目　录

1 OpenGL 的基本框架
1.1 OpenGL 简介……1
1.2 OpenGL 的工作方式……2
1.3 OpenGL 的操作步骤……3
1.4 OpenGL 的组成……3
1.5 OpenGL 的数据类型……4
1.6 OpenGL 函数命名约定……4
1.7 用 OpenGL 绘制图形……4
1.8 用 OpenGL 制作动画……9
2 图形的绘制
2.1 空间点的绘制……13
2.2 直线的绘制……14
2.3 多边形面的绘制……18
2.4 平面多面体的绘制……24
3 图形变换
3.1 OpenGL 中的变换……30
3.2 模型视图矩阵……31
3.3 矩阵堆栈……35
4 OpenGL 中的颜色、光照和材质
4.1 颜色……42
4.2 光照模型……42
4.3 材质属性……43
4.4 使用光照……43
4.5 使用光源……48
附录：参考函数
1.1 颜色使用……58
1.2 绘制几何图元……59
1.3 坐标转换……63
1.4 堆栈操作……66
1.5 使用光照和材质……68
1.6 帧缓存操作……72
1.7 查询函数……72
1.8 窗口初始化和启动事件处理……75
1.9 窗口管理……77
1.10 菜单管理……80
1.11 注册回调函数……82
1.12 几何图形绘制……84

1 OpenGL 的基本框架

1.1 OpenGL 简介

在计算机发展初期，人们就开始从事计算机图形的开发，但直到 20 世纪 80 年代末 90 年代初，三维图形才开始迅速发展。于是各种三维图形工具软件包相继推出，如 GL，RenderMan 等，但没有一种软件包能够在三维图形建模能力和编程方便程度上与 OpenGL 相比拟。

OpenGL（Open Graphics Library，开放图形库），是一个三维的计算机图形和模型库，它源于 SGI 公司为其图形工作站开发的 IRIS GL，在跨平台移植过程中发展成为 OpenGL。SGI 公司在 1992 年 6 月发布 1.0 版，后成为工业标准。目前，OpenGL 标准由 1992 年成立的独立财团 OpenGL Architecture Review Board（ARB）以投票方式产生，并制成规范文档公布，各软硬件厂商据此开发自己系统上的实现。目前最新版规范是 1999 年 5 月通过的 1.2.1。

OpenGL 作为一个性能优越的图形应用程序设计界面（API），它独立于硬件和窗口系统，在运行各种操作系统的各种计算机上都可用，并能在网络环境下以客户/服务器模式工作，是专业图形处理、科学计算等高端应用领域的标准图形库。

OpenGL 在军事、广播电视、CAD/CAM/CAE、娱乐、艺术造型、医疗影像、虚拟世界等领域都有着广泛的应用。它具有以下功能。

1. 模型绘制

OpenGL 能够绘制点、线和多边形。应用这些基本的形体，可以构造出几乎所有的三维模型。OpenGL 通常用模型的多边形的顶点来描述三维模型。

2. 模型观察

在建立了三维景物模型后，就需要用 OpenGL 描述如何观察所建立的三维模型。观察三维模型是通过一系列的坐标变换进行的。模型的坐标变换在使观察者能够在视点位置观察与视点相适应的三维模型景观。在整个三维模型的观察过程中，投影变换的类型决定观察三维模型的观察方式，不同的投影变换得到的三维模型的景象也是不同的。最后的视窗变换则对模型的景象进行裁剪缩放，即决定整个三维模型在屏幕上的图象。

3. 颜色模式的指定

OpenGL 应用了一些专门的函数来指定三维模型的颜色。程序开发者可以选择二个颜色模式，即 RGBA 模式和颜色表模式。在 RGBA 模式中，颜色直接由 RGB 值来指定；在颜色表模式中，颜色值则由颜色表中的一个颜色索引值来指定。开发者还可以选择平面着色和光滑着色二种着色方式对整个三维景观进行着色。

4. 光照应用

用 OpenGL 绘制的三维模型必须加上光照才能更加与客观物体相似。OpenGL 提供了管理四种光(辐射光、环境光、镜面光和漫射光)的方法，另外还可以指定模型表面的反射特性。

5. 图象效果增强

OpenGL 提供了一系列的增强三维景观的图象效果的函数，这些函数通过反走样、混合和雾化来增强图象的效果。反走样用于改善图象中线段图形的锯齿而更平滑，混合用于处理模型的半透明效果，雾使得影像从视点到远处逐渐褪色，更接近于真实。

6. 位图和图象处理

OpenGL 还提供了专门对位图和图象进行操作的函数。

7. 纹理映射

三维景物因缺少景物的具体细节而显得不够真实，为了更加逼真地表现三维景物，OpenGL 提供了纹理映射的功能。OpenGL 提供的一系列纹理映射函数使得开发者可以十分方便地把真实图象贴到景物的多边形上，从而可以在视窗内绘制逼真的三维景观。

8. 实时动画

为了获得平滑的动画效果，需要先在内存中生成下一幅图象，然后把已经生成的图象从内存拷贝到屏幕上，这就是 OpenGL 的双缓存技术(double buffer)。OpenGL 提供了双缓存技术的一系列函数。

9. 交互技术

目前有许多图形应用需要人机交互，OpenGL 提供了方便的三维图形人机交互接口，用户可以选择修改三维景观中的物体。

1.2 OpenGL 的工作方式

1.2.1 OpenGL 的体系结构

OpenGL 是一套图形标准，它严格按照计算机图形学原理设计而成，符合光学和视觉原理，非常适合可视化仿真系统。

由于 OpenGL 是一种 API，其中不包含任何窗口管理、用户交互或文件 I/O 函数。每个主机环境（如 Microsoft Windows）在这些方面都有自己的函数，由这些函数负责实现某些方法，以便把窗口或位图的绘制控制权移交给 OpenGL。通常，一个完整的窗口系统的 OpenGL 图形处理系统的结构如图 1.1 所示：最底层为图形硬件，第二层为操作系统，第三层为窗口系统，第四层为 OpenGL，最上面的层为应用软件。

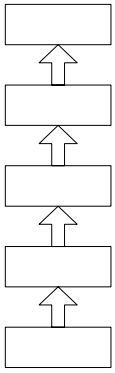

图 1-1 OpenGL 图形处理系统的层次结构

OpenGL 在 Windows NT 上的实现是基于客户机/服务器模式的，应用程序发出 OpenGL 命令，由动态链接库 OpenGL32.DLL 接受和打包后，发送到服务器端的 WINSRV.DLL，然后由它通过 DDI（Device Driver Interface，设备驱动程序接口）层发往视频显示驱动程序。如果系统安装了硬件加速器，则由硬件相关的 DDI 来处理。OpenGL/NT 的体系结构图如图 1.2 所示。

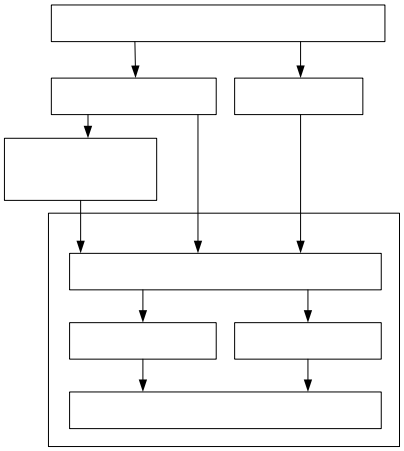

图 1-2 OpenGL/NT 体系结构

1.2.2 OpenGL 的流水线

当应用程序进行 OpenGL API 函数调用的时候，OpenGL 命令将被放在一个命令缓冲区中，这样，命令缓冲区中包含了大量的命令、顶点数据和纹理数据。当这个缓冲区被清空时，缓冲区中的命令和数据都将传递给流水线的下一个阶段，或者说，只有当命令缓冲区被清空时，OpenGL 命令才会被执行。图 1.3 显示了一条简化版的 OpenGL 流水线。

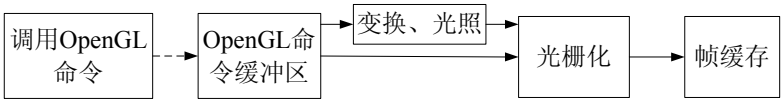

图 1-3 一条简化版的 OpenGL 流水线

在 OpenGL 中，命令缓冲区中的几何顶点数据通常还要进行几何变换以及光照计算，并通过指定的方法进行投影，为下一步光栅化做准备。光栅化根据图形的几何形状、颜色和纹理数据产生一系列图像的帧缓存地址和图元的二维描述值，光栅化的结果最后被放置在帧缓存中。帧缓存是图形显示设备的内存，这样图像就显示在屏幕上了。

1.2.3 OpenGL 状态机

OpenGL 是一种直接模式的 API，每条命令根据当前的渲染状态都会产生某种立即效果。渲染状态是各种标记，他们指出哪些特性是打开的，哪些是关闭的，以及应该如何应用他们。在 OpenGL 中，使用 glEnable 函数和 glDisable 函数来启用和禁用渲染特征。

1.3 OpenGL 的操作步骤

在 OpenGL 中进行的图形操作直至在计算机屏幕上渲染绘制出三维图形景观的基本步骤如下：

1. 根据基本图形单元建立景物模型，得到景物模型的数学描述（OpenGL 中把点、线、多边形、图像和位图都作为基本图形单元）；

2. 把景物模型放在三维空间中的合适的位置，并且设置视点(Viewpoint)以观察所感兴趣的景观；

3. 计算模型中所有物体的色彩，同时确定光照条件、纹理粘贴方式等；

4. 把景物模型的数学描述及其色彩信息转换至计算机屏幕上的像素，这个过程也就是光栅化(rasterization)。

在这些步骤的执行过程中，OpenGL 可能执行其他的一些操作，例如自动消隐处理等。另外，景物光栅化之后被送入帧缓冲器之前还可以根据需要对象素数据进行操作。

1.4 OpenGL 的组成

OpenGL 不是一种编程语言，而是一种 API（应用程序编程接口），它实际上是一种图形与硬件的接口，包括了多个图形函数。OpenGL 主要由以下函数库组成。

1. OpenGL 核心库

OpenGL 核心库中包含了 115 个最基本的命令函数，它们都是以“gl”为前缀，可以在任何 OpenGL 的工作平台上应用。这部分函数用于常规的、核心的图形处理，如建立各种各样的几何模型，产生光照效果，进行反走样以及进行纹理映射，以及进行投影变换等等。由于许多函数可以接收不同数据类型的参数，因此派生出来的函数原形有 300 多个。

2. OpenGL 实用程序库

OpenGL 的实用程序库包含有 43 个函数，以“glu”为前缀，在任何 OpenGL 平台都可以应用。这部分函数通过调用核心库的函数，来实现一些较为复杂的操作，如纹理映射、坐标变换、网格化、曲线曲面以及二次物体（圆柱、球体等）绘制等。

3. OpenGL 编程辅助库

OpenGL 的辅助库包含 31 个函数，以“aux”为前缀，但它们不能在所有的 OpenGL 平台上使用。OpenGL 的辅助库的函数主要用于窗口管理、输入输出处理以及绘制一些简单的三维形体。

4. OpenGL 实用程序工具包

OpenGL 实用程序工具包（OpenGL utility toolkit，GLUT）包含 30 多个函数，函数名前缀是“glut”。其中的函数主要提供基于窗口的工具，如窗口系统的初始化，多窗口管理，菜单管理，字体以及一些较复杂物体的绘制等。由于 glut 库中的窗口管理函数是不依赖于运行环境的，因此 OpenGL 中的工具库可以在所有的 OpenGL 平台上运行，在后面的示例中，我们均使用 glut 库建立 OpenGL 程序运行框架。

5. Windows 专用库

Windows 专用库函数包含有 6 个，每个函数以 wgl 开头，用于连接 OpenGL 和 Windows NT，这些函数用于在 Windows NT 环境下的 OpenGL 窗口能够进行渲染着色，在窗口内绘制位图字体以及把文本放在窗口的某一位置等这些函数把 Windows 和 OpenGL 揉合在一起。

6. Win32 API 函数库

这部分函数没有专用的前缀，主要用于处理像素存储格式和双帧缓存。

1.5 OpenGL 的数据类型

由于 OpenGL 具有平台无关性，它定义了自己的数据类型，这些数据类型将映射为常规的 C 数据类型，在程序中也可以直接使用这些 C 数据类型，下表列出了在 OpenGL 中定义的数据类型。

表 1-1　OpenGL 变量类型和相应的 C 数据类型

OpenGL 数据类型	内部表示法	定义为 C 类型	C 字面值后缀
GLbyte	8 位整数	signed char	b
GLshort	16 位整数	short	s
GLint，GLsizei	32 位整数	song	l
GLfloat，GLclampf	32 位浮点数	float	f
GLdouble，GLclampd	64 位浮点数	double	d
GLubyte，GLboolean	8 位无符号整数	unsigned char	ub
GLshort	16 位无符号整数	unsigned short	us
GLuint，GLenum，GLbitfield	32 位无符号整数	unsigned long	ui

1.6 OpenGL 函数命名约定

OpenGL 函数都遵循一个命名约定，通过这个约定可以了解函数来源于哪个库，需要多少个参数以及参数的类型。每一个函数都有一个根段，代表该函数相应的 OpenGL 命令。所有的 OpenGL 函数都采用以下格式：

<库前缀><根命令><可选的参数个数><可选的参数类型>

例如函数 glColor3f(…)，gl 表示这个函数来自库 gl.h，color 是该函数的根段，表示该函数用于颜色设定，3f 表示这个函数采用了三个浮点数参数。这种把参数数目和参数类型加入 OpenGL 函数结尾的约定使人们更容易记住参数列表而无需查找它。

1.7 用 OpenGL 绘制图形

由于 OpenGL 是一种 API，OpenGL 库遵循 C 调用约定，这意味着在 C 语言中编写的程

序可以很容易地调用 API 中的函数，本书的示例程序均以 C 语言编写。下面的程序利用 GLUT 库，在窗口的中心位置绘制一个矩形，其输出如图 1.4 所示。

程序清单 1.1：在窗口内绘制一个矩形

```
//GLRect.c
#include <windows.h>
#include <gl/glut.h>
#include<gl/gl.h>
#include<gl/glu.h>

// 函数 RenderScene 用于在窗口中绘制需要的图形
void RenderScene(void)
{
    //用当前清除色清除颜色缓冲区，即设定窗口的背景色
    glClear(GL_COLOR_BUFFER_BIT);

    //设置当前绘图使用的 RGB 颜色
    glColor3f(1.0f, 0.0f, 0.0f);

    //使用当前颜色绘制一个填充的矩形
    glRectf(100.0f, 150.0f, 150.0f, 100.0f);

    //刷新 OpenGL 命令队列
    glFlush();
}

// 函数 ChangeSize 是窗口大小改变时调用的登记函数
void ChangeSize(GLsizei w, GLsizei h)
{
    if(h == 0)      h = 1;

    //设置视区尺寸
    glViewport(0, 0, w, h);

    // 重置坐标系统，使投影变换复位
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    // 建立修剪空间的范围
    if (w <= h)
        glOrtho (0.0f, 250.0f, 0.0f, 250.0f*h/w, 1.0f, -1.0f);
    else
        glOrtho (0.0f, 250.0f*w/h, 0.0f, 250.0f, 1.0f, -1.0f);
```

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

//函数 SetupRC 用于初始化，常用来设置场景渲染状态
void SetupRC(void)
{
    // 设置窗口的清除色为白色
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
}

void main(void)
{
    //初始化 GLUT 库 OpenGL 窗口的显示模式
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);

    // 创建一个名为 GLRect 的窗口
    glutCreateWindow("GLRect");

    // 设置当前窗口的显示回调函数和窗口再整形回调函数
    glutDisplayFunc(RenderScene);
    glutReshapeFunc(ChangeSize);

    SetupRC();

    //启动主 GLUT 事件处理循环
    glutMainLoop();
}
```

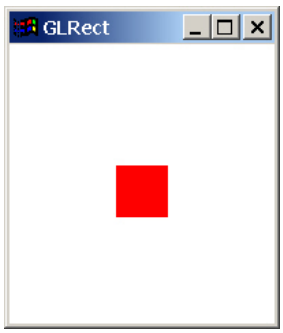

图 1-4 GLRect 程序运行结果

1.7.1 库和头文件

程序 GLRect 中包含了 4 个头文件，其中定义了程序所用的函数原形。此外，OpenGL 需要下列*.lib 包含在你的工程中：opengl.lib，glu.lib，glut32.lib；另外在运行程序路径下或\win98\system\（\winNT\system32）下需要一些动态连接库：opengl32.dll，glu32.dll，glut32.dll。

1.7.2 函数主体

我们先看所有 C 程序的入口点：

```
void main(void)
{
```

1. 显示模式

第一行代码如下：

```
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
```

它告诉 GLUT 库在创建窗口时使用单缓冲区窗口（GLUT_SINGLE）并使用 RGB 颜色模式（GLUT_RGB）。由于在屏幕上显示图形是由像素构成的，而像素的颜色及灰度信息都是存储在帧缓冲存储区中，单缓冲区窗口使用单一的帧缓冲存储区，这样所有的绘图命令都在显示窗口中执行；另一种是双缓冲区窗口（GLUT_DOUBLE），它使用了两个帧缓冲存储区，这样在窗口中执行的绘图命令实际上利用其中一个帧缓冲存储区创建一个场景，然后很快地交换到窗口视图中来，这种方法常用于产生动画效果的场合。RGB 颜色模式意味着要通过分别提供红、绿、蓝成分的浓度来指定颜色。

2. 创建 OpenGL 窗口

下一行代码：

```
glutCreateWindow("GLRect");
```

它利用 glut 库中的窗口管理函数在屏幕上创建一个标题为“GLRect”的窗口。

3. 回调函数

后面的两行代码

```
glutDisplayFunc(RenderScene);
glutReshapeFunc(ChangeSize);
```

分别用于指定当前窗口的显示回调函数和再整形回调函数。回调函数是响应某种事件而被调用的函数，他由程序员编制，通过 GLUT 注册函数连接到特定的函数。这样只要需要绘制窗口，GLUT 就会调用函数 RenderScene；而当窗口的大小或形状发生变化时，GLUT 会调用函数 ChangeSize。

4．设置上下文并执行

SetupRC()函数与 GLUT 框架无关，其作用是进行 OpenGL 的初始化。OpenGL 的初始化必须在渲染之前进行，由于 OpenGL 使用状态机，即每条 OpenGL 命令都使用当前的渲染状态完成，而对当前渲染状态的任何修改都会影响到之后的任何 OpenGL 命令，直至再次修改当前渲染状态，故此在一个场景中，许多的状态只要设置一次就可以了。

程序的结尾是最后一个 GLUT 函数调用：

```
glutMainLoop( );
}
```

该函数让 GLUT 框架开始运行，所有设置的回调函数开始工作，直到用户终止程序为止。

1.7.3 OpenGL 图形的绘制

1. 初始化

函数 SetupRC 主要用于初始化，其中只调用了一个函数

```
glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
```

这个函数用于设置清除窗口时使用的颜色，即设定窗口内的背景色。在 OpenGL 中，一种颜色用红、绿、蓝成分的混合来表示，每种成分的值可以是 0.0 到 1.0 之间的任意有效浮点数，这样虽理论上可以产生无限多种颜色，但实际可输出的颜色是有限的。这类似于 Windows 中使用 RGB 宏来创建 COLORREF 值时的颜色规范，不同的只是其中红、绿、蓝三种颜色

成分的取值范围是 0 到 255。表 1-2 给出了一些常见的混合色。

表 1-2　一些常用的混合色

混合色	红色成分（R）	绿色成分（G）	蓝色成分（B）
黑	0.0	0.0	0.0
红	1.0	0.0	0.0
绿	0.0	1.0	0.0
黄	1.0	1.0	0.0
蓝	0.0	0.0	1.0
紫	1.0	0.0	1.0
青	0.0	1.0	1.0
深灰	0.25	0.25	0.25
浅灰	0.75	0.75	0.75
棕	0.60	0.40	0.12
南瓜橙	0.98	0.625	0.12
粉红	0.98	0.04	0.70
紫红	0.60	0.40	0.70
白	1.0	1.0	1.0

glClearColor 的最后一个参数是 alpha 成分，主要用于混合的特殊效果，如半透明效果等。

2. 再整形回调函数

窗口在物理上是以象素数来测量的。开始在窗口中绘图之前，必须告诉 OpenGL 如何把指定的坐标对转换为屏幕坐标。为此，先指定窗口在笛卡儿空间中占据的区域，这一区域在计算机图形学中称为“窗口”，这里为了与操作系统的窗口有所区别，我们称这一区域为“修剪区”。而以象素计算的话，修剪区的宽度和高度很少会恰好与窗口的宽度和高度一致。所以坐标系必须从逻辑笛卡儿坐标映射为物理屏幕的象素坐标。这种映射是由称为“视区”的设置指定的。视区是窗口的客户区内用于绘制修剪区的区域，即把修剪区映射为窗口的一个区域。通常把视区定义充满整个窗口。

在程序 1.1 中，当窗口的大小改变，函数 ChangeSize 就会收到新的宽度和高度，在函数中利用这些信息可以完成修剪区到视区（坐标系到实际屏幕坐标）的映射。

（1）利用函数 glViewport 定义视区，glViewport 函数的定义如下：

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

x 和 y 参数指定窗口内视区的左下角，width 和 height 参数以象素为单位指定宽度和高度。通常，x 和 y 都是 0，但是可以用视区来渲染窗口不同区域中的多幅图形。

（2）利用函数 glOrtho 定义修剪区。这里需要注意一点，如果指定的视区不是正方形，而定义的修剪区是正方形，这样在完成修剪区到视区的映射后，显示的图形会发生变形，影响效果，为此必须保证定义的视区和修剪区的纵横比保持一致。glOrtho 函数的定义如下：

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

该函数在 3D 笛卡儿坐标空间中定义了一个修剪空间，left 和 right 指定 x 轴上显示的最小和最大坐标值；bottom 和 top 则用于 y 轴；near 和 far 参数用于 z 轴，通常是远离观察者的负值。由于这个修剪空间要映射到视区，这种三维空间到二维的映射，需要利用投影来实现，这里使用的是正投影。故此在调用函数 glOrtho 之前，调用了两个函数：

glMatrixMode(GL_PROJECTION);

glLoadIdentity();

这两个函数定义了投影矩阵，投影矩阵是实际定义视见空间的地方，而函数 glOrtho 并不是真正建立修剪空间，而是修改现有的修剪空间，它用描述了其参数中说明的修剪空间的矩阵乘以描述当前修剪空间的矩阵，为了避免每次调用 glOrtho 时对修剪空间造成进一步的破坏，所以使用函数 glLoadIdentity 使坐标系“复位”。

函数 ChangeSize 的最后为：

glMatrixMode(GL_MODELVIEW);

glLoadIdentity();

这两个函数告诉 OpenGL 将来所有变换都会影响模型。

3. 显示回调函数

显示回调函数 RenderScene 是调用 OpenGL 渲染函数的地方。

（1）实际清除

我们在初始化时指定了蓝色为窗口清除色，此时需要调用函数

glClear(GL_COLOR_BUFFER_BIT);

执行实际的清除操作。一幅图像的红、绿、蓝成分通常被并称为颜色缓冲区或象素缓冲区，用 glClear 清除颜色缓冲区会清除窗口中所有图形，并用指定的清除色填充整个窗口。

（2）指定当前绘图颜色

函数 glColor3f(1.0f，0.0f，0.0f)用于设置当前绘图颜色，即调用 glColor3f 设置以后，绘制操作（包括画线和填充）所用的颜色均为设置的当前颜色。

（3）图形绘制

这里我们使用函数

glRectf(100.0f, 150.0f, 150.0f, 100.0f);

来绘制一个矩形，该函数的四个参数表示矩形左上角点和右下角点的坐标。

（4）刷新队列

函数 glFlush（）用于让所有尚未执行的 OpenGL 命令都被执行。在内部，OpenGL 使用一条渲染流水线来顺序处理命令。OpenGL 命令和语言通常要排队，以便 OpenGL 驱动程序一次处理若干条“请求”。这种设置能提高性能，特别是在构造复杂对象的时候。GlFlush 函数只是告诉 OpenGL，它应处理到目前为止收到的绘图指令，而不要再等待更多的绘图命令。

由上面的程序可以看出，一个 OpenGL 程序的基本结构主要包括以下几个部分：

（1）定义窗口：包括指定窗口的大小、位置、显示模式以及设置各种回调函数；

（2）初始化设置：清除各种缓冲区，并设置各种 OpenGL 状态，例如设置背景色，打开光照，设置纹理等等。

（3）绘制场景：利用 OpenGL 函数绘制场景中的各种物体。

（4）变换：指定场景中需要显示的范围并指定由修剪区到视区的变换。

（5）结束运行：清除命令缓冲区，执行 OpenGL 命令。

1.8 用 OpenGL 制作动画

我们通常还可以用 OpenGL 程序创建动画效果，这里我们利用前面的例子，绘制正方形，并使这个正方形在窗口的边框反弹。这里需要创建一个循环，在每次调用显示回调函数之前改变正方形的位置，使其看起来像在窗口中移动。为了不断的调用显示回调函数，需要利用 GLUT 库中的函数：

glutTimerFunc(unsigned int msecs，(*func) (int value)，int value);

该函数用于指定一个定时器回调函数，即经过 msecs 毫秒后由 GLUT 调用指定的函数，并将 value 值传递给他。被定时器调用的函数原型如下：

