前言

随着计算机多媒体技术、可视化技术及图形学技术的发展，我们可以使用计算机来精确地再现现实世界中的绚丽多彩的三维物体，并充分发挥自身的创造性思维，通过人机交互来模拟、改造现实世界，这就是目前最为时髦的虚拟现实技术。通过这种技术，建筑工程师可以直接设计出美观的楼房模型；军事指挥员可以模拟战场进行军事推演，网民可以足不出户游览故宫博物馆等名胜古迹等。而虚拟现实技术最重要的一部分内容就是三维图形编程。当前，三维图形编程工具中最为突出的是 SGI 公司的 OpenGL（Open Graphics Language，开放式的图形语言）,它已经成为一个工业标准的计算机三维图形软件开发接口，并广泛应用于游戏开发、建筑、产品设计、医学、地球科学、流体力学等领域。值得一提的是，虽然微软有自己的三维编程开发工具 DirectX，但它也提供 OpenGL 图形标准，因此，OpenGL 可以在微机中广泛应用。

目前，OpenGL 在国内外都掀起了热潮，但国内对这一领域介绍的资料并不是很多，特别是有志于在图形图像方面进行深入研究的读者朋友，常常苦于不掌握 OpenGL 编程接口技术，无法向纵深领域扩展。为了开启三维图形编程这扇神秘大门，本讲座在结合 OpenGL 有关理论知识的基础上，着重介绍 Visual C++6.0 开发环境中的编程实现，由于水平有限，本讲座可能无法面面俱到，存在一些疏漏，但相信它可以将开启"神秘大门"的钥匙交给读者朋友们。

第一章 概述

一、OpenGL 的特点及功能

OpenGL 是用于开发简捷的交互式二维和三维图形应用程序的最佳环境，任何高性能的图形应用程序，从 3D 动画、CAD 辅助设计到可视化访真，都可以利用 OpenGL 高质量、高性能的特点。OpenGL 自 1992 年出现以来，逐渐发展完善，已成为一个唯一开放的，独立于应用平台的图形标准，一个典型的 OpenGL 应用程序可以在任何平台上运行--只需要使用目标系统的 OpenGL 库重新编译一下。

OpenGL 非常接近硬件，是一个图形与硬件的接口，包括了 100 多个图形函数用来建立三维模型和进行三维实时交互。OpenGL 强有力的图形函数不要求开发人员把三维物体模型的数据写成固定的数据格式，也不要求开发人员编写矩阵变换、外部设备访问等函数，大大地简化了编写三维图形的程序。例如：

1) OpenGL 提供一系列的三维图形单元（图元）供开发者调用。

2) OpenGL 提供一系列的图形变换函数。

3) OpenGL 提供一系列的外部设备访问函数，使开发者可以方便地访问鼠标、键盘、空间球、数据手套等外部设备。

由于微软在 Windows 中包含了 OpenGL，所以 OpenGL 可以与 Visual C++紧密接合，简单快捷地实现有关计算和图形算法，并保证算法的正确性和可靠性。简单地说，OpenGL 具有建模、变换、色彩处理、光线处理、纹理影射、图像处理、动画及物体运动模糊等功能：

1、建模

OpenGL 图形库除了提供基本的点、线、多边形的绘制函数外，还提供了复杂的三维物体，如球、锥、多面体、茶壶以及复杂曲线和曲面（例如 Bezier、Nurbs 等曲线或曲面）的绘制函数。

2、变换

OpenGL 图形库的变换包括基本变换和投影变换。基本变换有平移、旋转、变比、镜像四种变换，投影变换有平行投影（又称正射投影）和透视投影两种变换。

3、颜色模式设置

OpenGL 颜色模式有两种，即 RGBA 模式和颜色索引（Color Index）。

4、光照和材质设置

OpenGL 光有辐射光（Emitted Light）、环境光（Ambient Light）、漫反射光（Diffuse Light）和镜面光（Specular Light）。材质是用光反射率来表示。客观世界中的物体最终反映到人眼的颜色是光的红绿蓝分量与材质红绿蓝分量的反射率相乘后形成的颜色。

5、纹理映射（Texture Mapping）

利用 OpenGL 纹理映射功能可以十分逼真地表达物体表面细节。

6、位图显示和图象增强

OpenGL 的图象功能除了基本的拷贝和像素读写外，还提供融合（Blending）、反走样（Antialiasing）和雾（fog）的特殊图象效果处理。以上三条可是被仿真物更具真实感，增强图形显示的效果。

7、双缓存动画（Double Buffering）

OpenGL 使用了前台缓存和后台缓存交替显示场景（Scene）技术，简而言之，后台缓存计算场景、生成画面，前台缓存显示后台缓存已画好的画面。

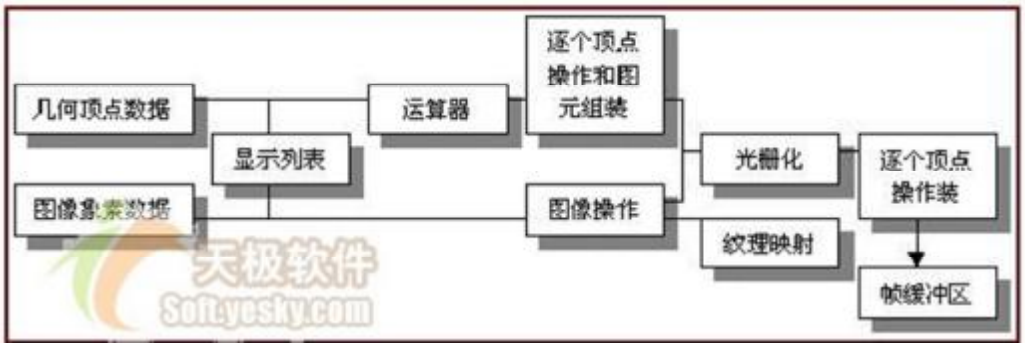
8、特殊效果

利用 OpenGL 还能实现深度暗示（Depth Cue）、运动模糊（Motion Blur）等特殊效果。运动模糊的绘图方式（motion-blured），模拟物体运动时人眼观察所感觉的动感现象。深度域效果（depth-of-effects），类似于照相机镜头效果，模型在聚焦点处清晰，反之则模糊。

这些三维物体绘图和特殊效果处理方式，说明 OpenGL 能够模拟比较复杂的三维物体或自然景观。

二、OpenGL 工作流程

OpenGL 的基本工作流程如下图：



图一、OpenGL 工作流程

如上图所示，几何顶点数据包括模型的顶点集、线集、多边形集，这些数据经过流程图的上部，包括运算器、逐个顶点操作等；图像数据包括象素集、影像集、位图集等，图像象素数据的处理方式与几何顶点数据的处理方式是不同的，但它们都经过光栅化、逐个片元（Fragment）处理直至把最后的光栅数据写入帧缓冲器。

在 OpenGL 中的所有数据包括几何顶点数据和象素数据都可以被存储在显示列表中或者立即可以得到处理。OpenGL 中，显示列表技术是一项重要的技术。

OpenGL 要求把所有的几何图形单元都用顶点来描述，这样运算器和逐个顶点计算操作都可以针对每个顶点进行计算和操作，然后进行光栅化形成图形碎片；对于象素数据，象素操作结果被存储在纹理组装用的内存中，再象几何顶点操作一样光栅化形成图形片元。整个流程操作的最后，图形片元都要进行一系列的逐个片元操作，这样最后的象素值送入帧缓冲器实现图形的显示。

根据这个流程，我们可以归纳出在 OpenGL 中进行主要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：

1）根据基本图形单元建立景物模型，并且对所建立的模型进行数学描述（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，其中的色彩根据应用要求来确定，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象素，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像素数据进行操作。

三、Windows 中 OpenGL 库函数及数据类型

（一）库函数

开发基于 OpenGL 的应用程序，必须先了解 OpenGL 的库函数。它采用 C 语言风格，提供大量的函数来进行图形的处理和显示。OpenGL 图形库一共有 100 多个函数，它们分别属于 OpenGL 的基本库、实用库、辅助库等不同的库。

1、核心库，包含的函数有 115 个，它们是最基本的函数，其前缀是 gl；这部分函数用于常规的、核心的图形处理，由 gl.dll 来负责解释执行。核心库中的函数可以进一步分为以下几类函数。

（1）绘制基本几何图元的函数。

glBegain()、glEnd()、glNormal*()、glVertex*()。

（2）矩阵操作、几何变换和投影变换的函数。

矩阵入栈函数 glPushMatrix()，矩阵出栈函数 glPopMatrix()，装载矩阵函数 glLoadMatrix()，矩阵相乘函数 glMultMatrix()，当前矩阵函数 glMatrixMode()和矩阵标准化函数 glLoadIdentity()，几何变换函数 glTranslate*()、glRotate*()和 glScale*()，投影变换函数 glOrtho()、glFrustum()和视口变换函数 glViewport()等等。

