第2章　命令和例程概述

许多OpenGL命令直接影响诸如点、线、多边形以及位图等 OpenGL对象的绘制。而另一些命令，例如那些用于反走样或纹理操作的命令，主要用来控制图像如何生成。还有一些命令则关注帧缓冲区的操作。

本章主要介绍所有 OpenGL命令是如何协同工作来建立 OpenGL处理流程的。同时也对OpenGL实用库（GLU）和对X窗口系统的OpenGL扩展（GLX）中的命令作了概述。

本章包括以下几个主要部分：

- OpenGL处理流程：在第1章的基础上讲解特定的OpenGL命令如何控制数据的处理。
- 其他OpenGL命令：讨论几个前一章中没有提及的 OpenGL命令集。
- OpenGL实用库：介绍了已有的GLU例程。
- 对X窗口系统 的OpenGL扩展：介绍GLX中有用的例程。

2.1　OpenGL处理流程

第1章介绍了OpenGL如何工作，本章将进一步讨论各阶段中数据处理的实际情况并且将各阶段与其用到的命令结合起来。图 2-1是一幅较为详细的OpenGL处理流程图。

从图中我们可以看到其中有三组箭头穿过了大多数的阶段。这三组箭头分别代表了顶点和与其相关的两个主要的数据类型 —— 颜色值和纹理坐标。值得注意的是顶点首先组合成图元，然后是片断，最后成为帧缓冲区中的像素。这一过程将在下面章节中作详细介绍。

一个OpenGL命令的效果将很大程度地依赖于某特定模式是否有效。例如，与光照有关的命令只有当你启动了光照功能才能有效地生成一个适当的光照对象。如果要启动一个特定的模式，请调用glEnable( )命令，并且要提供一个适当的常量来确定该模式（如 GL_LIGHTING）。下面章节中并没有介绍特定的模式，但在函数 glEnable()的使用说明中提供了一个完整的列表用来说明它可启动的模式。调用函数 glDisable()可以关闭一个模式。

2.1.1　顶点

本节介绍与在图2-1中与各顶点操作有关的OpenGL命令。它包含了有关顶点数组的各种信息。

1. 输入数据

你必须为OpenGL流程提供几种输入数据类型。

- 顶点 —— 顶点用来描述所需要的几何对象的形状。你可以通过在函数对 glBegin()/glEnd()之间调用函数glVertex*()来指定顶点，并用这些顶点建立点、线或多边形。你也可以用函数glRect*()来直接绘制一个完整的矩形。
- 边界标志 —— 在缺省情况下，多边形的所有边都是边界边。用函数 glEdgeFlag*()可以显式