```
void TimerFunction(int value);
```

注意，该函数与其他的回调函数不一样的地方在于该函数只会被激发一次。为了实现连续的动画，必须在定时器函数中再次重新设置定时器回调函数。

程序清单 1.2：在窗口内绘制一个移动的矩形

```
//MoveRect.c
#include <windows.h>
#include <gl/glut.h>
#include<gl/gl.h>
#include<gl/glu.h>

// 参数指定正方形的位置和大小
GLfloat x1 = 100.0f;
GLfloat y1 = 150.0f;
GLsizei rsize = 50;

// 正方形运动变化的步长
GLfloat xstep = 1.0f;
GLfloat ystep = 1.0f;

// 窗口的大小
GLfloat windowWidth;
GLfloat windowHeight;

void RenderScene(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0f, 0.0f, 0.0f);
    glRectf(x1, y1, x1+rsize, y1+rsize);

    //清空命令缓冲区并交换帧缓存
    glutSwapBuffers();
}

void ChangeSize(GLsizei w, GLsizei h)
{
    if(h == 0)      h = 1;

    glViewport(0, 0, w, h);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
```

```
    if (w <= h)
    {
        windowHeight = 250.0f*h/w;
        windowWidth = 250.0f;
    }
    else
    {
        windowWidth = 250.0f*w/h;
        windowHeight = 250.0f;
    }
    glOrtho(0.0f, windowWidth, 0.0f, windowHeight, 1.0f, -1.0f);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void TimerFunction(int value)
{
    // 处理到达窗口边界的正方形，使之反弹
    if(x1 > windowWidth-rsize || x1 < 0)    xstep = -xstep;
    if(y1 > windowHeight-rsize || y1 < 0)   ystep = -ystep;
    if(x1 > windowWidth-rsize)          x1 = windowWidth-rsize-1;
    if(y1 > windowHeight-rsize)         y1 = windowHeight-rsize-1;

    // 根据步长修改正方形的位置
    x1 += xstep;
    y1 += ystep;

    // 用新坐标重新绘图
    glutPostRedisplay();
    glutTimerFunc(33,TimerFunction, 1);
}

void SetupRC(void)
{
    //设置窗口清除色为蓝色
    glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
}

int main(void)
{
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutCreateWindow("Bounce");
```

```
    glutDisplayFunc(RenderScene);
    glutReshapeFunc(ChangeSize);
    glutTimerFunc(33, TimerFunction, 1);

    SetupRC();
    glutMainLoop();
}
```

这里，我们使用了双缓存技术来实现。双缓存技术使得执行的绘图代码能够在一个屏幕之外的缓冲区内进行渲染，然后用交换命令把图形瞬间放到屏幕上。这样在绘制动画的时候，每一帧都是在画面外的缓冲区中绘制，完成之后再很快地交换到屏幕上，这样会使动画比较平滑。在程序中，我们通过在窗口初始化时设定窗口模式为双缓冲区窗口，代码为：

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
```

此外，在显示回调函数的结尾我们使用了 glutSwapBuffers()函数代替 glFush()函数，该函数的作用是交换两个缓冲区的内容，即把隐藏的渲染好的图像放到屏幕上显示，并完成 OpenGL 流水线的刷新。

习题一

1. 在窗口的中心绘制一个球体。
2. 创建一个球体动画，使球体在窗口内做自由落体运动，并在撞击地面后能够弹回原来的高度。
3. 在第 2 题的基础上修改程序，使得球体可以沿着直线跳跃前进。

2 图形的绘制

2.1 空间点的绘制

最简单的计算机图形就是在屏幕上某个位置绘制一个点，并用特定的颜色绘制出来。请看下面的代码：

```
glBegin(GL_POINTS)；
    glVertex3f(0.0f，0.0f，0.0f)；
    glVertex3f(10.0f，10.0f，10.0f)；
glEnd()；
```

计算机中的图元只是把一组顶点或顶点列表解释为屏幕上绘制的某些形状，而顶点是用函数 glVertex3f 来定义，该函数中的参数指明定义点的 x、y 和 z 坐标。OpenGL 中定义的定点到放在函数 glBegin 和 glEnd 之间，由函数 glBegin 的参数指定绘制图元的类型，GL_POINTS 表示这个序列中绘制的是单个的点。注意一个 glBegin/glEnd 序列中可以包括任意多个相同类型的图元。表 2-1 列出了 glBegin 可支持的 OpenGL 图元。

表 2-1　glBegin 可支持的 OpenGL 图元

模式	图元类型
GL_POINTS	将指定的各个顶点用于创建单个的点
GL_LINES	将指定的顶点用于创建线段。每两个顶点指定一条单独的线段。如果顶点个数是奇数，则忽略最后一个
GL_LINE_STRIP	将指定的顶点用于创建线条。第一个顶点之后的每个顶点指定的是线条延伸到的下一个点
GL_LINE_LOOP	特性和 GL_LINE_STRIP 相似，只不过最后一条线段是在指定的最后一个和第一个顶点之间绘制。典型情况下，这用于绘制那些可能违反了 GL_POLYGON 用法规则的封闭区域
GL_TRIANGLES	将指定的顶点用于构造三角形。每三个顶点指定一个新三角形。如果顶点个数不是三的倍数，多余的顶点将被忽略
GL_TRIANGLE_STRIP	将指定的顶点用于创建三角条。指定前三个顶点之后，后继的每个顶点与它前面两个顶点一起用来构造下一个三角形。每个顶点三元组（在最初的组之后）会自动重新排列以确保三角形绕法的一致性。
GL_TRIANGLE_FAN	将指定的顶点用于构造三角扇形。第一个顶点充当原点，第三个顶点之后的每个顶点与它的前一个顶点还有原点一起组合。
GL_QUADS	每四个顶点一组用于构造一个四边形。如果顶点个数不是四的倍数，多余的顶点将被忽略
GL_QUADS_STRIP	将指定的顶点用于构造四条形边。在第一对顶点之后，每对顶点定义一个四边形。和 GL_QUADS 的顶点顺序不一样，每对顶点以指定顺序的逆序使用，以便保证绕法的一致
GL_POLYGON	将指定的顶点用于构造一个凸多边形。多边形的边缘决不能相交。最后一个顶点会自动连接到第一个顶点以确保多边形是封闭的

glVertex 函数用于指定顶点，它可以有 2，3，4 个参数。带 2 个参数时指定的是空间点的 x，y 坐标，其 z 坐标为默认值 0，在绘制平面图形时常常使用这类函数；带 3 个参数时

指定的是空间点的 x，y 和 z 坐标；带 4 个参数时，除了定义空间点的 x，y，z 坐标，还有一个不为 0 的 w 坐标，这样，点的坐标（x，y，z，w）实际上构成了一个齐次坐标。在 OpenGL 中，我们仍然使用规范化齐次坐标以保证点的齐次坐标与三维坐标的一一对应关系，最后指定的空间点的坐标为（x/w，y/w，z/w，1），w 成了坐标值的一个缩放因子。

在 OpenGL 中绘制一个点时，点大小的默认值是一个象素。可以用函数 glPointSize 修改这个值：

```
void glPointSize(GLfloat size);
```

这个函数采用一个参数来指定画点时以象素为单位的近似直径。但是不是任意大小点都支持，通常使用下面的代码来获取点大小的范围和它们之间最小的中间值：

```
GLfloat sizes[2];     //保存绘制点的尺寸范围
GLfloat step;         //保存绘制点尺寸的步长
glGetFloatv(GL_POINT_SIZE_RANGE，sizes);
glGetFloatv(GL_POINT_SIZE_GRANULARITY，&step);
```

在数组 seize 中包含两个元素，分别保存了 glPointSize 的最小有效值和最大有效值，而变量 step 将保存点大小之间允许的最小增量。指定范围之外的大小不会被解释为错误，而是使用最接近指定值的可支持的最大或最小尺寸。

在 OpenGL 程序中，我们常可以利用离散的点来拟合一些常见的曲线，如圆，螺旋线等等。

2.2 直线的绘制

使用模式 GL_LINES 可以在两点之间画线，如下面的代码在两点(0，0，0)和(10，10，10)之间画一条直线：

```
glBegin(GL_LINES);
    glVertex3f(0.0f，0.0f，0.0f);
    glVertex3f(10.0f，10.0f，10.0f);
glEnd();
```

注意，在 glBegin/glEnd 序列中两个顶点指定了一个图元（直线），如果序列中指定的点为奇数个，那么最后一个顶点将被忽略。

有时我们需要在一系列的顶点之间绘制连续直线，此时需要用到 GL_LINE_STRIP 或 GL_LINE_LOOP 模式。GL_LINE_STRIP 模式可以根据指定的一系列顶点，从一个顶点到另一个顶点用连续的线段画线。

```
glBegin(GL_LINE_STRIP);
    glVertex3f(0.0f，0.0f，0.0f);
    glVertex3f(10.0f，10.0f，0.0f);
    glVertex3f(20.0f，5.0f，0.0f);
glEnd();
```

上面这段代码实际上在 xy 平面内绘制了两条直线(0，0，0)到(10，0，0)和(0，10，0)到(20，5，0)。特别的，当沿着某条曲线指定一系列靠的很近的点，使用 GL_LINE_STRIP 模式可以绘制一条曲线。

GL_LINE_LOOP 模式与 GL_LINE_STRIP 模式类似，只是会在指定的最后一个顶点与第一个顶点之间画最后一条线。

直线的属性包括线宽和线型。在 OpenGL 中可用 glLineWidth 指定线宽：

```
void glLineWidth(GLfloat width)
```

与点的大小类似，glLineWidth 函数采用一个参数来指定要画的线以象素计的近似宽度，可以用下面的代码来获取线宽范围和它们之间的最小间隔：

```
GLfloat sizes[2];    //保存线宽的尺寸范围
GLfloat step;        //保存线宽尺寸的最小间隔
glGetFloarv(GL_LINE_WIDTH_RANGE，sizes);
glGetFloatv(GL_LINE_WIDTH_GRANULARITY，&step);
```

数组 seize 中保存了 glLineWidth 的最小有效值和最大有效值，而变量 step 将保存线宽之间允许的最小增量。OpenGL 规范只要求支持一种线宽：1.0。Microsoft 的 OpenGL 实现允许线宽从 0.5 到 10.0，最小增量为 0.125。

除了修改线宽，可以用虚线或短划线模式创建直线，为此需要先调用：

```
glEnable(GL_LINE_STIPPLE);
```

然后，函数 glLineStipple 将建立用于画线的模式：

```
glLineStipple(GLint factor，GLushort pattern);
```

其中，pattern 是一个 16 位值，他指定了画线时所用的模式。每一位代表线段的一部分是开还是关。默认情况下，每一位对应一个象素，但 factor 参数充当倍数可以增加模式的宽度。例如，将 factor 设为 2 会使模式中的每一位代表一行中 2 个象素的开或关。另外，在应用模式时，pattern 是逆向使用的，即模式的最低有效位最先作用于指定线段。图 2.1 说明了模式 0X00FF 是如何应用到线段上的。

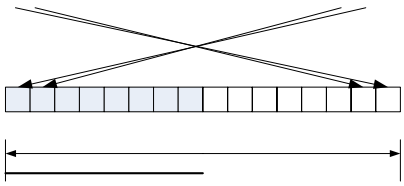

图 2-1 点划模式用于构造线段

在下面的程序演示了点的大小，线型以及线宽等等，输出图形如图 2.2 所示。

程序清单 2.1：点的大小，直线线型和线宽的示例

```
//PointAndLine.c
#include <windows.h>
#include <gl/glut.h>
#include<gl/gl.h>

void RenderScene(void)
{
    glClear(GL_COLOR_BUFFER_BIT); //用当前背景色填充窗口
    glColor3f(0.0f, 0.0f, 0.0f); //设置当前的绘图绘图 RGB 颜色

    GLfloat sizes[2];   //保存绘制点的尺寸范围
    GLfloat step;       //保存绘制点尺寸的步长
    GLfloat curSize;    //当前绘制的点的大小
    glGetFloatv(GL_POINT_SIZE_RANGE,sizes); //获得点的尺寸范围
    glGetFloatv(GL_POINT_SIZE_GRANULARITY,&step); //获得点尺寸的步长
```

Pattern = 0X00FF =

Binary = 0 0 0 0 0 0

```
    //绘制不同大小的点
    curSize=sizes[0];
    for (int i=0;i<25;i++)
    {
        glPointSize(curSize);   //设置点的大小
        glBegin(GL_POINTS);
            glVertex3f(25.0+i*8,200.0f,0.0f);
        glEnd();
        curSize +=step*2;
    }

    //绘制一条宽度为 5 的直线
    glLineWidth(5);   //设置线宽
    glBegin(GL_LINES);
        glVertex3f(25.0f,160.0f,0.0f);
        glVertex3f(225.0f,160.0f,0.0f);
    glEnd();

    //绘制一条虚线
    glEnable(GL_LINE_STIPPLE);
    glLineStipple(1,0x00FF); //设置点划线模式
    glBegin(GL_LINES);
        glVertex3f(25.0f,120.0f,0.0f);
        glVertex3f(225.0f,120.0f,0.0f);
    glEnd();

    //绘制一条宽度为 3 的点划线
    glLineWidth(3);
    glLineStipple(1,0xFF0C);
    glBegin(GL_LINES);
        glVertex3f(25.0f,80.0f,0.0f);
        glVertex3f(225.0f,80.0f,0.0f);
    glEnd();

    //增加重复因子绘制的点划线
    glLineStipple(4,0xFF0C);
    glBegin(GL_LINES);
        glVertex3f(25.0f,40.0f,0.0f);
        glVertex3f(225.0f,40.0f,0.0f);
    glEnd();
    glDisable(GL_LINE_STIPPLE);

    glFlush();//刷新 OpenGL 命令队列
}
```

```
void ChangeSize(GLsizei w, GLsizei h)
{
    if(h == 0)      h = 1;
    glViewport(0, 0, w, h); //设置视区尺寸

    // 重置坐标系统
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    // 建立修剪空间的范围
    if (w <= h)
        glOrtho (0.0f, 250.0f, 0.0f, 250.0f*h/w, 1.0f, -1.0f);
    else
        glOrtho (0.0f, 250.0f*w/h, 0.0f, 250.0f, 1.0f, -1.0f);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void SetupRC(void)
{
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f); // 设置窗口的背景色
}

void main(void)
{
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutCreateWindow("点与线");
    glutDisplayFunc(RenderScene); // 设置当前窗口的显示回调函数
    glutReshapeFunc(ChangeSize); // 为当前窗口设置窗口再整形回调函数
    SetupRC();
    glutMainLoop(); //启动主 GLUT 事件处理循环
}
```

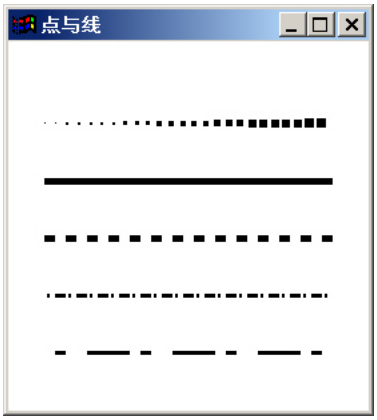

图 2-2 程序 PointAndLine 的运行结果

在上面的程序中，由于 OpenGL 使用的是一种状态机制，故此我们在绘制第一条直线前设置线宽为 5，不仅影响了第一条直线，还使绘制的虚线线宽也为 5，直到在绘制第一条点划线前将线宽改为 3 为止，这样后面绘制的直线线宽均为 3，直到再次修改为止。故此在绘制图形的时候，尽可能将状态相同或部分状态相同的图形放在一起绘制，可以提高程序的执行效率。

2.3 多边形面的绘制

1. 三角形的绘制

在 OpenGL 中，面是由多边形构成的。三角形可能是最简单的多边形，它有三条边。可以使用 GL_TRIANGLES 模式通过把三个顶点连接到一起而绘出三角形。下面的代码绘制了一个三角形：

```
glBegin(GL_TRIANGLES);
    glVertex2f(0.0，0.0);
    glVertex2f(15.0，15.0);
    glVertex2f(30.0，0.0);
glEnd();
```

注意，这里三角形将被用当前选定的颜色填充，如果尚未指定绘图的颜色，结果将是不确定的。

使用 GL_TRIANGLE_STRIP 模式可以绘制几个相连的三角形，系统根据前三个顶点绘制第一个多边形，以后每指定一个顶点，就与构成上一个三角形的后两个顶点绘制新的一个三角形。使用 GL_TRIANGLE_FAN 模式可以绘制一组相连的三角形，这些三角形绕着一个中心点成扇形排列。第一个顶点构成扇形的中心，用前三个顶点绘制出最初的三角形之后，随后的所有顶点都和扇形中心以及紧跟在它前面的顶点构成下一个三角形，此时是以顺时针方向穿过顶点。这两种模式的推进如图 2.3 和图 2.4 所示。

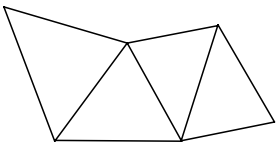

图 2-3 GL_TRIANGLE_STRIP 模式

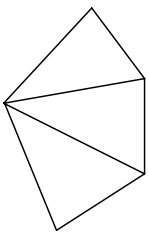

图 2-4 GL_TRIANGLE_FAN 模式

2. 绕法

在绘制三角形的过程中，三个顶点将三角形封闭的过程是有序的，即三角形的构成路径具有方向性，我们把指定顶点时顺序和方向的组合称为“绕法”。比如上面的例子中画出的三角形的绕法就是顺时针的，若把后两个顶点的位置互换，就得到了逆时针绕法。绕法是任何多边形图元的一个重要特性。一般默认情况下，OpenGL 认为逆时针绕法的多边形是正对着的，这一特性对于希望给多边形的正面和背面赋予不同的物理特性十分有用。如果要反转 OpenGL 的默认行为，可以调用函数：

```
glFrontFace(GL_CW);
```

CL_CW 告诉 OpenGL 应该把顺时针缠绕的多边形为正对着的。为了改回把逆时针绕法

视为正面，可以使用 CL_CCW。

3. 明暗处理

在绘制多边形时，我们常常要指定绘制的颜色，而在 OpenGL 中，颜色实际上是对各个顶点而不是对各个多边形指定的。多边形的轮廓或者内部用单一的颜色或许多不同的颜色来填充的处理方式称为明暗处理。在 OpenGL 中，用单一颜色处理的称为平面明暗处理（Flat Shading），用许多不同颜色处理的称为光滑明暗处理（Smooth Shading），也称为 Gourand 明暗处理（Gourand Shading）。

设置明暗处理模式的函数为：

void glShadeModel(GLenum mode);

其中参数 mode 的取值为 GL_FLAT 或 GL_SMOOTH，分别表示平面明暗处理和光滑明暗处理。应用平面明暗处理模式时，多边形内每个点的法向一致，且颜色也一致，OpenGL 用指定多边形最后一个顶点时的当前颜色作为填充多边形的纯色；应用光滑明暗处理模式时，多边形所有点的法向是由内插生成的，具有一定的连续性，因此每个点的颜色也相应内插，故呈现不同色。这种模式下，插值方法采用的是双线性插值法

Gouraud 明暗处理通常算法为：先用多边形顶点的光强线性插值出当前扫描线与多边形边交点处的光强，然后再用交点的光强线插值处扫描线位于多边形内区段上每一象素处的光强值。图 2.5 中显示出一条扫描线与多边形相交，交线的端点是 A 点和 B 点，P 点是扫描线上位于多边形内的任一点，多边形三个顶点的光强分别为 I1、I2 和 I3。取 A 点的光强 Ia 为 I1 和 I2 的线性插值，B 点的光强 Ib 为 I1 和 I3 的线性插值，P 点的光强 Ip 则为 Ia 和 Ib 的线性插值。

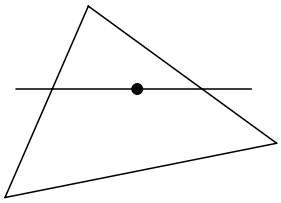

图 2-5 Gouraud 明暗处理

采用 Gouraud 明暗处理不但可以使用多边形表示的曲面光强连续，而且计算量很小。这种算法还可以以增量的形式改进，且能用硬件直接实现算法，从而广泛用于计算机实时图形生成。