（3）颜色、光照和材质的函数。

如设置颜色模式函数 glColor*()、glIndex*()，设置光照效果的函数 glLight*() 、glLightModel*()和设置材质效果函数 glMaterial()等等。

（4）显示列表函数。

主要有创建、结束、生成、删除和调用显示列表的函数 glNewList()、 glEndList()、glGenLists()、glCallList()和 glDeleteLists()等。

（5）纹理映射函数。

主要有一维纹理函数 glTexImage1D()、二维纹理函数 glTexImage2D()、 设置纹理参数、纹理环境和纹理坐标的函数 glTexParameter*()、glTexEnv*()和 glTetCoord*()等。

（6）特殊效果函数。

融合函数 glBlendFunc()、反走样函数 glHint()和雾化效果 glFog*()。

（7）光栅化、象素操作函数。

像素位置 glRasterPos*()、线型宽度 glLineWidth()、多边形绘制模式 glPolygonMode()，读取象素 glReadPixel()、复制象素 glCopyPixel()等。

（8）选择与反馈函数。

主要有渲染模式 glRenderMode()、选择缓冲区 glSelectBuffer()和反馈缓冲区 glFeedbackBuffer()等。

（9）曲线与曲面的绘制函数。

生成曲线或曲面的函数 glMap*()、glMapGrid*()，求值器的函数 glEvalCoord*() glEvalMesh*()。

（10）状态设置与查询函数。主要有 glGet*()、glEnable()、glGetError()等。

2、实用库（OpenGL utility library，GLU），包含的函数功能更高一些，如绘制复杂的曲线曲面、高级坐标变换、多边形分割等，共有 43 个，前缀为 glu。Glu 函数通过调用核心库的函数，为开发者提供相对简单的用法，实现一些较为复杂的操作。此类函数由 glu.dll 来负责解释执行。主要包括了以下几种：

（1）辅助纹理贴图函数。

有 gluScaleImage() 、gluBuild1Dmipmaps()、gluBuild2Dmipmaps()等。

（2）坐标转换和投影变换函数。

定义投影方式函数 gluPerspective()、gluOrtho2D() 、gluLookAt()，拾取投影视景体函数 gluPickMatrix()，投影矩阵计算 gluProject()和 gluUnProject()等。

（3）多边形镶嵌工具。

有 gluNewTess()、 gluDeleteTess()、gluTessCallback()、gluBeginPolygon() gluTessVertex()、gluNextContour()、gluEndPolygon()等。

（4）二次曲面绘制工具。

主要有绘制球面、锥面、柱面、圆环面 gluNewQuadric()、gluSphere()、gluCylinder()、gluDisk()、gluPartialDisk()、gluDeleteQuadric()等等。

（5）非均匀有理 B 样条绘制工具。

主要用来定义和绘制 Nurbs 曲线和曲面，包括 gluNewNurbsRenderer()、gluNurbsCurve()、gluBeginSurface()、gluEndSurface()、gluBeginCurve()、gluNurbsProperty()等函数。

（6）错误反馈工具。

获取出错信息的字符串 gluErrorString()等。

3、OpenGL 辅助库（OpenGL auxiliarylibrary，GLAUX），包括简单的窗口管理、输入事件处理、某些复杂三维物体绘制等函数，共有 31 个，前缀为 aux。此类函数由 glaux.dll 来负责解释执行。辅助库函数主要包括以下几类。

（1）窗口初始化和退出函数。

auxInitDisplayMode()和 auxInitPosition()。

（2）窗口处理和时间输入函数。

auxReshapeFunc()、auxKeyFunc()和 auxMouseFunc()。

（3）颜色索引装入函数。

auxSetOneColor()。

（4）三维物体绘制函数。

包括了两种形式网状体和实心体，如绘制立方体 auxWireCube()和 auxSolidCube()。这里以网状体为例，长方体 auxWireBox()、环形圆纹面 auxWireTorus()、圆柱 auxWireCylinder()、二十面体 auxWireIcosahedron()、八面体 auxWireOctahedron()、四面体 auxWireTetrahedron()、十二面体 auxWireDodecahedron()、圆锥体 auxWireCone()和茶壶 auxWireTeapot()。绘制实心体只要将上述函数中的确"Wire"更换成"Solid"就可以了。

（5）其他。

背景过程管理函数 auxIdleFunc()；程序运行函数 auxMainLoop()。

4、OpenGL 工具库（penGL Utility Toolkit）

包含大约 30 多个函数，函数名前缀为 glut，此函数由 glut.dll 来负责解释执行。这部分函数主要包括：

（1）窗口操作函数

窗口初始化、窗口大小、窗口位置等函数 glutInit() glutInitDisplayMode()、glutInitWindowSize() glutInitWindowPosition()等。

（2）回调函数。

响应刷新消息、键盘消息、鼠标消息、定时器函数等，GlutDisplayFunc()、glutPostRedisplay()、glutReshapeFunc()、glutTimerFunc()、glutKeyboardFunc()、 glutMouseFunc()。

（3）创建复杂的三维物体。这些和 aux 库的函数功能相同。创建网状体和实心体。如 glutSolidSphere()、glutWireSphere()等。

（4）菜单函数

创建添加菜单的函数 GlutCreateMenu()、glutSetMenu()、glutAddMenuEntry()、glutAddSubMenu() 和 glutAttachMenu()。

（5）程序运行函数。

glutMainLoop()。

5、16 个 WGL 函数，专门用于 OpenGL 和 Windows 窗口系统的联接，其前缀为 wgl，主要用于创建和选择图形操作描述表（renderingcontexts）以及在窗口内任一位置显示字符位图。这类函数主要包括以下几类

（1）绘图上下文相关函数。

wglCreateContext()、wglDeleteContext()、wglGetCurrentContent()、wglGetCurrentDC() wglDeleteContent()等。

（2）文字和文本处理函数。

wglUseFontBitmaps()、wglUseFontOutlines()。

（3）覆盖层、地层和主平面层处理函数。

wglCopyContext()、wglCreateLayerPlane()、 wglDescribeLayerPlane()、wglReakizeLayerPlatte() 等。

（4）其他函数。

wglShareLists()、wglGetProcAddress()等。

6、另外，还有五个 Win32 函数用来处理像素格式（pixel formats）和双缓存。由于它们是对 Win32 系统的扩展，因此不能应用在其它 OpenGL 平台上。

（二）OpenGL 数据类型

与 C 语言相对应，OpenGL 中也有整数、字节、浮点数等数据类型，为了说明两者的对应关系，下表将 OpenGL 的数据类型与相应的 C 类型进行了对比：

前缀	数据类型	相应 C 语言类型	OpenGL 类型
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	long	GLint,GLsizei
f	32-bit floating-point	float	GLfloat,GLclampf
d	64-bit floating-point	double	GLdouble,GLclampd
ub	8-bit unsigned integer	unsigned char	GLubyte,GLboolean

us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned long	GLuint,GLenum,GLbitfield

表一、OpenGL 数据类型表

此外，OpenGL 也定义 GLvoid 类型，如果用 C 语言编写，可以用它替代 void 类型。

（三）OpenGL 库函数的命名规律

了解了 OpenGL 的数据类型，让我们再回过头来看看 OpenGL 库函数的命名规律。所有 OpenGL 函数采用了以下格式：

<库前缀><根命令><可选的参数个数><可选的参数类型>

库前缀有 gl、glu、aux、glut、wgl、glx 等等，分别表示该函数属于 OpenGL 某开发库等，从函数名后面中还可以看出需要多少个参数以及参数的类型。I 代表 int 型，f 代表 float 型，d 代表 double 型，u 代表无符号整型。注意，有的函数参数类型后缀前带有数字 2、3、4。2 代表二维，3 代表三维，4 代表 alpha 值（以后介绍）。有些 OpenGL 函数最后带一个字母 v，表示函数参数可用一个指针指向一个向量（或数组）来替代一系列单个参数值。下面两种格式都表示设置当前颜色为红色，二者等价。

glColor3f(1.0,0.0,0.0);等价于：

```
float color_array[]={1.0,0.0,0.0};
glColor3fv(color_array);
```

除了以上基本命名方式外，还有一种带"*"星号的表示方法，例如 glColor*()，它表示可以用函数的各种方式来设置当前颜色。同理，glVertex*v()表示用一个指针指向所有类型的向量来定义一系列顶点坐标值。

第二章 基本图元

任何复杂的三维模型都是由基本的几何图元：点、线段和多边形组成的，有了这些图元，就可以建立比较复杂的模型。因此这部分内容是学习 OpenGL 编程的基础。

一、基本图元的描述及定义

OpenGL 图元是抽象的几何概念，不是真实世界中的物体，因此须用相关的数学模型来描述。所有的图元都是由一系列有顺序的顶点集合来描述的。OpenGL 中绘制几何图元，必须使用 glBegain()和 glEnd()这一对函数，传递给 glBegain()函数的参数唯一确定了要绘制何种几何图元，同时，在该函数对中给出了几何图元的定义，函数 glEnd()标志顶点列表的结束。例如，下面的代码绘制了一个多边形：

```
glBegin(GL_POLYGON);
glVertex2f(0.0,0.0);
glVertex2f(0.0,3.0);
glVertex2f(3.0,3.0);
```

