



OpenGL® Reference Manual

Third Edition

The Official Reference Document to OpenGL, Version 1.2



软件开发技术 丛书

openGL

参考手册

(第3版)



OpenGL Architecture Review Board 编

美 Dave Shreiner 主编

孙京国 王剑平 赵宇鹏 译



机械工业出版社
China Machine Press



Addison-Wesley

软件开发技术丛书

OpenGL® 参考手册

第3版

OpenGL Architecture Review Board 著

(美) Dave Shreiner 主编

孙守迁 王 剑 林宗楷 等译

李岳梅 罗仕鉴 审校



机械工业出版社
China Machine Press

本书经OpenGL结构评审委员会正式批准出版，全书全面权威地介绍了OpenGL的所有函数，OpenGL实用库（GLU 1.3）中的最新例程和对X窗口系统的OpenGL扩展（GLX 1.3）中的新增功能。此外，本书还对OpenGL的基本概念、所定义的常量及相关命令、对ARB扩展的描述等内容进行了专门介绍。

全书内容全面、结构严谨、叙述清晰，是一本有关OpenGL所有函数及命令的参考大全。对于专门从事三维图形开发的人员及高等院校师生，本书是一本必备的参考手册。

OpenGL Architecture Review Board; Dave Shreiner, editor: OpenGL Reference Manual, Third Edition : The Official Reference Document to OpenGL, Version 1.2.

Original edition copyright © 2000 by Silicon Graphics, Inc.

Chinese edition published by arrangement with Addison Wesley Longman, Inc.

All rights reserved.

未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

16486/04

本书版权登记号：图字：01-2000-0306

图书在版编目（CIP）数据

OpenGL® 参考手册 / OpenGL结构评审委员会著；孙守迁等译。—3版。—北京：机械工业出版社/辽宁教育出版社，2001.1

（软件开发技术丛书）

书名原文：OpenGL Reference Manual, Third Edition: The Official Reference Document to OpenGL, Version 1.2

ISBN

I. O… II. ①O… ②孙… III. 三维-动画-图形软件，OpenGL-技术手册 IV.TP391.41-62

中国版本图书馆CIP数据核字(2000)第52753号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：姚 蕾

山东省高青县印刷厂 印刷 新华书店北京发行所发行

2001年1月第1版第1次印刷

787mm×1092 mm 1/16 · 31.25 印张

印数：0 001-5 000册

定价：58.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

译 者 序

近年来，随着计算机技术的进步，我们跨入了一个三维时代。各种扣人心弦的三维游戏、能数字化地显示天气变化的气象服务、震撼人心的3D数字化特殊效果，无不使我们体验到三维世界的全新感觉。可视化、计算机动画、虚拟现实是当今图形学领域的三大热门话题，它们的技术核心都是三维图形。

1992年7月，SGI公司首次发布了作为三维图形编程接口的OpenGL。目前它已成为国际上通用的开放式三维图形标准。一方面，OpenGL规范由ARB（OpenGL Architecture Review Board，OpenGL结构评审委员会）负责管理，充分保证了它的独立性、开放性、前瞻性和跨平台性。它可被集成到Unix、Windows NT 4.0、Windows 98、X窗口等窗口系统中。另一方面，Compaq、IBM、Intel、Microsoft等在计算机界具有主导作用的公司纷纷采用OpenGL图形的国际标准。各种游戏加速卡、专用加速部件都能不同程度地提高OpenGL程序的运行性能。这些都推动了OpenGL的发展，并迅速成为三维图形的国际标准。再者，SGI公司不断推出以OpenGL为基础的高级开发工具，以满足对图形工具性能日益增长的需求。这一切使得OpenGL成为最流行的三维图形开发工具。目前它已被广泛应用于CAD/CAM/CAE、地质、航空、医学图像处理、广告、艺术造型、电影后期制作等领域。

OpenGL由大量功能强大的图形函数组成，它集成了所有曲面造型、图形变换、光照、材质、纹理、像素操作、融合、反选择、雾化等复杂的计算机图形学算法。开发人员可以利用这些函数对整个三维图形轻松进行渲染，从而达到数字化现实生活景象的目的。

本书是OpenGL参考手册的第3版，对OpenGL的函数进行了详细而简洁的说明，是程序员利用OpenGL进行程序开发的不可缺少的工具书。本书第1章是OpenGL入门，第2章对命令和例程进行了简介，第3章集中介绍了各种命令和例程，第4章介绍了定义的常量和相关命令，第5章是OpenGL参考说明，第6章是GLU的参考说明，第7章是GLX的参考说明。

本书在翻译过程中得到了国家863计划项目（863-511-942-016）的支持。参加翻译的人员还有王火亮、许宇荣、杨勤、杨颖、余牛、周贵仔、蒋丽、李岳梅，在此感谢他们的辛勤劳动。

由于计算机软件行业的飞速发展，加之时间仓促，翻译中难免会有不妥之处，如能得到您的及时指正将不胜感激。

我们的电子邮箱地址为caid@cs.zju.edu.cn.

译 者
2000年9月于求是园

前　　言

OpenGL是一个图形硬件的软件接口（“GL”即Graphics Library）。这一接口包含了数百个函数，图形程序员可以利用这些函数指定设计高品质的三维彩色图像所需的对象和操作。这些函数中有许多实际上是其他函数的简单变形，因此，实际上它仅包含大约180个左右完全不同的函数。

OpenGL实用库（OpenGL Utility Library，GLU）和对X窗口系统的OpenGL扩展（OpenGL Extension to the X Window System，GLX）为OpenGL提供了有用的支持特性和完整的OpenGL核心函数集。本书详细介绍了这些函数的功能。书中各章内容如下：

- 第1章 OpenGL简介

在概念上对OpenGL作了概述。它通过一个高层的模块图来阐述OpenGL所执行的所有主要处理阶段。

- 第2章 命令和例程概述

较详细地阐述了OpenGL对输入数据的处理过程（用顶点形式来指定一个几何体或用像素形式来定义一幅图像时），并告诉你如何用OpenGL函数来控制这个过程。此外，在本章中还对GLU和GLX函数作了讨论。

- 第3章 命令和例程一览

根据OpenGL命令所完成的功能列举说明了这些命令组。一旦了解了这些命令的功能，你就可以利用这些完整的函数原型作为快速参考。

- 第4章 定义的常量及相关命令

列举了在OpenGL中定义的常量和使用这些常量的命令。

- 第5章 OpenGL参考说明

本书的主体部分，它包括各组相关的OpenGL命令的描述。带参数的命令和与之一起描述的其他命令仅在数据类型方面有所不同。每个函数的参考说明介绍了参数、命令的作用和使用这些命令时可能发生的错误。

此外，本章还包含了有关OpenGL的ARB扩展——多重纹理和绘图子集的参考说明。需要说明的是并非所有的OpenGL的环境都支持ARB扩展。

- 第6章 GLU参考说明

本章包含了所有的GLU命令的参考说明。

- 第7章 GLX参考说明

本章包含了所有的GLX命令的参考说明。

0.1 阅读此书前的预备知识

本书是OpenGL Architecture Review Board，Mason Woo、Jackie Neider、Tom Davis 和 Dave Shreinre编著的《OpenGL编程指南（第3版）》(Reading, MA: Addison-Wesley, 1999)

的姊妹篇。阅读这两本书的前提是你已经懂得如何用C语言编程。

两本书的不同之处主要在于：《OpenGL编程指南》一书着重于介绍如何运用OpenGL，而本书的重点则是OpenGL的工作方式。当然要想彻底地了解OpenGL，这两方面的知识都是必需的。这两本书的另一个不同点是本书的大多数内容都是按字母次序编排的，这样编排的前提是假定你已经知道你所不明白的地方而仅仅想查找某个特定命令的用法。而《OpenGL编程指南》一书的编排则更像一本指南：它首先解释了OpenGL的简单概念，然后再导出更复杂的概念。虽然你不必通过阅读《OpenGL编程指南》一书来理解本书对命令的解释，但如果你已经读过它，你将会有对这些命令有更深刻的理解。

如果你对计算机图形学还不太了解，那么请先从《OpenGL编程指南》一书入手学习，并同时参考下面这些书：

- James D. Foley、Andries van Dam、Steven K. Feiner和John F. Hughes著，《计算机图形学：原理及应用》(Computer Graphics: Principles and Practice)。(Reading, MA: Addison-Wesley)。该书是一本计算机图形学的百科全书，它包含了丰富信息量，但最好在你对这门学科有一定的实践经验之后再读它。
- Andrew S. Glassner著，《3D计算机图形学：艺术家与设计师的用户指南》(3D Computer Graphics: A User's Guide for Artists and Designers)。(New York: Design Press)。这是一本非技术性的、综合介绍计算机图形学的书，它着重于所能获得的视觉效果而非如何获取这些效果的具体技巧。
- Olin Lathrop著，《计算机图形学的工作原理》(The Way Computer Graphics Work)。(New York: John Wiley and Sons, Inc.)。这本书概括性地介绍了计算机图形学，主要面向初级和中级计算机用户。它介绍了理解计算机图形学所必需的一般概念。

0.2 字体约定

本书使用如下的字体约定：

黑体字 (**Bold**) —— 命令和例行程序名；

斜体字 (*Italics*) —— 变量名、自变量名、参数名、空间维数和文件名；

正体字 (**Regular**) —— 枚举类型和定义的常量；

等宽字体 (**Monospace font**) —— 示例代码。

值得注意的是本书所使用的命令名称都是缩写形式。许多OpenGL命令只是其他命令的变种。简言之，这里只使用函数的基本名称。如果此命令上加有星号 (*)，则说明它所代表的实际的命令名称可能比显示的命令名称要多。如，**glVertex***代表所有指定顶点的命令变种所构成的命令。

多数命令的区别仅在于它们所带的自变量的数据类型。有些命令则在相关自变量的数目、这些自变量是否被指定为向量以及是否需在列表中单独指定等方面存在着区别。例如，你使用**glVertex2f**命令时必须以浮点数形式提供x和y的坐标；而使用**glVertex3sv**命令时你需为x, y, z提供一个包含三个短整型值的数组。

0.3 致谢

本手册的初版是许多人共同努力的结果。Silicon Graphics的Kurt Akeley, SABL Productions

的Sally Browning以及Silicon Graphics的Kevin P. Smith为第1版提供了大量的资料，另外还有Jackie Neider和Mark Segal(他们均来自Silicon Graphics)。Mark和Kurt合著《The OpenGL Graphics System: A Specification》，Kevin著《OpenGL Graphics System Utility Library》，Phil Karlton 著《OpenGL Graphics with the X Window System》为本书作者提供了文献来源。Phil Karlton和Kipp Hickman帮助在Silicon Graphics定义并创建了OpenGL，此外还有Gain Technology, Inc.的Raymond Drewry、Digital Equipment Corp.的Fred Fisher、Kubota Pacific Computer, Inc.的Randi Rost等人也为本书的编写提供了帮助。OpenGL 结构评审委员会的成员 Murray Cantor以及International Business Machines的Linas Vepstas、Digital Equipment Corporation的Paula Womack和Jeff Lane、Intel的Murali Sundaresan，还有Microsoft的Chuck Whitmer也提供了很多帮助。Thad Beier同Seth Katz以及Silicon Graphics的Inventor小组一起制作了封面图形。Silicon Graphics的Kay Maitz、Evans Technical Communications的Arthur Evans以及Susan Blau提供了产品援助，Tanya Kucak对本手册进行了编辑。当然，如果没有OpenGL，也就不会有本书的存在，所以要感谢Silicon Graphics的OpenGL小组所有成员，感谢他们的辛勤工作。他们是：Momi Akeley、Allen Akin、Chris Frazier、Bill Glazier、Paul Ho、Simon Hui、Lesley Kalmin、Pierre Tardif、Jim Winget，尤其是Wei Yen。另外，还有上面提到的Kurt、Phil、Mark、Kipp以及Kevin。当然还有许多其他的Silicon Graphics成员也为改进OpenGL的定义和功能做出了很多贡献，在这里也一并感谢他们。

Kempf的Renate Kempf及其同事、Silicon Graphics的Chris Frazier为《OpenGL Reference Manual for OpenGL, Version1.1》添加了所有OpenGL1.1 Specification中的新功能，并编辑审查了其他所有参考说明书。下列人员对该书进行了仔细的复审，他们是Allen Akin、David Blythe、Craig Dunwoody、Chris Frazier以及Silicon Graphics的Paula Womack、OpenGL 结构评审委员会中的成员，包括Silicon Graphics的Kurt Akeley、HP的Dave Arns、E&S的Bill Armstrong、Intergraph的Dale Kirkland和IBM的Bimal Poddar。Silicon Graphics的Simon Hui复审了GLX参考说明，John Spitzer复审了已校对的图形插页。

在本书中，SGI的Dave Shreiner添加了OpenGL 1.2和GLX 1.3的大部分新的功能，并在David Yu的帮助下重新修订了图面。Norman Chin重新修订了GLU 1.3的参考说明。下列人员认真地进行了手册复审这一艰巨的工作，他们是：Ron Bielaski、Steve Cunningham、Jeffery Galinovsky、Eric Haines、Mark Kilgard、Dale Kirkland、Seth Livingston、Bimal Poddar、David Nishimoto、Mike Schmitt、Scott Thompson、David Yu以及SGI的OpenGL小组的成员Craig Dunwoody、Jaya Kanajan、George Kyraizis、Jon Leech和Ken Nicholson。

尤其感谢Jon Leech，是他编辑了OpenGL 1.2.1、GLU 1.3和GLX 1.3的说明书，以及使OpenGL保持活力和生机的OpenGL ARB。同时也要感谢Laura Cooper和Dany Galgani为本手册的编写所提供的产品支持。

原书书号：ISBN 0-201-65765-1

原出版社网址：www.aw.com/cseng/

目 录

译者序	
前言	
第1章 OpenGL简介	1
1.1 OpenGL基础	1
1.1.1 OpenGL图元及命令	1
1.1.2 OpenGL是一种过程语言	1
1.1.3 OpenGL的执行模式	2
1.2 基本OpenGL操作	2
第2章 命令和例程概述	4
2.1 OpenGL处理流程	4
2.1.1 顶点	4
2.1.2 ARB绘图子集	8
2.1.3 片断	9
2.2 其他OpenGL命令	11
2.2.1 使用求值器	11
2.2.2 执行选择和反馈	11
2.2.3 显示列表的使用	12
2.2.4 模式和运行的管理	12
2.2.5 获取状态信息	12
2.3 OpenGL实用库	13
2.3.1 生成纹理操作所需的图形	13
2.3.2 坐标转换	13
2.3.3 多边形的镶嵌分块	14
2.3.4 绘制球体、圆柱和圆盘	14
2.3.5 NURBS曲线和曲面	14
2.3.6 错误处理	15
2.4 对X窗口系统的OpenGL扩展	15
2.4.1 初始化	15
2.4.2 控制绘制操作	15
第3章 命令和例程一览	18
3.1 注释	18
3.2 OpenGL命令	19
3.2.1 图元	19
3.2.2 顶点数组	19
3.2.3 坐标转换	20
3.2.4 着色与光照	20
3.2.5 剪切	21
3.2.6 光栅化	21
3.2.7 像素操作	22
3.2.8 纹理	22
3.2.9 雾	23
3.2.10 帧缓冲区操作	24
3.2.11 求值器	24
3.2.12 选择与反馈	25
3.2.13 显示列表	25
3.2.14 模式与执行	25
3.2.15 状态查询	26
3.3 ARB扩展	26
3.3.1 多重纹理	26
3.3.2 绘图子集	26
3.4 GLU例程	28
3.4.1 纹理图像	28
3.4.2 坐标转换	29
3.4.3 多边形镶嵌分块	29
3.4.4 二次对象	30
3.4.5 NURBS曲线和曲面	30
3.4.6 状态查询	31
3.5 GLX例程	31
3.5.1 初始化	31
3.5.2 控制绘图操作	31
第4章 定义的常量及相关命令	34
第5章 OpenGL参考说明	61
第6章 GLU参考说明	368
第7章 GLX参考说明	434

第1章 OpenGL简介

OpenGL是图形硬件的一个软件接口。它的主要作用是将二维或三维的对象绘入一个帧缓冲区中。对象被描述为一系列的顶点（用来定义几何对象）或像素（用来定义图像）。OpenGL对数据进行几个步骤的处理从而将其转换成像素，这些像素将在帧缓冲区中形成最终需要的图形。

本章将全面地介绍OpenGL的工作原理，包括以下两个主要部分：

- **OpenGL基础** 主要解释基本的OpenGL概念，例如什么是几何图元以及OpenGL如何实行客户端-服务器端的执行模式。
- **基本OpenGL操作** 通过一个高层的模块图来说明OpenGL在帧缓冲区中处理数据并生成相应图像的过程。

1.1 OpenGL基础

本节主要解释一些OpenGL固有的命令。

1.1.1 OpenGL图元及命令

OpenGL通过几个可选模式来绘制“图元”——点、线段或多边形。你可以对各种模式独立进行控制；也就是说，一个模式的设置并不影响其他模式的设置（尽管模式间的相互作用将影响帧缓冲区中的最后结果）。OpenGL的程序通过调用函数来指定图元、设置模式并描述其他操作。

在OpenGL中，图元由单个或多个顶点组来定义。一个顶点可以是一个点、一条线的端点或一个边的角。数据（由顶点坐标、颜色、法线、纹理坐标和边界标志所组成）与顶点是相对应的，并且每个顶点和与它相关的数据独立，按照次序，采用同样的方法进行操作。这里仅有一种情况例外，那就是当一组顶点必须被“剪切”从而使得某一特定的图元刚好在某一指定的区域内，则顶点数据可能会被修改并产生新的顶点。其剪切的类型由该组顶点所代表的图元决定。

虽然有些命令在生效前可能会有一段不确定的延时，但是OpenGL的所有命令都是依照其被接收的次序来执行的。也就是说，每个图元在被绘制完成之前，其后面的命令将不会有效。同时，这也意味着使用状态查询命令时它所返回的数据将只包含所有以前发布并已执行完毕的OpenGL命令。

1.1.2 OpenGL是一种过程语言

OpenGL从根本上说是一种过程语言而非描述性的语言：OpenGL提供了直接控制二维和三维几何体的基本操作。它包含了转换矩阵、光照方程系数、反走样方法和像素校正算子的描述。然而，OpenGL并不能直接描述或建模复杂的几何对象。

你所发布的OpenGL命令指定了怎样产生一个特定的结果（即接下来所应该采取的操作）而不是指定确切的结果。正是这种过程特性使我们能够了解OpenGL是如何工作的——只有明白了它的操作顺序才能对如何使用它有更深的理解。

1.1.3 OpenGL的执行模式

OpenGL使用了一种客户端-服务器端的模式来解释命令。应用程序（客户端）所发布的命令将通过OpenGL（服务器端）来编译和处理。服务器的操作既可以同客户端在同一台计算机上进行，又可以分别属于不同的机器。因此，从这个意义上讲，OpenGL是网络透明的。一个服务器可以维护数个GL上下文，每个上下文被封装在一个GL状态里。服务器可以同时包含几个GL上下文，每个上下文都被封装在一个GL状态里。每个客户端都可以连接到这些上下文中的任何一个。所需要的网络协议可以是扩充过的已有协议（如X Window系统）或是一个完全独立的协议。OpenGL并没有提供命令用来获取用户的输入。

窗口系统分配给帧缓冲区的资源将最终控制OpenGL命令对帧缓冲区的影响。窗口系统将决定OpenGL帧缓冲区的哪些部分可在给定的时间内被访问并将这些部分的结构传送给OpenGL。因此OpenGL中不存在配置帧缓冲区及初始化OpenGL的命令。帧缓冲区的配置是在OpenGL外，由与其相关联的窗口系统来完成的，而OpenGL的初始化则是当窗口系统为OpenGL绘图分配一个窗口时完成的。（GLX——OpenGL界面的X扩展，提供了这些功能。有关细节请参阅2.4节“对X窗口系统的OpenGL扩展”。）

1.2 基本OpenGL操作

图1-1是抽象的高层的模块图，它展示了OpenGL处理数据的过程。如图所示，命令由左边进入，然后经过了一个可被看作处理流程的处理过程。其中有一些命令指定了所要绘制的几何对象，而另一些命令则用于控制不同阶段中对象的处理方法。

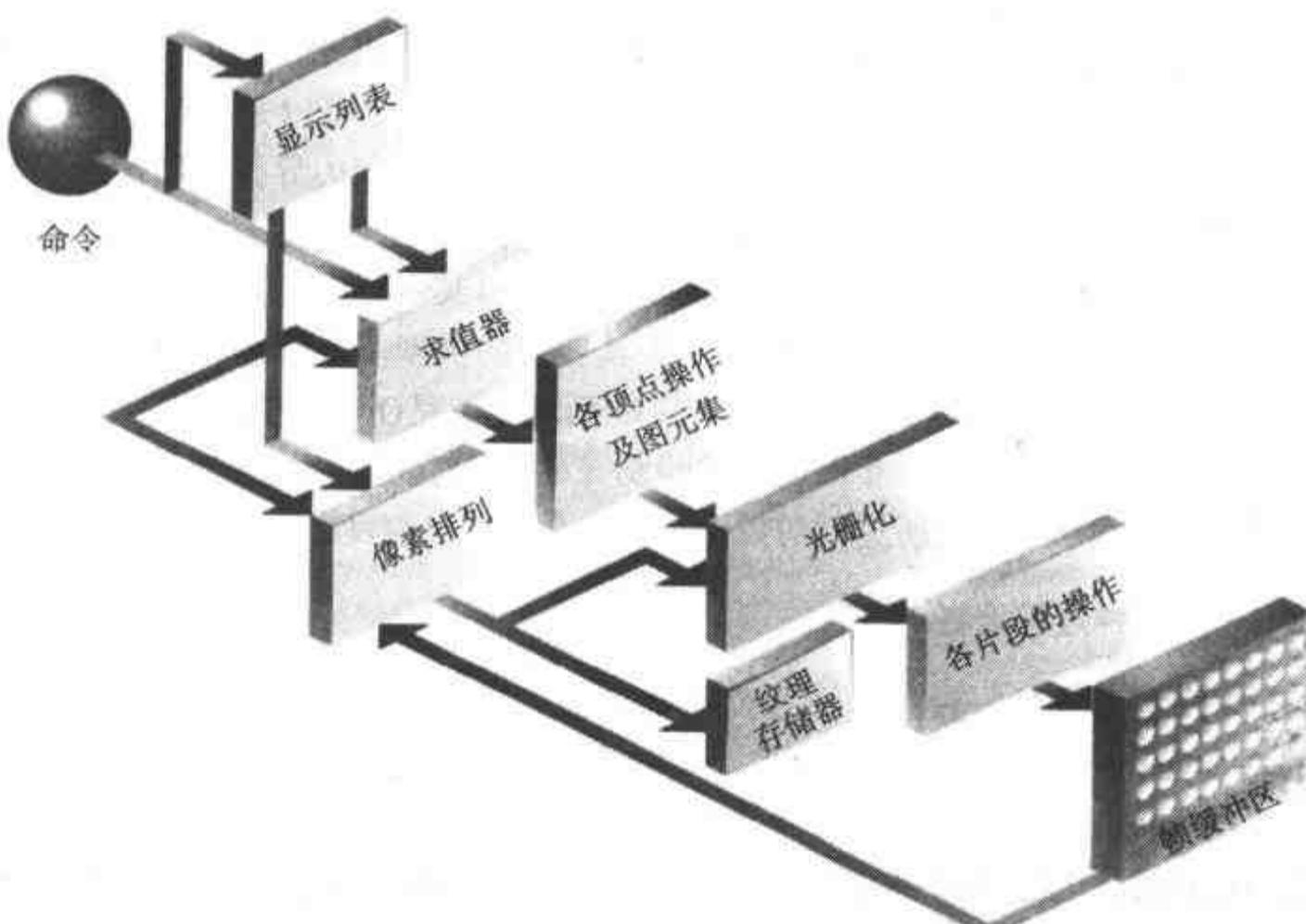


图1-1 OpenGL数据处理过程

当命令进入流程时，你可以选用两种方法对它们进行处理：一种是通过流程立即执行这些命令；另一种是将其中一些命令组织到一个“显示列表”，过一段时间再执行它们。

流程中的“求值器”阶段通过将输入值赋给多项式命令提供了一种非常有效的方法来生成几何曲线和曲面的近似值。接下来的“各顶点操作及图元集”阶段主要是处理OpenGL的几何图元——点、线段和多边形。所有这些图元均由顶点来描述。顶点可以被转换和照亮。接下来图元被剪切到视口，为下一阶段做好了准备。

“光栅化”生成了一系列的帧缓冲区地址以及相应的用于描述点、线段或多边形的二维值。这些生成的“片断”将被送到最后一个阶段——“各片断的操作”。这一阶段是数据以像素形式存入“帧缓冲区”之前的最后操作。这些操作包括根据帧缓冲区中原有的深度值（用于深度缓存操作）与输入值而有条件地更新帧缓冲区的操作，还包括对输入的像素颜色与已存储的颜色所进行的融合操作，对像素值所进行的屏蔽操作及其他逻辑操作。

数据是以像素形式而非顶点形式输入的。这些数据可以用来描述一个用于纹理映射的图像，它将跳过第一阶段（如前面所述），而通过“像素操作”阶段作为像素来处理。这一阶段的处理将导致两种结果：其一是被存入“纹理存储器”，以备光栅化阶段所用；其二是直接被光栅化。后者所形成的片断将被存入帧缓冲区，就好象它们是由几何数据生成的一样。

一个OpenGL应用程序可以获得OpenGL状态的所有元素，其中包括纹理存储器的内容，甚至还包括帧缓冲区的内容。

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。

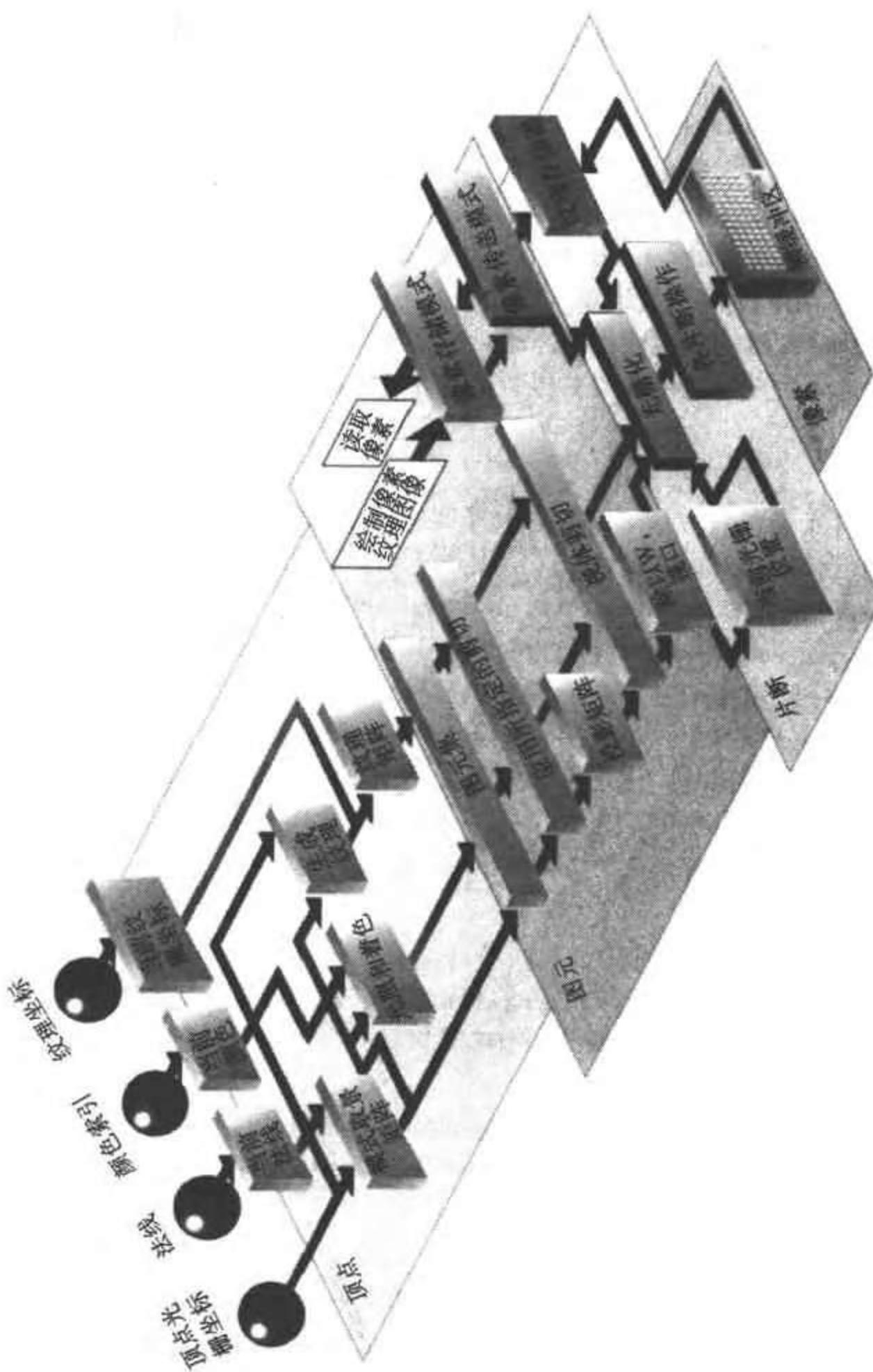


图2-1 OpenGL处理流程的各个阶段

- 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`glMatrixMode()`、`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`glPushMatrix()`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化成像素片断。

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()**指定的剪切平面而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()**来控制正而多边形和背而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**glResetHistogram()**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**glResetMinmax()**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alpha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
glGetColorTableParameter†
glGetConvolutionParameter†

glGetColorTable†
glGetConvolutionFilter†
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()**、**gluTessVertex()**和**gluTessEndPolygon()**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**gluPartialDisk()**。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**gluEndSurface()**。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**glXCreateWindow()**。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**glXCreatePixmap()**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**glXGetCurrentReadDrawable()**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**glXCopyContext()**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。

第3章 命令和例程一览

本章列出了OpenGL、OpenGL实用库以及对X窗口系统的OpenGL扩展的原型。这些原型按功能分组显示如下：

- OpenGL命令

图元	雾
顶点数组	帧缓冲区操作
坐标转换	求值器
着色与光照	选择与反馈
剪切	显示列表
光栅化	模式与执行
像素操作	状态查询
纹理	

- ARB扩展

多纹理	图形子集
-----	------

- GLU例程

纹理图形	二次对象
坐标转换	NURBS曲线和曲面
多边形镶嵌分块	状态查询

- GLX例程

初始化	控制绘制操作
-----	--------

3.1 注释

有些OpenGL命令仅在它们所接受的数据类型上存在不同。本书中使用如下的约定来简化地表示这些命令：

```
void glVertex2{sfid}{v} (TYPE x, TYPE y);
```

上例中，第一对大括号中所包含的字符用来确定数据TYPE可能的类型。（其中括号前的阿拉伯数字表示命令所包含的自变量的个数。）表3-1列出了每种可能的数据类型、其相应的字符以及OpenGL中用来表示它们的类型定义。

命令中如果包含第二对大括号，则其中的字母v表示命令中包含向量形式。当你选择使用向量形式时，所有的TYPE参数将合并为一个数组。例如，下面是一个命令的非向量形式和向量形式，它们都使用一个浮点数据类型：

```
void glVertex2f(GLfloat x, GLfloat y);
```

```
void glVertex2fv(GLfloat v[2]);
```

在向量形式的用法是不确定的时候，向量和非向量的形式都被列出。请注意并非所有多变量

的命令都有向量形式，而有些命令就只有一个向量形式，这时字母v将不包含在大括号中。

表 3-1

字 符	C语言类型	OpenGL类型定义
b	signed char	GLbyte
s	short	GLshort
i	int	GLint
f	float	GLfloat, GLclampf
d	double	GLdouble, GLclampd
ub	unsigned char	GLubyte, GLboolean
us	unsigned short	GLshort
ui	unsigned int	GLuint, GLenum, GLbitfield
	void	GLvoid

3.2 OpenGL命令

3.2.1 图元

指定顶点或矩形：

```
void glBegin (GLenum mode);
void glEnd (void);
void glVertex2{sfid}{v} (TYPE x, TYPE y);
void glVertex3{sfid}{v} (TYPE x, TYPE y, TYPE z);
void glVertex4{sfid}{v} (TYPE x, TYPE y, TYPE z, TYPE w);
void glRect{sfid} (TYPE x1, TYPE y1, TYPE x2, TYPE y2);
void glRect{sfid}v (const TYPE *v1, const TYPE *v2);
```

指定多边形边界的处理方式：

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

指定多边形的偏移量：

```
void glPolygonOffset (GLfloat factor, GLfloat units);
```

3.2.2 顶点数组

指定顶点数组：

```
void glVertexPointer (GLint size, GLenum type, GLsizei stride,
                      const GLvoid *pointer);
void glEdgeFlagPointer (GLsizei stride, const GLvoid *pointer);
void glIndexPointer (GLenum type, GLsizei stride,
                     const GLvoid *pointer);
void glColorPointer (GLint size, GLenum type, GLsizei stride,
                     const GLvoid *pointer);
void glNormalPointer (GLenum type, GLsizei stride,
                      const GLvoid *pointer);
```

```
void glTexCoordPointer (GLint size, GLenum type, GLsizei stride,
                      const GLvoid *pointer);
```

控制顶点数组及其组分的绘制：

```
void glInterleavedArrays (GLenum format, GLsizei stride,
                          const GLvoid *pointer);
void glArrayElement (GLint i);
void glDisableClientState (GLenum array);
void glEnableClientState (GLenum array);
void glDrawElements (GLenum mode, GLsizei count, GLenum type,
                     const GLvoid *indices);
void glDrawRangeElements (GLenum mode, GLuint start, GLuint end,
                          GLsizei count, GLenum type, const GLvoid *indices);
void glDrawArrays (GLenum mode, GLint first, GLsizei count);
```

存储与恢复顶点数组的值：

```
void glPushClientAttrib (GLbitfield mask);
void glPopClientAttrib (void);
```

获取指定的顶点数组的地址：

```
void glGetPointerv (GLenum pname, GLvoid **params);
```

3.2.3 坐标转换

转换当前矩阵：

```
void glRotate{fd} (TYPE angle, TYPE x, TYPE y, TYPE z);
void glTranslate{fd} (TYPE x, TYPE y, TYPE z);
void glScale{fd} (TYPE x, TYPE y, TYPE z);
void glMultMatrix{fd} (const TYPE *m);
void glFrustum (GLdouble left, GLdouble right, GLdouble bottom,
                GLdouble top, GLdouble near, GLdouble far);
void glOrtho (GLdouble left, GLdouble right, GLdouble bottom,
              GLdouble top, GLdouble near, GLdouble far);
```

替换当前矩阵：

```
void glLoadMatrix{fd} (const TYPE *m);
void glLoadIdentity (void);
```

操纵矩阵堆栈：

```
void glMatrixMode (GLenum mode);
void glPushMatrix (void);
void glPopMatrix (void);
```

指定视口：

```
void glDepthRange (GLclampd near, GLclampd far);
void glViewport (GLint x, GLint y, GLsizei width, GLsizei height);
```

3.2.4 着色与光照

设置当前颜色、颜色索引或法向量：

```
void glColor3{bsifd ubusui}{v} (TYPE red, TYPE green, TYPE blue);
void glColor4{bsifd ubusui}{v} (TYPE red, TYPE green, TYPE blue,
                               TYPE alpha);
```

指定光源、材料或光照模式参数值：

```
void glLight{if}{v} (GLenum light, GLenum pname, TYPE param);
void glMaterial{if}{v} (GLenum face, GLenum pname, TYPE param);
void glLightModel{if}{v} (GLenum pname, TYPE param);
```

选择一种阴影模式：

```
void glShadeModel (GLenum mode);
```

指定正面多边形：

```
void glFrontFace (GLenum dir);
```

使一种颜色跟踪当前颜色：

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

获取光源或材料的参数值：

```
void glGetLight{if}v (GLenum light, GLenum pname, TYPE *params);
void glGetMaterial{if}v (GLenum face, GLenum pname, TYPE *params);
```

3.2.5 剪切

指定一个剪切平面：

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

返回剪切平面系数：

```
void glGetClipPlane (GLenum plane, GLdouble *equation);
```

3.2.6 光栅化

设置当前光栅位置：

```
void glRasterPos2{sfid}{v}(TYPE x, TYPE y);
void glRasterPos3{sfid}{v}(TYPE x, TYPE y, TYPE z);
void glRasterPos4{sfid}{v}(TYPE x, TYPE y, TYPE z, TYPE w);
```

指定一个位图：

```
void glBitmap (GLsizei width, GLsizei height, GLfloat xorig, GLfloat yorig,
               GLfloat xmove, GLfloat ymove, const GLubyte *bitmap);
```

指定点或线的尺寸：

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

指定或返回线或多边形的点画模式：

```
void glLineStipple (GLint factor, GLushort pattern);
void glPolygonStipple (const GLubyte *mask);
void glGetPolygonStipple (GLubyte *mask);
```

选择如何光栅化多边形：

```
void glCullFace (GLenum mode);
```

3.2.7 像素操作

选择像素读取或复制操作的源：

```
void glReadBuffer (GLenum mode);
```

读、写或复制像素：

```
void glReadPixels ( GLint x, GLint y, GLsizei width, GLsizei height,
                    GLenum format, GLenum type, GLvoid *pixels);
void glDrawPixels ( GLsizei width, GLsizei height, GLenum format,
                    GLenum type, const GLvoid *pixels);
void glCopyPixels ( GLint x, GLint y, GLsizei width, GLsizei height,
                    GLenum type);
```

指定或查询像素的编码或处理方式：

```
void glPixelStore{if} (GLenum pname, TYPE param);
void glPixelTransfer{if} (GLenum pname, TYPE param);
void glPixelMap{f usui}v (GLenum map, GLsizei mapsiz, const TYPE *values);
void glGetPixelMap{f usui}v (GLenum map, TYPE *values);
```

控制像素的光栅化：

```
void glPixelZoom (GLfloat xfactor, GLfloat yfactor);
```

3.2.8 纹理

控制如何将一个纹理应用于片断：

```
void glTexParameter{if}{v} (GLenum target, GLenum pname, TYPE param);
void glTexEnv{if}{v} (GLenum target, GLenum pname, TYPE param);
```

设置当前纹理坐标：

```
void glTexCoord1{sifd}{v} (TYPE s);
void glTexCoord2{sifd}{v} (TYPE s, TYPE t);
void glTexCoord3{sifd}{v} (TYPE s, TYPE t, TYPE r);
void glTexCoord4{sifd}{v} (TYPE s, TYPE t, TYPE r, TYPE q);
```

控制纹理坐标的生成：

```
void glTexGen{ifd}{v} (GLenum coord, GLenum pname, TYPE param);
```

指定一个一维或二维的纹理图像或纹理子图：

```
void glTexImage1D (GLenum target, GLint level, GLint internalformat,
                   GLsizei width, GLint border, GLenum format,
                   GLenum type, const GLvoid *pixels);
void glTexImage2D (GLenum target, GLint level, GLint internalformat,
                   GLsizei width, GLsizei height, GLint border,
                   GLenum format, GLenum type, const GLvoid *pixels);
void glTexImage3D (GLenum target, GLint level, GLenum internalformat,
                   GLsizei width, GLsizei height, GLsizei depth, GLint border,
```

```

Glenum format, Glenum type, const GLvoid *pixels);
void glTexSubImage1D (GLenum target, GLint level, GLint xoffset,
    GLsizei width, GLenum format, GLenum type,
    const GLvoid *pixels);
void glTexSubImage2D (GLenum target, GLint level, GLint xoffset,
    GLint yoffset, GLsizei width, GLsizei height,
    GLenum format, GLenum type, const GLvoid *pixels);
void glTexSubImage3D (GLenum target, GLint level, GLint xoffset,
    GLint yoffset, GLint zoffset, GLsizei width, GLsizei height,
    GLsizei depth, GLenum format, GLenum type,
    const GLvoid *pixels);

```

测试一个名称是否对应于一个纹理并获取与纹理有关的参数值：

```

void glIsTexture (GLuint texture);
void glGetTexEnv{if}v (GLenum target, GLenum pname, TYPE *params);
void glGetTexGen{ifd}v (GLenum coord, GLenum pname, TYPE *params);
void glGetTexImage (GLenum target, GLint level, GLenum format,
    GLenum type, GLvoid *pixels);
void glGetTexLevelParameter{if}v (GLenum target, GLint level,
    GLenum pname, TYPE *params);
void glGetTexParameter{if}v (GLenum target, GLenum pname, TYPE *params);

```

复制一个或部分纹理：

```

void glCopyTexImage1D (GLenum target, GLint level,
    GLenum internalformat, GLint x, GLint y, GLsizei v,
    GLint border);
void glCopyTexImage2D (GLenum target, GLint level,
    GLenum internalformat, GLint x, GLint y,
    GLsizei width, GLsizei height, GLint border);
void glCopyTexSubImage1D (GLenum target, GLint level, GLint xoffset,
    GLint x, GLint y, GLsizei width);
void glCopyTexSubImage2D (GLenum target, GLint level, GLint xoffset,
    GLint yoffset, GLint x, GLint y, GLsizei width,
    GLsizei height);
void glCopyTexSubImage3D (GLenum target, GLint level, GLint xoffset,
    GLint yoffset, GLint zoffset, GLint x, GLint y,
    GLsizei width, GLsizei height);

```

建立一个指定的纹理并优化纹理存储驻留：

```

void glBindTexture (GLenum target, GLuint texture);
void glDeleteTextures (GLsizei n, const GLuint *textures);
GLboolean glAreTexturesResident (GLsizei n, const GLuint *textures,
    GLboolean *residences);
void glGenTextures (GLsizei n, GLuint *textures);
void glPrioritizeTextures (GLsizei n, const GLuint *textures,
    const GLclampf *priorities);

```

3.2.9 雾

设置雾参数：

```
void glFog{if}{v} (GLenum pname, TYPE param);
```

3.2.10 帧缓冲区操作

控制每个片断的测试：

```
void glScissor (GLint x, GLint y, GLsizei width, GLsizei height);
void glAlphaFunc (GLenum func, GLclampf ref);
void glStencilFunc (GLenum func, GLint ref, GLuint mask);
void glStencilOp (GLenum fail, GLenum pass, GLenum zpass);
void glDepthFunc (GLenum func);
```

结合片断与帧缓冲区中的值：

```
void glBlendFunc (GLenum sfactor, GLenum dfactor);
void glLogicOp (GLenum opcode);
```

清除部分或全部缓冲区：

```
void glClear (GLbitfield mask);
```

指定清除操作所需的颜色、深度和模板值：

```
void glClearAccum (GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha);
void glClearColor ( GLclampf red, GLclampf green, GLclampf blue,
                  GLclampf alpha);
void glClearDepth (GLclampd depth);
void glClearIndex (GLfloat c);
void glClearStencil (GLint s);
```

控制向缓冲区中写入：

```
void glDrawBuffer (GLenum mode);
void glIndexMask (GLuint mask);
void glColorMask ( GLboolean red, GLboolean green, GLboolean blue,
                  GLboolean alpha);
void glDepthMask (GLboolean flag);
void glStencilMask (GLuint mask);
```

操作累积缓冲区：

```
void glAccum (GLenum op, GLfloat value);
```

3.2.11 求值器

定义一个一维或二维求值器：

```
void glMap1{fd} (GLenum target, TYPE u1, TYPE u2, GLint stride,
                 GLint order, const TYPE *points);
void glMap2{fd} (GLenum target, TYPE u1, TYPE u2, GLint ustride,
                 GLint uorder, TYPE v1, TYPE v2, GLint vstride,
                 GLint vorder, const TYPE *points);
```

生成并求取一系列映射的域值：

```
void glMapGrid1{fd} (GLint n, TYPE u1, TYPE u2>);
void glMapGrid2{fd} (GLint un, TYPE u1, TYPE u2, GLint vn, TYPE v1,
                     TYPE v2);
void glEvalMesh1 (GLenum mode, GLint i1, GLint i2);
void glEvalMesh2 (GLenum mode, GLint i1, GLint i2, GLint j1, GLint j2);
void glEvalPoint1 (GLint i);
void glEvalPoint2 (GLint i, GLint j);
```

在一个指定的域坐标中求取一维或二维映射值：

```
void glEvalCoord1{fd}{v} (TYPE u);
void glEvalCoord2{fd}{v} (TYPE u, TYPE v);
```

获取求值器的参数值：

```
void glGetMap{idf}v (GLenum target, GLenum query, TYPE *v);
```

3.2.12 选择与反馈

控制模式及相应的缓冲区：

```
GLint glRenderMode (GLenum mode);
void glSelectBuffer (GLsizei size, GLuint *buffer);
void glFeedbackBuffer (GLsizei size, GLenum type, GLfloat *buffer);
```

为反馈模式提供一个标记：

```
void glPassThrough (GLfloat token);
```

控制选取模式中的名称堆栈：

```
void glInitNames (void);
void glLoadName (GLuint name);
void glPushName (GLuint name);
void glPopName (void);
```

3.2.13 显示列表

建立或删除显示列表：

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

执行一个或一组显示列表：

```
void glCallList (GLuint list);
void glCallLists (GLsizei n, GLenum type, const GLvoid *lists);
```

管理显示列表的索引：

```
GLuint glGenLists (GLsizei range);
GLboolean glIsList (GLuint list);
void glListBase (GLuint base);
```

3.2.14 模式与执行

启动、关闭及查询模式：

```
void glEnable (GLenum cap);
void glDisable (GLenum cap);
GLboolean glIsEnabled (GLenum cap);
```

等待，直到所有OpenGL命令执行完成：

```
void glFinish (void);
```

强迫执行所有已发布的OpenGL命令：

```
void glFlush (void);
```

指定OpenGL操作的提示：

```
void glHint (GLenum target, GLenum mode);
```

3.2.15 状态查询

获取有关错误及OpenGL当前连接的信息：

```
GLenum glGetError (void);
const GLubyte * glGetString (GLenum name);
```

查询状态变量：

```
void glGetBooleanv (GLenum pname, GLboolean *params);
void glGetDoublev (GLenum pname, GLdouble *params);
void glGetFloatv (GLenum pname, GLfloat *params);
void glGetIntegerv (GLenum pname, GLint *params);
```

存储与恢复状态变量集：

```
void glPushAttrib (GLbitfield mask);
void glPopAttrib (void);
```

3.3 ARB扩展

3.3.1 多重纹理

设置当前纹理坐标：

```
void glMultiTexCoord1{sifd}{v}ARB (GLenum target, TYPE s);
void glMultiTexCoord2{sifd}{v}ARB (GLenum target, TYPE s, TYPE t);
void glMultiTexCoord3{sifd}{v}ARB (GLenum target, TYPE s, TYPE t, TYPE r);
void glMultiTexCoord4{sifd}{v}ARB (GLenum target, TYPE s, TYPE t,
                                  TYPE r, TYPE q);
```

选择当前有效的纹理单元：

```
void glActiveTextureARB (GLenum texture);
```

选择当前有效的顶点数组：

```
void glClientActiveTextureARB (GLenum texture);
```

3.3.2 绘图子集

1. 颜色表

指定一个颜色查询表或子表：

```
void glColorTable (GLenum target, GLenum internalformat, GLsizei width,
                  GLenum format, GLenum type, const GLvoid *table);
void glColorSubTable (GLenum target, GLsizei start, GLsizei count,
                     GLenum format, GLenum type, const GLvoid *data);
```

复制一个颜色查询表或子表：

```
void glCopyColorTable (GLenum target, GLenum internalformat, GLint x,
                      GLint y, GLsizei width);
void glCopyColorSubTable (GLenum target, GLsizei start, GLint x, GLint y,
                         GLsizei width);
```

获取颜色表的相应值：

```
void glGetColorTable (GLenum target, GLenum format, GLenum type,
                     GLvoid *table);
void glGetColorTableParameter{if}v (GLenum target, GLenum pname,
                                   TYPE *params);
```

2. 卷积

控制如何将一个卷积滤波器应用子输入的图像：

```
void glConvolutionParameter{if}{v} (GLenum target, GLenum pname,
                                   TYPE params);
```

指定一维或二维卷积滤波器：

```
void glConvolutionFilter1D (GLenum target, GLenum internalformat,
                           GLsizei width, GLenum format, GLenum type,
                           const GLvoid *image);
void glConvolutionFilter2D (GLenum target, GLenum internalformat,
                           GLsizei width, GLsizei height, GLenum format,
                           GLenum type, const GLvoid *image);
```

指定二维可分离卷积滤波器：

```
void glSeparableFilter2D (GLenum target, GLenum internalformat,
                          GLsizei width, GLsizei height, GLenum format,
                          GLenum type, const GLvoid *row, const GLvoid *column);
```

获取卷积的相关参数值：

```
void glGetConvolutionFilter (GLenum target, GLenum format, GLenum type,
                            GLvoid *image);
void glGetConvolutionParameter{if}v (GLenum target, GLenum pname,
                                   TYPE *params);
void glGetSeparableFilter (GLenum target, GLenum format, GLenum type,
                          GLvoid *row, GLvoid *column, GLvoid *span);
```

复制一个或部分卷积滤波器：

```
void glCopyConvolutionFilter1D (GLenum target, GLenum internalformat,
                               GLint x, GLint y, GLsizei width);
void glCopyConvolutionFilter2D (GLenum target, GLenum internalformat,
                               GLint x, GLint y, GLsizei width, GLsizei height);
```

3. 直方图

指定直方图格式：

```
void glHistogram (GLenum target, GLsizei width, GLenum internalformat,
                  GLboolean sink);
```

获取直方图的值与参数：

```
void glGetHistogram (GLenum target, GLboolean reset, GLenum format,
                     GLenum type, GLvoid *values);
void glGetHistogramParameterfv (GLenum target, GLenum pname,
                                 GLfloat *params);
```

重新设置内部直方图表：

```
void glResetHistogram (GLenum target);
```

4. 最值

指定最值格式：

```
void glMinmax (GLenum target, GLenum internalformat, GLboolean sink);
```

获取最值及参数：

```
void glGetMinmax (GLenum target, GLboolean reset, GLenum format,
                   GLenum type, GLvoid *values);
void glGetMinmaxParameterfv (GLenum target, GLenum pname,
                             GLfloat *params);
```

重新设置内部最值表：

```
void glResetMinmax (GLenum target);
```

3.4 GLU例程

3.4.1 纹理图像

缩放一个图像：

```
int gluScaleImage (GLenum format, GLint widthin, GLint heightin,
                   GLenum typein, const void *datain, GLint widthout,
                   GLint heightout, GLenum typeout, void *dataout);
```

生成一个图像的mipmap：

```
int gluBuild1DMipmaps (GLenum target, GLint internalformat, GLsizei width,
                       GLenum format, GLenum type, const void *data);
int gluBuild2DMipmaps (GLenum target, GLint internalformat, GLsizei width,
                       GLint height, GLenum format, GLenum type,
                       const void *data);
int gluBuild3DMipmaps (GLenum target, GLint internalformat, GLsizei width,
                       GLsizei height, GLsizei depth, GLenum format,
                       GLenum type, const void *data);
```

生成一个图像的mipmap图层的范围：

```
int gluBuild1DMipmapLevels (GLenum target, GLint internalformat,
                            GLsizei width, GLenum format, GLenum type,
                            GLint level, GLint base, GLint max, const void *data);
int gluBuild2DMipmapLevels (GLenum target, GLint internalformat,
                            GLsizei width, GLsizei height, GLenum format,
                            GLenum type, GLint level, GLint base, GLint max,
                            const void *data);
```

```
int gluBuild3DMipmapLevels (GLenum target, GLint internalformat,
                           GLsizei width, GLsizei height, GLsizei depth,
                           GLenum format, GLenum type, GLint level,
                           GLint base, GLint max, const void *data);
```

3.4.2 坐标转换

建立投影或观察矩阵：

```
void gluOrtho2D (GLdouble left, GLdouble right, GLdouble bottom,
                  GLdouble top);
void gluPerspective (GLdouble fovy, GLdouble aspect, GLdouble zNear,
                     GLdouble zFar);
void gluPickMatrix (GLdouble x, GLdouble y, GLdouble width,
                    GLdouble height, GLint viewport[4]);
void gluLookAt (GLdouble eyex, GLdouble eyey, GLdouble eyez,
                GLdouble centerx, GLdouble centery, GLdouble centerz,
                GLdouble upx, GLdouble upy, GLdouble upz);
```

将对象坐标转换成屏幕坐标：

```
int gluProject (GLdouble objx, GLdouble objy, GLdouble objz,
                const GLdouble modelMatrix[16],
                const GLdouble projMatrix[16],
                const GLint viewport[4], GLdouble *winx,
                GLdouble *winy, GLdouble *winz);
int gluUnProject (GLdouble winx, GLdouble winy, GLdouble winz,
                  const GLdouble modelMatrix[16],
                  const GLdouble projMatrix[16],
                  const GLint viewport[4], GLdouble *objx,
                  GLdouble *objy, GLdouble *objz);
int gluUnProject4 (GLdouble winx, GLdouble winy, GLdouble winz,
                   GLdouble clipw, const GLdouble modelMatrix[16],
                   const GLdouble projMatrix[16],
                   const GLint viewport[4], GLdouble near,
                   GLdouble far, GLdouble* objx, GLdouble* objy,
                   GLdouble* objz, GLdouble *objw);
```

3.4.3 多边形镶嵌分块

管理镶嵌分块对象：

```
GLUtesselator* gluNewTess (void);
void gluTessCallback (GLUtesselator *tobj, GLenum which, void (*fn)());
void gluDeleteTess (GLUtesselator *tobj);
void gluGetTessProperty (GLUtesselator* tess, GLenum which,
                        GLdouble* data);
```

描述输入的多边形：

```
void gluTessBeginPolygon (GLUtesselator *tobj);
void gluTessEndPolygon (GLUtesselator *tobj);
void gluTessBeginContour (GLUtesselator *tess);
```

```

void gluTessEndContour (GLUtesselator *tess);
void gluTessVertex (GLUtesselator *tobj, GLdouble v[3], void *data);
void gluTessNormal (GLUtesselator *tess, GLdouble valuex, GLdouble valuey,
                      GLdouble valuez);
void gluTessProperty (GLUtesselator *tess, GLenum which, GLdouble data);

```

3.4.4 二次对象

管理二次对象：

```

GLUquadric* gluNewQuadric (void);
void gluDeleteQuadric (GLUquadric *state);
void gluQuadricCallback (GLUquadric *qobj, GLenum which, void (*fn)());

```

控制绘图操作：

```

void gluQuadricNormals (GLUquadric *quadObject, GLenum normals);
void gluQuadricTexture (GLUquadric *quadObject,
                      GLboolean textureCoords);
void gluQuadricOrientation (GLUquadric *quadObject,
                      GLenum orientation);
void gluQuadricDrawStyle (GLUquadric *quadObject, GLenum drawStyle);

```

指定一个二次图元：

```

void gluCylinder (GLUquadric *qobj, GLdouble baseRadius,
                      GLdouble topRadius, GLdouble height, GLint slices,
                      GLint stacks);
void gluDisk (GLUquadric *qobj, GLdouble innerRadius,
                      GLdouble outerRadius, GLint slices, GLint loops);
void gluPartialDisk (GLUquadric *qobj, GLdouble innerRadius,
                      GLdouble outerRadius, GLint slices, GLint loops,
                      GLdouble startAngle, GLdouble sweepAngle);
void gluSphere (GLUquadric *qobj, GLdouble radius, GLint slices,
                      GLint stacks);

```

3.4.5 NURBS曲线和曲面

管理一个NURBS对象：

```

GLUnurbs* gluNewNurbsRenderer (void);
void gluDeleteNurbsRenderer (GLUnurbs *nobj);
void gluNurbsCallback (GLUnurbs *nobj, GLenum which, void (*fn)());
void gluNurbsCallbackData (GLUnurbs *nurb, GLvoid *userData);

```

创建一条NURBS曲线：

```

void gluBeginCurve (GLUnurbs *nobj);
void gluEndCurve (GLUnurbs *nobj);
void gluNurbsCurve (GLUnurbs *nobj, GLint nknots, GLfloat *knot,
                      GLint stride, GLfloat *ctlarray,
                      GLint order, GLenum type);

```

创建一个NURBS曲面：

```

void gluBeginSurface (GLUnurbs *nobj);
void gluEndSurface (GLUnurbs *nobj);
void gluNurbsSurface (GLUnurbs *nobj, GLint uknot_count,
                      GLfloat *uknot, GLint vknot_count, GLfloat *vknot,
                      GLint u_stride, GLint v_stride, GLfloat *ctlarray,
                      GLint sorder, GLint torder, GLenum type);

```

定义一个修剪区域：

```

void gluBeginTrim (GLUnurbs *nobj);
void gluEndTrim (GLUnurbs *nobj);
void gluPwlCurve (GLUnurbs *nobj, GLint count, GLfloat *array,
                      GLint stride, GLenum type);

```

控制NURBS绘图：

```

void gluLoadSamplingMatrices (GLUnurbs *nobj,
                      const GLfloat modelMatrix[16],
                      const GLfloat projMatrix[16],
                      const GLint viewport[4]);
void gluNurbsProperty (GLUnurbs *nobj, GLenum property, GLfloat value);
void gluGetNurbsProperty (GLUnurbs *nobj, GLenum property,
                      GLfloat *value);

```

3.4.6 状态查询

由一个OpenGL出错代码生成一个出错字符串或描述GLU的版本或扩展：

```

const GLubyte* gluErrorString (GLenum errorCode);
const GLubyte* gluGetString (GLenum name);
GLboolean gluCheckExtension (const GLubyte *extName,
                      const GLubyte *extString);

```

3.5 GLX例程

3.5.1 初始化

确定GLX扩展是否已经在X服务器上被定义：

```

Bool glXQueryExtension (Display *dpy, int *errorBase, int *eventBase);
Bool glXQueryVersion (Display *dpy, int *major, int *minor);

```

获取所需的视觉环境信息：

```

XVisualInfo* glXChooseVisual (Display *dpy, int screen, int *attribList);
int glXGetConfig (Display *dpy, XVisualInfo *vis, int attrib, int *value);

```

查询服务器端或客户端所支持的特性：

3.5.2 控制绘图操作

管理或查询一个OpenGL绘图环境：

```

GLXContext glXCreateContext (Display *dpy, XVisualInfo *vis,
                           GLXContext shareList, Bool direct);
void glXDestroyContext (Display *dpy, GLXContext ctx);
void glXCopyContext (Display *dpy, GLXContext src,
                     GLXContext dst, GLuint mask);
Bool glXIsDirect (Display *dpy, GLXContext ctx);
Bool glXMakeCurrent (Display *dpy, GLXDrawable draw, GLXContext ctx);
GLXContext glXGetCurrentContext (void);
GLXDrawable glXGetCurrentDrawable (void);

```

执行屏幕外的绘制：

```

GLXPixmap glXCreateGLXPixmap (Display *dpy, XVisualInfo *vis,
                               Pixmap pixmap);
void glXDestroyGLXPixmap (Display *dpy, GLXPixmap pix);

```

同步执行：

```

void glXWaitGL (void);
void glXWaitX (void);

```

交换前后缓冲区：

```

void glXSwapBuffers (Display *dpy, Window window);

```

使用一种X字体：

```

void glXUseXFont (Font font, int first, int count, int listBase);

```

注：在GLX1.3机制中，下面的命令为前面所列的命令提供了一组高级功能。

查询或请求关于帧缓冲区或视觉环境的信息：

```

GLXFBConfig* glXGetFBConfigs (Display *dpy, int screen, int *nElements);
GLXFBConfig* glXChooseFBConfig (Display *dpy, int screen,
                                 const int *attribList, int *nElements);
int glXGetFBConfigAttrib (Display *dpy, GLXFBConfig config, int attribute,
                           int *value);
XVisualInfo* glXGetVisualFromFBConfig (Display *dpy, GLXFBConfig config);

```

管理GLX的可绘区域：

```

GLXWindow glXCreateWindow (Display *dpy, GLXFBConfig config,
                            Window window, const int *attribList);
void glXDestroyWindow (Display *dpy, GLXWindow window);
GLXPixmap glXCreatePixmap (Display *dpy, GLXFBConfig config,
                            Pixmap pixmap, const int *attribList);
void glXDestroyPixmap (Display *dpy, GLXPixmap pixmap);
GLXPbuffer glXCreatePbuffer (Display *dpy, GLXFBConfig config,
                             const int *attribList);
void glXDestroyPbuffer (Display *dpy, GLXPbuffer pbuffer);
void glXQueryDrawable (Display *dpy, GLXDrawable drawable,
                      int attribute, unsigned int *value);

```

管理OpenGL绘图环境：

第4章 定义的常量及相关命令

本章列出了OpenGL中所定义的所有常量及其相关命令。一个常量可以代表一个参数的名称、一个参数的值、一种模式、一个查询目标或一个返回值。你可以将本章中的表4-1作为本书参考章节的一种索引：如果你记得一个常量的名称，那么你就可以用这张表来查出哪些函数使用了这个常量，从而就可以通过这些函数的参考说明来了解更多的信息。值得注意的是本章所有列出的常数都是可被相应的命令直接使用的，在这些函数的参考说明中还列出了你可能会感兴趣的其他一些命令。

表 4-1

常量	相关命令
GL_2D, GL_3D, GL_3D_COLOR, GL_3D_COLOR_TEXTURE, GL_4D_COLOR_TEXTURE	glFeedbackBuffer()
GL_2_BYTES, GL_3_BYTES, GL_4_BYTES	glCallLists()
GL_ACCUM	glAccum()
GL_ACCUM_ALPHA_BITS, GL_ACCUM_BLUE_BITS	glGet*
GL_ACCUM_BUFFER_BIT	glClear(), glPushAttrib()
GL_ACCUM_CLEAR_VALUE, GL_ACCUM_GREEN_BITS, GL_ACCUM_RED_BITS	glGet*
GL_ACTIVE_TEXTURE_ARB	glGet()
GL_ADD	glAccum(), glTexEnv*
GL_ALIASED_LINE_WIDTH_GRANULARITY, GL_ALIASED_LINE_WIDTH_RANGE, GL_ALIASED_POINT_SIZE_GRANULARITY, GL_ALIASED_POINT_SIZE_RANGE	glGet()
GL_ALL_ATTRIB_BITS	glPushClientAttrib()
GL_ALPHA, GL_ALPHA4, GL_ALPHA8, GL_ALPHA12, GL_ALPHA16	glColorSubTable(), glColorTable(), glConvolutionFilter1D(), glConvolutionFilter2D(), glCopyColorSubTable(), glCopyConvolutionFilter1D(),

(续)

常量	相关命令
GL_ALPHA, GL_ALPHA4, GL_ALPHA8, GL_ALPHA12, GL_ALPHA16	glColorSubTable(), glColorTable(), glConvolutionFilter1D(), glConvolutionFilter2D(), glCopyColorSubTable(), glCopyConvolutionFilter1D(), glCopyConvolutionFilter2D(), glDrawPixels(), glGetColorTable(), glGetConvolutionFilter(), glGetHistogram(), glGetMinmax(), glGetSeparableFilter(), glGetTexImage(), glHistogram(), glMinmax(), glReadPixels(), glResetMinmax(), glSeparableFilter2D(), glTexImage1D(), glTexImage2D(), glTexImage3D(), glTexSubImage1D(), glTexSubImage2D(), glTexSubImage3D()
GL_ALPHA_BIAS	glGet*(), glPixelTransfer*()
GL_ALPHA_BITS	glGet*()
GL_ALPHA_SCALE	glGet*(), glPixelTransfer*()
GL_ALPHA_TEST	glDisable(), glEnable(), glGet*(), glIsEnabled()
GL_ALPHA_TEST_FUNC, GL_ALPHA_TEST_REF	glGet*()
GL_ALWAYS	glAlphaFunc(), glDepthFunc(), glStencilFunc()
GL_AMBIENT	glColorMaterial(), glGetLight*(), glGetMaterial*(), glLight*(), glMaterial*()
GL_AMBIENT_AND_DIFFUSE	glColorMaterial(), glGetMaterial*(), glMaterial*()
GL_AND, GL_AND_INVERTED, GL_AND_REVERSE	glLogicOp()
GL_ATTRIB_STACK_DEPTH	glGet*()

(续)

常量	相关命令
<code>GL_AUTO_NORMAL</code>	<code>glDisable()</code> , <code> glEnable()</code> , <code> glGet*()</code> , <code> glIsEnabled()</code>
<code>GL_AUX0</code> through <code>GL_AUX3</code>	<code>glDrawBuffer()</code> , <code> glReadBuffer()</code>
<code>GL_AUX_BUFFERS</code>	<code> glGet*()</code>
<code>GL_BACK</code>	<code> glColorMaterial()</code> , <code> glCullFace()</code> , <code> glDrawBuffer()</code> , <code> glGetMaterial*</code> (<code>),</code> <code> glMaterial*</code> (<code>),</code> <code> glPolygonMode()</code> , <code> glReadBuffer()</code>
<code>GL_BACK_LEFT</code> , <code>GL_BACK_RIGHT</code>	<code>glDrawBuffer()</code> , <code> glReadBuffer()</code>
<code>GL_BGR</code> , <code>GL_BGRA</code>	<code> glColorSubTable()</code> , <code> glColorTable()</code> , <code> glConvolutionFilter1D()</code> , <code> glConvolutionFilter2D()</code> , <code> glDrawPixels()</code> , <code> glGetColorTable()</code> , <code> glGetConvolutionFilter()</code> , <code> glGetHistogram()</code> , <code> glGetMinmax()</code> , <code> glGetSeparableFilter()</code> , <code> glGetTexImage()</code> , <code> glGetHistogram()</code> , <code> glReadPixels()</code> , <code> glResetMinmax()</code> , <code> glSeparableFilter2D()</code> , <code> glTexImage1D()</code> , <code> glTexImage2D()</code> , <code> glTexImage3D()</code> , <code> glTexSubImage1D()</code> , <code> glTexSubImage2D()</code> , <code> glTexSubImage3D()</code>
<code>GL_BITMAP</code>	<code> glDrawPixels()</code> , <code> glGetTexImage()</code> , <code> glReadPixels()</code> , <code> glTexImage1D()</code> , <code> glTexImage2D()</code> , <code> glTexSubImage1D()</code> , <code> glTexSubImage2D()</code>
<code>GL_BITMAP_TOKEN</code>	<code> glPassThrough()</code>
<code>GL_BLEND</code>	<code> glDisable()</code> , <code> glEnable()</code> , <code> glGet*()</code> , <code> glIsEnabled()</code> , <code> glTexEnv*</code> (<code>)</code>
<code>GL_BLEND_COLOR</code> , <code>GL_BLEND_DST</code> , <code>GL_BLEND_EQUATION</code> , <code>GL_BLEND_SRC</code>	<code> glGet*()</code>
<code>GL_BLUE</code>	<code> glColorSubTable()</code> , <code> glColorTable()</code> , <code> glConvolutionFilter1D()</code>

(续)

常量	相关命令
	glConvolutionFilter2D(), glDrawPixels(), glGetColorTable(), glGetConvolutionFilter(), glGetHistogram(), glGetMinmax(), glGetSeparableFilter(), glGetTexImage(), glHistogram(), glReadPixels(), glResetMinmax(), glSeparableFilter2D(), glTexImage1D(), glTexImage2D(), glTexImage3D(), glTexSubImage1D(), glTexSubImage2D(), glTexSubImage3D()
GL_BLUE_BIAS	glGet*, glPixelTransfer*
GL_BLUE_BITS	glGet*
GL_BLUE_SCALE	glGet*, glPixelTransfer*
GL_BYTE	glCallLists(), glColorPointer(), glColorSubTable(), glColorTable(), glConvolutionFilter1D(), glConvolutionFilter2D(), glDrawPixels(), glGetColorTable(), glGetConvolutionFilter(), glGetHistogram(), glGetMinmax(), glGetSeparableFilter(), glGetTexImage(), glHistogram(), glNormalPointer(), glReadPixels(), glResetHistogram(), glResetMinmax(), glSeparableFilter2D(), glTexImage1D(), glTexImage2D(), glTexImage3D(), glTexSubImage1D(), glTexSubImage2D(), glTexSubImage3D()
GL_C3F_V3F, GL_C4F_N3F_V3F, GL_C4UB_V2F, GL_C4UB_V3F	glInterleavedArrays()

(续)

常量	相关命令
GL_CCW	glFrontFace()
GL_CLAMP, GL_CLAMP_TO_EDGE	glTexParameter*
GL_CLEAR	glLogicOp()
GL_CLIENT_ACTIVE_TEXTURE_ARB	glGet*
GL_CLIENT_ALL_ATTRIB_BITS	glPushClientAttrib()
GL_CLIENT_ATTRIB_STACK_DEPTH	glGet*
GL_CLIENT_PIXEL_STORE_BIT, GL_CLIENT_VERTEX_ARRAY_BIT	glPushClientAttrib()
GL_CLIP_PLANE0 through GL_CLIP_PLANES5	glClipPlane(), glDisable(), glEnable(), glGet*, glGetClipPlane(), glIsEnabled()
GL_COEFF	glGetMap*
GL_COLOR	glCopyPixels(), glMatrixMode()
GL_COLOR_ARRAY	glDisableClientState(), glEnableClientState(), glGet*, glIsEnabled()
GL_COLOR_ARRAY_POINTER	glGetPointerv()
GL_COLOR_ARRAY_SIZE, GL_COLOR_ARRAY_STRIDE, GL_COLOR_ARRAY_TYPE	glGet*
GL_COLOR_BUFFER_BIT	glClear(), glPushAttrib()
GL_COLOR_CLEAR_VALUE	glGet*
GL_COLOR_INDEX	glDrawPixels(), glReadPixels(), glTexImage1D(), glTexImage2D(), glTexImage3D(), glTexSubImage1D(), glTexSubImage2D(), glTexSubImage3D()
GL_COLOR_INDEXES	glGetMaterial*, glMaterial*
GL_COLOR_LOGIC_OP, GL_COLOR_MATERIAL	glDisable(), glEnable(), glGet*, glIsEnabled()
GL_COLOR_MATERIAL_FACE, GL_COLOR_MATERIAL_PARAMETER, GL_COLOR_MATRIX,	glGet*

(续)

常量	相关命令
GL_COLOR_MATRIX_STACK_DEPTH	
GL_COLOR_TABLE	glColorSubTable(), glColorTable(), glColorTableParameter*(), glCopyColorSubTable(), glCopyColorTable(), glDisable(), glEnable(), glGet*(), glGetColorTable(), glGetColorTableParameter*(), glIsEnabled()
GL_COLOR_TABLE_ALPHA_SIZE	glGetColorTableParameter*()
GL_COLOR_TABLE_BIAS	glColorTableParameter*(), glGetColorTableParameter*()
GL_COLOR_TABLE_BLUE_SIZE, GL_COLOR_TABLE_FORMAT, GL_COLOR_TABLE_GREEN_SIZE, GL_COLOR_TABLE_INTENSITY_SIZE, GL_COLOR_TABLE_LUMINANCE_SIZE, GL_COLOR_TABLE_RED_SIZE	glGetColorTableParameter*()
GL_COLOR_TABLE_SCALE	glColorTableParameter*(), glGetColorTableParameter*()
GL_COLOR_TABLE_WIDTH	glGetColorTableParameter*()
GL_COLOR_WRITEMASK	glGet*()
GL_COMPILE, GL_COMPILE_AND_EXECUTE	glNewList()
GL_CONSTANT_ALPHA	glBlendFunc()
GL_CONSTANT_ATTENUATION	glGetLight*(), glLight*()
GL_CONSTANT_BORDER	glConvolutionParameter*()
GL_CONSTANT_COLOR	glBlendFunc()
GL_CONVOLUTION_1D	glConvolutionFilter1D(), glConvolutionParameter*(), glCopyConvolutionFilter1D(), glDisable(), glEnable(), glGetConvolutionFilter(), glGetConvolutionParameter*(), glIsEnabled()
GL_CONVOLUTION_2D	glConvolutionFilter2D(), glConvolutionParameter*(), glCopyConvolutionFilter2D()

(续)

常量	相关命令
	glDisable(), glEnable(), glGetConvolutionFilter(), glGetConvolutionParameter*(), glIsEnabled()
GL_CONVOLUTION_BORDER_COLOR GL_CONVOLUTION_BORDER_MODE, GL_CONVOLUTION_FILTER_BIAS, GL_CONVOLUTION_FILTER_SCALE	glConvolutionParameter*(), glGetConvolutionParameter*() glConvolutionParameter*(), glGetConvolutionParameter*()
GL_CONVOLUTION_FORMAT, GL_CONVOLUTION_HEIGHT, GL_CONVOLUTION_WIDTH	glGetConvolutionParameter*()
GL_COPY, GL_COPY_INVERTED	glLogicOp()
GL_COPY_PIXEL_TOKEN	glPassThrough()
GL_CULL_FACE	glDisable(), glEnable(), glGet*(), glIsEnabled()
GL_CULL_FACE_MODE	glGet*()
GL_CURRENT_BIT	glPushAttrib()
GL_CURRENT_COLOR, GL_CURRENT_INDEX, GL_CURRENT_NORMAL, GL_CURRENT_RASTER_COLOR, GL_CURRENT_RASTER_DISTANCE, GL_CURRENT_RASTER_INDEX, GL_CURRENT_RASTER_POSITION, GL_CURRENT_RASTER_POSITION_VALID, GL_CURRENT_RASTER_TEXTURE_COORDS, GL_CURRENT_TEXTURE_COORDS	glGet*()
GL_CW	glFrontFace()
GL_DECAL	glTexEnv*()
GL_DECR	glStencilOp()
GL_DEPTH	glCopyPixels()
GL_DEPTH_BIAS	glGet*, glPixelTransfer*
GL_DEPTH_BITS	glGet*()
GL_DEPTH_BUFFER_BIT	glClear(), glPushAttrib()
GL_DEPTH_CLEAR_VALUE	glGet*()

(续)

常量	相关命令
GL_DEPTH_COMPONENT	glDrawPixels(), glReadPixels()
GL_DEPTH_FUNC, GL_DEPTH_RANGE	glGet*()
GL_DEPTH_SCALE	glGet*(), glPixelTransfer*
GL_DEPTH_TEST	glDisable(), glEnable(), glGet*(), glIsEnabled()
GL_DEPTH_WRITEMASK	glGet*()
GL_DIFFUSE	glColorMaterial(), glGetLight*
GL_DITHER	glDisable(), glEnable(), glGet*, glIsEnabled()
GL_DOMAIN	glGetMap*
GL_DONT_CARE	glHint()
GL_DOUBLE	glColorPointer(), glIndexPointer(), glNormalPointer(), glTexCoordPointer(), glVertexPointer()
GL_DOUBLEBUFFER, GL_DRAW_BUFFER	glGet*
GL_DRAW_PIXEL_TOKEN	glPassThrough()
GL_DST_ALPHA, GL_DST_COLOR	glBlendFunc()
GL_EDGE_FLAG	glGet*
GL_EDGE_FLAG_ARRAY	glDisableClientState(), glEnableClientState(), glGet*, glIsEnabled()
GL_EDGE_FLAG_ARRAY_POINTER	glGetPointerv()
GL_EDGE_FLAG_ARRAY_STRIDE	glGet*
GL_EMISSION	glColorMaterial(), glGetMaterial*, glMaterial*
GL_ENABLE_BIT	glPushAttrib()
GL_EQUAL	glAlphaFunc(), glDepthFunc(), glStencilFunc()
GL_EQUIV	glLogicOp()

(续)

常量	相关命令
GL_EVAL_BIT	glPushAttrib()
GL_EXP, GL_EXP2	glFog*()
GL_EYE_LINEAR	glTexGen*()
GL_EYE_PLANE	glGetTexGen*(), glTexGen*()
GL_FALSE	glAreTexturesResident(), glColorMask(), glEdgeFlag(), glGet*(), glGetHistogram(), glGetMinmax(), glHistogram(), glIsEnabled(), glIsTexture(), glLightModel*(), glPixelStore*()
GL_FASTEST	glHint()
GL_FEEDBACK	glRenderMode()
GL_FEEDBACK_BUFFER_POINTER	glGetPointerv()
GL_FEEDBACK_BUFFER_SIZE, GL_FEEDBACK_BUFFER_TYPE	glGet*()
GL_FILL	glEvalMesh2(), glPolygonMode()
GL_FLAT	glShadeModel()
GL_FLOAT	glCallLists(), glColorPointer(), glColorSubTable(), glColorTable(), glConvolutionFilter1D(), glConvolutionFilter2D(), glConvolutionParameterf(), glConvolutionParameterfv(), glDrawPixels(), glGetColorTable(), glGetConvolutionFilter(), glGetConvolutionParameterfv(), glGetHistogram(), glGetHistogramParameterfv(), glGetMinmax(), glGetMinmaxParameterfv(), glGetSeparableFilter(), glGetTexImage(), glHistogram(), glIndexPointer(), glNormalPointer(), glReadPixels(), glResetHistogram(), glResetMinmax(), glSeparableFilter2D().

(续)

常量	相关命令
	glTexCoordPointer(), glTexImage1D(), glTexImage2D(), glTexImage3D(), glTexSubImage1D(), glTexSubImage2D(), glTexSubImage3D(), glVertexPointer()
GL_FOG	glDisable(), glEnable(), glGet*(), glIsEnabled()
GL_FOG_BIT	glPushAttrib()
GL_FOG_COLOR, GL_FOG_DENSITY, GL_FOG_END	glFog*, glGet*
GL_FOG_HINT	glGet*, glHint()
GL_FOG_INDEX, GL_FOG_MODE, GL_FOG_START	glFog*, glGet*
GL_FRONT	glColorMaterial(), glCullFace(), glDrawBuffer(), glGetMaterial*, glMaterial*, glPolygonMode(), glReadBuffer()
GL_FRONT_AND_BACK	glColorMaterial(), glCullFace(), glDrawBuffer(), glGetMaterial*, glMaterial*, glPolygonMode()
GL_FRONT_FACE	glGet*
GL_FRONT_LEFT, GL_FRONT_RIGHT	glDrawBuffer(), glReadBuffer()
GL_FUNC_ADD, GL_FUNC_REVERSE_SUBTRACT, GL_FUNC_SUBTRACT	glBlendEquation()
GL_GEQUAL, GL_GREATER	glAlphaFunc(), glDepthFunc(), glStencilFunc()
GL_GREEN	glColorSubTable(), glColorTable(), glConvolutionFilter1D(), glConvolutionFilter2D(), glDrawPixels(), glGetColorTable(), glGetConvolutionFilter(), glGetHistogram(), glGetMinmax(), glGetSeparableFilter(),

(续)

常量	相关命令
	glGetTexImage(), glHistogram(), glReadPixels(), glResetMinmax(), glSeparableFilter2D(), glTexImage1D(), glTexImage2D(), glTexImage3D(), glTexSubImage1D(), glTexSubImage2D(), glTexSubImage3D()
GL_GREEN_BIAS	glGet*(), glPixelTransfer*()
GL_GREEN_BITS	glGet*()
GL_GREEN_SCALE	glGet*(), glPixelTransfer*()
GL_HINT_BIT	glPushAttrib()
GL_HISTOGRAM	glDisable(), glEnable(), glGet*(), glGetHistogram(), glGetHistogramParameter*(), glHistogram(), glIsEnabled(), glResetHistogram()
GL_HISTOGRAM_ALPHA_SIZE, GL_HISTOGRAM_BLUE_SIZE, GL_HISTOGRAM_FORMAT, GL_HISTOGRAM_GREEN_SIZE, GL_HISTOGRAM_LUMINANCE_SIZE, GL_HISTOGRAM_RED_SIZE, GL_HISTOGRAM_SINK, GL_HISTOGRAM_WIDTH	glGetHistogramParameter*()
GL_INCR	glStencilOp()
GL_INDEX_ARRAY	glDisableClientState(), glEnableClientState(), glGet*(), glIsEnabled()
GL_INDEX_ARRAY_POINTER	glGetPointerv()
GL_INDEX_ARRAY_STRIDE, GL_INDEX_ARRAY_TYPE, GL_INDEX_BITS, GL_INDEX_CLEAR_VALUE	glGet*()
GL_INDEX_LOGIC_OP	glDisable(), glEnable(), glGet*(), glIsEnabled()
GL_INDEX_MODE	glGet*()
GL_INDEX_OFFSET, GL_INDEX_SHIFT	glGet*(), glPixelTransfer*()

(续)

常量	相关命令
GL_INDEX_WRITEMASK	glGet*()
GL_INT	glCallLists(), glColorPointer(), glColorSubTable(), glColorTable(), glConvolutionFilter1D(), glConvolutionFilter2D(), glConvolutionParameteri(), glConvolutionParameteriv(), glDrawPixels(), glGetColorTable(), glGetConvolutionFilter(), glGetConvolutionParameteriv(), glGetHistogram(), glGetHistogramParameteriv(), glGetMinmax(), glGetMinmaxParameteriv(), glGetSeparableFilter(), glGetTexImage(), glHistogram(), glIndexPointer(), glNormalPointer(), glReadPixels(), glResetHistogram(), glResetMinmax(), glSeparableFilter2D(), glTexCoordPointer(), glTexImage1D(), glTexImage2D(), glTexImage3D(), glTexSubImage1D(), glTexSubImage2D(), glTexSubImage3D(), glVertexPointer()
GL_INTENSITY	glColorTable(), glConvolutionFilter1D(), glConvolutionFilter2D(), glCopyColorSubTable(), glCopyColorTable(), glCopyConvolutionFilter1D(), glCopyConvolutionFilter2D(), glSeparableFilter2D()
GL_INTENSITY4, GL_INTENSITY8, GL_INTENSITY12, GL_INTENSITY16	glColorTable(), glConvolutionFilter1D(), glConvolutionFilter2D(), glCopyColorTable(), glCopyConvolutionFilter1D(), glCopyConvolutionFilter2D(),

(续)

常量	相关命令
GL_INVALID_ENUM, GL_INVALID_OPERATION, GL_INVALID_VALUE	glSeparableFilter2D()
GL_INVERT	glLogicOp(), glStencilOp()
GL_KEEP	glStencilOp()
GL_LEFT	glDrawBuffer(), glReadBuffer()
GL_EQUAL, GL_LESS	glAlphaFunc(), glDepthFunc(), glStencilFunc()
GL_LIGHT0 through GL_LIGHT7	glDisable(), glEnable(), glGet*(), glGetLight*(), glIsEnabled(), glLight*()
GL_LIGHTING	glDisable(), glEnable(), glGet*(), glIsEnabled()
GL_LIGHTING_BIT	glPushAttrib()
GL_LIGHT_MODEL_AMBIENT, GL_LIGHT_MODEL_COLOR_CONTROL, GL_LIGHT_MODEL_LOCAL_VIEWER, GL_LIGHT_MODEL_TWO_SIDE	glGet*(), glLightModel*()
GL_LINE	glEvalMesh1(), glEvalMesh2(), glPolygonMode()
GL_LINEAR	glFog*(), glTexParameter*
GL_LINEAR_ATTENUATION	glGetLight*(), glLight*
GL_LINEAR_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST	glTexParameter*
GL_LINES	glBegin(), glDrawArrays(), glDrawElements(), glDrawRangeElements()
GL_LINE_BIT	glPushAttrib()
GL_LINE_LOOP	glBegin(), glDrawArrays(), glDrawElements(), glDrawRangeElements()
GL_LINE_RESET_TOKEN	glPassThrough()
GL_LINE_SMOOTH	glDisable(), glEnable(), glGet*(), glIsEnabled()

(续)

常量	相关命令
GL_LINE_SMOOTH_HINT	glGet*, glHint()
GL_LINE_STIPPLE	glDisable(), glEnable(), glGet*, glIsEnabled()
GL_LINE_STIPPLE_PATTERN, GL_LINE_STIPPLE_REPEAT	glGet()
GL_LINE_STRIP	glBegin(), glDrawArrays(), glDrawElements(), glDrawRangeElements()
GL_LINE_TOKEN	glPassThrough()
GL_LINE_WIDTH, GL_LINE_WIDTH_GRANULARITY, GL_LINE_WIDTH_RANGE, GL_LIST_BASE	glGet()
GL_LIST_BIT	glPushAttrib()
GL_LIST_INDEX, GL_LIST_MODE	glGet()
GL_LOAD	glAccum()
GL_LOGIC_OP_MODE	glGet()
GL_LUMINANCE, GL_LUMINANCE4, GL_LUMINANCE8, GL_LUMINANCE12, GL_LUMINANCE16, GL_LUMINANCE_ALPHA, GL_LUMINANCE4_ALPHA4, GL_LUMINANCE6_ALPHA2, GL_LUMINANCE8_ALPHA8, GL_LUMINANCE12_ALPHA4, GL_LUMINANCE12_ALPHA12, GL_LUMINANCE16_ALPHA16	glColorSubTable(), glColorTable(), glConvolutionFilter1D(), glConvolutionFilter2D(), glCopyColorSubTable(), glCopyConvolutionFilter1D(), glCopyConvolutionFilter2D(), glDrawPixels(), glGetColorTable(), glGetConvolutionFilter(), glGetHistogram(), glGetMinmax(), glGetSeparableFilter(), glGetTexImage(), glHistogram(), glMinmax(), glReadPixels(), glSeparableFilter2D(), glTexImage1D(), glTexImage2D(), glTexImage3D(), glTexSubImage1D(), glTexSubImage2D(), glTexSubImage3D()
GL_MAP1_COLOR_4	glDisable(), glEnable(), glGet*, glGetMap*, glIsEnabled()

(续)

常量	相关命令
GL_MAP1_GRID_DOMAIN, GL_MAP1_GRID_SEGMENTS	glMap1*()
GL_MAP1_INDEX, GL_MAP1_NORMAL, GL_MAP1_TEXTURE_COORD_1 through GL_MAP1_TEXTURE_COORD_4, GL_MAP1_VERTEX_3, GL_MAP1_VERTEX_4	glDisable(), glEnable(), glGet*(), glGetMap*(), glIsEnabled(), glMap1*()
GL_MAP2_COLOR_4	glDisable(), glEnable(), glGet*(), glGetMap*(), glIsEnabled(), glMap2*()
GL_MAP2_GRID_DOMAIN, GL_MAP2_GRID_SEGMENTS	glGet*()
GL_MAP2_INDEX, GL_MAP2_NORMAL, GL_MAP2_TEXTURE_COORD_1, GL_MAP2_TEXTURE_COORD_4, GL_MAP2_VERTEX_3, GL_MAP2_VERTEX_4	glDisable(), glEnable(), glGet*(), glGetMap*(), glIsEnabled(), glMap2*()
GL_MAP_COLOR, GL_MAP_STENCIL	glGet*(), glPixelTransfer*()
GL_MATRIX_MODE	glGet*()
GL_MAX	glBlendEquation()
GL_MAX_3D_TEXTURE_SIZE, GL_MAX_ATTRIB_STACK_DEPTH, GL_MAX_CLIENT_ATTRIB_STACK_DEPTH, GL_MAX_CLIP_PLANES, GL_MAX_COLOR_MATRIX_STACK_DEPTH	glGet*()
GL_MAX_CONVOLUTION_HEIGHT, GL_MAX_CONVOLUTION_WIDTH	glGetConvolutionParameter*()
GL_MAX_ELEMENTS_INDICES, GL_MAX_ELEMENTS_VERTICES, GL_MAX_EVAL_ORDER, GL_MAX_LIGHTS, GL_MAX_LIST_NESTING, GL_MAX_MODELVIEW_STACK_DEPTH, GL_MAX_NAME_STACK_DEPTH, GL_MAX_PIXEL_MAP_TABLE, GL_MAX_PROJECTION_STACK_DEPTH, GL_MAX_TEXTURE_SIZE, GL_MAX_TEXTURE_STACK_DEPTH, GL_MAX_TEXTURE_UNITS_ARB, GL_MAX_VIEWPORT_DIMS	glGet*()
GL_MIN	glBlendEquation()

(续)

常量	相关命令
GL_MINMAX	glDisable(), glEnable(), glGet*(), glGetMinmax(), glGetMinmaxParameter*(), glIsEnabled(), glMinmax(), glResetMinmax()
GL_MINMAX_FORMAT, GL_MINMAX_SINK	glGetMinmaxParameter*()
GL_MODELVIEW	glMatrixMode()
GL_MODELVIEW_MATRIX, GL_MODELVIEW_STACK_DEPTH	glGet*()
GL_MODULATE	glTexEnv*()
GL_MULT	glAccum()
GL_N3F_V3F	glInterleavedArrays()
GL_NAME_STACK_DEPTH	glGet*()
GL NAND	glLogicOp()
GL_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_NEAREST_MIPMAP_NEAREST	glTexParameter*()
GL_NEVER	glAlphaFunc(), glDepthFunc(), glStencilFunc()
GL_NICEST	glHint()
GL_NONE	glClear(), glDrawBuffer()
GL_NOOP, GL_NOR	glLogicOp()
GL_NORMALIZE	glDisable(), glEnable(), glGet*(), glIsEnabled()
GL_NORMAL_ARRAY	glDisableClientState(), glEnableClientState(), glGet*(), glIsEnabled()
GL_NORMAL_ARRAY_POINTER	glGetPointerv()
GL_NORMAL_ARRAY_STRIDE, GL_NORMAL_ARRAY_TYPE	glGet*()
GL_NOTEQUAL	glAlphaFunc(), glDepthFunc(), glStencilFunc()
GL_NO_ERROR	glGetError()

(续)

常量	相关命令
GL_OBJECT_LINEAR	glTexGen*()
GL_OBJECT_PLANE	glGetTexGen*(), glTexGen*()
GL_ONE, GL_ONE_MINUS_CONSTANT_ALPHA, GL_ONE_MINUS_CONSTANT_COLOR, GL_ONE_MINUS_DST_ALPHA, GL_ONE_MINUS_DST_COLOR, GL_ONE_MINUS_SRC_ALPHA, GL_ONE_MINUS_SRC_COLOR	glBlendFunc()
GL_OR	glLogicOp()
GL_ORDER	glGetMap*()
GL_OR_INVERTED, GL_OR_REVERSE	glLogicOp()
GL_OUT_OF_MEMORY	glGetError()
GL_PACK_ALIGNMENT, GL_PACK_IMAGE_HEIGHT, GL_PACK_LSB_FIRST, GL_PACK_ROW_LENGTH, GL_PACK_SKIP_IMAGES, GL_PACK_SKIP_PIXELS, GL_PACK_SKIP_ROWS, GL_PACK_SWAP_BYTES	glGet*, glPixelStore*()
GL_PASS_THROUGH_TOKEN	glPassThrough()
GL_PERSPECTIVE_CORRECTION_HINT	glGet*, glHint()
GL_PIXEL_MAP_A_TO_A, GL_PIXEL_MAP_B_TO_B, GL_PIXEL_MAP_G_TO_G, GL_PIXEL_MAP_I_TO_A, GL_PIXEL_MAP_I_TO_B, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_I, GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_R_TO_R, GL_PIXEL_MAP_S_TO_S	glGetPixelMap*, glPixelMap()
GL_PIXEL_MAP_A_TO_A_SIZE, GL_PIXEL_MAP_B_TO_B_SIZE, GL_PIXEL_MAP_G_TO_G_SIZE, GL_PIXEL_MAP_I_TO_A_SIZE, GL_PIXEL_MAP_I_TO_B_SIZE, GL_PIXEL_MAP_I_TO_G_SIZE, GL_PIXEL_MAP_I_TO_I_SIZE,	glGet()

(续)

常量	相关命令
GL_PIXEL_MAP_I_TO_R_SIZE, GL_PIXEL_MAP_R_TO_R_SIZE, GL_PIXEL_MAP_S_TO_S_SIZE	
GL_PIXEL_MODE_BIT	glPushAttrib()
GL_POINT	glEvalMesh1(), glEvalMesh2(), glPolygonMode()
GL_POINTS	glBegin(), glDrawArrays(), glDrawElements(), glDrawRangeElements()
GL_POINT_BIT	glPushAttrib()
GL_POINT_SIZE, GL_POINT_SIZE_GRANULARITY, GL_POINT_SIZE_RANGE	glGet*()
GL_POINT_SMOOTH	glDisable(), glEnable(), glGet*(), glIsEnabled()
GL_POINT_SMOOTH_HINT	glGet*(), glHint()
GL_POINT_TOKEN	glPassThrough()
GL_POLYGON	glBegin(), glDrawArrays(), glDrawElements(), glDrawRangeElements()
GL_POLYGON_BIT	glPushAttrib()
GL_POLYGON_MODE, GL_POLYGON_OFFSET_FACTOR	glGet*()
GL_POLYGON_OFFSET_FILL, GL_POLYGON_OFFSET_LINE, GL_POLYGON_OFFSET_POINT	glDisable(), glEnable(), glGet*(), glIsEnabled()
GL_POLYGON_OFFSET_UNITS	glGet*()
GL_POLYGON_SMOOTH	glDisable(), glEnable(), glGet*(), glIsEnabled()
GL_POLYGON_SMOOTH_HINT	glGet*(), glHint()
GL_POLYGON_STIPPLE	glDisable(), glEnable(), glGet*(), glIsEnabled()
GL_POLYGON_STIPPLE_BIT	glPushAttrib()
GL_POLYGON_TOKEN	glPassThrough()
GL_POSITION	glGetLight*(), glLight*()

(续)

常量	相关命令
GL_POST_COLOR_MATRIX_ALPHA_BIAS, GL_POST_COLOR_MATRIX_ALPHA_SCALE, GL_POST_COLOR_MATRIX_BLUE_BIAS, GL_POST_COLOR_MATRIX_BLUE_SCALE	glGet*, glPixelTransfer*
GL_POST_COLOR_MATRIX_COLOR_TABLE	glColorSubTable(), glColorTable(), glColorTableParameter*, glCopyColorSubTable(), glCopyColorTable(), glDisable(), glEnable(), glGet*, glGetColorTable(), glGetColorTableParameter*, glIsEnabled()
GL_POST_COLOR_MATRIX_GREEN_BIAS, GL_POST_COLOR_MATRIX_GREEN_SCALE, GL_POST_COLOR_MATRIX_RED_BIAS, GL_POST_COLOR_MATRIX_RED_SCALE, GL_POST_CONVOLUTION_ALPHA_BIAS, GL_POST_CONVOLUTION_ALPHA_SCALE, GL_POST_CONVOLUTION_BLUE_BIAS, GL_POST_CONVOLUTION_BLUE_SCALE	glGet*, glPixelTransfer*
GL_POST_CONVOLUTION_COLOR_TABLE	glColorSubTable(), glColorTable(), glColorTableParameter*, glCopyColorSubTable(), glCopyColorTable(), glDisable(), glEnable(), glGet*, glGetColorTable(), glGetColorTableParameter*, glIsEnabled()
GL_POST_CONVOLUTION_GREEN_BIAS, GL_POST_CONVOLUTION_GREEN_SCALE, GL_POST_CONVOLUTION_RED_BIAS, GL_POST_CONVOLUTION_RED_SCALE	glGet*, glPixelTransfer*
GL_PROJECTION	glMatrixMode()
GL_PROJECTION_MATRIX, GL_PROJECTION_STACK_DEPTH	glGet()
GL_PROXY_COLOR_TABLE	glColorSubTable(), glColorTable()
GL_PROXY_HISTOGRAM	glHistogram()
GL_PROXY_POST_COLOR_MATRIX_COLOR_TABLE, GL_PROXY_POST_CONVOLUTION_COLOR_TABLE	glColorSubTable(), glColorTable()

(续)

常量	相关命令
GL_PROXY_TEXTURE_1D, GL_PROXY_TEXTURE_2D, GL_PROXY_TEXTURE_3D	glTexImage1D(), glTexImage2D(), glTexImage3D()
GL_Q	glGetTexGen*(), glTexGen*()
GL_QUADRATIC_ATTENUATION	glGetLight*(), glLight*()
GL_QUADS	glBegin(), glDrawArrays(), glDrawElements(), glDrawRangeElements()
GL_QUAD_STRIP	glBegin(), glDrawArrays(), glDrawElements(), glDrawRangeElements()
GL_R	glGetTexGen*(), glTexGen*()
GL_R3_G3_B2	glColorTable(), glConvolutionFilter1D(), glConvolutionFilter2D(), glCopyColorTable(), glCopyConvolutionFilter1D(), glCopyConvolutionFilter2D(), glCopyTexImage1D(), glCopyTexImage2D(), glHistogram(), glMinmax(), glSeparableFilter2D(), glTexImage3D()
GL_READ_BUFFER	glGet*()
GL_RED	glColorSubTable(), glColorTable(), glConvolutionFilter1D(), glConvolutionFilter2D(), glDrawPixels(), glGetColorTable(), glGetConvolutionFilter(), glGetHistogram(), glGetMinmax(), glGetSeparableFilter(), glGetTexImage(), glHistogram(), glReadPixels(), glResetMinmax(), glSeparableFilter2D(), glTexImage1D(), glTexImage2D(), glTexImage3D(), glTexSubImage1D(), glTexSubImage2D(), glTexSubImage3D()

(续)

常量	相关命令
GL_REDUCE	glConvolutionParameter*
GL_RED_BIAS	glGet*, glPixelTransfer*
GL_RED_BITS	glGet()
GL_RED_SCALE	glGet*, glPixelTransfer*
GL_RENDER	glRenderMode()
GL_RENDERER	glGetString()
GL_RENDER_MODE	glGet()
GL_REPEAT	glTexParameter*
GL_REPLACE	glStencilOp(), glTexEnv*
GL_REPLICATE_BORDER	glConvolutionParameter*
GL_RESCALE_NORMAL	glDisable(), glEnable(), glGet(), glIsEnabled()
GL_RETURN	glAccum()
GL_RGB	glColorSubTable(), glColorTable(), glConvolutionFilter1D(), glConvolutionFilter2D(), glCopyColorSubTable(), glCopyConvolutionFilter1D(), glCopyConvolutionFilter2D(), glDrawPixels(), glGetColorTable(), glGetConvolutionFilter(), glGetHistogram(), glGetMinmax(), glGetSeparableFilter(), glGetTexImage(), glHistogram(), glMinmax(), glReadPixels(), glResetMinmax(), glSeparableFilter2D(), glTexImage1D(), glTexImage2D(), glTexImage3D(), glTexSubImage1D(), glTexSubImage2D(), glTexSubImage3D()
GL_RGB4, GL_RGB5, GL_RGB8, GL_RGB10, GL_RGB12, GL_RGB16, GL_RGB5_A1,	glColorTable(), glConvolutionFilter1D(),

(续)

常量	相关命令
GL_RGB10_A2	glConvolutionFilter2D(), glCopyColorTable(), glCopyConvolutionFilter1D(), glCopyConvolutionFilter2D(), glCopyTexImage1D(), glCopyTexImage2D(), glHistogram(), glMinmax(), glSeparableFilter2D(), glTexImage3D()
GL_RGBA	glColorSubTable(), glColorTable(), glConvolutionFilter1D(), glConvolutionFilter2D(), glCopyColorSubTable(), glCopyColorTable(), glCopyConvolutionFilter1D(), glCopyConvolutionFilter2D(), glCopyTexImage1D(), glCopyTexImage2D(), glCopyTexSubImage1D(), glCopyTexSubImage2D(), glCopyTexSubImage3D(), glDrawPixels(), glGetColorTable(), glGetConvolutionFilter(), glGetHistogram(), glGetMinmax(), glGetSeparableFilter(), glGetTexImage(), glHistogram(), glMinmax(), glReadPixels(), glSeparableFilter2D(), glTexImage1D(), glTexImage2D(), glTexImage3D(), glTexSubImage1D(), glTexSubImage2D(), glTexSubImage3D()
GL_RGBA2, GL_RGBA4, GL_RGBA8, GL_RGBA12, GL_RGBA16	glColorTable(), glConvolutionFilter1D(), glConvolutionFilter2D(), glCopyColorTable(), glCopyConvolutionFilter1D(), glCopyConvolutionFilter2D(), glCopyTexImage1D(), glCopyTexImage2D(), glHistogram(), glMinmax(), glSeparableFilter2D(),

(续)

常量	相关命令
	glTexImage1D(), glTexImage2D(), glTexImage3D()
GL_RGBA_MODE	glGet*()
GL_RIGHT	glDrawBuffer(), glReadBuffer()
GL_S	glGetTexGen*(), glTexGen*()
GL_SCISSOR_BIT	glPushAttrib()
GL_SCISSOR_BOX	glGet*()
GL_SCISSOR_TEST	glDisable(), glEnable(), glGet*(), glIsEnabled()
GL_SELECT	glRenderMode()
GL_SELECTION_BUFFER_POINTER	glGetPointerv()
GL_SELECTION_BUFFER_SIZE	glGet*()
GL_SEPARABLE_2D	glDisable(), glEnable(), glGetSeparableFilter(), glIsEnabled(), glSeparableFilter2D()
GL_SEPARATE_SPECULAR_COLOR	glLightModel*()
GL_SET	glLogIcOp()
GL_SHADE_MODEL	glGet*()
GL_SHININESS	glGetMaterial*(), glMaterial*()
GL_SHORT	glCallLists(), glColorPointer(), glColorSubTable(), glColorTable(), glConvolutionFilter1D(), glConvolutionFilter2D(), glDrawPixels(), glGetColorTable(), glGetConvolutionFilter(), glGetHistogram(), glGetMinmax(), glGetSeparableFilter(), glGetTexImage(), glHistogram(), glIndexPointer(), glNormalPointer(), glReadPixels(), glResetHistogram(), glResetMinmax(),

(续)

常量	相关命令
	glSeparableFilter2D(), glTexCoordPointer(), glTexImage1D(), glTexImage2D(), glTexImage3D(), glTexSubImage1D(), glTexSubImage2D(), glTexSubImage3D(), glVertexPointer()
GL_SINGLE_COLOR	glLightModel*()
GL_SMOOTH	glShadeModel()
GL_SMOOTH_LINE_WIDTH_GRANULARITY, GL_SMOOTH_LINE_WIDTH_RANGE, GL_SMOOTH_POINT_SIZE_GRANULARITY, GL_SMOOTH_POINT_SIZE_RANGE	glGet*
GL_SPECULAR	glColorMaterial(), glGetLight*, glGetMaterial*, glLight*, glMaterial()
GL_SPHERE_MAP	glTexGen*
GL_SPOT_CUTOFF, GL_SPOT_DIRECTION, GL_SPOT_EXPONENT	glGetLight*, glLight*
GL_SRC_ALPHA, GL_SRC_ALPHA_SATURATE, GL_SRC_COLOR	glBlendFunc()
GL_STACK_OVERFLOW, GL_STACK_UNDERFLOW	glGetError()
GL_STENCIL	glCopyPixels()
GL_STENCIL_BITS	glGet*
GL_STENCIL_BUFFER_BIT	glClear(), glPushAttrib()
GL_STENCIL_CLEAR_VALUE, GL_STENCIL_FAIL, GL_STENCIL_FUNC	glGet*
GL_STENCIL_INDEX	glDrawPixels(), glReadPixels()
GL_STENCIL_PASS_DEPTH_FAIL, GL_STENCIL_PASS_DEPTH_PASS, GL_STENCIL_REF	glGet*
GL_STENCIL_TEST	glDisable(), glEnable(), glGet*, glIsEnabled()

(续)

常量	相关命令
GL_STENCIL_VALUE_MASK, GL_STENCIL_WRITEMASK, GL_STEREO, GL_SUBPIXEL_BITS	glGet*
GL_T	glGetTexGen*, glTexGen()
GL_T2F_C3F_V3F, GL_T2F_C4F_N3F_V3F, GL_T2F_C4UB_V3F, GL_T2F_N3F_V3F, GL_T2F_V3F, GL_T4F_C4F_N3F_V4F, GL_T4F_V4F	glInterleavedArrays()
GL_TABLE_TOO_LARGE	glGetError()
GL_TEXTURE	glMatrixMode()
GL_TEXTURE0_ARB through GL_TEXTURE31_ARB	glActiveTextureARB(), glClientActiveTextureARB(), glMultiTexCoord*ARB()
GL_TEXTURE_1D	glBindTexture(), glCopyTexImage1D(), glCopyTexSubImage1D(), glDisable(), glEnable(), glGet*, glGetTexImage(), glGetTexLevelParameter*, glGetTexParameter*, glIsEnabled(), glTexImage1D(), glTexParameter*, glTexSubImage1D()
GL_TEXTURE_2D	glBindTexture(), glCopyTexImage2D(), glCopyTexSubImage2D(), glDisable(), glEnable(), glGet*, glGetTexImage(), glGetTexLevelParameter*, glGetTexParameter*, glIsEnabled(), glTexImage2D(), glTexParameter*, glTexSubImage2D()
GL_TEXTURE_3D	glBindTexture(), glDisable(), glEnable(), glGet*, glGetTexImage(), glGetTexLevelParameter*, glGetTexParameter*, glIsEnabled(), glTexImage3D(), glTexParameter*, glTexSubImage3D()

(续)

常量	相关命令
GL_TEXTURE_ALPHA_SIZE	glGetTexLevelParameter*(), glGetTexParameter*()
GL_TEXTURE_BASE_LEVEL	glGetTexLevelParameter*(), glGetTexParameter*(), glTexParameter*()
GL_TEXTURE_BINDING_1D, GL_TEXTURE_BINDING_2D, GL_TEXTURE_BINDING_3D	glGet*()
GL_TEXTURE_BIT	glPushAttrib()
GL_TEXTURE_BLUE_SIZE, GL_TEXTURE_BORDER	glGetTexLevelParameter*(), glGetTexParameter*()
GL_TEXTURE_BORDER_COLOR	glGetTexParameter*(), glTexParameter*()
GL_TEXTURE_COORD_ARRAY	glDisableClientState(), glEnableClientState(), glGet*(), glIsEnabled()
GL_TEXTURE_COORD_ARRAY_POINTER	glGetPointerv()
GL_TEXTURE_COORD_ARRAY_SIZE, GL_TEXTURE_COORD_ARRAY_STRIDE, GL_TEXTURE_COORD_ARRAY_TYPE	glGet*()
GL_TEXTURE_DEPTH	glGetTexLevelParameter*(), glGetTexParameter*()
GL_TEXTURE_ENV, GL_TEXTURE_ENV_COLOR, GL_TEXTURE_ENV_MODE	glGetTexEnv*(), glTexEnv*()
GL_TEXTURE_GEN_MODE	glGetTexGen*(), glTexGen*()
GL_TEXTURE_GEN_Q, GL_TEXTURE_GEN_R, GL_TEXTURE_GEN_S, GL_TEXTURE_GEN_T	glDisable(), glEnable(), glGet*(), glIsEnabled()
GL_TEXTURE_GREEN_SIZE, GL_TEXTURE_HEIGHT, GL_TEXTURE_INTENSITY_SIZE, GL_TEXTURE_INTERNAL_FORMAT, GL_TEXTURE_LUMINANCE_SIZE	glGetTexLevelParameter*(), glGetTexParameter*()
GL_TEXTURE_MAG_FILTER	glGetTexParameter*(), glTexParameter*()
GL_TEXTURE_MATRIX	glGet*()

(续)

常量	相关命令
GL_TEXTURE_MAX_LEVEL, GL_TEXTURE_MAX_LOD, GL_TEXTURE_MIN_FILTER, GL_TEXTURE_MIN_LOD, GL_TEXTURE_PRIORITY	glGetTexParameter*(), glTexParameter*()
GL_TEXTURE_RED_SIZE, GL_TEXTURE_RESIDENT	glGetTexLevelParameter*(), glGetTexParameter*()
GL_TEXTURE_STACK_DEPTH	glGet*()
GL_TEXTURE_WIDTH	glGetTexLevelParameter*(), glGetTexParameter*()
GL_TEXTURE_WRAP_R, GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T	glGetTexParameter*(), glTexParameter*()
GL_TRANSFORM_BIT	glPushAttrib()
GL_TRIANGLES, GL_TRIANGLE_FAN, GL_TRIANGLE_STRIP	glBegin(), glDrawArrays(), glDrawElements(), glDrawRangeElements()
GL_TRUE	glAreTexturesResident(), glBindTexture(), glCallLists(), glGet*, glIsTexture(), glPixelStore(), glPopAttrib(), glPrioritizeTextures()
GL_UNPACK_ALIGNMENT, GL_UNPACK_IMAGE_HEIGHT, GL_UNPACK_LSB_FIRST, GL_UNPACK_ROW_LENGTH, GL_UNPACK_SKIP_IMAGES, GL_UNPACK_SKIP_PIXELS, GL_UNPACK_SKIP_ROWS, GL_UNPACK_SWAP_BYTES	glGet*, glPixelStore*()

第5章 OpenGL 参考说明

本章按字母次序列出了所有OpenGL命令（见表5-1）的参考说明。每一页参考说明都描述了多个相关的命令。关于OpenGL实用库(GLU)和对X窗口系统的OpenGL扩展(GLX)请参阅后面有关章节的介绍。

表 5-1

glAccum	glClear	glColorTable
glActiveTextureARB	glClearAccum	glColorTableParameter
glAlphaFunc	glClearColor	glConvolutionFilter1D
glAreTexturesResident	glClearDepth	glConvolutionFilter2D
glArrayElement	glClearIndex	glConvolutionParameter
glBegin, glEnd	glClearStencil	glCopyColorSubTable
glBindTexture	glClientActiveTextureARB	glCopyColorTable
glBitmap	glClipPlane	glCopyConvolutionFilter1D
glBlendColor	glColor	glCopyConvolutionFilter2D
glBlendEquation	glColorMask	glCopyPixels
glBlendFunc	glColorMaterial	glCopyTexImage1D
glCallList	glColorPointer	glCopyTexImage2D
glCallLists	glColorSubTable	glCopyTexSubImage1D
glCopyTexSubImage2D	glFustum	glGetTexLevelParameter
glCopyTexSubImage3D	glGenLists	glGetTexParameter
glCullFace	glGenTextures	glHint
glDeleteLists	glGet	glHistogram
glDeleteTextures	glGetClipPlane	glIndex
glDepthFunc	glGetColorTable	glIndexMask
glDepthMask	glGetColorTableParameter	glIndexPointer
glDepthRange	glGetConvolutionFilter	glInitNames
glDrawArrays	glGetConvolutionParameter	glInterleavedArrays

(续)

glDrawBuffer	glGetError	glIsEnabled
glDrawElements	glGetHistogram	glIsList
glDrawPixels	glGetHistogramParameter	glIsTexture
glDrawRangeElements	glGetLight	glLight
glEdgeFlag	glGetMap	glLightModel
glEdgeFlagPointer	glGetMaterial	glLineStipple
glEnable, glDisable	glGetMinmax	glLineWidth
glEnableClientState, glDisableClientState	glGetMinmaxParameter	glListBase
glEvalCoord	glGetPixelMap	glLoadIdentity
glEvalMesh1, glEvalMesh2	glGetPointerv	glLoadMatrix
glEvalPoint	glGetPolygonStipple	glLoadName
glFeedbackBuffer	glGetSeparableFilter	glLogicOp
glFinish	glGetString	glMap1
glFlush	glGetTexEnv	glMap2
glFog	glGetTexGen	glMapGrid
glFrontFace	glGetTexImage	glMaterial
glMatrixMode	glPushAttrib, glPopAttrib	glStencilMask
glMinmax	glPushClientAttrib, glPopClientAttrib	glStencilOp
glMultiTexCoord	glPushMatrix, glPopMatrix	glTexCoord
glMultMatrix	glPushName, glPopName	glTexCoordPointer
glNewList, glEndList	glRasterPos	glTexEnv
glNormal	glReadBuffer	glTexGen
glNormalPointer	glReadPixels	glTexImage1D
glOrtho	glRect	glTexImage2D
glPassThrough	glRenderMode	glTexImage3D
glPixelMap	glResetHistogram	glTexParameter
glPixelStore	glResetMinmax	glTexSubImage1D

(续)

glPixelTransfer	glRotate	glTexSubImage2D
glPixelZoom	glScale	glTexSubImage3D
glPointSize	glScissor	glTranslate
glPolygonMode	glSelectBuffer	glVertex
glPolygonOffset	glSeparableFilter2D	glVertexPointer
glPolygonStipple	glShadeModel	glViewport
glPrioritizeTextures	glStencilFunc	

■ glAccum

- 名称:

glAccum()

- 功能:

在累积缓冲区中进行操作。

- C描述:

```
void glAccum( GLenum op,
              GLfloat value )
```

- 参数说明:

op 指定累积缓冲区的操作。它可取符号常量: **GL_ACCUM**、**GL_LOAD**、**GL_ADD**、**GL_MULT**和**GL_RETURN**。

value 指定一个用于累积缓冲区操作的浮点值。参数*op*决定了将怎样使用*value*。

- 说明:

累积缓冲区是一种扩展了范围的颜色缓冲区。图像并不直接绘入累积缓冲区。相反地，绘入一个颜色缓冲区的图像是在绘制后才被加上累积缓冲区中的内容。不同转换矩阵所产生的累积图像可以用于反走样(点、线段和多边形的)、动模糊以及域的深度。

累积缓冲区中的每个像素都包含红、绿、蓝和alpha值。累积缓冲区中用于存储每个组分的位数与具体的实现有关。你可以调用函数glGetIntegerv(**GL_ACCUM_RED_BITS**)、glGetIntegerv(**GL_ACCUM_GREEN_BITS**)、glGetIntegerv(**GL_ACCUM_BLUE_BITS**)和glGetIntegerv(**GL_ACCUM_ALPHA_BITS**)来检测这个位数的具体值。不管每一组分的位数究竟如何，存储于每个组分中的值的范围均是[-1, 1]。累积缓冲区中的像素与帧缓冲区中的像素是一一映射的。

函数**glAccum()**是在累积缓冲区中进行操作的。它的第一个自变量*op*是一个符号常量，用来选择累积缓冲区的一种操作。它的第二个自变量*value*是该操作中用到的浮点值。共定义了五种操作: **GL_ACCUM**、**GL_LOAD**、**GL_ADD**、**GL_MULT**和**GL_RETURN**。

所有累积缓冲区的操作均被限制在当前裁剪箱的范围内，且被同样地应用于每个像素的红、绿、蓝和alpha组分。如果函数**glAccum()**运行的结果超出了范围[-1, 1]，则累积缓冲区中像素组分的内容将是未定义的。

具体的操作如下：

GL_ACCUM	从当前选择的缓冲区中读取（请参阅glReadBuffer()）R、G、B和A值。每一组分除以 2^n-1 ，这里n是当前所选定的缓冲区中表示每个颜色组分的位数。其结果是一个在范围[0, 1]内的浮点值。用value乘上它们以后，再加上累积缓冲区中相应的像素组分，然后用它来更新累积缓冲区。
GL_LOAD	同GL_ACCUM相类似，唯一不同之处是累积缓冲区中的当前值不再用来计算新值。也就是说，由当前所选定的缓冲区中读出的R、G、B和A值除以 2^n-1 后再乘上value所得的值被直接存入相应的累积缓冲单元，从而覆盖当前的值。
GL_ADD	将value直接加到累积缓冲区中的R、G、B和A上。
GL_MULT	用value乘上累积缓冲区中的R、G、B和A值，然后将这些放大的组分返回它们相应的累积缓冲存储单元。
GL_RETURN	将累积缓冲区中的值传送到当前所选定的用于写入的颜色缓冲区中。每一个R、O、B和A组分分别乘上value后再乘上 2^n-1 ，这时其取值范围是[0, 2^n-1]，然后存入相应的显示缓冲单元中。可以采用这种传送的片断操作只能是像素所有权测试、裁剪操作、抖动操作和颜色写入的屏蔽操作。

如果要清除累积缓冲区，你可以首先调用带有R、G、B和A值的函数glClearAccum()来设置它，接下来在已启动累积缓冲区的情况下调用函数glClear()即可。

- 注意：

只有在当前裁剪箱范围内的像素才可以被glAccum()操作更新。

- 出错提示：

如果参数op不是一个可接受的值，则产生**GL_INVALID_ENUM**提示。

如果没有累积缓冲区存在，则产生**GL_INVALID_OPERATION**提示。

如果函数glAccum()在函数对glBegin()/glEnd()之间执行，则产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

```
glGet( GL_ACCUM_RED_BITS )
glGet( GL_ACCUM_GREEN_BITS )
glGet( GL_ACCUM_BLUE_BITS )
glGet( GL_ACCUM_ALPHA_BITS )
```

- 请参阅：

```
glClear(), glClearAccum(), glCopyPixels(), glDrawBuffer(), glGet(), glReadBuffer(),
glReadPixels(), glScissor(), glStencilOp()
```

glActiveTextureARB

- 名称：

glActiveTextureARB()

- 功能：

选择当前有效的纹理单元。

- C描述：

void glActiveTextureARB(GLenum *texture*)

- 参数说明：

texture 指定哪一个纹理单元将被激活。纹理单元的数目由具体实现决定，但至少要两个。参数*texture*必须取**GL_TEXTURE*i*_ARB**之一，这里 $0 \leq i < \text{GL_MAX_TEXTURE_UNITS_ARB}$ ，它是一个与实现有关的值。其缺省值是**GL_TEXTURE0_ARB**。

- 说明：

函数**glActiveTextureARB()**用来选择哪个纹理单元后面的纹理状态调用将有效。纹理单元的数目与具体实现有关，但至少要两个以上。

顶点数组是一种客户端的GL资源，它由函数**glClientActiveTextureARB()**选取。

- 注意：

当调用函数**glGetString(GL_EXTENSIONS)**所返回的字符串中包含**GL_ARB_multitexture**时，函数**glActiveTextureARB()**才被支持。

- 出错提示：

如果参数*texture*不是任何一个**GL_TEXTURE*i*_ARB**，则产生**GL_INVALID_ENUM**提示。这里 $0 \leq i < \text{GL_MAX_TEXTURE_UNITS_ARB}$ 。

- 请参阅：

glClientActiveTextureARB(), **glGetInteger()**, **glMultiTexCoordARB()**,
glTexParameter()

glAlphaFunc

- 名称：

glAlphaFunc()

- 功能：

指定alpha测试函数。

- C描述：

**void glAlphaFunc(GLenum *func*,
 GLclampf *ref*)**

- 参数说明：

func 指定alpha比较函数。它可以取以下各符号常量之一：**GL_NEVER**、**GL_LESS**、**GL_EQUAL**、**GL_LEQUAL**、**GL_GREATER**、**GL_NOTEQUAL**、**GL_GEQUAL**和**GL_ALWAYS**。其缺省值是**GL_ALWAYS**。

ref 指定与输入的alpha值相比较的参考值。参数*ref*的取值范围是[0, 1]，这里0代表可能的最低alpha值，1代表可能的最高alpha值。其缺省值是0。

- 说明：

alpha测试将根据一个输入片断的alpha值与一个常数参考值的比较结果而删除一些片断。函数`glAlphaFunc()`用来指定参考值和比较函数。只有当alpha测试开启后，比较操作才可以进行。在缺省情况下，该测试是关闭的。（请参阅`glEnable(GL_ALPHA_TEST)`和`glDisable(GL_ALPHA_TEST)`）。

参数`func`和`ref`用于指定绘制像素时的状态。输入的alpha值将通过参数`func`指定的函数与参数`ref`进行比较。如果该值通过了这项比较，并通过了随后的模板和深度缓冲测试，则输入的片断将被绘出。如果比较失败，则帧缓冲区中的像素存储单元将不发生任何变化。可以采用的比较函数如下：

GL_NEVER	永不通过。
GL_LESS	如果输入的alpha值小于参考值，则通过。
GL_EQUAL	如果输入的alpha值等于参考值，则通过。
GL_LEQUAL	如果输入的alpha值小于或等于参考值，则通过。
GL_GREATER	如果输入的alpha值大于参考值，则通过。
GL_NOTEQUAL	如果输入的alpha值不等于参考值，则通过。
GL_GEQUAL	如果输入的alpha值大于或等于参考值，则通过。
GL_ALWAYS	总是通过（初始值）。

函数`glAlphaFunc()`将作用于所有像素的写操作，包括点、线、多边形和位图的搜索转化的结果及对像素进行的绘制和拷贝操作。函数`glAlphaFunc()`对清除屏幕操作无效。

- 注意：

该测试只能在RGBA模式下执行。

- 出错提示：

如果参数`func`不是一个可取值，则产生`GL_INVALID_ENUM`提示。

如果函数`glAlphaFunc()`在函数对`glBegin()`/`glEnd()`之间执行，则产生`GL_INVALID_OPERATION`提示。

- 有关数据的获取：

```
glGet(GL_ALPHA_TEST_FUNC)
glGet(GL_ALPHA_TEST_REF)
glIsEnabled(GL_ALPHA_TEST)
```

- 请参阅：

`glBlendFunc()`, `glClear()`, `glDepthFunc()`, `glEnable()`, `glStencilFunc()`

glAreTexturesResident

- 名称：

`glAreTexturesResident()`

- 功能：

确定是否将纹理装入纹理存储器。

- C描述：

```
GLboolean glAreTexturesResident( GLsizei n,
                                const GLuint *textures,
                                GLboolean *residences)
```

- 参数说明：

n 指定要查询的纹理个数。

textures 指定一个包含要查询的纹理名称的数组。

residences 指定一个数组，其中存放返回的纹理驻留状况。参数*textures*中某个元素指定的纹理驻留状况将被返回到*residences*相应的元素。

- 说明：

GL建立了一个驻留在纹理存储器中的纹理“工作集”。它们与那些没有驻留在纹理存储器中的纹理相比，能更有效地被绑定到纹理目标。

glAreTexturesResident()查询由*textures*中的元素指定的*n*个纹理的驻留状况。如果所有指定的纹理都驻留，函数**glAreTexturesResident()**将返回**GL_TRUE**，并且参数*residences*中的内容将不受干扰。如果并非所有指定的纹理都已存入存储器，函数**glAreTexturesResident()**将返回**GL_FALSE**，并且具体情况将返回到*residences*的*n*个元素中。如果*residences*中的一个元素是**GL_TRUE**，则由*texture*中相应元素指定的纹理是驻留的。

单个的一个被绑定的纹理的驻留状况也可通过查询函数**glGetTexParameter()**得到。其中：参数*target*被设置为该纹理绑定的目标，*pname*被设置为**GL_TEXTURE_RESIDENT**。这是查询一个默认纹理驻留状况的唯一方法。

- 注意：

函数**glAreTextureResidents()**只有在GL 1.1以上的版本中才可以使用。

函数**glAreTextureResidents()**返回的是调用时的纹理驻留状况，因此它不能保证其他任何时候纹理仍然处于驻留状态。

如果纹理是驻留在虚拟存储器中（这时没有纹理存储器），它们仍被认为处于驻留状态。

某些具体实现在直到第一次使用一个纹理时才载入该纹理。

- 出错提示：

如果*n*是负数，则产生**GL_INVALID_VALUE**提示。

如果参数*textures*中的所有元素均为0或没有指定一个纹理，则产生**GL_INVALID_VALUE**提示。这时函数返回**GL_FALSE**，并且参数*residences*的内容是不确定的。

