
  auxMainLoop(drawScene);
}

GLvoid  CALLBACK  resize(GLsizei width,  GLsizei height)
```

```
{
    GLfloat aspect;

    glViewport(0, 0, width, height);
    aspect = (GLfloat)width/height;
    glMatrixMode(GL_PROJECTION );
    glLoadIdentity();
    gluPerspective(45.0, aspect, 3.0, 7.0);
    glMatrixMode(GL_MODELVIEW);
}

GLvoid initialize(GLvoid)
{
     GLfloat  maxObjectSize, aspect;
     GLdouble near_plane, far_plane;

     GLsizei  width, height;

     GLfloat  ambientProperties[] = {0.7, 0.7, 0.7, 1.0};
     GLfloat  diffuseProperties[]  = {0.8,  0.8, 0.8, 1.0};
     GLfloat  specularProperties[] = {1.0, 1.0, 1.0, 1.0};

     width = 1024.0;
     height = 768.0;

     auxInitPosition(width/4, height/4, width/2, height/2);
     auxInitDisplayMode(AUX_RGB | AUX_DEPTH | AUX_DOUBLE);
     auxInitWindow(  "AUX Library Demo"  );
     auxIdleFunc(drawScene);
     auxReshapeFunc( resize );
     glClearColor( 0.0, 0.0, 0.0, 1.0 );
     glClearDepth( 1.0 );
     glEnable(GL_DEPTH_TEST);
     glEnable(GL_LIGHTING);

     glLightfv(GL_LIGHT0, GL_AMBIENT, ambientProperties);
     glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseProperties);
     glLightfv(GL_LIGHT0, GL_SPECULAR, specularProperties);
     glLightModelf(GL_LIGHT_MODEL_TWO_SIDE, 1.0);

     glEnable( GL_LIGHT0 );

     glMatrixMode( GL_PROJECTION );
     aspect =  (GLfloat) width / height;
```

```
    gluPerspective( 45.0, aspect, 3.0, 7.0);
    glMatrixMode( GL_MODELVIEW );

    near_plane = 3.0;
    far_plane = 7.0;
    maxObjectSize = 3.0;
    radius = near_plane + maxObjectSize/2.0;
    latitude = 0.0;
    longitude = 0.0;
}

void polarView(GLdouble radius, GLdouble twist, GLdouble latitude, GLdouble
longitude)
{
     glTranslated(0.0,  0.0,  -radius);
     glRotated(-twist,  0.0,  0.0,  1.0 );
     glRotated( -latitude, 1.0, 0.0, 0.0);
     glRotated( longitude,  0.0,  0.0, 1.0);
}

GLvoid CALLBACK drawScene(GLvoid)
 {
      static GLfloat   whiteAmbient[]  = {0.3, 0.3,  0.3, 1.0};
      static GLfloat   redAmbient[] = {0.3, 0.1, 0.1, 1.0};
      static GLfloat   greenAmbient[] = {0.1, 0.3, 0.1, 1.0};
      static GLfloat   blueAmbient[] = {0.1, 0.1,  0.3,  1.0};
      static GLfloat   whiteDiffuse[] = {1.0, 1.0, 1.0, 1.0};
      static GLfloat   redDiffuse[] = {1.0, 0.0, 0.0, 1.0};
      static GLfloat   greenDiffuse[]  = {0.0, 1.0, 0.0,  1.0};
      static GLfloat   blueDiffuse[] = {0.0, 0.0, 1.0, 1.0};
      static GLfloat   whiteSpecular[]  = {1.0, 1.0, 1.0,  1.0};
      static GLfloat   redSpecular[] = {1.0,  0.0,  0.0,  1.0};
      static GLfloat   greenSpecular[] = {0.0, 1.0, 0.0, 1.0};
      static GLfloat   blueSpecular[]  = {0.0,  0.0,  1.0, 1.0};

      static GLfloat   lightPosition0[]  = {1.0, 1.0,  1.0, 1.0};
      static GLfloat   angle = 0.0;

      glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
      glPushMatrix();
        latitude += 4.0;
        longitude += 2.5;
        polarView( radius, 0, latitude, longitude );
```

```
        glPushMatrix();
             angle += 6.0;
             glRotatef(angle, 1.0, 0.0, 1.0);
             glTranslatef( 0.0,  1.5,  0.0);
             glLightfv(GL_LIGHT0, GL_POSITION, lightPosition0);
             drawLight();
        glPopMatrix();

        glPushAttrib(GL_LIGHTING_BIT);
          glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, redAmbient);
          glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, redDiffuse);
          glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, whiteSpecular);
          glMaterialf(GL_FRONT, GL_SHININESS, 100.0);

          auxSolidCone( 0.3, 0.6 );

        glPopAttrib();

        auxWireSphere(1.5);

        glPushAttrib(GL_LIGHTING_BIT);

           glMaterialfv(GL_BACK, GL_AMBIENT, greenAmbient);
           glMaterialfv(GL_BACK, GL_DIFFUSE, greenDiffuse);
           glMaterialfv(GL_FRONT, GL_AMBIENT, blueAmbient);
           glMaterialfv(GL_FRONT, GL_DIFFUSE, blueDiffuse);
           glMaterialfv(GL_FRONT, GL_SPECULAR, blueSpecular);
           glMaterialf(GL_FRONT, GL_SHININESS,  50.0);

           glPushMatrix();
                 glTranslatef(0.8, -0.65, 0.0);
                 glRotatef(30.0,  1.0,  0.5,  1.0);
                 auxSolidCylinder( 0.3,  0.6 );
           glPopMatrix();
        glPopAttrib();
      glPopMatrix();

    auxSwapBuffers();
}


GLvoid drawLight(GLvoid)
{
   glPushAttrib(GL_LIGHTING_BIT);
```

```
    glDisable(GL_LIGHTING_BIT);
    glColor3f(1.0, 1.0, 1.0);
    auxSolidDodecahedron(0.1);
  glPopAttrib();
}
```

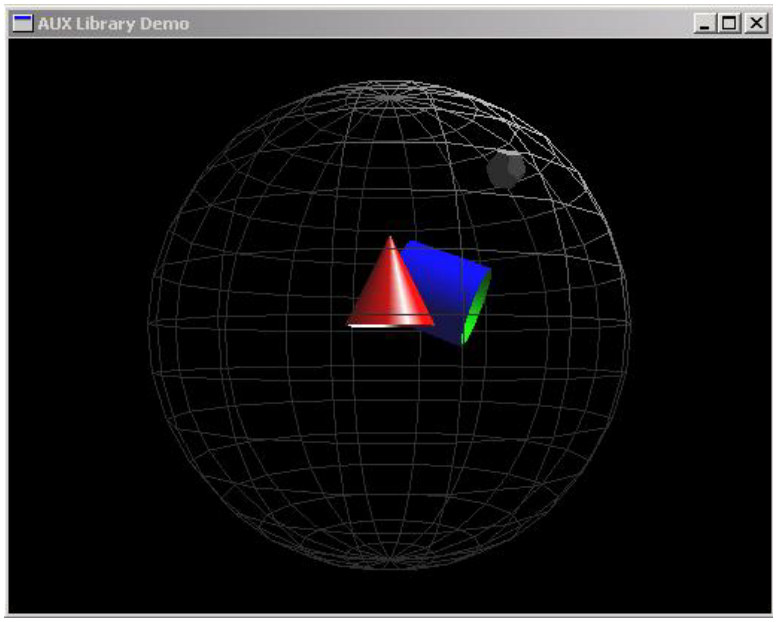

图 1-3 aux 库函数的演示程序

1．3．3 在 OpenGL 中使用颜色

(1) 计算机的颜色

OpenGL 应用程序要在屏幕的窗口上显示彩色图形，而窗口是由像素矩阵构成的一个矩形，每个像素又有各自的颜色，像素的颜色不同组成的图形也不同，因此 OpenGL 的一切操作都是为了最终确定窗口的各像素颜色。计算机屏幕上像素的颜色是红、绿、蓝 3 种颜色按一定比例混合得到的，称为 RGB 值；有时又会出现另一个参数 α，构成 RGBA 值，像素的颜色信息可以用 RGBA 模式存贮，每个像素 4 个值；也可以用色彩索引模式存贮，每个像素一个值，对应颜色映射表中一个特定的 RGBA 组合。

R、G、B 的取值范围为[0.0，1.0]，图 1-4 是颜色组合的示意图，图中仅标出了立方体顶点上的颜色值，这个立方体内的每一点都对应成千上万颜色中的一个特定的颜色。

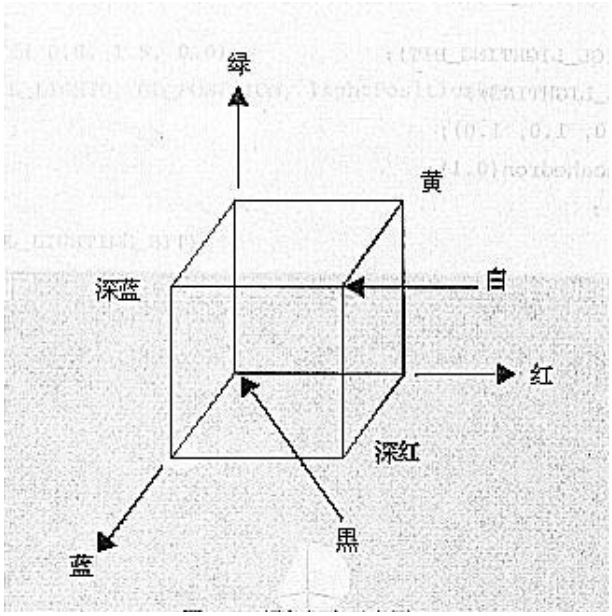

图 1—4 颜色组合示意图

确定像素的颜色要经过一个漫长的过程。首先，程序初始化时，要选择色彩显示模式，即使用 RGBA 模式或色彩索引模式。随着程序的执行，为每个几何要素的顶点指定颜色值，可以用函数指定；若允许光照操作，变换矩阵与表面法线和材质特性等相互作用，也会给顶点定义颜色；完成光照计算后，按选定的明暗模式(平直或光滑处理)对几何要素做处理。

其后，几何要素光栅化或转换成一个二维图像，光栅化确定几何要素在窗口坐标下占用的整数网格方块，并为其赋颜色或其它值。网格方块连同颜色、深度、纹理坐标值称为像素段，若允许下面的操作，对像素段进行纹理、雾、反走样操作，再对帧缓冲区内的像素段做混合、抖动、逻辑位操作，这时才确定了像素的颜色，每个像素的颜色数由帧缓冲区的位平面数目确定，若有 8 个位平面，最多有 2^8 个颜色。

(2)OpenGL 的色彩模式

RGBA 模式下，硬件为 R、G、B、A 组分分配一定数目的位平面，R、G、B、A 值以整数形式存贮，使用时再除以 2^m，m 为分配到的位平面数，RGBA 模式示意图见图 1-5。

色彩索引模式下，OpenGL 用与画家用的调色板相似的色彩映射表(或查找表)，不过这个调色板的基本色调只有红、绿、蓝。色彩索引模式的示意图见图 1-6。色彩索引模式下，可以同时显示的颜色由颜色映射的尺寸和位平面数目确定。若颜色映射的尺寸为 2^n，位平面数为 m，则可用颜色为 $\min(2^n, 2^m)$。在 RGBA 模式，每个像素的颜色是独立的，而在色彩索引模式下，在位平面上有相同索引的像素在颜色映射表中位置相同，若这个位置上的内容变化，则所有具有这个索引值的像素颜色都要变化，如图 1-7 所示。

以上介绍的两种色彩模式各有特色，在使用中如何选择模式还要根据硬件性能和应用程序的需要确定。对于多数系统，RGBA 模式可以同时显示的颜色比较多，用在明暗光照、雾、纹理映射等效果处理上，RGBA 模式更灵活；而在下面提到的几种情况，则要考虑选用色彩索引模式：

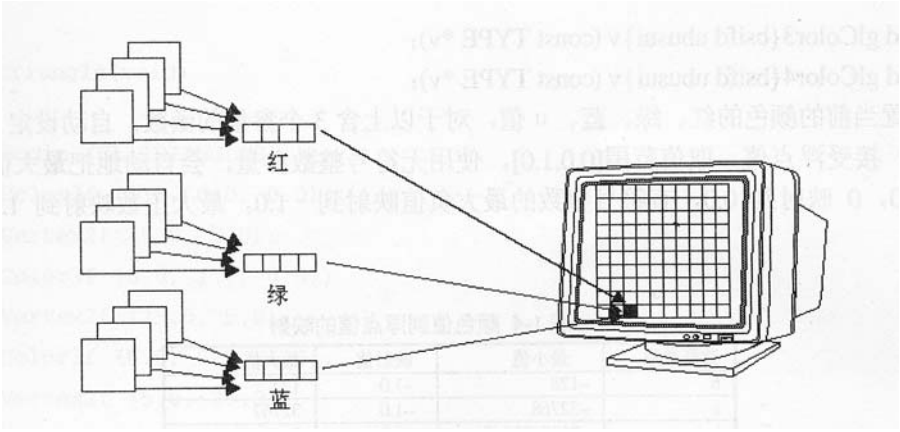

图 1-5 RGBA 模式示意图

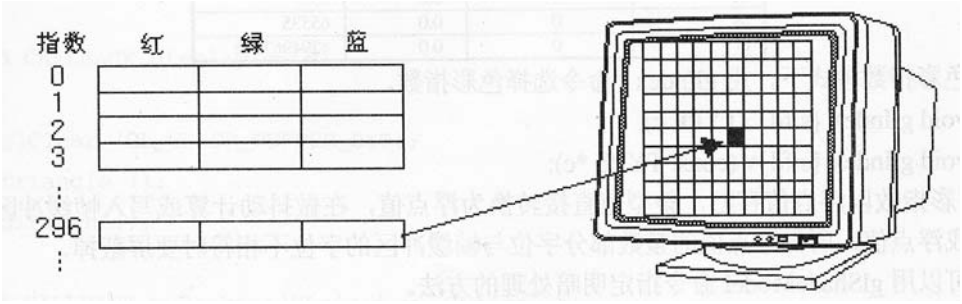

图 1—6 色彩索引模式的示意图

. 移植已经大量使用色彩索引模式的应用程序。

. 若位平面数 n 较小，程序中使用的颜色少于 2^n 个。　　、

. 若位平面数目较少，RGBA 模式下明暗处理的结果很粗糙，若对明暗处理要求不高，比如只用灰度。

. 色彩索引模式在做颜色映射动画、在层平面绘图等操作时更有用处。

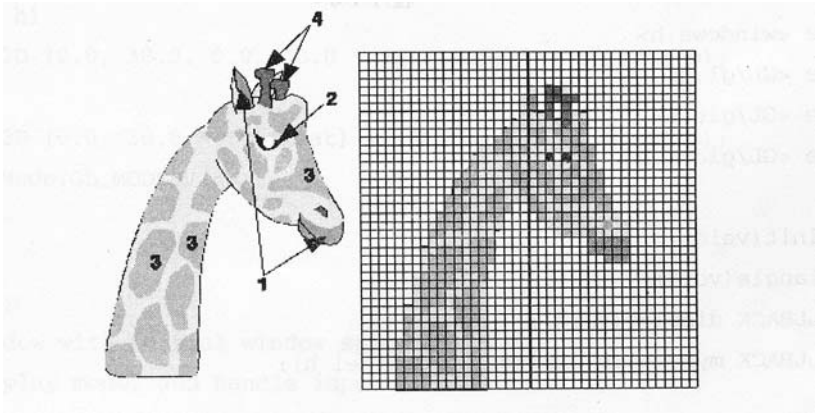

图 1—7　在色彩索引模式下绘图

(3)指定颜色和明暗模式

RGBA 模式下，用 glColor*命令设定颜色。

```
void glColor3{bsifd ubusui}(TYPE r, TYPE g, TYPE b);
void glColor4{bsifd ubusui}(TYPE r, TYPE g, TYPE b, TYPE a);
```

void glColor3{bsifd ubusui}v(const TYPE *v)：

voidglColor4(bsifdubusui)v(const TYPE *v)；

设置当前的颜色的红、绿、蓝、α值，对于以上含3个参量的函数，自动设定α=1.0。glColor* 接受浮点值，取值范围[0.0, 1.0]，使用无符号整数参量，会自动地把最大值线性映射到1.0，0映射到0.0，有符号整数的最大负值映射到-1.0，最大正数映射到1.0，参见表1-4。

表1—4 颜色值到浮点值的映射

数据类型	最小值	映射值	最大值
b	-128	-1.0	127
s	-32768	-1.0	32767
i	-2147483648	-1.0	2147483647
ub	0	0.0	255
us	0	0.0	65535
ui	0	0.0	4294967295

色彩索引模式下，用glIndex*命令选择色彩索引：

void glIndex{sifd}(TYPE c)；

void glIndex{sifd}v(const TYPE *c)；

色彩索引以浮点值存贮，整型值直接转换为浮点值，在做抖动计算或写入帧缓冲区时要转换成浮点值，所得浮点值的整数部分字位与帧缓冲区的字位不相符时要屏蔽掉。

可以用glShadeModel命令指定明暗处理的方法：

void glShadeModel(GLenum mode)；

mode是GL_SMOOTH或GL_FLAT，缺省值是GL_SMOOTH，平直明暗处理时所有顶点的颜色相同，光滑明暗处理时各个顶点的颜色单独处理，沿直线线段的颜色由两顶点的颜色插值，多边形内的颜色也是在顶点间插值得到的。以下是一个明暗处理方法的例子，在程序1-3中绘制两个用不同方法处理的三角形。

程序1-3

```
/* Prog1_3.c */
#include<windows.h>
#include<GL/gl.h>
#include<GL/glu.h>
#include<GL/glaux.h>

void myinit(void);
void triangle(void);
void CALLBACK display(void);

void CALLBACK myReshape(GLsizei w, GLsizei h);

/*GL_SMOOTH is actually the default shading model.   */

void myinit(void)
```

```
{
   glShadeModel(GL_SMOOTH);
}

void triangle(void)
{
    glBegin(GL_TRIANGLES);
      glColor3f(1.0,  0.0,  0.0);
      glVertex2f(5.0,  5.0);
      glColor3f(0.0, 1.0, 0.0);
      glVertex2f(25.0, 5.0);
      glColor3f(0.0, 0.0, 1.0);
      glVertex2f(5.0, 25.0);
    glEnd();
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    triangle();
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    if (!h)  return;
    glViewport(0,  0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if  (w <= h)
      gluOrtho2D(0.0, 30.0, 0.0, 30.0 * (GLfloat)h/(GLfloat)w);
    else
      gluOrtho2D(0.0, 30.0 * (GLfloat)w/(GLfloat)h, 0.0, 30.0);

    glMatrixMode(GL_MODELVIEW);
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, and handle input events.
 */
int main(int argc,  char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
```

```
    auxInitPosition (0,  0,  500,  500);
    auxInitWindow ("Smooth Shading");
    myinit();
    auxReshapeFunc(myReshape);
    auxMainLoop(display);
    return(0);
}
```

在 RGBA 模式下，相邻像素的颜色值略有不同，多边形内颜色是逐渐变化的，而在色彩索引模式下，相邻像素在颜色映射中引用的值可能在不同的位置，颜色可能不相似，这样多边形内的颜色变化比较明显。因此要用 auxSetOneColor 函数在颜色映射的相邻索引间产生光滑变化的颜色。

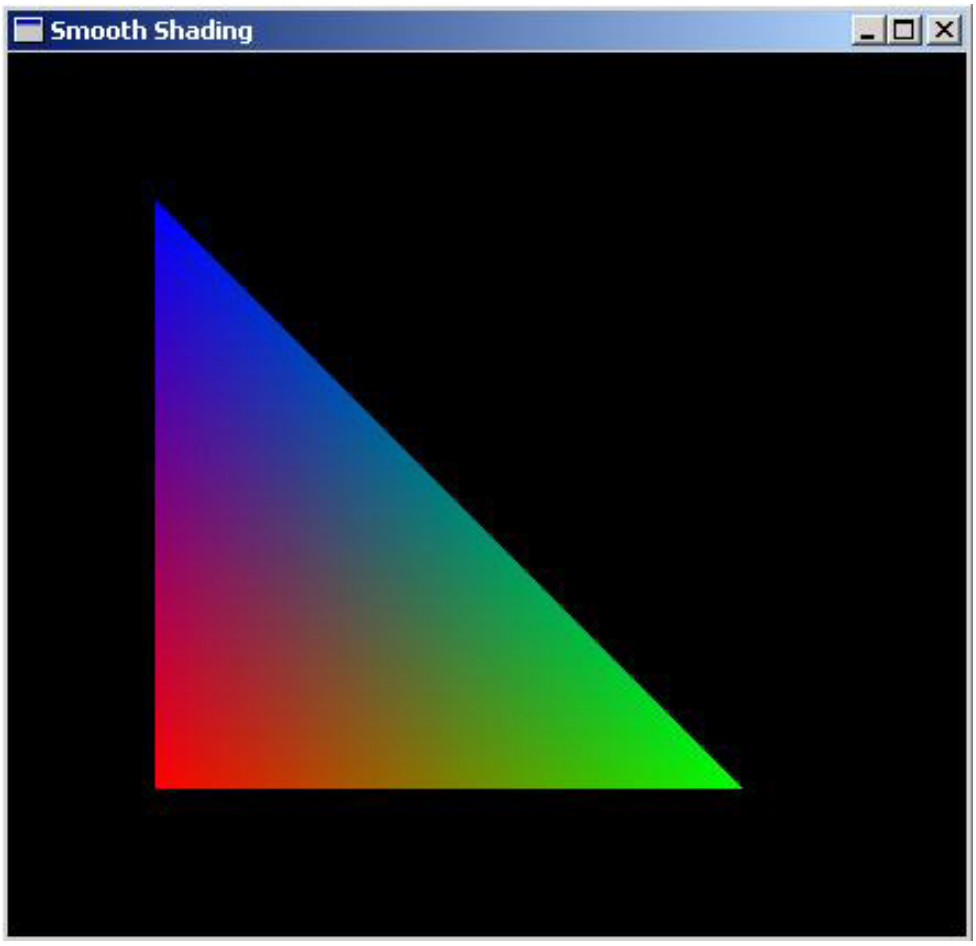

图 1-8 颜色的平滑处理

第二章　用 OpenGL 画几何体

从根本上看，OpenGL 绘制的所有复杂的三维物体都是由一定数量的基本图形要素构成的，曲线、曲面分别是由一系列直线段、多边形近似得到的。本章首先讲述在 OpenGL 中如何描述几何要素、设置并使用几何要素的属性。

2．1　绘图前的一些准备工作作

在开始绘制新图形前，计算机屏幕上可能已有一些图形，OpenGL 存储了那些图形绘图状态的信息，所以必须清除当前窗口的内容，以免影响绘图的效果，常用的函数有：

① void glClearColor(Glclampf red, GLclampf green, GLclampf blue, GLclampf alpha)函数，给定当前屏幕的背景设置颜色，red，green，blue，alpha 为 RGBA 颜色值；

② void glClear(GLbitfield mask)函数，标识要清除的缓冲区，例如：