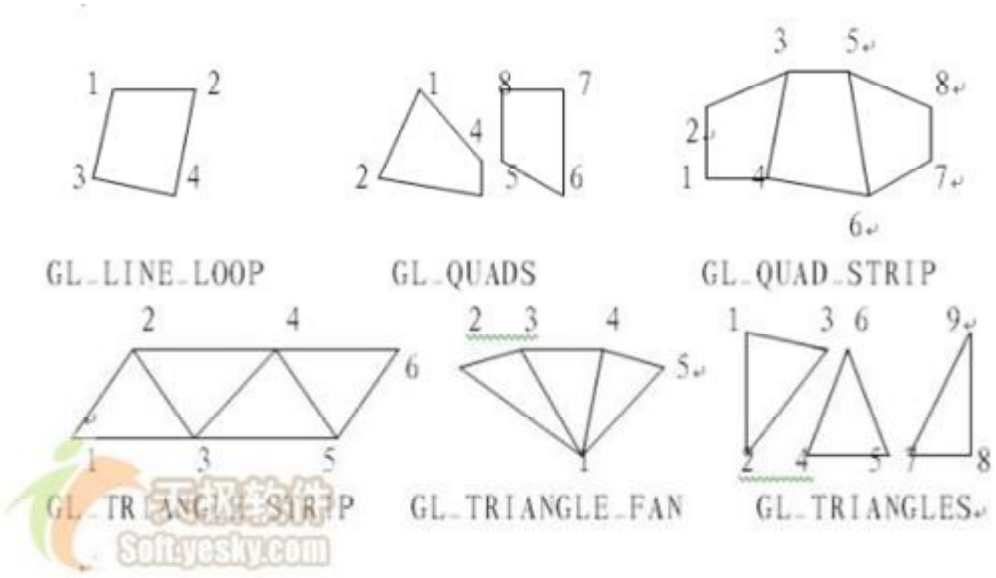
```
glVertex2f(4.0,1.5);
glVertex2f(3.0,0.0);
glEnd();
```

函数 glBegin(GLenum mode)标志描述一个几何图元的顶点列表的开始，其参数 mode 表示几何图元的描述类型，具体类型见表一：

类型	说明
GL_POINTS	单个顶点集
GL_LINES	多组双顶点线段
GL_POLYGON	单个简单填充凸多边形
GL_TRAINGLES	多组独立填充三角形
GL_QUADS	多组独立填充四边形
GL_LINE_STRIP	不闭合折线
GL_LINE_LOOP	闭合折线
GL_TRAINGLE_STRIP	线型连续填充三角形串
GL_TRAINGLE_FAN	扇形连续填充三角形串
GL_QUAD_STRIP	连续填充四边形串

表一、几何图元类型说明

部分几何图元的示意图：



图一、部分几何图元示意图

在 glBegin()和 glEnd()之间最重要的信息就是由函数 glVertex*()定义的顶点，必要时也可为每个顶点指定颜色（只对当前点或后续点有效）、法向、纹理坐标或其他，即调用相关的函数：

函数	函数意义
glColor*()	设置当前颜色

glIndex*()	设置当前颜色表
glNormal*()	设置法向坐标
glEvalCoord*()	产生坐标
glCallList(),glCallLists()	显示列表
glTexCoord*()	设置纹理坐标
glEdgeFlag*()	控制边界绘制
glMaterial*()	设置材质

表二、在 glBegin()和 glEnd()之间可调用的函数

需要指出的是：OpenGL 所定义的点、线、多边形等图元与一般数学定义不太一样，存在一定的差别。一种差别源于基于计算机计算的限制。OpenGL 中所有浮点计算精度有限，故点、线 、多边形的坐标值存在一定的误差。另一种差别源于位图显示的限制。以这种方式显示图形，最小的显示图元是一个象素，尽管每个象素宽度很小，但它们仍然比数学上所定义的点或线宽要大得多。当用 OpenGL 进行计算时，虽然是用一系列浮点值定义点串，但每个点仍然是用单个象素显示，只是近似拟合。

二、点(Point)

用浮点值表示的点称为顶点(Vertex)。所有顶点在 OpenGL 内部计算时都使用三维坐标（x,y,z）来处理，用二维坐标(x,y)定义的点在 OpenGL 中默认 z 值为 0。顶点坐标也可以用齐次坐标(x,y,z,w)来表示，如果 w 不为 0.0，这些齐次坐标表示的顶点即为三维空间点(x/w,y/w,z/w),一般来说，w 缺省为 1.0。

可以用 glVertex{234}{sifd}[V](TYPE cords)函数来定义一个顶点。例如：

```
glVertex2f(2.0f,3.0f);//二维坐标定义顶点；
```

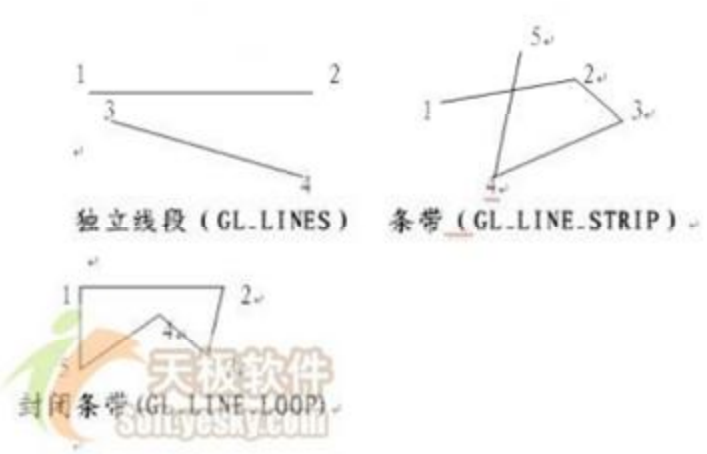
OpenGL 中定义的点可以有不同的尺寸，其函数形式为：

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

参数 size 设置点的宽度（以象素为单位）,必须大于 0.0，缺省时为 1.0。

三、线(Line)

在 OpenGL 中，线代表线段(Line Segment)，它由一系列顶点顺次连结而成。具体的讲，线有独立线段、条带、封闭条带三种，如图二所示：



图二、线段的三种连结方式

OpenGL 能指定线的宽度并绘制不同的虚点线，如点线、虚线等。相应的函数形式如下：

1、void glLineWidth(GLfloat width);

设置线宽（以象素为单位）。参数 width 必须大于 0.0，缺省时为 1.0。

2、void glLineStipple(GLint factor,GLushort pattern);

设置当前线为虚点模式。参数 pattern 是一系列的 16 位二进制数（0 或 1），它重复地赋给所指定的线，从低位开始，每一个二进制位代表一个象素， 1 表示用当前颜色绘制一个象素（或比例因子指定的个数），0 表示当前不绘制，只移动一个象素位（或比例因子指定的个数）。参数 factor 是个比例因子，它用来拉伸 pattern 中的元素，即重复绘制 1 或移动 0，比如，factor 为 2，则碰到 1 时就连续绘制 2 次，碰到 0 时连续移动 2 个单元。factor 的大小范围限制在 1 到 255 之间。

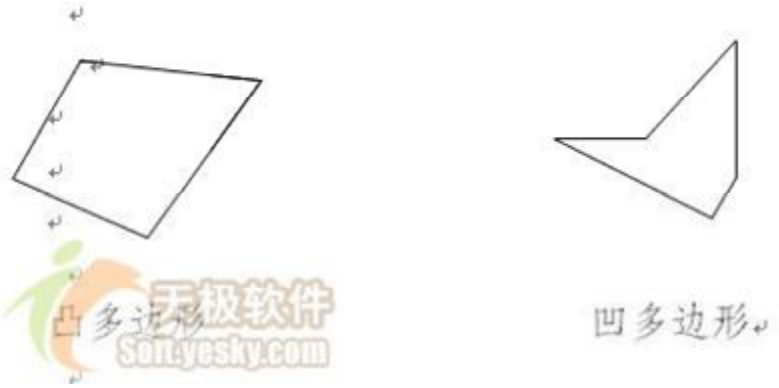
在绘制虚点线之前必须先启动虚点模式，即调用函数 glEnable(GL_LINE_STIPPLE);结束时，调用 glDisable(GL_LINE_STIPPLE)关闭。下面代码绘制了一个点线：

```
void line2i(GLint x1,GLint y1,GLint x2,GLint y2)
{
    glBegin(GL_LINES);
    glVertex2f(x1,y1);
    glVertex2f(x2,y2);
    glEnd();
}
glLineStipple (1, 0x1C47);   /* 虚点线 */
glEnable(GL_LINE_STIPPLE);
glColor3f(0.0,1.0,0.0);
line2i (450 , 250 , 600 , 250 );
```