4. 多边形的模式

多边形不是必须用当前颜色填充的。默认情况下绘制的多边形是实心的，但可以通过指定把多边形绘制为轮廓或只是点（只画出顶点）来修改这项默认行为。函数 glPolygonMode 允许把多边形渲染为填充的实心、轮廓线或只是点。另外，可以把这项渲染模式应用到多边形的两面或只应用到正面或背面。使用下面的函数可以改变多边形模式：

glPolygonMode(Glenum face，Glenum mode)；

其中，参数 face 指定多边形的哪一面受模式改变的影响——GL_FRONT，GL_BACK 或 GL_FRONT_AND_BACK。参数 mode 用于指定新的绘图模式。GL_FILL 是默认值，生成填充的多边形；GL_LINE 生成多边形的轮廓；而 GL_POINT 只画出顶点。GL_LINE 和 GL_POINT 绘制的点和线受 glEdgeFlag 所设置边缘标记的影响。

5. 多边形的绘制规则

在使用大量多边形构造一个复杂表面时，有两条重要规则。

P1

A

第一条规则是所有多边形都必须是平面的，也就是说，多边形的所有顶点必须位于一个平面上，不能在空间中扭曲。

第二条规则是多边形的边缘决不能相交，而且多边形必须是凸的。如果有任意两条边交叉，就称这个多边形与自己相交。“凸的”是指任意经过多边形的直线进入和离开多边形的次数不超过一次。

对于非凸多边形，可以把它分割成几个凸多边形（通常是三角形），再将它绘制出来。这样有出现了一个问题，如果这样的多边形被填充时看不到任何边缘，但如果切换到轮廓图形，就会看到组成大表面的所有小三角形，这会分散你的注意力。OpenGL 提供了一个特殊标记来处理这些边缘，称为边缘标记。在指定顶点的列表时，通过设置和清除边缘标记，可以通知 OpenGL 哪些线段被认为是边线（围绕图形边界的线），哪些线段不是（不应该显示的内部线段）。glEdgeFlag 函数用一个参数把边缘标记设为 True 或 false。当函数被设置为 True 时，后面的顶点将标记出边界线段的起点。

下面的程序演示了一个三角形，可以通过菜单修改其状态。

程序清单 2.2：三角形的绘制

```
//Triangle.c
#include <windows.h>
#include <gl/gl.h>
#include <gl/glut.h>

// 旋转参数
static GLfloat xRot = 0.0f;
static GLfloat yRot = 0.0f;

//确定多边形绕法的方向
BOOL bWinding = TRUE;

//初始化窗口
void SetupRC(void)
{
    // 设置窗口背景颜色为黑色
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f );
}

void ChangeSize(int w, int h)
{
    if(h == 0)  h = 1;

    glViewport(0, 0, w, h);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    if (w <= h)
```

```
            glOrtho (-100.0f, 100.0f, -100.0f*h/w, 100.0f*h/w, -100.0f, 100.0f);
        else
            glOrtho (-100.0f*w/h, 100.0f*w/h, -100.0f, 100.0f, -100.0f, 100.0f);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
}

// 在窗口中绘制图形
void RenderScene(void)
{
        glClear(GL_COLOR_BUFFER_BIT);

        // 旋转图形
        glPushMatrix();
        glRotatef(xRot, 1.0f, 0.0f, 0.0f);
        glRotatef(yRot, 0.0f, 1.0f, 0.0f);

        //设置点的大小以及线宽
        glPointSize(5);
        glLineWidth(5);

        //设置多边形绕法的方向是顺时针还是逆时针
        if (bWinding)
            glFrontFace(GL_CW);
        else
            glFrontFace(GL_CCW);

        /绘制三角形
        glBegin(GL_TRIANGLES);
            glColor3f(0.0f, 1.0f, 0.0f);
            glVertex3f(0, 60, 0);
            glColor3f(1.0f, 0.0f, 0.0f);
            glVertex3f(-60, -60, 0);
            glColor3f(0.0f, 0.0f, 1.0f);
            glVertex3f(60, -60, 0);
        glEnd();

        glPopMatrix();

        glutSwapBuffers();   // 刷新命令缓冲区
}
```

```
void SpecialKeys(int key, int x, int y)
{
    if(key == GLUT_KEY_UP)      xRot-= 5.0f;
    if(key == GLUT_KEY_DOWN)    xRot += 5.0f;
    if(key == GLUT_KEY_LEFT)    yRot -= 5.0f;
    if(key == GLUT_KEY_RIGHT)   yRot += 5.0f;

    if(xRot > 356.0f)    xRot = 0.0f;
    if(xRot < -1.0f)     xRot = 355.0f;
    if(yRot > 356.0f)    yRot = 0.0f;
    if(yRot < -1.0f)     yRot = 355.0f;

    // 刷新窗口
    glutPostRedisplay();
}

void ProcessMenu(int value)
{
    switch(value)
    {
        case 1:
            //修改多边形正面为填充模式
            glPolygonMode(GL_FRONT,GL_FILL);
            break;
        case 2:
            //修改多边形正面为线模式
            glPolygonMode(GL_FRONT,GL_LINE);
            break;
        case 3:
            //修改多边形正面为点充模式
            glPolygonMode(GL_FRONT,GL_POINT);
            break;
        case 4:
            //修改多边形反面为填充模式
            glPolygonMode(GL_FRONT,GL_FILL);
            break;
        case 5:
            //修改多边形反面为线模式
            glPolygonMode(GL_FRONT,GL_LINE);
            break;
        case 6:
            //修改多边形反面为点模式
            glPolygonMode(GL_FRONT,GL_POINT);
            break;
```

```
            case 7:
                //设置多边形的阴影模式为平面明暗模式
                glShadeModel(GL_FLAT);
                break;
            case 8:
                //设置多边形的阴影模式为光滑明暗模式
                glShadeModel(GL_SMOOTH);
                break;
            case 9:
                bWinding = !bWinding;
                break;
            default:
                break;
        }
        glutPostRedisplay();
}

int main(int argc, char* argv[])
{
        int nModeMenu;
        int nMainMenu;
        int nColorMenu;

        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB );
        glutCreateWindow("多边形演示");
        glutReshapeFunc(ChangeSize);
        glutSpecialFunc(SpecialKeys);   //设置特殊键响应回调函数
        glutDisplayFunc(RenderScene);

        // 创建一个子菜单并定义菜单回调函数
        nModeMenu = glutCreateMenu(ProcessMenu);
        //添加菜单项，1 表示选择菜单条目时传递的参数值
        glutAddMenuEntry("正面多边形填充模式",1);
        glutAddMenuEntry("正面线框模型",2);
        glutAddMenuEntry("正面点模式",3);
        glutAddMenuEntry("反面多边形填充模式",4);
        glutAddMenuEntry("反面线框模型",5);
        glutAddMenuEntry("反面点模式",6);

        //增加一个子菜单
        nColorMenu = glutCreateMenu(ProcessMenu);
        glutAddMenuEntry("平面明暗模式",7);
        glutAddMenuEntry("光滑明暗模式",8);
```

```
    //创建主菜单
    nMainMenu = glutCreateMenu(ProcessMenu);
    glutAddSubMenu("多边形模式", nModeMenu);
    glutAddSubMenu("颜色模式", nColorMenu);
    glutAddMenuEntry("改变绕法",9);

    //将创建的菜单与右键关联，即把菜单设置为右键弹出式菜单
    glutAttachMenu(GLUT_RIGHT_BUTTON);

    SetupRC();
    glutMainLoop();
    return 0;
}
```

在程序 4 中，我们增加了特殊键的控制，利用函数 glutSpecialFunc(SpecialKeys)设置了特殊键响应回调函数，并在这个函数中修改了两个用于控制 x 和 y 方向旋转角度的参数 xRot,yRot，然后在显示回调函数 RenderScene 中增加以下代码：

```
    glPushMatrix();
    glRotatef(xRot, 1.0f, 0.0f, 0.0f);
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);
    ……
    glPopMatrix();
```

这样使得通过方向键可以使整个场景绕着 x 轴和 y 轴旋转，关于旋转变换的内容在第三章将详细介绍。

2.4 平面多面体的绘制

最常见的形体是平面多面体，平行多面体是由多个多边形表面构成的封闭实体，这样我们可以通过构造平面多面体的各个面组合而成。当然，在 glut 库中还提供了一些常见实体的绘制函数，如球体、圆环、正四面体等，参见附录中的相关内容。下面的例子演示了利用 4 个三角形构造出了一个三棱锥的过程，其输出如图 2.6 所示。

程序清单 2.3：构造三棱锥

```
#include <windows.h>
#include <gl/gl.h>
#include <gl/glut.h>

// 旋转参数
static GLfloat xRot = 0.0f;
static GLfloat yRot = 0.0f;

BOOL bDepth = FALSE;//深度测试开关
BOOL bCull = FALSE;//剔除开关
```

```
void SetupRC(void)
{
    // 设置窗口背景颜色为黑色
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f );
    //指定多边形的阴影模式为平面阴暗模式
    glShadeModel(GL_FLAT);
}

void ChangeSize(int w, int h)
{
    if(h == 0)h = 1;

    glViewport(0, 0, w, h);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    if (w <= h)
        glOrtho (-100.0f, 100.0f, -100.0f*h/w, 100.0f*h/w, -100.0f, 100.0f);
    else
        glOrtho (-100.0f*w/h, 100.0f*w/h, -100.0f, 100.0f, -100.0f, 100.0f);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

// 在窗口中绘制图形
void RenderScene(void)
{
    //清除颜色及深度缓冲区
    glClear(GL_COLOR_BUFFER_BIT| GL_DEPTH_BUFFER_BIT);

    //是否开启设度测试
    if(bDepth)
        glEnable(GL_DEPTH_TEST);
    else
        glDisable(GL_DEPTH_TEST);

    //是否打开剔除
    if(bCull)
        glEnable(GL_CULL_FACE);
    else
        glDisable(GL_CULL_FACE);
```

```
    // 旋转图形
    glPushMatrix();
    glRotatef(xRot, 1.0f, 0.0f, 0.0f);
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);

    //指定顺时针绕法的多边形为多边形正面
    glFrontFace(GL_CW);

    //绘制三棱锥的三个棱面，他们的颜色分别为红、绿、蓝
    glBegin(GL_TRIANGLE_FAN);
        glVertex3f(0.0, 0.0, 80.0);
        glVertex3f(0.0, 50.0, 0.0);
        glColor3f(1.0,0.0,0.0);
        glVertex3f(50.0, -50.0, 0.0);
        glColor3f(0.0,1.0,0.0);
        glVertex3f(-50.0, -50.0, 0.0);
        glColor3f(0.0,0.0,1.0);
        glVertex3f(0.0, 50.0, 0.0);
    glEnd();

    //绘制三棱锥的底面，其颜色为黄色
    glBegin(GL_TRIANGLE_FAN);
        glVertex3f(0.0, 50.0, 0.0);
        glVertex3f(50.0, -50.0, 0.0);
        glColor3f(1.0,1.0,0.0);
        glVertex3f(-50.0, -50.0, 0.0);
    glEnd();

    glPopMatrix();

    // 刷新命令缓冲区
    glutSwapBuffers();
}

void SpecialKeys(int key, int x, int y)
{
    if(key == GLUT_KEY_UP)      xRot-= 5.0f;
    if(key == GLUT_KEY_DOWN)    xRot += 5.0f;
    if(key == GLUT_KEY_LEFT)    yRot -= 5.0f;
    if(key == GLUT_KEY_RIGHT)   yRot += 5.0f;

    if(xRot > 356.0f)    xRot = 0.0f;
    if(xRot < -1.0f)     xRot = 355.0f;
    if(yRot > 356.0f)    yRot = 0.0f;
```

```
    if(yRot < -1.0f)        yRot = 355.0f;

    // 刷新窗口
    glutPostRedisplay();
}

void ProcessMenu(int value)
{
    switch(value)
    {
        case 1:
            bDepth = !bDepth;
            break;
        case 2:
            bCull = !bCull;
            break;
        default:
            break;
    }
    glutPostRedisplay();
}

int main(int argc, char* argv[])
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB );
    glutCreateWindow("三棱锥");
    glutReshapeFunc(ChangeSize);
    glutSpecialFunc(SpecialKeys);
    glutDisplayFunc(RenderScene);

    // 创建右键菜单
    glutCreateMenu(ProcessMenu);
    glutAddMenuEntry("深度测试",1);
    glutAddMenuEntry("剔除背面",2);
    glutAttachMenu(GLUT_RIGHT_BUTTON);

    SetupRC();
    glutMainLoop();

    return 0;
}
```

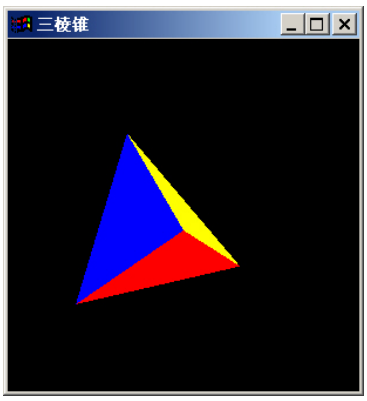

图 2-6 启用深度测试的三棱锥

在这个例子中，我们使用扇形绘制模式绘制了三棱锥的三个棱面，然后再绘制了三棱锥的底面。我们发现，黄色的底面先显示在屏幕上，而在旋转的过程中，红色的棱面始终不能显示出来，这是因为红色的棱面是最先绘制的，它总是被后面绘制的绿色、蓝色或黄色的多边形面所遮挡，要改变这种状况就需要启用深度测试。

1. 深度测试

在绘制图形的过程中，有时一个物体的一部分会被其前方的物体挡住（从观察者的角度看），如果这是这个物体在挡在其前面的物体绘制完成之后绘制，那么屏幕中显示的图形将不是我们所希望的，即后面的物体挡住了前面的物体。

只需启用一项称为深度测试的功能就可以解决这一问题，深度测试是一种移除被挡住表面的有效技术，它的过程是：在绘制一个象素时，会给他分配一个值（称为 z 值），这个值表示它与观察者的距离。然后，如果需要在同一个位置上绘制另一个象素，将比较新象素和已经保存在该位置的象素的 z 值。如果新象素的 z 值较大，即它离观察者更近因而在原来那个象素的前面，原来的象素就会被新象素挡住。这一操作在内部由深度缓冲区完成。

为了启用深度测试，只要调用：

```
glEnable(GL_DEPTH_TEST);
```

如果要关闭深度测试，则只需调用 glDisable(GL_DEPTH_TEST)即可。

另外为了使深度缓冲区正常完成深度测试功能，在每次渲染场景时，必须先清除深度缓冲区：

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

2. 隐藏表面

启用深度测试之后，我们得到了良好的视觉效果，但是还是付出了一些性能代价，因为每个画出的象素都必须与先前象素的 z 值比较。但是如果我们知道某些表面无论如何也不必画出，我们可以将其指出来，这种技术称为“剔除”，这种技术可以将已知的永远看不到的几何图形消除掉，这样可以显著的改善性能。

那么哪些表面是永远看不到的呢？最常见的例子就是封闭物体的内部表面，上面例子中的锥体是一个封闭表面，我们永远看不到它的内部。一种称为回溯剔除的技术可以消除表面的背面。

前面我们已经介绍过，OpenGL 使用绕法来确定多边形的正面和背面，这对以一致方向来缠绕那些确定对象外表面的多边形很重要。通过排除多边形的背面，我们可以大量减少渲染图像时必要的处理工作量。回溯剔除通过程序中的如下代码在程序中启用或禁用：

```
if(bCull)
    glEnable(GL_CULL_FACE);
```

```
else
    glDisable(GL_CULL_FACE);
```

启用剔除技术后，我们发现三棱锥的底面消失了，这是因为在绘制过程中，我们都使用了顺时针绕法的多边形为正面，但这样底面的正面正对着三棱锥的内部，故此启用剔除技术后把底面剔出掉了，要改变这种状况，可以在绘制三棱锥的底面前调用函数：

```
glFrontFace(GL_CCW);
```

3 图形变换

为了把一组图形融合为一个场景，必须把他们按照彼此之间的关系和与观察者的关系排列起来，这就要用到变换。变换使得把 3D 坐标投影到 2D 场景成为可能。变换还可以实现旋转对象、移动对象，甚至拉伸、压缩和弯曲他们。与直接修改对象不同的是，变换修改的是坐标系。

3.1 OpenGL 中的变换

OpenGL 中常用的变换包括视图变换、模型变换、模型视图变换、投影变换和视见区变换。

1. 视图变换

视图变换是第一个应用到场景上的变换。它用于确定场景的有利位置。默认情况下，观察点位于原点，顺着 z 轴的负向看。视图变换使你可以将观察点放置在任何期望的位置上，从任何方向进行观察。确定视图变换相当于在场景中放置一部摄像机并确定其指向。

在工作时间表上，必须先指定视图变换再指定其他任何变换。

2. 模型变换

模型变换用于处理模型和模型内的特定对象。这些变换把对象移动到合适的位置，旋转他们，并对它们进行缩放。模型变换实际上就是几何变换。

3. 模型视图变换

从内部效果和它们对场景最终外观的效果来说，视图变换和模型变换是相同的，把这两者分开完全是为了程序员的方便，向后移动对象和向前移动参考系之间并没有本质差别。术语“模型视图”表示你可以把这类变换视为模型变换或视图变换，但实际上并无区别，因此称它为模型视图变换。

4. 投影变换

投影变换应用到最终的模型视图方向。投影实际上定义了视见空间并建立了修剪平面。更明确一些，投影变换指定已完成场景如何转换成屏幕上的最终图象。常用的投影变换包括正投影和透视投影。

5. 视见区变换

完成上述变换之后，需要把场景的一个二维投影映射到屏幕上某处的窗口。这个到物理窗口坐标的映射是最后完成的一项变换，它称为视见区变换。

在计算机图形学中，所有的变换都是通过矩阵乘法来实现的，即将物体顶点构成的齐次坐标矩阵乘以三维变换矩阵就得到了变换后的物体顶点的齐次坐标矩阵，这样只要求出物体的三维变换矩阵，就可以得到变换后的物体。在 OpenGL 中，对象的坐标变换也通过矩阵来实现的（虽然并不一定要自己定义矩阵）。通常，在 OpenGL 中使用了两个重要的矩阵：模型视图矩阵和投影矩阵，其中模型视图矩阵用于物体的模型视图变换，投影矩阵用于投影变换。

有了模型视图矩阵和投影矩阵，在 OpenGL 中从未加工的顶点数据到屏幕坐标的过程就包括如下步骤

1. 将顶点坐标转换为规范化齐次坐标矩阵；

2. 将顶点的规范化齐次坐标矩阵乘以模型视图矩阵，生成变换后的顶点齐次坐标矩阵；

3. 将变换后的顶点齐次坐标矩阵乘以投影矩阵，生成修剪坐标矩阵，这样就有效地排除了视见空间之外的所有数据。

4. 由修剪坐标除以 w 坐标，得到规格化的设备坐标。注意，在变换过程中，模型视图矩阵和投影视图矩阵都有可能改变坐标的 w 值。

5. 坐标三元组被视见区变换影射到一个 2D 平面。这也是一个矩阵变换，但在 OpenGL 中不能直接指定或修改它，系统会根据指定给 glViewport 的值在内部设置它。

3.2 模型视图矩阵

模型视图矩阵用于指定场景的视图变换和几何变换，模型视图矩阵是一个 4×4 的矩阵，它代表了你用来放置和定位对象的经过变换的坐标系。

在计算机图形学中，物体进行的几何变换可以分解成多个基本几何变换，而三维几何变换矩阵是多个基本几何变换矩阵相乘的结果。OpenGL 提供了一些函数用于把任意矩阵设置成当前操作的矩阵（模型视图矩阵或投影矩阵），请看下面的例子：

```
glfloat m[] = { 1.0f,   0.0f,   3.0f,   0.0f,
                0.0f,   1.0f,   0.0f,   1.0f,
                0.0f,   0.0f,   1.0f,   1.0f,
                0.0f,   0.0f,   0.0f,   1.0f, };
glMatrixMode(GL_MODELVIEW);
glLoadMatrixf(m);
```

这段程序中，先声明了一个数组来保存 4×4 矩阵的值，注意这里矩阵按列优先顺序保存，这意味着先从上往下遍历每一列；然后使用 glLoadMatrix 函数将定义的矩阵设置为当前操作的矩阵。如果需要执行变换，即把定义的矩阵乘到模型视图矩阵中，可以使用函数 glMultmatrixf，其函数原型如下：

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

其中参数 m 为一个以列优先顺序保存 16 个连续值的数组。

实际上，在 OpenGL 中，我们通过一些高级矩阵函数将模型视图矩阵乘以指定的变换矩阵，并将结果矩阵设置成当前的模型视图矩阵。常用的高级矩阵函数如下。

1. 平移变换

平移变换函数如下：

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

三个参数就是目标分别沿三个轴的正向平移的偏移量。这个函数用这三个偏移量生成的矩阵完成乘法。当参数是(0. 0, 0. 0, 0. 0)时，生成的矩阵是单位矩阵，此时对物体没有影响。

2. 旋转变换

旋转变换函数如下：

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

函数中第一个参数是表示目标沿从原点到指定点(x,y,z)的方向矢量逆时针旋转的角度，后三个参数则是指定旋转方向矢量的坐标。这个函数表示用这四个参数生成的矩阵完成乘法。当角度参数是 0.0 时，表示对物体没有影响。

3. 缩放变换

缩放变换函数如下：

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

三个参数值就是目标分别沿三个轴向缩放的比例因子。这个函数用这三个比例因子生成的矩阵完成乘法。这个函数能完成沿相应的轴对目标进行拉伸、压缩和反射三项功能。当参

数是(1.0,1.0,1.0)时，对物体没有影响。当其中某个参数为负值时，表示将对目标进行相应轴的反射变换，且这个参数不为 1.0，则还要进行相应轴的缩放变换。最好不要令三个参数值都为零，这将导致目标沿三轴都缩为零。

通过上述的高级矩阵函数，我们可以很方便地实现变换，但是这里存在一个问题：在调用函数时，修改的是当前的模型视图矩阵。新的矩阵随后将成为当前的模型视图矩阵并影响此后绘制的图形。这样模型视图矩阵函数在调用时，就会有造成效果的积累，请看下面一段代码：

```
//沿 x 轴正向移动 10 个单位
glTranslatef(10.0f, 0.0f, 0.0f);
glutSolidSphere(1.0f, 15, 15);
//沿 y 轴正向移动 10 个单位
glTranslatef(0.0f, 10.0f, 0.0f);
glutSolidSphere(1.0f);
```

这段代码绘制了两个球，第一个绘制的球的球心沿 x 轴正向移动了 10 个单位，而第二个球不仅沿 y 轴正向移动 10 个单位，也沿 x 轴正向移动 10 个单位。这就是效果积累。如果想要第二个球只沿 y 轴正向移动 10 个单位，一种简单的方法是把模型矩阵复位，即通过给模型视图矩阵加载上单位矩阵来复位原点。下面的代码把单位矩阵加载到模型视图矩阵中：