地设置边界标志。

- 当前光栅位置 —— 当前光栅位置用来确定绘制像素和位图时的光栅坐标。它由函数glRasterPos*()指定。

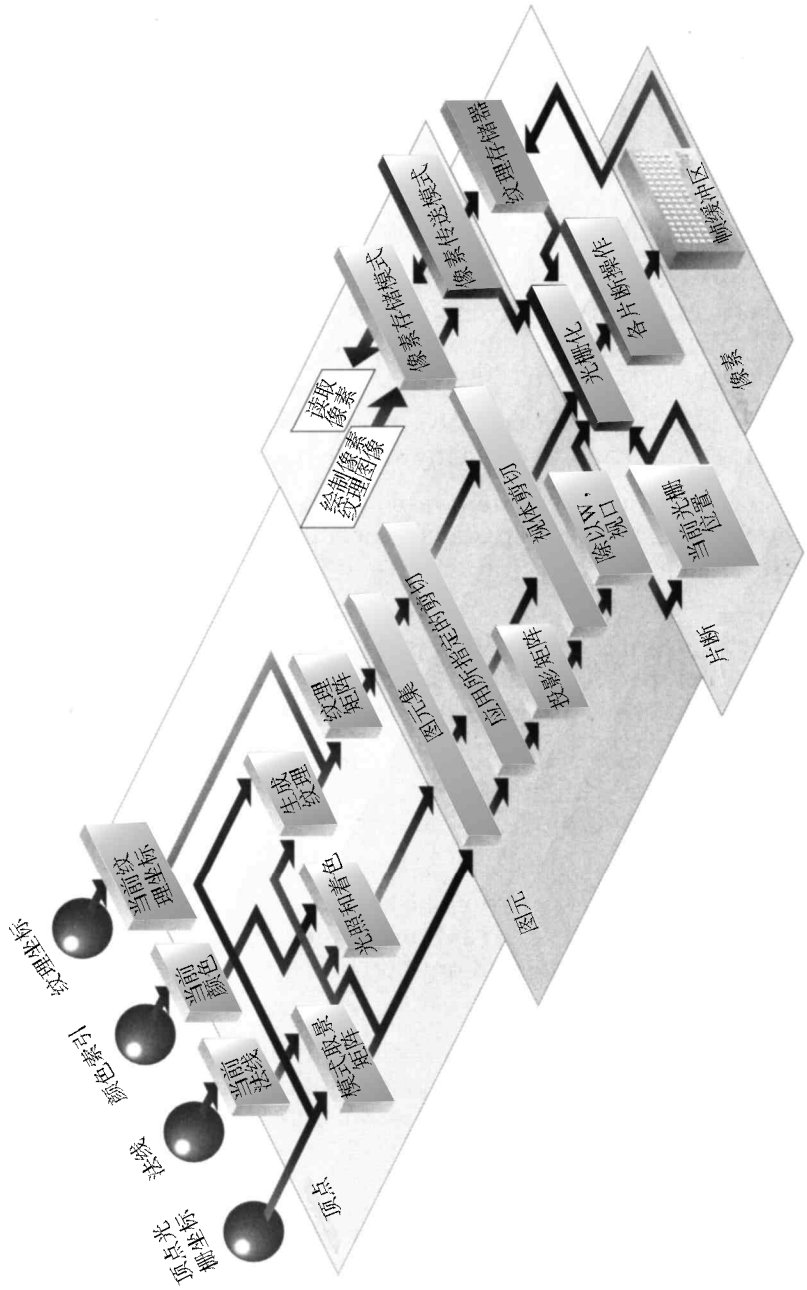

图2-1 OpenGL处理流程的各个阶段

- 当前法线 —— 每个法向量都与一个特定的顶点相对应，它用来确定顶点处的表面在三维空间中的方向。它同时又影响该顶点所接收的光照的多少。函数 glNormal*()用来指定一个法向量。
- 当前颜色 —— 一个顶点的颜色用来确定光照对象最终的颜色。在 RGBA模式下，可以通过函数glColor*()来指定颜色；在颜色索引模式下则需使用函数 glIndex*()。
- 当前纹理坐标 —— 纹理坐标用来确定在纹理映射表中的位置，此位置与一个对象的某个顶点相关联。它们可以由函数 glTexCoord*()指定。当系统支持 ARB多重纹理扩展时用函数 glMultiTexCoord*ARB()。

当调用函数 glVertex*( )时，生成的顶点将继承当前的边界标志、法线、颜色和纹理坐标。因此，必须在函数 glVertex*( )之前调用函数 glEdgeFlag*()、glNormal*()、glColor*()和glTexCoord*( )来影响所生成的新顶点。

上面所列的所有顶点的输入数据可以通过使用 “ 顶点数组 ” 来指定，在下面的叙述中会有详细介绍。它允许通过调用单个函数来传送顶点数组数据。某些 OpenGL机制可能用该方法来指定顶点会更有效。

2. 矩阵转换

顶点和法线首先要各自通过矩阵转换后才能用于在帧缓冲区中生成图像。顶点通过模式取景矩阵和投影矩阵转换，而发光的法线则由模式取景矩阵转换。你可以使用诸如 glMatrix-Mode()、glMultMatrix*()、glRotate*()、glTranslate*()和glScale*()等函数来组成所需的转换。或者，也可直接用函数 glLoadMatrix*()和glLoadIdentity()指定矩阵。用函数 glPush- Matrix()和glPopMatrix()可以在各自的堆栈中存储和恢复模式取景矩阵和投影矩阵。

3. 光照和着色

除了指定颜色和法向量外，你还可以用函数 glLight*()和glLightModel*()指定所需的光照环境，用函数 glMaterial*()指定所需的材料属性。用于控制光照计算的相关命令有：glShadeModel()、glFrontFace()和glColorMaterial()。

4. 生成纹理坐标

OpenGL并不明确地提供纹理坐标，而是用其他顶点数据的函数来生成它们。这项工作由函数glTexGen*( )完成。当纹理坐标被指定或生成之后，它们将通过纹理矩阵实现转换。控制这些矩阵所使用的命令与前面 “ 矩阵转换 ” 一节中所提及的命令是一样的。

5. 图元集

一旦所有的计算执行完，这些顶点 —— 连同各顶点相应的边界标志、颜色和纹理信息 —— 将被组合成图元（包括点、线段和多边形）。

6. 顶点数组

你只需调用有限的几个命令就可以通过顶点数组来指定几何图元。当你调用函数 glDrawArrays()来绘制图元时，你不需要再通过调用一个个的 OpenGL函数来传送每个单独的顶点、法线或颜色，而只要调用一个 glDrawArrays()函数来分别指定顶点数组、法线数组和颜色数组就可以用它们来定义一系列要绘制图元（所有同一类型的）。函数 glVertexPointer()、glNormalPointer()、glColorPointer()、glIndexPointer()、glTexCoordPointer()和

glEdgeFlagPointer()用来描述数组的组织结构和存储单元的位置。函数 glEnableClientState()和glDisableClientState()用来指定将访问哪个顶点数组中的顶点坐标和属性。

一个顶点的所有当前有效的数据都可以通过在函数对 glBegin()/glEnd()之间调用函数glArrayElement()来指定。此外，函数glDrawElements()和glDrawRangeElements()可以随机访问顶点数组。

7. 图元

在流程的下一个阶段中，图元将被转化成像素片断。该过程有以下几个步骤：适当地剪切图元；对颜色和纹理作必要的相关调整；将有关坐标转换成窗口坐标；最后，将剪切好的图元通过光栅化处理而转换成像素片断。