```
glClearColor(1. 0，1. 0，0. 0，0. 0)；
glClear(GL_COLOR_BUFFER_BIT)；
```

把屏幕的窗口背景变为黄色。

glClear 命令也可以用于清除其他缓冲区，mask 为下列值的逻辑位或组合，见表 2-1。

表 2—1　清除缓冲区

缓冲区	名　　称
颜色缓冲区	GL_COLOR_BUFFER_BIT
深度缓冲区	GL_DEPTH_BUFFER_BIT
累加缓冲区	GL_ACCUM_BUFFER_BIT
模板缓冲区	GL_STENCIL_BUFFER_BIT

```
glClearColor(1. 0，1. 0，0. 0，0. 0)；
glClearDepth(0. 0)；
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)；
```

把屏幕的窗口背景设置为黄色，深度缓冲区的所有像素值设为零，另外，用 glClearColor，glClearDepth，glClearIndex，glClearStencil，glClearAcc 函数为各自对应的缓冲区清零。值得注意的是，若要同时清除多个缓冲区，使用上面提到的 mask 位或组合，要比使用多次调用 glClear 函数要快得多。

2．2 OpenGL 的几何要素

2．2．1　OpanGL 的几何要素

OpenGL 的几何要素有点、线、多边形。　OpenGL 中的点是三维的，　用户设定二维坐标(x，y)，此时自动地 z=0，线是用一系列相连的顶点定义的，多边形是一个封闭的线段，通过选择属性，既可以得到填充的多边形，也可以是轮廓线，或是一系列点，多边形的边不能交叉且应该是凸多边形，若要绘制的多边形不满足这些要求，则应调用 GLU 库函数，进行详细描述和镶嵌。

不在一个平面上的多边形有时经过旋转变换，有时会不满足上述要求，为避免这种情况发生，可以使用三角形代替多边形，如图 2-1 所示。

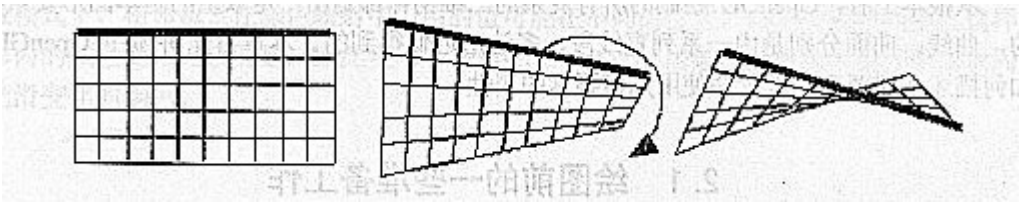

图 2—1　不共面的多边形

OpenGL 提供 glRect 命令把矩形当成一个特殊的四边形绘制:

void glRect{sifd}(TYPE x1, TYPE y1, TYPE x2, TYPE y2);

Void glRect{sifd}v(TYPE *v1, TYPE *v2);

其中(x1, y1)为矩形左下角坐标，(x2, y2)为右上角坐标，而 v1，v2 分别为矩形对角线上的两个顶点的矢量。

光滑的曲线、曲面都是由一系列线段、多边形近似得到的，OpenGL 不直接提供绘制曲线、曲面的命令，如图示 2-2 所示。

图 2—2　用线段近似曲线

由上面的叙述可知，描述几何要素就是按一定顺序给出几何要素的顶点，glVertex 命令指定一个顶点，并在生成顶点后，把当前颜色、纹理坐标、法线等值赋给这个顶点。

Void glVertex{234}{sifd}{v}(TYPE coords)函数有时用矢量形式定义顶点，执行效率高。应该注意的是，只有在 glBegin 与 glEnd 之间调用 glVertex 函数才有意义，而 glBegin 标识顶点的定义开始，glEnd 命令标志结束一个几何要素的定义。

Void glBegin(GLenum mode);

mode 的值见下表 2-2，这些值的示意图见图 2-3。下面举例说明如何在程序 2-1 中实现这些 OpenGL 几何要素的调用。

表 2-2 mode 的取值

mode 的值	解　　释
GL_POINTS	一系列独立的点
GL_LINES	每两点相连成为线段
GL_POLYGON	简单、凸多边形的边界
GL_TRIANGLES	三点相连成为一个三角形
GL_QUADS	四点相连成为一个四边形
GL_LINE_STRIP	顶点相连成为一系列线段 1
GL_LINE_LOOP	顶点相连成为一系列线段，连接最后一点与第一点
GL_TRIANGLE_STRIP	相连的三角形带
GL_TRIANGLE_FAN	相连的三角形扇形
GL_QUAD_STRIP	相连的四边形带

在 glBegin 与 glEnd 之间，只有一些命令可调用，见表 2—3，而其他命令是无效的，并

会使 OpenGL 出现错误。

表 2-3 可以在 glBegin/glEnd 间调用的函数

命　　令	作用
glVertex	设置顶点坐标
glColor	设置当前颜色
GlIndex	设置当前色彩索引
glNormal	设置当前法线矢量坐标
glEvalCoord	生成坐标
glCallList, glCallLists	执行一个或多个显示列表
glTexCoord	设置纹理坐标
glEdgeFlag	控制边缘的绘制
glMaterial	设置材质的属性

程序 2—1

```
/* Prog2_1.c */
#include <windows.h>
#include <GL/gl.h>
#include <GL/glaux.h>

void myinit(void);
void DrawMyObjects(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

void myinit(void)
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glClear(GL_COLOR_BUFFER_BIT);
   glShadeModel(GL_FLAT);
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if(w<=h)
     glOrtho(-20.0, 20.0, -20.0*(GLfloat)h/(GLfloat)w,
                  20.0*(GLfloat)h/(GLfloat)w, -50.0, 50.0);
   else
     glOrtho(-20.0*(GLfloat)h/(GLfloat)w,
            20.0*(GLfloat)h/(GLfloat)w, -20.0, 20.0, -50.0, 50.0);
```

```
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void CALLBACK display(void)
{
    glColor3f(1.0, 1.0, 0.0);
    DrawMyObjects();
    glFlush();
}

void DrawMyObjects(void)
{
    /* draw some points  */
    glBegin(GL_POINTS);
      glColor3f(1.0, 0.0, 0.0);
      glVertex2f(-10.0, 11.0);
      glColor3f(1.0, 1.0, 0.0);
      glVertex2f(-9.0, 10.0);
      glColor3f(0.0, 1.0, 1.0);
      glVertex2f(-8.0, 12.0);
    glEnd();

    /* draw some line segments */
    glBegin(GL_LINES);
      glColor3f(1.0, 1.0, 0.0);
      glVertex2f(-11.0, 8.0);
      glVertex2f(-7.0, 7.0);
      glColor3f(1.0, 0.0, 1.0);
      glVertex2f(-11.0, 9.0);
      glVertex2f(-8.0, 6.0);
    glEnd();

    /* draw one opened_line  */
    glBegin(GL_LINE_STRIP);
      glColor3f(0.0, 1.0, 0.0);
      glVertex2f(-3.0, 9.0);
      glVertex2f(2.0, 6.0);
      glVertex2f(3.0, 8.0);
      glVertex2f(-2.5, 6.5);
    glEnd();

    /* draw one closed_line  */
     glBegin(GL_LINE_LOOP);
```

```
  glColor3f(0.0, 1.0, 1.0);
  glVertex2f(7.0, 7.0);
  glVertex2f(8.0, 8.0);
  glVertex2f(9.0, 6.5);
  glVertex2f(10.3, 7.5);
  glVertex2f(11.5, 6.0);
  glVertex2f(7.5, 6.0);
glEnd();

/* draw ene filled polygen  */
glBegin(GL_POLYGON);
  glColor3f(0.5, 0.3, 0.7);
  glVertex2f(-7.0, 2.0);
  glVertex2f(-8.0, 3.0);
  glVertex2f(-10.3, 0.5);
  glVertex2f(-7.5, -2.0);
  glVertex2f(-6.0, -1.0);
glEnd();

/* draw some filled quandrangles  */
glBegin(GL_QUADS);
  glColor3f(0.7, 0.5, 0.2);
  glVertex2f(0.0, 2.0);
  glVertex2f(-1.0, 3.0);
  glVertex2f(-3.3, 0.5);
  glVertex2f(-0.5, -1.0);
  glColor3f(0.5, 0.7, 0.2);
  glVertex2f(3.0, 2.0);
  glVertex2f(2.0, 3.0);
  glVertex2f(0.0, 0.5);
  glVertex2f(2.5, -1.0);
glEnd();

/* draw some filled strip_quandrangles  */
glBegin(GL_QUAD_STRIP);
  glVertex2f(6.0, -2.0);
  glVertex2f(5.5, 1.0);
  glVertex2f(8.0, 1.0);
  glColor3f(0.8, 0.0, 0.0);
  glVertex2f(9.0, 2.0);
  glVertex2f(11.0, -2.0);
  glColor3f(0.0, 0.0, 0.8);
  glVertex2f(11.0, 2.0);
  glVertex2f(13.0, -1.0);
```

```
      glColor3f(0.0, 0.8, 0.0);
      glVertex2f(14.0, 1.0);
    glEnd();

    /* draw some filled_triangles */
    glBegin(GL_TRIANGLES);
      glColor3f(0.2, 0.5, 0.7);
      glVertex2f(-10.0, -5.0);
      glVertex2f(-12.3, -7.5);
      glVertex2f(-8.5,-6.0);

      glColor3f(0.2, 0.7, 0.5);
      glVertex2f(-8.0, -7.0);
      glVertex2f(-7.0, -4.5);
      glVertex2f(-5.5, -9.0);
    glEnd();

    /* draw some filled strip_triangles  */
    glBegin(GL_TRIANGLE_STRIP);
    glVertex2f(-1.0, -8.0);
    glVertex2f(-2.5, -5.0);
    glColor3f(0.8, 0.8, 0.0);
    glVertex2f(1.0, -7.0);
    glColor3f(0.0,0.8,0.8);
    glVertex2f(2.0, -4.0);
    glColor3f(0.8, 0.0, 0.8);
    glVertex2f(4.0, -6.0);
  glEnd();

  /* draw some filled fan triangles  */
  glBegin(GL_TRIANGLE_FAN);
    glVertex2f(8.0, -6.0);
    glVertex2f(10.0, -3.0);
    glColor3f(0.8, 0.2, 0.5);
    glVertex2f(12.5, -4.5);
    glColor3f(0.2, 0.5, 0.8);
    glVertex2f(13.0, -7.5);
    glColor3f(0.8, 0.5, 0.2);
    glVertex2f(10.5, -9.0);
  glEnd();
}

/*  Main Loop
*  Open window with initial window size, title bar,
```

```
*  RGBA display mode, and handle input events.
*/

void main(void)
{
   auxInitDisplayMode(AUX_SINGLE | AUX_RGBA);
   auxInitPosition(0, 0, 500, 500);
   auxInitWindow("Geometric Primitive Types");
   myinit();
   auxReshapeFunc(myReshape);
   auxMainLoop(display);
}
```

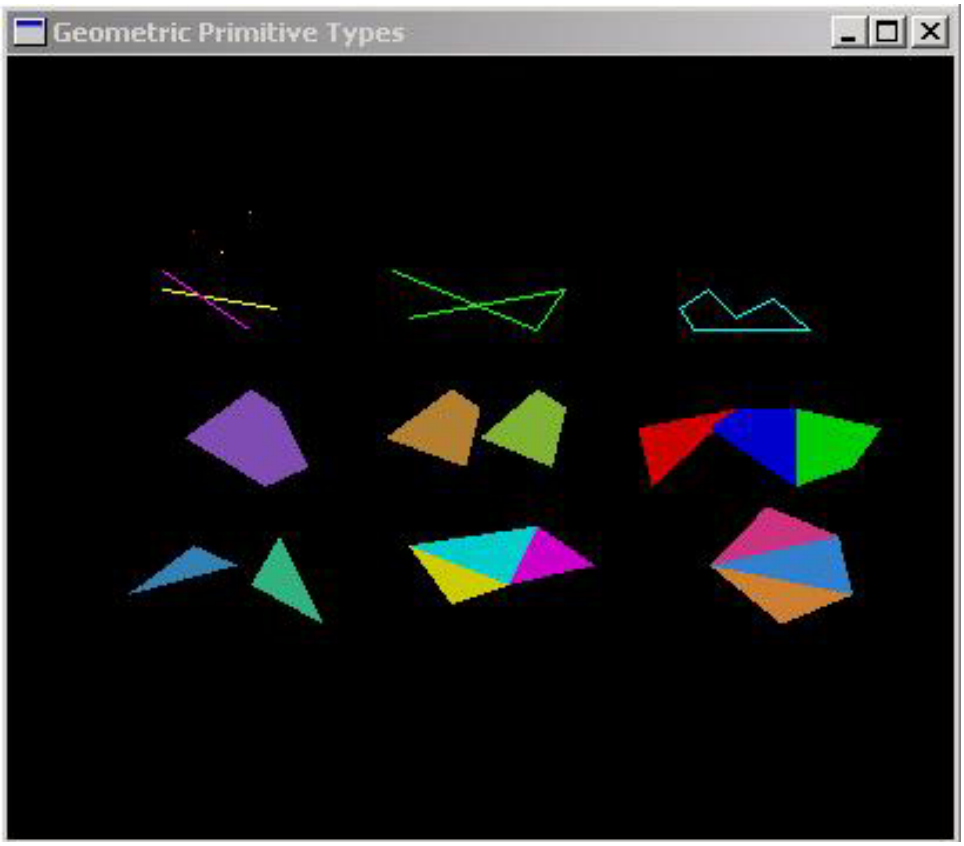

图 2-3　mode 的示意图

2．2．2　如何使用几何要素

缺省时，点是屏幕上的一个像素，线为一个像素宽度的实线，绘出的多边形是填充多边形，然而这些属性可以根据需要加以改动。

(1)点

把绘图点大小设为 size，若不允许使用反走样，把宽度的小数部分圆整到整数，得到与窗口的边框对齐的正方形，若使用反走样，所得到的点是图形的像素束，边缘光滑，不进行圆整。可以用 glGetFloatv 的 GL_POIN_SIZE_RANGE 参量查询最大走样点尺寸。

Void glPointSize(Glfloat size);

(2)线

设置线的像素宽度，与点的大小相似，走样时，线宽为整数值，反走样时，允许有浮点值的线宽，部分填充边界像素，可以用 glGetFloatv 的 GL_LINE_WIDTH 参量查询最大走样线宽度。

Void glLineWidth(Glfloat width);

若直线不是实线，可以用 glLineStipple 函数定义填充图案，并首先调用 glEnable 命令设置允许使用线填充。

Void glLineStipple(GLint factor, GLushort pattern);

glEnable(GL_LINE_STIPPLE);

Pattern 是 16 位二进制码，1 表示填充，从低位到高位重复使用，定义填充方式，factor 是伸长因子，取值范围[1, 255]。各参量的作用如图 2-4 所示。

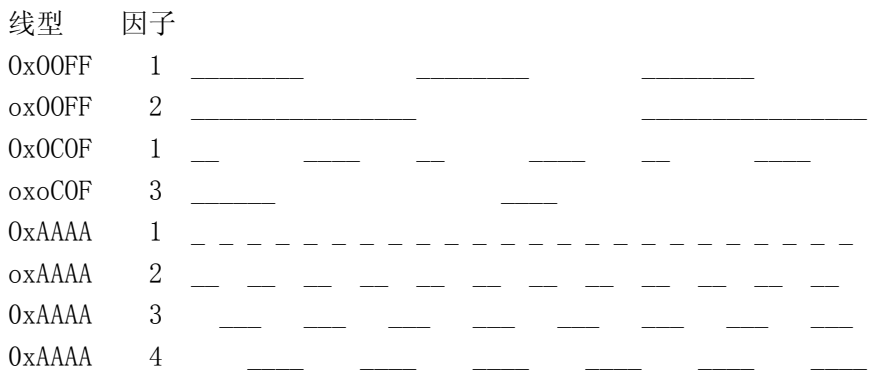

图 2-4 线填充的参数示意图

glLineStipple 函数可以与 glLineWidth 函数相结合，得到所需要的各种线型，下面给出一个利用上述两个函数绘制各种线型的程序(程序 2-2)，结果如图 2-5 所示。

程序 2—2

```
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK display(void);
#define drawOneLine(x1,y1,x2,y2) glBegin(GL_LINES);  \
     glVertex2f((x1), (y1)); glVertex2f((x2), (y2)); glEnd();

void myinit(void)
{
   /*  background to be cleared to black   */
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_FLAT);
}
```

```
void CALLBACK display(void)
{
      int i;
      glClear(GL_COLOR_BUFFER_BIT);
/*  draw all lines in white */
      glColor3f(1.0,  1.0,  1.0);
/*  in 1st row,  3 lines drawn,  each with a different stipple    */
      glEnable(GL_LINE_STIPPLE);
      glLineStipple(1,  0x0101);   /*  dotted  */
      drawOneLine(50.0,  125.0,  150.0,  125.0);
      glLineStipple(1,  0x00FF);  /*  dashed  */
      drawOneLine (150.0, 125.0, 250.0, 125.0);
      glLineStipple(1,  0x1C47);  /*  dash/dot/dash   */
      drawOneLine (250.0, 125.0, 350.0, 125.0);
/*  in 2nd row,  3 wide lines drawn, each with different stipple */
      glLineWidth(5.0);
      glLineStipple(1, 0x0101);
      drawOneLine(50.0, 100.0, 150.0, 100.0);
      glLineStipple(1, 0x00FF);
      drawOneLine(150.0,  100.0,  250.0,  100.0);
      glLineStipple  (1,  0x1C47);
      drawOneLine(250.0, 100.0, 350.0, 100.0);
      glLineWidth(1.0);
/*  in 3rd row,  6 lines drawn, with dash/dot/dash stipple,  */
/*  as part of a single connect line strip          */
      glLineStipple(1,  0x1C47);
      glBegin (GL_LINE_STRIP);
        for  (i = 0;  i < 7;  i++)
          glVertex2f(50.0 + ((GLfloat)i*50.0), 75.0);
      glEnd ();
/*  in 4th row,  6 independent lines drawn,   */
/*  with dash/dot/dash stipple           */
      for  (i = 0;  i < 6;  i++)
      {
        drawOneLine(50.0 + ((GLfloat)i*50.0),
            50.0, 50.0 + ((GLfloat)(i+1)*50.0), 50.0);
      }
/*  in 5th row,  1 line drawn, with dash/dot/dash stipple    */
/*  and repeat factor of 5           */
      glLineStipple(5, 0x1C47);
      drawOneLine (50.0,  25.0,  350.0,  25.0);
      glFlush ();
}
```

```
/*  Main Loop
    Open window with initial window size,  title bar,
    RGBA display mode, and handle input events.
  */

int main(int argc,  char** argv)
{
      auxInitDisplayMode(AUX_SINGLE | AUX_RGB);
      auxInitPosition(0,  0,  400,  150);
      auxInitWindow("Lines");
      myinit();
      auxMainLoop(display);
      return(0);
}
```

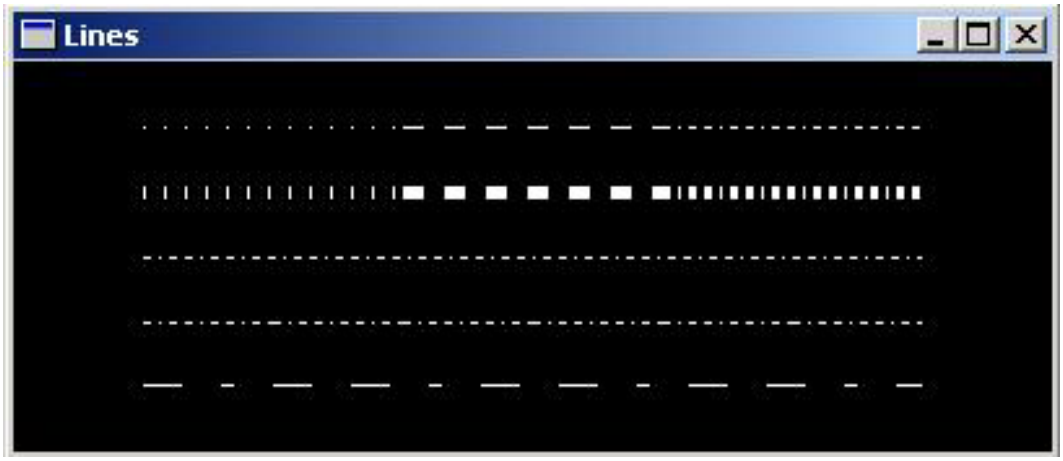

图 2-5 线填充程序绘制的图形

(3)多边形

与数学上的多边形不同，OpenGL 的多边形是有前、后面的，通常把顶点排列顺序为逆时针的多边形定义为朝前，显示时不同的面朝向观察者效果可能不同，这就如同剖视实体，看到里面与外面的部分是截然不同的，可以用

void glPolygonMode(Glenum face，Glenum mode)；

控制多边形图形的属性，face 是 GL_FRONT_AND_BACK，GL_FRONT，GL_BACK 之一，mode 是 GL_POINT，GL_LINE，GL_FILL 之一。缺省时，画出的多边形是前后面同时绘出的填充多边形，即 face = GL_FRONT_AND_BACK，mode = GL_FILL。

多边形的方向性可以用如下的函数加以改变：

void glFrontFace(Glenum mode)；

其中 mode = GL_CCW，GL_CW 之一，前者代表逆时针方向，后者为顺时针方向，若绘图过程中，只用一种多边形方向定义，则另一方向始终是不可见的，这种情况下，则可以用函数

void glCullFace(Glenum mode)；

其中 mode 是 GL_FRONT，GL_BACK，GL_FRONT_AND_BACK 之一。

此时必须是已经调用 glEnable(Gl_CULL_FACE)，不画出朝后的多边形，从而加快绘图速度。

缺省时，多边形内部用当前颜色填充，也可以用 32×32 位与窗口平行的位图图案填充，即调用函数

void glPolygonStipple(const GLubyte *mask);

mask 为指向 32×32 位图的指针，glEnable(GL_POLYGON_STIPPLE)允许使用这个功能。为了直观说明位图图案填充多边形的使用方法，下面举例(程序 2-3)说明如何分别在矩形和三角形内填充两个 32×32 位图，结果如图 2-6 所示。

程序 2—3