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

第一个函数用于指定当前操作的矩阵，其函数原型如下：

```
glMatrixMode(GLenum mode);
```

其中参数 mode 用于确定将哪个矩阵堆栈用于矩阵操作，它的取值有：GL_MODELVIEW（模型视图矩阵堆栈），GL_PROJECTION（投影矩阵堆栈）和 GL_TEXTURE（纹理矩阵堆栈）。一旦设置了当前操作矩阵，它就将保持为活动的矩阵，直到你修改它为止。

第二个函数作用就是给当前操作矩阵加载上单位矩阵，。

下面的程序演示了一个三角形进行的平移、旋转和缩放等变换，其输出如图 3-1 所示。

程序 6：几何变换

```
#include <windows.h>
#include <gl/gl.h>
#include <gl/glut.h>

void SetupRC(void)
{
    // 设置窗口背景颜色为白色
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f );
}

//绘制三角形
void DrawTriangle(void)
{
    glBegin(GL_TRIANGLES);
        glVertex2f(0.0f, 0.0f);
        glVertex2f(40.0f, 0.0f);
        glVertex2f(20.0f, 40.0f);
    glEnd();
```

```
}

void ChangeSize(int w, int h)
{
    if(h == 0)h = 1;

    glViewport(0, 0, w, h);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    if (w <= h)
        glOrtho (-100.0f, 100.0f, -100.0f*h/w, 100.0f*h/w, -100.0f, 100.0f);
    else
        glOrtho (-100.0f*w/h, 100.0f*w/h, -100.0f, 100.0f, -100.0f, 100.0f);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void RenderScene(void)
{
    glClear(GL_COLOR_BUFFER_BIT );

    // 设置当前操作矩阵为模型视图矩阵，并复位为单位矩阵
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    //绘制黑色的坐标轴
    glColor3f(0.0f, 0.0f, 0.0f);
    glBegin(GL_LINES);
        glVertex2f(-100.0f, 0.0f);
        glVertex2f(100.0f, 0.0f);
        glVertex2f(0.0f, -100.0f);
        glVertex2f(0.0f, 100.0f);
    glEnd();

    //绘制出第一个红色的三角形
    glColor3f(1.0f, 0.0f, 0.0f);
    DrawTriangle();

    //绘制出逆时针旋转 200 度角的绿色三角形
    glRotatef(200.0f,0.0f,0.0f,1.0f);
    glColor3f(0.0f, 1.0f, 0.0f);
```

```
        DrawTriangle();

        //绘制出沿 x 轴负方向平移 40 单位的黄色三角形
        glLoadIdentity();
        glTranslatef(-60.0f,0.0f,0.0f);
        glColor3f(1.0f, 1.0f, 0.0f);
        DrawTriangle();

        //绘制出缩放因子为 1.0，2.0，1.0 的蓝色三角形
        glTranslatef(100.0f, 10.0f, 0.0f);
        glScalef(1.0f, 2.0f, 1.0f);
        glColor3f(0.0f, 0.0f, 1.0f);
        DrawTriangle();

        glutSwapBuffers();
}

int main(int argc, char* argv[])
{
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB );
        glutCreateWindow("±ä»»ÊµÀý");
        glutReshapeFunc(ChangeSize);
        glutDisplayFunc(RenderScene);

        SetupRC();
        glutMainLoop();

        return 0;
}
```

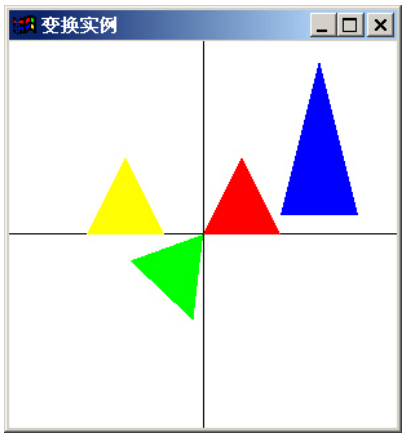

图 3-1 三角形的变换

在上面的程序中，我们首先确定了当前操作的矩阵为模型视图矩阵，然后绘制三角形。注意，在绘制第二个黄色的三角形之前，我们调用了函数 glLoadIdentity()使模型视图矩阵复

位，以抵消积累效果；而在绘制第三个蓝色三角形时，我们没有调用函数 glLoadIdentity()，这样，蓝色三角形进行的变换实际上包括三个步骤：先沿着 x 轴负方向平移 40 个单位，再沿着 x 轴正向平移 100 单位且沿 y 轴正方向平移 10 个单位，最后进行缩放操作，沿 z 轴放大 2 倍。

3.3 矩阵堆栈

实际上，在创建、装入、相乘模型变换和投影变换矩阵时，都已用到堆栈操作。一般说来，矩阵堆栈常用于构造具有继承性的模型，即由一些简单目标构成复杂模型。而在构造复杂模型时，我们常常需要保存中间的变化状态，以便在进行一些变换后能够恢复。为了简化这种操作，OpenGL 为模型视图矩阵和投影矩阵各维护着一个“矩阵堆栈”，他的工作方式与通常的程序堆栈完全一样。我们可以把当前矩阵压到堆栈中保存它，然后对当前矩阵进行修改。把矩阵弹出堆栈即可恢复。使用的函数如下：

```
void glPushMatrix(void);
void glPopMatrix(void);
```

函数 glPushMatrix 用于将当前矩阵压入当前矩阵堆栈，常用于保存当前变换矩阵。函数 glPopMatrix 用于将最后一个（最顶上的）矩阵弹出当前矩阵堆栈，常用于恢复当前变换矩阵原先的状态，如果曾经调用 glPushMatrix 把它保存起来的话。

堆栈是有深度的，如果超出了堆栈深度或当堆栈为空时试图弹出一个矩阵值，都会发生错误。可以用下面的函数获取堆栈深度的最大值：

```
glGet(GL_MAX_MODELVIEW_STACK_DEPTH);
glGet(GL_MAX_PROJECTION_STACK_DEPTH);
```

3.4 使用投影

OpenGL 中只提供了两种投影方式，一种是平行投影（正射投影），另一种是透视投影。不管是调用哪种投影函数，为了避免不必要的变换，其前面必须加上以下两句，以指定当前处理的矩阵是投影变换矩阵：

```
glMAtrixMode(GL_PROJECTION)；
glLoadIdentity()；
```

1. 正射投影

正射投影，又叫平行投影，它的视景体是一个矩形的平行管道，也就是一个长方体，其特点是无论物体距离相机多远，投影后的物体大小尺寸不变。OpenGL 中正射投影函数共有两个。一个函数是：

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

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

另一个函数是：

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

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

角点为（left，bottom）、右上角点为（right，top）的矩形。

2. 透视投影

透视投影的特点是距离视点近的物体大，距离视点远的物体小，远到极点即为消失，成为灭点。它的视景体类似于一个顶部和底部都被切除掉的棱椎，也就是棱台。OpenGL 透视投影函数也有两个，其中一个函数是：

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

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

另一个函数是：

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

它也创建一个对称透视视景体，但它的参数定义于前面的不同，其操作是创建一个对称的透视投影矩阵，并且用这个矩阵乘以当前矩阵。参数 fovy 定义视野在 X-Z 平面（垂直方向上的可见区域）的角度，范围是[0.0, 180.0]；参数 aspect 是投影平面的纵横比（宽度与高度的比值）；参数 zNear 和 Far 分别是远近裁剪面沿 Z 负轴到视点的距离，它们总为正值。

下面的程序演示了一个原子动画模型，其中用一个在中间的球体表示原子核，另外三个球体表示电子，三个电子沿着不同的轨道绕着原子核旋转。其输出如图 3-2 所示。

程序 7：原子核示例

```
#include <windows.h>
#include <gl/gl.h>
#include <gl/glu.h>
#include <gl/glut.h>

void SetupRC()
{
    glEnable(GL_DEPTH_TEST);    // 启用深度测试
    glFrontFace(GL_CCW);        // 指定逆时针绕法表示多边形正面
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f );      //背景为白色
}

void ChangeSize(int w, int h)
{
    if(h == 0)h = 1;

    // 设置视区尺寸
    glViewport(0, 0, w, h);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    // 设置修剪空间
```

```
    GLfloat fAspect;
    fAspect = (float)w/(float)h;
    gluPerspective(45.0, fAspect, 1.0, 500.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void RenderScene(void)
{
    // 绕原子核旋转的角度
    static float fElect1 = 0.0f;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // 重置模型视图矩阵
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    //将图形沿 z 轴负向移动
    glTranslatef(0.0f, 0.0f, -250.0f);

    // 绘制红色的原子核
    glColor3f(1.0f, 0.0f, 0.0f);
    glutSolidSphere(10.0f, 15, 15);

    // 当前绘制颜色变为黑色
    glColor3f(0.0f, 0.0f, 0.0f);

    //绘制第一个电子
    //保存当前的模型视图矩阵
    glPushMatrix();
    glRotatef(fElect1, 0.0f, 1.0f, 0.0f);//绕 y 轴旋转一定的角度
    glTranslatef(90.0f, 0.0f, 0.0f);//平移一段距离
    glutSolidSphere(6.0f, 15, 15);//画出电子

    // 恢复矩阵
    glPopMatrix();

    // 第二个电子
    glPushMatrix();
    glRotatef(45.0f, 0.0f, 0.0f, 1.0f);
    glRotatef(fElect1, 0.0f, 1.0f, 0.0f);
    glTranslatef(-70.0f, 0.0f, 0.0f);
```

```
        glutSolidSphere(6.0f, 15, 15);
        glPopMatrix();

        // 第三个电子
        glPushMatrix();
        glRotatef(-45.0f,0.0f, 0.0f, 1.0f);
        glRotatef(fElect1, 0.0f, 1.0f, 0.0f);
        glTranslatef(0.0f, 0.0f, 60.0f);
        glutSolidSphere(6.0f, 15, 15);
        glPopMatrix();

        // 增加旋转步长
        fElect1 += 10.0f;
        if(fElect1 > 360.0f) fElect1 = 10.0f;

        glutSwapBuffers();
}

void TimerFunc(int value)
{
        glutPostRedisplay();
        glutTimerFunc(100, TimerFunc, 1);
}

int main(int argc, char* argv[])
{
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
        glutCreateWindow("原子示例");
        glutReshapeFunc(ChangeSize);
        glutDisplayFunc(RenderScene);
        glutTimerFunc(500, TimerFunc, 1);

        SetupRC();
        glutMainLoop();
        return 0;
}
```

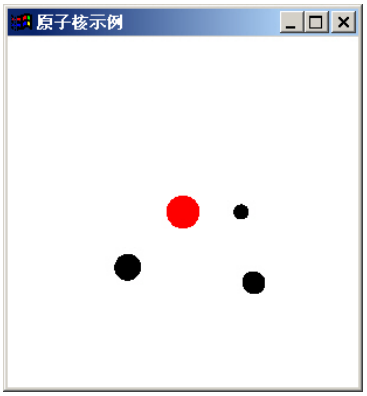

图 3-2 原子电子示例

本程序使用了透视投影，可以清楚地看到电子在旋转到远处时，体积逐渐变小；而旋转到近处时，又渐渐变大。

6. 裁剪变换

在 OpenGL 中，除了利用投影变换中视景体定义的六个裁剪平面（上、下、左、右、前、后）外，我们还可继续再定义一个或多个附加裁剪平面，对场景进一步进行裁减，以去掉场景中无关的目标。

附加平面裁剪函数为：

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

这个函数定义一个附加的裁剪平面。其中参数 equation 指向一个包含四个双精度浮点数的数组，这四个系数分别是裁剪平面 Ax+By+Cz+D=0 的 A、B、C、D 值。因此，由这四个系数就能确定一个裁剪平面。参数 plane 是符号常数 GL_CLIP_PLANEi(i=0,1,...)，用于指定裁剪面号。

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

在下面的程序中，显示对一个网状球体的裁减，其输出如图 3-3 所示。

程序 8：裁减平面示例

```
#include <windows.h>
#include <gl/glut.h>
#include<gl/gl.h>
#include<gl/glu.h>

// 旋转参数
static GLfloat xRot = 0.0f;
static GLfloat yRot = 0.0f;

void SetupRC(void)
{
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
    glShadeModel (GL_FLAT);
}
```

```
void ChangeSize(GLsizei w, GLsizei h)
{
    if(h == 0)      h = 1;

    // 设置视区尺寸
    glViewport(0, 0, w, h);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    // 建立修剪空间的范围
    gluPerspective(60.0f, (GLfloat) w/(GLfloat) h, 1.0f, 20.0f);

    glMatrixMode(GL_MODELVIEW);
}

// 在窗口中绘制图形
void RenderScene(void)
{
    //定义裁减平面方程系数，这里平面方程为 x = 0
    GLdouble eqn[4] = {1.0, 0.0, 0.0, 0.0};

    glClear(GL_COLOR_BUFFER_BIT);

    glColor3f (1.0f, 0.0f, 0.0f);

    glPushMatrix();

    //沿 z 轴负向平移一段距离，保证良好的观察效果
    glTranslatef (0.0, 0.0, -5.0);

    //设置裁减平面
    glClipPlane (GL_CLIP_PLANE0, eqn);
    glEnable (GL_CLIP_PLANE0);

    //使球体旋转
    glRotatef(xRot, 1.0f, 0.0f, 0.0f);
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);

    glutWireSphere(2.0f, 20 , 20);

    glPopMatrix();

    glFlush();
```

```
}

void SpecialKeys(int key, int x, int y)
{
    if(key == GLUT_KEY_UP)      xRot-= 5.0f;
    if(key == GLUT_KEY_DOWN)    xRot += 5.0f;
    if(key == GLUT_KEY_LEFT)    yRot -= 5.0f;
    if(key == GLUT_KEY_RIGHT)   yRot += 5.0f;

    if(xRot > 356.0f)   xRot = 0.0f;
    if(xRot < -1.0f)    xRot = 355.0f;
    if(yRot > 356.0f)   yRot = 0.0f;
    if(yRot < -1.0f)    yRot = 355.0f;

    glutPostRedisplay();
}

int main(int argc, char* argv[])
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutCreateWindow("GLClipPlane");
    glutDisplayFunc(RenderScene);
    glutSpecialFunc(SpecialKeys);
    glutReshapeFunc(ChangeSize);

    SetupRC();
    glutMainLoop();
    return 0;
}
```

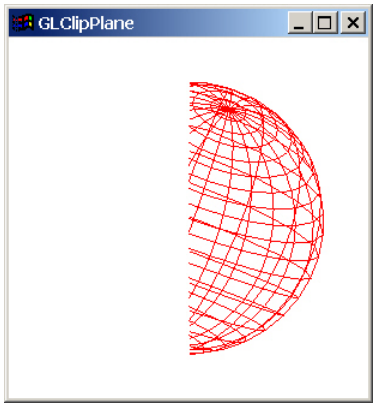

图 3-3 网状球体的裁减

4 OpenGL 中的颜色、光照和材质

4.1 颜色

我们前面已经多次的使用了颜色，而 OpenGL 颜色模式一共有两个：RGB（RGBA）模式和颜色表模式。在 RGB 模式下，所有的颜色定义全用 R、G、B 三个值来表示，有时也加上 Alpha 值（与透明度有关），即 RGBA 模式。而在颜色表模式下，每一个象素的颜色是用颜色表中的某个颜色索引值表示，而这个索引值指向了相应的 R、G、B 值。

在 RGBA 模式下，可以用 glColor*()来定义当前颜色。其函数形式为：

void glColor<x><t> (red, green, blue, alpha);

函数名中的 x 表示参数的数目，当它等于 3 的时候，三个参数分别代表 R、G、B 值，alpha 值缺省为 1.0；当它等于 4 的时候，还包括了 Alpha 值，其范围从 0.0 到 1.0。函数名中的 t 指定参数数据的类型，可以取 b、d、f、i、s、ub、ui 或 us，它们分别代表字节型、双精度型、浮点型、整型、短整型、无符号字节型和无符号短整型。另一个版本的函数还可以增加后缀 v，它用一个数组来放置这些参数。

常用的指定颜色的函数是 glColor3f，其中每个颜色分量的值在[0.0，1.0]范围内。还有一个函数 glColor3ub，这个版本使用的颜色分量的取值范围是 0 到 255 之间的无符号字节，与 Windows 的 RGB 宏指定颜色的方法类似：

glColor3ub(128, 128, 128) = RGB(128, 128, 128)

在颜色表模式（Color_Index Mode）下，可以调用 glIndex*()函数从颜色表中选取当前颜色。其函数形式为：

void glIndex{sifd}(TYPE c);

void glIndex{sifd}v(TYPE *c);

其中，参数值 c 用于设置当前颜色索引值，即调色板号，若值大于颜色位面数时则取模。

4.2 光照模型

当光照射到一个物体表面上时，会出现三种情形。首先，光可以通过物体表面向空间反射，产生反射光；其次，对于透明物体，光可以穿透该物体并从另一端射出，产生透射光；最后，部分光被物体表面吸收而转换成热。在上述三部分光中，仅仅是透射光和反射光能够进入人眼产生视觉效果。此外，物体本身还有可能发光，比如发光的灯泡。这里我们暂时不考虑透明物体，这样场景中可能存在以下几种类型的光，即环境光、散射光、镜面光和辐射光。

1. 环境光(Ambient Light)

环境光有光源，但是由于被周围的环境，如地面、天空、墙壁等多次反射，变得无法确定其方向。环境光均匀地从周围环境入射至景物表面并等量地向各个方向反射出去。一般说来，房间里的环境光成分要多些，户外的相反要少得多，因为大部分光按相同的方向照射，而在户外很少有其他物体反射的光。

2. 漫射光(Diffuse Light)

漫射光来自某个方向，它垂直于物体时比倾斜时更明亮。一旦它照射到物体上，则在各个方向上均匀地发散出去。于是，无论视点在哪里它都一样亮。来自特定位置和特定方向的任何光，都可能有散射成分。

3. 镜面光(Specular Light)

镜面光也具有方向性，但被物体强烈地反射到另一个特定的方向。如一个点光源照射一个金属球时会在球面上形成一块特别亮的区域，呈现所谓“高光(Highlight)”，它是光源在金属球面上产生的镜面反射光。对于较光滑物体，其镜面反射光的高光区域小而亮；相反，粗糙表面的镜面反射光呈发散状态，其高光区域大而不亮。

4. 辐射光

辐射光是最简单的一种光，它直接从物体发出并且不受任何光源影响。

在 OpenGL 中，任何一种光源都由三种光照成分组成：环境光、散射光和镜面光，当然光源本身还有可能发出辐射光。由于我们知道光是一种波，而颜色仅仅是我们可以看见的一种光波，所以每种光照成分都是由 RGBA 值定义的。

4.3 材质属性

当我们使用光照来描述多边形，总是说它由具有某些反射属性的材质组成，而不说它具有特殊的颜色。这样我们指定物体的颜色，就必须指定物体材质对环境光、漫射光和镜面光源的反射属性。通常，我们用材质对光的红、绿、蓝三原色的反射率来近似定义材质属性。象光源一样，材质颜色也分成环境、漫反射和镜面反射成分，它们决定了材质对环境光、漫反射光和镜面反射光的反射程度。

在进行光照计算时，物体的最终颜色是由其材质属性的 RGB 值 和光照属性的 RGB 值共同决定的。例如，如果当前的环境光源的 RGB 值为（0.5，1.0，0.5），而物体的材质的环境反射成分的 RGB 值为（0.5，0.5，0.5），那么物体最终的颜色为：

（0.5×0.5，1.0×0.5，0.5×0.5）=（0.25，0.5，0.25）

即将每个环境光源的成分与材质的环境反射率相乘。这样，物体表面的颜色为三项 RGB 值的叠加：材质对环境光的反射率与环境光结合的 RGB 值；材质对漫反射光的反射率与漫反射光结合的 RGB 值；材质对镜面光的反射率与镜面反射光结合的 RGB 值。注意，当叠加的 RGB 中任何一个颜色分量的值大于 1.0，那么就用 1.0 计算。

4.4 使用光照

在场景中增加光线需要几个步骤，下面的程序演示了这个过程，程序的输出如图 4.1 和 4.2 所示。

程序 4.1：光照环境的三棱锥

```
#include <windows.h>
#include <gl/gl.h>
#include <gl/glut.h>

// 旋转参数
static GLfloat xRot = 0.0f;
static GLfloat yRot = 0.0f;

BOOL bColorMaterial = FALSE;//颜色跟踪模式

//初始化窗口
void SetupRC(void)
{
    // 设置窗口背景颜色为黑色
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f );
    glShadeModel(GL_FLAT);
}

void ChangeSize(int w, int h)
{
```

```
    if(h == 0)h = 1;

    glViewport(0, 0, w, h);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    if (w <= h)
        glOrtho (-100.0f, 100.0f, -100.0f*h/w, 100.0f*h/w, -100.0f, 100.0f);
    else
        glOrtho (-100.0f*w/h, 100.0f*w/h, -100.0f, 100.0f, -100.0f, 100.0f);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

// 在窗口中绘制图形
void RenderScene(void)
{
    //清除颜色及深度缓冲区
    glClear(GL_COLOR_BUFFER_BIT| GL_DEPTH_BUFFER_BIT);
    glEnable(GL_DEPTH_TEST);

    //设置光照
    glEnable(GL_LIGHTING); //启动光照
    GLfloat ambient[]={0.8f, 0.8f, 0.8f, 1.0f}; //环境光 RGBA 值
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambient); //设置全局环境光

    //设置材质属性
    if(bColorMaterial)
    {
        glEnable(GL_COLOR_MATERIAL); //启动颜色跟踪法
        //指定多边形的正面使用颜色跟踪法
        glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
    }
    else
    {
        glDisable(GL_COLOR_MATERIAL); //关闭颜色跟踪法
        GLfloat material_ambiemt[]={0.75f, 0.75f, 0.75f, 1.0f};
        //指定多边形的正面的环境反射和漫反射值
        glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, material_ambiemt);
    }

    // 旋转图形
```

```
    glPushMatrix();
    glRotatef(xRot, 1.0f, 0.0f, 0.0f);
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);

    glFrontFace(GL_CW);

    //绘制三棱锥
    glBegin(GL_TRIANGLE_FAN);
        glVertex3f(0.0, 0.0, 80.0);
        glVertex3f(0.0, 50.0, 0.0);
        glColor3f(1.0,0.0,0.0);
        glVertex3f(50.0, -50.0, 0.0);
        glColor3f(0.0,1.0,0.0);
        glVertex3f(-50.0, -50.0, 0.0);
        glColor3f(0.0,0.0,1.0);
        glVertex3f(0.0, 50.0, 0.0);
    glEnd();

    glBegin(GL_TRIANGLE_FAN);
        glVertex3f(0.0, 50.0, 0.0);
        glVertex3f(50.0, -50.0, 0.0);
        glColor3f(1.0,1.0,0.0);
        glVertex3f(-50.0, -50.0, 0.0);
    glEnd();

    glPopMatrix();

    // 刷新命令缓冲区
    glutSwapBuffers();
}

void SpecialKeys(int key, int x, int y)
{
    if(key == GLUT_KEY_UP)      xRot-= 5.0f;
    if(key == GLUT_KEY_DOWN)    xRot += 5.0f;
    if(key == GLUT_KEY_LEFT)    yRot -= 5.0f;
    if(key == GLUT_KEY_RIGHT)   yRot += 5.0f;

    if(xRot > 356.0f)    xRot = 0.0f;
    if(xRot < -1.0f)     xRot = 355.0f;
    if(yRot > 356.0f)    yRot = 0.0f;
    if(yRot < -1.0f)     yRot = 355.0f;

    // 刷新窗口
```

```
    glutPostRedisplay();
}

void ProcessMenu(int value)
{
    switch(value)
        {
        case 1:
            bColorMaterial = FALSE;
            break;
        case 2:
            bColorMaterial = TRUE;
            break;
        default:
            break;
    }
    glutPostRedisplay();
}

int main(int argc, char* argv[])
{

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB );
    glutCreateWindow("加光照的三棱锥");
    glutReshapeFunc(ChangeSize);
    glutSpecialFunc(SpecialKeys);
    glutDisplayFunc(RenderScene);

    // 创建右键菜单
    glutCreateMenu(ProcessMenu);
    glutAddMenuEntry("普通材质",1);
    glutAddMenuEntry("颜色跟踪材质",2);
    glutAttachMenu(GLUT_RIGHT_BUTTON);

    SetupRC();
    glutMainLoop();
    return 0;
}
```