8. 剪切

当点、线段和多边形需要剪切时，OpenGL对它们的处理稍有不同。对于点而言，要么维持其原始状态（当它包含在剪切体积内时），要么被丢弃（当它处于剪切体积外时）。对于线段和多边形则不尽相同。如果线段或多边形的一部分处于剪切体积外，则在剪切点的位置上生成一个新的顶点。而对于多边形，这些新的顶点之间还需要重新连一条完整的边。当线段和多边形被剪切时，其边界标志、颜色和纹理信息都被赋给新顶点。

剪切实际上有两个步骤：

1) 由应用所指定的剪切。图元一旦被组合而成，它们将根据应用的要求，通过函数glClipPlane()指定的剪切平面在*眼坐标*中进行剪切。（任何的OpenGL机制都支持至少六个这样的应用相关的剪切平面）。

2) 视体剪切。接下来，图元将由投影矩阵转换（成*剪切坐标*）并被相应的视体剪切。你可以通过矩阵转换命令来控制这些矩阵。但更多地，它们由函数 glFrustum()和glOrtho()指定。

9. 转换成窗口坐标

在剪切坐标转换成*窗口坐标*前，它们先要被归一化，即通过除以 *w*值从而生成*归一化设备坐标*。之后，通过视口转换将这些归一化的坐标变为窗口坐标。你可以用函数 glDepthRange()和glViewport()来控制这些视口 —— 它们将决定显示图像的窗口屏幕区域。

10. 光栅化

光栅化是将一个图元转化为一个二维图像的操作。该图像中的每个点都包含这样一些信息：颜色、深度和纹理数据。点及其相关信息被称为一个“片断”。

当前光栅位置（由函数glRasterPos*()指定）在该阶段的像素绘制和位图中有多种用途。三种不同类型的图元的光栅化是各不相同的。另外，像素矩形和位图均需被光栅化。

- 图元。下面的命令允许你通过选择图元的尺寸及点画模式对图元的光栅化进行控制：glPointSize()、glLineWidth()、glLineStipple()和glPolygonStipple()。你也可以通过命令glCullFace()、glFrontFace()和glPolygonMode()来控制正面多边形和背面多边形将如何被光栅化。
- 像素。有几个命令用于控制像素的存储和传送模式。命令 glPixelStore*()用来控制像素在客户端存储器中的编码方式，glPixelTransfer*()和glPixelMap*()控制像素存入帧缓冲区之前的处理方式。另外，当你使用的 OpenGL机制支持“ ARB绘图子集 ”（参见2.1.2节）时，

就可以对像素进行其他处理。像素矩形由函数 glDrawPixels()指定，其光栅化由函数glPixelZoom()控制。

- 位图。位图是由0和1所组成的矩形，它用来指定一个将要生成的特定格式的片断图案。每个这样的片断都有相同的相关数据。位图由函数 glBitmap()指定。
- 纹理。当纹理功能启动后，它将把一个指定的纹理图像的一部分映射到每个图元上。要想实现这种映射，你需要使用由片断的纹理坐标所确定的存储单元中的纹理图像的颜色来修改该片断的RGBA颜色。

  你可以用函数glTexImage1D(), glTexImage2D()或glTexImage3D()来指定一个纹理图像。如果要通过拷贝帧缓冲区中的数据来建立一个纹理图像，请使用函数 glCopyTexImage1D()或glCopyTexImage2D()。你也可以通过 glTexSubImage1D(), glTexSubImage2D()或 glTexImage3D()载入子图像，或通过函数 glCopyTexSubImage1D(), glCopyTexSubImage2D()或glCopySubImage3D()用从帧缓冲区中拷贝来的数据替换部分纹理图像。函数glTexParameter*()和glTexEnv*()用来控制对纹理值进行解释并应用于一个片断。

  要指定具体哪些纹理优先存入纹理内存，需先调用函数 glBindTexture()生成指定的纹理（纹理对象），然后再调用函数 glPrioritizeTextures()给它们排序。函数glDeleteTexture()可以删除一个纹理对象。
- 颜色总和。是指在纹理操作之后，将由镜面光照计算而来的颜色片断加到片段上。通过函数glLightModel()且将GL_LIGHT_MODEL_COLOR_CONTROL参数赋值为GL_SEPARATE_SPECULAR_COLOR可以指定镜面光照颜色进行分别计算。
- 雾。OpenGL可以将一种雾颜色和一种已光栅化的片断的纹理颜色用一个融合因子融合在一起。该融合因子由观察点与片断之间的距离决定。用函数glFog*()可指定雾颜色和融合因子。
- 多边形偏移。当你绘制消隐图形或对表面进行贴面时，应该考虑使用函数glPolygonOffset() 替换一个片断的深度值。该深度值是在绘制多边形时将一个特定的偏移量加到一个可调节的量上而生成的。该可调节量依赖于多边形深度值的变化量及它的屏幕尺寸。这种替换允许在同一个面上绘制多边形而相互间不产生影响。你可以启动如下所示的三种多边形模式之一来确定将采用怎样的替换方法。这三种多边形模式是 GL_POLYGON_OFFSET_FILL，GL_POLYGON_OFFSET_LINE和GL_POLYGON_OFFSET_POINT。