四、多边形(Polygon)

（一）凸、凹多边形。

OpenGL 定义的多边形是由一系列线段依次连结而成的封闭区域，多边形可以是平面多边形，即所有顶点在一个平面上，也可以是空间多边形。OpenGL 规定多边形中的线段不能交叉，区域内不能有空洞，也即多边形必须是凸多边形（指多边形任意非相邻的两点的连线位于多边形的内部），不能是凹多边形，否则不能被 OpenGL 函数接受。凸多边形和凹多边形见图三。



图三、凸凹多边形

（二）边界标志问题。

实际应用中，往往需要绘制一些凹多边形，通常解决的办法是对它们进行分割，用多个三角形来替代。显然，绘制这些三角形时，有些边不应该进行绘制，否则，多边形内部就会出现多余的线框。OpenGL 提供的解决办法是通过设置边标志命令 glEdgeFlag（）来控制某些边产生绘制，而另外一些边不产生绘制，这也称为边界标志线或非边界线。这个命令的定义如下；

```
void glEdgeFlag(GLboolean flag);
void glEdgeFlag(PGLboolean pflag);
```

（三）多边形绘制模式。

多边形的绘制模式包含有：全填充式、轮廓点式、轮廓线式、图案填充式及指定正反面等。下面分别介绍相应的 OpenGL 函数形式。

1）多边形模式设置。其函数为：

```
void glPolygonMode(GLenum face,GLenum mode);
```

参数 face 为 GL_FRONT、GL_BACK 或 GL_FRONT_AND_BACK；参数 mode 为 GL_POINT、GL_LINE 或 GL_FILL，分别表示绘制轮廓点式多边形、轮廓线式多边形或全填充式多边形。在 OpenGL 中，多边形分为正面和反面，对这两个面都可以进行操作，在缺省状况下，OpenGL 对多边形正反面是以相同的方式绘制的，要改变绘制状态，必须调用 PolygonMode（）函数，

2）设置图案填充式多边形。其函数为：

```
void glPolygonStipple(const GLubyte *mask);
```

参数 mask 是一个指向 32x32 位图的指针。与虚点线绘制的道理一样，某位为 1 时绘制，为 0 时什么也不绘。注意，在调用这个函数前，必须先启动 glEnable(GL_POLYGON_STIPPLE)；不用时用 glDisable(GL_POLYGON_STIPPLE)关闭。下面举出一个多边形扩展绘制实例：

```
void CALLBACK display(void)
{
    /* 填充模式定义 （32x32） */
    GLubyte pattern[]= {
        0x00, 0x01, 0x80, 0x00,
        0x00, 0x03, 0xc0, 0x00,
            0x00, 0x07, 0xe0, 0x00,
            0x00, 0x0f, 0xf0, 0x00,
            0x00, 0x1f, 0xf8, 0x00,
            0x00, 0x3f, 0xfc, 0x00,
            0x00, 0x7f, 0xfe, 0x00,
            0x00, 0xff, 0xff, 0x00,
            0x01, 0xff, 0xff, 0x80,
            0x03, 0xff, 0xff, 0xc0,
            0x07, 0xff, 0xff, 0xe0,
            0x0f, 0xff, 0xff, 0xf0,
            0x1f, 0xff, 0xff, 0xf8,
            0x3f, 0xff, 0xff, 0xfc,
            0x7f, 0xff, 0xff, 0xfe,
            0xff, 0xff, 0xff, 0xff,
            0xff, 0xff, 0xff, 0xff,
            0x7f, 0xff, 0xff, 0xfe,
            0x3f, 0xff, 0xff, 0xfc,
            0x1f, 0xff, 0xff, 0xf8,
            0x0f, 0xff, 0xff, 0xf0,
            0x07, 0xff, 0xff, 0xe0,
            0x03, 0xff, 0xff, 0xc0,
            0x01, 0xff, 0xff, 0x80,
            0x00, 0xff, 0xff, 0x00,
            0x00, 0x7f, 0xfe, 0x00,
            0x00, 0x3f, 0xfc, 0x00,
            0x00, 0x1f, 0xf8, 0x00,
            0x00, 0x0f, 0xf0, 0x00,
            0x00, 0x07, 0xe0, 0x00,
            0x00, 0x03, 0xc0, 0x00,
```

```
            0x00, 0x01, 0x80, 0x00
        };
        glClear (GL_COLOR_BUFFER_BIT);
      /* 绘制一个指定图案填充的三角形 */
        glColor3f(0.9,0.86,0.4);
        glPolygonStipple (pattern);
        glBegin(GL_TRIANGLES);
            glVertex2i(310,310);
            glVertex2i(220,80);
            glVertex2i(405,80);
        glEnd();
        glDisable (GL_POLYGON_STIPPLE);
        glFlush ();
    }
```

3）指定多边形的正反面。

其函数为：

```
void glFrontFace(GLenum mode);
```

在正常情况下，OpenGL 中的多边形的正面和反面是由绘制的多边形的顶点顺序决定的，逆时针绘制的面是多边形的正面，但是，在 OpenGL 中使用该函数可以自定义多边形的正面。该函数的参数 mode 指定了正面的方向。它可以是 CL_CCW 和 CL_CW，分别指定逆时针和顺时针方向为多边形的正方向。

四、法向量的计算及指定

法向量是几何图元的重要属性之一。几何对象的法向量是垂直与曲面切面的单位向量，它定义了几何对象的空间方向，特别定义了它相对于光源的方向，决定了在该点上可接受多少光照。

OpenGL 本身没有提供计算法向量的函数（计算法向量的任务由程序员自己去完成），但它提供了赋予当前顶点法向的函数。

（一）平面法向的计算方法。

在一个平面内，有两条相交的线段，假设其中一条为矢量 W，另一条为矢量 V，平面法向为 N，则平面法向就等于两个矢量的叉积（遵循右手定则），即 N＝WxV。例如：一个三角形平面三个顶点分别为 P0、P1、P2，相应两个向量为 W、V，则三角平面法向的计算方式如下列代码所示：

```
void getNormal(GLfloat gx[3],GLfloat gy[3],
GLfloat gz[3],GLfloat *ddnv)
{
   GLfloat w0,w1,w2,v0,v1,v2,nr,nx,ny,nz;
   w0=gx[0]-gx[1]; w1=gy[0]-gy[1];w2=gz[0]-gz[1];
```

```
  v0=gx[2]-gx[1]; v1=gy[2]-gy[1];v2=gz[2]-gz[1];
  nx=(w1*v2-w2*v1);ny=(w2*v0-w0*v2);nz=(w0*v1-w1*v0);
  nr=sqrt(nx*nx+ny*ny+nz*nz); //向量单位化。
  ddnv[0]=nx/nr; ddnv[1]=ny/nr;ddnv[2]=nz/nr;
}
```

以上函数的输出参数为指针 ddnv，它指向法向的三个分量，并且程序中已经将法向单位化（或归一化）了。

（二）曲面法向量的计算。

对于曲面各顶点的法向计算有很多种，如根据函数表达式求偏导的方法等。但是，在大多数情况，OpenGL 中的多边形并不是由曲面方程建立起来的，而是由模型数组构成，这时候求取法向量的办法是将曲面细分成多个小多边形，然后选取小多边形上相邻的三个点 v1、v2、v3（当然三个点不能在同一直线上），按照平面法向量的求取方法就可以了。

（三）法向量的定义。

OpenGL 法向量定义函数为：

```
void glNormal3{bsifd}(TYPE nx,TYPE ny,TYPE nz);
void glNormal3{bsifd}v(const TYPE *v);
```

非向量形式定义法向采用第一种方式，即在函数中分别给出法向三个分量值 nx、ny 和 nz；向量形式定义采用第二种，即将 v 设置为一个指向拥有三个元素的指针，例如 v[3]={nx,ny,nz}。

五、显示列表

（一）定义显示列表。

前面所举出的例子都是瞬时给出函数命令，OpenGL 瞬时执行相应的命令，这种绘图方式叫做立即或瞬时方式（immediate mode）。OpenGL 显示列表（Display List）是由一组预先存储起来的留待以后调用的 OpenGL 函数语句组成的，当调用显示列表时就依次执行表中所列出的函数语句。显示列表可以用在以下场合：