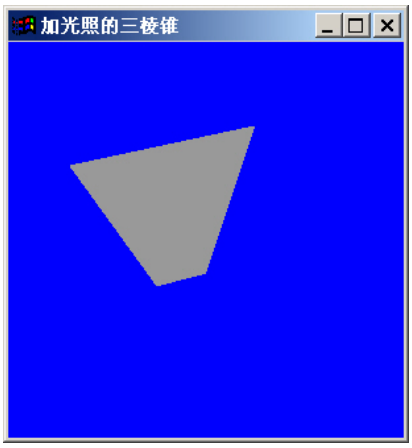

图 4-1 glMaterial 函数定义材质

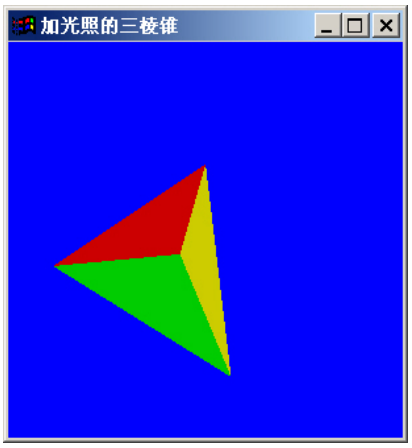

图 4-2 颜色跟踪法设定材质

1. 启动光照

让 OpenGL 进行光照计算，必须明确指出光照是否有效或无效。如果光照无效，OpenGL 只是简单地将当前颜色映射到当前顶点上去，而不进行法向、光源、材质等复杂计算。要使光照有效，首先得启动光照，即：

```
glEnable(GL_LIGHTING);
```

若使光照无效，则调用 gDisable(GL_LIGHTING) 可关闭当前光照。

2. 设置光照模型

允许进行光照计算后，需要设置光照模型，影响光照模型的三个参数是通过 glLightModel 函数中设置的。在上面的示例中，我们使用了参数 GL_LIGHT_MODEL_AMBIENT，这个参数允许指定当前场景的全局环境光，它可以从各边均匀照射所有的物体。

```
//设置光照
glEnable(GL_LIGHTING);
GLfloat ambient[]={0.8f, 0.8f, 0.8f, 1.0f};
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambient);
```

此时第二个参数为一个储存 RGBA 值的数组。注意，默认的全局环境光的 RGBA 值为（0.2，0.2，0.2，1.0），光线相当黯淡的。

3. 设置材质属性

在设置了光照模型之后，我们还需要设置绘制物体的材质属性，使多边形能够反射光，这样就可以看到三棱锥了。设置材质属性的方法有两种，一种是在绘制每个多边形或者多边形集和之前利用 glMaterial 函数。下面的代码指定了后面绘制的多边形都具有相同的材质属性：

```
GLfloat material_ambiemt[]={0.75f, 0.75f, 0.75f, 1.0f};
glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, material_ambiemt);
```

glMaterialfv 函数的第一个参数指定多边形的前面（GL_FRONT），背面（GL_BACK）或者两面（GL_FRONT_AND_BACK）具有指定的材质属性。第二个参数则指定当前设置的是哪一种材质属性，这里我们使用 GL_AMBIENT_AND_DIFFUSE，即设置环境光和漫反射光的值。最后一个参数是一个数组，它包含了构成属性的 RGBA 值。

注意，在大多数情况下，环境光和漫射光的成分是相同的，而一般不需要指定镜面反射指数，除非需要镜面亮斑。

第二种更常用的方法是使用颜色跟踪法。通过颜色跟踪法，只需要通过调用 glColor 就可以设置材质属性。要使颜色跟踪法有效，必须调用带有参数 GL_COLOR_MATERIAL 的 glEnable 函数。然后用 glColorMaterial 函数指定多边形的哪一个面使用颜色跟踪法。代码如

下：

```
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
```

4.5 使用光源

4.5.1 光源参数设置

在程序 4.1 中，我们仅仅使用了全局环境光，改变全局环境光的 RGBA 值，可以使场景的明暗发生变化。此外，OpenGL 还提供了 8 个独立的光源，他们可以放在场景中的任何地方，也可放在视见空间之外。当将光源放在无穷远处就可以得到平行的光线。

每个光源包含了环境光、漫射光和镜面光三个成分，这些属性都是通过函数 glLight 来指定的，我们看下面一段代码：

```
GLfloat light0_diffuse[]= { 0.0f, 0.0f, 1.0f, 1.0f};
glLightfv(GL_LIGHT0, GL_DIFFUSE, light0_diffuse);
```

函数 glLightfv 函数的第一个参数是一个符号常数，指定设置的是哪个光源的参数。第二个参数指定了设置的是哪一个参数，可以选择 GL_DIFFUSE、GL_AMBIENT、GL_SPECULAR，分别用于指定漫射光、环境光和镜面光成分。第三个参数是一个包含四个浮点数的数组，用于指定相应的参数。这段代码指定的光源 LIGHT0 只包含了漫射光成分，是一个漫反射光源。

指定了光源成分之后，需要指定光源的位置，如下面一段代码：

```
GLfloat light0_position[] = { 1.0f, 1.0f, 1.0f, 0.0f };
glLightfv(GL_LIGHT0, GL_POSITION, light0_position);
```

这段代码指定了光源 LIGHT0 所在的位置，指定光源位置参数的数组 light0_position 包含四个值，其中最后一个值为 1.0 时，则表明指定的坐标就是光源的位置。如果最后一个值是 0.0，则表明沿数组指定的矢量方向的光源位于无限远的地方，即光源发出的光是平行光。OpenGL 光源的位置和方向与其它几何图元的位置和方向一样都必须经过变换矩阵的作用。尤其是当用 glLight 函数说明光源的位置和方向时，位置和方向都要经过当前变换矩阵的作用并保存在视点坐标系中，但是投影矩阵变换对它们不起作用。

在定义了光源的属性参数之后，必须使用 glEnable 函数使光源有效。下面的程序演示了一个移动的光源，其输出如图 4.3 所示。

程序 4.2：移动光源

```
#include <windows.h>
#include <gl/gl.h>
#include <gl/glu.h>
#include <gl/glut.h>

static GLfloat xRot = 0.0f;
static GLfloat yRot = 0.0f;

void SetupRC()
{
    //设置光源 LIGHT0 的参数
    GLfloat light_diffuse[]={1.0f,1.0f,1.0f,1.0f};
    GLfloat light_ambient[]={0.0f,0.5f,0.5f,1.0f};
    glLightfv(GL_LIGHT0,GL_DIFFUSE,light_diffuse);
    glLightfv(GL_LIGHT0,GL_AMBIENT,light_ambient);
```

```
    //使光源有效
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    //开启深度测试
    glEnable(GL_DEPTH_TEST);

    glClearColor(0.0f, 0.0f, 0.0f,1.0f);
}

void ChangeSize(int w, int h)
{
    GLfloat nRange = 80.0f;
    if(h == 0)        h = 1;

    glViewport(0, 0, w, h);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    gluPerspective(40.0f, (GLfloat) w/(GLfloat) h, 1.0f, 20.0f);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void RenderScene(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    GLfloat position[] = { 0.0f, 0.0f, 1.5f, 1.0f };

    glPushMatrix ();
    glTranslatef (0.0f, 0.0f, -5.0f);
    glPushMatrix ();
    glRotated (yRot, 0.0f, 1.0f, 0.0f);
    glRotated (xRot, 1.0f, 0.0f, 0.0f);

    //设置光源的位置
    glLightfv (GL_LIGHT0, GL_POSITION, position);

    glTranslated (0.0f, 0.0f, 1.5f);
    //绘制一个黄色的光球
```

```
        glDisable (GL_LIGHTING);
        glColor3f (1.0f, 1.0f, 0.0f);
        glutSolidSphere (0.1f, 50.0f, 50.0f);
        glEnable (GL_LIGHTING);

        glPopMatrix ();
        //设置材质属性
        GLfloat mat_diffuse[]={0.0,0.5,1.0,1.0};
        glMaterialfv(GL_FRONT_AND_BACK,GL_DIFFUSE,mat_diffuse);

        glutSolidTorus (0.275, 0.85, 50, 50);
        glPopMatrix ();
        glutSwapBuffers();
}

void SpecialKeys(int key, int x, int y)
{
        if(key == GLUT_KEY_UP)          xRot-= 5.0f;
        if(key == GLUT_KEY_DOWN)    xRot += 5.0f;
        if(key == GLUT_KEY_LEFT)      yRot -= 5.0f;
        if(key == GLUT_KEY_RIGHT)   yRot += 5.0f;
        if(key > 356.0f)        xRot = 0.0f;
        if(key < -1.0f)         xRot = 355.0f;
        if(key > 356.0f)        yRot = 0.0f;
        if(key < -1.0f)         yRot = 355.0f;
        glutPostRedisplay();
}

int main(int argc, char* argv[])
{
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
        glutCreateWindow("移动的光源");
        glutReshapeFunc(ChangeSize);
        glutSpecialFunc(SpecialKeys);
        glutDisplayFunc(RenderScene);
        SetupRC();
        glutMainLoop();
        return 0;
}
```

图 4-3 移动的光源

4.5.2 聚光源

定位光源可以定义成聚光灯形式，即将光的形状限制在一个圆锥内。OpenGL 中聚光的定义包括以下几步：

1. 定义聚光源位置。

因为聚光源也是定向光源，所以他的位置同一般定向光一样。

2. 定义聚光截止角。

利用函数 glLightf 使用参数 GL_SPOT_CUTOFF 定义聚光光锥的轴与中心线的夹角，也可说成是光锥顶角的一半。缺省时，这个参数为 180.0，即顶角为 360 度，光向所有的方向发射，因此聚光关闭。一般在聚光启动情况下，聚光截止角限制在[0.0,90.0]之间，如下面一行代码设置截止角为 45 度：

```
glLightf(GL_LIGHT0,GL_SPOT_CUTOFF,45.0);
```

3. 定义聚光方向。

聚光方向决定聚光光锥的轴，它使用齐次坐标定义，缺省值为(0.0,0.0,-1.0)，即指向 Z 负轴。聚光方向也要进行几何变换，其结果保存在视点坐标系中。聚光方向是通过函数 glLightfv 使用参数 GL_SPOT_DIRECTION 定义，如下面一段代码：

```
GLfloat spot_direction[]={-1.0f, -1.0f, 0.0f};
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spot_direction);
```

4. 定义聚光指数。参数 GL_SPOT_EXPONENT 控制光的集中程度，光锥中心的光强最大，越靠边的光强越小，缺省时为 0。如下面的代码：

```
glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, 2.0);
```

此外，除了定义聚光指数控制光锥内光强的分布，还可利用衰减因子来设置光沿着传播方向的衰减。OpenGL 的光衰减是通过光源的发光量乘以衰减因子计算出来的。常用的衰减因子有常值因子（GL_CONSTANT_ATTENUATION）、线形因子（GL_LINEAR_ATTENUATION）和二次因子（GL_QUADRATIC_ATTENUATION），其中缺省的常数衰减因子是 1.0，其余两个因子都是 0.0。用户也可以自己定义这些值，如：

```
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 2.0);
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.5);
```

4.5.3 镜面光斑

在定义了光源中的镜面光成分之后，物体上并不会产生镜面光斑的效果，还需要定义材质的镜面反射率和镜面指数。

1. 镜面反射率

镜面反射率决定了材质表面对于镜面光反射的成分，如下面的代码：

```
GLfloat mat_specular[] = { 1.0f, 1.0f, 1.0f, 1.0f };
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
```

指定了其后绘制的表面几乎反射了所有的入射镜面光。

2. 镜面指数

高度的镜面光和材质的镜面反射率使物体的颜色变亮，我们还可以使用下面的代码指定材质的镜面指数。

```
glMaterialfv(GL_FRONT, GL_SHININESS, 128);
```

镜面指数说明如何确定镜面光亮斑的大小和聚光程度。值为 0 时指定一个没有焦点镜面光亮斑，它能够对整个多边形的颜色均匀加亮。如果增大这个值，可以减少镜面光亮斑的尺寸、增加聚焦度，就出现了闪亮的光斑，该参数值的范围在[1，128]之间。

在下面的例子中，设置了两个光源，一个是漫反射的蓝色点光源，另一个是红色聚光光源，它们都照在一个球体上，产生亮斑，其输出如图 4.4 所示。

程序 4.3 多光源

```
#include <windows.h>
#include <gl/glut.h>
#include<gl/gl.h>
#include<gl/glu.h>

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

    //定义材质属性
    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);

    //light0 为漫反射的蓝色点光源
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light0_diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION,light0_position);

    //light1 为红色聚光光源
    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);
    glEnable(GL_DEPTH_TEST);

    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
}

void ChangeSize(GLsizei w, GLsizei h)
{
    if(h == 0)      h = 1;

    glViewport(0, 0, w, h);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    if (w <= h)
        glOrtho (-5.5f, 5.5f, -5.5f*h/w, 5.5f*h/w, -10.0f, 10.0f);
    else
        glOrtho (-5.5f*w/h, 5.5f*w/h, -5.5f, 5.5f, -10.0f, 10.0f);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void RenderScene(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    glTranslated (-3.0f, -3.0f, 3.0f);
    glPopMatrix ();
    glutSolidSphere(2.0f, 50, 50);

    glFlush();
}

void main(void)
{
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutCreateWindow("多光源球");
```

```
    glutDisplayFunc(RenderScene);
    glutReshapeFunc(ChangeSize);

    SetupRC();
    glutMainLoop();
}
```

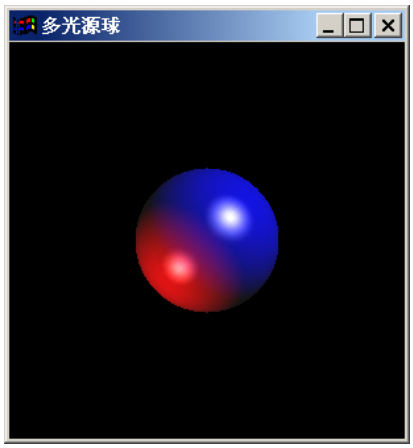

图 4-4 多光源球

4.5.4 辐射光

辐射光可以使物体看起来发出某种颜色的光，通过给 GL_EMISSION 定义一个 RGBA 值，可以达到这种特殊效果。我们可以利用这一特性来模拟灯和其他光源。代码如下：

```
GLfloat mat_emission[]={0.3f, 0.3f, 0.5f, 0.0f};
glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
```

这样，物体看起来稍微有点发光。比如绘制一个打开的台灯，就可以将一个小球的材质定义成上述形式，并且在小球内部建立一个聚光源，这样台灯的灯泡效果就出来了

4.5.5 双面光照

光照计算通常是对所有多边形进行的，无论其是正面或反面。一般情况下，设置光照条件时总是正面多边形，因此不能对背面多边形进行正确地光照。对于一个封闭的物体，只有正面多边形能看到，这种情况下不必考虑背面光照。而如果物体不封闭，其内部的曲面是可见的，那么对内部多边形需进行光照计算，这时应该调用如下函数：

```
glLightModeli(LIGHT_MODEL_TWO_SIDE, GL_TRUE);
```

启动双面光照。要关闭双面光照，只需将参数 GL_TRUE 改为 GL_FALSE 即可。下面的例子中，演示了双面光照的效果，其输出如图 4.5 所示。

程序 4.4：双面光照效果

```
#include <windows.h>
#include <gl/gl.h>
#include <gl/glu.h>
#include <gl/glut.h>

void SetupRC()
{
    GLfloat ambient[] = { 0.0f, 0.0f, 0.0f, 1.0f };
    GLfloat diffuse[] = { 1.0f, 1.0f, 1.0f, 1.0f };
    GLfloat specular[] = { 1.0f, 1.0f, 1.0f, 1.0f };
    GLfloat position[] = { -5.0f, 5.0f, 5.0f, 1.0f };

    //设定光源参数
```

```
    glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, specular);
    glLightfv(GL_LIGHT0, GL_POSITION, position);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_DEPTH_TEST);
    glFrontFace(GL_CW);

    glClearColor(0.0, 0.1, 0.1, 0.0);
}

void ChangeSize(int w, int h)
{
    if(h == 0)      h = 1;

    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    if (w <= h)
        glOrtho (-4.0f, 4.0f, -4.0f*h/w, 4.0f*h/w, -10.0f, 10.0f);
    else
        glOrtho (-4.0f*w/h, 4.0f*w/h, -4.0f, 4.0f, -10.0f, 10.0f);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void RenderScene(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPushMatrix();

    /* 第一个茶壶使用了 GL_FRONT 材质和单面光照*/
    glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_FALSE);
    GLdouble equ[4] = { -1.0f, 2.3f, 2.3f, 2.3f };
    glPushMatrix();
    glTranslatef(-3.0f, 0.0f, 0.0f);
    glRotatef(180.0f, 0.0f, 1.0f, 0.0f );

    //glClipPlane 定义裁减平面，使得可以看到茶壶的内部
    glClipPlane(GL_CLIP_PLANE0, equ);
    glEnable(GL_CLIP_PLANE0);

    GLfloat tea_ambient[] = { 0.0f, 0.2f, 1.0f, 1.0f };
    GLfloat tea_diffuse[] = { 0.8f, 0.5f, 0.2f, 1.0f };
    GLfloat tea_specular[] = { 1.0f, 1.0f, 1.0f, 1.0f };

    glMaterialfv(GL_FRONT, GL_AMBIENT, tea_ambient);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, tea_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, tea_specular);
    glMaterialf(GL_FRONT, GL_SHININESS, 20.0);
    glutSolidTeapot(1.0);
    glPopMatrix();
```

```
    //开启双面光照
    glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);

    /* 第二个茶壶的正面和反面使用相同的材质 */
    glPushMatrix();
    glRotatef(180.0f, 0.0f, 1.0f, 0.0f );
    glClipPlane(GL_CLIP_PLANE0, equ);
    glEnable(GL_CLIP_PLANE0);

    GLfloat teafb_ambient[] = { 1.0f, 0.2f, 0.0f, 1.0f };
    GLfloat teafb_diffuse[] = { 0.9f, 0.5f, 0.8f, 1.0f };
    GLfloat teafb_specular[] = { 1.0f, 1.0f, 1.0f, 1.0f };

    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, teafb_ambient);
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, teafb_diffuse);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, teafb_specular);
    glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 55.0f);
    glutSolidTeapot(1.0);
    glPopMatrix();

    /* 第三个茶壶的正面和反面使用了不同的材质 */
    glPushMatrix();
    glTranslatef(3.0f, 0.0f, 0.0f);
    glRotatef(180.0f, 0.0f, 1.0f, 0.0f );
    glClipPlane(GL_CLIP_PLANE0, equ);
    glEnable(GL_CLIP_PLANE0);

    GLfloat teaf_ambient[] = { 0.6f, 0.4f, 0.4f, 1.0f };
    GLfloat teaf_diffuse[] = { 1.0f, 0.4f, 0.1f, 1.0f };
    GLfloat teaf_specular[] = { 1.0f, 1.0f, 1.0f, 1.0f };

    glMaterialfv(GL_FRONT, GL_AMBIENT, teaf_ambient);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, teaf_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, teaf_specular);
    glMaterialf(GL_FRONT, GL_SHININESS, 100.0f);

    GLfloat teab_ambient[] = { 0.4f, 0.1f, 1.0f, 1.0f };
    GLfloat teab_diffuse[] = { 0.2f, 0.8f, 0.1f, 1.0f };
    GLfloat teab_specular[] = { 1.0f, 1.0f, 1.0f, 1.0f };

    glMaterialfv(GL_BACK, GL_AMBIENT, teab_ambient);
    glMaterialfv(GL_BACK, GL_DIFFUSE, teab_diffuse);
    glMaterialfv(GL_BACK, GL_SPECULAR, teab_specular);
    glMaterialf(GL_BACK, GL_SHININESS, 8.0f);

    glutSolidTeapot(1.0);
    glPopMatrix();

    glDisable(GL_CLIP_PLANE0);
    glPopMatrix();
    glutSwapBuffers();
}

int main(int argc, char* argv[])
{
    glutInit(&argc, argv);
```

```
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB );
    glutInitWindowSize(400,400);
    glutInitWindowPosition(200,200);
    glutCreateWindow("不同材质属性的茶壶");
    glutReshapeFunc(ChangeSize);
    glutDisplayFunc(RenderScene);
    SetupRC();
    glutMainLoop();

    return 0;
}
```

图 4-5 不同材质属性的茶壶

这里要说明的一点是，由于显示回调函数在程序运行过程中会被多次调用，故此对于光照和材质设定，如果整个场景使用同一属性，那么尽可能的将这些设置放在初始化函数中，以提高程序的运行效率。

附录 参考函数

1.1 颜色使用

1. glShadeModel：选择平面明暗模式或光滑明暗模式

C 语言描述

void glShadeModel(GLenum mode)

参数

mode 指定表示明暗模式的符号值，可以选择 GL_FLAT（平面明暗模式）和 GL_SMOOTH（光滑明暗模式），缺省值为 GL_SMOOTH。

说明

OpenGL 图元需要进行明暗处理，处理得模式可以为平面明暗模式或光滑（Gouraud 着色）明暗模式。光滑明暗模式时，多边形各个内部点的颜色是根据各顶点指定的颜色来插值得到的，这意味着两个顶点之间的颜色是从一顶点的颜色渐变到另一顶点的颜色。对于平面明暗模式，整个图元区域的颜色就是最后一个顶点指定的颜色，唯一例外的是 GL_POLYGON，这是整个区域的颜色是第一个顶点所指定的颜色。但要注意，如果激活了光照，计算到的顶点颜色都是光照后的结果颜色，若光照关闭，计算到的颜色就是指定顶点时的当前颜色。