```
/* Prog2_3.c */
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK display(void);

void myinit (void)
{
  /*  clear background to black   */
  glClearColor(0.0,  0.0,  0.0,  0.0);
  glShadeModel(GL_FLAT);
}

void CALLBACK display(void)
{
  /* Pattern Defined. */
  GLubyte pattern[]= {
  0x00,  0x01,  0x80,  0x00,
  0x00,  0x03,  0xc0,  0x00,
  0x00,  0x07,  0xe0,  0x00,
  0x00,  0x0f,  0xf0,  0x00,
  0x00,  0x1f,  0xf8,  0x00,
  0x00,  0x3f,  0xfc,  0x00,
  0x00,  0x7f,  0xfe,  0x00,
  0x00,  0xff,  0xff,  0x00,
  0x01,  0xff,  0xff,  0x80,
  0x03,  0xff,  0xff,  0xc0,
  0x07,  0xff,  0xff,  0xe0,
  0x0f,  0xff,  0xff,  0xf0,
  0x1f,  0xff,  0xff,  0xf8,
  0x3f,  0xff,  0xff,  0xfc,
  0x7f,  0xff,  0xff,  0xfe,
  0xff,  0xff,  0xff,  0xff,
```

```
  0xff,  0xff,  0xff,  0xff,
  0x7f,  0xff,  0xff,  0xfe,
  0x3f,  0xff,  0xff,  0xfc,
  0x1f,  0xff,  0xff,  0x18,
  0x0f,  0xff,  0xff,  0xf0,
  0x07,  0xff,  0xff,  0xe0,
  0x03,  0xff,  0xff,  0xc0,
  0x01,  0xff,  0xff,  0x80,
  0x00,  0xff,  0xff,  0x00,
  0x00,  0x7f,  0xfe,  0x00,
  0x00,  0x3f,  0xfc,  0x00,
  0x00,  0x1f,  0x18,  0x00,
  0x00,  0x0f,  0xf0,  0x00,
  0x00,  0x07,  0xe0,  0x00,
  0x00,  0x03,  0xc0,  0x00,
  0x00,  0x01,  0x80,  0x00};

  glClear(GL_COLOR_BUFFER_BIT);
  /*  draw one stippled rectangle  */
  glColor3f(0.1, 0.8, 0.7);
  glEnable(GL_POLYGON_STIPPLE);
  glPolygonStipple(pattern);
  glRectf(48.0,  80.0, 210.0,  305.0);
  /*  draw one stippled triangle  */
  glColor3f(0.9, 0.86, 0.4);
  glPolygonStipple(pattern);
  glBegin(GL_TRIANGLES);
     glVertex2i(310, 310);
     glVertex2i(220, 80);
     glVertex2i(405, 80);
  glEnd();
  glDisable(GL_POLYGON_STIPPLE);
  glFlush();
}

  /*  Main Loop
   *  Open window with initial windew size,  title bar,
   *  RGBA display mede,  and handle input events.
   */

void main(void)
{
      auxInitDisplayMode(AUX_SINGLE | AUX_RGBA);
      auxInitPosition (0,  0,  500,  400);
```

```
    auxInitWindow("Polygon Stippling");
    myinit();
    auxMainLoop(display);
}
```

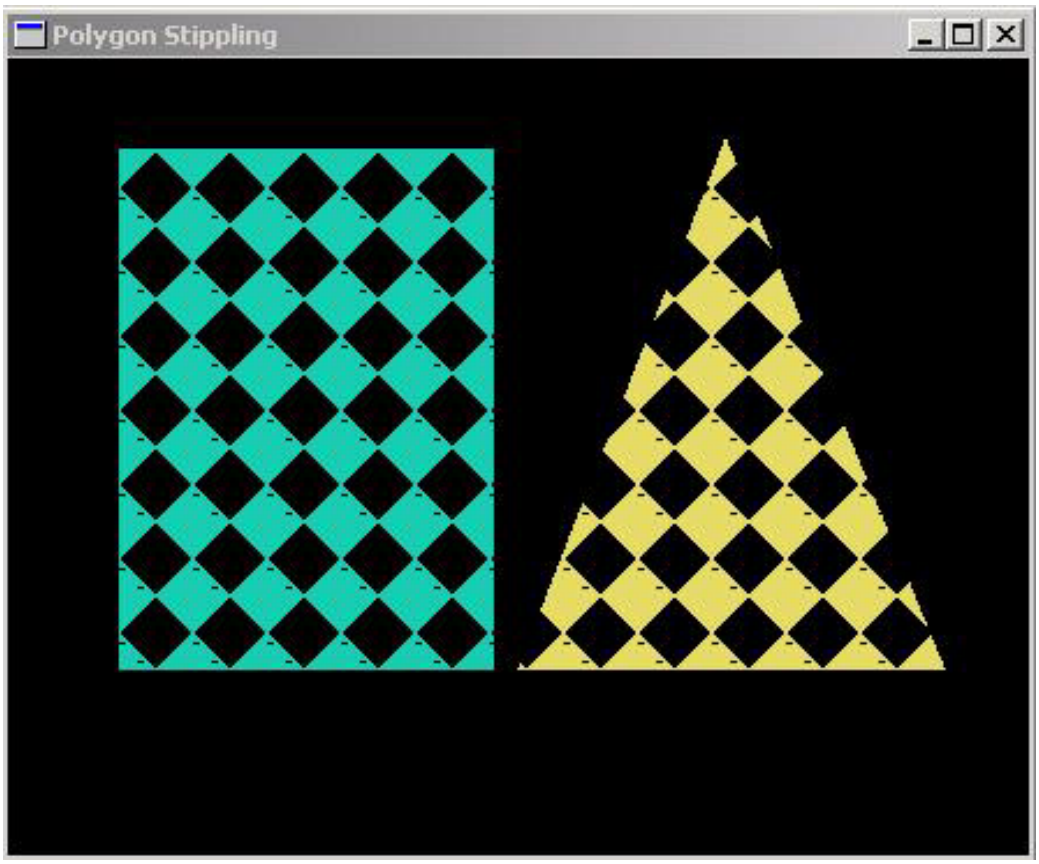

图 2-6　多边形位图图案

从上面介绍的内容可知，OpenGL 只能绘制凸多边形：若要使用凹多边形则会遇到问题；若把它分解成三角形，则不能真正使用 glPolygonMode 的功能，这时可以把多边形顶点的信息传递给 OpenGL，单独绘出这部分图形。而函数

```
void glEdgeFlag(Glboolean flag);
void glEdgeFlagv(const Glboolean *flag);
```

用来识别顶点是否在多边形的边缘，若 flag = GL_TRUE，标识为 TRUE(缺省值)，这个命令只适用于多边形、三角形、四边形，不适用于三角形、四边形的带。

2．3　法向矢量

OpenGL 可以为每一个顶点设定一个法线，而物体的法线代表其空间曲面的方向，缺省时，方向是指向曲面外的这个矢量，顶点的法线可以决定该顶点上接受的光强。

```
Void glNormal3{bidf}(TYPE nx, TYPE ny, TYPE nz);
Void glNormal3{bidf}v(const TYPE *v);
```

其中 nx，ny,nz 为法线向量的坐标分量，v 为法线矢量，函数为当前法线向量赋值，而其后定义的顶点使用这个法线值，例如：

```
glBegin(GL_POLYGON);
  glNormal3fv(n0);
  glVertexf(v0);
  glNormal3f(n1);
  glVertex3f(v1);
  glNormal3f(n2);
  glVertex3f(v2);
  glNormal3f(n3);
  glVertex3f(v3);
glEnd();
```

因为法线矢量只表示方向，任意长度的法线在执行光照前都要最终把长度转换为单位长度，通常要给定有单位长度的法线矢量。若要 OpenGL 自动在变换后归一化法线矢量，则应调用 glEnable(GL_NORMALIZE)，缺省时，不做归一化操作。

OpenGL 不提供直接求解曲面法线矢量的函数，需要用户自己经过计算得出。计算法线矢量的方法很重要，其质量的高低直接影响光照的效果，并最终影响绘图的质量，如图 2-7 所示。下面着重讲述求解法线的方法。

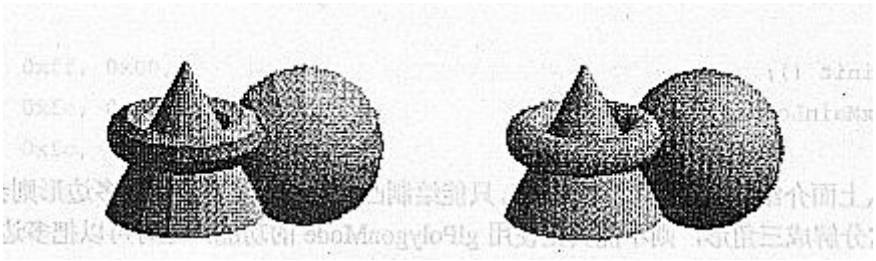

图 2—7　求解法线方法对绘图效果的影响

解析曲面是由数学方程给出的光滑、可微曲面，若曲面的显式方程为

$V(s, t)=V[X(s, t), Y(s, t), Z(s, t)]$

s，t 为曲面的参数，X、Y、Z 为可微函数，$\frac{\partial V}{\partial s}\times\frac{\partial V}{\partial t}$ 即为曲面的法线，若方程为隐式的，即 $V(s, t) = V[s, t, G(s, t)]$，此时得不到显式定义的法线，但是法线矢量是由函数以梯度给出的，即

$$\Delta F=[\frac{\partial F}{\partial X}\frac{\partial F}{\partial Y}\frac{\partial F}{\partial Z}]$$

OpenGL 绘制的曲面由多边形近似得到，因此法线向量的计算也只能由多边形的数据得到。对于在一个平面上的多边形，取其任意不共线的三点，其叉积(V1-V2)×(V2 一 V3)垂直于其平面，然后为避免在相邻平面上平均法线某个值过大，如图 2-8 所示，n1，n2，n3，n4 相交于 P 点，这时求出 n1 + n2 + n3 + n4，做归一化处理，即为 P 点的法线。在有些特殊情况下，如 Q 点在边界上，则要进行判断法线的取法，有时取边界上多边表法线的平均值较好。

另外，曲面有时光滑，有时变化比较剧烈，如 R 点，这时就不能做算术平均，要分别处理交界面上的法线。

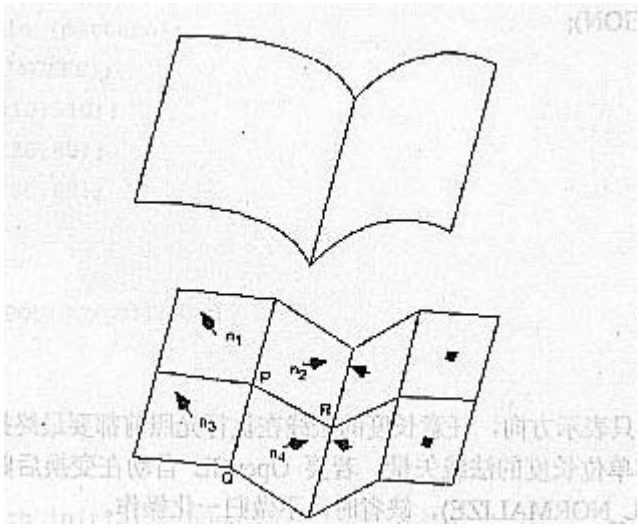

图 2-8　计算法线向量的示意图

2．4　用多边形建立曲面的技巧

正如前面提到的，OpenGL 用多边形近似光滑曲面，而建立这个近似需要注意以下事项：

① 多边形方向应该一致，从曲面外面看，表面上的多边形方向要一致，这是绘制曲面时要牢记的第一件事情。

② 除三角形外，多边形的各顶点可能不共面，有时不是简单多边形，使 OpenGL 不能正常工作。

③ 要处理好显示速度与图像质量的关系，对于比较光滑的曲面可以用较长的多边形，离视角比较远的曲面可以用比较少的多边形。

④ 生成高质量的图像，轮廓边缘用的多边形比内部的多边形密，若法向矢量与视角到平面的矢量垂直，则应多用一些多边形。

⑤ 在模型中避免 T 型交叉，因为有时会在曲面交界处出现裂缝。

⑥ 若构造封闭曲面，要使封闭循环的开始和结束使用相同数目的坐标，否则会由于舍入误差出现裂缝。例如，因为 sin，cos 函数的 0 与 2*PI*EDGES / EDGES 不会完全相等，会带来误差。

下面给出一个用多面体近似圆球图形的例子，分别用 20、80、320 个多边形近以，可以看出，随着多边形数目的增加，逐渐逼近一个真实的球体，如图 2—9 所示。

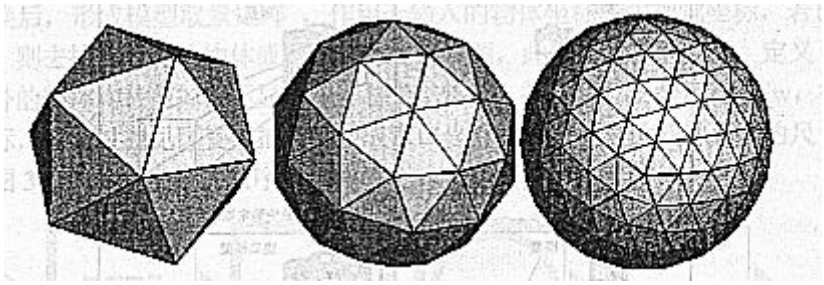

图 2—9　多面体近似圆球

第三章　在 OpenGL 中观察物体

OpenGL 的强大绘图功能表现在绘制各种复杂的三维图形，针对的是一个丰富多彩的三维世界。与二维图形不同的是，首先需要在三维空间合适的位置布置模型，选择一个有利的观察点，然后才能看到一幅视觉效果良好的逼真的三维图形。本章着重讲述 OpenGL 的取景变换、投影变换、视见区变换以及如何控制变换矩阵堆栈存贮、恢复 OpenGL 的状态，以上这些内容对完成复杂的三维图形是十分必要的。

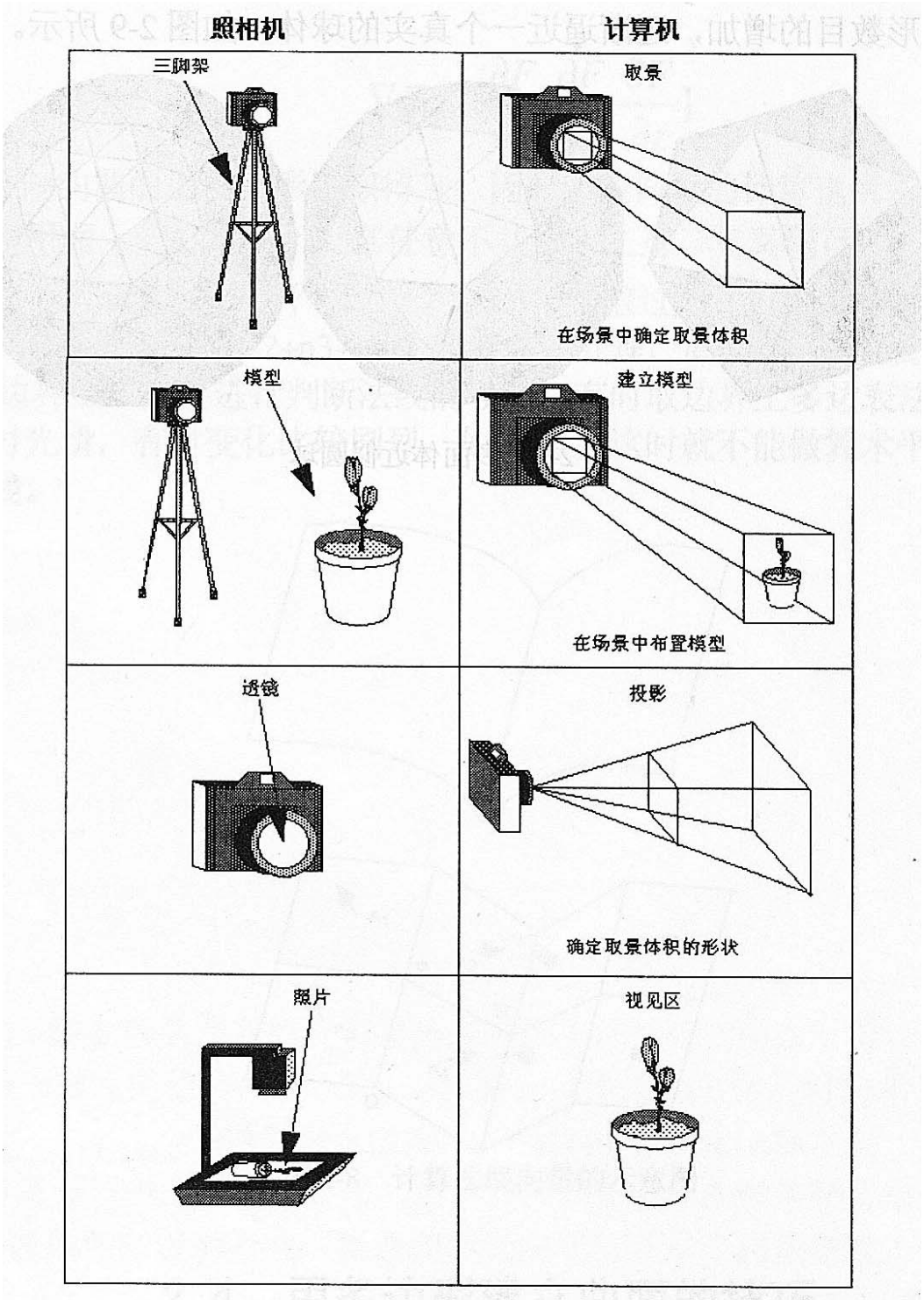

图 3-1 照相机比拟

3．1　OpenGL 基本变换命令

3．1．1 OpenGL 变换过程概述

OpenGL 对图形进行变换与照相机拍摄物体的过程是相似的，如图 3—1 所示。

- 支起三角架，把照相机放在场景中，相当于 OpenGL 的取景变换。
- 把要拍摄的场景固定在要拍摄的物体上，相当于 OpenGL 的模型变换。
- 选择照相机镜头或调节焦距，相当于 OpenGL 的投影变换。
- 确定照片的大小，可以放大照片的某一部分，相当于 OpenGL 的视见区变换。

从物体的顶点坐标变换到屏幕的窗口坐标，具体的变换过程如图 3-2 所示，在做取景和模型变换过程中，法线矢量也自动地进行变换，这样才能保证法线与顶点坐标的关系，经过取景、模型变换后，形成模型取景矩阵，作用于输入的物体坐标产生视觉坐标，若已经定义了裁剪平面，则去掉要裁剪的物体或得到物体的剖视图，此时产生裁剪坐标，定义了一个取景体，在其外的物体均裁剪掉，不显示在屏幕上，然后，把 x，y，z 坐标除以 w，得到归一化的设备坐标，再经过视见区变换把它转换成窗口坐标，可以通过改变视见区的尺寸改变图形的大小，图 3-2 中顶点变换的顺序是不能改变的。

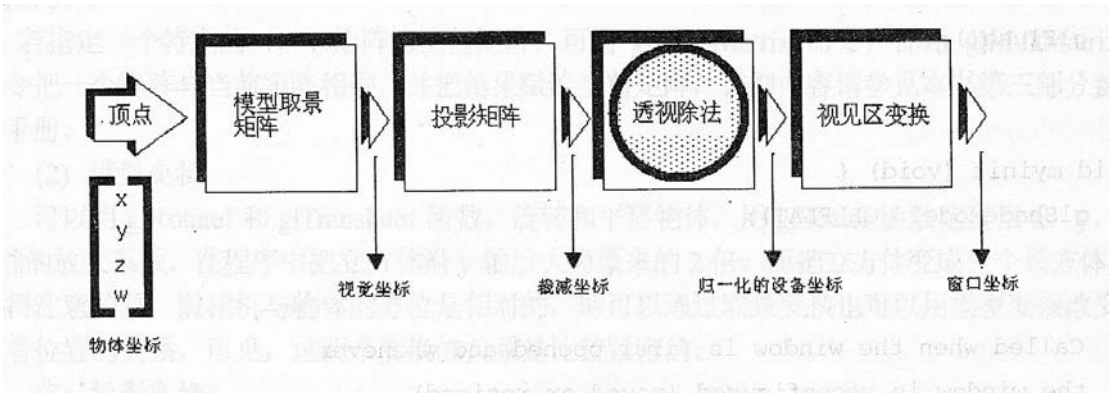

图 3—2 顶点变换的过程

值得注意的是，在 OpenGL 中，二维、三维顶点在内部都当成由 4 个坐标组成的三维齐次坐标(x，y，z，w)，若 α ≠ 0，(α x，α y，α z，α w)和(x，y，z，w)代表齐次坐标相同点，而(x，y，z，1.0)则是三维 Euler 坐标的一点(x，y，z)。若 w≠0，齐次坐标(x，y，z，w)代表 Euler 坐标(x/w，y/w，z/w)，若 w=0 则代表在无穷远处的点，若 w<0，有时 OpenGL 处理齐次裁剪平面时会弄错，因此应该尽量使用 w>0 的值。

3．1．2 OpenGL 的基本变换命令

本节将结合一个简单的程序简要地讲述 OpenGL 的基本变换命令及其使用方法，下面是一个在三维空间绘制立方体的程序(程序 3—1)。

程序 3—1