多边形偏移可用于绘制消隐图形、绘制高亮边的实心体及将贴面应用于物体表面。

2.1.2 ARB绘图子集

OpenGL机制可以支持任选的OpenGL ARB绘图子集。该集合是由数个附加的像素处理操作组成的。下面的功能仅当调用函数 glLightString(GL_EXTENSIONS)的返回值是字符串GL_ARB_imaging时才有效。

ARB图形子集所包含的功能如下所示：

- 颜色表。颜色查询表提供了一种替换单个像素的方法。颜色表可用 glColorTable例程指定。如果要指定一个基于帧缓冲区值的颜色表，可以调用函数 glCopyColorTable( )。函数

glCopySubTable()允许你替换颜色表的一部分。而函数 glCopyColorSubTable()将使用帧缓冲区中的值替换指定的部分。在指定颜色表时可以对它进行缩放和偏移。用函数 glColorTableParameter*()来指定缩放和偏移值。

- 卷积滤波器。卷积是结合附近的像素来计算一个最终的像素值的方法。卷积滤波器可由函数 glConvolutionFilter1D()、glConvolutionFilter2D()或glSeparableFilter2D()指定。另外，卷积滤波器也可以用帧缓冲区中的值来指定，这时需要调用函数glCopyConvolutionFilter1D() 和 glCopyConvolutionFilter2D()。卷积滤波器可以通过函数glConvolutionParameter*()指定的值对它进行缩放和偏移。经过卷积操作后，所得的像素值也可以进行缩放和偏移，函数 glPixelTransfer*()用来指定缩放和偏移值。
- 颜色矩阵转换。矩阵转换可通过颜色矩阵堆栈而应用于像素。函数 glMatrixMode (GL_COLOR)用来修正当前的颜色矩阵。有关内容请参阅“矩阵转换”。经过颜色矩阵转换之后，像素值可以通过函数 glPixelTransfer*()指定的值进行缩放和偏移。
- 直方图。直方图用来确定像素矩形中值的分布情况。函数 glHistogram()用来指定哪些颜色组件将被计数。用函数 glGetHistogram()可以返回直方图的计算结果。函数 glReset-Histogram()可以重新设置直方图表，而调用函数 glGetHistogramParameter*()将返回用于描述直方图表的值。
- 最值。每个像素矩形的最小和最大值可以用函数glMinmax()来计算。函数glGetMinmax()返回通过函数 glMinmax()指定的颜色组件而计算所得的最小和最大值。调用函数 glReset-Minmax()可以重新设置内部的最值表，而调用函数 glGetMinmaxParameter*()将返回用于描述该表的参数值。
- 融合方程。除了汇总外，像素还可以用融合而非叠加方式进行合并。函数glBlendEquation() 用来指定源和目标像素值将如何被合并。
- 常数融合颜色。除了标准融合函数外，ARB绘图子集允许用常数作为源或目标颜色值的系数。调用函数glBlendColor()来指定常数融合颜色，它将与函数 glBlendFunc()所指定的融合系数一起使用。

2.1.3 片断

当一个通过光栅化而生成的片断能通过一系列的测试时，OpenGL允许通过该片断来修正帧缓冲区中相应的像素。如果该片断没通过测试，则该片断可被用来直接替换帧缓冲区中的值，或者与帧缓冲区中已存在的值合并。具体情况将视特定模式的状态而定。

1. 像素所有权测试

第一项测试用来检验与一个特定的片断相应的帧缓冲区中的像素是否属于当前的 OpenGL环境。如果是，则对片断进行下一项测试；否则，将通过窗口系统来决定是丢弃该片断还是要对该片断进行后面的操作。当一个 OpenGL窗口不明确时，这一测试允许窗口系统来控制 OpenGL的行为。

2. 裁剪测试

裁剪测试将丢弃通过函数 glScissor()指定的任意屏幕上的校正矩形区域外的片断。

3. Alpha测试

Alpha测试（它仅在RGBA模式下执行）将根据片断的alpah值和一个常数参考值之间的比较结果而丢弃一个片断。该比较命令和参考值由函数glAlphaFunc()指定。

4. 模板测试

模板测试将基于模板缓冲区中的值和一个参考值的比较结果而丢弃一个片断。函数glStencilFunc()用来指定所使用的比较命令及参考值。不管片断是否能通过模板测试，模板缓冲区中值的修正都由函数glStencilOp()决定。

5. 深度缓冲测试

当对一个片断进行的深度比较操作失败时，深度缓冲测试将丢弃该片断。函数glDepthFunc()用来指定比较命令。当模板缓冲区有效时，深度比较的结果也将影响模板缓冲区的更新值。

6. 融合

融合是将一个片断的R、G、B和A值与存放于帧缓冲区相应存储单元中的值合并。该操作仅在RGBA模式下才能使用。它根据片断的alpha值与当前存储的像素相应值而产生不同的操作，它也受RGB值的限制。为了控制融合操作，你可以调用函数glBlendFunc()来指定源和目标融合因子。