1）矩阵操作

大部分矩阵操作需要 OpenGL 计算逆矩阵，矩阵及其逆矩阵都可以保存在显示列表中。

2）光栅位图和图像

程序定义的光栅数据不一定是适合硬件处理的理想格式。当编译组织一个显示列表时，OpenGL 可能把数据转换成硬件能够接受的数据，这可以有效地提高画位图的速度。

3）光、材质和光照模型

当用一个比较复杂的光照环境绘制场景时，因为材质计算可能比较慢。若把材质定义放在显示列表中，则每次改换材质时就不必重新计算了，因此能更快地绘制光照场景。

4）纹理

因为硬件的纹理格式可能与 OpenGL 格式不一致，若把纹理定义放在显示列表中，则在编译显示列表时就能对格式进行转换，而不是在执行中进行，这样就能大大提高效率。

5）多边形的图案填充模式，即可将定义的图案放在显示列表中。

OpenGL 提供类似于绘制图元的结构即类似于 glBegin()与 glEnd()的形式创建显示列表，其相应的函数为：

```
void glNewList(GLuint list,GLenum mode);
void glEndList(void);
```

glNewList（）函数说明一个显示列表的开始，其后的 OpenGL 函数存入显示列表中，直至调用结束表的函数 glEndList(void)。glNewList（）函数中的参数 list 是一个正整数，它标志唯一的显示列表；参数 mode 的可能值有 GL_COMPILE 和 GL_COMPILE_AND_EXECUTE；若要使列表中函数语句只存入而不执行，则用 GL_COMPILE；若要使列表中的函数语句存入表中且按瞬时方式执行一次，则用 GL_COMPILE_AND_EXECUTE。

注意：并不是所有的 OpenGL 函数都可以在显示列表中存储且通过显示列表执行。一般来说，用于传递参数或返回数值的函数语句不能存入显示列表，因为这张表有可能在参数的作用域之外被调用；如果在定义显示列表时调用了这样的函数，则它们将按瞬时方式执行并且不保存在显示列表中，有时在调用执行显示列表函数时会产生错误。以下列出的是不能存入显示列表的 OpenGL 函数：

```
glDeleteLists()          glIsEnable()
glFeedbackBuffer()       glIsList()
glFinish()               glPixelStore()
glGenLists()             glRenderMode()
glGet*()                 glSelectBuffer()
```

在建立显示列表以后就可以调用执行显示列表的函数来执行它，并且允许在程序中多次执行同一显示列表，同时也可以与其它函数的瞬时方式混合使用。显示列表执行的函数形式如下：

```
void glCallList(GLuint list);
```

参数 list 指定被执行的显示列表。显示列表中的函数语句按它们被存放的顺序依次执行；若 list 没有定义，则不会产生任何事情。

（二）管理显示列表

在实际应用中，一般调用函数 glGenList()来创建多个显示列表，这样可以避免意外删除，产生一个没有用过的显示列表。此外，在管理显示列表的过程中，还可调用函数 glDeleteLists()来删除一个或一个范围内的显示列表。

1）GLuint glGenList(GLsizei range)

该函数分配 range 个相邻的未被占用的显示列表索引。这个函数返回的是一个正整数索引值，它是一组连续空索引的第一个值。返回的索引都标志为空且已被占用，以后再调用这个函数时不再返回这些索引。若申请索引的指定数目不能满足或 range 为 0 则函数返回 0。

2）GLboolean glIsList(GLuint list)

该函数询问显示列表是否已被占用的情况，若索引 list 已被占用，则函数返回 TURE；反之，返回 FAULSE。

3）void glDeleteLists(GLuint list,GLsizei range)

该函数删除一组连续的显示列表，即从参数 list 所指示的显示列表开始，删除 range 个显示列表，并且删除后的这些索引重新有效。

（三）多级显示列表

多级显示列表的建立就是在一个显示列表中调用另一个显示列表，也就是说，在函数 glNewList()与 glEndList()之间调用 glCallList()。多级显示列表对于构造由多个元件组成的物体十分有用，尤其是某些元件需要重复使用的情况。但为了避免无穷递归，显示列表的嵌套深度最大为 64（也许更高些，这依赖于不同的 OpenGL 实现），当然也可调用函数 glGetIntegerv()来获得这个最大嵌套深度值。OpenGL 也允许用一个显示列表包含几个低级的显示列表来模拟建立一个可编辑的显示列表。

下面的一段代码使用了列表嵌套来显示一个三角形：

```
glNewList(1,GL_COMPILE);
glVertex3fv(v1);
glEndList();
glNewList(2,GL_COMPILE);
glVertex3fv(v2);
glEndList();

glNewList(3,GL_COMPILE);
glVertex3fv(v3);
glEndList();

glNewList(4,GL_COMPILE);
glBegin(GL_POLYGON);
```

```
glCallList(1);
glCallList(2);
glCallList(3);
glEnd();
glEndList();
```

第三章 坐标变换

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

一、OpenGL 中的三维物体的显示

（一）坐标系统

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

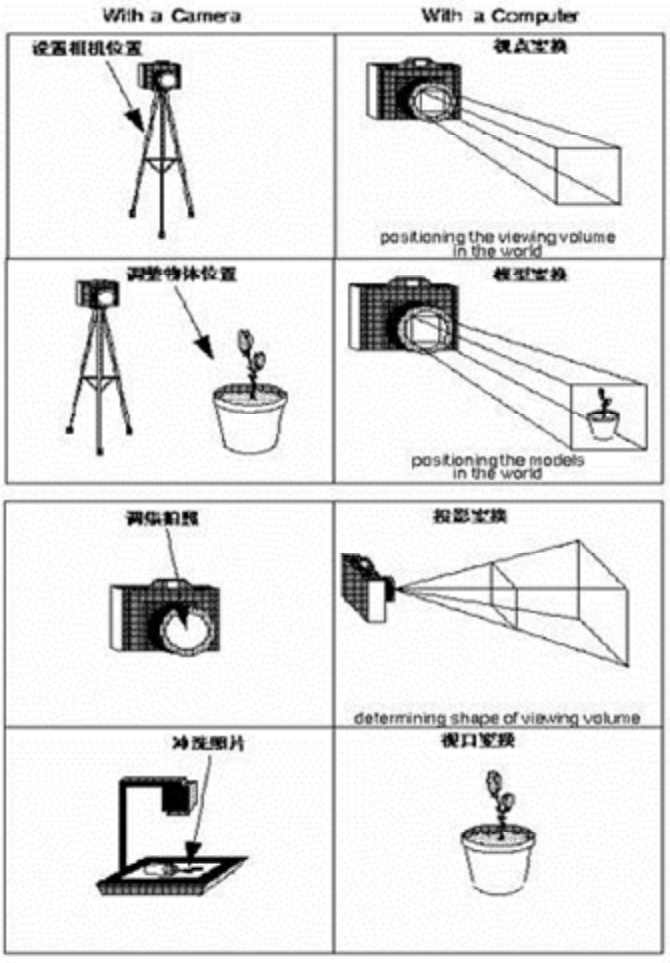
为了使被显示的三维物体数字化，要在被显示的物体所在的空间中定义一个坐标系。这个坐标系的长度单位和坐标轴的方向要适合对被显示物体的描述，这个坐标系称为世界坐标系。世界坐标系是始终固定不变的。

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

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

（二）三维物体的相机模拟

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



图一、相机模拟 OpenGL 中的各种坐标变换

从三维物体到二维图象，就如同用相机拍照一样，通常都要经历以下几个步骤：

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

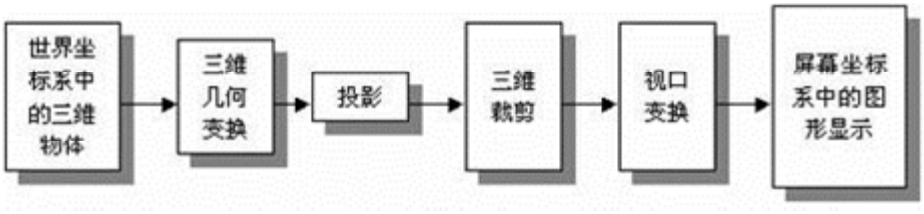
2、将三维物体放在场景中的适当位置，它相当于 OpenGL 中的模型变换（Modeling Transformation），即对模型进行旋转、平移和缩放。

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

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

屏幕上显示场景的范围和尺寸。

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



图二、三维物体的显示过程

二、OpenGL 中的几种变换

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

（一）视点变换

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

执行视点变换的命令和执行模型转换的命令是相同的，想一想，在用相机拍摄物体时，我们可以保持物体的位置不动，而将相机移离物体，这就相当于视点变换；另外，我们也可以保持相机的固定位置，将物体移离相机，这就相当于模型转换。这样，在 OpenGL 中，以逆时针旋转物体就相当于以顺时针旋转相机。因此，我们必须把视点转换和模型转换结合在一起考虑，而对这两种转换单独进行考虑是毫无意义的。除了用模型转换命令执行视点转换之外，OpenGL 实用库还提供了 gluLookAt()函数，该函数有三个变量，分别定义了视点的位置、相机瞄准方向的参考点以及相机的向上方向。该函数的原型为：