```
/* Prog3_1 */
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
```

```
void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

/*  Clear the screen.  Set the current color to white.
 *  Draw the wire frame cube.
 */
void CALLBACK display(void)
{
      glClear(GL_COLOR_BUFFER_BIT);
      glColor3f(1.0, 1.0, 1.0);
      glLoadIdentity();   /*  clear the matrix    */
      glTranslatef(0.0,  0.0,  -2.0);  /*  viewing transformation  */
      glScalef(1.0,  1.0,  1.0);   /*  modeling transformation */
      auxWireCube(1.0);   /*  draw the cube   */
      glFlush();
}

void myinit(void)
{
      glShadeModel(GL_FLAT);
}

/*  Called when the window is first opened and whenever
 *  the window is recenfigured (moved or resized).
 */
void CALLBACK myReshape(GLsizei w, GLsizei h)
{
      glMatrixMode(GL_PROJECTION);   /*  prepare for and then  */
      glLoadIdentity();  /*  define the projection  */
      gluPerspective(70.0, 2.0, 1.5, 40.0);    /*  transformation  */
      glMatrixMode(GL_MODELVIEW);    /*  back to modelview matrix    */
      glViewport(0,  0, w, h);    /*  define the viewport */
}

/*  Main Loop
 *  Open window with initial window size,  title bar,
 *  RGBA display mode,  and handle input events.
 */
int main(int argc,  char** argv)
{
     auxInitDisplayMode(AUX_SINGLE | AUX_RGB);
     auxInitPosition(0,  0,  500,  500);
```

```
    auxInitWindow("Perspective 3-D Cube");
    myinit();
    auxReshapeFunc(myReshape);
    auxMainLoop(display);
    return(0);
}
```

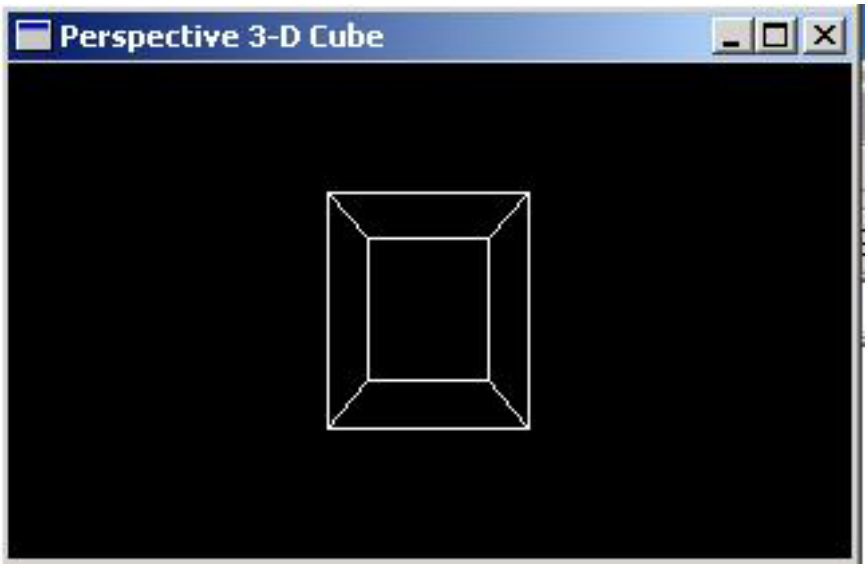

图 3-3 3D 透视立方体

(1)取景变换

在程序中，定义取景变换前，应该把当前矩阵设置成单位阵，即调用 glLoadIdentity 函数，这是在做变换操作前必需的一步，因为变换就是把指定的矩阵乘以当前矩阵，再把这个结果赋给当前矩阵，若当前矩阵不是单位阵，则所得的值必然会包括以前变换的结果。

完成矩阵初始化后，glTranslatef 函数定义在 x，y，z 方向如何移动照相机，在程序中向负 z 轴移动 5 个单位长度，缺省时，照相机和物体均位于原点，glRotatef 函数改变照相机的方向。

若指定一个特定的 4×4 矩阵为当前矩阵，可用 glLoadMatrix 命令，而用 glMultMatrix 命令把一个矩阵与当前矩阵相乘，并把结果赋给当前矩阵，详细内容请参见本书第三部分参考手册。

(2)模型变换

可以用 glRotatef 和 glTranslatef 函数，旋转和平移物体，用 glScalef 函数定义沿 x，y，z 轴的放大系数，在程序中把立方体沿 y 轴放大为原来的 2 倍，而把立方体变成一个长方体。值得注意的是，照相机与物体的方位是相对的，既可以通过取景变换也可以通过模型变换改变两者的位置关系。可见，这两个变换的关系是比较紧密的。

(3)投影变换

投影变换除了确定观察范围外，还决定物体投影到屏幕的方式，OpenGL 提供透视投影和正交投影两种方式，同时提供以不同方式描述相应参数的命令，透视投影遵守物体近大远小的投影准则，与景物在照相机底片上的投影相同，在程序中 glFrustum 命令定义了透视投影。而正交投影直接把物体投影到屏幕上，不改变其相对尺寸，常用于建筑和机械 CAD 绘图，反映出物体的真实尺寸。

在程序中，调用 glFrustum 函数之前，要在 myReshape 子程序中调用

glMatrixMode(GL_PROJECTION)，表明当前矩阵操作是针对投影变换的，其后的变换只影响投影变换，而 glMatrixMode(GL_MODELVIEW)则指明是模型取景矩阵。

(4)视见区变换

投影变换和视见区变换都决定三维场景映射到屏幕上的方式，投影变换确定映射的机理，视见区表明映射到屏幕的可见形状，glViewport 函数给出可见屏幕范围的原点和宽、高，均以像素为单位，在程序中原点为(0，0)，宽和高则由主程序根据窗口的大小传入。

完成上述所有变换的定义后，OpenGL 对物体的每个顶点进行模型和取景变换，然后经过投影变换，裁剪掉在取景范围外的物体，把变换后的顶点除以 w，映射到视见区。

3.2 取景和模型变换

由于变换的实质就是用变换矩阵 M 乘以当前矩阵 C，使 C=CM，从而改变当前矩阵的值，很显然，若同时做多次变换操作，则要做多次矩阵乘法，当前矩阵 C 的值会因为变换顺序的不同而不同，因此，变换的顺序很重要，例如，对一个物体先旋转再平移的效果与先平移再旋转的效果截然不同，如图 3-4 所示，针对第一种情况下，程序应该是：

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultiMatrix(T);     /* translation */
glMultiMatrix(R);     /* rotation */
draw_the_object();
```

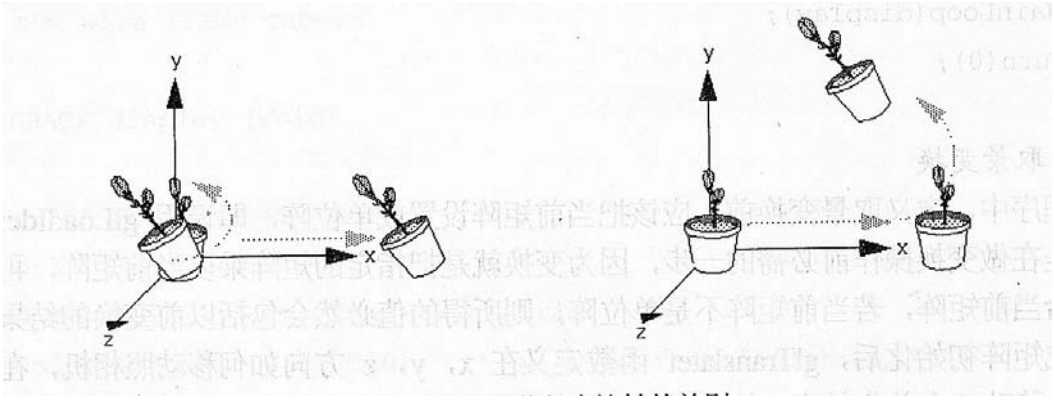

图 3—4　先平移与先旋转的差别

若在以地面为参照的坐标系中，这时当前的矩阵为 C=ITR=TR，而物体的每个顶点坐标矢量 v 变换或 v′=TRv=T(Rv)，是与程序中出现的顺序相反进行的；若在与物体固连的坐标系中，变换的顺序则与程序中出现的顺序是相同的。这是由参考坐标不同造成的，但是程序是相同的，因此在编程时，这一点应该引起注意。

由上一节的内容可知，取景变换和模型变换是相互关联的，下面将结合具体例子详细讲述模型变换和取景变换。

3．2．1　模型变换

程序 3-2 用于对三角形模型变换，按照变换的次序，结果分别为实线三角形、虚线三角形、变形的三角形和旋转的三角形，分别对应模型变换的基本操作（见图 3-5）。

程序 3—2

```
/* Prog3_2.c */
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>

#include <GL/glaux.h>

void myinit(void);
void draw_triangle(void);
void CALLBACK display(void);
void CALLBACK myReshape(GLsizei w, GLsizei 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();
}

/*  Clear the screen.  For each triangle,  set the current
 *  color and modify the modelview matrix.
 */
void CALLBACK display(void)
{
      glClearColor(0.0, 0.0, 0.0, 1.0);
      glClear(GL_COLOR_BUFFER_BIT);
      glLoadIdentity();
      glColor3f(1.0,  1.0,  1.0);
      draw_triangle();
      glEnable(GL_LINE_STIPPLE);
      glLineStipple(1, 0xF0F0);
      glLoadIdentity();
      glTranslatef(-20.0,  0.0,  0.0);
      draw_triangle();
      glLineStipple(1,  0xF00F);
      glLoadIdentity();
      glScalef(1.5,  0.5,  1.0);
      draw_triangle();
      glLineStipple(1,  0x8888);
      glLoadIdentity();
```

```
      glRotatef(90.0,  0.0,  0.0,  1.0);
      draw_triangle();
      glDisable(GL_LINE_STIPPLE);
      glFlush();
}

void myinit(void)
{
       glShadeModel(GL_FLAT);
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
      if  (!h)  return;
      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);
}

/*  Hain Loop
 *  Open window with initial window size,  title bar,
 *  RGBA display mode, and handle input events.
 */
int main(int argc,  char** argv)
{
     auxInitDisplayMode(AUX_SINGLE | AUX_RGB);
     auxInitPosition(0,  0,  500,  500);
     auxInitWindow("Modeling Transformations");
     myinit();
     auxReshapeFunc(myReshape);
     auxMainLoop(display);
     return(0);
}
```

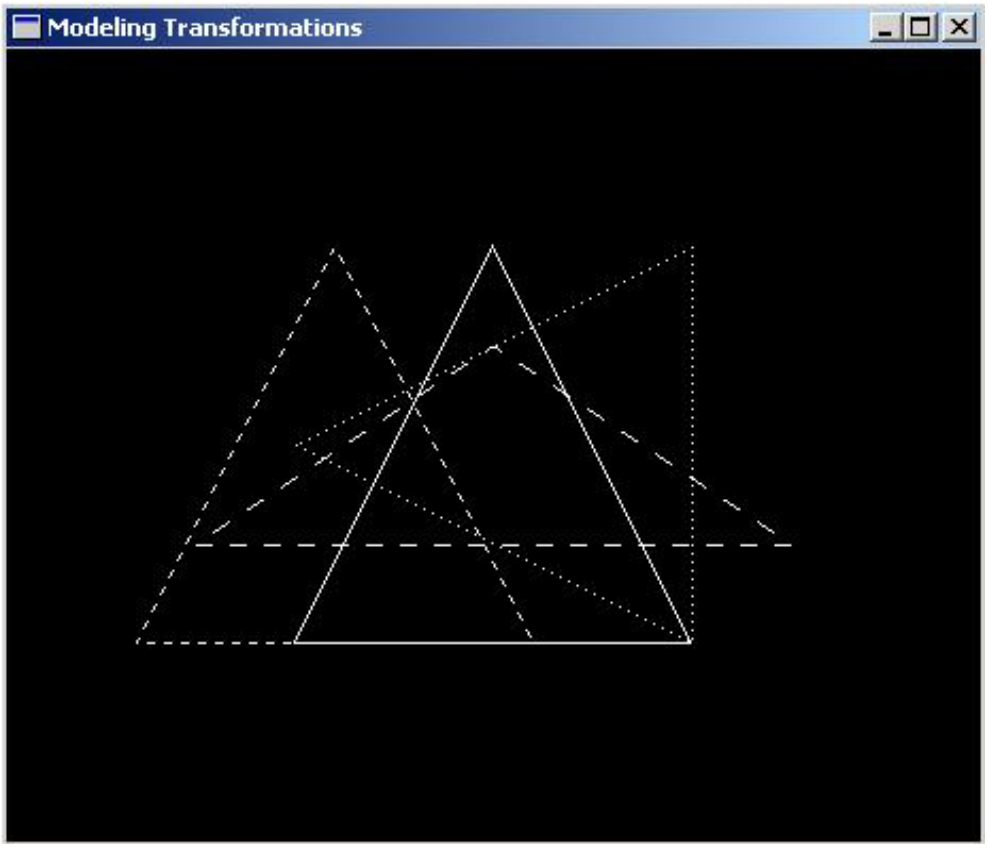

图 3-5　三角形模型变换

在上述例子中，用于模型变换的命令如下：

① void glTranslate{fd}(TYPE x，TYPE y，TYPE z)；

把物体在 x，y，z 轴上分别移动 x，y，z 个长度单位。

② void glRotate{fd}(TYPE angle，TYPE x，TYPE y，TYPE z)

绕矢量$(x，y，2)^T$逆针时旋转 angle 度。

③ void glScale{fd}(TYPE x，TYPE y，TYPE z)

把物体沿 x，y，z 轴分别放大为原来的 x，y，z 倍，若其值为负数，则把物体放置在相应坐标轴的相反方向，再放大。另外值得注意的是，只有在必要时才放大，否则会在做光照计算时，重新把法线向量归一化，而降低绘图速度。

在程序中，分别把三角形沿-x 轴平移 20 个单位，单位长度在 x 方向放大为原来的 1．5 倍，y 方向为原来的 0．5 倍，z 方向不变，绕 z 轴方向逆时针旋转 90º，缺省的模型变换为物体不平移、不旋转、不放大，上述 3 个函数的详细内容请参见第三部分参考手册。

3．2．2　取景变换

取景变换改变观察点的位置和方向，包括平移和旋转两个操作，由于上面提到的变换作用顺序的原因，应该在模型变换前调用取景变换，以使模型变换先作用于物体。做取景变换的方法有如下几种。

① 用 glTranslate 和 glRotate 命令，相当于移动照相机参见 3．1．1 节，缺省的取景变换是观察点位于原点，方向指向负 z 轴。可以用 glTranslate 命令把观察点与物体分开，参见图 3-6，把物体向后移(沿负 z 轴)，或等价地把照相机前移，用 glRotate 命令观察物体的侧面，这样，就完成了取景变换。

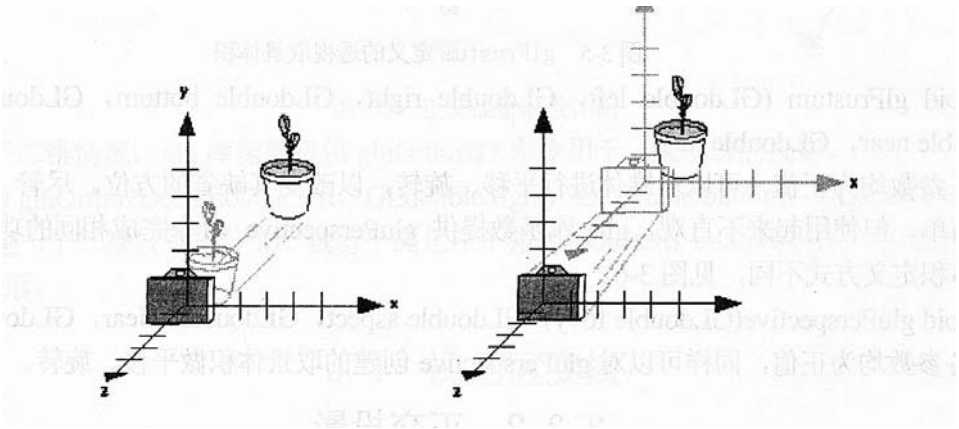

图 3—6　glTranslate 命令分开观察点与物体

② 用 gluLookAt 函数，这是一个 glu 库函数，由一系列平移和旋转命令构成，可以在空间任一点观察场景，由三维参数定义观察点的位置，定义照相机所对准的参考点和向上的方向，这个函数用于浏览场景时很有用处，使取景体积在 x，y 方向均对称，(eyex，eyey，eyez) 为屏幕上图象的中心，不断改变观察点，即可浏览场景，详细内容请参见本书的第三部分。

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

④ 创建自己的子程序，对于一些特殊场合，如飞行模拟器，可能需要比较复杂的取景变换，而这些变换又要经常引用，这时可以创建自己的取景子程序。

3.3 投影变换

投影变换就是要确定一个取景体积，其作用有两个：

. 确定物体投影到屏幕的方式，即是透视投影还是正交投影。

. 确定从图形上裁剪掉哪些物体或物体的某一部分。

下面，分别讲述透视投影和正交投影。

3．3．1　透视投影

透视投影的示意图见图 3-7，其取景体积是一个截头锥体，在这个体积内的物体投影到锥的顶点，由 glFrustum 命令定义这个截头锥体，这个取景体积可以是不对称的，计算透视投影矩阵 M，并乘以当前矩阵 C，使 C=CM。

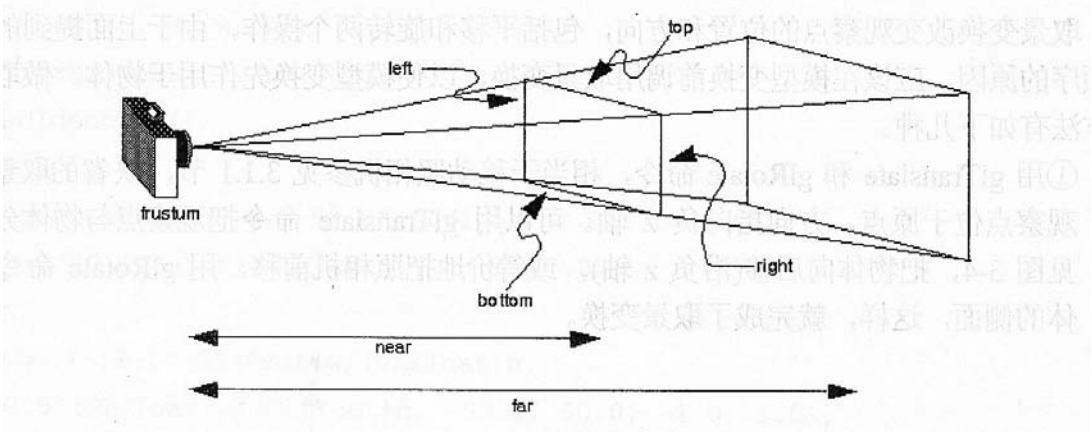

图 3—7　glFrustum 定义的透视取景体积

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

各参数均为正值，可以对锥体进行平移、旋转，以改变其缺省的方位。尽管 glFrustum 概念简单，但使用起来不直观，glu 库函数提供 gluPerspective 命令完成相同的功能，但对取景体积定义方式不同，见图 3-8。

Void gluPerspective(Gldouble fovy, Gldouble aspect, Gldouble zNear, Gldouble zFar)；各参数均为正值，同样可以对 gluPerspective 创建的取景体积做平移、旋转。

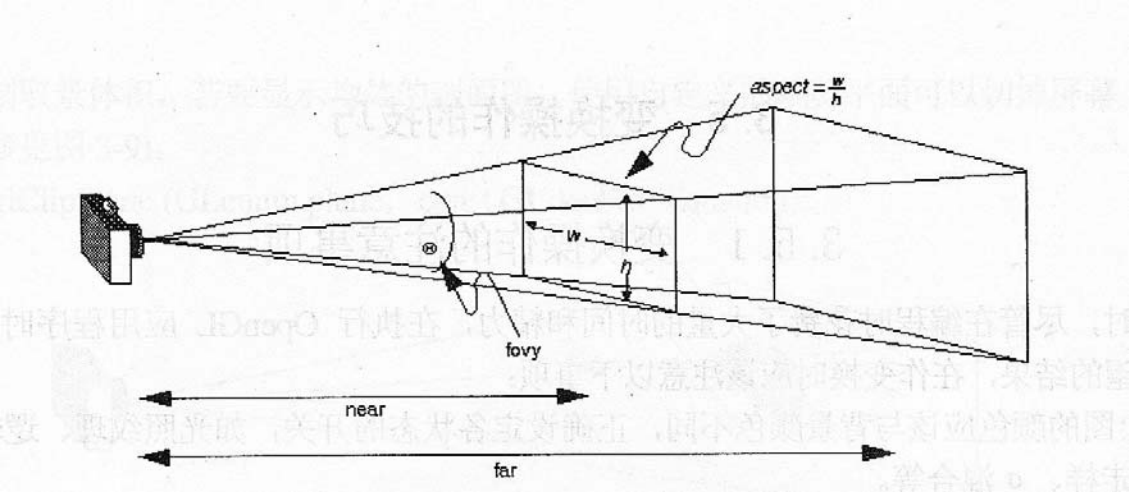

图 3—8　gluPerspective 定义的透视取景体积

3．3．2　正交投影

正交投影的示意图见图 3-9，其取景体积是一个各面均为矩形的六面体，用 glOrtho 命令创建正交平行的取景体积，计算正交平行取景体积矩阵 M，并乘以当前矩阵 C 使 C=CM，值得注意的是，若 z=far,near 的平面在观察点前，为负值，否则为正值。

Void glOrtho(Gldouble left, Gldouble right, Gldouble bottem, Gldouble top, Gldouble near, Gldouble far);