当函数**glAreTexturesResident()**在函数对**glBegin()**/**glEnd()**之间执行，则产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取

函数**glGetTexParameter(GL_TEXTURE_RESIDENT)**用于检索一个当前绑定的纹理的驻留状态。

- 请参阅：

glBindTexture(), **glGetTexParameter()**, **glPrioritizeTextures()**, **glTexImage1D()**,
glTexImage2D(), **glTexImage3D()**, **glTexParameter()**

glArrayElement

- 名称:

glArrayElement()

- 功能:

用指定的顶点数组元素绘制一个顶点。

- C描述:

void glArrayElement(GLint *i*)

- 参数说明:

i 为可用的顶点数据数组指定一个索引。

- 说明:

在函数对**glBegin()**/**glEnd()**内使用函数**glArrayElement()**，其作用是为点、线和多边形图元指定顶点和属性数据。当函数**glArrayElement()**被调用时，如果**GL_VERTEX_ARRAY**是有效的，则它将通过使用位于有效数组的存储单元*i*中的顶点和属性数据绘出一个顶点。否则，**GL_VERTEX_ARRAY**无效，将不会有绘制动作发生，但与有效数组相关联的属性将被修正。

函数**glArrayElement()**是通过索引顶点数据的方式来绘制图元的，而不是按由前到后的次序通过数据数组。由于每次调用仅指定一个顶点，因此可以明确地指定每一个图元的属性。比如说可以明确地指定每个独立三角形的一个单独的法向量。

在函数对**glBegin()**/**glEnd()**之间改变数组数据将对该函数对中不连续的函数**glArrayElement()**的调用产生影响。也就是说，如果对函数**glArrayElement()**的一次调用是先于对数组数据的改变操作的，它将访问原始的数据；反之，如果对函数**glArrayElement()**的一次调用是在对数组数据的改变操作之后，它将访问已改变的数组。

- 注意:

函数**glArrayElement()**只有在GL 1.1以上的版本中才可以使用。

函数**glArrayElement()**是包含在显示列表中的。如果函数**glArrayElement()**被放进了一个显示列表，则它所需的数组数据（由数组指针确定且要有效）也应被放进显示列表。由于数组指针和该数组的有效化是客户端状态，所以它们的值将影响显示列表的建立的而不是显示列表的执行。

- 请参阅:

glClientActiveTextureARB(), **glColorPointer()**, **glDrawArrays()**, **glEdgeFlagPointer()**,
glGetPointerv(), **glIndexPointer()**, **glInterleavedArrays()**, **glNormalPointer()**,
glTexCoordPointer(), **glVertexPointer()**

glBegin, glEnd

- 名称:

glBegin(), **glEnd()**

- 功能:

界定一个或一组相似图元的顶点。

- C描述:

```
void glBegin( GLenum mode )
```

- 参数说明：

mode 指定由函数对glBegin()/glEnd()提供的顶点所要创建的图元。它可取10个符号常量：**GL_POINTS**、**GL_LINES**、**GL_LINE_STRIP**、**GL_LINE_LOOP**、**GL_TRIANGLES**、**GL_TRIANGLE_STRIP**、**GL_TRIANGLE_FAN**、**GL_QUADS**、**GL_QUAD_STRIP**和**GL_POLYGON**。

- C描述：

```
void glEnd( void )
```

- 说明：

函数glBegin()和glEnd()用来界定一个或一组相似图元的顶点。函数glBegin()带有一个自变量*mode*，该值用来说明指定的顶点具体属于以下所列的10种类型中的哪一种。这里*n*是由1开始的整数，*N*是被指定的总的顶点个数。

GL_POINTS

把每个顶点看作一个单独的点。顶点*n*定义了点*n*，共绘出*N*个点。

GL_LINES

把每对相邻的顶点绘成一条独立的线段。顶点 $2n-1$ 和 $2n$ 定义了线段*n*，共绘出*N/2*条线段。

GL_LINE_STRIP

把*n*个的顶点顺次连成一条连续的折线。顶点*n*和*n+1*定义了线段*n*，共绘出*N-1*条线段。

GL_LINE_LOOP

把*n*个的顶点顺次连成一条连续的折线框。顶点*n*和*n+1*定义了线段*n*，共绘出*N*条线段。

GL_TRIANGLES

相邻三个顶点构成一个三角形，且它们两两独立。顶点 $3n-2$ 、 $3n-1$ 和 $3n$ 定义了三角形*n*，共绘出*N/3*个三角形。

GL_TRIANGLE_STRIP

绘出一组相互衔接的三角形，每个三角形均由连续的三个顶点确定。对于奇数*n*，顶点*n*、*n+1*和*n+2*定义了三角形*n*；对于偶数*n*，顶点*n+1*、*n*和*n+2*定义了三角形*n*，共绘出*N-2*个三角形。

GL_TRIANGLE_FAN

绘出一组相互衔接的三角形，每个三角形均由顶点1和其他任意两个连续的顶点确定。顶点1、*n+1*和*n+2*定义了三角形*n*，共绘出*N-2*个三角形。

GL_QUADS

相邻四个顶点构成一个四边形，且它们两两独立。顶点 $4n-3$ 、 $4n-2$ 、 $4n-1$ 和 $4n$ 定义了四边形*n*，共绘出*N/4*个四边形。

GL_QUAD_STRIP

绘出一组相互衔接的四边形，每一个四边形由第一对顶点和随后的一对顶点确定。顶点 $2n-1$ 、 $2n$ 、 $2n+2$ 和 $2n+1$ 定义了四边形*n*，共绘出*N/2-1*个四边形。注意：用来构造一个四边形的顶点的顺序与相互独立的顶点数据的顺序不同。

GL_POLYGON

绘出一个凸多边形。该多边形由顶点顺次连接而成。

只有部分GL命令能在函数对glBegin()/glEnd()之间使用。这些命令是glVertex()、glColor()、glIndex()、glNormal()、glTexCoord()、glEvalCoord()、glEvalPoint()、glArrayElement()、glMaterial()和glEdgeFlag()。当然，在函数对glBegin()/glEnd()之间调用函数glCallList()或glCallLists()执行只包含上面所列命令的显示列表也是允许的。除此之外的其他GL命令如果在函数对glBegin()/glEnd()之间执行，则出错标记会置位并忽略这些命令。

不管参数*mode*选取了怎样的值，可在函数对glBegin()/glEnd()之间定义的顶点的个数都不受限制。指定的线段、三角形、四边形和多边形若不完整则不被绘出。当提供了太少的顶点来绘制一个多边形或指定的顶点数是一个不正确的倍数时，就产生了一种不完整指定。不完整的图元被忽略，其他的图元被绘出。

绘制每一图元时应指定的最小顶点个数如下：点—1个；线段—2个；三角形—3个；四边形—4个；多边形—3个。需要特定多个顶点的模式有：GL_LINE(2)、GL_TRIANGLES(3)、GL_QUADS(4)和GL_QUAD_STRIP(2)。

- 出错提示：

如果参数*mode*被设置成了一个不能接受的值，则产生GL_INVALID_ENUM提示。

当函数glBegin()在函数对glBegin()/glEnd()之间执行时产生GL_INVALID_OPERATION提示。

当函数glEnd()在函数对glBegin()/glEnd()之间执行时产生GL_INVALID_OPERATION提示。

如果在函数对glBegin()/glEnd()之间执行的命令不是glVertex()、glColor()、glIndex()、glNormal()、glTexCoord()、glEvalCoord()、glEvalPoint()、glArrayElement()、glMaterial()、glEdgeFlag()、glCallList()或glCallLists()，则产生GL_INVALID_OPERATION提示。

在函数对glBegin()/glEnd()之间执行glEnableClientState()、glDisableClientState()、glEdgeFlagPointer()、glTexCoordPointer()、glColorPointer()、glIndexPointer()、glNormalPointer()、glVertexPointer()、glInterleavedArrays()或glPixelStore()是不允许的，但这时不一定会产生出错提示。

- 请参阅：

glArrayElement(), **glCallList()**, **glCallLists()**, **glColor()**, **glEdgeFlag()**, **glEvalCoord()**,
glEvalPoint(), **glIndex()**, **glMaterial()**, **glNormal()**, **glTexCoord()**, **glVertex()**

glBindTexture

- 名称：

glBindTexture()

- 功能：

将一个指定的纹理绑定到一个纹理目标。

- C描述：

```
void glBindTexture( GLenum target,
```

GLuint *texture*)

- 参数说明:

target 指定纹理绑定的目标。它必须是**GL_TEXTURE_1D**、**GL_TEXTURE_2D**或**GL_TEXTURE_3D**。

texture 指定一个纹理的名称。

- 说明:

函数**glBindTexture()**允许用户建立或使用一个指定的纹理。参数*target*可被设置为**GL_TEXTURE_1D**、**GL_TEXTURE_2D**和**GL_TEXTURE_3D**。参数*texture*用于指定一个纹理的名称。调用带有参数*target*和*texture*的函数**glBindTexture()**将把一个名为*texture*的新纹理绑定到纹理目标中。当一个纹理被绑定到某个目标时，该目标以前的绑定将自动断开。

纹理名是不带符号的整数。值0被保留以便作为每个纹理目标的缺省纹理。纹理的名称和相应的纹理内容被存入当前GL绘图环境的共享显示列表空间（请参阅**glXCreateContext()**）。仅当两个绘图环境共用一个显示列表时它们才共用一个纹理名称。

你可以用函数**glGenTextures()**来产生一组新的纹理名称。

当一个纹理第一次被绑定时，它将假定目标的维数：首先绑定到**GL_TEXTURE_1D**的纹理是一维的；首先绑定到**GL_TEXTURE_2D**的纹理是二维的；首先绑定到**GL_TEXTURE_3D**的纹理是三维的。第一次绑定的一维纹理的状态与GL初始化时其缺省值是**GL_TEXTURE_1D**时所产生的纹理的状态是完全一样的。二维纹理和三维纹理的情况同一维类似。

当一个纹理被绑定后，GL对于其连接的目标的操作将影响这一纹理，并且对它所绑定的目标进行的查询将返回绑定纹理的情况。如果一个纹理的绑定目标的维数的纹理映射操作是处于激活状态，则该纹理可用。事实上，这时纹理目标已经成了与它相连接的纹理的别名。初始化时，纹理名0代表与它相连接的缺省纹理。

由函数**glBindTexture()**建立的一个纹理绑定将一直有效直到另一个不同的纹理连接到与它相同的目标上，或者绑定的纹理被函数**glDeleteTextures()**删除为止。

一个指定的纹理一旦建立后，只要需要，它就可以在任意时刻被再次连接到与它维数匹配的其他目标上。通常情况下，用函数**glBindTexture()**绑定一个已存在的指定纹理到一个纹理目标要比用函数**glTexImage1D()**、**glTexImage2D()**和**glTexImage3D()**重新装入一个纹理图像快得多。在函数执行时如果要增加另外的控制，可用函数**glPrioritizeTextures()**。

- 注意:

函数**glBindTexture()**只有在GL 1.1以上的版本中才可以使用。

- 出错提示:

如果参数*target*不是一个允许的值，则产生**GL_INVALID_ENUM**提示。

当参数*texture*的维数与*target*的维数不匹配时产生**GL_INVALID_OPERATION**提示。

当函数**glBindTexture()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取:

glGet(GL_TEXTURE_BINDING_1D)

`glGet(GL_TEXTURE_BINDING_2D)`
`glGet(GL_TEXTURE_BINDING_3D)`

- 请参阅：

`glAreTexturesResident()`, `glDeleteTextures()`, `glGenTextures()`, `glGet()`, `glGetTexParameter()`,
`glIsTexture()`, `glPrioritizeTextures()`, `glTexImage1D()`, `glTexImage2D()`, `glTexParameter()`

• `glBitmap`

- 名称：

`glBitmap()`

- 功能：

绘制一个位图。

- C 描述：

```
void glBitmap( GLsizei width,
               GLsizei height,
               GLfloat xorig,
               GLfloat yorig,
               GLfloat xmove,
               GLfloat ymove,
               const GLubyte *bitmap )
```

- 参数说明：

width, *height* 指定一个位图的宽度和高度。

xorig, *yorig* 指定位图中原点的位置。原点位置从位图的左下角位置开始度量，向右和向上的方向是轴的正方向。

xmove, *ymove* 指定位图绘制完成后当前光栅在*x*轴和*y*轴上的偏移量。

bitmap 指定位图的地址。

- 说明：

位图是一种二进制图像。绘制时，位图的位置与当前光栅的位置紧密相关。当帧缓冲区中的像素在位图中相应的值为1时，用当前光栅的颜色或索引绘制；当帧缓冲区中的像素在位图中相应的值为0时，将不对其进行修改。

函数`glBitmap()`带有七个参数：第一对参数用来指定位图的宽度和高度；第二对参数用来指定位图原点相对于位图左下角的位置；第三对参数用来指定位图绘制完成后当前光栅在*x*轴和*y*轴上的偏移量；最后一个参数是位图自身的一个指针。

位图跟图像数据一样用命令`glDrawPixels()`绘制。命令中的宽度和高度参数用位图的*width*和*height*替换，*type*设置为`GL_BITMAP`，*format*设置为`GL_COLOR_INDEX`。由函数`glPixelStore()`指定的模式将对位图数据的解释产生影响，而由函数`glPixelTransfer()`指定的模式则不会。如果当前光栅位置无效，则函数`glBitmap()`将被忽略。否则，位图的左下角位置在窗口中的坐标如下：

$$x_w = x_r - x_o$$

$$y_w = y_r - y_o$$

这里 (x_r, y_r) 表示光栅的位置， (x_o, y_o) 表示位图原点。与每个像素相应的在位图中的值如果是1，将产生片断。这些片断将使用当前光栅的z坐标、颜色或颜色索引以及当前光栅的纹理坐标生成。它们将被等同于由点、线或多边形所生成的一样。这里包括纹理映射、雾化以及所有对单个片断所进行的诸如alpha和深度测试等操作。

当位图被绘制完成后，当前光栅的x和y坐标可由xmove和ymove偏移量计算得到。这时光栅的z坐标、当前光栅的颜色、纹理坐标和索引都保持不变。

- 注意：

如果要设置一个视口之外的有效光栅位置，请首先在视口中设置一个有效的位置，然后调用函数glBitmap()并使用NULL作为参数bitmap的值，利用偏移量xmove和ymove来设定新的光栅位置。这一技巧对于沿视口产生一个图像非常有用。

- 出错提示：

如果参数width或height是负数，则产生GL_INVALID_VALUE提示。

当函数glBitmap()在函数对glBegin()/glEnd()之间执行时产生GL_INVALID_OPERATION提示。

- 有关数据的获取：

```
glGet( GL_CURRENT_RASTER_POSITION )
glGet( GL_CURRENT_RASTER_COLOR )
glGet( GL_CURRENT_RASTER_DISTANCE )
glGet( GL_CURRENT_RASTER_INDEX )
glGet( GL_CURRENT_RASTER_TEXTURE_COORDS )
glGet( GL_CURRENT_RASTER_POSITION_VALID )
```

- 请参阅：

glDrawPixels(), **glPixelStore()**, **glPixelTransfer()**, **glRasterPos()**

glBlendColor

- 名称：

glBlendColor()

- 功能：

设置融合颜色。

- C描述：

```
void glBlendColor( GLclampf red,
                    GLclampf green,
                    GLclampf blue,
                    GLclampf alpha )
```

- 参数说明：

red, green, blue, alpha 指定**GL_BLEND_COLOR**的组分。

- 说明：

GL_BLEND_COLOR可以被用来计算源融合因子和目的融合因子。存储前，颜色组分的取值范围是[0, 1]。请参阅**glBlendFunc()**以获取有关融合操作的完整说明。初始化时，**GL_BLEND_COLOR**被设置为(0, 0, 0, 0)。

- 注意：

函数**glBlendColor()**是**GL_ARB_imaging**子集的一部分。当调用函数**glGetString(GL_EXTENSIONS)**时，如果其返回值是**GL_ARB_imaging**，函数**glBlendColor()**才可以使用。

- 出错提示：

当函数**glBlendColor()**在函数对**glBegin()/glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGet(GL_BLEND_COLOR)

- 请参阅：

glBlendEquation(), glBlendFunc(), glGetString()

glBlendEquation

- 名称：

glBlendEquation()

- 功能：

设置融合方程。

- C描述：

void glBlendEquation(GLenum mode)

- 参数说明：

mode 指定源颜色和目标颜色的融合方式。它必须取**GL_FUNC_ADD**、**GL_FUNC_SUBTRACT**、**GL_FUNC_REVERSE_SUBTRACT**、**GL_MIN**或**GL_MAX**。

- 说明：

融合方程用来决定一个新的像素（“源”颜色）如何跟帧缓存中已有的像素（“目标”颜色）相融合。

GL_MIN:

设置一个融合方程，使得通过该方程所得到的颜色组分是相应的源颜色组分和目标颜色组分的最小值。

GL_MAX:

设置一个融合方程，使得通过该方程所得到的颜色组分是相应的源颜色组分和目标颜色组分的最大值。

剩下的融合方程将用到由函数**glBlendFunc()**所指定的源和目标融合因子。有关各种融合因子的介绍请参阅**glBlendFunc()**。

在下面的方程中，源颜色组分和目标颜色组分分别由 (R_s, G_s, B_s, A_s) 和 (R_d, G_d, B_d, A_d) 表示。所生成的颜色由 (R_f, G_f, B_f, A_f) 表示。源缩放因子和目标缩放因子分别由 (s_s, s_G, s_B, s_A) 和 (d_s, d_G, d_B, d_A) 表示。对于这些方程而言，所有的颜色组分值都被看作在范围[0, 1]内。

$$\begin{aligned} R_f &= \min(1, R_s s_s + R_d d_s) \\ G_f &= \min(1, G_s s_G + G_d d_G) \\ B_f &= \min(1, B_s s_B + B_d d_B) \\ A_f &= \min(1, A_s s_A + A_d d_A) \end{aligned}$$

GL_FUNC_ADD:

它将设置一个融合方程，通过该方程所得到的颜色组分是相应的源颜色组分和目标颜色组分的加和值。每个源颜色组分将乘上相应的源因子，每个目标颜色组分将乘上相应的目标因子。最后的组分值将是这两个乘积值的加和形式（如下），最后所得的值都被限制在范围[0, 1]内。

$$\begin{aligned} R_f &= \max(0, R_s s_s + R_d d_s) \\ G_f &= \max(0, G_s s_G + G_d d_G) \\ B_f &= \max(0, B_s s_B + B_d d_B) \\ A_f &= \max(0, A_s s_A + A_d d_A) \end{aligned}$$

GL_FUNC_SUBTRACT:

与**GL_FUNC_ADD**类似，唯一不同的是通过该方程所得到的颜色组分是源颜色因子和源颜色的乘积减去相应的目标因子和目标颜色的乘积所得的值（如下），最后所得的值都被限制在范围[0, 1]内。

$$\begin{aligned} R_f &= \max(0, R_d d_s - R_s s_s) \\ G_f &= \max(0, G_d d_G - G_s s_G) \\ B_f &= \max(0, B_d d_B - B_s s_B) \\ A_f &= \max(0, A_d d_A - A_s s_A) \end{aligned}$$

GL_FUNC_REVERSE_SUBTRACT:

与**GL_FUNC_ADD**类似，唯一不同的是通过该方程所得到的颜色组分是目标因子和目标颜色的乘积减去相应源颜色因子和源颜色的乘积所得的值（如下），最后所得的值都被限制在范围[0, 1]内。

GL_MIN和**GL_MAX**方程更多地用来分析图像数据（例如，当图像的极限是一种常量颜色时）。**GL_FUNC_ADD**则更多地用于其他物体中的反走样和透明度操作中。

初始情况下的融合方程是**GL_FUNC_ADD**。

- 注意：

函数**glBlendEquation()**是**GL_ARB_imaging**子集的一部分。只有当调用函数**glGetString(GL_EXTENSIONS)**的返回值是**GL_ARB_imaging**时，函数**glBlendEquation()**才被支持。

GL_MIN和**GL_MAX**方程并不使用源因子和目标因子，仅用到源颜色和目标颜色。

- 出错提示：

当参数*mode*不是**GL_FUNC_ADD**、**GL_FUNC_SUBTRACT**、**GL_FUNC_REVERSE_SUBTRACT**、**GL_MAX**或**GL_MIN**之一时产生**GL_INVALID_ENUM**操作。

当函数glBlendEquation()在函数对glBegin()/glEnd()之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGet(GL_BLEND_EQUATION)

- 请参阅：

glGetString(), **glBlendColor()**, **glBlendFunc()**

glBlendFunc

- 名称：

glBlendFunc()

- 功能：

指定像素运算方法。

- C描述：

```
void glBlendFunc( GLenum sfactor,
                  GLenum dfactor )
```

- 参数说明：

sfactor 指定如何计算红、绿、蓝和alpha的源融合因子。它可取的9个符号常量为：

GL_ZERO、**GL_ONE**、**GL_DST_COLOR**、**GL_ONE_MINUS_DST_COLOR**、**GL_SRC_ALPHA**、**GL_ONE_MINUS_SRC_ALPHA**、**GL_DST_ALPHA**、**GL_ONE_MINUS_DST_ALPHA**和**GL_SRC_ALPHA_SATURATE**。其缺省值是**GL_ONE**。

另外，如果系统支持**GL_ARB_imaging**扩展，它还可取以下的值：**GL_CONSTANT_COLOR**、**GL_ONE_MINUS_CONSTANT_COLOR**、**GL_CONSTANT_ALPHA**和**GL_ONE_MINUS_CONSTANT_ALPHA**。

dfactor 指定如何计算红、绿、蓝和alpha的目标融合因子。它可以取下面8个符号常量之一：

GL_ZERO、**GL_ONE**、**GL_SRC_COLOR**、**GL_ONE_MINUS_SRC_COLOR**、**GL_SRC_ALPHA**、**GL_ONE_MINUS_SRC_ALPHA**、**GL_DST_ALPHA**和**GL_ONE_MINUS_DST_ALPHA**。其缺省值是**GL_ZERO**。

另外，如果系统支持**GL_ARB_imaging**扩展，它还可以取下面的值：**GL_CONSTANT_COLOR**、**GL_ONE_MINUS_CONSTANT_COLOR**、**GL_CONSTANT_ALPHA**和**GL_ONE_MINUS_CONSTANT_ALPHA**。

- 说明：

在RGBA模式下，可用一个函数把输入的（源的）RGBA值和已存在于帧缓冲区中的RGBA值（目标值）融合在一起。初始情况下，融合操作被关闭。你可以调用函数**glEnable(GL_BLEND)**和**glDisable(GL_BLEND)**来启动和关闭融合操作。

当融合操作开启后，可以用函数**glBlendFunc()**来定义具体的融合方式。参数*sfactor*用来指定究竟采用9种方式中的哪种来对源颜色组分进行缩放。参数*dfactor*用来指定究竟采用8种方式

中的哪种来对目标颜色组分进行缩放。具体采用的方式见表5-2。每种方式都定义了4个缩放因子，分别为红、绿、蓝和alpha所用。

在表5-2和随后的方程中，源颜色组分和目标颜色组分分别由 (R_s, G_s, B_s, A_s) 和 (R_d, G_d, B_d, A_d) 表示。由函数glBlendFunc()指定的颜色由 (R_c, G_c, B_c, A_c) 表示。所有这些数都被认为是 $0 \sim (k_r, k_g, k_b, k_a)$ 之间的整数。在此

$$k_c = 2^{m_c} - 1$$

且 (m_r, m_g, m_b, m_a) 是红、绿、蓝和alpha组分中位面的数目。

源缩放因子和目标缩放因子分别由 (s_r, s_g, s_b, s_a) 和 (d_r, d_g, d_b, d_a) 表示。表5-2中所示的 (f_r, f_g, f_b, f_a) 既表示源缩放因子也表示目标缩放因子。所有缩放因子的取值范围均为[0, 1]。

表 5-2

参数	(f_r, f_g, f_b, f_a)
GL_ZERO	$(0, 0, 0, 0)$
GL_ONE	$(1, 1, 1, 1)$
GL_SRC_COLOR	$(R_s/k_r, G_s/k_g, B_s/k_b, A_s/k_a)$
GL_ONE_MINUS_SRC_COLOR	$(1, 1, 1, 1) - (R_s/k_r, G_s/k_g, B_s/k_b, A_s/k_a)$
GL_DST_COLOR	$(R_d/k_r, G_d/k_g, B_d/k_b, A_d/k_a)$
GL_ONE_MINUS_DST_COLOR	$(1, 1, 1, 1) - (R_d/k_r, G_d/k_g, B_d/k_b, A_d/k_a)$
GL_SRC_ALPHA	$(A_s/k_a, A_s/k_a, A_s/k_a, A_s/k_a)$
GL_ONE_MINUS_SRC_ALPHA	$(1, 1, 1, 1) - (A_s/k_a, A_s/k_a, A_s/k_a, A_s/k_a)$
GL_DST_ALPHA	$(A_d/k_a, A_d/k_a, A_d/k_a, A_d/k_a)$
GL_ONE_MINUS_DST_ALPHA	$(1, 1, 1, 1) - (A_d/k_a, A_d/k_a, A_d/k_a, A_d/k_a)$
GL_SRC_ALPHA_SATURATE	$(i, i, i, 1)$
GL_CONSTANT_COLOR	(R_c, G_c, B_c, A_c)
GL_ONE_MINUS_CONSTANT_COLOR	$(1, 1, 1, 1) - (R_c, G_c, B_c, A_c)$
GL_CONSTANT_ALPHA	(A_c, A_c, A_c, A_c)
GL_ONE_MINUS_CONSTANT_ALPHA	$(1, 1, 1, 1) - (A_c, A_c, A_c, A_c)$

表5-2中，

$$i = \min(A_s, k_A - A_d) / k_A$$

在RGBA模式下，系统通过下面的方程来确定所要绘制的像素的RGBA融合值：

$$\begin{aligned} R_d &= \min(k_R, R_s s_R + R_d d_R) \\ G_d &= \min(k_G, G_s s_G + G_d d_G) \\ B_d &= \min(k_B, B_s s_B + B_d d_B) \\ A_d &= \min(k_A, A_s s_A + A_d d_A) \end{aligned}$$

虽然以上方程表面上看很精确，但由于融合操作是针对不精确的整型颜色值所采取的一种操作，所以融合算法不可能很精确。但是，当一个融合因子等于1时，就意味着不必再修正其被乘的数；当一个融合因子等于0时，等同于与它相乘的数减小为0。例如，当sfactor是GL_SRC_ALPHA，dfactor是GL_ONE_MINUS_SRC_ALPHA，并且 A_s 等于 k_a 时，上述方程将简化为简单的等式：

$$\begin{aligned} R_d &= R_s \\ G_d &= G_s \\ B_d &= B_s \\ A_d &= A_s \end{aligned}$$

- 示例：

透明度的最佳实现是通过应用融合函数（**GL_SRC_ALPHA**和**GL_ONE_MINUS_SRC_ALPHA**）。其中的图元由远到近关系排序。值得注意的是这里的透明度计算不需要提供存储于帧缓冲区中的alpha位面。

融合函数（**GL_SRC_ALPHA**, **GL_ONE_MINUS_SRC_ALPHA**）对任意顺序的点和线的反走样也很有用。多边形反走样也是用融合函数（**GL_SRC_ALPHA**, **GL_ONE**）和从近到远的多边形排序顺序来进行优化的。（请参阅**glEnable(GL_POLYGON_SMOOTH)**和**glDisable(GL_POLYGON_SMOOTH)**）。为了使操作正确，必须为函数提供目标alpha位面来存放累积覆盖区域。

- 注意：

输入的（源的）alpha值被看作是材质的不透明度，其变动范围从1.0 (K_a) 完全不透明到0.0 (0)完全透明。当允许在不止一个颜色缓冲区中进行绘制时，**GL**将对每一个允许的缓冲区单独进行融合操作，并把各个颜色缓冲区中的内容作为目标颜色（请参阅**glDrawBuffer()**）。

融合操作仅在RGBA模式下有效。在颜色索引模式下，它将被忽略。

只有当你的设备支持**GL_ARB_imaging**扩展时，**GL_CONSTANT_COLOR**、**GL_ONE_MINUS_CONSTANT_COLOR**、**GL_CONSTANT_ALPHA**和**GL_ONE_MINUS_CONSTANT_ALPHA**才可以被使用。

- 出错提示：

如果参数*sfactor*或*dfactor*不是一个可接受的值，则产生**GL_INVALID_ENUM**提示。

当函数**glBlendFunc()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGet(GL_BLEND_SRC)
glGet(GL_BLEND_DST)
glIsEnabled(GL_BLEND)

- 请参阅：

glAlphaFunc(), **glBlendColor()**, **glBlendEquation()**, **glClear()**, **glDrawBuffer()**, **glEnable()**,
glLogicOp(), **glStencilFunc()**

glCallList

- 名称：

glCallList()

- 功能：

执行一个显示列表。

- C描述：

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

- 参数说明：

list 指定要执行的显示列表的整数名称。

- 说明：

函数glCallList()的作用是执行一个指定的显示列表。存储于显示列表中的命令将按序执行，其结果跟不使用显示列表而直接执行这些命令一样。如果list没有被定义成一个显示列表，命令glCallList()将被忽略。

函数glCallList()自身也可以出现在显示列表中。为避免由显示列表调用其他显示列表时出现死循环，GL限定了一个显示列表执行时的最大嵌套层数。具体的值与实现有关，但至少是64。

函数glCallList()执行时，GL的状态不被存储和返回。因此，在一个显示列表的执行过程中对GL状态的改变将一直持续到显示列表执行完成为止。可用函数glPushAttrib()、glPopAttrib()、glPushMatrix()和glPopMatrix()来跨多个glCallList()调用保存GL的状态。

- 注意：

只要显示列表中包含的命令都是允许在函数对glBegin()/glEnd()之间执行的，那么显示列表就可以在该函数对之间执行。

- 有关数据的获取：

```
glGet( GL_MAX_LIST_NESTING )
```

glIsList()

- 请参阅：

glCallLists(), glDeleteLists(), glGenLists(), glNewList(), glPushAttrib(), glPushMatrix()

glCallLists

- 名称：

glCallLists()

- 功能：

执行一组显示列表。

- C描述：

```
void glCallLists(GLsizei n,
```

```
                      GLenum type,
```

```
                      const GLvoid *lists )
```

- 参数说明：

n 指定要执行的显示列表的数目。

type 指定lists中数据的类型。它可以是下面的值：GL_BYTE、GL_UNSIGNED_BYTE、GL_SHORT、GL_UNSIGNED_SHORT、GL_INT、GL_UNSIGNED_INT、GL_FLOAT、GL_2_BYTES、GL_3_BYTES和GL_4_BYTES。

lists 指定显示列表中名称数组地址的偏移量。这一指针的类型是void，因为偏移量可以是字节型、短整型、整型、长整型或浮点型，具体取值由参数*type* 的值决定。

- 说明：

函数glCallLists()使通过*lists*而指定的名称序列中的每个显示列表被执行。这样，存储于每一显示列表中的命令被顺序执行，其结果就像不用显示列表而直接调用这些命令一样。未定义名称的显示列表将被忽略。

当多个显示表需要执行时，函数glCallLists()提供了一种非常有效的方法。参数*type*允许显示列表接受以下几种类型名称格式：

GL_BYTE 参数*lists*被看作一个带符号单字节构成的数组。其范围是[-128, 127]。

GL_UNSIGNED_BYTE

参数*lists*被看作一个无符号单字节的数组。其范围是[0, 255]。

GL_SHORT 参数*lists*被看作一个带符号双字节整数构成的数组。其范围是[-32768, 32767]。

GL_UNSIGNED_SHORT

参数*lists*被看作一个无符号双字节整数构成的数组。其范围是[0, 65535]。

GL_INT 参数*lists*被看作一个带符号四字节整数构成的数组。

GL_UNSIGNED_INT

参数*lists*被看作一个无符号四字节整数构成的数组。

GL_FLOAT 参数*lists*被看作一个四字节浮点值构成的数组。

GL_2_BYTES 参数*lists*被看作一个无符号单字节构成的数组。每一对字节指定一个显示列表的名称。每一对字节的值是由第一个无符号单字节的值乘上256再加上第二个无符号单字节的值而得到的。

GL_3_BYTES 参数*lists*被看作一个无符号单字节构成的数组。每三个字节指定一个显示列表的名称。该名称是由第一个无符号单字节的值乘上65536加上第二个无符号单字节的值乘上256再加上第三个无符号单字节的值而得到的。

GL_4_BYTES 参数*lists*被看作一个无符号单字节构成的数组。每四个字节指定一个显示列表的名称。该名称由第一个无符号单字节的值乘上1677216加上第二个无符号单字节的值乘上65536加上第三个无符号单字节的值乘上256再加上第四个无符号单字节的值而得到。

显示列表的名称列表不是以空结束。而是用参数*n*指定由*lists*中取出的显示列表的个数。

函数glListBase()可以设置一个附加的间接手段。它被用来在显示列表执行前，将一个无符号的偏移量加到*lists*指定的每个显示列表名称上。

函数glCallLists()可以出现在显示列表中。为避免由显示列表调用其他显示列表时出现死循环，GL限定了一个显示列表执行时的最大嵌套层数。具体的值与实现有关，但至少是64。

函数glCallLists()执行时，GL的状态不被存储和返回。因此，在一个显示列表的执行过程中对GL状态的改变将一直持续到显示列表执行完成为止。你可用函数glPushAttrib()、glPopAttrib()、glPushMatrix()和glPopMatrix()来跨多个glCallLists()调用保存GL的状态。

- 注意：

只要显示列表中包含的命令都允许在函数对glBegin()/glEnd()之间执行，那么它就可以在该函数对之间执行。

- 出错提示：

如果参数*n*为负数，则产生**GL_INVALID_VALUE**提示。

当参数*type*不是**GL_BYTE**、**GL_UNSIGNED_BYTE**、**GL_SHORT**、**GL_UNSIGNED_SHORT**、**GL_INT**、**GL_UNSIGNED_INT**、**GL_FLOAT**、**GL_2_BYTES**、**GL_3_BYTES**和**GL_4_BYTES**之一时，产生**GL_INVALID_ENUM**提示。

- 有关数据的获取：

glGet(GL_LIST_BASE)
glGet(GL_MAX_LIST_NESTING)

glIsList()

- 请参阅：

glCallList(), **glDeleteLists()**, **glGenLists()**, **glListBase()**, **glNewList()**, **glPushAttrib()**,
glPushMatrix()

glClear

- 名称：

glClear()

- 功能：

用预先设定的值清除缓冲区。

- C描述：

void glClear(GLbitfield mask)

- 参数说明：

mask 用二进制逻辑或操作来屏蔽指定的缓冲区，从而消除该缓冲区。这里有4种屏蔽操作：**GL_COLOR_BUFFER_BIT**、**GL_DEPTH_BUFFER_BIT**、**GL_ACCUM_BUFFER_BIT**和**GL_STENCIL_BUFFER_BIT**。

- 说明：

函数glClear()的作用是用预先选定的值设置窗口中的位面区域。这些预先选定的值由函数glClearColor()、glClearIndex()、glClearDepth()、glClearStencil()和glClearAccum()来选取。可以通过函数glDrawBuffer()一次选择多个缓冲区达到同时清除多个被选定的颜色缓冲区的目的。

像素的所有权测试、裁剪测试、抖动和缓冲区写屏蔽等操作都将影响到函数glClear()的使用。裁剪箱限定了清除的区域。Alpha函数、融合函数、逻辑操作、模板测试、纹理映射及深度

缓冲测试等操作都被函数glClear()忽略。

函数glClear()带有一个自变量，该值指定了一个对若干值进行的二进制逻辑或操作，用以清除指定的缓冲区。

参数mask的取值如下：

GL_COLOR_BUFFER_BIT	表示缓冲区当前对颜色写入有效。
GL_DEPTH_BUFFER_BIT	表示深度缓冲区。
GL_ACCUM_BUFFER_BIT	表示累积缓冲区。
GL_STENCIL_BUFFER_BIT	表示模板缓冲区。

究竟哪个缓冲区将被清除，取决于该缓冲区清除值的设置。

- 注意：

如果一个缓冲区不是当前缓冲区，则用函数glClear()清除那个缓冲区将不产生任何作用。

- 出错提示：

如果参数mask不是所指定的四个值之一，则产生**GL_INVALID_VALUE**提示。

当函数glClear()在函数对glBegin()/glEnd()之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

```
glGet( GL_ACCUM_CLEAR_VALUE )
glGet( GL_DEPTH_CLEAR_VALUE )
glGet( GL_INDEX_CLEAR_VALUE )
glGet( GL_COLOR_CLEAR_VALUE )
glGet( GL_STENCIL_CLEAR_VALUE )
```

- 请参阅：

glClearAccum(), **glClearColor()**, **glClearDepth()**, **glClearIndex()**, **glClearStencil()**,
glColorMask(), **glDepthMask()**, **glDrawBuffer()**, **glScissor()**, **glStencilMask()**

■ **glClearAccum**

- 名称：

glClearAccum()

- 功能：

指定累积缓冲区的清除值。

- C描述：

```
void glClearAccum( GLfloat red,
                   GLfloat green,
                   GLfloat blue,
                   GLfloat alpha )
```

- 参数说明：

red, *green*, *blue*, *alpha*

指定清除累积缓冲区时用的红、绿、蓝和alpha值。它们的初始值都是0。

- 说明：

函数glClearAccum()指定当用函数glClear()清除累积缓冲区时用的红、绿、蓝和alpha值。由函数glClearAccum()指定的值在范围[-1, 1]内。

- 出错提示：

当函数glClearAccum()在函数对glBegin()/glEnd()之间执行时产生GL_INVALID_OPERATION提示。

- 有关数据的获取：

`glGet(GL_ACCUM_CLEAR_VALUE)`

- 请参阅：

`glAccum()`, `glClear()`

glClearColor

- 名称：

`glClearColor()`

- 功能：

指定颜色缓冲区的清除值。

- C描述：

```
void glClearColor( GLclampf red,
                  GLclampf green,
                  GLclampf blue,
                  GLclampf alpha )
```

- 参数说明：

red, green, blue, alpha

指定清除颜色缓冲区时用的红、绿、蓝和alpha值。它们的初始值都是0。

- 说明：

函数glClearColor()指定当用函数glClear()清除颜色缓冲区时用的红、绿、蓝和alpha值。

由函数glClearColor()指定的值在范围[0, 1]内。

- 出错提示：

当函数glClearColor()在函数对glBegin()/glEnd()之间执行时产生GL_INVALID_OPERATION提示。

- 有关数据的获取：

`glGet(GL_COLOR_CLEAR_VALUE)`

- 请参阅：

`glClear()`

glClearDepth

- 名称：

`glClearDepth()`

- 功能：

指定深度缓冲区的清除值。

- C描述：

void glClearDepth(GLclampd *depth*)

- 参数说明：

depth 指定清除深度缓冲区时用的深度值。其初始值是1。

- 说明：

函数glClearDepth()指定当用函数glClear()清除深度缓冲区时用的深度值。

由函数glClearDepth()指定的值在范围[0, 1]内。

- 出错提示：

当函数glClearDepth()在函数对glBegin()/glEnd()之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGet(GL_DEPTH_CLEAR_VALUE)

- 请参阅：

glClear()

glClearIndex

- 名称：

glClearIndex()

- 功能：

指定颜色索引缓冲区的清除值。

- C描述：

void glClearIndex(GLfloat *c*)

- 参数说明：

c 指定清除颜色索引缓冲区时用的索引值。其初始值是0。

- 说明：

函数glClearIndex()指定当用函数glClear()清除颜色索引缓冲区时用的颜色索引值。参数*c*不被截断。反之，通过在右边补上若干个不指定精度的二进制点，*c*将被转换成一种固定点的格式。该值的整数部分将被 $2^m - 1$ 屏蔽，这里*m*是存储于帧缓冲区中的颜色索引的二进制位的数目。

- 出错提示：

当函数glClearIndex()在函数对glBegin()/glEnd()之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGet(GL_INDEX_CLEAR_VALUE)

glGet(GL_INDEX_BITS)

- 请参阅：

glClear()

glClearStencil

- 名称:

glClearStencil()

- 功能:

指定模板缓冲区的清除值。

- C描述:

void glClearStencil(GLint *s*)

- 参数说明:

s 指定清除模板缓冲区时用的索引值。其初始值是0。

- 说明:

函数**glClearStencil()**指定当用函数**glClear()**清除模板缓冲区时用的索引值。*s*将被 $2^m - 1$ 屏蔽，这里*m*是模板缓冲区的二进制位的数目。

- 出错提示:

当函数**glClearStencil()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取:

glGet(GL_STENCIL_CLEAR_VALUE)

glGet(GL_STENCIL_BITS)

- 请参阅:

glClear(), **glStencilFunc()**, **glStencilOp()**, **glStencilMask()**

glClientActiveTextureARB

- 名称:

glClientActiveTextureARB()

- 功能:

选择激活的纹理单元。

- C描述:

void glClientActiveTextureARB(GLenum *texture*)

- 参数说明:

texture 指定哪一个纹理单元将被激活。纹理单元的个数由具体实现决定，但至少要两个。

参数*texture*必须取**GL_TEXTURE*i*_ARB**之一，这里 $0 \leq i < \text{GL_MAX_TEXTURE_UNITS_ARB}$ ，它是一个与具体实现有关的值。其初始值是**GL_TEXTURE0_ARB**。

- 说明:

函数**glClientActiveTextureARB()**用来选择由函数**glTexCoordPointer()**修改的顶点数组的客户状态参数。通过调用函数**glEnableClientState(GL_TEXTURE_COORD_ARRAY)**和**glDisableClientState(GL_TEXTURE_COORD_ARRAY)**来启动和关闭该操作。

- 注意：

当调用函数glGetString(GL_EXTENSIONS)所返回的信息串中包含**GL_ARB_multitexture**时，函数glClientActiveTextureARB()才被支持。

函数glClientActiveTextureARB()将把**GL_CLIENT_ACTIVE_TEXTURE_ARB**设置成被激活的纹理单元。

- 出错提示：

如果参数*texture*不是任何一个**GL_TEXTURE*i*_ARB**，则产生**GL_INVALID_ENUM**提示。这里 $0 \leq i < \text{GL_MAX_TEXTURE_UNITS_ARB}$ 。

- 请参阅：

glActiveTextureARB(), **glDisableClientState()**, **glEnableClientState()**, **glMultiTexCoordARB()**, **glTexCoordPointer()**

glClipPlane

- 名称：

glClipPlane()

- 功能：

指定一个剪切所有几何体所用的平面。

- C描述：

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

- 参数说明：

plane 指定剪切面的名称。符号名称**GL_CLIP_PLANE*i***都可以被接受，这里*i*是从0到**GL_MAX_CLIP_PLANES-1**之间的一个整数。

equation 指定一个包含4个双精度浮点数的数组的地址。这些值定义了一个平面方程。

- 说明：

一个几何体总是被一个六面体的边界沿x、y和z轴方向剪切而得到。函数glClipPlane()允许用户定义另外的剪切平面。这些剪切平面不必与x、y和z轴垂直。剪切面外的物体同样被剪切掉。要查询系统允许的附加剪切平面的最大值可调用函数glGetInteger(GL_MAX_CLIP_PLANES)。所有的实现机制均提供至少六个这样的剪切平面。剪切后所得的区域是已定义的半空间的交集，所以它们均为凸多边形。

函数glClipPlane()用四个变量的平面方程指定一个半空间。当函数glClipPlane()被调用时，参数*equation*将通过模式取景矩阵的逆来进行变换，并被存储为眼坐标形式。随后对模式取景矩阵进行的改变将不再对已存储的平面方程的组分产生影响。如果通过已存储的平面方程组分求得的一个顶点的眼坐标是大于或等于0的，则说明该顶点在剪切平面内。否则，该顶点在剪切平面外。

如果要启动和关闭某一个剪切平面，请调用函数glEnable(GL_CLIP_PLANE*i*)和glDisable(GL_CLIP_PLANE*i*)。这里*i*是平面号。

初始情况下，所有剪切平面在眼坐标体系中均由(0, 0, 0, 0)定义，且处于关闭状态。

- 注意：

下面等式总是成立的：

$$\text{GL_CLIP_PLANE}_i = \text{GL_CLIP_PLANE}0 + i$$

- 出错提示：

如果参数*plane*是一个不能接受的值，则产生**GL_INVALID_ENUM**提示。

当函数**glClipPlane()**在函数对**glBegin()/glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGetClipPlane()

glIsEnabled(GL_CLIP_PLANE*i*)

- 请参阅：

glEnable()

3. glColor

- 名称：

glColor3b(), **glColor3d()**, **glColor3f()**, **glColor3i()**, **glColor3s()**, **glColor3ub()**, **glColor3ui()**,
glColor3us(), **glColor4b()**, **glColor4d()**, **glColor4f()**, **glColor4i()**, **glColor4s()**, **glColor4ub()**,
glColor4ui(), **glColor4us()**, **glColor3bv()**, **glColor3dv()**, **glColor3fv()**, **glColor3iv()**,
glColor3sv(), **glColor3ubv()**, **glColor3uiv()**, **glColor3usv()**, **glColor4bv()**, **glColor4dv()**,
glColor4fv(), **glColor4iv()**, **glColor4sv()**, **glColor4ubv()**, **glColor4uiv()**, **glColor4usv()**

- 功能：

设置当前颜色。

- C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)

```

• 参数说明：

red, green, blue

指定当前颜色新的红、绿和蓝值。

alpha 指定当前颜色新的alpha值。仅用于含有四个参数的glColor4()命令。

• C 描述：

```

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)

```

• 参数说明：

v 指定一个指针，指向一个包含红、绿、蓝和alpha（有时有）值的数组。

• 说明：

GL保存着当前的单值颜色索引和当前的四值的RGBA颜色。函数glColor()用于设置一组新的RGBA值。函数glColor()主要有两个变种：glColor3()和glColor4()。函数glColor3()明确地指定新的红、绿和蓝值，并隐含地将当前alpha值设置为1.0（满强度）。函数glColor4()则明确地指定了所有四个颜色组分。

函数glColor3b()、glColor4b()、glColor3s()、glColor4s()、glColor3i()和glColor4i()带有三个或四个带符号单字节、短整型和长整型参数。如果函数名中有字母“v”，则说明此函数可带有一个指向这些值的指针。

当前颜色值被存储为浮点形式，它是一种带有未定义尾数的指数形式。当指定的颜色组分是无符号的整型值时，它将被线性地映射为整型浮点格式：最大的正数映射为1.0（满强度），0映射为0.0（零强度）。当指定的颜色组分是带符号的整型值时，它将被线性地映射为整型浮点格式：最大的正数映射为1.0，最小的负数映射为-1.0。（请注意，这时的0不一定被精确地映射为0.000）浮点值则直接映射。

当前颜色被更新前，浮点值和带符号的整型值取值范围都不是[0, 1]。然而，当颜色组分被插入或写入一个颜色缓冲区时，它们应被截断到这一范围。

- 注意：

当前颜色的初始值是(1, 1, 1, 1)。

当前颜色可以被随时更新。尤其是，函数glColor()可以在函数对glBegin()/glEnd()之间调用。

- 有关数据的获取：

glGet(GL_CURRENT_COLOR)

glGet(GL_RGBA_MODE)

- 请参阅：

glIndex()

◆ **glColorMask**

- 名称：

glColorMask()

- 功能：

开启和关闭对帧缓冲区中的颜色组分的写操作。

- C描述：

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

- 参数说明：

red, green, blue, alpha

决定红、绿、蓝和alpha组分是否能被写入帧缓冲区。它们的初始值都是**GL_TRUE**，表明颜色组分可以被写入。

- 说明：

函数glColorMask()的作用是指定帧缓冲区中每个单独的颜色组分是否可以被写入。比如说，当*red*是**GL_FALSE**时，即使试图进行绘图操作，任一颜色缓冲区中的任何像素的红组分都不会改变。

对组分中单独二进制位的更改是不可控制的。然而，却可以开启和关闭对整个颜色组分的更改。

- 出错提示：

当函数glColorMask()在函数对glBegin()/glEnd()之间执行时产生GL_INVALID_OPERATION提示。

- 有关数据的获取：

glGet(GL_COLOR_WRITEMASK)

glGet(GL_RGBA_MODE)

- 请参阅：

glClear(), glColor(), glColorPointer(), glDepthMask(), glIndex(), glIndexPointer(), glIndexMask(), glStencilMask()

• glColorMaterial

- 名称：

glColorMaterial()

- 功能：

使一个材料的颜色跟踪当前颜色。

- C描述：

```
void glColorMaterial( GLenum face,
                      GLenum mode )
```

- 参数说明：

face 指定究竟是正面、反面还是正反两面的材料参数将跟踪当前的颜色。它的可取值有：GL_FRONT、GL_BACK和GL_FRONT_AND_BACK。其初始值是GL_FRONT_AND_BACK。

mode 指定跟踪当前颜色的材料参数。它的可取值有：GL_EMISSION、GL_AMBIENT、GL_DIFFUSE、GL_SPECULAR和GL_AMBIENT_AND_DIFFUSE。其初始值是GL_AMBIENT_AND_DIFFUSE。

- 说明：

函数glColorMaterial()用于指定哪个材料参数将跟踪当前颜色。当GL_COLOR_MATERIAL有效时，由参数*mode*和参数*face*指定的材料参数将随时跟踪当前的颜色。

调用函数glEnable(GL_COLOR_MATERIAL)和glDisable(GL_COLOR_MATERIAL)可以设置GL_COLOR_MATERIAL是有效还是无效。初始情况下，GL_COLOR_MATERIAL是无效的。

- 注意：

可以通过调用函数glColorMaterial()实现只使用函数glColor()来改变每个顶点的材料参数子集的目的，而不必使用函数glMaterial()。当每个顶点的这样一个子集已被指定时，使用函数glColorMaterial()要比函数glMaterial()更合适。

在GL_COLOR_MATERIAL有效前，请先调用函数glColorMaterial()。

当颜色数组有效时，调用函数glDrawElements()、glDrawArrays()或glDrawRange-

`Elements()`会使当前颜色不确定。如果当前颜色不确定，则在调用函数`glColorMaterial()`时由参数`face`和`mode`指定的光照材料状态也不确定。

在GL 1.1以上的版本中，当**GL_COLOR_MATERIAL**有效时，所求得的颜色值将对光照方程的求值产生影响，就好像当前颜色被修正过一样。但是它对当前颜色的跟踪光照参数不产生影响。

- 出错提示：

如果参数`face`和`mode`不是一个可接受的值，则产生**GL_INVALID_ENUM**提示。

当函数`glColorMaterial()`在函数对`glBegin()`/`glEnd()`之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

```
glIsEnabled( GL_COLOR_MATERIAL )
glGet( GL_COLOR_MATERIAL_PARAMETER )
glGet( GL_COLOR_MATERIAL_FACE )
```

- 请参阅：

`glColor()`, `glColorPointer()`, `glDrawArrays()`, `glDrawElements()`, `glDrawRangeElements()`,
`glEnable()`, `glLight()`, `glLightModel()`, `glMaterial()`

• `glColorPointer`

- 名称：

`glColorPointer()`

- 功能：

定义一个颜色数组。

- C描述：

```
void glColorPointer( GLint size,
                      GLenum type,
                      GLsizei stride,
                      const GLvoid *pointer )
```

- 参数说明：

`size` 指定每一颜色的组分个数。它必须是3或4。其初始值是4。

`type` 指定数组中每一颜色组分的数据类型。它的可取值有：**GL_BYTE**、**GL_UNSIGNED_BYTE**、**GL_SHORT**、**GL_UNSIGNED_SHORT**、**GL_INT**、**GL_UNSIGNED_INT**、**GL_FLOAT**和**GL_DOUBLE**。其初始值是**GL_FLOAT**。

`stride` 指定连续颜色之间的字节偏移量。如果参数`stride`是0（初始值），则这些颜色被认为是在数组中连续存放的。初始值是0。

`pointer` 指定一个指针，指向数组中的第一个颜色元素的第一个组分的位置。

- 说明：

当进行绘制操作时，函数`glColorPointer()`用来指定各颜色组分数组的数据格式及存放地址。

参数 *size* 指定每一颜色组分的个数。它必须是3或4。参数 *type* 指定数组中每一颜色组分的数据类型。参数 *stride* 指定从一个颜色到下一个封装了顶点集及属性集的数组之间的字节跨距，其中被封装的顶点集及属性集应可以存放在单一数组或不同数组中。(单一数组存放形式在某些实现中更有效一些，请参阅 **glInterleavedArrays()**)。

一个颜色数组一旦被指定，参数 *size*、*type*、*stride* 和 *pointer* 将被存储成客户端。

要启动和关闭颜色数组，可调用函数 **glEnableClientState(GL_COLOR_ARRAY)** 和 **glDisableClientState(GL_COLOR_ARRAY)**。启动之后，颜色数组可以被函数 **glDrawArrays()**、**glDrawElements()**、**glDrawRangeElements()** 或 **glArrayElement()** 访问。

- 注意：

函数 **glColorPointer()** 只有在 GL 1.1 以上的版本中才可以使用。

颜色数组在初始情况下是关闭的，所以此时函数 **glDrawArrays()**、**glDrawElements()**、**glDrawRangeElements()** 或 **glArrayElement()** 并不能访问它。

函数 **glColorPointer()** 不允许在函数对 **glBegin()**/**glEnd()** 之间执行，但这时可能不产生出错提示。如果没有产生出错提示，则该操作是未定义的。

函数 **glColorPointer()** 通常在客户端实现。

颜色数组的参数是一种客户端状态，所以它们不能由函数 **glPushAttrib()** 和 **glPopAttrib()** 存储和返回，而必须用函数 **glPushClientAttrib()** 和 **glPopClientAttrib()** 代替。

- 出错提示：

如果参数 *size* 不是3或4，则产生 **GL_INVALID_VALUE** 提示。

如果参数 *type* 不是一个可接受的值，则产生 **GL_INVALID_ENUM** 提示。

如果参数 *stride* 是负数，则产生 **GL_INVALID_VALUE** 提示。

- 有关数据的获取：

```
glIsEnabled( GL_COLOR_ARRAY )
glGet( GL_COLOR_ARRAY_SIZE )
glGet( GL_COLOR_ARRAY_TYPE )
glGet( GL_COLOR_ARRAY_STRIDE )
glGetPointerv( GL_COLOR_ARRAY_POINTER )
```

- 请参阅：

```
glArrayElement(), glDrawArrays(), glDrawElements(), glEdgeFlagPointer(), glEnable(),
glGetPointerv(), glIndexPointer(), glInterleavedArrays(), glNormalPointer(),
glPopClientAttrib(), glPushClientAttrib(), glTexCoordPointer(), glVertexPointer()
```

glColorSubTable

- 名称：

glColorSubTable()

- 功能：

重新指定部分颜色表。

- C 描述：

```
void glColorSubTable( GLenum target,
                      GLsizei start,
                      GLsizei count,
                      GLenum format,
                      GLenum type,
                      const GLvoid *data )
```

- 参数说明：

target 必须是下面各值之一： **GL_COLOR_TABLE**、**GL_POST_CONVOLUTION_COLOR_TABLE**或**GL_POST_COLOR_MATRIX_COLOR_TABLE**。

start 重新替换的部分颜色表的开始位置的索引。

count 重新替换的颜色表的条目个数。

format 参数*data*中的像素数据格式。它的允许值是：**GL_RED**、**GL_GREEN**、**GL_BLUE**、**GL_ALPHA**、**GL_LUMINANCE**、**GL_LUMINANCE_ALPHA**、**GL_RGB**、**GL_BGR**、**GL_RGBA**和**GL_BGRA**。

type 参数*data*中的像素数据类型。它的允许值是：**GL_UNSIGNED_BYTE**、**GL_BYTE**、**GL_UNSIGNED_SHORT**、**GL_SHORT**、**GL_UNSIGNED_INT**、**GL_INT**、**GL_FLOAT**、**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**。

data 它是一个指向一维数组的指针。该数组中存放着用于替换颜色表指定部分的像素数据。

- 说明：

函数**glColorSubTable()**的作用是重新指定已由函数**glColorTable()**定义的颜色表的一个连续部分。由*data*提供的像素将替换已有颜色表的*start*到*start+count-1*（包含）部分。这一区域可以不包括原来指定的颜色表范围之外的任何条目。指定一个宽度为0的子纹理是允许的，但这样的指定没有任何作用。

- 注意：

当调用函数**glGetString(GL_EXTENSIONS)**的返回值是**GL_ARB_imaging**时，函数**glColorSubTable()**才被支持。

- 出错提示：

如果参数*target*不是一个可接受值，则产生**GL_INVALID_ENUM**提示。

如果*start+count>width*，则产生**GL_INVALID_VALUE**提示。

如果参数*format*不是一个可接受值，则产生**GL_INVALID_ENUM**提示。

如果参数*type*不是一个可接受值，则产生**GL_INVALID_ENUM**提示。

当函数**glColorSubTable()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGetColorTable()

glGetColorTableParameter()

- 请参阅：

glColorSubTable(), **glColorTableParameter()**, **glCopyColorTable()**, **glCopyColorSubTable()**, **glGetColorTable()**

• **glColorTable**

- 名称：

glColorTable()

- 功能：

定义一个颜色查询表。

- C 描述：

```
void glColorTable( GLenum target,
                   GLenum internalformat,
                   GLsizei width,
                   GLenum format,
                   GLenum type,
                   const GLvoid *table )
```

- 参数说明：

target

必须是下面各值之一：**GL_COLOR_TABLE**、**GL_POST_CONVOLUTION_COLOR_TABLE**、**GL_POST_COLOR_MATRIX_COLOR_TABLE**、**GL_PROXY_COLOR_TABLE**、**GL_PROXY_POST_CONVOLUTION_COLOR_TABLE**或**GL_PROXY_POST_COLOR_MATRIX_COLOR_TABLE**

internalformat

颜色表的内部格式。它可取的值有：**GL_ALPHA**、**GL_ALPHA4**、**GL_ALPHA8**、**GL_ALPHA12**、**GL_ALPHA16**、**GL_LUMINANCE**、**GL_LUMINANCE4**、**GL_LUMINANCE8**、**GL_LUMINANCE12**、**GL_LUMINANCE16**、**GL_LUMINANCE_ALPHA**、**GL_LUMINANCE4_ALPHA4**、**GL_LUMINANCE6_ALPHA2**、**GL_LUMINANCE8_ALPHA8**、**GL_LUMINANCE12_ALPHA4**、**GL_LUMINANCE12_ALPHA12**、**GL_LUMINANCE16_ALPHA16**、**GL_INTENSITY**、**GL_INTENSITY4**、**GL_INTENSITY8**、**GL_INTENSITY12**、**GL_INTENSITY16**、**GL_R3_G3_B2**、**GL_RGB**、**GL_RGB4**、**GL_RGB5**、**GL_RGB8**、**GL_RGB10**、**GL_RGB12**、

GL_RGB16、**GL_RGBA**、**GL_RGBA2**、**GL_RGBA4**、**GL_RGB5_A1**、**GL_RGBA8**、**GL_RGB10_A2**、**GL_RGBA12**和**GL_RGBA16**。

width 由参数*table*指定的颜色查询表的条目个数。

format 参数*table*中的像素数据格式。它允许取的值是：**GL_RED**、**GL_GREEN**、**GL_BLUE**、**GL_ALPHA**、**GL_LUMINANCE**、**GL_LUMINANCE_ALPHA**、**GL_RGB**、**GL_BGR**、**GL_RGBA**和**GL_BGRA**。

type 参数*table*中的像素数据类型。它允许取的值有：**GL_UNSIGNED_BYTE**、**GL_BYTE**、**GL_UNSIGNED_SHORT**、**GL_SHORT**、**GL_UNSIGNED_INT**、**GL_INT**、**GL_FLOAT**、**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**。

table 指向一个被用来生成颜色表的一维像素数据的数组。

• 说明：

函数glColorTable()有两种用途：测试由一个特定的参数集给出的一个查询表的实际尺寸和颜色分辨率；或者载入一个颜色查询表的内容。第一种情况来用目标**GL_PROXY_***；第二种情况采用别的目标。

如果参数*target*是**GL_COLOR_TABLE**、**GL_POST_CONVOLUTION_COLOR_TABLE**或**GL_POST_COLOR_MATRIX_COLOR_TABLE**，函数glColorTable()将由一个像素数组建立一个颜色查询表。从内存中抽取由参数*width*、*format*、*type*和*table*指定的像素数组，其处理过程就像调用了函数glDrawPixels()一样。但最后扩展成RGBA颜色以后，该过程也就完成了。

这时将为查询表定义四个缩放参数和四个位移参数，它们将用于缩放和位移每个像素的R、G、B和A组分。（函数glColorTableParameter()可用来设置这些参数。）

接下来，R、G、B和A值将被截断到范围[0, 1]。于是每个像素被转化成参数*internalformat*指定的内部格式。这一转化过程是从像素的组分值（R、G、B和A）到内部格式（红、绿、蓝、alpha、亮度和强度）值的一种简单映射。具体方法见表5-3。

表 5-3

内部格式	红	绿	蓝	alpha	亮度	强度
GL_ALPHA				A		
GL_LUMINANCE					R	
GL_LUMINANCE_ALPHA				A	R	
GL_INTENSITY						R
GL_RGB	R	G	B			
GL_RGBA	R	G	B	A		

最后，产生的像素的红、绿、蓝、alpha、亮度和/或强度组分被存入颜色表。它们构成了一

一个一维颜色查询表，其索引范围是[0, width-1]。

如果参数`target`是`GL_PROXY_*`，函数`glColorTable()`将重新计算和存储某些代理颜色表的状态变量的值。这些状态变量有：`GL_COLOR_TABLE_FORMAT`、`GL_COLOR_TABLE_WIDTH`、`GL_COLOR_TABLE_RED_SIZE`、`GL_COLOR_TABLE_GREEN_SIZE`、`GL_COLOR_TABLE_BLUE_SIZE`、`GL_COLOR_TABLE_ALPHA_SIZE`、`GL_COLOR_TABLE_LUMINANCE_SIZE`和`GL_COLOR_TABLE_INTENSITY_SIZE`。这时将不影响实际颜色表的图像或状态。如果指定的颜色表太大而超出了所支持的范围，则上面所列的代理状态变量的值将全部设为0。否则，颜色表可由函数`glColorTable()`根据相应的非代理目标提供，同时像定义该目标一样对代理状态变量进行设置。

代理状态变量可通过调用带有目标`GL_PROXY_*`的函数`glGetColorTableParameter()`取得。这一方法可以确定一个特殊的`glColorTable()`命令是否成功，并决定得到的颜色表属性是怎样的。

当一个颜色表有效并且其宽度非0时，其内容将按照表的内部格式替换每个RGBA像素组的一个组分子集。

每一像素组所包含的颜色组分（R, G, B, A）的取值范围都是[0.0,1.0]。颜色组分将被重新缩放到颜色查询表所要求的尺寸范围，以便形成一个索引。接下来，一个按颜色查询表的内部格式形成的组分子集将由该索引选定的表中的条目所替代。如果颜色组分和表的内容表示如表5-4所示。

表 5-4

表 示	含 义
r	通过R计算的表的索引
g	通过G计算的表的索引
b	通过B计算的表的索引
a	通过A计算的表的索引
L[i]	在表的索引i处的亮度值
I[i]	在表的索引i处的强度值
R[i]	在表的索引i处的R值
G[i]	在表的索引i处的G值
B[i]	在表的索引i处的B值
A[i]	在表的索引i处的A值

则颜色表查询结果如表5-5所示。

表 5-5

表的内部格式	所得的纹理组分			
	R	G	B	A
<code>GL_ALPHA</code>	R	G	B	A[a]
<code>GL_LUMINANCE</code>	L[r]	L[g]	L[b]	A[t]
<code>GL_LUMINANCE_ALPHA</code>	L[r]	L[g]	L[b]	A[a]
<code>GL_INTENSITY</code>	I[r]	I[g]	I[b]	I[a]
<code>GL_RGB</code>	R[r]	G[g]	B[b]	A
<code>GL_RGBA</code>	R[r]	G[g]	B[b]	A[a]

当**GL_COLOR_TABLE**有效时，由像素映射操作（如果已开启）得到的颜色在被传递到卷积操作前将由颜色查询表映射。当**GL_POST_CONVOLUTION_COLOR_TABLE**有效时，由卷积操作得到的颜色将被随后的卷积颜色查询表修正。这些修正的结果将被送到颜色矩阵操作中。最后，如果**GL_POST_COLOR_MATRIX_COLOR_TABLE**是有效的，这些由颜色矩阵操作所得的颜色在直方图操作之前要先被随后的颜色矩阵的颜色查询表映射。

- 注意：

当调用函数**glGetString(GL_EXTENSIONS)**的返回值是**GL_ARB_imaging**时，函数**glColorTable()**才被支持。

当参数*target*被设置为**GL_COLOR_TABLE**、**GL_POST_CONVOLUTION_COLOR_TABLE**或**GL_POST_COLOR_MATRIX_COLOR_TABLE**时，参数*width*必须是2的幂值，否则将产生**GL_INVALID_VALUE**提示。

- 出错提示：

如果参数*target*不是一个可接受值，则产生**GL_INVALID_ENUM**提示。

如果参数*internalformat*不是其可取值之一，则产生**GL_INVALID_ENUM**提示。

当参数*width*小于0时产生**GL_INVALID_VALUE**提示

当参数*format*不是一个可接受值时产生**GL_INVALID_ENUM**提示。

当参数*type*不是一个可接受值时产生**GL_INVALID_ENUM**提示。

当所需的颜色表太大而不能由实现支持，并且参数*target*不是**GL_PROXY***时，则产生**GL_TABLE_TOO_LARGE**。

当函数**glColorTable()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGetColorTableParameter()

- 请参阅：

glColorSubTable(), **glColorTableParameter()**, **glCopyColorTable()**, **glCopyColorSubTable()**, **glGetColorTable()**

■ **glColorTableParameter**

- 名称：

glColorTableParameterfv(), **glColorTableParameteriv()**

- 功能：

设置颜色查询表参数。

- C描述：

```
void glColorTableParameterfv( GLenum target,
                             GLenum pname,
                             const GLfloat *params)
void glColorTableParameteriv( GLenum target,
                             GLenum pname,
```

```
const GLint *params )
```

- 参数说明：

target 目标颜色表。它必须是**GL_COLOR_TABLE**、**GL_POST_CONVOLUTION_COLOR_TABLE**或**GL_POST_COLOR_MATRIX_COLOR_TABLE**。

pname 纹理颜色查询表参数的符号名称。它必须是**GL_COLOR_TABLE_SCALE**或**GL_COLOR_TABLE_BIAS**。

params 一个指针，指向一个用于存储参数值的数组。

- 说明：

函数**glColorTableParameter()**的作用是指定当一个颜色组分被载入颜色表时用于颜色组分的缩放因子和偏移项。参数*target*用于确定缩放因子和偏移项应用于哪个颜色表。它必须被设置成**GL_COLOR_TABLE**、**GL_POST_CONVOLUTION_COLOR_TABLE**或**GL_POST_COLOR_MATRIX_COLOR_TABLE**。

参数*pname*必须用**GL_COLOR_TABLE_SCALE**，来设置缩放因子。这时，参数*params*指向一个四元的数组，依次代表红、绿、蓝和alpha缩放因子。

同样，参数*pname*必须用**GL_COLOR_TABLE_BIAS**来设置偏移项。这时，参数*params*指向一个四元的数组，依次代表红、绿、蓝和alpha偏移项。

颜色表本身要通过调用函数**glColorTable()**来指定。

- 注意：

当调用函数**glGetString(GL_EXTENSIONS)**的返回值是**GL_ARB_imaging**时，函数**glColorTableParameter()**才被支持。

- 出错提示：

如果参数*target*或*pname*不是一个可接受值，则产生**GL_INVALID_ENUM**提示。

当函数**glColorTableParameter()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGetColorTableParameter()

- 请参阅：

glColorTable(), **glPixelTransfer()**

- **glConvolutionFilter1D**

- 名称：

glConvolutionFilter1D()

- 功能：

定义一个一维的卷积滤波器。

- C描述：

```
void glConvolutionFilter1D( GLenum target,
                           GLenum internalformat,
                           GLsizei width,
```

```

GLenum format,
GLenum type,
const GLvoid *image )

```

• 参数说明：

target 必须是**GL_CONVOLUTION_1D**。

internalformat

卷积滤波器内核的内部格式。其允许的取值是：**GL_ALPHA**、**GL_LUMINANCE**、**GL_LUMINANCE_ALPHA**、**GL_INTENSITY**、**GL_RGB**和**GL_BGRA**。

width 由参数*image*提供的像素数组的宽度。

format 参数*image*中像素数据的格式。它可取的值有：**GL_ALPHA**、**GL_ALPHA4**、**GL_ALPHA8**、**GL_ALPHA12**、**GL_ALPHA16**、**GL_LUMINANCE**、**GL_LUMINANCE4**、**GL_LUMINANCE8**、**GL_LUMINANCE12**、**GL_LUMINANCE16**、**GL_LUMINANCE_ALPHA**、**GL_LUMINANCE4_ALPHA4**、**GL_LUMINANCE6_ALPHA2**、**GL_LUMINANCE8_ALPHA8**、**GL_LUMINANCE12_ALPHA4**、**GL_LUMINANCE12_ALPHA12**、**GL_LUMINANCE16_ALPHA16**、**GL_INTENSITY**、**GL_INTENSITY4**、**GL_INTENSITY8**、**GL_INTENSITY12**、**GL_INTENSITY16**、**GL_R3_G3_B2**、**GL_RGB**、**GL_RGB4**、**GL_RGB5**、**GL_BGB8**、**GL_RGB10**、**GL_RGB12**、**GL_RGB16**、**GL_RGBA**、**GL_RGBA2**、**GL_RGBA4**、**GL_RGB5_A1**、**GL_RGBA8**、**GL_RGB10_A2**、**GL_RGBA12**和**GL_RGBA16**。

type 参数*image*中像素数据的类型。它允许取的值有：**GL_UNSIGNED_BYTE**、**GL_BYTE**、**GL_BITMAP**、**GL_UNSIGNED_SHORT**、**GL_SHORT**、**GL_UNSIGNED_INT**、**GL_INT**、**GL_FLOAT**、**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**。

image 一个指向像素数据的一维数组的指针。该数据用于产生卷积矩阵的内核。

• 说明：

函数**glConvolutionFilter1D()**的作用是由一个像素数组产生一个一维卷积滤波器的内核。

由参数*width*、*format*、*type*和*image*指定的像素数组是从内存中提取的。其处理过程就像是调用了函数**glDrawPixels()**一样，但当它最后完全扩展到ROBA时该过程将停止。

接下来每个像素的R、G、B和A组分将被四个1D的**GL_CONVOLUTION_FILTER_SCALE**参数进行缩放，并由四个1D的**GL_CONVOLUTION_FILTER_BIAS**参数进行偏移。

(缩放和偏移参数由函数glConvolutionParameter()用目标GL_CONVOLUTION_1D和名称GL_CONVOLUTION_FILTER_SCALE及GL_CONVOLUTION_FILTER_BIAS来设定。这些参数都是由四个值的矢量所构成的，这四个值依次应用于红、绿、蓝和alpha。) 在这个过程中，R、G、B和A值不被截断到范围[0, 1]内。

然后，每个像素被转换成由参数internalformat指定的内部格式。这种转换仅仅是由像素的组分值（R、G、B和A）简单地映射到内部格式所包含的值（红、绿、蓝、alpha、亮度及强度）。其映射关系如表5-6所示。

表 5-6

内部格式	红	绿	蓝	alpha	亮度	强度
GL_ALPHA				A		
GL_LUMINANCE				R		
GL_LUMINANCE_ALPHA			A	R		
GL_INTENSITY						R
GL_RGB	R	G	B			
GL_RGBA	R	G	B	A		

所得像素的红、绿、蓝、alpha、亮度和/或强度组分将被存储为浮点型而非整型格式。它们所形成的一维滤波器内核的图像将通过坐标*i*检索，*i*从0开始并从左到右递增。位于坐标*i*处的内核是指从0开始计数的第*i*个像素。

值得注意的是经过这样的操作所得的颜色组分同样要由它们相应的GL_POST_CONVOLUTION_c_SCALE参数缩放，并由参数GL_POST_CONVOLUTION_c_BIAS进行偏移（这里的c代表RED、GREEN、BLUE和ALPHA）。这两个参数由函数glPixelTransfer()设置。

- 注意：

当调用函数glGetString(GL_EXTENSIONS)的返回值是GL_ARB_imaging时，函数glConvolutionFilter1D()才被支持。

- 出错提示：

当参数target不是GL_CONVOLUTION_1D时产生GL_INVALID_ENUM提示。

当参数internalformat不是其可取值之一时产生GL_INVALID_ENUM提示。

当参数width小于0或大于其最大支持值时产生GL_INVALID_VALUE提示。这个最大支持值需要由函数glGetConvolutionParameter()通过使用目标GL_CONVOLUTION_1D和名称GL_MAX_CONVOLUTION_WIDTH来查询。

当参数format不是一个可接受值时产生GL_INVALID_ENUM提示。

当参数type不是一个可接受值时产生GL_INVALID_ENUM提示。

当函数glConvolutionFilter1D()在函数对glBegin()/glEnd()之间执行时产生GL_INVALID_OPERATION提示。

当参数type为GL_UNSIGNED_BYTE_3_3_2、GL_UNSIGNED_BYTE_2_3_3_REV、GL_UNSIGNED_SHORT_5_6_5或GL_UNSIGNED_SHORT_5_6_5_REV之一，但format却不是GL_RGB格式时产生GL_INVALID_OPERATION提示。

当参数*type*为**GL_UNSIGNED_SHORT_4_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**或**GL_UNSIGNED_INT_2_10_10_10_REV**之一，但*format*既不是**GL_RGBA**格式又不是**GL_BGRA**格式时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGetConvolutionParameter()

glGetConvolutionFilter()

- 请参阅：

glConvolutionFilter2D(), **glSeparableFilter2D()**, **glConvolutionParameter()**,
glPixelTransfer()

glConvolutionFilter2D

- 名称：

glConvolutionFilter2D()

- 功能：

定义一个二维的卷积滤波器。

- C描述：

```
void glConvolutionFilter2D( GLenum target,
                           GLenum internalformat,
                           GLsizei width,
                           GLsizei height,
                           GLenum format,
                           GLenum type,
                           const GLvoid *image )
```

- 参数说明：

target 必须是**GL_CONVOLUTION_2D**。

internalformat

卷积滤波器内核的内部格式。它可取的值有：**GL_ALPHA**、**GL_ALPHA4**、**GL_ALPHA8**、**GL_ALPHA12**、**GL_ALPHA16**、**GL_LUMINANCE**、**GL_LUMINANCE4**、**GL_LUMINANCE8**、**GL_LUMINANCE12**、**GL_LUMINANCE16**、**GL_LUMINANCE_ALPHA**、**GL_LUMINANCE4_ALPHA4**、**GL_LUMINANCE6_ALPHA2**、**GL_LUMINANCE8_ALPHA8**、**GL_LUMINANCE12_ALPHA4**、**GL_LUMINANCE12_ALPHA12**、**GL_LUMINANCE16_ALPHA16**、**GL_INTENSITY**、**GL_INTENSITY4**、**GL_INTENSITY8**、**GL_INTENSITY12**、**GL_INTENSITY16**、**GL_R3_G3_B2**、**GL_RGB**、**GL_RGB4**、**GL_RGB5**、**GL_RGB8**、**GL_RGB10**、**GL_RGB12**、**GL_RGB16**、**GL_RGBA**、**GL_RGBA2**、**GL_RGBA4**、**GL_RGB5_A1**、**GL_RGBA8**、**GL_RGB10_A2**、**GL_RGBA12**和**GL_RGBA16**。

<i>width</i>	由参数 <i>image</i> 提供的像素数组的宽度。
<i>height</i>	由参数 <i>image</i> 提供的像素数组的高度。
<i>format</i>	参数 <i>image</i> 中像素数据的格式。它允许取的值是： GL_RED 、 GL_GREEN 、 GL_BLUE 、 GL_ALPHA 、 GL_LUMINANCE 、 GL_LUMINANCE_ALPHA 、 GL_RGB 、 GL_BGR 、 GL_RGBA 和 GL_BGRA 。
<i>type</i>	参数 <i>image</i> 中像素数据的类型。它允许取的值有： GL_UNSIGNED_BYTE 、 GL_BYTE 、 GL_BITMAP 、 GL_UNSIGNED_SHORT 、 GL_SHORT 、 GL_UNSIGNED_INT 、 GL_INT 、 GL_FLOAT 、 GL_UNSIGNED_BYTE_3_3_2 、 GL_UNSIGNED_BYTE_2_3_3_REV 、 GL_UNSIGNED_SHORT_5_6_5 、 GL_UNSIGNED_SHORT_5_6_5_REV 、 GL_UNSIGNED_SHORT_4_4_4 、 GL_UNSIGNED_SHORT_4_4_4_REV 、 GL_UNSIGNED_SHORT_5_5_5_1 、 GL_UNSIGNED_SHORT_1_5_5_5_REV 、 GL_UNSIGNED_INT_8_8_8_8 、 GL_UNSIGNED_INT_8_8_8_8_REV 、 GL_UNSIGNED_INT_10_10_10_2 和 GL_UNSIGNED_INT_2_10_10_10_REV 。
<i>image</i>	一个指向存放像素数据的二维数组的指针。该数据用于产生卷积滤波器的内核。

• 说明：

函数glConvolutionFilter2D()的作用是由一个像素数组产生一个二维卷积滤波器的内核。

由参数*width*、*height*、*format*、*type*和*image*指定的像素数组是从内存中提取的。其生成过程就像是调用了函数glDrawPixels()一样，但当它最后完全扩展到RGBA时该过程将停止。

接下来每个像素的R、G、B和A组分将由四个2D的**GL_CONVOLUTION_FILTER_SCALE**参数进行缩放，并由四个2D的**GL_CONVOLUTION_FILTER_BIAS**参数进行偏移。（缩放和偏移参数由函数glConvolutionParameter()用目标**GL_CONVOLUTION_2D**和名称**GL_CONVOLYTION_FILTER_SCALE**及**GL_CONVOLUTION_FILTER_BIAS**来设定。这些参数都是由四个值所构成的矢量，这四个值依次应用于红、绿、蓝和alpha。）在这个过程中，R、G、B和A值不必被截断到范围[0, 1]内。

然后，每个像素将被转换成由参数*internalformat*指定的内部格式。这种转换仅仅是由像素的组分值（R、G、B和A）简单地映射到内部格式所包含的值（红、绿、蓝、alpha、亮度及强度）。其映射关系如表5-7所示。

表 5-7

内部格式	红	绿	蓝	alpha	亮度	强度
GL_ALPHA				A		
GL_LUMINANCE				R		
GL_LUMINANCE_ALPHA			A	R		
GL_INTENSITY						R
GL_RGB	R	G	B			
GL_RGBA	R	G	B	A		

所得像素的红、绿、蓝、alpha、亮度和/或强度组分将被存储为浮点型而非整型格式。它们所

形成的二维滤波器内核的图像将通过坐标*i*和*j*检索，*i*从0开始并从左到右递增；*j*从0开始并从低到高递增。位于坐标*i*, *j*的内核是指第*N*个像素，这里的*N*是*i+j × width*。

值得注意的是经过这样的操作所得的颜色组分同样要由它们相应的**GL_POST_CONVOLUTION_c_SCALE**参数缩放，并由参数**GL_POST_CONVOLUTION_c_BIAS**进行偏移（这里的c代表**RED**、**GREEN**、**BLUE**和**ALPHA**）。这两个参数由函数**glPixelTransfer()**设置。

- 注意：

当调用函数**glGetString(GL_EXTENSIONS)**的返回值是**GL_ARB_imaging**时，函数**glConvolutionFilter2D()**才被支持。

- 出错提示：

如果参数*target*不是**GL_CONVOLUTION_2D**，则产生**GL_INVALID_ENUM**提示。

当参数*internalformat*不是其可取值之一时产生**GL_INVALID_ENUM**提示。

当参数*width*小于0或大于其最大支持值时产生**GL_INVALID_VALUE**提示。这个最大支持值需要由函数**glGetConvolutionParameter()**通过使用目标**GL_CONVOLUTION_2D**和名称**GL_MAX_CONVOLUTION_WIDTH**来查询。

当参数*format*不是一个可接受值时产生**GL_INVALID_ENUM**提示。

当参数*type*不是一个可接受值时产生**GL_INVALID_ENUM**提示。

当函数**glConvolutionFilter2D()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_BYTE_3_3_**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**或**GL_UNSIGNED_SHORT_5_6_5_REV**之一，但*format*却不是**GL_RGB**格式时产生**GL_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_SHORT_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**或**GL_UNSIGNED_INT_2_10_10_10_REV**之一，但参数*format*既不是**GL_RGBA**格式又不是**GL_BGRA**格式时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGetConvolutionParameter()

glGetConvolutionFilter()

- 请参阅：

glConvolutionFilter1D(), **glSeparableFilter2D()**, **glConvolutionParameter()**,
glPixelTransfer()

* **glConvolutionParameter**

- 名称：

glConvolutionParameterf(), **glConvolutionParameteri()**, **glConvolutionParameterfv()**,
glConvolutionParameteriv()

- 功能：

设置卷积参数。

• C描述：

```
void glConvolutionParameterf( GLenum target,
                               GLenum pname,
                               GLfloat params )
void glConvolutionParameteri( GLenum target,
                               GLenum pname,
                               GLint params )
```

• 参数说明：

target 卷积参数目标。它必须是**GL_CONVOLUTION_1D**、**GL_CONVOLUTION_2D**或**GL_SEPARABLE_2D**之一。

pname 待设置的参数名。它只能是**GL_CONVOLUTION_BORDER_MODE**。

params 参数值。它必须是**GL_REDUCE**、**GL_CONSTANT_BORDER**或**GL_REPLICATE_BORDER**之一。

• C描述：

```
void glConvolutionParameterfv( GLenum target,
                               GLenum pname,
                               const GLfloat *params )
void glConvolutionParameteriv( GLenum target,
                               GLenum pname,
                               const GLint *params )
```

• 参数说明：

target 卷积参数目标。它必须是**GL_CONVOLUTION_1D**、**GL_CONVOLUTION_2D**或**GL_SEPARABLE_2D**之一。

pname 设置的参数名。它可以是**GL_CONVOLUTION_BORDER_MODE**、**GL_CONVOLUTION_BORDER_COLOR**、**GL_CONVOLUTION_FILTER_SCALE**和**GL_CONVOLUTION_FILTER_BIAS**。

params 参数值。当参数*pname*是**GL_CONVOLUTION_BORDER_MODE**时，它必须是**GL_REDUCE**、**GL_CONSTANT_BORDER**或**GL_REPLICATE_BORDER**之一；否则，它必须是一个包含四个值（分别用于红、绿、蓝和alpha）的矢量，用于缩放（当参数*pname*是**GL_CONVOLUTION_FILTER_SCALE**时），或是用于对一个卷积滤波器内核进行偏移（当参数*pname*是**GL_CONVOLUTION_FILTER_BIAS**时），或是用于设定边界颜色常数（当参数*pname*是**GL_CONVOLUTION_BORDER_COLOR**时）。

• 说明：

函数**glConvolutionParameter()**的作用是设置一个卷积参数。

参数*target*用以选择将被影响的卷积滤波器。其取值可以是**GL_CONVOLUTION_1D**、**GL_CONVOLUTION_2D**或**GL_SEPARABLE_2D**。分别代表1D、2D或可分离的2D滤波器。

参数*pname*用以选择将要改变的参数。**GL_CONVOLUTION_FILTER_SCALE**和**GL_CONVOLUTION_FILTER_BIAS**将对定义卷积滤波器的内核产生影响；有关细节请参阅函数glConvolutionFilter1D()、glConvolutionFilter2D和glSeparableFilter2D()。在这种情况下，参数*params*是一个包含四个值的数组。这四个值分别应用于红、绿、蓝和alpha值。初始情况下，**GL_CONVOLUTION_FILTER_SCALE**取(1,1,1,1)，**GL_CONVOLUTION_FILTER_BIAS**取(0,0,0,0)。

参数*pname*的一个值**GL_CONVOLUTION_BORDER_MODE**用于控制卷积边界模式。它可取的值有：

GL_REDUCE:

卷积操作所得的图像比源图像要小。如果滤波器的宽度是*Wf*、高度是*Hf*、源图像的宽度是*Ws*、高度是*Rs*，则卷积后图像的宽度将是*Ws-Wf+1*、高度是*Rs-Hf+1*。(如果这种减法导致了一个图像的宽度和(或)高度出现了零或负数的情况，这时仅会出现输出为空，并不会产生出错提示)。由卷积产生的图像的坐标在宽度方向从0到*Ws-Wf*，在高度方向从0到*Rs-Hf*。

GL_CONSTANT_BORDER:

通过卷积操作产生的图像与源图像等大，并且其生成过程就像是源图像用像素按**GL_CONSTANT_BORDER_COLOR**指定的颜色环绕而成。

GL_REPLICATE_BORDER

通过卷积操作产生的图像与源图像等大，并且其生成过程就像是源图像边界上的最外面的像素被复制过一样。

- 注意：

当调用函数glGetString(**GL_EXTENSIONS**)的返回值是**GL_ARB_imaging**时，函数glConvolutionParameter()才被支持。

当指定了无效的图像尺寸时，将会有错误发生。这里所说的尺寸是指卷积操作后被测试的尺寸，并不是源图像的尺寸。比如说函数glTexImage1D()所需要的尺寸是2的幂值。当**GL_REDUCE**边界模式有效时，源图像的尺寸必须是大于最后一个2的幂值，但同时又小于1D的滤波器内核的那个尺寸。

- 出错提示：

如果参数*target*不是一个可接受的值，则产生**GL_INVALID_ENUM**提示。

如果参数*pname*不是一个可接受的值，则产生**GL_INVALID_ENUM**提示。

当参数*pname*是**GL_CONVOLUTION_BORDER_MODE**，但参数*params*不是**GL_REDUCE**、**GL_CONSTANT_BORDER**或**GL_REPLICATE_BORDER**之一时产生**GL_INVALID_ENUM**提示。

当函数glConvolutionParameter()在函数对glBegin()/glEnd()之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGetColorConvolutionParameter()

- 请参阅：

glConvolutionFilter1D(), glConvolutionFilter2D(), glSeparableFilter2D(), glGetConvolutionParameter()

• **glCopyColorSubTable**

- 名称:

glCopyColorSubTable()

- 功能:

重新指定部分颜色表。

- C描述:

```
void glCopyColorSubTable( GLenum target,
                          GLsizei start,
                          GLint x,
                          GLint y,
                          GLsizei width )
```

- 参数说明:

target 必须是下面各值之一: **GL_COLOR_TABLE**、**GL_POST_CONVOLUTION_COLOR_TABLE**或**GL_POST_COLOR_MATRIX_COLOR_TABLE**。

start 重新替换的部分颜色表开始位置的索引。

x, *y* 被拷贝的像素行的左下角的窗口坐标。

width 重新替换的颜色表条目的数目。

- 说明:

函数**glCopyColorSubTable()**用于重新指定已经由函数**glColorTable()**定义的颜色表的一个连续部分。从帧缓冲区中拷贝的像素将替换已有颜色表的*start*到*start+x-1* (包含) 部分。当然这一区域也可能不包括原来指定范围之外的任何条目。指定一个宽度为0的子纹理是允许的, 但这样的指定没有任何作用。

- 注意:

当调用函数**glGetString(GL_EXTENSIONS)**的返回值是**GL_ARB_imaging**时, 函数**glCopyColorSubTable()**才被支持。

- 出错提示:

如果参数*target*不是一个预先指定的颜色表, 则产生**GL_INVALID_VALUE**提示。

当参数*target*不是一个可接受值时产生**GL_INVALID_VALUE**提示。

当*start+count>width*时产生**GL_INVALID_VALUE**提示。

当函数**glCopyColorSubTable()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示

- 有关数据的获取:

glGetColorTable()

glGetColorTableParameter()

- 请参阅:

glColorSubTable(), **glColorTableParameter()**, **glCopyColorTable()**, **glCopyColorSubTable()**, **glGetColorTable()**

***glCopyColorTable**

- 名称:

glCopyColorTable()

- 功能:

将像素复制到一个颜色表中。

- C 描述:

```
void glCopyColorTable( GLenum target,
                      GLsizei internalformat,
                      GLint x,
                      GLint y,
                      GLsizei width )
```

- 参数说明:

target 颜色表指标。它必须是下面各值之一: **GL_COLOR_TABLE**、**GL_POST_CONVERSION_COLOR_TABLE**或**GL_POST_COLOR_MATRIX_COLOR_TABLE**。

internalformat

纹理图像的内部存储格式。它必须是下列符号常量之一: **GL_ALPHA**、**GL_ALPHA4**、**GL_ALPHA8**、**GL_ALPHA12**、**GL_ALPHA16**、**GL_LUMINANCE**、**GL_LUMINANCE4**、**GL_LUMINANCE8**、**GL_LUMINANCE12**、**GL_LUMINANCE16**、**GL_LUMINANCE_ALPHA**、**GL_LUMINANCE4_ALPHA4**、**GL_LUMINANCE6_ALPHA2**、**GL_LUMINANCES8_ALPHA8**、**GL_LUMINANCE12_ALPHA4**、**GL_LUMINANCE12_ALPHA12**、**GL_LUMINANCE16_ALPHA16**、**GL_INTENSITY**、**GL_INTENSITY4**、**GL_INTENSITY8**、**GL_INTENSITY12**、**GL_INTENSITY16**、**GL_R3_G3_B2**、**GL_RGB**、**GL_RGB4**、**GL_RGB5**、**GL_RGB8**、**GL_RGB10**、**GL_RGB12**、**GL_RGB16**、**GL_RGBA**、**GL_RGBA2**、**GL_RGBA4**、**GL_RGB5_A1**、**GL_RGBA8**、**GL_RGB10_A2**、**GL_RGBA12**和**GL_RGBA16**。

x

将被转换成颜色表的像素矩形的左下角的x坐标。

y

将被转换成颜色表的像素矩形的左下角的y坐标。

width

像素矩形的宽度。

- 说明:

函数**glCopyColorTable()**将从当前的**GL_READ_BUFFER** (而不是像函数**glColorTable()**一样由主存储器) 中载入一个颜色表。

屏幕上的左下角在 (*x*, *y*)、宽为*width*、高为1的像素矩形被载入颜色表。如果该区间的任何一个像素在与GL环境相应的窗口之外，则由这些像素得到的值是未定义的。

这一矩形中像素的生成过程就象调用了函数**glReadPixels()**一样，并通过参数*internal-format*设置成**RGBA**值，但当它最后完全转换成**RGBA**时该过程将停止。

然后，为该颜色表定义的四个缩放参数和四个偏移参数将被用来缩放和偏移每个像素的R、


```
GLint x,
GLint y,
GLsizei width )
```

• 参数说明：

target 必须是**GL_CONVOLUTION_1D**。

internalformat

指定卷积滤波器内核的内部格式。它必须是下列符号常量之一： **GL_ALPHA**、
GL_ALPHA4、**GL_ALPHA8**、**GL_ALPHA12**、**GL_ALPHA16**、
GL_LUMINANCE、**GL_LUMINANCE4**、**GL_LUMINANCE8**、**GL_LUMINANCE12**、**GL_LUMINANCE16**、**GL_LUMINANCE_ALPHA**、**GL_LUMINANCE4_ALPHA4**、
GL_LUMINANCE6_ALPHA2、**GL_LUMINANCE8_ALPHA8**、**GL_LUMINANCE12_ALPHA4**、**GL_LUMINANCE12_ALPHA12**、**GL_LUMINANCE16_ALPHA16**、**GL_INTENSITY**、**GL_INTENSITY4**、**GL_INTENSITY8**、**GL_INTENSITY12**、**GL_INTENSITY16**、**GL_RGB**、**GL_R3_G3_B2**、**GL_RGB4**、**GL_RGB5**、**GL_RGB8**、**GL_RGB10**、**GL_RGB12**、
GL_RGB16、**GL_RGBA**、**GL_RGBA2**、**GL_RGBA4**、**GL_RGB5_A1**、**GL_RGBA8**、**GL_RGB10_A2**、**GL_RGBA12**或**GL_RGBA16**。

x, *y* 指定被拷贝的像素数组左下角的窗口坐标。

width 要拷贝像素数组的宽度。

• 说明：

函数**glCopyConvolutionFilter1D()**的作用是用当前**GL_READ_BUFFER**中（而不是从主内存中，请参阅**glConvolutionFilter1D()**）的像素定义一个一维卷积滤波器内核。

屏幕上的左下角在 (*x*, *y*)，宽度为*width*、高度为1的像素矩形定义了该卷积滤波器。如果这一区域内的所有像素都超出了GL环境所对应的窗口范围，则由这些像素得到的值将是未定义。

这个矩形范围内的像素的生成过程跟调用函数**glReadPixels()**生成格式为*format*的RGBA颜色是一样的，但最后转换之前该过程将停止。接下来每个像素的R、G、B和A组分将被四个1D的**GL_CONVOLUTION_FILTER_SCALE**参数进行缩放，并由四个1D的**GL_CONVOLUTION_FILTER_BIAS**参数进行偏移。（缩放和偏移参数由函数**glConvolution Parameter()**用目标为**GL_CONVOLUTION_1D**和名称为**GL_CONVOLYTION_FILTER_SCALE**及**GL_CONVOLUTION_FILTER_BIAS**的参数来设定。这些参数都是由四个值所构成的矢量，这四个值依次为红、绿、蓝和alpha。）在这个过程中，R、G、B和A值不必被截断到范围[0, 1]内。

接下来，每个像素将被转换成由参数*internalformat*指定的内部格式。这种转换仅仅是由像素的组分值（R、G、B和A）简单地映射到内部格式所包含的值（红、绿、蓝、alpha、亮度及强度）。其映射关系如表5-9所示：

所得像素的红、绿、蓝、alpha、亮度和/或强度组分将被存储为浮点型而非整型格式。

像素按以下方法排序：较小的*x*屏幕坐标对应于小的滤波器图像坐标*i*。

值得注意的是经过这样的卷积操作所得的颜色组分同样要由它们相应的**GL_POST_**

CONVOLUTION_c_SCALE参数进行缩放，并由参数**GL_POST_CONVOLUTION_c_BIAS**进行偏移（这里的c代表红、绿、蓝和**ALPHA**）。这两个参数由函数**glPixelTransfer()**设置。

表 5-9

内部格式	红	绿	蓝	alpha	亮度	强度
GL_ALPHA				A		
GL_LUMINANCE				R		
GL_LUMINANCE_ALPHA			A	R		
GL_INTENSITY						R
GL_RGB	R	G	B			
GL_RGBA	R	G	B	A		

- 注意：

当调用函数**glGetString(GL_EXTENSIONS)**的返回值是**GL_ARB_imaging**时，函数**glCopyConvolutionFilter1D()**才被支持。

- 出错提示：

当参数*target*不是**GL_CONVOLUTION_1D**时产生**GL_INVALID_ENUM**提示。

当参数*internalformat*不是其可取值之一时产生**GL_INVALID_ENUM**提示。

当参数*width*小于0或大于其最大支持值时产生**GL_INVALID_VALUE**提示。这个最大支持值需要由函数**glGetConvolutionParameter()**通过使用目标为**GL_CONVOLUTION_1D**和名称为**GL_MAX_CONVOLUTION_WIDTH**的参数来查询。

当函数**glCopyConvolutionFilter1D()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGetConvolutionParameter()

glGetConvolutionFilter()

- 请参阅：

glConvolutionFilter1D(), **glConvolutionParameter()**, **glPixelTransfer()**

- **glCopyConvolutionFilter2D**

- 名称：

glCopyConvolutionFilter2D()

- 功能：

将像素拷贝进一个二维卷积滤波器中。

- C描述：

```
void glCopyConvolutionFilter2D( GLenum target,
                               GLenum internalformat,
                               GLint x,
                               GLint y,
                               GLsizei width,
```

`GLsizei height)`

- 参数说明：

target 必须是**GL_TEXTURE_2D**。

internalformat

指定卷积滤波器内核的内部格式。它必须是下列符号常量之一：**GL_ALPHA**、**GL_ALPHA4**、**GL_ALPHA8**、**GL_ALPHA12**、**GL_ALPHA16**、**GL_LUMINANCE**、**GL_LUMINANCE4**、**GL_LUMINANCE8**、**GL_LUMINANCE12**、**GL_LUMINANCE16**、**GL_LUMINANCE_ALPHA**、**GL_LUMINANCE4_ALPHA4**、**GL_LUMINANCE6_ALPHA2**、**GL_LUMINANCE8_ALPHA8**、**GL_LUMINANCE12_ALPHA4**、**GL_LUMINANCE12_ALPHA12**、**GL_LUMINANCE16_ALPHA16**、**GL_INTENSITY**、**GL_INTENSITY4**、**GL_INTENSITY8**、**GL_INTENSITY12**、**GL_INTENSITY16**、**GL_RGB**、**GL_R3_G3_B2**、**GL_RGB4**、**GL_RGB5**、**GL_RGB8**、**GL_RGB10**、**GL_RGB12**、**GL_RGB16**、**GL_RGBA**、**GL_RGBA2**、**GL_RGBA4**、**GL_RGB5_A1**、**GL_RGBA8**、**GL_RGB10_A2**、**GL_RGBA12**或**GL_RGBA16**。

x, y 指定被拷贝的像素数组左下角的窗口坐标。

width 指定要拷贝的像素数组的宽度。

height 指定要拷贝的像素数组的高度。

- 说明：

函数**glCopyConvolutionFilter2D()**的作用是用当前**GL_READ_BUFFER**中（而不是从主内存中，请参阅**glConvolutionFilter2D()**）的像素定义一个二维卷积滤波器内核。

屏幕上的左下角在(*x, y*)，宽度为*width*、高度为*height*的像素矩形定义了该卷积滤波器。如果这一区域内的所有像素都超出了GL环境所对应的窗口范围，则由这些像素得到的值将未定义。

这个矩形范围内的像素的生成过程就象调用函数**glDrawPixels()**生成格式为*format*的RGBA颜色一样，但最后转换之前该过程将停止。接下来每个像素的R、G、B和A组分将被四个2D的**GL_CONVOLUTION_FILTER_SCALE**参数进行缩放，并由四个2D的**GL_CONVOLUTION_FILTER_BIAS**参数进行偏移。（缩放和偏移参数由函数**glConvolutionParameter()**用目标为**GL_CONVOLUTION_2D**和名称为**GL_CONVOLYTION_FILTER_SCALE**及**GL_CONVOLUTION_FILTER_BIAS**的参数来设定。这些参数都是由四个值所构成的矢量，这四个值依次为红、绿、蓝和alpha。）在这个过程中，R、G、B和A值不必被截断到范围[0, 1]内。

接下来，每个像素将被转换成由参数*internalformat*指定的内部格式。这种转换仅仅是将像素的组分值(R、G、B和A)简单地映射到内部格式所包含的值(红、绿、蓝、alpha、亮度及强度)。其映射关系如表5-10所示：

所得像素的红、绿、蓝、alpha、亮度和/或强度组分将被存储为浮点型而非整型格式。

像素按以下方法排序：较小的*x*屏幕坐标对应于小的滤波器图像坐标*i*，较小的*y*屏幕坐标对应于小的滤波器图像坐标*j*。

值得注意的是经过这样的卷积操作所得的颜色组分同样要由它们相应的**GL_POST_**

CONVOLUTION_c_SCALE参数进行缩放，并由参数**GL_POST_CONVOLUTION_c_BIAS**进行偏移（这里的c代表红、绿、蓝和ALPHA）。这两个参数由函数glPixelTransfer()设置。

表 5-10

内部格式	红	绿	蓝	alpha	亮度	强度
GL_ALPHA				A		
GL_LUMINANCE				R		
GL_LUMINANCE_ALPHA			A	R		
GL_INTENSITY						R
GL_RGB	R	G	B			
GL_RGBA	R	G	B	A		

- 注意：

当调用函数glGetString(GL_EXTENSIONS)的返回值是**GL_ARB_imaging**时，函数glCopyConvolutionFilter2D()才被支持。

- 出错提示：

当参数*target*不是**GL_CONVOLUTION_2D**时产生**GL_INVALID_ENUM**提示。

当参数*internalformat*不是其可取值之一时产生**GL_INVALID_ENUM**提示。

当参数*width*小于0或大于其最大支持值时产生**GL_INVALID_VALUE**提示。这个最大支持值需要由函数glGetConvolutionParameter()通过使用目标为**GL_CONVOLUTION_2D**和名称为**GL_MAX_CONVOLUTION_WIDTH**的参数来查询。

当参数*height*小于0或大于其最大支持值时产生**GL_INVALID_VALUE**提示。这个最大支持值需要由函数glGetConvolutionParameter()通过使用目标为**GL_CONVOLUTION_2D**和名称为**GL_MAX_CONVOLUTION_HEIGHT**的参数来查询。

当函数glConvolutionFilter2D()在函数对glBegin()/glEnd()之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGetConvolutionParameter()

glGetConvolutionFilter()

- 请参阅：

glConvolutionFilter2D(), glGetConvolutionParameter(), glPixelTransfer()

• glCopyPixels

- 名称：

glCopyPixels()

- 功能：

在帧缓冲区中拷贝像素。

- C描述：

```
void glCopyPixels( GLint x,  
                   GLint y,
```

```
GLsizei width,
GLsizei height,
GLenum type)
```

• 参数说明：

x, y 指定被拷贝的像素矩形区域的左下角的窗口坐标。

width, height

指定被拷贝的像素矩形区域的尺寸。它们都必须是非负值。

type 指定将拷贝的是颜色值、深度值还是模板值。它可取下面的符号常量：**GL_COLOR**、**GL_DEPTH**和**GL_STENCIL**。

• 说明：

函数**glCopyPixels()**从指定的帧缓冲区中与当前光栅位置有关的区域内拷贝一个像素的屏幕校准矩形。只有当整个像素源区域在窗口显示区内时，这样操作才完全有效。如果要从窗口之外拷贝图形，或拷贝窗口显示区外的部分，则需要相关的硬件支持，在此未作定义。参数*x*和*y*指定了被拷贝的像素矩形区域的左下角的窗口坐标。参数*width*和*height*指定了被拷贝的像素矩形区域的尺寸。参数*width*和*height*都必须是非负值。

这里有几个参数用于控制像素的拷贝过程。它们由下面的三个命令设置：**glPixelTransfer()**、**glPixelMap()**和**glPixelZoom()**。本节主要介绍由这3个命令设置的大部分参数对函数**glCopyPixels()**的影响。

函数**glCopyPixels()**从每个像素的左下角，即(*x+i, y+j*)处开始拷贝其值。这里 $0 \leq i < width$, $0 \leq j < height$ 。这个像素被称作第*j*行的第*i*个像素。不同的行像素被由低到高地拷贝，同行内像素被由左到右拷贝。

参数*type*用于指定将拷贝的是颜色值、深度值还是模板值。每种数据类型的转换方式如下：

GL_COLOR:

将从当前指定的读源缓冲区中读出索引或RGBA颜色（请参阅**glReadBuffer()**）。当GL是在颜色索引模式下时，通过在二进制点的右边加上若干位而将它转化为一种确定的点格式。然后这些索引数值左移**GL_INDEX_SHIFT**位后加到**GL_INDEX_OFFSET**上。如果**GL_INDEX_SHIFT**的值为负，则右移。不管是左移还是右移，结果中未指定的其他位都补0。当**GL_MAP_COLOR**是**TRUE**时，索引值将由查询表**GL_PIXEL_MAP_I_TO_I**表所提供的值替换。不管索引值是否被查询表替换，其整数部分都将加上 $2^b - 1$ ，其中**b**为颜色索引缓冲区中二进制位的个数。

当GL是在RGBA模式下时，读取的每个像素的红、绿、蓝和alpha值将被转化为一个不指定精度的内部整型浮点格式。数据被线性地映射为整型浮点格式：最大的值映射为1.0，0映射为0.0。然后把所得的浮点颜色值乘以**GL_c_SCALE**，再加上**GL_c_BIAS**即得最后值。此处**c**代表相关颜色的RED、GREEN、BLUE和ALPHA组分。最后这些值的取值范围是[0, 1]。这时如果**GL_MAP_COLOR**为**TRUE**，各颜色组分将乘以查询表中**GL_PIXEL_MAP_c_TO_c**的值，然后替换表中的值。这里**c**是R、G、B或A之一。

当系统支持**GL_ARB_imaging**扩展时，颜色值将另外地进行颜色表查询、颜色矩阵转换和卷积滤波操作。

接下来GL将把当前的光栅位置的z坐标和纹理坐标赋给每一个像素，从而把所得的索引或RGBA颜色值转化为片断，并指定相应的窗口坐标 (x_r+i, y_r+j) ，这里 (x_r, y_r) 是当前光栅位置。该像素是第j行的第i个像素。这些像素片断与由光栅化点、线和多边形产生的片断具有一样的性质。在该片断被写入帧缓冲区之前将要进行纹理映射、雾化及所有的片断操作。

GL_DEPTH:

深度值将从深度缓冲区中读取，然后被直接转化为一个不指定精度的内部浮点格式。然后把所得的深度值乘以**GL_DEPTH_SCALE**，再加上**GL_DEPTH_BIAS**即得最后的深度值。其值在[0, 1]之间。

接下来GL将把当前的光栅位置的z坐标和纹理坐标赋给每一个像素，从而把所得的RGBA颜色值转化为片断，并指定相应的窗口坐标 (x_r+i, y_r+j) ，这里 (x_r, y_r) 是当前光栅位置。该像素是第j行的第i个像素。这些像素片断与由光栅化点、线和多边形产生的片断具有一样的性质。在该片断被写入帧缓冲区之前将要进行纹理映射、雾化及所有片断操作。

GL_STENCIL:

模板索引将从模板缓冲区中读取。这些读取值通过在二进制点的右边加上若干位而将它转化为一种内部的确定的点格式。然后这些索引数值将左移**GL_INDEX_SHIFT**位后加到**GL_INDEX_OFFSET**上。如果**GL_INDEX_SHIFT**的值为负，则右移。不管是左移还是右移，结果中未指定的其他位都补0。当**GL_MAP_STENCIL**是TRUE时，索引值将由查询表**GL_PIXEL_MAP_S_TO_S**表所提供的值替换。不管索引值是否被查询表替换，其整数部分都将加上 2^b-1 ，其中b为模板缓冲区中二进制位的个数。所得的模板索引值被按如下方式写入模板缓冲区：由第j行的第i个存储单元中读取的内容被写入 (x_r+i, y_r+j) 存储单元，这里 (x_r, y_r) 是当前光栅位置。对这些写操作能够产生影响的测试只有像素所有权测试和裁剪测试和模板写屏蔽。

目前为止介绍的光栅化操作中我们都假设像素的缩放因子(zoom factor)为1。如果已用函数**glPixelZoom()**改变了x和y的缩放因子，下式将把像素转化为片断。设 (x_r, y_r) 是当前光栅位置，给定的像素位于像素矩形的第j行第i列。则产生的片断位于一像素矩形的中央。此像素矩形的两个角为：

$$(X_r+zoom_x \cdot i, y_r+zoom_y \cdot j) \text{ 和 } (X_r+zoom_x \cdot (i+1), y_r+zoom_y \cdot (j+1))$$

这里 $zoom_x$ 是**GL_ZOOM_X**的值， $zoom_y$ 是**GL_ZOOM_Y**的值。

- 示例：

如果要将窗口左下角的彩色像素拷贝到当前光栅位置，请使用下面命令：

```
glCopyPixels(0, 0, 1, 1, GL_COLOR);
```

- 注意：

由函数**glPixelStore()**指定的模式对函数**glCopyPixels()**的操作没有影响。

- 出错提示：

当参数type不是一个可接受值，则产生**GL_INVALID_ENUM**提示。

当参数width或height为负数时产生**GL_INVALID_VALUE**提示。

当参数type是**GL_DEPTH**，但没有深度缓冲区时，产生**GL_INVALID_OPERATION**提示。

当参数type是**GL_STENCIL**，但没有模板缓冲区时，产生**GL_INVALID_OPERATION**提示。

当函数**glCopyPixels()**在函数对**glBegin()/glEnd()**之间执行时产生**GL_INVALID**

_OPERATION提示。

- 有关数据的获取：

glGet(GL_CURRENT_RASTER_POSITION)
glGet(GL_CURRENT_RASTER_POSITION_VALID)

- 请参阅：

glColorTable(), **glConvolutionFilter1D()**, **glConvolutionFilter2D()**, **glDepthFunc()**,
glDrawBuffer(), **glDrawPixels()**, **glMatrixMode()**, **glPixelMap()**, **glPixelTransfer()**, **glPixelZoom()**,
glRasterPos(), **glReadBuffer()**, **glReadPixels()**, **glSeparableFilter2D()**, **glStencilFunc()**

***glCopyTexImage1D**

- 名称：

glCopyTexImage1D()

- 功能：

将像素拷贝进一个1D纹理图像。

- C描述：

```
void glCopyTexImage1D( GLenum target,
                      GLint level,
                      GLenum internalformat,
                      GLint x,
                      GLint y,
                      GLsizei width,
                      GLint border )
```

- 参数说明：

target 指定目标纹理。它必须是**GL_TEXTURE_1D**。

level 指定多重纹理的纹理级别号。0级是图像的基本级。*n*级是指第*n*个mipmap的简化图像。

internalformat

指定纹理的内部格式。它必须是下列符号常量之一：**GL_ALPHA**、**GL_ALPHA4**、**GL_ALPHA8**、**GL_ALPHA12**、**GL_ALPHA16**、**GL_LUMINANCE**、**GL_LUMINANCE4**、**GL_LUMINANCE8**、**GL_LUMINANCE12**、**GL_LUMINANCE16**、**GL_LUMINANCE_ALPHA**、**GL_LUMINANCE4_ALPHA4**、**GL_LUMINANCE6_ALPHA2**、**GL_LUMINANCE8_ALPHA8**、**GL_LUMINANCE12_ALPHA4**、**GL_LUMINANCE12_ALPHA12**、**GL_LUMINANCE16_ALPHA16**、**GL_INTENSITY**、**GL_INTENSITY4**、**GL_INTENSITY8**、**GL_INTENSITY12**、**GL_INTENSITY16**、**GL_RGB**、**GL_R3_G3_B2**、**GL_RGB4**、**GL_RGB5**、**GL_RGB8**、**GL_RGB10**、**GL_RGB12**、**GL_RGB16**、**GL_RGBA**、**GL_RGBA2**、**GL_RGBA4**、**GL_RGB5_A1**、

GL_RGBA8、GL_RGB10_A2、GL_RGBA12或GL_RGBA16。

x, y 指定被拷贝的像素行左下角的窗口坐标。

width 指定纹理图像的宽度。它必须是0或 $2^n + 2 \times border$, 此处*n*是一个正整数。纹理图像的高度是1。

border 指定纹理边界的宽度。它必须是0或1。

- 说明:

函数glCopyTexImage1D()的作用是用当前**GL_READ_BUFFER**中的像素定义一个一维纹理图像。

屏幕上的左下角在(*x, y*)、宽为*width+2 × border*的像素行定义了mipmap图层的纹理数组，该mipmap图层的级别由参数*level*指定。参数*internalformat*指定纹理数组的内部格式。

该行中像素的形成过程与调用函数glCopyPixels()形成像素的过程很相似，但该过程将在最后转换前结束。此处，所有像素的组分值都被截断到范围[0, 1]，并被转换成纹理的内部格式，从而存储到纹理数组中。

像素按以下方法排序：较小的x屏幕坐标对应较小的纹理坐标。

如果当前**GL_READ_BUFFER**的指定行中的任何像素超出了当前绘制内容所对应的窗口范围，则由这些像素得到的值将未定义。

- 注意:

函数glCopyTexImage1D()只有在GL1.1以上的版本才被支持。

纹理操作在颜色索引模式下无效。

参数*internalformat*的值不能取1、2、3或4。

一个宽度为0的图像是一个NULL纹理。

当系统支持**GL_ARB_imaging**扩展时，从帧缓冲区中拷贝的RGBA组分可以被绘图流程处理。详细情况请参阅glTexImage1D()

- 出错提示:

当参数*target*不是一个可取值时，产生**GL_INVALID_ENUM**提示。

当参数*level*小于0时，产生**GL_INVALID_VALUE**提示。

当参数*level*大于 $\log_{\sqrt{2}} max$ 时，产生**GL_INVALID_VALUE**提示。此处*max*是**GL_MAX_TEXTURE_SIZE**的返回值。

当参数*internalformat*不是一个可取值时，产生**GL_INVALID_VALUE**提示。

当参数*width*小于0或大于 $2 + GL_MAX_TEXTURE_SIZE$ ，或不能表示成 $2^n + 2^* (border)$ (此处*n*是一个正整数)时，产生**GL_INVALID_VALUE**提示。

当参数*border*既不是0也不是1时，产生**GL_INVALID_VALUE**提示。

当函数glCopyTexImage1D()在函数对glBegin()/glEnd()之间执行时，产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取:

glGetTexImage()

glIsEnabled(GL_TEXTURE_1D)

- 请参阅：

`glCopyPixels()`, `glCopyTexImage2D()`, `glCopyTexSubImage1D()`, `glCopyTexSubImage2D()`,
`glPixelStore()`, `glPixelTransfer()`, `glTexEnv()`, `glTexGen()`, `glTexImage1D()`, `glTexImage2D()`,
`glTexSubImage1D()`, `glTexSubImage2D()`, `glTexParameter()`

`: glCopyTexImage2D`

- 名称：

`glCopyTexImage2D()`

- 功能：

将像素拷贝进一个2D纹理图像。

- C描述：

```
void glCopyTexImage2D( GLenum target,
                      GLint level,
                      GLenum internalformat,
                      GLint x,
                      GLint y,
                      GLsizei width,
                      GLsizei height,
                      GLint border )
```

- 参数说明：

target 指定目标纹理。它必须是`GL_TEXTURE_2D`。

level 指定多重纹理的纹理级别号。0级是图像的基本级。*n*级是指第*n*个mipmap的简化图像。

internalformat

指定纹理的内部格式。它必须是下列符号常量之一：`GL_ALPHA`、`GL_ALPHA4`、`GL_ALPHA8`、`GL_ALPHA12`、`GL_ALPHA16`、`GL_LUMINANCE`、`GL_LUMINANCE4`、`GL_LUMINANCE8`、`GL_LUMINANCE12`、`GL_LUMINANCE16`、`GL_LUMINANCE_ALPHA`、`GL_LUMINANCE4_ALPHA4`、`GL_LUMINANCE6_ALPHA2`、`GL_LUMINANCE8_ALPHA8`、`GL_LUMINANCE12_ALPHA4`、`GL_LUMINANCE12_ALPHA12`、`GL_LUMINANCE16_ALPHA16`、`GL_INTENSITY`、`GL_INTENSITY4`、`GL_INTENSITY8`、`GL_INTENSITY12`、`GL_INTENSITY16`、`GL_RGB`、`GL_R3_G3_B2`、`GL_RGB4`、`GL_RGB5`、`GL_RGB8`、`GL_RGB10`、`GL_RGB12`、`GL_RGB16`、`GL_RGBA`、`GL_RGBA2`、`GL_RGBA4`、`GL_RGB5_AI`、`GL_RGBA8`、`GL_RGB10_A2`、`GL_RGBA12`或`GL_RGBA16`。

x, *y* 指定被拷贝的像素矩形区域的左下角的窗口坐标。

width 指定纹理图像的宽度。它必须是0或 $2^n + 2 \times border$ ，此处*n*是一个正整数。

height 指定纹理图像的高度。它必须是0或 $2^m + 2 \times border$ ，此处*m*是一个正整数。

border 指定纹理边界的宽度。它必须是0或1。

- 说明：

函数glCopyTexImage2D()的作用是用当前的**GL_READ_BUFFER**中的像素定义一个二维纹理图像。

屏幕上的左下角在(*x*, *y*)，宽为*width*+2×*border*、高为*height*+2×*border*的像素矩形定义了mipmap图层的纹理数组，该mipmap纹理级别号由参数*level*指定。参数*internalformat*指定纹理数组的内部格式。

该矩形中像素的形成过程与调用函数glCopyPixels()形成像素的过程很相似，但该过程将在最后转换前结束。此处，所有像素的组分值都被截断到范围[0, 1]，并被转换成纹理的内部格式，从而存储到纹理数组中。

像素按以下方法排序：较小的*x*和*y*屏幕坐标对应较小的*s*和*t*纹理坐标。

如果当前**GL_READ_BUFFER**的指定矩形中的任何像素超出了与当前绘制内容相应的窗口范围，则由这些像素得到的值将未定义。

- 注意：

函数glCopyTexImage2D()只有在GL 1.1以上的版本中才被支持。

纹理操作在颜色索引模式下无效。

参数*internalformat*的值不能取1、2、3或4。

一个宽度或高度为0的图像是一个NULL纹理。

当系统支持**GL_ARB_imaging** 扩展时，从帧缓冲区中拷贝的RGBA组分可以被绘图流程处理。详细情况请参阅glTexImage2D()

- 出错提示：

当参数*target*不是**GL_TEXTURE_2D**时，产生**GL_INVALID_ENUM**提示。

当参数*level*小于0时，产生**GL_INVALID_VALUE**提示。

当参数*level*大于 $\log_2 max$ 时，产生**GL_INVALID_VALUE**提示。此处*max*是**GL_MAX_TEXTURE_SIZE**的返回值。

当参数*width*或*height*小于0或大于 $2 + \text{GL_MAX_TEXTURE_SIZE}$ ，或不能表示成 $2^k + 2 \times border$ （此处*k*是一个正整数）时，产生**GL_INVALID_VALUE**提示。

当参数*border*既不是0也不是1时，产生**GL_INVALID_VALUE**提示。

当参数*internalformat*不是一个可取值时，产生**GL_INVALID_VALUE**提示。

当函数glCopyTexImage2D()在函数对glBegin()/glEnd ()之间执行时，产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGetTexImage()

glIsEnabled(GL_TEXTURE_2D)

- 请参阅：

glCopyPixels(), **glCopyTexImage1D()**, **glCopyTexSubImage1D()**, **glCopyTexSubImage2D()**,
glPixelStore(), **glPixelTransfer()**, **glTexEnv()**, **glTexGen()**, **glTexImage1D()**, **glTexImage2D()**,

glTexSubImage1D(), glTexSubImage2D(), glTexParameter()

• **glCopyTexSubImage1D**

- 名称：

glCopyTexSubImage1D()

- 功能：

拷贝一个一维纹理子图。

- C描述：

```
void glCopyTexSubImage1D( GLenum target,
                          GLint level,
                          GLint xoffset,
                          GLint x,
                          GLint y,
                          GLsizei width )
```

- 参数说明：

target 指定目标纹理。它必须是**GL_TEXTURE_1D**。

level 指定多重纹理的纹理级别号。0级是图像的基本级。*n*级是指第*n*个mipmap的简化图像。

xoffset 指定纹理数组中沿x方向的一个纹理偏移量。

x, y 指定被拷贝的像素行左下角的窗口坐标。

width 指定纹理子图的宽度。

- 说明：

函数**glCopyTexSubImage1D()**的作用是用从当前**GL_READ_BUFFER**中（而不是像函数**glTexSubImage1D()**一样，从主内存中）提取的像素替换部分一维纹理图像。

屏幕上的左下角在(*x, y*)、宽度为*width*的像素行将替换已存在的纹理数组中的一部分，该部分位于*x*方向的*xoffset*和*xoffset+width-1*（包含端点）之间。指定的目标纹理数组中可以不包含任何以前指定的源纹理数组范围以外区域中的像素。

该行中像素的形成过程与调用函数**glCopyPixels()**形成像素的过程相似，但该过程将在最后转换前结束。这时，所有像素的组分值将被截断到范围[0, 1]，并被转换成纹理的内部格式，从而存储到纹理数组中。

指定一个宽度为0的子纹理是允许的，但这种指定将不产生任何效果。如果当前**GL_READ_BUFFER**中指定行中的所有像素都超出了当前绘制内容所对应的读取窗口范围，则由这些像素得到的值将未定义。

已指定的纹理数组的参数*internalformat*、*width*和*border*或是位于指定的子区间之外的纹理值将不会被影响。

- 注意：

函数**glCopyTexSubImage1D()**只有在GL 1.1以上的版本中才有效。

纹理操作在颜色索引模式下不起作用。

函数`glPixelStore()`和`glPixelTransfer()`对纹理图像的影响同它们对函数`glDrawPixels()`的影响相似。

当系统支持`GL_ARB_imaging`时，从帧缓冲区中拷贝的RGBA组分可以被绘图流程处理。详细情况请参阅`glTexImage1D()`。

- 出错提示：

当参数`target`不是`GL_TEXTURE_1D`时产生`GL_INVALID_ENUM`提示。

如果纹理数组没被以前的函数`glTexImage1D()`或`glCopyTexImage1D()`定义过，则产生`GL_INVALID_OPERATION`提示。

当参数`level`小于0时产生`GL_INVALID_VALUE`提示。

当参数`level`大于`log2max`时产生`GL_INVALID_VALUE`提示。此处`max`是`GL_MAX_TEXTURE_SIZE`的返回值。

当 $y < -b$ 或 $width < -b$ 时产生`GL_INVALID_VALUE`提示。此处`b`是纹理数组的边界宽度。

当 $xoffset < -b$ ，或 $(xoffset + width) > (w - b)$ 时产生`GL_INVALID_VALUE`提示。此处`w`是值`GL_TEXTURE_WIDTH`，`b`是被修改的纹理图像的宽度值`GL_TEXTURE_BORDER`。请注意此处的`w`是边界宽度的两倍。

- 有关数据的获取：

`glGetTexImage()`

`glIsEnabled(GL_TEXTURE_1D)`

- 请参阅：

`glCopyPixels()`, `glCopyTexImage1D()`, `glCopyTexImage2D()`, `glCopyTexSubImage2D()`,
`glCopyTexSubImage3D()`, `glPixelStore()`, `glPixelTransfer()`, `glReadBuffer()`, `glTexEnv()`,
`glTexGen()`, `glTexImage1D()`, `glTexImage2D()`, `glTexImage3D()`, `glTexParameter()`,
`glTexSubImage1D()`, `glTexSubImage2D()`, `glTexSubImage3D()`

glCopyTexSubImage2D

- 名称：

`glCopyTexSubImage2D()`

- 功能：

拷贝一个二维纹理子图。

- C描述：