2. glColor：设置当前颜色

C 语言描述

void glcolor3b(GLbyte red，GLbyte green，GLbyte blue);
void glcolor3d(GLdouble red，GLdouble green，GLdouble blue);
void glcolor3f(GLfloat red，GLfloat green，GLfloat blue);
void glcolor3i(GLint red，GLint green，GLint blue);
void glcolor3s(GLshort red，GLshort green，GLshort blue);
void glcolor3ub(GLubyte red，GLubyte green，GLubyte blue);
void glcolor3ui(GLuint red，GLuint green，GLuint blue);
void glcolor3us(GLushort red，GLushort green，GLushort blue);
void glcolor4b(GLbyte red，GLbyte green，GLbyte blue，GLbyte alpha);
void glcolor4d(GLdouble red，GLdouble green，GLdouble blue，GLdouble alpha);
void glcolor4f(GLfloat red，GLfloat green，GLfloat blue，GLfloat alpha);
void glcolor4i(GLint red，GLint green，GLint blue，GLint alpha);
void glcolor4s(GLshort red，GLshort green，GLshort blue，GLshort alpha);
void glcolor4ub(GLubyte red，GLubyte green，GLubyte blue，GLubyte alpha);
void glcolor4ui(GLuint red，GLuint green，GLuint blue，GLuint alpha);
void glcolor4us(GLushort red，GLushort green，GLushort blue，GLushort alpha);
void glcolor3bv(const GLbyte *v);
void glcolor3dv(const GLdouble *v);
void glcolor3fv(const GLfloat *v);
void glcolor3iv(const GLint *v);
void glcolor3sv(const GLshort *v);
void glcolor3ubv(const GLubyte *v);
void glcolor3uiv(const GLuint *v);

void glcolor3usv(const GLushort *v);
void glcolor4bv(const GLbyte *v);
void glcolor4dv(const GLdouble *v);
void glcolor4fv(const GLfloat *v);
void glcolor4iv(const GLint *v);
void glcolor4sv(const GLshort *v);
void glcolor4ubv(const GLubyte *v);
void glcolor4uiv(const GLuint *v);
void glcolor4usv(const GLushort *v);

参数

red，green, blue　指定当前颜色中的红、绿和蓝色成分。

alpha　指定颜色中的 α 成分。只有在 glColor4 函数带 4 个变量时才指定此参数。

*v　指定一个指向包含红、绿、蓝和 alpha 值的数组指针。

说明

本函数通过指定红、绿、蓝的颜色成分来设置当前的颜色，同时部分函数可以接受 α 成分。每个成分亮度的表示范围是从零（0.0）到全光强（1.0），当指定了非浮点类型时，该类型从 0 到最大值的表示法与浮点类型范围的 0.0 到 1.0 相映射。

3. glColorMask：激活或关闭帧缓存颜色分量的写操作

C 语言描述

void glColorMask(GLboolean red，GLboolean green，GLboolean blue，GLboolean alpha)

参数

red，green，blue，alpha　指定红、绿、蓝和 α 成分是否可以修改。

说明

本函数指定是否可以将单个的颜色分量写入帧缓存。例如，若 red 为 GL_FALSE，那么不管执行什么样的绘制操作，任何颜色缓存中任何像素的红色分量均不被改变。

1.2 绘制几何图元

1. glVertex：指定顶点

C 语言描述

glVertex2d(GLdouble x, GLdouble y);
glVertex2f(GLfloat x, GLfloat y);
glVertex2i(GLint x, GLint y);
glVertex2s(GLshort x, GLshort y);
glVertex3d(GLdouble x, GLdouble y, GLdouble z);
glVertex3f(GLfloat x, GLfloat y, GLfloat z);
glVertex3i(GLint x, GLint y, GLint z);
glVertex3s(GLshort x, GLshort y, GLshort z);
glVertex4d(GLdouble x, GLdouble y, GLdouble z, GLdouble w);
glVertex4f(GLfloat x, GLfloat y, GLfloat z, GLfloat w);
glVertex4i(GLint x, GLint y, GLint z, GLint w);
glVertex4s(GLshort x, GLshort y, GLshort z, GLshort w);
glVertex2dv(const GLdouble *v);
glVertex2fv(const GLfloat *v);

glVertex2iv(const GLint *v);
glVertex2sv(const GLshort *v);
glVertex3dv(const GLdouble *v);
glVertex3fv(const GLfloat *v);
glVertex3iv(const GLint *v);
glVertex3sv(const GLshort *v);
glVertex4dv(const GLdouble *v);
glVertex4fv(const GLfloat *v);
glVertex4iv(const GLint *v);
glVertex4sv(const GLshort *v);

参数

x，y，z　顶点的 x，y，z 坐标，不指定 z 时，默认值为 0.0。

w　顶点的 w 坐标，该坐标用于缩放，默认情况下设置为 1.0。

*v　一个数值数组，它保存着 2 个，3 个或 4 个用于指定顶点的值。

说明

glVertex 函数使用在 glBegin/glEnd 语句之间，用来指定点、线和多边形的顶点。当调用 glVertex 函数指定顶点时，可以为该顶点指定相关的数据，即顶点的当前颜色、法向量和纹理坐标。

2. glBegin，glEnd：限定一个或多个图元顶点的绘制

C 语言描述

void glBegin(GLenum mode);
void glEnd(void);

参数

mode　指定在 glBegin 和 glEnd 之间将要用顶点创建的一个或多个图元，可以有以下 10 个供选择的常数：GL_POINTS、GL_LINES、GL_LINE_STRIP、GL_LINE_LOOP、GL_TRIANGLES 、 GL_TRIANGLE_STRIP 、 GL_TRIANGLE_FAN 、 GL_QUADS 、 GL_QUAD_STRIP、GL_POLYGON。

说明

glBegin 和 glEnd 函数限定了一组或多组图元的顶点定义。glBegin 函数的变量 mode 指定了顶点可以绘制的 10 种方式。以 n 作为以 1 开始的整数，N 作为指定的全部顶点数，则各种绘制方式如下：

GL_POINTS：把每一个顶点作为一个点进行处理，顶点 n 即定义了点 n，共绘制 N 个点。

GL_LINES：把每一个顶点作为一个独立的线段，顶点 2n-1 和 2n 之间共定义了 n 个线段，总共绘制 N/2 条线段。，如果 N 为奇数，则忽略最后一个顶点。

GL_LINE_STRIP：绘制从第一个顶点到最后一个顶点依次相连的一组线段，第 n 和 n+1 个顶点定义了线段 n，总共绘制 N-1 条线段。

GL_LINE_LOOP：绘制从定义第一个顶点到最后一个顶点依次相连的一组线段，然后最后一个顶点与第一个顶点相连。第 n 和 n+1 个顶点定义了线段 n，然后最后一个线段是由顶点 N 和 1 之间定义，总共绘制 N 条线段。

GL_TRIANGLES：把每三个顶点作为一个独立的三角形。顶点 3n-2，3n-1 和 3n 定义了第 n 个三角形，总共绘制 N/3 个三角形。

GL_TRIANGLE_STRIP：绘制一组相连的三角形。对于奇数点 n，顶点 n，n+1 和 n+2

定义了第 n 个三角形；对于偶数 n，顶点 n+1，n 和 n+2 定义了第 n 个三角形，总共绘制 N-2 个三角形。

GL_TRIANGLE_FAN：绘制一组相连的三角形。三角形是由第一个顶点及其后给定的顶点所确定。顶点 1，n+1 和 n+2 定义了第 n 个三角形，总共绘制 N-2 个三角形。

GL_QUADS：绘制由 4 个顶点组成的一组单独的四边形。顶点 4n-3，4n-2，4n-1 和 4n 定义了第 n 个四边形，总共绘制 N/4 个四边形。

GL_QUAD_STRIP：绘制一组相连的四边形。每个四边形是由一对顶点及其后给定的一对顶点共同确定的。顶点 2n-1，2n，2n+2 和 2n+1 定义了第 n 个四边形，总共绘制 N/2-1 个四边形。

GL_POLYGON：绘制一个凸多边形，顶点 1 到 N 定义了这个多边形。

可以在 glBegin 和 glEnd 之间调用的 OpenGL 函数为：glVertex，glColor，glIndex，glNormal，glTexCoord，glEvalCoord，glEvalPoint，glMaterial 和 glEdgeFlag。

3. glEdgeFlag，glEdgeFlagv：指定边界标记

C 语言描述

void glEdgeFlag(GLboolean flag);

void glEdgeFlagv(const GLboolean *flag);

参数

flag　指定当前的边界标记值，为 GL_TRUE 或 GL_FALSE。

*flag　指向包含单个布尔元素的数组指针，该布尔元素表示当前的边界标记值。

说明

在 glBegin/glEnd 之间指定的多边形、单个的三角形或单个的四边形上每个顶点的边界标志，该顶点或者是边界标志或者不是边界标志。指定顶点时，如果当前边界标记为 GL_TRUE，那么该顶点标记为边界的起点，否则为非边界标记的起点。注意只有在 GL_POLYGON_MODE 设置为 GL_POINT 或 GL_LINE 时，边界标记和非边界标记才有意义。初始时，边界标记为 GL_TREU。

4. glPointSize：指定光栅化点的直径

C 语言描述

void glPointSize(GLfloat size);

参数

size　指定光栅化点的直径，缺省值为 1.0。

说明

本函数指定走样和反走样的点的光栅化的直径。如果 size 变量取非 1.0 的值，那么激活和未激活反走样的情况下，点的大小在屏幕上显示不同的结果。可以调用 glEnable 和 glDisable 函数，变量为 GL_POINT_SMOOTH 控制点的反走样。

如果关闭反走样，点的实际大小是通过将指定的数值四舍五入为整数来决定。如果激活反走样，那么点光栅化为每个像素正方形产生一个片元，该正方形的直径等于当前点的大小，中心位于点（xw，yw）（下标 w 表示窗口坐标）的圆形区域相交，对于每个片元，覆盖值是圆形区域与对应的像素正方形相交的窗口坐标区域，这个覆盖值将被保存起来并使用在最后的光栅化步骤中。

注意，当激活点反走样时，点的大小并不是任意的，它有一定的范围，可以用 glGet 函数，变量为 GL_POINT_SIZE_RANGE 查询出，如果设定的值超出这个范围，那么将会使用该范围内最接近设定值的值。

5. glLineWidth：指定光栅化直线的宽度

C 语言描述

void glLineWidth(GLfloat width);

参数

width 指定光栅化直线的宽度，缺省值为 1.0。

说明

本函数指定被光栅化的走样和反走样直线的宽度。在激活和未激活反走样的情况下，不同的线宽具有不同的效果。可以调用 glEnable 和 glDisable 函数，变量为 GL_LINE_SMOOTH 控制直线的反走样。

如果关闭直线反走样，实际的线宽是通过将指定的数值四舍五入为整数来决定。如果激活反走样，直线光栅化操作为每个像素正方形产生一个片元，这个正方形与位于矩形内的区域相互交叉，该矩形的宽度等于当前的线宽，长度等于线的实际长度，并且位于线段的中心。对于每个片元，覆盖值是相邻像素正方形的矩形区域交叉的窗口坐标区域，这个覆盖值将被保存起来并使用在最后的光栅化操作中。

注意，当激活直线反走样时，直线的宽度并不是任意的，它有一定的范围，可以用 glGet 函数，变量为 GL_LINE_WIDTH_RANGE 查询出，如果设定的值超出这个范围，那么将会使用该范围内最接近设定值的值。

6. glLineStipple：指定点划线

C 语言描述

void glLineStipple(GLint factor, GLushort pattern)

参数

factor 指定线的点划图中每个位的倍数。例如，如果 factor 为 3，那么点划图中的每个位是设置值的 3 倍。factor 的取值范围为[0，255]，缺省值为 0。

pattern 指定一个 16 位整数，当线被光栅化时，这 16 位确定了线中的哪一段需要绘制，缺省时点划图中的每个位均为 1。

说明

本函数使用位模式来绘制点划线。位模式从第 0 位（最右边的位）开始，所以实际的绘图模式是指定模式的逆序。factor 参数用于展宽模式中每一位在点划线中指定要画或不要画的像素数。为了使用点划线，必须先启用点划线，调用 glEnable（GL_LINE_STIPPLE）。如果正在绘制多条线段，每次画新线段时模式将被复位，这样在绘制一条线段在模式的中间终止将不会影响下一条线段。

7. glPolygonMode：选择多边形光栅化模式

C 语言描述

void glPolygonMode(GLenum face, GLenum mode);

参数

face 指定多边形的哪一个面受模式改变的影响——GL_FRONT，GL_BACK 或 GL_FRONT_AND_BACK。

mode 指定新的绘图模式。GL_FILL 为默认值，生成填充的多边形；GL_LINE 生成多边形的轮廓；GL_POINT 只画出顶点。GL_LINE 和 GL_POINT 绘制的点和直线受 glEdgeFlag 所设置的边缘标记的影响。

说明

本函数允许改变多边形的渲染方式。默认情况下，用当前颜色或材质属性给多边形进行填充或加上阴影。

8. glFrontFace：定义正面多边形和反面多边形。

C 语言描述

void glFrontFace(GLenum mode);

参数

mode　指定正对多边形的方向——顺时针（GL_CW）或逆时针（GL_CCW）。

说明

本函数用于定义多边形的哪一面被视为正面。如果从正面看时，多边形顶点的指定顺序是按顺时针方向绕多边形一周，则说这个多边形具有顺时针绕法；反之，顶点是按逆时针方向绕多边形一周，则说这个多边形具有逆时针绕法。通过本函数可以把顺时针或逆时针绕法的一面指定为多边形的正面。

9. glCullFace：指定剔出操作的多边形面

C 语言描述

void glCullFace(GLenum mode);

参数

mode　指定应剔除多边形的哪一个面，不是 GL_FRONT 就是 GL_BACK。

说明

本函数可以禁用多边形正面或背面上的光照、阴影和颜色计算及操作，消除不必要的渲染计算是因为无论对象如何进行旋转或变换，都不会看到多边形的背面。用GL_CULL_FACE参数调用 glEnable 和 glDisable 可以启用或禁用剔除。

10. glRect：绘制矩形

C 语言描述

void glRectd(GLdouble x1, GLdouble y1, GLdouble x2, GLdouble y2);
void glRectf(GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2);
void glRecti(GLint x1, GLint y1, GLint x2, GLint y2);
void glRects(GLshort x1, GLshort y1, GLshort x2, GLshort y2);
void glRectdv(const GLdouble *v1, const GLdouble *v2);
void glRectfv(const GLfloat *v1, const GLfloat *v2);
void glRectiv(const GLint *v1, const GLint *v2);
void glRectsv(const GLshort *v1, const GLshort *v2);

参数

x1，y1　指定矩形的左上角点。
x2，y2　指定矩形的右下角点。
*v1 两个值的数组，指定矩形的左上角点。
*v2 两个值的数组，指定矩形的右下角点。

说明

本函数根据指定的两个顶点坐标绘制矩形。矩形位于 z=0 的 xy 平面上。

1.3 坐标转换

1. gltranslate：用当前矩阵乘以平移矩阵

C 语言描述

void glTranslated(GLdouble x, GLdouble y, GLdouble z);

void glTranslatef(GLfloat x, GLfloat y, GLfloat z);

参数

x，y，z　指定平移矢量的 x，y，z 坐标。

说明

本函数用当前矩阵乘以由平移矢量指定的平移矩阵，并用结果矩阵替代当前矩阵。如果矩阵模式为 GL_MODELVIEW 或 GL_PROJECTION，则在调用本函数之后绘制的所有物体均被平移。

2. glRotate：用当前矩阵乘以旋转矩阵

C 语言描述

void glRotated(GLdouble angle, GLdouble x, GLdouble y, GLdouble z);

void glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z);

参数

angle 指定旋转的角度，单位为度。

x，y，z　指定一个自原点的方向矢量作为旋转轴。

说明

本函数用来计算旋转矩阵，该矩阵围绕原点到点（x，y，z）的向量逆时针旋转 angle 角，然后用当前矩阵乘以旋转矩阵，并用结果矩阵替代当前矩阵。如果矩阵模式为 GL_MODELVIEW 或 GL_PROJECTION，则在调用本函数之后绘制的所有物体均被旋转。

3. glScale：用当前矩阵乘以缩放矩阵

C 语言描述

void glScaled(GLdouble x, GLdouble y, GLdouble z);

void glScalef(GLfloat x, GLfloat y, GLfloat z);

参数

x，y，z　指定沿着 x，y，z 轴三个方向的缩放因子。

说明

本函数用当前矩阵乘以由三个轴向缩放因子指定的缩放矩阵，并用结果矩阵替代当前矩阵。如果矩阵模式为 GL_MODELVIEW 或 GL_PROJECTION，则在调用本函数之后绘制的所有物体均被缩放。

4. glViewport：设置视口

C 语言描述

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

参数

x，y 指定视口矩形的左下角坐标，单位为像素。缺省值为（0，0）。

width，height　指定视口矩形的宽度和高度。

说明

本函数在窗口内设置一个区域，这个区域用于将修剪空间的坐标映射到物理窗口的坐标。

5. glFrustum：用当前矩阵乘以透视矩阵

C 语言描述

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

参数

left，right　左修剪平面和右修剪平面的坐标。

bottom，top　下修剪平面和上修剪平面的坐标。

near，far　近修剪平面和远修剪平面的坐标，这两个值必须都是正值。

说明

本函数创建一个透视矩阵，该矩阵生成透视投影。假设视点位于（0，0，0），那么（left，bottom，-near）和（right，top，-near）分别指定近修剪平面中映射到窗口左下角和右上角的坐标，-far 指定远修剪平面中映射到窗口左下角和右上角的坐标，near 和 far 都必须为正值，相应的矩阵如下：

$$\begin{bmatrix} \frac{2near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & \frac{far+near}{far-near} & \frac{2\times far\times near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

本函数用当前矩阵乘以透视矩阵，并将结果矩阵替换当前矩阵。

6. glOrtho：用当前矩阵乘以正视矩阵

C 语言描述

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

参数

left，right，bottom，top　设置修剪空间最左、最右、最下和最上边的坐标。

near　从原点指向观察者的最大距离。

far　从原点远离观察者的最大距离。

说明

该函数描述了一个平行修剪空间。这种投影意味着离观察者较远的对象看起来不会变小。在三维笛卡尔坐标中想象这个修剪空间，左边和右边是最小和最大的 x 值，上边和下边是最小和最大的 y 值，近处和远处是最小和最大的 z 值。

7. glClipPlane：指定切割几何物体的平面

C 语言描述

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

参数

plane 指定定位的切割平面，这些平面的符号名称形式为 GL_CLIP_PLANEi，其中 i 是从 0 到 GL_MAX_CLIP_PLANES-1 之间的整数。

*equation 指定由四个双精度浮点数值组成的数组地址，这些值组成一个平面方程

说明

本函数用于指定附加的切割平面，这些平面可以不必垂直于 x，y 或 z 轴，最多可以指

定 GL_MAX_CLIP_PLANES 个。由于最终的切割区域在所定义的半个空间中是相互交叉的，因此它总是凸的。本函数用由四个分量组成的平面方程（Ax+By+Cz+D=0）来定义半空间，当调用本函数时，equation 经由模型视图矩阵的逆矩阵转换，并将结果存储在最终的视点坐标中，随后对模型视图矩阵所作的变换并不会影响到存储的平面方程的分量。如果顶点的视点坐标与存储的平面方程的分量的点积为正值或 0，则该顶点就在此切割平面上，否则就不在此平面上。

8. gluOrtho2D：定义二维正视投影矩阵

C 语言描述

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

参数

left，right　指定远端左边和远端右面的修剪平面。

bottom，top　指定远端下边和远端上面的修剪平面。

说明

本函数定义一个 2D 正射投影矩阵，该投影矩阵等价于调用 glOrtho 时把 near 和 far 分别设置为 0 和 1。在绘制平面图形时常常使用本函数。

9. gluPerspective：创建透视投影矩阵

C 语言描述

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

参数

Fovy　y 方向上可见区域的夹角（视场角），以度为单位。

Aspect　纵横比，用于确定 x 方向的可见区域。纵横比为 x（宽度）/y（高度）。

zNear，zFar　从观察者到近修剪平面和远修剪平面的距离，这两个值一定为正值。

说明

本函数创建一个矩阵，描述完全坐标中一个视图平截头体。纵横比应该与视见区的纵横比一致。透视分界线基于可见区域的夹角和到近修剪平面和远修剪平面的距离。

10. gluLookAt：定义视景转换

C 语言描述

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

参数

eyex，eyey，eyez　视点的位置。

centerx，centery，centerz　参考点的位置。

upx，upy，upz　向上矢量的方向。

说明

本函数根据视点，参考点和向上矢量创建视景矩阵，参考点表示场景的中心点，视景矩阵将参考点映射到负 z 轴方向，视点指向原点，以使在使用投影矩阵时，场景的中心映射到视区的中心，同样，投影到视场平面上的向上矢量的方向映射到正 y 轴上，以便它向上指向视口。向上矢量不必平行于由视点到参考点的视线方向。

1.4 堆栈操作

1.glLoadMatrix：用任意矩阵替换当前矩阵

C 语言描述

void glLoadMatrixf(const GLfloat *m);

void glLoadMatrixd(const GLdouble *m);

参数

m　指定一个指向 4×4 矩阵的指针，该矩阵按照已列为主的顺序存储 16 个相邻的数值。

说明

本函数用 m 中指定的值替换当前变换矩阵，当前变换矩阵按照当前的矩阵模式为投影矩阵、模型视图矩阵或纹理矩阵。m 指向一个 4×4 矩阵，该矩阵以列为主的顺序存储着单精度浮点数值或双精度浮点数值，即，矩阵按如下方式存储：

$$\begin{bmatrix} a0 & a4 & a8 & a12 \\ a1 & a5 & a9 & a13 \\ a2 & a6 & a10 & a14 \\ a3 & a7 & a11 & a15 \end{bmatrix}$$

2. glMultMatrix：用当前矩阵乘以任意矩阵

C 语言描述

void glMultMatrixf(const GLfloat *m);

void glMultMatrixd(const GLdouble *m);

参数

m　指定一个指向 4×4 矩阵的指针，该矩阵按照已列为主的顺序存储 16 个相邻的数值。

说明

本函数用当前选定的矩阵堆栈乘以由 m 指定的矩阵，然后把得到的矩阵存为矩阵堆栈顶部的当前矩阵。m 指向一个 4×4 矩阵，该矩阵以列为主的顺序存储着单精度浮点数值或双精度浮点数值。

3. glMatrixMode：指定哪一个矩阵是当前操作的矩阵

C 语言描述

void glMatrixMode(GLenum mode);

参数

mode　指定哪一个矩阵堆栈是下一个矩阵操作的目标，可以选择的数值为：GL_MODELVIEW，GL_PROJECTION，GL_TEXTURE。

说明

本函数设置当前的矩阵模式，mode 可以选择下面三个数值：

GL_MODELVIEW：对模型视图矩阵堆栈应用随后的矩阵操作。

GL_PROJECTION：对投影矩阵堆栈应用随后的矩阵操作。

GL_TEXTURE：对纹理矩阵堆栈应用随后的矩阵操作。

4. glPushMatrix,glPopMatrix：压入和弹出当前矩阵堆栈

C 语言描述

void glPushMatrix(void);

void glPopMatrix(void);