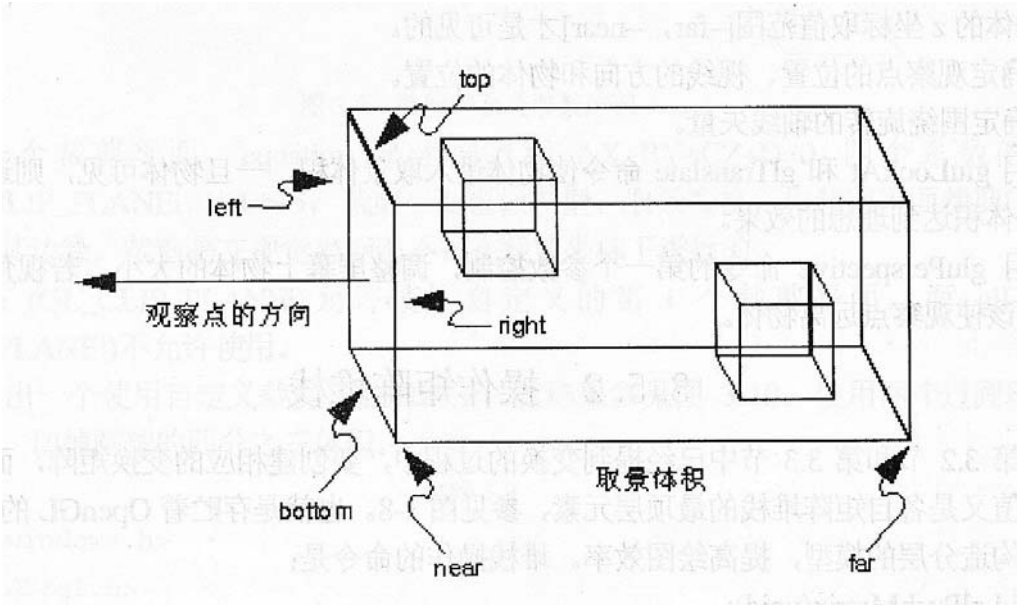

图 3-9　正交投影取景体积

对于二维情况，glu 库函数提供 gluOrtho2D 命令用于二维图形的投影。

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

创建一个二维投影矩阵 M，裁剪平面是左下角坐标为(left，buttom)、右上角坐标为(right，top)的矩形。

3．4　视见区变换

用窗口管理器在屏幕上打开一个窗口时，已经自动地把视见区设为整个窗口的大小，可以用 glViewport 命令选定一个较小的绘图区，利用这个命令可以在同一窗口上同时显示多个视图，达到分屏显示的目的。

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

初始视见区为(0，0，width，height)，应该使视见区的长宽比与取景体积的长宽比相等，否则会使图像变形，在程序中要及时接入窗口变化的事件，正确调整视见区。

在视见区变换时，保留了 z 坐标，可以用 glDepthRange 命令把 z 值放大到所需的范围，与 x,y 窗口坐标不同，OpenGL 把 z 窗口坐标的取值范围按[0，1]处理。

void glDepthRange(GLclampd near，GLclampd far)；

near，far 值是存入深度缓冲区的最小、最大值调整量，缺省时，分别为 0.0、1.0。

3．5　变换操作的技巧

3．5．1　变换操作的注意事项

有时，尽管在编程时花费了大量的时间和精力，在执行 OpenGL 应用程序时可能得不到所希望的结果，在作变换时应该注意以下事项：

. 绘图的颜色应该与背景颜色不同，正确设定各状态的开关，如光照纹理、逻辑操作、反走样、α 混合等。

. 使用投影命令时，缺省的视线方向为沿负 z 轴，near，far 值是物体与观察点的距离，物体的 z 坐标取值范围[-far，-near)才是可见的。

. 确定观察点的位置、视线的方向和物体的位置。

. 确定围绕旋转的轴线矢量。

. 用 gluLookAt 和 glTranslate 命令使物体进入取景体积，一旦物体可见，则逐步调整取景体积达到理想的效果。

. 用 gluPerspective 命令的第一个参数控制，调整屏幕上物体的大小，若视角过大，则应该使观察点远离物体。

3．5．2 操作矩阵堆栈

在第 3．2 节和第 3．3 节中已经提到变换的过程中，要创建相应的变换矩阵，而这些矩阵的当前值又是各自矩阵堆栈的最顶层元素，参见图 3-10。也就是存贮着 OpenGL 的一些状态，　有助于构造分层的模型，提高绘图效率。堆栈操作的命令是：

void glPushMatrix(void)；

把当前堆栈中所有矩阵向下推一层，当前堆栈的类型由 glMatrixMode 的参数决定，最顶层矩阵由其下一层矩阵复制，若堆栈满，则出错。

void glPopMatrix(void)；　　.

弹出最顶层矩阵，其下一层的矩阵成为最顶层矩阵，若堆栈中只有一个矩阵，则出错。

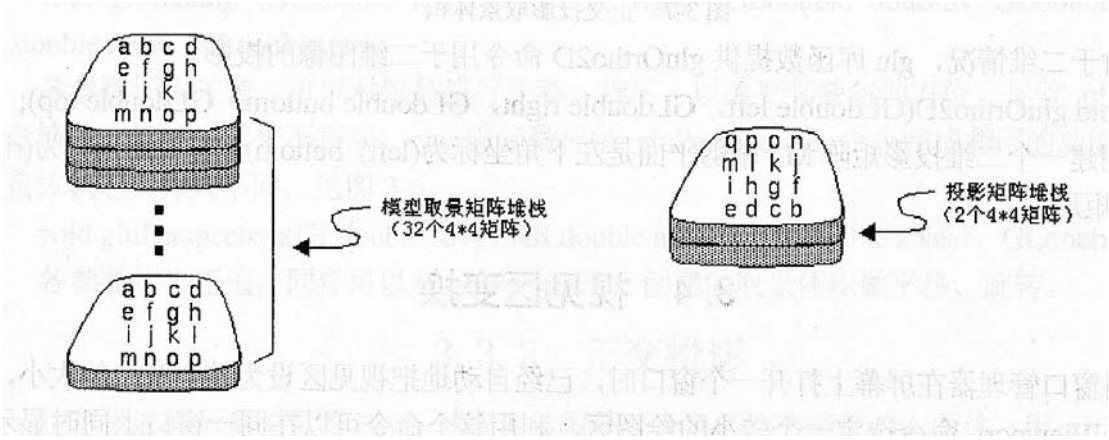

图 3—10 矩阵堆栈示意图

3．5．3 裁剪平面

在取景体积中已经使用了六个裁剪平面(参见图 3-7)，用户可以定义至多 6 个裁剪平面进一步限制取景体积，若要显示物体的剖面图，使用自定义的裁剪平面可以切掉屏幕上多余的部分(参见图 3-11)。

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

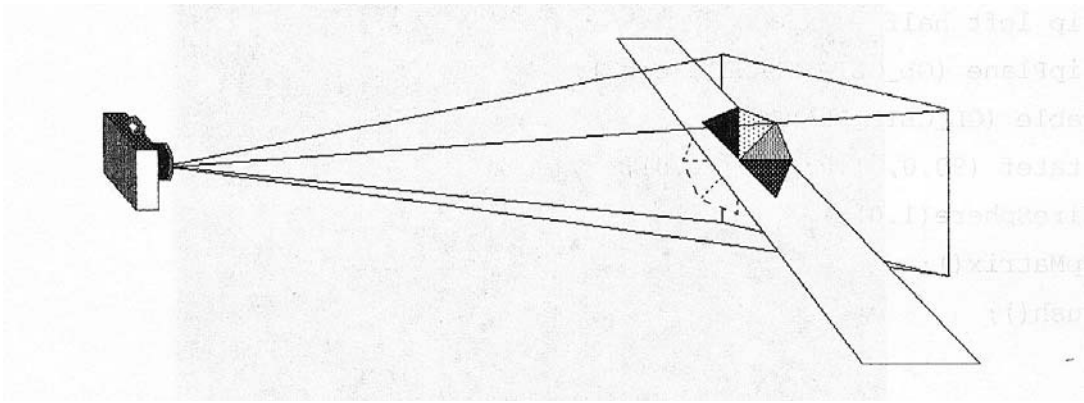

图 3—11 裁剪平面及取景体积

定义一个裁剪平面，equation 是平面方程 AX+BY+CZ+D=0 四个系数的指针 plane=GL_CLIP_PLANEi，i=0—5，裁剪平面也做模型、取景变换，由裁剪平面裁剪的多边形自动重构其边缘，裁剪是在视觉坐标而不是在裁剪坐标下进行的。

由 glEnable(GL_CLIP_PLANEi)允许使用自定义的第 i 个裁剪平面，而 glDisable (GL_CLIP_PLANEi)则禁止使用。

下面给出一个使用自定义裁剪平面的例子，其结果参见图 3—12，使用两个过圆球中心的裁剪平面，切掉圆球的四分之三体积。

程序 3—3

```
/* Prog3_3.c */
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
```

```
#include <GL/glaux.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

void CALLBACK display(void)
{
      GLdouble eqn[4] = {0.0, 1.0, 0.0, 0.0};
      GLdouble eqn2[4] = {1.0, 0.0, 0.0, 0.0};
      glClear(GL_COLOR_BUFFER_BIT);

      glColor3f(1.0,  1.0,  1.0);
      glPushMatrix();
      glTranslatef(0.0, 0.0, -5.0);

/*      clip lower half -- y < 0       */
      glClipPlane(GL_CLIP_PLANE0,  eqn);
      glEnable (GL_CLIP_PLANE0);
/*      clip left half -- x < 0        */
      glClipPlane(GL_CLIP_PLANE1, eqn2);
      glEnable(GL_CLIP_PLANE1);
      glRotatef(90.0,  1.0,  0.0,  0.0);
      auxWireSphere(1.0);
      glPopMatrix();
      glFlush();
}

void myinit(void)
{
      glShadeModel(GL_FLAT);
}

void CALLBACK myReshape(GLsizei w,  GLsizei h)
{
      if  (!h)  return;
      glViewport(0,  0, w, h);
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      gluPerspective(60.0,  (GLfloat)w/(GLfloat)h,  1.0,  20.0);
      glMatrixMode(GL_MODELVIEW);
}

/*  Main Loop
```

```
 *  Open window with initial window smze,  title bar,
 *  RGBA display mode, and handle input events.
 */

int main(int argc, char** argv)
{
      auxInitDisplayMode(AUX_SINGLE  | AUX_RGB);
      auxInitPosition(0,  0,  500,  500);
      auxInitWindow ("Arbitrary Clipping Planes");
      myinit();
      auxReshapeFunc(myReshape);
      auxMainLoop(display);
      return(0);
}
```

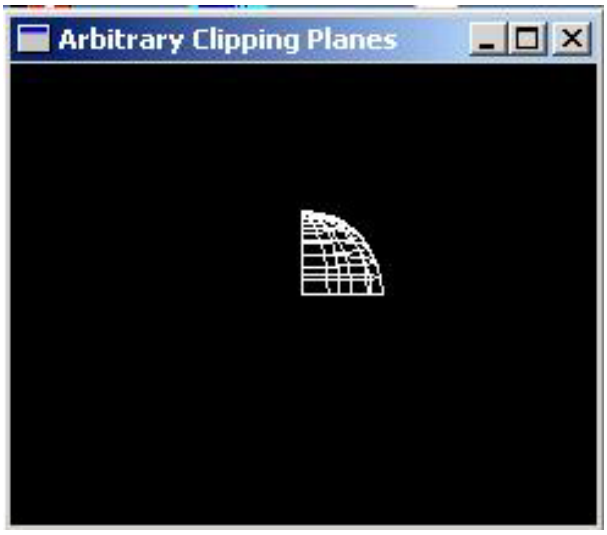

图 3-12　经过裁剪的圆球线框图

3.6　应用变换的一个实例

本节将综合本章前面几节的内容，给出一个应用各种变换的例子。通过分析这个例子，有助于加深理解前面讲述的各部分内容，以便在创建自己的应用程序中灵活使用这些变换命令。

程序 3-4 用于创建一个简单的机器人手臂，手臂的各部分通过转轴联接起来，可以活动，如图 3-13 所示。通过进一步扩展程序，可以绘制出带有多个手指的机器人手臂，如图 3-14 所示。需要注意的是，程序中用到变换命令较多，分别针对各个对象，要清楚这些变换的相互关系，在绘制手指时，要使用 glPushMatrix 和 glPopMatrix 保存和恢复变换矩阵，可以在不同的位置绘制相同的几何形状，这些对于复杂程序设计都是有益的。

程序 3-4

```
/* Prog3_4.c */
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
```

```
#include <GL/glaux.h>

void myinit(void);
void drawPlane(void);
void CALLBACK elbowAdd(void);
void CALLBACK elbowSubtract(void);
void CALLBACK shoulderAdd(void);
void CALLBACK shoulderSubtract(void);
void CALLBACK display(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);

static int shoulder = 0, elbow = 0;

void CALLBACK elbowAdd(void)
{
      elbow =  (elbow + 5)  % 360;
}

void CALLBACK elbowSubtract(void)
{
      elbow =  (elbow - 5)  % 360;
}

void CALLBACK shoulderAdd(void)
{
      shoulder =  (shoulder + 5)  % 360;
}

void CALLBACK shoulderSubtract(void)
{
      shoulder = (shoulder - 5) % 360;
}

void CALLBACK display(void)
{
      glClear(GL_COLOR_BUFFER_BIT);
      glColor3f(1.0, 1.0, 1.0);
      glPushMatrix();
      glTranslatef(-1.0,  0.0,  0.0);
      glRotatef((GLfloat)shoulder, 0.0, 0.0, 1.0);
      glTranslatef(1.0,  0.0,  0.0);
      auxWireBox(2.0,  0.4,  1.0);
      glTranslatef(1.0,  0.0,  0.0);
      glRotatef((GLfloat)elbow,  0.0,  0.0,  1.0);
```

```
	glTranslatef(1.0,  0.0,  0.0);
	auxWireBox(2.0,  0.4,  1.0);
	glPopMatrix();
	glFlush();
}

void myinit(void)
{
	glShadeModel(GL_FLAT);
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
	if  (!h) return;
	glViewport(0,  0, w, h);
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	gluPerspective(65.0,  (GLfloat)w/(GLfloat)h,  1.0,  20.0);
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
	glTranslatef(0.0,  0.0,  -5.0);  /* viewing transform  */
}

 /*  Main Loop
  *  Open window with initial window size,  title bar,
  *  RGBA display mode, and handle input events.
  */
int main(int argc,  char** argv)
{
	auxInitDisplayMode(AUX_SINGLE | AUX_RGB);
	auxInitPosition(0,  0,  400,  400);
	auxInitWindow("Composite Modeling Transformations");
	myinit();
	auxKeyFunc(AUX_LEFT,  shoulderSubtract);
	auxKeyFunc(AUX_RIGHT, shoulderAdd);
	auxKeyFunc(AUX_UP, elbowAdd);
	auxKeyFunc(AUX_DOWN,  elbowSubtract);
	auxReshapeFunc(myReshape);
	auxMainLoop(display);
	return(0);
}
```

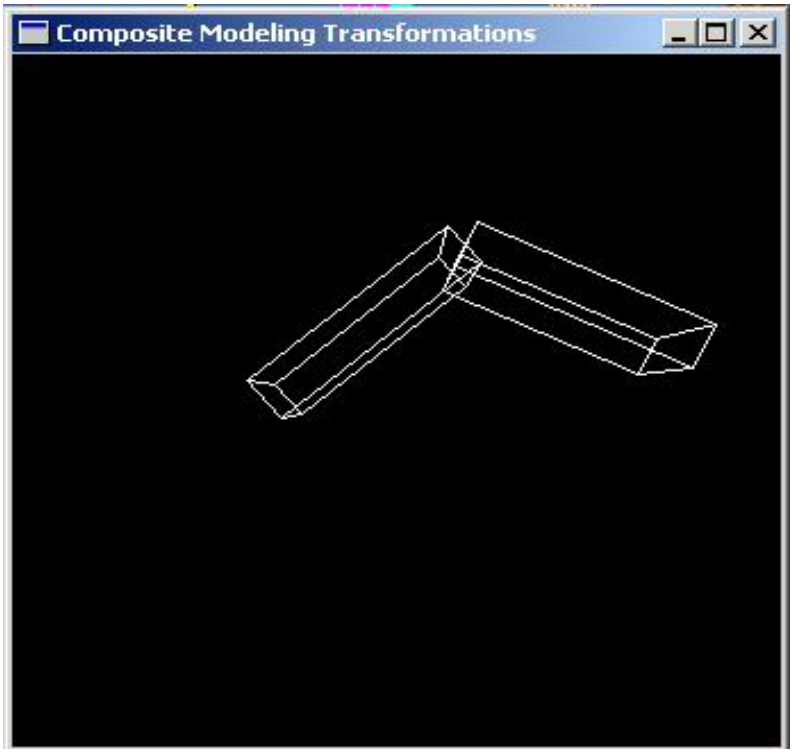

图 3-13 简单的机器人手臂

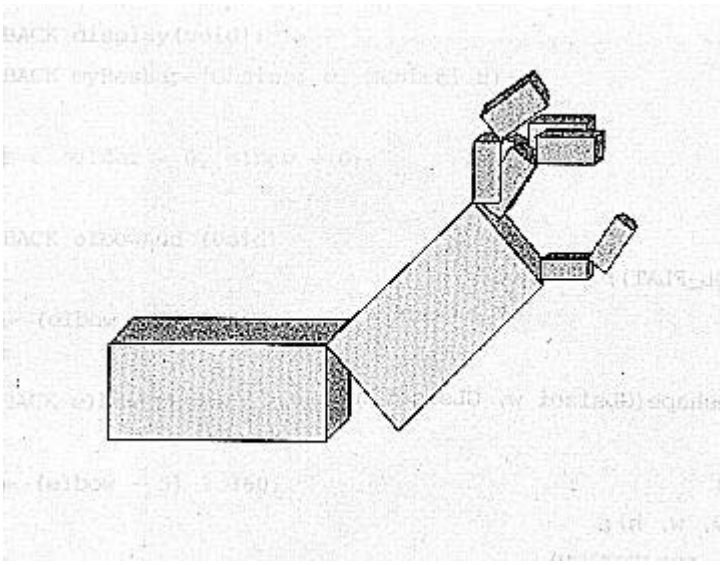

图 3-14　带有手指的机器人手臂

第四章　显示列表

显示列表是一组存贮起来的 OpenGL 命令。引用显示列表时，按顺序执行其中的 OpenGL 命令，可以在程序中不同地方使用这些命令。与子程序不同的是，这些命令是经过编译的，执行效率高，从而可以有效地提高 OpenGL 的绘图性能。前面几章用到的 OpenGL 命令都是立即执行模式的，本章将讨论如何使用显示列表功能，这一概念在较深入地研究更复杂的 OpenGL 功能如字符、位图等时经常用到。下面将结合具体实例，从显示列表的基本概念出发，讲述如何创建、管理、执行显示列表，并给出一些使用技巧。

4.1　显示列表的基本概念

以绘制由 100 条线段近似的圆为例，若按以往的思想，用立即执行模式程序可以写成：

```
drawCircle()
{
    GLinit i;
    GLfloat cosine,   sine;
    glBegin(GL_POLYGON);
        for(i=0;   I<100;   I++)  {
            cosine=cos(i*2*PI / 100.0);
            sine=sin(i*2*PI / 100.0);
            glVertex2f(cosine, sine);
        }
    glEnd();
}
```

若要用这段程序绘制多个圆，每次都要计算三角函数，从而浪费大量时间，降低绘图效率。若把上述顶点坐标存入一个数组，可以改善一下情况，但是在内存中查找变量及下标也会浪费机时，而显示列表的优势在于，OpenGL 只需绘制一个圆，就会绘制其他同样的圆。也就是说，在这种情况下，使用显示列表是既方便又经济的方法，具体的程序如下：

```
#define MY_CIRCLE_LIST 1
buildCircle()
{
    GLinit i;
    GLfloat cosine, sine;
    glNewList(MY_CIRCLE_LIST, GL_COMPILE);
        glBegin(GL_POLYGON);
            for(i=0; i<100; i++)  {
                cosine = cos(i*2*Pi/100.0);
                sine = sin(i*2*Pi/100.0);
                glVertex2f(cosine, sine);
            }
        glEnd();
    glEndList();
}
```

在 glNewlist 与 glEndList 命令之间定义了一个名为 MY_CIRCLE_LIST 的显示列表名称，可以用 glCallList(MY_CIRCLE_LIST)调用，经过编译，在显示列表中只保存坐标和其他变量的值，而不包括三角函数的正弦、余弦。显示列表一经编译，这些值不能改变，可以删除显示列表，创建一个新的显示列表，但不能编辑一个已有的显示列表。

OpenGL 显示列表是一系列命令的高速缓存，而不是内存中的动态数据库，不必进行内存管理，使绘图性能大幅度提高，即使使用显示列表只绘制一个简单图形，也不会比立即执行模式的速度慢。对于在网络上运行的程序，由于显示列表驻留在服务器，减少了网络上的传输，速率提高幅度更大，而对当地机器，绘图命令更能充分发挥硬件的性能，使用显示列表的效率也会提高，因为 OpenGL 的实现不同，在不同的硬件上执行同一条命令效率会相差很多。当然若显示列表很短，也不能体现出优势，而在下面几种情况下，会优化性能。

. 矩阵操作。由于多数矩阵要求 OpenGL 计算矩阵的逆阵，每次计算要用大量的时间，对于某个特定的 OpenGL 实现会把矩阵和其逆阵存入显示列表。

. 光栅位图和图象。定义光栅数据的格式可能不完全发挥硬件性能。而编译显示列表后，数据会转变成适用于硬件的格式，这时会大幅度提高绘图速度。

. 光照、材料属性。在光照模型中，若在场景中布置有复杂的光照条件，设置材料需要大量的计算，在显示列表中存入材料的定义，因为存贮了计算结果，不必每次切换材料时都重复这些计算，这样也会提高绘图速度。

. 纹理。用显示列表可以最大限度地提高绘图效率，因为显示列表会在编译时把纹理定义成硬件纹理格式，不必在显示时转换这些数据。

. 多边形点图案。

4．2　创建并执行一个显示列表

本节举例说明如何创建并调用一个显示列表，在程序 4-1 中，glNewList 与 glEndList 之间创建一个红色三角形，然后把坐标系原点平移到(1．5，0．0)处，在 display 子程序中，由 glCallList 调用已经建立的显示列表，绘制十个三角形，由于受到坐标系平移的作用，绘制的直线位置也发生变化，参见图 4—1，也就是说，在显示列表中调用的变换命令，会对程序中以后的命令起作用，这一点是应该引起注意的。

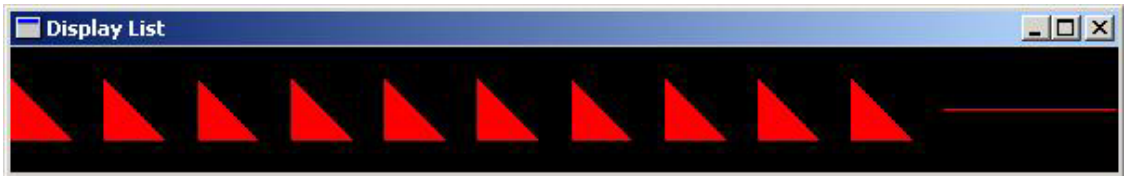

图 4—1　用显示列表绘制一系列三角形

(1)创建一个显示列表

```
void glNewList(Gluint list GLenum mode);
void glEndlist(void);
```