```
void glCopyTexSubImage2D( GLenum target,
                           GLint level,
                           GLint xoffset,
                           GLint yoffset,
                           GLint x,
                           GLint y,
                           GLsizei width,
                           GLsizei height )
```

- 参数说明：

target 指定目标纹理。它必须是**GL_TEXTURE_2D**。

level 指定多重纹理的纹理级别号。0级是图像的基本级。*n*级是指第*n*个mipmap的简化图像。

xoffset 指定纹理数组中沿x方向的一个纹理偏移量。

yoffset 指定纹理数组中沿y方向的一个纹理偏移量。

x, y 指定被拷贝的像素行左下角的窗口坐标。

width 指定纹理子图的宽度。

height 指定纹理子图的高度。

- 说明：

函数glCopyTexSubImage2D()的作用是用从当前**GL_READ_BUFFER**中（而不像函数glTexSubImage2D()一样，从主内存中）提取的像素替换部分二维纹理图像。

屏幕上的左下角在(*x, y*)、宽度为*width*、高度为*height*的像素矩形将替换已存在的纹理数组中的一部分，该部分位于x方向的*xoffset*和*xoffset+width-1*（包含端点）及y方向的*yoffset*和*yoffset+height-1*（包含端点）之间。其mipmap纹理级别号由参数*level*指定。

该像素矩形的形成过程与调用函数glCopyPixels()形成像素矩形的过程相似，但该过程将在最后转换前结束。这时，所有像素的组分值将被截断到范围[0, 1]，并被转换成纹理的内部格式，从而存储到纹理数组中。

纹理数组的目标矩形中可以不包含任何以前指定的源纹理数组范围以外区域中的像素，但指定一个宽度为0或高度为0的子纹理是允许的，只是这种指定将不产生任何效果。

如果当前**GL_READ_BUFFER**中指定矩形中的所有像素都超出了当前绘图环境所对应的读取窗口范围，则由这些像素得到的值将未定义。

已指定的纹理数组的参数*internalformat*、*width*、*height*和*border*或是位于指定的子区间之外的纹理值将不会被影响。

- 注意：

函数glCopyTexSubImage2D()只有在GL 1.1以上的版本中才有效。

纹理操作在颜色索引模式下不起作用。

函数glPixelStore()和glPixelTransfer()对纹理图像的影响同它们对函数glDrawPixels()的影响相似。

当系统支持**GL_ARB_imaging**时，从帧缓冲区中拷贝的RGBA组分可以被绘图流程处理。详细情况请参阅glTexImage1D()。

- 出错提示：

当参数*target*不是**GL_TEXTURE_2D**时产生**GL_INVALID_ENUM**提示。

当纹理数组从来没被以前的函数glTexImage2D()或glCopyTexImage2D()定义过时产生**GL_INVALID_OPERATION**提示。

当参数*level*小子0时产生**GL_INVALID_VALUE**提示。

当参数*level*大于 $\log_2 max$ 时产生**GL_INVALID_VALUE**提示。此处*max*是**GL_MAX_**

TEXTURE_SIZE的返回值。

当 $x < -b$ 或 $y < -b$ 时产生**GL_INVALID_VALUE**提示。此处 b 是纹理数组的边界宽度。

当 $xoffset < -b$, $(xoffset+width) > (w-b)$, $yoffset < -b$, 或 $(yoffset+height) > (h-b)$ 时产生**GL_INVALID_VALUE**提示。此处 w 是值**GL_TEXTURE_WIDTH**, h 是值**GL_TEXTURE_HEIGHT**, b 是被修改的纹理图像的宽度值**GL_TEXTURE_BORDER**。请注意此处的 w 和 h 是边界宽度的两倍。

当函数**glCopyTexSubImage2D()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取:

glGetTexImage()

glIsEnabled(GL_TEXTURE_2D)

- 请参阅:

glCopyPixels(), **glCopyTexImage1D()**, **glCopyTexImage2D()**, **glCopyTexSubImage1D()**, **glCopyTexSubImage3D()**, **glPixelStore()**, **glPixelTransfer()**, **glReadBuffer()**, **glTexEnv()**, **glTexGen()**, **glTexImage1D()**, **glTexImage2D()**, **glTexImage3D()**, **glTexParameter()**, **glTexSubImage1D()**, **glTexSubImage2D()**, **glTexSubImage3D()**

■ **glCopySubImage3D**

- 名称:

glCopySubImage3D()

- 功能:

拷贝一个三维纹理子图。

- C描述:

```
void glCopySubImage3D( GLenum target,
                      GLint level,
                      GLint xoffset,
                      GLint yoffset,
                      GLint zoffset,
                      GLint x,
                      GLint y,
                      GLsizei width,
                      GLsizei height )
```

- 参数说明:

target 指定目标纹理。它必须是**GL_TEXTURE_3D**。

level 指定多重纹理的纹理级别号。0级是图像的基本级。*n*级是指第*n*个mipmap的简化图像。

xoffset 指定纹理数组中沿x方向的一个纹理偏移量。

yoffset 指定纹理数组中沿y方向的一个纹理偏移量。

zoffset 指定纹理数组中沿z方向的一个纹理偏移量。

x, y 指定被拷贝的像素矩形的左下角的窗口坐标。

width 指定纹理子图的宽度。

height 指定纹理子图的高度。

- 说明：

函数glCopySubImage3D()的作用是用从当前**GL_READ_BUFFER**中（而不是像函数glTexSubImage3D()一样，从主内存中）提取的像素替换部分三维纹理图像。

屏幕上的左下角在(*x, y*)、宽度为*width*、高度为*height*的像素矩形将替换已存在的纹理数组中的一部分，该部分位于*x*方向的*xoffset*和*xoffset+width-1*（包含端点）之间，*y*方向的*yoffset*和*yoffset+height-1*（包含端点），*z*方向的*zoffset*和*zoffset+depth-1*（包含端点）之间。其mipmap纹理级别号由参数*level*指定。

该矩形中像素的形成过程与调用函数glCopyPixels()形成像素的过程相似，但该过程将在最后转换前结束。这时，所有像素的组分值将被截断到范围[0, 1]，并被转换成纹理的内部格式，从而存储到纹理数组中。

纹理数组的目标矩形中可以不包含任何以前指定的纹理数组范围以外区域中的像素，但指定一个宽度为0或高度为0的子纹理是允许的，只是这种指定将不产生任何效果。

如果当前**GL_READ_BUFFER**中指定矩形中的所有像素都超出了当前绘制环境所对应的读取窗口范围，则由这些像素得到的值未定义。

已指定的纹理数组的参数*internalformat*、*width*、*height*、*depth*和*border*或是位于指定的子区间之外的纹理值将不会被影响。

- 注意：

函数glCopyTexSubImage3D()只有在GL 1.2以上的版本中才有效。

纹理操作在颜色索引模式下不起作用。

函数glPixelStore()和glPixelTransfer()对纹理图像的影响同它们对函数glDrawPixels()的影响相似。

当系统支持**GL_ARB_imaging**时，从帧缓冲区中拷贝的RGBA组分可以被绘图流程处理，就好像它们是一个二维的纹理。详细情况请参阅glTexImage2D()。

- 出错提示：

当参数*target*不是**GL_TEXTURE_3D**时产生**GL_INVALID_ENUM**提示。

当纹理数组从来没被以前的函数glTexImage3D()或glCopyTexImage3D()定义过时产生**GL_INVALID_OPERATION**提示。

当参数*level*小于0时产生**GL_INVALID_VALUE**提示。

当参数*level*大于*log₂max*时产生**GL_INVALID_VALUE**提示。此处*max*是**GL_MAX_3D_TEXTURE_SIZE**的返回值。

当*x < -b*或*y < -b*时产生**GL_INVALID_VALUE**提示。此处*b*是纹理数组的边界宽度。

当*xoffset < -b*, (*xoffset + width*) > (*w-b*) ; *yoffset < -b*, (*yoffset + height*) > (*h-b*) ; *zoffset < -b*, 或 (*zoffset + depth*) > (*d-b*) 时产生**GL_INVALID_VALUE**提示。此处*w*是值**GL_TEXTURE_WIDTH**, *h*是值**GL_TEXTURE_HEIGHT**, *d*是值**GL_TEXTURE_DEPTH**, *b*是被修改

的纹理图像的宽度值**GL_TEXTURE_BORDER**。请注意此处的 w 、 h 和 d 是边界宽度的两倍。

当函数**glCopyTexSubImage3D()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGetTexImage()

glIsEnabled(GL_TEXTURE_3D)

- 请参阅：

glCopyPixels(), **glCopyTexImage1D()**, **glCopyTexImage2D()**, **glCopyTexSubImage1D()**,
glCopyTexSubImage2D(), **glPixelStore()**, **glPixelTransfer()**, **glReadBuffer()**, **glTexEnv()**,
glTexGen(), **glTexImage1D()**, **glTexImage2D()**, **glTexImage3D()**, **glTexParameter()**,
glTexSubImage1D(), **glTexSubImage2D()**, **glTexSubImage3D()**

◆ **glCullFace**

- 名称：

glCullFace()

- 功能：

指定被拣选掉的面是正面还是反面。

- C 描述：

`void glCullFace(GLenum mode)`

- 参数说明：

mode 指定被拣选掉的面是正面的还是反面。它可取下面的符号常量：**GL_FRONT**、
GL_BACK或**GL_FRONT_AND_BACK**。其缺省值是**GL_BACK**。

- 说明：

函数**glCullFace()**的作用是当启动拣选操作时指定被拣选掉的面是正面的还是反面（由参数*mode*指定）。缺省情况下，拣选操作是关闭的。可以用参数**glEnable(GL_CULL_FACE)**和**glDisable(GL_CULL_FACE)**来启动和关闭拣选操作。这里所提到的拣选面可以是三角形、四边形、多边形和矩形面。

函数**glFrontFace()**用来指定是顺时针方向还是逆时针方向的面来作为正面。请参阅**glFrontFace()**。

- 注意：

如果参数*mode*是**GL_FRONT_AND_BACK**，则没有面被绘出。但其他图元，如点或线将被绘出。

- 出错提示：

当参数*mode*不是一个可接受的值时产生**GL_INVALID_ENUM**提示。

当函数**glCullFace()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

`glIsEnabled(GL_CULL_FACE)`
`glGet(GL_CULL_FACE_MODE)`
• 请参阅：
`glEnable()`, `glFrontFace()`

*** glDeleteLists**

- 名称：
glDeleteLists()
 - 功能：
删除一组连续的显示列表。

```
void glDeleteLists( GLuint list,  
                      GLsizei range );
```

- #### • 参数说明：

list 指定要删除的第一个显示列表的整型名称。

range 指定要删除的显示列表的数目。

- ### • 说明:

函数glDeleteLists()的作用是删除一组连续的显示列表。参数list是要删除的第一个显示列表的名称，参数range是要删除的显示列表的数目。所有在范围 $list \leq d \leq list + range - 1$ 内的显示列表都将被删除。

所有分配给指定显示列表的存储单元将被释放，这时允许重新使用这些存储单元。在上述范围内的名称如果没有一个相关的显示列表，该操作将被忽略。如果参数*range*是0，将不会发生任何操作。

- ### * 出错提示:

当参数*range*是负数时产生**GL_INVALID_VALUE**提示。

当函数glDeleteLists()在函数对glBegin()/glEnd()之间执行时产生**GL_INVALID_OPERATION**提示。

- 请参阅:

glCallList(), **glCallLists()**, **glGenLists()**, **glIsList()**, **glNewList()**

~glDeleteTextures

- 名称：
glDeleteTextures()
 - 功能：
删除指定的纹理。

• C 描述:

```
void glDeleteTextures( GLsizei n  
                      const GLuint *textures );
```

- 参数说明：

n 指定要删除的纹理数目

textures 指定一个要删除纹理的数组。

- 说明：

函数glDeleteTextures()的作用是删除由数组*textures*的元素所指定*n*个纹理。当一个纹理被删除后，它将不再含有任何内容或维数，它的名称将被释放以供重新使用（比如供函数glGenTextures()使用）。如果将要删除一个当前连接的纹理，该连接将被恢复为0（默认纹理）。

函数glDeleteTextures()自动忽略0和其他任何不与已存在的纹理相对应的名称。

- 注意：

函数glDeleteTextures()仅在GL 1.1以上的版本中才有效。

- 出错提示：

当参数*n*是负数时产生**GL_INVALID_VALUE**提示。

当函数glDeleteTextures()在函数对glBegin()/glEnd()之间执行时产生**GL_INVALID_OPERATION**提示。

- 请参阅：

glAreTexturesResident(), glBindTexture(), glCopyTexImage1D(), glCopyTexImage2D(),
glGenTextures(), glGet(), glGetTexParameter(), glPrioritizeTextures(), glTexImage1D(),
glTexImage2D(), glTexParameter()

glDepthFunc

- 名称：

glDepthFunc()

- 功能：

指定用于深度缓冲比较的值。

- C 描述：

```
void glDepthFunc( GLenum func )
```

- 参数说明：

func 指定深度比较函数。它可取的符号常量有：**GL_NEVER**、**GL_LESS**、**GL_EQUAL**、**GL_LEQUAL**、**GL_GREATER**、**GL_NOTEQUAL**、**GL_GEQUAL** 和**GL_ALWAYS**。其缺省值是**GL_LESS**。

- 说明：

函数glDepthFunc()指定一个比较函数，用来对每个输入像素的深度值与深度缓冲区中提供的深度值进行比较。仅当启动深度测试后，比较操作才能进行。（请参阅glEnable(**GL_DEPTH_TEST**)和glDisable(**GL_DEPTH_TEST**)）。

参数*func*用来指定绘制像素时的状态。比较函数含义如下：

GL_NEVER 不通过。

GL_LESS 如果输入的深度值小于参考值，则通过。

GL_EQUAL 如果输入的深度值等于参考值，则通过。

GL_LESS 如果输入的深度值小于参考值，则通过。

GL_GREATER 如果输入的深度值大于参考值，则通过。

GL_NOTEQUAL 如果输入的深度值不等于参考值，则通过。

GL_EQUAL 如果输入的深度值大于或等于参考值，则通过。

GL_ALWAYS 总是通过。

参数*func*的默认值是**GL_LESS**。缺省情况下，深度测试关闭。即使已存在深度缓冲区且深度屏蔽是非零的，如果深度测试已关闭，则深度缓冲区也不能被更新。

- 出错提示：

当参数*func*不是一个可取值时产生**GL_INVALID_ENUM**提示。

当函数**glDepthFunc()**在函数对**glBegin()/glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGet(GL_DEPTH_FUNC)

glIsEnabled(GL_DEPTH_TEST)

- 请参阅：

glDepthRange(), glEnable(), glPolygonOffset()

glDepthMask

- 名称：

glDepthMask()

- 功能：

启动和关闭深度缓冲区的写入操作。

- C描述：

void glDepthMask(GLboolean flag)

- 参数说明：

flag 指定是否可以对深度缓冲区进行写入操作。当参数*flag*是**GL_FALSE**时，深度缓冲区的写入操作关闭；否则，该操作启动。缺省情况下，允许对深度缓冲区进行写操作。

- 说明：

函数**glDepthMask()**的作用是指定是否可以对深度缓冲区进行写入操作。当参数*flag*是**GL_FALSE**时，深度缓冲区的写入操作关闭；否则，该操作启动。缺省情况下，允许对深度缓冲区进行写操作。

- 出错提示：

当函数**glDepthMask()**在函数对**glBegin()/glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGet(GL_DEPTH_WRITEMASK)

- 请参阅：

glColorMask(), glDepthFunc(), glDepthRange(), glIndexMask(), glStencilMask()

• **glDepthRange**

- 名称：

glDepthRange()

- 功能：

指定一种从归一化深度坐标到窗口深度坐标的映射方法。

- C描述：

```
void glDepthRange( GLclampd zNear,
                  GLclampd zFar )
```

- 参数说明：

zNear 指定从最近的剪切平面到窗口坐标的映射。其缺省值时0。

zFar 指定从最远的剪切平面到窗口坐标的映射。其缺省值时1。

- 说明：

深度坐标在剪切和除以 w 后，其范围将变为[-1, 1]。其中-1和1分别与最近和最远的剪切平面相对应。函数**glDepthRange()**将指定该范围内从归一化深度坐标到窗口深度坐标的一种线性映射。不管具体实现的实际深度缓冲区如何，窗口坐标中的深度值将被看作其范围是[0, 1]（像颜色组分一样）。因此，由函数**glDepthRange()**得到的值将首先被截断到这一范围内。

这里，0和1被分别映射成最近和最远的剪切平面。在这种映射下，深度缓冲区的范围可以被完全利用。

- 注意：

这里并不要求*zNear*小于*zFar*。反之，*zNear*=1、*zFar*=0同样可以。

- 出错提示：

当函数**glDepthRange()**在函数对**glBegin()/glEnd()**之间执行时，产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGet(GL_DEPTH_RANGE)

- 请参阅：

glDepthFunc(), glPolygonOffset(), glViewport()

• **glDrawArrays**

- 名称：

glDrawArrays()

- 功能：

从数组数据中绘制图元。

- C描述：

```
void glDrawArrays( GLenum mode,
                  GLint first,
```

`GLsizei count)`

- 参数说明：

mode 指定将被绘制的图元种类。它可取下面的符号常量：**GL_POINTS**、**GL_LINE_STRIP**、**GL_LINE_LOOP**、**GL_LINES**、**GL_TRIANGLE_STRIP**、**GL_TRIANGLE_FAN**、**GL_TRIANGLES**、**GL_QUAD_STRIP**、**GL_QUADS**和**GL_POLYGON**。

first 指定有效的数组中的开始索引。

count 指定绘制的索引数目。

- 说明：

函数`glDrawArrays()`只要调用很少几个子程序就可以指定多个几何图元。用户不用再使用一个GL过程来生成各个单独的定点、法线、纹理坐标、边界标志或颜色；而可以通过预先指定几个单独的顶点数组（这些数组中可以是顶点、法向量和颜色），用一个函数`glDrawArrays()`将这些顶点构成一个图元序列。

当函数`glDrawArrays()`被调用时，它将从每个有效的数组中取出*count*个连续的图元，从而构造一个几何图元序列。参数*first*指定其开始位置，参数*mode*指定构成图元的种类及这些图元的构成方式。当**GL_VERTEX_ARRAY**无效时，将不会产生任何几何图元。

由函数`glDrawArrays()`修正的顶点属性在该函数返回后将产生一个不确定的值。例如，当**GL_COLOR_ARRAY**有效时，函数`glDrawArrays()`执行后当前颜色的值将是未定义的。没有被修正的属性将仍保持原来的值。

- 注意：

函数`glDrawArrays()`只有GL 1.1以上的版本中才有效。

函数`glDrawArrays()`可以被包括在显示列表中。当它被包括在一个显示列表中时，其所需的数组数据（由数组指针确定并启动）也进入了显示列表。由于数组指针和启动操作都是客户端状态，所以它们的值将对显示列表的建立产生影响，而并不影响显示列表的执行。

- 出错提示：

当参数*mode*不是一个可接受的值时产生**GL_INVALID_ENUM**提示。

当参数*count*为负数时产生**GL_INVALID_VALUE**提示。

当函数`glDrawArrays()`在函数对`glBegin()`/`glEnd()`之间执行时产生**GL_INVALID_OPERATION**提示。

- 请参阅：

`glArrayElement()`, `glColorPointer()`, `glDrawElements()`, `glDrawRangeElements()`,
`glEdgeFlagPointer()`, `glGetPointerv()`, `glIndexPointer()`, `glInterleavedArrays()`,
`glNormalPointer()`, `glTexCoordPointer()`, `glVertexPointer()`

• `glDrawBuffer`

- 名称：

`glDrawBuffer()`

- 功能：

指定需要写入的颜色缓冲区。

- C 描述：

```
void glDrawBuffer( GLenum mode )
```

- 参数说明：

mode 指定需要写入的四种颜色缓冲区。它可取下面的符号常量：**GL_NONE**、**GL_FRONT_LEFT**、**GL_FRONT_RIGHT**、**GL_BACK_LEFT**、**GL_BACK_RIGHT**、**GL_FRONT**、**GL_BACK**、**GL_LEFT**、**GL_RIGHT**、**GL_FRONT_AND_BACK**和**GL_AUX*i***，此处*i*是在范围0到“**GL_AUX_BUFFERS**”-1之间（**GL_AUX_BUFFERS**不是它的上限，用户可通过函数glGet()来查询有效的辅助缓冲区的数目）。单缓冲区环境的默认值是**GL_FRONT**，双缓冲区环境的默认值是**GL_BACK**。

- 说明：

当颜色值被写入帧缓冲区时，这些颜色值将被写入由函数glDrawBuffer()指定的颜色缓冲区中。其说明如下：

GL_NONE	不写入颜色缓冲区。
GL_FRONT_LEFT	仅写入左前颜色缓冲区。
GL_FRONT_RIGHT	仅写入右前颜色缓冲区。
GL_BACK_LEFT	仅写入左后颜色缓冲区。
GL_BACK_RIGHT	仅写入右后颜色缓冲区。
GL_FRONT	仅写入左前和右前颜色缓冲区。如果没有右前颜色缓冲区，则仅向左前颜色缓冲区写入。
GL_BACK	仅写入左后和右后颜色缓冲区。如果没有右后颜色缓冲区，则仅向左后颜色缓冲区写入。
GL_LEFT	仅写入左前和左后颜色缓冲区。如果没有左后颜色缓冲区，则仅向左前颜色缓冲区写入。
GL_RIGHT	仅写入右前和右后颜色缓冲区。如果没有右后颜色缓冲区，则仅向右前颜色缓冲区写入。
GL_FRONT_AND_BACK	写入所有的颜色缓冲区（左前、右前、左后和右后）。如果没有后面的颜色缓冲区，则仅向左前和右前颜色缓冲区写入。如果没有右面的颜色缓冲区，则仅向左前和左后颜色缓冲区写入。如果没有右面和后面的颜色缓冲区，则仅向左前颜色缓冲区写入。
GL_AUX<i>i</i>	仅写入辅助颜色缓冲区 <i>i</i> 。

当选定向多个颜色缓冲区写入时，融合操作和逻辑操作将对每个独立的颜色缓冲区单独进行，并在每个缓冲区产生不同的结果。

单原子环境中仅含有*left*缓冲区，而立体环境中则含有*left*和*right*缓冲区。类似地，单缓冲区环境仅包含*front*缓冲区，而双缓冲区环境则包含*front*和*back*缓冲区。环境是由GL在初始化时选择的。

- 注意：

下式总是成立的：

$$\text{GL_AUX}_i = \text{GL_AUX0} + i$$

- 出错提示：

当参数*mode*不是一个可接受的值时产生**GL_INVALID_ENUM**提示。

当参数*mode*指定的缓冲区都不存在时产生**GL_INVALID_OPERATION**提示。

当函数**glDrawBuffer()**在函数对**glBegin()/glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGet(GL_DRAW_BUFFER)

glGet(GL_AUX_BUFFERS)

- 请参阅：

glBlendFunc(), glColorMask(), glIndexMask(), glLogicOp(), glReadBuffer()

◆ **glDrawElements**

- 名称：

glDrawElements()

- 功能：

由数组数据绘制图元。

- C描述：

```
void glDrawElements( GLenum mode,
                     GLsizei count,
                     GLenum type,
                     const GLvoid *indices )
```

- 参数说明：

mode 指定要绘制的图元的种类。它可取下面的符号常量：**GL_POINTS**、**GL_LINE_STRIP**、**GL_LINE_LOOP**、**GL_LINES**、**GL_TRIANGLE_STRIP**、**GL_TRIANGLE_FAN**、**GL_TRIANGLES**、**GL_QUAD_STRIP**、**GL_QUADS**和**GL_POLYGON**。

count 指定要绘制的图元数目。

type 指定*indices*中数值的类型。它必须是下列各值之一：**GL_UNSIGNED_BYTE**、**GL_UNSIGNED_SHORT**或**GL_UNSIGNED_INT**。

indices 指定一个指向索引的存储单元的指针。

- 说明：

函数**glDrawArrays()**只要调用很少几个子程序就可以指定多个几何图元。用户不用再通过调用一个GL函数来生成各个单独的定点、法线、纹理坐标、边界标志或颜色；而可以通过预先指定几个单独的数组（这些数组中可以是顶点、法向量和颜色），用一个函数**glDrawElements()**来将它们构成一个图元序列。

当函数glDrawArrays()被调用时，它将从每个有效的数组中由参数*indices*指定的位置开始取出*count*个连续的图元从而生成一个几何图元序列。参数*mode*指定了构成图元的种类及由数组元素构成这些图元的方式。当多个数组有效时，每个数组都可使用。当**GL_VERTEX_ARRAY**无效时，将不会产生任何几何图元。

由函数glDrawArrays()修正的顶点属性在该函数返回后将产生一个不确定的值。例如，当**GL_COLOR_ARRAY**有效时，函数glDrawElements()执行后当前颜色的值将是未定义的。没有被修正的属性将仍保持原来的值。

- 注意：

函数glDrawElements()只有在GL 1.1以上的版本中才有效。

函数glDrawElements()可以被包括在显示列表中。当它被包括在一个显示列表中时，其所需的数据（由数组指针确定并启动）也进入了显示列表。由于数组指针和启动操作都是客户端状态，所以它们的值将对显示列表的建立产生影响，但并不影响显示列表的执行。

- 出错提示：

当参数*mode*不是一个可接受的值时产生**GL_INVALID_ENUM**提示。

当参数*count*是负数时产生**GL_INVALID_VALUE**提示。

当函数glDrawElements()在函数对glBegin()/glEnd()之间执行时产生**GL_INVALID_OPERATION**提示。

- 请参阅：

glArrayElement(), **glColorPointer()**, **glDrawArrays()**, **glDrawRangeElements()**,
glEdgeFlagPointer(), **glGetPointerv()**, **glIndexPointer()**, **glInterleavedArrays()**,
glNormalPointer(), **glTexCoordPointer()**, **glVertexPointer()**

glDrawPixels

- 名称：

glDrawPixels()

- 功能：

向帧缓冲区写入一个像素块。

- C描述：

```
void glDrawPixels( GLsizei width,
                   GLsizei height,
                   GLenum format,
                   GLenum type,
                   const GLvoid *pixels )
```

- 参数说明：

width, height

指定要写入帧缓冲区中的像素矩形的尺寸。

format 指定像素数据的格式。它可以是以下符号常量之一：**GL_COLOR_INDEX**、**GL_STENCIL_INDEX**、**GL_DEPTH_COMPONENT**、**GL_RGB**、**GL_BGR**、

GL_RGBA、**GL_BGRA**、**GL_RED**、**GL_GREEN**、**GL_BLUE**、**GL_ALPHA**、**GL_LUMINANCE**和**GL_LUMINANCE_ALPHA**。

type 为*pixels*指定数据类型。它可以是以下符号常量之一：**GL_UNSIGNED_BYTE**、**GL_BYTE**、**GL_BITMAP**、**GL_UNSIGNED_SHORT**、**GL_SHORT**、**GL_UNSIGNED_INT**、**GL_INT**、**GL_FLOAT**、**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**。

pixels 指定一个指向像素数据的指针。

- 说明：

在当前光栅位置有效的前提下,函数glDrawPixels()的作用是从内存中读取像素数据并把它写入帧缓冲区的当前光栅位置。使用函数glRasterPos()设置当前光栅位置。使用函数glGet(**GL_CURRENT_RASTER_POSITION_VALID**)来决定指定的光栅位置的有效性, 使用函数glGet(**GL_CURRENT_RASTER_POSITION**)查询光栅的位置。

有几个参数用来定义内存中的像素数据的编码方式, 并在像素数据被写入帧缓冲区之前控制对像素数据的处理。这些参数由以下四个命令设置: glPixelStore(), glPixelTransfer(), glPixelMap()和glPixelZoom()。本节主要介绍函数glDrawPixels()的用法, 没有介绍到的参数的设定请参考这四个命令的介绍。

从*pixels*中读取的数据可以是一串带符号或无符号字节、带符号或无符号短整数、带符号或无符号整数、或单精度浮点数。其具体类型由*type*决定。当*type*是**GL_UNSIGNED_BYTE**、**GL_BYTE**、**GL_UNSIGNED_SHORT**、**GL_SHORT**、**GL_UNSIGNED_INT**、**GL_INT**或**GL_FLOAT**之一时, 这些值将根据*format*指定的格式的不同而作为颜色或深度的一个组分或索引。当*type*是**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_4_4_4**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_10_10_10_2**之一时, 每个无符号的数将被认为代表单一像素的所有组分, 组分的排列顺序由参数*format*确定。当*type*是**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4_REV**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_2_10_10_10_REV**之一时, 根据*format*指定的格式的不同, 每个无符号的数将被以相反的顺序解释为某一像素颜色的所有组分。索引总是被单独对待。颜色组分则按单值、双值、三值或四值的组来看待, 具体的排列顺序由参数*format*确定。单个索引和颜色组分组都可确定像素。当*type*是**GL_BITMAP**时, 数据只接受无符号字节型数据, 且*format*只能是**GL_COLOR_INDEX**或**GL_STENCIL_INDEX**。每个无符号字节型数据都被认为是八个二进制位的像素。其中二进制位的顺序由参数**GL_UNPACK_LSB_FIRST**确定(请参阅glPixelStore())。

从存储单元`pixels`开始，从内存中读取 $width \times height$ 个像素。缺省情况下，这些像素总是从内存中的相邻存储单元中读取，除非读取完所有 $width$ 方向的像素后，指针才指向下一个四字节边界。四字节行的排列由函数 `glPixelStore(GL_UNPACK_ALIGNMENT)` 指定。它可以被设置成单字节、双字节、四字节和八字节。其他像素存取参数决定不同的像素读取指针。在开始读取第一个像素前和所有 $width$ 宽度的像素读完后，可以通过改变像素存取参数来改变读取方式。有关细节请参考函数 `glPixelStore()`。

从内存中读取的 $width \times height$ 个像素的方法是相同的，具体方法由函数 `glPixelTransfer()` 和 `glPixelMap()` 指定的几个参数值来确定。情况像素将被写入哪个目标缓冲区等细节在不同的格式下是不相同的，具体情况由参数 `format` 确定。参数 `format` 可采用以下 13 个符号常量之一：

GL_COLOR_INDEX:

每个像素是一个单值，一个颜色索引。不管内存中数据的类型如何，通过在二进制点的右边加上若干位而将它转化为一种固定点格式。浮点数被转化为一个真正的固定点值。带符号和不带符号的整型数据的小数部分被设为 0。位图数据被转化为 0 或 1。

然后这些固定点索引值将左移 `GL_INDEX_SHIFT` 个二进制位并加到 `GL_INDEX_OFFSET` 上。如果 `GL_INDEX_SHIFT` 的值为负，则右移。结果中的其他不确定的二进制位补 0。

当 GL 处于 RGBA 模式时，所得的索引值通过 `GL_PIXEL_MAP_I_TO_R` 表、`GL_PIXEL_MAP_I_TO_G` 表、`GL_PIXEL_MAP_I_TO_B` 表和 `GL_PIXEL_MAP_I_TO_A` 表转化为一个 RGBA 像素。当 GL 处于颜色索引模式且 `GL_MAP_COLOR` 为 TRUE 时，索引值被查询表 `GL_PIXEL_MAP_I_TO_I` 的值替代。不管索引值是否被查询表替换，其整数部分都将和 $2^b - 1$ 进行逻辑与运算，其中 b 为颜色索引缓冲区中二进制位的数目。

接下来 GL 将把当前光栅位置的 z 坐标和纹理坐标赋给每一个像素，从而把所得的索引值或颜色值转化为片断。并用下式计算第 n 个片断的 x 和 y 窗口坐标：

$$x_n = x_r + n \bmod width$$

$$y_n = y_r + n / width$$

式中 (x, y) 是当前光栅的位置。这些像素片断被当作和通过光栅化点、线和多边形而得到的片断一样处理。在片断写入帧缓冲区以前将对其进行纹理映射、雾化等片断操作。

GL_STENCIL_INDEX:

每个像素是一个单值，一个模板索引值。不管内存中数据的类型如何，通过在二进制点的右边加上若干位而将它转化为一种固定点格式。浮点数被转化为一个真正的固定点值。带符号和不带符号的整型数据的小数部分被设为 0。位图数据被转化为 0 或 1。

然后这些固定点索引数值将左移 `GL_INDEX_SHIFT` 位并加到 `GL_INDEX_OFFSET` 上。

_OFFSET上。如果**GL_INDEX_SHIFT**的值为负，则右移。结果中的其他不确定的二进制位补0。

当**GL_MAP_STENCIL**为TRUE时，索引值被查询表**GL_PIXEL_MAP_S_TO_S**的值替换。不管索引值是否被查询表替换，其整数部分都将和 $2^b - 1$ 进行逻辑与运算，其中b为模板缓冲区中二进制位的数目。

接下来通过下式计算写入模板缓冲区存储单元中的第n个模板索引的x和y窗口坐标：

$$x_n = x_r + n \bmod width$$

$$y_n = y_r + n / width$$

式中(x_r, y_r)是当前光栅的位置。只有像素所有权测试、裁剪测试和模板写屏蔽会对这些写操作产生影响。

GL_DEPTH_COMPONENT:

每个像素是一个单深度的组分值。浮点数被直接转化为一个不确定精度的内部浮点格式。带符号整型数据被线性地映射为内部浮点格式：所代表的最大正整数映射为1.0，所代表的最小负整数映射为-1.0。不带符号的整型数据则简单地映射：最大的整数映射为1.0，0映射为0.0。然后把所得的深度值乘以**GL_DEPTH_SCALE**，再加到**GL_DEPTH_BIAS**上。然后所得的结果将被截断到范围[0, 1]内。

接下来GL将把当前光栅位置的颜色值或颜色索引值和纹理坐标赋给每一个像素，从而把所得的深度组分值转化为片断。并用下式计算第n个片断的x和y窗口坐标：

$$x_n = x_r + n \bmod width$$

$$y_n = y_r + n / width$$

式中(x_r, y_r)是当前光栅的位置。这些像素片断被当作和通过光栅化点、线和多边形而得到的片断一样处理。在片断写入帧缓冲区以前将对其进行纹理映射、雾化等片断操作。

GL_RGBA

GL_BGRA:

每个像素是一个四组分构成的组：对于**GL_RGBA**模式，依次为红、绿、蓝和alpha值；对于**GL_BGRA**模式，则依次为蓝、绿、红和alpha值。浮点数被直接转化为一个不确定精度的内部浮点格式。带符号整型数据被线性地映射为内部浮点格式：所代表的最大正整数映射为1.0，所代表的最小负整数映射为-1.0（请注意这里0并非被精确地映射为0.0）。不带符号的整型数据则被简单地映射：最大的整数映射为1.0，0映射为0.0。然后把所得的颜色值乘以**GL_c_SCALE**，再加到**GL_c_BIAS**上。此处c代表各个颜色组分**RED**、**GREEN**、**BLUE**和**ALPHA**。所得的值将被截断到范围[0, 1]内。

这时如果**GL_MAP_COLOR**为TRUE，各颜色组分将按查询表中**GL_PIXEL**

_MAP_c_TO_c的尺寸进行缩放，然后被它所指定的查询表中的值所替换。这里c分别是**R**、**G**、**B**和**A**。

接下来GL将把当前光栅位置的z坐标和纹理坐标赋给每一个像素，从而把所得的RGBA颜色值转化为片断。并用下式计算第n个片断的x和y窗口坐标：

$$x_n = x_r + n \bmod width$$

$$y_n = y_r + n / width$$

式中(x_r, y_r)是当前光栅的位置。这些像素片断被当作和通过光栅化点、线和多边形而得到的片断一样处理。在片断写入帧缓冲区以前将对其进行纹理映射、雾化等片断操作。

GL_RED:

每个像素是一个单一的红组分。这一组分被转化成内部浮点格式的方式和一个RGBA像素的红组分一样。然后把绿色和蓝色组分设置为0，alpha设置为1，从而转化为一个RGBA像素。通过这种方法得到的像素被当作和直接读取的RGBA像素一样处理。

GL_GREEN:

每个像素是一个单一的绿组分。这一组分被转化成内部浮点格式的方式和一个RGBA像素的绿组分一样。然后把红色和蓝色组分设置为0，alpha设置为1，从而转化为一个RGBA像素。通过这种方法得到的像素被当作和直接读取的RGBA像素一样处理。

GL_BLUE:

每个像素是一个单一的蓝组分。这一组分被转化成内部浮点格式的方式和一个RGBA像素的蓝组分一样。然后把红色和绿色组分设置为0，alpha设置为1，从而转化为一个RGBA像素。通过这种方法得到的像素被当作和直接读取的RGBA像素一样处理。

GL_ALPHA:

每个像素是一个单一的alpha组分。这一组分被转化成内部浮点格式的方式和一个RGBA像素的alpha组分一样。然后把红色、绿色和蓝色组分设置为0，从而转化为一个RGBA像素。通过这种方法得到的像素被当作和直接读取的RGBA像素一样处理。

GL_RGB

GL_BGR:

每个像素是一个三组分构成的组：对于**GL_RGB**模式，依次为红、绿、蓝；对于**GL_BGR**模式，则依次为蓝、绿、红。每一组分转化成内部浮点格式的方式和一个RGBA像素的红、绿和蓝组分一样。然后把alpha组分设置为1，从而将此三组分组转化为一个RGBA像素。通过这种方法得到的像素被当作和直接读取的RGBA像素一样处理。

GL_LUMINANCE:

每个像素是一个单一的亮度组分。这一组分转化成内部浮点格式的方式和一个RGBA像素的红组分一样。然后把红色、绿色和蓝色组分设置为已转化的亮度值，alpha组分设置为1，从而转化为一个RGBA像素。通过这种方法得到的像素被当作和直接读取的RGBA像素一样处理。

GL_LUMINANCE_ALPHA:

每个像素是一个两组分构成的组：亮度组分和alpha组分。这两种组分转化成内部浮点格式的方式和一个RGBA像素的红组分一样。然后把红色、绿色和蓝色组分设置为已转化的亮度值，alpha组分设置为相应的alpha值，从而转化为一个RGBA像素。通过这种方法得到的像素被当作和直接读取的RGBA像素一样处理。

表5-11概括了type可接受的有效常量的含义：

表 5-11

类 型	相 应 的 数据 类型
GL_UNSIGNED_BYTE	无符号8位整数
GL_BYTE	带符号8位整数
GL_BITMAP	无符号8位整数中的单个位
GL_UNSIGNED_SHORT	无符号16位整数
GL_SHORT	带符号16位整数
GL_UNSIGNED_INT	无符号32位整数
GL_INT	32位整数
GL_FLOAT	单精度浮点数
GL_UNSIGNED_BYTE_3_3_2	无符号8位整数
GL_UNSIGNED_BYTE_2_3_3_REV	具有相反组分顺序的无符号8位整数
GL_UNSIGNED_SHORT_5_6_5	无符号16位整数
GL_UNSIGNED_SHORT_5_6_5_REV	具有相反组分顺序的无符号16位整数
GL_UNSIGNED_SHORT_4_4_4	无符号16位整数
GL_UNSIGNED_SHORT_4_4_4_REV	具有相反组分顺序的无符号16位整数
GL_UNSIGNED_SHORT_5_5_5_1	无符号16位整数
GL_UNSIGNED_SHORT_1_5_5_5_REV	具有相反组分顺序的无符号16位整数
GL_UNSIGNED_INT_8_8_8_8	无符号32位整数
GL_UNSIGNED_INT_8_8_8_8_REV	具有相反组分顺序的无符号32位整数
GL_UNSIGNED_INT_10_10_10_2	无符号32位整数
GL_UNSIGNED_INT_2_10_10_10_REV	具有相反组分顺序的无符号32位整数

在以上的光栅化介绍中我们都假设像素的缩放因子为1。如果已用函数**glPixelZoom()**改变了x和y像素缩放因子，可通过下式把像素转化为片断。设(x_r, y_r)是当前光栅位置，给定的像素位于像素矩形的第n列第m行。则产生的片断位于一个像素矩形的中央。此像素矩形的四个角为：

$$(x_r + \text{zoom}_x n, y_r + \text{zoom}_y m)$$

$$(x_r + \text{zoom}_x (n + 1), y_r + \text{zoom}_y (m + 1))$$

此处 zoom_x 是**GL_ZOOM_X**的值， zoom_y 是**GL_ZOOM_Y**的值。

- 注意：

只有在GL 1.2以上的版本中，**GL_BGR**和**GL_BGRA**才是参数**format**的有效值。

只有在GL 1.2以上的版本中，`GL_UNSIGNED_BYTE_3_3_2`、`GL_UNSIGNED_BYTE_2_3_3_REV`、`GL_UNSIGNED_SHORT_5_6_5`、`GL_UNSIGNED_SHORT_5_6_5_REV`、`GL_UNSIGNED_SHORT_4_4_4_4`、`GL_UNSIGNED_SHORT_4_4_4_4_REV`、`GL_UNSIGNED_SHORT_5_5_5_1`、`GL_UNSIGNED_SHORT_1_5_5_5_REV`、`GL_UNSIGNED_INT_8_8_8_8`、`GL_UNSIGNED_INT_8_8_8_8_REV`、`GL_UNSIGNED_INT_10_10_10_2`和`GL_UNSIGNED_INT_2_10_10_10_REV`才是参数*type*的有效值。

- 出错提示：

当参数*width*或*height*是负数时产生`GL_INVALID_VALUE`提示。

当参数*format*和*type*不是一个可接受的值时产生`GL_INVALID_ENUM`提示。

当参数*format*是`GL_RED`、`GL_GREEN`、`GL_BLUE`、`GL_ALPHA`、`GL_RGB`、`GL_RGBA`、`GL_BGR`、`GL_BGRA`、`GL_LUMINANCE`或`GL_LUMINANCE_ALPHA`，但GL处于颜色索引模式下时产生`GL_INVALID_OPERATION`提示。

当参数*type*是`GL_BITMAP`，但参数*format*既非`GL_COLOR_INDEX`又非`GL_STENCIL_INDEX`时产生`GL_INVALID_ENUM`提示。

当参数*format*是`GL_STENCIL_INDEX`，但没有模板缓冲区时产生`GL_INVALID_OPERATION`提示。

当函数`glDrawPixels()`在函数对`glBegin()`/`glEnd()`之间执行时产生`GL_INVALID_OPERATION`提示。

当参数*type*是`GL_UNSIGNED_BYTE_3_3_2`、`GL_UNSIGNED_BYTE_2_3_3_REV`、`GL_UNSIGNED_SHORT_5_6_5`、`GL_UNSIGNED_SHORT_5_6_5_REV`之一，但*format*却不是`GL_RGB`时产生`GL_INVALID_OPERATION`提示。

当参数*type*为`GL_UNSIGNED_SHORT_4_4_4_4`、`GL_UNSIGNED_SHORT_4_4_4_4_REV`、`GL_UNSIGNED_SHORT_5_5_5_1`、`GL_UNSIGNED_SHORT_1_5_5_5_REV`、`GL_UNSIGNED_INT_8_8_8_8`、`GL_UNSIGNED_INT_8_8_8_8_REV`、`GL_UNSIGNED_INT_10_10_10_2`或`GL_UNSIGNED_INT_2_10_10_10_REV`之一，但*format*既不是`GL_RGBA`又不是`GL_BGRA`时产生`GL_INVALID_OPERATION`提示。

- 有关数据的获取：

`glGet(GL_CURRENT_RASTER_POSITION)`

`glGet(GL_CURRENT_RASTER_POSITION_VALID)`

- 请参阅：

`glAlphaFunc()`, `glBlendFunc()`, `glCopyPixels()`, `glDepthFunc()`, `glLogicOp()`, `glPixelMap()`, `glPixelStore()`, `glPixelTransfer()`, `glPixelZoom()`, `glRasterPos()`, `glReadPixels()`, `glScissor()`, `glStencilFunc()`

- **glDrawRangeElements**

- 名称：

`glDrawRangeElements()`

- 功能：

从数组数据中绘制图元。

- C 描述：

```
void glDrawRangeElements( GLenum mode,
                          GLuint start,
                          GLuint end,
                          GLsizei count,
                          GLenum type,
                          const GLvoid *indices )
```

- 参数说明：

mode 指定要绘制的图元的种类。它可取下面的符号常量：**GL_POINTS**、**GL_LINE_STRIP**、**GL_LINE_LOOP**、**GL_LINES**、**GL_TRIANGLE_STRIP**、**GL_TRIANGLE_FAN**、**GL_TRIANGLES**、**GL_QUAD_STRIP**、**GL_QUADS**和**GL_POLYGON**。

start 指定*indices*中包含的最小的数组索引。

end 指定*indices*中包含的最大的数组索引。

count 指定要绘制的图元数目。

type 指定在*count*中值的类型。它必须是下列各值之一：**GL_UNSIGNED_BYTE**，**GL_UNSIGNED_SHORT**或**GL_UNSIGNED_INT**。

indices 指定一个指向索引的存储单元的指针。

- 说明：

函数**glDrawRangeElements()**是函数**glDrawElements()**的带约束格式。参数*mode*、*start*、*end*和*count*与函数**glDrawElements()**的相应参数是相匹配的，其附加的约束是数组中的*count*个值都应位于*start*和*end*（包括*start*, *end*）之间。

用户可用函数**glGet(GL_MAX_ELEMENTS_VERTICES)**和**glGet(GL_MAX_ELEMENTS_INDICES)**来查询具体实现的顶点和索引数据的最大推荐值。当*end-start+1*大于**GL_MAX_ELEMENTS_VERTICES**或当参数*count*大于**GL_MAX_ELEMENTS_INDICES**时，该操作将在一个降低的性能上进行。这里并没有要求提供所有范围[*start*, *end*]内的顶点。然而，具体实现可以对无用的顶点进行部分地操作，产生比优化索引集要差一些的性能。

当调用函数**glDrawRangeElements()**时，它将从参数*start*开始，使用从一个有效的数组中得到的*count*个连续元素，构成一个几何图元序列。参数*mode*指定了所构成图元的种类及这些图元的构成方式。当多个数组有效时，每个数组都可使用。当**GL_VERTEX_ARRAY**无效时，将不产生任何几何图元。

由函数**glDrawRangeElements()**修正的顶点属性在该函数返回后将产生一个不确定的值。例如，当**GL_COLOR_ARRAY**有效时，函数**glDrawRangeElements()**执行后当前颜色的值将是未定义的。没有被修正的属性将仍保持原来的值。

- 注意：

函数**glDrawRangeElements()**只有在GL 1.2以上的版本中才有效。

函数`glDrawRangeElements()`可以被包括在显示列表中。当它被包括在一个显示列表中时，其所需的数组数据（由数组指针确定并启动）也进入了显示列表。由于数组指针和启动操作都是客户端状态，所以它们的值将对显示列表的建立产生影响，但并不影响显示列表的执行。

- 出错提示：

索引位于范围[*start*, *end*]之外是错误的。但具体实现中可能不检测这种情况。这样的索引将导致与具体实现有关的行为。

当参数*mode*不是一个可接受的值时产生**GL_INVALID_ENUM**提示。

当参数*count*是负数时产生**GL_INVALID_VALUE**提示。

当*end < start*时产生**GL_INVALID_VALUE**提示。

当函数`glDrawRangeElements()`在函数对`glBegin()`/`glEnd()`之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

`glGet(GL_MAX_ELEMENTS_VERTICES)`

`glGet(GL_MAX_ELEMENTS_INDICES)`

- 请参阅：

`glArrayElement()`, `glColorPointer()`, `glDrawArrays()`, `glDrawElements()`,
`glEdgeFlagPointer()`, `glGetPointerv()`, `glIndexPointer()`, `glInterleavedArrays()`,
`glNormalPointer()`, `glTexCoordPointer()`, `glVertexPointer()`

glEdgeFlag

- 名称：

`glEdgeFlag()`, `glEdgeFlagv()`

- 功能：

指定某边是被当作边界还是被当作非边界。

- C描述：

`void glEdgeFlag(GLboolean flag)`

- 参数说明：

flag 指定当前边界标志的值，它可以是**GL_TRUE**或**GL_FALSE**。其默认值为**GL_TRUE**。

- C描述：

`void glEdgeFlagv(const GLboolean *flag)`

- 参数说明：

flag 指定一个指针，指向一个包含单一boolean元素的数组。这一元素将用来替换当前边界标志的值。

- 说明：

每一个在函数对`glBegin()`/`glEnd()`之间定义的多边形、单个的三角形、单个的四边形的顶点，都被标记为一个边界边或一个非边界边的开始。当指定顶点时其当前边界标记是“真”，则这一顶点被标记为一个边界边的起始点。否则，顶点被标记为一个非边界边的起始点。如果参数*flag*

为**GL_TRUE**, 函数**glEdgeFlag()**把边界标志位设置为**GL_TRUE**。否则, 把边界标志位设置为**GL_FALSE**。

相互连接的三角形和四边形的顶点通常都是被设置为边界边而不考虑其边界标志的具体值。

只有当**GL_POLYGON_MODE**被设置成**GL_POINT**或**GL_LINE**时, 顶点的边界边和非边界边标志才有意义。请参阅**glPolygonMode()**。

- 注意:

当前的边界标志可以被随时更新。特别是, 函数**glEdgeFlag()**可以在函数对**glBegin()**/**glEnd()**之间被调用。

- 有关数据的获取:

glGet(GL_EDGE_FLAG)

- 请参阅:

glBegin(), **glEdgeFlagPointer()**, **glPolygonMode()**

glEdgeFlagPointer

- 名称:

glEdgeFlagPointer()

- 功能:

定义一个边界标志数组。

- C描述:

```
void glEdgeFlagPointer( GLsizei stride,
                        const GLboolean *pointer )
```

- 参数说明:

stride 指定连续边界标志间的字节偏移量。缺省情况下, 其值是0。如果参数*stride*是0, 则认为顶点被连续地存放在数组中。

pointer 指定一个指针, 指向数组中的第一个边界标志。其缺省值是0。

- 说明:

函数**glEdgeFlagPointer()**的作用是指定一个布尔型边界标志数组的存放位置和数据格式。该边界标志将用于绘图操作。参数*stride*指定存放在一个数组中或分别存放在多个数组中的、所允许的顶点和属性的一个边界标志到下一个边界标志之间的字节偏移量。(在一些实现中, 单一数组的存放形式可能更有效。请参阅**glInterleavedArrays()**)

当指定一个边界标志数组后, 参数*stride*和*pointer*将被存储为客户端状态。用函数 **glEnableClientState(GL_EDGE_FLAG_ARRAY)**和 **glDisableClientState(GL_EDGE_FLAG_ARRAY)**可以启动和关闭边界标志数组。启动边界标志数组后, 当调用函数**glDrawArrays()**、**glDrawElements()**或**glArrayElement()**时可使用这个边界标志数组。

函数**glDrawArrays()**用来将预先指定的顶点和顶点属性数组构造成一个图元序列(所有相同的类型)。函数**glArrayElement()**可以通过检索顶点和顶点属性来指定图元。函数**glDrawElements()**则是通过检索顶点和顶点属性来构造一个图元序列。

- 注意:

函数`glEdgeFlagPointer()`只有在GL 1.1以上的版本中才有效。

边界标志数组在缺省情况下是关闭的，这时函数`glDrawArrays()`、`glDrawElements()`或`glArrayElement()`不能访问它。

函数`glEdgeFlagPointer()`不允许在函数对`glBegin()`/`glEnd()`之间执行，否则可能产生一个出错提示，但也有可能不产生出错提示。如果没有产生出错提示，其运行将是未定义的。

函数`glEdgeFlagPointer()`是典型的客户端执行方式。

因为边界标志数组参数是客户端状态，所以它不能用函数`glPushAttrib()`和`glPopAttrib()`存储和恢复，而应用函数`glPushClientAttrib()`和`glPopClientAttrib()`代替进行存储和恢复操作。

- 出错提示：

当参数`stride`是负数时产生`GL_INVALID_ENUM`提示。

- 有关数据的获取：

```
glIsEnabled( GL_EDGE_FLAG_ARRAY )
glGet( GL_EDGE_FLAG_ARRAY_STRIDE )
glGetPointerv( GL_EDGE_FLAG_ARRAY_POINTER )
```

- 请参阅：

`glArrayElement()`, `glColorPointer()`, `glDrawArrays()`, `glDrawElements()`, `glEnable()`,
`glGetPointerv()`, `glIndexPointer()`, `glNormalPointer()`, `glPopClientAttrib()`, `glPushClientAttrib()`,
`glTexCoordPointer()`, `glVertexPointer()`

- **glEnable, glDisable**

- 名称：

`glEnable()`, `glDisable()`

- 功能：

启动和关闭服务器端的GL功能。

- C描述：

```
void glEnable( GLenum cap )
```

- 参数说明：

`cap` 指定一个代表某种GL功能的符号常量。

- C描述：

```
void glDisable( GLenum cap )
```

- 参数说明：

`cap` 指定一个代表某种GL功能的符号常量。

- 说明：

函数`glEnable()`和`glDisable()`的作用是启动和关闭各种GL功能。用函数`glIsEnabled()`和`glGet()`来确定任一功能的当前设置。除`GL_DITHER`外，其他功能的缺省值都是`GL_FALSE`。`GL_DITHER`的缺省值是`GL_TRUE`。

函数`glEnable()`和`glDisable()`都只带有一个自变量`cap`，该参数可取下面所列各值：

<code>GL_ALPHA_TEST</code>	当启动该功能时，将进行alpha测试。请参阅 <code>glAlphaFunc()</code>
----------------------------	---

GL_AUTO_NORMAL	当启动该功能时，如果正使用 GL_MAP2_VERTEX_3 或 GL_MAP2_VERTEX_4 生成顶点，它将产生一个法向量。请参阅 glMap2o() 。
GL_BLEND	当启动该功能时，它将对颜色缓冲区中的值和输入的RGBA颜色值进行融合操作。请参阅 glBlendFunc() 。
GL_CLIP_PLANE<i>i</i>	当启动该功能时，它将用用户指定的剪切平面 <i>i</i> 对几何体进行剪切操作。请参阅 glClipPlane() 。
GL_COLOR_LOGIC_OP	当启动该功能时，它将对颜色缓冲区中的值和输入的RGBA值进行当前选定的逻辑操作。请参阅 glLogicOp() 。
GL_COLOR_MATERIAL	当启动该功能时，它将用一个或多个材料参数跟踪当前的颜色。请参阅 glColorMaterial() 。
GL_COLOR_TABLE	当启动该功能时，它将对输入的RGBA颜色值进行颜色表查询操作。请参阅 glColorTable() 。
GL_CONVOLUTION_1D	当启动该功能时，它将对输入的RGBA颜色值进行1D卷积操作。请参阅 glConvolutionFilter1D() 。
GL_CONVOLUTION_2D	当启动该功能时，它将对输入的RGBA颜色值进行2D卷积操作。请参阅 glConvolutionFilter2D() 。
GL_CULL_FACE	当启动该功能时，它将按多边形在窗口坐标中的缠绕方向拣选多边形。请参阅 glCullFace() 。
GL_DEPTH_TEST	当启动该功能时，它将进行深度比较操作并更新深度缓冲区。请注意即使深度缓冲区已经存在并且其屏蔽值不为0，如果深度测试功能是关闭的，该缓冲区也不能被更新。请参阅 glDepthFunc() 和 glDepthRange() 。
GL_DITHER	当启动该功能时，它将在颜色组分或索引写入颜色缓冲区之前对它们进行抖动操作。
GL_FOG	当启动该功能时，它将把雾颜色混入随后的纹理颜色中。请参阅 glFog() 。
GL_HISTOGRAM	当启动该功能时，它将对输入的RGBA颜色值进行直方图操作。请参阅 glHistogram() 。
GL_INDEX_LOGIC_OP	当启动该功能时，它将对输入的索引和颜色缓冲区中的颜色使用当前选定的逻辑操作。请参阅 glLogicOp() 。
GL_LIGHT<i>i</i>	当启动该功能时，它将把光源 <i>i</i> 包含进光照方程中。请参阅

	glLightModel()和glLight()。
GL_LIGHTING	当启动该功能时，它将使用当前的光照参数计算顶点的颜色或索引。否则，将当前的颜色或索引简单对应到每个顶点。请参阅glMaterial()、glLightModel()和glLight()。
GL_LINE_SMOOTH	当启动该功能时，它将使用正确的滤波方式绘制线段。否则，它将绘出走样线段。请参阅glLineWidth()。
GL_LINE_STIPPLE	当启动该功能时，它将使用线段当前线段的点画模板绘制线段。请参阅glLineStipple()。
GL_MAP1_COLOR_4	当启动该功能时，它将调用函数glEvalCoord1()、glEvalMesh1()和glEvalPoint1()来生成RGBA值。请参阅glMap1()。
GL_MAP1_INDEX	当启动该功能时，它将调用函数glEvalCoord1()、glEvalMesh1()和glEvalPoint1()来生成颜色索引值。请参阅glMap1()。
GL_MAP1_NORMAL	当启动该功能时，它将调用函数glEvalCoord1()、glEvalMesh1()和glEvalPoint1()来生成法线。请参阅glMap1()。
GL_MAP1_TEXTURE_COORD_1	当启动该功能时，它将调用函数glEvalCoord1()、glEvalMesh1()和glEvalPoint1()来生成s纹理坐标。请参阅glMap1()。
GL_MAP1_TEXTURE_COORD_2	当启动该功能时，它将调用函数glEvalCoord1()、glEvalMesh1()和glEvalPoint1()来生成s和t纹理坐标。请参阅glMap1()。
GL_MAP1_TEXTURE_COORD_3	当启动该功能时，它将调用函数glEvalCoord1()、glEvalMesh1()和glEvalPoint1()来生成s、t和r纹理坐标。请参阅glMap1()。
GL_MAP1_TEXTURE_COORD_4	当启动该功能时，它将调用函数glEvalCoord1()、glEvalMesh1()和glEvalPoint1()来生成s、t、r和q纹理坐标。请参阅glMap1()。
GL_MAP1_VERTEX_3	当启动该功能时，它将调用函数glEvalCoord1()、glEvalMesh1()和glEvalPoint1()来生成x、y和z顶点坐标。请参阅glMap1()。
GL_MAP1_VERTEX_4	当启动该功能时，它将调用函数glEvalCoord1()、glEvalMesh1()和glEvalPoint1()来生成x、y、z和w顶点坐标。请参阅glMap1()。
GL_MAP2_COLOR_4	

GL_MAP2_INDEX

当启动该功能时，它将调用函数glEvalCoord2()、glEvalMesh2()和glEvalPoint2()来生成RGBA值。请参阅glMap2()。

GL_MAP2_NORMAL

当启动该功能时，它将调用函数glEvalCoord2()、glEvalMesh2()和glEvalPoint2()来生成颜色索引值。请参阅glMap2()。

GL_MAP2_TEXTURE_COORD_1

当启动该功能时，它将调用函数glEvalCoord2()、glEvalMesh2()和glEvalPoint2()来生成法线值。请参阅glMap2()。

GL_MAP2_TEXTURE_COORD_2

当启动该功能时，它将调用函数glEvalCoord2()、glEvalMesh2()和glEvalPoint2()来生成s纹理坐标。请参阅glMap2()。

GL_MAP2_TEXTURE_COORD_3

当启动该功能时，它将调用函数glEvalCoord2()、glEvalMesh2()和glEvalPoint2()来生成s、t和r纹理坐标。请参阅glMap2()。

GL_MAP2_TEXTURE_COORD_4

当启动该功能时，它将调用函数glEvalCoord2()、glEvalMesh2()和glEvalPoint2()来生成s、t、r和q纹理坐标。请参阅glMap2()。

GL_MAP2_VERTEX_3

当启动该功能时，它将调用函数glEvalCoord2()、glEvalMesh2()和glEvalPoint2()来生成x、y和z顶点坐标。请参阅glMap2()。

GL_MAP2_VERTEX_4

当启动该功能时，它将调用函数glEvalCoord2()、glEvalMesh2()和glEvalPoint2()来生成x、y、z和w顶点坐标。请参阅glMap2()。

GL_MINMAX

当启动该功能时，它将计算输入的RGBA颜色值的最小和最大值。请参阅glMinmax()。

GL_NORMALIZE

当启动该功能时，它将在转换结束后把由函数glNormal()指定的法向量缩放成单位长度。请参阅glNormal()。

GL_POINT_SMOOTH

当启动该功能时，它将使用合适的滤波方式绘制点。否则，它将绘出走样点。请参阅glPointSize()。

GL_POLYGON_OFFSET_FILL

当启动该功能时，它将对GL_FILL模式下绘制的多边形进行如下操作：在执行深度比较前先将一个多边形片断的深度值加上一个偏移量。请参阅glPolygonOffset()。

GL_POLYGON_OFFSET_LINE

当启动该功能时，它将对**GL_LINE**模式下绘制的多边形进行如下操作：在执行深度比较前先将一个多边形片断的深度值加上一个偏移量。请参阅glPolygonOffset()。

GL_POLYGON_OFFSET_POINT

当启动该功能时，它将对**GL_POINT**模式下绘制的多边形进行如下操作：在执行深度比较前先将一个多边形片断的深度值加上一个偏移量。请参阅glPolygonOffset()。

GL_POLYGON_SMOOTH

当启动该功能时，它将使用合适的滤波方式绘制多边形。否则，它将绘出走样多边形。对于一个正确的反走样多边形来说，必须有一个alpha深度缓冲区且该多边形应该是从前向后存储的。

GL_POLYGON_STIPPLE

当启动该功能时，它将使用当前多边形的点画模板绘制多边形。请参阅glPolygonStipple()。

GL_POST_COLOR_MATRIX_COLOR_TABLE

当启动该功能时，它将在颜色矩阵转换结束后对RGBA颜色值执行一个颜色表查询。请参阅glColorTable()。

GL_POST_CONVOLUTION_COLOR_TABLE

当启动该功能时，它将在卷积操作结束后对RGBA颜色值执行一个颜色表查询。请参阅glColorTable()。

GL_RESCALE_NORMAL

当启动该功能时，它将在转换结束后把由函数glNormal()指定的法向量缩放成单位长度。请参阅glNormal()。

GL_SEPARABLE_2D

当启动该功能时，它将使用一个分离的卷积滤波器对输入的ROBA颜色值进行二维卷积操作。请参阅glSeparableFilter2D()。

GL_SCISSOR_TEST

当启动该功能时，它将把裁剪矩形之外的片断裁剪掉。请参阅glScissor()。

GL_STENCIL_TEST

当启动该功能时，它将执行模板测试操作并更新模板缓冲区。请参阅glStencilFunc()和glStencilOp()。

GL_TEXTURE_1D

当启动该功能时，它将执行一维纹理操作（除非二维或三维纹理操作有效）。请参阅glTexImage1D()。

GL_TEXTURE_2D

当启动该功能时，它将执行二维纹理操作（除非三维纹理操作有效）。请参阅glTexImage2D()。

GL_TEXTURE_3D

当启动该功能时，它将执行三维纹理操作。请参阅glTex-

Image3D()。

GL_TEXTURE_GEN_Q

当启动该功能时，它将用由函数glTexGen()定义的纹理生成函数计算 q 纹理坐标。否则，使用当前的 q 纹理坐标。请参阅glTexGen()。

GL_TEXTURE_GEN_R

当启动该功能时，它将用由函数glTexGen()定义的纹理生成函数计算 r 纹理坐标。否则，使用当前的 r 纹理坐标。请参阅glTexGen()。

GL_TEXTURE_GEN_S

当启动该功能时，它将用由函数glTexGen()定义的纹理生成函数计算 s 纹理坐标。否则，使用当前的 s 纹理坐标。请参阅glTexGen()。

GL_TEXTURE_GEN_T

当启动该功能时，它将用由函数glTexGen()定义的纹理生成函数计算 t 纹理坐标。否则，使用当前的 t 纹理坐标。请参阅glTexGen()。

- 注意：

GL_POLYGON_OFFSET_FILL、**GL_POLYGON_OFFSET_LINE**、**GL_POLYGON_OFFSET_POINT**、**GL_COLOR_LOGIC_OP**和**GL_INDEX_LOGIC_OP**只有在GL 1.1以上的版本中才有效。

GL_RESCALE_NORMAL和**GL_TEXTURE_3D**只有在GL 1.2以上的版本中才有效。

当调用函数glGet(GL_EXTENSIONS)的返回值是**GL_ARB_imaging**时，**GL_COLOR_TABLE**、**GL_CONVOLUTION_1D**、**GL_CONVOLUTION_2D**、**GL_HISTOGRAM**、**GL_MINMAX**、**GL_POST_COLOR_MATRIX_COLOR_TABLE**、**GL_POST_CONVOLUTION_COLOR_TABLE**和**GL_SEPARABLE_2D**才有效。

当系统支持**GL_ARB_multitexture**时，**GL_TEXTURE_1D**、**GL_TEXTURE_2D**、**GL_TEXTURE_3D**、**GL_TEXTURE_GEN_S**、**GL_TEXTURE_GEN_T**、**GL_TEXTURE_GEN_R**和**GL_TEXTURE_GEN_Q**可以启动或关闭由函数glActiveTextureARB指定的活动纹理单位的各个状态。

- 出错提示：

当参数 cap 不是前面所列的各值之一时产生**GL_INVALID_ENUM**提示。

当函数glEnable()或glDisable()在函数对glBegin()/glEnd()之间执行时产生**GL_INVALID_OPERATION**提示。

- 请参阅：

glActiveTextureARB(), glAlphaFunc(), glBlendFunc(), glClipPlane(), glColorMaterial(), glCullFace(), glDepthFunc(), glDepthRange(), glEnableClientState(), glFog(), glGet(),

glIsEnabled(), **glLight()**, **glLightModel()**, **glLineWidth()**, **glLineStipple()**, **glLogicOp()**, **glMap1()**, **glMap2()**, **glMaterial()**, **glNormal()**, **glPointSize()**, **glPolygonMode()**, **glPolygonOffset()**, **glPolygonStipple()**, **glScissor()**, **glStencilFunc()**, **glStencilOp()**, **glTexGen()**, **glTexImage1D()**, **glTexImage2D()**, **glTexImage3D()**

• **glEnableClientState**, **glDisableClientState**

- 名称:

glEnableClientState(), **glDisableClientState()**

- 功能:

启动和关闭客户端功能。

- C 描述:

`void glEnableClientState(GLenum array)`

- 参数说明:

array 指定要启动的功能。它可取如下的符号常量: **GL_COLOR_ARRAY**、**GL_EDGE_FLAG_ARRAY**、**GL_INDEX_ARRAY**、**GL_NORMAL_ARRAY**、**GL_TEXTURE_COORD_ARRAY** 和 **GL_VERTEX_ARRAY**。

- C 描述:

`void glDisableClientState(GLenum array)`

- 参数说明:

array 指定要关闭的功能

- 说明:

函数**glEnableClientState()**和**glDisableClientState()**的作用是启动和关闭某个客户端功能。缺省情况下，所有客户端功能都处于关闭状态。函数**glEnableClientState()**和**glDisableClientState()**都只带有一个参数*array*，该参数可取如下的值：

GL_COLOR_ARRAY 当启动该功能时，颜色矩阵可以在调用函数**glArrayElement()**、**glDrawArrays()**、**glDrawElements()**或**glDrawRangeElements()**绘图过程中被使用和写入。请参阅**glColorPointer()**。

GL_EDGE_FLAG_ARRAY

当启动该功能时，边界标志矩阵可以在调用函数**glArrayElement()**、**glDrawArrays()**、**glDrawElements()**或**glDrawRangeElements()**绘图过程中被使用和写入。请参阅**glEdgeFlagPointer()**。

GL_INDEX_ARRAY

当启动该功能时，索引矩阵可以在调用函数**glArrayElement()**、**glDrawArrays()**、**glDrawElements()**或**glDrawRangeElements()**绘图过程中被使用和写入。请参阅**glIndexPointer()**。

GL_NORMAL_ARRAY

当启动该功能时，法线矩阵可以在调用函数**glArrayElement()**、**glDrawArrays()**、**glDrawElements()**或**glDrawRangeElements()**绘图过程中被使用和写入。请参阅**glNormalPointer()**。

GL_TEXTURE_COORD_ARRAY

当启动该功能时，纹理坐标矩阵可以在调用函数glArrayElement()、glDrawArrays()、glDrawElements()或glDrawRangeElements()绘图过程中被使用和写入。请参阅glTexCoordPointer()。

GL_VERTEX_ARRAY

当启动该功能时，顶点矩阵可以在调用函数glArrayElement()、glDrawArrays()、glDrawElements()或glDrawRangeElements()绘图过程中被使用和写入。请参阅glVertexPointer()。

- 注意：

函数glEnableClientState()仅在GL 1.1以上的版本中才有效。

当系统支持**GL_ARB_multitexture**时，启动和关闭**GL_TEXTURE_COORD_ARRAY**将对当前有效的客户纹理单元产生影响。该客户纹理单元由函数glClientActiveTextureARB()控制。

- 出错提示：

当参数array不是一个可接受的值时产生**GL_INVALID_ENUM**提示。

函数glEnableClientState()在函数对glBegin()/glEnd()之间执行是不允许的，但这时可能不出现出错提示。如果没有出现出错提示，系统的行为将是未定义的。

- 请参阅：

glArrayElement(), **glClientActiveTextureARB()**, **glColorPointer()**, **glDrawArrays()**,
glDrawElements(), **glEdgeFlagPointer()**, **glEnable()**, **glGetPointerv()**, **glIndexPointer()**,
glInterleavedArrays(), **glNormalPointer()**, **glTexCoordPointer()**, **glVertexPointer()**

glEvalCoord

- 名称：

glEvalCoord1d(), **glEvalCoord1f()**, **glEvalCoord2d()**, **glEvalCoord2f()**, **glEvalCoord1dv()**,
glEvalCoord1fv(), **glEvalCoord2dv()**, **glEvalCoord2fv()**

- 功能：

求取有效的一维和二维映射的值。

- C描述：

```
void glEvalCoord1d( GLdouble u )
void glEvalCoord1f( GLfloat u )
void glEvalCoord2d( GLdouble u,
                   GLdouble v )
void glEvalCoord2f( GLfloat u,
                   GLfloat v )
```

- 参数说明：

u 指定一个已经由函数glMap1()或glMap2()定义的基础函数的域坐标*u*的值。

v 指定一个已经由函数glMap1()或glMap2()定义的基础函数的域坐标*v*的值。函数glEvalCoord1()不需要提供该参数。

- C描述：

```

void glEvalCoord1dv( const GLdouble *u )
void glEvalCoord1fv( const GLfloat *u )
void glEvalCoord2dv( const GLdouble *u )
void glEvalCoord2fv( const GLfloat *u )

```

• 参数说明：

u 指定一个指针，指向一个包含一维或二维域坐标的数组。它的第一个坐标是*u*，第二个坐标是*v*，第二个坐标仅提供给函数glEvalCoord2()。

• 说明：

函数glEvalCoord1()用自变量*u*来求取有效的一维映射值。函数glEvalCoord2()用两个域值*u*和*v*来求取有效的二维映射的值。函数glMap1()和glMap2()用来定义一个映射，函数glEnable()和glDisable()可以启动和关闭该映射。

当一个glEvalCoord()命令被发布时，所有指定维数的当前有效的映射将求值。对于每个有效的映射来说，就相当于发布了带有计算所得值的相应的GL命令。也就是说，当GL_MAP1_INDEX或GL_MAP2_INDEX有效时，模拟了一个glIndex()命令；当GL_MAP1_COLOR_4或GL_MAP2_COLOR_4有效时，模拟了一个glColor()命令；当GL_MAP1_NORMAL或GL_MAP2_NORMAL有效时，生成了法向量；当GL_MAP1_TEXTURE_COORD_1、GL_MAP1_TEXTURE_COORD_2、GL_MAP1_TEXTURE_COORD_3、GL_MAP1_TEXTURE_COORD_4、GL_MAP2_TEXTURE_COORD_1、GL_MAP2_TEXTURE_COORD_2、GL_MAP2_TEXTURE_COORD_3或GL_MAP2_TEXTURE_COORD_4有效时，模拟了一个适当的glTexCoord()命令。

对于GL求值所使用的颜色、颜色索引、法线和纹理坐标，当相应的求值器有效时，这些求得的值将替换该求值器的当前值。否则，所求得的值将不更新当前值。这样，如果glVertex()命令由glEvalCoord()命令替换，则与命令glVertex()相关的颜色、法线及纹理坐标将不受命令glEvalCoord()生成的值影响，但却受最新的glColor()、glIndex、glNormal()和glTexCoord()命令的影响。

当映射无效时，用于映射的命令将无效。对于特殊维数而言，当多个纹理求值器有效时（比如GL_MAP2_TEXTURE_COORD_1和GL_MAP2_TEXTURE_COORD_2同时有效），这时仅仅生成坐标数目多的映射求值被执行（如上述中的GL_MAP2_TEXTURE_COORD_2）。因此，GL_MAP1_VERTEX_4取代GL_MAP1_VERTEX_3，同样GL_MAP2_VERTEX_4取代GL_MAP2_VERTEX_3。如果对于指定的维数，三组分或四组分的顶点映射均无效，则忽略glEvalCoord()命令。

如果已启动法线自动生成功能，则调用函数glEnable(GL_AUTO_NORMAL)，glEvalCoord2()将解析地生成表面法线。这时将不考虑GL_MAP2_NORMAL的具体内容或它的映射是否有效。设

$$\mathbf{m} = \frac{\partial \mathbf{p}}{\partial u} \times \frac{\partial \mathbf{p}}{\partial v}$$

则生成的法线n为

$$\mathbf{n} = \frac{\mathbf{m}}{||\mathbf{m}||}$$

如果已关闭法线的自动生成功能，则当相应的法线映射**GL_MAP2_NORMAL**有效时，它将生成法线。如果当法线自动生成及法线映射均无效时，命令`glEvalCoord2()`将不生成法线。

- 有关数据的获取：

```
glIsEnabled( GL_MAP1_VERTEX_3 )
glIsEnabled( GL_MAP1_VERTEX_4 )
glIsEnabled( GL_MAP1_INDEX )
glIsEnabled( GL_MAP1_COLOR_4 )
glIsEnabled( GL_MAP1_NORMAL )
glIsEnabled( GL_MAP1_TEXTURE_COORD_1 )
glIsEnabled( GL_MAP1_TEXTURE_COORD_2 )
glIsEnabled( GL_MAP1_TEXTURE_COORD_3 )
glIsEnabled( GL_MAP1_TEXTURE_COORD_4 )
glIsEnabled( GL_MAP2_VERTEX_3 )
glIsEnabled( GL_MAP2_VERTEX_4 )
glIsEnabled( GL_MAP2_INDEX )
glIsEnabled( GL_MAP2_COLOR_4 )
glIsEnabled( GL_MAP2_NORMAL )
glIsEnabled( GL_MAP2_TEXTURE_COORD_1 )
glIsEnabled( GL_MAP2_TEXTURE_COORD_2 )
glIsEnabled( GL_MAP2_TEXTURE_COORD_3 )
glIsEnabled( GL_MAP2_TEXTURE_COORD_4 )
glIsEnabled( GL_AUTO_NORMAL )
```

glGetMap

- 请参阅：

`glBegin()`, `glColor()`, `glEnable()`, `glEvalMesh()`, `glEvalPoint()`, `glIndex()`, `glMap1()`,
`glMap2()`, `glMapGrid()`, `glNormal()`, `glTexCoord()`, `glVertex()`

*** glEvalMesh**

- 名称：

`glEvalMesh1()`, `glEvalMesh2()`

- 功能：

计算点或线的一维或二维网格。

- C 描述：

```
void glEvalMesh1( GLenum mode,
                  GLint i1,
                  GLint i2)
```

- 参数说明：

mode 在函数glEvalMesh1()中，指定是计算点还是线的一维网格。它可取的符号常量有：GL_POINT和GL_LINE。

i1, i2 指定网格区域变量*i*的第一个和最后一个整数值。

• C 描述：

```
void glEvalMesh2( GLenum mode,
                  GLint i1,
                  GLint i2,
                  GLint j1,
                  GLint j2 )
```

• 参数说明：

mode 在函数glEvalMesh2()中，指定是计算点、线还是多边形的二维网格。它可取的符号常量有：GL_POINT、GL_LINE和GL_FILL。

i1, i2 指定网格区域变量*i*的第一个和最后一个整数值。

j1, j2 指定网格区域变量*j*的第一个和最后一个整数值。

• 说明：

函数glMapGrid()和glEvalMesh()用于从前到后地生成和求取一系列平坦空间映射域的值。

函数glEvalMesh()穿过一个一维或二维网格的整数区域，其范围是函数glMap1()和glMap2()所指定的求值映射的区域。参数*mode*决定所得的顶点是由点、线还是填充多边形连接的。

对于一维情况，即函数glEvalMesh1()，网格的生成过程相当于执行下面的代码：

```
glBegin( type );
for ( i = i1; i <= i2; i += 1 )
    glEvalCoord1( i · Δu + u1 );
glEnd();
```

这里

$$\Delta u = (u_2 - u_1) / n$$

n、*u₁*和*u₂*是距其最近的函数glMapGrid1()的自变量。当*mode*是GL_POINT时，参数*type*是GL_POINTS；当*mode*是GL_LINE时，参数*type*是GL_LINES。

这里唯一的要求是当*i = n*时，由*i · Δu + u₁*计算所得的值要精确地等于*u₂*。

对于二维的情况，即函数glEvalMesh2()，设

$$\Delta u = (u_2 - u_1) / n$$

$$\Delta v = (v_2 - v_1) / m,$$

这里*n*、*u₁*、*u₂*、*m*、*v₁*和*v₂*是距其最近的函数glMapGrid2()的自变量。则，当参数*mode*是GL_FILL时，函数glEvalMesh2()等价于：

```
for ( j = j1; j < j2; j += 1 ) {
    glBegin( GL_QUAD_STRIP );
    for ( i = i1; i <= i2; i += 1 ) {
```

```

        glEvalCoord2( i · Δu + u1, j · Δv + v1 );
        glEvalCoord2( i · Δu + u1, (j+1) · Δv + v1 );
    }
    glEnd();
}

```

当参数*mode*是**GL_LINE**时，函数**glEvalMesh2()**等价于：

```

for ( j = j1; j <= j2; j += 1 ) {
    glBegin( GL_LINE_STRIP );
    for ( i = i1; i <= i2; i += 1 )
        glEvalCoord2( i · Δu + u1, j · Δv + v1 );
    glEnd();
}

for ( i = i1; i <= i2; i += 1 ) {
    glBegin( GL_LINE_STRIP );
    for ( j = j1; j <= j1; j += 1 )
        glEvalCoord2( i · Δu + u1, j · Δv + v1 );
    glEnd();
}

```

最后，当参数*mode*是**GL_POINT**时，函数**glEvalMesh2()**等价于：

```

glBegin( GL_POINTS );
for ( j = j1; j <= j2; j += 1 )
    for ( i = i1; i <= i2; i += 1 )
        glEvalCoord2( i · Δu + u1, j · Δv + v1 );
glEnd();

```

在上述三种情况下，这里唯一的要求是当*i=n*时，由*i · Δu+u₁*所求得的值要精确地等于₁；且当*j=m*时，由*j · Δv+v₁*所求得的值要精确地等于v₁。

- 出错提示：

当参数*mode*不是一个可取值时产生**GL_INVALID_ENUM**提示。

当函数**glEvalMesh()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGet(GL_MAP1_GRID_DOMAIN)
glGet(GL_MAP2_GRID_DOMAIN)
glGet(GL_MAP1_GRID_SEGMENTS)
glGet(GL_MAP2_GRID_SEGMENTS)

- 请参阅：

glBegin(), **glEvalCoord()**, **glEvalPoint()**, **glMap1()**, **glMap2()**, **glMapGrid()**

- **glEvalPoint**

- 名称：

glEvalPoint1(), **glEvalPoint2()**

- 功能：

生成并求取网格中的一个点。

- C 描述：

```
void glEvalPoint1( GLint i )
void glEvalPoint2( GLint i,
                  GLint j )
```

- 参数说明：

i 指定网格区域变量*i*的整数值。

j 指定网格区域变量*j*的整数值。(仅适用于glEvalPoint2())。

- 说明：

函数glMapGrid()和glEvalMesh()用来从前到后地生成和求取一系列平坦空间映射域的值。函数glEvalPoint()用来在由函数glEvalMesh()生成的网格空间中生成一个单独的网格点。函数glEvalPoint1()相当于执行如下的代码：

```
glEvalCoord1( i · Δu + u1 );
```

这里

$$\Delta u = (u_2 - u_1) / n$$

n、*u₁*和*u₂*是距其最近的函数glMapGrid1()的自变量。这里唯一的要求是当*i=n*时，由*i · Δu + u₁*所求得的值要精确地等于*u₂*。

对于二维的情况，即函数glEvalMesh2()，设

$$\Delta u = (u_2 - u_1) / n$$

$$\Delta v = (v_2 - v_1) / m,$$

这里*n*、*u₁*、*u₂*、*m*、*v₁*和*v₂*是距其最近的函数glMapGrid2()的自变量，则，函数glEvalMesh2()等价于：

```
glEvalCoord2( i · Δu + u1, j · Δv + v1 );
```

这里唯一的要求是当*i=n*时，由*i · Δu + u₁*计算所得的值要精确地等于*u₂*；且当*j=m*时，由*j · Δv + v₁*计算所得的值要精确地等于*v₂*。

- 有关数据的获取：

```
glGet( GL_MAP1_GRID_DOMAIN )
glGet( GL_MAP2_GRID_DOMAIN )
glGet( GL_MAP1_GRID_SEGMENTS )
glGet( GL_MAP2_GRID_SEGMENTS )
```

- 请参阅：

glEvalCoord(), glEvalMesh(), glMap1(), glMap2(), glMapGrid()

• glFeedbackBuffer

- 名称：

glFeedbackBuffer()

- 功能：

控制反馈模式。

- C 描述：

```
void glFeedbackBuffer( GLsizei size,
                      GLenum type,
                      GLfloat *buffer )
```

- 参数说明：

size 指定可以写入*buffer*中的数值的最大数目。

type 指定一个符号常量，用来描述返回给每个顶点的信息。它可以是**GL_2D**、**GL_3D**、**GL_3D_COLOR**、**GL_3D_COLOR_TEXTURE**和**GL_4D_COLOR_TEXTURE**。

buffer 返回的反馈数据。

- 说明：

函数**glFeedbackBuffer()**用来控制反馈。跟选择一样，反馈也是GL的模式之一。可通过调用函数**glRenderMode(GL_FEEDBACK)**选择反馈模式。当GL处于反馈模式下时，不能通过光栅化产生像素。将被光栅化的图元信息被GL反馈回应用程序。

函数**glFeedbackBuffer()**带有三个参数：参数*buffer*是一个指针，它指向一个用来存放反馈信息的浮点数组。参数*size*表明数组的大小。参数*type*是一个符号常量，用来描述返回给每个顶点的信息。函数**glFeedbackBuffer()**必须在进入反馈模式（用函数**glRenderMode(GL_FEEDBACK)**）之前发布。没有建立反馈缓冲区就设置**GL_FEEDBACK**或在反馈模式下调用函数**glFeedbackBuffer()**都是错误的。

在反馈模式下调用函数**glRenderMode()**时，它的返回值是存储于反馈数组中的条目的数目，并把反馈数组指针重设为指向反馈缓冲区的底部。返回值不能大于*size*。如果反馈的数据所要求的空间大于参数*buffer*所拥有的空间，则函数**glRenderMode()**返回一个负值。要跳出反馈模式，可调用一个参数值不是**GL_FEEDBACK**的函数**glRenderMode()**。

当在反馈模式下时，每一个被光栅化的图元、位图或像素矩形都将产生一个数值块，并把这些数值块拷贝到反馈数组中。如果这样做的结果使条目的数目超出了反馈数组所能容纳的最大值，这些数据块将被部分地写入数组（如果数组中还有部分剩余空间），并设置一个溢出标志。每一个数据块开始的地方都有一个代码用来说明图元的类型，然后才是描述图元的顶点和相关数据的值。位图和像素矩形也可以写入条目中。反馈操作只能在多边形拣选并且用函数**glPolygonMode()**编译多边形后才会发生。所以反馈缓冲区不返回拣选后的多边形。反馈也可能发生在多于三条边的多边形分解为三角形之后（如果你的特定的GL实现通过执行这种分解来绘制多边形）。

函数**glPassThrough()**被用作向反馈缓冲区插入一个标记。请参阅**glPassThrough()**。

下面是数据块写入反馈缓冲区的语法。每一图元都由单独的识别值说明，且后面跟有若干顶点数。多边形条目包含有一个整数值用来指明后面跟有的顶点的数目。每一个顶点都被反馈为几个由

参数`type`指定的浮点值。颜色被反馈为四个值（RGBA模式）或一个值（颜色索引模式）。

`feedbackList` \leftarrow `feedbackItem` `feedbackList` | `feedbackItem`

`feedbackItem` \leftarrow `point` | `lineSegment` | `polygon` | `bitmap` | `pixelRectangle` | `passThru`

`point` \leftarrow **GL_POINT_TOKEN** `vertex`

`lineSegment` \leftarrow **GL_LINE_TOKEN** `vertex` `vertex` | **GL_LINE_RESET_TOKEN** `vertex` `vertex`

`polygon` \leftarrow **GL_POLYGON_TOKEN** `n` `polySpec`

`polySpec` \leftarrow `polySpec` `vertex` | `vertex` `vertex` `vertex`

`bitmap` \leftarrow **GL_BITMAP_TOKEN** `vertex`

`pixelRectangle` \leftarrow **GL_DRAW_PIXEL_TOKEN** `vertex` | **GL_COPY_PIXEL_TOKEN** `vertex`

`passThru` \leftarrow **GL_PASS_THROUGH_TOKEN** `value`

`vertex` \leftarrow `2d` | `3d` | `3dColor` | `3dColorTexture` | `4dColorTexture`

`2d` \leftarrow `value` `value`

`3d` \leftarrow `value` `value` `value`

`3dColor` \leftarrow `value` `value` `value` `color`

`3dColorTexture` \leftarrow `value` `value` `value` `color` `tex`

`4dColorTexture` \leftarrow `value` `value` `value` `value` `color` `tex`

`color` \leftarrow `rgba` | `index`

`rgba` \leftarrow `value` `value` `value` `value`

`index` \leftarrow `value`

`tex` \leftarrow `value` `value` `value` `value`

这里`value`是一个浮点数，`n`是一个浮点整数，代表多边形中顶点的数目。**GL_POINT_TOKEN**、**GL_LINE_TOKEN**、**GL_LINE_RESET_TOKEN**、**GL_POLYGON_TOKEN**、**GL_BITMAP_TOKEN**、**GL_DRAW_PIXEL_TOKEN**、**GL_COPY_PIXEL_TOKEN**、**GL_PASS_THROUGH_TOKEN**为浮点型符号常量。当重设线的点画模式时，返回参数**GL_LINE_RESET_TOKEN**。作为一个顶点返回的数据将由反馈`type`决定。

表5-12说明了类型*type*和每个顶点的相关数值。其中*k*分别为1（颜色索引模式）和4（RGBA模式）。

表 5-12

类 型	坐 标	颜 色	纹 理	数 值 的 总 数
GL_2D	<i>x</i> , <i>y</i>			2
GL_3D	<i>x</i> , <i>y</i> , <i>z</i>			3
GL_3D_COLOR	<i>x</i> , <i>y</i> , <i>z</i>	<i>k</i>		3 + <i>k</i>
GL_3D_COLOR_TEXTURE	<i>x</i> , <i>y</i> , <i>z</i>	<i>k</i>	4	7 + <i>k</i>
GL_4D_COLOR_TEXTURE	<i>x</i> , <i>y</i> , <i>z</i> , <i>w</i>	<i>k</i>	4	8 + <i>k</i>

除*w*是剪切坐标外，其他返回顶点的坐标都是窗口坐标。如果已经启动光照模式，反馈颜色将会是亮的。如果已经启动纹理坐标产生模式，则会产生反馈纹理坐标。这些坐标均通过纹理矩阵转换。

- 注意：

当在显示列表中调用函数**glFeedbackBuffer()**时，它并不被编入显示列表而是被立即执行。

当系统支持**GL_ARB_multitexture**扩展时，函数**glFeedbackBuffer()**仅返回纹理单元**GL_TEXTURE0_ARB**的纹理坐标。

- 出错提示：

当参数*type*不是一个不可接受的值时产生**GL_INVALID_ENUM**提示。

当参数*size*是负数时产生**GL_INVALID_VALUE**提示。

当绘图模式是**GL_FEEDBACK**时调用函数**glFeedbackBuffer()**，或者还没有调用函数**glFeedbackBuffer()**就先调用了函数**glRenderMode(GL_FEEDBACK)**，都会产生**GL_INVALID_OPERATION**提示。

当函数**glFeedbackBuffer()**在函数对**glBegin()/glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGet(GL_RENDER_MODE)

glGet(GL_FEEDBACK_BUFFER_POINTER)

glGet(GL_FEEDBACK_BUFFER_SIZE)

glGet(GL_FEEDBACK_BUFFER_TYPE)

- 请参阅：

glBegin(), **glLineStipple()**, **glPassThrough()**, **glPolygonMode()**, **glRenderMode()**, **glSelectBuffer()**

- **glFinish**

- 名称：

glFinish()

- 功能：

在所有GL操作完成之前，产生阻塞。

- C 描述:

void glFinish(void)

- 说明:

函数**glFinish()**并不立即返回，直到所有以前调用的GL命令都已完成后再返回。这些命令包括所有对GL状态的改变，所有对连接状态的改变，和所有对帧缓冲区中内容的改变。

- 注意:

函数**glFinish()**需要与服务器往返通信。

- 出错提示:

当函数**glFinish()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 请参阅:

glFlush()

■ glFlush

- 名称:

glFlush()

- 功能:

在有限时间内强制执行GL命令。

- C 描述:

void glFlush(void)

- 说明:

当在不同的存储单元（包括网络缓冲区和图形加速器本身）中含有不同GL实现的缓冲区命令时，函数**glFlush()**将清空这些缓冲区，使所有已发布的命令尽可能快地执行。即使不能精确地算出执行所用的时间，也可以确定它将在有限时间内完成。

由于一些GL程序将通过网络执行，或通过一个缓冲区指定的加速器工作，所以所有程序都必须使用函数**glFlush()**以使以前调用的命令尽快完成。例如，如果用户的输入依赖于已产生的图形，则在等待用户输入前应调用函数**glFlush()**。

- 注意:

函数**glFlush()**可在任何时候返回。它的返回并不等待所有以前发布的GL命令执行完成。

- 出错提示:

当函数**glFlush()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 请参阅:

glFinish()

■ glFog

- 名称:

glFogf(), **glFogl()**, **glFogfv()**, **glFogiv()**

- 功能:

指定雾参数。

• C描述：

```
void glFogf( GLenum pname,
              GLfloat params )
void glFogi( GLenum pname,
              GLint params )
```

• 参数说明：

pname 指定一个雾参数。它可以是**GL_FOG_MODE**、**GL_FOG_DENSITY**、**GL_FOG_START**、**GL_FOG_END**和**GL_FOG_INDEX**。

params 指定参数*pname*的具体设定值。

• C描述：

```
void glFogfv( GLenum pname,
               const GLfloat *params )
void glFogiv( GLenum pname,
               const GLint *params )
```

• 参数说明：

pname 指定一个雾参数。它可以是**GL_FOG_MODE**、**GL_FOG_DENSITY**、**GL_FOG_START**、**GL_FOG_END**、**GL_FOG_INDEX**和**GL_FOG_COLOR**。

params 指定参数*pname*的一个或多个具体设定值。**GL_FOG_COLOR**需要一个四元数组。其他参数仅需要一个一元数组。

• 说明：

默认情况下，雾化操作是关闭的。当启动雾化操作后，雾化操作将对光栅化的几何体、位图和像素块产生影响，但不影响清除缓冲区的操作。函数**glEnable(GL_FOG)**用来启动雾化操作，函数**Disable(GL_FOG)**用于关闭雾化操作。

函数**glFog()**将把由参数*pname*指定的雾参数放入了参数*params*中。下面列出了参数*pname*的可取值：

GL_FOG_MODE	参数 <i>params</i> 是一个单整型值或浮点值，用来指定计算雾化融合因子 <i>f</i> 的方程。它可接受三个符号常量： GL_LINEAR ， GL_EXP 和 GL_EXP2 。对应于这三个符号常量的方程见下面的有关叙述。缺省的雾化模式为 GL_EXP 。
GL_FOG_DENSITY	参数 <i>params</i> 是一个单整型值或浮点值，用来指定用于双指数雾化方程的雾浓度 <i>density</i> 。雾浓度只取非负数。缺省的雾浓度为1。
GL_FOG_START	参数 <i>params</i> 是一个单整型值或浮点值，用来指定参数 <i>start</i> ，即线性雾化方程中最近的距离。其缺省值是0。
GL_FOG_END	参数 <i>params</i> 是一个单整型值或浮点值，用来指定参数 <i>end</i> ，即线性雾化方程中最远的距离。其缺省值是1。
GL_FOG_INDEX	参数 <i>params</i> 是一个单整型值或一个浮点值，用来指定雾的颜色

索引值*i*。其缺省值是0。

GL_FOG_COLOR

参数`params`包含四个整型值或浮点值，它们用来指定雾的颜色值 C_f 。整型数据被线性地映射：所代表的最大正数映射为1.0，所代表的最小负数映射为-1.0。浮点型数据则被直接映射。转化后所有的颜色组分的取值范围均为[0, 1]。颜色的缺省值是(0, 0, 0, 0)。

雾化就是用一个融合因子 f 把一种雾颜色与每一种光栅化的像素片断的已经纹理化后颜色进行融合。融合因子 f 的计算方法有三种，具体方法由雾化模式决定。设 z 是眼坐标中从原点到要雾化的片断的距离，则，

模式**GL_LINEAR**下的雾化方程为：

$$f = \frac{end - z}{end - start}$$

模式**GL_EXP**下的雾化方程为：

$$f = e^{-(density \cdot z)}$$

模式**GL_EXP2**下的雾化方程为：

$$f = e^{-(density \cdot z)^2}$$

不管雾化模式究竟哪种，计算后的雾化因子 f 都将被截断到范围[0, 1]内。

在RGBA模式下，如果预设的雾化因子是 C_f ，那么片断的红、绿和蓝颜色组分由下式计算：

$$C'_r = f C_r + (1-f) C_f$$

雾化操作不会影响片断的alpha组分。

在颜色索引模式下，片断的颜色索引*i*由下式计算：

$$I'_r = I_r + (1-f) I_f$$

- 出错提示：

当参数`pname`不是个可接受的值，或当参数`pname`是**GL_FOG_MODE**而参数`params`不是一个可接受的值时，产生**GL_INVALID_ENUM**提示。

当参数`pname`是**GL_FOG_DENSITY**而参数`params`是负数时，产生**GL_INVALID_VALUE**提示。

当函数`glFog()`在函数对`glBegin()`/`glEnd()`之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

```
glIsEnabled( GL_FOG )
glGet( GL_FOG_COLOR )
glGet( GL_FOG_INDEX )
glGet( GL_FOG_DENSITY )
glGet( GL_FOG_START )
glGet( GL_FOG_END )
glGet( GL_FOG_MODE )
```

- 请参阅：

glEnable()***glFrontFace**

- 名称：

glFrontFace()

- 功能：

指定正面和反面多边形。

- C 描述：

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

- 参数说明：

mode 指定正面多边形的方向。它可以是**GL_CW**和**GL_CCW**。其缺省值是**GL_CCW**。

- 说明：

一个由不透明的表面所围成的场景，其反面的多边形是不可见的。除去这些不可见的多边形将大大提高图形的绘制速度。函数**glEnable(GL_CULL_FACE)**和**glDisable(GL_CULL_FACE)**用来启动和关闭除去反面多边形的操作。

如果一个假想的对象沿它的第一个顶点出发，顺着第二个顶点，第三个顶点……然后又回到第一个顶点的路径，在多边形的内部沿顺时针方向移动，则多边形在窗口坐标系中的投影被称为沿顺时针方向缠绕；否则称其为沿逆时针方向缠绕。函数**glFrontFace()**指定在窗口坐标系中的正面是指沿顺时针方向缠绕的多边形还是沿逆时针方向缠绕的多边形。参数**GL_CCW**指定沿顺时针方向缠绕的多边形是正面，参数**GL_CW**指定沿逆时针方向缠绕的多边形是正面。缺省值是沿逆时针方向缠绕的多边形作为正面。

- 出错提示：

当参数*mode*不是一个可接受的值时产生**GL_INVALID_ENUM**提示。

当函数**glFrontFace()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGet(GL_FRONT_FACE)

- 请参阅：

glCullFace(), **glLightModel()**

***glFrustum**

- 名称：

glFrustum()

- 功能：

用一个透视矩阵乘以当前的矩阵。

- C 描述：

```
void glFrustum( GLdouble left,
                 GLdouble right,
                 GLdouble bottom,
```

```
GLdouble top,
GLdouble zNear,
GLdouble zFar)
```

• 参数说明：

left, right

指定左、右垂直剪切平面的坐标。

bottom, top

指定下、上水平剪切平面的坐标。

zNear, zFar

指定视点到最近和最远深度剪切平面的距离。二者都必须是正数。

• 说明：

函数glFrustum()描述了一个用来生成透视投影的透视矩阵。当前矩阵（请参阅glMatrixMode()）将与该矩阵相乘，所得的结果将替换当前矩阵。就像调用函数glMultMatrix()时使用下列矩阵作为其自变量：

$$\begin{matrix} \frac{2 \cdot zNear}{right - left} & 0 & A & 0 \\ 0 & \frac{2 \cdot zNear}{top - bottom} & B & 0 \\ 0 & 0 & C & D \\ 0 & 0 & -1 & 0 \end{matrix}$$

$$A = \frac{right + left}{right - left}$$

$$B = \frac{top + bottom}{top - bottom}$$

$$C = -\frac{zFar + zNear}{zFar - zNear}$$

$$D = -\frac{2 \cdot zFar \cdot zNear}{zFar - zNear}$$

通常，矩阵模式是**GL_PROJECTION**，并且(*left, bottom, -zNear*)和(*right, top, -zNear*)分别指定最近的剪切平面上映射到窗口中的左下角和右上角的点。（这时假设视点是在(0, 0, 0)位置。）*-zFar*指定视点到最远深度剪切平面的位置。*zNear*和*zFar*都必须是正数。

可用函数glPushMatrix()和glPopMatrix()来存储和恢复当前的矩阵堆栈。

• 注意：

深度缓冲区的精度将受指定的*zNear*和*zFar*值的影响。*zNear*和*zFar*的比值越大，它对深度缓冲区的影响将越小。当

$$r = \frac{zFar}{zNear}$$

时，大约 $\log_2(r)$ 位的深度缓冲区的精度将被丢失。由于当 $zNear$ 趋近于0时 r 将趋于无穷大，所以 $zNear$ 不能被设置为0。

- 出错提示：

当 $zNear$ 或 $zFar$ 不是正数，或 $left=right$ ，或 $bottom=top$ 时产生**GL_INVALID_VALUE**提示。

当函数**glFrustum()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGet(GL_MATRIX_MODE)
glGet(GL_MODELVIEW_MATRIX)
glGet(GL_PROJECTION_MATRIX)
glGet(GL_TEXTURE_MATRIX)
glGet(GL_COLOR_MATRIX)

- 请参阅：

glOrtho(), **glMatrixMode()**, **glMultMatrix()**, **glPushMatrix()**, **glViewport()**

• **glGenLists**

- 名称：

glGenLists()

- 功能：

建立一组连续的空显示列表。

- C描述：

GLuint glGenLists(GLsizei range)

- 参数说明：

range 指定要产生的连续的空显示列表的数目。

- 说明：

函数**glGenList()**带有一个参数*range*。它返回一个整数*n*，这样名为*n*, *n+1*, ..., *n+range-1*的*range*个连续空显示列表将被建立。如果参数*range*为0，或者没有*range*个连续有效名称的显示列表组存在，或者出现了任何错误；都不会建立显示列表，并返回0。

- 出错提示：

当参数*range*为负数时产生**GL_INVALID_VALUE**提示。

当函数**glGenLists()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glIsList()

- 请参阅：

glCallList(), **glCallLists()**, **glDeleteLists()**, **glNewList()**

• **glGenTextures**

- 名称：

glGenTextures()

- 功能：

生成纹理名称。

- C描述：

```
void glGenTextures( GLsizei n,
                    GLuint *textures )
```

- 参数说明：

n 指定将生成的纹理名称的数目。

textures 指定用来存放生成的纹理名称的数组。

- 说明：

函数glGenTextures()将*n*个纹理名称返回到*textures*中。此处并不能保证所有的名称将构成一个连续的整数集。然而，它却可以保证在调用函数glGenTextures()前所返回的名称中没有一个可以被立刻使用。

生成的纹理没有维数，它们的维数被设定为等于它们第一次绑定到的纹理目标的维数（请参阅函数glBindTexture()）。

调用函数glGenTextures()返回的名称将不能再被随后的调用所返回，除非先用函数glDeleteTextures()将它们删除掉。

- 注意：

函数glGenTextures()在GL 1.1以上的版本中才有效。

- 出错提示：

当参数*n*是负数时产生GL_INVALID_VALUE提示。

当函数glGenTextures()在函数对glBegin()/glEnd()之间执行时产生GL_INVALID_OPERATION提示。

- 有关数据的获取：

glIsTexture()

- 请参阅：

glBindTexture(), glCopyTexImage1D(), glCopyTexImage2D(), glDeleteTextures(), glGet(),
glGetTexParameter(), glTexImage1D(), glTexImage2D(), glTexImage3D(), glTexParameter()

glGet

- 名称：

glGetBooleanv(), glGetDoublev(), glGetFloatv(), glGetIntegerv()

- 功能：

返回所选参数的值。

- C描述：

```
void glGetBooleanv( GLenum pname,
                    GLboolean *params )
```

- C描述：

```
void glGetDoublev( GLenum pname,
                   GLdouble *params )
```

• C 描述:

```
void glGetFloatv( GLenum pname,
                  GLfloat *params )
```

• C 描述:

```
void glGetIntegerv( GLenum pname,
                     GLint *params )
```

• 参数说明:

pname 指定要返回的参数值。它可取的符号常量见下面的介绍。

params 指定参数的返回值。

• 说明:

这四个命令都返回GL的简单状态变量值。参数*pname*是一个符号常量，表示要返回的状态变量。参数*params*是一个指针，指向一个给定类型的数组，用来存放返回的数据。

如果参数*params*的类型与状态变量要求的类型不一样，将发生类型转化。当调用函数glGetBooleanv()时，只有浮点数0.0（或整型数0）转化为GL_FALSE，否则转化为GL_TRUE。当调用函数glGetIntegerv()时，返回的布尔值为GL_TRUE或GL_FALSE，大多数浮点值返回为最接近的整型值。只有浮点颜色值和法线值返回一个线性映射值：1.0映射为所代表的最大正数，-1.0映射为所代表的最小负数。当调用函数glGetFloatv()和glGetDoublev()时，返回布尔值为GL_TRUE或GL_FALSE返回，整型值转化为浮点值。

参数*pname*可以取以下的符号常量：

GL_ACCUM_ALPHA_BITS

参数*params*返回一个值，该值表示在累积缓冲区中alpha组分的位面的数目。

GL_ACCUM_BLUE_BITS

参数*params*返回一个值，该值表示在累积缓冲区中蓝组分的位面的数目。

GL_ACCUM_CLEAR_VALUE

参数*params*返回四个值：红、绿、蓝和alpha值，这四个值用来清空累积缓冲区。如果需要整型值，则将内部浮点值进行如下的线性映射：1.0映射为所代表的最大正整数，-1.0映射为所代表的最小负整数。其缺省值为(0, 0, 0, 0)。请参阅glClearAccum()。

GL_ACCUM_GREEN_BITS

参数*params*返回一个值，该值表示在累积缓冲区中绿组分的位面的数目。

GL_ACCUM_RED_BITS

参数*params*返回一个值，该值表示在累积缓冲区中红组分的位面的数目。

GL_ACTIVE_TEXTURE_ARB

参数*params*返回一个单值，该值表示当前有效的多重纹理单元。其缺省值为GL_TEXTURE0_ARB。请参阅glActiveTextureARB()。

GL_ALIASED_POINT_SIZE_RANGE

参数*params*返回两个值，它们表示所支持的失真点的最小和最大尺寸。

GL_ALIASED_LINE_WIDTH_RANGE

参数*params*返回两个值，它们表示所支持的失真线的最小和最大尺寸。

GL_ALPHA_BIAS

参数*params*返回一个值，该值表示在像素转换过程中所使用的alpha组分的偏移因子。其缺省值是0。请参阅glPixelTransfer()。

GL_ALPHA_BITS

参数*params*返回一个值，该值表示每个颜色缓冲区中alpha位面的数目。

GL_ALPHA_SCALE

参数*params*返回一个值，该值表示在像素转换过程中所使用的alpha组分的缩放因子。其缺省值是1。请参阅glPixelTransfer()。

GL_ALPHA_TEST

参数*params*返回一个boolean值，该值表明是否启动片断的alpha测试功能。其缺省值是**GL_FALSE**。请参阅glAlphaFunc()。

GL_ALPHA_TEST_FUNC

参数*params*返回一个值，该值表示alpha测试函数的符号名称。其缺省值是**GL_ALWAYS**。请参阅glAlphaFunc()。

GL_ALPHA_TEST_REF

参数*params*返回一个值，该值表示用于alpha测试的参考值。其缺省值是0。请参阅glAlphaFunc()。如果需要一个整型值，则将内部浮点数进行如下的线性映射：1.0映射为所代表的最大正整数，-1.0映射为所代表的最小负整数。

GL_ATTRIB_STACK_DEPTH

参数*params*返回一个值，该值表示属性堆栈的深度。如果该堆栈是空的，它将返回0。其缺省值是0。请参阅glPushAttrib()。

GL_AUTO_NORMAL

参数*params*返回一个boolean值，该值表示2D映射求值是否自动生成表面法线。其缺省值是**GL_FALSE**。请参阅glMap2()。

GL_AUX_BUFFERS

参数*params*返回一个值，该值表示辅助颜色缓冲区的数目。其缺省值是0。

GL_BLEND

参数*params*返回一个boolean值，该值表示是否启动融合功能。其缺省值是**GL_FALSE**。请参阅glBlendFunc()。

GL_BLEND_COLOR

参数*params*返回四个值，这些值分别表示融合颜色的红、绿、蓝和alpha组分值。请参阅glBlendColor()。

GL_BLEND_DST

参数*params*返回一个值，该值是一个符号常量，它用来识别目标融合函数。其缺省值是**GL_ZERO**。请参阅glBlendFunc()。

GL_BLEND_EQUATION

参数*params*返回一个值，该值是一个符号常量，它表示融合方

程是**GL_FUNC_ADD**、**GL_MIN**还是**GL_MAX**。请参阅**glBlendEquation()**。

GL_BLEND_SRC

参数*params*返回一个值，该值是一个符号常量，它用来识别源融合函数。其缺省值是**GL_ONE**。请参阅**glBlendFunc()**。

GL_BLUE_BIAS

参数*params*返回一个值，该值表示在像素转换过程中所使用的蓝组分的偏移因子。其缺省值是0。请参阅**glPixelTransfer()**。

GL_BLUE_BITS

参数*params*返回一个值，该值表示每个颜色缓冲区中蓝组分的位面的数目。

GL_BLUE_SCALE

参数*params*返回一个值，该值表示在像素转换过程中所使用的蓝组分的缩放因子。其缺省值是1。请参阅**glPixelTransfer()**。

GL_CLIENT_ACTIVE_TEXTURE_ARB

参数*params*返回一个单值，该值是一个整型量，它表示当前客户端有效的多重纹理单元。其缺省值是**GL_TEXTURE0_ARB**。请参阅**glClientActiveTextureARB()**。

GL_CLIENT_ATTRIB_STACK_DEPTH

参数*params*返回一个值，该值用来确定属性堆栈的深度。其缺省值是0。请参阅**glPushClientAttrib()**。

GL_CLIP_PLANE*i*

参数*params*返回一个boolean值，该值表示指定的剪切平面是否已被启动。其缺省值是**GL_FALSE**。请参阅**glClipPlane()**。

GL_COLOR_ARRAYV

参数*params*返回一个boolean值，该值表示是否启动颜色数组功能。其缺省值是**GL_FALSE**。请参阅**glColorPointer()**。

GL_COLOR_ARRAY_SIZE

参数*params*返回一个值，该值表示颜色数组中每个颜色的组分数。其缺省值是4。请参阅**glColorPointer()**。

GL_COLOR_ARRAY_STRIDE

参数*params*返回一个值，该值表示颜色数组中相邻颜色间的字节偏移量。其缺省值是0。请参阅**glColorPointer()**。

GL_COLOR_ARRAY_TYPE

参数*params*返回一个值，该值表示颜色数组中每个组分的数据类型。其缺省值是**GL_FLOAT**。请参阅**glColorPointer()**。

GL_COLOR_CLEAR_VALUE

参数*params*返回四个值，这些值分别表示用于清除颜色缓冲区的红、绿、蓝和alpha值。如果需要整型值，则将内部浮点值进行如下的线性映射：1.0映射为所代表的最大正整数，-1.0映射为所代表的最小负整数。其缺省值是(0, 0, 0, 0)。请参阅**glClearColor()**。

GL_COLOR_LOGIC_OP

参数*params*返回一个boolean值，该值表示是否用一个逻辑操作将一个片断的RGBA颜色值并入帧缓冲区中。其缺省值是**GL_FALSE**。请参阅glLogicOp()。

GL_COLOR_MATERIAL

参数*params*返回一个boolean值，该值表示是否用一个或多个材质参数跟踪当前颜色。其缺省值是**GL_FALSE**。请参阅glColorMaterial()。

GL_COLOR_MATERIAL_FACE

参数*params*返回一个值，该值是一个符号常量，表示哪个材质带有一个跟踪当前颜色的参数。其缺省值是**GL_FRONT_AND_BACK**。请参阅glColorMaterial()。

GL_COLOR_MATERIAL_PARAMETER

参数*params*返回一个值，该值是一个符号常量，它表示将用哪个材质参数跟踪当前颜色。其缺省值是**GL_AMBIENT_AND_DIFFUSE**。请参阅glColorMaterial()。

GL_COLOR_MATRIX

参数*params*返回六个值，这些值表示位于颜色矩阵堆栈栈顶的颜色矩阵。其缺省值是单位矩阵。请参阅glPushMatrix()。

GL_COLOR_MATRIX_STACK_DEPTH

参数*params*返回一个值，该值表示投影矩阵堆栈所支持的最大深度值。该值至少应是2。请参阅glPushMatrix()。

GL_COLOR_TABLE

参数*params*返回一个boolean值，该值表示是否启动颜色表查询功能。请参阅glColorTable()。

GL_COLOR_WRITEMASK

参数*params*返回四个boolean值，这些值表示是否向颜色缓冲区写入红、绿、蓝和alpha值。其缺省值是(**GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE**)。请参阅glColorMask()。

GL_CONVOLUTION_1D

参数*params*返回一个boolean值，该值表示是否启动1D卷积功能。其缺省值是**GL_FALSE**。请参阅glConvolutionFilter1D()。

GL_CONVOLUTION_2D

参数*params*返回一个boolean值，该值表示是否启动2D卷积功能。其缺省值是**GL_FALSE**。请参阅glConvolutionFilter2D()。

GL_CULL_FACE

参数*params*返回一个boolean值，该值表示是否启动多边形拣选功能。其缺省值是**GL_FALSE**。请参阅glCullFace()。

GL_CULL_FACE_MODE

参数*params*返回一个值，该值是一个符号常量，它表示哪些多边形将被拣选掉。其缺省值是**GL_BACK**。请参阅glCullFace()。

GL_CURRENT_COLOR

参数*params*返回四个值，这些值分别表示当前颜色的红、绿、蓝和alpha组分。如果需要整型值，则将内部浮点值进行如下的线性映射：1.0映射为所表示的最大正整数，-1.0映射为所表示的最小负整数。其缺省值是(1, 1, 1, 1)。请参阅glColor()。

GL_CURRENT_INDEX 参数*params*返回一个值，该值表示当前的颜色索引值。其缺省值是1。请参阅glIndex()。

GL_CURRENT_NORMAL

参数*params*返回三个值，这些值分别表示当前法线的x、y和z值。如果需要整型值，则将内部浮点值进行以下的线性映射：1.0映射为所表示的最大正整数，-1.0映射为所表示的最小负整数。其缺省值是(0, 0, 1)。请参阅glNormal()。

GL_CURRENT_RASTER_COLOR

参数*params*返回四个值，这些值分别表示当前光栅位置的红、绿、蓝和alpha组分。如果需要整型值，则将内部浮点值进行以下的线性映射：1.0映射为所表示的最大正整数，-1.0映射为所表示的最小负整数。其缺省值是(1, 1, 1, 1)。请参阅glRasterPos()。

GL_CURRENT_RASTER_DISTANCE

参数*params*返回一个值，该值表示从视点到当前光栅位置的距离。其缺省值是0。请参阅glRasterPos()。

GL_CURRENT_RASTER_INDEX

参数*params*返回一个值，该值表示当前光栅位置的颜色索引值。其缺省值是1。请参阅glRasterPos()。

GL_CURRENT_RASTER_POSITION

参数*params*返回四个值，这些值分别表示当前光栅位置的x、y、z和w组分。其中x、y和z表示窗口坐标，w表示剪切坐标。其缺省值是(0, 0, 0, 1)。请参阅glRasterPos()。

GL_CURRENT_RASTER_POSITION_VALID

参数*params*返回一个boolean值，该值表示当前光栅位置是否有效。其缺省值是GL_TRUE。请参阅glRasterPos()。

GL_CURRENT_RASTER_TEXTURE_COORDS

参数*params*返回四个值，这些值分别表示当前的纹理坐标s、t、r和q。其缺省值是(0, 0, 0, 1)。请参阅glTexCoord()。

GL_DEPTH_BIAS 参数*params*返回一个值，该值表示在像素转换过程中所使用的深度偏移因子。其缺省值是0。请参阅glPixelTransfer()。

GL_DEPTH_BITS 参数*params*返回一个值，该值表示每个深度缓冲区中位面的数目。

GL_DEPTH_CLEAR_VALUE

参数*params*返回一个值，该值用来清除深度缓冲区。如果需要整型值，则将内部浮点值进行如下的线性映射：1.0映射为所表示的最大正整数，-1.0映射为所表示的最小负整数。其缺省值是1。请参阅glClearDepth()。

GL_DEPTH_FUNC

参数*params*返回一个值，该值是一个表示深度比较函数的符号常量。其缺省值是**GL_LESS**。请参阅glDepthFunc()。

GL_DEPTH_RANGE

参数*params*返回两个值，它们分别表示深度缓冲区的最近和最远的映射极限值。如果需要整型值，则将内部浮点值进行如下的线性映射：1.0映射为所代表的最大正整数，-1.0映射为所代表的最小负整数。其缺省值是(0, 1)。请参阅glDepthRange()。

GL_DEPTH_SCALE

参数*params*返回一个值，该值表示在像素转换过程中所使用的深度缩放因子。其缺省值是1。请参阅glPixelTransfer()。

GL_DEPTH_TEST

参数*params*返回一个boolean值，该值表明是否启动片断的深度测试功能。其缺省值是**GL_FALSE**。请参阅glDepthFunc()和glDepthRange()。

GL_DEPTH_WRITEMASK

参数*params*返回一个boolean值，该值表示深度缓冲区是否允许写入。其缺省值是**GL_TRUE**。请参阅glDepthMask()。

GL_DITHER

参数*params*返回一个boolean值，该值表示是否允许对片断颜色和索引进行抖动操作。其缺省值是**GL_TRUE**。

GL_DOUBLEBUFFER

参数*params*返回一个boolean值，该值表示是否支持双缓冲区。

GL_DRAW_BUFFER

参数*params*返回一个值，该值是一个符号常量，它表示将绘入哪个缓冲区。请参阅glDrawBuffer()。当后缓冲区存在时，其缺省值是**GL_BACK**。否则，其缺省值是**GL_FRONT**。

GL_EDGE_FLAG

参数*params*返回一个boolean值，该值说明当前边界标志是**GL_TRUE**还是**GL_FALSE**。其缺省值是**GL_TRUE**。请参阅glEdgeFlag()。

GL_EDGE_FLAG_ARRAY

参数*params*返回一个boolean值，该值表示是否已启动边界标志数组。其缺省值是**GL_FALSE**。请参阅glEdgeFlagPointer()。

GL_EDGE_FLAG_ARRAY_STRIDE

参数*params*返回一个值，该值表示边界标志数组中相邻边界标志之间的字节偏移量。其缺省值是0。请参阅glEdgeFlagPointer()。

GL_FEEDBACK_BUFFER_SIZE

参数*params*返回一个值，该值表示反馈缓冲区的尺寸。请参阅

`glFeedbackBuffer()`。

GL_FEEDBACK_BUFFER_TYPE

参数*params*返回一个值，该值表示反馈缓冲区的类型。请参阅`glFeedbackBuffer()`。

GL_FOG

参数*params*返回一个boolean值，该值表示是否启动雾化功能。其缺省值是**GL_FALSE**。请参阅`glFog()`。

GL_FOG_COLOR

参数*params*返回四个值，这些值分别表示雾颜色的红、绿、蓝和alpha组分。如果需要整型值，则将内部浮点值将进行如下的线性映射：1.0映射为所代表的最大正整数，-1.0映射为所代表的最小负整数。其缺省值是(0, 0, 0, 0)。请参阅`glFog()`。

GL_FOG_DENSITY

参数*params*返回一个值，该值表示雾的浓度参数。其缺省值是1。请参阅`glFog()`。

GL_FOG_END

参数*params*返回一个值，该值表示线性雾化方程的结束因子。其缺省值是1。请参阅`glFog()`。

GL_FOG_HINT

参数*params*返回一个值，该值是一个符号常量，它表示雾化提示的模式。其缺省值是**GL_DONT_CARE**。请参阅`glHint()`。

GL_FOG_INDEX

参数*params*返回一个值，该值表示雾的颜色索引。其缺省值是0。请参阅`glFog()`。

GL_FOG_MODE

参数*params*返回一个值，该值是一个符号常量，它表示哪个雾化方程将被选择。其缺省值是**GL_EXP**。请参阅`glFog()`。

GL_FOG_START

参数*params*返回一个值，该值表示线性雾化方程的开始因子。其缺省值是0。请参阅`glFog()`。

GL_FRONT_FACE

参数*params*返回一个值，该值是一个符号常量，它表示是将顺时针缠绕的多边形作为正面多边形还是将逆时针缠绕的多边形作为正面多边形。其缺省值是**GL_CCW**。请参阅`glFrontFace()`。

GL_GREEN_BIAS

参数*params*返回一个值，该值表示在像素转换过程中所使用的绿组分的偏移因子。其缺省值是0。

GL_GREEN_BITS

参数*params*返回一个值，该值表示每个颜色缓冲区中绿组分的位面的数目。

GL_GREEN_SCALE

参数*params*返回一个值，该值表示在像素转换过程中所使用的绿组分的缩放因子。其缺省值是1。请参阅`glPixelTransfer()`。

GL_HISTOGRAM

参数*params*返回一个boolean值，该值表示是否启动直方图功能。其缺省值是**GL_FALSE**。请参阅`glHistogram()`。

GL_INDEX_ARRAY

参数*params*返回一个boolean值，该值表示是否启动颜色索引数组。其缺省值是**GL_FALSE**。请参阅`glIndexPointer()`

GL_INDEX_ARRAY_STRIDE

参数*params*返回一个值，该值表示颜色索引数组中相邻颜色索引

间的字节偏移量。其缺省值是0。请参阅glIndexPointer()。

GL_INDEX_ARRAY_TYPE

参数*params*返回一个值，该值表示索引颜色数组中每个组分的数据类型。其缺省值是**GL_FLOAT**。请参阅glIndexPointer()。

GL_INDEX_BITS

参数*params*返回一个值，该值表示每个颜色索引缓冲区中位面的数目。

GL_INDEX_CLEAR_VALUE

参数*params*返回一个值，该值表示用于清除颜色索引缓冲区的颜色索引值。其缺省值是0。请参阅glClearIndex()。

GL_INDEX_LOGIC_OP

参数*params*返回一个boolean值，该值表示是否用一个逻辑操作将一个片断的颜色索引值并入帧缓冲区中。其缺省值是**GL_FALSE**。请参阅glLogicOp()。

GL_INDEX_MODE

参数*params*返回一个boolean值，该值表示GL是处于颜色索引模式 (**GL_TRUE**) 还是RGBA模式 (**GL_FALSE**)。

GL_INDEX_OFFSET

参数*params*返回一个boolean值，该值表示在像素转换过程中加到颜色和模板索引上的偏移量。其缺省值是0。请参阅glPixelTransfer()。

GL_INDEX_SHIFT

参数*params*返回一个boolean值，该值表示在像素转换过程中颜色和模板索引被移位的数量。其缺省值是0。请参阅glPixelTransfer()。

GL_INDEX_WRITEMASK

参数*params*返回一个值，该值是一个屏蔽值，它表示每个颜色索引缓冲区中的哪个位面将被写入。其缺省值是全1的。请参阅glIndexMask()。

GL_LIGHT*i*

参数*params*返回一个boolean值，该值表示是否启动了指定的光源。其缺省值是**GL_FALSE**。请参阅glLight()和glLightModel()。

GL_LIGHTING

参数*params*返回一个boolean值，该值表示是否启动光照功能。其缺省值是**GL_FALSE**。请参阅glLightModel()。

GL_LIGHT_MODEL_AMBIENT

参数*params*返回四个值，它们分别表示全景的环境光强的红、绿、蓝和alpha组分。如果需要整型值，则将内部浮点值进行以下的线性映射：1.0映射为所代表的最大正整数，-1.0映射为所代表的最小负整数。其缺省值是(0.2, 0.2, 0.2, 1.0)。请参阅glLightModel()。

GL_LIGHT_MODEL_COLOR_CONTROL

参数*params*返回一个boolean值，该值表示镜面反射计算是否将从一般的光照计算中分离出来。其缺省值是**GL_SINGLE_COLOR**。

GL_LIGHT_MODEL_LOCAL_VIEWER

参数*params*返回一个boolean值，该值表示在镜面反射计算时是否将视点放入场景中。其缺省值是**GL_FALSE**。请参阅glLightModel()。

GL_LIGHT_MODEL_TWO_SIDE

参数*params*返回一个boolean值，该值表示是否用各自的材料计算正面和反面多边形的光照。其缺省值是**GL_FALSE**。请参阅glLightModel()。

GL_LINE_SMOOTH

参数*params*返回一个boolean值，该值表示是否启动线的反走样功能。其缺省值是**GL_FALSE**。请参阅glLineWidth()。

GL_LINE_SMOOTH_HINT

参数*params*返回一个值，该值是一个符号常量，它表示线的反走样提示的模式。其缺省值是**GL_DONT_CARE**。请参阅glHint()。

GL_LINE_STIPPLE

参数*params*返回一个boolean值，该值表示是否启动线的点画绘图功能。其缺省值是**GL_FALSE**。请参阅glLineStipple()。

GL_LINE_STIPPLE_PATTERN

参数*params*返回一个值，该值表示16位线的点画模式。其缺省值是全1的。请参阅glLineStipple()。

GL_LINE_STIPPLE_REPEAT

参数*params*返回一个值，该值表示线的点画重复因子。其缺省值是1。请参阅glLineStipple()。

GL_LINE_WIDTH

参数*params*返回一个值，该值表示由函数glLineWidth()指定的线段宽度。其缺省值是1。

GL_LINE_WIDTH_GRANULARITY

参数*params*返回一个值，该值表示反走样线所支持的相邻宽度之间的宽度差别。请参阅glLineWidth()。

GL_LINE_WIDTH_RANGE

参数*params*返回两个值，它们分别表示反走样线所支持的最小和最大线宽。请参阅glLineWidth()。

GL_LIST_BASE

参数*params*返回一个值，该值表示加到提供给函数glCallLists()的数组中的所有名称上的基础偏移。其缺省值是0。请参阅glListBase()。

GL_LIST_INDEX

参数*params*返回一个值，该值表示当前结构中的显示列表的名称。如果当前结构中没有显示列表，它将返回0。其缺省值是0。请参阅glNewList()。

GL_LIST_MODE

参数*params*返回一个值，该值是一个符号常量，它表示当前结构中显示列表的结构模式。其缺省值是0。请参阅glNewList()。

GL_LOGIC_OP_MODE

参数*params*返回一个值，该值是一个符号常量，它表示所选择的

	逻辑操作模式。其缺省值是 GL_COPY 。请参阅glLogicOp()。
GL_MAP1_COLOR_4	参数 <i>params</i> 返回一个boolean值，该值表示是否用1D求值器生成颜色。其缺省值是 GL_FALSE 。请参阅glMap1()。
GL_MAP1_GRID_DOMAIN	参数 <i>params</i> 返回两个值，它们表示1D映射的网格区域的端点。其缺省值是(0, 1)。请参阅glMapGrid()。
GL_MAP1_GRID_SEGMENTS	参数 <i>params</i> 返回一个值，它表示1D映射的网格区域中分块的数目。其缺省值是1。请参阅glMapGrid()。
GL_MAP1_INDEX	参数 <i>params</i> 返回一个boolean值，该值表示是否用1D求值器生成颜色索引。其缺省值是 GL_FALSE 。请参阅glMap1()。
GL_MAP1_NORMAL	参数 <i>params</i> 返回一个boolean值，该值表示是否用1D求值器生成法线。其缺省值是 GL_FALSE 。请参阅glMap1()。
GL_MAP1_TEXTURE_COORD_1	参数 <i>params</i> 返回一个boolean值，该值表示是否用1D求值器生成1D纹理坐标。其缺省值是 GL_FALSE 。请参阅glMap1()。
GL_MAP1_TEXTURE_COORD_2	参数 <i>params</i> 返回一个boolean值，该值表示是否用1D求值器生成2D纹理坐标。其缺省值是 GL_FALSE 。请参阅glMap1()
GL_MAP1_TEXTURE_COORD_3	参数 <i>params</i> 返回一个boolean值，该值表示是否用1D求值器生成3D纹理坐标。其缺省值是 GL_FALSE 。请参阅glMap1()。
GL_MAP1_TEXTURE_COORD_4	参数 <i>params</i> 返回一个boolean值，该值表示是否用1D求值器生成4D纹理坐标。其缺省值是 GL_FALSE 。请参阅glMap1()。
GL_MAP1_VERTEX_3	参数 <i>params</i> 返回一个boolean值，该值表示是否用1D求值器生成3D顶点坐标。其缺省值是 GL_FALSE 。请参阅glMap1()。
GL_MAP1_VERTEX_4	参数 <i>params</i> 返回一个boolean值，该值表示是否用1D求值器生成4D顶点坐标。其缺省值是 GL_FALSE 。请参阅glMap1()。
GL_MAP2_COLOR_4	参数 <i>params</i> 返回一个boolean值，该值表示是否用2D求值器生成颜色。其缺省值是 GL_FALSE 。请参阅glMap2()。
GL_MAP2_GRID_DOMAIN	参数 <i>params</i> 返回四个值，它们分别表示2D映射的 <i>i</i> 和 <i>j</i> 网格区域的端点。其缺省值是(0, 1; 0, 1)。请参阅glMapGrid()。
GL_MAP2_GRID_SEGMENTS	参数 <i>params</i> 返回两个值，它们分别表示2D映射的 <i>i</i> 和 <i>j</i> 网格区域中分块的数目。其缺省值是(1, 1)。请参阅glMapGrid()。

GL_MAP2_INDEX	参数 <i>params</i> 返回一个boolean值，该值表示是否用2D求值器生成颜色索引。其缺省值是 GL_FALSE 。请参阅glMap2()。
GL_MAP2_NORMAL	参数 <i>params</i> 返回一个boolean值，该值表示是否用2D求值器生成法线。其缺省值是 GL_FALSE 。请参阅glMap2()。
GL_MAP2_TEXTURE_COORD_1	参数 <i>params</i> 返回一个boolean值，该值表示是否用2D求值器生成1D纹理坐标。其缺省值是 GL_FALSE 。请参阅glMap2()。
GL_MAP2_TEXTURE_COORD_2	参数 <i>params</i> 返回一个boolean值，该值表示是否用2D求值器生成2D纹理坐标。其缺省值是 GL_FALSE 。请参阅glMap2()。
GL_MAP2_TEXTURE_COORD_3	参数 <i>params</i> 返回一个boolean值，该值表示是否用2D求值器生成3D纹理坐标。其缺省值是 GL_FALSE 。请参阅glMap2()。
GL_MAP2_TEXTURE_COORD_4	参数 <i>params</i> 返回一个boolean值，该值表示是否用2D求值器生成4D纹理坐标。其缺省值是 GL_FALSE 。请参阅glMap2()。
GL_MAP2_VERTEX_3	参数 <i>params</i> 返回一个boolean值，该值表示是否用2D求值器生成3D顶点坐标。其缺省值是 GL_FALSE 。请参阅glMap2()。
GL_MAP2_VERTEX_4	参数 <i>params</i> 返回一个boolean值，该值表示是否用2D求值器生成4D顶点坐标。其缺省值是 GL_FALSE 。请参阅glMap2()。
GL_MAP_COLOR	参数 <i>params</i> 返回一个boolean值，该值表示在像素转换过程中是否用查询表替换颜色和颜色索引。其缺省值是 GL_FALSE 。请参阅glPixelTransfer()。
GL_MAP_STENCIL	参数 <i>params</i> 返回一个boolean值，该值表示在像素转换过程中是否用查询表替换模板索引。其缺省值是 GL_FALSE 。请参阅glPixelTransfer()。
GL_MATRIX_MODE	参数 <i>params</i> 返回一个值，该值是一个符号常量，它表示哪个矩阵堆栈是当前所有矩阵操作的目标堆栈。其缺省值是 GL_MODELVIEW 。请参阅glMatrixMode()。
GL_MAX_3D_TEXTURE_SIZE	参数 <i>params</i> 返回一个值，该值表示GL所能处理的最大的3D纹理的一个粗略估计值。在GL 1.2以上版本中可以用 GL_PROXY_TEXTURE_3D 来确定一个纹理是否太大。请参阅glTexImage3D()。
GL_MAX_CLIENT_ATTRIB_STACK_DEPTH	参数 <i>Params</i> 返回一个值，该值表示客户端属性堆栈所支持的最大深度。请参阅glPushClientAttrib()。
GL_MAX_ATTRIB_STACK_DEPTH	

参数*params*返回一个值，该值表示属性堆栈的最大支持深度。该值至少是16。请参阅glPushAttrib()。

GL_MAX_CLIP_PLANES

参数*params*返回一个值，该值表示由应用所定义的剪切平面的最大值。该值至少是6。请参阅glClipPlane()。

GL_MAX_COLOR_MATRIX_STACK_DEPTH

参数*params*返回一个值，该值表示颜色矩阵堆栈的最大支持深度。该值至少是2。请参阅glPushMatrix()。

GL_MAX_ELEMENTS_INDICES

参数*params*返回一个值，该值表示顶点数组中索引的最大推荐值。请参阅glDrawRangeElements()。

GL_MAX_ELEMENTS_VERTICES

参数*params*返回一个值，该值表示顶点数组中顶点的最大推荐值。请参阅glDrawRangeElements()。

GL_MAX_EVAL_ORDER

参数*params*返回一个值，该值表示1D和2D求值器支持的方程的最大阶次。该值至少是8。请参阅glMap1O和glMap2O。

GL_MAX_LIGHTS

参数*params*返回一个值，该值表示光源的最大数目。该值至少是8。请参阅glLight()。

GL_MAX_LIST_NESTING

参数*params*返回一个值，该值表示显示列表横向递归（遍历）时所允许的最大递归深度。该值至少是64。请参阅glCallList()。

GL_MAX_MODELVIEW_STACK_DEPTH

参数*params*返回一个值，该值表示模式取景矩阵堆栈的最大支持深度。该值至少是32。请参阅glPushMatrix()。

GL_MAX_NAME_STACK_DEPTH

参数*params*返回一个值，该值表示选择名称堆栈的最大支持深度。该值至少是64。请参阅glPushName()。

GL_MAX_PIXEL_MAP_TABLE

参数*params*返回一个值，该值表示glPixelMap()查询表的最大支持尺寸。该值至少是32。请参阅glPixelMap()。

GL_MAX_PROJECTION_STACK_DEPTH

参数*params*返回一个值，该值表示投影矩阵堆栈的最大支持深度。该值至少是2。请参阅glPushMatrix()。

GL_MAX_TEXTURE_SIZE

参数*params*返回一个值，该值表示GL所能处理的最大纹理的一个粗略估计值。在GL1.1以上版本中可以用**GL_PROXY_**

TEXUTRE_1D或GL_PROXY_TEXTURE_2D来确定一个纹理是否太大。请参阅glTexImage1D()和glTexImage2D()。

GL_MAX_TEXTURE_STACK_DEPTH

参数*params*返回一个值，该值表示纹理矩阵堆栈的最大支持深度。该值至少是2。请参阅glPushMatrix()。

GL_MAX_TEXTURE_UNITS_ARB

参数*params*返回一个值，该值表示所支持的纹理单元的数目。该值至少是1。请参阅glActiveTextureARB()。

GL_MAX_VIEWPORT_DIMS

参数*params*返回两个值，它们分别表示视口的最大支持宽度和高度。它们至少应该等于将用于绘图的显示器的可视尺寸。请参阅glViewport()。

GL_MINMAX

参数*params*返回一个boolean值，该值表示是否计算像素的极(最大和最小)值。其缺省值是GL_FALSE。请参阅glMinmax()。

GL_MODELVIEW_MATRIX

参数*params*返回16个值，它们表示位于模式取景矩阵堆栈栈顶的模式取景矩阵堆栈。缺省情况下，该矩阵的单位阵。请参阅glPushMatrix()。

GL_MODELVIEW_STACK_DEPTH

参数*params*返回一个值，该值表示在模式取景矩阵堆栈中矩阵的数目。其缺省值是1。请参阅glPushMatrix()。

GL_NAME_STACK_DEPTH

参数*params*返回一个值，该值表示在选择名称堆栈中名称的数目。其缺省值是0。请参阅glPushName()。

GL_NORMAL_ARRAY

参数*params*返回一个boolean值，该值表示是否启动了法线数组。其缺省值是GL_FALSE。请参阅glNormalPointer()。

GL_NORMAL_ARRAY_STRIDE

参数*params*返回一个值，该值表示法线数组中相邻法线间的字节偏移量。其缺省值是0。请参阅glNormalPointer()。

GL_NORMAL_ARRAY_TYPE

参数*params*返回一个值，该值表示法线数组中每个坐标的的数据类型。其缺省值是GL_FLOAT。请参阅glNormalPointer()。

GL_NORMALIZE

参数*params*返回一个boolean值，该值表示当法线被转换到眼坐标后是否自动将其缩放成单位长度。其缺省值是GL_FALSE。请参阅glNormal()

GL_PACK_ALIGNMENT

参数*params*返回一个值，该值表示当把像素数据写入内存时使用的字节对齐方式。其缺省值是4。请参阅glPixelStore()。

GL_PACK_IMAGE_HEIGHT

参数*params*返回一个值，该值表示当把像素数据写入内存时使用的图像高度。其缺省值是0。请参阅glPixelStore()。

GL_PACK_LSB_FIRST 参数*params*返回一个boolean值，该值表示当把单个二进制位的像素写入内存中时第一个像素是否从每个无符号的字节最次要的位开始。其缺省值是GL_FALSE。请参阅glPixelStore()。

GL_PACK_ROW_LENGTH

参数*params*返回一个值，该值表示当把像素数据写入内存时使用的行的长度。其缺省值是0。请参阅glPixelStore()。

GL_PACK_SKIP_IMAGE

参数*params*返回一个值，该值表示当向内存中写入第一个像素前所跳过的像素图像的数目。其缺省值是0。请参阅glPixelStore()。

GL_PACK_SKIP_PIXELS

参数*params*返回一个值，该值表示当向内存中写入第一个像素前所跳过的像素存储单元的数目。其缺省值是0。请参阅glPixelStore()。

GL_PACK_SKIP_ROWS

参数*params*返回一个值，该值表示当向内存中写入第一个像素时所跳过的像素存储单元的行数。其缺省值是0。请参阅glPixelStore()。

GL_PACK_SWAP_BYTES

参数*params*返回一个boolean值，该值表示当把像素写入内存前是否将双字节和四字节的像素索引和组分交换。其缺省值是GL_FALSE。请参阅glPixelStore()。

GL_PERSPECTIVE_CORRECTION_HINT

参数*params*返回一个值，该值是一个符号常量，它表示透视修正提示的模式。其缺省值是GL_DONT_CARE。请参阅glHint()。

GL_PIXEL_MAP_A_TO_A_SIZE

参数*params*返回一个值，该值表示像素的alpha→alpha组分转换表的尺寸。其缺省值是1。请参阅glPixelMap()。

GL_PIXEL_MAP_B_TO_B_SIZE

参数*params*返回一个值，该值表示像素的蓝组分→蓝组分转换表的尺寸。其缺省值是1。请参阅glPixelMap()。

GL_PIXEL_MAP_G_TO_G_SIZE

参数*params*返回一个值，该值表示像素的绿组分→绿组分转换表的尺寸。其缺省值是1。请参阅glPixelMap()。

GL_PIXEL_MAP_I_TO_A_SIZE

参数*params*返回一个值，该值表示像素的索引→alpha组分转换表的尺寸。其缺省值是1。请参阅glPixelMap()。

GL_PIXEL_MAP_I_TO_B_SIZE

参数*params*返回一个值，该值表示像素的索引→蓝组分转换表的尺寸。其缺省值是1。请参阅glPixelMap()。

GL_PIXEL_MAP_I_TO_G_SIZE

参数*params*返回一个值，该值表示像素的索引→绿组分转换表的尺寸。其缺省值是1。请参阅glPixelMap()。

GL_PIXEL_MAP_I_TO_I_SIZE

参数*params*返回一个值，该值表示像素的索引→索引转换表的尺寸。其缺省值是1。请参阅glPixelMap()。

GL_PIXEL_MAP_I_TO_R_SIZE

参数*params*返回一个值，该值表示像素的索引→红组分转换表的尺寸。其缺省值是1。请参阅glPixelMap()。

GL_PIXEL_MAP_R_TO_R_SIZE

参数*params*返回一个值，该值表示像素的红组分→红组分转换表的尺寸。其缺省值是1。请参阅glPixelMap()。

GL_PIXEL_MAP_S_TO_S_SIZE

参数*params*返回一个值，该值表示由像素的模板→模板转换表的尺寸。其缺省值是1。请参阅glPixelMap()。

GL_POINT_SIZE

参数*params*返回一个值，该值表示由函数glPointSize()所指定的点的尺寸。其缺省值是1。

GL_POINT_SIZE_GRANULARITY

参数*params*返回一个值，该值表示反走样点所支持的相邻尺寸之间的尺寸差值。请参阅glPointSize()。

GL_POINT_SIZE_RANGE

参数*params*返回两个值，它们分别表示反走样点的最小支持和最大支持值。这里，最小值不能大于1，最大值不能小于1。请参阅glPointSize()。

GL_POINT_SMOOTH

参数*params*返回一个boolean值，该值表示是否启动点的反走样功能。其缺省值是**GL_FALSE**。请参阅glPointSize()。

GL_POINT_SMOOTH_HINT

参数*params*返回一个值，该值是一个符号常量，它表示点反走样提示的模式。其缺省值是**GL_DONT_CARE**。请参阅

glHint()。

GL_POLYGON_MODE 参数*params*返回两个值，它们都是符号常量，用来表示是否将正面和反面多边形光栅化成点、线或填充多边形。其缺省值是**GL_FILL**。请参阅**glPolygonMode()**。

GL_POLYGON_OFFSET_FACTOR

参数*params*返回一个值，该值是一个缩放因子，它将确定光栅化一个多边形时加到所产生的每个片断的深度值上的偏移量。其缺省值是0。请参阅**glPolygonOffset()**。

GL_POLYGON_OFFSET_UNITS

参数*params*返回一个值，该值在光栅化一个多边形时将乘上一个由具体实现指定的值，然后加到所产生的每个片断的深度值上。其缺省值是0。请参阅**glPolygonOffset()**。

GL_POLYGON_OFFSET_FILL

参数*params*返回一个boolean值，该值表示在多边形的填充模式中偏移量是否有效。其缺省值是**GL_FALSE**。请参阅**glPolygonOffset()**。

GL_POLYGON_OFFSET_LINE

参数*params*返回一个boolean值，该值表示在多边形的线画模式中偏移量是否有效。其缺省值是**GL_FALSE**。请参阅**glPolygonOffset()**。

GL_POLYGON_OFFSET_POINT

参数*params*返回一个boolean值，该值表示在多边形的点画模式中偏移量是否有效。其缺省值是**GL_FALSE**。请参阅**glPolygonOffset()**。

GL_POLYGON_SMOOTH

参数*params*返回一个boolean值，该值表示是否启动多边形的反走样功能。其缺省值是**GL_FALSE**。请参阅**glPolygonMode()**。

GL_POLYGON_SMOOTH_HINT

参数*params*返回一个值，该值是一个符号常量，它表示多边形反走样提示的模式。其缺省值是**GL_DONT_CARE**。请参阅**glHint()**。

GL_POLYGON_STIPPLE

参数*params*返回一个boolean值，该值表示是否启动多边形的点画绘制功能。其缺省值是**GL_FALSE**。请参阅**glPolygonStipple()**。

GL_POST_COLOR_MATRIX_COLOR_TABLE

参数*params*返回一个boolean值，该值表示是否启动随后的颜色矩

阵转换查询功能。其缺省值是**GL_FALSE**。请参阅**glColorTable()**

GL_POST_COLOR_MATRIX_RED_BIAS

参数*params*返回一个值，该值表示在颜色矩阵转换操作之后应用于RGBA片断的红组分偏移因子。其缺省值是0。请参阅**glPixelTransfer()**

GL_POST_COLOR_MATRIX_GREEN_BIAS

参数*params*返回一个值，该值表示在颜色矩阵转换操作之后应用于RGBA片断的绿组分偏移因子。其缺省值是0。请参阅**glPixelTransfer()**。

GL_POST_COLOR_MATRIX_BLUE_BIAS

参数*params*返回一个值，该值表示在颜色矩阵转换操作之后应用于RGBA片断的蓝组分偏移因子。其缺省值是0。请参阅**glPixelTransfer()**。

GL_POST_COLOR_MATRIX_ALPHA_BIAS

参数*params*返回一个值，该值表示在颜色矩阵转换操作之后应用于RGBA片断的alpha组分偏移因子。其缺省值是0。请参阅**glPixelTransfer()**。

GL_POST_COLOR_MATRIX_RED_SCALE

参数*params*返回一个值，该值表示在颜色矩阵转换操作之后应用于RGBA片断的红组分缩放因子。其缺省值是1。请参阅**glPixelTransfer()**。

GL_POST_COLOR_MATRIX_GREEN_SCALE

参数*params*返回一个值，该值表示在颜色矩阵转换操作之后应用于RGBA片断的绿组分缩放因子。其缺省值是1。请参阅**glPixelTransfer()**。

GL_POST_COLOR_MATRIX_BLUE_SCALE

参数*params*返回一个值，该值表示在颜色矩阵转换操作之后应用于RGBA片断的蓝组分缩放因子。其缺省值是1。请参阅**glPixelTransfer()**。

GL_POST_COLOR_MATRIX_ALPHA_SCALE

参数*params*返回一个值，该值表示在颜色矩阵转换操作之后应用于RGBA片断的alpha组分缩放因子。其缺省值是1。请参阅**glPixelTransfer()**。

GL_POST_CONVOLUTION_COLOR_TABLE

参数*params*返回一个boolean值，该值表示是否启动随后的卷积查询功能。其缺省值是**GL_FALSE**。请参阅**glColorTable()**。

GL_POST_CONVOLUTION_RED_BIAS

参数*params*返回一个值，该值表示在卷积操作之后应用于RGBA片断的红组分偏移因子。其缺省值是0。请参阅glPixelTransfer()。

GL_POST_CONVOLUTION_GREEN_BIAS

参数*params*返回一个值，该值表示在卷积操作之后应用于RGBA片断的绿组分偏移因子。其缺省值是0。请参阅glPixelTransfer()。

GL_POST_CONVOLUTION_BLUE_BIAS

参数*params*返回一个值，该值表示在卷积操作之后应用于RGBA片断的蓝组分偏移因子。其缺省值是0。请参阅glPixelTransfer()。

GL_POST_CONVOLUTION_ALPHA_BIAS

参数*params*返回一个值，该值表示在卷积操作之后应用于RGBA片断的alpha组分偏移因子。其缺省值是0。请参阅glPixelTransfer()。

GL_POST_CONVOLUTION_RED_SCALE

参数*params*返回一个值，该值表示在卷积操作之后应用于RGBA片断的红组分缩放因子。其缺省值是1。请参阅glPixelTransfer()。

GL_POST_CONVOLUTION_GREEN_SCALE

参数*params*返回一个值，该值表示在卷积操作之后应用于RGBA片断的绿组分缩放因子。其缺省值是1。请参阅glPixelTransfer()。

GL_POST_CONVOLUTION_BLUE_SCALE

参数*params*返回一个值，该值表示在卷积操作之后应用于RGBA片断的蓝组分缩放因子。其缺省值是1。请参阅glPixelTransfer()。

GL_POST_CONVOLUTION_ALPHA_SCALE

参数*params*返回一个值，该值表示在卷积操作之后应用于RGBA片断的alpha组分缩放因子。其缺省值是1。请参阅glPixelTransfer()。

GL_PROJECTION_MATRIX

参数*params*返回16个值，它们表示投影矩阵堆栈栈顶的投影矩阵。其缺省值是单位矩阵。请参阅glPushMatrix()。

GL_PROJECTION_STACK_DEPTH

参数*params*返回一个值，该值表示投影矩阵堆栈中矩阵的数目。其缺省值是1。请参阅glPushMatrix()。

GL_READ_BUFFER

参数*params*返回一个值，该值是一个符号常量，它表示选取哪个颜色缓冲区用来读取数据。当有后缓冲区时，其缺省值是GL_BACK。否则，其缺省值是GL_FRONT。请参阅glReadPixels()和glAccum()。

GL_RED_BIAS

参数*params*返回一个值，该值表示在像素转换过程中所使用的

	红组分偏移因子。其缺省值是0。
GL_RED_BITS	参数 <i>params</i> 返回一个值，该值表示每个颜色缓冲区中红组分位面的数目。
GL_RED_SCALE	参数 <i>params</i> 返回一个值，该值表示在像素转换过程中所使用的红组分缩放因子。其缺省值是1。请参阅glPixelTransfer()。
GL_RENDER_MODE	参数 <i>params</i> 返回一个值，该值是一个符号常量，它表示GL是处于绘制模式、选择模式还是反馈模式。其缺省值是 GL_RENDER 。请参阅glRenderMode()。
GL_RESCALE_NORMAL	参数 <i>params</i> 返回一个boolean值，该值表示是否启动法线重新缩放功能。请参阅glEnable()。
GL_RGBA_MODE	参数 <i>params</i> 返回一个boolean值，该值表明GL是处于RGBA模式(TRUE)还是颜色索引模式(FALSE)。请参阅glColor()。
GL_SCISSOR_BOX	参数 <i>params</i> 返回四个值，这些值分别代表裁剪箱的x和y窗口坐标及其宽度和高度。缺省情况下，x和y坐标都被设置为0，裁剪箱的宽度和高度被设置成等于窗口的宽度和高度。请参阅glScissor()。
GL_SCISSOR_TEST	参数 <i>params</i> 返回一个boolean值，该值表示是否启动裁剪功能。其缺省值是 GL_FALSE 。请参阅glScissor()。
GL_SELECTION_BUFFER_SIZE	参数 <i>params</i> 返回一个值，该值表示选择缓冲区的尺寸。请参阅glSelectBuffer()。
GL_SEPARABLE_2D	参数 <i>params</i> 返回一个boolean值，该值表示是否启动了2D分离卷积功能。其缺省值是 GL_FALSE 。请参阅glSeparableFilter2D()。
GL_SHADE_MODEL	参数 <i>params</i> 返回一个值，该值是一个符号常量，它表示浓淡处理模式是平坦方式还是光滑方式。其缺省值是 GL_SMOOTH 。请参阅glShadeModel()。
GL_SMOOTH_LINE_WIDTH_RANGE	参数 <i>params</i> 返回两个值，它们分别代表反走样线的最小和最大支持宽度值。请参阅glLineWidth()。
GL_SMOOTH_LINE_WIDTH_GRANULARITY	参数 <i>params</i> 返回一个值，该值表示反走样线的间隔尺寸。请参阅glLineWidgb()。
GL_SMOOTH_POINT_SIZE_RANGE	参数 <i>params</i> 返回两个值，它们分别代表反走样点的最小和最大支持宽度值。请参阅glPointSize()。
GL_SMOOTH_POINT_SIZE_GRANULARITY	

	参数 <i>params</i> 返回一个值，该值表示反走样点的间隔尺寸。请参阅glPointSize()。
GL_STENCIL_BITS	参数 <i>params</i> 返回一个值，该值表示模板缓冲区中位面的数目。
GL_STENCIL_CLEAR_VALUE	参数 <i>params</i> 返回一个值，该值表示用来清除模板位面的索引值。其缺省值是0。请参阅glClearStencil()。
GL_STENCIL_FAIL	参数 <i>params</i> 返回一个值，该值是一个符号常量，它表示当模板测试失败时将采取的动作。其缺省值是 GL_KEEP 。请参阅glStencilOp()。
GL_STENCIL_FUNC	参数 <i>params</i> 返回一个值，该值是一个符号常量，它表示可采用哪个函数比较模板缓冲区中的值和模板参考值。其缺省值是 GL_ALWAYS 。请参阅glStencilFunc()。
GL_STENCIL_PASS_DEPTH_FAIL	参数 <i>params</i> 返回一个值，该值是一个符号常量，它表示当模板测试通过但深度测试失败时所采取的动作。其缺省值是 GL_KEEP 。请参阅glStencilOp()。
GL_STENCIL_PASS_DEPTH_PASS	参数 <i>params</i> 返回一个值，该值是一个符号常量，它表示当模板测试和深度测试都通过时所采取的动作。其缺省值是 GL_KEEP 。请参阅glStencilOp()。
GL_STENCIL_REF	参数 <i>params</i> 返回一个值，该值表示用来同模板缓冲区中的内容进行比较的参考值。其缺省值是0。请参阅glStencilFunc()。
GL_STENCIL_TEST	参数 <i>params</i> 返回一个boolean值，该值表示是否启动了片断的模板测试功能。其缺省值是 GL_FALSE 。请参阅glStencilFunc()和glStencilOp()。
GL_STENCIL_VALUE_MASK	参数 <i>params</i> 返回一个值，该值表示在进行模板缓冲区中的值和模板参考值的比较操作前用于屏蔽这两个值的屏蔽值。其缺省值是全1的。请参阅glStencilFunc()。
GL_STENCIL_WRITEMASK	参数 <i>params</i> 返回一个值，该值表示用于控制模板位面写入操作的屏蔽值。其缺省值是全1的。请参阅glStencilMask()。
GL_STEREO	参数 <i>params</i> 返回一个boolean值，该值表示系统是否支持立体缓冲区（左缓冲区和右缓冲区）。
GL_SUBPIXEL_BITS	参数 <i>params</i> 返回一个值，该值表示在窗口坐标系中用来定位光栅化的几何体的子像素分辨率的二进制位数目的一个估计值。其缺省值是4。

GL_TEXTURE_1D	参数 <i>params</i> 返回一个boolean值，该值表示是否启动了1D纹理映射功能。其缺省值是 GL_FALSE 。请参阅glTexImage1D()。
GL_TEXTURE_BINDING_1D	参数 <i>params</i> 返回一个值，该值表示与目标 GL_TEXTURE_1D 相绑定的当前纹理名称。其缺省值是0。请参阅glBindTexture()。
GL_TEXTURE_2D	参数 <i>params</i> 返回一个boolean值，该值表示是否启动了2D纹理映射功能。其缺省值是 GL_FALSE 。请参阅glTexImage2D()。
GL_TEXTURE_BINDING_2D	参数 <i>params</i> 返回一个值，该值表示与目标 GL_TEXTURE_2D 相绑定的当前纹理名称。其缺省值是0。请参阅glBindTexture()。
GL_TEXTURE_3D	参数 <i>params</i> 返回一个boolean值，该值表示是否启动了3D纹理映射功能。其缺省值是 GL_FALSE 。请参阅glTexImage3D()。
GL_TEXTURE_BINDING_3D	参数 <i>params</i> 返回一个值，该值表示与目标 GL_TEXTURE_3D 相绑定的当前纹理名称。其缺省值是0。请参阅glBindTexture()。
GL_TEXTURE_COORD_ARRAY	参数 <i>params</i> 返回一个boolean值，该值表示是否启动了纹理坐标数组。其缺省值是 GL_FALSE 。请参阅glTexCoordPointer()。
GL_TEXTURE_COORD_ARRAY_SIZE	参数 <i>params</i> 返回一个值，该值表示纹理坐标数组中每个元素的坐标数目。其缺省值是4。请参阅glTexCoordPointer()。
GL_TEXTURE_COORD_ARRAY_STRIDE	参数 <i>params</i> 返回一个值，该值表示纹理坐标数组中相邻元素之间的字节偏移量。其缺省值是0。请参阅glTexCoordPointer()
GL_TEXTURE_COORD_ARRAY_TYPE	参数 <i>params</i> 返回一个值，该值表示纹理坐标数组中坐标的数据类型。其缺省值是 GL_FLOAT 。请参阅glTexCoordPointer()。
GL_TEXTURE_GEN_Q	参数 <i>params</i> 返回一个boolean值，该值表示是否启动了自动生成q纹理坐标的功能。其缺省值是 GL_FALSE 。请参阅glTexGen()。
GL_TEXTURE_GEN_R	参数 <i>params</i> 返回一个boolean值，该值表示是否启动了自动生成r纹理坐标的功能。其缺省值是 GL_FALSE 。请参阅glTexGen()。
GL_TEXTURE_GEN_S	参数 <i>params</i> 返回一个boolean值，该值表示是否启动了自动生成s纹理坐标的功能。其缺省值是 GL_FALSE 。请参阅glTexGen()。
GL_TEXTURE_GEN_T	参数 <i>params</i> 返回一个boolean值，该值表示是否启动了自动生成t纹理坐标的功能。其缺省值是 GL_FALSE 。请参阅glTexGen()。
GL_TEXTURE_MATRIX	参数 <i>params</i> 返回16个值，这些值表示位于纹理矩阵堆栈栈顶的纹理矩阵。其缺省值是单位矩阵。请参阅glPushMatrix()。

GL_TEXTURE_STACK_DEPTH

参数*params*返回一个值，该值表示纹理矩阵堆栈中纹理矩阵的数目。其缺省值是1。请参阅glPushMatrix()。

GL_UNPACK_ALIGNMENT

参数*params*返回一个值，该值表示当把像素读出内存时使用的字节对齐方式。其缺省值是4。请参阅glPixelStore()。

GL_UNPACK_IMAGE_HEIGHT

参数*params*返回一个值，该值表示当把像素读出内存时使用的图像的高度。其缺省值是0。请参阅glPixelStore()。

GL_UNPACK_LSB_FIRST

参数*params*返回一个boolean值，该值表示当从内存中读出单一二进制位的像素时第一个像素是否从每个无符号的字节最次要的位中读出的。其缺省值是GL_FALSE。请参阅glPixelStore()。

GL_UNPACK_ROW_LENGTH

参数*params*返回一个值，该值表示当从内存中读出数据时使用的行长度。其缺省值是0。请参阅glPixelStore()。

GL_UNPACK_SKIP_IMAGES

参数*params*返回一个值，该值表示当从内存中读出第一个像素前所跳过的像素图像的数目。其缺省值是0。请参阅glPixelStore()。

GL_UNPACK_SKIP_PIXELS

参数*params*返回一个值，该值表示当从内存中读出第一个像素前所跳过的像素存储单元的数目。其缺省值是0。请参阅glPixelStore()。

GL_UNPACK_SKIP_ROWS

参数*params*返回一个值，该值表示当从内存中读出第一个像素前所跳过的像素存储单元的行数。其缺省值是0。请参阅glPixelStore()。

GL_UNPACK_SWAP_BYTES

参数*params*返回一个boolean值，该值表示当从内存中读出像素前是否将双字节和四字节的像素索引和组分进行交换。其缺省值是GL_FALSE。请参阅glPixelStore()。

GL_VERTEX_ARRAY

参数*params*返回一个boolean值，该值表示是否启动了顶点数组。其缺省值是GL_FALSE。请参阅glVertexPointer()。

GL_VERTEX_ARRAY_SIZE

参数*params*返回一个值，该值表示顶点数组中每个顶点的坐标

数。其缺省值是4。请参阅glVertexPointer()。

GL_VERTEX_ARRAY_STRIDE

参数*params*返回一个值，该值表示顶点数组中相邻顶点之间的字节偏移量。其缺省值是0。请参阅glVertexPointer()。

GL_VERTEX_ARRAY_TYPE

参数*params*返回一个值，该值表示顶点数组中每个坐标的的数据类型。其缺省值是**GL_FLOAT**。请参阅glVertexPointer()。

GL_VIEWPORT

参数*params*返回四个值，这些值分别代表视口的x和y窗口坐标及它的宽度和高度。缺省情况下，x和y窗口坐标都被设置为0，视口的宽度和高度被设置成等于GL将绘入的窗口的宽度和高度。请参阅glViewport()。

GL_ZOOM_X

参数*params*返回一个值，该值表示像素的x缩放因子。其缺省值是1。请参阅glPixelZoom()。

GL_ZOOM_Y

参数*params*返回一个值，该值表示像素的y缩放因子。其缺省值是1。请参阅glPixelZoom()。

通过函数glIsEnabled()可以很容易地查询上述许多boolean参数。

- 注意：

GL_COLOR_LOGIC_OP、**GL_COLOR_ARRAY**、**GL_COLOR_ARRAY_SIZE**、**GL_COLOR_ARRAY_STRIDE**、**GL_COLOR_ARRAY_TYPE**、**GL_EDGE_FLAG_ARRAY**、**GL_EDGE_FLAG_ARRAY_STRIDE**、**GL_INDEX_ARRAY**、**GL_INDEX_ARRAY_STRIDE**、**GL_INDEX_ARRAY_TYPE**、**GL_INDEX_LOGIC_OP**、**GL_NORMAL_ARRAY**、**GL_NORMAL_ARRAY_STRIDE**、**GL_NORMAL_ARRAY_TYPE**、**GL_POLYGON_OFFSET_UNITS**、**GL_POLYGON_OFFSET_FACTOR**、**GL_POLYGON_OFFSET_FILL**、**GL_POLYGON_OFFSET_LINE**、**GL_POLYGON_OFFSET_POINT**、**GL_TEXTURE_COORD_ARRAY**、**GL_TEXTURE_COORD_ARRAY_SIZE**、**GL_TEXTURE_COORD_ARRAY_STRIDE**、**GL_TEXTURE_COORD_ARRAY_TYPE**、**GL_VERTEX_ARRAY**、**GL_VERTEX_ARRAY_SIZE**、**GL_VERTEX_ARRAY_STRIDE**、**GL_VERTEX_ARRAY_TYPE**仅在GL 1.1以上的版本中才有效。

GL_ALIASED_POINT_SIZE_RANGE、**GL_ALIASED_POINT_SIZE_GRANULARITY**、**GL_FEEDBACK_BUFFER_SIZE**、**GL_FEEDBACK_BUFFER_TYPE**、**GL_LIGHT_MODEL_AMBIENT**、**GL_LIGHT_MODEL_COLOR_CONTROL**、**GL_MAX_3D_TEXTURE_SIZE**、**GL_MAX_ELEMENTS_INDICES**、**GL_MAX_ELEMENTS_VERTICES**、**GL_PACK_IMAGE_HEIGHT**、**GL_PACK_SKIP_IMAGES**、**GL_RESCALE_NORMAL**、**GL_SELECTION_BUFFER_SIZE**、**GL_SMOOTH_LINE_WIDTH_GRANULARITY**、**GL_SMOOTH_LINE_WIDTH_RANGE**、**GL_SMOOTH_POINT_SIZE_GRANULARITY**、**GL_SMOOTH_POINT_SIZE_RANGE**、**GL_TEXTURE_3D**、**GL_TEXTURE_BINDING_3D**、**GL_UNPACK_IMAGE_HEIGHT**和**GL_UNPACK_SKIP_IMAGES**仅在GL 1.2以上的版本中才有效。

GL_LINE_WIDTH_GRANULARITY在GL 1.2版本中无效，该功能由**GL_SMOOTH_LINE_WIDTH_GRANULARITY**代替。

GL_LINE_WIDTH_RANGE在GL 1.2版本中无效，该功能由**GL_SMOOTH_LINE_WIDTH_RANGE**代替。

GL_POINT_SIZE_GRANULARITY在GL 1.2版本中无效，该功能由**GL_SMOOTH_POINT_SIZE_GRANULARITY**代替。

GL_POINT_SIZE_RANGE在GL 1.2版本中无效，该功能由**GL_SMOOTH_POINT_SIZE_RANGE**代替。

当调用函数glGet(GL_EXTENSIONS)的返回值是**GL_ARB_imaging**时，**GL_BLEND_COLOR**、**GL_BLEND_EQUATION**、**GL_COLOR_MATRIX**、**GL_COLOR_MATRIX_STACK_DEPTH**、**GL_COLOR_TABLE**、**GL_CONVOLUTION_1D**、**GL_CONVOLUTION_2D**、**GL_HISTOGRAM**、**GL_MAX_COLOR_MATRIX_STACK_DEPTH**、**GL_MINMAX**、**GL_POST_COLOR_MATRIX_COLOR_TABLE**、**GL_POST_COLOR_MATRIX_RED_BIAS**、**GL_POST_COLOR_MATRIX_GREEN_BIAS**、**GL_POST_COLOR_MATRIX_BLUE_BIAS**、**GL_POST_COLOR_MATRIX_ALPHA_BIAS**、**GL_POST_COLOR_MATRIX_RED_SCALE**、**GL_POST_COLOR_MATRIX_GREEN_SCALE**、**GL_POST_COLOR_MATRIX_BLUE_SCALE**、**GL_POST_COLOR_MATRIX_ALPHA_SCALE**、**GL_POST_CONVOLUTION_COLOR_TABLE**、**GL_POST_CONVOLUTION_RED_BIAS**、**GL_POST_CONVOLUTION_GREEN_BIAS**、**GL_POST_CONVOLUTION_BLUE_BIAS**、**GL_POST_CONVOLUTION_ALPHA_BIAS**、**GL_POST_CONVOLUTION_RED_SCALE**、**GL_POST_CONVOLUTION_GREEN_SCALE**、**GL_POST_CONVOLUTION_BLUE_SCALE**、**GL_POST_CONVOLUTION_ALPHA_SCALE**和**GL_SEPARABLE_2D**才有效。

当调用函数glGet(GL_EXTENSIONS)的返回值是**GL_ARB_multitexture**时，参数**GL_ACTIVE_TEXTURE_ARB**、**GL_CLIENT_ACTIVE_TEXTURE_ARB**和**GL_MAX_TEXTURE_UNITS_ARB**才有效。

当系统支持**GL_ARB_multitexture**扩展时，下面的参数返回当前有效的纹理单元的相应值：**GL_CURRENT_RASTER_TEXTURE_COORDS**、**GL_TEXTURE_1D**、**GL_TEXTURE_BINDING_1D**、**GL_TEXTURE_2D**、**GL_TEXTURE_BINDING_2D**、**GL_TEXTURE_3D**、**GL_TEXTURE_BINDING_3D**、**GL_TEXTURE_GEN_S**、**GL_TEXTURE_GEN_T**、**GL_TEXTURE_GEN_R**、**GL_TEXTURE_GEN_Q**、**GL_TEXTURE_MATRIX**和**GL_TEXTURE_STACK_DEPTH**。类似地，下列参数返回当前有效的客户端纹理单元的相应值：**GL_TEXTURE_COORD_ARRAY**、**GL_TEXTURE_COORD_ARRAY_SIZE**、**GL_TEXTURE_COORD_ARRAY_STRIDE**和**GL_TEXTURE_COORD_ARRAY_TYPE**。

- 出错提示：

当参数*pname*不是一个可取值时产生**GL_INVALID_ENUM**提示。

当函数glGet()在函数对glBegin()/glEnd()之间执行时，产生**GL_INVALID_OPERATION**提示。

- 请参阅：

glGetClipPlane(), **glGetColorTable()**, **glGetColorTableParameter()**, **glGetConvolutionFilter()**, **glGetColvolutionParameter()**, **glGetError()**, **glGetHistogram()**, **glGet HistogramParameter()**, **glGetLight()**, **glGetMap()**, **glGetMaterial()**, **glGetMinmax()**, **glGetMinmaxParameter()**, **glGetPixelMap()**, **glGetPointerv()**, **glGetPolygonStipple()**, **glGetSeparableFilter()**, **glGetString()**, **glGetTexEnv()**, **glGetTexGen()**, **glGetTexImage()**, **glGetTexLevelParameter()**, **glGetTexParameter()**, **glIsEnabled()**

• **glGetClipPlane**

- 名称:

glGetClipPlane()

- 功能:

返回指定剪切平面的系数。

- C描述:

```
void glGetClipPlane( GLenum plane,
                    GLdouble *equation )
```

- 参数说明:

plane 指定一个剪切平面。剪切平面的个数与GL具体实现有关，但至少支持六个剪切平面。它们通过符号名称**GL_CLIP_PLANE*i***来相互区别。此处 $0 \leq i < \text{GL_MAX_CLIP_PLANES}$ 。

equation 返回四个双精度值，这些值是眼坐标系中*plane*平面方程的系数。其缺省值是(0, 0, 0, 0)。

- 说明:

函数**glGetClipPlane()**将*plane*平面方程的四个系数返回到*equation*中。

- 注意:

下列等式总是成立的:

$$\text{GL_CLIP_PLANE}_i = \text{GL_CLIP_PLANE}_0 + i$$

当有错误发生时，参数*equation*的内容将不发生变化。

- 出错提示:

当参数*plane*不是一个可接受值时产生**GL_INVALID_ENUM**提示。

当函数**glGetClipPlane()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 请参阅:

glClipPlane()

• **glGetColorTable**

- 名称:

glGetColorTable()

- 功能:

获取一个颜色查询表的内容。

• C描述：

```
void glGetColorTable( GLenum target,
                      GLenum format,
                      GLenum type,
                      GLvoid *table )
```

• 参数说明：

target 必须是**GL_COLOR_TABLE**、**GL_POST_CONVOLUTION_COLOR_TABLE**或**GL_POST_COLOR_MATRIX_COLOR_TABLE**。

format 参数*table*中像素数据的格式。它可取的值有：**GL_RED**、**GL_GREEN**、**GL_BLUE**、**GL_ALPHA**、**GL_LUMINANCE**、**GL_LUMINANCE_ALPHA**、**GL_RGB**、**GL_BGR**、**GL_RGBA**和**GL_BGRA**。

type 参数*table*中像素数据的类型。它可以是下面的符号常量：**GL_UNSIGNED_BYTE**、**GL_BYTE**、**GL_BITMAP**、**GL_UNSIGNED_SHORT**、**GL_SHORT**、**GL_UNSIGNED_INT**、**GL_INT**、**GL_FLOAT**、**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**。

table 指定一个指针，指向像素数据的一个一维数组。该数组包含着颜色表的内容。

• 说明：

函数**glGetColorTable()**将参数*target*所指定的颜色表的内容返回到*table*中。这里不发生任何的像素转换操作，但将执行应用于函数**glReadPixels()**的像素存储模式。

指定的*format*所要求，但并不包含在颜色查询表内部格式中的颜色组分将被返回为0。内部颜色组分与参数*format*所要求的组分之间的分配关系如表5-13：

表 5-13

内 部 组 分	结 果 组 分
红	红
绿	绿
蓝	蓝
alpha	alpha
亮度	红
强度	红

• 注意：

当函数**glGetString(GL_EXTENSIONS)**的返回值是**GL_ARB_imaging**时，才提供函数

glGetColorTable()

- 出错提示：

当参数*target*不是一个允许值时产生**GL_INVALID_ENUM**提示。

当参数*format*不是一个允许值时产生**GL_INVALID_ENUM**提示。

当参数*type*不是一个允许值时产生**GL_INVALID_ENUM**提示。

当参数*type*为**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**或**GL_UNSIGNED_SHORT_5_6_5_REV**之一，但参数*format*却不是**GL_RGB**格式时产生**GL_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_SHORT_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**或**GL_UNSIGNED_INT_2_10_10_10_REV**之一，但参数*format*既不是**GL_RGBA**格式又不是**GL_BGRA**格式时产生**GL_INVALID_OPERATION**提示。

当函数**glGetColorTable()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 请参阅：

glColorTable(), **glColorTableParameter()**, **glGetColorTableParameter()**

***glGetColorTableParameter**

- 名称：

glGetColorTableParameterfv(), **glGetColorTableParameteriv()**

- 功能：

获取颜色查询表参数。

- C 描述：

```
void glGetColorTableParameterfv( GLenum target,
                                 GLenum pname,
                                 GLfloat *params )
void glGetColorTableParameteriv( GLenum target,
                                 GLenum pname,
                                 GLint *params )
```

- 参数说明：

target 指定目标颜色查询表。必须是**GL_COLOR_TABLE**、**GL_POST_CONVOLUTION_COLOR_TABLE**、**GL_POST_COLOR_MATRIX_COLOR_TABLE**、**GL_PROXY_COLOR_TABLE**、**GL_PROXY_POST_CONVOLUTION_COLOR_TABLE**、**GL_PROXY_POST_COLOR_MATRIX_COLOR_TABLE**。

pname 指定颜色查询表参数的符号名称。它必须是下面各值之一：**GL_COLOR_TABLE_BIAS**、**GL_COLOR_TABLE_SCALE**、**GL_COLOR_TABLE_FORMAT**、**GL_COLOR_TABLE_WIDTH**、**GL_COLOR_TABLE_RED_SIZE**、

GL_COLOR_TABLE_GREEN_SIZE、**GL_COLOR_TABLE_BLUE_SIZE**、**GL_COLOR_TABLE_ALPHA_SIZE**、**GL_COLOR_TABLE_LUMINANCE_SIZE**或**GL_COLOR_TABLE_INTENSITY_SIZE**。

params 指定数组指针。该数组用来存放参数值。

- 说明：

返回颜色查询表*target*的特定参数值。

当参数*pname*被设置为**GL_COLOR_TABLE_SCALE** 或**GL_COLOR_TABLE_BIAS**时，函数**glGetColorTableParameter()**将返回*target*所指定的颜色查询表的缩放或偏移参数。这时，参数*target*必须设置成**GL_COLOR_TABLE**, **GL_POST_CONVOLUTION_COLOR_TABLE**或**GL_POST_COLOR_MATRIX_COLOR_TABLE**、且参数*params*应该指向一个四元数组，该数组用来顺次接收红、绿、蓝和alpha 颜色组分的缩放因子或偏移因子。

函数**glGetColorTableParameter()**也可以用来返回一个颜色表的格式和尺寸参数。这时，参数*target*将设置成颜色表目标或代理颜色表目标。其格式及尺寸参数由函数**glColorTable()**设置。

表5-14列出了所需的格式和尺寸参数。对于下表中*pname*的每个符号常量，参数*params*都必须指向一个给定长度的数组，用来接收指定的值。

表 5-14

参 数	N	含 义
GL_COLOR_TABLE_FORMAT	1	内部格式（例如 GL_RGBA ）
GL_COLOR_TABLE_WIDTH	1	表中元素的数目
GL_COLOR_TABLE_RED_SIZE	1	红组分的尺寸（以二进制位为单位）
GL_COLOR_TABLE_GREEN_SIZE	1	绿组分的尺寸（以二进制位为单位）
GL_COLOR_TABLE_BLUE_SIZE	1	蓝组分的尺寸（以二进制位为单位）
GL_COLOR_TABLE_ALPHA_SIZE	1	alpha组分的尺寸（以二进制位为单位）
GL_COLOR_TABLE_LUMINANCE_SIZE	1	亮度组分的尺寸（以二进制位为单位）
GL_COLOR_TABLE_INTENSITY_SIZE	1	强度组分的尺寸（以二进制位为单位）

- 注意：

当函数**glGetString(GL_EXTENSIONS)**的返回值是**GL_ARB_imaging**时，才提供函数**glGetColorTableParameter()**。

- 出错提示：

当参数*target*或*pname*不是一个可接受值时产生**GL_INVALID_ENUM**提示。

当函数**glGetColorTableParameter()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 请参阅：

glColorTable(), **glTexParameter()**, **glColorTableParameter()**

- **glGetConvolutionFilter**

- 名称：

glGetConvolutionFilter()

• 功能：

获取当前的1D或2D卷积滤波器的内核。

• C描述：

```
void glGetConvolutionFilter( GLenum target,
                             GLenum format,
                             GLenum type,
                             GLvoid *image )
```

• 参数说明：

target 指定要返回的滤波器。它必须是**GL_CONVOLUTION_1D**或**GL_CONVOLUTION_2D**。

format 指定输出图像的格式。它的可取值必须是下列值之一：**GL_RED**、**GL_GREEN**、**GL_BLUE**、**GL_ALPHA**、**GL_RGB**、**GL_BGR**、**GL_RGBA**、**GL_BGRA**、**GL_LUMINANCE**或**GL_LUMINANCE_ALPHA**。

type 指定输出图像中组分的数据类型。它可以是下面的符号常量之一：**GL_UNSIGNED_BYTE**、**GL_BYTE**、**GL_BITMAP**、**GL_UNSIGNED_SHORT**、**GL_SHORT**、**GL_UNSIGNED_INT**、**GL_INT**、**GL_FLOAT**、**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**。

image 指定一个指针，指向用于存放输出图像的存储单元。

• 说明：

函数**glGetConvolutionFilter()**将当前的1D或2D卷积滤波器的内核作为图像返回。将一维或二维的图像以*format*和*type*指定的形式放入*image*中。这里不发生任何的像素转换操作，但将使用有关的像素存储模式。

参数*format*所提供的，但并不包含在滤波器内部格式中的颜色组分将被返回为0。内部颜色组分与参数*format*所要求的组分之间的分配关系如表5-15：

表 5-15

内部组分	结果组分
红	红
绿	绿
蓝	蓝
Alpha	Alpha
亮度	红
强度	红

- 注意：

当函数glGetString(GL_EXTENSIONS)的返回值是GL_ARB_imaging时，才提供函数glGetConvolutionFilter()。

当前的可分离的2D滤波器必须用函数glGetSeparableFilter()返回而不用函数glGetConvolutionFilter()返回。

- 出错提示：

当参数*target*不是一个允许值时产生GL_INVALID_ENUM提示。

当参数*format*不是一个允许值时产生GL_INVALID_ENUM提示。

当参数*type*不是一个允许值时产生GL_INVALID_ENUM提示。

当函数glGetConvolutionFilter()在函数对glBegin()/glEnd()之间执行时产生GL_INVALID_OPERATION提示。

当参数*type*为GL_UNSIGNED_BYTE_3_3_2、GL_UNSIGNED_BYTE_2_3_3_REV、GL_UNSIGNED_SHORT_5_6_5、GL_UNSIGNED_SHORT_5_6_5_REV之一，但参数*format*却不是GL_RGB格式时产生GL_INVALID_OPERATION提示。

当参数*type*为GL_UNSIGNED_SHORT_4_4_4_4、GL_UNSIGNED_SHORT_4_4_4_4_REV、GL_UNSIGNED_SHORT_5_5_5_1、GL_UNSIGNED_SHORT_1_5_5_5_REV、GL_UNSIGNED_INT_8_8_8_8、GL_UNSIGNED_INT_8_8_8_8_REV、GL_UNSIGNED_INT_10_10_10_2或GL_UNSIGNED_INT_2_10_10_10_REV之一，但参数*format*既不是GL_RGBA格式又不是GL_BGRA格式时产生GL_INVALID_OPERATION提示。

- 有关数据的获取：

glGetConvolutionParameter()

- 请参阅：

glGetSeparableFilter(), glConvolutionParameter(), glConvolutionFilter1D(), glConvolutionFilter2D()

* glGetConvolutionParameter

- 名称：

glGetConvolutionParameterfv(), glGetConvolutionParameteriv()

- 功能：

获取卷积参数。

- C描述：

```
void glGetConvolutionParameterfv( GLenum target,
                                  GLenum pname,
                                  GLfloat *params )
```

```
void glGetConvolutionParameteriv( GLenum target,
                                  GLenum pname,
                                  GLint *params )
```

- 参数说明：

target 指定一个滤波器，其参数将被返回。它必须是**GL_CONVOLUTION_1D**、**GL_CONVOLUTION_2D**或**GL_SEPARABLE_2D**之一。

pname 指定将要返回的参数。它必须取下面各值之一：**GL_CONVOLUTION_BORDER_MODE**、**GL_CONVOLUTION_BORDER_COLOR**、**GL_CONVOLUTION_FILTER_SCALE**、**GL_CONVOLUTION_FILTER_BIAS**、**GL_CONVOLUTION_FORMAT**、**GL_CONVOLUTION_WIDTH**、**GL_CONVOLUTION_HEIGHT**、**GL_MAX_CONVOLUTION_WIDTH**或**GL_MAX_CONVOLUTION_HEIGHT**。

params 指定一个指针，指向用于存放返回的参数的存储单元。

- 说明：

函数glGetConvolutionParameter()的作用是返回卷积参数。参数*target*确定所需的卷积滤波器。参数*pname*确定将返回的参数：

GL_CONVOLUTION_BORDER_MODE

卷积边界模式。请参阅glConvolutionParameter()中边界模式列表。

GL_CONVOLUTION_BORDER_COLOR

当前卷积边界的颜色。参数*params*必须是一个指向四元数组的指针。这四个值分别是返回的红、绿、蓝和alpha值边界颜色。

GL_CONVOLUTION_FILTER_SCALE

当前滤波器的缩放因子。参数*params*必须是一个指向四元数组的指针。这四个值分别是依次返回的红、绿、蓝和alpha滤波器缩放因子。

GL_CONVOLUTION_FILTER_BIAS

当前滤波器的偏移因子。参数*params*必须是一个指向四元数组的指针。这四个值分别是依次返回的红、绿、蓝和alpha滤波器偏移因子。

GL_CONVOLUTION_FORMAT

当前的内部格式。所允许的值请参阅glConvolutionFilter1D()、glConvolutionFilter2D()和glSeparableFilter2D()。

GL_CONVOLUTION_WIDTH

当前滤波图像的宽度。

GL_CONVOLUTION_HEIGHT

当前滤波图像的高度。

GL_MAX_CONVOLUTION_WIDTH

可接受的最大滤波图像宽度。

GL_MAX_CONVOLUTION_HEIGHT

可接受的最大滤波图像高度。

- 出错提示：

当参数*target*不是一个允许值时产生**GL_INVALID_ENUM**提示。

当参数*pname*不是一个允许值时产生**GL_INVALID_ENUM**提示。

当参数*target*是**GL_CONVOLUTION_1D**, 而参数*pname*是**GL_CONVOLUTION_HEIGHT**或**GL_MAX_CONVOLUTION_HEIGHT**时产生**GL_INVALID_ENUM**提示。

当函数**glGetConvolutionParameter()**在函数对**glBegin() / glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 请参阅：

glGetConvolutionFilter(), **glGetSeparableFilter2D()**, **glConvolutionParameter()**

* **glGetError**

- 名称：

glGetError()

- 功能：

返回出错信息。

- C 描述：

GLenum glGetError(void)

- 说明：

函数**glGetError()**将返回出错标志的值。每个可检测到的错误都对应一个数字代码和符号名称。当一个错误发生时，出错标志就被设置成适当的出错代码值。在调用函数**glGetError()**之前，其他的错误都不再被记录。当调用函数**glGetError()**后，出错代码将被返回，并且其标志将被重新设置成**GL_NO_ERROR**。如果调用函数**glGetError()**返回的是**GL_NO_ERROR**，就说明自从上次调用函数**glGetError()**后或从GL被初始化以来并没有发现任何可检测到的错误。

对于分布式实现机制而言，它可能会有多个出错标志。只要任何一个出错标志记录了一个错误，当调用函数**glGetError()**时，都会返回标志值并将其重新设置成**GL_NO_ERROR**。当不止一个标志都记录了一个错误时，函数**glGetError()**将返回并刷新一个任意出错标志值。这时，必须循环调用函数**glGetError()**，直到所有出错标志都被重新设置为止。这时函数**glGetError()**将返回**GL_NO_ERROR**。

初始情况下，所有的出错标志都被设置成**GL_NO_ERROR**。

下面是通常情况下的出错提示：

GL_NO_ERROR 没有出错记录。该符号常量的值一定是0。

GL_INVALID_ENUM 为枚举型自变量指定了一个不能接受的值。出错的命令将被忽略。除了重新设置出错标志外，它不产生其他方面的影响。

GL_INVALID_VALUE 数值型自变量超出了指定的范围。出错的命令将被忽略。除了重新设置出错标志外它不产生其他方面的影响。

GL_INVALID_OPERATION

当前状态下，指定的操作被禁止。出错的命令将被忽略。除了重新设置出错标志外它，不产生其他方面的影响。

GL_STACK_OVERFLOW

此命令导致了一个堆栈的上溢出。出错的命令将被忽略。除了重新设置出错标志外，它不产生其他方面的影响。

GL_STACK_UNDERFLOW

此命令导致了一个堆栈的下溢出。出错的命令将被忽略。除了重新设置出错标志外，它不产生其他方面的影响。

GL_OUT_OF_MEMORY

没有足够的存储空间来执行这个命令。该错误被记录后，除了出错标志的状态外，其他的GL状态将变得未定义。

GL_TABLE_TOO_LARGE

指定的表尺寸超出了具体实现所支持的最大表尺寸。出错的命令将被忽略。除了重新设置出错标志外，它不产生其他方面的影响。

当设置一个出错标志后，只有在产生**GL_OUT_OF_MEMORY**时，GL的操作结果才是未定义的。其他的所有情况都只是忽略出错的命令，它们并不对GL的状态及帧缓冲区中的内容产生影响。当生成命令返回一个值时，它返回0。当函数**glGetError()**本身产生错误时，它返回0。

- 注意：

GL_TABLE_TOO_LARGE是在GL 1.2版本中被引入的。

- 出错提示：

当函数**glGetError()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。这时函数**glGetError()**返回0。

• glGetHistogram

- 名称：

glGetHistogram()

- 功能：

获取直方图表。

- C描述：

```
void glGetHistogram( GLenum target,
                     GLboolean reset,
                     GLenum format,
                     GLenum type,
                     GLvoid *values )
```

- 参数说明：

target 必须是**GL_HISTOGRAM**。

reset 当它是**GL_TRUE**时，实际返回的每个组分计数器的值被重新设置成0（其他计数器不受影响）。当它是**GL_FALSE**时，直方图表中的任何计数器都不被修改。

format 返回到参数*values*中的值的格式。它必须是下面各值之一：**GL_RED**、**GL_GREEN**、**GL_BLUE**、**GL_ALPHA**、**GL_RGB**、**GL_BGR**、**GL_RGBA**、**GL_BGRA**、**GL_LUMINANCE**或**GL_LUMINANCE_ALPHA**。

type 返回到参数*values*中的值的类型。它可以是下面的符号常量之一：**GL_UNSIGNED_BYTE**、**GL_BYTE**、**GL_BITMAP**、**GL_UNSIGNED_SHORT**、

GL_SHORT、**GL_UNSIGNED_INT**、**GL_INT**、**GL_FLOAT**、**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**。

values 一个指针，指向存放返回的直方图表的存储单元。

- 说明：

函数**glGetHistogram()**将当前的直方图表用1D的图像返回。这个返回的图像的宽度等于直方图的宽度。该图像不执行像素转换操作，但将执行适用于1D图像的像素存储模式。

参数*format*指定的，但并不包含在直方图表内部格式中的颜色组分将被返回为0。

内部颜色组分与参数*format*所要求的组分之间的分配关系表5-16：

表 5-16

内部组分	结果组分
红	红
绿	绿
蓝	蓝
Alpha	Alpha
亮度	红

- 注意：

当函数**glGetString(GL_EXTENSIONS)**的返回值是**GL_ARB_imaging**时，才提供函数**glGetHistogram()**。

- 出错提示：

当参数*target*不是**GL_HISTOGRAM**时产生**GL_INVALID_ENUM**提示。

当参数*format*不是一个允许值时产生**GL_INVALID_ENUM**提示。

当参数*type*不是一个允许值时产生**GL_INVALID_ENUM**提示。

当函数**glGetHistogram()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**之一，但参数*format*却不是**GL_RGB**格式时产生**GL_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_SHORT_4_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**或**GL_UNSIGNED_INT_2_10_10_10_REV**之一，但参数*format*既不是**GL_RGBA**格式又不是**GL_BGRA**格式时产生**GL_INVALID_OPERATION**提示。

- 请参阅：

glHistogram(), **glResetHistogram()**, **glGetHistogramParameter()**

• **glGetHistogramParameter**

- 名称：

glGetHistogramParameterfv(), **glGetHistogramParameteriv()**

- 功能：

获取直方图参数。

- C 描述：

```
void glGetHistogramParameterfv( GLenum target,
                                GLenum pname,
                                GLfloat *params )
```

```
void glGetHistogramParameteriv( GLenum target,
                                GLenum pname,
                                GLint *params )
```

- 参数说明：

target 它必须是**GL_HISTOGRAM**或**GL_PROXY_HISTOGRAM**。

pname 将要返回的参数名称。它必须取下面各值之一：**GL_HISTOGRAM_WIDTH**、**GL_HISTOGRAM_FORMAT**、**GL_HISTOGRAM_RED_SIZE**、**GL_HISTOGRAM_GREEN_SIZE**、**GL_HISTOGRAM_BLUE_SIZE**、**GL_HISTOGRAM_ALPHA_SIZE**、**GL_HISTOGRAM_LUMINANCE_SIZE**、或**GL_HISTOGRAM_SINK**。

params 一个指针，指向存放返回值的存储单元。

- 说明：

函数**glGetHistogramParameter** ()用来查询当前直方图或一个代理的参数值。该函数也可以用来查询直方图的状态信息。这时参数*target*应该是**GL_HISTOGRAM**（获取当前直方图表的信息）或**GL_PROXY_HISTOGRAM**（从最近的代理请求中获取信息），并且参数*pname*取下面表 5-17 所示的各值之一：

表 5-17

参数	描述
GL_HISTOGRAM_WIDTH	直方图表的宽度
GL_HISTOGRAM_FORMAT	内部格式
GL_HISTOGRAM_RED_SIZE	红组分计数器尺寸（以二进制位为单位）
GL_HISTOGRAM_GREEN_SIZE	绿组分计数器尺寸（以二进制位为单位）
GL_HISTOGRAM_BLUE_SIZE	蓝组分计数器尺寸（以二进制位为单位）
GL_HISTOGRAM_ALPHA_SIZE	Alpha组分计数器尺寸（以二进制位为单位）
GL_HISTOGRAM_LUMINANCE	亮度组分计数器尺寸（以二进制位为单位）
GL_HISTOGRAM_SINK	参数 <i>sink</i> 的值

- 注意：

当函数glGetString(GL_EXTENSIONS)的返回值是GL_ARB_imaging时，才提供函数glGetHistogramParameter()。

- 出错提示：

当参数target不是一个允许值时产生GL_INVALID_ENUM提示。

当参数pname不是一个允许值时产生GL_INVALID_ENUM提示。

当函数glGetHistogramParameter()在函数对glBegin()/glEnd()之间执行时产生GL_INVALID_OPERATION提示。

- 请参阅：

glGetHistogram(), glHistogram()

• glGetLight

- 名称：

glGetLightfv(), glGetLightiv()

- 功能：

返回光源参数的值。

- C描述：

```
void glGetLightf( GLenum light,
                  GLenum pname,
                  GLfloat *params )
void glGetLighti( GLenum light,
                  GLenum pname,
                  GLint *params)
```

- 参数说明：

light 指定一个光源。光源的具体个数与GL具体实现有关，但至少支持八个光源。它们可通过符号型名称GL_LIGHT*i*进行识别，此处 $0 \leq i < \text{GL_MAX_LIGHTS}$ 。

pname 为*light*指定一个光源参数。可接受的符号型名称是GL_AMBIENT、GL_DIFFUSE、GL_SPECULAR、GL_POSITION、GL_SPOT_DIRECTION、GL_SPOT_EXPONENT、GL_SPOT_CUTOFF、GL_CONSTANT_ATTENUATION、GL_LINEAR_ATTENUATION和GL_QUADRATIC_ATTENUATION。

params 返回所需的值。

- 说明：

函数glGetLight()的作用是将返回的光源参数值放入*params*。参数*light*用来给光源命名，它是一个符号型的值，形如GL_LIGHT*i*，此处 $0 \leq i < \text{GL_MAX_LIGHTS}$ 。其中GL_MAX_LIGHTS是一个与具体实现有关的常数，它至少是8。参数*pname*指定为十种光源参数中的一种参数，它也是一个符号型值。

参数*pname*可取的十个参数为：

GL_AMBIENT	参数 <i>params</i> 返回四个整数值或浮点值，它们用来指定光源的环境光强度。如果需要整数，则内部的浮点值将按下述方法线性映射：
-------------------	---

1.0映射为所代表的最大正整数，-1.0映射为所代表的最小负整数。当内部值超出范围[-1, 1]时，相应的整型返回值将是未定义的。其缺省值是(0, 0, 0, 1)。

GL_DIFFUSE

参数*params*返回四个整数值或浮点值，它们用来指定光源的散射光强度。如果需要整数，则内部的浮点值将按上述方法线性映射：1.0映射为所代表的最大正整数，-1.0映射为所代表的最小负整数。当内部值超出范围[-1, 1]时，相应的整型返回值将是未定义的。光源**GL_LIGHT0**的缺省值是(1, 1, 1, 1)，其他光源的缺省值是(0, 0, 0, 0)。

GL_SPECULAR

参数*params*返回四个整数值或浮点值，它们用来指定光源的反射光强度。如果需要整数，则内部的浮点值将按上述方法线性映射：1.0映射为所代表的最大正整数，-1.0映射为所代表的最小负整数。当内部值超出范围[-1, 1]时，相应的整型返回值将是未定义的。光源**GL_LIGHT0**的缺省值是(1, 1, 1, 1)，其他光源的缺省值是(0, 0, 0, 0)。

GL_POSITION

参数*params*返回四个整数值或浮点值，它们用来指定光源的位置。如果需要整数，则将内部的浮点值圆整为最接近的整数值。这里的返回值是眼坐标中所包含的值。它们将不等于由函数**glLight()**指定的值，除非当调用函数**glLight()**时所使用的模式取景矩阵是单位矩阵。其缺省值是(0, 0, 1, 0)。

GL_SPOT_DIRECTION

参数*params*返回三个整数值或浮点值，它们用来指定光源的方向。如果需要整数，则将内部的浮点值圆整为最接近的整数值。这里的返回值是眼坐标中所包含的值。它们将不等于由函数**glLight()**指定的值，除非当调用函数**glLight()**时所使用的模式取景矩阵是单位矩阵。尽管在将聚光灯的方向带入光照方程前已进行了归一化，但指定值的转换后的值将优先于归一化的值返回。其缺省值是(0, 0, -1)。

GL_SPOT_EXPONENT

参数*params*返回一个整数值或浮点值，用来指定光源的聚光指数。如果需要整数，则将内部的浮点值圆整为最接近的整数值。其缺省值是0。

GL_SPOT_CUTOFF

参数*params*返回一个整数值或浮点值，用来指定光源的聚光截止角度。如果需要整数，则将内部的浮点值圆整为最接近的整数值。其缺省值是180。

GL_CONSTANT_ATTENUATION

参数*params*返回一个整数值或浮点值，用来指定光源的常数衰

减因子（即与距离无关）。如果需要整数，则将内部的浮点值圆整为最接近的整数值。其缺省值是1。

GL_LINEAR_ATTENUATION

参数*params*返回一个整数值或浮点值，用来指定光源的线性衰减因子。如果需要整数，则将内部的浮点值圆整为最接近的整数值。其缺省值是0。

GL_QUADRATIC_ATTENUATION

参数*params*返回一个整数值或浮点值，用来指定光源的二次衰减因子。如果需要整数，则将内部的浮点值圆整为最接近的整数值。其缺省值是0。

- 注意：

下式总是成立的：

$$\mathbf{GL_LIGHT}_i = \mathbf{GL_LIGHT0} + i$$

当有错误发生时，参数*params*中的内容将不发生改变。

- 出错提示：

当参数*light*或*pname*不是一个可接受的值时产生**GL_INVALID_ENUM**提示。

当函数**glGetLight()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 请参阅：

glLight()

glGetMap

- 名称：

glGetMapdv(), **glGetMapfv()**, **glGetMapiv()**

- 功能：

返回求值器参数。

- C描述：

```
void glGetMapdv( GLenum target,
```

```
                      GLenum query,
```

```
                      GLdouble *v )
```

```
void glGetMapfv( GLenum target,
```

```
                      GLenum query,
```

```
                      GLfloat *v )
```

```
void glGetMapiv( GLenum target,
```

```
                      GLenum query,
```

```
                      GLint *v )
```

- 参数说明：

target 指定一个映射的符号名称。它可取的值有：**GL_MAP1_COLOR_4**、**GL_**

MAP1_INDEX、**GL_MAP1_NORMAL**、**GL_MAP1_TEXTURE_COORD_1**、**GL_MAP1_TEXTURE_COORD_2**、**GL_MAP1_TEXTURE_COORD_3**、**GL_MAP1_TEXTURE_COORD_4**、**GL_MAP1_VERTEX_3**、**GL_MAP1_VERTEX_4**、**GL_MAP2_COLOR_4**、**GL_MAP2_INDEX**、**GL_MAP2_NORMAL**、**GL_MAP2_TEXTURE_COORD_1**、**GL_MAP2_TEXTURE_COORD_2**、**GL_MAP2_TEXTURE_COORD_3**、**GL_MAP2_TEXTURE_COORD_4**、**GL_MAP2_VERTEX_3**和**GL_MAP2_VERTEX_4**。

query 指定哪个参数将被返回。它可以取**GL_COEFF**、**GL_ORDER**和**GL_DOMAIN**。
v 返回所需要的数据。

- 说明：

函数**glMap1()**和**glMap2()**用来定义求值器。而函数**glGetMap()**的作用是返回求值器的参数值。参数**target**用来选取一个映射，参数**query**用来选取一个指定的参数，参数**v**用来指向存放将返回值的存储单元。

参数**target**的可取值在函数**glMap1()**和**glMap2()**的参考说明中描述。

参数**query**可取下面各值：

GL_COEFF 参数**v**返回求值器函数的控制点。一维求值器返回**order**个控制点，二维求值器返回**uorder**×**vorder**个控制点。每个控制点包含一个、两个、三个或四个整型、单精度浮点或双精度浮点值，具体情况与求值器的类型有关。GL返回二维控制点时是遵循行优先次序的，也就是**uorder**索引的增加速度要快，**vorder**索引跟在每行的后面。如果需要整数，则将内部的浮点值圆整为最接近的整数值。

GL_ORDER 参数**v**返回求值器函数的阶次。一维求值器返回一个值**order**。其缺省值是1。二维求值器返回两个值**uorder**和**vorder**。其缺省值是1，1。

GL_DOMAIN 参数**v**返回线性映射参数**u**和**v**。一维求值器返回两个值**u1**和**u2**，它们均由函数**glMap1()**指定。二维求值器返回四个值（**u1**，**u2**，**v1**和**v2**），它们均由函数**glMap2()**指定。如果需要整数，则将内部的浮点值圆整为最接近的整数值。

- 注意：

当有错误发生时，参数**v**中的内容将不发生改变。

- 出错提示：

当参数**target**或**query**不是一个可接受的值时产生**GL_INVALID_ENUM**提示。

当函数**glGetMap()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 请参阅：

glEvalCoord(), **glMap1()**, **glMap2()**

- **glGetMaterial**

- 名称：

glGetMaterialfv(), glGetMaterialiv()

- 功能：

返回材质参数。

- C 描述：

```
void glGetMaterialfv( GLenum face,
                      GLenum pname,
                      GLfloat *params )
void glGetMaterialiv( GLenum face,
                      GLenum pname,
                      GLint *params )
```

- 参数说明：

face 指定两种材质中哪种材质被查询。**GL_FRONT**和**GL_BACK**分别代表正向和反向材质。

pname 指定要返回的材质参数。它可以取**GL_AMBIENT**、**GL_DIFFUSE**、**GL_SPECULAR**、**GL_EMISSION**、**GL_SHININESS**和**GL_COLOR_INDEXS**之一。

params 返回所需的数据。

- 说明：

函数**glGetMaterial()**将返回材质*face*的参数值*pname*，并将它放入*params*中。它定义了六个参数：

GL_AMBIENT 参数*params*返回四个整数值或浮点值，它们代表材质的环境光的反射比。如果需要整数，则内部的浮点值将按下述方法线性映射：1.0映射为所代表的最大正整数，-1.0映射为所代表的最小负整数。当内部值超出范围[-1, 1]时，相应的整型返回值将是未定义的。其缺省值是(0.2, 0.2, 0.2, 1.0)。

GL_DIFFUSE 参数*params*返回四个整数值或浮点值，它们代表材质的散射光的反射比。如果这里需要整数，内部的浮点值将按下述方法线性映射：1.0映射为所代表的最大正整数，-1.0映射为所代表的最小负整数。当内部值超出范围[-1, 1]时，相应的整型返回值将是未定义的。其缺省值是(0.8, 0.8, 0.8, 1.0)。

GL_SPECULAR 参数*params*返回四个整数值或浮点值，它们代表材质的镜面反射光的反射比。如果需要整数，则内部的浮点值将按下述方法线性映射：1.0映射为所代表的最大的正整数，-1.0映射为所代表的最小负整数。当内部值超出范围[-1, 1]时，相应的整型返回值将是未定义的。其缺省值是(0, 0, 0, 1)。

GL_EMISSION 参数*params*返回四个整数值或浮点值，它们代表材质的辐射光的强度。如果需要整数，则内部的浮点值将按下述方法线性映射：1.0映射为所代表的最大正整数，-1.0映射为所代表的最小负整数。当内部值超出范围[-1, 1]时，相应的整型返回值将是未定义的。其缺省值是(0, 0, 0, 1)。

GL_SHININESS 参数*params*返回一个整数值或浮点值，它代表材质的镜面反射指数。如果需要整数，则将内部的浮点值圆整为最接近的整数值。其缺省值是0。

GL_COLOR_INDEX

参数*params*返回三个整数值或浮点值，它们代表材质的环境光索引、散射光索引和镜面反射光索引。这些索引仅用于颜色索引光照模式（所有其他参数仅供RGBA光照模式使用。）如果需要整数，则将内部的浮点值圆整为最接近的整数值。

- 注意：

当有错误发生时，参数*params*中的内容将不发生改变。

- 出错提示：

当参数*face*或*pname*不是一个可接受的值时产生**GL_INVALID_ENUM**提示。

当函数**glGetMaterial()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 请参阅：

glMaterial()

glGetMinmax

- 名称：

glGetMinmax()

- 功能：

获取像素的最小和最大值。

- C 描述：

```
void glGetMinmax( GLenum target,
                  GLboolean reset,
                  GLenum format,
                  GLenum type,
                  GLvoid *values )
```

- 参数说明：

target 必须是**GL_MINMAX**。

reset 当它是**GL_TRUE**时，实际返回的极值表中的每个条目值都被重新设置成它们的初始值。（其他条目不受影响。）当它是**GL_FALSE**时，极值表不被修改。

format 指定返回到*value*中的数据的格式。它可取的值有：**GL_RED**、**GL_GREEN**、**GL_BLUE**、**GL_ALPHA**、**GL_RGB**、**GL_BGR**、**GL_RGBA**、**GL_BGRA**、**GL_LUMINANCE**或**GL_LUMINANCE_ALPHA**。

type 指定返回到*value*中的数据的类型。它可以是下面的符号常量之一：**GL_UNSIGNED_BYTE**、**GL_BYTE**、**GL_BITMAP**、**GL_UNSIGNED_SHORT**、**GL_SHORT**、**GL_UNSIGNED_INT**、**GL_INT**、**GL_FLOAT**、**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED**

`_INT_8_8_8_8`、`GL_UNSIGNED_INT_8_8_8_8_REV`、`GL_UNSIGNED_INT_10_10_10_2`和`GL_UNSIGNED_INT_2_10_10_10_REV`。

values 指定一个指针，指向存放返回值的存储单元。

- 说明：

函数`glGetMinmax()`将返回宽度为2的一维图像中累积的最小和最大的像素值（以各组分为基准计算）。返回值中的第一个集合是最小值集，第二个集合是最大值集。这些返回值的格式由参数`format`确定，类型由参数`type`确定。

返回的值不进行像素转换操作，但将执行适用于1D图像的像素存储模式。指定参数`format`所要求的，但并不包含在极值表的内部格式中的颜色组分将被返回为0。内部颜色组分与参数`format`所要求的组分之间的分配关系表5-18：

表 5-18

内部组分	结果组分
红	红
绿	绿
蓝	蓝
Alpha	Alpha
亮度	红

当参数`reset`是`GL_TRUE`时，与返回值相对应的极值表中的条目将被重新设置成它们的初始值。即使当参数`reset`是`GL_TRUE`时，没有被返回的最小值和最大值将不被修改。

- 注意：

当函数`glGetString(GL_EXTENSIONS)`的返回值是`GL_ARB_imaging`时，才提供函数`glGetMinmax()`。

- 出错提示：

当参数`target`不是`GL_MINMAX`时产生`GL_INVALID_ENUM`提示。

当参数`format`不是一个允许值时产生`GL_INVALID_ENUM`提示。

当参数`type`不是一个允许值时产生`GL_INVALID_ENUM`提示。

当函数`glGetMinmax()`在函数对`glBegin()`/`glEnd()`之间执行时产生`GL_INVALID_OPERATION`提示。

当参数`type`为`GL_UNSIGNED_BYTE_3_3_2`、`GL_UNSIGNED_BVTE_2_3_3_REV`、`GL_UNSIGNED_SHORT_5_6_5`、`GL_UNSIGNED_SHORT_5_6_5_REV`之一，但参数`format`却不是`GL_RGB`格式时产生`GL_INVALID_OPERATION`提示。

当参数`type`为`GL_UNSIGNED_SHORT_4_4_4_4`、`GL_UNSIGNED_SHORT_4_4_REV`、`GL_UNSIGNED_SHORT_5_5_5_1`、`GL_UNSIGNED_SHORT_1_5_5_5_REV`、`GL_UNSIGNED_INT_8_8_8_8`、`GL_UNSIGNED_INT_8_8_8_8_REV`、`GL_UNSIGNED_INT_10_10_10_2`或`GL_UNSIGNED_INT_2_10_10_10_REV`之一，但参数`format`既不是`GL_RGBA`格式又不是`GL_BGRA`格式时产生`GL_INVALID_OPERATION`提示。

- 请参阅：

glMinmax(), glResetMinmax(), glGetMinmaxParameter()**glGetMinmaxParameter**

- 名称:

glGetMinmaxParameterfv(), glGetMinmaxParameteriv()

- 功能:

获取极值(最大和最小)参数。

- C描述:

```
void glGetMinmaxParameterfv(GLenum target,
                           GLenum pname,
                           GLfloat *params)
void glGetMinmaxParameteriv(GLenum target,
                           GLenum pname,
                           GLint *params)
```

- 参数说明:

target 它必须是**GL_MINMAX**。

pname 指定将要返回的参数。它必须是**GL_MINMAX_FORMAT**或**GL_MINMAX_SINK**之一。

params 指定一个指针，指向存放返回值的存储单元。

- 说明:

函数**glGetMinmaxParameter()**通过将参数*pname*设置成表5-19所示的值之一的方法返回当前极值表的参数:

表 5-19

参数	描述
GL_MINMAX_FORMAT	极值表的内部格式
GL_MINMAX_SINK	参数 <i>sink</i> 的值

- 注意:

当函数**glGetString(GL_EXTENSIONS)**的返回值是**GL_ARB_imaging**时，才提供函数**glGetMinmaxParameter()**。

- 出错提示:

当参数*target*不是**GL_MINMAX**时产生**GL_INVALID_ENUM**提示。

当参数*pname*不是一个允许值时产生**GL_INVALID_ENUM**提示。

当函数**glGetMinmaxParameter()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 请参阅:

glMinmax(), glGetMinmax()

◆ glGetPixelMap

- 名称：

glGetPixelMapfv(), **glGetPixelMapuiv()**, **glGetPixelMapusv()**

- 功能：

返回指定的像素映射。

- C 描述：

```
void glGetPixelMapfv( GLenum map,
                     GLfloat *values )
void glGetPixelMapuiv( GLenum map,
                     GLuint *values )
void glGetPixelMapusv( GLenum map,
                     GLushort *values )
```

- 参数说明：

map 指定要返回的像素映射名称。它必须取下面符号常量之**GL_PIXEL_MAP_I_TO_I**、**GL_PIXEL_MAP_S_TO_S**、**GL_PIXEL_MAP_I_TO_R**、**GL_PIXEL_MAP_I_TO_G**、**GL_PIXEL_MAP_I_TO_B**、**GL_PIXEL_MAP_I_TO_A**、**GL_PIXEL_MAP_R_TO_R**、**GL_PIXEL_MAP_G_TO_G**、**GL_PIXEL_MAP_B_TO_B**和**GL_PIXEL_MAP_A_TO_A**。

values 返回像素映射的内容。

- 说明：

有关参数*map*的可接受的值，请参阅**glPixelMap()**的参考说明。函数**glGetPixelMap()**用来返回由参数*map*指定的像素映射的内容，并将它放入*values*中。在某些函数把颜色索引、模板索引、颜色组分和深度组分映射成其他值时将使用像素映射。这些函数有：**glReadPixels()**、**glDrawPixels()**、**glCopyPixels()**、**glTexImage1D()**、**glTexImage2D()**、**glTexImage3D()**、**glTexSubImage1D()**、**glTexSubImage2D()**、**glTexSubImage3D()**、**glCopyTexImage1D()**、**glCopyTexImage2D()**、**glCopyTexSubImage1D()**、**glCopyTexSubImage2D()**、**glCopyTexSubImage3D()**、**glColorTable()**、**glColorSubTable()**、**glCopyColorTable()**、**glCopyColorSubTable()**、**glConvolutionFilter1D()**、**glConvolutionFilter2D()**、**glSeparableFilter2D()**、**glGetHistogram()**、**glGetMimmax()**和**glGetTexImage()**。

如果需要无符号的整数值，则内部的定点值或浮点值将按下述方法线性映射：1.0映射为所代表的最大整数，0.0映射为0。当映射值超出范围[0, 1]时，返回的无符号整数值将是未定义的。

如果想要确定参数*map*所要求的尺寸，可以调用带有相应符号常量的函数**glGet()**。

- 注意：

当有错误发生时，参数*values*中的内容将不发生改变。

- 出错提示：

当参数*map*不是一个可接受值时产生**GL_INVALID_ENUM**提示。

当函数**glGetPixelMap()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID**

_OPERATION提示。

- 有关数据的获取：

```
glGet( GL_PIXEL_MAP_I_TO_I_SIZE )
glGet( GL_PIXEL_MAP_S_TO_S_SIZE )
glGet( GL_PIXEL_MAP_I_TO_R_SIZE )
glGet( GL_PIXEL_MAP_I_TO_G_SIZE )
glGet( GL_PIXEL_MAP_I_TO_B_SIZE )
glGet( GL_PIXEL_MAP_I_TO_A_SIZE )
glGet( GL_PIXEL_MAP_R_TO_R_SIZE )
glGet( GL_PIXEL_MAP_G_TO_G_SIZE )
glGet( GL_PIXEL_MAP_B_TO_B_SIZE )
glGet( GL_PIXEL_MAP_A_TO_A_SIZE )
glGet( GL_MAX_PIXEL_MAP_TABLE )
```

- 请参阅：

```
glColorSubTable(), glColorTable(), glConvolutionFilter1D(), glConvolutionFilter2D(),
glCopyColorSubTable(), glCopyColorTable(), glCopyPixels(), glCopyTexImage1D(),
glCopyTexImage2D(), glCopyTexSubImage1D(), glCopyTexSubImage2D(), glCopyTexSub-
Image3D(), glDrawPixels(), glGetHistogram(), glGetMinmax(), glGetTexImage(),
glPixelMap(), glPixelTransfer(), glReadPixels(), glSeparableFilter2D(), glTexImage1D(),
glTexImage2D(), glTexImage3D(), glTexSubImage1D(), glTexSubImage2D(), glTexSub-
Image3D()
```

***glGetPointerv**

- 名称：

glGetPointerv()

- 功能：

返回指定指针的地址。

- C描述：

```
void glGetPointerv( GLenum pname,
                    GLvoid *params )
```

- 参数说明：

pname 指定要返回的数组或缓冲区指针。它可取的符号常量如下：GL_COLOR_ARRAY_POINTER、GL_EDGE_FLAG_ARRAY_POINTER、GL_FEEDBACK_BUFFER_POINTER、GL_INDEX_ARRAY_POINTER、GL_NORMAL_ARRAY_POINTER、GL_TEXTURE_COORD_ARRAY_POINTER、GL_SELECTION_BUFFER_POINTER和GL_VERTEX_ARRAY_POINTER。

params 返回由*pname*指定的指针值。

- 说明：

函数`glGetPointerv()`的作用是返回指针信息。参数`pname`是一个符号常量，它表示要返回的指针，而参数`params`是一个指针，它指向存放返回数据的存储单元。

- 注意：

函数`glGetPointerv()`在GL 1.1以上的版本中才有效。

所有指针都是客户端状态。

每个指针的缺省值都是0。

当系统支持**GL_ARB_multitexture**时，查询**GL_TEXTURE_COORD_ARRAY_POINTER**将返回当前有效的客户端纹理单元的值。

- 出错提示：

当参数`pname`不是一个可接受的值时产生**GL_INVALID_ENUM**提示。

- 请参阅：

`glArrayElement()`, `glClientActiveTextureARB()`, `glColorPointer()`, `glDrawArrays()`,
`glEdgeFlagPointer()`, `glFeedbackBuffer()`, `glIndexPointer()`, `glInterleavedArrays()`,
`glNormalPointer()`, `glSelectBuffer()`, `glTexCoordPointer()`, `glVertexPointer()`

glGetPolygonStipple

- 名称：

`glGetPolygonStipple()`

- 功能：

返回多边形的点画绘制模式。

- C描述：

`void glGetPolygonStipple(GLubyte *mask)`

- 参数说明：

`mask` 返回点画绘制模式。其缺省值是全1的。

- 说明：

函数`glGetPolygonStipple()`将把一个 32×32 的多边形点画绘制模式返回给`mask`。该模式被装入内存中。这一过程就好象在`type`是**GL_BITMAP**、`format`是**GL_COLOR_INDEX**、`height`和`width`都是32的情况下调用函数`glReadPixels()`并将点画绘制模式存入一个内部 32×32 颜色索引缓冲区中。然而，与调用函数`glReadPixels()`不同的是，这时并不对返回的点画图像进行像素转换操作（替换、偏移及像素映射）。

- 注意：

当有错误发生时，参数`mask`中的内容将不发生改变。

- 出错提示：

当函数`glGetPolygonStipple()`在函数对`glBegin()`/`glEnd()`之间执行时产生**GL_INVALID_OPERATION**提示。

- 请参阅：

`glPixelStore()`, `glPixelTransfer()`, `glPolygonStipple()`, `glReadPixels()`

• **glGetSeparableFilter**

- 名称:

glGetSeparableFilter()

- 功能:

获取可分离的卷积滤波器核心图像。

- C 描述:

```
void glGetSeparableFilter( GLenum target,
                           GLenum format,
                           GLenum type,
                           GLvoid *row,
                           GLvoid *column,
                           GLvoid *span )
```

- 参数说明:

target 指定要返回的可分离滤波器。它必须是**GL_SEPARABLE_2D**。

format 指定输出图像的格式。它必须是下列各值之一: **GL_RED**、**GL_GREEN**、**GL_BLUE**、**GL_ALPHA**、**GL_RGB**、**GL_BGR**、**GL_RGBA**、**GL_BGRA**、**GL_LUMINANCE**或**GL_LUMINANCE_ALPHA**。

type 指定输出图像中组分的数据类型。它可以是下面的符号常量之一: **GL_UNSIGNED_BYTE**、**GL_BYTE**、**GL_BITMAP**、**GL_UNSIGNED_SHORT**、**GL_SHORT**、**GL_UNSIGNED_INT**、**GL_INT**、**GL_FLOAT**、**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_REV**。

row 指定指向存放行滤波器图像的存储单元的指针。

column 指定指向存放列滤波器图像的存储单元的指针。

span 指定指向存放所涵盖滤波器图像(当前没被使用)的存储单元的指针。

- 说明:

函数**glGetSeparableFilter()**将两个一维的滤波器核心图像返回给当前的可分离2D卷积滤波器。根据指定的格式*format*和类型*type*, 将行图形和列图形分别放入了*row*和*column*中。(在当前实现机制中, *span*总是无效的。)这里不发生任何的像素转换操作, 但将使用相应的像素存储模式。

参数*format*所提供的, 但并不包含在颜色查询表内部格式中的颜色组分将被返回为0。内部颜色组分与参数*format*所要求的组分之间的分配关系表5-20:

表 5-20

内部组分	结果组分
红	红
绿	绿
蓝	蓝
Alpha	Alpha
亮度	红
强度	红

- 注意：

当函数glGetString(GL_EXTENSIONS)的返回值是GL_ARB_imaging时，才提供函数glGetSeparableFilter()。

不可分离的2D滤波器必须用函数glGetConvolutionFilter()恢复。

- 出错提示：

当参数target不是GL_SEPARABLE_2D时产生GL_INVALID_ENUM提示。

当参数format不是一个允许值时产生GL_INVALID_ENUM提示。

当参数type不是一个允许值时产生GL_INVALID_ENUM提示。

当函数glGetSeparableFilter()在函数对glBegin()/glEnd()之间执行时产生GL_INVALID_OPERATION提示。

当参数type为GL_UNSIGNED_BYTE_3_3_2、GL_UNSIGNED_BYTE_2_3_3_REV、GL_UNSIGNED_SHORT_5_6_5、GL_UNSIGNED_SHORT_5_6_5_REV之一，但参数format却不是GL_RGB格式时产生GL_INVALID_OPERATION提示。

当参数type为GL_UNSIGNED_SHORT_4_4_4、GL_UNSIGNED_SHORT_4_4_4_REV、GL_UNSIGNED_SHORT_5_5_5_1、GL_UNSIGNED_SHORT_1_5_5_5_REV、GL_UNSIGNED_INT_8_8_8_8、GL_UNSIGNED_INT_8_8_8_REV、GL_UNSIGNED_INT_10_10_10_2和GL_UNSIGNED_INT_2_10_10_10_REV之一，但参数format既不是GL_RGBA格式又不是GL_BGRA格式时产生GL_INVALID_OPERATION提示。

- 有关数据的获取：

glGetConvolutionParameter()

- 请参阅：

glGetConvolutionFilter(), glConvolutionParameter(), glSeparableFilter2D()

• glGetString

- 名称：

glGetString()

- 功能：

返回一个描述当前GL的关联内容的字符串。

- C描述：

```
const GLubyte*glGetString( GLenum name )
```

- 参数说明：

name 指定一个符号常量。它可以是**GL_VENDOR**, **GL_RENDERER**, **GL_VERSION** 或 **GL_EXTENSIONS**。

- 说明：

函数**glGetString()**返回一个指针。该指针指向一个静态字符串。这个字符串描述了与当前GL相关联的某些情况。参数*name*可以取下面的值：

GL_VENDOR 返回一个对该GL具体实现负责的公司。该名称不会随版本的改变而发生变化。

GL_RENDERER 返回绘制者的名字。该名称通常专用于一个硬件平台的特定配置。该名称不会随版本的改变而发生变化。

GL_VERSION 返回一个版本号。

GL_EXTENSIONS 返回GL所支持的一个用空格隔开的扩展清单。

由于GL中并不包含对一个实现机制的执行特性的查询，所以某些应用被写入了公认的已知平台，并基于这些平台已知的执行特性而修整它们的GL用法。字符串**GL_VENDOR**和**GL_RENDERER**共同指定了一个唯一的平台。它们不会随版本的改变而发生变化，而且它们被该平台的公认规则使用。

如果有些应用想要使用标准GL所没有的特征，则这些特征可以作为标准GL的扩展来实现。字符串**GL_EXTENSIONS**是一个GL所支持的用空格隔开的扩展清单。（扩展名中不能包含空格。）

字符串**GL_VERSION**是以一个版本号开始的。该版本号使用下列形式之一：

major_number.minor_number

major_number.minor_number.release_number

生产厂商所指定的信息也可以跟在版本号后而。它的格式与所采用的具体实现有关，但版本号与生产厂商指定的信息总是用空格来隔开的。

所有字符串都是空结束的。

- 注意：

当有错误发生时，函数**glGetString()**返回0。

客户端和服务器端可能支持不同的版本或扩展。函数**glGetString()**总是返回一个可兼容的版本号或扩展清单。版本号总是用来描述服务器。

- 出错提示：

当参数*name*不是一个可接受的值时产生**GL_INVALID_ENUM**提示。

当函数**glGetString()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

glGetTexEnv

- 名称：

glGetTexEnvfv(), **glGetTexEnviv()**

- 功能：

返回纹理环境参数。

- C 描述：

```
void glGetTexEnvfv( GLenum target,
                     GLenum pname,
                     GLfloat *params )
void glGetTexEnviv( GLenum target,
                     GLenum pname,
                     GLint *params )
```

- 参数说明：

target 指定一个纹理环境。它必须是**GL_TEXTURE_ENV**。

pname 指定一个纹理环境参数的符号名称。它可以取**GL_TEXTURE_ENV_MODE**和**GL_TEXTURE_ENV_COLOR**。

params 返回所需的数据。

- 说明：

函数**glGetTexEnv()**将由函数**glTexEnv()**指定的纹理环境的选定值返回到*params*中。参数*target*用来指定一个纹理环境。当前，被定义并支持的纹理环境只有一个：**GL_TEXTURE_ENV**。

参数*pname*用来指定一个纹理环境参数的符号名称。它可以是：

GL_TEXTURE_ENV_MODE

参数*params*返回一个单值的纹理环境模式，它是一个符号常量。其缺省值是**GL_MODULATE**。

GL_TEXTURE_ENV_COLOR

参数*params*返回四个整型或浮点型的值，它们代表纹理环境的颜色。如果需要整数，则内部的浮点值将按上述方法映射：1.0 映射为可代表的最大的正整数，-1.0 映射为可代表的最小的负整数。其缺省值是(0, 0, 0, 0)。

- 注意：

当有错误发生时，参数*params*中的内容将不发生改变。

当系统支持**GL_ARB_multitexture**扩展时，函数**glGetTexEnv()**将返回当前有效纹理单元的纹理环境参数。

- 出错提示：

当参数*target*或*pname*不是一个可接受的值时产生**GL_INVALID_ENUM**提示。

当函数**glGetTexEnv()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 请参阅：

glActiveTextureARB(), **glTexEnv()**

• **glGetTexGen**

- 名称:

glGetTexGendv(), **glGetTexGenfv()**, **glGetTexGeniv()**

- 功能:

返回纹理坐标生成参数。

- C描述:

```
void glGetTexGendv( GLenum coord,
                    GLenum pname,
                    GLdouble *params )
```

```
void glGetTexGenfv( GLenum coord,
                    GLenum pname,
                    GLfloat *params )
```

```
void glGetTexGeniv( GLenum coord,
                    GLenum pname,
                    GLint *params )
```

- 参数说明:

coord 指定一个纹理坐标。它必须是**GL_S**, **GL_T**, **GL_R**或**GL_Q**。

pname 指定返回值的符号名称。它必须是**GL_TEXTURE_GEN_MODE**, 或是纹理生成平面方程: **GL_OBJECT_PLANE**或**GL_EYE_PLANE**的名称。

params 返回所需的数据。

- 说明:

函数**glGetTexGen()**将由函数**glTexGen()**指定的纹理坐标生成函数的选定参数值返回到*params*中。参数*coord*用符号常量**GL_S**, **GL_T**, **GL_R**或**GL_Q**来指定纹理坐标(*s*, *t*, *r*, *q*)中的一个坐标。

参数*pname*可以取下面三个符号名称之一:

GL_TEXTURE_GEN_MODE	参数 <i>params</i> 返回一个单值的纹理生成函数, 它是一个符号常量。其缺省值是 GL_EYE_LINEAR 。
----------------------------	---

GL_OBJECT_PLANE	参数 <i>params</i> 返回四个平面方程系数, 它们用来确定线性对象坐标的生成。如果需要整数, 将直接由内部的浮点值映射。
------------------------	--

GL_EYE_PLANE	参数 <i>params</i> 返回四个平面方程系数, 它们用来确定线性眼坐标的生成。如果需要整数, 将直接由内部的浮点值映射。这些返回的值是包含在眼坐标中的值。除非当调用函数 glTexGen() 时的模式取景矩阵是单位矩阵, 否则这些值将不等于由函数 glTexGen() 指定的值。
---------------------	---

- 注意:

当有错误发生时, 参数*params*中的内容将不发生改变。

当系统支持**GL_ARB_multitexture**扩展时, 函数**glGetTexGen()**将返回当前有效纹理单元的

纹理坐标生成参数。

- 出错提示：

当参数*coord*或*pname*不是一个可接受的值时，产生**GL_INVALID_ENUM**提示。

当函数**glGetTexGen()**在函数对**glBegin()**/**glEnd()**之间执行时，产生**GL_INVALID_OPERATION**提示。

- 请参阅：

glActiveTextureARB(), **glTexGen()**

• **glGetTexImage**

- 名称：

glGetTexImage()

- 功能：

返回一个纹理图像。

- C 描述：

```
void glGetTexImage( GLenum target,
                     GLint level,
                     GLenum format,
                     GLenum type,
                     GLvoid *pixels )
```

- 参数说明：

target 指定哪种纹理将被获取。它可以是**GL_TEXTURE_1D**, **GL_TEXTURE_2D**或**GL_TEXTURE_3D**。

level 指定多重纹理的级别号。0级是图像的基本级。*n*级是第*n*个mipmap的简化图像。

format 指定返回数据的一个像素格式。它所支持的格式有：**GL_RED**、**GL_GREEN**、**GL_BLUE**、**GL_ALPHA**、**GL_RGB**、**GL_BGR**、**GL_RGBA**、**GL_BGRA**、**GL_LUMINANCE**或**GL_LUMINANCE_ALPHA**。

type 指定返回数据的一个像素类型。它所支持的类型有：**GL_UNSIGNED_BYTE**、**GL_BYTE**、**GL_UNSIGNED_SHORT**、**GL_SHORT**、**GL_UNSIGNED_INT**、**GL_INT**、**GL_FLOAT**、**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**。

pixels 指定返回的纹理图像。它必须是一个指向某数组的指针。该数组的类型由*type*指定。

- 说明：

函数`glGetTexImage()`将一个纹理图像返回到`pixels`。参数`target`指定要得到的纹理图像是由`glTexImage1D()`指定的（`GL_TEXTURE_1D`），还是由`glTexImage2D()`指定的（`GL_TEXTURE_2D`），或是由`glTexImage3D()`指定的（`GL_TEXTURE_3D`）。参数`level`指定想得到的多重纹理的级别号。参数`format`和`type`指定想得到的图像数组的格式和类型。关于参数`format`和`type`可取值的说明请参阅函数`glTexImage1D()`和`glDrawPixels()`。

要想理解函数`glGetTexImage()`的操作过程，首先应该认为在一个RGBA颜色缓冲区中选定了一个所需尺寸的内部四组分纹理图像。接下来，函数`glGetTexImage()`所进行的操作的语义就基本上与函数`glReadPixels()`一样了，它除了没有执行像素转换之外，就相当于用相同的`format`和`type`调用了函数`glReadPixels()`。这时将`x`和`y`设置为0，`width`设置为纹理图像的宽度（当边界指定时，它将包含边界），`height`设置为1（1D的图像）或设置为纹理图像的高度（2D的图像。当边界指定时，它将包含边界）。由于内部纹理图像是RGBA图像，所以像素格式`GL_COLOR_INDEX`、`GL_STENCIL_INDEX`和`GL_DEPTH_COMPONENT`不能被接受，像素类型`GL_BITMAP`也不能被接受。

如果所选择的纹理不包含四个组分，则将使用如下的映射方式：RGBA缓冲区将把单组分纹理看作是将红组分设置成单组分值、绿和蓝组分设置成0、alpha组分设置成1形成的；把二组分纹理看作是将红组分设置成组分0、alpha组分设置成组分1、绿和蓝组分设置成0形成的；把三组分纹理看作是将红组分设置成组分0、绿组分设置成组分1、蓝组分设置成组分2、alpha组分设置成1形成的。

如果要决定`pixels`所需的尺寸，首先应通过调用函数`glGetTexLevelParameter()`来决定内部纹理图像的尺寸，然后在`format`和`type`的基础上，再由每个像素所需的存储请求放大相应的像素个数。这时要确保像素存储参数被计数，尤其是`GL_PACK_ALIGNMENT`。

- 注意：

当有错误发生时，`pixels`中的内容将不改变。

当系统支持`GL_ARB_multitexture`扩展时，函数`glGetTexImage()`将返回当前有效纹理单元的纹理图像。

仅在GL 1.2以上的版本中，`GL_UNSIGNED_BYTE_3_3_2`、`GL_UNSIGNED_BYTE_2_3_3_REV`、`GL_UNSIGNED_SHORT_5_6_5`、`GL_UNSIGNED_SHORT_5_6_5_REV`、`GL_UNSIGNED_SHORT_4_4_4_4`、`GL_UNSIGNED_SHORT_4_4_4_4_REV`、`GL_UNSIGNED_SHORT_5_5_5_1`、`GL_UNSIGNED_SHORT_1_5_5_5_REV`、`GL_UNSIGNED_INT_8_8_8_8`、`GL_UNSIGNED_INT_8_8_8_8_REV`、`GL_UNSIGNED_INT_10_10_10_2`和`GL_UNSIGNED_INT_2_10_10_10_REV`才是有效的类型值。

- 出错提示：

当参数`target`、`format`或`type`不是一个可接受的值时产生`GL_INVALID_ENUM`提示。

当参数`level`小于0时产生`GL_INVALID_VALUE`提示。

当参数`level`大于`log2max`时产生`GL_INVALID_VALUE`提示。此处`max`是`GL_MAX_TEXTURE_SIZE`的返回值。

当函数`glGetTexImage()`在函数对`glBegin()`/`glEnd()`之间执行时产生`GL_INVALID_`

OPERATION提示。

当参数`type`为**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**之一，但参数`format`却不是**GL_RGB**格式时产生**GL_INVALID_OPERATION**提示。

当参数`type`为**GL_UNSIGNED_SHORT_4_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**之一，但参数`format`既不是**GL_RGBA**格式又不是**GL_BGRA**格式时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

```
glGetTexLevelParameter( GL_TEXTURE_WIDTH )
glGetTexLevelParameter( GL_TEXTURE_HEIGHT )
glGetTexLevelParameter( GL_TEXTURE_BORDER )
glGetTexLevelParameter( GL_TEXTURE_INTERNALFORMAT )
glGet( GL_PACK_ALIGNMENT )
```

- 请参阅：

```
glActiveTextureARB(), glDrawPixels(), glReadPixels(), glTexEnv(), glTexGen(), glTexImage1D(),
glTexImage2D(), glTexImage3D(), glTexSubImage1D(), glTexSubImage2D(),
glTexSubImage3D(), glTexParameter()
```

• **glGetTexLevelParameter**

- 名称：

```
glGetTexLevelParameterfv(), glGetTexLevelParameteriv()
```

- 功能：

返回指定分辨率级别的纹理参数值。

- C描述：

```
void glGetTexLevelParameterfv( GLenum target,
                               GLint level,
                               GLenum pname,
                               GLfloat *params )
void glGetTexLevelParameteriv( GLenum target,
                               GLint level,
                               GLenum pname,
                               GLint *params )
```

- 参数说明：

`target` 指定目标纹理的符号名称。它可以是**GL_TEXTURE_1D**、**GL_TEXTURE_2D**、**GL_TEXTURE_3D**、**GL_PROXY_TEXTURE_1D**、**GL_PROXY_**

TEXTURE_2D或GL_PROXY_TEXTURE_3D。

level 指定多重纹理的级别号。0级是图像的基本级。*n*级是第*n*个mipmap的简化图像。

pname 指定一个纹理参数的符号名称。它可取的值有：GL_TEXTURE_WIDTH、GL_TEXTURE_HEIGHT、GL_TEXTURE_DEPTH、GL_TEXTURE_INTERNAL_FORMAT、GL_TEXTURE_BORDER、GL_TEXTURE_RED_SIZE、GL_TEXTURE_GREEN_SIZE、GL_TEXTURE_BLUE_SIZE、GL_TEXTURE_ALPHA_SIZE、GL_TEXTURE_LUMINANCE_SIZE和GL_TEXTURE_INTENSITY_SIZE。

params 返回被请求的数据。

• 说明：

函数glGetTexLevelParameter()把参数*level*指定的多级分辨率的纹理参数值返回到*params*中。参数*target*用来定义目标纹理，它的可取值有：GL_TEXTURE_1D、GL_TEXTURE_2D、GL_TEXTURE_3D、GL_PROXY_TEXTURE_1D、GL_PROXY_TEXTURE_2D或GL_PROXY_TEXTURE_3D。

用GL_MAX_TEXTURE_SIZE和GL_MAX_3D_TEXTURE_SIZE并不能充分说明该图像。这里必须提供能容纳mipmap和边界的正方形纹理图像的最大值，而不是一个长的表面纹理或一个不带mipmap和边界的纹理。这样它才能更容易地适应纹理存储器。对于代理目标而言，用户可以更精确地查询GL是否能容纳一个给定配置的纹理。当该纹理不能被容纳时，可由函数glGetTexLevelParameter()查询的纹理状态变量将被设置为0。如果该纹理是可以被容纳的，纹理状态值将被设置成非代理目标时所设置的值。

参数*pname*指定将返回的纹理参数值。

可接受的参数名称如下：

GL_TEXTURE_WIDTH

参数*params*返回一个值，它表示纹理图像的宽度。该值包含纹理图像的边界。其缺省值是0。

GL_TEXTURE_HEIGHT

参数*params*返回一个值，它表示纹理图像的高度。该值包含纹理图像的边界。其缺省值是0。

GL_TEXTURE_DEPTH

参数*params*返回一个值，它表示纹理图像的深度。该值包含纹理图像的边界。其缺省值是0。

GL_TEXTURE_INTERNAL_FORMAT

参数*params*返回一个值，它表示纹理图像的内部格式。

GL_TEXTURE_BORDER

参数*params*返回一个值，它表示纹理图像边界像素的宽度。其缺省值是0。

GL_TEXTURE_RED_SIZE**GL_TEXTURE_GREEN_SIZE**

GL_TEXTURE_BLUE_SIZE
GL_TEXTURE_ALPHA_SIZE
GL_TEXTURE_LUMINANCE_SIZE
GL_TEXTURE_INTENSITY_SIZE:

各组分的内部存储分辨率。由GL选择的分辨率将是用户所需求的分辨率的一个最接近的匹配值。用户可以通过函数glTexImage1D()、glTexImage2D()、glTexImage3D()、glCopyTexImage1D和glCopyTexImage2D()来设置分辨率。各组分的内部存储分辨率的缺省值是0。

- 注意：

当有错误发生时，参数*params*中的内容将不改变。

GL_TEXTURE_INTERNAL_FORMAT在GL 1.1以上的版本中才有效。在1.0版本中，由**GL_TEXTURE_COMPONENTS**代替。

GL_PROXY_TEXTURE_1D和**GL_PROXY_TEXTURE_2D**在GL 1.1以上的版本中才有效。

GL_TEXTURE_3D、**GL_PROXY_TEXTURE_3D**和**GL_TEXTURE_DEPTH**只有在GL 1.2以上的版本中才有效。

当系统支持**GL_ARB_multitexture**扩展时，函数glGetTexLevelParameter()将返回当前有效纹理单元的纹理级参数。

- 出错提示：

当参数*target*或*pname*不是一个可接受值时产生**GL_INVALID_ENUM**提示。

当参数*level*小于0时产生**GL_INVALID_VALUE**提示。

当参数*level*大于log₂*max*时产生**GL_INVALID_VALUE**提示。此处*max*是**GL_MAX_TEXTURE_SIZE**的返回值。

当函数glGetTexLevelParameter()在函数对glBegin()/glEnd()之间执行时产生**GL_INVALID_OPERATION**提示。

- 请参阅：

glActiveTextureARB(), glGetTexParameter(), glCopyTexImage1D(), glCopyTexImage2D(),
 glCopyTexSubImage1D(), glCopyTexSubImage2D(), glCopyTexSubImage3D(), glTexEnv(),
 glTexGen(), glTexImage1D(), glTexImage2D(), glTexImage3D(), glTexSubImage1D(),
 glTexSubImage2D(), glTexSubImage3D(), glTexParameter()

■ glGetTexParameter

- 名称：

glGetTexParameterfv(), glGetTexParameteriv()

- 功能：

返回纹理参数值。

- C描述：

void glGetTexParameterfv(GLenum *target*,

```

        GLenum pname,
        GLfloat *params )

void glGetTexParameteriv( GLenum target,
                         GLenum pname,
                         GLint *params )

```

• 参数说明：

target 指定目标纹理的符号名称。它的可接受值有**GL_TEXTURE_1D**、**GL_TEXTURE_2D**或**GL_TEXTURE_3D**。

pname 指定纹理参数的符号名称。它的可取值有：**GL_TEXTURE_MAG_FILTER**、**GL_TEXTURE_MIN_FILTER**、**GL_TEXTURE_MIN_LOD**、**GL_TEXTURE_MAX_LOD**、**GL_TEXTURE_BASE_LEVEL**、**GL_TEXTURE_MAX_LEVEL**、**GL_TEXTURE_WRAP_S**、**GL_TEXTURE_WRAP_T**、**GL_TEXTURE_WRAP_R**、**GL_TEXTURE_BORDER_COLOR**、**GL_TEXTURE_PRIORITY**和**GL_TEXTURE_RESIDENT**。

params 返回纹理参数值。

• 说明：

函数glGetTexParameter()把由参数*pname*指定的纹理参数值返回到*params*中。参数*target*用来定义目标纹理，它的可取值有**GL_TEXTURE_1D**、**GL_TEXTURE_2D**或**GL_TEXTURE_3D**，分别用于指定一维、二维或三维的纹理操作。参数*pname*的可接受符号常量与glTexParameter()相同，其含义也是一样的：

GL_TEXTURE_MAG_FILTER:

返回一个单值的纹理放大滤波器，它是一个符号常量。其缺省值是**GL_LINEAR**。

GL_TEXTURE_MIN_FILTER

返回一个单值的纹理缩小滤波器，它是一个符号常量。其缺省值是**GL_NEAREST_MIPMAP_LINEAR**。

GL_TEXTURE_MIN_LOD

返回一个单值的最小级别值。其缺省值是-1000。

GL_TEXTURE_MAX_LOD

返回一个单值的最大级别值。其缺省值是1000。

GL_TEXTURE_BASE_LEVEL

返回一个单值的纹理mipmap的基本级。其缺省值是0。

GL_TEXTURE_MAX_LEVEL

返回一个单值的最大纹理mipmap的数组级。其缺省值是1000。

GL_TEXTURE_WRAP_S

返回纹理坐标s的一个单值缠绕函数，它是一个符号常量。其缺省值是**GL_REPEAT**。

GL_TEXTURE_WRAP_T

返回纹理坐标 t 的一个单值缠绕函数，它是一个符号常量。其缺省值是**GL_REPEAT**。

GL_TEXTURE_WRAP_R

返回纹理坐标 r 的一个单值缠绕函数，它是一个符号常量。其缺省值是**GL_REPEAT**。

GL_TEXTURE_BORDER_COLOR

返回四个整型或浮点型数值，它们包含了纹理边界的RGBA颜色组分。返回的浮点值在范围[0, 1]内。如果这里需要整数，内部的浮点值将按下述方法进行线性映射：1.0映射为最大的正整数，-1.0映射为最小的负整数。其缺省值是(0, 0, 0, 0)。

GL_TEXTURE_PRIORITY

返回目标纹理（或与它相连接的纹理）的驻留优先级。其缺省值是1。请参阅**glPrioritizeTextures()**。

GL_TEXTURE_RESIDENT

返回目标纹理的驻留状态。如果返回到*params*中的值是**GL_TRUE**，则该纹理驻留在纹理存储器中。请参阅**glAreTextureResident()**。

- 注意：

GL_TEXTURE_PRIORITY和**GL_TEXTURE_RESIDENT**只有在GL1.1以上的版本中才有效。

GL_TEXTURE_3D, **GL_TEXTURE_MIN_LOD**, **GL_TEXTURE_MAX_LOD**, **GL_TEXTURE_BASE_LEVEL**, **GL_TEXTURE_MAX_LEVEL**和**GL_TEXTURE_WRAP_R**只在GL 1.2以上版本中才有效。

当有错误发生时，参数*params*中的内容将不发生改变。

- 出错提示：

当参数*target*或*pname*不是一个可接受值时产生**GL_INVALID_ENUM**提示。

当函数**glGetTexParameter()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 请参阅：

glAreTextureResident(), **glPrioritizeTextures()**, **glTexParameter()**

- **glHint**

- 名称：

glHint()

- 功能：

指定一个提示，定义特定OpenGL实现的绘图或图像保真的质量。

- C 描述:

```
void glHint( GLenum target,
             GLenum mode )
```

- 参数说明:

target 指定一个符号常量, 说明哪些行为将受到控制。它可以是**GL_FOG_HINT**, **GL_LINE_SMOOTH_HINT**, **GL_PERSPECTIVE_CORRECTION_HINT**, **GL_POINT_SMOOTH_HINT** 和**GL_POLYGON_SMOOTH_HINT**。

mode 指定一个符号常量, 说明所需要采取的方法。它可以是**GL_FASTEST**, **GL_NICEST** 和**GL_DONT_CARE**。

- 说明:

GL的某些操作, 如果有编译所需的空间, 就可以通过提示来控制某些特殊的操作。每个提示都由两个自变量来指定。参数*target*是一个符号常量, 用来说明被控制的行为。参数*mode*是另一个符号常量, 用来说明希望得到的行为质量。每个参数*target*的缺省值都是整型常量**GL_DONT_CARE**。参数*mode*可取的值如下:

GL_FASTEST 选择最有效的操作。

GL_NICEST 选择最正确的或质量最好的操作。

GL_DONT_CARE 不优先选择。

尽管可以充分地定义那些能被提示的实现方式, 但各种提示的编译却是跟特定实现有关的。这些可以由参数*target*指定的提示方式及其含义如下:

GL_FOG_HINT 表示雾化计算操作的精确性。如果你的GL系统不能有效地支持对每一个像素点的雾化操作, 选择提示**GL_DONT_CARE**或**GL_FASTEST**可影响对每个顶点的雾化效果的计算。

GL_LINE_SMOOTH_HINT

表示反走样线的采样质量。采用的过滤方程越多, 提示参数**GL_NICEST**在光栅化时产生的像素片断就越多。

GL_PERSPECTIVE_CORRECTION_HINT

指定颜色和纹理坐标插值的质量。如果你的GL系统不能有效地支持透视校正参数插值法, 选用提示参数**GL_DONT_CARE**或**GL_FASTEST**可产生颜色和/或纹理坐标的简单线性插值。

GL_POINT_SMOOTH_HINT

指定反走样点的采样质量。采用的过滤方程越多, 提示参数**GL_NICEST**在光栅化时产生的像素片断就越多。

GL_POLYGON_SMOOTH_HINT

指定反走样多边形的采样质量。采用的过滤方程越多, 提示参数**GL_NICEST**在光栅化时产生的像素片断就越多。

- 注意:

各种提示的编译与特定OpenGL实现有关, 有些实现忽略glHint()操作。

- 出错提示：

当参数*target*或*mode*不是一个可接受的值时产生**GL_INVALID_ENUM**提示。

当函数**glHint()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

* **glHistogram**

- 名称：

glHistogram()

- 功能：

定义直方图表。

- C描述：

```
void glHistogram( GLenum target,
                  GLsizei width,
                  GLenum internalformat,
                  GLboolean sink )
```

- 参数说明：

target 指定其参数将被设置的直方图。它必须是**GL_HISTOGRAM**或**GL_PROXY_HISTOGRAM**。

width 指定直方图表中条目的个数。它必须是2的幂值。

internalformat

指定直方图中条目的格式。它必须是下面所列的符号常量之一：**GL_ALPHA**、**GL_ALPHA4**、**GL_ALPHA8**、**GL_ALPHA12**、**GL_ALPHA16**、**GL_LUMINANCE**、**GL_LUMINANCE4**、**GL_LUMINANCE8**、**GL_LUMINANCE12**、**GL_LUMINANCE16**、**GL_LUMINANCE_ALPHA**、**GL_LUMINANCE4_ALPHA4**、**GL_LUMINANCE6_ALPHA2**、**GL_LUMINANCE8_ALPHA8**、**GL_LUMINANCE12_ALPHA4**、**GL_LUMINANCE12_ALPHA12**、**GL_LUMINANCE16_ALPHA16**、**GL_R3_G3_B2**、**GL_RGB**、**GL_RGB4**、**GL_RGB5**、**GL_RGB8**、**GL_RGB10**、**GL_RGB12**、**GL_RGB16**、**GL_RGBA**、**GL_RGBA2**、**GL_RGBA4**、**GL_RGB5_A1**、**GL_RGBA8**、**GL_RGB10_A2**、**GL_RGBA12**或**GL_RGBA16**。

sink 当它是**GL_TRUE**时，像素在经过直方图操作后将被消耗，这时没有绘制或像素载入操作发生。如果它是**GL_FALSE**，在直方图操作结束后，还将对像素进行最值操作。

- 说明：

当启动**GL_HISTOGRAM**功能后，RGBA颜色组分将转化成直方图表索引。该转化过程是首先将其截断到范围[0, 1]内，然后乘上直方图表的宽度并将其圆整成最接近的整数。这样由RGBA索引选定的直方图表的条目将会增加。（如果直方图的内部格式中包含亮度值，则亮度表

条目的增加值将由R颜色组分所得的索引来决定。) 如果直方图表条目的增加值超出了它的最大值，则它的值将成为未定义的(但这并不是一个错误)。

直方图处理仅对RGBA像素有效(这些像素可以是由最初指定的颜色索引通过索引查询表转化而来)。启动与关闭直方图功能，请使用函数glEnable(GL_HISTOGRAM)和glDisable(GL_HISTOGRAM)。

当target是GL_HISTOGRAM时，函数glHistogram()将重新定义当前直方图表包含width个条目，条目的格式由internalformat指定。这时条目将由0到width-1来检索，并被初始化为0。先前的直方图条目中如果有值的话，这些值将丢失。如果sink是GL_TRUE，则直方图操作完成后，像素将被丢弃，不再对像素进行其他、的操作，也不再有绘制、像素载入或像素反馈发生。

当target是GL_PROXY_HISTOGRAM时，函数glHistogram()将计算所有的状态信息，就好象直方图表被重新定义过一样，但实际上并不重新定义新表。如果所需要的直方图表太大，系统无法支持，则状态信息将被设置为0。这里也就提供了一种判断给定参数的直方图表是否被支持的方法。

- 注意：

当调用函数glGetString(GL_EXTENSIONS)的返回值是GL_ARB_imaging时，才提供函数glHistogram()。

- 出错提示：

当参数target不是一个允许值时产生GL_INVALID_ENUM提示。

当参数width小于0或不是一个2的幂值时，产生GL_INVALID_VALUE提示。

当参数internalformat不是一个允许值时产生GL_INVALID_ENUM提示。

当参数target为GL_HISTOGRAM，并且指定的直方图表对于实现而言太大时，产生GL_TABLE_TOO_LARGE提示。

当函数glHistogram()在函数对glBegin()/glEnd()之间执行时产生GL_INVALID_OPERATION提示。

- 有关数据的获取：

glGetHistogramParameter()

- 请参阅：

glGetHistogram(), glResetHistogram()

• glIndex

- 名称：

glIndexd(), glIndexf(), glIndexi(), glIndexes(), glIndexub(), glIndexdv(), glIndexfv(), glIndexiv(), glIndexsv(), glIndexubv()

- 功能：

设置当前颜色索引。

- C描述：

`void glIndexd(GLdouble c)`

```
void glIndexf( GLfloat c )
void glIndexi( GLint c )
void glIndexes( GLshort c )
void glIndexub( GLubyte c )
```

- 参数说明：

c 指定当前颜色索引的新值。

- C描述：