当OpenGL机制支持ARB绘图子集时，它将提供一些附加的融合功能，有关细节请参阅2.1.2节“ARB图形子集”。

7. 抖动

当抖动功能启动后，OpenGL将对片断的颜色或颜色索引应用一种抖动算法。该算法仅由片断的值及其*x*和*y*窗口坐标决定。

8. 逻辑操作

逻辑操作可在片断和存储于帧缓冲区相应位置中的值之间使用。其结果将替换当前帧缓冲区中的值。你可以通过函数glLogicOp( )来选择理想的逻辑操作。逻辑操作只能应用于颜色索引值，它不能用于RGBA值。

9. 像素

在OpenGL流程的前一阶段中，片断被转换成了帧缓冲区中的像素。帧缓冲区实际上组成了一系列逻辑缓冲区，这些缓冲区包括：颜色缓冲区、深度缓冲区、模板缓冲区和累积缓冲区。其中颜色缓冲区由前左、前右、后左、后右以及一定数量的辅助缓冲区所构成。你可以用命令控制这些缓冲区，也可以直接从它们中读取或拷贝像素。（请注意：你所使用的特定的OpenGL环境不一定提供了所有这些缓冲区。）

10. 帧缓冲区操作

你可以用函数glDrawBuffer()来选取你想写入颜色值的缓冲区。另外，当所有片断操作已执行完成以后，你还可以用四个不同的命令对每个逻辑帧缓冲区中位的写入进行屏蔽。这四个命令是glIndexMask()、glColorMask()、glDepthMask()和glStencilMask()。累积缓冲区的操作由函数glAccum()控制。此外，函数glClear()可以将一个指定的缓冲区子集中的每个像素设置成某个指定值，该指定值由函数glClearColor()、glClearIndex()、glClearDepth()、glClearStencil()

或glClearAccum()指定。

11. 读取或复制像素

你可以将由帧缓冲区中读出的像素读入内存中，也可以用各种方式将它们编码，并将编码结果存入内存中。函数glReadPixels()可用来完成上述工作。另外，你可以用函数glCopyPixels()将帧缓冲区的某个区域中的一个像素矩形的值复制到另一个区域。函数glReadBuffer()用来控制从哪个颜色缓冲区中读取或复制像素。

2.2 其他OpenGL命令

本节主要介绍一组特殊的命令。这些命令在图2-1中并没有被明确地作为OpenGL处理流程的一部分而显示出来。这些命令实现了诸如多项式求值、显示列表的使用以及获取OpenGL状态变量的值等多项任务。

2.2.1 使用求值器

OpenGL的求值器命令允许你用一个有理多项式映射来生成顶点、法线、纹理坐标及颜色。这些计算所得的值将通过流程，就象它们是被直接指定的一样。该求值工具也被NURBS（Non-Uniform Rational B-Spline）命令所使用，该命令允许你定义曲线和曲面。具体情况请参阅2.3节“OpenGL实用库”及本书第6章。

使用求值器前，你首先必须用函数glMap*()定义适当的一维或二维多项式映射。你可以用下面两种方法之一来指定和求取该映射的域值：

- 首先用函数glMapGrid*()定义一系列用于映射的等间隔域值，然后用函数glEvalMesh*()求取该网格的一个矩形子集。如果要想求取网格中的一个单独的点，请你使用函数glEvalPoint*()。
- 将一个期望的域值明确地指定为函数glEvalCoord*()的自变量，这时将求取该域值的映射。

2.2.2 执行选择和反馈

选择、反馈和绘制是三种互斥的操作模式。在通过光栅化而生成片断的过程中，绘制是它的默认模式。在选择和反馈模式中，没有片断生成，因此也不对帧缓冲区进行修改。在选择模式中，你可以决定将把哪些图元绘入窗口的某个区域；而在反馈模式中，即将被光栅化的图元的信息被返回给应用程序。你可以通过函数glRenderMode()来选取所需的模式。

1. 选择

选择操作是通过返回的名称堆栈的当前内容来工作的。名称堆栈是一个整型名称数组。你可以在模式代码中指定名称并建立名称堆栈。该模式代码的作用是指定将要绘制的对象的几何形状。

一旦图元与剪切体积相交，将有一个选择命中产生。该命中纪录将被写入函数glSelectBuffer()提供的选择矩阵中。命中纪录中包含有命中发生时名称堆栈的内容。（请注意：函数glSelectBuffer()必须在OpenGL被函数glRenderMode()设置成选择模式之前调用。同时，在函数glRenderMode()将OpenGL退出选择模式之前也不能保证名称堆栈中所包含的内容都被返回。）

你可以使用函数glInitNames()、glLoadName()、glPushName()和glPopName()来操作名称堆栈。对于选择模式我们可以考虑使用OpenGL实用库（GLU）中的例程gluPickMatrix()，该例程在2.3节“OpenGL实用库”及本书第6章中将有详细介绍。