两个函数成对出现，分别标志显示列表定义的开始和结束，两者之间的命令被定义成为一个由 list 标识的显示列表，若把这些命令放入显示列表而不执行时，mode 为 GL_COMPILE，而把这些命令放入显示列表同时立即执行时，mode 为 GL_COMPILE_AND_EXECUTE。

创建显示列表时要注意，显示列表只存贮数值，在创建显示列表后，即使所引用的数值改变，也不会影响显示列表的内容，这是因为显示列表是一系列 OpenGL 命令的高速缓存，而不是一个动态数据库(见上一节内容)；对于 glNewList 与 glEndList 之间调用的 OpenGL 函数

是有限制的，下列命令不能在创建显示列表时使用，glDeleteLists，glIsEnabled，glFeedbackBuffer, glIsList，glFinish，glFlush，glGetLists，glGet，glPixelstore，glReadPixels，glRenderPixels, glSelectBuffer。

(2)执行一个显示列表　　”

在创建显示列表后，可以用 glCallList 调用标识为 list 的显示列表，可以对同一个显示列表调用多次。

void glCallList(Gluint list);

调用时，显示列表中的命令按排列的顺序执行，若 list 没有定义，不进行任何操作。本节的程序中，调用显示列表，改变了 OpenGL 的状态，若不希望在执行显示列表后改变状态，则要分别用 glPushMatrix 和 glPopMatrix 存贮和恢复当前矩阵，此时，程序 4-2 中的显示列表定义改为：

程序 4-2

```
/* Prog4_2.c */
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void drawLine(void);
void CALLBACK display(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);

GLuint listName;

void myinit(void)
{
  GLfloat color_vector[3]  = {1.0, 0.0, 0.0};
  listName = glGenLists(1);
  glNewList(listName,  GL_COMPILE);
    glPushAttrib(GL_CURRENT_BIT);
      glColor3fv(color_vector);
      glBegin(GL_TRIANGLES);
        glVertex2f(0.0,  0.0);
        glVertex2f(1.0,  0.0);
        glVertex2f(0.0,  1.0);
      glEnd();
      glTranslatef(1.5,  0.0,  0.0);
    glPopAttrib();
  glEndList();
  glShadeModel(GL_FLAT);
}
```

```
void drawLine(void)
{
  glBegin(GL_LINES);
    glVertex2f(0.0,  0.5);
    glVertex2f(15.0, 0.5);
  glEnd();
}

void CALLBACK display(void)
{
  GLuint i;
  GLfloat new_color[3]  = {0.0,  1.0,  0.0};

  glClear(GL_COLOR_BUFFER_BIT);
  glColor3fv(new_color);
  glPushMatrix();
    for (i = 0;  i < 10;  i++)
        glCallList(listName);
  glPopMatrix();
  drawLine();
  glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
  if (!h) return;
  glViewport(0,  0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  if (w <= h)
     gluOrtho2D(0.0, 2.0, -0.5*(GLfloat)h/(GLfloat)w,
                          1.5*(GLfloat)h/(GLfloat)w);
  else
    gluOrtho2D(0.0, 2.0*(GLfloat)w/(GLfloat)h, -0.5, 1.5);
  glMatrixMode(GL_MODELVIEW);
}

 /*  Main Loop
  *  Open window with initial window size,  title bar,
  *  RGBA display mode, and handle input events.
  */
int main(int argc,  char** argv)
{
```

```
  auxInitDisplayMode(AUX_SINGLE | AUX_RGB);
  auxInitPosition(0, 0, 400, 50);
  auxInitWindow("Display List");
  myinit();
  auxReshapeFunc(myReshape);
  auxMainLoop(display);
  return(0);
}
```

直线的位置不会因为调用显示列表而改变，参见图 4—2。一旦创建了显示列表，就可以通过显示列表的标识，在程序的任何地方调用它。

图 4—2　修改后的显示列表

(3)分层的显示列表

在 glNewList 与 glEndList 之间，可以用 glCallList 调用其它已经创建的显示列表，即显示列表是可以嵌入的，若调用尚未定义的显示列表，不做任何操作。对于绘制由多个部件组合成的物体，尤其是在物体中某些部件多次使用时，很有用处，嵌套的层次至少是 64，可以用 glGetIntegerv(GL_MAX_LIST_NESTING, GLint *data)查询最大嵌套次数，利用这个特性，可以通过重新定义嵌套的底层显示列表，改动上一层显示列表的内容，可以改变显示列表 1，2，3 的内容，改变显示列表 4 绘制的多边形形状。但是应该看到，这样做并不会使系统绘图效率提高。

```
        glNewList(1, GL_COMPILE);
              glVertex3f(v1);
        glEndlist();
        glNewList(2, GL_COMPILE);
              glVertex3f(v2);
        glEndlist();
        glNewList(3, GL_COMPILE);
              glVertex3f(v3);
        glEndlist();
        glNewList(4, GL_COMPILE);
              glBegin(GL_POLYGON);
                    giCallList(1);
                    glCallList(2);
                    glCallList(3);
              glEnd();
        glEndlist();
```

4．3　进一步使用显示列表遇到的问题

本章前面提到的内容只是基本的知识，在实际使用时，情况会复杂得多。本节将结合使用中遇到的一些问题，详细介绍显示列表的使用技巧。

(1)显示列表的索引

显示列表的索引是一个任意正整数，若创建显示列表时，使用的索引已经是一个有定义的值，则所对应的已有的显示列表将被覆盖，为防止发生这种事情，用 glGenLists 命令生成一个未用过的索引，用 glIsList 判定这个索引是否正在使用，而 glDeleteLists 可以删除一个或多个显示列表。

Gluint glGenLists(Glsizei range)

Glboolean glIsLists(Gluint list);

若 List 正在使用，返回值为 TRUE，否则为 FALSE。

void glDeleteLists(Gluint list,Glsizei range);　　.

删除从 list 开始的 range 个显示列表，若删除尚未创建的显示列表，将被忽略。删除后，上述的显示列表可以重新使用。　　·

(2)执行多个显示列表

若把显示列表指针存入数组，调用 glCallLists 函数，可以按顺序执行多个显示列表，在使用字体时，每个 ASCII 字符对应一个显示列表，并按字母顺序把这些显示列表排列好，则可以在 OpenGL 中使用该字体，用 glListBase 命令为第一个字符指定显示列表的索引。

void glListBase(Gluint base);

指定调用 glCallLists 函数时显示列表索引所加的偏移量，缺省值为 0，该命令不影响 glCallList 和 glNewlist 命令的使用。

void glCallLists(Glsizei n，GLenum type，const Glvoid *lists);

执行 n 个显示列表,这些显示列表的索引由 lists 指向的数组中的值加上当前偏移量(见 glListBase 函数)，type 为数据类型的索引数组元素的字节数，对多字节数据，从高位开始读取数据，详细内容参见本书的第三部分。

为说明执行多个显示列表的使用方法，给出一个绘制字符的程序 4-3，这些字符是由线段构成的，glGenLists 分配 128 个连续排列的显示列表，第一个索引由 glListBase 给出，每个字母对应一个显示列表，其索引是由偏移量与字符的 ASCII 值相加得到的，由 glCallLists 命令显示这些字符串。如图 4-3 所示。

程序 4—3

```
/* Prog4_3.c */
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

typedef struct charpoint
{
       GLfloat x, y;
       int     type;
}CP;
```

```
void myinit(void);
void drawLetter(CP *l);
void printStrokedString(char *s);
void CALLBACK display(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);

#define PT 1
#define STROKE 2
#define END 3
CP Adata[] = {{0, 0, PT}, {0, 9, PT}, {1, 10, PT}, {4, 10, PT},
              {5, 9, PT}, {5, 0, STROKE}, {0, 5, PT}, {5, 5, END}};
CP Edata[] = {{5, 0, PT}, {0, 0, PT}, {0, 10, PT}, {5, 10, STROKE},
              {0, 5, PT}, {4, 5, END}};
CP Pdata[] = {{0, 0, PT}, {0, 10, PT}, {4, 10, PT}, {5, 9, PT}, {5, 6, PT},
              {4, 5, PT}, {0, 5, END}};
CP Rdata[] = {{0, 0, PT}, {0, 10, PT}, {4, 10, PT}, {5, 9, PT}, {5, 6, PT},
              {4, 5, PT}, {0, 5, STROKE}, {3, 5, PT}, {5, 0, END}};
CP Sdata[] = {{0, 1, PT}, {1, 0, PT}, {4, 0, PT}, {5, 1, PT}, {5, 4, PT},
              {4, 5, PT}, {1, 5, PT}, {0, 6, PT}, {0, 9, PT}, {1, 10, PT},
              {4, 10, PT}, {5, 9, END}};

/*  drawLetter()  interprets the instructions from the array
 *  for that letter and renders the letter with line segments.
 */
void drawLetter(CP *l)
{
  glBegin(GL_LINE_STRIP);
    while (1)  {
      switch (l->type)  {
         case PT:
            glVertex2fv(&l->x);
            break;
         case STROKE:
            glVertex2fv(&l->x);
            glEnd();
            glBegin(GL_LINE_STRIP);
            break;
         case END:
            glVertex2fv(&l->x);
            glEnd();
            glTranslatef(8.0,  0.0,  0.0);
            return;
      }
      l++;
```

```
    }
}

/*  Create a display list for each of 6 characters  */
void myinit(void)
{
  GLuint base;
  glShadeModel(GL_FLAT);
  base = glGenLists(128);
  glListBase(base);
  glNewList(base+'A', GL_COMPILE); drawLetter(Adata); glEndList();
  glNewList(base+'E', GL_COMPILE); drawLetter(Edata); glEndList();
  glNewList(base+'P', GL_COMPILE); drawLetter(Pdata); glEndList();
  glNewList(base+'R', GL_COMPILE); drawLetter(Rdata); glEndList();
  glNewList(base+'S', GL_COMPILE); drawLetter(Sdata); glEndList();
  glNewList(base+' ', GL_COMPILE); glTranslatef(8.0, 0.0, 0.0); glEndList();
}

char *test1 = "A SPARE SERAPE APPEARS AS";
char *test2 = "APES PREPARE RARE PEPPERS";

void printStrokedString(char *s)
{
  GLsizei len = strlen(s);
  glCallLists(len, GL_BYTE, (GLbyte *)s);
}

void CALLBACK display(void)
{
  glClear(GL_COLOR_BUFFER_BIT);
  glColor3f(1.0,  1.0,  1.0);
  glPushMatrix();
    glScalef(2.0, 2.0, 2.0);
    glTranslatef(10.0,  30.0,  0.0);
    printStrokedString(test1);
  glPopMatrix();
  glPushMatrix();
    glScalef(2.0, 2.0, 2.0);
    glTranslatef(10.0, 13.0, 0.0);
    printStrokedString(test2);
  glPopMatrix();
  glFlush();
}
```

```
/*  Main Loop
 *  Open window with initial window size,  title bar,
 *  RGBA display mode, and handle input events.
 */
int main(int argc,  char** argv)
{
  auxInitDisplayMode(AUX_SINGLE  | AUX_RGB);
  auxInitPosition(0,  0,  440,  120);
  auxInitWindow("Stroke Font");
  myinit();
  auxMainLoop(display);
  return(0);
}
```

图 4-3 用显示列表构造字体

第五章　光照处理

要绘制逼真的三维物体，必须做光照处理。如图 5—1 所示，没有光照的圆球与二维圆盘没有任何差别，而有光照的球体才是真正的三维物体。OpenGL 可以控制光照与物体的关系，产生多种不同的视觉效果。本章将着重介绍 OpenGL 光照的基本概念、如何创建光源、选择光源模式、定义材料属性。结合具体的程序讲述使用光照的步骤及使用技巧。

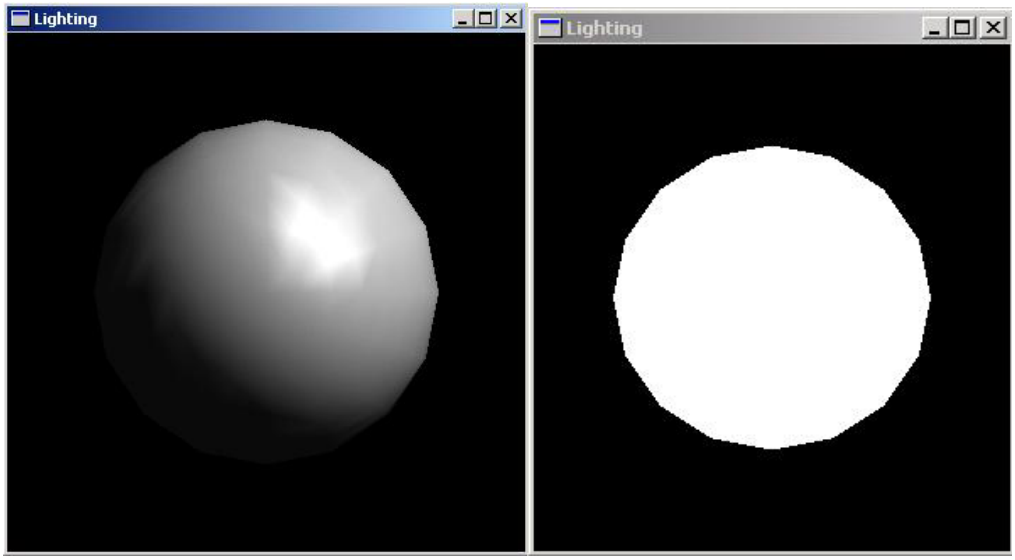

图 5—1　有无光照情况下的对比

5．1　OpenGL 光照概念

5．1．1　OpenGL 光照基本概念

在屏幕上最终显示的像素颜色，受到 glColor 命令指定的颜色影响，同时也要反映出在场景中使用光照的特性，以及物体反射和吸收光的属性，OpenGL 的光是由红、绿、蓝组成的，光源的颜色由其所发出的红、绿、蓝颜色的数量决定，材料的属性是由在不同方向反射、入射的红、绿、蓝的百分数决定的，OpenGL 的光照方程只是一个近似的，但是计算量较小，也比较精确。OpenGL 的光可来自多个光源，每光源可以单独控制开关，有的光来自某个特定的方向、位置，也有的光源分散在整个场景，如墙壁的泛光；经过来自光源的光线多次反射，无法确定其光线的方向，要绘制真实的三维物体，仅有光源是不够的，只有物体表面吸收、反射光线时，光源才起作用，而材料本身可能发光，也可能漫反射光线，或在特定方向反射光线，光照只对有上述属性的材料起作用。

① 光线由四个成分构成：发射、泛光、漫反射、反射，这四个成分单独计算，然后加起来。

发射光：来自物体，不受光源的影响。

泛光：来自环境的泛光光源，泛光照到表面上，在各方向均匀散布。

漫反射光：来自一个方向，直接照射到表面的要比仅仅扫过表面亮，一旦照射到表面上，无论在何处观察，亮度相同。

反射光：来自一个特定方向，以一个特定方向离开，可以把这个成分看成材料的光洁度。

②材料颜色取决于反射的红、绿、蓝光的百分数，与光线的特性相似，材料也有泛光、扩散、反射颜色。材料的泛光与每个入射光源的泛光组份相结合，漫反射与光源的漫反射组合，镜面反射与镜面反射组合相结合。泛光和漫反射定义材料的颜色，两者通常是相似的，镜面反射通常是白或灰。

5．1．2 光照处理的步骤

下列的程序 5-1 绘制一个有光照的球体，绘制的图形如图 5—1 所示。

程序 5-1

```
/* Prog5_1.c */
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

/* Initialize material property,light source,l ighting model,
 *  and depth buffer.
 */
void myinit(void)
{
  GLfloat mat_specular[] = {1.0, 1.0, 1.0, 1.0};
  GLfloat mat_shininess[] = {50.0};
  GLfloat light_position[] = {1.0, 1.0, 1.0, 0.0};
  glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
  glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
  glLightfv(GL_LIGHT0, GL_POSITION, light_position);
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glDepthFunc(GL_LESS);
  glEnable(GL_DEPTH_TEST);
}

void CALLBACK display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  auxSolidSphere(1.0);
  glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
```

```
  if (!h) return;
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  if (w <= h)
     glOrtho(-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w,
                          1.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
  else
     glOrtho(-1.5*(GLfloat)w/(GLfloat)h,
              1.5*(GLfloat)w/(GLfloat)h, -1.5, 1.5, -10.0, 10.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}

 /*  Main Loop
  *  Open window with initial window size,  title bar,
  *  RGBA display mode, and handle input events.
  */
int main(int argc,  char** argv)
{
  auxInitDisplayMode(AUX_SINGLE | AUX_RGB | AUX_DEPTH16);
  auxInitPosition(0,  0,  500,  500);
  auxInitWindow("Lighting");
  myinit();
  auxReshapeFunc(myReshape);
  auxMainLoop(display);
  return(0);
}
```

从上面的程序可以看出，对物体做光照处理的步骤是：

① 定义物体各顶点的法向矢量，借助这些法线可以确定物体与光源的相对方向。OpenGL 由此计算出每个顶点接受来自光照的光强度，在程序中，球体的法线在调用 auxSolidSphere 函数时已经自动求出，不必自己定义。

②创建、放置、打开光源。在程序中，只使用一个白光光源，位置由 glLightfv 函数给出，使用光源的缺省颜色，若要定义其他颜色，调用 glLight 函数，同时使用的光源数目最多可以是 8 个，(有的 OpenGL 实现允许使用更多的光源)，其他未指定的光源是黑色的。可以在空间任意点布置光源，可以距离场景较近，也可以是无穷远的，可以控制光束的形状，窄的聚合光束或是宽的光束,值得注意的是,每引入一个光源会给 OpenGL 带来大量的计算工作，最终会影响绘图的性能。用 glEnable(GL_LIGHTING)打开 OpenGL 光照处理功能，用 glEnable(GL_LIGHTi)打开第 i 个光源。

③选择光照模型。glLighModel 函数描述光照模型，在程序中，定义光照模型的唯一元素是场景的泛光，光照模型也可以定义观察者是在无穷远处还是在场景附近，另外，对物体的前、后面处理是否不同，程序中使用了缺省值，即在无穷远观察、单侧光照，若在场景附近观察，由于 OpenGL 要计算观察点与每个物体的角度，要增加很多计算量，而在无穷远观

察，忽略了这些角度，结果有些失真。详细的使用方法，请参见下一节内容。

⑤ 定义场景中物体的材料属性。物体的材料属性决定物体反光性质，材料的组成成分，由于物体的表面与入射光相互作用较为复杂，掌握如何定义材料属性，得到所需的物体外观需要一定的技巧。在程序中只定义了镜面反射颜色和光洁度两个属性，详细的使用方法，请参见第 5．3 节。

5．2 如何定义光源的特性

光源有很多属性，如颜色、位置、方向等，本节将讲述如何控制这些属性，以得到所需要的光线。glLight 命令有三个参量：光源标识 GL_LIGLITi，i=0~8、光源的属性(见表 5-1)及其属性值，若 param 不是矢量，只能设定光源属性的一个值，可以定义光源的所有属性。

void glLight{if}{V}(GLenum light，GLenum pname，TYPE param)；

表 5-1 pname 的缺省值

参 数 名	缺 省 值	解释
GL_AMBIENT	(0.0， 0.0， 0.0， 1．0)	光源泛光强度的 RGBA 值
GL_DIFFUSE	(1.0，1.0，1.0，1.0)	光源漫反射强度的 RGBA 值
GL_SPECULAR	(1.0，1.0，1.0，1.0)	光源锗面反射强度的 RGBA 值
GL_POSITION	(0.0，0.0，1.0，0.0)	光源的位置(x,y,z,w)
GL_SPOT_DIRCTION	(0.0，0.0,-1.0)	聚光灯的方向(x,y,z)
GL_SPOT_EXPONENT	0．0	聚光灯指数
GL SPOT_CUTOFF	180．0	聚光灯的截止角度
GL_CONSTANT_ATTENUATION	1．0	衰减因子常量
GL_LINEAR_ATTENUATION	0．0	线形衰减因子
GL_QUADRIC_ATTENUATION	0．0	二次衰减因子

上述定义的特性要与场景的光照模型及物体材料属性相互作用，在以后的章节中将详细介绍这些内容。下面分别讲述光源的各种属性的使用方法。

(1)颜色

OpenGL 允许把与颜色相关的 GL_AMBIENT、GL_DIFFUSE、GL_SPECULAR 三个参数与任何一个特定的光线联系起来。GL_AMBIENT 指光源在场景中添加的 RGBA 泛光强度， 缺省值为(0.0，0．0，0．0，1．0)，无泛光。GL_DIFFUSE 与光源颜色联系最密切，定义光源加入场景的漫反射光 RGBA 颜色，对于 GL_LIGHT0 的缺省值为(1．0，1．0，1．0，1．0)，即为白光，而对于 GL_LIGHT1~ GL_LIGHT7，缺省值为(0.0，0.0，0.0，0.0)，对应黑色。GL_SPECULAR 影响物体的反射强光颜色，若要得到真实的效果，应取与 GL_DIFFUSE 相同的值，对 GL_LIGHT0，缺省值为(1.0，1.0，1.0，1.0)，而对其他光源，缺省值为(0.0，0.0，0.0，0.0)。

(2)位置和衰减 .