说明

每个矩阵模式都有一个矩阵堆栈，在 GL_MODELVIEW 模式中，堆栈深度至少为 32，而在 GL_PROJECTION 和 GL_TEXTURE 模式中，堆栈深度至少为 2，在任何模式下，当前

矩阵总是该模式下堆栈中的最顶层矩阵。glPushMatrix 函数把矩阵压入当前矩阵堆栈，并复制当前矩阵，也就是说，在调用 glPushMatrix 函数之后，堆栈顶层的矩阵同其下面的矩阵是一样的。glPopMatrix 函数从堆栈中弹出当前矩阵，并用堆栈中该矩阵下面的矩阵替代当前矩阵。

5. glLoadIdentity：将当前矩阵设置为单位矩阵

C 语言描述

void glLoadIdentity(void);

说明

本函数将当前变换矩阵替换为单位矩阵，这实质上是把坐标系复位成眼坐标。

1.5 使用光照和材质

1. glLight：设置光源参数

C 语言描述

void glLightf(GLenum light, GLenum pname, GLfloat param);

void glLighti(GLenum light, GLenum pname, GLint param);

void glLightfv(GLenum light, GLenum pname, const GLfloat *params);

void glLightiv(GLenum light, GLenum pname, const GLint *params);

参数

light 指定光照。光照的数目取决于实现，但至少可以支持 8 个光照，指定这些光照的符号常数为 GL_Lighti，其中 0≤i≤GL_MAX_LIGHTS。

pname 指定光照 light 的光源参数。

param 指定光源 light 的 pname 参数的设置值。

*params 指定指向光源 light 的 pname 参数的设置值指针。

说明

本函数用于设置单个光源的参数值。pname 的光照参数如下：

GL_AMBIENT params 参数包含四个整数值或浮点数值，这些值指定光照的环境 RGBA 浓度。整数值进行线形映射，最大的正整数值映射为 1.0，最小的负整数值映射为-1.0，浮点数值直接进行映射。缺省的环境光照浓度为（0.0，0.0，0.0，1.0）。

GL_DIFFUSE params 参数包含四个整数值或浮点数值，这些值指定光照的漫反射 RGBA 浓度。整数值进行线形映射，最大的正整数值映射为 1.0，最小的负整数值映射为-1.0，浮点数值直接进行映射。对于光源 0，缺省的漫反射光照浓度为（1.0，1.0，1.0，1.0），而对于其他光源，缺省的漫反射光浓度为（0.0，0.0，0.0，1.0）。

GL_SPECULAR params 参数包含四个整数值或浮点数值，这些值指定光照的镜面 RGBA 浓度。整数值进行线形映射，最大的正整数值映射为 1.0，最小的负整数值映射为-1.0，浮点数值直接进行映射。对于光源 0，缺省的镜面光照浓度为（1.0，1.0，1.0，1.0），而对于其他光源，缺省的镜面光浓度为（0.0，0.0，0.0，1.0）。

GL_POSITION params 参数包含四个整数值或浮点数值，这些值以齐次物体坐标指定光照的位置。整数值和浮点数值直接进行映射。当调用 glLight 函数时，光照位置使用模型视图矩阵进行转换，并存储在视点坐标中。如果位置坐标的 w 分量为 0.0，则光源为方向光源，这时漫反射光和镜面光的计算考虑光照的方向，但不考虑实际的位置，而且关闭衰减。若 w 分量为 1.0，则光源为位置光源，漫反射光和镜面光的计算要考虑光照的方向，且激活衰减。缺省时的光照位置为（0，0，1，0），这是一个方向光源，其方向平行于-z 轴。

GL_SPOT_DIRECTION params 参数包含三个整数值或浮点数值，这些值以齐次物体

坐标指定光照的方向。整数值和浮点数值直接进行映射。当调用 glLight 函数时，聚光源的方向使用模型视图矩阵的逆矩阵进行转换，并存储在视点坐标中。只有当 GL_SPOT_CUTOFF 的值不等于 180（度）时，该参数值才是有意义的。缺省的 GL_SPOT_CUTOFF 值等于 180（度），缺省时的方向为（0，0，-1）。

GL_SPOT_EXPONENT　params 参数为单个整数值或浮点数值，它表示光照的浓度分布。整数值和浮点数值直接进行映射。有效的光照浓度是按照夹角余弦，并逐渐增加聚光源的指数进行衰减的，夹角是指光照方向和光照到所照射的顶点方向之间的角度，这样不管聚光源的夹角有多大，较大的聚光指数会产生有更强聚焦效果的光源。缺省时的聚光指数为 0，它是均匀照射的光源。

GL_SPOT_CUTOFF　params 参数是单个整数值或浮点数值，它表示光源的最大散布角。整数值和浮点数值直接进行映射。这个角度只可以选择[0，90]之间的值以及特定值 180。如果光照方向和光照到所照射的顶点方向之间的角度大于这个角度，那么光照完全被屏蔽。否则，光照浓度受聚光指数和衰减因子的控制。缺省时的聚光角度为 180，它是均匀照射的光源。

GL_CONSTANT_ATTENUATION，GL_LINEAR_ATTENUATION，GL_QUADRATIC_ATTENUATION　params 参数是单个整数值或浮点数值，它是三个光照衰减因子中的一个。整数值和浮点数值直接进行映射，params 只能选择非负值。如果光源是位置光源而不是方向光源，那么光照浓度按照和值的倒数进行衰减，这个和值是下面数值之和：常值因子、线形因子乘以光照到所照射点之间的距离，以及二次因子乘以相同距离的平方。缺省时的衰减因子为（1，0，0），它表示光照不进行衰减。

注意，光照计算需要调用 glEnable（GL_LIGHTING）激活。

2. glLightModel：设置光照模型参数

C 语言描述

```
void glLightModelf(GLenum pname, GLfloat param);
void glLightModeli(GLenum pname, GLint param);
void glLightModelfv(GLenum pname, const GLfloat *param);
void glLightModeliv(GLenum pname, const GLint *param);
```

参数

pname　指定单值光照模型参数。

param　指定 pname 的设置值。

*param　指向 pname 设置值的指针。

说明

本函数指定光照模型参数。有效的光照模型参数有三个：

GL_LIGHT_MODEL_AMBIENT　params 参数包含四个整数值或浮点数值，这些值指定了整个场景中的环境 RGBA 浓度。整数值进行线形映射，最大的正整数值映射为 1.0，最小的负整数值映射为-1.0，浮点数值直接进行映射。缺省的环境场景浓度为（0.2，0.2，0.2，1.0）。

GL_LIGHT_MODEL_LOCAL_VIEWER　params 参数是单个整数值或浮点数值，这个值指定了如何计算镜面反射角度。如果 params 为 0，那么不管视点坐标中顶点的位置在哪里，镜面反射角度认为视线方向是平行于 z 轴负方向，否则，镜面反射从视点坐标系统的原点来计算，缺省值为 0。

GL_LIGHT_MODEL_TWO_SIDE　params 参数是单个整数值或浮点数值，它表示对多边形是执行单面光照计算还是双面光照计算，但他对于点、线或位图的光照计算没有影响。

如果 params 为 0，则指定的是单面光照，在光照方程中使用 front 材质参数，否则指定的是双面光照，这时反面多边形的顶点使用 back 材质参数照射，在求取光照方程之前，翻转法向量。正面多边形的顶点总是使用 front 材质参数照射，而且不改变法向量。缺省值为 0。

在 RGBA 模式下，顶点的光照颜色是下列数值之和：材质散射浓度、材质环境反射和光照模型全景环境浓度的乘积、以及每个激活光源的基值。每个光源都提供下面三项的和值：环境光、漫射光和镜面光。环境光源的基值是材质环境反射和光照环境浓度的乘积。漫射光源的基值是下列数值的乘积：材质漫反射、光照的漫射浓度、顶点法向量同顶点到光源的单位化向量的点积。镜面光源的基值是下列数值的乘积：材质镜面反射、光照的镜面浓度、单位化的顶点到视点的向量与顶点到光源的向量之间的点积。三个光源基值按照顶点到光源的距离和光源的方向、散布的光照指数和散布的光照角度进行等量的衰减。如果点积的计算结果是负值，那么就用零替代所有的点积结果。

3. glMaterial：为光照模型指定材质参数

C 语言描述

```
void glMaterialf(GLenum face, GLenum pname, GLfloat param);
void glMateriali(GLenum face, GLenum pname, GLint param);
void glMaterialfv(GLenum face, GLenum pname, const GLfloat *params);
void glMaterialiv(GLenum face, GLenum pname, const GLfloat *params);
```

参数

face　指定需要更改材质的那些面，必须为 GL_FRONT、GL_BACK 或 GL_FRONT_AND_BACK。

pname　指定需要更改的那些面的材质参数。

param　指定 pname 的设置值。

*params　指定指向 pname 设置值的数值指针。

说明

本函数为材质参数应用参数值，该函数有三个变量，第一个变量 face 指定需要更改的是哪个面的材质；第二个变量 pname 指定需要更改的是这些材质中的哪些参数；第三个变量 params 设置更改的参数值。pname 可以选择的参数值如下：

GL_AMBIENT　params 参数包含四个整数值或浮点数值，这些值指定材质环境反射的 RGBA 值。整数值进行线形映射，最大的正整数值映射为 1.0，最小的负整数值映射为-1.0，浮点数值直接进行映射。对于正面材质和反面材质，缺省的环境反射值为（0.2，0.2，0.2，1.0）。

GL_DIFFUSE　params 参数包含四个整数值或浮点数值，这些值指定材质漫反射的 RGBA 值。整数值进行线形映射，最大的正整数值映射为 1.0，最小的负整数值映射为-1.0，浮点数值直接进行映射。对于正面材质和反面材质，缺省的漫反射值为（0.8，0.8，0.8，1.0）。

GL_SPECULAR　params 参数包含四个整数值或浮点数值，这些值指定材质镜面反射的 RGBA 值。整数值进行线形映射，最大的正整数值映射为 1.0，最小的负整数值映射为-1.0，浮点数值直接进行映射。对于正面材质和反面材质，缺省的镜面反射值为（0.0，0.0，0.0，1.0）。

GL_EMISSION　params 参数包含四个整数值或浮点数值，这些值指定材质散射光照浓度的 RGBA 值。整数值进行线形映射，最大的正整数值映射为 1.0，最小的负整数值映射为-1.0，浮点数值直接进行映射。对于正面材质和反面材质，缺省的散射浓度为（0.0，0.0，0.0，1.0）。

GL_SHININESS　params 参数是单个整数值或浮点数值，这个值指定材质的 RGBA 镜

面指数。整数值和浮点数值直接进行映射，只可以选择[0，128]范围之内的值。对于正面材质和反面材质，缺省的镜面指数为 0。

GL_AMBIENT_AND_DIFFUSE　相当于使用相同的参数值调用两次 glMaterial 函数，一个材质参数为 GL_AMBIENT，另一个参数为 GL_DIFFUSE。

GL_COLOR_INDEXES　params 参数包含三个整数值或浮点数值，这些值指定环境光、漫射光和镜面光的颜色索引，这三个值及 GL_SHININESS 值是颜色索引模式下光照方程使用的唯一的材质值。

4. glColorMaterial：使材质颜色跟踪当前颜色

C 语言描述

void glColorMaterial(GLenum face, GLenum mode);

参数

face　指定正面材质、反面材质或正反面材质参数是否跟踪当前颜色，可以选择的值为 GL_FRONT、GL_BACK 或 GL_FRONT_AND_BACK，缺省值为 GL_FRONT_AND_BACK。

mode　指定哪些材质参数跟踪当前颜色，可以选择的值为：GL_EMISSION、GL_AMBIENT、GL_DIFFUSE、GL_SPECULAR 和 GL_AMBIENT_AND_DIFFUSE，缺省值为 GL_ AMBIENT_AND_DIFFUSE。

说明

本函数指定哪些材质参数跟踪当前颜色。当激活 GL_COLOR_MATERIAL 时，由 face 指定的材质中 mode 指定的材质参数将会一直跟踪当前颜色，可以使用 glEnable 和 glDisable 函数，变量为 GL_COLOR_MATERIAL 激活或关闭颜色跟踪，缺省时颜色跟踪是关闭的。使用本函数，只要调用 glColor 就可以改变每个顶点的材质参数子集，而不必调用 glMaterial 函数。

5. glNormal：设置当前的法向量

C 语言描述

```
void glNormal3b(GLbyte nx, GLbyte ny, GLbyte nz);
void glNormal3d(GLdouble nx, GLdouble ny, GLdouble nz);
void glNormal3f(GLfloat nx, GLfloat ny, GLfloat nz);
void glNormal3i(GLint nx, GLint ny, GLint nz);
void glNormal3s(GLshort nx, GLshort ny, GLshort nz);
void glNormal3bv(const GLbyte *v);
void glNormal3dv(const GLdouble *v);
void glNormal3fv(const GLfloat *v);
void glNormal3iv(const GLint *v);
void glNormal3sv(const GLshort *v);
```

参数

nx，ny，nz　指定新的当前法向量的 x，y，z 坐标，当前法向量的初始值为（0，0，1）。

*v　指定一个指向数组的指针，该数组由三个元素组成：新的当前法向量的 x，y，z 坐标。

说明

无论何时调用 glNormal，当前法向量均被设置为给定的坐标值。用 glNormal 指定的法向量不必是单位向量，如果激活了单位化，那么在转换之后，由 glNormal 指定的法向量自动进行单位化。单位化可以使用 glEnable 和 glDisable 函数，变量为 GL_NORMALIZE 激活

和关闭。缺省时单位化是关闭的

1.6 帧缓存操作

1. glClearColor：设置颜色缓存的清除值

C 语言描述

void glClearColor(GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha);

参数

red，green，blue，alpha　指定清除颜色缓存时所使用的红、绿、蓝和 alpha 值。

说明

指定由 glClear 清除颜色缓存时所使用的红、绿、蓝和 alpha 值，指定值的范围固定为[0.0f，1.0f]。

2. glClear：将缓存清除为预先的设置值

C 语言描述

void glClear(GLbitfield mask);

参数

mask　对指定的需要清除的缓存进行按位或屏蔽操作，这四个屏蔽值如下：GL_COLOR_BUFFER_BIT，GL_DEPTH_BUFFER_BIT，GL_ACCUM_BUFFER_BIT 和 GL_STENCIL_BUFFER_BIT。

说明

本函数只有一个变量，这个变量对所清除的缓存值进行按位或操作，这些值如下：

GL_COLOR_BUFFER_BIT　指定当前被激活为写操作的颜色缓存。

GL_DEPTH_BUFFER_BIT　指定深度缓存。

GL_ACCUM_BUFFER_BIT　指定累加缓存。

GL_STENCIL_BUFFER_BIT　指定模板缓存。

3. glClearDepth：设置深度缓存的清除值

C 语言描述

void glClearDepth(GLclampd depth);

参数

depth　指定清除深度缓存时使用的深度值。

说明

本函数指定用 glClear 清除深度缓存时所使用的深度值，该值的范围在[0，1]之间。

1.7 查询函数

1. glGet：返回所选择的参数值

C 语言描述

void glGetBooleanv(GLenum pname, GLboolean *params);

void glGetDoublev(GLenum pname, GLdouble *params);

void glGetFloatv(GLenum pname, GLfloat *params);

void glGetIntegerv(GLenum pname, GLint *params);

参数

pname　指定返回的参数值，可以选择的符号常数看下面的说明。

params　返回指定的参数值。

说明

本函数为 OpenGL 中简单状态变量返回数值，如果 params 的数据类型并不是状态变量要求的数据类型，则执行类型转换。

下面是 pname 可以选择的部分符号常数：

GL_COLOR_CLEAR_VALUE　params 返回四个值：用来清除颜色缓存的红色、绿色、蓝色和 alpha 值。如果需要，整数值由内部的浮点表达式进行线型映射，1.0 返回最大的正整数，-1.0 返回最小的正整数值。

GL_COLOR_MATERIAL　params 返回一个布尔数值，该值表示是否有一个或多个材质参数正在跟踪当前颜色。

GL_COLOR_MATERIAL_FACE　params 返回一个数值，该值为符号常数，表示有哪些材质正在跟踪当前颜色的参数。

GL_COLOR_MATERIAL_PARAMETER　params 返回一个数值，该值为符号常数，表示哪些材质参数正在跟踪当前颜色。

GL_CULL_FACE　params 返回一个布尔数值，该值表示是否能使多边形切割。

GL_CULL_FACE_MODE　params 返回一个数值，该值为符号常数，它表示哪些多边形的面被剔除。

GL_CURRENT_COLOR　params 返回四个数值：当前颜色的红、绿、蓝和 alpha 值。

GL_CURRENT_NORMAL　params 返回三个数值：当前法向量的 x，y 和 z 值。

GL_DOUBLEBUFFER　params 返回一个布尔值，该值表示是否能支持双缓存。

GL_EDGE_FLAG　params 返回一个布尔值，该值表示当前的边界标记是 TRUE 还是 FALSE。

GL_FRONT_FACE　params 返回一个数值，该值为符号常数，表示是顺时针多边形还是逆时针多边形为正面多边形。

GL_LIGHT_MODEL_AMBIENT　params 返回四个数值：整个场景中环境浓度的红、绿、蓝和 alpha 值。

GL_LIGHT_MODEL_TWO_SIDE　params 返回一个布尔值，该值表示是否使用分离的材质计算正面多边形和反面多边形的光照。

GL_LINE_STIPPLE_PATTERN　params 返回一个数值，该值为 16 位直线点划图。

GL_LINE_STIPPLE_REPEAT　params 返回一个数值，该值为直线点划图的重复因子。

GL_LINE_WIDTH　params 返回一个数值，该值为当前指定的线宽。

GL_LINE_WIDTH_RANGE　params 返回两个数值，线段所支持的最小宽度和最大宽度。

GL_MAX_CLIP_PLANES　params 返回一个数值，该值为应用程序定义的切割平面的最大数目。

GL_MAX_LIGHTS　params 返回一个数值，该值为光照的最大数目。

GL_MAX_MODELVIEW_STACK_DEPTH　params 返回一个数值，该值为模型视图矩阵堆栈支持的最大深度。

GL_MAX_PROJECTION_STACK_DEPTH　params 返回一个数值，该值为投影矩阵堆栈支持的最大深度。

GL_MODELVIEW_MATRIX　params 返回 16 个数值：在模型视图矩阵堆栈顶层的模型视图矩阵。

GL_MODELVIEW_STACK_DEPTH　params 返回一个数值，该值为模型视图矩阵堆栈的矩阵数目。

GL_POINT_SIZE　params 返回一个数值，该值为当前指定的点的大小。

GL_POINT_SIZE_RANGE　params 返回两个数值：点大小的最小值和最大值。

GL_POLYGON_MODE　params 返回两个数值：指定是正面多边形还是反面多边形被光栅化为点、线或实多边形的符号常数。

GL_PROJECTION_MATRIX　params 返回 16 个数值：在投影矩阵堆栈顶层的模型视图矩阵。

GL_ PROJECTION_STACK_DEPTH　params 返回一个数值，该值投影矩阵堆栈的矩阵数目。

GL_SHADE_MODEL　params 返回一个数值，该值为符号常数，表示阴影模式是平面明暗模式还是光滑明暗模式。

GL_VIEWPORT　params 返回四个数值：视口的 x 和 y 窗口坐标，接下来是视口的宽度和高度。

2.glGetClipPlane：返回指定的切平面系数

C 语言描述

void glGetClipPlane(GLenum plane, GLdouble *equation);

参数

plane　指定用 GL_CLIP_PLANEi 标示的切割平面。

*equation　返回视点坐标中 plane 平面方程的系数。

说明

本函数可以获得切割平面方程的四个系数。

3. glGetLight：返回光源参数值

C 语言描述

void glGetLightfv(GLenum light, GLenum pname, GLfloat *params);

void glGetLightiv(GLenum light, GLenum pname, GLint *params);

参数

light　指定名为 GL_LIGHTi(0≤i≤GL_MAX_LIGHTS)。

pname　指定光源参数。

*param　返回请求的数据。

说明

本函数用于获取光源的参数，pname 指定的光源参数如下：

GL_AMBIENT　params 返回四个整数值或浮点数值，这些值表示光源的环境 RGBA 浓度。

GL_DIFFUSE　params 返回四个整数值或浮点数值，这些值表示光源的漫反射 RGBA 浓度。

GL_SPECULAR　params 返回四个整数值或浮点数值，这些值表示光源的镜面 RGBA 浓度。

GL_POSITION　params 返回四个整数值或浮点数值，这些值表示光源的位置。

GL_SPOT_DIRECTION　params 返回三个整数值或浮点数值，这些值表示光源的方向。

GL_SPOT_EXPONENT　params 返回一个整数值或浮点数值，它表示聚光源指数。

GL_SPOT_CUTOFF　params 返回单个整数值或浮点数值，它表示聚光源的角度。

GL_CONSTANT_ATTENUATION　params 返回单个整数值或浮点数值，它表示光照的恒定衰减值（与距离无关）。

GL_LINEAR_ATTENUATION　params 返回单个整数值或浮点数值，它表示光照的线性

衰减值。

GL_QUADRATIC_ATTENUATION　params 返回单个整数值或浮点数值，它表示光照的二次衰减值。

4. glGetMaterial：返回材质参数

C 语言描述

void glGetMaterialfv(GLenum face, GLenum panem, GLfloat *params);

void glGetMaterialiv(GLenum face, GLenum panem, GLint *params);

参数

face　指定查询的是哪个面的材质，必须为 GL_FRONT、GL_BACK 或 GL_FRONT_AND_BACK。

pname　指定返回的材质参数。

*params　返回请求的数据。

说明

本函数用于获取材质参数。pname 可以选择的参数值如下：

GL_AMBIENT　params 返回四个整数值或浮点数值，这些值表示材质环境反射的 RGBA 值。

GL_DIFFUSE　params 返回四个整数值或浮点数值，这些值表示材质漫反射的 RGBA 值。

GL_SPECULAR　params 返回四个整数值或浮点数值，这些值表示材质镜面反射的 RGBA 值。

GL_EMISSION　params 返回四个整数值或浮点数值，这些值表示材质漫反射光浓度的 RGBA 值。

GL_SHININESS　params 返回单个整数值或浮点数值，这个值表示材质的 RGBA 镜面指数。

GL_COLOR_INDEXES　params 返回三个整数值或浮点数值，这些值表示材质的环境反射、漫射反射和镜面反射的颜色索引值。

1.8 窗口初始化和启动事件处理

1. glutInit：初始化 GLUT 库

C 语言描述

void glutInit(int *argcp, char **argv);