2. 反馈

在反馈模式中，每个光栅化的图元将生成一个数值块，并将其拷贝到反馈数组中。你可用函数glFeedbackBuffer()提供这个数组。而这一函数必须在OpenGL设置成反馈模式前被调用。每个数值块开始的一个代码用来指明图元的类型，接下来是描述图元的顶点和相关数据的值。它们也可以被写入位图和像素矩形。在调用函数glRenderMode()使OpenGL退出反馈模式之前，并不能保证已将数据写入了反馈数组。在反馈模式下，你可以使用函数glPassThrough()提供一个标记，它在反馈模式里被返回，就象它是个图元一样。

2.2.3 显示列表的使用

一个显示列表就是一组被存储起来以备以后执行的OpenGL命令。函数glNewList()用来创建一个显示列表，函数glEndList()结束创建工作。绝大多数在glNewList()和glEndList()之间被调用的OpenGL命令都被添加到显示列表并被选择执行。（函数glNewList()的参考说明中列出了所有不能在显示表中存储并执行的命令。）如果要执行一个或一组显示表，你可调用函数glCallList()或glCallLists()，并同时提供用于识别一个或一组特定显示表的数字。你可以通过函数glGenLists()、glListBase()和glIsList()来管理用于识别显示表的索引。函数glDeleteLists()用来删除一组显示表。

2.2.4 模式和运行的管理

许多OpenGL命令的执行结果都跟某一特定的模式是否有效有关。你可以使用函数glEnable()和glDisable()来设置这样的模式，也可以用函数glIsEnabled()来确认某一特定的模式是否已被设置。

你可以使用函数glFinish()来控制以前已发布的OpenGL命令的执行，该函数将强迫所有的命令完成。你也可以用函数glFlush()，它将确保所有这些命令在有限的时间内完成。

OpenGL的一个特殊实现是函数glHint()。你可以通过该函数使用“提示”来控制绘制的某些方面。你同样可以控制颜色和纹理坐标插入值的质量、雾化计算的精度以及反走样点、线或多边形的样本质量。

2.2.5 获取状态信息

OpenGL含有大量的状态变量，它们对许多命令的行为都将产生影响。下列变量指定了特定的查询命令：

glGetClipPlane	glGetColorTable†
glGetColorTableParameter†	glGetConvolutionFilter†
glGetConvolutionParameter†	glGetHistogram†

glGetHistogramParameter† glGetLight
glGetMap glGetMaterial
glGetMinmax† glGetMinmaxParameter†
glGetPixelMap glGetPointerv
glGetPolygonStipple glGetSeparableFilter
glGetTexEnv glGetTexGen
glGetTexImage glGetTexLevelParameter
glGetTexParameter

注：带“†”号的例程仅当OpenGL机制支持ARB绘图子集时才可以使用。

如果要获取其他状态变量的值，你可以调用函数 glGetBooleanv()、glGetDoublev()、glGetFloatv()或glGetIntegerv()。函数glGet*()的参考说明中介绍了如何使用这些命令。另外，你还可以使用glGetError()、glGetString()和glIsEnabled()等查询命令。（与出错处理有关的例程的细节见2.3.6节“错误处理”。）你可以用函数glPushAttrib()和glPopAttrib()来存储和恢复状态变量集。

2.3 OpenGL实用库

OpenGL实用库（GLU）包含了几组命令，这些命令通过提供对辅助特性的支持，补充了核心OpenGL界面。由于这些实用例程是使用核心的 OpenGL命令，所以任何的OpenGL机制都能保证支持这些实用例程。这些实用库例程的前缀是 *glu*而非*gl* 。

2.3.1 生成纹理操作所需的图形

GLU提供了缩放图形及自动进行 mipmap的例程来简化纹理图形的指定过程。例程gluScaleImage()用来将一个指定的图形缩放成一个可接受的纹理尺寸。所得的图形然后作为一个纹理传送给 OpenGL。自动 mipmap例程 gluBuild1DMipmaps()、gluBuild2DMipmaps()和gluBuild3DMipmaps()将从一个指定的图形中生成一个mipmap的纹理图形，然后将它们分别传送给glTexImage1D()、glTexImage2D()和glTexImage3D()。另外，例程gluBuild1DMipmapLevels()、gluBuild2DMipmapLevels()和gluBuild3DMipmapLevels()将为一个指定的mipmap图层建立一个mipmap纹理图形范围。

2.3.2 坐标转换

这里提供了几个普通用途的矩阵转换例程。你可以用例程 gluOrtho2D()来建立一个二维的正交观察区域，用例程gluPerspective()建立一个透视观察体积，或用例程gluLookAt()建立一个中心在指定眼点的观察体积。每个例程都建立了一个所需的矩阵，并通过函数 glMultMatrix() 将它应用于当前矩阵。

例程gluPickMatrix()通过建立一个矩阵而简化了选择操作。该矩阵用来将绘图约束到视口中的一个小区域中。如果你是在使用这个矩阵之后的选择模式下绘制图像，则光标附近所有要被绘制的对象将被选取，并且它们的相关信息将被存入选择缓冲区中。（有关选择模式的详细情况见2.2.2节“执行选择和反馈”。）