若光源在无穷远处，到达物体时光线是平行的，而放置在场景中的光源位置和方向决定对整个场景的影响，在上一节的程序中

```
GLfloat light_position[]={1.0，1.0，1.0，0.0};
glLighw(G_LIGHT0，GL_POSITION，light_position);
```

给出了光源 GL_LIGHT0 的位置属性 GL_POSITION，光源在齐次坐标(x，y，z，w)=(1.0，1.0，1.0，0.0)处，即无穷远，x,y,z 分量给出光照的方向，这个方向矢量为同法线矢量一样要做模型取景变换，缺省状态的(0.0，0.0，1.0，0.0)，即无穷远处沿负 z 轴方向光照，若 w≠0，则光源在场景内，(x，y，z)代表光源的空间位置，经过模型取景变换，存贮成视觉

坐标。缺省时，光源向所有方向发射光束，用户可以控制其发光的角度。

在场景内的光线从光源发射出来，其强度会随传播距离逐渐衰减，而在无穷远处的光源则感觉不到其强度的衰减，OpenGL 定义的衰减因子为：

$$\frac{1}{k_c + k_l d + k_q d^2}$$

其中，d 为光源位置与顶点的距离；

k_l = GL_LINEAR_ATTENUATION，缺省值为 0．0；

k_q = GL_QUADRATIC_ATTENUATION，缺省值为 0.0;

k_c = GL_CONSTANT_ATTENUATION，缺省值为 1.0

可以用 glLight 命令设定上述属性的值，值得注意的是，除发射光和全景泛光值不衰减外，泛光、漫反射、镜面反射光都要衰减。

(3)聚光灯

上面提到用户可以控制光源发光的角度，即光源成为一个聚光灯，发射的光线限制在一个圆锥内，GL_SPOT_CUTOFF 是圆锥顶角的一半，缺省时为 180°，glLight 命令中 GL_SPOT_CUTOFF 的取值范围为[0.0，90.0]，下面的语句定义光线和截止角度所在圆锥的轴线矢量，即聚光灯方向。

```
glLightf(GL_LIGHT0，GL_SPOT_CUTOFF,45.0)；
GLfloat spot_direction={-1.0，-1.0，0.0}；
glLightv(GL_LIGHT0，GL_SPOT_DIRECTION，spot_direction)；
```

缺省的光照方向为(0.0，0.0，-1.0)，沿负 z 轴方向，聚光灯的方向也要做模型取景变换，存贮成视觉坐标。

除了上面提到的两个属性外，聚光灯的属性还包括一个控制圆锥内光强分布的指数因子 GL_SPOT_EXPONENT,在圆锥轴线上光强最大，边缘处最暗。这个因子越大，光束的强度越向轴线集中，缺省值为 GL_SPOT_EXPONENT=0.0。

(4)使用多个光源

前面的内容中，所举实例均针对光源 GLLIGHT0，当然在场景中还可以使用其他光源 GL_LIGHTi，i=1~7，用 glLight 命令也可以为这些光源设置属性。需要注意的是，这些光源属性的有些缺省值与 GL_LIGHT0 有所不同。

为了说明如何使用聚光灯以及在场景中布置多个光源，下面给出一个例子，程序 5-2 中同时使用了一个标准光源和一个黄色聚光灯照亮一个环面，见图 5-2，其中红色方块代表聚光灯的位置。

程序 5—2

```
/* Prog5_2.c */
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);
```

```
/*  Initialize material property,  light source,  lighting model,
 *  and depth buffer.
 */

void myinit(void)
{
  GLfloat mat_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
  GLfloat mat_diffuse[] = { 0.8, 0.8, 0.8, 1.0 };
  GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
  GLfloat mat_shininess[] = { 50.0 };
  GLfloat light0_diffuse[] = { 0.0, 0.0, 1.0, 1.0 };
  GLfloat light0_position[] = { 1.0, 1.0, 1.0, 0.0 };
  GLfloat light1_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
  GLfloat light1_diffuse[] = { 1.0, 0.0, 0.0, 1.0 };
  GLfloat light1_specular[] = { 1.0, 0.6, 0.6, 1.0 };
  GLfloat light1_position[] = { -3.0, -3.0, 3.0, 1.0 };
  GLfloat spot_direction[] = { 1.0, 1.0, -1.0};

  glMaterialfv(GL_FRONT, GL_AMBIENT,  mat_ambient);
  glMaterialfv(GL_FRONT, GL_DIFFUSE,  mat_diffuse);
  glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
  glMaterialfv(GL_FRONT, GL_SHININESS,mat_shininess);

  glLightfv(GL_LIGHT0, GL_DIFFUSE, light0_diffuse);
  glLightfv(GL_LIGHT0, GL_POSITION, light0_position);

  glLightfv(GL_LIGHT1, GL_AMBIENT, light1_ambient);
  glLightfv(GL_LIGHT1, GL_DIFFUSE, light1_diffuse);
  glLightfv(GL_LIGHT1, GL_SPECULAR,light1_specular);
  glLightfv(GL_LIGHT1, GL_POSITION,light1_position);

  glLightf (GL_LIGHT1, GL_SPOT_CUTOFF, 30.0);
  glLightfv(GL_LIGHT1, GL_SPOT_DIRECTION, spot_direction);

  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glEnable(GL_LIGHT1);
  glDepthFunc(GL_LESS);
  glEnable(GL_DEPTH_TEST);
}

void CALLBACK display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
  glPushMatrix();
    glTranslated (-3.0,  -3.0, 3.0);
    glDisable(GL_LIGHTING);
    glColor3f(1.0, 0.0, 0.0);
    auxWireCube(0.1);
    glEnable(GL_LIGHTING);
  glPopMatrix ();
  auxSolidSphere(2.0);
  glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  if (w <= h)
    glOrtho(-5.5, 5.5, -5.5*(GLfloat)h/(GLfloat)w,
                        5.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
  else
    glOrtho(-5.5*(GLfloat)w/(GLfloat)h,
             5.5*(GLfloat)w/(GLfloat)h, 5.5, 5.5, -10.0, 10.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}

/*  Main Loop
 *  Open window with initial window size,  title bar,
 *  RGBA display mode, and handle input events.
 */
void main(void)
{
  auxInitDisplayMode(AUX_SINGLE | AUX_RGBA);
  auxInitPosition(0,  0,  500,  500);
  auxInitWindow("Spotlight and Multi lights");


  myinit();
  auxReshapeFunc(myReshape);
  auxMainLoop(display);
}
```

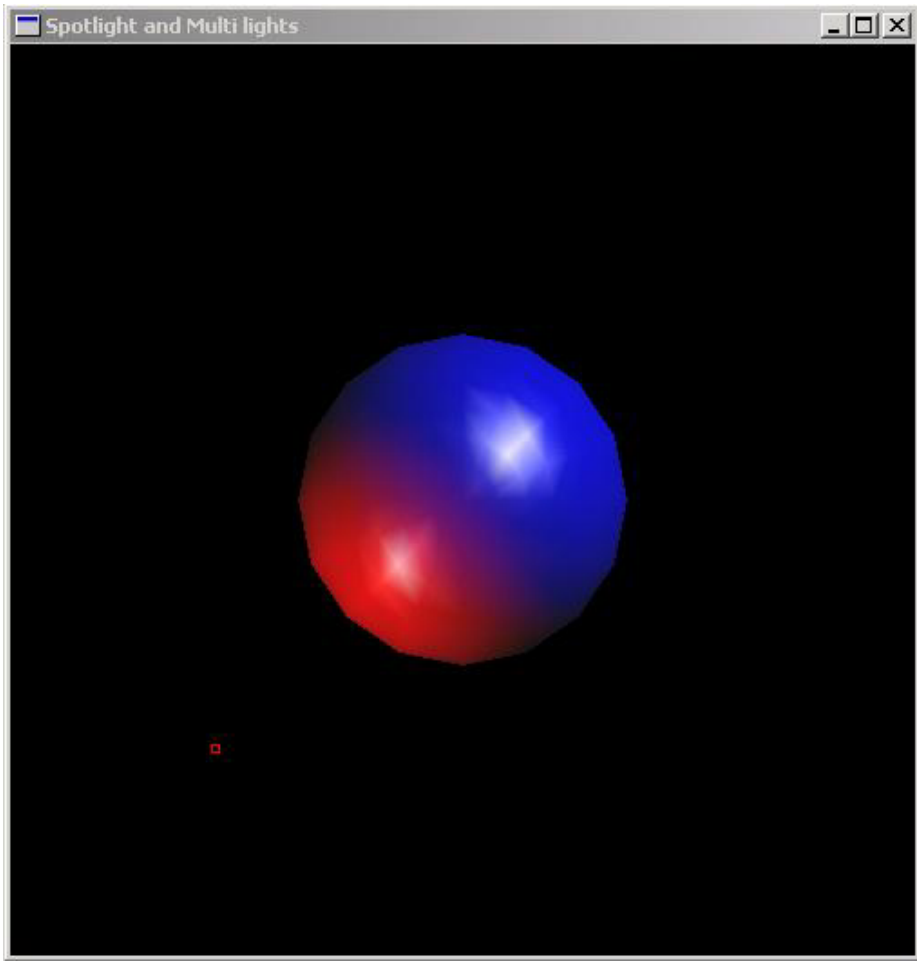

图 5-2 使用多个光源

(5)光源的位置和方向

OpenGL 把光源的位置和方向也看成是几何要素，即要做同样的矩阵变换，可以通过改变模型取景矩阵堆栈，改变光源的位置和方向(投影矩阵对它没有影响)，下面讲述三种改变程序的模型，取景变换对光源位置的影响：

①光源位置静止不动，参见第 5．1 节的程序 5-1，在定义取景或模型变换后，设置光源的位置，即在 myinit 子程序中定义变换矩阵，在 myReshape 子程序中设定光源的位置。

②光源绕一个静止物体运动，可以在设置模型变换前，定义光源的位置，通过改变模型变换，改变光源的位置，例如在 display 子程序中，光源绕静止圆球转过 spin 度，由于只需改变模型变换，而取景变换保持不变，要用 glPushMatrix 与 glPopMatrix 把两者分开。

```
void display(GLint spin);
{
        GLftoat light_position[] = {0.0,   0.0,   1.5,   1.0};
        glClear(GLCOLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glPushMatrix();
                glTranslatef(0.0,   0.0,   -5.0);
                glPushMatrix();
                        glRotated((GLdouble)spin, 1.0, 0.0, 0.0);
                        glLightv(GL_LIGHT), GL_POSITION,   light_position);
                glPopMatrix();
```

```
            auxSolidTorus();
        glPopMatrix();
        glFlush();
}
```

③光源随观察点一起运动，可以在设置取景变换前，定义光源的位置，使取景变换同时作用于光源和观察点，例如在 myinit 子程序中：

```
GLfloat light_position[]={0.0,  0.0,  1.0,  1.0};
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(40.0,  (GLfloat)w/(GLfloat)h,  1.0,  100.0);
glMatrixMode(GL_MODELVIEW);
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

而对于 display 子程序

```
glCtear(GL_OLOR_BUFFER_MASK | GL_DEPTH_BUFFER_MASK);
glPushMatrix();
        glTranslatef(0.0,  0.0,  -5.0);
        glRotated((GLdouble)spin,  1.0,  0.0,  0.0);
        auxSolidTorus();
glPopMatrix();
glFlush();
```

下面的程序 5-3 说明如何用模型变换命令 glRotate 和 glTranslate 移动光源，见图 5-3 所示，其中黄色线框圆球代表光源，照亮一个环面，每次单击鼠标器左键，光源移动 15°，并在新的光源位置重新绘制场景。

程序 5-3

```
/* Prog5_3.c */
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK movelight(AUX_EVENTREC *event);
void CALLBACK display(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);

static int step = 0;

void CALLBACK movelight(AUX_EVENTREC *event)
{
  step = (step + 15) % 360;
}

void myinit(void)
```

```
{
  GLfloat mat_diffuse[] = {0.0, 0.5, 1.0, 1.0};
  GLfloat mat_ambient[] = {0.0, 0.2, 1.0, 1.0};
  GLfloat light_diffuse[] = {1.0, 1.0, 1.0, 1.0};
  GLfloat light_ambient[] = {0.0, 0.5, 0.5, 1.0};

  glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
  glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);

  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glDepthFunc(GL_LESS);
  glEnable(GL_DEPTH_TEST);
}
/*  Here is where the light position is reset after the modeling
 *  transformation (glRotated)  is called.  This places the
 *  light at a new position in world coordinates.  The yellow
 *  sphere represents the position of the light.
 */
void CALLBACK display(void)
{
    GLfloat position[] = { 0.0, 0.0, 1.5, 1.0 };

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    glTranslatef(0.0,  0.0,  -5.0);
    glPushMatrix();
      glRotated((GLdouble)step, -1.0, 1.0, 1.0);
      glRotated(0.0, 1.0, 0.0, 0.0);
      glLightfv(GL_LIGHT0, GL_POSITION, position);
      glTranslated(0.0, 0.0, 1.5);
      glDisable(GL_LIGHTING);
      glColor3f(1.0, 1.0, 0.0);
      auxWireSphere(0.1);
      glEnable(GL_LIGHTING);
    glPopMatrix();
    auxSolidTorus(0.275, 0.85);
  glPopMatrix();
  glFlush();
}
void CALLBACK myReshape(GLsizei w, GLsizei h)
{
  glViewport(0, 0, w, h);
```

```
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(40.0, (GLfloat)w/(GLfloat)h, 1.0, 20.0);
  glMatrixMode(GL_MODELVIEW);
}

/*  Main Loop
 *  Open window with initial window size,  title bar,
 *  RGBA display mede, and handle input events.
 */
void main (void)
{
  auxInitDisplayMode(AUX_SINGLE | AUX_RGBA);
  auxInitPosition(0,  0,  500,  500);
  auxInitWindow("Moving Light");
  myinit();
  auxMouseFunc(AUX_LEFTBUTTON, AUX_MOUSEDOWN, movelight);
  auxReshapeFunc(myReshape);
  auxMainLoop(display);
}
```

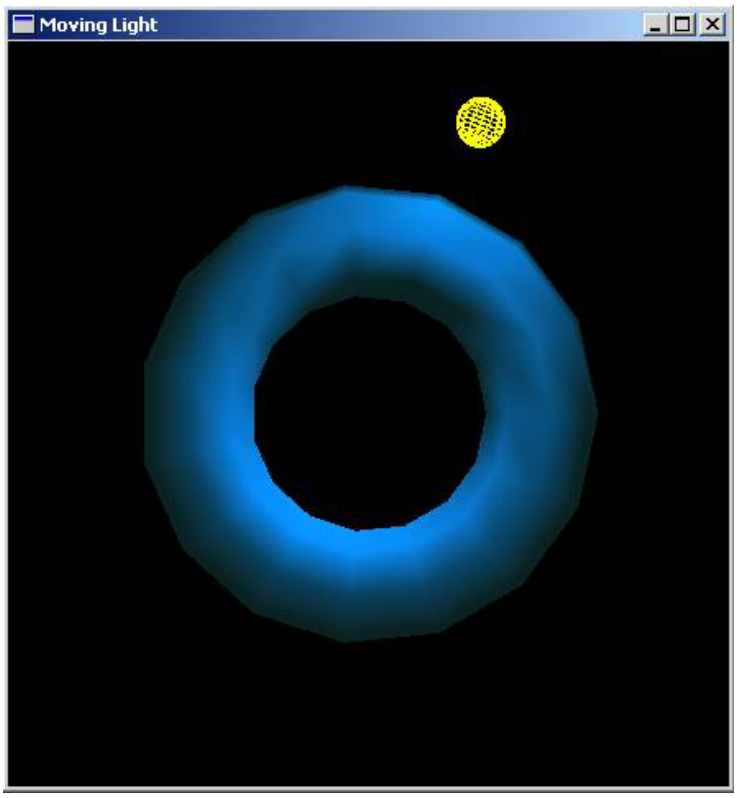

图 5-3 移动光源

(6)选取光照模型

OpenGL 的光照模型由三部分组成：全景泛光强度，观察点的位置，对前、后面光照计算的处理。

. 全景泛光强度

场景中除了有每个光源发出的泛光，还有不属于任何光源的泛光存在，这个泛光即为全景泛光，用 GL_LIGHT_MODEL_AMBIENT 参数定义全景泛光的 RGBA 强度，例如

GLfloat lmodel_ambient[]={0.2，0.2，0.2，1.0};

glLightModelfv(GL_LIGHT_MODEL_AMBIENT，lmodel_ambient);

上述 lmodel_ambient 值为缺省时的 GL_LIGHT_MODEL_AMBIENT 参数值。

. 观察点的位置

观察点的位置影响镜面反射产生强光的计算，顶点上强光的亮度依赖该处的法线值、顶点到光源的方向和顶点到观察点的方向。对于无穷远处的观察点，与场景中任一顶点的方向均相同，虽然用场景中的观察点得到的结果更逼真，但是由于要对每个顶点计算方向，会降低绘图的性能，缺省时使用无穷远处的观察点，可以用

glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER，GL_TRUE);

设置成场景中的观察点，位于视觉坐标(0.0，0.0，0.0)，把上述命令的 GL_TRUE 换成 GL_FALSE，则切换回无穷远处的观察点。

. 两面光照

光照计算是针对每个多边形的，无论是朝前，还是朝后，通常只设定朝前多边形的光照条件，使朝后多边形光照不正确，在第 5．1 节的实例中前面部分是可见的，则不影响绘图的效果，若剖视球体，则应按光照条件进行光照处理，这时需调用

glLightModeli(GL_LIGHT_MODEL_TWO_SIDE，GL_TRUE);

允许两面光照，OpenGL 把朝后多边形法线取逆，这样才能正确处理表面光照。把上述命令的第二个参数替换成 GL_FALSE，则不进行两面光照。

5．3 定义材料属性

本节讲述的场景中物体的材料属性：泛光、漫反射和镜面反射颜色，光洁度、发射光的颜色，多数属性的概念与上一节谈到的光源属性在概念上相近，设定这些属性的方法也相近。

void glMaterial{if}{v}(GLenum face，GLenum pname，TYPE param);

定义光照方程中物体的当前材料属性，face 为 GL_FRONT,GL_BACK 和 GL_FRONT_AND_BACK 之一，标识这些属性适用的表面，pname 为属性名称，param 为属性的值，见下表 5-2，所有属性的定义都是在 RGBA 模式下的。

表 5-2 glMaterial 的 pname 参数缺省值

参 数 名	缺 省 值	解 释
GL_AMBIENT	(0.2，0.2，0.2，1.0)	材料泛光强度的 RGBA 值
GL_DIFFUSE	(0.8，0.8，0．8，1．0)	材料漫反射强度的 RGBA 值
GL_AMBIENT_DIFFUSE	(0.8，0.8，0.8，1.0)	材料的泛光、漫反射强度的 RGBA 值
GL_SPECULAR	(0.0，0.0，0.0，1.0)	材料镜面反射强度的 RGBA 值
GL_SHININESS	0．0	材料镜面反射指数
GL_EMISSION	(0.0，0.0，0.0，1.0)	材料发射光的颜色
GL_COLOR_INDEX	(0，1，1)	材料镜面反射光的色彩索引

(1)泛光和漫反射

由 glMaterial 命令设定的 GL_DIFFUSE 和 GL_AMBIENT 参数分别影响物体反射的漫反射和

泛光颜色。漫反射在表现物体颜色方面起主要作用，它受到入射漫反射光颜色和入射光与法线夹角影响，而观察点的位置不会影响到漫反射光，泛光影响物体的整体颜色，因为直射到物体上的漫反射最亮，而没有直射处物体泛光最明显，物体的泛光受全景泛光和各光源泛光的影响，与漫反射相似，泛光也不受观察点位置的影响。

对于真实物体，泛光和漫反射通常具有相同颜色，OpenGL 提供了同时为这两个属性赋值的简便方式，例如：

```
GLfloat mat_amb_diff[]={0. 1，0. 5，0. 8，1. 0};
glMaterialfv(GL_FRONT_AND_BACK,GL_AMBIENT_AND_DIFFUSE，mat_amb_diff};
```

(2)镜面反射

物体发出的镜面反射产生强光，镜面反射强度受观察点位置影响，即在反射角方向光最强，OpenGL 用 GL_SPECULAR 属性定义镜面反射强光的 RGBA 值，GL_SHININESS 控制强光的尺寸和亮度，取值范围[0. 0，128. 0]，GL_SHININESS 值越大，强度越集中，越亮，例如：

```
GLfloat mat_specular[] = {1. 0，1. 0，1. 0，1. 0};
GLfloat low_shininess[] = {5. 0};
glMaterialfv(GL_FRONT,GL_SPECULAR，mat SPECULAR);
glMaterialfv(GL_FRONT, GL_SHININESS，low_shininess);
```

(3)发射

GL_EMISSION 属性定义物体发射光的颜色，可以用来模拟场景中的发光体例如：

```
GLfloat mat_emission[] = {0. 3，0. 2，0. 2，0. 0};
glMaterialfv(GL_FRONT,GL_EMISSION，mat_emission);
```

若在物体所在位置创建一个光源，可以模拟台灯的视觉效果。

与第 5. 1 节的实例不同，下列的程序 5-4 应用不同的材料属性，绘制多种不同视觉效果的十二个球体，参见图 5-4，第一行球体，无泛光，第二行球体，灰色泛光，第三行球体，蓝色泛光，第一列球体，蓝色漫反射，无镜面反射，第二列球体加入白色镜面反射，光洁度指数较小，第三列球体光洁度指数高，因此强光更集中，第四列球体蓝色扩散，加入了发射属性。

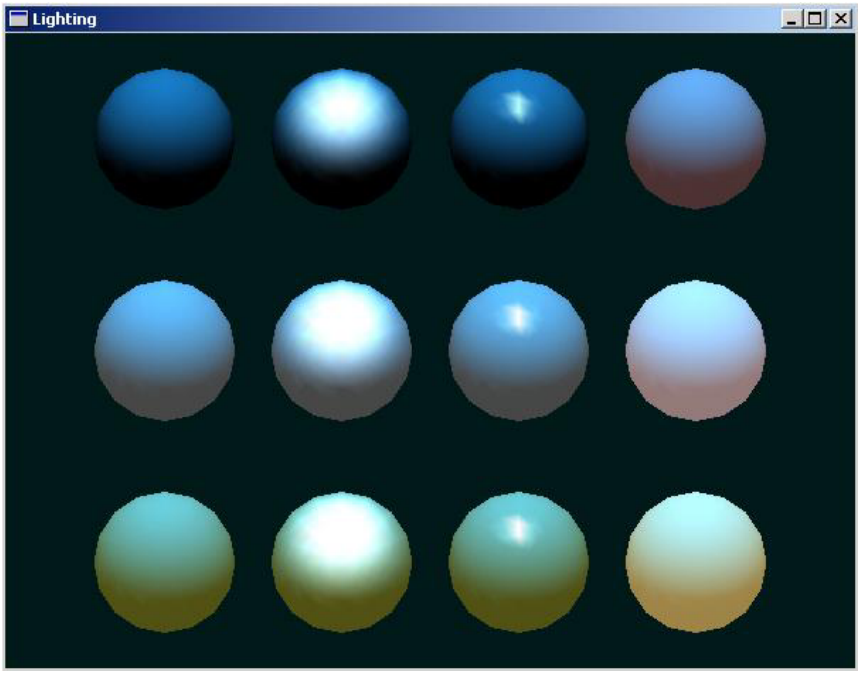

图 5-4 不同材质的视觉效果

程序 5-4