```
void glIndexdv( const GLdouble *c )
void glIndexfv( const GLfloat *c )
void glIndexiv( const GLint *c )
void glIndexsv( const GLshort *c )
void glIndexubv( const GLubyte *c )
```

- 参数说明：

c 指定一个指针，指向一个一元数组，该数组包含当前颜色索引的新值。

- 说明：

函数glIndex()用来更新当前的颜色索引（单值）。它只带有一个自变量——当前颜色索引的新值。

当前颜色索引值是以浮点形式存放的。整型值未经任何特殊映射，直接转化为浮点值。其缺省值是1。

在颜色索引缓冲区所代表的范围之外的索引值不被截断。然而，在一个索引被抖动（如果允许抖动）并被写入帧缓冲区之前，它会被转化成定点格式。所得的定点值的整数部分中的二进制位如果不能同帧缓冲区中的位相对应，它们将被屏蔽掉。

- 注意：

函数glIndexub()和glIndexubv()只有在GL 1.1以上的版本中才有效。

当前的索引可被随时更新。尤其，函数glIndex()可在函数对glBegin()/glEnd()中被调用。

- 有关数据的获取：

glGet(GL_CURRENT_INDEX)

- 请参阅：

glColor(), glIndexPointer()

◆ glIndexMask

- 名称：

glIndexMask()

- 功能：

控制颜色索引缓冲区中各自的二进制位的写操作。

- C描述：

```
void glIndexMask( GLuint mask )
```

- 参数说明：

mask 指定一个位屏蔽，它表示是否可以对颜色索引缓冲区中的每个单独的二进制位进行写入操作。缺省情况下的屏蔽全是1。

- 说明：

函数glIndexMask()用来控制颜色索引缓冲区中各自的二进制位的写操作。参数*mask*中的至少*n*个有意义的位指定了一个屏蔽，此处*n*是颜色索引缓冲区中二进制位的数目。如果在屏蔽中出现的是1，则表示颜色索引缓冲区中相应的位允许写入；相反，如果出现的是0，则表示颜色索引缓冲区中相应的位是写保护的。

这种屏蔽只能用于颜色索引模式，并且它仅对当前选定用来写入的缓冲区有效（请参阅glDrawBuffer()）。缺省情况下，所有的位都允许写入。

- 出错提示：

当函数glIndexMask()在函数对glBegin() / glEnd()之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGet(GL_INDEX_WRITEMASK)

- 请参阅：

glColorMask(), glDepthMask(), glDrawBuffer(), glIndex(), glIndexPointer(),
glStencilMask()

• glIndexPointer

- 名称：

glIndexPointer()

- 功能：

定义一个颜色索引数组。

- C描述：

```
void glIndexPointer( GLenum type,
                     GLsizei stride,
                     const GLvoid *pointer )
```

- 参数说明：

type 指定数组中每个颜色索引的数据类型。它可取的符号常量有：**GL_UNSIGNED_BYTE**、**GL_SHORT**、**GL_INT**、**GL_FLOAT**和**GL_DOUBLE**。其缺省值是**GL_FLOAT**。

stride 指定相邻的颜色索引之间的字节偏移量。如果参数*stride*是0（默认值），则认为颜色索引被一个接一个的放入在数组中。

pointer 指定一个指针，指向数组中的第一个颜色索引。其缺省值是0。

- 说明：

函数glIndexPointer()的作用是绘图时指定一个颜色索引数组的存放位置和数据格式。参数

*type*用来指定每个颜色索引的数据类型，参数*stride*给出了允许的顶点和属性存放在一个数组中或分别存放在多个数组中的一个颜色索引到下一个颜色索引之间的字节偏移量。(在一些实现中，单一数组的存放形式可能更有效。请参阅glInterleavedArrays()。)

参数*type*、*stride*和*pointer*都被存储为客户端状态。

缺省情况下，颜色索引数组是关闭的。用函数glEnableClientState(GL_INDEX_ARRAY)和glDisableClientState(GL_INDEX_ARRAY)可以启动和关闭颜色索引数组。当启动颜色索引数组后，颜色索引数组可被函数glDrawArrays()、glDrawElements()或glArrayElement()使用。

函数glDrawArrays()用来将预先指定的顶点和顶点属性数组构造成一个图元序列（所有相同的类型）。函数glArrayElement()可以通过检索顶点和顶点属性来指定图元。函数glDrawElements()则是通过检索顶点和顶点属性来构造一个图元序列。

- 注意：

函数glIndexPointer()只有在GL 1.1以上版本中才有效。

颜色索引数组在缺省情况下是关闭的，这时不能调用函数glArrayElement()、glDrawElement()或glDrawArrays()来访问它。

函数glIndexPointer()不允许在函数对glBegin()/glEnd()之间执行，否则可能产生一个出错提示，但也有可能不产生出错提示。如果没有产生出错提示，其操作将是未定义的。

函数glIndexPointer()是典型的客户端执行方式。

因为颜色索引数组参数是客户端状态，所以它不能用函数glPushAttrib()和glPopAttrib()存储和恢复，而应该用函数glPushClientAttrib()和glPopClientAttrib()代替。

- 出错提示：

当参数*type*不是一个可取的值时产生GL_INVALID_ENUM提示。

当参数*stride*是负数时产生GL_INVALID_VALUE提示。

- 有关数据的获取：

```
glIsEnabled( GL_INDEX_ARRAY )
glGet( GL_INDEX_ARRAY_TYPE )
glGet( GL_INDEX_ARRAY_STRIDE )
glGetPointerv( GL_INDEX_ARRAY_POINTER )
```

- 请参阅：

glArrayElement(), glColorPointer(), glDrawArrays(), glDrawElements(), glEdgeFlagPointer(),
glEnable(), glGetPointerv(), glInterleavedArrays(), glNormalPointer(), glPopClientAttrib(),
glPushClientAttrib(), glTexCoordPointer(), glVertexPointer()

• glInitNames

- 名称：

glInitNames()

- 功能：

初始化名称堆栈。

- C 描述:

```
void glInitNames( void )
```

- 说明:

在选择模式中, 名称堆栈是区别绘图命令集中的命令的唯一标志。它由一个有序的无符号整数集所构成。函数glInitNames()的作用是初始化名称堆栈, 即将它初始化成其默认的空堆栈。

如果绘图模式不是GL_SELECT, 名称堆栈总是空的。这时函数glInitNames()将被忽略。

- 出错提示:

当函数glInitNames()在函数对glBegin()/glEnd()之间执行时产生GL_INVALID_OPERATION提示。

- 有关数据的获取:

```
glGet( GL_NAME_STACK_DEPTH )
```

```
glGet( GL_MAX_NAME_STACK_DEPTH )
```

- 请参阅:

glLoadName(), **glPushName()**, **glRenderMode()**, **glSelectBuffer()**

glInterleavedArrays

- 名称:

glInterleavedArrays()

- 功能:

同时指定并启动几个交叉存取的数组。

- C 描述:

```
void glInterleavedArrays( GLenum format,
                           GLsizei stride,
                           const GLvoid *pointer)
```

- 参数说明:

format 指定要启动的数组的类型。它可以是以下符号常量之一: **GL_V2F**、**GL_V3F**、**GL_C4UB_V2F**、**GL_C4UB_V3F**、**GL_C3F_V3F**、**GL_N3F_V3F**、**GL_C4F_N3F_V3F**、**GL_T2F_V3F**、**GL_T4F_V4F**、**GL_T2F_C4UB_V3F**、**GL_T2F_C3F_V3F**、**GL_T2F_N3F_V3F**、**GL_T2F_C4F_N3F_V3F**和**GL_T4F_C4F_N3F_V4F**。

stride 指定各数组元素集合之间的字节偏移量。

pointer 指定一个指针, 指向第一个数组元素集合的一个元素。其缺省值是0。

- 说明:

当颜色、法线、纹理和顶点数组的元素是整个数组元素集合的一部分时, 函数glInterleavedArrays()允许你分别指定并启动各个数组。在某些实现过程中, 这种方法可能比分别指定各个数组更有效。

当参数*stride*是0时, 该元素集合将连续存储。否则, 两个数组元素集合的开始位置之间将有

*stride*的字节偏移量。

参数*format*可作为一个“关键字，”描述从数组集中提取各个分离的数组。当参数*format*中包含有字母T时，将从交叉存取的数组中提取纹理坐标；如果包含的是C，将提取颜色值；如果是N，则提取法线坐标。顶点坐标总是被提取。

阿拉伯数字2、3和4用来指定提取值的个数。F表示提取值是浮点型的。当字母C后面跟有字母4UB时，就表示将提取的是4个无符号的字节。如果一个颜色值是作为4个无符号字节来提取的，其后面的顶点数组元素将存入第一个可能的浮点值所对应的地址中。

- 注意：

函数glInterleavedArrays()仅在GL 1.1以上版本中才有效。

如果编辑一个显示列表时调用了函数glInterleavedArrays()，它将立即执行而不被编入显示列表。

函数glInterleavedArrays()不允许在函数对glBegin()/glEnd()之间执行，但即使这样做了，也可能不产生出错提示，但这时操作将是未定义的。

函数glInterleavedArrays()是通常在客户端实现。

由于顶点数组参数是客户端状态，因此不能用函数glPushAttrib()和glPopAttrib()来存储和恢复，而应该使用函数glPushClientAttrib()和glPopClientAttrib()代替。

当系统支持GL_ARB_multitexture扩展时，函数glInterleavedArrays()仅更新纹理坐标数组中当前有效的纹理单元。而对于其他客户纹理单元的纹理坐标状态将不更新，这时并不考虑它们是否已被启动。

- 出错提示：

当参数*format*不是一个可接受值时产生GL_INVALID_ENUM提示。

当参数*stride*是负数时产生GL_INVALID_VALUE提示。

- 请参阅：

glArrayElement(), glClientActiveTextureARB(), glColorPointer(), glDrawArrays(),
glDrawElements(), glEdgeFlagPointer(), glEnableClientState(), glGetPointer(),
glIndexPointer(), glNormalPointer(), glTexCoordPointer(), glVertexPointer()

■ glIsEnabled

- 名称：

glIsEnabled()

- 功能：

测试一种功能是否已启动。

- C描述：

GLboolean glIsEnabled(GLenum *cap*)

- 参数说明：

cap 指定一个符号常量，该常量用来表示一种GL功能。

- 说明：

当参数*cap*是一种已启动的功能时，函数glIsEnabled()将返回GL_TRUE。否则，返回GL_FALSE。缺省情况下，除GL_DITHER外，所有功能都是关闭的；GL_DITHER在缺省情况下是启动的。

参数*cap*可接受如下的功能，见表5-21。

表 5-21

常量	请参阅
GL_ALPHA_TEST	glAlphaFunc()
GL_AUTO_NORMAL	glEvalCoord()
GL_BLEND	glBlendFunc(), glLogicOp()
GL_CLIP_PLANEi	glClipPlane()
GL_COLOR_ARRAY	glColorPointer()
GL_COLOR_LOGIC_OP	glLogicOp()
GL_COLOR_MATERIAL	glColorMaterial()
GL_COLOR_TABLE	glColorTable()
GL_CONVOLUTION_1D	glConvolutionFilter1D()
GL_CONVOLUTION_2D	glConvolutionFilter2D()
GL_CULL_FACE	glCullFace()
GL_DEPTH_TEST	glDepthFunc(), glDepthRange()
GL_DITHER	glEnable()
GL_EDGE_FLAG_ARRAY	glEdgeFlagPointer()
GL_FOG	glFog()
GL_HISTOGRAM	glHistogram()
GL_INDEX_ARRAY	glIndexPointer()
GL_INDEX_LOGIC_OP	glLogicOp()
GL_LIGHTi	glLightModel(), glLight()
GL_LIGHTING	glMaterial(), glLightModel(), glLight()
GL_LINE_SMOOTH	glLineWidth()
GL_LINE_STIPPLE	glLineStipple()
GL_MAP1_COLOR_4	glMap1()
GL_MAP1_INDEX	glMap1()
GL_MAP1_NORMAL	glMap1()
GL_MAP1_TEXTURE_COORD_1	glMap1()
GL_MAP1_TEXTURE_COORD_2	glMap1()
GL_MAP1_TEXTURE_COORD_3	glMap1()
GL_MAP1_TEXTURE_COORD_4	glMap1()
GL_MAP2_COLOR_4	glMap2()
GL_MAP2_INDEX	glMap2()
GL_MAP2_NORMAL	glMap2()
GL_MAP2_TEXTURE_COORD_1	glMap2()
GL_MAP2_TEXTURE_COORD_2	glMap2()
GL_MAP2_TEXTURE_COORD_3	glMap2()
GL_MAP2_TEXTURE_COORD_4	glMap2()
GL_MAP2_VERTEX_3	glMap2()

(续)

常量	请参阅
GL_MAP2_VERTEX_4	glMap2()
GL_MINMAX	glMinmax()
GL_NORMAL_ARRAY	glNormalPointer()
GL_NORMALIZE	glNormal()
GL_POINT_SMOOTH	glPointSize()
GL_POLYGON_SMOOTH	glPolygonMode()
GL_POLYGON_OFFSET_FILL	glPolygonOffset()
GL_POLYGON_OFFSET_LINE	glPolygonOffset()
GL_POLYGON_OFFSET_POINT	glPolygonOffset()
GL_POLYGON_STIPPLE	glPolygonStipple()
GL_POST_COLOR_MATRIX_COLOR_TABLE	glColorTable()
GL_POST_CONVOLUTION_COLOR_TABLE	glColorTable()
GL_RESCALE_NORMAL	glNormal()
GL_SCISSOR_TEST	glScissor()
GL_SEPARABLE_2D	glSeparableFilter2D()
GL_STENCIL_TEST	glStencilFunc(), glStencilOp()
GL_TEXTURE_1D	glTexImage1D()
GL_TEXTURE_2D	glTexImage2D()
GL_TEXTURE_3D	glTexImage3D()
GL_TEXTURE_COORD_ARRAY	glTexCoordPointer()
GL_TEXTURE_GEN_Q	glTexGen()
GL_TEXTURE_GEN_R	glTexGen()
GL_TEXTURE_GEN_S	glTexGen()
GL_TEXTURE_GEN_T	glTexGen()
GL_VERTEX_ARRAY	glVertexPointer()

• 注意：

当有错误发生时，函数glIsEnabled()返回0。

GL_COLOR_LOGIC_OP、**GL_COLOR_ARRAY**、**GL_EDGE_FLAG_ARRAY**、**GL_INDEX_ARRAY**、**GL_INDEX_LOGIC_OP**、**GL_NORMAL_ARRAY**、**GL_POLYGON_OFFSET_FILL**、**GL_POLYGON_OFFSET_LINE**、**GL_POLYGON_OFFSET_POINT**、**GL_TEXTURE_COORD_ARRAY**、**GL_VERTEX_ARRAY**仅在GL 1.1以上版本中才有效。

GL_RESCALE_NORMAL和**GL_TEXTURE_3D**仅在GL 1.2以上版本中才有效。

当调用函数glGet(GL_EXTENSIONS)的返回值是**GL_ARB_imaging**时，**GL_COLOR_TABLE**、**GL_CONVOLUTION_1D**、**GL_CONVOLUTION_2D**、**GL_HISTOGRAM**、**GL_MINMAX**、**GL_POST_COLOR_MATRIX_COLOR_TABLE**、**GL_POST_CONVOLUTION_COLOR_TABLE**和**GL_SEPARABLE_2D**才有效。

当系统支持**GL_ARB_multitexture**扩展时，下面的参数返回与当前有效的纹理单元相应的值：**GL_TEXTURE_1D**、**GL_TEXTURE_BINDING_1D**、**GL_TEXTURE_2D**、**GL_TEXTURE_BINDING_2D**、**GL_TEXTURE_3D**、**GL_TEXTURE_BINDING_3D**、**GL_TEXTURE_GEN_S**、**GL_TEXTURE_GEN_T**、**GL_TEXTURE_GEN_R**、

GL_TEXTURE_GEN_Q、**GL_TEXTURE_MATRIX**和**GL_TEXTURE_STACK_DEPTH**。类似地，下列参数返回与当前有效的客户端纹理单元相应的值：**GL_TEXTURE_COORD_ARRAY**、**GL_TEXTURE_COORD_ARRAY_SIZE**、**GL_TEXTURE_COORD_ARRAY_STRIDE**和**GL_TEXTURE_COORD_ARRAY_TYPE**。

- 出错提示：

当参数*cap*不是一个可接受的值时产生**GL_INVALID_ENUM**提示。

当函数**glIsEnabled()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 请参阅：

glEnable(), **glEnableClientState()**, **glGet()**

• **glIsList**

- 名称：

glIsList()

- 功能：

确定一个名称是否对应于一个显示列表。

- C描述：

GLboolean glIsList(GLuint list)

- 参数说明：

list 指定一个可能的显示列表名称。

- 说明：

如果函数**glIsList()**的返回值是**GL_TRUE**，说明*list*是一个显示列表的名称；否则，说明*list*不是一个显示列表的名称。

- 出错提示：

当函数**glIsList()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 请参阅：

glCallList(), **glCallLists()**, **glDeleteLists()**, **glGenLists()**, **glNewList()**

• **glIsTexture**

- 名称：

glIsTexture()

- 功能：

确定一个名称是否对应于一个纹理。

- C描述：

GLboolean glIsTexture(GLuint texture)

- 参数说明：

texture 指定一个可能是纹理名称的值。

- 说明：

当*texture*是一个当前的纹理名称时，函数glIsTexture()返回GL_TRUE；当*texture*是0，或虽然是一个非0的值，但却不是一个当前的纹理名称时，函数glIsTexture()返回GL_FALSE。

- 注意：

函数glIsTexture()仅在GL 1.1以上的版本中才有效。

- 出错提示：

当函数glIsTexture()在函数对glBegin()/glEnd()之间执行时产生GL_INVALID_OPERATION提示。

- 请参阅：

glBindTexture(), **glCopyTexImage1D()**, **glCopyTexImage2D()**, **glDeleteTextures()**,
glGenTextures(), **glGet()**, **glGetTexParameter()**, **glTexImage1D()**, **glTexImage2D()**,
glTexImage3D(), **glTexParameter()**

glLight

- 名称：

glLightf(), **glLighti()**, **glLightfv()**, **glLightiv()**

- 功能：

设置光源参数。

- C描述：

```
void glLightf( GLenum light,
               GLenum pname,
               GLfloat param )
void glLighti( GLenum light,
               GLenum pname,
               GLint param )
```

- 参数说明：

light 指定一个光源。光源的数目与实现有关，但至少支持八个光源。它们用符号型名称**GL_LIGHTi**相互区别，这里 $0 \leq i < \text{GL_MAX_LIGHTS}$ 。

pname 为光源*light*指定一个单值的光源参数。它可以是**GL_SPOT_EXPONENT**、**GL_SPOT_CUTOFF**、**GL_CONSTANT_ATTENUATION**、**GL_LINEAR_ATTENUATION**和**GL_QUADRATIC_ATTENUATION**。

param 指定光源*light*的参数*pname*的具体设定值。

- C描述：

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

```
GLenum pname,
const GLint *params)
```

• 参数说明：

light 指定一个光源。光源的数目与实现有关，但至少支持八个光源。它们用符号型名称**GL_LIGHT*i***相互区别，这里 $0 \leq i < \text{GL_MAX_LIGHTS}$ 。

pname 为光源*light*指定一个光源参数。它可以是**GL_AMBIENT**、**GL_DIFFUSE**、**GL_SPECULAR**、**GL_POSITION**、**GL_SPOT_CUTOFF**、**GL_SPOT_DIRECTION**、**GL_SPOT_EXPONENT**、**GL_CONSTANT_ATTENUATION**、**GL_LINEAR_ATTENUATION**和**GL_QUADRATIC_ATTENUATION**。

params 指定一个指针，指向光源*light*的参数*pname*的具体设定值。

• 说明：

函数glLight()用来设置单一光源的参数值。参数*light*用来指定光源，它是一个符号型的名称，形如**GL_LIGHT*i***，这里 $0 \leq i < \text{GL_MAX_LIGHTS}$ 。参数*pname*指定为十个光源参数中的一个参数，它也是一个符号型值。参数*params*是一个单值，或一个指向包含新值的数组的指针。

如果要启用和关闭光照算法，请调用函数glEnable(**GL_LIGHT**)和glDisable(**GL_LIGHT**)。缺省情况下，光照算法是关闭的。当启用光照算法后，它将对光照算法产生影响。glEnable(**GL_LIGHT*i***)和glDisable(**GL_LIGHT*i***)用于启用和关闭光源*i*。

可取的十个光源参数为：

GL_AMBIENT 参数*params*包含四个整数值或浮点值，它们用来指定光源的**RGBA**环境光的强度。整型数值被线性地映射为浮点格式：最大的正数映射为1.0，最小的负数映射为-1.0。浮点型数值则直接映射。二者都不被截断。缺省的环境光强度为(0, 0, 0, 1)。

GL_DIFFUSE 参数*params*包含四个整数值或浮点值，它们用来指定光源的**RGBA**散射光的强度。整型数值被线性地映射为浮点格式：最大的正数映射为1.0，最小的负数映射为-1.0。浮点型数值则直接映射。二者都不被截断。光源**GL_LIGHT0**的缺省散射光强度为(1, 1, 1, 1)，其他光源的缺省散射光强度为(0, 0, 0, 0)。

GL_SPECULAR 参数*params*包含四个整数值或浮点值，它们用来指定光源的**RGBA**反射光的强度。整型数值被线性地映射为浮点格式：最大的正数映射为1.0，最小的负数映射为-1.0。浮点型数值则直接映射。二者都不被截断。光源**GL_LIGHT0**的缺省散射光强度为(1, 1, 1, 1)，其他光源的缺省散射光强度为(0, 0, 0, 0)。

GL_POSITION 参数*params*包含四个整数值或浮点值，它们用来指定光源在齐次对象坐标中的位置。整型数值和浮点数值都直接映射。二者都不被截断。当函数glLight()被调用的时候，光源的位置将通过模式取景矩阵转换得到（就好象它是一个点），并被储存为一个眼坐标。当位置中的w成分为0时，该光源被认为是一个方向光源。这时对光进行的散射和反射计算将使用

其方向，与它的实际位置无关。并且，这时不对光进行衰减处理。相反，如果w不是0，对光进行的散射和反射计算则是基于它在眼坐标中的实际位置，且需要对它进行衰减处理。缺省情况下，光源的位置是(0, 0, 1, 0)。也就是说，缺省的光源是方向光源，它们相互平行，并且沿-z轴方向。

GL_SPOT_DIRECTION

参数*params*包含三个整数值或浮点值，它们用来指定光源在齐次对象坐标中的方向。整型数值和浮点数值都直接映射。二者都不被截断。当函数glLight()被调用的时候，光源的聚光方向通过模式取景矩阵的逆阵转换得到（就好象它是一个法向量）。它被储存为一个眼坐标。仅当**GL_SPOT_CUTOFF**不是180时它才有意义。（**GL_SPOT_CUTOFF**的默认值即180）。默认的聚光方向是(0, 0, -1)。

GL_SPOT_EXPONENT

参数*params*是一个整数值或浮点值，它用来指定光源的强度分布。整型数值和浮点数值都直接映射。其取值范围是[0, 128]。

有效的光强度是按光照方向与它到所照射的顶点的方向之间的夹角的余弦规律衰减的，一直衰减到聚光指数的幂。因此，较高的聚光指数就会导致一个较高的汇聚光源，而不管其点截止角究竟如何（请参阅下面的**GL_SPOT_CUTOFF**）。缺省的聚光指数是0，它产生了均匀的光强衰减方式。

GL_SPOT_CUTOFF

参数*params*是一个整数值或浮点值，它用来指定一个光源的最大发散角。整型数值和浮点数值都直接映射。其取值范围是[0, 90]和特殊值180。如果光照方向与它到所照射的顶点的方向之间的夹角大于聚光灯的截止角，光线将被完全屏蔽掉。否则，光强将受聚光指数和发散因子所控制。缺省的聚光灯截止角是180，它产生了均匀的发散光。

GL_CONSTANT_ATTENUATION

GL_LINEAR_ATTENUATION

GL_QUADRATIC_ATTENUATION

参数*params*是一个整数值或浮点值，它用来指定光源的三个衰减因子。整型数值和浮点数值都直接映射。它只能取非负值。如果一个光源是位置光源而非方向光源，它将按某一个代数和的倒数衰减，这一代数和等于常数衰减因子加上线性衰减因子乘以光源与被照亮的顶点之间的距离再加上二次衰减因子乘以这一距离的平方所得的和。衰减因子的缺省值是(1, 0, 0)，这时光强不衰减。

- 注意：

下式总是成立的：

GL_LIGHT*i* = GL_LIGHT0 + *i*

- 出错提示：

当参数*light*或*pname*不是一个可接受的值时产生**GL_INVALID_ENUM**提示。

当聚光指数值超出范围[0, 128], 或聚光灯的截止角度超出范围 [0, 90] (除特殊值180之外), 或指定的衰减因子是一个负数时产生**GL_INVALID_VALUE**提示。

当函数**glDrawPixels()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGetLight()

glIsEnabled(GL_LIGHTING)

- 请参阅：

glColorMaterial(), **glLightModel()**, **glMaterial()**

glLightModel

- 名称：

glLightModelf(), **glLightModeli()**, **glLightModelfv()**, **glLightModeliv()**

- 功能：

设置光照模式参数。

- C描述：

```
void glLightModelf( GLenum pname,
                    GLfloat param )
```

```
void glLightModeli( GLenum pname,
                    GLint param )
```

- 参数说明：

pname 指定一个单值的光照模式参数。它可以是**GL_LIGHT_MODEL_LOCAL_VIEWER**、**GL_LIGHT_MODEL_COLOR_CONTROL**和**GL_LIGHT_MODEL_TWO_SIDE**。

param 指定参数*pname*的具体设定值。

- C描述：

```
void glLightModelfv( GLenum pname,
                     const GLfloat *params )
```

```
void glLightModeliv( GLenum pname,
                     const GLint *params )
```

- 参数说明：

pname 指定一个单值的光照模式参数。它可以是**GL_LIGHT_MODEL_AMBIENT**、**GL_LIGHT_MODEL_COLOR_CONTROL**、**GL_LIGHT_MODEL_LOCAL_VIEWER**和**GL_LIGHT_MODEL_TWO_SIDE**。

params 指定一个指针，指向参数*pname*的具体设定值。

• 说明：

函数glLightModel()用来设定光照模式参数。参数*pname*指定一种光照模式，参数*params*给出光照模式的新值。下面是四种光照模式参数：

GL_LIGHT_MODEL_AMBIENT

参数*params*包含四个整数值或四个浮点值，它们用来指定全局环境光的RGBA强度。整型数值被线性地映射为浮点格式：最大的正数映射为1.0，最小的负数映射为-1.0。浮点型数值则直接映射。二者都不被截断。初始的环境光强度为(0.2, 0.2, 0.2, 1.0)。

GL_LIGHT_MODEL_COLOR_CONTROL

参数*params*必须是**GL_SEPARATE_SPECULAR_COLOR**和**GL_SINGLE_COLOR**二者之一。**GL_SINGLE_COLOR**指定通过对每一顶点进行光照运算而产生的光是单色光。

GL_SEPARATE_SPECULAR_COLOR

指定光照计算所得的镜面光将同光照计算式的余项分别存储。在纹理映射操作应用之后（如果这一操作开启），这些镜面光将相加而生成片断的颜色。其初始值是**GL_SINGLE_COLOR**。

GL_LIGHT_MODEL_LOCAL_VIEWER

参数*params*是一个整数值或一个浮点值，用来指定镜面反射角如何计算。当参数*pname*是0（或0.0）时，镜面反射光将平行于照射光并沿-z轴方向，这时顶点在眼坐标中的位置将不予考虑。否则，镜面反射将从眼坐标体系的原点计算。其初始值是0。

GL_LIGHT_MODEL_TWO_SIDE

参数*params*是一个整数值或一个浮点值，用来指定对多边形进行的光照计算是按单面光照还是按双面光照。点、线和位图的光照计算将不受这一参数的影响。当参数*pname*是0（或0.0）时，采用单面光照模式，这时只有**front**材质参数被带入光照方程。否则，采用双面光照模式。这时多边形的反面顶点被**back**材质照亮，且用光照方程求值时，这些法向量将反向。正面多边形的顶点被**front**材质照亮，其法向量不变。参数*pname*的初始值是0。

在RGBA模式下，顶点的光照颜色是材质的辐射光强度、材质的环境反射光和光照模式的全局环境光强度的合成光同每一个启用的光源产生的光强度的总和。每一个启用的光源产生的光强有三部分组成：环境光、散射光和反射光。其中环境光源是由材质的环境反射光和光源的环境光强所形成的。散射光源是由材质的散射反射率、光源的散射强度以及从顶点到光源的法线（含有从顶点到光源的归一化法向量）所生成的点所决定的。镜面反射光源是由材质的镜面反射率、光源的反射强度以及归一化的顶点到观察点与顶点到光源的向量所生成的点所决定。它可以一直达到材质的发光强度。光源的光照强度是随它与顶点之间的距离、光照方向、发散指数和发散截止角而衰减的。如果所求值为负数，则该点的光照为0。

将最后所得的光照颜色的alpha组分值设为等于材质的散射光的alpha组分值。

在颜色索引模式下，一个顶点的从环境光到反射光的光照索引值将用**GL_COLOR_INDEXES**传给函数**glMaterial()**。使用加权值(0.30, 0.59, 0.11)由光源的颜色计算而来散射和镜面反射系数、材质发出的光以及与RGBA模式相同的反射光和衰减情况最终决定了所得的索引含有多少上面提到的环境光。

- 注意：

GL_LIGHT_MODEL_COLOR_CONTROL只有在GL 1.2以上的版本中才可以使用。

- 出错提示：

当参数*pname*不是一个可接受的值时产生**GL_INVALID_ENUM**提示。

当参数*pname*为**GL_LIGHT_MODEL_COLOR_CONTROL**，但参数*params*不是**GL_SINGLE_COLOR**或**GL_SEPARATE_SPECULAR_COLOR**时产生**GL_INVALID_ENUM**提示。

当函数**glLightMode()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

```
glGet( GL_LIGHT_MODEL_AMBIENT )
glGet( GL_LIGHT_MODEL_COLOR_CONTROL )
glGet( GL_LIGHT_MODEL_LOCAL_VIEWER )
glGet( GL_LIGHT_MODEL_TWO_SIDE )
glIsEnabled( GL_LIGHTING )
```

- 请参阅：

glLight(), **glMaterial()**

• **glLineStipple**

- 名称：

glLineStipple()

- 功能：

指定线的点画绘制模板。

- C描述：

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

- 参数说明：

factor 指定点画绘制模板中每个二进制位的重复次数。例如，如果参数*factor*是3，则模板中相应的二进制位重复使用三次后才使用下一位。参数*factor*的取值范围是[1, 256]，其默认值是1。

pattern 指定一个16位整数。它的二进制位模板决定在线段光栅化时线段中哪个片断将被绘出。首先使用零位。模板的默认值全是1。

- 说明：

线的点画画法指出了光栅化时哪些片断将被屏蔽掉，这些片断将不被绘出。你可以通过三个参数来获得这个屏蔽值：16位的线点画模板`pattern`，重复计数器`factor`和一个整型点画计数器`s`。

无论何时，只要函数`glBegin()`被调用或由函数`glBegin(GL_LINES)/glEnd()`绘制线段之前，计数器`s`均被清零。当走样线段的每一单元宽度的片断被绘出或宽度为`i`的线段的`i`个片断被绘出时它都自动加1。如果式

$$pattern \text{ bit } (s / factor) \bmod 16$$

的值是0时，与计数器`s`相对应的片断`i`将被屏蔽掉。否则，这些片断将被送入帧缓冲区。`pattern`中的零位是最小的有意义位。

反走样的线被看作是一系列的 $1 \times width$ 的用来点画绘制的长方形。长方形`s`是否被光栅化是由走样线的描述规则所决定的。只是这时是用长方形来计数而不是片断组。

用函数`glEnable(GL_LINE_STIPPLE)`和`glDisable(GL_LINE_STIPPLE)`可以启用和关闭线的点画绘制模式。当启用点画绘制模式时，线的点画绘制模板的用法如上所述。当关闭点画绘制模式时，就好象模板全是1。初始情况下，点画绘制模式关闭。

- 出错提示：

当函数`glLineStipple()`在函数对`glBegin()/glEnd()`之间执行时产生`GL_INVALID_OPERATION`提示。

- 有关数据的获取：

`glGet(GL_LINE_STIPPLE_PATTERN)`
`glGet(GL_LINE_STIPPLE_REPEAT)`
`glIsEnabled(GL_LINE_STIPPLE)`

- 请参阅：

`glLineWidth()`, `glPolygonStipple()`

; glLineWidth

- 名称：

`glLineWidth()`

- 功能：

指定光栅化线的宽度。

- C描述：

`void glLineWidth(GLfloat width)`

- 参数说明：

`width` 指定光栅化线的宽度。其初始值是1。

- 说明：

函数`glLineWidth()`用来指定光栅化的走样线和反走样线的宽度。采用一个不为1的线宽将会产生不同的效果，它与是否启动线段的反走样模式有关。函数`glEnable(GL_LINE_SMOOTH)`和`glDisable(GL_LINE_SMOOTH)`用来启动和关闭反走样模式。线的反走样模式被默认为关闭状态。

当关闭线的反走样模式时，线的实际宽度被圆整为系统所提供的最接近的整数值（如果圆整值为0，则相当于这时的线宽为1）。如果 $|\Delta x| \geq |\Delta y|$ ，则光栅化时用*i*个像素填充每一列。这里*i*为参数*width*的圆整值。否则，用*i*个像素填充每一行。

当启用线的反走样模式时，线的光栅化将为每一像素块产生一个片断。反走样线的绘制过程就是用精确度等同于以实际直线为中心线、长度和宽度都等于实际线的矩形块填充每个片断的过程。每个片断的覆盖值就是长方形区域与相应的像素块相交的窗口坐标区域。这一值被存储起来并将在最后的光栅化过程中用到。

反走样操作并不支持所有的线宽。当需要一种不被支持的线宽时，线的宽度将由最接近的支持线宽代替。线宽为1时，它一定支持反走样操作，其他值的情况由具体的系统实现决定。同样，线的走样宽度也有一个范围。函数glGet(GL_ALIASED_LINE_WIDTH_RANGE)、glGet(GL_SMOOTH_LINE_WIDTH_RANGE)、glGet(GL_SMOOTH_LINE_WIDTH_GRANULARITY)分别用于查询走样线所支持的线宽范围、反走样线所支持的线宽范围以及反走样所支持的线宽范围内具体尺寸的不同。

- 注意：

GL_LINE_WIDTH用来查询函数glLineWidth()指定的线宽。对于走样线和反走样线而言，对指定的线宽进行截断和近似操作是无效的。

非反走样线的宽度可以截断为一个系统所允许的最大值。调用函数glGet(GL_ALIASED_LINE_WIDTH_RANGE)可以获取这一最大值。

- 出错提示：

当参数*width*小于或等于0时产生**GL_INVALID_VALUE**提示。

当函数glLineWidth()在函数对glBegin()/glEnd()之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

```
glGet( GL_LINE_WIDTH )
glGet( GL_ALIASED_LINE_WIDTH_RANGE )
glGet( GL_SMOOTH_LINE_WIDTH_RANGE )
glGet( GL_SMOOTH_LINE_WIDTH_GRANULARITY )
glIsEnabled( GL_LINE_SMOOTH )
```

- 请参阅：

glEnable()

- **glListBase**

- 名称：

glListBase()

- 功能：

为函数glCallLists()设置显示列表的基值。

- C描述：

void glListBase(GLuint base)

- 参数说明：

base 指定一个整型偏移基值，这个偏移基值将被加到函数glCallLists()中的偏移量上，从而生成最终的显示列表名称。其初始值为0。

- 说明：

函数glCallLists()指定一个偏移量数组。显示列表的名称由每一个偏移量加上偏移基值而得到。它所提供的有效显示列表将被执行，其他的显示列表被忽略。

- 出错提示：

当函数glListBase()在函数对glBegin()/glEnd()之间执行时产生GL_INVALID_OPERATION提示。

- 有关数据的获取：

glGet(GL_LIST_BASE)

- 请参阅：

glCallLists()

• glLoadIdentity

- 名称：

glLoadIdentity()

- 功能：

用单位矩阵替换当前的矩阵。

- C描述：

void glLoadIdentity(void)

- 说明：

函数glLoadIdentity()用单位矩阵替换当前的矩阵。它在语法上等同于调用函数glLoadMatrix()时使用单位矩阵

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

但有时调用函数glLoadIdentity()更有效。

- 出错提示：

当函数glLoadIdentity()在函数对glBegin()/glEnd()之间执行时产生GL_INVALID_OPERATION提示。

- 有关数据的获取：

glGet(GL_MATRIX_MODE)

glGet(GL_COLOR_MATRIX)

glGet(GL_MODELVIEW_MATRIX)

glGet(GL_PROJECTION_MATRIX)

glGet(GL_TEXTURE_MATRIX)

- 请参阅：

glLoadMatrix(), glMatrixMode(), glMultMatrix(), glPushMatrix()

• **glLoadMatrix**

- 名称：

glLoadMatrixd(), glLoadMatrixf()

- 功能：

用指定的矩阵替换当前矩阵。

- C 描述：

`void glLoadMatrixd(const GLdouble *m)`

`void glLoadMatrixf(const GLfloat *m)`

- 参数说明：

m 指定一个指针，指向16个连续的值。这16个值按列组成一个 4×4 的矩阵。

- 说明：

函数**glLoadMatrix** 用*m*指定的矩阵替换当前矩阵。当前矩阵可以是投影矩阵、模式取景矩阵或纹理矩阵，具体由当前矩阵模式决定（请参阅**glMatrixMode()**）。

当前矩阵M定义了一种坐标转换方式。比如，假设M是一个模式取景矩阵。如果*v*=(*v*[0], *v*[1], *v*[2], *v*[3])是顶点的对象坐标集，并且*m*指向一个包含16个单精度或双精度浮点值*m*[0], *m*[1], ..., *m*[15]的数组，那么模式取景转换*M(v)*按下式计算：

$$M(v) = \begin{matrix} m[0] & m[4] & m[8] & m[12] & v[0] \\ m[1] & m[5] & m[9] & m[13] & v[1] \\ m[2] & m[6] & m[10] & m[14] & \times \\ m[3] & m[7] & m[11] & m[15] & v[2] \end{matrix}$$

这里符号“×”表示矩阵乘积。

投影变换和纹理变换可以类似地定义。

- 注意：

当矩阵中的元素可以被指定为单精度或双精度值时，GL系统至少可以把这些值当作单精度值进行存储或操作。

- 出错提示：

当函数**glLoadMatrix()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGet(GL_MATRIX_MODE)

glGet(GL_COLOR_MATRIX)

glGet(GL_MODELVIEW_MATRIX)

glGet(GL_PROJECTION_MATRIX)**glGet(GL_TEXTURE_MATRIX)**

- 请参阅：

glLoadIdentity(), glMatrixMode(), glMultMatrix(), glPushMatrix()

glLoadName

- 名称：

glLoadName()

- 功能：

载入一个名称到名称堆栈中。

- C描述：

void glLoadName(GLuint name)

- 参数说明：

name 指定一个将用来替换名称堆栈的栈顶值的新名称。

- 说明：

名称堆栈是选择模式下用来唯一地确定绘图命令的集合。它由一组有序的无符号整数组成。

函数**glLoadName()**用*name*替换名称堆栈的栈顶值。初始情况下，它是空的。

当绘图模式不是**GL_SELECT**时，名称堆栈总是空的。这时调用函数**glLoadName()**的操作将被忽略。

- 出错提示：

当名称堆栈是空堆栈时调用函数**glLoadName()**将产生**GL_INVALID_OPERATION**提示。

当函数**glLoadName()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGet(GL_NAME_STACK_DEPTH)**glGet(GL_MAX_NAME_STACK_DEPTH)**

- 请参阅：

glInitNames(), glPushName(), glRenderMode(), glSelectBuffer()

glLogicOp

- 名称：

glLogicOp()

- 功能：

在颜色索引模式下绘图时，指定一种逻辑像素操作。

- C描述：

void glLogicOp(GLenum opcode)

- 参数说明：

opcode 指定一个符号常量，从而选择一种逻辑操作。它可以是**GL_CLEAR**、**GL_SET**、

GL_COPY、**GL_COPY_INVERTED**、**GL_NOOP**、**GL_INVERT**、**GL_AND**、**GL_NAND**、**GL_OR**、**GL_NOR**、**GL_XOR**、**GL_EQUIV**、**GL_AND_REVERSE**、**GL_AND_INVERTED**、**GL_OR_REVERSE**和**GL_OR_INVERTED**。初始值是**GL_COPY**。

- 说明：

函数**glLogicOp()**用来指定一种逻辑操作。当启用这一操作后，它将在输入的颜色索引值或RGBA值与帧缓冲区中相应的颜色索引值或RGBA值之间进行逻辑操作。在RGBA模式下，启用和关闭逻辑操作可以调用函数**glEnable(GL_COLOR_LOGIC_OP)**和**glDisable(GL_COLOR_LOGIC_OP)**；在颜色索引模式下，启用和关闭逻辑操作可以调用函数**glEnable(GL_INDEX_LOGIC_OP)**和**glDisable(GL_INDEX_LOGIC_OP)**。初始情况下，逻辑操作在这两种模式下都关闭。

参数*opcode*是一个符号常量，它可取的值如表5-22所示。表5-22中，*s*代表输入的颜色索引值，*d*代表帧缓冲区中的颜色索引值。它使用标准的C语言。表5-22所示的操作均为二进制运算，这些逻辑操作在源索引（或颜色）值与目标索引（或颜色）值的相应位之间进行。

- 注意：

颜色索引的逻辑操作被任何版本的GL支持，但RGBA逻辑操作仅在GL1.1以上的版本中才被支持。

表 5-22

Opcode	操作结果
GL_CLEAR	0
GL_SET	1
GL_COPY	<i>s</i>
GL_COPY_INVERTED	$\neg s$
GL_NOOP	<i>d</i>
GL_INVERT	$\neg d$
GL_AND	<i>s&d</i>
GL_NAND	$\neg(s \& d)$
GL_OR	<i>s d</i>
GL_NOR	$\neg(s d)$
GL_XOR	$s \oplus d$
GL_EQUIV	$\neg(s \oplus d)$
GL_AND_REVERSE	<i>s&$\neg d$</i>
GL_AND_INVERTED	$\neg s \& d$
GL_OR_REVERSE	<i>s $\neg d$</i>
GL_OR_INVERTED	$\neg s d$

如果可以向不止一个的RGBA颜色或索引缓冲区中写入，则逻辑操作将用每个缓冲区中的目标值对各个启动的缓冲区分别操作（请参阅**glDrawBuffer()**）。

- 出错提示：

当参数*opcode*不是一个可接受的值时产生**GL_INVALID_ENUM**提示。

当函数glLogicOp()在函数对glBegin()/glEnd()之间执行时产生**GL_INVALID_OPERATION** 提示。

- 有关数据的获取：

```
glGet( GL_LOGIC_OP_MODE )
glIsEnabled( GL_COLOR_LOGIC_OP )
glIsEnabled( GL_INDEX_LOGIC_OP )
```

- 请参阅：

glAlphaFunc(), **glBlendFunc()**, **glDrawBuffer()**, **glEnable()**, **glStencilOp()**

• **glMap1**

- 名称：

glMap1d(), **glMap1f()**

- 功能：

定义一个一维求值器。

- C 描述：

```
void glMap1d( GLenum target,
              GLdouble u1,
              GLdouble u2,
              GLint stride,
              GLint order,
              const GLdouble *points )
```

```
void glMap1f( GLenum target,
              GLfloat u1,
              GLfloat u2,
              GLint stride,
              GLint order,
              const GLdouble *points )
```

- 参数说明：

target 指定由求值器所生成的值的种类。它可以是符号常量**GL_MAP1_VERTEX_3**、**GL_MAP1_VERTEX_4**、**GL_MAP1_INDEX**、**GL_MAP1_COLOR_4**、**GL_MAP1_NORMAL**、**GL_MAP1_TEXTURE_COORD_1**、**GL_MAP1_TEXTURE_COORD_2**、**GL_MAP1_TEXTURE_COORD_3**和**GL_MAP1_TEXTURE_COORD_4**。

u1, *u2* 指定*u*的线性映射方式，并把它提供给函数glEvalCoord1()。这个变量将由此命令指定的方程求值。

stride 指定在由*points*提供的数据结构中，从一个控制点的开始位置到下一个控制点的开始位置之间跨越的浮点或双精度值的存储单元数目。这里允许将控制点插入

任意的数据结构中。它唯一的约束条件是一个特定的控制点必须占据连续的内存存储单元。

order 指定控制点的数目。它必须是正数。

points 指定一个指向控制点数组的指针。

• 说明：

求值器提供了一种用多项式或有理多项式映射来生成顶点、法线、纹理坐标和颜色的方法。由求值器所生成的值将被送到后面的GL处理过程中，这个值与由函数glVertex()、glNormal()、glTexCoord()和glColor()提供的值基本相同，只是求值器产生的值不能更新当前的法线、纹理坐标和颜色。

你可以使用求值器对任何次数（直到GL系统所支持的最大值）的多项式或有理多项式形式的样条曲线进行描述。这几乎包括了所有计算机图形所使用的曲面：B样条曲线、Bezier曲线和Hermite样条曲线等等。

求值器以Bernstein多项式为基础定义曲线。定义为

$$p(\hat{u}) = \sum_{i=0}^n B_i^n(\hat{u}) R_i$$

这里 R_i 是一个控制点， $B_i^n(\hat{u})$ 是次数为 n ($order=n+1$)的第*i*个Bernstein多项式：

$$B_i^n(\hat{u}) = \frac{n}{i} \hat{u}^i (1-\hat{u})^{n-i}$$

并且：

$$0^0 \equiv 1 \text{ and } \frac{n}{0} \equiv 1$$

函数glMap1()定义了一个基值并指定了所生成的值的种类。此函数执行后可以用函数glEnable(映射名)和glDisable(映射名)来启动和关闭一个映射。这里的映射名是由参数target预先定义的九个值之一，详述如下。这时可用函数glEvalCoord1()来求取一维映射值。当函数glEvalCoord1()提供了一个u值后，Bernstein方程用来求取这个变量u的值，这里

$$\hat{u} = \frac{u - u_1}{u_2 - u_1}$$

参数target是一个符号常量，用来表示在*points*中提供的控制点的种类和映射求值后的输出。参数target可以被预先定义为下面九个值之一：

GL_MAP1_VERTEX_3 每一个控制点是三个浮点值x, y和z。当映射求值时，产生内部命令glVertex3()。

GL_MAP1_VERTEX_4 每一个控制点是四个浮点值x, y, z和w。当映射求值时，产生内部命令glVertex4()。

GL_MAP1_INDEX 每一个控制点是一个浮点值，它代表一个颜色索引。当映射求值时，产生内部命令glIndex()。但当前的索引值不能由这些glIndex()命令所生成的值更新。

GL_MAP1_COLOR_4 每一个控制点是四个浮点值，它们分别代表红、绿、蓝和alpha值。当映射求值时，产生内部命令glColor4()。但当前的颜色值不能由这些glColor4()命令更新。

GL_MAP1_NORMAL 每一个控制点是三个浮点值，它们分别代表一个法向量的x、y和z组分值。当映射求值时，产生内部命令glNormal()。但当前的法线值不能由这些glNormal()命令所生成的值更新。

GL_MAP1_TEXTURE_COORD_1

每一个控制点是一个浮点值，它代表一个s纹理坐标。当映射求值时，产生内部命令glTexCoord1()。但当前的纹理坐标值不能由这些glTexCoord()命令所生成的值更新。

GL_MAP1_TEXTURE_COORD_2

每一个控制点是两个浮点值，它们分别代表s和t纹理坐标。当映射求值时，产生内部命令glTexCoord2()。但当前的纹理坐标值不能由这些glTexCoord()命令所生成的值更新。

GL_MAP1_TEXTURE_COORD_3

每一个控制点是三个浮点值，它们分别代表s、t和r纹理坐标。当映射求值时，产生内部命令glTexCoord3()。但当前的纹理坐标值不能由这些glTexCoord()命令所生成的值更新。

GL_MAP1_TEXTURE_COORD_4

每一个控制点是四个浮点值，它们分别代表s、t、r和q纹理坐标。当映射求值时，产生内部命令glTexCoord4()。但当前的纹理坐标值不能由这些glTexCoord()命令所生成的值更新。

参数*stride*, *order*和*points*定义了访问控制点时的数组寻址。参数*points*是第一个控制点的存储单元地址，它占据了一个、两个、三个或四个连续的内存单元，具体情况由定义的映射决定。在数组中有*order*个控制点。参数*stride*指定内存指针从一个控制点到下一个控制点需要跨过的的浮点或双精度的存储单元数目。

- 注意：

所有GL命令都可以采用指针指向数据，就好象参数*points*中的内容是由函数glMap1()在返回前拷贝而来。当函数glMap1()被调用后，改变参数*points*中内容的操作将无效。

- 出错提示：

当参数*target*不是一个可接受的值时产生**GL_INVALID_ENUM**提示。

当*u1=u2*时产生**GL_INVALID_VALUE**提示。

当*stride*小于一个控制点中数值的数目时产生**GL_INVALID_VALUE**提示。

当*order*小于1或大于**GL_MAX_EVAL_ORDER**的返回值时产生**GL_INVALID_VALUE**提示。

当函数glMap1()在函数对glBegin()/glEnd()之间运行时产生**GL_INVALID_OPERATION**提示。

当系统支持**GL_ARB_multitexture**扩展时，如果**GL_ACTIVE_TEXTURE_ARB**的值不是**GL_TEXTURE0_ARB**，这时调用函数glMap1()将产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

```
glGetMap()
glGet( GL_MAX_EVAL_ORDER )
glIsEnabled( GL_MAP1_VERTEX_3 )
glIsEnabled( GL_MAP1_VERTEX_4 )
glIsEnabled( GL_MAP1_INDEX )
glIsEnabled( GL_MAP1_COLOR_4 )
glIsEnabled( GL_MAP1_NORMAL )
glIsEnabled( GL_MAP1_TEXTURE_COORD_1 )
glIsEnabled( GL_MAP1_TEXTURE_COORD_2 )
glIsEnabled( GL_MAP1_TEXTURE_COORD_3 )
glIsEnabled( GL_MAP1_TEXTURE_COORD_4 )
```

- 请参阅：

glBegin(), **glColor()**, **glEnable()**, **glEvalCoord()**, **glEvalMesh()**, **glEvalPoint()**, **glMap2()**,
glMapGrid(), **glNormal()**, **glTexCoord()**, **glVertex()**

***glMap2**

- 名称：

glMap2d(), **glMap2f()**

- 功能：

定义一个二维求值器。

- C 描述：

```
void glMap2d( GLenum target,
            GLdouble u1,
            GLdouble u2,
            GLint ustride,
            GLint uorder,
            GLdouble v1,
            GLdouble v2,
            GLint vstride,
            GLint vorder,
            const GLdouble *points )
void glMap2f( GLenum target,
            GLfloat u1,
            GLfloat u2,
            GLint ustride,
            GLint uorder,
            GLfloat v1,
```

```

    GLfloat v2,
    GLint vstride,
    GLint vorder,
    const GLfloat *points )

```

• 参数说明：

- target* 指定求值器所生成的值的种类。它可以是符号常量**GL_MAP2_VERTEX_3**、**GL_MAP2_VERTEX_4**、**GL_MAP2_INDEX**、**GL_MAP2_COLOR_4**、**GL_MAP2_NORMAL**、**GL_MAP2_TEXTURE_COORD_1**、**GL_MAP2_TEXTURE_COORD_2**、**GL_MAP2_TEXTURE_COORD_3**和**GL_MAP2_TEXTURE_COORD_4**。
- u1*, *u2* 指定*u*的线性映射方式，并把它提供给函数**glEvalCoord2()**。这两个变量中的一个变量将由此命令所指定的方程求值。初始情况下，*u1*为0，*u2*为1。
- ustride* 指定控制点*R_{i,j}*开始位置和控制点*R_{i+1,j+1}*开始位置之间跨越的浮点或双精度值的存储单元数目。这里*i*和*j*分别是*u*和*v*的控制点索引。这里允许将控制点插入任意的数据结构中。它唯一的约束条件是一个特定的控制点必须占据连续的内存存储单元。参数*ustride*的初始值是0。
- uorder* 指定*u*轴方向的控制点数组的维数。它必须是正数。其初始值是1。
- v1*, *v2* 指定*v*的线性映射方式，并把它提供给函数**glEvalCoord2()**。这两个变量中的一个变量将由此命令所指定的方程求值。初始情况下，*v1*为0，*v2*为1。
- vstride* 指定控制点*R_{i,j}*开始位置和控制点*R_{i+1,j+1}*开始位置之间跨越的浮点或双精度值的存储单元数目。这里*i*和*j*分别是*u*和*v*的控制点索引。这里允许将控制点插入任意的数据结构中。它唯一的约束条件是一个特定的控制点必须占据连续的内存存储单元。参数*vstride*的初始值是0。
- vorder* 指定*v*轴方向的控制点数组的维数。它必须是正数。其初始值是1。
- points* 指定一个指向控制点数组的指针。

• 说明：

求值器提供了一种用多项式或有理多项式映射来生成顶点、法线、纹理坐标和颜色的方法。由求值器所生成的值将被送到后面的GL处理过程中，这个值与由函数**glVertex()**, **glNormal()**, **glTexCoord()**和**glColor()**提供的值基本相同，只是求值器产生的值不能更新当前的法线、纹理坐标和颜色。

你可以使用求值器对任何次数（直到GL系统所支持的最大值）的多项式或有理多项式样条曲线进行描述。这几乎包括了所有计算机图形所使用的曲面：B样条曲面、NURBS(非规一化的有理B样条)曲面、Bezier曲面等等。

求值器以双变量的Bernstein多项式为基础定义曲面。定义为

$$p(\hat{u}, \hat{v}) = \sum_{i=0}^n \sum_{j=0}^m B_i^n(\hat{u}) B_j^m(\hat{v}) R_{ij}$$

这里 R_i 是一个控制点, $B_i^n(u)$ 是次数为 n ($uorder=n+1$)的第*i*个Bernstein多项式。

这里:

$$B_i^n(\hat{u}) = \frac{n}{i} \hat{u}^i (1-\hat{u})^{n-i}$$

并且, $B_j^m(\hat{v})$ 是次数为 m ($vorder = m + 1$)的第*j*个Bernstein多项式。

$$B_j^m(\hat{v}) = \frac{m}{j} \hat{v}^j (1-\hat{v})^{m-j}$$

并且:

$$0^0 = 1 \text{ and } \frac{n}{0} = 1$$

函数glMap2()定义了一个基值并指定了所产生的值的种类。此函数执行后可以用函数glEnable(映射名)和glDisable(映射名)来启动和关闭一个映射。这里的映射名是由参数target预先定义的九个值之一, 详述如下。当函数glEvalCoord2()提供了*u*和*v*的值后, 双变量的Bernstein多项式用来求取变量*u*和*v*的值, 这里

$$\hat{u} = \frac{u - u1}{u2 - u1}$$

$$\hat{v} = \frac{v - v1}{v2 - v1}$$

参数target是一个符号常量, 用来表示在points中提供的控制点的种类和映射求值后的输出。参数target可以被预先定义为下面的九个值之一:

GL_MAP2_VERTEX_3 每一个控制点是三个浮点值x, y和z。当映射求值时, 产生内部命令glVertex3()。

GL_MAP2_VERTEX_4 每一个控制点是四个浮点值x, y, z和w。当映射求值时, 产生内部命令glVertex4()。

GL_MAP2_INDEX 每一个控制点是一个浮点值, 它代表一个颜色索引。当映射求值时, 产生内部命令glIndex()。但当前的索引值不能由这些glIndex()命令所生成的值更新。

GL_MAP2_COLOR_4 每一个控制点是四个浮点值, 它们分别代表红、绿、蓝和alpha值。当映射求值时, 产生内部命令glColor4()。但当前的颜色值不能由这些glColor4()命令所生成的值更新。

GL_MAP2_NORMAL 每一个控制点是三个浮点值, 它们分别代表一个法向量的x、y和z组分值。当映射求值时, 产生内部命令glNormal()。但当前的法线值不能由这些glNormal()命令所生成的值更新。

GL_MAP2_TEXTURE_COORD_1

每一个控制点是一个浮点值, 它代表一个s纹理坐标。当映射求值时, 产生内部命令glTexCoord1()。但当前的纹理坐标值不能由这些glTexCoord()命令所生成的值更新。

GL_MAP2_TEXTURE_COORD_2

每一个控制点是两个浮点值，它们分别代表s和t纹理坐标。当映射求值时，产生内部命令glTexCoord2()。但当前的纹理坐标值不能由这些glTexCoord()命令所生成的值更新。

GL_MAP2_TEXTURE_COORD_3

每一个控制点是三个浮点值，它们分别代表s、t和r纹理坐标。当映射求值时，产生内部命令glTexCoord3()。但当前的纹理坐标值不能由这些glTexCoord()命令所生成的值更新。

GL_MAP2_TEXTURE_COORD_4

每一个控制点是四个浮点值，它们代表s、t、r和q纹理坐标。当映射求值时，产生内部命令glTexCoord4()。但当前的纹理坐标值不能由这些glTexCoord()命令所生成的值更新。

参数 $ustride$ 、 $uorder$ 、 $vstride$ 、 $vorder$ 和 $points$ 定义了访问控制点时的数组寻址。参数 $points$ 是第一个控制点的存储单元地址，它占据了一个、两个、三个或四个连续的内存单元，具体情况由定义的映射决定。在数组中有 $vorder \times uorder$ 个控制点。参数 $ustride$ 指定内存指针从控制点 R_{ij} 到控制点 $R_{i+1,j}$ 需要跨过的浮点或双精度的存储单元数目。参数 $vstride$ 指定内存指针从控制点 R_{ij} 到控制点 $R_{i,j+1}$ 需要跨过的浮点或双精度的存储单元数目。

- 注意：

所有GL命令都可用指针指向数据，就好象参数 $points$ 中的内容是由函数glMap2()在返回前拷贝而来。当函数glMap2()被调用后，改变参数 $points$ 中内容的操作将无效。

初始情况下，**GL_AUTO_NORMAL**有效。这时**GL_MAP2_VERTEX_3**或**GL_MAP2_VERTEX_4**生成顶点时将同时生成法向量。

- 出错提示：

当参数 $target$ 不是一个可接受的值时产生**GL_INVALID_ENUM**提示。

当 $u1=u2$ 或 $v1=v2$ 时产生**GL_INVALID_VALUE**提示。

当 $ustride$ 或 $vstride$ 小于一个控制点中数值的数目时产生**GL_INVALID_VALUE**提示。

当 $uorder$ 或 $vorder$ 小于1或大于**GL_MAX_EVAL_ORDER**的返回值时产生**GL_INVALID_VALUE**提示。

当函数glMap2()在函数对glBegin()/glEnd()之间运行时产生**GL_INVALID_OPERATION**提示。

当系统支持**GL_ARB_multitexture**扩展时，如果**GL_ACTIVE_TEXTURE_ARB**的值不是**GL_TEXTURE0_ARB**，这时调用函数glMap2()将产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGetMap()

glGet(GL_MAX_EVAL_ORDER)

glIsEnabled(GL_MAP2_VERTEX_3)

glIsEnabled(GL_MAP2_VERTEX_4)

glIsEnabled(GL_MAP2_INDEX)
glIsEnabled(GL_MAP2_COLOR_4)
glIsEnabled(GL_MAP2_NORMAL)
glIsEnabled(GL_MAP2_TEXTURE_COORD_1)
glIsEnabled(GL_MAP2_TEXTURE_COORD_2)
glIsEnabled(GL_MAP2_TEXTURE_COORD_3)
glIsEnabled(GL_MAP2_TEXTURE_COORD_4)

- 请参阅：

glBegin(), **glColor()**, **glEnable()**, **glEvalCoord()**, **glEvalMesh()**, **glEvalPoint()**, **glMap1()**,
glMapGrid(), **glNormal()**, **glTexCoord()**, **glVertex()**

***glMapGrid**

- 名称：

glMapGrid1d(), **glMapGrid1f()**, **glMapGrid2d()**, **glMapGrid2f()**

- 功能：

定义一个一维或二维的网格。

- C 描述：

```
void glMapGrid1d( GLint un,
                  GLdouble u1,
                  GLdouble u2 )
void glMapGrid1f( GLint un,
                  GLfloat u1,
                  GLfloat u2 )
void glMapGrid2d( GLint un,
                  GLdouble u1,
                  GLdouble u2,
                  GLint vn,
                  GLdouble v1,
                  GLdouble v2 )
void glMapGrid2f( GLint un,
                  GLfloat u1,
                  GLfloat u2,
                  GLint vn,
                  GLfloat v1,
                  GLfloat v2 )
```

- 参数说明：

un 指定网格范围[*u1*, *u2*]之间的等份数。它必须是正数。

- $u1, u2$ 指定整型网格域值 $i=0$ 和 $i=un$ 的映射值。
- vn 指定网格范围 $[v1, v2]$ 之间的等份数（仅用于函数 **glMapGrid2()**）。
- $v1, v2$ 指定整型网格域值 $j=0$ 和 $j=vn$ 的映射值（仅用于函数 **glMapGrid2()**）。
- 说明：

函数 **glMapGrid()** 和函数 **glEvalMesh()** 成对使用，用来产生和求取均匀空间中的一系列映射域值。函数 **glEvalMesh()** 的作用是把某一整型区域分为一个一维或二维的网格，该区域是由函数 **glMap1()** 和 **glMap2()** 指定的求值映射域。

函数 **glMap1()** 和 **glMap2()** 用来指定整型网格坐标 i （或 i, j ）与浮点型求值映射坐标 u （或 u, v ）之间的线性网格映射。具体如何求取 u 和 v 坐标，请参阅函数 **glMap1()** 和 **glMap2()**。

函数 **glMapGrid1()** 指定一个单线性映射，这时整型网格坐标 0 被精确地映射为 $u1$ ，整型网格坐标 un 被精确地映射为 $u2$ 。其他整型网格坐标 i 按下式映射：

$$u = i(u2-u1)/un + u1$$

函数 **glMapGrid2()** 指定两个线性映射。其中的一个映射将整型网格坐标 $i=0$ 精确地映射为 $u1$ ，整型网格坐标 $i=un$ 精确地映射为 $u2$ 。另一个映射将整型网格坐标 $j=0$ 精确地映射为 $v1$ ，整型网格坐标 $j=vn$ 精确地映射为 $v2$ 。其他整型网格坐标 i 和 j 按下式映射：

$$u = i(u2-u1)/un + u1$$

$$v = j(v2-v1)/vn + v1$$

函数 **glMapGrid()** 所指定的映射和函数 **glEvalMesh()** 及 **glEvalPoint()** 指定的映射一样。

- 出错提示：

当参数 un 或 vn 不是正数时产生 **GL_INVALID_VALUE** 提示。

当函数 **glMapGrid()** 在函数对 **glBegin()**/**glEnd()** 之间执行时产生 **GL_INVALID_OPERATION** 提示。

- 有关数据的获取：

```
glGet( GL_MAP1_GRID_DOMAIN )
glGet( GL_MAP2_GRID_DOMAIN )
glGet( GL_MAP1_GRID_SEGMENTS )
glGet( GL_MAP2_GRID_SEGMENTS )
```

- 请参阅：

glEvalCoord(), **glEvalMesh()**, **glEvalPoint()**, **glMap1()**, **glMap2()**

glMaterial

- 名称：
- 功能：
- C 描述：

为光照模型指定材质参数。

```
void glMaterialf( GLenum face,
                  GLenum pname,
                  GLfloat param )
void glMateriali( GLenum face,
                  GLenum pname,
                  GLint param )
```

• 参数说明：

face 指定哪个或哪些面将被更新。它必须是**GL_FRONT**、**GL_BACK**和**GL_FRONT_AND_BACK**之一。

pname 为将被更新的面指定一个单值的材质参数。它必须取**GL_SHININESS**。

param 指定参数**GL_SHININESS**的具体设定值。

• C 描述：

```
void glMaterialfv( GLenum face,
                   GLenum pname,
                   const GLfloat *params )
void glMaterialiv( GLenum face,
                   GLenum pname,
                   const GLint *params )
```

• 参数说明：

face 指定哪个或哪些面将被更新。它必须是**GL_FRONT**、**GL_BACK**和**GL_FRONT_AND_BACK**之一。

pname 为将被更新的面指定一个单值的材质参数。它必须取**GL_AMBIENT**、**GL_DIFFUSE**、**GL_SPECULAR**、**GL_EMISSION**、**GL_SHININESS**、**GL_AMBIENT_AND_DIFFUSE**和**GL_COLOR_INDEXES**之一。

param 指定一个指针，指向参数*pname*的具体设定值。

• 说明：

函数**glMaterial()**用来为材质参数赋值。它有两个适合的材质参数集合：一个是“正面”材质集，用来形成点、线、位图和所有多边形的阴影（当双面光照模式关闭时），或刻画正面多边形（当双面光照模式开启时）；另一个是“反面”材质集，当双面光照模式开启时，它用来形成反面多边形的阴影。请参阅**glLightMode()**获取有关单面光照模式和双面光照模式的更多信息。

函数**glMaterial()**带有三个自变量：自变量*face*用来指定是**GL_FRONT**材质、**GL_BACK**材质还是**GL_FRONT_AND_BACK**材质将被修正。自变量*pname*用来指定一个或两个材质集之中的哪些参数将被修正。自变量*params*用来指定将把怎样的值赋给选定的参数。

材质参数被用于每个顶点的光照方程中。有关光照方程的细节请参阅**glLightModel()**。这些参数可以由函数**glMaterial()**指定。它们在光照方程中的作用如下：

GL_AMBIENT 参数*params*包含四个整数值或浮点值，它们用来指定材质的环境光的

RGBA反射系数。整型数值被线性地映射为浮点格式：最大的正数映射为1.0，最小的负数映射为-1.0。浮点数则直接映射。二者都不被截断。正面和反面材质初始的环境光的反射系数均为(0.2, 0.2, 0.2, 1.0)。

GL_DIFFUSE 参数*params*包含四个整数值或浮点值，它们用来指定材质的散射光的RGBA反射系数。整型数值被线性地映射为浮点格式：最大的正数映射为1.0，最小的负数映射为-1.0。浮点数则直接映射。二者都不被截断。正面和反面材质初始的散射光的反射系数均为(0.8, 0.8, 0.8, 1.0)。

GL_SPECULAR 参数*params*包含四个整数值或浮点值，它们用来指定材质的反射光的RGBA反射系数。整型数值被线性地映射为浮点格式：最大的正数映射为1.0，最小的负数映射为-1.0。浮点数则直接映射。二者都不被截断。正面和反面材质初始的反射光系数均为(0, 0, 0, 1)。

GL_EMISSION 参数*params*包含四个整数值或浮点值，它们用来指定材质的RGBA辐射光的强度。整型数值被线性地映射为浮点格式：最大的正数映射为1.0，最小的负数映射为-1.0。浮点数则直接映射。二者都不被截断。正面和反面材料初始的辐射光强均为(0, 0, 0, 1)。

GL_SHININESS 参数*params*包含一个整数值或浮点值，用来指定材质的RGBA镜面反射指数。整型数值和浮点数都直接映射。其取值范围是[0, 128]。正面和反面材质初始的镜面反射指数均为0。

GL_AMBIENT_AND_DIFFUSE

相当于用相同参数两次调用函数glMaterial()，参数分别为**GL_AMBIENT** 和**GL_DIFFUSE**。

GL_COLOR_INDEXES

参数*params*包含三个整数值或浮点值，它们用来指定材质的环境光、散射光和反射光的颜色索引。在颜色索引模式下的光照方程只能用这三个值以及**GL_SHININESS**。有关颜色索引模式下的光照请参阅函数glLightModel()。

- 注意：

材质参数可以被随时更新。特别是，它能在函数对glBegin()/glEnd()之间被调用。当每一个顶点仅有一个材质参数需要改变时，函数glColorMaterial()比glMaterial()更常用。

虽然环境光、散射光、反射光和辐射光的材质参数中都含有alpha组分，但在光照计算中只采用散射光的alpha组分。

- 出错提示：

当参数*face*或*pname*不是一个可接受的值时产生**GL_INVALID_ENUM**提示。

当镜面反射指数值超出范围[0, 128]时产生**GL_INVALID_VALUE**提示。

- 有关数据的获取：

glGetMaterial()

- 请参阅：

glColorMaterial(), glLight(), glLightModel()**glMatrixMode**

- 名称:

glMatrixMode()

- 功能:

指定当前矩阵。

- C描述:

void glMatrixMode(GLenum mode)

- 参数说明:

mode 指定后续矩阵操作的对象是哪个矩阵堆栈。它可取三个值: **GL_MODELVIEW**、**GL_PROJECTION**和**GL_TEXTURE**。它的初始值是**GL_MODELVIEW**。

此外, 如果系统支持**GL_ARB_imaging**扩展, 也可使用**GL_COLOR**。

- 说明:

函数**glMatrixMode()**用于设置当前的矩阵模式, 参数*mode*可取以下四个值:

GL_MODELVIEW 指定后续矩阵操作的对象是模式取景矩阵堆栈

GL_PROJECTION 指定后续矩阵操作的对象是投影矩阵堆栈

GL_TEXTURE 指定后续矩阵操作的对象是纹理矩阵堆栈

GL_COLOR 指定后续矩阵操作的对象是颜色矩阵堆栈

你可以通过调用函数**glGet(GL_MATRIX_MODE)**来查询当前所有矩阵操作的对象是哪个矩阵堆栈。其初始值是**GL_MODELVIEW**。

- 出错提示:

当参数*mode*不是一个可接受的值时产生**GL_INVALID_ENUM**提示。

当函数**glMatrixMode()**在函数对**glBegin()/glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取:

glGet(GL_MATRIX_MODE)

- 请参阅:

glLoadIdentity(), **glPushMatrix()**

glMinmax

- 名称:

glMinmax()

- 功能:

定义极值表。

- C描述:

**void glMinmax(GLenum target,
 GLenum internalformat,**

`GLboolean sink)`

- 参数说明：

<code>target</code>	指定参数将被设置的极值表。它必须是 GL_MINMAX 。
<code>internalformat</code>	指定极值表中条目的格式。它必须是下面所列的符号常量之一： GL_ALPHA 、 GL_ALPHA4 、 GL_ALPHA8 、 GL_ALPHA12 、 GL_ALPHA16 、 GL_LUMINANCE 、 GL_LUMINANCE4 、 GL_LUMINANCE8 、 GL_LUMINANCE12 、 GL_LUMINANCE16 、 GL_LUMINANCE_ALPHA 、 GL_LUMINANCE4_ALPHA4 、 GL_LUMINANCE6_ALPHA2 、 GL_LUMINANCE8_ALPHA8 、 GL_LUMINANCE12_ALPHA4 、 GL_LUMINANCE12_ALPHA12 、 GL_LUMINANCE16_ALPHA16 、 GL_R3_G3_B2 、 GL_RGB 、 GL_RGB4 、 GL_RGB5 、 GL_RGB8 、 GL_RGB10 、 GL_RGB12 、 GL_RGB16 、 GL_RGBA 、 GL_RGBA2 、 GL_RGBA4 、 GL_RGB5_A1 、 GL_RGBA8 、 GL_RGB10_A2 、 GL_RGBA12 或 GL_RGBA16 。
<code>sink</code>	当它是 GL_TRUE 时，像素在经过极值操作后将被消耗，这时将没有绘制和纹理装载操作发生。如果它是 GL_FALSE ，在极值操作结束后，将对像素进行最后的转换操作。

- 说明：

当启动**GL_MINMAX**后，输入像素的RGBA颜色组分将与存有两个元素的极值表中的各像素的最小值和最大值进行比较。（极值表中的第一个元素是最小值，第二个元素是最大值。）如果一个像素组分大于极值表中的最大值，则该最大值将被这个组分值更新；相反，如果一个像素组分小于极值表中的最小值，则该最小值也同样被更新。（在这两种情况中，如果极值的内部格式中包含亮度值，则用输入像素的R颜色组分来进行比较。）通过随后调用的函数**glGetMinmax()**可以查询极值表的内容。如果要启动与关闭极值操作功能，请使用函数**glEnable(GL_MINMAX)**和**glDisable(GL_MINMAX)**。

函数**glMinmax()**将重新将当前极值表的条目定义成由`internalformat`指定格式。最大值被初始化成可能的最小组分值，而最小值则被初始化成可能的最大组分值。原先极值表中的值将丢失。如果`sink`是**GL_TRUE**，则极值操作完成后，像素将被丢弃，不再对像素进行其他的操作，也不再有绘制、像素载入或像素反馈发生。

- 注意：

当调用函数**glGetString(GL_EXTENSIONS)**的返回值是**GL_ARB_imaging**时，才提供函数**glMinmax()**。

- 出错提示：

当参数`target`不是一个允许值时产生**GL_INVALID_ENUM**提示。

当参数`internalformat`不是一个允许值时产生**GL_INVALID_ENUM**提示。

当函数**glMinmax()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGetMinmaxParameter()

- 请参阅：

glGetMinmax(), glResetMinmax()

glMultiTexCoordARB

- 名称：

**glMultiTexCoord1dARB(), glMultiTexCoord1fARB(), glMultiTexCoord1iARB(),
glMultiTexCoord1sARB(), glMultiTexCoord2dARB(), glMultiTexCoord2fARB(),
glMultiTexCoord2iARB(), glMultiTexCoord2sARB(), glMultiTexCoord3dARB(),
glMultiTexCoord3fARB(), glMultiTexCoord3iARB(), glMultiTexCoord3sARB(),
glMultiTexCoord4dARB(), glMultiTexCoord4fARB(), glMultiTexCoord4iARB(),
glMultiTexCoord4sARB(), glMultiTexCoord1dvARB(), glMultiTexCoord1fvARB(),
glMultiTexCoord1ivARB(), glMultiTexCoord1svARB(), glMultiTexCoord2dvARB(),
glMultiTexCoord2fvARB(), glMultiTexCoord2ivARB(), glMultiTexCoord2svARB(),
glMultiTexCoord3dvARB(), glMultiTexCoord3fvARB(), glMultiTexCoord3ivARB(),
glMultiTexCoord3svARB(), glMultiTexCoord4dvARB(), glMultiTexCoord4fvARB(),
glMultiTexCoord4ivARB(), glMultiTexCoord4svARB()**

- 功能：

设置当前纹理坐标。

- C 描述：

```
void glMultiTexCoord1dARB( GLenum target,
                           GLdouble s )
void glMultiTexCoord1fARB( GLenum target,
                           GLfloat s )
void glMultiTexCoord1iARB( GLenum target,
                           GLint s )
void glMultiTexCoord1sARB( GLenum target,
                           GLshort s )
void glMultiTexCoord2dARB( GLenum target,
                           GLdouble s,
                           GLdouble t )
void glMultiTexCoord2fARB( GLenum target,
                           GLfloat s,
                           GLfloat t )
void glMultiTexCoord2iARB( GLenum target,
                           GLint s,
                           GLint t )
```

```
void glMultiTexCoord2sARB( GLenum target,
                           GLshort s,
                           GLshort t )
void glMultiTexCoord3dARB( GLenum target,
                           GLdouble s,
                           GLdouble t,
                           GLdouble r )
void glMultiTexCoord3fARB( GLenum target,
                           GLfloat s,
                           GLfloat t,
                           GLfloat r )
void glMultiTexCoord3iARB( GLenum target,
                           GLint s,
                           GLint t,
                           GLint r )
void glMultiTexCoord3sARB( GLenum target,
                           GLshort s,
                           GLshort t,
                           GLshort r )
void glMultiTexCoord4dARB( GLenum target,
                           GLdouble s,
                           GLdouble t,
                           GLdouble r,
                           GLdouble q )
void glMultiTexCoord4fARB( GLenum target,
                           GLfloat s,
                           GLfloat t,
                           GLfloat r,
                           GLfloat q )
void glMultiTexCoord4iARB( GLenum target,
                           GLint s,
                           GLint t,
                           GLint r,
                           GLint q )
void glMultiTexCoord4sARB( GLenum target,
                           GLshort s,
                           GLshort t,
```

GLshort *r*,
GLshort *q*)

• 参数说明：

target 指定将被修改坐标的纹理单元。该纹理单元的数目由具体实现决定，但至少是2。它必须是**GL_TEXTURE*i*_ARB**之一，这里 $0 \leq i < \text{GL_MAX_TEXTURE_UNITS_ARB}$ 。

s, t, r, q

指定*target*纹理单元的*s*、*t*、*r*和*q*纹理坐标。并非所有命令都需要提供全部坐标。

• C描述：

```
void glMultiTexCoord1dvARB(GLenum target,
                           const GLdouble *v)
void glMultiTexCoord1fvARB(GLenum target,
                           const GLfloat *v)
void glMultiTexCoord1ivARB(GLenum target,
                           const GLint *v)
void glMultiTexCoord1svARB(GLenum target,
                           const GLshort *v)
void glMultiTexCoord2dvARB(GLenum target,
                           const GLdouble *v)
void glMultiTexCoord2fvARB(GLenum target,
                           const GLfloat *v)
void glMultiTexCoord2ivARB(GLenum target,
                           const GLint *v)
void glMultiTexCoord2svARB(GLenum target,
                           const GLshort *v)
void glMultiTexCoord3dvARB(GLenum target,
                           const GLdouble *v)
void glMultiTexCoord3fvARB(GLenum target,
                           const GLfloat *v)
void glMultiTexCoord3ivARB(GLenum target,
                           const GLint *v)
void glMultiTexCoord3svARB(GLenum target,
                           const GLshort *v)
void glMultiTexCoord4dvARB(GLenum target,
                           const GLdouble *v)
void glMultiTexCoord4fvARB(GLenum target,
                           const GLfloat *v)
```

```
void glMultiTexCoord4ivARB( GLenum target,
                           const GLint *v )
void glMultiTexCoord4svARB( GLenum target,
                           const GLshort *v )
```

- 参数说明：

target 指定将被修改坐标的纹理单元。该纹理单元的数目由具体实现决定，但至少是2。它必须是**GL_TEXTURE*i*_ARB**之一，这里 $0 \leq i < \text{GL_MAX_TEXTURE_UNITS_ARB}$ 。

v 指定一个指针，指向一个包含一个、二个、三个或四个元素的数组，它们分别指定纹理坐标*s*、*t*、*r*和*q*。

- 说明：

函数**glMultiTexCoordARB()**用来指定一维、二维、三维或四维的纹理坐标。函数**glMultiTexCoord1ARB()**将当前纹理坐标设置成(*s*, 0, 0, 1)，函数**glMultiTexCoord2ARB()**将当前纹理坐标设置成(*s*, *t*, 0, 1)，函数**glMultiTexCoord3ARB()**将当前纹理坐标设置成(*s*, *t*, *r*, 1)，函数**glMultiTexCoord4ARB()**将当前纹理坐标设置成(*s*, *t*, *r*, *q*)。当前纹理坐标是相应的每个顶点和当前光栅位置数据的一部分。在缺省情况下，*s*、*r*、*t*和*q*的值是(0, 0, 0, 1)。

- 注意：

当函数**glGetString(GL_EXTENSIONS)**的返回值是**GL_ARB_multitexture**时，才支持函数**glMultiTexCoordARB()**。

当前纹理坐标可被随时更新。特别是，函数**glMultiTexCoordARB()**可在函数对**glBegin()**/**glEnd()**之间被调用。

下面等式总是成立的：

$$\text{GL_TEXTURE}_i\text{_ARB} = \text{GL_TEXTURE}_0\text{_ARB} + i.$$

- 有关数据的获取：

glGet(GL_CURRENT_TEXTURE_COORDS)

- 请参阅：

glActiveTextureARB(), **glClientActiveTextureARB()**, **glTexCoord()**, **glTexCoordPointer()**, **glVertex()**

• **glMultMatrix**

- 名称：

glMultMatrixd(), **glMultMatrixf()**

- 功能：

用指定的矩阵乘以当前矩阵。

- C描述：

```
void glMultMatrixd(const GLdouble *m )
```

```
void glMultMatrixf( const GLfloat *m )
```

- 参数说明：

m 指定一个指针，指向16个连续的值。这16个值按列组成一个 4×4 的矩阵。

- 说明：

函数glMultMatrix()用*m*指定的矩阵乘以当前矩阵，并把结果保存为当前矩阵。

当前矩阵由当前矩阵模式（请参阅glMatrixMode()）来确定，它可以是投影矩阵、模式取景矩阵或纹理矩阵。

- 示例：

如果当前矩阵是*C*，且变换坐标为*v*=(*v*[0], *v*[1], *v*[2], *v*[3])。则当前变换为*C*×*v*，或：

$$\begin{array}{cccccc} c[0] & c[4] & c[8] & c[12] & & v[0] \\ c[1] & c[5] & c[9] & c[13] & & v[1] \\ c[2] & c[6] & c[10] & c[14] & \times & v[2] \\ c[3] & c[7] & c[11] & c[15] & & v[3] \end{array}$$

调用函数glMultMatrix()（其自变量是*m*=*m*[0], *m*[1], ...*m*[15]）替换的当前变换为(*C*×*M*)×*v*，或：

$$\begin{array}{cccccc} c[0] & c[4] & c[8] & c[12] & m[0] & m[4] & m[8] & m[12] & v[0] \\ c[1] & c[5] & c[9] & c[13] & m[1] & m[5] & m[9] & m[13] & v[1] \\ c[2] & c[6] & c[10] & c[14] & \times & m[2] & m[6] & m[10] & m[14] & \times & v[2] \\ c[3] & c[7] & c[11] & c[15] & & m[3] & m[7] & m[11] & m[15] & & v[3] \end{array}$$

这里符号“×”表示矩阵乘积，*v*表示一个 4×1 的矩阵。

- 注意：

矩阵中的元素可以被指定成单精度或双精度的值，GL系统至少可以把这些值当作单精度值进行存储或操作。

许多计算机语言中的 4×4 数组是按行序排列的。这里我们仅讨论按列序排列的矩阵的变换。相乘的顺序是非常重要的。比如，如果当前的变换是旋转变换，调用函数glMultMatrix()对一个矩阵进行变换，这时变换操作将直接在要变换的坐标上进行，旋转操作也将在其所得的结果上进行。

- 出错提示：

当函数glMultMatrixd()在函数对glBegin()/glEnd()之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

```
glGet( GL_MATRIX_MODE )
glGet( GL_COLOR_MATRIX )
glGet( GL_MODELVIEW_MATRIX )
glGet( GL_PROJECTION_MATRIX )
glGet( GL_TEXTURE_MATRIX )
```

- 请参阅：

glLoadIdentity(), glLoadMatrix(), glMatrixMode(), glPushMatrix()

• **glNewList, glEndList**

• 名称:

glNewList(), **glEndList()**

• 功能:

建立或替换一个显示列表。

• C描述:

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

• 参数说明:

list 指定显示列表的名称。

mode 指定编译模式。它可以是**GL_COMPILE**或**GL_COMPILE_AND_EXECUTE**。

• C描述:

```
void glEndList( void )
```

• 说明:

显示列表是一组GL命令，这些命令被存储起来随后再执行。函数**glNewList()**可以用来建立显示列表。所有后续执行的命令都被编入显示列表中，这些命令将按顺序发布，直到函数**glEndList()**被调用为止。

函数**glNewList()**带有两个自变量。第一个自变量*list*是一个正的整型值，它是显示列表的唯一名称。显示列表的名称可由函数**glGenLists()**建立并保存，函数**glIsList()**用来测试显示列表名称的唯一性。第二个自变量*mode*是一个符号常量，它可取的两个值是：

GL_COMPILE 命令只被编译。

GL_COMPILE_AND_EXECUTE 在命令被编译进入显示列表时要执行此命令。

有一些函数是不能被编译进入显示列表中的，因为它们不管显示列表模式如何都立刻执行。这些函数是**glAreTexturesResident()**、**glColorPointer()**、**glDeleteLists()**、**glDeleteTextures()**、**glDisableClientState()**、**glEdgeFlagPointer()**、**glEnableClientState()**、**glFeedbackBuffer()**、**glFinish()**、**glFlush()**、**glGenLists()**、**glGenTextures()**、**glIndexPointer()**、**glInterleavedArrays()**、**glIsEnabled()**、**glIsList()**、**glIsTexture()**、**glNormalPointer()**、**glPopClientAttrib()**、**glPixelStore()**、**glPushClientAttrib()**、**glReadPixels()**、**glRenderMode()**、**glSelectBuffer()**、**glTexCoordPointer()**、**glVertexPointer()**和所有**glGet()**命令。

类似地，如果函数**glTexImage1D()**、**glTexImage2D()**和**glTexImage3D()**的第一个参数是**GL_PROXY_TEXTURE_1D**、**GL_PROXY_TEXTURE_2D**和**GL_PROXY_TEXTURE_3D**时，这些函数也将立刻执行。

当系统支持**GL_ARB_imaging**扩展时，如果函数**glHistogram()**的参数是**GL_PROXY_HISTOGRAM**时，它将立刻执行。类似地，如果函数**glColorTable()**的参数是**GL_PROXY_COLOR_TABLE**、**GL_PROXY_POST_CONVOLUTION_COLOR_TABLE**或**GL_PROXY_POST_COLOR_MATRIX_COLOR_TABLE**时，它也将立刻执行。

当系统支持**GL_ARB_multitexture**扩展时，函数**glClientActiveTextureARB()**不能被编译进入显示列表而将立刻执行。

函数**glEndList()**执行后，将完成由函数**glNewList()**指定的名为*list*的显示列表建立。如果名为*list*的显示列表已经存在，执行函数**glEndList()**的结果是显示列表中的内容被替换。

- 注意：

函数**glCallList()**和**glCallLists()**可以是显示列表的一部分。即使显示列表的建立模式是**GL_COMPILE_AND_EXECUTE**，由函数**glCallList()**和**glCallLists()**执行的显示列表命令却不包含在将要建立的显示列表中。

显示列表只是一组命令和参数，所以当一个显示列表在正在执行时，如果其中的命令产生了错误，则必须产生出错提示。如果显示列表被建立为**GL_COMPILE**模式，那么只有在显示列表执行时才会有出错提示产生。

- 出错提示：

当参数*list*为0时产生**GL_INVALID_VALUE**提示。

当参数*mode*为不可接受的值时产生**GL_INVALID_ENUM**提示。

当函数**glEndList()**被调用但却没有与其相对应的函数**glNewList()**或当一个显示列表的建立过程中又一次调用了函数**glNewList()**，这时将产生**GL_INVALID_OPERATION**提示。

当函数**glNewList()**和**glEndList()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

如果编译显示列表时内存空间太少将产生**GL_OUT_OF_MEMORY**提示。在GL1.1以上的版本中，如果已经存在显示列表，则先前的显示器中的内容将不被改变。同时，GL的其他状态也不改变（就好象并没有试图建立新的显示列表）。

- 有关数据的获取：

glIsList()

glGet(GL_LIST_INDEX)

glGet(GL_LIST_MODE)

- 请参阅：

glCallList(), glCallLists(), glDeleteLists(), glGenLists()

• **glNormal**

- 名称：

glNormal3b(), **glNormal3d()**, **glNormal3f()**, **glNormal3i()**, **glNormal3s()**, **glNormal3bv()**,
glNormal3dv(), **glNormal3fv()**, **glNormal3iv()**, **glNormal3sv()**

- 功能：

设置当前法向量。

- C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 )

```

• 参数说明：

nx, ny, nz

指定新的当前法向量的x, y和z坐标。当前法线的初始值是单位向量(0, 0, 1)。

• C描述：

```

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 )

```

• 参数说明：

v 指定一个指针，指向一个包含三个元素的数组，这三个元素分别是新的当前法向量的x, y和z坐标。

• 说明：

当函数glNormal()执行时，当前的法向量由该函数提供的坐标重新设定。字节型、短整型和整型数据都被线性地映射为浮点格式：最大的正数映射为1.0，最小的负数映射为-1.0。

由函数glNormal()指定的法向量不必是单位长度的向量。如果已经启动了GL_NORMALIZE，函数glNormal()所指定的任意长度的法向量都将被归一化。如果已经启动了GL_RESCALE_NORMAL，法向量将由模式取景矩阵提供的缩放因子进行缩放。GL_RESCALE_NORMAL要求最初指定的法向量是单位长度的向量，并要求模式取景矩阵只包含统一的缩放因子以期得到较好的结果。调用函数glEnable()和glDisable()并取参数GL_NORMALIZE或GL_RESCALE_NORMAL可以启动和关闭归一化操作。初始情况下，归一化操作是关闭的。

• 注意：

当前法向量可被随时更新。特别是，函数glNormal()可在函数对glBegin()/glEnd()之间被调用。

- 有关数据的获取：

glGet(GL_CURRENT_NORMAL)
glIsEnabled(GL_NORMALIZE)
glIsEnabled(GL_RESCALE_NORMAL)

- 请参阅：

glBegin(), **glColor()**, **glIndex()**, **glNormalPointer()**, **glTexCoord()**, **glVertex()**

► **glNormalPointer**

- 名称：

glNormalPointer()

- 功能：

定义一个法向量数组。

- C 描述：

```
void glNormalPointer( GLenum type,
                     GLsizei stride,
                     const GLvoid *pointer )
```

- 参数说明：

type 指定数组中每个坐标的的数据类型。它可取符号常量**GL_BYTE**、**GL_SHORT**、**GL_INT**、**GL_FLOAT**和**GL_DOUBLE**。其初始值是**GL_FLOAT**。

stride 指定相邻的法线之间的字节偏移量。如果参数*stride*是0（初始值），则认为法线被一个接一个的排列在数组中。

pointer 指定一个指针，指向数组中的第一个法向量的第一个坐标。其初始值是0。

- 说明：

函数**glNormalPointer()**的作用是指定绘图时采用的法向量数组的存放位置和数据格式。参数*type*用来指定每个法向量的数据类型，参数*stride*给出了顶点和属性存放在一个数组中或分别存放在多个数组中的一个法向量到下一个法向量所允许的字节偏移量。（在一些GL实现中，单一数组的存放形式可能更有效。请参阅**glInterleavedArrays()**）。当一个法向量数组被指定后，参数*size*、*type*、*stride*和*pointer*将被存储为客户端状态。

用函数**glEnableClientState(GL_NORMAL_ARRAY)**和**glDisableClientState(GL_NORMAL_ARRAY)**可以启动和关闭法向量数组。当启动法向量数组后，它可以被函数**glDrawArrays()**、**glDrawElements()**或**glArrayElement()**使用。

函数**glDrawArrays()**用来将预先指定的顶点和顶点属性数组构造成一个图元序列（所有相同的类型）。函数**glArrayElement()**可以通过检索顶点和顶点的属性来指定图元。函数**glDrawElements()**则是通过检索顶点和顶点属性来构造一个图元序列。

- 注意：

函数**glNormalPointer()**只有在GL 1.1以上的版本中才可以使用。

法向量数组在初始情况下是关闭的，这时不能通过调用函数**glDrawArrays()**、**glDrawElement()**

或`glArrayElement()`对它进行访问。

函数`glNormalPointer()`不允许在函数对`glBegin()`/`glEnd()`之间执行，否则可能产生一个出错提示，但也有可能不出现出错提示。如果没有出现出错提示，其运行将是未定义的。

函数`glNormalPointer()`是通常在客户端执行。

因为法向量数组参数是客户端状态，所以它不能用函数`glPushAttrib()`和`glPopAttrib()`存储和恢复，而应由函数`glPushClientAttrib()`和`glPopClientAttrib()`代替。

- 出错提示：

当参数`type`不是一个可取的值时产生`GL_INVALID_ENUM`提示。

当参数`stride`是负数时产生`GL_INVALID_VALUE`提示。

- 有关数据的获取：

```
glIsEnabled( GL_NORMAL_ARRAY )
glGet( GL_NORMAL_ARRAY_TYPE )
glGet( GL_NORMAL_ARRAY_STRIDE )
glGetPointerv( GL_NORMAL_ARRAY_POINTER )
```

- 请参阅：

```
glArrayElement(), glColorPointer(), glDrawArrays(), glDrawElements(),
glEdgeFlagPointer(), glEnable(), glGetPointerv(), glIndexpointer(), glInterleaveArrays(),
glPopClientAttrib(), glPushClientAttrib(), glTexCoordPointer(), glVertexPointer()
```

- **glOrtho**

- 名称：

`glOrtho()`

- 功能：

用一个正交矩阵乘以当前矩阵。

- C 描述：

```
void glOrtho( GLdouble left,
              GLdouble right,
              GLdouble bottom,
              GLdouble top,
              GLdouble zNear,
              GLdouble zFar )
```

- 参数说明：

`left, right`

指定左、右垂直剪切平面的坐标。

`bottom, top`

指定下、上水平剪切平面的坐标。

`zNear, zFar`

指定视点到最近和最远深度剪切平面的距离。

如果剪切平面位于视点的后面，这些值将是负数。

- 说明：

函数glOrtho()提供了一种产生平行投影的变换方法。当前矩阵（请参阅glMatrixMode()）被正交矩阵与当前矩阵相乘的结果所替代。就好象调用函数glMultMatrix()时使用下面矩阵作为它的参数：

$$\begin{matrix} \frac{2}{right-left} & 0 & 0 & t_x \\ 0 & \frac{2}{top-bottom} & 0 & t_y \\ 0 & 0 & \frac{-2}{zFar-zNear} & t_z \\ 0 & 0 & 0 & 1 \end{matrix}$$

这里

$$t_x = -\frac{right + left}{right - left}$$

$$t_y = -\frac{top + bottom}{top - bottom}$$

$$t_z = -\frac{zFar + zNear}{zFar - zNear}$$

通常，矩阵模式为**GL_PROJECTION**，且(*left*, *bottom*, *-zNear*)和(*right*, *top*, *-zNear*)指定了最近的剪切平面上、分别映射到窗口坐标系中的左下角和右上角的点。假定这时的视点在(0, 0, 0)位置。*-zFar*指定了最远的剪切平面的位置。*zNear*和*zFar*既可以是正数也可以是负数。

函数glPushMatrix()和glPopMatrix()用来存储和恢复当前矩阵堆栈。

- 出错提示：

当函数glOrtho()在函数对glBegin()/glEnd()之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGet(GL_MATRIX_MODE)

glGet(GL_COLOR_MATRIX)

glGet(GL_MODELVIEW_MATRIX)

glGet(GL_PROJECTION_MATRIX)

glGet(GL_TEXTURE_MATRIX)

- 请参阅：

glFrustum(), **glMatrixMode()**, **glMultMatrix()**, **glPushMatrix()**, **glViewport()**

- **glPassThrough**

- 名称:

glPassThrough()

- 功能:

在反馈缓冲区中插入一个标记。

- C描述:

void glPassThrough(GLfloat token)

- 参数说明:

token 指定一个插入反馈缓冲区中的标记值。这一值将跟在**GL_PASS_THROUGH_TOKEN**后面。

- 说明:

反馈是一种GL绘图模式。这一模式通过调用函数**glRenderMode(GL_FEEDBACK)**来选取。当GL在反馈模式下时，不能通过光栅化来产生像素。相反，光栅化的图元的信息将被GL反馈回应用程序。请参阅**glFeedbackBuffer()**获取有关反馈缓冲区及其中的值的相关信息。

反馈模式下，当函数**glPassThrough()**执行时，它将在反馈缓冲区中插入一个用户定义的标记。参数*token*将被返回，就好象它是一个图元。它由其唯一的识别值**GL_PASS_THROUGH_TOKEN**指定。函数**glPassThrough()**指定的有关图元性质的次序将被保留。

- 注意:

如果GL不在反馈模式下，函数**glPassThrough()**将被忽略。

- 出错提示:

当函数**glPassThrough()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示

- 有关数据的获取:

glGet(GL_RENDER_MODE)

- 请参阅:

glFeedbackBuffer(), **glRenderMode()**

glPixelMap

- 名称:

glPixelMapfv(), **glPixelMapuiv()**, **glPixelMapusv()**

- 功能:

建立像素转换映射。

- C描述:

```
void glPixelMapfv( GLenum map,
                    GLint mapsize,
                    const GLfloat *values )
void glPixelMapuiv( GLenum map,
                    GLint mapsize,
                    const GLuint *values )
```

```
void glPixelMapusv( GLenum map,
                     GLint mapsize,
                     const GLushort *values )
```

• 参数说明：

map 指定一个符号型的映射名称。它必须取下面符号常量之一：**GL_PIXEL_MAP_I_TO_I**、**GL_PIXEL_MAP_S_TO_S**、**GL_PIXEL_MAP_I_TO_R**、**GL_PIXEL_MAP_I_TO_G**、**GL_PIXEL_MAP_I_TO_B**、**GL_PIXEL_MAP_I_TO_A**、**GL_PIXEL_MAP_R_TO_R**、**GL_PIXEL_MAP_G_TO_G**、**GL_PIXEL_MAP_B_TO_B**和**GL_PIXEL_MAP_A_TO_A**。

mapsize 指定被定义的映射的尺寸。

values 指定一个存放参数*mapsize*值的数组。

• 说明：

函数glPixelMap()可以建立转换表或映射。这些转换表和映射将被下列函数使用：glCopyPixels()、glCopyTexImage1D()、glCopyTexImage2D()、glCopyTexSubImage1D()、glCopyTexSubImage2D()、glCopyTexSubImage3D()、glDrawPixels()、glReadPixels()、glTexImage1D()、glTexImage2D()、glTexImage3D()、glTexSubImage1D()、glTexSubImage2D()和glTexSubImage3D()。此外，如果系统支持**GL_ARB_imaging**子集，函数glColorTable()、glColorSubTable()、glConvolutionFilter1D()、glConvolutionFilter2D()、glHistogram()、glMinmax()和glPixelTransfer()也将用到这些转换表和映射。函数glPixelTransfer()的参考说明中对如何使用这些映射表作了详细的说明。在像素和纹理图像的有关章节中也对这一问题作了部分说明。这里我们仅就这些映射本身作出说明。

参数*map*是一个符号型的映射名称，它用来说明具体采取十种映射方式中的哪一种映射方式。参数*mapsize*用来指定映射表中条目的数目。参数*value*是一个指针，指向一个数组，该数组中存放着参数*mapsize*的具体值。

十种具体的映射方式如下：

GL_PIXEL_MAP_I_TO_I	颜色索引到颜色索引的映射。
GL_PIXEL_MAP_S_TO_S	模板索引到模板索引的映射。
GL_PIXEL_MAP_I_TO_R	颜色索引到红组分的映射。
GL_PIXEL_MAP_I_TO_G	颜色索引到绿组分的映射。
GL_PIXEL_MAP_I_TO_B	颜色索引到蓝组分的映射。
GL_PIXEL_MAP_I_TO_A	颜色索引到alpha组分的映射。
GL_PIXEL_MAP_R_TO_R	红组分到红组分的映射。
GL_PIXEL_MAP_G_TO_G	绿组分到绿组分的映射。
GL_PIXEL_MAP_B_TO_B	蓝组分到蓝组分的映射。
GL_PIXEL_MAP_A_TO_A	alpha组分到alpha组分的映射。

映射表中的条目可以被定义为单精度浮点数、无符号短整型数或无符号长整型数。用于存储颜色组分值的映射表（除**GL_PIXEL_MAP_I_TO_I**和**GL_PIXEL_MAP_S_TO_S**）都被保存为带

有不确定的尾数和指数大小的浮点格式。由函数glPixelMapfv()指定的浮点值被直接转化为映射表的浮点格式，并被截断到范围[0, 1]内。由函数glPixelMapusv()和glPixelMapuiv()指定的无符号整型值被用如下的方法线性映射为浮点格式：最大的正数映射为1.0，0映射为0.0。

由映射方式GL_PIXEL_MAP_I_TO_I和GL_PIXEL_MAP_S_TO_S产生的用于存储索引值的映射被保存为一个固定点格式，把若干个不确定的位补到二进制点的右边。由函数glPixelMapfv()指定的浮点值被直接转化为这些映射表所要求的固定点格式。由函数glPixelMapusv()和glPixelMapuiv()指定的无符号整型值被用如下的方法线性映射为浮点格式：二进制点的右边全部补0。

表5-23显示了每一种映射的初始尺寸和初始值。由颜色索引和模板索引指定的映射必须存在关系式 $mapsize=2^n$ ，这里n为某一不确定的值。每个映射的尺寸与你所用的设备有关，可用函数glGet(GL_MAP_PIXEL_MAP_TABLE)来查询。为每一个映射提供的尺寸值最小是32。

表 5-23

map (映射方式)	查询索引	查询值	初始尺寸	初始值
GL_PIXEL_MAP_I_TO_I	颜色索引	颜色索引	1	0
GL_PIXEL_MAP_S_TO_S	模板索引	模板索引	1	0
GL_PIXEL_MAP_I_TO_R	颜色索引	R	1	0
GL_PIXEL_MAP_I_TO_G	颜色索引	G	1	0
GL_PIXEL_MAP_I_TO_B	颜色索引	B	1	0
GL_PIXEL_MAP_I_TO_A	颜色索引	A	1	0
GL_PIXEL_MAP_R_TO_R	R	R	1	0
GL_PIXEL_MAP_G_TO_G	G	G	1	0
GL_PIXEL_MAP_B_TO_B	B	B	1	0
GL_PIXEL_MAP_A_TO_A	A	A	1	0

- 出错提示：

当参数map是一个不可接受的值时产生GL_INVALID_ENUM提示。

当参数mapsize小于1或大于GL_MAX_PIXEL_MAP_TABLE时产生GL_INVALID_VALUE提示。

当参数map是GL_PIXEL_MAP_I_TO_I、GL_PIXEL_MAP_S_TO_S、GL_PIXEL_MAP_I_TO_R、GL_PIXEL_MAP_I_TO_G、GL_PIXEL_MAP_I_TO_B、GL_PIXEL_MAP_I_TO_A，但参数mapsize不是2的幂值时产生GL_INVALID_VALUE提示。

当函数glPixelMap()在函数对glBegin()/glEnd()之间执行时产生GL_INVALID_OPERATION提示。

- 有关数据的获取：

glGetPixelMap()

glGet(GL_PIXEL_MAP_I_TO_I_SIZE)

glGet(GL_PIXEL_MAP_S_TO_S_SIZE)

glGet(GL_PIXEL_MAP_I_TO_R_SIZE)

glGet(GL_PIXEL_MAP_I_TO_G_SIZE)

`glGet(GL_PIXEL_MAP_I_TO_B_SIZE)
 glGet(GL_PIXEL_MAP_I_TO_A_SIZE)
 glGet(GL_PIXEL_MAP_R_TO_R_SIZE)
 glGet(GL_PIXEL_MAP_G_TO_G_SIZE)
 glGet(GL_PIXEL_MAP_B_TO_B_SIZE)
 glGet(GL_PIXEL_MAP_A_TO_A_SIZE)
 glGet(GL_MAX_PIXEL_MAP_TABLE)`

- 请参阅：

`glColorTable(), glColorSubTable(), glConvolutionFilter1D(), glConvolutionFilter2D(),
 glCopyPixels(), glCopyTexImage1D(), glCopyTexImage2D(), glCopyTexSubImage1D(),
 glCopyTexSubImage2D(), glDrawPixels(), glHistogram(), glMinmax(), glPixelStore(),
 glPixelTransfer(), glReadPixels(), glSeparableFilter2D(), glTexImage1D(), glTexImage2D(),
 glTexImage3D(), glTexSubImage1D(), glTexSubImage2D(), glTexSubImage3D()`

• `glPixelStore`

- 名称：

`glPixelStoref(), glPixelStorei()`

- 功能：

设置像素存储模式。

- C 描述：

```
void glPixelStoref( GLenum pname,  

                    GLfloat param )  

void glPixelStorei( GLenum pname,  

                    GLint param )
```

- 参数说明：

pname 指定要设置参数的符号型名称。它可取的值有：(1) 影响像素打包进内存的六个值：`GL_PACK_SWAP_BYTES`、`GL_PACK_LSB_FIRST`、`GL_PACK_ROW_LENGTH`、`GL_PACK_IMAGE_HEIGHT`、`GL_PACK_SKIP_PIXELS`、`GL_PACK_SKIP_ROWS`、`GL_PACK_SKIP_IMAGES`和`GL_PACK_ALIGNMENT`；(2) 影响从内存中将数据解包的六个值：`GL_UNPACK_SWAP_BYTES`、`GL_UNPACK_LSB_FIRST`、`GL_UNPACK_ROW_LENGTH`、`GL_UNPACK_IMAGE_HEIGHT`、`GL_UNPACK_SKIP_PIXELS`、`GL_UNPACK_SKIP_ROWS`、`GL_UNPACK_SKIP_IMAGES`和`GL_UNPACK_ALIGNMENT`。

param 指定参数*pname*的具体设定值。

- 说明：

函数`glPixelStore()`用来设置像素的存储模式，它将影响后续函数`glDrawPixels()`和`glReadPixels()`的操作，就象解包多边形的点画模式（请参阅`glPolygonStipple()`）、位图（请参阅`glBitmap()`）

和纹理模式（请参阅glTexImage1D()、glTexImage2D()、glTexImage3D()、glTexSubImage1D()、glTexSubImage2D()和glTexSubImage3D()）。此外，如果系统支持GL_ARB_imaging扩展，像素存储模式将对卷积滤波器（请参阅glConvolutionFilter1D()、glConvolutionFilter2D()和glSeparableFilter2D()）、颜色表（请参阅glColorTable()和glColorSubTable()）、解包直方图（请参阅glHistogram()）和极值（请参阅glMinmax()）数据等产生影响。

参数`pname`是一个符号常量，它表示被设置的参数。参数`param`是一个新的设定值。在十二种存储模式参数中有六种将对数据返回客户端内存的方式产生影响。它们是：

GL_PACK_SWAP_BYTES

如果该参数为TRUE，内存中的多字节颜色组分、深度组分、颜色索引或模板索引的次序将被颠倒。也就是说，如果一个四字节组分是由字节 b_0 、 b_1 、 b_2 和 b_3 构成的，则当GL_PACK_SWAP_BYTES为TRUE时，这个四字节组分在内存中的存放顺序为 b_3 、 b_2 、 b_1 和 b_0 。参数GL_PACK_SWAP_BYTES不影响单个像素的内部组分在内存中的排列次序，它只影响组分或索引之间的字节的排列次序。例如，一个GL_RGB格式的像素总是按红、绿、蓝的次序存储的，它不受GL_PACK_SWAP_BYTES的值影响。

GL_PACK_LSB_FIRST

如果该参数为TRUE，在字节中的二进制位的排列次序是从最次要的位开始一直排到最重要的位；否则，字节中的第一位将用来放置最重要的位。该参数只对位图数据有效。

GL_PACK_ROW_LENGTH

如果该参数大于0，它将用来定义在一行中像素的数目。如果一行中的第一个像素在内存中位于第 p 个存储单元，那么，存储下一行中的第一个像素的存储单元将跳过 k 个组分或索引而得到，这里 k 为：

$$k = \begin{cases} nl & s \geq a \\ \frac{a}{s} \frac{snl}{a} & s < a \end{cases}$$

上式中， n 是每一个像素中包含的组分或索引的个数； l 是每一行中包含的像素的个数（当GL_PACK_ROW_LENGTH大于0时；否则，由像素例程的`width`参数替代。）； a 是参数GL_PACK_ALIGNMENT的值； s 是每个组分中的字节大小（当 $a < s$ 时，它相当于 $a=s$ ）。当值中只有一个二进制位时，下一行的存储位置是在跳过 k 个组分或索引的地方。这里

$$k = 8a \frac{nl}{8a}$$

上面所提到的“组分”是指非索引值的红、绿、蓝、alpha和深度值。比如，在GL_RGB存储模式下，每一像素含有三个组分，依次为红、绿、蓝。

GL_PACK_IMAGE_HEIGHT

如果该参数大于0，它将用于定义在一个图像的三维纹理体积中像素的数目。这里“图像”是由所有共用相同的第三维索引的像素所定义的。如果一行中的第一个像素在内存中位于第 p 个存储单元，那么，下一行中的第一个像素的存储位置将跳过 k 个组分或索引而得到，这里 k 为：

$$k = \begin{cases} nlh & s \geq a \\ \frac{a}{s} \frac{snlh}{a} & s < a \end{cases}$$

上式中， n 是每个像素中所包含的组分或索引的数目； l 是每一行中包含的像素的数目（当**GL_PACK_ROW_LENGTH**大于0时；否则，由函数**glTexImage3d()**的**width**参数替代）； h 是每个像素图像中的行数（当**GL_PACK_IMAGE_HEIGHT**大于0时；否则，由函数**glTexImage3d()**的**height**参数替代）。 a 是参数**GL_PACK_ALIGNMENT**的值； s 是每个组分中的字节大小（当 $a < s$ 时，相当于 $a=s$ ）。

上面所提到的“组分”是指非索引值的红、绿、蓝、alpha和深度值。比如，在**GL_RGB**存储模式下，每一像素含有三个组分，依次为红、绿、蓝。

GL_PACK_SKIP_PIXELS, GL_PACK_SKIP_ROWS和GL_PACK_SKIP_IMAGES

这些值为程序设计人员提供了很大的便利，它们所提供的功能都能通过增加传送给函数**glReadPixels()**的指针而被简单地复制。将**GL_PACK_SKIP_PIXELS**设置为*i*就相当于增加*in*个组分或索引指针。这里*n*是每个像素中包含的组分或索引的数目。将**GL_PACK_SKIP_ROWS**设置为*j*就相当于增加*jm*个组分或索引指针。这里*m*是每一行中包含的组分或索引的数目。它同**GL_PACK_ROW_LENGTH**中所用的计算方法一样。将**GL_PACK_SKIP_IMAGES**设置为*k*就相当于增加*kp*个组分或索引指针。这里*p*是每一个图像中包含的组分或索引的数目。它同**GL_PACK_IMAGE_HEIGHT**中所用的计算方法一样。

GL_PACK_ALIGNMENT

指定在内存中每一像素行的开始位置的排列方式。它的允许值为1（按单字节排列）、2（按双字节排列）、4（按字排列）和8（按双字排列）。

其他的六种存储参数用来控制如何从客户端内存中读出像素数据。这些值对函数**glDrawPixels()**、**glTexImage1D()**、**glTexImage2D()**、**glTexImage3D()**、**glTexSubImage1D()**、**glTexSubImage2D()**、**glTexSubImage3D()**、**glBitmap()**和**glPolygonStipple()**都有很重要的意义。

另外，如果系统支持**GL_ARB_imaging**扩展，它们还将对函数**glColorTable()**、**glColorSubTable()**、**glConvolutionFilter1D()**、**glConvolutionFilter2D()**和**glSeparableFilter2D()**产生重大的影响。这些参数是：

GL_UNPACK_SWAP_BYTES

如果该参数为**TRUE**，内存中的多字节颜色组分、深度组分、颜色索引或模板索

引的次序将被颠倒。也就是说，如果一个四字节组分是由字节 b_0 、 b_1 、 b_2 和 b_3 构成的，当**GL_UNPACK_SWAP_BYTES**为TRUE时，这个四字节组分在内存中的存放顺序为 b_3 、 b_2 、 b_1 和 b_0 。参数**GL_UNPACK_SWAP_BYTES**不影响单个像素的内部组分在内存中的排列次序，它只影响组分或索引之间的字节的排列次序。如，一个**GL_RGB**格式的像素总是按红、绿、蓝的次序存储的，它不受**GL_UNPACK_SWAP_BYTES**的值影响。

GL_UNPACK_LSB_FIRST

如果该参数为TRUE，在字节中的二进制位的排列次序是从最次要的位开始一直排到最重要的位；否则，字节中的第一位将用来放置最重要的位。该参数只对位图数据有效。

GL_UNPACK_ROW_LENGTH

如果该参数大于0，它将用于定义在一行中像素的数目。如果一行中的第一个像素在内存中位于第 p 个存储单元，那么，下一行中的第一个像素的存储位置将跳过 k 个组分或索引而得到，这里 k 为：

$$k = \begin{cases} nl & s \geq a \\ \frac{a}{s} \frac{snl}{a} & s < a \end{cases}$$

上式中， n 是每个像素中包含的组分或索引的数目； l 是每行中包含的像素的数目（当**GL_UNPACK_ROW_LENGTH**大于0时；否则，由像素例程的width参数替代）； a 是参数**GL_UNPACK_ALIGNMENT**的值； s 是每个组分中的字节大小（当 $a < s$ 时，相当于 $a = s$ ）。当值中只有一个二进制位时，下一行的存储位置是在跳过 k 个组分或索引的地方。这里

$$k = 8a \frac{nl}{8a}$$

上面所提到的“组分”是指非索引值的红、绿、蓝、alpha和深度值。比如，在**GL_RGB**存储模式下，每一像素含有三个组分，依次为红、绿、蓝。

GL_UNPACK_IMAGE_HEIGHT

如果该参数大于0，它将用于定义在一个图像的三维纹理体积中像素的数目。这里“图像”是由所有具有相同的第三维索引的像素所定义的。如果一行中的第一个像素在内存中位于第 p 个存储单元，那么，下一行中的第一个像素的存储位置将跳过 k 个组分或索引而得到，这里 k 为：

$$k = \begin{cases} nlh & s \geq a \\ \frac{a}{s} \frac{snlh}{a} & s < a \end{cases}$$

上式中， n 是每个像素中包含的组分或索引的个数； l 是每行中包含的像素的数目

(当**GL_UNPACK_ROW_LENGTH**大于0时；否则，由函数glTexImage3d()的width参数替代)；*h*是像素图像中的行数(当**GL_UNPACK_IMAGE_HEIGHT**大于0时；否则，由函数glTexImage3d()的height参数替代)。*a*是参数**GL_UNPACK_ALIGNMENT**的值；*s*是每个组分中的字节大小(当*a*<*s*时，相当于*a*=*s*)。

上面所提到的“组分”是值非索引值的红、绿、蓝、alpha和深度值。比如，在**GL_RGB**存储模式下，每一像素含有三个组分，依次为红、绿、蓝。

GL_UNPACK_SKIP_PIXELS、**GL_UNPACK_SKIP_ROWS**

这些值为程序设计人员提供了很大的便利，它们所提供的功能都能通过增加传送给函数glDrawPixels()、glTexImage1D()、glTexImage2D()、glTexSubImage1D()、glTexSubImage2D()、glBitmap()或glPolygonStipple的指针而被简单地复制。将**GL_UNPACK_SKIP_PIXELS**设置为*i*就相当于增加*in*个组分或索引指针。这里*n*是每个像素中包含的组分或索引的数目。将**GL_UNPACK_SKIP_ROWS**设置为*j*就相当于增加*jm*个组分或索引指针。这里*m*是每行中包含的组分或索引的数目。它同**GL_UNPACK_ROW_LENGTH**中所使用的计算方法一样。

GL_UNPACK_ALIGNMENT

指定在内存中每一像素行的开始位置的排列方式。它的允许值为1(按字节排列)、2(按双字节排列)、4(按字排列)和8(按双字排列)。

表5-24给出了函数glPixelStore()设定的存储参数值的类型、初值和有效范围。

表 5-24

<i>pname</i>	类 型	初 值	有 效 范 围
GL_PACK_SWAP_BYTES	boolean	false	true 或 false
GL_PACK_LSB_FIRST	boolean	false	true 或 false
GL_PACK_ROW_LENGTH	integer	0	[0, ∞)
GL_PACK_IMAGE_HEIGHT	integer	0	[0, ∞)
GL_PACK_SKIP_PIXELS	integer	0	[0, ∞)
GL_PACK_SKIP_ROWS	integer	0	[0, ∞)
GL_PACK_SKIP_IMAGES	integer	0	[0, ∞)
GL_PACK_ALIGNMENT	integer	1	1, 2, 4 或8
GL_UNPACK_SWAP_BYTES	boolean	false	true 或 false
GL_UNPACK_LSB_FIRST	boolean	false	true 或 false
GL_UNPACK_ROW_LENGTH	integer	0	[0, ∞)
GL_UNPACK_IMAGE_HEIGHT	integer	0	[0, ∞)
GL_UNPACK_SKIP_PIXELS	integer	0	[0, ∞)
GL_UNPACK_SKIP_ROWS	integer	0	[0, ∞)
GL_UNPACK_SKIP_IMAGES	integer	0	[0, ∞)
GL_UNPACK_ALIGNMENT	integer	1	1, 2, 4 或8

函数glPixelStoref()能用来设置任何类型的像素存储参数。如果参数类型是boolean，则当参数param被设置成0时表示FALSE；否则表示TRUE。如果参数pname是一个整型参数，则参数

*param*被圆整为一个最接近的整数。

同样，函数glPixelStorei()也能设置任何类型的像素存储参数。如果参数类型是boolean，则当参数*param*为0时表示FALSE，否则表示TRUE。

- 注意：

当函数glDrawPixels()、glReadPixels()、glTexImage1D()、glTexImage2D()、glTexImage3D()、glTexSubImage1D()、glTexSubImage2D()、glTexSubImage3D()、glBitmap()或glPolygonStipple()被放入一个显示列表中时，像素存储模式将会影响内存中数据的编译。同样，如果系统支持GL_ARB_imaging扩展，当函数glColorTable()、glColorSubTable()、glConvolutionFilter1D()、glConvolutionFilter2D()和glSeparableFilter2D()被放入一个显示列表中时，像素存储模式也将会对内存中的数据编译产生影响。当一个显示列表正在执行时，像素存储模式的影响将是无效的。

像素存储模式是一种客户端状态模式，它必须用函数glPushClientAttrib()和glPopClientAttrib()来存储和恢复。

- 出错提示：

当参数*pname*不是一个可接受的值时产生**GL_INVALID_ENUM**提示。

当行的长度、跳过的像素个数或跳过的行数被指定成一个负数或指定的排列模式不是值1、2、4或8时产生**GL_INVALID_VALUE**提示。

当函数glPixelStore()在函数对glBegin()/glEnd()之间执行时产生**GL_INVALID_OPERATION**提示

- 有关数据的获取：

```
glGet( GL_PACK_SWAP_BYTES )
glGet( GL_PACK_LSB_FIRST )
glGet( GL_PACK_ROW_LENGTH )
glGet( GL_PACK_IMAGE_HEIGHT )
glGet( GL_PACK_SKIP_ROWS )
glGet( GL_PACK_SKIP_PIXELS )
glGet( GL_PACK_SKIP_IMAGES )
glGet( GL_PACK_ALIGNMENT )
glGet( GL_UNPACK_SWAP_BYTES )
glGet( GL_UNPACK_LSB_FIRST )
glGet( GL_UNPACK_ROW_LENGTH )
glGet( GL_UNPACK_IMAGE_HEIGHT )
glGet( GL_UNPACK_SKIP_ROWS )
glGet( GL_UNPACK_SKIP_PIXELS )
glGet( GL_UNPACK_SKIP_IMAGES )
glGet( GL_UNPACK_ALIGNMENT )
```

- 请参阅：

glBitmap(),glColorTable(), glColorSubTable(), glConvolutionFilter1D(), glConvolution-

`Filter2D()`, `glSeparableFilter2D()`, `glDrawPixels()`, `glHistogram()`, `glMinmax()`, `glPixelMap()`,
`glPixelTransfer()`, `glPixelZoom()`, `glPolygonStipple()`, `glPushClientAttrib()`, `glReadPixels()`,
`glTexImage1D()`, `glTexImage2D()`, `glTexImage3D()`, `glTexSubImage1D()`, `glTexSubImage2D()`,
`glTexSubImage3D()`

• `glPixelTransfer`

- 名称:

`glPixelTransferf()`, `glPixelTransferi()`

- 功能:

设置像素转换模式。

- C 描述:

```
void glPixelTransferf( GLenum pname,
                      GLfloat param )
```

```
void glPixelTransferi( GLenum pname,
                      GLint param )
```

- 参数说明:

pname 指定要设置的像素转换参数的符号型名称。它可取的值为: `GL_MAP_COLOR`、
`GL_MAP_STENCIL`、`GL_INDEX_SHIFT`、`GL_INDEX_OFFSET`、
`GL_RED_SCALE`、`GL_RED_BIAS`、`GL_GREEN_SCALE`、`GL_GREEN_BIAS`、
`GL_BLUE_SCALE`、`GL_BLUE_BIAS`、`GL_ALPHA_SCALE`、`GL_ALPHA_BIAS`、`GL_DEPTH_SCALE` 或 `GL_DEPTH_BIAS`。

此外, 如果系统支持`GL_ARB_imaging`扩展, 它还可取下面的符号型名称:

`GL_POST_COLOR_MATRIX_RED_SCALE`, `GL_POST_COLOR_MATRIX_GREEN_SCALE`,
`GL_POST_COLOR_MATRIX_BLUE_SCALE`, `GL_POST_COLOR_MATRIX_ALPHA_SCALE`, `GL_POST_COLOR_MATRIX_RED_BIAS`,
`GL_POST_COLOR_MATRIX_GREEN_BIAS`, `GL_POST_COLOR_MATRIX_BLUE_BIAS`, `GL_POST_COLOR_MATRIX_ALPHA_BIAS`,
`GL_POST_CONVOLUTION_RED_SCALE`, `GL_POST_CONVOLUTION_GREEN_SCALE`, `GL_POST_CONVOLUTION_BLUE_SCALE`,
`GL_POST_CONVOLUTION_ALPHA_SCALE`, `GL_POST_CONVOLUTION_RED_BIAS`, `GL_POST_CONVOLUTION_GREEN_BIAS`,
`GL_POST_CONVOLUTION_BLUE_BIAS` 和 `GL_POST_CONVOLUTION_ALPHA_BIAS`。

param 指定参数*pname*的具体设定值。

- 说明:

函数`glPixelTransfer()`设定的像素转换模式将对随后的一些函数的操作产生影响。这些函数有:
`glCopyPixels()`、`glCopyTexImage1D()`、`glCopyTexImage2D()`、`glCopyTexSubImage1D()`、

`glCopyTexSubImage2D()`、`glCopyTexSubImage3D()`、`glDrawPixels()`、`glReadPixels()`、`glTexImage1D()`、`glTexImage2D()`、`glTexImage3D()`、`glTexSubImage1D()`、`glTexSubImage2D()`和`glTexSubImage3D()`。此外，如果系统支持`GL_ARB_imaging`子集，函数`glColorTable()`、`glColorSubTable()`、`glConvolutionFilter1D()`、`glConvolutionFilter2D()`、`glHistogram()`、`glMinmax()`和`glSeparableFilter2D()`也将受到影响。当从帧缓冲区读出像素（用函数`glCopyPixels()`、`glCopyTexImage1D()`、`glCopyTexImage2D()`、`glCopyTexSubImage1D()`、`glCopyTexSubImage2D()`、`glCopyTexSubPimage3D()`和`glReadPixels()`），或由客户端内存中解包（用函数`glDrawPixels()`、`glTexImage1D()`、`glTexImage2D()`、`glTexImage3D()`、`glTexSubImage1D()`、`glTexSubImage2D()`和`glTexSubImage3D()`）时，不管对像素进行操作的命令如何，像素的转换操作的发生次序及方式都是一样的。像素的存储模式（请参阅`glPixelStore()`）将控制从客户端内存中读出像素的解包过程和把像素写入客户端内存的打包过程。