如果你想确定在窗口的什么位置绘制物体，可以使用例程 gluProject()。该例程将把指定对象的对象坐标转换成窗口坐标，而例程gluUnProject()和gluUnProject4()则执行相反的操作。

2.3.3 多边形的镶嵌分块

多边形镶嵌分块例程用一个或多个轮廓线将一个凹多边形分割成三角形。使用这个 GLU功能时，首先用例程gluNewTess()建立一个镶嵌分块的对象，并用 gluTessCallBack()定义一个反馈例程，该例程将通过镶嵌分块器来处理三角形的生成。接下来用命令 gluTessBeginPolygon()、glTessVertex()和glTessEndPolygon()来指定将被镶嵌分块的凹多边形。你也可以在例程对gluTessBeginPolygon()/gluTessEndPolygon()之间使用 gluTessBeginContour()和gluTessEndContour()来定界轮廓线。如果要删除一个不需要的镶嵌分块对象，请使用例程gluDeleteTess()。

GLU镶嵌分块例程将把所有多边形投影到一个平面上，并镶嵌分块该投影。用命令gluTessNormal()可以为平面指定一个法线（并作为平面自身的一个结果）。如果该分块平面法线被设置为（0，0，0）—— 它的初始值，则命令 gluTessNormal()将基于命令gluTessVertex()所指定的值而选取一个平面。

2.3.4 绘制球体、圆柱和圆盘

你可以使用GLU的二次曲面例程来绘制球体、圆柱和圆盘。要完成这些工作，你需要首先使用gluNewQuadric()建立一个二次对象。如果你对默认值感到不满意，可以使用下面的例程来指定期望的绘制模式：

- gluQuadricNormals()决定是否应该生成表面法线。如果是，确定每个顶点都要一条法线或是否每个面上都要一条法线。
- gluQuadricTexture()决定是否应生成纹理坐标。
- gluQuadricOrientation()决定二次曲面的哪一边应被认为是外部，哪一边应是内部。
- gluQuadricDrawStyle()决定二次曲面是否应被画成为一组多边形、线或点的集合。

当你已经指定好绘制模式时，就可以为所希望的二次对象调用绘制例程，请用：gluSphere()、gluCylinder()、gluDisk()或gluParticalDisk()。如果在绘制过程中发生了一个错误，就会触发由例程gluQuadricCallBack()指定的出错处理例程。当你使用完一个二次对象后，如果想删除它，请使用gluDeleteQuadric()。

2.3.5 NURBS曲线和曲面

本节描述了将NURBS(非归一化的有理B样条)曲线和曲面转换到OpenGL的求值器的例程。你可以使用gluNewNurbsRenderer()和gluDeleteNurbsRenderer()来建立和删除一个NURBS对象，用gluNurbsCallBack()来建立一个错误处理例程。

你可以用不同的例程集来指定所需的曲线和曲面。指定曲线的例程有： gluBeginCurve()、gluNurbsCurve()和gluEndCurve()，指定曲面的例程有： gluBeginSurface()、gluNurbsSurface()和gluEndCurbsSurface()。你也可以指定一个修剪区域，这个区域将用来指定一个用

于求值的NURBS曲面域的子集。这样，你便可以建立具有光滑边界或包含孔洞的曲面。这些修整例程有：gluBeginTrim()、gluPwlCurve()、gluNurbsCurve()和gluEndTrim()。

与二次对象类似，你同样可以控制NURBS曲线和曲面的绘制：

- 决定当一个曲线或曲面的控制多面体位于当前视口外时是否丢弃它们。
- 决定用于绘制曲线和曲面的多边形边的最大长度（像素形式）。
- 决定是将投影矩阵、模式取景矩阵和视口从OpenGL服务器中取走还是用gluLoadSamplingMatrices()明确地支持它们。

你可以使用例程gluNurbsProperty()来设置这些特性，或使用默认值。要查询一个NURBS对象的绘制模式，请使用例程gluGetNurbsProperty()。

2.3.6 错误处理

例程gluErrorString()返回一个与OpenGL或GLU出错代码相应的出错字符串。现有的OpenGL出错代码在函数glGetError()的介绍中已作了说明。有关GLU的出错代码请参阅gluErrorString()、gluTessCallback()、gluQuadricCallback()及gluNurbsCallback()的介绍。GLX例程所产生的错误在有关例程的介绍中都作了说明。

2.4 对X窗口系统的OpenGL扩展

在X窗口系统中，OpenGL绘制被作为一个向正式的X环境的X扩展——它使用普通的X机制实现了连接和确认。如同使用其他的X扩展一样，有一个为被封装在X字节流中的OpenGL绘制命令定义的网络协议。由于三维绘制的效率是至关重要的，因此OpenGL向X的扩展允许OpenGL忽略X服务器对数据的编码、复制和编译，而直接向图形流程绘制。

本节简要讨论了作为GLX一部分的例程。这些例程都带有前缀*glX*。要全面理解以下各节和成功使用GLX，你需要有一些关于X的知识。

2.4.1 初始化

你可以通过例程glXQueryExtension()和glXQueryVersion()来确定是否为一个X服务器定义了GLX扩展。如果已定义，还要确定服务器中使用的是哪个版本。如果你要决定GLX机制的功能，请使用例程glXQueryServerString()和glXQueryExtensionsString()。它们将返回X服务器所支持的扩展信息。例程glXGetClientString()描述了由GLX客户库所提供的功能。