参数

*argcp　指向 main 函数 argc 变量的指针。由于 glutInit 用于初始化 GLUT 库的命令行选项，因此当函数返回时，argcp 指向的数值会被更新。

*argv　main 函数的 argv 变量。

说明

本函数用来初始化 GLUT 库并同窗口系统对话协商。在此过程中，如果 GLUT 不能正确初始化，glutInit 函数将会终止 GLUT 程序，并将错误信息回送给用户。

2. glutInitWindowPosition：设置初始窗口的位置

C 语言描述

void glutInitWindowPosition(int x, int y);

参数

x　窗口左上角的 x 坐标，以像素为单位。

y　窗口左上角的 y 坐标，以像素为单位。

说明

本函数用当前初始的窗口位置和初始的窗口大小创建新窗口。对应初始窗口位置的 GLUT 状态值开始时为-1 和-1。创建窗口时，如果初始窗口位置的任意一个分量为零，那么窗口的实际位置就由窗口系统来决定。注意，初始窗口位置的目的是向窗口系统建议窗口的初始位置，窗口系统未必使用该信息。

3. glutInitWindowSize：设置初始窗口的大小

C 语言描述

void glutInitWindowSize(int width, int height);

参数

width　初始窗口的宽度，以像素为单位。

height　初始窗口的高度，以像素为单位。

说明

本函数用当前初始的窗口大小和初始的窗口位置创建新窗口。对应初始窗口大小的 GLUT 状态值开始时为 300 和 300。初始窗口大小的分量值必须大于 0。注意，初始窗口大小的目的是向窗口系统建议窗口的初始大小，窗口系统未必使用该信息。

4. glutInitDisplayMode：设置初始显示模式

C 语言描述

void glutInitDisplayMode(unsigned int mode);

参数

mode　指定初始窗口的显示模式，通常是若干指明窗口帧缓存特性的标识的组合。GLUT 定义了如下标识：

GLUT_RGBA 或 GLUT_RGB　指定 RGBA 颜色模式的窗口。

GLUT_INDEX　指定颜色索引模式的窗口。

GLUT_SINGLE　指定单缓存窗口。

GLUT_DOUBLE　指定双缓存窗口。

GLUT_ACCUM　窗口使用累加缓存。

GLUT_ALPHA　窗口的颜色分量包含 alpha 值。

GLUT_DEPTH　窗口使用深度缓存。

GLUT_STENCIL　窗口使用模板缓存。

GLUT_MULTISAMPLE　指定支持多样本功能的窗口，如果系统不支持多样本功能，则自动选择非多样本窗口。

GLUT_STEREO　指定立体窗口。

GLUT_LUMINANCE　窗口使用亮度颜色模型。亮度颜色模型提供了 OpenGL RGBA 颜色模型的功能，但是帧缓存中没有绿色和蓝色分量，而且每个像素的红色分量转换为 0 到 glutGet(GLUT_WINDOW_COLORMAP_SIZE)-1 之间的索引值。然后再在颜色索引表中查找来确定该像素的颜色。注意，大多数 OpenGL 平台都不支持这种模型。

说明

当创建窗口时，本函数用于确定所创建窗口的显示模式。注意，当使用 GLUT_RGBA 选择 RGBA 颜色模式时，alpha 缓存并没有创建，因此，如果需要用到 alpha，应同时指定 GLUT_ALPHA。

5. glutMainLoop：进入 GLUT 事件处理循环

C 语言描述

void glutMainLoop(void);

说明

调用本函数进入 GLUT 事件处理循环。glutMainLoop 函数在 GLUT 程序中最多只能调用一次，它一旦调用就不再返回，并且调用注册过的回调函数。

1.9 窗口管理

1. glFlush：刷新 OpenGL 命令队列和缓冲区

C 语言描述

void glFlush(void);

说明

刷新 OpenGL 命令队列和缓冲区。OpenGL 命令常常会排队并成批处理以便优化性能。glFlush 命令使得所有正在等待的命令得到执行。

2. glutCreateWindow：创建顶层窗口

C 语言描述

void glutCreateWindow(char *name);

参数

name 用来作为窗口标题的 ASCII 码字符串。

说明

本函数用于创建窗口时，以 name 作为窗口的名字，同时新建窗口被置为当前窗口。每个窗口都有一个与之关联的唯一的 OpenGL 上下文，窗口创建后，能够立即对 OpenGL 上下文中的状态进行变动。

窗口的显示状态开始是可视的，但在调用 glutMainLoop 函数后，窗口才显示在屏幕上，这意味着，直到调用 glutMainLoop 函数前，所有的绘图命令都无效，因为窗口还没有绘制出来。

3. glutCreateSubWindow：创建子窗口

C 语言描述

void glutCreateSubWindow(int win, int x, int y, int width, int height);

参数

win 子窗口的父窗口的标识符。

x，y 子窗口相对于父窗口原点的 x 和 y 坐标（以像素为单位）。

width，height 子窗口的宽度和高度（以像素为单位）。

说明

本函数在当前窗口 win 中创建一个宽为 width，高为 height 的位于（x，y）处的子窗口，同时新建的子窗口被设置成当前窗口。子窗口可以任意嵌套。

4. glutHideWindow：隐藏当前窗口的显示状态

C 语言描述

void glutHideWindow(void);

说明

本函数改变当前窗口的显示状态，使其隐藏，但是该函数的调用结果直到返回主事件循环才生效。

5. glutShowWindow：改变当前窗口的显示状态，使其显示

C 语言描述

void glutShowWindow(void);

说明

本函数改变当前窗口的显示状态，使其显示，但是该函数的调用结果直到返回主事件循环才生效。

6. glutSetWindowTitle：设置当前顶层窗口的窗口标题

C 语言描述

void glutSetWindowTitle(char *name);

参数

name　由 glutCreateWindow 函数中的 name 变量确定的顶层窗口名称。

说明

本函数应该在当前窗口为顶层窗口时调用。当创建顶层窗口时，顶层窗口的窗口标题由 glutCreateWindow 函数中的 name 变量确定，但窗口一旦建立后，窗口标题就可以由本函数来修改。

7. glutPostRedisplay：标记当前窗口需要重新绘制

C 语言描述

void glutPostRedisplay(void);

说明

本函数标记当前窗口的图像层需要重新绘制，在 glutMainLoop 函数的事件处理循环的下一个反复中，将调用该窗口的显示回调函数重绘该窗口的图像层。

8. glutSwapBuffers：交换当前窗口的缓存

C 语言描述

void glutSwapBuffers(void);

说明

本函数交换当前窗口的使用层的缓存，它将后台缓存中的内容交换到前台缓存中，该函数执行的结果直到显示器垂直回扫后才看得到。如果使用层不是双缓存结构，本函数将不起任何作用。

9. glutFullScreen：使窗口全屏显示

C 语言描述

void glutFullScreen(void);

说明

本函数申请把当前窗口用全屏显示。本函数的申请不会立即被处理，只有返回到主事件循环之后它才被执行。

10. glutPositionWindow：申请改变当前窗口的位置

C 语言描述

void glutPositionWindow(int x, int y);

参数

x　新的窗口位置坐标的 x 分量，以像素为单位。

y　新的窗口位置坐标的 y 分量，以像素为单位。

说明

本函数申请改变当前窗口的位置，对于顶层窗口，x 和 y 参数是相对于屏幕原点的像素偏移值，对于子窗口，x 和 y 参数是相对于父窗口原点的像素偏移值。本函数的申请并不会立即被处理，只有返回到主事件循环之后它才被执行。

11. glutReshapeWindow：申请改变当前窗口的大小

C 语言描述

void glutReshapeWindow(int width, int height);

参数

width　新窗口的宽度，以像素为单位。

height　新窗口的高度，以像素为单位。

说明

本函数申请改变当前窗口的大小，width 和 height 变量指定窗口的新尺寸，它们必须为正值。本函数的申请并不会立即被处理，只有返回到主事件循环之后它才被执行。

12. glutSetWindow：设置当前窗口

C 语言描述

void glutSetWindow(int win);

参数

win　被设置成当前窗口的窗口标识符。

说明

本函数设置当前窗口，但不改变窗口的使用层。

13.glutGetWindow：获得当前窗口的标识符

C 语言描述

int glutGetWindow(void);

说明

本函数返回当前窗口的标识符，如果窗口不存在，或者删除了前一个当前窗口，则本函数返回 0。

14. glutPopWindow：改变当前窗口的位置，使其前移

C 语言描述

void glutPopWindow(void);

说明

本函数对顶层窗口和子窗口都起作用，但是，弹出窗口的结果并不会立即产生，相反，弹出操作会一直保存，直到进入事件循环，该操作才会执行。

15. glutPushWindow：改变当前窗口的位置，使其后移

C 语言描述

void glutPushWindow(void);

说明

本函数对顶层窗口和子窗口都起作用，但是，压入窗口的结果并不会立即产生，相反，压入操作会一直保存，直到进入事件循环，该操作才会执行。

16. glutDestroyWindow：销毁指定的窗口

C 语言描述

void glutDestroyWindow(int win);

参数

win　指定要销毁的 GLUT 窗口的窗口标识符。

说明

本函数销毁由 win 指定的窗口以及与窗口关联的 OpenGL 上下文、逻辑颜色表（若窗口是颜色索引模式）和重叠层（如果存在），该窗口的任何子窗口也被一同销毁，如果 win 为当前窗口，则当前窗口在此函数调用后变成空值。

1.10 菜单管理

1. glutCreateMenu：创建一个新的弹出式菜单

C 语言描述

int glutCreateMenu(void (*func)(int value));

参数

func　当选择菜单上的一个菜单条目时，func 定义被调用的回调函数。

value　传递给回调函数的数值，它由所选择的菜单条目对应的整型值决定。

说明

本函数创建一个新的弹出式菜单并返回一个唯一标识此菜单的整型标识符，并将新建的弹出菜单与 func 函数关联在一起，这样，当选择此菜单中的一个菜单条目时，调用回调函数 func，传递给他的参数 value 指定选取的菜单条目。

2. glutAddMenuEntry：在当前菜单的底部增加一个菜单条目

C 语言描述

void glutAddMenuEntry(char *name, int value);

参数

name　显示在新菜单条目上的 ASCII 码字符串。

valur　当选择该菜单条目时，传递到菜单回调函数中的数值。

说明

本函数在当前菜单的底部增加一个菜单条目，字符串 name 将显示在新增加的菜单条目上。如果用户选择了这个菜单条目，对应的菜单回调函数就被调用，并以 value 值作为传递给此回调函数的参数。

3.glutAddSubMenu：在当前菜单的底部增加一个子菜单触发条目

C 语言描述

void glutAddSubMenu(char *name, int menu);

参数

name　显示在新子菜单触发条目上的 ASCII 码字符串。

valur　当选择该子菜单触发条目时弹出的子菜单的标识符。

说明

本函数在当前菜单的底部增加一个子菜单触发条目，字符串 name 将显示在新增加的子菜单触发条目上。如果用户选择了这个菜单项，则 menu 标识的子菜单就会弹出，使用户可以在子菜单中做进一步的选择。

4. glutAttachMenu：把当前窗口的一个鼠标按键与当前菜单的标识符联系起来

C 语言描述

void glutAttachMenu(int botton);

参数

botton　指明何种鼠标按键，它可以选择的符号常数为：GLUT_LEFT_BUTTON、GLUT_MIDDLE_BUTTON 或 GLUT_RIGHT_BUTTON，分别表示鼠标左键、中键和右键。

说明

本函数把当前窗口的一个鼠标按键与当前菜单的标识符联系起来，通过把菜单标识附到一个鼠标按键上，当用户点取该按键时，菜单标识符标识的菜单将弹出。

5. glutGetMenu：获取当前菜单的标识符

C 语言描述

int glutgetMenu(void);

说明

本函数获取当前菜单的标识符，如果没有菜单存在或前一个当前菜单被删除了，本函数返回 0 值。

6. glutSetMenu：设置当前菜单

C 语言描述

void glutSetMenu(int menu);

参数

menu　指明菜单标识符，其标识的菜单将被设置成当前菜单。

说明

本函数将 menu 标识的菜单设置成当前菜单。

7. glutDestroyMenu：删除指定的菜单

C 语言描述

void glutDestroyMenu(int menu);

参数

menu　被删除的菜单的标识符。

说明

本函数删除由 menu 指定的菜单。如果 menu 是当前菜单，则当前菜单变为空值，此时调用 glutGetMenu 返回 0 值。当一个菜单被删除时，对其能够触发的子菜单没有影响。

8. glutRemoveMenuItem：删除指定的菜单项

C 语言描述

void glutRemoveMenuItem(int entry);

参数

entry　当前菜单中要删除的菜单项的索引（最顶层的菜单项的索引值为 1）。

说明

本函数删除由 entry 指定的菜单项，不管它是一个菜单条目还是一个子菜单触发条目。

1.11 注册回调函数

1. glutDisplayFunc：注册当前窗口的显示回调函数

C 语言描述

void glutDisplayFunc(void (*func)(void));

参数

func　指明显示回调函数。

说明

本函数注册当前窗口的显示回调函数。当一个窗口的图像层需要重新绘制时，GLUT 将调用该窗口的显示回调函数。GLUT 根据窗口的重绘状态判定是否调用该窗口的显示回调函数。窗口的重绘状态在如下情况下设置：一是调用了 glutPostRedisplay 函数；二是用户进行了某些窗口操作（使窗口的大小改变了，使窗口被挡住的部分重新显示出来等）。多个要求窗口重新绘制的请求将被结合在一起，以便使显示回调函数的调用次数减少到最少的程度。

当窗口创建时，该窗口的显示回调函数并不存在，显示回调函数在窗口显示前必须注册，否则窗口的显示将导致致命错误。

2. glutReshapeFunc：注册当前窗口的形状变化回调函数

C 语言描述

void glutReshapeFunc(void(*func)(int width, int height));

参数

func　指定形状变化回调函数。

width　新窗口的宽度。

height　新窗口的高度。

说明

本函数注册当前窗口的形状变化回调函数。当改变窗口的大小时，该窗口的形状改变回调函数将被调用。此外，在下列情况下也将调用该窗口的形状改变回调函数：一是当一个窗口创建后，在第一次调用它的显示回调函数之前；二是当创建窗口的重叠层时。回调函数中的 width 和 height 参数是窗口的新尺寸（以像素为单位）。在回调前，当前窗口的大小设置为改变后窗口的大小。

如果一个窗口没有注册形状改变回调函数，或者用 NULL 作为参数调用了 glutReshapeFunc（注销前面注册了的形状变化回调函数），那么当窗口的大小变化时，调用缺省的形状改变回调函数。缺省的形状改变回调函数仅包含一条语句：glViewPort（0，0，width，height）。

当顶层窗口的大小改变时，子窗口的大小改变时，也只产生一次形状变化回调，因此，形状变化回调函数有责任同时更新窗口的图像层和重叠层。

3.glutMouseFunc：注册当前窗口的鼠标回调函数

C 语言描述

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

参数

func　指定鼠标回调函数。

button　定义鼠标的按键。

state　定义鼠标按键的动作。

x　当按下鼠标时，鼠标相当于窗口左上角的 x 坐标。

y　当按下鼠标时，鼠标相当于窗口左上角的 y 坐标。

说明

本函数注册当前窗口的鼠标回调函数。当用户在窗口中按下或释放鼠标键时，每一次的按下动作和每一次的释放动作都产生一次鼠标回调。button 变量可以选择下列常值中的一个：GLUT_LEFT_BUTTON，GLUT_MIDDLE_BUTTON，GLUT_RIGHT_BUTTON 回调。state 变量的值是 GLUT_UP 和 GLUT_DOWN 中的一个，GLUT_UP 表示回调是由按下动作产生的，GLUT_DOWN 表示回调是由释放动作产生的。x 和 y 变量指定鼠标按键的状态变化时鼠标相对于窗口左上角的位置（以像素为单位）。当对一个按键的 GLUT_DOWN 回调产生时，程序可以假定当释放这个按键时，对同一个按键的 GLUT_UP 回调也将产生，即使鼠标已经移到了窗口外。

如果有菜单与窗口的一个鼠标按键连到一块，则对这个按键的鼠标回调将不再产生。

4.glutKeyboardFunc：注册当前窗口的键盘回调函数

C 语言描述

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

参数

func　指明键盘回调函数。

key　生成的 ASCII 字符。

x，y　当按下 key 键时，鼠标相对于窗口左上角的 x，y 坐标。

说明

本函数注册当前窗口的键盘回调函数。当用户在窗口中敲击键盘时，每一次产生 ASCII 字符的按键动作都将产生一次键盘回调。键盘回调函数中，key 变量是生成的 ASCII 字符，x 和 y 变量指定键按下时鼠标相对于窗口左上角的位置（以像素为单位）。

5. glutSpecialFunc：注册当前窗口的特定键回调函数

C 语言描述

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

参数

func　指明特定键回调函数。

key　指定按下的特定键。

x，y　当按下键时，鼠标相对于窗口左上角的 x，y 坐标。

说明

本函数设置当前窗口的特定键回调函数。当键盘中的功能键和方向键按下时，调用特定键回调函数。变量 key 是按下的特定键对应的 GLUT_KEY_常量。x 和 y 变量指定键按下时鼠标相对于窗口左上角的坐标。

对于变量 key，可以选择的常数如下：

GLUT_KEY_F1	F1 功能键
GLUT_KEY_F2	F2 功能键
GLUT_KEY_F3	F3 功能键
GLUT_KEY_F4	F4 功能键
GLUT_KEY_F5	F5 功能键
GLUT_KEY_F6	F6 功能键
GLUT_KEY_F7	F7 功能键

GLUT_KEY_F8	F8 功能键
GLUT_KEY_F9	F9 功能键
GLUT_KEY_F10	F10 功能键
GLUT_KEY_F11	F11 功能键
GLUT_KEY_F12	F12 功能键
GLUT_KEY_LEFT	左方向键
GLUT_KEY_UP	上方向键
GLUT_KEY_RIGHT	右方向键
GLUT_KEY_DOWN	下方向键
GLUT_KEY_PAGE_UP	PageUp 键
GLUT_KEY_PAGE_DOWN	PageDown 键
GLUT_KEY_HOME	Home 键
GLUT_KEY_END	End 键
GLUT_KEY_INSERT	Insert 键

6. glutTimerFunc：注册按一定时间间隔触发的定时器回调函数

C 语言描述

void glutTimerFunc(unsigned int msecs, void(*func)(int value));

参数

msecs　本次注册的定时器回调函数两次邻近触发之间的时间间隔（以毫秒为单位）。

func　指定定时器回调函数。

value　传送到定时器回调函数中的整型值。

说明

本函数注册按 msecs 毫秒时间间隔触发的定时器回调函数 func。传给定时器回调函数的 value 参数等于注册此回调函数时传给函数 glutTimerFunc 的参数 value 的值。多个按同一时间间隔或不同时间间隔触发的定时器回调函数可以同时注册。

定时器回调函数的触发间隔只是该函数调用前必须等待的最小时间，当定时器回调函数对应的触发时间期满后，GLUT 将尽可能快地调用该函数。

1.12 几何图形绘制

1. glutSolidSphere，glutWireSphere：绘制实心球体和线框球体

C 语言描述

void glutSolidSphere(GLdouble radius, GLint slices, GLint stacks);

void glutWireSphere(GLdouble radius, GLint slices, GLint stacks);

参数

radius　球体的半径。

slices　球体围绕 z 轴（相对于经度线）分割的数目。

stacks　球体沿着 z 轴（相对于纬度线）分割的数目。

说明

绘制中心在模型坐标原点、半径为 radius 的球体，球体围绕 z 轴分割为 slices 个数目，沿着 z 轴分割 stacks 个的数目。

2. glutSolidCube，glutWireCube：绘制实心立方体和线框立方体

C 语言描述

void glutSolidCube(GKdouble size);

void glutWireCube(GKdouble size);

参数

size　立方体的边长。

说明

这两个函数用于绘制以模型坐标原点为中心、边长为 size 的实心立方体和线框立方体。

3. glutSolidCone，glutWireCone：绘制实心圆锥体和线框圆锥体

C 语言描述

void glutSolidCone(GLdouble radius, GLdouble height, GLint slices, GLint stacks);

void glutWireCone(GLdouble radius, GLdouble height, GLint slices, GLint stacks);

参数

radius　圆锥体基底的半径。

height　圆锥体的高度。

slices　圆锥体围绕 z 轴的分割数。

stacks　圆锥体数沿着 z 轴的分割数。

说明

这两个函数分别绘制沿着 z 轴方向定位的实心圆锥体和线框圆锥体。圆锥体的基底定位于 z = 0 平面内，顶点 z = height，圆锥体围绕 z 轴分割为 slices 个数目，沿着 z 轴分割为 stacks 个数目。

4. glutSolidTorus，glutWireTorus：绘制实心圆环和线框圆环

C 语言描述

void glutSolidTorus(GLdouble innerRadius, GLdouble outerRadius, GLint nsides, GLint rings);

void glutWireTorus(GLdouble innerRadius, GLdouble outerRadius, GLint nsides, GLint rings);

参数

innerRadius　圆环的内半径。

outerRadius　圆环的外半径。

nsides　沿着圆环方向的分割数。

rings　圆环的环线数。

说明

这两个函数分别绘制中心位于模型坐标原点的实心圆环和线框圆环，圆环的轴线沿着 z 轴方向。

5. glutSolidDodecahedron，glutWireDodecahedron：绘制实心十二面体和线框十二面体

C 语言描述

void glutSolidDodecahedron(void);

void glutWireDodecahedron(void);

说明

这两个函数分别绘制中心位于模型坐标原点的实心 12 面体和线框 12 面体，12 面体的半径为 $\sqrt{3}$ 。

6. glutSolidOctahedron，glutWireOctahedron：绘制实心八面体和线框八面体

C 语言描述

void glutSolidOctahedron(void);

void glutWireOctahedron(void);

说明

这两个函数分别绘制中心位于模型坐标原点的实心八面体和线框八面体，八面体的半径为 1.0。

7. glutSolidTetrahedron，glutWireTetrahedron：绘制实心四面体和线框四面体

C 语言描述

void glutSolidTetrahedron(void);

void glutWireTetrahedron(void);

说明

这两个函数分别绘制中心位于模型坐标原点的实心四面体和线框四面体，四面体的半径为 $\sqrt{3}$ 。

8. glutSolidIcosahedron，glutWireIcosahedron：绘制实心二十面体和线框二十面体

C 语言描述

void glutSolidIcosahedron(void);

void glutWireIcosahedron(void);

说明

这两个函数分别绘制中心位于模型坐标原点的实心 20 面体和线框 20 面体，20 面体的半径为 1.0。

9. glutSolidTeapot，glutWireTeapot：绘制实心茶壶和线框茶壶

C 语言描述

void glutSolidTeapot(GLdouble size);

void glutWireTeapot(GLdouble size);

参数

size　茶壶的相对尺寸。

说明

这两个函数分别绘制实心茶壶和线框茶壶，茶壶的曲面法向量和纹理坐标也同时生成。茶壶是用 OpenGL 的求值器构建的。