像素转换操作将对四种基本的像素类型进行处理：*color*（颜色）、*color index*（颜色索引）、*depth*（深度）和*stencil*（模板）。*Color*像素由四个浮点值组成，这些值带有不确定的尾数和指数大小，并且这些值将按适当的比例进行缩放：这时0代表零强度，1代表满强度。*Color index*包含一个定点值，此时二进制小数点右边含有未指定的精度。*Depth*像素包含一个浮点值，这些值带有未定义的尾数和指数大小，并且这些值将按适当的比例进行缩放，这时0.0代表最小的深度缓存值，1.0代表最大的深度缓存值。*Stencil*像素包括一个定点值，此时二进制小数点右边含有未指定的精度。

像素转换操作所处理的四种基本的像素类型如下：

Color	四个颜色组分中的每一个都将乘上一个缩放比例因子，再加上一个偏移量。即，红组分将乘上缩放比例因子 <code>GL_RED_SCALE</code> ，再加上偏移量 <code>GL_RED_BIAS</code> ；绿组分将乘上缩放比例因子 <code>GL_GREEN_SCALE</code> ，再加上偏移量 <code>GL_GREEN_BIAS</code> ；蓝组分将乘上缩放比例因子 <code>GL_BLUE_SCALE</code> ，再加上偏移量 <code>GL_BLUE_BIAS</code> ；alpha组分将乘上缩放比例因子 <code>GL_ALPHA_SCALE</code> ，再加上偏移量 <code>GL_ALPHA_BIAS</code> 。经过以上的变换后，所有颜色组分还将被截断到范围[0, 1]内。所有颜色、缩放因子和偏移量的值都由函数 <code>glPixelTransfer()</code> 确定。
-------	--

如果`GL_MAP_COLOR`是TRUE，每一个颜色组分都由相应的颜色 - 颜色的映射表中的尺寸值进行缩放，然后由缩放后的值所指定的颜色索引表中相应的值替换。即，红组分先被`GL_PIXEL_MAP_R_TO_R_SIZE`进行缩放，然后由它所指定的`GL_PIXEL_MAP_R_TO_R`索引表中的值所替换。绿组分先被`GL_PIXEL_MAP_G_TO_G_SIZE`进行缩放，然后由它所指定的`GL_PIXEL_MAP_G_TO_G`索引表中的值所替换。蓝组分先被`GL_PIXEL_MAP_B_TO_B_SIZE`进行缩放，然后由它所指定的`GL_PIXEL_MAP_B_TO_B`索引表中的值所替换。alpha组分先被`GL_PIXEL_MAP_A_TO_A_SIZE`进行缩放，然后由它所指定的`GL_PIXEL_MAP_A_TO_A`索引表中的值所替换。所有从映射表中得到

的组分值还将被截断到范围[0, 1]内。值**GL_MAP_COLOR**由函数**glPixelTransfer()**指定。各种映射表的内容由函数**glPixelMap()**指定。

如果系统支持**GL_ARB_imaging**扩展，每个颜色的四个组分可以经过颜色矩阵转换后再进行缩放和偏移。即，红组分将乘上缩放比例因子**GL_POST_COLOR_MATRIX_RED_SCALE**，再加上偏移量**GL_POST_COLOR_MATRIX_RED_BIAS**；绿组分将乘上缩放比例因子**GL_POST_COLOR_MATRIX_GREEN_SCALE**，再加上偏移量**GL_POST_COLOR_MATRIX_GREEN_BIAS**；蓝组分将乘上缩放比例因子**GL_POST_COLOR_MATRIX_BLUE_SCALE**，再加上偏移量**GL_POST_COLOR_MATRIX_BLUE_BIAS**；alpha组分将乘上缩放比例因子**GL_POST_COLOR_MATRIX_ALPHA_SCALE**，再加上偏移量**GL_POST_COLOR_MATRIX_ALPHA_BIAS**。经过以上的变换后，所有颜色组分还将被截断到范围[0, 1]内。

类似地，如果系统支持**GL_ARB_imaging**扩展，当卷积滤波器已经启动时，颜色的四个组分中的每个组分都可以经过卷积过滤后再进行缩放和偏移。即，红组分将乘上缩放比例因子**GL_POST_CONVOLUTION_RED_SCALE**，再加上偏移量**GL_POST_CONVOLUTION_RED_BIAS**；绿组分将乘上缩放因子**GL_POST_CONVOLUTION_GREEN_SCALE**，再加上偏移量**GL_POST_CONVOLUTION_GREEN_BIAS**；蓝组分将乘上缩放比例因子**GL_POST_CONVOLUTION_BLUE_SCALE**，再加上偏移量**GL_POST_CONVOLUTION_BLUE_BIAS**；alpha组分将乘上缩放比例因子**GL_POST_CONVOLUTION_ALPHA_SCALE**，再加上偏移量**GL_POST_CONVOLUTION_ALPHA_BIAS**。经过以上的变换后，所有颜色组分还将被截断到范围[0, 1]内。

Color index

每一颜色索引左边的二进制位将被**GL_INDEX_SHIFT**替换；任何超出定点索引所指定片断的二进制位数量的那些位都将被补0。如果**GL_INDEX_SHIFT**是负数，它将替换的是颜色索引的右边，0的填充分法与替换颜色索引左边时相同。接着再加上**GL_INDEX_OFFSET**。值**GL_INDEX_SHIFT**和**GL_INDEX_OFFSET**均由函数**glPixelTransfer()**指定。

这里，最后生成的像素所要求的格式决定了具体操作的不同。如果最后生成的像素被写入一个颜色索引缓冲区，或按**GL_COLOR_INDEX**格式读回客户端的内存中，则像素仍按索引格式对待。如果**GL_MAP_COLOR**是TRUE，则每个像素将被 $2^n - 1$ 所屏蔽，这里n是**GL_PIXEL_MAP_I_TO_I_SIZE**，然后它将被屏蔽后的值所指向的**GL_PIXEL_MAP_I_TO_I**映射表中的值所替换。值**GL_MAP_**

COLOR由函数glPixelTransfer()指定。索引映射的内容由函数glPixelMap()指定。

如果最后生成的像素被写入一个**RGBA**颜色缓冲区，或按除**GL_COLOR_INDEX**格式之外的其他格式读回客户端的内存中，则像素将参照下面四个映射表由索引模式转化为颜色模式：**GL_PIXEL_MAP_I_TO_R**、**GL_PIXEL_MAP_I_TO_G**、**GL_PIXEL_MAP_I_TO_B**和**GL_PIXEL_MAP_I_TO_A**。在查表前，每一个像素还将被 $2^n - 1$ 屏蔽，这里n对红组分映射而言是**GL_PIXEL_MAP_I_TO_R_SIZE**；对绿组分映射而言是**GL_PIXEL_MAP_I_TO_G_SIZE**；对蓝组分映射而言是**GL_PIXEL_MAP_I_TO_B_SIZE**；对alpha组分映射而言是**GL_PIXEL_MAP_I_TO_A_SIZE**。经过以上的变换后，所有颜色组分还将被截断到范围[0, 1]内。这四个映射表的内容由函数glPixelMap()指定。

Depth	每一个深度值将乘以 GL_DEPTH_SCALE ，再加上 GL_DEPTH_BIAS 。然或被截断到范围[0, 1]内。
Stencil	与颜色索引类似，每一个模板索引的相应位将由 GL_INDEX_SHIFT 替换，然后加上 GL_INDEX_OFFSET 。如果 GL_MAP_STENCIL 是 TRUE ，则每一个像素将被 $2^n - 1$ 屏蔽，这里n是 GL_PIXEL_MAP_S_TO_S_SIZE 。那么，它将被屏蔽后的值所指定 GL_PIXEL_MAP_S_TO_S 映射表中的值所替换。

表5-25给出了函数glPixelTransfer()设定的像素转换参数值的类型、初值和有效值的范围。

表 5-25

pname	类 型	初 值	有 效 范 围
GL_MAP_COLOR	boolean	false	true/false
GL_MAP_STENCIL	boolean	false	true/false
GL_INDEX_SHIFT	integer	0	(-∞, ∞)
GL_INDEX_OFFSET	integer	0	(-∞, ∞)
GL_RED_SCALE	float	1	(-∞, ∞)
GL_GREEN_SCALE	float	1	(-∞, ∞)
GL_BLUE_SCALE	float	1	(-∞, ∞)
GL_ALPHA_SCALE	float	1	(-∞, ∞)
GL_DEPTH_SCALE	float	1	(-∞, ∞)
GL_RED_BIAS	float	0	(-∞, ∞)
GL_GREEN_BIAS	float	0	(-∞, ∞)
GL_BLUE_BIAS	float	0	(-∞, ∞)
GL_ALPHA_BIAS	float	0	(-∞, ∞)
GL_DEPTH_BIAS	float	0	(-∞, ∞)
GL_POST_COLOR_MATRIX_RED_SCALE	float	1	(-∞, ∞)

(续)

<i>pname</i>	类 型	初 值	有 效 范 围
GL_POST_COLOR_MATRIX_GREEN_SCALE	float	1	(-∞, ∞)
GL_POST_COLOR_MATRIX_BLUE_SCALE	float	1	(-∞, ∞)
GL_POST_COLOR_MATRIX_ALPHA_SCALE	float	1	(-∞, ∞)
GL_POST_COLOR_MATRIX_RED_BIAS	float	0	(-∞, ∞)
GL_POST_COLOR_MATRIX_GREEN_BIAS	float	0	(-∞, ∞)
GL_POST_COLOR_MATRIX_BLUE_BIAS	float	0	(-∞, ∞)
GL_POST_COLOR_MATRIX_ALPHA_BIAS	float	0	(-∞, ∞)
GL_POST_CONVOLUTION_RED_SCALE	float	1	(-∞, ∞)
GL_POST_CONVOLUTION_GREEN_SCALE	float	1	(-∞, ∞)
GL_POST_CONVOLUTION_BLUE_SCALE	float	1	(-∞, ∞)
GL_POST_CONVOLUTION_ALPHA_SCALE	float	1	(-∞, ∞)
GL_POST_CONVOLUTION_RED_BIAS	float	0	(-∞, ∞)
GL_POST_CONVOLUTION_GREEN_BIAS	float	0	(-∞, ∞)
GL_POST_CONVOLUTION_BLUE_BIAS	float	0	(-∞, ∞)
GL_POST_CONVOLUTION_ALPHA_BIAS	float	0	(-∞, ∞)

函数glPixelTransferf()能设置任何类型的像素转换参数。如果参数类型是boolean，则0代表**FALSE**，其他非0值都代表**TRUE**。如果参数*pname*是一个整型参数，则参数*param*将被圆整为一个最接近的整数。

同样，函数glPixelTransferi()也能设置任何类型的像素转换参数。如果参数类型是boolean，则0代表**FALSE**，其他非0值表示**TRUE**。参数*param*被赋值给实型参数值之前要先转化为浮点型。

- 注意：

当函数glColorTable()、glColorSubTable()、glConvolutionFilter1D()、glConvolutionFilter2D()、glCopyPixels()、glCopyTexImage1D()、glCopyTexImage2D()、glCopyTexSubImage1D()、glCopyTexSubImage2D()、glCopyTexSubImage3D()、glDrawPixels()、glReadPixels()、glSeparableFilter2D()、glTexImage1D()、glTexImage2D()、glTexImage3D()、glTexSubImage1D()、glTexSubImage2D()或glTexSubImage3D()被编入一个显示列表中（请参阅glNewList()和glCallList()）时，显示列表执行时所起作用的像素转换模式是正在被使用的那个像素转换模式。因此，它跟命令被编译进显示列表时的设置可以是不同的。

- 出错提示：

当参数*pname*不是一个可接受的值时产生**GL_INVALID_ENUM**提示。

当函数glPixelTransfer()在函数对glBegin()/glEnd()之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

```
glGet( GL_MAP_COLOR )
glGet( GL_MAP_STENCIL )
glGet( GL_INDEX_SHIFT )
```

```

glGet( GL_INDEX_OFFSET )
glGet( GL_RED_SCALE )
glGet( GL_RED_BIAS )
glGet( GL_GREEN_SCALE )
glGet( GL_GREEN_BIAS )
glGet( GL_BLUE_SCALE )
glGet( GL_BLUE_BIAS )
glGet( GL_ALPHA_SCALE )
glGet( GL_ALPHA_BIAS )
glGet( GL_DEPTH_SCALE )
glGet( GL_DEPTH_BIAS )
glGet( GL_POST_COLOR_MATRIX_RED_SCALE )
glGet( GL_POST_COLOR_MATRIX_RED_BIAS )
glGet( GL_POST_COLOR_MATRIX_GREEN_SCALE )
glGet( GL_POST_COLOR_MATRIX_GREEN_BIAS )
glGet( GL_POST_COLOR_MATRIX_BLUE_SCALE )
glGet( GL_POST_COLOR_MATRIX_BLUE_BIAS )
glGet( GL_POST_COLOR_MATRIX_ALPHA_SCALE )
glGet( GL_POST_COLOR_MATRIX_ALPHA_BIAS )
glGet( GL_POST_CONVOLUTION_RED_SCALE )
glGet( GL_POST_CONVOLUTION_RED_BIAS )
glGet( GL_POST_CONVOLUTION_GREEN_SCALE )
glGet( GL_POST_CONVOLUTION_GREEN_BIAS )
glGet( GL_POST_CONVOLUTION_BLUE_SCALE )
glGet( GL_POST_CONVOLUTION_BLUE_BIAS )
glGet( GL_POST_CONVOLUTION_ALPHA_SCALE )
glGet( GL_POST_CONVOLUTION_ALPHA_BIAS )

```

• 请参阅：

glCallList(), **glColorTable()**, **glColorSubTable()**, **glConvolutionFilter1D()**, **glConvolutionFilter2D()**, **glCopyPixels()**, **glCopyTexImage1D()**, **glCopyTexImage2D()**, **glCopyTexSubImage1D()**, **glCopyTexSubImage2D()**, **glCopyTexSubImage3D()**, **glDrawPixels()**, **glNewList()**, **glPixelMap()**, **glPixelStore()**, **glPixelZoom()**, **glReadPixels()**, **glTexImage1D()**, **glTexImage2D()**, **glTexImage3D()**, **glTexSubImage1D()**, **glTexSubImage2D()**, **glTexSubImage3D()**

* **glPixelZoom**

• 名称：

glPixelZoom()

• 功能：

指定像素的缩放比例因子。

- C描述：

```
void glPixelZoom( GLfloat xfactor,
                  GLfloat yfactor )
```

- 参数说明：

xfactor, *yfactor*

指定在对像素进行写操作时的x和y缩放比例因子。

- 说明：

函数glPixelZoom()用来指定x和y缩放比例因子。在函数glDrawPixels()和glCopyPixels()的执行过程中，如果(*xr*, *yr*)是当前的光栅位置，并且给定元素位于像素矩形中第*m*行第*n*列，则中心位于矩形中的像素将可能被替换。这个矩形的四个角的坐标为：

$$(\textit{xr} + n \cdot \textit{xfactor}, \textit{yr} + m \cdot \textit{yfactor})$$

$$(\textit{xr} + (n+1) \cdot \textit{xfactor}, \textit{yr} + (m+1) \cdot \textit{yfactor})$$

那些中心位于此矩形区的底部或左边界像素也将被修正。

像素缩放因子也可以是负数。负的像素缩放因子将把所生成的图像以当前光栅位置为基准进行反射。

- 出错提示：

当函数glPixelZoom()在函数对glBegin()/glEnd()之间执行时产生GL_INVALID_OPERATION提示。

- 有关数据的获取：

glGet(GL_ZOOM_X)

glGet(GL_ZOOM_Y)

- 请参阅：

glCopyPixels(), glDrawPixels()

• glPointSize

- 名称：

glPointSize()

- 功能：

指定光栅化的点的直径。

- C描述：

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

- 参数说明：

size 指定光栅化的点的直径。它的初始值是1。

- 说明：

函数glPointSize()用来指定走样和反走样点的光栅化直径。是否启动反走样操作将对大小不

为1的点产生不同的显示效果。用函数glEnable(GL_POINT_SMOOTH)和glDisable(GL_POINT_SMOOTH)可以启动和关闭点的反走样操作。点的反走样操作在初始情况下是关闭的。

如果没有启动反走样操作，GL将把它所支持的尺寸圆整成最接近的整数值，从而确定点的实际大小。（如果圆整后的值是0，则相当于点的大小是1。）如果圆整后的尺寸是奇数，那么，像素片段的中点(x, y)将由下式计算：

$$(x_w + .5, y_w + .5)$$

这里的下标w表示它是窗口坐标。这时该片断是由所有位于一个正方形块中的像素表示的，这个正方形块的中点位于(x, y)。如果圆整后的尺寸是偶数，那么，像素片段的中点(x, y)将由下式计算：

$$(x_w + .5, y_w + .5)$$

这时光栅化的片断的中点是个半整数的窗口坐标。它位于中心在(x, y)的正方形块中。所有由光栅化一个非反走样点而产生的像素片断都被赋以相同的相关数据，也就是与该点相应的顶点。

如果已启动了反走样操作，那么，通过光栅化一个点而为每个像素正方形生成的一个片断将被显示为一个圆形像素区域，该圆的直径等于当前点的尺寸，中心位于点的(x_w, y_w)位置。每一像素的覆盖值是圆形区域和相应的像素正方形相交的窗口坐标的面积。这一值将被储存起来并用于最后的光栅化阶段。每个片断的相关值也就是被光栅化的点的相关值。

点的反走样操作并不支持所有的尺寸值。当遇到一个不支持的尺寸值时，它将被最接近的支持值所代替。只有值1被所有系统支持，其他的值则由具体的实现决定。函数glGet(GL_SMOOTH_POINT_SIZE_RANGE)用来查询系统所支持的尺寸范围。函数glGet(GL_SMOOTH_POINT_SIZE_GRANULARITY)用于查询在支持范围内所支持的值之间的区别。类似地，走样点用函数glGet(GL_ALIASED_POINT_SIZE_RANGE)和glGet(GL_ALIASED_POINT_SIZE_GRANULARITY)来查询。

- 注意：

当查询GL_POINT_SIZE时，总是返回由函数glPointSize()所指定的点的尺寸。对走样点和反走样点进行的截断和圆整操作将不影响这些指定值。

一个非反走样点的尺寸可能被截断为一个由具体实现所决定的最大值。尽管这些最大值不能被查询，但它圆整后的整型值必须不小于反走样点的最大值。

GL_POINT_SIZE_RANGE和GL_POINT_SIZE_GRANULARITY在GL1.2以上版本中不能使用，其相应功能由GL_SMOOTH_POINT_SIZE_RANGE和GL_SMOOTH_POINT_SIZE_GRANULARITY所替代。

- 出错提示：

当参数size小于或等于0时产生GL_INVALID_VALUE提示。

当函数glPointSize()在函数对glBegin()/glEnd()之间执行时产生GL_INVALID_OPERATION提示。

- 有关数据的获取：

```
glGet( GL_POINT_SIZE )
glGet( GL_ALIASED_POINT_SIZE_RANGE )
glGet( GL_ALIASED_POINT_SIZE_GRANULARITY )
glGet( GL_SMOOTH_POINT_SIZE_RANGE )
glGet( GL_SMOOTH_POINT_SIZE_GRANULARITY )
glGet( GL_POINT_SMOOTH )
```

- 请参阅：

glEnable()

glPolygonMode

- 名称：

glPolygonMode()

- 功能：

指定一个多边形的光栅化模式。

- C 描述：

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

- 参数说明：

face 指定应用模式*mode*的多边形。它可以是**GL_FRONT**、**GL_BACK**和**GL_FRONT_AND_BACK**。其中**GL_FRONT**指定正面多边形，**GL_BACK**指定反面多边形，**GL_FRONT_AND_BACK**指定正面和反面多边形。

mode 指定多边形的光栅化方式。它可以是**GL_POINT**、**GL_LINE**和**GL_FILL**。初始情况下，正面和反面多边形都按**GL_FILL**方式绘制。

- 说明：

函数**glPolygonMode()**用于控制多边形的光栅化方式。参数*face*指定应用模式*mode*的多边形是正面多边形（**GL_FRONT**）、反面多边形（**GL_BACK**）还是正面和反面多边形（**GL_FRONT_AND_BACK**）。多边形的绘制模式仅对最后多边形的光栅化产生影响。尤其是，在应用这些模式之前，多边形的顶点可以被照亮、多边形可以被剪切、甚至有可能被拣选掉。

参数*mode*所代表的模式有以下三种：

GL_POINT 多边形的顶点被标记为一个边界边的开始，这些顶点将按点的方式绘制。点的属性参数诸如**GL_POINT_SIZE**和**GL_POINT_SMOOTH**等将控制这些点的光栅化。除**GL_POLYGON_MODE**之外的其他多边形光栅化属性将无效。

GL_LINE 多边形边界边的绘制将按线段的方式绘制。多边形将被看作是由线的点画方式绘制的一些连续的线段。在线段之间，线的点画计数器和模板将不被重设（请参阅**glLineStipple()**）。线段属性参数诸如**GL_LINE_WIDTH**和

GL_LINE_SMOOTH等将控制这些线段的光栅化。除**GL_POLYGON_MODE**之外的其他多边形光栅化属性将无效。

GL_FILL 多边形的内部将被填充。多边形的属性参数诸如**GL_POLYGON_STIPPLE**和**GL_POLYGON_SMOOTH**等将控制多边形的光栅化。

`glPolygonMode(GL_FRONT, GL_LINE);`

- 示例：

绘制一个反面是填充多边形、正面是轮廓图多边形的一个表面，可调用函数：

- 注意：

可用一个边界标志来把顶点标记为边界边或非边界边。边界标志是在分解一个多边形时由GL系统内部生成的。用户可以用命令**glEdgeFlag()**来明确地设置这些边界标志。

- 出错提示：

当参数`face`或`mode`不是一个可接受的值时产生**GL_INVALID_ENUM**提示。

当函数**glPolygonMode()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

`glGet(GL_POLYGON_MODE)`

- 请参阅：

`glBegin()`, `glEdgeFlag()`, `glLineStipple()`, `glLineWidth()`, `glPointSize()`, `glPolygonStipple()`

glPolygonOffset

- 名称：

`glPolygonOffset()`

- 功能：

设置用于计算深度值的缩放比例因子和单元。

- C 描述：

```
void glPolygonOffset( GLfloat factor,
                      GLfloat units )
```

- 参数说明：

factor 指定一个缩放比例因子，它将用来建立每个多边形的可变的深度偏移量。其初始值是0。

units 它将乘上一个具体实现指定的值从而建立一个固定的深度偏移量。其初始值是0。

- 说明：

当启动**GL_POLYGON_OFFSET_FILL**、**GL_POLYGON_OFFSET_LINE**或**GL_POLYGON_OFFSET_POINT**时，每个片断的`depth`值将从一个适当顶点的`depth`值之中插入，然后进行偏移。这里的偏移量是`factor*DZ + r*units`，其中`DZ`是多边形所对应的屏幕区域在深度上的一种度量，`r`是给定具体实现中能够生成一个可分辨的最小偏移值。在进行深度测试和写入深度缓冲区之前，首先要加上该偏移量。

函数glPolygonOffset()对于绘制消隐图像、对物体表面进行贴面以及用高亮边界绘制实心体是很有用的。

- 注意：

函数glPolygonOffset()只有在GL 1.1以上版本中才可以使用。

函数glPolygonOffset()并不对存放在反馈缓冲区中的深度坐标产生影响。

函数glPolygonOffset()在选择模式下无效。

- 出错提示：

当函数glPolygonOffset()在函数对glBegin()/glEnd()之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

```
glIsEnabled( GL_POLYGON_OFFSET_FILL )
glIsEnabled( GL_POLYGON_OFFSET_LINE )
glIsEnabled( GL_POLYGON_OFFSET_POINT )
glGet( GL_POLYGON_OFFSET_FACTOR )
glGet( GL_POLYGON_OFFSET_UNITS )
```

- 请参阅：

glDepthFunc(), **glEnable()**, **glGet()**, **glIsEnabled()**

• **glPolygonStipple**

- 名称：

glPolygonStipple()

- 功能：

设置多边形的点画绘制方式。

- C描述：

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

- 参数说明：

mask 指定一个指针，指向一个 32×32 的点画图案的位置。该模板将从内存中取出。其取出的方法与函数glDrawPixels()相同。

- 说明：

多边形的点画绘图方式跟线的点画绘图方式相似（请参阅glLineStipple()）。它也是通过建立一个图案，从而在绘图时屏蔽掉一些通过光栅化而生成的片断。点画绘图模式与多边形的反走样无关。

参数*mask*是一个指针，它指向一个存储于内存中的 32×32 的点画图案的位置。其执行情况就像在像素格式为**GL_COLOR_INDEX**、数据类型为**GL_BITMAP**的情况下，把一个*height*和*width*都是32的像素数据提供给函数glDrawPixels()。也就是说，点画图案是用一个 32×32 的二进制位颜色索引数组来表示的，该数组中放置的是无符号字节。函数glPixelStore()的参数如**GL_UNPACK_SWAP_BYT**E和**GL_UNPACK_LSB_FIRST**将影响二进制位放入一个点画模式

中时的排列方式。然而，像素转换操作（替换、偏移、像素映射等）将不影响点画模式所形成的图像。

调用函数glEnable(GL_POLYGON_STIPPLE)和glDisable(GL_POLYGON_STIPPLE)可以启动和关闭多边形的点画绘图模式。初始情况下，点画绘图模式是关闭的。当启动这一模式后，一个光栅化后的窗口坐标为 x_w 和 y_w 的多边形片断将被送到GL的下一环节的充要条件是该点画模板的第 $(x_w \bmod 32)$ 行的第 $(y_w \bmod 32)$ 位是1(one)。当关闭多边形的点画绘图模式后，它相当于点画模板全由1构成。

- 出错提示：

当函数glPolygonMode()在函数对glBegin()/glEnd()之间执行时产生GL_INVALID_OPERATION提示

- 有关数据的获取：

glGetPolygonStipple()

glIsEnabled(GL_POLYGON_STIPPLE)

- 请参阅：

glDrawPixels(), glLineStipple(), glPixelStore(), glPixelTransfer()

• **glPrioritizeTextures**

- 名称：

glPrioritizeTextures()

- 功能：

设置纹理驻留的优先级。

- C描述：

```
void glPrioritizeTextures( GLsizei n,
                           const GLuint *textures,
                           const GLclampf *priorities )
```

- 参数说明：

n 指定将优先排列次序的纹理的数目。

textures 指定一个用来存放将被优先化处理的纹理名称的数组。

priorities 指定一个用来存放纹理优先级的数组。通过优先级排列，可以将具有一定优先级的*priorities*中的元素应用到*textures*中相应的元素所指定的纹理中去。

- 说明：

函数glPrioritizeTextures()将*priorities*中给定的*n*个纹理优先级赋给*textures*所指定的*n*个纹理。

GL所建立的所谓纹理的“工作组”是驻留在纹理存储器中的。当这些纹理与一个纹理目标相连接时可能比没有驻留的纹理要有效得多。通过指定每个纹理的优先级，函数glPrioritizeTextures()将可以指导GL具体实现环境确定哪个纹理将驻留。

参数*priorities*中给定的优先级在使用前是在范围[0, 1]内的。其中0表示最低的优先级。当

一个纹理是0优先级时，就表示该纹理最不可能驻留。而1表示最高的优先级，优先级为1的纹理将最有可能驻留。然而，纹理在被使用前是不能保证被驻留的。

函数**glPrioritizeTextures()**在默认情况下将忽略0优先级的纹理，或者说这时任何的纹理名称都不与一个实际存在的纹理相对应。

函数**glPrioritizeTextures()**并不要求任何由*textures*指定的纹理与一个纹理目标相连接。函数**glTexParameter()**也可以用来设置一个纹理的优先级，但它仅适用于当前连接的纹理。它是唯一用来设置默认纹理的方法。

- 注意：

函数**glPrioritizeTextures()**只有在GL 1.1以上的版本中才可以使用。

- 出错提示：

当参数*n*是负数时产生**GL_INVALID_VALUE**提示。

当函数**glPrioritizeTextures()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGetTexParameter(GL_TEXTURE_PRIORITY)将返回当前连接纹理的优先级。

- 请参阅：

glAreTexturesResident(), **glBindTexture()**, **glCopyTexImage1D()**, **glCopyTexImage2D()**,
glTexImage1D(), **glTexImage2D()**, **glTexImage3D()**, **glTexParameter()**

◆ **glPushAttrib**, **glPopAttrib**

- 名称：

glPushAttrib(), **glPopAttrib()**

- 功能：

压入和弹出服务器属性堆栈。

- C描述：

`void glPushAttrib(GLbitfield mask)`

- 参数说明：

mask 为那些将要存储的属性指定一个屏蔽码。参数*mask*的值见下面叙述。

- C描述：

`void glPopAttrib(void)`

- 说明：

函数**glPushAttrib()**带有一个参数*mask*，它是一个屏蔽码，用以确定将要存入属性堆栈的状态变量组。该屏蔽码将使用符号常量来设置其中的位。参数*mask*的典型结构是采用逻辑或运算将这些常量中的某几个连在一起。特殊屏蔽码**GL_ALL_ATTRIB_BITS**用于存储所有可进栈的状态。

符号型屏蔽常量及它们的相关GL状态如下（其中的第二列为可被存储的属性）：

GL_ACCUM_BUFFER_BIT	累积缓存区刷新值
----------------------------	----------

GL_COLOR_BUFFER_BIT	GL_ALPHA_TEST 的启动标志位 Alpha测试函数及参考值 GL_BLEND 的启动标志位 融合的源函数和目标函数 融合的颜色常数 融合方程 GL_DITHER 的启动标志位 GL_DRAW_BUFFER 设置值 GL_COLOR_LOGIC_OP 的启动标志位 GL_INDEX_LOGIC_OP 的启动标志位 逻辑操作函数 颜色模式和索引模式的刷新值 颜色模式和索引模式的写入标志
GL_CURRENT_BIT	当前RGBA颜色 当前颜色索引 当前法向量 当前纹理坐标 当前光栅位置 GL_CURRENT_RASTER_POSITION_VALID 标记 与当前光栅位置相关的RGBA颜色 与当前光栅位置相关的颜色索引 与当前光栅位置相关的纹理坐标 GL_EDGE_FLAG 标记
GL_DEPTH_BUFFER_BIT	GL_DEPTH_TEST 的启动标志位 深度缓冲区测试函数 深度缓冲区的刷新值 GL_DEPTH_WRITEMASK 的启动标志位
GL_ENABLE_BIT	GL_ALPHA_TEST 的开启标志位 GL_AUTO_NORMAL 标志 GL_BLEND 标志 用户所定义的剪切平面的启动标志位 GL_COLOR_MATERIAL GL_CULL_FACE 标志 GL_DEPTH_TEST 标志 GL_DITHER 标志 GL_FOG 标志 GL_LIGHT<i>i</i> , 这里 $0 \leq i < \text{GL_MAX_LIGHTS}$

	GL_LIGHTING 标志
	GL_LINE_SMOOTH 标志
	GL_LINE_STIPPLE 标志
	GL_COLOR_LOGIC_OP 标志
	GL_INDEX_LOGIC_OP 标志
	GL_MAP1_x , 这里x是一种映射类型
	GL_MAP2_x , 这里x是一种映射类型
	GL_NORMALIZE 标志
	GL_POINT_SMOOTH 标志
	GL_POLYGON_OFFSET_LINE 标志
	GL_POLYGON_OFFSET_FILL 标志
	GL_POLYGON_OFFSET_POINT 标志
	GL_POLYGON_SMOOTH 标志
	GL_POLYGON_STIPPLE 标志
	GL_SCISSOR_TEST 标志
	GL_STENCIL_TEST 标志
	GL_TEXTURE_1D 标志
	GL_TEXTURE_2D 标志
	GL_TEXTURE_3D 标志
	GL_TEXTURE_GEN_x 标志, 这里x是S、T、R或Q
GL_EVAL_BIT	GL_MAP1_x 的启动标志位, 这里x是一种映射类型
	GL_MAP2_x 的开启标志位, 这里x是一种映射类型
	1D网格的端点和分界线
	2D网格的端点和分界线
	GL_AUTO_NORMAL 的启动标志位
GL_FOG_BIT	GL_FOG 的启动标志位
	雾的颜色
	雾的浓度
	线性雾化的开始点
	线性雾化的终止点
	雾化索引
	GL_FOG_MODE 值
GL_HINT_BIT	GL_PERSPECTIVE_CORRECTION_HINT 设置值
	GL_POINT_SMOOTH_HINT 设置值
	GL_LINE_SMOOTH_HINT 设置值
	GL_FOG_HINT 设置值

GL_LIGHTING_BIT	GL_COLOR_MATERIAL 的启动标志位 GL_COLOR_MATERIAL_FACE 值 跟踪当前颜色的颜色材质参数 环境光颜色 GL_LIGHT_MODEL_LOCAL_VIEWER 值 GL_LIGHT_MODEL_TWO_SIDE 设置值 GL_LIGHTING 的启动标志位 各光源的启动标志位 各光源的环境光、散射光和反射光分量的强度 各光源的方向、位置、聚光指数和截止角 各光源的常数、线性和二次衰减因子 各材质的环境、散射、反射和发散颜色 各材质的环境、散射、反射颜色索引 各材质的反射指数 GL_SHADE_MODEL 设置值
GL_LINE_BIT	GL_LINE_SMOOTH 标志 GL_LINE_STIPPLE 的启动标志位 线段的点画模式及重复计数器 线宽
GL_LIST_BIT	GL_LIST_BASE 设置值
GL_PIXEL_MODE_BIT	GL_RED_BIAS 和 GL_RED_SCALE 设置值 GL_GREEN_BIAS 和 GL_GREEN_SCALE 设置值 GL_BLUE_BIAS 和 GL_BLUE_SCALE 设置值 GL_ALPHA_BIAS 和 GL_ALPHA_SCALE 设置值 GL_DEPTH_BIAS 和 GL_DEPTH_SCALE 设置值 GL_INDEX_OFFSET 和 GL_INDEX_SHIFT 设置值 GL_MAP_COLOR 和 GL_MAP_STENCIL 标志位 GL_ZOOM_X 和 GL_ZOOM_Y 缩放因子 GL_READ_BUFFER 设置值
GL_POINT_BIT	GL_POINT_SMOOTH 标志位 点的尺寸值
GL_POLYGON_BIT	GL_CULL_FACE 的启动标志位 GL_CULL_FACE_MODE 设置值 GL_FRONT_FACE 指示器 GL_POLYGON_MODE 设置值 GL_POLYGON_SMOOTH 标志位 GL_POLYGON_STIPPLE 的启动标志位

GL_POLYGON_OFFSET_FILL	标志位
GL_POLYGON_OFFSET_LINE	标志位
GL_POLYGON_OFFSET_POINT	标志
GL_POLYGON_OFFSET_FACTOR	
GL_POLYGON_OFFSET_UNITS	
 GL_POLYGON_STIPPLE_BIT	
	点画方式绘制的多边形图像
 GL_SCISSOR_BIT	
	GL_SCISSOR_TEST 标志
	剪切板
 GL_STENCIL_BUFFER_BIT	
	GL_STENCIL_TEST 的启动标志位
	模板函数及参考值
	模板值的屏蔽
	模板测试失败、通过及深度缓冲区通过的动作
	模板缓冲区的刷新值
	模板缓冲区的写屏蔽值
 GL_TEXTURE_BIT	
	四个纹理坐标的启动标志位
	各纹理图像的边界颜色
	各纹理图像的缩小函数
	各纹理图像的放大函数
	各纹理图像的纹理坐标和缠绕模式
	各纹理环境的颜色和模式
	GL_TEXTURE_GEN_x 的启动标志位，这里x是S、T、R和Q
	GL_TEXTURE_GEN_MODE 对S、T、R和Q的设置值
	glTexGen() 对S、T、R和Q的平面方程
	当前的纹理连接（如 GL_TEXTURE_BINDING_2D ）
 GL_TRANSFORM_BIT	
	六个剪切平面的系数
	用户定义的剪切平面的启动标志位
	GL_MATRIX_MODE 值
	GL_NORMALIZE 标志
	GL_RESCALE_NORMAL 标志
 GL_VIEWPORT_BIT	
	深度范围（最近及最远值）
	视口原点及范围

函数**glPopAttrib()**用来恢复最后一个**glPushAttrib()**命令所存储的状态变量。那些没被存储的变量将不发生变化。

把属性值压入一个满堆栈或从一个空堆栈中弹出属性值都是错误的。在这两种情况下，都将设置一个出错标记，并且不改变GL的状态。初始情况下，属性堆栈是空的。

- 注意：

并非所有GL状态值都能被存入属性堆栈。比如，绘制模式状态及选择和反馈状态就不能被存储。客户端的状态必须用函数`glPushClientAttrib()`存储。

属性堆栈的深度值与所使用的具体实现有关，但它至少应该是16。

当系统支持**GL_ARB_multitexture**扩展时，压入和弹出纹理状态的操作对所有支持的纹理单元都有效。

- 出错提示：

当属性堆栈存满以后继续调用函数`glPushAttrib()`时将产生**GL_STACK_OVERFLOW**提示。

当属性堆栈是空堆栈时调用函数`glPopAttrib()`时产生**GL_STACK_UNDERFLOW**提示。

当函数`glPushAttrib()`或`glPopAttrib()`在函数对`glBegin()`/`glEnd()`之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

`glGet(GL_ATTRIB_STACK_DEPTH)`

`glGet(GL_MAX_ATTRIB_STACK_DEPTH)`

- 请参阅：

`glGet()`, `glGetClipPlane()`, `glGetError()`, `glGetLight()`, `glGetMap()`, `glGetMaterial()`,
`glGetPixelMap()`, `glGetPolygonStipple()`, `glGetString()`, `glGetTexEnv()`, `glGetTexGen()`,
`glGetTexImage()`, `glGetTexLevelParameter()`, `glGetTexParameter()`, `glIsEnabled()`,
`glPushClientAttrib()`

- **glPushClientAttrib**, **glPopClientAttrib**

- 名称：

`glPushClientAttrib()`, `glPopClientAttrib()`

- 功能：

压入和弹出客户属性堆栈。

- C描述：

`void glPushClientAttrib(GLbitfield mask)`

- 参数说明：

mask 为那些将要存储的属性指定一个屏蔽码。参数*mask*的值见下面叙述。

- C描述：

`void glPopClientAttrib(void)`

- 说明：

函数`glPushClientAttrib()`带有一个参数*mask*，它是一个屏蔽码，用来确定将要存入客户属性堆栈的客户状态变量组。符号常量用来设置屏蔽码中的二进制位。参数*mask*的典型结构是采用逻辑或运算将这些常量中的某几个连在一起。特殊的屏蔽码**GL_CLIENT_ALL_ATTRIB_BITS**用于存储所有可进栈的客户端状态。

符号型屏蔽常量及它们的相关GL状态如下（其中的第二列为可被存储的属性）：

GL_CLIENT_PIXEL_STORE_BIT	像素存储模式
GL_CLIENT_VERTEX_ARRAY_BIT	顶点数组（及启动值）

函数`glPopClientAttrib()`用以恢复最后一个由`glPushAttrib()`命令存储的客户端状态变量。那些没被存储的变量将不发生变化。

把属性值压入一个满的客户属性堆栈或从一个空的堆栈中弹出属性值都是错误的。在这两种情况下，GL将设置一个出错标记，并且不改变GL的状态。

初始情况下，客户端属性堆栈是空的。

- 注意：

函数`glPushClientAttrib()`只有在GL 1.1以上的版本中才可以使用。

并非所有GL客户状态值都能被存入属性堆栈。比如，选择和反馈状态就不能被存储。

属性堆栈的深度值与GL的具体实现有关，但它至少应该是16。

函数`glPushAttrib()`和`glPopAttrib()`用来压入和恢复服务器的状态。函数`glPushClientAttrib()`和`glPopClientAttrib()`仅用来压入和弹出像素存储模式及顶点数组的状态。

当系统支持**GL_ARB_multitexture**扩展时，压入和弹出客户端顶点数组状态的操作对所有支持的纹理单元和活动的客户端纹理状态都是有效的。

- 出错提示：

当属性堆栈存满以后继续调用函数`glPushClientAttrib()`时将产生**GL_STACK_OVERFLOW**提示。

当属性堆栈是空堆栈时调用函数`glPopClientAttrib()`将产生**GL_STACK_UNDERFLOW**提示。

- 有关数据的获取：

`glGet(GL_ATTRIB_STACK_DEPTH)`

`glGet(GL_MAX_CLIENT_ATTRIB_STACK_DEPTH)`

- 请参阅：

`glColorPointer()`, `glDisableClientState()`, `glEdgeFlagPointer()`, `glEnableClientState()`,
`glGet()`, `glGetError()`, `glIndexPointer()`, `glNormalPointer()`, `glNewList()`, `glPixelStore()`,
`glPushAttrib()`, `glTexCoordPointer()`, `glVertexPointer()`

- **glPushMatrix, glPopMatrix**

- 名称：

`glPushMatrix()`, `glPopMatrix()`

- 功能：

压入和弹出当前矩阵堆栈。

- C描述：

`void glPushMatrix(void)`

- C描述：

`void glPopMatrix(void)`

- 说明：

每个矩阵模式都含有一个矩阵堆栈。在**GL_MODELVIEW**模式下，堆栈深度至少是32。在其他模式下，如**GL_COLOR**、常值（PROJECTION）及**GL_TEXTURE**，深度至少是2。在任何模式下，当前矩阵都是指该模式下矩阵堆栈栈顶的矩阵。

函数**glPushMatrix()**将把当前的矩阵栈压下一层，把当前矩阵复制后放入当前层。也就是说，调用函数**glPushMatrix()**后，栈顶的矩阵和下一层的矩阵相同。

函数**glPopMatrix()**的作用是从当前矩阵堆栈中弹出一个矩阵，把当前矩阵替换为其下一层的矩阵。

初始情况下，每一个堆栈包含一个矩阵，一个相同的矩阵。

对一个满矩阵堆栈进行压入操作或对一个只包含单一矩阵的堆栈进行弹出操作都是错误的。在这两种情况下，将设置一个出错标记，并且不改变GL的状态。

- 出错提示：

当当前矩阵堆栈存满以后继续调用函数**glPushMatrix()**时产生**GL_STACK_OVERFLOW**提示。

当当前矩阵堆栈中只含有一个矩阵时继续调用函数**glPopAttrib()**时产生**GL_STACK_UNDERFLOW**提示。

当函数**glPushMatrix()**或**glPopMatrix()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

```
glGet( GL_MATRIX_MODE )
glGet( GL_COLOR_MATRIX )
glGet( GL_MODELVIEW_MATRIX )
glGet( GL_PROJECTION_MATRIX )
glGet( GL_TEXTURE_MATRIX )
glGet( GL_COLOR_STACK_DEPTH )
glGet( GL_MODELVIEW_STACK_DEPTH )
glGet( GL_PROJECTION_STACK_DEPTH )
glGet( GL_TEXTURE_STACK_DEPTH )
glGet( GL_MAX_MODELVIEW_STACK_DEPTH )
glGet( GL_MAX_PROJECTION_STACK_DEPTH )
glGet( GL_MAX_TEXTURE_STACK_DEPTH )
```

- 请参阅：

glFrustum(), **glLoadIdentity()**, **glLoadMatrix()**, **glMatrixMode()**, **glMultMatrix()**,
glOrtho(), **glRotate()**, **glScale()**, **glTranslate()**, **glViewport()**

- **glPushName**, **glPopname**

- 名称：

glPushName(), glPopname()

- 功能：

压入和弹出名称堆栈。

- C 描述：

```
void glPushName( GLuint name )
```

- 参数说明：

name 指定一个将被压入名称堆栈顶部的名称。

- C 描述：

```
void glPopname( void )
```

- 说明：

名称堆栈用来在选择模式中设置绘图命令的唯一标志。它包含一个有序的无符号整型设置值，并且在初始情况下它是空的。

函数 **glPushName()** 将 *name* 压入名称堆栈的顶部，函数 **glPopName()** 将名称堆栈顶部的一个名称弹出堆栈。

名称堆栈的最大深度与 GL 的具体实现有关。调用 **GL_MAX_NAME_STACK_DEPTH** 来查询一个特定实现的名称堆栈的深度值。把一个名称压入一个满堆栈或从一个空堆栈中弹出名称都是错误的。在函数对 **glBegin()**/**glEnd()** 之间使用名称堆栈也是错误的。在这些情况下，将设置一个出错标记，并不改变 GL 的状态。

当绘图模式不是 **GL_SELECT** 模式时，名称堆栈总是空的。如果绘图模式不是 **GL_SELECT**，调用函数 **glPushName()** 或 **glPopname()** 的操作也将被忽略。

- 出错提示：

当名称堆栈存满以后继续调用函数 **glPushName()** 时产生 **GL_STACK_OVERFLOW** 提示。

当名称堆栈是空堆栈时调用函数 **glPopname()** 产生 **GL_STACK_UNDERFLOW** 提示。

当函数 **glPushName()** 或 **glPopname()** 在函数对 **glBegin()**/**glEnd()** 之间执行时产生 **GL_INVALID_OPERATION** 提示。

- 有关数据的获取：

```
glGet( GL_NAME_STACK_DEPTH )
glGet( GL_MAX_NAME_STACK_DEPTH )
```

- 请参阅：

glInitNames(), **glLoadName()**, **glRenderMode()**, **glSelectBuffer()**

glRasterPos

- 名称：

glRasterPos2d(), **glRasterPos2f()**, **glRasterPos2i()**, **glRasterPos2s()**, **glRasterPos3d()**,
glRasterPos3f(), **glRasterPos3i()**, **glRasterPos3s()**, **glRasterPos4d()**, **glRasterPos4f()**,
glRasterPos4i(), **glRasterPos4s()**, **glRasterPos2dv()**, **glRasterPos2fv()**, **glRasterPos2iv()**,
glRasterPos2sv(), **glRasterPos3dv()**, **glRasterPos3fv()**, **glRasterPos3iv()**, **glRasterPos3sv()**,

glRasterPos4dv(), glRasterPos4fv(), glRasterPos4iv(), glRasterPos4sv()

- 功能：

指定像素操作的光栅位置。

- C描述：

```
void glRasterPos2d( GLdouble x,  
                      GLdouble y )  
void glRasterPos2f( GLfloat x,  
                      GLfloat y )  
void glRasterPos2i( GLint x,  
                     GLint y )  
void glRasterPos2s( GLshort x,  
                     GLshort y )  
void glRasterPos3d( GLdouble x,  
                      GLdouble y,  
                      GLdouble z )  
void glRasterPos3f( GLfloat x,  
                      GLfloat y,  
                      GLfloat z )  
void glRasterPos3i( GLint x,  
                     GLint y,  
                     GLint z )  
void glRasterPos3s( GLshort x,  
                     GLshort y,  
                     GLshort z )  
void glRasterPos4d( GLdouble x,  
                      GLdouble y,  
                      GLdouble z,  
                      GLdouble w )  
void glRasterPos4f( GLfloat x,  
                      GLfloat y,  
                      GLfloat z,  
                      GLfloat w )  
void glRasterPos4i( GLint x,  
                     GLint y,  
                     GLint z,  
                     GLint w )  
void glRasterPos4s( GLshort x,
```

```
    GLshort y,
    GLshort z,
    GLshort w )
```

- 参数说明：

x, y, z, w

指定光栅位置的对象坐标*x, y, z*和*w*（如果已给出）。

- C描述：

```
void glRasterPos2dv(const GLdouble *v )
void glRasterPos2fv( const GLfloat *v )
void glRasterPos2iv( const GLint *v )
void glRasterPos2sv( const GLshort *v )
void glRasterPos3dv(const GLdouble *v )
void glRasterPos3fv( const GLfloat *v )
void glRasterPos3iv( const GLint *v )
void glRasterPos3sv( const GLshort *v )
void glRasterPos4dv(const GLdouble *v )
void glRasterPos4fv( const GLfloat *v )
void glRasterPos4iv( const GLint *v )
void glRasterPos4sv( const GLshort *v )
```

- 参数说明：

v 指定一个指向一元、二元、三元或四元数组的指针，从而分别指定相应的*x, y, z*和*w*坐标。

- 说明：

在GL中含有一个三维的窗口坐标。它们被称作光栅坐标，像素和位图的写操作将用这些光栅坐标进行定位。它们保持子像素的精度。请参阅glBitmap(), glDrawPixels()和glCopyPixels()。

当前的光栅位置包含三个窗口坐标 (*x, y, z*) 和一个剪切坐标值 (*w*)，它还包含一个眼坐标距离、一个有效位和相关的颜色数据及纹理坐标。这里*w*坐标是一个剪切坐标，这是因为*w*没被投射到窗口坐标中。函数glRasterPos4()明确指定了对象坐标*x, y, z*和*w*。函数glRasterPos3()明确指定了对象坐标*x, y*和*z*，并且隐含地将*w*设置为1。函数glRasterPos2()明确指定了对象坐标*x*和*y*，并且隐含地设置*z*和*w*为0和1。

由函数glRasterPos()提供的对象坐标可以看作是由函数glVertex()通过以下方式产生的：由当前模式取景和投影矩阵转换并且送到剪切环节。如果该顶点没被拣选掉，那么它将被投影并被缩放为窗口坐标，从而成为一个新的当前光栅位置并设置**GL_CURRENT_RASTER_POSITION_VALID**标记。如果该顶点被拣选掉，则有效位被刷新且当前的光栅位置及相关的颜色和纹理坐标是未定义的。

当前的光栅位置也包括了一些相关的颜色数据和纹理坐标。如果已启动光照模式，则通过光照计算（请参阅glLight(), glLightModel()和glShadeModel()）产生的颜色被设置为**GL_CURRENT_COLOR**。

_RASTER_COLOR (RGBA模式) 或**GL_CURRENT_RASTER_INDEX** (颜色索引模式)。如果已关闭光照模式，则当前光栅的颜色被当前颜色 (RGBA模式，状态变量**GL_CURRENT_COLOR**) 或颜色索引 (颜色索引模式，状态变量**GL_CURRENT_INDEX**) 更新。

类似地，**GL_CURRENT_RASTER_TEXTURE_COORDS**将由**GL_CURRENT_TEXTURE_COORDS**通过纹理矩阵和纹理生成函数 (请参阅glTexGen()) 而生成的值更新。最后，从眼坐标系统的原点至顶点的距离仅通过模式取景矩阵变换后替换**GL_CURRENT_RASTER_DISTANCE**。

初始情况下，当前的光栅位置是 (0, 0, 0, 1)，当前光栅距离是0，有效位被设置，相关的RGBA颜色是 (1, 1, 1, 1)，相关的颜色索引是1，相关的纹理坐标是 (0, 0, 0, 1)。RGBA模式下，**GL_CURRENT_RASTER_INDEX**总是1；颜色索引模式下，当前光标的RGBA颜色总保持它的初始值。

- 注意：

光栅位置可被函数glRasterPos()和glBitmap()修改。

当光栅位置坐标无效时，基于光栅位置的绘图命令将被忽略 (也就是说，它将对GL状态不产生影响)。

调用函数glDrawElements()或glDrawRangeElements()可能会导致当前的颜色或索引不确定。如果当前的颜色或索引是不确定时，这时执行函数glRasterPos()将导致当前光栅颜色或当前光栅索引也无法确定。

为了要设置一个视口外的有效光栅位置，首先应该设置一个有效的光栅位置，然后调用参数**bitmap**为NULL的函数glBitmap()。

当系统支持**GL_ARB_imaging**扩展时，每一个纹理单元都有独自的光栅纹理坐标。每一个纹理单元的当前光栅纹理坐标由函数glRasterPos()更新。

- 出错提示：

当函数glRasterPos()在函数对glBegin()/glEnd()之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

```
glGet( GL_CURRENT_RASTER_POSITION )
glGet( GL_CURRENT_RASTER_POSITION_VALID )
glGet( GL_CURRENT_RASTER_DISTANCE )
glGet( GL_CURRENT_RASTER_COLOR )
glGet( GL_CURRENT_RASTER_INDEX )
glGet( GL_CURRENT_RASTER_TEXTURE_COORDS )
```

- 请参阅：

glBitmap(), glCopyPixels(), glDrawArrays(), glDrawElements(), glDrawRangeElements(), glDrawPixels(), glTexCoord(), glTexGen(), glVertex()

- **glReadBuffer**

- 名称：

glReadBuffer()

- 功能：

为像素选择一个源颜色缓冲区。

- C描述：

```
void glReadBuffer( GLenum mode )
```

- 参数说明：

mode 指定一个颜色缓冲区。它的可取值是**GL_FRONT_LEFT**、**GL_FRONT_RIGHT**、**GL_BACK_LEFT**、**GL_BACK_RIGHT**、**GL_FRONT**、**GL_BACK**、**GL_LEFT**、**GL_RIGHT**和**GL_AUX*i***。此处*i*位于0和**GL_AUX_BUFFERS-1**之间。

- 说明：

函数**glReadBuffer()**为随后的一些命令函数指定一个源颜色缓冲区。这些命令函数是**glReadPixels()**、**glCopyTexImage1D()**、**glCopyTexImage2D()**、**glCopyTexSubImage1D()**、**glCopyTexSubImage2D()**、**glCopyTexSubImage3D()**和**glCopyPixels()**。参数*mode*可取十二个或更多的预先定义的值。（**GL_AUX0~GL_AUX3**总是被定义的。）在一个完整配置的系统中，**GL_FRONT**、**GL_LEFT**和**GL_FRONT_LEFT**都是指前左缓冲区，**GL_FRONT_RIGHT**和**GL_RIGHT**指前右缓冲区，**GL_BACK_LEFT**和**GL_BACK**指后左缓冲区。

在非立体的双缓冲区配置中只有前左和后左两个缓冲区，在立体的单缓冲区配置中有前左和前右两个缓冲区，在非立体的单缓冲区配置中只有一个前左缓冲区。为函数**glReadBuffer()**指定一个非立体缓冲区是错误的。

- 出错提示：

当参数*mode*不是十二个（或更多）可接受值之一时产生**GL_INVALID_ENUM**提示。

当参数*mode*指定了一个并不存在的缓冲区时产生**GL_INVALID_OPERATION**提示。

当函数**glReadBuffer()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGet(GL_READ_BUFFER)

- 请参阅：

glCopyPixels(), **glCopyTexImage1D()**, **glCopyTexImage2D()**, **glCopyTexSubImage1D()**,
glCopyTexSubImage2D(), **glCopyTexSubImage3D()**, **glDrawBuffer()**, **glReadPixels()**

: glReadPixels

- 名称：

glReadPixels()

- 功能：

从帧缓冲区中读出一个像素块。

- C描述：

```
void glReadPixels ( GLint x,
                    GLint y,
                    GLsizei width,
                    GLsizei height,
                    GLenum format,
                    GLenum type,
                    GLvoid *pixels)
```

• 参数说明：

x, *y*

指定从帧缓冲区中读出的第一个像素的窗口坐标。它位于一个像素矩形块的左下角。

width, *height*

指定像素矩形的尺寸。它们与单个像素相一致。

format

指定像素数据的格式。它可以是以下符号常量值：GL_COLOR_INDEX、GL_STENCIL_INDEX、GL_DEPTH_COMPONENT、GL_RED、GL_GREEN、GL_BLUE、GL_ALPHA、GL_RGB、GL_BGR、GL_RGBA、GL_BGRA、GL_LUMINANCE和GL_LUMINANCE_ALPHA。

type

指定像素数据的数据类型。它可以是以下符号常量之一：GL_UNSIGNED_BYTE、GL_BYTE、GL_BITMAP、GL_UNSIGNED_SHORT、GL_SHORT、GL_UNSIGNED_INT、GL_INT、GL_FLOAT、GL_UNSIGNED_BYTE_3_3_2、GL_UNSIGNED_BYTE_2_3_3_REV、GL_UNSIGNED_SHORT_5_6_5、GL_UNSIGNED_SHORT_5_6_5_REV、GL_UNSIGNED_SHORT_4_4_4_4、GL_UNSIGNED_SHORT_4_4_4_4_REV、GL_UNSIGNED_SHORT_5_5_5_1、GL_UNSIGNED_SHORT_1_5_5_5_REV、GL_UNSIGNED_INT_8_8_8_8、GL_UNSIGNED_INT_8_8_8_8_REV、GL_UNSIGNED_INT_10_10_10_2和GL_UNSIGNED_INT_2_10_10_10_REV。

pixels

返回像素数据。

• 说明：

函数glReadPixels()的作用是从帧缓冲区中左下角位于(*x*, *y*)的像素矩阵中读出像素数据，并把读出的值存入客户端内存的*pixels*起始位置中。这里有几个参数在像素数据写入客户端内存之前控制对像素数据的处理。这些参数由以下三个命令设置：glPixelStore(), glPixelTransfer()和glPixelMap()。本节主要介绍函数glReadpixels()的用法，有关这三个命令对参数的设定请参考相关命令的介绍。

当系统支持GL_ARB_imaging扩展时，像素数据也可以由其他的操作处理，这些操作包括查找颜色表、颜色矩阵转换、卷积、直方图操作、最小及最大像素值计算等。

函数glReadPixels()从左下角在(*x+i*, *y+j*)的像素矩形中返回像素值，此处 $0 \leq i < width$, $0 \leq j < height$ 。

$\leq j < height$ 。这一像素被称作第*j*行的第*i*个像素。像素返回的次序是从最低的一行到最高的一行，同一行中则由左到右。

参数*format*指定返回的像素值的格式，它可取以下的值：

GL_COLOR_INDEX

颜色索引从函数glReadBuffer()选定的颜色缓冲区中读取。每个索引被转化为一种定点格式。这种转化操作即根据参数**GL_INDEX_SHIFT**的值和符号的不同替换索引的左边或右边的若干二进制位，然后再加上**GL_INDEX_OFFSET**。如果**GL_MAP_COLOR**是**GL_TRUE**，则索引值由映射表**GL_PIXEL_MAP_I_TO_I**中的映射值替换。

GL_STENCIL_INDEX

模板值从模板缓冲区中读取。每个索引值被转化为一种定点格式。这种转化操作即根据参数**GL_INDEX_SHIFT**的值和符号的不同替换索引的左边或右边的若干二进制位，然后再加上**GL_INDEX_OFFSET**。如果**GL_MAP_STENCIL**是**GL_TRUE**，则索引值由映射表**GL_PIXEL_MAP_S_TO_S**中的映射值替换。

GL_DEPTH_COMPONENT

深度值从深度缓冲区中读取。每个颜色组分被转化为浮点格式。这时最小的深度值被映射为0，最大的深度值被映射为1。然后把所得的深度值乘以**GL_DEPTH_SCALE**，再加上**GL_DEPTH_BIAS**即得最后的深度组分值。其取值范围是[0, 1]。

GL_RED

GL_GREEN

GL_BLUE

GL_ALPHA

GL_RGB

GL_BGR

GL_RGBA

GL_BGRA

GL_LUMINANCE

GL_LUMINANCE_ALPHA

它们处理过程的不同取决于颜色缓冲区中存储的是颜色索引还是RGBA颜色组分。如果存储的是颜色索引，则这些颜色索引将从函数glReadBuffer()选定的颜色缓冲区中读取。每个索引被转化为一种定点格式。这种转化操作即根据参数**GL_INDEX_SHIFT**的值和符号的不同替换索引的左边或右边的若干二进制位，然后加上**GL_INDEX_OFFSET**。索引中的红、绿、蓝和alpha值通过查找映射表**GL_PIXEL_MAP_I_TO_R**、**GL_PIXEL_MAP_I_TO_G**、**GL_PIXEL_MAP_I_TO_B**和**GL_PIXEL_MAP_I_TO_A**而获得。每一个映射表的尺寸必须是 2^n ，但*n*对于不同的映射表可以不同。用一个索引在尺寸为 2^n 的

映射表中查找一个值时，它必须被 2^c-1 所屏蔽。

如果颜色缓冲区中存储的是RGBA颜色组分，则这些组分值将从函数glReadBuffer()选定的颜色缓冲区中读取。每个颜色组分被转化为浮点格式。这时零强度映射为0.0，满强度映射为1.0。然后把所得的每一组分值乘以GL_c_SCALE，再加上GL_c_BIAS即得最后的组分值，这里c分别代表RED、GREEN、BLUE和ALPHA。每一组分值取值范围都是[0, 1]。当每一组分放大到其相关映射表的尺寸后，它将被GL_PIXEL_MAP_c_TO_c中的映射值所替代，这里c是R、G、B或A。

不需要的数据将被丢弃。比如，GL_RED中丢弃绿、蓝和alpha组分，GL_RED中仅丢弃alpha组分。GL_LUMINANCE中计算出一个单组分值作为红、绿和蓝组分的总和，GL_LUMINANCE_ALPHA也作同样的工作，只是它把第二个值作为alpha值。最终值的取值范围是[0, 1]。

上面所讨论的替换、放大、偏移和查询因子都由函数glPixelTransfer()指定。查询表自身的内容由函数glPixelMap()指定。最后，索引或颜色组分都转换为参数type指定的正确格式。如果参数format是GL_COLOR_INDEX或GL_STENCIL_INDEX，并且参数type不是GL_FLOAT时，每一个索引将被屏蔽为一个下表所给出的屏蔽值。如果参数type是GL_FLOAT，则每一个整型索引值将被转化为一个单精度浮点格式。

如果参数format是GL_RED、GL_GREEN、GL_BLUE、GL_ALPHA、GL_RGB、GL_BGR、GL_RGBA、GL_BGRA、GL_LUMINANCE或GL_LUMINANCE_ALPHA，并且参数type不是GL_FLOAT时，每一个颜色组分将乘上下表所给出的乘子。如果参数type是GL_FLOAT，则每个颜色组分被原样传送（或当客户端格式与GL使用的格式不一样时，它会被转化为一种客户端所需的单精度浮点格式）。见表5-26。

表 5-26

type	索引标记	组分转化方式
GL_UNSIGNED_BYTE	2^{8-1}	$(2^{8-1})c$
GL_BYTE	2^{7-1}	$[(2^{8-1})c-1]/2$
GL_BITMAP	1	1
GL_UNSIGNED_SHORT	2^{16-1}	$(2^{16-1})c$
GL_SHORT	2^{15-1}	$[(2^{16-1})c-1]/2$
GL_UNSIGNED_INT	2^{32-1}	$(2^{32-1})c$
GL_INT	2^{31-1}	$[(2^{32-1})c-1]/2$
GL_FLOAT	none	c

返回值按下面方式放入内存。如果format是GL_COLOR_INDEX、GL_STENCIL_INDEX、GL_DEPTH_COMPONENT、GL_RED、GL_GREEN、GL_BLUE、GL_ALPHA或GL_LUMINANCE时，返回一个单值，且第j行的第i个像素的数据被放在存储单元(j)width+i。对于每一个像素，GL_RGB和GL_BGR返回三个值，GL_RGBA和GL_BGRA返回四个值，GL_LUMINANCE_ALPHA返回二个值。每一像素的所有返回值占据数组pixels中的连续空间。

由函数glPixelStore()所设置的如GL_PACK_LSB_FIRST和GL_PACK_SWAP_BYTES这样的存储参数将影响数据向内存中的写操作的方式。具体情况详见glPixelStore()。

- 注意：

位于与当前GL环境相关联的窗口以外的像素值将是未定义的。

如果有错误发生，数组*pixels*中的内容将不发生变化。

- 出错提示：

当参数*format*和*type*不是一个可接受值时产生GL_INVALID_ENUM提示。

当参数*type*是GL_BITMAP，而参数*format*既非GL_COLOR_INDEX又非GL_STENCIL_INDEX时，产生GL_INVALID_ENUM提示。

当参数*width*或*height*是负数时，产生GL_INVALID_VALUE提示。

当参数*format*是GL_COLOR_INDEX，但颜色缓冲区中存储的却是RGBA颜色组分时，产生GL_INVALID_OPERATION提示。

当参数*format*是GL_STENCIL_INDEX，但却没有模板缓冲区时，产生GL_INVALID_OPERATION提示。

当参数*format*是GL_DEPTH_COMPONENT，但却没有深度缓冲区时，产生GL_INVALID_OPERATION提示。

当函数glReadPixels()在函数对glBegin()/glEnd()之间执行时，产生GL_INVALID_OPERATION提示。

当参数*type*为GL_UNSIGNED_BYTE_3_3_2、GL_UNSIGNED_BYTE_2_3_3_REV、GL_UNSIGNED_SHORT_5_6_5、GL_UNSIGNED_SHORT_5_6_5_REV之一，但*format*却不是GL_RGB格式时，产生GL_INVALID_OPERATION提示。

当参数*type*为GL_UNSIGNED_SHORT_4_4_4、GL_UNSIGNED_SHORT_4_4_4_REV、GL_UNSIGNED_SHORT_5_5_5_1、GL_UNSIGNED_SHORT_1_5_5_5_REV、GL_UNSIGNED_INT_8_8_8_8、GL_UNSIGNED_INT_8_8_8_8_REV、GL_UNSIGNED_INT_10_10_2和GL_UNSIGNED_INT_2_10_10_10_REV之一，但*format*既不是GL_RGBA格式又不是GL_BGRA格式时，产生GL_INVALID_OPERATION提示。

仅在GL 1.2以上的版本中，GL_BGR和GL_BGRA才是参数*format*的有效值。

仅在GL 1.2以上的版本中，GL_UNSIGNED_BYTE_3_3_2、GL_UNSIGNED_BYTE_2_3_3_REV、GL_UNSIGNED_SHORT_5_6_5、GL_UNSIGNED_SHORT_5_6_5_REV、GL_UNSIGNED_SHORT_4_4_4、GL_UNSIGNED_SHORT_4_4_4_REV、GL_UNSIGNED_SHORT_5_5_5_1、GL_UNSIGNED_SHORT_1_5_5_5_REV、GL_UNSIGNED_INT_8_8_8_8、GL_UNSIGNED_INT_8_8_8_8_REV、GL_UNSIGNED_INT_10_10_2和GL_UNSIGNED_INT_2_10_10_10_REV才是参数*type*的有效值。

- 有关数据的获取：

glGet(GL_INDEX_MODE)

- 请参阅：

glCopyPixels(), glDrawPixels(), glPixelMap(), glPixelStore(), glPixelTransfer(),

glReadBuffer()**glRect**

- 名称:

glRectd(), **glRectf()**, **glRecti()**, **glRects()**, **glRectdv()**, **glRectfv()**, **glRectiv()**, **glRectsv()**

- 功能:

绘制一个矩形。

- C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)
```

- 参数说明:

x1, *y1*

指定矩形的一个顶点。

x2, *y2*

指定矩形的另一个相对的顶点。

- C描述:

```
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 )
```

- 参数说明：

v1 指定一个指针，指向矩形的一个顶点。

v2 指定一个指针，指向矩形的另一个相对的顶点。

- 说明：

函数glRect()通过指定矩形的两个角从而有效地定义了一个矩形。每个函数所带的四个参数指定了两个相关的(x, y)坐标对；或由两个指针指定两个数组，每个数组包含一个(x, y)坐标对。最后形成的矩形被定义在z=0平面上。

函数glRect(*x1*, *y1*, *x2*, *y2*)与下面的程序完全等价：

```
glBegin(GL_POLYGON);
glVertex2(x1, y1);
glVertex2(x2, y1);
glVertex2(x2, y2);
glVertex2(x1, y2);
glEnd();
```

- 注意：如果第二个顶点在第一个顶点的右上方，则该矩形将沿逆时针方向形成。

- 出错提示：

当函数glRect()在函数对glBegin()/glEnd()之间执行时产生**GL_INVALID_OPERATION**提示。

- 请参阅：

glBegin(), glVertex()

• glRenderMode

- 名称：

glRenderMode()

- 功能：

设置光栅化模式。

- C描述：

```
GLint glRenderMode( GLenum mode )
```

- 参数说明：

mode 指定光栅化模式。它可取下面三个值：**GL_RENDER**、**GL_SELECT**和**GL_FEEDBACK**。其初始值是**GL_RENDER**。

- 说明：

函数glRenderMode()用于设置光栅化的模式。它带有一个参数*mode*，该值可以取三个预先定义的值之一：

GL_RENDER

绘图模式。在这种模式中，图元被光栅化后产生像素片断，然后这些片断被写入帧缓存区中。这是一种常用模式，也是默认模式。

GL_SELECT

选择模式。在这种模式中，不产生像素片断，也不改变帧缓存区中的内容。然而，如果绘图模式是**GL_RENDER**，要绘制的一个图元的名称记录将被返回到一个选择缓冲区中。这样，在进入选择模式之前，首先应该创建该缓冲区（请参阅**glSelectBuffer()**）。

GL_FEEDBACK

反馈模式。在这种模式中，不产生像素片断，也不改变帧缓存区中的内容。然而，如果绘图模式是**GL_RENDER**，要绘制的顶点的坐标和属性将被返回到一个反馈缓冲区中。这时，在进入选择模式之前，首先应该创建该缓冲区（请参阅**glFeedbackBuffer()**）。

函数**glRenderMode()**的返回值是由调用该函数时的绘图模式而不是由参数*mode*来决定的。三种绘图模式的返回值如下：

GL_RENDER 0。

GL_SELECT 传送到选择缓冲区的命中记录个数。

GL_FEEDBACK 传送到反馈缓冲区的值的个数。

有关选择和反馈操作的细节请参考函数**glSelectBuffer()**和**glFeedbackBuffer()**。

- 注意：

当有错误发生时，不管当前处于哪种绘图模式，函数**glRenderMode()**的返回值均为0。

- 出错提示：

当参数*mode*不是一个可接受的值时产生**GL_INVALID_ENUM**提示。

当绘图模式是**GL_SELECT**时调用函数**glSelectBuffer()**，或调用函数**glRenderMode(GL_SELECT)**前一次都没调用过函数**glSelectBuffer()**，将产生**GL_INVALID_OPERATION**提示。

当绘图模式是**GL_FEEDBACK**时调用函数**glFeedbackBuffer()**，或调用函数**glRenderMode(GL_FEEDBACK)**前一次都没调用过函数**glFeedbackBuffer()**，将产生**GL_INVALID_OPERATION**提示。

函数**glRenderMode()**在函数对**glBegin()/glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGet(GL_RENDER_MODE)

- 请参阅：

glFeedbackBuffer(), **glInitNames()**, **glLoadName()**, **glPassThrough()**, **glPushName()**,
glSelectBuffer()

glResetHistogram

- 名称：

glResetHistogram()

- 功能：

将直方图表的条目重新设置成0。

- C描述：

```
void glResetHistogram( GLenum target )
```

- 参数说明：

target 必须是**GL_HISTOGRAM**。

- 说明：

函数**glResetHistogram()**将把当前直方图表中的元素重新设置成0。

- 注意：

当调用函数**glGetString(GL_EXTENSIONS)**的返回值是**GL_ARB_imaging**时，才支持函数**glResetHistogram()**。

- 出错提示：

当参数*target*不是**GL_HISTOGRAM**时产生**GL_INVALID_ENUM**提示。

当函数**glResetHistogram()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 请参阅：

glHistogram()

glResetMinmax

- 名称：

glResetMinmax()

- 功能：

将极值表的条目重新设置成初始值。

- C描述：

```
void glResetMinmax( GLenum target )
```

- 参数说明：

target 必须是**GL_MINMAX**。

- 说明：

函数**glResetMinmax()**将把当前极值表中的元素重新设置成它们的初始值：“最大”的元素重新设置成最小可能的组分值，“最小”的元素重新设置成最大可能的组分值。

- 注意：

当调用函数**glGetString(GL_EXTENSIONS)**的返回值是**GL_ARB_imaging**时，才支持函数**glResetMinmax()**。

- 出错提示：

当参数*target*不是**GL_MINMAX**时产生**GL_INVALID_ENUM**提示。

当函数**glResetMinmax()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 请参阅：

glMinmax()**• glRotate**

- 名称:

glRotated(), **glRotatef()**

- 功能:

把当前矩阵乘上一个旋转矩阵。

- C 描述:

```
void glRotated ( GLdouble angle,
```

```
    GLdouble x,
```

```
    GLdouble y,
```

```
    GLdouble z )
```

```
void glRotatef ( GLfloat angle,
```

```
    GLfloat x,
```

```
    GLfloat y,
```

```
    GLfloat z )
```

- 参数说明:

angle 指定旋转角度。其单位是“度”(°)。

x, *y*, *z*

分别指定一个向量的*x*、*y*和*z*坐标。

- 说明:

函数**glRotate()**的作用是绕向量(*x*, *y*, *z*)产生一个*angle*角度的旋转。当前矩阵(请参阅**glMatrixMode()**)将被它与一个旋转矩阵相乘后所得的矩阵所替代。这一操作相当于调用了将如下矩阵作为其自变量的**glMultMatrix()**函数:

$$\begin{matrix} x^2(1 - c) + c & xy(1 - c) - zs & xz(1 - c) + ys & 0 \\ yx(1 - c) + zs & y^2(1 - c) + c & yz(1 - c) - xs & 0 \\ xz(1 - c) - ys & yz(1 - c) + xs & z^2(1 - c) + c & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

在此, $c = \cos(\text{angle})$, $s = \sin(\text{angle})$, 且 $\|(x, y, z)\| = 1$ (否则, GL将归一化该向量)。

如果矩阵模式是**GL_MODELVIEW**或**GL_PROJECTION**时, 函数**glRotate()**调用后所有要绘制的对象将被旋转。你可以使用函数**glPushMatrix()**和**glPopMatrix()**存储和恢复未旋转的坐标系统。

- 注意:

这种旋转操作遵循右手规则, 所以如果向量(*x*, *y*, *z*)朝向用户方向, 旋转将按逆时针方向进行。

- 出错提示:

当函数**glRotate()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示

- 有关数据的获取：

```
glGet( GL_MATRIX_MODE )
glGet( GL_COLOR_MATRIX )
glGet( GL_MODELVIEW_MATRIX )
glGet( GL_PROJECTION_MATRIX )
glGet( GL_TEXTURE_MATRIX )
```

- 请参阅：

glMatrixMode(), **glMultMatrix()**, **glPushMatrix()**, **glScale()**, **glTranslate()**

• **glScale**

- 名称：

glScaled(), **glScalef()**

- 功能：

将当前矩阵乘上一个普通缩放矩阵。

- C 描述：

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

- 参数说明：

x, *y*, *z* 分别指定沿*x*、*y*和*z*轴方向的缩放因子。

- 说明：

函数**glScale()**产生一个沿*x*、*y*和*z*轴方向的缩放因子。这三个参数分别代表沿三个轴所需的缩放因子。

当前矩阵（请参阅**glMatrixMode()**）将被它与这个缩放矩阵相乘的结果所替换。这一操作相当于调用了用下面的矩阵作为其自变量的**glScale()**函数：

$$\begin{matrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

如果矩阵模式是**GL_MODELVIEW**或**GL_PROJECTION**时，函数**glScale()**调用后所有要绘制的对象都将被缩放。

你可以使用函数**glPushMatrix()**和**glPopMatrix()**存储和恢复未缩放的坐标系统。

- 注意：

如果缩放因子不是1并且已经启动了模式取景矩阵和光照操作，则光照操作会经常出现错误。

这时，系统会调用函数glEnable(GL_NORMALIZE)对法向量进行自动地归一化处理。

- 出错提示：

函数glScale()在函数对glBegin()/glEnd()之间执行时产生GL_INVALID_OPERATION提示。

- 有关数据的获取：

```
glGet( GL_MATRIX_MODE )
glGet( GL_COLOR_MATRIX )
glGet( GL_MODELVIEW_MATRIX )
glGet( GL_PROJECTION_MATRIX )
glGet( GL_TEXTURE_MATRIX )
```

- 请参阅：

glMatrixMode(), glMultMatrix(), glPushMatrix(), glRotate(), glTranslate()

• glScissor

- 名称：

glScissor()

- 功能：

定义裁剪箱。

- C描述：

```
void glScissor( GLint x,
                GLint y,
                GLsizei width,
                GLsizei height )
```

- 参数说明：

x, *y*

指定裁剪箱的左下角。初始值为(0, 0)。

width, *height*

指定裁剪箱的宽度和高度。当一个GL环境第一次与一个窗口连接时，参数*width*和*height*将被设置成与窗口的尺寸。

- 说明：

函数glScissor()的作用是在窗口坐标中定义一个名为裁剪箱的矩形。其前两个参数*x*和*y*用于指定裁剪箱的左下角。参数*width*和*height*用于指定裁剪箱的宽度和高度。

函数glEnable(GL_SCISSOR_TEST)和glDisable(GL_SCISSOR_TEST)用来启动和关闭裁剪测试。初始情况下，该测试是关闭的。当启动裁剪测试后，只有位于裁剪箱内的像素才能被绘图命令修改。帧缓冲区中的像素在共用的角上含有整型的窗口坐标值。命令glScissor(0, 0, 1, 1)仅允许修改窗口中左下角的像素。命令glScissor(0, 0, 0, 0)不允许修改窗口中的任何像素。

当关闭裁剪测试后，相当于裁剪箱包含整个窗口。

- 出错提示：

当参数*width*或*height*是负数时产生**GL_INVALID_VALUE**提示。

当函数**glScissor()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

```
glGet( GL_SCISSOR_BOX )
glIsEnabled( GL_SCISSOR_TEST )
```

- 请参阅：

glEnable(), **glViewPoint()**

• **glSelectBuffer**

- 名称：

glSelectBuffer()

- 功能：

为选择模式产生的值建立一个缓冲区。

- C 描述：

```
void glSelectBuffer( GLsizei size,
                    GLfloat *buffer )
```

- 参数说明：

size 指定 *buffer* 的大小。

buffer 返回选择数据。

- 说明：

函数**glSelectBuffer()**带有两个自变量：*buffer*是一个指向一个无符号整型数组的指针，*size*确定数组的大小。当绘图模式是**GL_SELECT**时，*buffer*将从名称堆栈中返回数据（请参阅**glInitNames()**, **glLoadName()**, **glPushName()**）。函数**glSelectBuffer()**必须在启动选择模式之前就被发布，并且当绘图模式是**GL_SELECT**时是不能发布该函数的。

程序员可以采用选择操作来决定哪些图元将被绘入一个窗口的某些区域。这些区域由当前的模式取景和投射矩阵定义。

在选择模式下，不能通过光栅化产生任何像素片断。相反地，如果一个图元或一个光栅位置与一个由用户定义的剪切平面或观察截锥体所定义的剪切体积相交时，这个图元产生一个选择命中。（对于多边形而言，如果该多边形被拣选，将不产生选择命中）。当名称堆栈发生改变或当函数**glRenderMode()**被调用时，如果已产生了选择命中，则这个命中记录将被拷贝到*buffer*中。该命中记录包含了事件发生时名称堆栈中名称的数目，然后是上一次事件发生时所有顶点的最小和最大深度值，再接下来是名称堆栈的内容和第一个底部的名称。

放入命中记录前，深度值（其取值范围是[0, 1]）要先乘上 $2^{32}-1$ 。

进入选择模式前，*buffer*中的一个内部索引被重设为0。命中记录被拷贝入*buffer*一次，这个索引就把一个点通过名称块的尾部加到单元中——它也就是下一个有效的单元。如果命中记录的数目超过了*buffer*中剩余的存储单元的个数，则尽可能多地把命中记录拷贝入*buffer*中，并设置一个溢出标记。如果一个命中记录被拷贝时名称堆栈时空的，则该记录中最小和最大深度值后面将是0。

如果调用函数glRenderMode()时所带的参数不是GL_SELECT，GL将退出选择模式。当绘图模式是GL_SELECT时，无论何时调用函数glRenderMode()，它都将返回被拷贝进buffer中的命中记录的个数，并重新设置溢出标记和选择缓冲区指针，然后把名称堆栈初始化为空堆栈。当调用函数glRenderMode()时，如果已设置溢出标志，则返回一个负的命中记录数值。

- 注意：

在调用函数glRenderMode()时所带的参数不是GL_SELECT之前，buffer中的内容是未定义的。

用函数对glBegin()/glEnd()绘制图元和调用函数glRasterPos()将产生命中。

- 出错提示：

当参数size是负数时产生GL_INVALID_VALUE提示。

当绘图模式不是GL_SELECT时调用函数glSelectBuffer()，或调用函数glRenderMode(GL_SELECT)前已经调用过函数glSelectBuffer()，将产生GL_INVALID_OPERATION提示。

当函数glSelectBuffer()在函数对glBegin()/glEnd()之间执行时产生GL_INVALID_OPERATION提示。

- 有关数据的获取：

```
glGet( GL_NAME_STACK_DEPTH )
glGet( GL_SELECTION_BUFFER_SIZE )
glGetPointerv( GL_SELECTION_BUFFER_POINTER )
```

- 请参阅：

glFeedbackBuffer(), glInitNames(), glLoadName(), glPushName(), glRenderMode()

• glSeparableFilter2D

- 名称：

glSeparableFilter2D()

- 功能：

定义一个可分离的二维卷积滤波器。

- C描述：

```
void glSeparableFilter2D( GLenum target,
                           GLenum internalformat,
                           GLsizei width,
                           GLsizei height,
                           GLenum format,
                           GLenum type,
                           const GLvoid *row,
                           const GLvoid *column )
```

- 参数说明：

target 必须是GL_SEPARABLE_2D。

internalformat

卷积滤波器内核的内部格式。它可取的值有：**GL_ALPHA**、**GL_ALPHA4**、**GL_ALPHA8**、**GL_ALPHA12**、**GL_ALPHA16**、**GL_LUMINANCE**、**GL_LUMINANCE4**、**GL_LUMINANCE8**、**GL_LUMINANCE12**、**GL_LUMINANCE16**、**GL_LUMINANCE_ALPHA**、**GL_LUMINANCE4_ALPHA4**、**GL_LUMINANCE6_ALPHA2**、**GL_LUMINANCE8_ALPHA8**、**GL_LUMINANCE12_ALPHA4**、**GL_LUMINANCE12_ALPHA12**、**GL_LUMINANCE16_ALPHA16**、**GL_INTENSITY**、**GL_INTENSITY4**、**GL_INTENSITY8**、**GL_INTENSITY12**、**GL_INTENSITY16**、**GL_R3_G3_B2**、**GL_RGB**、**GL_RGB4**、**GL_RGB5**、**GL_RGB8**、**GL_RGB10**、**GL_RGB12**、**GL_RGB16**、**GL_RGBA**、**GL_RGBA2**、**GL_RGBA4**、**GL_RGB5_A1**、**GL_RGBA8**、**GL_RGB10_A2**、**GL_RGBA12**和**GL_RGBA16**。

width 由参数*row*提供的像素数组中元素的数目。(它也就是可分离滤波器内核的宽度。)

height 由参数*column*提供的像素数组中元素的数目。(它也就是可分离滤波器内核的高度。)

format 参数*row*和*column*中像素数据的格式。它允许取的值是：**GL_RED**、**GL_GREEN**、**GL_BLUE**、**GL_ALPHA**、**GL_RGB**、**GL_BGR**、**GL_RGBA**、**GL_BGRA**、**GL_INTENSITY**、**GL_LUMINANCE**和**GL_LUMINANCE_ALPHA**。

type 参数*row*和*column*中像素数据的类型。它允许取的值有：**GL_UNSIGNED_BYTE**、**GL_BYTE**、**GL_BITMAP**、**GL_UNSIGNED_SHORT**、**GL_SHORT**、**GL_UNSIGNED_INT**、**GL_INT**、**GL_FLOAT**、**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**。

row 一个指向像素数据的一维数组的指针。该数据用来建立行滤波器的内核。

column 一个指向像素数据的一维数组的指针。该数据用来建立列滤波器的内核。

• 说明：

函数**glSeparableFilter2D()**的作用是由两个像素数组产生一个二维的可分离的卷积滤波器的内核。

由参数(*width*, *format*, *type*, *row*)和(*height*, *format*, *type*, *column*)指定的像素数组的处理过程跟将它们送入函数**glDrawPixels()**是一样的，但当它最后完全扩展成RGBA后，该过程将停止。

接下来，两个数组中的所有像素的R、G、B和A组分将被四个分离的2D **GL_CONV**-

LUTION_FILTER_SCALE参数进行缩放，并由四个分离的2D **GL_CONVOLUTION_FILTER_BIAS**参数进行偏移。（缩放和偏移参数由函数**glConvolutionParameter()**用目标值**GL_SEPARABLE_2D**和名称值**GL_CONVOLUTION_FILTER_SCALE**及**GL_CONVOLUTION_FILTER_BIAS**来设定。这些参数都是由四个值所构成的向量，这四个值依次为红、绿、蓝和alpha。）在这个过程中，R、G、B和A值不必被截断到范围[0, 1]内。

然后，每个像素将被转换成由参数*internalformat*指定的内部格式。这种转换仅仅是由像素的组分值（R、G、B和A）简单地映射到内部格式所包含的值（红、绿、蓝、alpha、亮度及强度）。其映射关系如表5-27：

表 5-27

内部格式	红	绿	蓝	Alpha	亮度	强度
GL_LUMINANCE					R	
GL_LUMINANCE_ALPHA				A	R	
GL_INTENSITY						R
GL_RGB	R	G	B			
GL_RGBA	R	G	B	A		

所得像素的红、绿、蓝、alpha、亮度和/或强度组分将被存储为浮点型而非整型格式。它们组成两个一维滤波器内核的图像。行图像将通过坐标*i*从0开始由左到右地检索，每个行图像中的存储单元由*row*中的第*i*个元素得到。列图像将通过坐标*j*从0开始由低到高地检索，每个列图像中的存储单元由*column*中的第*j*个元素得到。

值得注意的是经过这样的操作所得的颜色组分同样要由它们相应的参数**GL_POST_CONVOLUTION_c_SCALE**进行缩放，并由参数**GL_POST_CONVOLUTION_c_BIAS**进行偏移（这里的c代表值RED、GREEN、BLUE和ALPHA）。这两个参数由函数**glPixelTransfer()**设置。

- 注意：

只有当调用函数**glGetString(GL_EXTENSIONS)**的返回值是**GL_ARB_imaging**时，函数**glSeparableFilter2D()**才被支持。

- 出错提示：

当参数*target*不是**GL_SEPARABLE_2D**时产生**GL_INVALID_ENUM**提示。

当参数*internalformat*不是其可取值之一时产生**GL_INVALID_ENUM**提示。

当参数*width*小于0或大于其最大支持值时产生**GL_INVALID_VALUE**提示。这个最大支持值需要由函数**glGetConvolutionParameter()**通过使用目标值**GL_SEPARABLE_2D**和名称值**GL_MAX_CONVOLUTION_WIDTH**来查询。

当参数*height*小于0或大于其最大支持值时产生**GL_INVALID_VALUE**提示。这个最大支持值需要由函数**glGetConvolutionParameter()**通过使用目标值**GL_SEPARABLE_2D**和名称值**GL_MAX_CONVOLUTION_HEIGHT**来查询。

当参数*format*不是一个允许的值时产生**GL_INVALID_ENUM**提示。

当参数*type*不是一个允许的值时产生**GL_INVALID_ENUM**提示。

当函数glSeparableFilter2D()在函数对glBegin()/glEnd()之间执行时产生GL_INVALID_OPERATION提示。

当参数*type*为GL_UNSIGNED_BYTE_3_3_2、GL_UNSIGNED_BYTE_2_3_3_REV、GL_UNSIGNED_SHORT_5_6_5或GL_UNSIGNED_SHORT_5_6_5_REV之一，但*format*却不是GL_RGB格式时产生GL_INVALID_OPERATION提示。

当参数*type*为GL_UNSIGNED_SHORT_4_4_4_4、GL_UNSIGNED_SHORT_4_4_4_4_REV、GL_UNSIGNED_SHORT_5_5_5_1、GL_UNSIGNED_SHORT_1_5_5_5_REV、GL_UNSIGNED_INT_8_8_8_8、GL_UNSIGNED_INT_8_8_8_8_REV、GL_UNSIGNED_INT_10_10_10_2或GL_UNSIGNED_INT_2_10_10_10_REV之一，但*format*既不是GL_RGBA格式又不是GL_BGRA格式时产生GL_INVALID_OPERATION提示。

- 有关数据的获取：

glGetConvolutionParameter()

glGetSeparableFilter()

- 请参阅：

glConvolutionFilter1D(), **glConvolutionFilter2D()**, **glConvolutionParameter()**,
glPixelTransfer()

• **glShadeModel**

- 名称：

glShadeModel()

- 功能：

选择平坦或光滑浓淡处理。

- C描述：

void glShadeModel(GLenum mode)

- 参数说明：

mode 指定一个表示一种浓淡处理技巧的符号值。它可取的值为GL_FLAT和GL_SMOOTH。其初始值是GL_SMOOTH。

- 说明：

GL图元可以采用平坦或光滑两种浓淡处理方法。光滑浓淡处理是它的默认方法。这种方法是当一个图源被光栅化时用内插法计算顶点的颜色，从而为每个得到的像素片断分配各自不同的颜色。而平坦浓淡处理是指选取一个顶点来计算颜色，并把计算值赋给由光栅化一个图元产生的所有像素片断。在这两种浓淡处理中，计算所得的顶点颜色值在光照模式启动时是指光照的结果；如果光照模式是关闭的，它则代表这时指定顶点的当前颜色。

对于点而言，平坦浓淡处理和光滑浓淡处理的结果是没有区别的。当函数glBegin()发布时，它将从1开始计算顶点和图元。GL将把线段*i*的下一个顶点*i*+1的计算颜色赋给平坦浓淡处理的线段*i*。类似地，GL也从1开始计算多边形。赋给平坦浓淡处理的多边形计算颜色的顶点如表5-28。除了单多边形之外，其他多边形都是由最后一个顶点来确定多边形的颜色。

表 5-28

多边形的图元类型	顶 点
单多边形 ($i=1$)	1
带状三角形	$i+2$
扇形三角形	$i+2$
独立的三角形	$3i$
相连的四边形	$2i+2$
独立的四边形	$4i$

平坦浓淡处理和光滑浓淡处理分别由函数 **glShadeModel(GL_FLAT)** 和 **glShadeModel(GL_SMOOTH)** 指定。

- 出错提示：

如果参数 *mode* 不是 **GL_FLAT** 或 **GL_SMOOTH** 之一，则产生 **GL_INVALID_ENUM** 提示。

当函数 **glShadeModel()** 在函数对 **glBegin()**/**glEnd()** 之间执行时产生 **GL_INVALID_OPERATION** 提示。

- 有关数据的获取：

glGet(GL_SHADE_MODEL)

- 请参阅：

glBegin(), **glColor()**, **glLight()**, **glLightModel()**

glStencilFunc

- 名称：

glStencilFunc()

- 功能：

为模板测试设置函数和参考值。

- C 描述：

```
void glStencilFunc( GLenum func,
                    GLint ref,
                    GLuint mask )
```

- 参数说明：

func 指定测试函数。它可取的值是 **GL_NEVER**, **GL_LESS**, **GL_EQUAL**, **GL_GREATER**, **GL_GEQUAL**, **GL_NOTEQUAL** 和 **GL_ALWAYS**。其初始值是 **GL_ALWAYS**。

ref 为模板测试设定参考值。参数 *ref* 的取值范围是 $[0, 2^n - 1]$ ，此处 *n* 是模板缓冲区中位面的数目。其初始值是 0。

mask 指定一个屏蔽，当执行测试时对存储的模板值和参考值进行逻辑“与”操作。其初始值是全 1。

- 说明：

像深度缓存测试一样，模板测试用于控制对每一像素基的绘制动作。你可以把GL图元绘入模板平面，然后就可以用模板平面屏蔽掉部分绘制的几何体和图像在屏幕中的显示。模板测试的典型应用是用多通道绘制算法获取特殊的效果图，如贴纸、轮廓图和实心几何体绘制等。

基于模板中的值与参考值的比较结果，模板测试将有条件地删除一些像素。调用函数 `glEnable(GL_STENCIL_TEST)` 和 `glDisable(GL_STENCIL_TEST)` 可以启动和关闭测试操作。如果要指定基于模板测试输出的操作，请调用函数 `glStencilOp()`。

参数 `func` 是一个符号常量，用来确定模板比较函数。它可取的八个值如下。参数 `ref` 是一个整型参考值，它用于模板的比较。它的取值范围是 $[0, 2^n - 1]$ ，此处 n 是模板缓冲区中位面的数目。参数 `mask` 是一个同参考值及模板存储值进行逻辑“与”操作的二进制位值，其结果将参与比较。

当参数 `stencil` 提供的值是存储在相关的模板缓冲区的逻辑单元中的值时，下面显示了被参数 `func` 指定的每个比较函数所产生的影响。只有当比较成功后像素才能被送到光栅化进程的下一阶段（请参阅 `glStencilOp()`）。所有测试过程中参数 `stencil` 都被看作 $[0, 2^n - 1]$ 范围内的无符号整型值，此处 n 是模板缓冲区中位面的个数。

参数 `func` 的可取值及其含义如下：

<code>GL_NEVER</code>	总是失败
<code>GL_LESS</code>	如果(<code>ref & mask</code>) $<$ (<code>stencil & mask</code>)，则通过
<code>GL_EQUAL</code>	如果(<code>ref & mask</code>) \leq (<code>stencil & mask</code>)，则通过
<code>GL_GREATER</code>	如果(<code>ref & mask</code>) $>$ (<code>stencil & mask</code>)，则通过
<code>GL_GEQUAL</code>	如果(<code>ref & mask</code>) \geq (<code>stencil & mask</code>)，则通过
<code>GL_EQUAL</code>	如果(<code>ref & mask</code>) $=$ (<code>stencil & mask</code>)，则通过
<code>GL_NOTEQUAL</code>	如果(<code>ref & mask</code>) \neq (<code>stencil & mask</code>)，则通过
<code>GL_ALWAYS</code>	总是通过

- 注意：

初始情况下，模板测试关闭。当没有模板缓冲区时，模板修改将不发生，这种情况相当于模板测试总是通过。

- 出错提示：

当参数 `func` 不是八个可取值之一时产生 `GL_INVALID_ENUM` 提示。

当函数 `glStencilFunc()` 在函数对 `glBegin()/glEnd()` 之间执行时产生 `GL_INVALID_OPERATION` 提示。

- 有关数据的获取：

```
glGet( GL_STENCIL_FUNC )
glGet( GL_STENCIL_VALUE_MASK )
glGet( GL_STENCIL_REF )
glGet( GL_STENCIL_BITS )
glIsEnabled( GL_STENCIL_TEST )
```

- 请参阅：

`glAlphaFunc()`, `glBlendFunc()`, `glDepthFunc()`, `glEnable()`, `glIsEnabled()`, `glLogicOp()`,

glStencilOp()*** glStencilMask**

- 名称：

glStencilMask()

- 功能：

控制模板平面中单独的二进制位的写操作。

- C描述：

```
void glStencilMask( GLuint mask )
```

- 参数说明：

mask 指定一个二进制位的屏蔽，用来控制模板平面中单独的二进制位的写操作。初始情况下，该屏蔽是全1的。

- 说明：

函数glStencilMask()的作用是控制模板平面中单独的二进制位的写操作。参数*mask*中至少应该包含*n*个有效位才可以指定一个屏蔽，此处*n*是模板平面包含的二进制位的数目。当屏蔽中某一位是1时，就表示允许在模板缓冲区的相应位进行写操作。相反，当屏蔽中某一位是0时，则表示模板缓冲区的相应位是写保护的。初始情况下，所有的位均允许写入。

- 出错提示：

当函数glStencilMask()在函数对glBegin()/glEnd()之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

```
glGet( GL_STENCIL_WRITEMASK )
```

```
glGet( GL_STENCIL_BITS )
```

- 请参阅：

glColorMask(), **glDepthMask()**, **glIndexMask()**, **glStencilFunc()**, **glStencilOp()**

*** glStencilOp**

- 名称：

glStencilOp()

- 功能：

设置模板测试操作。

- C描述：

```
void glStencilOp( GLenum fail,
                  GLenum zfail,
                  GLenum zpass )
```

- 参数说明：

fail 指定模板测试失败时将采取的操作。它可取六个符号常量：**GL_KEEP**、**GL_ZERO**、**GL_REPLACE**、**GL_INCR**、**GL_DECR**和**GL_INVERT**。其初始值是

GL_KEEP。

zfail 指定模板测试通过但深度测试失败时将采取的操作。参数*zfail*可取的值与参数*fail*相同。其初始值是**GL_KEEP**。

zpass 指定模板测试和深度测试都通过，或当模板测试通过但没有深度缓冲区或深度缓冲区无效时将采取的操作。参数*zpass*可取的值与参数*fail*相同。其缺省值是**GL_KEEP**。

- 说明：

像深度缓存测试一样，模板测试用于控制对每一像素基的绘制。你可以把GL图元绘入模板平面中，然后用模板平面来屏蔽掉部分绘制的几何体和图像在屏幕中的显示。模板测试的典型应用是用多通道绘制算法来获取特殊的效果图，如贴纸、轮廓图和实心几何体制作等。

基于模板中的值与参考值的比较结果，模板测试将有条件地删除一些像素。用函数glEnable(GL_STENCIL_TEST)和glDisable(GL_STENCIL_TEST)来启动和关闭测试操作。要控制模板测试，可调用函数glStencilFunc()。

函数glStencilOp()所带的三个自变量用于指定当模板测试关闭后所应采取的操作。如果模板测试失败，像素的颜色或深度缓冲区将不发生变化，参数*fail*用来指定将对模板缓冲区中的内容所采取的操作。下面列出了六种可采取的操作及其含义：

GL_KEEP	保持当前值。
GL_ZERO	模板缓冲区中的值置0。
GL_REPLACE	用函数glStencilFunc()指定的参考值ref替代模板缓冲区中的值。
GL_INCR	将模板缓冲区中的当前值加1，直至指定的最大无符号值。
GL_DECR	将模板缓冲区中的当前值减1，直至0。
GL_INVERT	将模板缓冲区中的当前值进行二进制反转。

模板缓冲区中的值将被看作是无符号的整数。当进行加1或减1处理时，这些值必须是在范围[0, $2^n - 1$]之间，此处n是查询**GL_STENCIL_BITS**时所返回的值。

函数glStencilOp()的其他两个参数用来指定模板缓冲区的动作，它们将由随后的深度缓冲区测试成功(*zpass*)还是失败(*zfail*)决定(请参阅glDepthFunc())。这些动作同样使用参数*fail*所使用的六个符号常量。值得注意的是当没有深度缓冲区存在或没有启动深度缓冲区测试时，参数*zfail*将被忽略。这时如果模板测试失败，将分别由参数*fail*和*zpass*来指定模板的动作。

- 注意：

初始情况下，模板测试关闭。当没有模板缓冲区时，模板修改将不发生，这种情况相当于模板测试总是通过。这时将不考虑函数glStencilOp()的操作。

- 出错提示：

当参数*fail*、*zfail*或*zpass*不是指定的六个常量值之一时产生**GL_INVALID_ENUM**提示。

当函数glStencilOp()在函数对glBegin()/glEnd()之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGet(GL_STENCIL_FAIL)

```
glGet( GL_STENCIL_PASS_DEPTH_PASS )
glGet( GL_STENCIL_PASS_DEPTH_FAIL )
glGet( GL_STENCIL_BITS )
glIsEnabled( GL_STENCIL_TEST )
```

- 请参阅：

glAlphaFunc(), **glBlendFunc()**, **glDepthFunc()**, **glEnable()**, **glLogicOp()**, **glStencilFunc()**

• **glTexCoord**

- 名称：

```
glTexCoord1d(), glTexCoord1f(), glTexCoord1i(), glTexCoord1s(), glTexCoord2d(),
glTexCoord2f(), glTexCoord2i(), glTexCoord2s(), glTexCoord3d(), glTexCoord3f(),
glTexCoord3i(), glTexCoord3s(), glTexCoord4d(), glTexCoord4f(), glTexCoord4i(),
glTexCoord4s(), glTexCoord1dv(), glTexCoord1fv(), glTexCoord1iv(), glTexCoord1sv(),
glTexCoord2dv(), glTexCoord2fv(), glTexCoord2iv(), glTexCoord2sv(), glTexCoord3dv(),
glTexCoord3fv(), glTexCoord3iv(), glTexCoord3sv(), glTexCoord4dv(), glTexCoord4fv(),
glTexCoord4iv(), glTexCoord4sv()
```

- 功能：

设置当前纹理坐标。

- C描述：

```
void glTexCoord1d( GLdouble s )
void glTexCoord1f( GLfloat s )
void glTexCoord1i( GLint s )
void glTexCoord1s( GLshort s )
void glTexCoord2d( GLdouble s,
                  GLdouble t )
void glTexCoord2f( GLfloat s,
                  GLfloat t )
void glTexCoord2i( GLint s,
                  GLint t )
void glTexCoord2s( GLshort s,
                  GLshort t )
void glTexCoord3d( GLdouble s,
                  GLdouble t,
                  GLdouble r )
void glTexCoord3f( GLfloat s,
                  GLfloat t,
                  GLfloat r )
```

```

void glTexCoord3i( GLint s,
                  GLint t,
                  GLint r )
void glTexCoord3s( GLshort s,
                  GLshort t,
                  GLshort r )
void glTexCoord4d( GLdouble s,
                  GLdouble t,
                  GLdouble r,
                  GLdouble q )
void glTexCoord4f( GLfloat s,
                  GLfloat t,
                  GLfloat r,
                  GLfloat q )
void glTexCoord4i( GLint s,
                  GLint t,
                  GLint r,
                  GLint q )
void glTexCoord4s( GLshort s,
                  GLshort t,
                  GLshort r,
                  GLshort q )

```

• 参数说明：

s, t, r, q

指定纹理坐标*s, t, r*和*q*。并不是所有的命令形式都需要提供所有的参数。

• C 描述：

```

void glTexCoord1dv( const GLdouble *v )
void glTexCoord1fv( const GLfloat *v )
void glTexCoord1iv( const GLint *v )
void glTexCoord1sv( const GLshort *v )
void glTexCoord2dv( const GLdouble *v )
void glTexCoord2fv( const GLfloat *v )
void glTexCoord2iv( const GLint *v )
void glTexCoord2sv( const GLshort *v )
void glTexCoord3dv( const GLdouble *v )
void glTexCoord3fv( const GLfloat *v )
void glTexCoord3iv( const GLint *v )

```

```

void glTexCoord3sv( const GLshort *v )
void glTexCoord4dv( const GLdouble *v )
void glTexCoord4fv( const Gfloat *v )
void glTexCoord4iv( const GLint *v )
void glTexCoord4sv( const GLshort *v )

```

- 参数说明：

v 指定一个指向一元、二元、三元或四元数组的指针，它将顺次指定相应的纹理坐标*s*、*t*、*r*和*q*。

- 说明：

函数**glTexCoord()**指定的纹理坐标可以是一维、二维、三维或四维。函数**glTexCoord1()**设置当前的纹理坐标为(*s*, 0, 0, 1)形式；函数**glTexCoord2()**设置当前的纹理坐标为(*s*, *t*, 0, 1)形式；函数**glTexCoord3()**设置当前的纹理坐标为(*s*, *t*, *r*, 1)形式；函数**glTexCoord4()**设置当前的纹理坐标为(*s*, *t*, *r*, *q*)形式。当前纹理坐标都是与其相应的顶点和当前光栅位置的部分数据。初始情况下，*s*, *t*, *r*和*q*的值是(0, 0, 0, 1)。

- 注意：

当前纹理坐标可被随时更新。尤其地，函数**glTexCoord()**可在函数对**glBegin()**/**glEnd()**之间被调用。

当系统支持**GL_ARB_imaging**扩展时，函数**glTexCoord()**总是更新纹理单元**GL_TEXTURE0_ARB**。

- 有关数据的获取：

glGet(GL_CURRENT_TEXTURE_COORDS)

- 请参阅：

glTexCoordPointer(), **glVertex()**

• **glTexCoordPointer**

- 名称：

glTexCoordPointer()

- 功能：

定义一个纹理坐标数组。

- C描述：

```

void glTexCoordPointer(GLint size,
                       GLenum type,
                       GLsizei stride,
                       const GLvoid *pointer)

```

- 参数说明：

size 指定每个数组元素的坐标个数。它必须是1、2、3或4。其初始值是4。

type 指定每个纹理坐标的数据类型。它可取符号常量**GL_SHORT**、**GL_INT**、

GL_FLOAT和**GL_DOUBLE**。其初始值是**GL_FLOAT**。

stride 指定相邻的数组元素之间的字节偏移量。如果参数*stride*是0，则认为数组元素被一个接一个的排列在数组中。其初始值是0。

pointer 指定一个指针，指向数组中的第一个元素的第一个坐标位置。其初始值是0。

- 说明：

函数**glTexCoordPointer()**的作用是绘图时指定一个数组的纹理坐标的存放位置和数据格式。参数*size*用以指定每个元素的坐标个数，它必须时1、2、3或4。参数*type*用以指定每个纹理坐标的数据类型。参数*Stride* 指定从一个数组元素到下一个封装了顶点集及属性集的数组之间的字节跨距，其中被封装的顶点集及属性集应可以存放在单一数组或不同数组中。在某些GL实现机制中，单一数组的存放形式可能更有效）。当一个纹理坐标数组被指定后，参数*size*、*type*、*stride* 和*pointer*将被存储为客户端状态。

用函数**glEnableClientState(GL_TEXTURE_COORD_ARRAY)**和**glDisableClientState(GL_TEXTURE_COORD_ARRAY)**可以启动和关闭纹理坐标数组。当启动纹理坐标数组后，它可被函数**glDrawArrays()**、**glDrawElements()**、**glDrawRangeElement()**或**glArrayElement()**使用。

函数**glDrawArrays()**用于将预先指定的顶点和顶点属性数组构造成一个图元序列（所有相同的类型）。函数**glArrayElement()**可以通过检索顶点和顶点属性来指定图元。函数**glDrawElements()**则是通过检索顶点和顶点属性来构造一个图元序列。

- 注意：

函数**glTexCoordPointer()**只有在GL 1.1以上的版本中才可以使用。

顶点数组在初始情况下是关闭的，这时不能用函数**glDrawArrays()**、**glDrawElements()**、**glDrawRangeElements()**或**glArrayElement()**访问它。

函数**glTexCoordPointer()**不允许在函数对**glBegin()**/**glEnd()**之间执行，否则可能产生一个出错提示，但也有可能不出现出错提示。如果没有出现出错提示，其运行将是未定义的。

函数**glTexCoordPointer()**通常是在客户端执行的，并且可以不需任何协议。

因为纹理坐标数组参数是客户端状态，所以它不能用函数**glPushAttrib()**和**glPopAttrib()**存储或返回，而应用函数**glPushClientAttrib()**和**glPopClientAttrib()**代替。

- 出错提示：

当参数*size*不是1、2、3和4时产生**GL_INVALID_VALUE**提示。

当参数*type*不是一个可取的值时产生**GL_INVALID_ENUM**提示。

当参数*stride*是负数时产生**GL_INVALID_VALUE**提示。

- 有关数据的获取：

```
glIsEnabled( GL_TEXTURE_COORD_ARRAY )
glGet( GL_TEXTURE_COORD_ARRAY_SIZE )
glGet( GL_TEXTURE_COORD_ARRAY_TYPE )
glGet( GL_TEXTURE_COORD_ARRAY_STRIDE )
glGetPointerv( GL_TEXTURE_COORD_ARRAY_POINTER )
```

- 请参阅：

glArrayElement(), **glClientActiveTextureARB()**, **glColorPointer()**, **glDrawArrays()**,
glDrawElements(), **glDrawRangeElements()**, **glEdgeFlagPointer()**, **glEnable()**,
glGetPointerv(), **glIndexPointer()**, **glNormalPointer()**, **glPopClientAttrib()**,
glPushClientAttrib(), **glTexCoord()**, **glVertexPointer()**

* **glTexEnv**

- 名称：

glTexEnvf(), **glTexEnvi()**, **glTexEnvfv()**, **glTexEnviv()**

- 功能：

设置纹理环境参数。

- C描述：

```
void glTexEnvf( GLenum target,
                GLenum pname,
                GLfloat param )
void glTexEnvi( GLenum target,
                GLenum pname,
                GLint param )
```

- 参数说明：

target 指定一个纹理环境。它必须是**GL_TEXTURE_ENV**。

pname 指定一个单值纹理环境参数的符号名。它必须是**GL_TEXTURE_ENV_MODE**。

param 指定一个符号常量。它可以是**GL_MODULATE**、**GL_DECAL**、**GL_BLEND**和**GL_REPLACE**。

- C描述：

```
void glTexEnvf (GLenum target,
                GLenum pname,
                const GLfloat *params )
void glTexEnvi( GLenum target,
                GLenum pname,
                const GLint *params )
```

- 参数说明：

target 指定一个纹理环境。它必须是**GL_TEXTURE_ENV**。

pname 指定一个纹理环境参数的符号名。它可以是**GL_TEXTURE_ENV_MODE**或**GL_TEXTURE_ENV_COLOR**。

params 指定一个指向参数数组的指针。该数组包含一个单符号常量或一个RGBA颜色。

- 说明：

纹理环境指定当一个片断纹理化时如何将它编译成纹理值。参数*target*必须是**GL_**

TEXTURE_ENV。参数*pname*可以是**GL_TEXTURE_ENV_MODE**或**GL_TEXTURE_ENV_COLOR**。如果参数*pname*是**GL_TEXTURE_ENV_MODE**,则参数*params*是(或指向)一个纹理函数的符号名。可以指定的纹理函数有四个：**GL_MODULATE**, **GL_DECAL**, **GL_BLEND**和**GL_REPLACE**。

纹理函数作用于要纹理化的片断的过程就是将纹理图像值应用于片断(请参阅**glTexParameter()**)从而为每个片断生成一个RGBA颜色的过程。表5-29列出了三个可选的纹理函数所产生的RGBA颜色的情况。其中字母C代表一个三颜色值(RGB),字母A是相应的alpha值。每一图像的RGBA值的取值范围都是[0,1]。表5-29中的下标*f*表示输入的片断,下标*t*表示纹理图像,下标*c*表示纹理环境颜色,下标*v*表示由纹理函数产生的值。对于一个纹理图像而言,每个纹理元素可以包含一到四个组分(请参阅**glTexImage1D()**、**glTexImage2D()**、**glTexImage3D()**、**glCopyTexImage1D()**和**glCopyTexImage2D()**)。对于一个单组分图像,*L*表示该单组分;一个双组分图像用*L₁*和*A₁*表示组分;一个三组分图像仅有一个颜色值*C₁*,一个四组分图像则包含一个颜色值*L*和一个alpha值*A₁*。

表 5-29

组分形式		纹理函数			
<i>format</i>		GL_MODULATE	GL_DECAL	GL_BLEND	
GL_ALPHA		$C_v = C_f$ $A_v = A_f A_t$	未定义	$C_v = C_f$ $A_v = A_f$	$C_v = C_f$ $A_v = A_t$
GL_LUMINANCE		$C_v = L_t C_f$	未定义	$C_v = (1-L_t) C_f$ $+L_t C_c$	$C_v = L_t$
1		$A_v = A_f$		$A_v = A_f$	$A_v = A_f$
GL_LUMINANCE_ALPHA		$C_v = L_t C_f$	未定义	$C_v = (1-L_t) C_f$ $+L_t C_c$	$C_v = L_t$
2		$A_v = A_t A_f$		$A_v = A_t A_f$	$A_v = A_t$
GL_INTENSITY		$C_v = C_f I_t$	未定义	$C_v = (1-I_t) C_f$ $+I_t C_c$	$C_v = I_t$
c		$A_v = A_t I_t$		$A_v = (1-I_t) A_f$	$A_v = I_t$
GL_RGB		$C_v = C_t C_f$	$C_v = C_t$	$C_v = (1-C_t) C_f$ $+C_t C_c$	$C_v = C_t$
3		$A_v = A_f$	$A_v = A_f$	$A_v = A_f$	$A_v = A_f$
GL_RGBA		$C_v = C_t C_f$	$C_v = (1-A_f) C_f$ $+A_f C_t$	$C_v = (1-C_t) C_f$ $+C_t C_c$	$C_v = C_t$
4		$A_v = A_t A_f$	$A_v = A_f$	$A_v = A_t A_f$	$A_v = A_t$

当参数*pname*是**GL_TEXTURE_ENV_COLOR**时,参数*params*将是一个指向一个数组的指针,该数组包含RGBA颜色的四个值。整型颜色组分值被线性地映射为正的浮点格式:最大的正数映射为1.0,最小的负数映射为-1.0。当它们被指定时,它们将取[0,1]范围内的值。*C*取这四个值。

GL_TEXTURE_ENV_MODE的默认值是**GL_MODULATE**, **GL_TEXTURE_ENV_COLOR**的默认值是(0,0,0,0)。

- 注意：

GL_REPLACE只有在GL 1.1以上的版本中才可以使用。

除1、2、3和4之外,其他内部格式只有在GL 1.1以上的版本中才可以使用。

当系统支持**GL_ARB_imaging**扩展时,函数**glTexEnv()**用于控制当前有效的纹理单元所处的纹理环境,该单元可用函数**glActiveTextureARB()**选取。

- 出错提示：

当参数*target*和*pname*不是一个可接受的已定义值时产生**GL_INVALID_ENUM**提示。

当参数*params*应该有一个已定义的常量值(由*pname*的值决定)而没有时,也将产生**GL_INVALID_ENUM**提示。

当函数**glTexEnv()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGetTexEnv()

- 请参阅：

glActiveTextureARB(), **glCopyPixels()**, **glCopyTexImage1D()**, **glCopyTexImage2D()**,
glCopyTexSubImage1D(), **glCopyTexSubImage2D()**, **glCopyTexSubImage3D()**,
glTexImage1D(), **glTexImage2D()**, **glTexImage3D()**, **glTexParameter()**, **glTexSubImage1D()**,
glTexSubImage2D(), **glTexSubImage3D()**

◆ **glTexGen**

- 名称：

glTexGend(), **glTexGenf()**, **glTexGeni()**, **glTexGendv()**, **glTexGenfv()**, **glTexGeniv()**

- 功能：

控制纹理坐标的生成。

- C描述：

```
void glTexGend( GLenum coord,
                  GLenum pname,
                  GLdouble param )
void glTexGenf( GLenum coord,
                  GLenum pname,
                  GLfloat param )
void glTexGeni( GLenum coord,
                  GLenum pname,
                  GLint param )
```

- 参数说明：

coord 指定一个纹理坐标。它可取**GL_S**、**GL_T**、**GL_R**或**GL_Q**。

pname 指定一个纹理坐标生成函数的符号名。它必须是**GL_TEXTURE_GEN**

_MODE。

param 指定一个单值纹理生成参数。它可以是**GL_OBJECT_LINEAR**、**GL_EYE_LINEAR**或**GL_SPHERE_MAP**之一。

• C 描述：

```
void glTexGendv( GLenum coord,
                  GLenum pname,
                  const GLdouble *params )
void glTexGenfv( GLenum coord,
                  GLenum pname,
                  const GLfloat *params )
void glTexGeniv( GLenum coord,
                  GLenum pname,
                  const GLint *params )
```

• 参数说明：

coord 指定一个纹理坐标。它可取**GL_S**、**GL_T**、**GL_R**或**GL_Q**。

pname 指定一个纹理坐标生成函数或函数参数的符号名。它必须是**GL_TEXTURE_GEN_MODE**、**GL_OBJECT_PLANE**或**GL_EYE_PLANE**。

params 指定一个指向纹理生成参数数组的指针。如果参数*pname*是**GL_TEXTURE_GEN_MODE**，则该数组必须包含一个单符号常量，它可以是**GL_OBJECT_LINEAR**、**GL_EYE_LINEAR**或**GL_SPHERE_MAP**之一。否则，参数*params*将取一个由参数*pname*指定的纹理坐标生成函数的系数。

• 说明：

函数**glTexGen()**的作用是选择一种纹理坐标生成函数或提供一种生成函数的系数。参数*coord*定义一个纹理坐标(*s*, *t*, *r*, *q*)；它必须是符号值**GL_S**、**GL_T**、**GL_R**或**GL_Q**之一。参数*pname*必须是符号常量**GL_TEXTURE_GEN_MODE**、**GL_OBJECT_PLANE**或**GL_EYE_PLANE**之一。如果参数*pname*是**GL_TEXTURE_GEN_MODE**，则参数*Params*必须选择模式**GL_OBJECT_LINEAR**、**GL_EYE_LINEAR**或**GL_SPHERE_MAP**之一。如果参数*pname*是**GL_OBJECT_PLANE**或**GL_EYE_PLANE**之一，则参数*params*将包含相应的纹理坐标生成函数的系数。

当纹理生成函数是**GL_OBJECT_LINEAR**时，使用方程

$$g = p_1 x_o + p_2 y_o + p_3 z_o + p_4 w_o$$

此外*g*是由参数*coord*指定的坐标的计算值，*p₁*，*p₂*，*p₃*和*p₄*是由参数*params*提供的四个值，*x_o*，*y_o*，*z_o*和*w_o*是顶点的对象坐标。这个方程也可以用来生成以海平面为参考平面（由*p₁*，*p₂*，*p₃*和*p₄*定义）的地形纹理映射。由**GL_OBJECT_LINEAR**坐标生成函数计算得到的某一地形顶点的高度是指它与海平面的距离，这一高度可以用来作为在山顶上映射白雪和在山脚下映射绿草的依据。

当纹理生成函数是**GL_EYE_LINEAR**时，使用方程

$$g = p'_1 x_e + p'_2 y_e + p'_3 z_e + p'_4 w_e$$

生成纹理坐标，此处

$$(p'_1 \ p'_2 \ p'_3 \ p'_4) = (p_1 \ p_2 \ p_3 \ p_4) M^{-1}$$

x_e, y_e, z_e 和 w_e 是顶点的眼坐标， p_1, p_2, p_3 和 p_4 是由参数 *params* 提供的四个值， M 是函数 **glTexGen()** 调用时的模式取景矩阵。当 M 是不充分条件矩阵或奇异矩阵，则由函数生成的纹理坐标将可能不精确或不能确定。

这里值得注意的是由参数 *params* 提供的值定义了一个眼坐标中的参考平面。应用于其上的模式取景矩阵同多边形顶点转换的矩阵在效果上是不尽相同的。这一方程提供了一种在移动的物体上生成纹理坐标的动态轮廓线的方法。

当参数 *pname* 是 **GL_SPHERE_MAP** 且参数 *coord* 是 **GL_S** 或 **GL_T** 时，纹理坐标 *s* 和 *t* 的生成方法如下。设 u 是从原点指向多边形顶点的单位向量（在眼坐标体系中），设 n 是转换成眼坐标后的当前法向量，设

$$f = (f_x \ f_y \ f_z)^T$$

为

$$f = u - 2n'n^T u$$

的转置向量，并设 $m = 2\sqrt{f_x^2 + f_y^2 + (f_z + 1)^2}$ 。则纹理坐标 *s* 和 *t* 的值为：

$$s = \frac{f_x}{m} + \frac{1}{2}$$

$$t = \frac{f_y}{m} + \frac{1}{2}$$

启动和关闭纹理坐标生成函数，可调用以一个纹理坐标符号名称（**GL_TEXTURE_GEN_S**、**GL_TEXTURE_GEN_T**、**GL_TEXTURE_GEN_R** 或 **GL_TEXTURE_GEN_Q**）为参数的函数 **glEnable()** 和 **glDisable()**。当这一功能启动后，指定的纹理坐标将由相应的生成函数计算得到。当这样功能关闭后，后面的顶点取当前纹理坐标设置的指定纹理坐标值。初始情况下，所有纹理生成函数都被设置为 **GL_EYE_LINEAR**，且纹理坐标生成函数都是关闭的。这时，*s* 平面方程为 $(1, 0, 0, 0)$ ，*t* 平面方程为 $(0, 1, 0, 0)$ ，*r* 和 *q* 平面方程均为 $(0, 0, 0, 0)$ 。

当系统支持 **GL_ARB_imaging** 扩展时，函数 **glTexEnv()** 用于设置当前有效的纹理单元的纹理生成参数，它可用函数 **glActiveTextureARB()** 选取。

- 出错提示：

当参数 *coord* 或 *pname* 不是一个已定义的可接受值时产生 **GL_INVALID_ENUM** 提示。

当参数 *pname* 为 **GL_TEXTURE_GEN_MODE**，参数 *params* 是 **GL_SPHERE_MAP**，而参数 *coord* 是 **GL_R** 或 **GL_Q** 时，将产生 **GL_INVALID_ENUM** 提示。

当函数 **glTexGen()** 在函数对 **glBegin()**/**glEnd()** 之间执行时产生 **GL_INVALID_OPERATION** 提示。

- 有关数据的获取：

glGetTexGen()

glIsEnabled(GL_TEXTURE_GEN_S)
glIsEnabled(GL_TEXTURE_GEN_T)
glIsEnabled(GL_TEXTURE_GEN_R)
glIsEnabled(GL_TEXTURE_GEN_Q)

- 请参阅：

glActiveTextureARB(), **glCopyPixels()**, **glCopyTexImage2D()**, **glCopyTexSubImage1D()**,
glCopyTexSubImage2D(), **glCopyTexSubImage3D()**, **glTexEnv()**, **glTexImage1D()**,
glTexImage2D(), **glTexImage3D()**, **glTexParameter()**, **glTexSubImage1D()**, **glTexSubImage2D()**,
glTexSubImage3D()

• **glTexImage1D**

- 名称：

glTexImage1D()

- 功能：

指定一个一维纹理图像。

- C 描述：