例程glXChooseFBConfig()返回一个与指定属性相匹配的GLXFBConfig的数组。用例程glXGetFBConfigAttrib()可以返回一个与特定的GLXFBConfig相应的特定属性值，并可选取你的应用所需的最佳GLXFBConfig单元。如果你要获得一个所有可用的GLXFBConfig的完整清单，请调用例程glXGetFBConfigs()。调用例程glXGetVisualFromFBConfig()将获得一个与指定的GLXFBConfig相应的XVisualInfo结构。

2.4.2 控制绘制操作

为了使用OpenGL的GLX绘图，首先应该具备绘图区域和管理OpenGL状态所需的环境。

GLX提供了几个命令用于建立、删除和管理 OpenGL的绘图区域和绘图环境，并将它们联合起来以便实现OpenGL的绘图功能。

另外，还提供了附加指令用于实现 X和OpenGL流之间的同步、交换前后缓冲区以及使用 X字体。

1. 管理屏幕绘图区域

要想在OpenGL窗口的屏幕上绘图，首先应该使用一个合适的 X可视环境建立一个 X窗口（一般使用由例程 glXGetVisualFromFBConfig()所得到的结构 XVisualInfo建立）。如果要将X窗口转换成一个GLXWindow，请使用例程glXCreatWindow()。如果要删除一个GLXWindow，请使用例程glXDestroyWindow()。

2. 管理屏幕外的绘图区域

GLX支持两种类型的屏幕外绘图区域： GLXPixmaps和GLXPbuffers。GLXPixmaps是与一个GLX像素映射资源相应的X像素映射。同支持OpenGL绘入像素映射一样，它也同样支持 X绘图。GLXPbuffers是一个GLX的独有资源。因此除GLX外，其他设备不能用 X或一个X扩展来绘制图形。

要建立一个GLXPixmaps，应首先建立一个X像素映射，然后通过例程 glXCreatPixmap()将它转换成一个GLXPixmaps。如果要删除一个 GLXPixmaps，请使用glXDestroyPixmap()，然后还需要删除最初的X像素映射。

由于GLXPbuffers没有相应的X可绘区域，所以它只需要调用 glXCreatePbuffer()来建立一个GLXPbuffers资源。类似地，可以用 glXDestroyPbuffer()来删除GLXPbuffers。

3. 管理OpenGL绘图环境

例程glXCreateNewContext()可以创建一个OpenGL绘图环境。该例程的一个自变量允许你忽略X服务器而直接申请一个绘图环境（请注意：如果要直接绘图， X服务器必须是局部的，并且OpenGL机制需要支持直接绘制）。你可以用 glXIsDirect()来确定一个OpenGL环境是否是直接的。如果要获取GLX环境的其他属性，可调用例程 glXQueryContext()。

如果要使一个绘图环境成为当前的（将一个 GLX绘图区域与一个GLX环境相关联），请使用 glXMakeContextCurrent()；而 glXGetCurrentContext()返回当前的环境。你也可以用 glXGetCurrentDrawable()来获取当前的可绘区域。同样，你还可以用 glXGetCurrent-ReadDrawable() 来获取当前读取的可绘区域。另外，例程 glXGetCurrentDisplay()可以返回与X显示相关的当前可绘区域和相关环境。

在任何时刻，对任何线程都只有一个当前环境。如果你有多个环境，你可以用 glXCopy-Context()来从一个环境向另一个环境复制所选定的 OpenGL状态变量组。当你不再需要一个特定的环境时，请用glXDestroyContext()删除它。

4. 同步执行

要想在任何未完成的OpenGL绘制结束前阻止X请求发生，请调用 glXwaitGL()。这样，任何以前发布的OpenGL命令将能确保在glXwaitGL()执行后所产生的X绘制调用发生之前被执行。尽管调用glFinish()也能得到同样的结果，但当客户端和服务器端在不同的机器上时，前者将更有效；这是因为X服务器将等待OpenGL绘制的完成而不是象函数 glFlush()那样等待客户端应用的

完成。

要想在任何未完成的X请求完成之前阻止一个OpenGL命令序列的执行，请调用glXWaitX()。该例程将确保在它执行后所产生的任何OpenGL命令执行之前首先执行以前发布的X绘制命令。

5. 事件处理

除了由X服务器提供的普通事件流之外，GLX还另外添加了应用中可能会处理到的其他事件。用例程glXSelectEvent()可以选择应用中希望被提示的GLX事件。现在，当改变一个GLXPbuffers时只有一个GLX事件被发送。如果要返回一个GLX事件，请调用glXGetSelectedEvent()。

6. 交换缓冲区

对于双缓冲区的可绘环境，通过例程glXSwapBuffers()可以实现前后缓冲区的互换。一个隐含的glFlush()将被作为这个例程的一部分来执行。

7. 使用X字体

命令glXUseXFont()为在OpenGL中使用X字体提供了一个捷径。