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

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

通常，视点转换操作在模型转换操作之前发出，以便模型转换先对物体发生作用。场景中物体的顶点

经过模型转换之后移动到所希望的位置，然后再对场景进行视点定位等操作。模型转换和视点转换共同构成模型视景矩阵。

（二）模型变换

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

1、模型平移

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

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

2、模型旋转

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

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

3、模型缩放

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

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

（三）投影变换

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

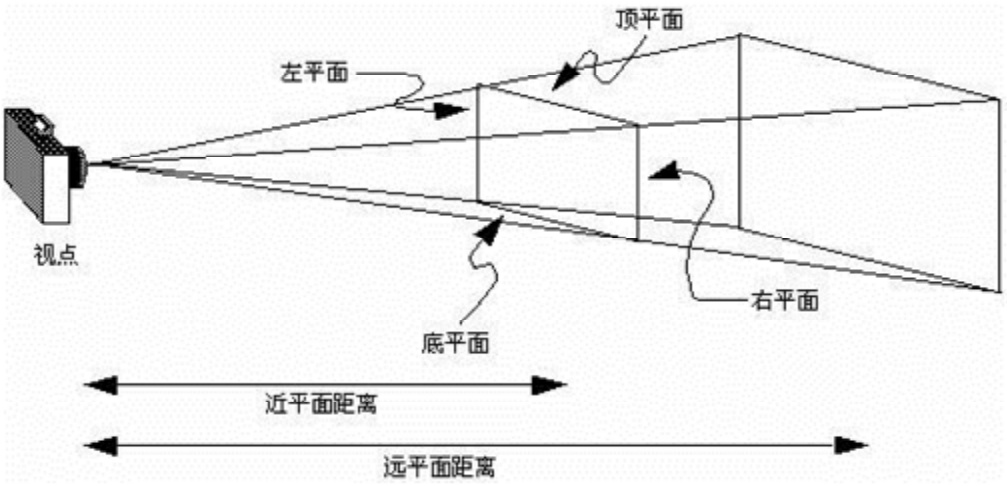
事实上，投影变换的目的就是定义一个视景体，使得视景体外多余的部分裁剪掉，最终进入图像的只是视景体内的有关部分。投影包括透视投影（Perspective Projection）和正视投影（Orthographic Projection）两种。

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

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

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

它创建一个透视视景体。其操作是创建一个透视投影矩阵，并且用这个矩阵乘以当前矩阵。这个函数的参数只定义近裁剪平面的左下角点和右上角点的三维空间坐标，即（left，bottom，-near）和（right，top，-near）；最后一个参数 far 是远裁剪平面的 Z 负值，其左下角点和右上角点空间坐标由函数根据透视投影原理自动生成。near 和 far 表示离视点的远近，它们总为正值。该函数形成的视景体如图三所示。

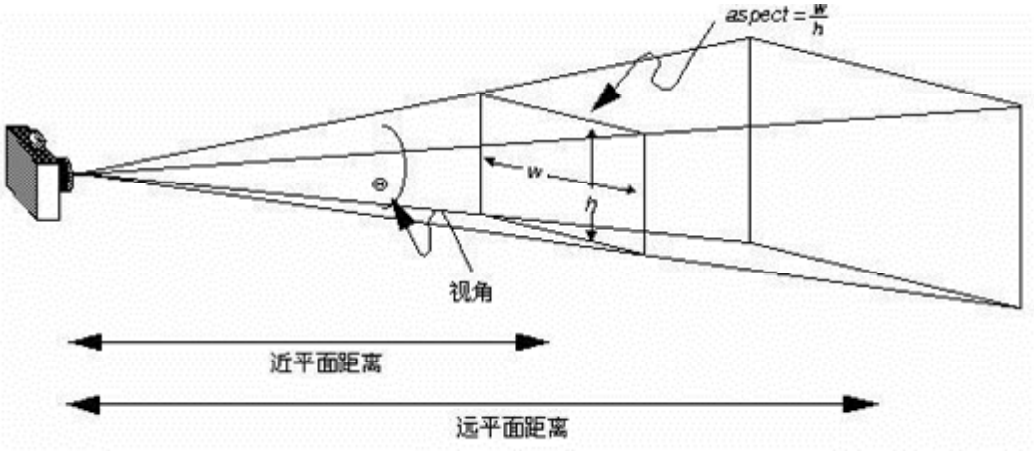


图三、透视投影视景体

另一个透视函数是：

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

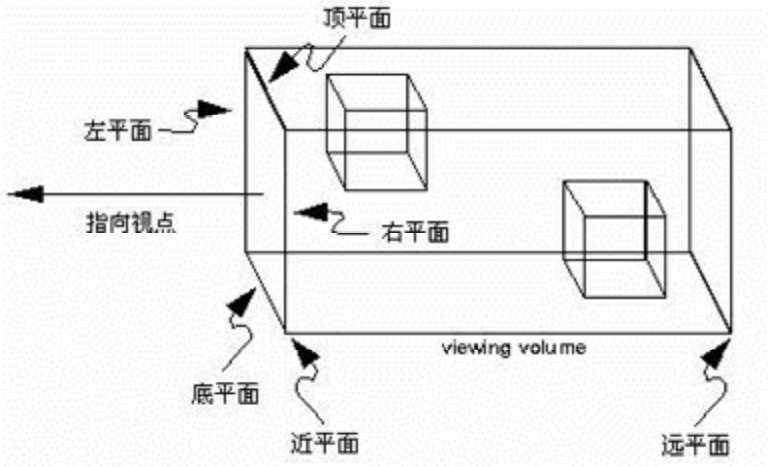
它也创建一个对称透视视景体，但它的参数定义于前面的不同，参数 fovy 定义视野在 X-Z 平面的角度，范围是[0.0, 180.0]；参数 aspect 是投影平面宽度与高度的比率；参数 zNear 和 Far 分别是远近裁剪面沿 Z 负轴到视点的距离，它们总为正值。



图四、透视投影视景体

以上两个函数缺省时，视点都在原点，视线沿 Z 轴指向负方向。

正射投影，又叫平行投影。这种投影的视景体是一个矩形的平行管道，也就是一个长方体，如图五所示。正射投影的最大一个特点是无论物体距离相机多远，投影后的物体大小尺寸不变。这种投影通常用在建筑蓝图绘制和计算机辅助设计等方面，这些行业要求投影后的物体尺寸及相互间的角度不变，以便施工或制造时物体比例大小正确。



图五、正射投影视景体

OpenGL 正射投影函数也有两个，一个函数是：

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

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

另一个函数是：

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

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

（四）视口变换。

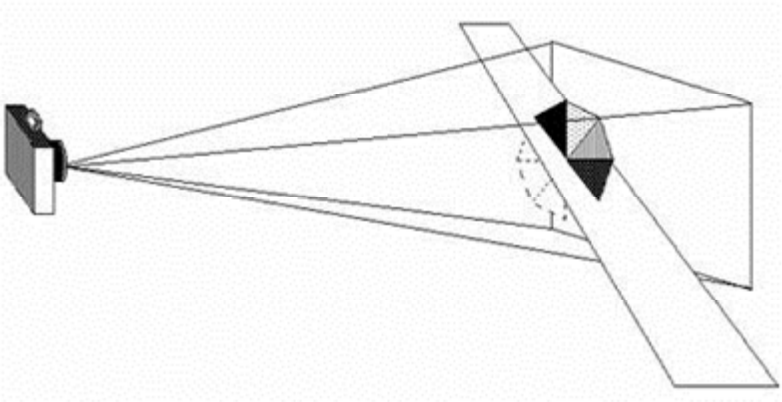
视口变换就是将视景体内投影的物体显示在二维的视口平面上。运用相机模拟方式，我们很容易理解视口变换就是类似于照片的放大与缩小。在计算机图形学中，它的定义是将经过几何变换、投影变换和裁剪变换后的物体显示于屏幕窗口内指定的区域内，这个区域通常为矩形，称为视口。OpenGL 中相关函数是：

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

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

（5）裁剪变换

在 OpenGL 中，除了视景体定义的六个裁剪平面（上、下、左、右、前、后）外，用户还可自己再定义一个或多个附加裁剪平面，以去掉场景中无关的目标，如图六所示。



图六、附加裁剪平面

附加平面裁剪函数为：

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

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

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

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

```
#include "glos.h"
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);
void CALLBACK display(void)
{
      GLdouble eqn[4] = {1.0, 0.0, 0.0, 0.0};
      glClear(GL_COLOR_BUFFER_BIT);
      glColor3f (1.0, 0.0, 1.0);
      glPushMatrix();
      glTranslatef (0.0, 0.0, -5.0);
      /* clip the left part of wire_sphere : x<0 */
      glClipPlane (GL_CLIP_PLANE0, eqn);
      glEnable (GL_CLIP_PLANE0);
      glRotatef (-90.0, 1.0, 0.0, 0.0);
      auxWireSphere(1.0);
      glPopMatrix();
      glFlush();
}
void myinit (void)
{
      glShadeModel (GL_FLAT);
}
void CALLBACK myReshape(GLsizei w, GLsizei h)
{
      glViewport(0, 0, w, h);
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      gluPerspective(60.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
      glMatrixMode(GL_MODELVIEW);
}
void main(void)
{
      auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
      auxInitPosition (0, 0, 500, 500);
      auxInitWindow ("Arbitrary Clipping Planes");
      myinit ();
      auxReshapeFunc (myReshape);
      auxMainLoop(display);
}
```