```
/* Prog5_4.c */
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK display(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);

/*  Initialize z-buffer,  projection matrix,  light source,
 *  and lighLing model.  Do not specify a material property here.
 */
void myinit(void)
{
  GLfloat ambient[] = { 0.0, 0.0, 0.0, 1.0 };
  GLfloat diffuse[] = { 1.0,  1.0,  1.0,  1.0 };
  GLfloat specular[] = { 1.0,  1.0,  1.0,  1.0 };
  GLfloat position[]  = { 0.0,  3.0,  2.0,  0.0 };
  GLfloat lmodel_ambient[]  = { 0.4,  0.4,  0.4,  1.0 };
  GLfloat local_view[]  = { 0.0 };

  glEnable(GL_DEPTH_TEST);
  glDepthFunc(GL_LESS);
  glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
  glLightfv(GL_LIGHT0, GL_POSITION, position);
  glLightModelfv(GL_LIGHT_MODEL_AMBIENT,  lmodel_ambient);
  glLightModelfv(GL_LIGHT_MODEL_LOCAL_VIEWER,  local_view);

  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glClearColor(0.0,  0.1,  0.1,  0.0);
}


/*  Draw twelve spheres in 3 rows with 4 columns.
 *  The spheres in the first row have materials with no ambient reflection.
 *  The second row has materials with significant ambient reflection.
 *  The third row has materials with colored ambient reflection.
 *  The first column has materials with blue, diffuse reflection only.
 *  The second column has blue diffuse reflection, as well as specular
```

```
 *  reflection with a low shininess exponent.
 *  The third column has blue diffuse reflection, as well as specular
 *  reflection with a high shininess exponent  (a more concentrated highlight).
 *  The fourth column has materials which also include an emissive component.
 *  glTranslatef()  is used to move spheres to their appropriate locations.
 */
void CALLBACK display(void)
{
      GLfloat no_mat[] = { 0.0, 0.0, 0.0, 1.0 };
      GLfloat mat_ambient[] = { 0.7,  0.7,  0.7,  1.0 };
      GLfloat mat_ambient_color[]  = { 0.8,  0.8,  0.2,  1.0 };
      GLfloat mat_diffuse[] =  { 0.1,  0.5,  0.8,  1.0 };
      GLfloat mat_specular[] = { 1.0,  1.0,  1.0,  1.0 };
      GLfloat no_shininess[] = { 0.0 };
      GLfloat low_shininess[]  = { 5.0 };
      GLfloat high_shininess[] = { 100.0 };
      GLfloat mat_emission[]  = {0.3,  0.2,  0.2,  0.0};

      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

/*  draw sphere in first row, first column
 *  diffuse reflection only; no ambient or specular
 */

    glPushMatrix();
      glTranslatef(-3.75,  3.0,  0.0);
      glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
      glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
      glMaterialfv(GL_FRONT, GL_SPECULAR, no_mat);
      glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);
      glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
      auxSolidSphere(1.0);
    glPopMatrix();

/*  draw sphere in first row,  second column
 *  diffuse and specular reflectien;  low shininess; no ambient
 */
    glPushMatrix();
      glTranslatef(-1.25,  3.0,  0.0);
      glMaterialfv(GL_FRONT, GL_AMBIENT,  no_mat);
      glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
      glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
      glMaterialfv(GL_FRONT, GL_SHININESS, low_shininess);
      glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
```

```
      auxSolidSphere(1.0);
    glPopMatrix();

/*  draw sphere in first row,  third column
 *  diffuse and specular reflection; high shininess; no ambient
 */
   glPushMatrix();
     glTranslatef (1.25, 3.0, 0.0);
     glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
     glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
     glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
     glMaterialfv(GL_FRONT, GL_SHININESS, high_shininess);
     glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
     auxSolidSphere(1.0);
   glPopMatrix();

/*  draw sphere in first row,  fourth column
 *  diffuse reflection; emission; no ambient or specular reflection
 */
   glPushMatrix();
     glTranslatef(3.75,  3.0,  0.0);
     glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
     glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
     glMaterialfv(GL_FRONT, GL_SPECULAR, no_mat);
     glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);
     glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
     auxSolidSphere(1.0);
   glPopMatrix();

/*  draw sphere in second row,  first column
 *  ambient and diffuse reflection; no specular
 */
   glPushMatrix();
     glTranslatef(-3.75,  0.0,  0.0);
     glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
     glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
     glMaterialfv(GL_FRONT, GL_SPECULAR,no_mat);
     glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);
     glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
     auxSolidSphere(1.0);
   glPopMatrix();

/*  draw sphere in second row,  second column
 *  ambient, diffuse and specular reflection; low shininess
```

```
 */
    glPushMatrix();
      glTranslatef(-1.25,  0.0,  0.0);
      glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
      glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
      glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
      glMaterialfv(GL_FRONT, GL_SHININESS,  low_shininess);
      glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
      auxSolidSphere(1.0);
    glPopMatrix();

/*  draw sphere in second row,  third column
 *  ambient, diffuse and specular reflection; high shininess
 */
    glPushMatrix();
      glTranslatef(1.25, 0.0, 0.0);
      glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
      glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
      glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
      glMaterialfv(GL_FRONT, GL_SHININESS, high_shininess);
      glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
      auxSolidSphere(1.0);
    glPopMatrix();

/*  draw sphere in second row,  fourth column
 *  ambient and diffuse reflection; emission; no specular
 */
    glPushMatrix();
      glTranslatef(3.75,  0.0,  0.0);
      glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
      glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
      glMaterialfv(GL_FRONT, GL_SPECULAR, no_mat);
      glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);
      glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
      auxSolidSphere(1.0);
    glPopMatrix();

/*  draw sphere in third row,  first column
 *  colored ambient and diffuse reflection; no specular
 */
    glPushMatrix();
      glTranslatef(-3.75,  -3.0,  0.0);
      glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient_color);
      glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
```

```
      glMaterialfv(GL_FRONT, GL_SPECULAR, no_mat);
      glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);
      glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
      auxSolidSphere(1.0);
    glPopMatrix();

/*  draw sphere in third row,  second column
 *  colored ambient, diffuse and specular reflection; low shininess
 */
    glPushMatrix();
      glTranslatef(-1.25, -3.0, 0.0);
      glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient_color);
      glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
      glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
      glMaterialfv(GL_FRONT, GL_SHININESS, low_shininess);
      glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
      auxSolidSphere(1.0);
    glPopMatrix();

 /*  draw sphere in third row,  third column
  *  colored ambient, diffuse and specular reflection; high shininess
  */
    glPushMatrix();
      glTranslatef(1.25,  -3.0,  0.0);
      glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient_color);
      glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
      glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
      glMaterialfv(GL_FRONT, GL_SHININESS, high_shininess);
      glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
      auxSolidSphere(1.0);
    glPopMatrix();

/*  draw sphere in third row,  fourth column
 *  colored ambient and diffuse reflection;  emission; no specular
 */
    glPushMatrix();
      glTranslatef(3.75, -3.0, 0.0);
      glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient_color);
      glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
      glMaterialfv(GL_FRONT, GL_SPECULAR, no_mat);
      glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);
      glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
      auxSolidSphere(1.0);
    glPopMatrix();
```

```
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
      if (!h) return;
      glViewport(0, 0, w, h);
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      if  (w <=  (h * 2))
        glOrtho(-6.0, 6.0,  -3.0*((GLfloat)h*2)/(GLfloat)w,
                            3.0*((GLfloat)h*2)/(GLfloat)w, -10.0, 10.0);
      else
        glOrtho(-6.0*(GLfloat)w/((GLfloat)h*2),
                 6.0*(GLfloat)w/((GLfloat)h*2),  -3.0,  3.0,  -10.0,  10.0);
     glMatrixMode(GL_MODELVIEW);
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mede, and handle input events.
 */
int main(int argc,  char** argv)
{
      auxInitDisplayMode(AUX_SINGLE | AUX_RGB | AUX_DEPTH16);
      auxInitPosition(0,  0,  600,  450);
      auxInitWindow ("Lighting");
      myinit();
      auxReshapeFunc(myReshape);
      auxMainLoop(display);
      return(0);
}
```

从上面程序可以看出，要多次重复调用 glMaterial 命令，而每次仅改变一个属性，而多次调用 glMaterial 命令又会降低绘图性能,用 glColorMaterial 命令可以在改变材料属性的同时，尽量减少对绘图性能的影响。

void glColorMaterial(GLenum face, GLenum mode);

使 face 面的材料属性 mode 一直跟踪当前颜色，即改变当前颜色就立即改变指定的属性值，face、mode 的定义与 glMaterial 命令的参数相同，在调用 glColorMaterial 函数后，应该用 glEnable(GL_COLOR_MATERIAL)使 OpenGL 可以通过改变当前颜色改变材料的属性，若不希望使用这个功能时一定要用 glDisable(GL_COLOR_MATERIAL)关闭它，否则可能导致不需要的属性设置，若要同时改变多个材料属性，还是应该用 glMaterial 命令。下面是一个用 glColorMaterial 命令改变材料参数的例子，在程序 5-5 中，按下键盘 R，G，B 可以分别改变球体漫反射的红、绿、蓝颜色，图 5-5 为程序的运行结果。

程序 5-5

```
/* Prog5_5.c */
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK changeRedDiffuse(void);
void CALLBACK changeGreenDiffuse(void);
void CALLBACK changeBlueDiffuse(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

GLfloat diffuseMaterial[4] = { 0.5, 0.5, 0.5, 1.0 };

/*  Initialize values for material property,  light source,
 *  lighting model,  and depth buffer.
 */
void myinit(void)
{
    GLfloat mat_specular[]   = { 1.0,  1.0,  1.0,  1.0 };
    GLfloat light_position[] = { 1.0,  1.0,  1.0,  0.0 };
    glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuseMaterial);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialf(GL_FRONT, GL_SHININESS,  25.0);
    glLightfv(GL_LIGHT0, GL_POSITION,  light_position);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);
    glColorMaterial(GL_FRONT, GL_DIFFUSE);
    glEnable(GL_COLOR_MATERIAL);
}

void CALLBACK changeRedDiffuse(void)
{
    diffuseMaterial[0] += 0.1;
    if  (diffuseMaterial[0]  > 1.0)
      diffuseMaterial[0]  = 0.0;
    glColor4fv(diffuseMaterial);
}
void CALLBACK changeGreenDiffuse(void)
```

```
{
    diffuseMaterial[1] += 0.1;
    if  (diffuseMaterial[1]  > 1.0)
      diffuseMaterial[1] = 0.0;
    glColor4fv(diffuseMaterial);
}
void CALLBACK changeBlueDiffuse(void)
{
    diffuseMaterial[2]  += 0.1;
    if  (diffuseMaterial[2]  > 1.0)
      diffuseMaterial[2] = 0.0;
    glColor4fv(diffuseMaterial);
}
void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    auxSolidSphere(1.0);
    glFlush();
}
void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    if  (!h)  return;
    glViewport(0,  0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if  (w <= h)
        glOrtho(-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w,
                           1.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
    else
        glOrtho(-1.5*(GLfloat)w/(GLfloat)h,
                 1.5*(GLfloat)w/(GLfloat)h, -1.5, 1.5, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
int main(int argc,  char** argv)
{
    auxInitDisplayMode(AUX_SINGLE | AUX_RGB | AUX_DEPTH16);
    auxInitPosition(0, 0, 500, 500);
    auxInitWindow ("Color Material Mode");
    myinit();
    auxKeyFunc(AUX_R, changeRedDiffuse);
    auxKeyFunc(AUX_r, changeRedDiffuse);
    auxKeyFunc(AUX_G, changeGreenDiffuse);
    auxKeyFunc(AUX_g, changeGreenDiffuse);
```

```
    auxKeyFunc(AUX_B, changeBlueDiffuse);
    auxKeyFunc(AUX_b, changeBlueDiffuse);
    auxReshapeFunc(myReshape);
    auxMainLoop(display);
    return(0);
}
```

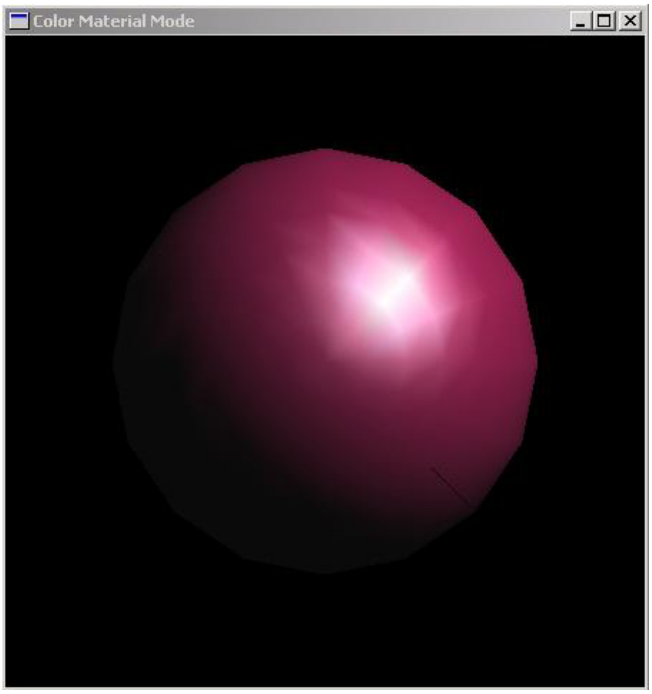

图 5-5 glColorMaterial 函数的使用

5．4　光照处理的注意事项

要得到理想的光照处理效果，需要有一定的使用技巧，也需要通过在程序中尝试调整光照、材料属性的参数来实现。本节讲述光照处理的一些注意事项，然后结合具体实例，讲述如何在色彩索引模式下进行光照处理。

5．4．1　OpenGL 顶点的颜色值

如果不做光照处理，顶点的颜色就是当前的绘图颜色。若允许使用光照，则在视觉坐标中做光照计算，此时顶点的颜色为顶点处材料发射光的颜色+顶点上的材料泛光属性放大的全景泛光颜色+经过衰减的来自各光源的泛光，漫反射、镜面反射颜色，在 RGBA 模式下最终的颜色值调整到(0，1)范围内。值得注意的是，OpenGL 光照计算不考虑物体间光线的遮挡，并不自动产生暗影。

①材料发射项为 GL_EMISSION 参数的 RGB 值。

②放大的全景泛光项为由 GL_LIGHT_MODEL_AMBIENT 参数定义的全景泛光与 glMaterial 命令的 GL_AMBIENT 参数定义的材料泛光属性值相乘所得的值，RGB 值为(R1R2，G1G2，B1B2)，(R1，G1，R1)，(R:，G,，R：)分别为上述两个参数的 RGB 值。

③光源作用于顶点的 RGB 值为衰减因子 X 聚光灯效应(泛光+漫反射+镜面反射)：

衰减因子 = $(k_c + k_l d + k_q d^2)^{-1}$，若光源在无穷远处，则衰减因子为 1．0。

聚光灯效应：若 GL_SPOT_CUTOFF=180．0，则为 1；

若在聚光灯照亮的范围外，则为 0。

若在聚光灯照亮的范围内，max{v·d,0}×GL_SPOT_EXPONENT,其中 $v=(v_x, v_y, v_z)^T$ 为聚光灯所在位置(GL_POSITION)指向顶点的单位矢量，$d=(d_x, d_y, d_z)^T$，聚光灯方向(GL_SPOT_DIRECTION)，v·d 为矢量的点积。

泛光项为光源泛光颜色值×材料泛光属性的颜色值。

漫反射项为 max{$l\cdot n$,0}×光源漫反射属性值×材料漫反射属性值。

其中 $l=(l_x, l_y, l_z)^T$ 为顶点到光源位置(GL_POSITION)的单位矢量，$n=(n_x, n_y, n_z)^T$ 为顶点的法向矢量。

镜面反射项：　若上述 $l\cdot n\leqslant 0$，则在顶点无该项，否则其值为 $max\{s\cdot n,0\}\times$ GL_SHININESS×光源镜面反射×材料镜面反射，其中 $s=(s_x, s_y, s_z)^T$ 为顶点到光源位置与顶点到观察点两矢量之和，经过归一化处理得到的单位矢量，n 为顶点的法向量。

上述各项仅为一个光源对顶点的作用。若场景中有多个光源，则对每一个顶点要把各光源的影响累加到一起。

5．4．2　色彩索引模式下的光照处理

前面所讲的光照处理函数均是在 RGBA 模式下使用的,在色彩索引模式下要取得某些效果比较困难，所以应该尽量使用 RGBA 模式，在色彩索引模式下，仅使用光源参数 GL_DIFFUSE(d1)和 GL_SPECULAR(s1)以及材料参数 GL_SHININESS 的 RGB 值，用于计算色彩扩散指数 d_{ci} 和反光强度 s_{ci}:

d_{ci} = 0.30R(d1)+0.59G(d1)+0.11B(d1)

s_{ci} = 0.30R(s1)+0.59G(s1)+0.11B(s1)

因为眼睛对绿光最敏感，对蓝光最不敏感，所以 RGB 值的加权值不相等。

在色彩索引模式下，定义材料颜色的形式如下：

GLfloat mat_colormap[]={16.0，47.0，79.0};

glMaterialfv(GL_FORNT,GL_COLOR_INDEX，mat_colormap);

其中数组的 3 个值分别对应材质的泛光、漫反射和反射色彩指数，缺省时，数组的值为{0,0，1.0，1.0},即泛光色彩指数为 0.0,漫反射和反射色彩指数均为 1.0,glColorMaterial 命令在色彩指数模式下的光照处理中不起任何作用。

程序 5-6 与第 5．1 节的例子相似，绘制一个有光照的球体，如图 5-6 所示。不同的是，这个程序在色彩索引模式下为材料、光源属性赋值，用 auxSetOneColor 命令定义了一系列色彩索引对应的颜色。

程序 5-6

```
/* Prog5_6.c */
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK display(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);

/*  Initialize material property,  light source, and lighting model.
 */
void myinit(void)
```

```
{
  GLint i;
  GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
  GLfloat mat_colormap[] = { 16.0, 48.0, 79.0 };
  GLfloat mat_shininess[]  = { 10.0 };
  glMaterialfv(GL_FRONT, GL_COLOR_INDEXES, mat_colormap);
  glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
  glLightfv(GL_LIGHT0, GL_POSITION, light_position);
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glDepthFunc(GL_LESS);
  glEnable(GL_DEPTH_TEST);
  for (i = 0; i < 32; i++)
  {
     auxSetOneColor(16 + i, 1.0*(i/32.0),  0.0, 1.0*(i/32.0));
     auxSetOneColor(48 + i, 1.0, 1.0*(i/32.0), 1.0);
  }
  glClearIndex(0);
}
void CALLBACK display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  auxSolidSphere(1.0);
  glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
  if  (!h)  return;
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  if  (w <= h)
     glOrtho(-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w,
                         1.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
  else
     glOrtho(-1.5*(GLfloat)w/(GLfloat)h,
              1.5*(GLfloat)w/(GLfloat)h, -1.5, 1.5, -10.0, 10.0);
     glMatrixMode(GL_MODELVIEW);
     glLoadIdentity();
}
/*  Main Loop
 *  Open window with initial window size,  title bar, color
 *  index display mode, and handle input events.
```

```
 */
int main(int argc, char** argv)
{
  auxInitDisplayMode(AUX_SINGLE | AUX_INDEX | AUX_DEPTH16);
  auxInitPosition(0, 0, 500, 500);
  auxInitWindow("Lighting in Color Map Mode");
  myinit();
  auxReshapeFunc(myReshape);
  auxMainLoop(display);
  return(0);
}
```

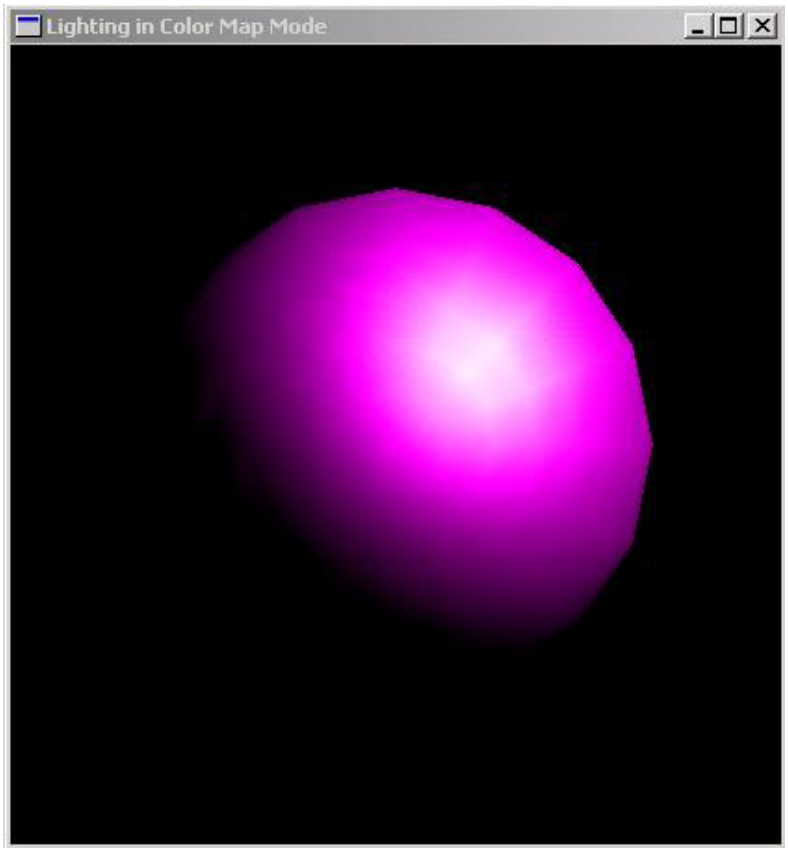

图 5-6 色彩索引模式下的光照