（六）矩阵栈的操作

在讲述矩阵栈之前，首先介绍两个基本 OpenGL 矩阵操作函数：

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

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

M = | m0 m4 m8 m12 |
| m1 m5 m9 m13 |
| m2 m6 m10 m14 |
| m3 m7 m11 M15 |

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

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

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

·void glPushMatrix(void);

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

·void glPopMatrix(void);

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

第四章 曲线和曲面

计算机图形学中，所有的光滑曲线、曲面都采用线段或三角形逼近来模拟，但为了精确地表现曲线，通常需要成千上万个线段或三角形来逼近，这种方法对于计算机的硬件资源有相当高的要求。然而，许多有用的曲线、曲面在数学上只需要用少数几个参数（如控制点等）来描述。这种方法所需要的存储空间比线段、三角形逼近的方法来所需要的空间要小得多，并且控制点方法描述的曲线、曲面比线段、三角形逼近的曲线、曲面更精确。

为了说明如何在 OpenGL 中绘制复杂曲线和曲面，我们对上述两类比方法都进行了介绍。下面我们先来介绍有关基础知识，然后再看是如何实现的吧。

一、曲线的绘制

OpenGL 通过一种求值器的机制来产生曲线和曲面，该机制非常灵活，可以生成任意角度的多项式曲线，并可以将其他类型的多边形曲线和曲面转换成贝塞尔曲线和曲面。这些求值器能在任何度的曲线及曲面上计算指定数目的点。随后，OpenGL 利用曲线和曲面上的点生成标准 OpenGL 图元，例如与曲线或曲面近似的线段和多边形。由于可让 OpenGL 计算在曲线上所需的任意数量的点，因此可以达到应用所需的精度。

对于曲线，OpenGL 中使用 glMap1*（）函数来创建一维求值器，该函数原型为：

```
void glMap1{fd}(GLenum target,TYPE u1,TYPE u2,GLint stride, GLint order,const TYPE *points);
```

函数的第一个参数 target 指出控制顶点的意义以及在参数 points 中需要提供多少值，具体值见表一所示。参数 points 指针可以指向控制点集、RGBA 颜色值或纹理坐标串等。例如若 target 是 GL_MAP1_COLOR_4，则就能在 RGBA 四维空间中生成一条带有颜色信息的曲线，这在数据场可视化中应用极广。参数 u1 和 u2，指明变量 U 的范围，U 一般从 0 变化到 1。参数 stride 是跨度，表示在每块存储区内浮点数或双精度数的个数，即两个控制点间的偏移量，比如上例中的控制点集 ctrpoint[4][3]的跨度就为 3，即单个控制点的坐标元素个数。函数参数 order 是次数加 1，叫阶数，与控制点数一致。

参数	意义
GL_MAP1_VERTEX_3	x,y,z 顶点坐标
GL_MAP1_VERTEX_4	x,y,z,w 顶点坐标
GL_MAP1_INDEX	颜色表
GL_MAP1_COLOR_4	R,G,B,A
GL_MAP1_NORMAL	法向量
GL_MAP1_TEXTURE_COORD_1	s 纹理坐标
GL_MAP1_TEXTURE_COORD_2	s,t 纹理坐标
GL_MAP1_TEXTURE_COORD_3	s,t,r 纹理坐标
GL_MAP1_TEXTURE_COORD_4	s,t,r,q 纹理坐标

表一、参数 target 的取值表

使用求值器定义曲线后，必须要启动求值器，才能进行下一步的绘制工作。启动函数仍是 glEnable()，其中参数与 glMap1*()的第一个参数一致。同样，关闭函数为 glDisable()，参数也一样。

一旦启动一个或多个求值器，我们就可以构造近似曲线了。最简单的方法是通过调用计算坐标函数 glEvalcoord1*()替换所有对函数 glVertex*()的调用。与 glVertex*()使用二维、三维和四维坐标不同，glEvalcoord1*()将 u 值传给所有已启动的求值器，然后由这些已启动的求值器生成坐标、法向量、颜色或纹理坐标。OpenGL 曲线坐标计算的函数形式如下：

```
void glEvalCoord1{fd}[v](TYPE u);
```

该函数产生曲线坐标值并绘制。参数 u 是定义域内的值，这个函数调用一次只产生一个坐标。在使用

glEvalCoord1*()计算坐标，因为 u 可取定义域内的任意值，所以由此计算出的坐标值也是任意的。

使用 glEvalCoord1*()函数的优点是，可以对 U 使用任意值，然而，如果想对 u 使用 N 个不同的值，就必须对 glEvalCoord1*()函数执行 N 次调用，为此，OpenGL 提供了等间隔值取值法，即先调用 glMapGrid1*()定义一个间隔相等的一维网格，然后再用 glEvalMesh1()通过一次函数执行，将求值器应用在网格上，计算相应的坐标值。下面详细解释这两个函数：

1、void glMapGrid1{fd}(GLint n,TYPE u1,TYPE u2);

定义一个网格，从 u1 到 u2 分为 n 步，它们是等间隔的。实际上，这个函数定义的是参数空间网格。

2、void glEvalMesh1(GLenum mode,GLint p1,GLint p2);

计算并绘制坐标点。参数 mode 可以是 GL_POINT 或 GL_LINE，即沿曲线绘制点或沿曲线绘制相连的线段。这个函数的调用效果同在 p1 和 p2 之间的每一步给出一个 glEvalCoord1()的效果一样。从编程角度来说，除了当 i=0 或 i=n，它准确以 u1 或 u2 作为参数调用 glEvalCoord1()之外，它等价于一下代码：

```
glBegin(GL_POINT);   /* glBegin(GL_LINE_STRIP); */
    for(i=p1;i<=p2;i++)
        glEvalCoord1(u1+i*(u2-u1)/n);
    glEnd();
```

为了进一步说明 OpenGL 中曲线的绘制方法。下面我们来看一个简单的例子，这是用四个控制顶点来画一条三次 Bezier 曲线。程序如下（注：这是本讲座中提供的第一个完整的 OpenGL 实例代码，如果读者朋友对整个程序结构有些迷惑的话，也不要紧，慢慢地往下看，先有一个感官上的印象，主要是掌握如何实现曲线绘制这一部分。关于 OpenGL 的程序整体结构实现，笔者将在第五讲中专门阐述）：

```
#include "glos.h"
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);
GLfloat ctrlpoints[4][3] = {
      { -4.0, -4.0, 0.0 }, { -2.0, 4.0, 0.0 },
      { 2.0,    -4.0, 0.0 }, { 4.0,    4.0, 0.0 }
};
void myinit(void)
{
      glClearColor(0.0, 0.0, 0.0, 1.0);
glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4,
&ctrlpoints[0][0]);
      glEnable(GL_MAP1_VERTEX_3);
      glShadeModel(GL_FLAT);
}
void CALLBACK display(void)
{
      int i;
      glClear(GL_COLOR_BUFFER_BIT);
      glColor3f(1.0, 1.0, 1.0);
      glBegin(GL_LINE_STRIP);
      for (i = 0; i <= 30; i++)
               glEvalCoord1f((GLfloat) i/30.0);
      glEnd();
      /*  显示控制点  */
      glPointSize(5.0);
      glColor3f(1.0, 1.0, 0.0);
      glBegin(GL_POINTS);
      for (i = 0; i < 4; i++)
            glVertex3fv(&ctrlpoints[i][0]);
      glEnd();
   glFlush();
}
void CALLBACK myReshape(GLsizei w, GLsizei h)
{
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
glOrtho(-5.0, 5.0, -5.0*(GLfloat)h/(GLfloat)w,
```

```
5.0*(GLfloat)h/(GLfloat)w, -5.0, 5.0);
   else
glOrtho(-5.0*(GLfloat)w/(GLfloat)h, 5.0*(GLfloat)w/(GLfloat)h, -5.0, 5.0, -5.0, 5.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
      }
```

二、曲面构造

曲面的绘制方法基本上与曲线的绘制方法是相同的，所不同的是曲面使用二维求值器，并且控制点连接起来形成一个网格。

对于曲面，求值器除了使用二个参数 U、V 之外，其余与一维求值器基本相同。顶点坐标 、颜色、法线矢量和纹理坐标都对应于曲面而不是曲线。在 OpenGL 中定义二维求值器的函数是：

```
void glMap2{fd}(GLenum target,TYPE u1,TYPE u2,GLint ustride,GLint uorder,TYPE v1,TYPE v2,
GLint vstride,GLint vorder,TYPE points);
```

参数 target 可以是表一中任意值，不过需将 MAP1 改为 MAP2。同样，启动曲面的函数仍是 glEnable()，关闭是 glDisable()。u1、u2 为 u 的最大值和最小值；v1、v2 为 v 的最大值和最小值。参数 ustride 和 vstride 指出在控制点数组中 u 和 v 向相邻点的跨度，即可从一个非常大的数组中选择一块控制点长方形。例如，若数据定义成如下形式：

```
GLfloat ctlpoints[100][100][3];
```

并且，要用从 ctlpoints[20][30]开始的 4x4 子集，选择 ustride 为 100*3，vstride 为 3，初始点设置为 ctlpoints[20][30][0]。最后的参数都是阶数，uorder 和 vorder，二者可以不同。

曲面坐标计算函数为：

```
void glEvalCoord2{fd}[v](TYPE u,TYPE v);
```

该函数产生曲面坐标并绘制。参数 u 和 v 是定义域内的值。下面看一个绘制 Bezier 曲面的例子：

```
/*  控制点的坐标  */
      GLfloat ctrlpoints[4][4][3] = {
{{-1.5, -1.5, 2.0}, {-0.5, -1.5, 2.0},
    {0.5, -1.5, -1.0}, {1.5, -1.5, 2.0}},
  {{-1.5, -0.5, 1.0}, {-0.5, 1.5, 2.0},
{0.5, 0.5, 1.0}, {1.5, -0.5, -1.0}},
{{-1.5, 0.5, 2.0}, {-0.5, 0.5, 1.0},
{0.5, 0.5, 3.0}, {1.5, -1.5, 1.5}},
{{-1.5, 1.5, -2.0}, {-0.5, 1.5, -2.0},
{0.5, 0.5, 1.0}, {1.5, 1.5, -1.0}}
};
void myinit(void)
{
            glClearColor (0.0, 0.0, 0.0, 1.0);
            glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4, 0, 1, 12, 4,
&ctrlpoints[0][0][0]);
glEnable(GL_MAP2_VERTEX_3);
glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
glEnable(GL_DEPTH_TEST);
      }
      void CALLBACK display(void)
```

```
    {
        int i, j;
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glColor3f(0.3, 0.6, 0.9);
        glPushMatrix ();
        glRotatef(35.0, 1.0, 1.0, 1.0);
        for (j = 0; j <= 8; j++)
        {
            glBegin(GL_LINE_STRIP);
            for (i = 0; i <= 30; i++)
                glEvalCoord2f((GLfloat)i/30.0, (GLfloat)j/8.0);
            glEnd();
            glBegin(GL_LINE_STRIP);
                for (i = 0; i <= 30; i++)
                    glEvalCoord2f((GLfloat)j/8.0,
(GLfloat)i/30.0);
            glEnd();
        }
        glPopMatrix ();
        glFlush();
    }
```

OpenGL 中定义均匀间隔的曲面坐标值的函数与曲线的类似，其函数形式为：

```
void glMapGrid2{fd}(GLenum nu,TYPE u1,TYPE u2,GLenum nv,TYPE v1,TYPE v2);
void glEvalMesh2(GLenum mode,GLint p1,GLint p2,GLint q1,GLint q2);
```

第一个函数定义参数空间的均匀网格，从 u1 到 u2 分为等间隔的 nu 步，从 v1 到 v2 分为等间隔的 nv 步，然后 glEvalMesh2()把这个网格应用到已经启动的曲面计算上。第二个函数参数 mode 除了可以是 GL_POINT 和 GL_LINE 外，还可以是 GL_FILL，即生成填充空间曲面。

下面举出一个用网格绘制一个经过光照和明暗处理的 Bezier 曲面的例程：

```
#include "glos.h"
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
void myinit(void);
void initlights(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);
/* 控制点坐标 */
GLfloat ctrlpoints[4][4][3] = {
    {{-1.5, -1.5, 2.0}, {-0.5, -1.5, 2.0},
```

```
    {0.5, -1.5, -1.0}, {1.5, -1.5, 2.0}},
  {{-1.5, -0.5, 1.0}, {-0.5, 1.5, 2.0},
{0.5, 0.5, 1.0}, {1.5, -0.5, -1.0}},
{{-1.5, 0.5, 2.0}, {-0.5, 0.5, 1.0},
{0.5, 0.5, 3.0}, {1.5, -1.5, 1.5}},
{{-1.5, 1.5, -2.0}, {-0.5, 1.5, -2.0},
{0.5, 0.5, 1.0}, {1.5, 1.5, -1.0}}
};
void initlights(void)
{
      GLfloat ambient[] = { 0.4, 0.6, 0.2, 1.0 };
      GLfloat position[] = { 0.0, 1.0, 3.0, 1.0 };
      GLfloat mat_diffuse[] = { 0.2, 0.4, 0.8, 1.0 };
      GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
      GLfloat mat_shininess[] = { 80.0 };
      glEnable(GL_LIGHTING);
      glEnable(GL_LIGHT0);
      glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
      glLightfv(GL_LIGHT0, GL_POSITION, position);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
  glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
}
void CALLBACK display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glPushMatrix();
  glRotatef(35.0, 1.0, 1.0, 1.0);
  glEvalMesh2(GL_FILL, 0, 20, 0, 20);
  glPopMatrix();
  glFlush();
}
void myinit(void)
{
          glClearColor (0.0, 0.0, 0.0, 1.0);
      glEnable (GL_DEPTH_TEST);
  glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4, 0, 1, 12,
4, &ctrlpoints[0][0][0]);
  glEnable(GL_MAP2_VERTEX_3);
  glEnable(GL_AUTO_NORMAL);
  glEnable(GL_NORMALIZE);
  glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
initlights();
}
```

```
void CALLBACK myReshape(GLsizei w, GLsizei h)
{
      glViewport(0, 0, w, h);
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      if (w <= h)
glOrtho(-4.0, 4.0, -4.0*(GLfloat)h/(GLfloat)w,
4.0*(GLfloat)h/(GLfloat)w, -4.0, 4.0);
else
    glOrtho(-4.0*(GLfloat)w/(GLfloat)h,
4.0*(GLfloat)w/(GLfloat)h, -4.0, 4.0, -4.0, 4.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}
void main(void)
{
      auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
   auxInitPosition (0, 0, 500, 500);
   auxInitWindow ("Lighted and Filled Bezier Surface");
   myinit();
   auxReshapeFunc (myReshape);
   auxMainLoop(display);
}
```

三、图元逼近法绘制三维物体

在 OpenGL 的辅助库中，提供了绘制 11 种基本几何图形的函数，具体参考第一讲的有关内容，在此不再赘述。这里我们讨论用另外一种方法来绘制三维物体 。

需要注意的是，这里我们用来近似曲面的多边形最好选择三角形，而不是四边形或其他形状的多边形，这是因为三角形的三个顶点在任何时候都位于同一平面内，它一定是非常简单的非凹多边形，而四边形或其他多边形的顶点可能不在同一平面内，也就有可能不是简单多边形，对于这样的多边形，OpenGL 是不能正常处理的。假设我们绘制一个球体，球体表面用很多个小三角形拼接而成，显然，用来近似球面的三角形越小、三角形越多，那么球面就越光滑。为了简要地说明如何用三角形逼近球体，这里我们使用三角形来构造一个 20 面体，二十面体的顶点坐标定义在 vdata[][]数组中，tindinces[][]数组定义了构成二十面体的二十个三角形顶点的绘制顺序。下面是主要实现代码：

```
#define x 5.25731
#define z 8.50651
static GLfloat vdata[12][3]={
{x,0.0,z},{x,0.0,z},{-x,0.0,-z},{x,0.0,-z},
{0.0,z,x},{0.0,z,-x},{0.0,-z-x},{0.0,-z,-x},
{z,x,0.0},{-z,x,0.0},{z,-x,0.0},{-z,-x,0.0}
};
static GLint tindices[20][3]={
```

```
{0,4,1},{0,9,4},{9,5,4},{4,5,8},{4,8,1},
{8,10,1},{8,3,10},{5,3,8},{5,2,3},{2,7,3},
{7,10,3},{7,6,10},{7,11,6},{11,0,6},
{6,1,10},{9,0,11},{9,11,2},{9,2,5},{7,2,11}
};
glColor3f(1.0, 0.0, 0.0);
for(int i=0;i<20;i++){
glBegin(GL_TRIANGLES);
glVertex3fv(&vdata[tindices[i][0]][0]);
glVertex3fv(&vdata[tindices[i][1]][0]);
glVertex3fv(&vdata[tindices[i][2]][0]);
glEnd();
}
```

显然，用正二十面体来表示一个球体显得过于粗糟，可以通过增加面数的方法使正多面体和求更为接近，一种简单的方法是剖分法，即将前面定义的三角形面分成几个面，例如，一分为四，形成 4 个多边形等，具体实现方法这里就不再赘述了。