```
void glTexImage1D( GLenum target,
                    GLint level,
                    GLint internalformat,
                    GLsizei width,
                    GLint border,
                    GLenum format,
                    GLenum type,
                    const GLvoid *pixels )
```

- 参数说明：

target 指定目标纹理。必须是**GL_TEXTURE_1D**或**GL_PROXY_TEXTURE_1D**。

level 指定多重纹理的级别号。0 级是图像的基本级。*n* 级是指第*n* 个 mipmap 简化图像。

internalformat

指定纹理中的颜色组分数。它必须是 1、2、3、4 或 下面所列的符号常量之一：

GL_ALPHA、**GL_ALPHA4**、**GL_ALPHA8**、**GL_ALPHA12**、**GL_ALPHA16**、
GL_LUMINANCE、**GL_LUMINANCE4**、**GL_LUMINANCE8**、**GL_LUMINANCE12**、**GL_LUMINANCE16**、**GL_LUMINANCE_ALPHA**、
GL_LUMINANCE4_ALPHA4、**GL_LUMINANCE6_ALPHA2**、**GL_LUMINANCE8_ALPHA8**、**GL_LUMINANCE12_ALPHA4**、**GL_LUMINANCE12_ALPHA12**、**GL_LUMINANCE16_ALPHA16**、**GL_INTENSITY**、
GL_INTENSITY4、**GL_INTENSITY8**、**GL_INTENSITY12**、**GL_INTENSITY16**、

	GL_RGB 、 GL_R3_G3_B2 、 GL_RGB4 、 GL_RGB5 、 GL_RGB8 、 GL_RGB10 、 GL_RGB12 、 GL_RGB16 、 GL_RGBA 、 GL_RGBA2 、 GL_RGBA4 、 GL_RGB5_A1 、 GL_RGBA8 、 GL_RGB10_A2 、 GL_RGBA12 或 GL_RGBA16 。
<i>width</i>	指定纹理图像的宽度。它必须是 $2^n + 2$ (有边界时), 这里 <i>n</i> 是任一整数。任何实现所支持的纹理图像的宽度至少是64像素。1D纹理图像的高度是1。
<i>border</i>	指定边界的宽度。它必须是0或1。
<i>format</i>	指定像素数据的格式。它可取下面所列的符号型数值： GL_COLOR_INDEX 、 GL_RED 、 GL_GREEN 、 GL_BLUE 、 GL_ALPHA 、 GL_RGB 、 GL_BGR 、 GL_RGBA 、 GL_BGRA 、 GL_LUMINANCE 和 GL_LUMINANCE_ALPHA 。
<i>type</i>	指定像素数据的类型。它可取的值如下： GL_UNSIGNED_BYTE 、 GL_BYTE 、 GL_BITMAP 、 GL_UNSIGNED_SHORT 、 GL_SHORT 、 GL_UNSIGNED_INT 、 GL_INT 、 GL_FLOAT 、 GL_UNSIGNED_BYTE_3_3_2 、 GL_UNSIGNED_BYTE_2_3_3_REV 、 GL_UNSIGNED_SHORT_5_6_5 、 GL_UNSIGNED_SHORT_5_6_5_REV 、 GL_UNSIGNED_SHORT_4_4_4_4 、 GL_UNSIGNED_SHORT_4_4_4_4_REV 、 GL_UNSIGNED_SHORT_5_5_5_1 、 GL_UNSIGNED_SHORT_1_5_5_5_REV 、 GL_UNSIGNED_INT_8_8_8_8 、 GL_UNSIGNED_INT_8_8_8_8_REV 、 GL_UNSIGNED_INT_10_10_10_2 和 GL_UNSIGNED_INT_2_10_10_10_REV 。
<i>pixels</i>	指定一个指针, 指向内存中的图像数据。

• 说明:

纹理操作是把一个指定的纹理图像的一部分映射到每个允许该操作的图元上。函数**glEnable(GL_TEXTURE_1D)**和**glDisable(GL_TEXTURE_1D)**用来启动和关闭一维纹理操作。

纹理图像由函数**glTexImage1D()**定义。自变量描述了纹理图像的参数, 如宽度、边界宽度、多重纹理的级别号 (请参阅**glTexParameter()**)、储存图像用的内部分辨率和格式。最后的三个参数描述了图像在内存中的存放形式。它们与用于函数**glDrawPixels()**的像素的格式是相同的。

当参数*target*是**GL_PROXY_TEXTURE_1D**时, 将不从*pixels*中读取数据, 但所有纹理图像的状态将被重新计算, 并进行一致性检测和实现设备的性能检测。如果实现设备不能处理一个要求尺寸的纹理, 它将把所有的图像状态设为0, 但却不产生出错提示 (请参阅**glGetError()**)。如果你要查询整个mipmap数组, 请使用一个层数大于或等于1的图像数组。

当参数*target*是**GL_TEXTURE_1D**时, 从*pixels*中读取的数据将排成一个序列, 这些数据将按参数*type*的不同分为带符号或无符号的字节、短整型、长整型或浮点型数据。它们将依参数*format*的不同, 被分成单值、双值、三值或四值的数据集合, 从而构成一个元素。如果参数*type*是**GL_BITMAP**, 则数据将被看作一个无符号字节串 (这时参数*format*必须是**GL_COLOR_INDEX**)。每个字节的数据被看作包含8个二进制位的元素, 其中二进制位的排列次序由**GL_UNPACK_LSB_FIRST**确定 (请参阅**glPixelStore()**)。

第一个元素对应于纹理数组的最左端, 接下来的元素被从左到右地排入纹理序列, 最后一个元素与纹理序列的右端相对应。

参数*format*用于确定*pixels*中的每个元素的组分，它可以被设定为下面的11个符号值之一：

GL_COLOR_INDEX

每个元素是一个单一的色彩索引。GL将它转化为一种定点的格式（在二进制点的右边加上不确定个零位）。替换左边还是右边由**GL_INDEX_SHIFT**的值和符号决定。然后把它加到**GL_INDEX_OFFSET**上（请参阅glPixelTransfer()）。这样得到的索引值通过**GL_PIXEL_MAP_I_TO_R**表、**GL_PIXEL_MAP_I_TO_G**表、**GL_PIXEL_MAP_I_TO_B**表和**GL_PIXEL_MAP_I_TO_A**表转化为一个颜色组分集。其取值范围是[0,1]。

GL_RED

每个元素是一个单一的红色组分。GL将把这一组分转化成浮点值。再把绿色和蓝色组分设置为0，alpha设置为1，从而转化为一个RGBA元素。然后每一组分将被乘上带符号的缩放因子**GL_c_SCALE**，并被加到带符号的偏移量**GL_c_BIAS**上。其取值范围是[0, 1]（请参阅glPixelTransfer()）。

GL_GREEN

每个元素是一个单一的绿色组分。GL将把这一组分转化成浮点值。再把红色和蓝色组分设置为0，alpha设置为1，从而转化为一个RGBA元素。然后每一组分将被乘上带符号的缩放因子**GL_c_SCALE**，并被加到带符号的偏移量**GL_c_BIAS**上。其取值范围是[0, 1]（请参阅glPixelTransfer()）。

GL_BLUE

每个元素是一个单一的蓝色组分。GL将把这一组分转化成浮点值。再把红色和绿色组分设置为0，alpha设置为1，从而转化为一个RGBA元素。然后每一组分将被乘上带符号的缩放因子**GL_c_SCALE**，并被加到带符号的偏移量**GL_c_BIAS**上。其取值范围是[0, 1]（请参阅glPixelTransfer()）。

GL_ALPHA

每个元素是一个单一的alpha组分。GL将把这一组分转化成浮点值。再把红色、绿色和蓝色组分设置为0，从而转化为一个RGBA元素。然后每一组分将被乘上带符号的缩放因子**GL_c_SCALE**，并被加到带符号的偏移量**GL_c_BIAS**上。其取值范围是[0, 1]（请参阅glPixelTransfer()）。

GL_RGB

GL_BGR

每个元素是一个RGB三元组。GL将把它转化成浮点值。再把alpha组分设置为1，从而转化为一个RGBA元素。然后每一组分将被乘上带符号的缩放因子**GL_c_SCALE**，并被加到带符号的偏移量**GL_c_BIAS**上。其取值范围是[0, 1]（请参阅glPixelTransfer()）。

GL_RGBA

GL_BGRA

每个元素包含四个组分。然后每一组分将被乘上带符号的缩放因子**GL_c_**

SCALE, 并被加到带符号的偏移量**GL_c_BIAS**上。其取值范围是[0, 1] (请参阅glPixelTransfer())。

GL_LUMINANCE

每个元素是一个单一的亮度值。GL将把它转化成浮点值。然后把红色、绿色和蓝色组分设置为已转化的亮度值, alpha组分设置为1, 从而转化为一个RGBA元素。最后每一组分将被乘上带符号的缩放因子**GL_c_SCALE**, 并被加到带符号的偏移量**GL_c_BIAS**上。其取值范围是[0, 1] (请参阅glPixelTransfer())。

GL_LUMINANCE_ALPHA

每个像素是一个亮度/alpha对。GL将把它转化成浮点值。然后把红色、绿色和蓝色组分设置为已转化的亮度值, 从而转化为一个RGBA元素。最后每一组分将被乘上带符号的缩放因子**GL_c_SCALE**, 并被加到带符号的偏移量**GL_c_BIAS**上。其取值范围是[0, 1] (请参阅glPixelTransfer())。

如果一个应用需要将纹理存储为一种特定的分辨率或特定的格式, 它可通过参数*internal-format*来请求相应的分辨率或格式。GL将选取一种与其相接近的内部显示方式, 但它可能并不与所需要的分辨率或格式完全匹配。(由**GL_LUMINANCE**, **GL_LUMINANCE_ALPHA**, **GL_RGB**和**GL_RGBA**指定的显示方式必须完全匹配。数值1、2、3和4也可用于指定前面提到的显示方式。)

目标值**GL_PROXY_TEXTURE_1D**用来测试一种分辨率和格式。实现设备将更新并重新计算它, 从而对所要求的存储显示方式和格式进行最佳匹配。要查询这一状态, 可调用函数glGetTexLevelParameter()。如果不能适应这一纹理的要求, 纹理状态将被置0。

一个单组分纹理图像仅用到由pixels中取出的RGBA颜色中的红组分。一个双组分图像用到了R和A值。类似地, 一个三组分图像用到了R、G和B值, 一个四组分图像用到了所有的RGBA组分。

- 注意:

纹理操作在颜色索引模式下无效。

如果系统支持**GL_ARB_imaging**扩展, RGBA元素也可以被图像流程处理。在颜色组分被截断到[0, 1]范围前, 下面的状态也可以应用于一个RGBA颜色:

1. 如果允许, 颜色组分将由**GL_COLOR_TABLE**指定的颜色表替换。请参阅glColorTable()。
2. 如果允许, 用一维卷积滤波器进行过滤。请参阅glConvolutionFilter1D()。
3. 如果允许, RGBA颜色组分将被乘以**GL_POST_CONVOLUTION_c_SCALE**后加上**GL_POST_CONVOLUTION_c_BIAS**。请参阅glPixelTransfer()。
4. 如果允许, 颜色组分将由**GL_POST_CONVOLUTION_COLOR_TABLE**指定的颜色表替换。请参阅glColorTable()。
5. 通过颜色矩阵转换。请参阅glMatrixMode()。

6. 如果允许，RGBA颜色组分将被乘以**GL_POST_COLOR_MATRIX_c_SCALE**后加上**GL_POST_COLOR_MATRIX_c_BIAS**。请参阅**glPixelTransfer()**。

7. 如果允许，颜色组分将由**GL_POST_COLOR_MATRIX_COLOR_TABLE**指定的颜色表替换。请参阅**glColorTable()**。

除**GL_STENCIL_INDEX**和**GL_DEPTH_COMPONENT**之外，纹理图像能被表示为与函数**glDrawPixels()**中的像素相同的数据格式。函数**glPixelStore()**和**glPixelTransfer()**对纹理图像的影响情况同它们对函数**glDrawPixels()**的影响情况完全相同。

GL_PROXY_TEXTURE_1D只有在GL 1.1以上版本中才可以使用。

除1、2、3或4之外的其他内部格式仅在GL 1.1以上版本中可用。

在GL1.1以上的版本中，*pixels*可能是一个空指针。这时，纹理存储器用来存储一个宽度为*width*的纹理。你可以下载子纹理来初始化纹理存储器。当用户试图将一个未初始化的纹理图像的一部分应用于一个图元时，该图像将是未定义。

格式**GL_BGR**和**GL_BGRA**及类型**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**只有在GL 1.2以上的版本中才可以使用。

当系统支持**GL_ARB_multitexture**扩展时，函数**glTexImage1D()**将为当前纹理单元（由函数**glActiveTextureARB()**指定）指定一维纹理。

- 出错提示：

当参数*target*不是**GL_TEXTURE_1D**或**GL_PROXY_TEXTURE_1D**时产生**GL_INVALID_ENUM**提示。

当参数*format*不是一个可接受的格式常量时产生**GL_INVALID_ENUM**提示。除**GL_STENCIL_INDEX**和**GL_DEPTH_COMPONENT**外的其他格式常量都可以被接受。

当参数*type*不是一个类型常量时产生**GL_INVALID_ENUM**提示。

当参数*type*取**GL_BITMAP**，但参数*format*不是**GL_COLOR_INDEX**时，产生**GL_INVALID_ENUM**提示。

当参数*level*小于0时产生**GL_INVALID_VALUE**提示。

当参数*level*大于*log_nmax*时产生**GL_INVALID_VALUE**提示。此处*max*是**GL_MAX_TEXTURE_SIZE**的返回值。

当参数*internalformat*既不是1、2、3、4，又不是一个可取的分辨率和格式符号常量时，产生**GL_INVALID_VALUE**提示。

当参数*width*小于0或大于**2+GL_MAX_TEXTURE_SIZE**，或不能表示成 $2^n + 2$ （有边界时）（此处*n*是某个整数）时，产生**GL_INVALID_VALUE**提示。

当参数*border*既不是0也不是1时产生**GL_INVALID_VALUE**提示。

当函数**glTexImage1D()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_**

OPERATION提示。

当参数*type*为**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**之一，但参数*format*却不是**GL_RGB**格式时，产生**GL_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_SHORT_4_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**之一，但参数*format*既不是**GL_RGBA**格式又不是**GL_BGRA**格式时，产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGetTexImage()

glIsEnabled(GL_TEXTURE_ID)

- 请参阅：

glActiveTextureARB(), **glColorTable()**, **glConvolutionFilter1D()**, **glCopyPixels()**,
glCopyTexImage1D(), **glCopyTexImage2D()**, **glCopyTexSubImage1D()**, **glCopyTexSubImage2D()**,
glCopyTexSubImage3D(), **glDrawPixels()**, **glMatrixMode()**, **glPixelStore()**,
glPixelTransfer(), **glTexEnv()**, **glTenGen()**, **glTexImage2D()**, **glTexImage3D()**,
glTexSubImage1D(), **glTexSubImage2D()**, **glTexSubImage3D()**, **glTexParameter()**

:glTexImage2D

- 名称：

glTexImage2D()

- 功能：

指定一个二维纹理图像。

- C 描述：

```
void glTexImage2D( GLenum target,
                   GLint level,
                   GLint internalformat,
                   GLsizei width,
                   GLsizei height,
                   GLint border,
                   GLenum format,
                   GLenum type,
                   const GLvoid *pixels )
```

- 参数说明：

target 指定目标纹理。必须是**GL_TEXTURE_2D**或**GL_PROXY_TEXTURE_2D**。

level 指定多重纹理的级别号。0级是图像的基本级。*n*级是指第*n*个mipmap的简化图像。

internalformat

指定纹理中的颜色组分数。它必须是1、2、3、4或下面所列的符号常量之一：**GL_ALPHA**、**GL_ALPHA4**、**GL_ALPHA8**、**GL_ALPHA12**、**GL_ALPHA16**、**GL_LUMINANCE**、**GL_LUMINANCE4**、**GL_LUMINANCE8**、**GL_LUMINANCE12**、**GL_LUMINANCE16**、**GL_LUMINANCE_ALPHA**、**GL_LUMINANCE4_ALPHA4**、**GL_LUMINANCE6_ALPHA2**、**GL_LUMINANCE8_ALPHA8**、**GL_LUMINANCE12_ALPHA4**、**GL_LUMINANCE12_ALPHA12**、**GL_LUMINANCE16_ALPHA16**、**GL_INTENSITY**、**GL_INTENSITY4**、**GL_INTENSITY8**、**GL_INTENSITY12**、**GL_INTENSITY16**、**GL_R3_G3_B2**、**GL_RGB**、**GL_RGB4**、**GL_RGB5**、**GL_RGB8**、**GL_RGB10**、**GL_RGB12**、**GL_RGB16**、**GL_RGBA**、**GL_RGBA2**、**GL_RGBA4**、**GL_RGB5_A1**、**GL_RGBA8**、**GL_RGB10_A2**、**GL_RGBA12**或**GL_RGBA16**。

width 指定纹理图像的宽度。它必须是 2^n+2 （有边界时），此处n是某个整数。任何实现设备所支持的纹理图像的宽度至少是64像素。

height 指定纹理图像的高度。它必须是 2^m+2 （有边界时），此处m是某个整数。任何实现设备所支持的纹理图像的高度至少是64像素。

border 指定边界的宽度。它必须是0或1。

format 指定像素数据的格式。它可取下面所列的符号型数值：**GL_COLOR_INDEX**、**GL_RED**、**GL_GREEN**、**GL_BLUE**、**GL_ALPHA**、**GL_RGB**、**GL_BGR**、**GL_RGBA**、**GL_BGRA**、**GL_LUMINANCE**和**GL_LUMINANCE_ALPHA**。

type 指定像素数据的类型。它可取的值如下：**GL_UNSIGNED_BYTE**、**GL_BYTE**、**GL_BITMAP**、**GL_UNSIGNED_SHORT**、**GL_SHORT**、**GL_UNSIGNED_INT**、**GL_INT**、**GL_FLOAT**、**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**。

pixels 指定一个指针，指向内存中的图像数据。

• 说明：

纹理操作是把一个指定的纹理图像的一部分映射到每个允许该操作的图元上。函数**glEnable(GL_TEXTURE_2D)**和**glDisable(GL_TEXTURE_2D)**用来启动和关闭二维纹理操作。

纹理图像由函数**glTexImage2D()**定义。自变量描述了纹理图像的参数，如高度、宽度、边界宽度、多重纹理的级别号（请参阅**glTexParameter()**）和提供的颜色组分的数目。最后的三个参数描述了图像在内存中的存放形式。它们与用于函数**glDrawPixels()**的像素的格式是相同的。

如果参数*target*是**GL_PROXY_TEXTURE_2D**, 将不从*pixels*中读取数据, 但所有纹理图像的状态将被重新计算, 并进行一致性检测和实现设备的性能检测。如果实现设备不能处理一个要求尺寸的纹理, 它将把所有的图像状态设为0, 但却不产生出错提示(请参阅glGetError())。可用一个层数大于或等于1的图像数组来查询整个mipmap数组。

当参数*target*是**GL_TEXTURE_2D**时, 从*pixels*中读取的数据将排成一个序列, 这些数据将按参数*type*的不同分为带符号或无符号的字节、短整型、长整型或浮点型数据。它们将依参数*format*的不同, 被分成单值、双值、三值或四值的数据集合, 从而构成一个元素。如果参数*type*是**GL_BITMAP**, 则数据将被看作一个无符号字节串(这时参数*format*必须是**GL_COLOR_INDEX**)。每个字节的数据被看作包含8个二进制位的元素, 其中二进制位的排列次序由**GL_UNPACK_LSB_FIRST**确定(请参阅glPixelStore())。

第一个元素对应于纹理图像的左下角, 接下来的元素被从左到右、从低到高地排入纹理图像中, 最后一个元素与纹理图像的右上角相对应。

参数*format*用于确定*pixels*中的每个元素的组分, 它可以被设定为下面的11个符号值之一:

GL_COLOR_INDEX

每个元素是一个单一的色彩索引。GL将它转化为一种定点的格式(在二进制点的右边加上不确定个零位)。替换左边还是右边由**GL_INDEX_SHIFT**的值和符号决定。然后把它加到**GL_INDEX_OFFSET**上(请参阅glPixelTransfer())。这样得到的索引值通过**GL_PIXEL_MAP_I_TO_R**表、**GL_PIXEL_MAP_I_TO_G**表、**GL_PIXEL_MAP_I_TO_B**表和**GL_PIXEL_MAP_I_TO_A**表转化为一个颜色组分集。其取值范围是[0, 1]。

GL_RED

每个元素是一个单一的红色组分。GL将把这一组分转化成浮点值。再把绿色和蓝色组分设置为0, alpha设置为1, 从而转化为一个RGBA元素。然后每一组分将被乘上带符号的缩放因子**GL_c_SCALE**, 并被加到带符号的偏移量**GL_c_BIAS**上。其取值范围是[0, 1](请参阅glPixelTransfer())。

GL_GREEN

每个元素是一个单一的绿色组分。GL将把这一组分转化成浮点值。再把红色和蓝色组分设置为0, alpha设置为1, 从而转化为一个RGBA元素。然后每一组分将被乘上带符号的缩放因子**GL_c_SCALE**, 并被加到带符号的偏移量**GL_c_BIAS**上。其取值范围是[0, 1](请参阅glPixelTransfer())。

GL_BLUE

每个元素是一个单一的蓝色组分。GL将把这一组分转化成浮点值。再把红色和绿色组分设置为0, alpha设置为1, 从而转化为一个RGBA元素。然后每一组分将被乘上带符号的缩放因子**GL_c_SCALE**, 并被加到带符号的偏移量**GL_c_BIAS**上。其取值范围是[0, 1](请参阅glPixelTransfer())。

GL_ALPHA

每个元素是一个单一的alpha组分。GL将把这一组分转化成浮点值。再把红色、

绿色和蓝色组分设置为0，从而转化为一个RGBA元素。然后每一组分将被乘上带符号的缩放因子**GL_c_SCALE**，并被加到带符号的偏移量**GL_c_BIAS**上。其取值范围是[0, 1]（请参阅glPixelTransfer()）。

GL_RGB

GL_BGR

每个元素是一个RGB三元组。GL将把它转化成浮点值。再把alpha组分设置为1，从而转化为一个RGBA元素。然后每一组分将被乘上带符号的缩放因子**GL_c_SCALE**，并被加到带符号的偏移量**GL_c_BIAS**上。其取值范围是[0, 1]（请参阅glPixelTransfer()）。

GL_RGBA

GL_BGRA

每个元素包含四个组分。然后每一组分将被乘上带符号的缩放因子**GL_c_SCALE**，并被加到带符号的偏移量**GL_c_BIAS**上。其取值范围是[0, 1]（请参阅glPixelTransfer()）。

GL_LUMINANCE

每个元素是一个单一的亮度值。GL将把它转化成浮点值。然后把红色、绿色和蓝色组分设置为已转化的亮度值，alpha组分设置为1，从而转化为一个RGBA元素。最后每一组分将被乘上带符号的缩放因子**GL_c_SCALE**，并被加到带符号的偏移量**GL_c_BIAS**上。其取值范围是[0, 1]（请参阅glPixelTransfer()）。

GL_LUMINANCE_ALPHA

每个像素是一个亮度/alpha对。GL将把它转化成浮点值。然后把红色、绿色和蓝色组分设置为已转化的亮度值，从而转化为一个RGBA元素。最后每一组分将被乘上带符号的缩放因子**GL_c_SCALE**，并被加到带符号的偏移量**GL_c_BIAS**上。其取值范围是[0, 1]（请参阅glPixelTransfer()）。

有关参数*type*的取值情况，请参阅函数glDrawPixels()。

如果一个应用需要将纹理存储为一种特定的分辨率或特定的格式，它可通过参数*internal format* 来请求相应的分辨率或格式。GL将选取一种与其相接近的内部显示方式，但它可能并不与所需要的分辨率或格式完全匹配。（由**GL_LUMINANCE**, **GL_LUMINANCE_ALPHA**, **GL_RGB**和**GL_RGBA**指定的显示方式必须完全匹配。数值1、2、3和4也可用于指定前面提到的显示方式。）

目标值**GL_PROXY_TEXTURE_2D**用于测试一种分辨率和格式。实现设备将更新并重新计算它，从而对所要求的存储显示方式和格式进行最佳匹配。要查询这一状态，可调用函数glGetTexParameter()。如果不能适应这一纹理的要求，纹理状态将被置0。

一个单纹理图像仅用到由*pixels*中取出的RGBA颜色中的红组分。一个双组分图像用到了R和A值。类似地，一个三组分图像用到了R、G和B值，一个四组分图像用到了所有的RGBA组分。

- 注意：

纹理操作在颜色索引模式下无效。

如果系统支持**GL_ARB_imaging**扩展，RGBA元素也可以被图像流程处理。在颜色组分被截断到[0, 1]范围前，下面的状态也可以应用于一个RGBA颜色：

1. 如果允许，颜色组分将由**GL_COLOR_TABLE**指定的颜色表替换。请参阅**glColorTable()**。
2. 如果允许，用二维卷积滤波器进行过滤。请参阅**glConvolutionFilter2D()**。

如果一个卷积滤波器改变了纹理的宽度（例如通过**GL_REDUCE**的**GL_CONVOLUTION_BORDER_MODE**处理），过滤后的*width*必须是 2^n+2 （当有高度时），此处*n*是某个整数。同样地，过滤后的*height*必须是 2^m+2 （当有边界时），此处*m*是某个整数。

3. 如果允许，RGBA颜色组分将被乘以**GL_POST_CONVOLUTION_c_SCALE**后加上**GL_POST_CONVOLUTION_c_BIAS**。请参阅**glPixelTransfer()**。

4. 如果允许，颜色组分将由**GL_POST_CONVOLUTION_COLOR_TABLE**指定的颜色表替换。请参阅**glColorTable()**。

5. 通过颜色矩阵转换。请参阅**glMatrixMode()**。

6. 如果允许，RGBA颜色组分将被乘以**GL_POST_COLOR_MATRIX_c_SCALE**后加上**GL_POST_COLOR_MATRIX_c_BIAS**。请参阅**glPixelTransfer()**。

7. 如果允许，颜色组分将由**GL_POST_COLOR_MATRIX_COLOR_TABLE**指定的颜色表替换。请参阅**glColorTable()**。

除**GL_STENCIL_INDEX**和**GL_DEPTH_COMPONENT**之外，纹理图像能被表示为与函数**glDrawPixels()**中的像素相同的数据格式。函数**glPixelStore()**和**glPixelTransfer()**对纹理图像的影响情况同它们对函数**glDrawPixels()**的影响情况完全相同。

glTexImage2D()和**GL_PROXY_TEXTURE_2D**仅在GL 1.1以上版本中可用。

除1、2、3或4外的其他内部格式仅在GL 1.1以上版本中可用。

在GL 1.1以上版本中，*pixels*可以是一个空指针。这时，纹理存储器用来存储一个宽度为*width*、高度为*height*的纹理。你可以通过下载子纹理来初始化纹理存储器。当用户试图将一个未初始化的纹理图像的一部分应用于一个图元时，该图像将是未定义的。

格式**GL_BGR**和**GL_BGRA**及类型**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**仅在GL 1.2以上的版本中才可用。

当系统支持**GL_ARB_multitexture**扩展时，函数**glTexImage2D()**将为当前纹理单元（由函数**glActiveTextureARB()**指定）指定二维纹理。

- 出错提示：

当参数*target*不是**GL_TEXTURE_2D**或**GL_PROXY_TEXTURE_2D**时产生**GL_INVALID_ENUM**提示。

当参数*format*不是一个可接受的格式常量时产生**GL_INVALID_ENUM**提示。除**GL_STENCIL_INDEX**和**GL_DEPTH_COMPONENT**外的其他格式常量都可以被接受。

当参数*type*不是一个类型常量时产生**GL_INVALID_ENUM**提示。

当参数*type*取**GL_BITMAP**, 但参数*format*不是**GL_COLOR_INDEX**时, 产生**GL_INVALID_ENUM**提示。

当参数*level*小于0时产生**GL_INVALID_VALUE**提示。

当参数*level*大于log₂*max*时产生**GL_INVALID_VALUE**提示。此处*max*是**GL_MAX_TEXTURE_SIZE**的返回值。

当参数*internalformat*既不是1、2、3、4, 又不是一个可取的分辨率和格式符号常量时, 产生**GL_INVALID_VALUE**提示。

当参数*width*或*height*小于0或大于2+**GL_MAX_TEXTURE_SIZE**, 或不能表示成 $2^k + 2$ (有边界时) (此处*k*是某个整数) 时, 产生**GL_INVALID_VALUE**提示。

当参数*border*既不是0也不是1时产生**GL_INVALID_VALUE**提示。

当函数**glTexImage2D()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BVTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**之一, 但参数*format*却不是**GL_RGB**格式时, 产生**GL_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_SHORT_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**之一, 但参数*format*既不是**GL_RGBA**格式又不是**GL_BGRA**格式时, 产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取:

glGetTexImage()

glIsEnabled(GL_TEXTURE_2D)

- 请参阅:

glColorTable(), **glConvolutionFilter2D()**, **glCopyPixels()**, **glCopyTexImage1D()**, **glCopyTexImage2D()**, **glCopyTexSubImage1D()**, **glCopyTexSubImage2D()**, **glCopyTexSubImage3D()**, **glDrawPixels()**, **glMatrixMode()**, **glPixelStore()**, **glPixelTransfer()**, **glSeparableFilter2D()**, **glTexEnv()**, **glTexGen()**, **glTexImage1D()**, **glTexImage3D()**, **glTexSubImage1D()**, **glTexSubImage2D()**, **glTexSubImage3D()**, **glTexParameter()**

* **glTexImage3D**

- 名称:

glTexImage3D()

- 功能:

指定一个三维纹理图像。

- C描述:

```
void glTexImage3D( GLenum target,
                   GLint level,
                   GLint internalformat,
                   GLsizei width,
                   GLsizei height,
                   GLsizei depth,
                   GLint border,
                   GLenum format,
                   GLenum type,
                   const GLvoid *pixels )
```

• 参数说明：

target 指定目标纹理。必须是**GL_TEXTURE_3D**或**GL_PROXY_TEXTURE_3D**。

level 指定多重纹理的级别号。0级是图像的基本级。*n*级是指第*n*个mipmap的简化图像。

internalformat

指定纹理中的颜色组分数。它必须是1, 2, 3, 4或下面所列的符号常量之一：

GL_ALPHA、**GL_ALPHA4**、**GL_ALPHA8**、**GL_ALPHA12**、**GL_ALPHA16**、**GL_LUMINANCE**、**GL_LUMINANCE4**、**GL_LUMINANCE8**、**GL_LUMINANCE12**、**GL_LUMINANCE16**、**GL_LUMINANCE_ALPHA**、**GL_LUMINANCE4_ALPHA4**、**GL_LUMINANCE6_ALPHA2**、**GL_LUMINANCE8_ALPHA8**、**GL_LUMINANCE12_ALPHA4**、**GL_LUMINANCE12_ALPHA12**、**GL_LUMINANCE16_ALPHA16**、**GL_INTENSITY**、**GL_INTENSITY4**、**GL_INTENSITY8**、**GL_INTENSITY12**、**GL_INTENSITY16**、**GL_R3_G3_B2**、**GL_RGB**、**GL_RGB4**、**GL_RGB5**、**GL_RGB8**、**GL_RGB10**、**GL_RGB12**、**GL_RGB16**、**GL_RGBA**、**GL_RGBA2**、**GL_RGBA4**、**GL_RGB5_A1**、**GL_RGBA8**、**GL_RGB10_A2**、**GL_RGBA12**或**GL_RGBA16**。

width 指定纹理图像的宽度。它必须是 2^n+2 （有边界时），此处*n*是某个整数。任何实现设备所支持的纹理图像的宽度至少是64像素。

height 指定纹理图像的高度。它必须是 2^m+2 （有边界时），此处*m*是某个整数。任何实现设备所支持的纹理图像的高度至少是64像素。

depth 指定纹理图像的深度。它必须是 2^k+2 （有边界时），此处*k*是某个整数。任何实现设备所支持的纹理图像的深度至少是64像素。

border 指定边界的宽度。它必须是0或1。

format 指定像素数据的格式。它可取下面所列的符号型数值：**GL_COLOR_INDEX**、**GL_RED**、**GL_GREEN**、**GL_BLUE**、**GL_ALPHA**、**GL_RGB**、**GL_BGR**、**GL_RGBA**、**GL_BGRA**、**GL_LUMINANCE**和**GL_LUMINANCE_ALPHA**。

type 指定像素数据的类型。它可取的值如下：**GL_UNSIGNED_BYTE**、**GL_BYTE**、**GL_BITMAP**、**GL_UNSIGNED_SHORT**、**GL_SHORT**、**GL_UNSIGNED_INT**、**GL_INT**、**GL_FLOAT**、**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**。

pixels 指定一个指针，指向内存中的图像数据。

- 说明：

纹理操作是把一个指定的纹理图像的一部分映射到每个允许该操作的图元上。函数**glEnable(GL_TEXTURE_3D)**和**glDisable(GL_TEXTURE_3D)**用来启动和关闭三维纹理操作。

纹理图像由函数**glTexImage3D()**定义。自变量描述了纹理图像的参数，如高度、宽度、深度、边界宽度、多重纹理的级别号（请参阅**glTexParameter()**）和提供的颜色组分的数目。最后的三个参数描述了图像在内存中的存放形式。它们与用于函数**glDrawPixels()**的像素的格式是相同的。

如果参数*target*是**GL_PROXY_TEXTURE_3D**，将不从*pixels*中读取数据，但所有纹理图像的状态将被重新计算，并进行一致性检测和实现设备的性能检测。如果实现设备不能处理一个要求尺寸的纹理，它将把所有的图像状态设为0，但却不产生出错提示（请参阅**glGetError()**）。可用一个层数大于或等于1的图像数组来查询整个mipmap数组。

当参数*target*是**GL_TEXTURE_3D**时，从*pixels*中读取的数据将排成一个序列，这些数据将按参数*type*的不同分为带符号或无符号的字节、短整型、长整型或浮点型数据。它们将依参数*format*的不同，被分成单值、双值、三值或四值的数据集合，从而构成一个元素。如果参数*type*是**GL_BITMAP**，则数据将被看作一个无符号字节串（这时参数*format*必须是**GL_COLOR_INDEX**）。每个字节的数据被看作包含8个二进制位的元素，其中二进制位的排列次序由**GL_UNPACK_LSB_FIRST**确定（请参阅**glPixelStore()**）。

第一个元素对应于纹理图像的左下角，接下来的元素被从左到右、从低到高地排入纹理图像中，最后一个元素与纹理图像的右上角相对应。

参数*format*用于确定*pixels*中的每个元素的组分，它可以被设定为下面的11个符号值之一：

GL_COLOR_INDEX

每个元素是一个单一的色彩索引。GL将它转化为一种定点的格式（在二进制点的右边加上不确定个零位）。替换左边还是右边由**GL_INDEX_SHIFT**的值和符号决定。然后把它加到**GL_INDEX_OFFSET**上（请参阅**glPixelTransfer()**）。这样得到的索引值通过**GL_PIXEL_MAP_I_TO_R表**、**GL_PIXEL_MAP_I_TO_G表**、**GL_PIXEL_MAP_I_TO_B表**和**GL_PIXEL_MAP_I_TO_A表**转化为一个颜色组

分集。其取值范围是[0, 1]。

GL_RED

每个元素是一个单一的红色组分。GL将把这一组分转化成浮点值。再把绿色和蓝色组分设置为0, alpha设置为1, 从而转化为一个RGBA元素。然后每一组分将被乘上带符号的缩放因子**GL_c_SCALE**, 并被加到带符号的偏移量**GL_c_BIAS**上。其取值范围是[0, 1]内 (请参阅glPixelTransfer())。

GL_GREEN

每个元素是一个单一的绿色组分。GL将把这一组分转化成浮点值。再把红色和蓝色组分设置为0, alpha设置为1, 从而转化为一个RGBA元素。然后每一组分将被乘上带符号的缩放因子**GL_c_SCALE**, 并被加到带符号的偏移量**GL_c_BIAS**上。其取值范围是[0, 1] (请参阅glPixelTransfer())。

GL_BLUE

每个元素是一个单一的蓝色组分。GL将把这一组分转化成浮点值。再把红色和绿色组分设置为0, alpha设置为1, 从而转化为一个RGBA元素。然后每一组分将被乘上带符号的缩放因子**GL_c_SCALE**, 并被加到带符号的偏移量**GL_c_BIAS**上。其取值范围是[0, 1] (请参阅glPixelTransfer())。

GL_ALPHA

每个元素是一个单一的alpha组分。GL将把这一组分转化成浮点值。再把红色、绿色和蓝色组分设置为0, 从而转化为一个RGBA像素。然后每一组分将被乘上带符号的缩放因子**GL_c_SCALE**, 并被加到带符号的偏移量**GL_c_BIAS**上。其取值范围是[0, 1] (请参阅glPixelTransfer())。

GL_RGB

GL_BGR

每个元素是一个RGB三元组。GL将把它转化成浮点值。再把alpha组分设置为1, 从而转化为一个RGBA元素。然后每一组分将被乘上带符号的缩放因子**GL_c_SCALE**, 并被加到带符号的偏移量**GL_c_BIAS**上。其取值范围是[0, 1] (请参阅glPixelTransfer())。

GL_RGBA

GL_BGRA

每个元素包含四个组分。然后每一组分将被乘上带符号的缩放因子**GL_c_SCALE**, 并被加到带符号的偏移量**GL_c_BIAS**上。其取值范围是[0, 1] (请参阅glPixelTransfer())。

GL_LUMINANCE

每个元素是一个单一的亮度值。GL将把它转化成浮点值。然后把红色、绿色和蓝色组分设置为已转化的亮度值, alpha组分设置为1, 从而转化为一个RGBA元素。最后每一组分将被乘上带符号的缩放因子**GL_c_SCALE**, 并被加到带符号的偏移量**GL_c_BIAS**上。其取值范围是[0, 1] (请参阅glPixelTransfer())。

GL_LUMINANCE_ALPHA

每个像素是一个亮度/alpha对。GL将把它转化成浮点值。然后把红色、绿色和蓝色组分设置为已转化的亮度值，从而转化为一个RGBA元素。最后每一组分将被乘上带符号的缩放因子**GL_c_SCALE**，并被加到带符号的偏移量**GL_c_BIAS**上。其取值范围是[0, 1]（请参阅**glPixelTransfer()**）。

有关参数*type*的取值情况，请参阅函数**glDrawPixels()**。

如果一个应用需要将纹理存储为一种特定的分辨率或特定的格式，它可通过参数*internalformat*来请求相应的分辨率或格式。GL将选取一种与其相接近的内部显示方式，但它可能并不与所需要的分辨率或格式完全匹配。（由**GL_LUMINANCE**, **GL_LUMINANCE_ALPHA**, **GL_RGB**和**GL_RGBA**指定的显示方式必须完全匹配。数值1、2、3和4也可用于指定前面提到的显示方式。）

目标值**GL_PROXY_TEXTURE_3D**用于测试一种分辨率和格式。实现设备将更新并重新计算它，从面对所要求存储的分辨率和格式进行最佳匹配。要查询这一状态，可调用函数**glGetTexLevelParameter()**。如果不能适应这一纹理的要求，纹理状态将被置0。

一个单纹理图像仅用到由*pixels*中取出的RGBA颜色中的红组分。一个双组分图像用到了R和A值。类似地，一个三组分图像用到了R、G和B值，一个四组分图像用到了所有的RGBA组分。

• 注意：

纹理操作在颜色索引模式下无效。

除**GL_STENCIL_INDEX**和**GL_DEPTH_COMPONENT**之外，纹理图像能被表示为与函数**glDrawPixels()**中的像素相同的数据格式。函数**glPixelStore()**和**glPixelTransfer()**对纹理图像的影响情况同它们对函数**glDrawPixels()**的影响情况完全相同。

glTexImage3D()只有在GL1.2以上的版本中才可以使用。

除1、2、3或4外的其他内部格式仅在GL1.1以上版本中可用。

参数*pixels*可以是一个空指针。这时，纹理存储器用来存储一个宽度为*width*、高度为*height*、深度为*depth*的纹理。你可以通过下载子纹理来初始化纹理存储器。当用户试图将一个未初始化的纹理图像的一部分应用于一个图元时，该图像将是未定义的。

格式**GL_BGR**和**GL_BGRA**及类型**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**仅在GL1.2以上的版本中才可用。

当系统支持**GL_ARB_multitexture**扩展时，函数**glTexImage3D()**将为当前纹理单元（由函数**glActiveTextureARB()**指定）指定三维纹理。

如果系统支持**GL_ARB_imaging**扩展，RGBA元素也可以被图像流程处理。在颜色组分被截断到[0, 1]范围前，下面的状态也可以应用于一个RGBA颜色：

1. 如果允许，颜色组分将由**GL_COLOR_TABLE**指定的颜色表替换。请参阅**glColor-**

Table()。

2. 如果允许，颜色组分将由**GL_POST_CONVOLUTION_COLOR_TABLE**指定的颜色表替换。请参阅**glColorTable()**。

3. 通过颜色矩阵转换。请参阅**glMatrixMode()**。

4. 如果允许，**RGBA**颜色组分将乘以**GL_POST_COLOR_MATRIX_c_SCALE**后加上**GL_POST_COLOR_MATRIX_c_BIAS**。请参阅**glPixelTransfer()**。

5. 如果允许，颜色组分将由**GL_POST_COLOR_MATRIX_COLOR_TABLE**指定的颜色表替换。请参阅**glColorTable()**。

- 出错提示：

当参数*target*不是**GL_TEXTURE_3D**或**GL_PROXY_TEXTURE_3D**时产生**GL_INVALID_ENUM**提示。

当参数*format*不是一个可接受的格式常量时产生**GL_INVALID_ENUM**提示。除**GL_STENCIL_INDEX**和**GL_DEPTH_COMPONENT**外的其他格式常量都可以被接受。

当参数*type*不是一个类型常量时产生**GL_INVALID_ENUM**提示。

当参数*type*取**GL_BITMAP**，但参数*format*不是**GL_COLOR_INDEX**时，产生**GL_INVALID_ENUM**提示。

当参数*level*小于0时产生**GL_INVALID_VALUE**提示。

当参数*level*大于 $\log_2 max$ 时产生**GL_INVALID_VALUE**提示。此处*max*是**GL_MAX_TEXTURE_SIZE**的返回值。

当参数*internalformat*既不是1、2、3、4，又不是一个可接受的分辨率和格式符号常量时，产生**GL_INVALID_VALUE**提示。

当参数*width*、*height*或*depth*小于0或大于 $2 + GL_MAX_TEXTURE_SIZE$ ，或不能表示成 $2^k + 2$ （有边界时）（此处*k*是某个整数）时，产生**GL_INVALID_VALUE**提示。

当参数*border*既不是0也不是1时产生**GL_INVALID_VALUE**提示。

当函数**glTexImage3D()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**之一，但参数*format*却不是**GL_RGB**格式时产生**GL_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_SHORT_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**之一，但参数*format*既不是**GL_RGBA**格式又不是**GL_BGRA**格式时，产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGetTexImage()

glIsEnabled(GL_TEXTURE_3D)

- 请参阅：

glActiveTextureARB(), **glCopyPixels()**, **glCopyTexImage1D()**, **glCopyTexImage2D()**,
glCopyTexSubImage1D(), **glCopyTexSubImage2D()**, **glCopyTexSubImage3D()**,
glDrawPixels(), **glPixelStore()**, **glPixelTransfer()**, **glTexEnv()**, **glTexGen()**, **glTexImage1D()**,
glTexImage2D(), **glTexSubImage1D()**, **glTexSubImage2D()**, **glTexSubImage3D()**,
glTexParameter()

- **glTexParameter**

- 名称：

glTexParameterf(), **glTexParameteri()**, **glTexParameterfv()**, **glTexParameteriv()**

- 功能：

设置纹理参数。

- C描述：

```
void glTexParameterf( GLenum target,
                      GLenum pname,
                      GLfloat param )
void glTexParameteri( GLenum target,
                      GLenum pname,
                      GLint param )
```

- 参数说明：

target 指定目标纹理。必须是**GL_TEXTURE_1D**、**GL_TEXTURE_2D**或**GL_TEXTURE_3D**。

pname 指定一个单值纹理参数的符号名称。它可取的值如下：**GL_TEXTURE_MIN_FILTER**、**GL_TEXTURE_MAG_FILTER**、**GL_TEXTURE_MIN_LOD**、**GL_TEXTURE_MAX_LOD**、**GL_TEXTURE_BASE_LEVEL**、**GL_TEXTURE_MAX_LEVEL**、**GL_TEXTURE_WRAP_S**、**GL_TEXTURE_WRAP_T**、**GL_TEXTURE_WRAP_R**或**GL_TEXTURE_PRIORITY**。

param 指定参数*pname*的具体值。

- C描述：

```
void glTexParameterfv( GLenum target,
                      GLenum pname,
                      const GLfloat *params )
void glTexParameteriv( GLenum target,
                      GLenum pname,
                      const GLint *params )
```

- 参数说明：

target 指定目标纹理。必须是**GL_TEXTURE_1D**、**GL_TEXTURE_2D**或**GL_TEXTURE_3D**。

URE _3D。

pname 指定一个单值纹理参数的符号名。它可取的值如下：**GL_TEXTURE_MIN_FILTER**、**GL_TEXTURE_MAG_FILTER**、**GL_TEXTURE_MIN_LOD**、**GL_TEXTURE_MAX_LOD**、**GL_TEXTURE_BASE_LEVEL**、**GL_TEXTURE_MAX_LEVEL**、**GL_TEXTURE_WRAP_S**、**GL_TEXTURE_WRAP_T**、**GL_TEXTURE_WRAP_R**、**GL_TEXTURE_BORDER_COLOR**或**GL_TEXTURE_PRIORITY**。

params 指定一个指针，指向存放参数*pname*值的数组。

- 说明：

纹理映射是一种将图像应用于物体表面的技术，就像该图像是一种贴画纸或玻璃纸附着于物体的表面上。图像是在纹理空间中用(*s*, *t*)坐标体系创建的。一个纹理就是一个一维或二维的图像，并带有一组参数用以确定从图像中采样的方式。

函数glTexParameter()将某个目标纹理的纹理参数值赋给了*params*，这里所指的目标纹理是由参数*pname*和*target*定义的**GL_TEXTURE_1D**、**GL_TEXTURE_2D**和**GL_TEXTURE_3D**。下面列出了参数*pname*的取值：

GL_TEXTURE_MIN_FILTER

纹理缩小函数。当像素被纹理化并映射到大于一个纹理元素的区域时，将使用该纹理缩小函数。有六种定义的缩小函数。其中两种用最近的一个或四个纹理元素来计算纹理值。另外的四种用mipmap。

一个mipmap就是用一系列有序的数组按由低到高的分辨率来表示相同的图像。如果一个纹理的尺寸是 $2^n \times 2^m$ ，这时共有 $\max(n, m)+1$ 个mipmap。第一个mipmap是原始纹理，它的尺寸就是 $2^n \times 2^m$ 。接下来的每个mipmap的尺寸是 $2^{k-1} \times 2^{l-1}$ ，这里的 $2^k \times 2^l$ 是前一个mipmap的尺寸，直至 $k=0$ 或 $l=0$ 。这时，接下来的mipmap的尺寸将是 $1 \times 2^{l-1}$ 或 $2^{k-1} \times 1$ 直至最后一个mipmap，其尺寸将是 1×1 。要定义一个mipmap，可以用函数glTexImage1D()、glTexImage2D()、glTexImage3D()、glCopyTexImage1D()或glCopyTexImage2D()，同时这些函数应该带有自变量*level*来指定mipmaps的级别。当*level*取0时，对应于原始纹理；当它取 $\max(n, m)$ 时，对应于最后的 1×1 mipmap。

参数*params*提供的缩小纹理函数如下：

GL_NEAREST

返回与纹理化的像素中点距离(Manhattan 距离)最近的纹理元素的值。

GL_LINEAR

返回最靠近纹理化像素中点的四个纹理元素加权平均值。这些值包括边界上的纹理元素，具体由值**GL_TEXTURE_WRAP_S**和**GL_TEXTURE_WRAP_T**及是否精确映射而确定。

GL_NEAREST_MIPMAP_NEAREST

选择一个与纹理化的像素在尺寸上最匹配的mipmap，并用**GL_NEAREST**标准(即最靠近像素中点的纹理元素)产生一个纹理值。

GL_LINEAR_MIPMAP_NEAREST

选择一个与纹理化的像素在尺寸上最匹配的mipmap，并用**GL_LINEAR**标准（即最靠近纹理化像素中点的四个纹理元素加权平均值）产生一个纹理值。

GL_NEAREST_MIPMAP_LINEAR

选择两个与纹理化的像素在尺寸上最匹配的mipmap，并用**GL_NEAREST**标准（即最靠近像素中点的纹理元素）从每个mipmap中产生一个纹理值。最后的纹理值是这两个值的加权平均值。

GL_LINEAR_MIPMAP_LINEAR

选择两个与纹理化的像素在尺寸上最匹配的mipmap，并用**GL_LINEAR**标准（即最靠近纹理化像素中点的四个纹理元素加权平均值）从每个mipmap中产生一个纹理值。最后的纹理值是这两个值的加权平均值。

由于在缩小过程中有多个元素被采样，因此可能会出现个别混淆的情况。当然，**GL_NEAREST**和**GL_LINEAR**缩小方式的速度要比其他方式快一些，因为它们仅需采样一个和四个纹理元素就可以确定要绘制的像素的纹理值，但同时也产生了波动图案和不规则的转化。**GL_TEXTURE_MIN_FILTER**的缺省值是**GL_NEAREST_MIPMAP_LINEAR**。

GL_TEXTURE_MAG_FILTER

纹理放大函数。当像素被纹理化并映射到小于或等于一个纹理元素的区域时，将用到该纹理放大函数。纹理放大函数可以被设置成**GL_NEAREST**或**GL_LINEAR**（如下所示）。方式**GL_NEAREST**的生成速度要比**GL_LINEAR**方式快，但它将使纹理图像的边缘尖锐化，这是由于在两个纹理元素之间的转化不光滑所致。**GL_TEXTURE_MAG_FILTER**的缺省值是**GL_LINEAR**。

GL_NEAREST

返回与纹理化像素中点距离（Manhattan 距离）最近的纹理元素的值。

GL_LINEAR

返回最靠近纹理化像素中点的四个纹理元素加权平均值。这些值包括边界上的纹理元素，具体由值**GL_TEXTURE_WRAP_S**和**GL_TEXTURE_WRAP_T**及是否精确映射而确定。

GL_TEXTURE_MIN_LOD

设置最小的多重纹理级别参数。这一浮点值限定了mipmap的最高可选分辨率（最低的mipmap级）。其缺省值是-1000。

GL_TEXTURE_MAX_LOD

设置最大的多重纹理级别参数。这一浮点值限定了mipmap的最低可选分辨率（最高的mipmap级）。其缺省值是1000。

GL_TEXTURE_BASE_LEVEL

指定一个最低定义的mipmap级别的索引。它是一个整型数据。其缺省值是0。

GL_TEXTURE_MAX_LEVEL

设置一个最高定义的mipmap级别的索引。它是一个整型数据。其缺省值是1000。

GL_TEXTURE_WRAP_S

将纹理坐标s的缠绕参数设置为**GL_CLAMP**、**GL_CLAMP_TO_EDGE**或**GL_REPEAT**。

GL_CLAMP的作用是将坐标s截断到范围[0, 1]内。当把一个单独的图像映射到一个物体时，这种方式能有效地避免缠绕时人为因素的影响。**GL_CLAMP_TO_EDGE**的作用是将s坐标截断到范围[$\frac{1}{2N}$, $1 - \frac{1}{2N}$]内，这里N是截断方向上的纹理尺寸。**GL_REPEAT**的作用是忽略s坐标的整数部分，这时GL仅使用其小数部分，通过这种方式而创建一个重复的图案。仅当缠绕方式设置为**GL_CLAMP**时，边界上的纹理元素才可以访问。初始情况下，**GL_TEXTURE_WRAP_S**的值被设置成**GL_REPEAT**。

GL_TEXTURE_WRAP_T

将纹理坐标t的缠绕参数设置为**GL_CLAMP**、**GL_CLAMP_TO_EDGE**或**GL_REPEAT**。其具体情况同**GL_TEXTURE_WRAP_S**。初始情况下，**GL_TEXTURE_WRAP_T**的值被设置成**GL_REPEAT**。

GL_TEXTURE_WRAP_R

将纹理坐标r的缠绕参数设置为**GL_CLAMP**、**GL_CLAMP_TO_EDGE**或**GL_REPEAT**。其具体情况同**GL_TEXTURE_WRAP_S**。初始情况下，**GL_TEXTURE_WRAP_R**的值被设置成**GL_REPEAT**。

GL_TEXTURE_BORDER_COLOR

设置一个边界颜色。参数*params*包含的四个值组成了纹理边界的RGBA颜色。整型颜色组分数据被线性地映射为整型浮点格式：最大的正数映射为1.0，最小的负数映射为-1.0。这些值的取值范围是[0, 1]。默认的边界颜色是(0, 0, 0, 0)。

GL_TEXTURE_PRIORITY

指定当前连接纹理的纹理驻留优先级。其允许值的范围是[0, 1]。请参阅函数glPrioritizeTextures()和glBindTexture()以获取更多的信息。

- 注意：

GL_TEXTURE_3D、**GL_TEXTURE_MIN_LOD**、**GL_TEXTURE_MAX_LOD**、**GL_TEXTURE_BASE_LEVEL**和**GL_TEXTURE_MAX_LEVEL**仅在GL 1.2以上的版本中才可以使用。

假设一个程序已经启动了纹理化操作（通过调用带有参数**GL_TEXTURE_1D**、**GL_TEXTURE_2D**或**GL_TEXTURE_3D**的函数glEnable()）并且已经将**GL_TEXTURE_MIN_FILTER**设置成一个要求mipmap的函数。如果当前定义的纹理图像（当调用了函数glTexImage1D()、glTexImage2D()、glTexImage3D()、glCopyTexImage1D()或glCopyTexImage2D()）的尺寸后面跟有不正确的mipmap，或当定义的纹理图像比需要的纹理图像少，或设置的纹理图像含有不同的纹理组分数目时，这时相当于纹理映射功能已关闭。

线性滤波器只有在二维纹理中才访问最近的四个纹理元素。在一维纹理中，线性滤波器只能访问最近的两个纹理元素。

当系统支持**GL_ARB_multitexture**扩展时，函数glTexParameter()将确定由函数glActiveTextureARB()指定的当前纹理单元的纹理参数。

- 出错提示：

当参数*target*或*pname*不是一个可取的指定值时产生**GL_INVALID_ENUM**提示。

当参数*params*没有取指定的常数值（由参数*pname*的取值决定）时产生**GL_INVALID_ENUM**提示。

当函数**glTexParameter()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGetTexParameter()

glGetTexLevelParameter()

- 请参阅：

glActiveTextureARB(), **glBindTexture()**, **glCopyPixels()**, **glCopyTexImage1D()**,
glCopyTexImage2D(), **glCopyTexSubImage1D()**, **glCopyTexSubImage2D()**,
glCopyTexSubImage3D(), **glDrawPixels()**, **glPixelStore()**, **glPixelTransfer()**,
glPrioritizeTextures(), **glTexEnv()**, **glTexGen()**, **glTexImage1D()**, **glTexImage2D()**,
glTexImage3D(), **glTexSubImage1D()**, **glTexSubImage2D()**, **glTexSubImage3D()**

glTexSubImage1D

- 名称：

glTexSubImage1D()

- 功能：

指定一个一维纹理子图像。

- C 描述：

```
void glTexSubImage1D( GLenum target,
                      GLint level,
                      GLint xoffset,
                      GLsizei width,
                      GLenum format,
                      GLenum type,
                      const GLvoid *pixels )
```

- 参数说明：

target 指定目标纹理。必须是**GL_TEXTURE_1D**。

level 指定多重纹理的级别号。0 级是图像的基本级，*n* 级是指第*n* 个 mipmap 的简化图像。

xoffset 指定纹理数组中沿 *x* 方向的一个纹理偏移量。

width 指定纹理子图像的宽度。

format 指定像素数据的格式。它可取下面所列的符号型数值：**GL_COLOR_INDEX**、**GL_RED**、**GL_GREEN**、**GL_BLUE**、**GL_ALPHA**、**GL_RGB**、**GL_BGR**、**GL_RGBA**、**GL_BGRA**、**GL_LUMINANCE** 和 **GL_LUMINANCE_ALPHA**。

type 指定像素数据的类型。它可取的值如下：**GL_UNSIGNED_BYTE**、**GL_BYTE**、

GL_BITMAP、**GL_UNSIGNED_SHORT**、**GL_SHORT**、**GL_UNSIGNED_INT**、**GL_INT**、**GL_FLOAT**、**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**。

pixels 指定一个指针，指向内存中的图像数据。

- 说明：

纹理操作是把一个指定的纹理图像的一部分映射到每个允许该操作的图元上。函数**glEnable(GL_TEXTURE_1D)**和**glDisable(GL_TEXTURE_1D)**用来启动和关闭一维纹理操作。

函数**glTexSubImage1D()**的作用是在一个已存在的一维纹理图像中重新定义一个连续的子区间。由参数*pixels*指定的纹理将替换已存在的纹理数组中的这一部分，该部分位于x方向的*xoffset*和*xoffset+width-1*之间。该区域可能并不包含最初指定的纹理数组范围以外的任何像素。指定一个宽度为0的子纹理是允许的，但这种指定将不产生任何效果。

- 注意：

函数**glTexSubImage1D()**只有在GL 1.1以上的版本中可以使用。

纹理操作在颜色索引模式下无效。

函数**glPixelStore()**和**glPixelTransfer()**对纹理图像的影响同它们对函数**glDrawPixels()**的影响完全一样。

格式**GL_BGR**和**GL_BGRA**及类型**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**仅在GL 1.2以上的版本中才可以使用。

当系统支持**GL_ARB_multitexture**扩展时，函数**glTexSubImage1D()**将为当前纹理单元（由函数**glActiveTextureARB()**指定）指定一个一维子纹理。

当系统支持**GL_ARB_imaging**扩展时，RGBA元素也可以被图像流程处理。有关细节请参阅函数**glTexImage1D()**。

- 出错提示：

当参数*target*不是一个允许的值时产生**GL_INVALID_ENUM**提示。

当纹理数组没有被前面的函数**glTexImage1D()**定义时产生**GL_INVALID_OPERATION**提示。

当参数*level*小于0时产生**GL_INVALID_VALUE**提示。

当参数*level*大于*log₂max*时产生**GL_INVALID_VALUE**提示。此处*max*是**GL_MAX_TEXTURE_SIZE**的返回值。

当*xoffset* $< -b$, 或 (*xoffset* + *width*) $> (w - b)$ 时产生**GL_INVALID_VALUE**提示。此处*w*是**GL_TEXTURE_WIDTH**, *b*是被修正的纹理图像**GL_TEXTURE_BORDER**的宽度。请注意这里的*w*是包括两个边界的。

当参数*width*小于0时产生**GL_INVALID_VALUE**提示。

当参数*format*不是一个可接受的格式常量时产生**GL_INVALID_ENUM**提示。

当参数*type*不是一个类型常量时产生**GL_INVALID_ENUM**提示。

当参数*type*取**GL_BITMAP**, 但参数*format*不是**GL_COLOR_INDEX**时, 产生**GL_INVALID_ENUM**提示。

当函数**glTexSubImage1D()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**之一, 但参数*format*却不是**GL_RGB**格式时, 产生**GL_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_SHORT_4_4_4_4**, **GL_UNSIGNED_SHORT_4_4_4_4_REV**, **GL_UNSIGNED_SHORT_5_5_5_1**, **GL_UNSIGNED_SHORT_1_5_5_5_REV**, **GL_UNSIGNED_INT_8_8_8_8**, **GL_UNSIGNED_INT_8_8_8_8_REV**, **GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**之一, 但参数*format*既不是**GL_RGBA**格式又不是**GL_BGRA**格式时, 产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取:

glGetTexImage()

glIsEnabled(GL_TEXTURE_1D)

- 请参阅:

glActiveTextureARB(), **glCopyTexImage1D()**, **glCopyTexImage2D()**, **glCopyTexSubImage1D()**, **glCopyTexSubImage2D()**, **glCopyTexSubImage3D()**, **glDrawPixels()**, **glPixelStore()**, **glPixelTransfer()**, **glTexEnv()**, **glTexGen()**, **glTexImage1D()**, **glTexImage2D()**, **glTexImage3D()**, **glTexParameter()**, **glTexSubImage2D()**, **glTexSubImage3D()**

* **glTexSubImage2D**

- 名称:

glTexSubImage2D()

- 功能:

指定一个二维纹理子图像。

- C 描述:

```
void glTexSubImage2D(GLenum target,
                      GLint level,
```

```

    GLint xoffset,
    GLint yoffset,
    GLsizei width,
    GLsizei height,
    GLenum format,
    GLenum type,
    const GLvoid *pixels )

```

• 参数说明：

<i>target</i>	指定目标纹理。必须是 GL_TEXTURE_2D 。
<i>level</i>	指定多重纹理的级别号。0级是图像的基本级。 <i>n</i> 级是指第 <i>n</i> 个mipmap的简化图像。
<i>xoffset</i>	指定纹理数组中沿x方向的一个纹理偏移量。
<i>yoffset</i>	指定纹理数组中沿y方向的一个纹理偏移量。
<i>width</i>	指定纹理子图像的宽度。
<i>height</i>	指定纹理子图像的高度。
<i>format</i>	指定像素数据的格式。它可取下面所列的符号型数值： GL_COLOR_INDEX 、 GL_RED 、 GL_GREEN 、 GL_BLUE 、 GL_ALPHA 、 GL_RGB 、 GL_BGR 、 GL_RGBA 、 GL_BGRA 、 GL_LUMINANCE 和 GL_LUMINANCE_ALPHA 。
<i>type</i>	指定像素数据的类型。它可取的值如下： GL_UNSIGNED_BYTE 、 GL_BYTE 、 GL_BITMAP 、 GL_UNSIGNED_SHORT 、 GL_SHORT 、 GL_UNSIGNED_INT 、 GL_INT 、 GL_FLOAT 、 GL_UNSIGNED_BYTE_3_3_2 、 GL_UNSIGNED_BYTE_2_3_3_REV 、 GL_UNSIGNED_SHORT_5_6_5 、 GL_UNSIGNED_SHORT_5_6_5_REV 、 GL_UNSIGNED_SHORT_4_4_4_4 、 GL_UNSIGNED_SHORT_4_4_4_4_REV 、 GL_UNSIGNED_SHORT_5_5_5_1 、 GL_UNSIGNED_SHORT_1_5_5_5_REV 、 GL_UNSIGNED_INT_8_8_8_8 、 GL_UNSIGNED_INT_8_8_8_8_REV 、 GL_UNSIGNED_INT_10_10_10_2 和 GL_UNSIGNED_INT_2_10_10_10_REV 。
<i>pixels</i>	指定一个指针，指向内存中的图像数据。

• 说明：

纹理操作是把一个指定的纹理图像的一部分映射到每个允许该操作的图元上。函数**glEnable(GL_TEXTURE_2D)**和**glDisable(GL_TEXTURE_2D)**用来启动和关闭二维纹理操作。

函数**glTexSubImage2D()**的作用是在一个已存在的二维纹理图像中重新定义一个连续的子区间。由参数*pixels*指定的纹理将替换已存在的纹理数组中的这一部分，该部分位于*x*方向的*xoffset*和*xoffset+width-1*及*y*方向的*yoffset*和*yoffset+height-1*之间。该区域可以并不包含最初指定的纹理数组范围以外的任何像素。指定一个宽度为0或高度为0的子纹理是允许的，但这种指定将不产生任何效果。

• 注意：

函数`glTexSubImage2D()`只有在GL 1.1以上的版本中才可以使用。

纹理操作在颜色索引模式下无效。

函数`glPixelStore()`和`glPixelTransfer()`对纹理图像的影响同它们对函数`glDrawPixels()`的影响完全一样。

格式`GL_BGR`和`GL_BGRA`及类型`GL_UNSIGNED_BYTE_3_3_2`、`GL_UNSIGNED_BYTE_2_3_3_REV`、`GL_UNSIGNED_SHORT_5_6_5`、`GL_UNSIGNED_SHORT_5_6_5_REV`、`GL_UNSIGNED_SHORT_4_4_4_4`、`GL_UNSIGNED_SHORT_4_4_4_4_REV`、`GL_UNSIGNED_SHORT_5_5_5_1`、`GL_UNSIGNED_SHORT_1_5_5_5_REV`、`GL_UNSIGNED_INT_8_8_8_8`、`GL_UNSIGNED_INT_8_8_8_8_REV`、`GL_UNSIGNED_INT_10_10_10_2`和`GL_UNSIGNED_INT_2_10_10_10_REV`仅在GL1.2以上的版本中才可以使用。

当系统支持`GL_ARB_multitexture`扩展时，函数`glTexSubImage2D()`将为当前纹理单元（由函数`glActiveTextureARB()`指定）指定一个二维子纹理。

当系统支持`GL_ARB_imaging`扩展时，RGBA元素也可以被图像流程处理。有关细节请参阅函数`glTexImage2D()`。

- 出错提示：

当参数`target`不是`GL_TEXTURE_2D`时产生`GL_INVALID_ENUM`提示。

当纹理数组没有被前面的函数`glTexImage2D()`定义时产生`GL_INVALID_OPERATION`提示。

当参数`level`小于0时产生`GL_INVALID_VALUE`提示。

当参数`level`大于`log2max`时产生`GL_INVALID_VALUE`提示。此处`max`是`GL_MAX_TEXTURE_SIZE`的返回值。

当 $x_{\text{offset}} < -b$ ，或 $(x_{\text{offset}} + width) > (w - b)$ ，或当 $y_{\text{offset}} < -b$ ，或 $(y_{\text{offset}} + height) > (h - b)$ 时产生`GL_INVALID_VALUE`提示。此处`w`是`GL_TEXTURE_WIDTH`，`h`是`GL_TEXTURE_HEIGHT`，`b`是被修改的纹理图像的边界宽度。请注意这里的`w`和`h`是包括两个边界的。

当参数`width`或`height`小于0时产生`GL_INVALID_VALUE`提示。

当参数`format`不是一个可取的格式常量时产生`GL_INVALID_ENUM`提示。

当参数`type`不是一个类型常量时产生`GL_INVALID_ENUM`提示。

当参数`type`取`GL_BITMAP`，但参数`format`不是`GL_COLOR_INDEX`时，产生`GL_INVALID_ENUM`提示。

当函数`glTexSubImage2D()`在函数对`glBegin()`/`glEnd()`之间执行时产生`GL_INVALID_OPERATION`提示。

当参数`type`为`GL_UNSIGNED_BYTE_3_3_2`、`GL_UNSIGNED_BYTE_2_3_3_REV`、`GL_UNSIGNED_SHORT_5_6_5`、或`GL_UNSIGNED_SHORT_5_6_5_REV`之一，但参数`format`却不是`GL_RGB`格式时产生`GL_INVALID_OPERATION`提示。

当参数`type`为`GL_UNSIGNED_SHORT_4_4_4_4`、`GL_UNSIGNED_SHORT_4_4_4_4_REV`、

GL_UNSIGNED_SHORT_5_5_5_1、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**或**GL_UNSIGNED_INT_2_10_10_10_REV**之一，但参数*format*既不是**GL_RGBA**格式又不是**GL_BGRA**格式时，产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取：

glGetTexImage()

glIsEnabled(GL_TEXTURE_2D)

- 请参阅：

glActiveTextureARB(), **glCopyTexImage1D()**, **glCopyTexImage2D()**, **glCopyTexSubImage1D()**, **glCopyTexSubImage2D()**, **glCopyTexSubImage3D()**, **glDrawPixels()**, **glPixelStore()**, **glPixelTransfer()**, **glTexEnv()**, **glTexGen()**, **glTexImage1D()**, **glTexImage2D()**, **glTexImage3D()**, **glTexSubImage1D()**, **glTexSubImage3D()**, **glTexParameter()**

• **glTexSubImage3D**

- 名称：

glTexSubImage3D()

- 功能：

指定一个三维纹理子图像。

- C 描述：

```
void glTexSubImage3D( GLenum target,
                      GLint level,
                      GLint xoffset,
                      GLint yoffset,
                      GLint zoffset,
                      GLsizei width,
                      GLsizei height,
                      GLsizei depth,
                      GLenum format,
                      GLenum type,
                      const GLvoid *pixels )
```

- 参数说明：

target 指定目标纹理。必须是**GL_TEXTURE_3D**。

level 指定多重纹理的级别号。0级是图像的基本级。*n*层是指第*n*个mipmap的简化图像。

xoffset 指定纹理数组中沿x方向的一个纹理偏移量。

yoffset 指定纹理数组中沿y方向的一个纹理偏移量。

zoffset 指定纹理数组中沿z方向的一个纹理偏移量。

width 指定纹理子图像的宽度。

<i>height</i>	指定纹理子图像的高度。
<i>depth</i>	指定纹理子图像的深度。
<i>format</i>	指定像素数据的格式。它可取下面所列的符号值之一： GL_COLOR_INDEX 、 GL_RED 、 GL_GREEN 、 GL_BLUE 、 GL_ALPHA 、 GL_RGB 、 GL_BGR 、 GL_RGBA 、 GL_BGRA 、 GL_LUMINANCE 和 GL_LUMINANCE_ALPHA 。
<i>type</i>	指定像素数据的类型。它可取的值如下： GL_UNSIGNED_BYTE 、 GL_BYTE 、 GL_BITMAP 、 GL_UNSIGNED_SHORT 、 GL_SHORT 、 GL_UNSIGNED_INT 、 GL_INT 、 GL_FLOAT 、 GL_UNSIGNED_BYTE_3_3_2 、 GL_UNSIGNED_BYTE_2_3_3_REV 、 GL_UNSIGNED_SHORT_5_6_5 、 GL_UNSIGNED_SHORT_5_6_5_REV 、 GL_UNSIGNED_SHORT_4_4_4_4 、 GL_UNSIGNED_SHORT_4_4_4_4_REV 、 GL_UNSIGNED_SHORT_5_5_5_1 、 GL_UNSIGNED_SHORT_1_5_5_5_REV 、 GL_UNSIGNED_INT_8_8_8_8 、 GL_UNSIGNED_INT_8_8_8_8_REV 、 GL_UNSIGNED_INT_10_10_10_2 和 GL_UNSIGNED_INT_2_10_10_10_REV 。
<i>pixels</i>	指定一个指针，指向内存中的图像数据。

• 说明：

纹理操作是把一个指定的纹理图像的一部分映射到每个允许该操作的图元上。函数**glEnable(GL_TEXTURE_3D)**和**glDisable(GL_TEXTURE_3D)**用于启动和关闭三维纹理操作。

函数**glTexSubImage3D()**的作用是在一个已存在的三维纹理图像中重新定义一个连续的子区间。由参数*pixels*指定的纹理将替换已存在的纹理数组中的这一部分，该部分位于x方向的*xoffset*和*xoffset+width-1*之间，y方向的*yoffset*和*yoffset+height-1*以及z方向的*zoffset*和*zoffset+depth-1*之间。该区域可能并不包含最初指定的纹理数组范围以外的任何像素。指定一个宽度为0或高度为0或深度为0的子纹理并不是一个错误，但这种指定将不产生任何效果。

• 注意：

函数**glTexSubImage3D()**只有在GL 1.2以上的版本中才可以使用。

纹理操作在颜色索引模式下无效。

函数**glPixelStore()**和**glPixelTransfer()**对纹理图像的影响同它们对函数**glDrawPixels()**的影响完全一样。

格式**GL_BGR**和**GL_BGRA**及类型**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**仅在GL 1.2以上的版本中才可以使用。

当系统支持**GL_ARB_multitexture**扩展时，函数**glTexSubImage3D()**将为当前纹理单元（由函数**glActiveTextureARB()**指定）指定一个三维子纹理。

当系统支持**GL_ARB_imaging**扩展时，**RGBA**元素也可以被图像流程处理。有关细节请参

阅函数glTexImage3D()。

- 出错提示：

当参数*target*不是GL_TEXTURE_3D时产生GL_INVALID_ENUM提示。

当纹理数组没有被前面的函数glTexImage3D()定义时产生GL_INVALID_OPERATION提示。

当参数*level*小于0时产生GL_INVALID_VALUE提示。

当参数*level*大于log₂*max*时产生GL_INVALID_VALUE提示。此处*max*是GL_MAX_TEXTURE_SIZE的返回值。

当*xoffset*<-*b*，或(*xoffset*+*width*)>(*w*-*b*)，当*yoffset*<-*b*，或(*yoffset*+*height*)>(*h*-*b*)，当*zoffset*<-*b*，或(*zoffset*+*depth*)>(*d*-*b*)时产生GL_INVALID_VALUE提示。此处*w*是GL_TEXTURE_WIDTH，*h*是GL_TEXTURE_HEIGHT，*d*是GL_TEXTURE_DEPTH，*b*是被修改的纹理图像的边界宽度。请注意这里的*w*、*h*和*d*是包括两个边界的。

当参数*width*、*height*或*depth*小于0时产生GL_INVALID_VALUE提示。

当参数*format*不是一个接受的格式常量时产生GL_INVALID_ENUM提示。

当参数*type*不是一个类型常量时产生GL_INVALID_ENUM提示。

当参数*type*取GL_BITMAP，但参数*format*不是GL_COLOR_INDEX时，产生GL_INVALID_ENUM提示。

当函数glTexSubImage3D()在函数对glBegin()/glEnd()之间执行时产生GL_INVALID_OPERATION提示。

当参数*type*为GL_UNSIGNED_BYTE_3_3_2、GL_UNSIGNED_BYTE_2_3_3_REV、GL_UNSIGNED_SHORT_5_6_5、或GL_UNSIGNED_SHORT_5_6_5_REV之一，但参数*format*却不是GL_RGB格式时，产生GL_INVALID_OPERATION提示。

当参数*type*为GL_UNSIGNED_SHORT_4_4_4_4、GL_UNSIGNED_SHORT_4_4_4_4_REV、GL_UNSIGNED_SHORT_5_5_5_1、GL_UNSIGNED_SHORT_1_5_5_5_REV、GL_UNSIGNED_INT_8_8_8_8、GL_UNSIGNED_INT_8_8_8_8_REV、GL_UNSIGNED_INT_10_10_10_2和GL_UNSIGNED_INT_2_10_10_10_REV之--，但参数*format*既不是GL_RGBA格式又不是GL_BGRA格式时，产生GL_INVALID_OPERATION提示。

- 有关数据的获取：

glGetTexImage()

glIsEnabled(GL_TEXTURE_3D)

- 请参阅：

glActiveTextureARB(), glCopyTexImage1D(), glCopyTexImage2D(), glCopyTexSubImage1D(), glCopyTexSubImage2D(), glCopyTexSubImage3D(), glDrawPixels(), glPixelStore(), glPixelTransfer(), glTexEnv(), glTexGen(), glTexImage1D(), glTexImage2D(), glTexImage3D(), glTexSubImage1D(), glTexSubImage2D(), glTexParameter()。

- glTranslate

- 名称:

glTranslated(), **glTranslatef()**

- 功能:

把当前矩阵乘上一个平移矩阵。

- C 描述:

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

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

- 参数说明:

x, *y*, *z*

分别指定平移向量的*x*、*y*和*z*方向的坐标值。

- 说明:

函数**glTranslate()**通过(*x*, *y*, *z*)而产生一个平移。当前矩阵(请参阅**glMatrixMode()**)将由它与这个平移矩阵相乘的结果替换。这一操作相当于调用带有下面所列自变量的**glMultMatrix()**函数:

$$\begin{matrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{matrix}$$

如果矩阵模式是**GL_MODELVIEW**或**GL_PROJECTION**时, 调用函数**glTranslate()**后所有要绘制的对象都将被平移。

可用函数**glPushMatrix()**和**glPopMatrix()**存储和返回未平移的坐标系统。

- 出错提示:

当函数**glTranslate()**在函数对**glBegin()**/**glEnd()**之间执行时产生**GL_INVALID_OPERATION**提示。

- 有关数据的获取:

```
glGet( GL_MATRIX_MODE )
glGet( GL_COLOR_MATRIX )
glGet( GL_MODELVIEW_MATRIX )
glGet( GL_PROJECTION_MATRIX )
glGet( GL_TEXTURE_MATRIX )
```

- 请参阅:

glMatrixMode(), **glMultMatrix()**, **glPushMatrix()**, **glRotate()**, **glScale()**

■ **glVertex**

- 名称:

glVertex2d(), **glVertex2f()**, **glVertex2i()**, **glVertex2s()**, **glVertex3d()**, **glVertex3f()**,
glVertex3i(), **glVertex3s()**, **glVertex4d()**, **glVertex4f()**, **glVertex4i()**, **glVertex4s()**,
glVertex2dv(), **glVertex2fv()**, **glVertex2iv()**, **glVertex2sv()**, **glVertex3dv()**, **glVertex3fv()**,
glVertex3iv(), **glVertex3sv()**, **glVertex4dv()**, **glVertex4fv()**, **glVertex4iv()**, **glVertex4sv()**

- 功能:

指定一个顶点。

- C 描述:

```
void glVertex2d(GLdouble x,  
                GLdouble y)  
void glVertex2f(GLfloat x,  
                GLfloat y)  
void glVertex2i(GLint x,  
                GLint y)  
void glVertex2s(GLshort x,  
                GLshort y)  
void glVertex3d(GLdouble x,  
                GLdouble y,  
                GLdouble z)  
void glVertex3f(GLfloat x,  
                GLfloat y,  
                GLfloat z)  
void glVertex3i(GLint x,  
                GLint y,  
                GLint z)  
void glVertex3s(GLshort x,  
                GLshort y,  
                GLshort z)  
void glVertex4d(GLdouble x,  
                GLdouble y,  
                GLdouble z,  
                GLdouble w)  
void glVertex4f(GLfloat x,  
                GLfloat y,  
                GLfloat z,  
                GLfloat w)  
void glVertex4i(GLint x,
```

```

    GLint y,
    GLint z,
    GLint w )
void glVertex4s( GLshort x,
                  GLshort y,
                  GLshort z,
                  GLshort w )

```

• 参数说明：

x, *y*, *z*, *w*

指定一个顶点的*x*, *y*, *z*和*w*的坐标值。并不是所有的命令格式都需要提供全部的参数。

• C描述：

```

void glVertex2dv( const GLdouble *v )
void glVertex2fv( const GLfloat *v )
void glVertex2iv( const GLint *v )
void glVertex2sv( const GLshort *v )
void glVertex3dv( const GLdouble *v )
void glVertex3fv( const GLfloat *v )
void glVertex3iv( const GLint *v )
void glVertex3sv( const GLshort *v )
void glVertex4dv( const GLdouble *v )
void glVertex4fv( const GLfloat *v )
void glVertex4iv( const GLint *v )
void glVertex4sv( const GLshort *v )

```

• 参数说明：

v 指定一个指向二元、三元或四元数组的指针。二元数组中包含的元素是*x*和*y*; 三元数组中包含的元素是*x*、*y*和*z*; 四元数组中包含的元素是*x*、*y*、*z*和*w*。

• 说明：

函数glVertex()被用在函数对glBegin()/glEnd()之间，其作用是指定一个点、线和多边形的顶点。当调用函数glVertex()时，当前的颜色、法向量和纹理坐标将与顶点相关联。

当仅指定了*x*和*y*值时，*z*将取缺省值0，*w*将取缺省值1；当已指定了*x*、*y*和*z*值时，*w*将取缺省值1。

• 注意：

在函数对glBegin()/glEnd()之外调用函数glVertex()将导致未定义的动作。

• 请参阅：

glBegin(), glCallList(), glColor(), glEdgeFlag(), glEvalCoord(), glIndex(), glMaterial(),
glNormal(), glRect(), glTexCoord(), glVertexPointer()

• **glVertexPointer**

- 名称：

glVertexPointer()

- 功能：

定义一个顶点数据的数组。

- C 描述：

```
void glVertexPointer( GLint size,
                      GLenum type,
                      GLsizei stride,
                      const GLvoid *pointer )
```

- 参数说明：

size 指定每个顶点的坐标个数。它必须是2、3或4。其初始值是4。

type 指定数组中每个坐标的数据类型。它可取符号常量**GL_SHORT**, **GL_INT**, **GL_FLOAT**和**GL_DOUBLE**。其初始值是**GL_FLOAT**。

stride 指定相邻顶点之间的字节偏移量。如果参数*stride*是0，则认为顶点被一个接一个的排列在数组中。其初始值是0。

pointer 指定一个指针，指向数组中的第一个顶点的第一个坐标位置。其初始值是0。

- 说明：

函数**glVertexPointer()**的作用是绘图时指定一个顶点坐标数组的存放位置和数据格式。参数*size*用来指定每个顶点的坐标个数，参数*type*用来指定每个坐标的数据类型。参数*stride*用来指定从一个顶点到下一个顶点之间的字节偏移量，从而确定这些顶点是存放在一个数组中还是存放在多个数组中。（在一些实现设备中，单一数组的存放形式可能更有效，请参阅**glInterleaved- Arrays()**）。当一个顶点数组被指定后，参数*size*、*type*、*stride*和*pointer*将被存储为客户端状态。

用函数**glEnableClientState(GL_VERTEX_ARRAY)**和**glDisableClientState(GL_VERTEX_ARRAY)**可以启动和关闭顶点数组。当顶点数组启动后，顶点数组可被函数**glDrawArrays()**、**glDrawElements()**或**glArrayElement()**访问。

函数**glDrawArrays()**的作用是用预先指定的顶点和顶点属性数组构造一个图元序列（所有相同的类型）。函数**glArrayElement()**可以通过检索顶点和顶点属性来指定图元。函数**glDrawElements()**则是通过检索顶点和顶点属性来构造一个图元序列。

- 注意：

函数**glVertexPointer()**仅在GL 1.1以上版本中可用。

顶点数组在初始情况下是关闭的，这时不能调用函数**glArrayElement()**、**glDrawElements()**或**glDrawArrays()**访问它。

函数**glVertexPointer()**不允许在函数对**glBegin()**/**glEnd()**之间执行，否则可能产生一个出错提示，但也有可能不产生出错提示。如果没有出现出错提示，其运行将是未定义的。

函数**glVertexPointer()**通常在客户端执行。

因为顶点数组的参数是客户端状态，所以它不能用函数**glPushAttrib()**和**glPopAttrib()**存储和返回，而应由函数**glPushClientAttrib()**和**glPopClientAttrib()**代替进行存储和返回操作。

- 出错提示：

当参数*size*不是2、3和4时产生**GL_INVALID_VALUE**提示。

当参数*type*不是一个接受的值时产生**GL_INVALID_ENUM**提示。

当参数*stride*是负数时产生**GL_INVALID_VALUE**提示。

- 有关数据的获取：

```
glIsEnabled( GL_VERTEX_ARRAY )
glGet( GL_VERTEX_ARRAY_SIZE )
glGet( GL_VERTEX_ARRAY_TYPE )
glGet( GL_VERTEX_ARRAY_STRIDE )
glGetPointerv( GL_VERTEX_ARRAY_POINTER )
```

- 请参阅：

glArrayElement(), **glColorPointer()**, **glDrawArrays()**, **glDrawElements()**, **glDrawRangeElements()**, **glEdgeFlagPointer()**, **glEnable()**, **glGetPointerv()**, **glIndexPointer()**, **glInterleavedArrays()**, **glNormalPointer()**, **glPopClientAttrib()**, **glPushClientAttrib()**, **glTexCoordPointer()**

• **glViewport**

- 名称：

glViewport()

- 功能：

设置视口。

- C描述：

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

- 参数说明：

x, *y*

指定一个视口像素矩形的左下角。其初始值是(0, 0)。

width, *height*

指定一个视口的宽度和高度。当一个GL环境被第一次连接到一个窗口时，参数*width*和*height*按此窗口的大小设置。

- 说明：

函数**glViewport()**为*x*和*y*指定了一种由标准化的设备坐标到窗口坐标的仿射几何转化。设(*x_{nd}*, *y_{nd}*)是标准化的设备坐标，则窗口坐标(*x_w*, *y_w*)可由下式求得：

$$x_w = (x_{nd} + 1) \frac{width}{2} + x$$

$$y_w = (y_{nd} + 1) \frac{height}{2} + y$$

视口的宽度和高度被默认地截断到一定的范围，具体范围值由所处环境决定。可调用函数 **glGet(GL_MAX_VIEWPORT_DIMS)** 查询这一范围值。

- 出错提示：

当参数 *width* 或 *height* 是负数时产生 **GL_INVALID_VALUE** 提示。

当函数 **glViewport()** 在函数对 **glBegin()/glEnd()** 之间执行时产生 **GL_INVALID_OPERATION** 提示。

- 有关数据的获取：

glGet(GL_VIEWPORT)

glGet(GL_MAX_VIEWPORT_DIMS)

- 请参阅：

glDepthRange()

第6章 GLU参考说明

本章按字母顺序列出了OpenGL实用库（GLU）中所包含的所有例程（见表6-1）的参考说明。

表 6-1

glBeginCurve, glEndCurve	glGetString	glPwlCurve
glBeginPolygon, glEndPolygon	glGetTessProperty	glQuadricCallback
glBeginSurface, glEndSurface	glLoadSamplingMatrices	glQuadricDrawStyle
glBeginTrim, glEndTrim	glLookAt	glQuadricNormals
glBuild1DMipmapLevels	glNewNurbsRenderer	glQuadricOrientation
glBuild1DMipmaps	glNewQuadric	glQuadricTexture
glBuild2DMipmapLevels	glNewTess	glScaleImage
glBuild2DMipmaps	glNextContour	glSphere
glBuild3DMipmapLevels	glNurbsCallback	glTessBeginContour, glTessEndContour
glBuild3DMipmaps	glNurbsCallbackData	glTessBeginPolygon
glCheckExtension	glNurbsCurve	glTessCallback
glCylinder	glNurbsProperty	glTessEndPolygon
glDeleteNurbsRenderer	glNurbsSurface	glTessNormal
glDeleteQuadric	glOrtho2D	glTessProperty
glDeleteTess	glPartialDisk	glTessVertex
glDisk	glPerspective	glUnProject
glErrorString	glPickMatrix	glUnProject4
glGetNurbsProperty	glProject	

表6-2中的命令在GLU 1.2中是不可用的，但考虑到本书的完整性，在本章的末尾部分也将它们列了出来。

表 6-2

gluBeginPolygon, gluEndPolygon	gluNextContour
-----------------------------------	----------------

• gluBeginCurve, gluEndCurve

- 名称：

gluBeginCurve(), gluEndCurve()

- 功能：

限定一个NURBS曲线的定义。

- C描述：

```
void gluBeginCurve(GLUnurbs* nurb )
void gluEndCurve ( GLUnurbs* nurb )
```

- 参数说明：

nurb 指定NURBS对象（由gluNewNurbsRenderer()创建）。

- 说明：

例程gluBeginCurve()用来标记一个NURBS曲线定义的开始。然后通过调用一个或多个例程gluNurbsCurve()来定义该曲线的属性。如果要正确调用一个gluNurbsCurve()例程，必须将曲线类型指定为**GL_MAP1_VERTEX_3**或**GL_MAP1_VERTEX_4**。例程gluEndCurve()用来标记该曲线定义的结束。

GL求值器将把NURBS曲线作为一系列的线段来进行绘制。绘制过程中，求值器的状态由函数glPushAttrib(**GL_EVAL_BIT**)和glPopAttrib()保存。关于如何准确地保存这些状态，请参阅函数glPushAttrib()的参考说明。

- 示例：

以下命令绘制一个带有法线的纹理化的NURBS曲线，纹理坐标及法线也同样被指定为NURBS曲线。

```
gluBeginCurve(nobj);
  gluNurbsCurve(nobj, ..., GL_MAP1_TEXTURE_COORD_2);
  gluNurbsCurve(nobj, ..., GL_MAP1_NORMAL);
  gluNurbsCurve(nobj, ..., GL_MAP1_VERTEX_4);
gluEndCurve(nobj);
```

- 请参阅：

gluBeginSurface(), **gluBeginTrim()**, **gluNewNurbsRenderer()**, **gluNurbsCurve()**,
glPopAttrib(), **glPushAttrib()**

: gluBeginSurface, gluEndSurface

- 名称：

gluBeginSurface(), **gluEndSurface()**

- 功能：

限定一个NURBS曲面的定义。

- C描述：

```
void gluBeginSurface(GLUnurbs* nurb )
void gluEndSurface ( GLUnurbs* nurb )
```

- 参数说明：

nurb 指定NURBS对象（由gluNewNurbsRenderer()创建）。

- 说明：

例程`gluBeginSurface()`用来标记一个NURBS曲面定义的开始。然后通过调用一个或多个例程`gluNurbsSurface()`来定义该曲面的属性。如果要正确调用一个`gluNurbsSurface()`例程，必须将曲面类型指定为`GL_MAP2_VERTEX_3`或`GL_MAP2_VERTEX_4`。例程`gluEndSurface()`用来标记该曲面定义的结束。

NURBS曲面的修整由例程`gluBeginTrim()`、`gluPwlCurve()`、`gluNurbsCurve()`和`gluEndTrim()`来支持。有关细节请参阅例程`gluBeginTrim()`的参考说明。

GL求值器将把NURBS曲面作为一组多边形来进行绘制。绘制过程中，求值器的状态由函数`glPushAttrib(GL_EVAL_BIT)`和`glPopAttrib()`保存。关于如何准确地保存这些状态，请参阅函数`glPushAttrib()`的参考说明。

- 示例：

以下命令绘制一个带有法线的纹理化NURBS曲面，纹理坐标及法线也同样被描述为NURBS曲面。

```
gluBeginSurface(nobj);
  gluNurbsSurface(nobj, ..., GL_MAP2_TEXTURE_COORD_2);
  gluNurbsSurface(nobj, ..., GL_MAP2_NORMAL);
  gluNurbsSurface(nobj, ..., GL_MAP2_VERTEX_4);
gluEndSurface(nobj);
```

- 请参阅：

`gluBeginCurve()`, `gluBeginTrim()`, `gluNewNurbsRenderer()`, `gluNurbsCurve()`,
`gluNurbsSurface()`, `gluPwlCurve()`

■ `gluBeginTrim`, `gluEndTrim`

- 名称：

`gluBeginTrim()`, `gluEndTrim()`

- 功能：

限定一个NURBS修整环的定义。

- C描述：

```
void gluBeginTrim(GLUnurbs* nurb)
void gluEndTrim( GLUnurbs* nurb )
```

- 参数说明：

nurb 指定NURBS对象（由`gluNewNurbsRenderer()`创建）。

- 说明：

例程`gluBeginTrim()`用来标记一个修整环的开始，例程`gluEndTrim()`用来标记该修整环的结束。一个修整环是一组定向曲线段（构成一个封闭的曲线）。它用来定义一个NURBS曲面的边界。请将这些修整环包含在一个NURBS曲面的定义中，也就是说包含在例程对`gluBeginSurface()`/`gluEndSurface()`之间。

一个NURBS曲面的定义中可以包含多个修整环。比如说，当你定义的一个NURBS曲面形状

类似于在一个长方形中压出一个孔时，该曲面形状定义将包含两个修整环。一个环用来定义外面的长方形边界，而另一个环用来定义长方形中间的孔。这些修整环的定义都应该包含在例程对gluBeginTrim()/gluEndTrim()之间。

一个封闭的修整环的定义可以包含多条曲线段，而每个曲线段都可以作为分段的线性曲线（请参阅gluPwlCurve()）或一个单独的NURBS曲线（请参阅gluNurbsCurve()），也可以是这两种情况的任意形式的组合。能在修整环定义中（即例程对gluBeginTrim()和gluEndTrim()之间）出现的命令只能是gluPwlCurve()和gluNurbsCurve()。

NURBS曲面的显示区域是当曲线参数增加时修整曲线左边的区域。因此，NURBS曲面所包含的区域是指一个逆时针修整环以内而在一个顺时针修整环之外的部分。对于前面所提到的那个长方形而言，它外面的长方形边界是逆时针的，而内部孔的边界则是顺时针方向的。

当你用不止一个曲线定义一个单独的修整环时，所用的曲线段必须形成一个闭环（也就是说，每个曲线的终点必须是另一个曲线的起始点，而最后一条曲线的终点必须是第一条曲线的起始点）。如果曲线的端点是十分接近但却不完全重合的，那么它们将被强制地连在一起；但如果它们不是非常靠近，就将导致一个错误（请参阅gluNurbsCallback()）。

当一个修整环定义了多条曲线时，它们的方向必须是一致的（即，所有曲线的内部都应该是向左方向）。只要曲线方向之间的变化是正确的，修整环允许相互嵌套。但修整环自身相交或相互之间交叉都将导致错误的发生。

对于一个NURBS曲面而言，如果没有给出任何修整信息，那么它将绘出完整的表面。

- **示例：**

以下的代码定义了一个修整环，该修整环包含一个分段的线性曲线和两个NURBS曲线：

```
gluBeginTrim(nobj);
    gluPwlCurve(..., GLU_MAP1_TRIM_2);
    gluNurbsCurve(..., GLU_MAP1_TRIM_2);
    gluNurbsCurve(..., GLU_MAP1_TRIM_3);
gluEndTrim(nobj);
```

- **请参阅：**

gluBeginSurface(), **gluNewNurbsRenderer()**, **gluNurbsCallback()**, **gluNurbsCurve()**,
gluPwlCurve()

• **gluBuild1DMipmapLevels**

- **名称：**

gluBuild1DMipmapLevels()

- **功能：**

建立一个一维mipmap图层子集。

- **C描述：**

```
GLint gluBuild1DMipmapLevels( GLenum target,
                               GLint internalFormat,
                               GLsizei width,
```

```

GLenum format,
GLenum type,
GLint level,
GLint base,
GLint max,
const void *data )

```

• 参数说明：

target 指定目标纹理。必须是**GL_TEXTURE_1D**。

internalFormat

纹理图像所需要的内部存储格式。它必须是1、2、3、4或下面所列的符号常量之一：**GL_ALPHA**、**GL_ALPHA4**、**GL_ALPHA8**、**GL_ALPHA12**、**GL_ALPHA16**、**GL_LUMINANCE**、**GL_LUMINANCE4**、**GL_LUMINANCE8**、**GL_LUMINANCE12**、**GL_LUMINANCE16**、**GL_LUMINANCE_ALPHA**、**GL_LUMINANCE4_ALPHA4**、**GL_LUMINANCE6_ALPHA2**、**GL_LUMINANCE8_ALPHA8**、**GL_LUMINANCE12_ALPHA4**、**GL_LUMINANCE16_ALPHA16**、**GL_INTENSITY**、**GL_INTENSITY4**、**GL_INTENSITY8**、**GL_INTENSITY12**、**GL_INTENSITY16**、**GL_RGB**、**GL_R3_G3_B2**、**GL_RGB4**、**GL_RGB5**、**GL_RGB8**、**GL_RGB10**、**GL_RGB12**、**GL_RGB16**、**GL_RGBA**、**GL_RGBA2**、**GL_RGBA4**、**GL_RGB5_A1**、**GL_RGBA8**、**GL_RGB10_A2**、**GL_RGBA12**或**GL_RGBA16**。

width 指定纹理图像的像素宽度。它必须是2的幂值。

format 指定像素数据的格式。它必须是：**GL_COLOR_INDEX**、**GL_DEPTH_COMPONENT**、**GL_RED**、**GL_GREEN**、**GL_BLUE**、**GL_ALPHA**、**GL_RGB**、**GL_RGBA**、**GL_BGR**、**GL_BGRA**、**GL_LUMINANCE**或**GL_LUMINANCE_ALPHA**。

type 指定像素数据的类型。它可取的值如下：**GL_UNSIGNED_BYTE**、**GL_BYTE**、**GL_BITMAP**、**GL_UNSIGNED_SHORT**、**GL_SHORT**、**GL_UNSIGNED_INT**、**GL_INT**、**GL_FLOAT**、**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**或**GL_UNSIGNED_INT_2_10_10_10_REV**。

level 指定图形数据的mipmap图层。

base 指定传递给glTexImage1D()的最小mipmap图层。

max 指定传递给glTexImage1D()的最大mipmap图层。

data 指定一个指向内存中图形数据存放位置的指针。

- 说明：

例程gluBuild1DMipmapLevels()建立一个已预先滤波的一维纹理映射子集。该纹理映射是通过调用一个mipmap，降低一个mipmap分辨率而生成的。这一操作可用来实现通过纹理映射而得到的图元的反走样。

当返回值是0时，表示该操作已经成功；否则，它将返回一个GLU的出错代码（请参阅gluErrorString()）。

通过每次减半*data*直至达到 1×1 为止，可以建立一组从*base*到*max*的Mipmap图层。在每一层中，减半的每个mipmap中的纹理像素是它上一个mipmap图层中相应的两个纹理像素的平均值。可以通过调用函数glTexImage1D()来载入这些从*base*到*max*的Mipmap图层。如果*max*大于给定尺寸的纹理的最大mipmap图层，将返回一个GLU的出错代码（请参阅gluErrorString()），同时将不载入任何mipmap。

例如，当*level*是2，*width*是16时，下面的图层是可能的： 16×1 、 8×1 、 4×1 、 2×1 和 1×1 。它们分别对应2、3、4、5和6层。当*base*是3，*max*是5时，则只有 8×1 、 4×1 和 2×1 被载入。然而，如果*max*是7，则由于它大于这里所提供的最大mipmap图层6，因此只产生一个出错代码而并不载入任何mipmap图层。

最大的mipmap层可通过公式 $\log_2(\text{width } 2^{\text{level}})$ 而求得。

参数*type*的取值请参阅glTexImage1D()的参考说明。参数*level*的取值请参阅glDrawPixels()的参考说明。

- 注意：

例程gluBuild1DMipmapLevels()仅在GLU 1.3以上版本中才有效。

格式GL_BGR和GL_BGRA及类型GL_UNSIGNED_BYTE_3_3_2、GL_UNSIGNED_BYTE_2_3_3_REV、GL_UNSIGNED_SHORT_5_6_5、GL_UNSIGNED_SHORT_5_6_5_REV、GL_UNSIGNED_SHORT_4_4_4_4、GL_UNSIGNED_SHORT_4_4_4_4_REV、GL_UNSIGNED_SHORT_5_5_5_1、GL_UNSIGNED_SHORT_1_5_5_5_REV、GL_UNSIGNED_INT_8_8_8_8、GL_UNSIGNED_INT_8_8_8_8_REV、GL_UNSIGNED_INT_10_10_10_2和GL_UNSIGNED_INT_2_10_10_10_REV仅在GL 1.2以上的版本中才有效。

- 出错提示：

当参数*level*>*base*，*base*<0，*max*<*base*或*max*大于最高的mipmap图层时产生GLU_INVALID_VALUE提示。

当参数*width*<1时产生GLU_INVALID_VALUE提示。

当参数*internalFormat*、*format*或*type*不是一个合法值时产生GLU_INVALID_ENUM提示。

当参数*type*为GL_UNSIGNED_BYTE_3_3_2或GL_UNSIGNED_BYTE_2_3_3_REV，并且参数*format*不是GL_RGB格式时产生GLU_INVALID_OPERATION提示。

当参数*type*为GL_UNSIGNED_SHORT_5_6_5或GL_UNSIGNED_SHORT_5_6_5_REV，并且参数*format*不是GL_RGB格式时产生GLU_INVALID_OPERATION提示。

当参数*type*为**GL_UNSIGNED_SHORT_4_4_4_4**或**GL_UNSIGNED_SHORT_4_4_4_4_REV**, 并且参数*format*既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_SHORT_5_5_5_1**或**GL_UNSIGNED_SHORT_1_5_5_5_REV**, 并且参数*format*既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_INT_8_8_8_8**或**GL_UNSIGNED_INT_8_8_8_8_REV**, 并且参数*format*既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_INT_10_10_10_2**或**GL_UNSIGNED_INT_2_10_10_10_REV**, 并且参数*format*既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

- 请参阅:

glDrawPixels(), **glTexImage1D()**, **glTexImage2D()**, **glTexImage3D()**, **gluBuild1DMipmaps()**,
gluBuild2DMipmaps(), **gluBuild3DMipmaps()**, **gluErrorString()**, **glGetTexImage()**,
glGetTexLevelParameter(), **gluBuild2DMipmapLevels()**, **gluBuild3DMipmapLevels()**

• **gluBuild1DMipmaps**

- 名称:

gluBuild1DMipmaps()

- 功能:

建立一个一维mipmap。

- C描述:

```
GLint gluBuild1DMipmaps( GLenum target,
                         GLint internalFormat,
                         GLsizei width,
                         GLenum format,
                         GLenum type,
                         const void *data)
```

- 参数说明:

target 指定目标纹理。必须是**GL_TEXTURE_1D**。

internalFormat

纹理图像所需要的内部存储格式。它必须是1、2、3、4或下面所列的符号常量之一: **GL_ALPHA**、**GL_ALPHA4**、**GL_ALPHA8**、**GL_ALPHA12**、**GL_ALPHA16**、**GL_LUMINANCE**、**GL_LUMINANCE4**、**GL_LUMINANCE8**、**GL_LUMINANCE12**、**GL_LUMINANCE16**、**GL_LUMINANCE_ALPHA**、**GL_LUMINANCE4_ALPHA4**、**GL_LUMINANCE6_ALPHA2**、**GL_LUMINANCE8_ALPHA8**、**GL_LUMINANCE12_ALPHA4**、

GL_LUMINANCE12_ALPHA12、**GL_LUMINANCE16_ALPHA16**、**GL_INTENSITY**、**GL_INTENSITY4**、**GL_INTENSITY8**、**GL_INTENSITY12**、**GL_INTENSITY16**、**GL_RGB**、**GL_R3_G3_B2**、**GL_RGB4**、**GL_RGB5**、**GL_RGB8**、**GL_RGB10**、**GL_RGB12**、**GL_RGB16**、**GL_RGBA**、**GL_RGBA2**、**GL_RGBA4**、**GL_RGB5_A1**、**GL_RGBA8**、**GL_RGB10_A2**、**GL_RGBA12**或**GL_RGBA16**。

- width* 指定纹理图像的像素宽度。
- format* 指定像素数据的格式。它必须是**GL_COLOR_INDEX**、**GL_DEPTH_COMPONENT**、**GL_RED**、**GL_GREEN**、**GL_BLUE**、**GL_ALPHA**、**GL_RGB**、**GL_RGBA**、**GL_BGR**、**GL_BGRA**、**GL_LUMINANCE**或**GL_LUMINANCE_ALPHA**。
- type* 为*data*指定数据类型。它的可取值如下：**GL_UNSIGNED_BYTE**、**GL_BYTE**、**GL_BITMAP**、**GL_UNSIGNED_SHORT**、**GL_SHORT**、**GL_UNSIGNED_INT**、**GL_INT**、**GL_FLOAT**、**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**。
- data* 指定一个指向内存中图像数据存放位置的指针。

• 说明：

例程**gluBuild1DMipmaps()**建立一组已预先滤波的一维纹理映射。该纹理映射通过调用一个mipmap，降低一个mipmap的分辨率而生成的。这一操作可用来实现通过纹理映射而得到的图元的反走样。

当返回值是0时，表示该操作已经成功；否则，它将返回一个GLU的出错代码（请参阅**gluErrorString()**）。

开始时，GL将检测*data*的*width*是否是一个2的幂值。如果不是，*data*的一个拷贝将缩放至最接近的2的幂值。（如果*width*恰好是在两个2的幂值的正中间，则*data*的拷贝值将被放大成较大的值）。这个拷贝值将被用于随后的mipmap操作。例如，如果*width*是57，则在进行mipmap操作前，*data*的一个拷贝值将放大成64。

然后，代理纹理（请参阅**glTexImage1D()**）将被用来确定实现是否适合所要求的纹理。如果不适合，*width*将继续减半直到适合为止。

接下来，通过每次减半*data*的拷贝直至达到 1×1 为止，可以建立一组mipmap图层。在每一层中，减半的每个mipmap中的纹理像素是它上一个mipmap图层中相应的两个纹理像素的平均值。

可以通过调用函数**glTexImage1D()**来载入这些mipmap图层。0层就是*data*的一个拷贝。最高

层是 $\log_2(width)$ 。例如，当width是64并且实现能够存放这种尺寸的纹理时，它将建立下面的mipmap层： 64×1 、 32×1 、 16×1 、 8×1 、 4×1 、 2×1 和 1×1 。它们分别对应0、1、2、3、4、5和6层。

参数type的取值请参阅glTexImage1D()的参考说明。参数data的取值请参阅glDrawPixels()的参考说明。

- 注意：

请注意，这里并没有查询最大层的直接方法。但它可以通过函数glGetTexLevelParameter()来间接查询：首先查询0层中实际使用的宽度（由于代理纹理可能已将width缩放到适合实现所需的尺寸，因此这里的宽度可能不等于width）。然后最大层可以通过公式 $\log_2(width)$ 求得。

格式**GL_BGR**和**GL_BGRA**及类型**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**仅在GL 1.2以上的版本及GLU 1.3以上版本中才能使用。

- 出错提示：

当参数width<1时产生**GLU_INVALID_VALUE**提示。

当参数format或type不是一个合法值时产生**GLU_INVALID_ENUM**提示。

当参数type为**GL_UNSIGNED_BYTE_3_3_2**或**GL_UNSIGNED_BYTE_2_3_3_REV**，并且参数format不是**GL_RGB**格式时产生**GLU_INVALID_OPERATION**提示。

当参数type为**GL_UNSIGNED_SHORT_5_6_5**或**GL_UNSIGNED_SHORT_5_6_5_REV**，并且参数format不是**GL_RGB**格式时产生**GLU_INVALID_OPERATION**提示。

当参数type为**GL_UNSIGNED_SHORT_4_4_4_4**或**GL_UNSIGNED_SHORT_4_4_4_4_REV**，并且参数format既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

当参数type为**GL_UNSIGNED_SHORT_5_5_5_1**或**GL_UNSIGNED_SHORT_1_5_5_5_REV**，并且参数format既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

当参数type为**GL_UNSIGNED_INT_8_8_8_8**或**GL_UNSIGNED_INT_8_8_8_8_REV**，并且参数format既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

当参数type为**GL_UNSIGNED_INT_10_10_10_2**或**GL_UNSIGNED_INT_2_10_10_10_REV**，并且参数format既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

- 请参阅：

glDrawPixels(), **glTexImage1D()**, **glTexImage2D()**, **glTexImage3D()**, **gluBuild2DMipmaps()**,

gluBuild3DMipmaps(), **gluErrorString()**, **glGetTexImage()**, **glGetTexLevelParameter()**,
gluBuild1DMipmapLevels(), **gluBuild2DMipmapLevels()**, **gluBuild3DMipmapLevels()**

• **gluBuild2DMipmapLevels**

- 名称:

gluBuild2DMipmapLevels()

- 功能:

建立一个二维mipmap图层子集。

- C描述:

```
GLint gluBuild2DMipmapLevels( GLenum target,
                               GLint internalFormat,
                               GLsizei width,
                               GLsizei height,
                               GLenum format,
                               GLenum type,
                               GLint level,
                               GLint base,
                               GLint max,
                               const void *data )
```

- 参数说明:

target 指定目标纹理。必须是**GL_TEXTURE_2D**。

internalFormat

纹理图像所需要的内部存储格式。它必须是1、2、3、4或下面所列的符号常量之一: **GL_ALPHA**、**GL_ALPHA4**、**GL_ALPHA8**、**GL_ALPHA12**、**GL_ALPHA16**、**GL_LUMINANCE**、**GL_LUMINANCE4**、**GL_LUMINANCE8**、**GL_LUMINANCE12**、**GL_LUMINANCE16**、**GL_LUMINANCE_ALPHA**、**GL_LUMINANCE4_ALPHA4**、**GL_LUMINANCE6_ALPHA2**、**GL_LUMINANCE8_ALPHA8**、**GL_LUMINANCE12_ALPHA4**、**GL_LUMINANCE12_ALPHA12**、**GL_LUMINANCE16_ALPHA16**、**GL_INTENSITY**、**GL_INTENSITY4**、**GL_INTENSITY8**、**GL_INTENSITY12**、**GL_INTENSITY16**、**GL_RGB**、**GL_R3_G3_B2**、**GL_RGB4**、**GL_RGB5**、**GL_RGB8**、**GL_RGB10**、**GL_RGB12**、**GL_RGB16**、**GL_RGBA**、**GL_RGBA2**、**GL_RGBA4**、**GL_RGB5_A1**、**GL_RGBA8**、**GL_RGB10_A2**、**GL_RGBA12**或**GL_RGBA16**。

width, height

分别指定纹理图像的像素宽度和高度。它们必须是2的幂值。

format 指定像素数据的格式。它必须是**GL_COLOR_INDEX**、**GL_DEPTH**

_COMPONENT、GL_RED、GL_GREEN、GL_BLUE、GL_ALPHA、GL_RGB、GL_RGBA、GL_BGR、GL_BGRA、GL_LUMINANCE或GL_LUMINANCE_ALPHA。

type 指定*data*的数据类型。它必须是下面各值之一：**GL_UNSIGNED_BYTE、GL_BYTE、GL_BITMAP、GL_UNSIGNED_SHORT、GL_SHORT、GL_UNSIGNED_INT、GL_INT、GL_FLOAT、GL_UNSIGNED_BYTE_3_3_2、GL_UNSIGNED_BYTE_2_3_3_REV、GL_UNSIGNED_SHORT_5_6_5、GL_UNSIGNED_SHORT_5_6_5_REV、GL_UNSIGNED_SHORT_4_4_4、GL_UNSIGNED_SHORT_4_4_4_REV、GL_UNSIGNED_SHORT_5_5_5_1、GL_UNSIGNED_SHORT_1_5_5_5_REV、GL_UNSIGNED_INT_8_8_8_8、GL_UNSIGNED_INT_8_8_8_8_REV、GL_UNSIGNED_INT_10_10_10_2和GL_UNSIGNED_INT_2_10_10_10_REV。**

level 指定图像数据的mipmap图层。

base 指定传递给glTexImage2D()的最小mipmap图层。

max 指定传递给glTexImage2D()的最大mipmap图层。

data 指定一个指向内存中图像数据存放位置的指针。

- 说明：

例程gluBuild2DMipmapLevels()建立一个已预先滤波的二维纹理映射子集。该纹理映射通过调用一个mipmap，降低一个mipmap的分辨率来生成。这一操作可用来实现通过纹理映射而得到的图元的反走样。

当返回值是0时，表示该操作已经成功；否则，它将返回一个GLU的出错代码（请参阅gluErrorString()）。

递过每次减半*data*直至达到 1×1 为止，可以建立一组从*base*到*max*的Mipmap图层。在每一层中，减半的每个mipmap中的纹理像素是它上一个mipmap图层中相应的四个纹理像素的平均值。（在矩形图像中，最后将得到一个 $N \times 1$ 或 $1 \times N$ 的结构。这时，将由两个纹理像素来平均。）可以通过调用函数glTexImage2D()来载入这些从*base*到*max*的Mipmap图层。如果*max*大于给定尺寸的纹理的最大的mipmap图层，将返回一个GLU的出错代码（请参阅gluErrorString()），这时将不再建立任何mipmap。

例如，当*level*是2，*width*是16，*height*是8时，下面的图层是可能的： 16×8 、 8×4 、 4×2 、 2×1 和 1×1 。它们分别对于2、3、4、5和6层。当*base*是3，*max*是5时，则只有 8×4 、 4×2 和 2×1 被载入。然而，如果*max*是7，则由于它大于这里所提供的最大mipmap图层6，因此只产生一个出错代码而并不载入任何mipmap图层。

最大的mipmap图层可通过公式 $\log_2(\max(\text{width}, \text{height}) * (2^{\text{level}}))$ 而求得。

参数*format*的取值请参阅glTexImage1D()的参考说明。参数*type*的取值请参阅glDrawPixels()的参考说明。

- 注意：

例程gluBuild2DMipmapLevels()仅在GLU 1.3以上版本中才有效。

格式**GL_BGR**和**GL_BGRA**及类型**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**仅在GL 1.2以上的版本中才有效。

- 出错提示：

当参数*level*>*base*，*base*<0，*max*<*base*或*max*大于最高的mipmap图层时产生**GLU_INVALID_VALUE**提示。

当参数*width*<1或*height*<1时产生**GLU_INVALID_VALUE**提示。

当参数*internalFormat*、*format*或*type*不是一个合法值时产生**GLU_INVALID_ENUM**提示。

当参数*type*为**GL_UNSIGNED_BYTE_3_3_2**或**GL_UNSIGNED_BYTE_2_3_3_REV**，并且参数*format*不是**GL_RGB**格式时产生**GLU_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_SHORT_5_6_5**或**GL_UNSIGNED_SHORT_5_6_5_REV**，并且参数*format*不是**GL_RGB**格式时产生**GLU_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_SHORT_4_4_4_4**或**GL_UNSIGNED_SHORT_4_4_4_4_REV**，并且参数*format*既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_SHORT_5_5_5_1**或**GL_UNSIGNED_SHORT_1_5_5_5_REV**，并且参数*format*既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_INT_8_8_8_8**或**GL_UNSIGNED_INT_8_8_8_8_REV**，并且参数*format*既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_INT_10_10_10_2**或**GL_UNSIGNED_INT_2_10_10_10_REV**，参数*format*既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

- 请参阅：

glDrawPixels(), **glTexImage1D()**, **glTexImage2D()**, **glTexImage3D()**, **gluBuild1DMipmaps()**,
gluBuild2DMipmaps(), **gluBuild3DMipmaps()**, **gluErrorString()**, **glGetTexImage()**,
glGetTexLevelParameter(), **gluBuild1DMipmapLevels()**, **gluBuild3DMipmapLevels()**

gluBuild2DMipmaps

- 名称：

gluBuild2DMipmaps()

- 功能：

建立一个二维mipmap。

- C 描述:

```
GLint gluBuild2DMipmaps( GLenum target,
                          GLint internalFormat,
                          GLsizei width,
                          GLsizei height,
                          GLenum format,
                          GLenum type,
                          const void *data)
```

- 参数说明:

target 指定目标纹理。必须是**GL_TEXTURE_2D**。

internalFormat

纹理图像所需要的内部存储格式。它必须是1、2、3、4或下面所列的符号常量之一：

GL_ALPHA、**GL_ALPHA4**、**GL_ALPHA8**、**GL_ALPHA12**、**GL_ALPHA16**、**GL_LUMINANCE**、**GL_LUMINANCE4**、**GL_LUMINANCE8**、**GL_LUMINANCE12**、**GL_LUMINANCE16**、**GL_LUMINANCE_ALPHA**、**GL_LUMINANCE4_ALPHA4**、**GL_LUMINANCE6_ALPHA2**、**GL_LUMINANCE8_ALPHA8**、**GL_LUMINANCE12_ALPHA4**、**GL_LUMINANCE12_ALPHA12**、**GL_LUMINANCE16_ALPHA16**、**GL_INTENSITY**、**GL_INTENSITY4**、**GL_INTENSITY8**、**GL_INTENSITY12**、**GL_INTENSITY16**、**GL_RGB**、**GL_R3_G3_B2**、**GL_RGB4**、**GL_RGB5**、**GL_RGB8**、**GL_RGB10**、**GL_RGB12**、**GL_RGB16**、**GL_RGBA**、**GL_RGBA2**、**GL_RGBA4**、**GL_RGB5_A1**、**GL_RGBA8**、**GL_RGB10_A2**、**GL_RGBA12**或**GL_RGBA16**。

width, height

分别指定纹理图像的像素宽度和高度。

format 指定像素数据的格式。它必须是**GL_COLOR_INDEX**、**GL_DEPTH_COMPONENT**、**GL_RED**、**GL_GREEN**、**GL_BLUE**、**GL_ALPHA**、**GL_RGB**、**GL_RGBA**、**GL_BGR**、**GL_BGRA**、**GL_LUMINANCE**或**GL_LUMINANCE_ALPHA**。

type 指定*data*的数据类型。它的可取值如下：**GL_UNSIGNED_BYTE**、**GL_BYTE**、**GL_BITMAP**、**GL_UNSIGNED_SHORT**、**GL_SHORT**、**GL_UNSIGNED_INT**、**GL_INT**、**GL_FLOAT**、**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**。

data 指定一个指向内存中图像数据存放位置的指针。

- 说明：

例程`gluBuild2DMipmaps()`建立一个已预先滤波的二维纹理映射集。该纹理映射通过调用一个mipmap，降低一个mipmap的分辨率而生成。这一操作可用来实现通过纹理映射而得到的图元的反走样。

当返回值是0时，表示该操作已经成功；否则，它将返回一个GLU的出错代码（请参阅`gluErrorString()`）。

开始时，GL将检测数据的*width*和*height*是否是一个2的幂值。如果不是，*data*的一个拷贝（而非*data*本身）将缩放至最接近的2的幂值。（如果*width*或*height*正好是在两个2的幂值的正中间，则*data*的拷贝值将被缩放成较大的值。）这个拷贝值将用于随后的mipmap操作。例如，如果*width*是57，而*height*是23；则在进行mipmap操作前，*data*的一个拷贝将被放大成*width*为64，而*height*则被缩小为16。

然后，代理纹理（请参阅`glTexImage2D()`）将用来确定设备是否适合所要求的纹理。如果不适合，它们将继续减半直到适合为止。（如果OpenGL的版本≤1.0，则这两个最大值都将被截断成函数`glGetInteger(GL_MAX_TEXTURE_SIZE)`的返回值。）

接下来，通过每次减半*data*的拷贝直至达到1×1为止，可以建立一组mipmap图层。在每一层中，减半的每个mipmap中的纹理像素是它上一个mipmap图层中相应的四个纹理像素的平均值。（在矩形图像中，最后将得到一个N×1或1×N的结构。这时，将由两个纹理像素来平均。）

可以通过调用函数`glTexImage2D()`来载入这些mipmap图层。0层就是*data*的一个拷贝。最高层是 $\log_2(\max(width, height))$ 。例如，当*width*是64，*height*是16并且实现能够存放这种尺寸的纹理时，它将建立下面的mipmap层：64×16、32×8、16×4、8×2、4×1、2×1和1×1。它们分别对应0、1、2、3、4、5和6层。

参数*format*的取值请参阅`glTexImage1D()`的参考说明。参数*type*的取值请参阅`glDrawPixels()`的参考说明。

- 注意：

请注意，这里并没有直接查询最大层的方法。但它可以通过函数`glGetTexLevelParameter()`来间接查询：首先查询0层中实际使用的宽度和高度（由于代理纹理可能已将*width*和*height*缩放到适合实现所需的尺寸，因此这里的宽度和高度可能不等于*width*和*height*）。然后最大层可以通过公式 $\log_2(\max(width, height))$ 求得。

格式`GL_BGR`和`GL_BGRA`及类型`GL_UNSIGNED_BYTE_3_3_2`、`GL_UNSIGNED_BYTE_2_3_3_REV`、`GL_UNSIGNED_SHORT_5_6_5`、`GL_UNSIGNED_SHORT_5_6_5_REV`、`GL_UNSIGNED_SHORT_4_4_4`、`GL_UNSIGNED_SHORT_4_4_4_REV`、`GL_UNSIGNED_SHORT_5_5_5_1`、`GL_UNSIGNED_SHORT_1_5_5_5_REV`、`GL_UNSIGNED_INT_8_8_8_8`、`GL_UNSIGNED_INT_8_8_8_8_REV`、`GL_UNSIGNED_INT_10_10_10_2`和`GL_UNSIGNED_INT_2_10_10_10_REV`仅在GL 1.2以上的版本及GLU 1.3以上版本中才有效。

- 出错提示：

当参数*width*<1或*height*<1时产生`GLU_INVALID_VALUE`提示。

当参数*internalFormat*、*format*或*type*不是一个合法值时产生`GLU_INVALID_ENUM`提示。

当参数*type*为**GL_UNSIGNED_BYTE_3_3_2**或**GL_UNSIGNED_BYTE_2_3_3_REV**, 并且参数*format*不是**GL_RGB**格式时产生**GLU_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_SHORT_5_6_5**或**GL_UNSIGNED_SHORT_5_6_5_REV**, 并且参数*format*不是**GL_RGB**格式时产生**GLU_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_SHORT_4_4_4_4**或**GL_UNSIGNED_SHORT_4_4_4_4_REV**, 并且参数*format*既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_SHORT_5_5_5_1**或**GL_UNSIGNED_SHORT_1_5_5_5_REV**, 并且参数*format*既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_INT_8_8_8_8**或**GL_UNSIGNED_INT_8_8_8_8_REV**, 并且参数*format*既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_INT_10_10_10_2**或**GL_UNSIGNED_INT_2_10_10_10_REV**, 并且参数*format*既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

- 请参阅:

glDrawPixels(), **glTexImage1D()**, **glTexImage2D()**, **glTexImage3D()**, **gluBuild1DMipmaps()**,
gluBuild3DMipmaps(), **gluErrorString()**, **glGetTexImage()**, **glGetTexLevelParameter()**,
gluBuild1DMipmapLevels(), **gluBuild2DMipmapLevels()**, **gluBuild3DMipmapLevels()**

■ gluBuild3DMipmapLevels

- 名称:

gluBuild3DMipmapLevels()

- 功能:

建立一个三维mipmap图层子集。

- C描述:

GLint **gluBuild3DMipmapLevels(GLenum target,**

GLint *internalFormat*,

GLsizei *width*,

GLsizei *height*,

GLsizei *depth*,

GLenum *format*,

GLenum *type*,

GLint *level*,

GLint *base*,

GLint *max*,

const void **data*)

- 参数说明:

target 指定目标纹理。必须是**GL_TEXTURE_3D**。

internalFormat

纹理图像所需要的内部存储格式。它必须是1、2、3、4或下面所列的符号常量之一：**GL_ALPHA**、**GL_ALPHA4**、**GL_ALPHA8**、**GL_ALPHA12**、**GL_ALPHA16**、**GL_LUMINANCE**、**GL_LUMINANCE4**、**GL_LUMINANCE8**、**GL_LUMINANCE12**、**GL_LUMINANCE16**、**GL_LUMINANCE_ALPHA**、**GL_LUMINANCE4_ALPHA4**、**GL_LUMINANCE6_ALPHA2**、**GL_LUMINANCE8_ALPHA8**、**GL_LUMINANCE12_ALPHA4**、**GL_LUMINANCE12_ALPHA12**、**GL_LUMINANCE16_ALPHA16**、**GL_INTENSITY**、**GL_INTENSITY4**、**GL_INTENSITY8**、**GL_INTENSITY12**、**GL_INTENSITY16**、**GL_RGB**、**GL_R3_G3_B2**、**GL_RGB4**、**GL_RGB5**、**GL_RGB8**、**GL_RGB10**、**GL_RGB12**、**GL_RGB16**、**GL_RGBA**、**GL_RGBA2**、**GL_RGBA4**、**GL_RGB5_A1**、**GL_RGBA8**、**GL_RGB10_A2**、**GL_RGBA12**或**GL_RGBA16**。

width, height, depth

分别指定纹理图像的像素宽度、高度和深度。它们必须是2的幂值。

format 指定像素数据的格式。它必须是**GL_COLOR_INDEX**、**GL_DEPTH_COMPONENT**、**GL_RED**、**GL_GREEN**、**GL_BLUE**、**GL_ALPHA**、**GL_RGB**、**GL_RGBA**、**GL_BGR**、**GL_BGRA**、**GL_LUMINANCE**或**GL_LUMINANCE_ALPHA**。

type 指定*data*的数据类型。它必须是下面各值之一：**GL_UNSIGNED_BYTE**、**GL_BYTE**、**GL_BITMAP**、**GL_UNSIGNED_SHORT**、**GL_SHORT**、**GL_UNSIGNED_INT**、**GL_INT**、**GL_FLOAT**、**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**。

level 指定图像数据的mipmap图层。

base 指定传递给glTexImage3D()的最小mipmap图层。

max 指定传递给glTexImage3D()的最大mipmap图层。

data 指定一个指向内存中图像数据存放位置的指针。

• 说明：

例程gluBuild3DMipmapLevels()建立一个已预先滤波的三维纹理映射子集。该纹理映射通过调用一个mipmap，降低一个mipmap的分辨率而生成。这一操作可用来实现通过纹理映射而得

到的图元的反走样。

当返回值是0时，表示该操作已经成功；否则，它将返回一个GLU的出错代码（请参阅gluErrorString()）。

通过每次减半*data*直至达到 $1 \times 1 \times 1$ 为止，可以建立一组从*base*到*max*的mipmap图层。在每一层中，减半的每个mipmap中的纹理像素是它上一个mipmap层中相应的八个纹理像素的平均值。（如果其中的一个尺寸变成了1，则由四个纹理像素进行平均；如果其中的两个尺寸都变成了1，则由两个纹理像素进行平均。）可以通过调用函数glTexImage3D()来载入这些从*base*到*max*的Mipmap图层。如果*max*大于给定尺寸的纹理的最大的mipmap图层，将返回一个GLU的出错代码（请参阅gluErrorString()），这时将不再载入任何mipmap。

例如，当*level*是2，*width*是16，*height*是8，*depth*是4时，将建立下面的图层： $16 \times 8 \times 4$ 、 $8 \times 4 \times 2$ 、 $4 \times 2 \times 1$ 、 $2 \times 1 \times 1$ 和 $1 \times 1 \times 1$ 。它们分别对应2、3、4、5和6层。当*base*是3，*max*是5时，则只有映图图层 $8 \times 4 \times 2$ 、 $4 \times 2 \times 1$ 和 $2 \times 1 \times 1$ 被载入。然而，如果*max*是7，则由于它大于这里所提供的最大mipmap图层6，因此只产生一个出错代码而并不载入任何mipmap图层。

最大的mipmap图层可通过公式 $\log_2(\max(\text{width}, \text{height}, \text{depth})2^{\text{level}})$ 面求得。

参数*format*的取值请参阅glTexImage1D()的参考说明。参数*type*的取值请参阅glDrawPixels()的参考说明。

- 注意：

例程gluBuild3DMipmapLevels()仅在GLU 1.3以上版本中才有效。

格式**GL_BGR**和**GL_BGRA**及类型**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**仅在GL 1.2以上的版本中才有效。

- 出错提示：

当参数*level*>*base*，*base*<0，*max*<*base*或*max*大于最高的mipmap层时产生**GLU_INVALID_VALUE**提示。

当参数*width*<1、*height*<1或*depth*<1时产生**GLU_INVALID_VALUE**提示。

当参数*internalFormat*、*format*或*type*不是一个合法值时产生**GLU_INVALID_ENUM**提示。

当参数*type*为**GL_UNSIGNED_BYTE_3_3_2**或**GL_UNSIGNED_BYTE_2_3_3_REV**，并且参数*format*不是**GL_RGB**格式时产生**GLU_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_SHORT_5_6_5**或**GL_UNSIGNED_SHORT_5_6_5_REV**，并且参数*format*不是**GL_RGB**格式时产生**GLU_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_SHORT_4_4_4**或**GL_UNSIGNED_SHORT_4_4_4_REV**，并且参数*format*既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_SHORT_5_5_5_1**或**GL_UNSIGNED_SHORT_1_5_5_5**

REV, 并且参数*format*既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_INT_8_8_8_8**或**GL_UNSIGNED_INT_8_8_8_8_REV**, 并且参数*format*既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_INT_10_10_10_2**或**GL_UNSIGNED_INT_2_10_10_10_REV**, 并且参数*format*既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

- 请参阅:

glDrawPixels(), **glTexImage1D()**, **glTexImage2D()**, **glTexImage3D()**, **gluBuild1DMipmaps()**, **gluBuild2DMipmaps()**, **gluBuild3DMipmaps()**, **gluErrorString()**, **glGetTexImage()**, **glGetTexLevelParameter()**, **gluBuild1DMipmapLevels()**, **gluBuild2DMipmapLevels()**

• **gluBuild3DMipmaps**

- 名称:

gluBuild3DMipmaps()

- 功能:

建立一个三维mipmap。

- C描述:

```
GLint gluBuild3DMipmaps( GLenum target,
                          GLintut internalFormat,
                          GLsizei width,
                          GLsizei height,
                          GLsizei depth,
                          GLenum format,
                          GLenum type,
                          const void *data)
```

- 参数说明:

target 指定目标纹理。必须是**GL_TEXTURE_1D**。

internalFormat

纹理图像所需要的内部存储格式。它必须是1、2、3、4或下面所列的符号常量之一: **GL_ALPHA**、**GL_ALPHA4**、**GL_ALPHA8**、**GL_ALPHA12**、**GL_ALPHA16**、**GL_LUMINANCE**、**GL_LUMINANCE4**、**GL_LUMINANCE8**、**GL_LUMINANCE12**、**GL_LUMINANCE16**、**GL_LUMINANCE_ALPHA**、**GL_LUMINANCE4_ALPHA4**、**GL_LUMINANCE6_ALPHA2**、**GL_LUMINANCE8_ALPHA8**、**GL_LUMINANCE12_ALPHA4**、**GL_LUMINANCE12_ALPHA12**、**GL_LUMINANCE16_ALPHA16**、

GL_INTENSITY、**GL_INTENSITY4**、**GL_INTENSITY8**、**GL_INTENSITY12**、**GL_INTENSITY16**、**GL_RGB**、**GL_R3_G3_B2**、**GL_RGB4**、**GL_RGB5**、**GL_RGB8**、**GL_RGB10**、**GL_RGB12**、**GL_RGB16**、**GL_RGBA**、**GL_RGBA2**、**GL_RGBA4**、**GL_RGB5_A1**、**GL_RGBA8**、**GL_RGB10_A2**、**GL_RGBA12**或**GL_RGBA16**。

width, height, depth

分别指定纹理图像的像素宽度、高度和深度。

format 指定像素数据的格式。它必须是**GL_COLOR_INDEX**、**GL_DEPTH_COMPONENT**、**GL_RED**、**GL_GREEN**、**GL_BLUE**、**GL_ALPHA**、**GL_RGB**、**GL_RGBA**、**GL_BGR**、**GL_BGRA**、**GL_LUMINANCE**或**GL_LUMINANCE_ALPHA**。

type 指定*data*的数据类型。它的可取值如下：**GL_UNSIGNED_BYTE**、**GL_BYTE**、**GL_BITMAP**、**GL_UNSIGNED_SHORT**、**GL_SHORT**、**GL_UNSIGNED_INT**、**GL_INT**、**GL_FLOAT**、**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**。

data 指定一个指向内存中图像数据存放位置的指针。

• 说明：

例程**gluBuild3DMipmaps()**建立一个已预先滤波的三维纹理映射集。该纹理映射通过调用一个mipmap，降低一个mipmap的分辨率而生成。这一操作可用来实现通过纹理映射而得到的图元的反走样。

当返回值是0时，表示该操作已经成功；否则，它将返回一个GLU的出错代码（请参阅**gluErrorString()**）。

开始时，GL将检测数据的*width*、*height*和*depth*是否是一个2的幂值。如果不是，*data*的一个拷贝将被缩放至最接近的2的幂值。（如果*width*、*height*或*depth*正好是在两个2的幂值的正中间，则*data*的拷贝值将被放大成较大的值。）这个拷贝值将用于随后的mipmap操作。例如，如果*width*是57，*height*是23，而*depth*是24，则在进行mipmap操作前，*data*的一个拷贝将被放大成*width*为64，*height*被缩小为16，而*depth*则被放大为32。

然后，代理纹理（请参阅**glTexImage3D()**）将用来确定实现是否适合所要求的纹理。如果不适合，它们将继续减半直到适合为止。

通过每次减半*data*直至达到 $1 \times 1 \times 1$ 为止，可以建立一组mipmap图层。在每一层中，减半的每个mipmap中的纹理像素是它上一个mipmap图层中相应的八个纹理像素的平均值。（如果其中的一个尺寸变成了1，则由四个纹理像素进行平均；如果其中的两个尺寸都变成了1，则由两个

纹理像素进行平均。)

可以通过调用函数glTexImage3D()来载入这些mipmap图层。0级图层就是*data*的一个拷贝。最高层是 $\log_2(\max(width, height, depth))$ 。例如，当*width*是64, *height*是16, *depth*是32，并且实现能够存放这种尺寸的纹理时，它将建立下面的mipmap图层： $64 \times 16 \times 32$ 、 $32 \times 8 \times 16$ 、 $16 \times 4 \times 8$ 、 $8 \times 2 \times 4$ 、 $4 \times 1 \times 2$ 、 $2 \times 1 \times 1$ 和 $1 \times 1 \times 1$ 。它们分别对应0、1、2、3、4、5和6层。

参数*format*的取值请参阅glTexImage1D()的参考说明。参数*type*的取值请参阅glDrawPixels()的参考说明。

- 注意：

请注意，这里并没有直接查询最大层的方法。但它可以通过函数glGetTexLevelParameter()来间接查询：首先查询0层中实际使用的宽度（由于代理纹理可能已将*width*、*height*和*depth*缩放到适合实现需要的尺寸，因此这里的宽度、高度和深度可能不等于*width*、*height*和*depth*）。然后最大层可以通过公式 $\log_2(\max(width, height, depth))$ 求得。

例程gluBuild3DMipmaps()仅在GLU 1.3以上版本中才有效。

格式**GL_BGR**和**GL_BGRA**及类型**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**仅在GL 1.2以上的版本及GLU 1.3以上版本中才有效。

- 出错提示：

当参数*width*、*height*或*depth*<1时产生**GLU_INVALID_VALUE**提示。

当参数*internalFormat*、*format*或*type*不是一个合法值时产生**GLU_INVALID_ENUM**提示。

当参数*type*为**GL_UNSIGNED_BYTE_3_3_2**或**GL_UNSIGNED_BYTE_2_3_3_REV**，并且参数*format*不是**GL_RGB**格式时产生**GLU_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_SHORT_5_6_5**或**GL_UNSIGNED_SHORT_5_6_5_REV**，并且参数*format*不是**GL_RGB**格式时产生**GLU_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_SHORT_4_4_4_4**或**GL_UNSIGNED_SHORT_4_4_4_4_REV**，并且参数*format*既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_SHORT_5_5_5_1**或**GL_UNSIGNED_SHORT_1_5_5_5_REV**，并且参数*format*既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_INT_8_8_8_8**或**GL_UNSIGNED_INT_8_8_8_8_REV**，并且参数*format*既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

当参数*type*为**GL_UNSIGNED_INT_10_10_10_2**或**GL_UNSIGNED_INT_2_10_10_10_REV**

REV, 并且参数*format*既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

- 请参阅:

glDrawPixels(), **glTexImage1D()**, **glTexImage2D()**, **glTexImage3D()**, **gluBuild1DMipmaps()**,
gluBuild2DMipmaps(), **gluErrorString()**, **glGetTexImage()**, **glGetTexLevelParameter()**,
gluBuild1DMipmapLevels(), **gluBuild2DMipmapLevels()**, **gluBuild3DMipmapLevels()**

• **gluCheckExtension**

- 名称:

gluCheckExtension()

- 功能:

确定是否支持一个扩展名。

- C描述:

```
GLboolean gluCheckExtension( const GLubyte *extName,
                             const GLbyte *extString )
```

- 参数说明:

extName 指定一个扩展名。

extString 指定一个用空格隔开的被支持的扩展名清单。

- 说明:

当支持*extName*时, 例程**gluCheckExtension()**返回**GL_TRUE**, 否则, 返回**GL_FALSE**。

该例程可以通过**glGetString()**、**gluGetString()**、**glXGetClientString()**、**glXQueryExtensionsString()**或**glXQueryServerString()**返回的扩展字符串*extString*, 分别检验OpenGL、GLU或GLX的扩展名的有效性。

- 注意:

当一个扩展名是另一个的子串时, 它将能被正确地处理。

在*extString*中可能会在前面或后面出现空格。

扩展名中不能有嵌入的空格。

所有字符串都是以NULL结束。

- 请参阅:

glGetString(), **gluGetString()**, **glXGetClientString()**, **glXQueryExtensionsString()**

• **gluCylinder**

- 名称:

gluCylinder()

- 功能:

绘制圆柱。

- C描述:

```
void gluCylinder( GLUquadric* quad,
```

```
GLdouble base,
GLdouble top,
GLdouble height,
GLint slices,
GLint stacks )
```

• 参数说明：

quad 指定二次对象（由gluNewQuadric()建立）。

base 指定 $z=0$ 处的圆柱半径。

top 指定 $z=height$ 处的圆柱半径。

height 指定圆柱的高度。

slices 指定围绕 z 轴细分的数目。

stacks 指定沿 z 轴细分的数目。

• 说明：

例程gluCylinder()将绘制一个沿 z 轴方向的圆柱。圆柱的底在 $z=0$ 处，顶在 $z=height$ 处。与球体类似，圆柱体沿 z 轴方向被细分成片状，绕 z 轴方向被细分成层状。

请注意，当 top 被设置成0.0时，该例程将生成一个圆锥体。

当该圆柱的方向设置成GLU_OUTSIDE（用gluQuadricOrientation()）时，任一条生成的法线都由 z 轴向外指；否则，它们将指向 z 轴。

如果启动了纹理功能（用gluQuadricTexture()），它将生成纹理坐标。这时纹理坐标*t*将在如下的线性范围内：从 $z=0$ 处的0.0到 $z=height$ 处的1.0；类似地，纹理坐标*s*的范围是：从 $+y$ 轴处为0.0，到 $+x$ 轴处为0.25，到 $-y$ 轴处为0.5，然后到 $-x$ 轴处为0.75，最后又回到 $+y$ 轴处成为了1.0。

• 请参阅：

gluDisk(), gluNewQuadric(), gluPartialDisk(), gluQuadricTexture(), gluSphere()

• gluDeleteNurbsRenderer

• 名称：

gluDeleteNurbsRenderer()

• 功能：

删除一个NURBS对象。

• C描述：

```
void gluDeleteNurbsRenderer( GLUnurbs* nurb )
```

• 参数说明：

nurb 指定要删除的NURBS对象。

• 说明：

例程gluDeleteNurbsRenderer()用来删除一个NURBS对象（该对象由gluNewNurbsRenderer()建立）并释放它所占用的所有内存空间。一旦调用例程gluDeleteNurbsRenderer()，*nurb*将不能再被使用。

- 请参阅:

gluNewNurbsRenderer()

◆ **gluDeleteQuadric**

- 名称:

gluDeleteQuadric()

- 功能:

删除一个二次曲面对象。

- C描述:

void gluDeleteQuadric(GLUquadric*quad)

- 参数说明:

quad 指定要删除的二次曲面对象。

- 说明:

例程**gluDeleteQuadric()**用来删除一个二次曲面对象（该对象由**gluNewQuadric()**建立）并释放它所占用的所有内存空间。一旦调用例程**gluDeleteQuadric()**，*quad*将不能再被使用。

- 请参阅:

gluNewQuadric()

◆ **gluDeleteTess**

- 名称:

gluDeleteTess()

- 功能:

删除一个镶嵌分块对象。

- C描述:

void gluDeleteTess(GLUtesselator*tess)

- 参数说明:

tess 指定要删除的镶嵌分块对象。

- 说明:

例程**gluDeleteTess()**用来删除一个镶嵌分块对象（该对象由**gluNewTess()**建立）并释放它所占用的所有内存空间。

- 请参阅:

gluBeginPolygon(), **gluNewTess()**, **gluTessCallback()**

◆ **gluDisk**

- 名称:

gluDisk()

- 功能:

绘制圆盘。

- C描述:

```
void gluDisk( GLUquadric* quad,
              GLdouble inner,
              GLdouble outer,
              GLint slices,
              GLint loops )
```

- 参数说明:

quad 指定二次曲面对象 (由gluNewQuadric()建立)。
inner 指定圆盘的内部半径 (可能是0)。
outer 指定圆盘的外部半径。
slices 指定围绕z轴细分的数目。
loops 指定绕原点周围细分的同心环的数目。

- 说明:

例程gluDisk()在 $z=0$ 平面上绘制一个圆盘。该圆盘的外半径是*outer*, 并包含一个半径为*inner*的同心圆孔。该圆盘环绕z轴被细分成片状 (类似于比萨饼的薄片) 并且在z轴周围细分成环状 (它们的数目分别由*slices*和*loops*指定)。

至于它的方向, 圆盘的 $+z$ 轴一边被认为是它的“外边” (请参阅gluQuadricOrientation())。这就意味着如果它的方向被设置成GLU_OUTSIDE, 则生成的任何法线方向都是指向 $+z$ 轴方向的。否则, 它们将指向 $-z$ 轴方向。

如果启动了纹理功能 (用gluQuadricTexture()), 它将生成纹理坐标。这时纹理坐标*r*将通过下面方式线性地生成: 在 $(r, 0, 0)$ 处为 $(1, 0.5)$, 在 $(0, r, 0)$ 处为 $(0.5, 1)$, 在 $(-r, 0, 0)$ 处为 $(0, 0.5)$, 在 $(0, -r, 0)$ 处为 $(0.5, 0)$, 这里 $r = \text{outer}$ 。

- 请参阅:

gluCylinder(), **gluNewQuadric()**, **gluPartialDisk()**, **gluQuadricOrientation()**,
gluQuadricTexture(), **gluSphere()**

***gluErrorString**

- 名称:

gluErrorString()

- 功能:

产生一个来自GL或GLU出错代码的出错字符串。

- C描述:

```
const GLubyte * gluErrorString( GLenum error )
```

- 参数说明:

error 指定一个GL或GLU出错代码。

- 说明:

例程gluErrorString()将产生一个来自GL或GLU出错代码的出错字符串。该字符串为ISO

Latin 1格式。例如，**gluErrorString(GL_OUT_OF_MEMORY)**返回字符串“*out of memory*”。

标准的GLU出错代码是**GLU_INVALID_ENUM**、**GLU_INVALID_VALUE**和**GLU_OUT_OF_MEMORY**。某些其他的GLU函数可以通过回调返回特定的出错代码。有关GL的出错代码请参阅**glGetError()**的参考说明。

- 出错提示

如果参数*error*不是一个有效的GL或GLU出错代码，则返回NULL。

- 请参阅：

glGetError(), **gluNurbsCallback()**, **gluQuadricCallback()**, **gluTessCallback()**

• **gluGetNurbsProperty**

- 名称：

gluGetNurbsProperty()

- 功能：

获取一个NURBS特性。

- C描述：

```
void gluGetNurbsProperty( GLUnurbs* nurb,
                           GLenum property,
                           GLfloat* data )
```

- 参数说明：

nurb 指定NURBS对象（由**gluNewNurbsRenderer()**建立）。

property 指定将获取的特性值。它的可取有效值有：**GLU_CULLING**、**GLU_SAMPLING_TOLERANCE**、**GLU_DISPLAY_MODE**、**GLU_AUTO_LOAD_MATRIX**、**GLU_PARAMETRIC_TOLERANCE**、**GLU_SAMPLING_METHOD**、**GLU_U_STEP**、**GLU_V_STEP**和**GLU_NURBS_MODE**。

data 指定一个指针，指向指定的特性值所要写入的存储单元。

- 说明：

例程**gluGetNurbsProperty()**将返回一个NURBS对象所具有的特性。这些特性将影响NURBS曲线和曲面的绘制。具体特性及它们的作用请参阅**gluNurbsProperty()**的参考说明。

- 请参阅：

gluNewNurbsRenderer(), **gluNurbsProperty()**

• **gluGetString**

- 名称：

gluGetString()

- 功能：

返回一个描述GL版本或GLU扩展的字符串。

- C描述：

```
const GLubyte* gluGetString( GLenum name )
```

- 参数说明：

name 指定一个符号常量。它可以是**GL_VERSION**或**GL_EXTENSIONS**。

- 说明：

函数**glGetString()**返回一个指针，该指针指向用来描述所支持的GL版本或GLU扩展的一个静态字符串。

版本号使用下列形式之一：

major_number.minor_number

major_number.minor_number.release_number

版本字符串的形式如下：

version number <space> vendor-specific information

其中特定的卖方信息是可选的。它的格式和内容与具体所采用的机制有关。

标准的GLU中包含有一个基本的特性和功能集。当一个或一批公司希望支持其他的特性时，这些特性可以被包含在GLU扩展中。当*name*是**GL_EXTENSIONS**时，**gluGetString()**将返回一个被支持的用空格隔开的GLU扩展名。（扩展名中不能包含空格。）

所有字符串都是NULL结束的。

- 注意：

gluGetString()只能返回有关GLU扩展的信息。请调用函数**glGetString()**来获取一个GL扩展的清单。

gluGetString()是一个初始化命令。如果在**glNewList()**后面调用它，将导致不确定的行为。

- 出错提示：

当*name*不是**GLU_VERSION**或**GLU_EXTENSIONS**时，将返回NULL提示。

- 请参阅：

glGetString()

gluGetTessProperty

- 名称：

gluGetTessProperty()

- 功能：

获取一个镶嵌分块对象特性。

- C描述：

```
void gluGetTessProperty( GLUtesselator* tess,
                         GLenum which,
                         GLdouble* data )
```

- 参数说明：

tess 指定镶嵌分块对象（由**gluNewTess()**建立）。

which 指定将获取的特性值。它的可取有效值有：**GLU_TESS_WINDING_RULE**、**GLU_TESS_BOUNDARY_ONLY**和**GLU_TESS_TOLERANCE**。

data 指定一个指针，指向指定的特性值所要写入的存储单元。

- 说明：

例程**gluGetTessProperty()**将返回一个镶嵌分块对象所具有的特性。这些特性将影响镶嵌分块对象的编译和绘制。具体特性及它们的作用请参阅**gluTessProperty()**的参考说明。

- 请参阅：

gluNewTess(), **gluTessProperty()**

• **gluLoadSamplingMatrices**

- 名称：

gluLoadSamplingMatrices()

- 功能：

载入NURBS取样及拣选矩阵。

- C描述：

```
void gluLoadSamplingMatrices( GLUnurbs* nurb,
                               const GLfloat *model,
                               const GLfloat *perspective,
                               const GLint *view )
```

- 参数说明：

nurb 指定NURBS对象（由**gluNewNurbsRenderer()**建立）。

model 指定一个模式取景矩阵（通过调用**glGetFloatv()**而得到）。

perspective 指定一个投影矩阵（通过调用**glGetFloatv()**而得到）。

view 指定一个视口（通过调用**glGetIntegerv()**而得到）。

- 说明：

例程**gluLoadSamplingMatrices()**用*model*, *perspective*和*view*来重新计算取样和拣选矩阵并将它们存入*nurb*中。取样矩阵用来确定一个NURBS曲线或曲面应该被怎样镶嵌分块才能满足取样误差的要求（取样误差由**GLU_SAMPLING_TOLERANCE**确定）。拣选矩阵用来决定绘制一个NURBS曲线或曲面之前是否需要拣选（当启动**GLU_CULLING**属性时）。

例程**gluLoadSamplingMatrices()**只有在**GLU_AUTO_LOAD_MATRIX**属性关闭时才需要（请参阅**gluNurbsProperty()**）。尽管当**GLU_AUTO_LOAD_MATRIX**属性开启时可以方便地使用该例程，但同时它也有一些性能损失。（它将循环访问GL服务器以获取模式取景矩阵、投影矩阵以及视口的当前值。）

- 请参阅：

gluGetNurbsProperty(), **gluNewNurbsRenderer()**, **gluNurbsProperty()**

• **gluLookAt**

- 名称：

gluLookAt()

- 功能：

定义一个视点变换。

- C描述：

```
void gluLookAt( GLdouble eyeX,
                 GLdouble eyeY,
                 GLdouble eyeZ,
                 GLdouble centerX,
                 GLdouble centerY,
                 GLdouble centerZ,
                 GLdouble upX,
                 GLdouble upY,
                 GLdouble upZ )
```

- 参数说明：

eyeX, eyeY, eyeZ

指定视点的位置。

centerX, centerY, centerZ

指定参考点的位置。

upX, upY, upZ

指定up向量的方向。

- 说明：

例程gluLookAt()通过一个视点、一个表示场景中心的参考点和UP向量建立一个取景矩阵。

该矩阵将参考点映射为 $-z$ 轴方向，将视点映射为原点。当使用一个典型的投影矩阵时，场景的中心便映射到了视口的中心。类似地，UP向量在观察平面上的投影方向将被映射到 $+y$ 轴的方向，也就是说它指向视口的上方。UP向量的方向不能与视点到参考点的连线方向平行。

设：

$$\begin{aligned} F = & \begin{matrix} centerX - eyeX \\ centerY - eyeY \\ centerZ - eyeZ \end{matrix} \end{aligned}$$

并且设UP为向量 (*upX, upY, upZ*)，

则，归一化方式如下： $f = \frac{F}{\|F\|}$

最后，设 $s=f \times UP'$ ， $u=s \times f$ ，

$$UP' = \frac{UP}{\|UP\|}$$

于是，*M*的结构如下：

$$\begin{aligned} M = & \begin{matrix} s[0] & s[1] & s[2] & 0 \\ u[0] & u[1] & u[2] & 0 \\ -f[0] & -f[1] & -f[2] & 0 \\ 0 & 0 & 0 & 1 \end{matrix} \end{aligned}$$

例程**gluLookAt()**的作用等价于：

```
glMultMatrixf(M);
glTranslated(-eyex, -eyey, -eyez);
```

- 请参阅：

glFrustum(), **gluPerspective()**

• **gluNewNurbsRenderer**

- 名称：

gluNewNurbsRenderer()

- 功能：

建立一个NURBS对象。

- C描述：

GLUnurbs* gluNewNurbsRenderer(void)

- 说明：

例程**gluNewNurbsRenderer()**用来建立并返回一个指向新的NURBS对象的指针。当调用NURBS的绘图和控制函数时，必须提供这个对象。当返回值是0时，表示没有足够的内存空间来存放这个对象。

- 请参阅：

gluBeginCurve(), **gluBeginSurface()**, **gluBeginTrim()**, **gluDeleteNurbsRenderer()**,
gluNurbsCallback(), **gluNurbsProperty()**

• **gluNewQuadric**

- 名称：

gluNewQuadric()

- 功能：

建立一个二次曲面对象。

- C描述：

GLUquadric* gluNewQuadric(void)

- 说明：

例程**gluNewQuadric**用来建立并返回一个指向新的二次曲面对象的指针。当调用二次曲面的绘图和控制函数时，必须提供这个对象。当返回值是0时，表示没有足够的内存空间来存放这个对象。

- 请参阅：

gluCylinder(), **gluDeleteQuadric()**, **gluDisk()**, **gluPartialDisk()**, **gluQuadricCallback()**,
gluQuadricDrawStyle(), **gluQuadricNormals()**, **gluQuadricOrientation()**, **gluQuadricTexture()**,
gluSphere()

• **gluNewTess**

- 名称：

gluNewTess()

- 功能：

建立一个镶嵌分块对象。

- C描述：

GLUTesselator* gluNewTess(void)

- 说明：

例程**gluNewTess()**用来建立并返回一个指向新的镶嵌分块对象的指针。当调用镶嵌分块函数时，必须提供这个对象。当返回一个0值时，表示没有足够的内存空间来存放这个对象。

- 请参阅：

gluTessBeginPolygon(), gluDeleteTess(), gluTessCallback()

• **gluNurbsCallback**

- 名称：

gluNurbsCallback()

- 功能：

定义一个NURBS对象的回调。

- C描述：

```
void gluNurbsCallback( GLUnurbs* nurb,
                        GLenum which,
                        GLvoid (*CallBackFunc))
```

- 参数说明：

nurb 指定NURBS对象（由**gluNewNurbsRenderer()**建立）。

which 指定被定义的回调。它的有效值有：**GLU_NURBS_BEGIN**、**GLU_NURBS_VERTEX**、**GLU_NURBS_NORMAL**、**GLU_NURBS_COLOR**、**GLU_NURBS_TEXTURE_COORD**、**GLU_NURBS_END**、**GLU_NURBS_BEGIN_DATA**、**GLU_NURBS_VERTEX_DATA**、**GLU_NURBS_NORMAL_DATA**、**GLU_NURBS_COLOR_DATA**、**GLU_NURBS_TEXTURE_COORD_DATA**、**GLU_NURBS_END_DATA**和**GLU_NURBS_ERROR**。

CallBackFunc

指定回调所调用的函数。

- 说明：

例程**gluNurbsCallback()**用来定义一个NURBS对象所使用的回调。如果被指定的回调已经被定义，那么它将被替换。当*CallBackFunc*是NULL时，该回调将不被调用，这时如果存在相关的数据，那么这些相关数据将丢失。

除了出错回调之外，其他的回调将用于NURBS的镶嵌分块操作（当将**GLU_NURBS**

_MODE设置成**GLU_NURBS_TESSELLATOR**时), 并返回由镶嵌分块所得到的OpenGL多边形图元。值得注意的是每个回调都有两种形式: 一种形式带有用户数据指针而另一种形式则没有。如果某个特定的回调被指定了两种形式, 那么带有用户数据指针的形式将被使用。注意这里的“用户数据”是最后一次调用**gluNurbsCallbackData()**时所指定的指针的一个拷贝。

不管**GLU_NURBS_MODE**被设置成什么值, 出错回调函数都将有效。除此之外的其他所有回调函数都只有当**GLU_NURBS_MODE**被设置成**GLU_NURBS_TESSELLATOR**时才有效。

有效的回调形式如下:

GLU_NURBS_BEGIN

开始回调表示一个图元的初始。该函数带有一个GLenum类型的自变量, 这个自变量可以是**GL_LINES**, **GL_LINE_STRIP**, **GL_TRIANGLE_FAN**, **GL_TRIANGLE_STRIP**, **GL_TRIANGLES**或**GL_QUAD_STRIP**。缺省的开始回调函数是NULL。该回调的函数原型类似于:

```
void beginData (GLenum type, void *userData);
```

GLU_NURBS_BEGIN_DATA

除了它带有一个附加的指针自变量外, 它的用法同**GLU_NURBS_BEGIN**完全一样。该指针是最后一次调用**gluNurbsCallbackData()**时所指定的指针的一个拷贝。这个回调函数的缺省值是NULL。该回调的函数原型类似于:

```
void begin ( GLenum type );
```

GLU_NURBS_VERTEX

顶点回调表示一个图元的顶点。该顶点的坐标存放在参数“vertex”中。所有生成的顶点的维数都是3, 也就是说齐次坐标要转换成仿射几何坐标。缺省的顶点回调函数是NULL。该回调的函数原型类似于:

```
void vertex ( GLfloat *vertex );
```

GLU_NURBS_VERTEX_DATA

除了它带有一个附加的指针自变量外, 它的用法同**GLU_NURBS_VERTEX**完全一样。该指针是最后一次调用**gluNurbsCallbackData()**时所指定的指针的一个拷贝。这个回调函数的缺省值是NULL。该回调的函数原型类似于:

```
void vertexData ( GLfloat *vertex, void *userData );
```

GLU_NURBS_NORMAL

当生成顶点的法线时将调用法线回调。法线的组分存放在参数“normal”中。在NURBS曲线的情况下, 只有当用户提供了一个法线映射 (**GL_MAP1_NORMAL**) 时该反馈函数才有效。在NURBS曲面的情况下, 只有当用户提供了一个法线映射 (**GL_MAP2_NORMAL**) 时才能由法线映射计算出要生成的法线。如果没有提供一个法线映射, 则曲面法线的生成方式类似于**GL_AUTO_NORMAL**启动时用求值器生成法线的方式。缺省的法线回调函数是NULL。该回调的函数原型类似于:

```
void normal ( GLfloat *normal );
```

GLU_NURBS_NORMAL_DATA

除了它带有一个附加的指针自变量外，它的用法同**GLU_NURBS_NORMAL**完全一样。该指针是最后一次调用**gluNurbsCallbackData()**时所指定的指针的一个拷贝。这个回调函数的缺省值是**NULL**。该回调的函数原型类似于：

```
void normalData ( GLfloat *normal, void *userData );
```

GLU_NURBS_COLOR

当生成顶点的颜色时将调用颜色回调。颜色的组分存放在参数“color”中。只有当用户提供了一个颜色映射（**GL_MAP1_COLOR_4**或**GL_MAP2_COLOR_4**）时该回调函数才有效。“颜色”包含四个组分：R、G、B和A。缺省的颜色回调函数是**NULL**。该回调的函数原型类似于：

```
void color ( GLfloat *color );
```

GLU_NURBS_COLOR_DATA

除了它带有一个附加的指针自变量外，它的用法同**GLU_NURBS_COLOR**完全一样。该指针是最后一次调用**gluNurbsCallbackData()**时所指定的指针的一个拷贝。这个回调函数的缺省值是**NULL**。该回调的函数原型类似于：

```
void colorData ( GLfloat *color, void *userData );
```

GLU_NURBS_TEXTURE_COORD

当生成顶点的纹理坐标时将调用纹理回调。这些坐标存放在参数“texCoord”中。纹理坐标的个数可以是1、2、3或4，具体由指定的纹理映射类型（**GL_MAP1_TEXTURE_COORD_1**, **GL_MAP1_TEXTURE_COORD_2**, **GL_MAP1_TEXTURE_COORD_3**, **GL_MAP1_TEXTURE_COORD_4**, **GL_MAP2_TEXTURE_COORD_1**, **GL_MAP2_TEXTURE_COORD_2**, **GL_MAP2_TEXTURE_COORD_3**, **GL_MAP2_TEXTURE_COORD_4**）所确定。如果没有指定纹理映射，该回调函数将不被调用。缺省的纹理回调函数是**NULL**。该回调的函数原型类似于：

```
void texCoord ( GLfloat *texCoord );
```

GLU_NURBS_TEXTURE_COORD_DATA

除了它带有一个附加的指针自变量外，它的用法同**GLU_NURBS_TEXTURE_COORD**完全一样。该指针是最后一次调用**gluNurbsCallbackData()**时所指定的指针的一个拷贝。这个回调函数的缺省值是**NULL**。该回调的函数原型类似于：

```
void texCoordData ( GLfloat *texCoord, void *userData );
```

GLU_NURBS_END

当一个图元结束时调用结束回调。缺省的结束回调函数是**NULL**。该回调的函数原型类似于：

```
void end ( void );
```

GLU_NURBS_END_DATA

除了它带有一个附加的指针自变量外，它的用法同**GLU_NURBS_END**完全一样。该指针是最后一次调用**gluNurbsCallbackData()**时所指定的指针的一个拷贝。这

```
void endData ( void *userData );
```

个回调函数的缺省值是NULL。该回调的函数原型类似于：

GLU_NURBS_ERROR

当遇到一个错误时将调用出错回调。该函数带有一个GLenum类型的自变量，这个自变量表示所发生的特定错误。对于NURBS而言，它有从**GLU_NURBS_ERROR1**到**GLU_NURBS_ERROR37**共37种错误。用户可以遇过例程**gluErrorString()**返回这些错误的特征字符串。

- 注意：

例程**gluNurbsCallback()**仅在GLU 1.2以上的版本中才有效。

GLU 1.2仅支持**GLU_ERROR**参数*which*。**GLU_ERROR**在GLU 1.3中已经不再使用而由**GLU_NURBS_ERROR**代替。*CallBackFunc*的其他所有可接受的值都只在GLU 1.3以上的版本中才有效。

- 请参阅：

gluErrorString(), **gluNewNurbsRenderer()**, **gluNurbsCallbackData()**, **gluNurbsProperty()**

gluNurbsCallbackData

- 名称：

gluNurbsCallbackData()

- 功能：

设置一个用户数据指针。

- C描述：

```
void gluNurbsCallbackData( GLUnurbs* nurb,
                           GLvoid* userData )
```

- 参数说明：

nurb 指定NURBS对象（由**gluNewNurbsRenderer()**建立）。

userData 指定一个指向用户数据的指针。

- 说明：

例程**gluNurbsCallbackData()**用来将指向应用程序数据的指针传送给NURBS镶嵌分块操作。该指针的一个拷贝将通过NURBS回调函数的镶嵌分块操作来传送（请参阅**gluNurbsCallback()**）。

- 注意：

例程**gluNurbsCallbackData()**仅在GLU 1.3以上的版本中才有效。

- 请参阅：

gluNewNurbsRenderer(), **gluNurbsCallback()**

gluNurbsCallbackDataEXT

- 名称：

gluNurbsCallbackDataEXT()

- 功能：

设置一个用户数据指针。

- C描述：

```
void gluNurbsCallbackDataEXT( GLUnurbs* nurb,
                               GLvoid* userData )
```

- 参数说明：

nurb 指定NURBS对象（由gluNewNurbsRenderer()建立）。

userData 指定一个指向用户数据的指针。

- 说明：

例程gluNurbsCallbackDataEXT()用来将指向应用程序数据的指针传送给NURBS镶嵌分块操作。该指针的一个拷贝将通过NURBS回调函数的镶嵌分块操作来传送（请参阅gluNurbsCallback()）。

- 请参阅：

gluNurbsCallback()

gluNurbsCurve

- 名称：

gluNurbsCurve()

- 功能：

定义了一个NURBS曲线的外形。

- C描述：

```
void gluNurbsCurve( GLUnurbs* nurb,
                     GLint knotCount,
                     GLfloat* knots,
                     GLint stride,
                     GLfloat* control,
                     GLint order,
                     GLenum type )
```

- 参数说明：

nurb 指定NURBS对象（由gluNewNurbsRenderer()建立）。

knotCount

指定*knots*中结点的数目。参数*knotCount*等于控制点的数目加上阶数。

knots 指定一个*knotCount*的不减少结点值的数组。

stride 指定连续的曲线控制点之间的偏移量（它是一个单精度浮点型的数值）。

control 指定一个指向控制点数组的指针。其坐标必须适合于下面指定的类型*type*。

order 指定NURBS曲线的阶数。参数*order*等于度数+1。要得到一个立体曲线，其阶数

应该是4。

type 指定曲线的类型。如果该曲线是在例程对gluBeginCurve()/gluEndCurve()之间定义的，其类型可以是任意有效的一维求值器的类型（如GL_MAP1_VERTEX_3或GL_MAP1_COLOR_4）。如果该曲线是在例程对gluBeginTrim()/gluEndTrim()之间定义的，则它的有效类型只有GLU_MAP1_TRIM_2和GLU_MAP1_TRIM_3。

- 说明：

例程gluNurbsCurve()用来描述一条NURBS曲线。

当例程gluNurbsCurve()出现在例程对gluBeginCurve()/gluEndCurve()之间时，它用来描述一个将要绘制的曲线。例程对gluBeginCurve()/gluEndCurve()之间的每个单独的例程gluNurbsCurve()都提供了相应的位置、纹理和颜色坐标。在例程对gluBeginCurve()/gluEndCurve()之间可以不止一次地调用例程gluNurbsCurve()来生成颜色、位置和纹理数据。每一次正确的调用都应该指出曲线的位置（对于每一个GL_MAP1_VERTEX_3或GL_MAP1_VERTEX_4的*type*）。出现在例程对gluBeginTrim()/gluEndTrim()之间的例程gluNurbsCurve()用来描述一个NURBS曲面上的修整曲线。如果*type*是GLU_MAP1_TRIM_2，则它描述一条二维（*u*和*v*）参数空间中的曲线。如果*type*是GLU_MAP1_TRIM_3，则它描述一条二维齐次（*u*、*v*和*w*）参数空间中的曲线。有关修整曲线的详细内容请参阅gluBeginTrim()的参考说明。

- 示例：

下面的命令用来绘制一条带法线的纹理化NURBS曲线：

```
gluBeginCurve(nobj);
    gluNurbsCurve(nobj, ..., GL_MAP1_TEXTURE_COORD_2);
    gluNurbsCurve(nobj, ..., GL_MAP1_NORMAL);
    gluNurbsCurve(nobj, ..., GL_MAP1_VERTEX_4);
gluEndCurve(nobj);
```

- 注意：

定义一个已经“缝合好”的修整曲线，请使用例程gluPwlCurve()。

- 请参阅：

gluBeginCurve(), **gluBeginTrim()**, **gluNewNurbsReuderer()**, **gluPwlCurve()**

gluNurbsProperty

- 名称：

gluNurbsProperty()

- 功能：

设置一条NURBS对象的特性。

- C描述：

```
void gluNurbsProperty( GLUnurbs* nurb,
                        GLenum property,
                        GLfloat value )
```

• 参数说明：

- nurb* 指定NURBS对象（由gluNewNurbsRenderer()建立）。
- property* 指定要设置的特性。它的可取值有：GLU_SAMPLING_TOLERANCE, GLU_DISPLAY_MODE, GLU_CULLING, GLU_AUTO_LOAD_MATRIX, GLU_PARAMETRIC_TOLERANCE, GLU_SAMPLING_METHOD, GLU_U_STEP, GLU_V_STEP或GLU_NURBS_MODE。
- value* 指定所表示的特性值。它可以是一个数字值或下面各值之一：GLU_OUTLINE_Polygon, GLU_FILL, GLU_OUTLINE_PATCH, GLU_TRUE, GLU_FALSE, GLU_PATH_LENGTH, GLU_PARAMETRIC_ERROR, GLU_DOMAIN_DISTANCE, GLU_NURBS_RENDERER或GLU_NURBS_TESSELLATOR。

• 说明：

例程gluNurbsProperty()用来控制一个NURBS对象所具有的特性。这些特性将影响NURBS曲线的绘制。参数*property*可以取下面各值：

GLU_NURBS_MODE

参数*value*应设置成GLU_NURBS_RENDERER或GLU_NURBS_TESSELLATOR。当设置成GLU_NURBS_RENDERER时，NURBS对象将被镶嵌分块成OpenGL图元，并被送入流程中进行绘制。当设置成GLU_NURBS_TESSELLATOR时，NURBS对象将被镶嵌分块成OpenGL图元，但这时的顶点、法线、颜色、和（或）纹理将由一个反馈接口返回（请参阅gluNurbsCallback()）。这时允许用户存储镶嵌分块的结果用于后面的处理操作。其初始值是GLU_NURBS_RENDERER。

GLU_SAMPLING_METHOD

指定一个NURBS曲面的镶嵌分块方式。参数*value*可以是GLU_PATH_LENGTH, GLU_PARAMETRIC_ERROR, GLU_DOMAIN_DISTANCE, GLU_OBJECT_PATH_LENGTH或GLU_OBJECT_PARAMETRIC_ERROR。当设置成GLU_PATH_LENGTH时，绘制曲面时镶嵌分块的多边形的边的最大长度（像素长度）不能大于GLU_SAMPLING_TOLERANCE的指定值。

当设置成GLU_PARAMETRIC_ERROR时，表示由GLU_PARAMETRIC_TOLERANCE所指定的值是镶嵌分块多边形与它所近似的曲面之间的最大距离。

当设置成GLU_DOMAIN_DISTANCE时，表示允许用户在参数坐标上指定在u和v方向上每个单位长度内的采样点的数目。

当设置成GLU_OBJECT_PATH_LENGTH时，除了表示它是独立视图之外，其他都与GLU_PATH_LENGTH类似。也就是说在对象空间中绘制曲面时镶嵌分块的多边形的边的最大长度（像素长度）不能大于GLU_SAMPLING_

TOLERANCE的指定值。

当设置成**GLU_OBJECT_PARAMETRIC_ERROR**时，除了它是独立视图之外，其他都与**GLU_PARAMETRIC_ERROR**类似。也就是说由**GLU_PARAMETRIC_TOLERANCE**所指定的值是镶嵌分块多边形与它所近似的曲面之间的最大距离（在对象空间中）。

GLU_SAMPLING_METHOD的缺省值是**GLU_PATH_LENGTH**。

GLU_SAMPLING_TOLERANCE

指定当取样模式设置成**GLU_PATH_LENGTH**或**GLU_OBJECT_PATH_LENGTH**时的最大长度值（采用像素形式或对象空间长度单位）。当绘制一个曲线或曲面时，NURBS代码是有所保留的，因此实际的长度值要小一些。其初始值是50.0像素。

GLU_PARAMETRIC_TOLERANCE

指定当取样模式设置成**GLU_PARAMETRIC_ERROR**或**GLU_OBJECT_PARAMETRIC_ERROR**时的最大距离（采用像素形式或对象空间长度单位）。其初始值是0.5。

GLU_U_STEP

在参数坐标上指定沿u轴方向的每个单位长度中的采样点的数目。当**GLU_SAMPLING_METHOD**设置成**GLU_DOMAIN_DISTANCE**时将用到该值。其初始值时100。

GLU_V_STEP

在参数坐标上指定沿v轴方向的每个单位长度中的采样点的数目。当**GLU_SAMPLING_METHOD**设置成**GLU_DOMAIN_DISTANCE**时将用到该值。其初始值时100。

GLU_DISPLAY_MODE

参数*value*可以设置成**GLU_OUTLINE_POLYGON**, **GLU_FILL**或**GLU_OUTLINE_PATCH**。当**GLU_NURBS_MODE**设置成**GLU_NURBS_RENDERER**时，参数*value*用来指定一个NURBS曲面将如何绘制。当*value*设置成**GLU_FILL**时，曲面将绘制成一组多边形。当*value*设置成**GLU_OUTLINE_POLYGON**时，NURBS库仅绘出镶嵌分块建立的多边形的轮廓。当*value*设置成**GLU_OUTLINE_PATCH**时，仅绘出由用户定义的镶嵌分块的轮廓及修整过的曲面。

当**GLU_NURBS_MODE**设置成**GLU_NURBS_TESSELLATOR**时，参数*value*用来指定一个NURBS曲面将如何镶嵌分块。当**GLU_DISPLAY_MODE**设置成**GLU_FILL**或**GLU_OUTLINE_POLYGON**时，NURBS曲面将被镶嵌成能由反馈函数返回的OpenGL三角形图元。当**GLU_DISPLAY_MODE**设置成**GLU_OUTLINE_PATCH**时，仅仅生成镶嵌分块的轮廓和修整过的曲线。它们将作为线条序列由反馈函数返回。

其缺省值是**GLU_FILL**。

GLU_CULLING

参数*value*是一个boolean值。当它被设置为**GL_TRUE**时，表示如果一个NURBS曲线的控制点位于当前视口之外则优先丢弃该曲线而不是将它镶嵌分块。其初始值是**GL_FALSE**。

GLU_AUTO_LOAD_MATRIX

参数*value*是一个boolean值。当它被设置为**GL_TRUE**时，表示NURBS代码将用通过**GL**服务器下载投影矩阵、模式取景矩阵和视口来计算每个将要绘制的NURBS曲线的取样和拣选矩阵。取样和拣选矩阵的作用是用来确定将一个NURBS曲面镶嵌分块成线段还是多边形，并且用来确定当一个NURBS曲面位于视口外时是否将它拣选掉。当这一模式设置成**GL_FALSE**时，那么绘制NURBS时需要提供一个投影矩阵、一个模式取景矩阵及一个视口来构造取样和拣选矩阵。这些工作可以用例程**gluLoadSamplingMatrices()**来完成。

该模式的缺省值是**GL_TRUE**。在调用例程**gluLoadSamplingMatrices()**之前，模式由**GL_TRUE**变为**GL_FALSE**并不影响取样和拣选矩阵。

- 注意：

当**GLU_AUTO_LOAD_MATRIX**为True时，如果NURBS例程已被编入一个显示列表，则取样和拣选矩阵可能被错误执行。

参数*property*的值：**GLU_PARAMETRIC_TOLERANCE**、**GLU_SAMPLING_METHOD**、**GLU_U_STEP**或**GLU_V_STEP**，或参数*value*的值：**GLU_PATH_LENGTH**、**GLU_PARAMETRIC_ERROR**、**GLU_DOMAIN_DISTANCE**仅在**GL 1.1**以上的版本中才有效，它们不能在**GLU 1.0**中使用。

例程**gluGetString()**可用来确定**GLU**的版本。

GLU_NURBS_MODE仅在**GLU 1.3**以上的版本中才有效。

GLU_SAMPLING_METHOD的属性值**GLU_OBJECT_PATH_LENGTH**和**GLU_OBJECT_PARAMETRIC_ERROR**仅在**GLU 1.3**以上的版本中才有效。

- 请参阅：

gluGetNurbsProperty(), **gluLoadSamplingMatrices()**, **gluNewNurbsRenderer()**,
gluGetString(), **gluNurbsCallback**

gluNurbsSurface

- 名称：

gluNurbsSurface()

- 功能：

定义了一个NURBS曲面的外形。

- C描述：

```
void gluNurbsSurface( GLUnurbs* nurb,
```

```

    GLint sKnotCount,
    GLfloat* sKnots,
    GLint tKnotCount,
    GLfloat* tKnots,
    GLint sStride,
    GLint tStride,
    GLfloat *control,
    GLint sOrder,
    GLint tOrder,
    GLenum type )

```

• 参数说明：

nurb 指定NURBS对象（由*gluNewNurbsRenderer()*建立）。

sKnotCount

指定参数*u*方向的结点数目。

sKnots 指定参数*u*方向的一个*sKnotCount*的不减结点值的数组。

tKnotCount

指定参数*v*方向的结点数目。

tKnots 指定参数*v*方向的一个*tKnotCount*的不减结点值的数组。

sStride 指定*control*中的参数*u*方向的连续的控制点之间的偏移量（作为一个单精度浮点值的数目）。

tStride 指定*control*中的参数*v*方向的连续的控制点之间的偏移量（为一个单精度浮点型的数值）。

control 指定一个指针，指向含有NURBS曲面的控制点的数组。在参数*u*和*v*方向的连续控制点之间的偏移量由*sStride*和*tStride*给出。

sOrder 指定参数*u*方向的NURBS曲面的阶数。参数*order*等于度数+1。要得到一个参数*u*的立体曲面，则*u*的阶数应该是4。

tOrder 指定参数*v*方向的NURBS曲面的阶数。参数*order*等于度数+1。要得到一个参数*v*的立体曲面，则*v*的阶数应该是4。

type 指定曲面的类型。参数*type*可以是任意有效的二维求值器的类型（如**GL_MAP2_VERTEX_3**或**GL_MAP2_COLOR_4**）。

• 说明：

用一个NURBS（非均匀（一致）有理B样条曲线）曲面定义中的例程*gluNurbsSurface()*来描述这个NURBS曲面的形状（在进行任何的修整之前）。命令*gluBeginSurface()*表示一个NURBS曲面定义的开始，命令*gluEndSurface()*表示一个NURBS曲面定义的结束。命令*gluNurbsSurface()*只在一个NURBS曲面的定义中使用。

例程对*gluBeginSurface()*/*gluEndSurface()*之间的每个单独的*gluNurbsSurface()*例程都提供了一个曲面的相应位置、纹理和颜色坐标。在例程对*gluBeginSurface()*/*gluEndSurface()*之间可

以不止一次地调用例程`gluNurbsSurface()`来生成颜色、位置和纹理数据。每一次正确的调用都应该指出曲面的位置（对于每一个`GL_MAP2_VERTEX_3`或`GL_MAP2_VERTEX_4`的`type`）。

用例程对`gluBeginTrim()`/`gluEndTrim()`之间的例程`gluNurbsCurve()`和`gluPwlCurve()`可以修整一个NURBS曲面。

请注意：当一个例程`gluNurbsSurface()`带有`u`方向的参数`sKnotCount`、`v`方向的参数`tKnotCount`以及阶数`sOrder`和`tOrder`时，它必须有 $(sKnotCount-sOrder) \times (tKnotCount-tOrder)$ 个控制点。

- 示例：

下面的命令用来绘制一个带法线的纹理化NURBS曲面，纹理坐标和法线也是NURBS曲面：

```
gluBeginSurface(nobj);
    gluNurbsSurface(nobj, ..., GL_MAP2_TEXTURE_COORD_2);
    gluNurbsSurface(nobj, ..., GL_MAP2_NORMAL);
    gluNurbsSurface(nobj, ..., GL_MAP2_VERTEX_4);
gluEndSurface(nobj);
```

- 请参阅：

`gluBeginSurface()`, `gluBeginTrim()`, `gluNewNurbsRenderer()`, `gluNurbsCurve()`,
`gluPwlCurve()`

• `gluOrtho2D`

- 名称：

`gluOrtho2D()`

- 功能：

定义一个2D正投影矩阵。

- C描述：

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

- 参数说明：

left, right

指定左、右垂直剪切平面的坐标。

bottom, top

指定上和下水平剪切平面的坐标。

- 说明：

例程`gluOrtho2D()`建立了一个二维的正投影取景区域。它相当于函数`glOrtho()`取`near=-1`,
`far=1`的情况。

- 请参阅：

`glOrtho()`, `gluPerspective()`

• gluPartialDisk

- 名称：

gluPartialDisk()

- 功能：

绘制盘形的圆弧。

- C描述：

```
void gluPartialDisk( GLUquadric* quad,
                      GLdouble inner,
                      GLdouble outer,
                      GLint slices,
                      GLint loops,
                      GLdouble start,
                      GLdouble sweep )
```

- 参数说明：

quad 指定二次对象（由gluNewQuadric()建立）。

inner 指定不完整圆盘的内部半径（可能是0）。

outer 指定不完整圆盘的外部半径。

slices 指定围绕z轴细分的数目。

loops 指定不完整圆盘细分时绕原点分割的同心圆环的数目。

start 指定不完整圆盘的起始角（度数值）。

sweep 指定不完整圆盘所摆过的角度（度数值）。

- 说明：

例程gluPartialDisk()在 $z=0$ 的平面上绘制一个不完整圆盘。它与完整圆盘相类似，唯一不同之处时它只是完整圆盘的一个子集，它从*start*开始到*start+sweep*（包含）为止（这里 0° 是沿+y轴方向， 90° 沿+x轴方向， 180° 沿-y轴方向， 270° 沿-x轴方向）。

这个不完整圆盘的外半径是*outer*，并有一个半径为*inner*的同心圆孔。当*inner*是0时，将没有圆孔生成。该不完整圆盘绕z轴方向被细分成片状（类似于比萨饼的薄片），并且在z轴周围被分割成环状（它们的数目分别由*slices*和*loops*指定）。

至于它的方向，不完整圆盘+z轴的一边被认为是它的“外边”（请参阅gluQuadricOrientation()）。这就意味着如果它的方向被设置成GLU_OUTSIDE，则生成的任何法线方向都是指向+z轴方向的。否则，它们将指向-z轴方向。

如果启动了纹理功能（用gluQuadricTexture()），它将生成纹理坐标。这时纹理坐标*t*将通过下面方式线性的生成：在 $(r, 0, 0)$ 处为 $(1, 0.5)$ ，在 $(0, r, 0)$ 处为 $(0.5, 1)$ ，在 $(-r, 0, 0)$ 处为 $(0, 0.5)$ ，在 $(0, -r, 0)$ 处为 $(0.5, 0)$ ，这里 $r=outer$ 。

- 请参阅：

gluCylinder(), gluDisk(), gluNewQuadric(), gluQuadricOrientation(), gluQuadricTexture(), gluSphere()

• gluPerspective

- 名称:

gluPerspective()

- 功能:

建立一个透视投影矩阵。

- C描述:

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

- 参数说明:

fovy 指定y方向的取景区域的角度(°)。

aspect 指定x方向的用来确定取景区域的高宽比。高宽比是x(宽度)与y(高度)的比率。

zNear 指定视点到最近的裁剪平面的距离。(它必须是正数。)

zFar 指定视点到最远的裁剪平面的距离。(它必须是正数。)

- 说明:

例程**gluPerspective()**用来指定一个世界坐标体系中的截锥体的取景区域。一般来讲，例程**gluPerspective()**中的高宽比应该同相应的视口中的高宽比相匹配。例如，*aspect*=2.0意味着取景区域在x方向上的角度是它在y方向上的角度的2倍。这时，如果视口的宽度是高度的2倍，则它所显示的图像将没有扭曲变形。

由例程**gluPerspective()**生成的矩阵将同当前的矩阵相乘，就像用生成的矩阵调用函数**glMultMatrix()**一样。如果要将透视矩阵载入当前矩阵堆栈中，请在调用**gluPerspective()**前先调用函数**glLoadIdentity()**。

这里给出f的定义:

$$f = \cotangent\left(\frac{\text{fovy}}{2}\right)$$

则生成的矩阵为:

$$\begin{matrix} \frac{f}{\text{aspect}} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{zFar+zNear}{zNear-zFar} & \frac{2*zFar*zNear}{zNear-zFar} \\ 0 & 0 & -1 & 0 \end{matrix}$$

- 注意:

深度缓冲区的精度将受指定的*zNear*和*zFar*值的影响。*zNear*和*zFar*的比值越大，深度缓冲区中两个相近表面之间的区别的将越小。当

$$r = \frac{zFar}{zNear}$$

时，深度缓冲区的大约 $\log_2(r)$ 位的精度将被丢失。由于当 $zNear$ 趋近于0时 r 将趋近于无穷大，所以 $zNear$ 不能被设置为0。

- 请参阅：

glFrustum(), **glLoadIdentity()**, **glMultMatrix()**, **gluOrtho2D()**

• **gluPickMatrix**

- 名称：

gluPickMatrix()

- 功能：

定义一个拣选区域。

- C描述：

```
void gluPickMatrix( GLdouble x,
                     GLdouble y,
                     GLdouble delX,
                     GLdouble delY,
                     GLint *viewport )
```

- 参数说明：

x, y

指定窗口坐标中的一个拣选区域的中点。

delX, delY

分别指定窗口坐标中的拣选区域的宽度和高度。

viewport

指定当前视口（从调用函数**glGetIntegerv()**时起）。

- 说明：

例程**gluPickMatrix()**建立了一个投影矩阵，该矩阵可以用来限制在视口中的一个较小的范围内进行的绘制操作。它可以有效地确定光标附近的哪些对象将被绘出。用例程**gluPickMatrix()**来约束光标附近的一个小范围中的绘制操作。接着进入选择模式（用函数**glRenderMode()**）并绘制场景。这时所有光标附近要绘制的图元将被识别并存入选择缓冲区中。

由例程**gluPerspective()**生成的矩阵将同当前的矩阵相乘，就像用生成的矩阵调用函数**glMultMatrix()**一样。如果要有效地使用生成的拣选矩阵进行拣选操作，首先应该调用函数**glLoadIdentity()**来将一个单位矩阵载入到投影矩阵堆栈，接着调用**gluPickMatrix()**，最后调用一个命令（如**glMultMatrix()**）将投影矩阵同拣选矩阵相乘。

当使用例程**gluPickMatrix()**来拣选NURBS时，要仔细确认已经关闭了**GLU_AUTO_LOAD_MATRIX**。如果**GLU_AUTO_LOAD_MATRIX**没有被关闭，则任一个要绘制的NURBS曲面通过拣选矩阵而细分的情况将同没有通过该拣选矩阵细分的情况不一样。

- 示例：

当绘制的场景如下时：

```

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(...);
glMatrixMode(GL_MODELVIEW);
/* Draw the scene */

```

可以像下面这样选择局部视口作为拣选区域：

```

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPickMatrix(x, y, width, height, viewport);
gluPerspective(...);
glMatrixMode(GL_MODELVIEW);
/* Draw the scene */

```

- 请参阅：

glGet(), **glLoadIdentity()**, **glMultMatrix()**, **glRenderMode()**, **gluPerspective()**

gluProject

- 名称：

gluProject()

- 功能：

将对象坐标映射成窗口坐标。

- C描述：

```

GLint gluProject( GLdouble objX,
                  GLdouble objY,
                  GLdouble objZ,
                  const GLdouble *model,
                  const GLint *proj,
                  const GLint *view,
                  GLdouble* winX,
                  GLdouble* winY,
                  GLdouble* winZ )

```

- 参数说明：

objX, *objY*, *objZ*

指定对象坐标。

model 指定当前的模式取景矩阵（从调用**glGetDoublev()**时起）。

proj 指定当前的投影矩阵（从调用**glGetDoublev()**时起）。

view 指定当前的视口（从调用**glGetIntegerv()**时起）。

winX, *winY*, *winZ*

返回计算所得的窗口坐标。

- 说明：

例程`gluProject()`用`model`, `proj`和`view`将指定的对象坐标转换成窗口坐标。所得的结果被存放在`winX`, `winY`和`winZ`中。返回值如果是`GL_TRUE`表示操作成功, 如果是`GL_FALSE`表示操作失败。

为了计算窗口坐标, 设 $v = (objX, objY, objZ, 1.0)$, 表示一个4行1列的矩阵。则通过例程`gluProject()`计算所得的 v' 如下:

$$v' = P \times M \times v$$

这里, P 是当前的投影矩阵`proj`, M 是当前的模式取景矩阵`model`(二者均表示以列序为主的 4×4 矩阵), 并且这里的“ \times ”代表矩阵相乘。

则, 窗口矩阵由下式计算:

$$winX = view(0) + view(2) * (v'(0) + 1) / 2$$

$$winY = view(1) + view(3) * (v'(1) + 1) / 2$$

$$winZ = (v'(2) + 1) / 2$$

- 请参阅:

`glGet()`, `gluUnProject()`

• `gluPwlCurve`

- 名称:

`gluPwlCurve()`

- 功能:

描述一条分段线性的NURBS修整曲线。

- C描述:

```
void gluPwlCurve( GLUnurbs* nurb,
                   GLint count,
                   GLfloat* data,
                   GLint stride,
                   GLenum type )
```

- 参数说明:

`nurb` 指定NURBS对象(由`gluNewNurbsRenderer()`建立)。

`count` 指定曲线中控制点的数目。

`data` 指定一个包含曲线控制点的数组。

`stride` 指定连续的曲线控制点之间的偏移量(它是一个单精度浮点型的数值)。

`type` 指定曲线的类型。它只能是`GLU_MAP1_TRIM_2`和`GLU_MAP1_TRIM_3`。

- 说明:

例程`gluPwlCurve()`用来描述一条分段线性的NURBS曲面的修整曲线。一个分段线性曲线包含了一组参数空间中的点的坐标。它们用来修整NURBS曲面。这些点由线段连在了一起, 从而

形成了一条曲线。如果该曲线是没有分段线性化的曲线的一条近似曲线，则这些点在参数空间中应该尽量地靠近，以便在应用中的分辨率下看起来它们所连成的线是曲线。

如果*type*是**GLU_MAP1_TRIM_2**，则它描述一条二维(*u*和*v*)参数空间中的曲线。如果*type*是**GLU_MAP1_TRIM_3**，则它描述一条二维齐次(*u*、*v*和*w*)参数空间中的曲线。有关修整曲线的详细内容请参阅**gluBeginTrim()**的参考说明。

- 注意：

调用例程**gluNurbsCurve()**，描述一条紧靠NURBS曲面轮廓的修整曲线。

- 请参阅：

gluBeginCurve(), **gluBeginTrim()**, **gluNewNurbsRenderer()**, **gluNurbsCurve()**

• **gluQuadricCallback**

- 名称：

gluQuadricCallback()

- 功能：

定义一个二次曲面对象的回调。

- C描述：

```
void gluQuadricCallback( GLUquadric* quad,
                           GLenum which,
                           GLvoid (*CallBackFunc) )
```

- 参数说明：

quad 指定二次曲面对象（由**gluNewQuadric()**建立）。

which 指定要定义的回调。它唯一的有效值是**GLU_ERROR**。

CallBackFunc

指定将要调用的函数。

- 说明：

例程**gluQuadricCallback()**用来定义一个二次曲面对象所要使用的新回调。如果指定的回调已经定义过，那么它将被新指定的回调所替换。如果*CallBackFunc*是NULL，则所有已存在的回调都将被删除。

一个有效的回调是**GLU_ERROR**：

GLU_ERROR 当有错误产生时，该函数被调用。它的唯一自变量就是类型**GLenum**，它表示出现了指定的错误。描述这些错误的特征字符串将由例程**gluErrorString()**返回。

- 请参阅：

gluErrorString(), **gluNewQuadric()**

• **gluQuadricDrawStyle**

- 名称：

gluQuadricDrawStyle()

- 功能：

指定二次曲面所需的绘图类型。

- C 描述：

```
void gluQuadricDrawStyle( GLUquadric* quad,
                           GLenum draw )
```

- 参数说明：

quad 指定二次曲面对象（由 **gluNewQuadric()** 建立）。

draw 指定所需的绘图类型。它的有效值有：**GLU_FILL**、**GLU_LINE**、**GLU_SILHOUETTE** 和 **GLU_POINT**。

- 说明：

例程 **gluQuadricDrawStyle()** 用来指定由 *quad* 绘制的二次曲面的绘图类型。绘图类型的有效值如下：

GLU_FILL 用多边形图元来绘制二次曲面。多边形按逆时针方向绘制，这里的逆时针方向是由它的法线所决定的（像例程 **gluQuadricOrientation()** 指定的那样）。

GLU_LINE 用一组线段来绘制二次曲面。

GLU_SILHOUETTE

除了不绘出共面的分割边外，二次曲面其他部分都由一组线段绘出。

GLU_POINT 用一组点绘制二次曲面。

- 请参阅：

gluNewQuadric(), **gluQuadricNormals()**, **gluQuadricOrientation()**, **gluQuadricTexture()**

gluQuadricNormals

- 名称：

gluQuadricNormals()

- 功能：

指定二次曲面所需的法线类型。

- C 描述：

```
void gluQuadricNormal( GLUquadric* quad,
                        GLenum normal )
```

- 参数说明：

quad 指定二次曲面对象（由 **gluNewQuadric()** 建立）。

normal 指定所需的法线类型。它的有效值为：**GLU_NONE**、**GLU_FLAT** 和 **GLU_SMOOTH**。

- 说明：

例程 **gluQuadricNormal()** 用来指定由 *quad* 绘制二次曲面对象时所需的法线类型。法线类型的有效值如下：

GLU_NONE	不生成法线。
GLU_FLAT	为二次曲面的每个小平面都生成一条法线。
GLU_SMOOTH	为二次曲面的每个顶点都生成一条法线。这是默认值。
<ul style="list-style-type: none"> • 请参阅： gluNewQuadric(), gluQuadricDrawStyle(), gluQuadricOrientation(), gluQuadricTexture() 	

• **gluQuadricOrientation**

- 名称：

gluQuadricOrientation()

- 功能：

为二次曲面指定内部/外部方向。

- C描述：

```
void gluQuadricOrientation( GLUquadric* quad,
                           GLenum orientation )
```

- 参数说明：

quad 指定二次曲面对象 (由**gluNewQuadric()**建立)。

orientation 指定所需的方向。它的有效值为：**GLU_OUTSIDE**和**GLU_INSIDE**。

- 说明：

例程**gluQuadricOrientation()**用来指定由*quad*绘制二次曲面对象时所需的方向类型。方向类型的有效值如下：

GLU_OUTSIDE 绘制二次曲面时它的法线要指向外部(默认值)。

GLU_INSIDE 绘制二次曲面时它的法线要指向内部。

请注意：“外部” 和 “内部”的具体解释由将绘制的二次曲面决定。

- 请参阅：

gluNewQuadric(), **gluQuadricDrawStyle()**, **gluQuadricNormals()**, **gluQuadricTexture()**

• **gluQuadricTexture**

- 名称：

gluQuadricTexture()

- 功能：

指定绘制二次曲面时是否需要纹理。

- C描述：

```
void gluQuadricTexture( GLUquadric* quad,
                       GLboolean texture )
```

- 参数说明：

quad 指定二次曲面对象 (由**gluNewQuadric()**建立)。

texture 指定一个标志，用来表明是否需要生成纹理坐标。

- 说明：

例程`gluQuadricTexture()`用来指定由`quad`绘制二次曲面对象时是否生成纹理坐标。如果参数`texture`的值是`GL_TRUE`，则生成纹理坐标；如果是`GL_FALSE`，则不生成纹理坐标。它的默认值是`GL_FALSE`。

纹理坐标的生成方式由所绘制的特定二次曲线决定。

- 请参阅：

`gluNewQuadric()`, `gluQuadricDrawStyle()`, `gluQuadricNormals()`, `gluQuadricOrientation()`

• `gluScaleImage`

- 名称：

`gluScaleImage()`

- 功能：

将一个图像缩放成任意尺寸。

- C 描述：

GLint `gluScaleImage(GLenum format,`

```
          GLsizei wIn,
          GLsizei hIn,
          GLenum typeIn,
          const void *dataIn,
          GLsizei wOut,
          GLsizei hOut,
          GLenum typeOut,
          GLvoid* dataOut )
```

- 参数说明：

format

指定像素数据的格式。下列符号值是有效的：`GL_COLOR_INDEX`、`GL_STENCIL_INDEX`、`GL_DEPTH_COMPONENT`、`GL_RED`、`GL_GREEN`、`GL_BLUE`、`GL_ALPHA`、`GL_RGB`、`GL_RGBA`、`GL_BGR`、`GL_BGRA`、`GL_LUMINANCE`和`GL_LUMINANCE_ALPHA`。

wIn, *hIn*

分别指定源图像的像素宽度和高度。

typeIn

指定*dataIn*的像素数据类型。它必须是下面各值之一：`GL_UNSIGNED_BYTE`、`GL_BYTE`、`GL_BITMAP`、`GL_UNSIGNED_SHORT`、`GL_SHORT`、`GL_UNSIGNED_INT`、`GL_INT`、`GL_FLOAT`、`GL_UNSIGNED_BYTE_3_3_2`、`GL_UNSIGNED_BYTE_2_3_3_REV`、`GL_UNSIGNED_SHORT_5_6_5`、`GL_UNSIGNED_SHORT_5_6_5_REV`、`GL_UNSIGNED_SHORT_4_4_4_4`、`GL_UNSIGNED_SHORT_4_4_4_4_REV`、`GL_UNSIGNED_SHORT_5_5_5_1`、`GL_UNSIGNED_SHORT_1_5_5_5_REV`、

GL_UNSIGNED_INT_8_8_8、**GL_UNSIGNED_INT_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**。

dataIn

指定一个指向源图像的指针。

wOut, hOut

分别指定目标图像的像素宽度和高度。

typeOut

指定*dataOut*的像素数据类型。它必须是下面各值之一：**GL_UNSIGNED_BYTE**、**GL_BYTE**、**GL_BITMAP**、**GL_UNSIGNED_SHORT**、**GL_SHORT**、**GL_UNSIGNED_INT**、**GL_INT**、**GL_FLOAT**、**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**。

dataOut

指定一个指向目标图像的指针。

- 说明：

例程gluScaleImage()用适当的像素存储模式来缩放一个像素，它从源图像中取出数据，操作后放回到目标图像中。

当收缩一个图像时，例程gluScaleImage()用一个盒式滤波器采样源图像并为目标图像建立像素。当放大一个图像时，由源图像得到的像素通过线性内插的方法为目标图像生成像素。

当返回值是0时，表示该操作已经成功；否则，它将返回一个GLU的出错代码（请参阅gluErrorString()）。

参数*format*、*typeIn*和*typeOut*的取值请参阅glReadPixels()的参考说明。

- 注意：

格式**GL_BGR**和**GL_BGRA**及类型**GL_UNSIGNED_BYTE_3_3_2**、**GL_UNSIGNED_BYTE_2_3_3_REV**、**GL_UNSIGNED_SHORT_5_6_5**、**GL_UNSIGNED_SHORT_5_6_5_REV**、**GL_UNSIGNED_SHORT_4_4_4_4**、**GL_UNSIGNED_SHORT_4_4_4_4_REV**、**GL_UNSIGNED_SHORT_5_5_5_1**、**GL_UNSIGNED_SHORT_1_5_5_5_REV**、**GL_UNSIGNED_INT_8_8_8_8**、**GL_UNSIGNED_INT_8_8_8_8_REV**、**GL_UNSIGNED_INT_10_10_10_2**和**GL_UNSIGNED_INT_2_10_10_10_REV**仅在GL 1.2以上的版本中才有效。

- 出错提示：

当参数*wIn*、*hIn*、*wOut*或*hOut*是负数时产生**GLU_INVALID_VALUE**提示。

当参数*format*、*typeIn*或*typeOut*不是一个合法值时产生**GLU_INVALID_ENUM**提示。

当参数*typeIn*或*typeOut*为**GL_UNSIGNED_BYTE_3_3_2**或**GL_UNSIGNED_BYTE_2_3_3_REV**，并且参数*format*不是**GL_RGB**格式时产生**GLU_INVALID_OPERATION**提示。

当参数`typeIn`或`typeOut`为**GL_UNSIGNED_SHORT_5_6_5**或**GL_UNSIGNED_SHORT_5_6_5_REV**, 并且参数`format`不是**GL_RGB**格式时产生**GLU_INVALID_OPERATION**提示。

当参数`typeIn`或`typeOut`为**GL_UNSIGNED_SHORT_4_4_4_4**或**GL_UNSIGNED_SHORT_4_4_4_4_REV**, 并且参数`format`既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

当参数`typeIn`或`typeOut`为**GL_UNSIGNED_SHORT_5_5_5_1**或**GL_UNSIGNED_SHORT_1_5_5_5_REV**, 并且参数`format`既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

当参数`typeIn`或`typeOut`为**GL_UNSIGNED_INT_8_8_8_8**或**GL_UNSIGNED_INT_8_8_8_8_REV**, 并且参数`format`既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

当参数`typeIn`或`typeOut`为**GL_UNSIGNED_INT_10_10_10_2**或**GL_UNSIGNED_INT_2_10_10_10_REV**, 并且参数`format`既不是**GL_RGBA**格式也不是**GL_BGRA**格式时产生**GLU_INVALID_OPERATION**提示。

- 请参阅:

`glDrawPixels()`, `glReadPixels()`, `gluBuild1DMipmaps()`, `gluBuild2DMipmaps()`,
`gluBuild3DMipmaps()`, `gluErrorString()`

`:gluSphere`

- 名称:

`gluSphere()`

- 功能:

绘制一个球体。

- C 描述:

```
void gluSphere( GLUquadric* quad,
                GLdouble radius,
                GLint slices,
                GLint stacks )
```

- 参数说明:

`quad` 指定二次曲面对象 (由`gluNewQuadric()`建立)。

`radius` 指定球体的半径。

`slices` 指定围绕z轴细分时的数目 (类似于经线)。

`stacks` 指定沿z轴方向细分时的数目 (类似于纬线)。

- 说明:

例程`gluSphere()`用给定的半径绕原点绘制一个球体。该球体绕z轴方向被细分成片状, 沿z轴方向被细分成层状 (类似于经线和纬线)。

如果它的方向被设置成**GLU_OUTSIDE** (请参阅`gluQuadricOrientation()`), 则说明它所生

成的法线是由球心指向外的；否则，它们将指向球心。

如果启动了纹理功能（用**gluQuadricTexture()**），它将生成纹理坐标。这时纹理坐标*t*的范围将由*z=-radius*处的0.0变化到*z=radius*处的1.0（*t*沿径线方向线性增长）；纹理坐标*s*的范围将由+y处的0.0变到+x处的0.25，然后变到-y处的0.5，然后变到-x处的0.75，然后返回+y处变成为1.0。

- 请参阅：

gluCylinder(), **gluDisk()**, **gluNewQuadric()**, **gluPartialDisk()**, **gluQuadricOrientation()**, **gluQuadricTexture()**

• **gluTessBeginContour**, **gluTessEndContour**

- 名称：

gluTessBeginContour(), **gluTessEndContour()**

- 功能：

限定一个轮廓线的描述。

- C描述：

`void gluTessBeginContour(GLUtesselator* tess)`

`void gluTessEndContour (GLUtesselator* tess)`

- 参数说明：

tess 指定镶嵌分块对象（由**gluNewTess()**创建）。

- 说明：

例程**gluTessBeginContour()**和**gluTessEndContour()**用来限定一个多边形轮廓线的定义。在例程对**gluTessBeginContour()**/**gluTessEndContour()**之间可以没有也可以调用多个**gluTessVertex()**。顶点指定了一条封闭的轮廓线（每一条轮廓线的最后一个顶点自动地连到第一个顶点上）。有关细节请参阅**gluTessVertex()**的参考说明。例程**gluTessBeginContour()**只能在函数对**gluTessBeginPolygon()**/**gluTessEndPolygon()**之间被调用。

- 请参阅：

gluNewTess(), **gluTessBeginPolygon()**, **gluTessVertex()**, **gluTessCallback()**, **gluTessProperty()**, **gluTessNormal()**, **gluTessEndPolygon()**

• **gluTessBeginPolygon**

- 名称：

gluTessBeginPolygon()

- 功能：

限定一个多边形的描述。

- C描述：

`void gluTessBeginPolygon(GLUtesselator* tess,`
`GLvoid* data)`

- 参数说明：

tess 指定镶嵌分块对象（由gluNewTess()创建）。

data 指定一个指向用户多边形数据的指针。

- 说明：

例程gluTessBeginPolygon()和gluEndPolygon()用来限定一个凸多边形、凹多边形或自交叉多边形的定义。在每个例程对gluTessBeginPolygon()/gluTessEndPolygon()之间必须一次或多次调用例程对gluTessBeginContour()/gluTessEndContour()。在每根轮廓线之中可以没有也可以调用多个gluTessVertex()。顶点指定了一条封闭的轮廓线（每一条轮廓线的最后一个顶点自动地连到第一个顶点上）。有关细节请参阅gluTessVertex()、gluTessBeginContour()和gluTessEndContour()的参考说明。

参数*data*是一个指向用户定义的数据结构的指针。如果指定了适当的反馈（请参阅gluTessCallback()），则该指针返回反馈函数。这样它可以方便地存储每个多边形的信息。

一旦调用了例程gluTessEndPolygon()，多边形将被镶嵌分块，由此产生的三角形将通过反馈加以说明。反馈函数的有关细节请参阅例程gluTessCallback()。

- 示例：

以下命令用来定义一个含有三角形孔的四边形：

```
gluTessBeginPolygon(tobj, NULL);
gluTessBeginContour(tobj);
    gluTessVertex(tobj, v1, v1);
    gluTessVertex(tobj, v2, v2);
    gluTessVertex(tobj, v3, v3);
    gluTessVertex(tobj, v4, v4);
gluTessEndContour(tobj);
gluTessBeginContour(tobj);
    gluTessVertex(tobj, v5, v5);
    gluTessVertex(tobj, v6, v6);
    gluTessVertex(tobj, v7, v7);
gluTessEndContour(tobj);
gluTessEndPolygon(tobj);
```

- 请参阅：

gluNewTess(), gluTessBeginContour(), gluTessVertex(), gluTessCallback(),
gluTessProperty(), gluTessNormal(), gluTessEndPolygon()

gluTessCallback

- 名称：

gluTessCallback()

- 功能：

定义一个镶嵌分块对象的反馈。

- C描述：

```
void gluTessCallback( GLUtesselator* tess,
                      GLenum which,
```

`GLvoid (*CallBackFunc)()`

- 参数说明：

nurb 指定镶嵌分块对象（由`gluNewTess()`建立）。

which 指定被定义的反馈。它的有效值有：`GLU_TESS_BEGIN`、`GLU_TESS_BEGIN_DATA`、`GLU_TESS_EDGE_FLAG`、`GLU_TESS_EDGE_FLAG_DATA`、`GLU_TESS_VERTEX`、`GLU_TESS_VERTEX_DATA`、`GLU_TESS_END`、`GLU_TESS_END_DATA`、`GLU_TESS_COMBINE`、`GLU_TESS_COMBINE_DATA`、`GLU_TESS_ERROR`和`GLU_TESS_ERROR_DATA`。

CallBackFunc

指定反馈时所调用的函数。

- 说明：

例程`gluTessCallback()`用来定义一个镶嵌分块对象所使用的反馈。如果指定的反馈已经被定义，那么它将被替换。当`CallBackFunc`是NULL时，则已存在的反馈将变得未定义。

镶嵌分块对象用这些反馈来描述一个多边形如何被用户分解成三角形。值得注意的是每个反馈都有两种形式：一种形式带有用户指定的多边形数据而另一种形式则没有。如果某个特定的反馈同时指定了两种形式，那么带有用户指定的多边形数据的形式将被使用。被某些函数所使用的`polygon_data`参数是调用`gluTessBeginPolygon()`时所指定的指针的一个拷贝。

有效的反馈形式如下：

`GLU_TESS_BEGIN`

开始反馈函数如同函数`glBegin()`一样，它表示一个（三角形）图元的开始。该函数带有一个GLenum类型的自变量。当`GLU_TESS_BOUNDARY_ONLY`属性被设置成`GL_FALSE`时，该自变量可以被设置成`GL_TRIANGLE_FAN`、`GL_TRIANGLE_STRIP`或`GL_TRIANGLES`。当`GLU_TESS_BOUNDARY_ONLY`属性被设置成`GL_TRUE`时，这个自变量将被设置成`GL_LINE_LOOP`。该反馈的函数原型是：

```
void begin ( GLenum type );
```

`GLU_TESS_BEGIN_DATA`

除了它带有一个附加的指针自变量外，它的用法同`GLU_TESS_BEGIN`完全一样。该指针相当于调用`gluTessBeginPolygon()`时所提供的不透明指针。该反馈的函数原型是：

```
void beginData ( GLenum type, void *polygon_data );
```

`GLU_TESS_EDGE_FLAG`

边界标志反馈函数类似于函数`glEdgeFlag()`。该函数带有一个boolean标志，它表示哪条边是位于多边形的边界上。当标志是`GL_TRUE`时，表示每个紧随一条边开始的顶点位于多边形的边界上。即该边是多边形内部与外部的分界边。当标志是`GL_FALSE`时，表示每个紧随一条边开始的顶点位于多边形的内部。在调用第一个顶点反馈函数之前应该先调用边界标志反馈函数（如果已经定义）。

由于扇形三角形和带状三角形都不支持边界标志，所以当提供了一个不是NULL的边界标志时，开始反馈不能用**GL_TRIANGLE_FAN**或**GL_TRIANGLE_STRIP**来调用。（如果反馈被初始化为NULL，那么在执行过程中便不再有冲突）。相反的，扇形和带状三角形将转化成相互独立的三角形。该反馈函数的原型是：

```
void edgeFlag ( GLboolean flag );
```

GLU_TESS_EDGE_FLAG_DATA

除了它带有一个附加的指针自变量外，它的用法同**GLU_TESS_EDGE_FLAG**完全一样。该指针相当于调用**gluTessBeginPolygon()**时所提供的不透明指针。该反馈的函数原型是：

```
void edgeFlagData ( GLboolean flag, void *polygon_data );
```

GLU_TESS_VERTEX

顶点反馈函数在开始和结束反馈函数之间被调用。它类似于函数**glVertex()**，定义了镶嵌分块过程中建立的三角形的顶点。该函数带有一个指针作为其唯一的自变量。该指针相当于用户描述顶点时所提供的不透明指针（请参阅**gluTessVertex()**）。该反馈的函数原型类似于：

```
void vertex ( void *vertex_data );
```

GLU_TESS_VERTEX_DATA

除了它带有一个附加的指针自变量外，它的用法同**GLU_TESS_VERTEX**完全一样。该指针相当于调用**gluTessBeginPolygon()**时所提供的不透明指针。该反馈的函数原型类似于：

```
void vertexData ( void *vertex_data, void *polygon_data );
```

GLU_TESS_END

结束反馈函数的功能同**glEnd()**一样。它表示一个图元的结束，它不含自变量。该反馈的函数原型类似于：

```
void end ( void );
```

GLU_TESS_END_DATA

除了它带有一个附加的指针自变量外，它的用法同**GLU_TESS_END**完全一样。该指针相当于调用**gluTessBeginPolygon()**时所提供的不透明指针。该反馈的函数原型类似于：

```
void endData ( void *polygon_data );
```

GLU_TESS_COMBINE

当镶嵌分块操作出现了交叉，或想要合并特性时，将调用合并反馈函数建立一个新的顶点。该函数带有四个自变量：一个元素类型为**GLdouble**的三元数组，一个包含四个指针的数组，一个元素类型为**GLfloat**的四元数组，和一个指向指针的指针。它的函数原型是：

```
void combine ( GLdouble coords[3], void *vertex_data[4],
```

```
GLfloat weight[4], void **outData );
```

顶点被定义成最多四个已有的顶点的线性组合，并被存放在`vertex_data`中。线性组合的系数由`weight`提供，这些加权系数相加的和最大值是1。即使某些加权系数等于0，所有的顶点指针也将有效。参数`coords`给出了新顶点的存放位置。用户必须指定并分配另一个顶点才能用`vertex_data`和`weight`来通过内插的方法生成一个新的顶点，并将这个新顶点的指针返回到`outData`中。在绘图反馈时将提供这种处理方法。当调用`gluTessEndPolygon()`后，用户应该释放所占用的内存。例如，当多边形位于一个任意的三维空间平面上并且每个顶点都含有一种颜色时，`GLU_TESS_COMBINE`反馈函数看起来类似于：

```
void myCombine( GLdouble coords[3], VERTEX *d[4],
                GLfloat w[4], VERTEX **dataOut )
{
    VERTEX *new = new_vertex();

    new->x = coords[0];
    new->y = coords[1];
    new->z = coords[2];
    new->r = w[0]*d[0]->r + w[1]*d[1]->r + w[2]*d[2]->r + w[3]*d[3]->r;
    new->g = w[0]*d[0]->g + w[1]*d[1]->g + w[2]*d[2]->g + w[3]*d[3]->g;
    new->b = w[0]*d[0]->b + w[1]*d[1]->b + w[2]*d[2]->b + w[3]*d[3]->b;
    new->a = w[0]*d[0]->a + w[1]*d[1]->a + w[2]*d[2]->a + w[3]*d[3]->a;
    *dataOut = new;
}
```

如果镶嵌分块操作出现了交叉，则必须调用`GLU_TESS_COMBINE`或`GLU_TESS_COMBINE_DATA`（如下所示）反馈函数并将一个非空的指针写入`dataOut`中。否则将产生`GLU_TESS_NEED_COMBINE_CALLBACK`出错提示，这时将不产生任何输出。

`GLU_TESS_COMBINE_DATA`

除了它带有一个附加的指针自变量外，它的用法同`GLU_TESS_COMBINE`完全一样。该指针相当于调用`gluTessBeginPolygon()`时所提供的不透明指针。该反馈的函数原型类似于：

```
void combineData ( GLdouble coords[3], void *vertex_data[4],
                    GLfloat weight[4], void **outData,
                    void *polygon_data );
```

`GLU_TESS_ERROR`

当遇到一个错误时将调用出错反馈。该函数带有一个`GLenum`类型的自变量，这个自变量表示所发生的特定错误，它将被设置成下面各值之一：`GLU_TESS_MISSING_BEGIN_POLYGON`、`GLU_TESS_MISSING_END_POLYGON`、`GLU_TESS_MISSING_BEGIN_CONTOUR`、`GLU_TESS_MISSING_END_CONTOUR`、`GLU_TESS_COORD_TOO_LARGE`、

GLU_TESS_NEED_COMBINE_CALLBACK或**GLU_OUT_OF_MEMORY**。用户可以通过例程gluErrorString()返回这些错误的特征字符串。该反馈的函数原型类似于：

```
void error ( GLenum errno );
```

GLU库将通过插入遗漏调用的方法而从最开始四个错误中恢复。**GLU_TESS_COORD_TOO_LARGE**表示某些顶点坐标的绝对值超过了预先指定的常量**GLU_TESS_MAX_COORD**，这些超出的坐标值将被截断。（坐标值应该足够小以便两个值的乘积不产生溢出。）**GLU_TESS_NEED_COMBINE_CALLBACK**表示镶嵌分块操作检测到输入数据中的两个边之间产生了交叉。这时它不提供**GLU_TESS_COMBINE**或**GLU_TESS_COMBINE_DATA**反馈函数，也不产生输出。**GLU_OUT_OF_MEMORY**表示没有足够的内存空间，自然也就不会产生输出。

GLU_TESS_ERROR_DATA

除了它带有一个附加的指针自变量外，它的用法同**GLU_TESS_ERROR**完全一样。该指针相当于调用gluTessBeginPolygon()时所提供的不透明指针。该反馈的函数原型类似于：

```
void errorData ( GLenum errno, void *polygon_data );
```

- 示例：

多边形镶嵌分块操作可以被以下命令直接绘制：

```
gluTessCallback(tobj, GLU_TESS_BEGIN, glBegin);
gluTessCallback(tobj, GLU_TESS_VERTEX, glVertex3dv);
gluTessCallback(tobj, GLU_TESS_END, glEnd);
gluTessCallback(tobj, GLU_TESS_COMBINE, myCombine);
gluTessBeginPolygon(tobj, NULL);
gluTessBeginContour(tobj);
gluTessVertex(tobj, v, v);
...
gluTessEndContour(tobj);
gluTessEndPolygon(tobj);
```

通常，镶嵌分块的多边形应当被存放在一个显示列表中，这样当绘制它时，就不必再重新镶嵌。

- 请参阅：
glBegin(), **glEdgeFlag()**, **glVertex()**, **gluNewTess()**, **gluErrorString()**, **gluTessVertex()**, **gluTessBeginPolygon()**, **gluTessBeginContour()**, **gluTessProperty()**, **gluTessNormal()**

gluTessEndPolygon

- 名称：

gluTessEndPolygon()

- 功能：

限定一个多边形的描述。

- C 描述:

```
void gluTessEndPolygon( GLUTesselator* tess )
```

- 参数说明:

tess 指定镶嵌分块对象 (由gluNewTess()建立)。

- 说明:

例程gluTessBeginPolygon()和gluEndPolygon()用来限定一个凸多边形、凹多边形或自交叉多边形的定义。在例程对gluTessBeginPolygon()/gluTessEndPolygon()之间必须一次或多次调用例程对gluTessBeginContour()/gluTessEndContour()。在每根轮廓线之中可以没有也可以调用多个gluTessVertex()。顶点指定了一条封闭的轮廓线 (每一条轮廓线的最后一个顶点自动地连到第一个顶点上)。有关细节请参阅gluTessVertex()、gluTessBeginContour()和gluTessEndContour()的参考说明。

一旦调用了例程gluTessEndPolygon(), 多边形将被镶嵌分块, 由此产生的三角形将通过反馈加以说明。反馈函数的有关细节请参阅例程gluTessCallback()。

- 示例:

以下命令用来定义一个含有三角形孔的四边形:

```
gluTessBeginPolygon(tobj, NULL);
gluTessBeginContour(tobj);
    gluTessVertex(tobj, v1, v1);
    gluTessVertex(tobj, v2, v2);
    gluTessVertex(tobj, v3, v3);
    gluTessVertex(tobj, v4, v4);
gluTessEndContour(tobj);
gluTessBeginContour(tobj);
    gluTessVertex(tobj, v5, v5);
    gluTessVertex(tobj, v6, v6);
    gluTessVertex(tobj, v7, v7);
gluTessEndContour(tobj);
gluTessEndPolygon(tobj);
```

上面示例中的指针v1~v7必须指向不同的地址, 这是因为调用例程gluTessEndPolygon()之前, 镶嵌分块器并没有读取存储于这些地址中的值。

- 请参阅:

gluNewTess(), **gluTessBeginContour()**, **gluTessVertex()**, **gluTessCallback()**,
gluTessProperty(), **gluTessNormal()**, **gluTessBeginPolygon()**

gluTessNormal

- 名称:

gluTessNormal()

- 功能:

指定一个多边形的法线。

- C描述：

```
void gluTessNormal( GLUtesselator* tess,
                      GLdouble valueX,
                      GLdouble valueY,
                      GLdouble valueZ )
```

- 参数说明：

tess 指定镶嵌分块对象（由例程gluNewTess()建立）。

valueX 指定法线的第一个组分。

valueY 指定法线的第二个组分。

valueZ 指定法线的第三个组分。

- 说明：

例程gluTessNormal()描述了一个程序所定义的多边形的法线。在进行镶嵌分块前，所有输入的数据将被投射到一个与三坐标轴之中的一个坐标轴垂直的平面上，并且所有输出的三角形将按相应法线的CCW方向定向（CW方向可以通过将所提供的法线的符号反向面得到）。例如，当你知道所有多边形位于x-y平面上时，在绘制这些多边形之前请先调用例程gluTessNormal (tess, 0.0, 0.0, 1.0)。

当所提供的法线值是 (0.0, 0.0, 0.0) (默认值) 时，法线将通过下面的方法确定：法线的方向，也就是它的符号，将通过顶点与平面之间的匹配情况来确定，这时并不考虑该顶点的连接方式。这时最好输入的数据近似地位于这个平面上，否则如果向三个坐标轴的一个垂直面投影时几何体的形状将严重变形。法线的符号是这样确定的：它必须保证输入轮廓线的所有带符号区域的加和结果是非负的（这里，CCW轮廓线所围成的区域是正区域）。

- 请参阅：

gluTessBeginPolygon(), **gluTessEndPolygon()**

■ gluTessProperty

- 名称：

gluTessProperty()

- 功能：

设置一个镶嵌分块对象的属性。

- C描述：

```
void gluTessProperty( GLUtesselator* tess,
                      GLenum which,
                      GLdouble data )
```

- 参数说明：

tess 指定镶嵌分块对象（由gluNewTess()建立）。

which 指定要设置的属性。它的有效值有：GLU_TESS_WINDING_RULE、GLU_TESS_BOUNDARY_ONLY、GLU_TESS_TOLERANCE。

data 指定表示属性的值。

- 说明：

例程gluTessProperty()用来控制一个镶嵌分块对象所具有的属性。这些属性将影响多边形的翻译和绘制。参数*which*可以取下面各值：

GLU_TESS_WINDING_RULE

确定多边形的哪些部分是“内部”的。参数*data*应设置成**GLU_TESS_WINDING_ODD**、**GLU_TESS_WINDING_NONZERO**、**GLU_TESS_WINDING_POSITIVE**或**GLU_TESS_WINDING_NEGATIVE**、或**GLU_TESS_WINDING_ABS_GEQ_TWO**。

要想理解缠绕规则如何工作，首先要明白输入的轮廓线将平面分成了一个个区域。缠绕规则就是用来确定它们中的哪些区域是在多边形的内部。

对于一条单独的轮廓线C，某个点x的缠绕数就简单地对应为一个带符号的数。这个带符号的数就是沿轮廓线C走一圈时绕点x所转的圈数（在这里，CCW是正数）。当有多条轮廓线时，各个独立的缠绕数将进行相加。这时平面上的每个点都对应为一个相加和后所得的带符号的正数值。值得注意的是每个单独区域中的所有点的缠绕数是相同的。

当某个区域缠绕数是属于某种选定的种类（单数、非零、正数、负数或至少是2的绝对值）时，缠绕规则将把这些区域分类为“内部”。以前的GLU镶嵌分块器（GLU 1.2以前）使用“单数”规则。“非零”规则则通过另一种通用的方法来定义内部。另外的三个规则对于多边形的CSG操作很有用。

GLU_TESS_BOUNDARY_ONLY

它是一个boolean值（该值应被设置成**GL_TRUE**或**GL_FALSE**）。当它被设置成**GL_TRUE**时，多边形将不通过镶嵌分块操作而用一组封闭的轮廓线来划分内部和外部。外部的轮廓线将同它的法线构成CCW方向；内部轮廓线是CW方向的。对于每条轮廓线，这时的**GLU_TESS_BEGIN**和**GLU_TESS_BEGIN_DATA**反馈函数的类型是**GL_LINE_LOOP**。

GLU_TESS_TOLERANCE

指定通过合并特性来减小输出尺寸的偏差值。例如，当两个顶点非常接近以至于彼此之间可以相互替代。偏差值同任一个输入顶点的最大坐标值相乘的结果用来表示可被移去的特性的最大距离，它将作为单个合并操作的结果。当几个合并操作都用到某个特性值时，总的可移去的距离将增大。

特性合并操作是完全可选的，偏移量仅仅是一种提示。实现在某些情况下可以进行合并操作而在另外的情况下却不能，或干脆不能合并任何特性值。偏差的初始值是0。

只有两个顶点完全重合，当前实现才会将顶点进行合并，而不考虑当前的偏差值。只有当实现不能确定一个顶点是位于某条边的哪边时，它才将该顶点并入这条边中。对于两条边而言，只有当它们的两个端点都重合时才将它们并在一起。

- 请参阅：

gluGetTessProperty(), **gluNewTess()**

• **gluTessVertex()**

- 名称：

gluTessVertex()

- 功能：

指定一个多边形的顶点。

- C 描述：

```
void gluTessVertex( GLUtesselator* tess,
                     GLdouble *location,
                     GLvoid* data )
```

- 参数说明：

tess 指定镶嵌分块对象（由**gluNewTess()**建立）。

location 指定顶点的存放位置。

data 指定一个通过顶点反馈（由**gluTessCallback()**指定）而返回到多边形的不透明的指针。

- 说明：

当定义一个多边形时，例程**gluTessVertex()**用来描述一个顶点。连续调用**gluTessVertex()**便可以指定一个封闭的轮廓线。例如，描述一个四边形需要调用**gluTessVertex()**四次。例程**gluTessVertex()**只能用在例程对**gluTessBeginContour()**/**gluTessEndContour()**之间。

参数*data*通常指向一个含有顶点存放位置的结构，就像其他诸如颜色和法线这样的单个顶点的属性一样。进行镶嵌分块之后，这个指针将通过**GLU_TESS_VERTEX**或**GLU_TESS_VERTEX_DATA**反馈函数返回给用户（请参阅 **gluTessCallback()**的参考说明）。

- 示例：

以下命令用来定义一个含有三角形孔的四边形：

```
gluTessBeginPolygon(tobj, NULL);
gluTessBeginContour(tobj);
gluTessVertex(tobj, v1, v1);
gluTessVertex(tobj, v2, v2);
gluTessVertex(tobj, v3, v3);
gluTessVertex(tobj, v4, v4);
gluTessEndContour(tobj);
gluTessBeginContour(tobj);
gluTessVertex(tobj, v5, v5);
gluTessVertex(tobj, v6, v6);
gluTessVertex(tobj, v7, v7);
gluTessEndContour(tobj);
gluTessEndPolygon(tobj);
```

- 注意：

将*location*或*data*用作局部变量是不对的,在循环过程中将值存入它们中间也是不允许的。例如:

```
for (i = 0; i < NVERTICES; ++i) {
    GLdouble data[3];
    data[0] = vertex[i][0];
    data[1] = vertex[i][1];
    data[2] = vertex[i][2];
    gluTessVertex(tobj, data, data);
}
```

它将不工作。这是由于在执行例程*gluTessEndPolygon()*之前,参数*location*和*data*所指定的指针将不能被间接引用。这时所有顶点的坐标(除了最近一次设置的坐标之外)将在镶嵌分块操作之前被重写。

该问题的两个错误是:第一,它由单个指针组成(当*data*使用一个局部变量时);第二,它产生了一个**GLU_TESS_NEED_COMBINE_CALLBACK**错误(当*location*使用一个局部变量时)。

- 请参阅:

gluTessBeginPolygon(), gluNewTess(), gluTessBeginContour(), gluTessCallback(), gluTessProperty(), gluTessNormal(), gluTessEndPolygon()

gluUnProject

- 名称:

gluUnProject()

- 功能:

将窗口坐标映射成对象坐标。

- C描述:

```
GLint gluUnProject( GLdouble winX,
                      GLdouble winY,
                      GLdouble winZ,
                      const GLdouble *model,
                      const GLdouble *proj,
                      const GLint *view,
                      GLdouble* objX,
                      GLdouble* objY,
                      GLdouble* objZ)
```

- 参数说明:

winX, *winY*, *winZ*

指定要映射的窗口坐标。

model 指定当前的模式取景矩阵(从调用*glGetDoublev()*时起)。

proj 指定当前的投影矩阵（从调用glGetDoublev()时起）。

view 指定当前的视口（从调用glGetIntegerv()时起）。

objX, objY, objZ

返回计算所得的对象坐标。

- 说明：

例程gluProject()用*model*, *proj*和*view*将指定的窗口坐标映射成对象坐标。所得的结果被存放在*objX*, *objY*和*objZ*中。返回值如果是GL_TRUE表示操作成功，如果是GL_FALSE表示操作失败。

为了计算坐标(*objX*, *objY*, *objZ*)，例程gluUnProject()将用*model*proj*的逆矩阵乘上当前的归一化设备坐标，如下：

$$\begin{aligned} \textit{objX} &= \frac{2(\textit{winX} - \textit{view}[0])}{\textit{view}[2]} - 1 \\ \textit{objY} &= \textit{INV}(\textit{PM}) \frac{2(\textit{winY} - \textit{view}[1])}{\textit{view}[3]} - 1 \\ \textit{objZ} &= \frac{2(\textit{winZ}) - 1}{\textit{W}} \\ &\quad 1 \end{aligned}$$

*INV()*表示矩阵的逆。*W*是一个不用的变量，这里只是用作相容矩阵符号。

- 请参阅：

glGet(), gluProject()

• gluUnProject4

- 名称：

gluUnProject4()

- 功能：

将窗口坐标和剪切坐标映射成对象坐标。

- C描述：

```
void gluUnProject( GLdouble winX,
                    GLdouble winY,
                    GLdouble winZ,
                    GLdouble clipW,
                    const GLdouble *model,
                    const GLdouble *proj,
                    const GLint *view,
                    GLdouble near,
                    GLdouble far,
                    GLdouble* objX,
                    GLdouble* objY,
```

```
GLdouble* objZ  
GLdouble* objW)
```

- 参数说明：

winX, winY, winZ

指定要映射的窗口坐标。

clipW

指定要映射的w剪切坐标。

model

指定当前的模式取景矩阵（从调用glGetDoublev()时起）。

proj

指定当前的投影矩阵（从调用glGetDoublev()时起）。

view

指定当前的视口（从调用glGetIntegerv()时起）。

near, far

指定最近和最远的平面（从调用glGetDoublev()时起）。

objX, objY, objZ, objW

返回计算所得的对象坐标。

- 说明：

例程gluProject4()用*model*、*proj*和*view*将指定的窗口坐标*winX*、*winY*和*winZ*及它的w剪切坐标*clipW*映射成对象坐标(*objX*, *objY*, *objZ*, *objW*)。当函数glFeedbackBuffer()返回的数据类型是**GL_4D_COLOR_TEXTURE**时，其顶点的*clipW*可以不止一个。这里也处理*near*和*far*平面不是默认值0和1的情况。返回值如果是**GL_TRUE**表示操作成功，如果是**GL_FALSE**表示操作失败。

为了计算坐标(*objX*, *objY*, *objZ*和*objW*)，例程gluUnProject4()将用*model * proj*的逆矩阵乘上当前的规格化设备坐标，如下：

$$\begin{aligned} \textit{objX} &= \text{INV}(PM) \frac{2(\textit{winX} - \textit{view}[0])}{\textit{view}[2]} - 1 \\ \textit{objY} &= \text{INV}(PM) \frac{2(\textit{winY} - \textit{view}[1])}{\textit{view}[3]} - 1 \\ \textit{objZ} &= \text{INV}(PM) \frac{2(\textit{winZ} - \textit{near})}{(\textit{far} - \textit{near})} - 1 \\ \textit{objW} &= \textit{clipW} \end{aligned}$$

*INV()*表示矩阵的逆。

当*clipW*等于1, *near*和*far*分别是0和1时，例程gluUnProject4()等同于gluUnProject()

- 注意：

例程gluUnProject4()仅在GLU 1.3以上的版本中才有效。

- 请参阅：

glGet(), glFeedbackBuffer(), gluProject(), gluUnProject()

- **gluBeginPolygon, gluEndPolygon**

- 名称：

gluBeginPolygon(), gluEndPolygon()

- 功能：

限定一个多边形的描述。

- C 描述：

```
void gluBeginPolygon(GLUtesselator* tess)
void gluEndPolygon ( GLUtesselator* tess )
```

- 参数说明：

tess 指定镶嵌分块对象（由 **gluNewTess()** 建立）。

- 说明：

例程 **gluBeginPolygon()** 和 **gluEndPolygon()** 用来限定一个非凸多边形的定义。如果要定义一个这样的多边形，首先调用例程 **gluBeginPolygon()**，然后通过在每个顶点调用 **gluTessVertex()** 定义该多边形的轮廓线，然后调用例程 **gluNextContour()** 来开始每个新的轮廓线。最后用例程 **gluEndPolygon()** 来作为定义过程结束的标志。

一旦调用了例程 **gluEndPolygon()**，多边形将被镶嵌分块，由此产生的三角形将通过反馈加以说明。反馈函数的有关细节请参阅例程 **gluTessCallback()**。

- 注意：

该命令已不再使用而仅用于向后兼容。当调用例程 **gluBeginPolygon()** 时，它将被映射为紧随例程 **gluTessBeginContour()** 之后的例程 **gluTessBeginPolygon()**；调用例程 **gluEndPolygon()** 时，它将被映射为紧随例程 **gluTessEndContour()** 之后的例程 **gluTessEndPolygon()**。

- 示例：

以下命令用来定义一个含有三角形孔的四边形：

```
;gluBeginPolygon(tess)
;(tv ,tv ,tv ,tv
;IntTessVertRef(cop[ ],cop[ ]
;(sv ,sv ,sv ,sv
;IntTessVertRef(cop[ ],cop[ ]
;(ev ,ev ,ev ,ev
;IntTessVertRef(cop[ ],cop[ ]
;(av ,av ,av ,av
;IntTessVertRef(cop[ ],cop[ ]
;gluNextContour(cop[ ],GLU_INTERIOR)
;(sv ,sv ,sv ,sv
;IntTessVertRef(cop[ ],cop[ ]
;(ev ,ev ,ev ,ev
;IntTessVertRef(cop[ ],cop[ ]
;(tv ,tv ,tv ,tv
;IntTessVertRef(cop[ ],cop[ ]
;gluEndPolygon(tess)
```

- 请参阅：

gluNewTess(), **gluNextContour()**, **gluTessCallback()**, **gluTessVertex()**, **gluTessBeginPolygon()**, **gluTessBeginContour()**

gluNextContour

- 名称：

gluNextContour()

- 功能：

标记另一个轮廓线的开始。

- C 描述：

```
void gluNextContour( GLUtesselator* tess,
```

GLenum *type*)

- 参数说明：

tess 指定镶嵌分块对象（由gluNewTess()建立）。

type 指定所定义的轮廓线的类型。它的有效值有：**GLU_EXTERIOR**, **GLU_INTERIOR**, **GLU_UNKNOWN**, **GLU_CCW**和**GLU_CW**。

- 说明：

例程gluNextContour()用多重轮廓线来描述多边形。通过调用一组gluTessVertex()例程生成第一条轮廓线后，用例程gluNextContour()来表示前一个轮廓线已经完成，下一个轮廓线将要开始。接下来调用另外的一组gluTessVertex()例程生成一条新的轮廓线。通过不断重复这一过程而生成所有轮廓线。

参数*type*用来指定轮廓线的类型。它允许的值如下：

GLU_EXTERIOR 一条用来定义多边形外边界的外部轮廓线。

GLU_INTERIOR 一条用来定义多边形内边界的内部轮廓线（例如一个孔）。

GLU_UNKNOWN 一条未知的轮廓线，通过程序库来确定它是内部轮廓线还是外部轮廓线。

GLU_CCW,

GLU_CW 认为定义的第一条**GLU_CCW**或**GLU_CW**轮廓线是外部轮廓线。其余的轮廓线如果跟第一条轮廓线同向（顺时针或逆时针），则表示它是外部的；否则，表示它是内部的。

如果一条轮廓线的类型是**GLU_CCW**或**GLU_CW**，则其他所有的轮廓线都必须是同类型的（否则，所有**GLU_CCW**或**GLU_CW**类型的轮廓线都将变成**GLU_UNKNOWN**类型的轮廓线）。

请注意，**GLU_CCW**和**GLU_CW**类型的轮廓线之间并没有本质上的区别。

描述第一条轮廓线之前应该先用例程gluNextContour()定义它的类型。否则，第一条轮廓线将被标记为**GLU_EXTERIOR**。

该命令已不再使用而仅用于向后兼容。当调用例程gluNextContour()时，它将映射为紧随例程gluTessBeginContour()之后的例程gluTessEndContour()。

- 示例：

下面的代码描述了一个内部含有三角形孔的四边形：

```
gluBeginPolygon(tobj);
    gluTessVertex(tobj, v1, v1);
    gluTessVertex(tobj, v2, v2);
    gluTessVertex(tobj, v3, v3);
    gluTessVertex(tobj, v4, v4);
    gluNextContour(tobj, GLU_INTERIOR);
    gluTessVertex(tobj, v5, v5);
    gluTessVertex(tobj, v6, v6);
    gluTessVertex(tobj, v7, v7);
gluEndPolygon(tobj);
```

- 请参阅：

gluBeginPolygon(), **gluNewTess()**, **gluTessCallback()**, **gluTessVertex()**, **gluTessBeginContour()**

第7章 GLX参考说明

本章按字母顺序列出了对X窗口系统的OpenGL扩展（GLX）所包含的所有例程。请读者先从例程glXIntro()的参考说明开始阅读本章，因为该节就OpenGL在X窗口系统中的情况作了一个简单的介绍。表7-1是本章所介绍的例程：

表 7-1

glXChooseFBConfig	glXGetCurrentDisplay	glXQueryDrawable
glXCopyContext	glXGetCurrentDrawable	glXQueryExtension
glXCreateNewContext	glXGetCurrentReadDrawable	glXQueryExtensionsString
glXCreatePbuffer	glXGetFBConfigAttrib	glXQueryServerString
glXCreatePixmap	glXGetFBConfigs	glXQueryVersion
glXCreatePixmap	glXGetSelectedEvent	glXSelectEvent
glXDestroyContext	glXGetVisualFromFBConfig	glXSwapBuffers
glXDestroyPbuffer	glXIntro	glXUseXFont
glXDestroyPixmap	glXIsDirect	glXWaitGL
glXDestroyWindow	glXMakeContextCurrent	glXWaitX
glXGetClientString	glXMakeCurrent	
glXGetCurrentContext	glXQueryContext	

表7-2一些命令的功能已经由GLX1.3中的新例程所替代，所以将它们的说明放在本章的末尾部分。虽然GLX1.3对它们的使用仍然提供支持，但建议在新开发的程序中尽量不要使用它们。

表 7-2

glXChooseVisual	glXCreateGLXPixmap	glXGetConfig
glXCreateContext	glXDestroyGLXPixmap	

◆ glXChooseFBConfig

- 名称：

glXChooseFBConfig()

- 功能：

返回一个与指定属性相匹配的GLX帧缓冲区配置的列表。

- C描述：

```
GLXFBCConfig *glXchooseFbconfig( Display *dpy,
                                    int screen,
                                    const int *attrib_list,
                                    int *nelements )
```

- 参数说明：

dpy 指定到X服务器的连接。

screen 指定屏幕的数目。

attrib_list

指定一个属性/数值对的列表。最后一个属性必须是**None**。

nelements

返回由例程glXChooseFBConfig()所返回的列表中元素的个数。

- 说明：

例程glXChooseFBConfig()将返回与*attrib_list*中的指定属性相匹配的GLX帧缓冲区的配置。如果没有发现任何匹配的配置，它将返回**NULL**。如果*attrib_list*是**NULL**，例程glXChooseFBConfig()将返回指定屏幕所使用的GLX帧缓冲区配置的数组。当有错误发生，指定的屏幕没有帧缓冲区配置存在，或者没有帧缓冲区配置与指定的属性相匹配时，将返回**NULL**。请用XFree()来释放由glXChooseFBConfig()返回的内存。

在*attrib_list*中的所有属性（包括逻辑属性）都有相应的理想值。该列表用**None**结束。如果*attrib_list*没有指定某个属性，那么它将使用它的缺省值（见下的具体叙述，这时的属性值是默认指定的）。例如，如果没有指定**GLX_STEREO**，则它将被设定成**False**。对于某些属性，如果其缺省值是**GLX_DONT_CARE**，则意味着关于该属性的任何值都已确定，所以该属性将不再被检查。

这里将用一种属性指定的方式来匹配某个属性。像**GLX_LEVEL**这样的属性，就必须用精确的指定值来匹配；而其他的属性如**GLX_RED_SIZE**则必须用等于或大于指定的最小值的值进行匹配。当发现有不止一个GLX帧缓冲区匹配时，它将返回一个依据“最优”匹配准则而挑选出来的匹配列表。每个属性的匹配准则和确切的分类次序定义如下。

各种GLX视觉属性的解释：

GLX_FBCONFIG_1D 后面必须跟一个有效的XID，它表示所期望的GLX帧缓冲区配置。当指定一个**GLX_FBCONFIG_1D**时，所有的属性将被忽略。其缺省值是**GLX_DONT_CARE**。

GLX_BUFFER_SIZE 后面必须跟一个非负的整数，它表示所期望的颜色索引缓冲区的尺寸。它将选出至少是指定尺寸的最小索引缓冲区。当在**GLX_RENDER_TYPE**中没有设置**GLX_COLOR_INDEX_BIT**时，该属性被忽略。其缺省值是0。

GLX_LEVEL 后面必须跟一个整数的缓冲区级别的规格。这个规格是非常重要的。0级缓冲区表示缺省的显示缓冲区。1缓缓冲区表示第一

	个覆盖帧缓冲区，2级缓冲区表示第二个覆盖帧缓冲区……依此类推。负数级别的缓冲区表示基础帧缓冲区。其缺省值是0。
GLX_DOUBLEBUFFER	后面必须跟 True 或 False 。当指定 True 时，仅考虑双缓冲的帧缓冲区配置；当指定 False 时，仅考虑单缓冲的帧缓冲区配置。其缺省值是 GLX_DONT_CARE 。
GLX_STEREO	后面必须跟 True 或 False 。当指定 True 时，仅考虑立体的帧缓冲区配置；当指定 False 时，仅考虑单象管的帧缓冲区配置。其缺省值是 False 。
GLX_AUX_BUFFERS	后面必须跟一个非负的整数，它表示所期望的辅助缓冲区的数目。它将选出等于或大于指定个数的配置中含有的最小辅助缓冲区数目的那个配置。其缺省值是0。
GLX_RED_SIZE	后面必须跟一个非负的最小尺寸规格。如果这个值是0，它将选出最小的有效红组分缓冲区。否则，它将选出最小尺寸的有效红组分缓冲区中最大的一个红组分缓冲区。其缺省值是0。
GLX_GREEN_SIZE	后面必须跟一个非负的最小尺寸规格。如果这个值是0，它将选出最小的有效绿组分缓冲区。否则，它将选出最小尺寸的有效绿组分缓冲区中最大的一个绿组分缓冲区。其缺省值是0。
GLX_BLUE_SIZE	后面必须跟一个非负的最小尺寸规格。如果这个值是0，它将选出最小的有效蓝组分缓冲区。否则，它将选出至少是最小尺寸的有效蓝组分缓冲区中最大的一个蓝组分缓冲区。其缺省值是0。
GLX_ALPHA_SIZE	后面必须跟一个非负的最小尺寸规格。如果这个值是0，它将选出最小的有效alpha组分缓冲区。否则，它将选出最小尺寸的有效alpha组分缓冲区中最大的一个alpha组分缓冲区。其缺省值是0。
GLX_DEPTH_SIZE	后面必须跟一个非负的最小尺寸规格。如果这个值是0，它将选出没有深度缓冲区的帧缓冲区配置。否则，它将选出最小尺寸的有效深度缓冲区中最大的一个深度缓冲区。其缺省值是0。
GLX_STENCIL_SIZE	后面必须跟一个非负的整数，它表示所期望的模板位面的数目。它将选出指定尺寸的模板缓冲区中最小的一个模板缓冲区。如果期望值是0，那么它将选出没有模板缓冲区的帧缓冲区配置。其缺省值是0。
GLX_ACCUM_RED_SIZE	后面必须跟一个非负的最小尺寸规格。如果这个值是0，它将选出没有红组分累积缓冲区的帧缓冲区配置。否则，它将选出能达到最小尺寸的红组分累积缓冲区中最可能的那个红组分累积缓冲区。其缺省值是0。

GLX_ACCUM_GREEN_SIZE

后面必须跟一个非负的最小尺寸规格。如果这个值是0，它将选出没有绿组分累积缓冲区的帧缓冲区配置。否则，它将选出能达到最小尺寸的绿组分累积缓冲区中最可能的那个绿组分累积缓冲区。其缺省值是0。

GLX_ACCUM_BLUE_SIZE

后面必须跟一个非负的最小尺寸规格。如果这个值是0，它将选出没有蓝组分累积缓冲区的帧缓冲区配置。否则，它将选出能达到最小尺寸的蓝组分累积缓冲区中最可能的那个蓝组分累积缓冲区。其缺省值是0。

GLX_ACCUM_ALPHA_SIZE

后面必须跟一个非负的最小尺寸规格。如果这个值是0，它将选出没有alpha组分累积缓冲区的帧缓冲区配置。否则，它将选出能达到最小尺寸的alpha组分累积缓冲区中最可能的那个alpha组分累积缓冲区。其缺省值是0。

GLX_RENDER_TYPE

后面必须跟一个屏蔽值，它表示帧缓冲区配置必须支持的一种OpenGL绘制模式。有效的位值有**GLX_RGBA_BIT**和**GLX_COLOR_INDEX_BIT**。如果将该屏蔽值设置成**GLX_RGBA_BIT|GLX_COLOR_INDEX_BIT**，则只有可以与**RGBA**环境和颜色索引环境相匹配的帧缓冲区配置才可以被考虑。其缺省值是**GLX_RGBA_BIT**。

GLX_DRAWABLE_TYPE

后面必须跟一个屏蔽值，它表示帧缓冲区配置必须支持的一种GLX绘图类型。有效的位值有**GLX_WINDOW_BIT**、**GLX_PIXMAP_BIT**和**GLX_PBUFFER_BIT**。例如，如果将该屏蔽值设置成**GLX_WINDOW_BIT|GLX_PIXMAP_BIT**，则只有同时支持Windows和GLX像素映射的帧缓冲区配置才被考虑。其缺省值是**GLX_WINDOW_BIT**。

GLX_X_RENDERABLE

后面必须跟**True**或**False**。当指定**True**时，只有含有相应的X视觉环境（并能用于Windows和/或GLX像素映射绘制）的帧缓冲区配置才能被考虑。其缺省值是**GLX_DONT CARE**。

GLX_X_VISUAL_TYPE

后面必须跟**GLX_TRUE_COLOR**、**GLX_DIRECT_COLOR**、**GLX_PSEUDO_COLOR**、**GLX_STATIC_COLOR**、**GLX_GRAY_SCALE**或**GLX_STATIC_GRAY**之一。它表示所期望的X视觉环境的类型。当**GLX_DRAWABLE_TYPE**被在

*attrib_list*中指定并且后面跟着的屏蔽值不含有**GLX_WINDOW_BIT**设置时，该值将被忽略。另外，当**GLX_X_RENDERABLE**被指定成**False**时，该值也将被忽略。

RGBA绘制可以被以下的视觉环境类型所支持：**GLX_TRUE_COLOR**、**GLX_DIRECT_COLOR**、**GLX_PSEUDO_COLOR**或**GLX_STATIC_COLOR**；但颜色索引绘制仅被视觉环境类型：**GLX_PSEUDO_COLOR**或**GLX_STATIC_COLOR**（如单通道视觉环境）所支持。

记号**GLX_GRAY_SCALE**和**GLX_STATIC_GRAY**将无法与当前启动的OpenGL视觉环境相匹配，但它们却包含在将来的应用中。

GLX_X_VISUAL_TYPE的缺省值是**GLX_DONT_CARE**。

GLX_CONFIG_CAVEAT

后面必须跟**GLX_NONE**、**GLX_SLOW_CONFIG**或**GLX_NON_CONFORMANT_CONFIG**之一。当它被指定成**GLX_NONE**时，它将只考虑没有警告说明的帧缓冲区配置；当它被指定成**GLX_SLOW_CONFIG**时，它将只考虑慢帧缓冲区配置；当它被指定成**GLX_NON_CONFORMANT_CONFIG**时，它将只考虑非一致性的帧缓冲区配置。其缺省值是**GLX_DONT_CARE**。

GLX_TRANSPARENT_TYPE

后面必须跟**GLX_NONE**、**GLX_TRANSPARENT_RGB**或**GLX_TRANSPARENT_INDEX**。当它被指定成**GLX_NONE**时，它将只考虑不透明的帧缓冲区配置；当它被指定成**GLX_TRANSPARENT_RGB**时，它将只考虑支持RGBA绘制的透明帧缓冲区配置；当它被指定成**GLX_TRANSPARENT_INDEX**时，它将只考虑支持颜色索引绘制的透明帧缓冲区配置。其缺省值是**GLX_NONE**。

GLX_TRANSPARENT_INDEX_VALUE

后面必须跟一个整数值，它表示透明的索引值。该值必须在0和索引的最大帧缓冲区的值之间。只有使用指定的透明索引值的帧缓冲区配置才能被考虑。其缺省值是**GLX_DONT_CARE**。

除非**GLX_TRANSPARENT_TYPE**被包含在*attrib_list*中并且被指定为**GLX_TRANSPARENT_INDEX**，否则，该属性将被忽略。

GLX_TRANSPARENT_RED_VALUE

后面必须跟一个整数值，它表示透明的红组分值。该值必须在0和

红组分的最大帧缓冲区的值之间。只有使用指定的透明红组分值的帧缓冲区配置才能被考虑。其缺省值是**GLX_DONT_CARE**。

除非**GLX_TRANSPARENT_TYPE**被包含在*attrib_list*中并且被指定为**GLX_TRANSPARENT_RGB**, 否则, 该属性将被忽略。

GLX_TRANSPARENT_GREEN_VALUE

后面必须跟一个整数值, 它表示透明的绿组分值。该值必须在0和绿组分的最大帧缓冲区的值之间。只有使用指定的透明绿组分值的帧缓冲区配置才能被考虑。其缺省值是**GLX_DONT_CARE**。除非**GLX_TRANSPARENT_TYPE**被包含在*attrib_list*中并且被指定为**GLX_TRANSPARENT_RGB**, 否则, 该属性将被忽略。

GLX_TRANSPARENT_BLUE_VALUE

后面必须跟一个整数值, 它表示透明的蓝组分值。该值必须在0和蓝组分的最大帧缓冲区的值之间。只有使用指定的透明蓝组分值的帧缓冲区配置才能被考虑。其缺省值是**GLX_DONT_CARE**。除非**GLX_TRANSPARENT_TYPE**被包含在*attrib_list*中并且被指定为**GLX_TRANSPARENT_RGB**, 否则, 该属性将被忽略。

GLX_TRANSPARENT_ALPHA_VALUE

后面必须跟一个整数值, 它表示透明的alpha组分值。该值必须在0和alpha组分的最大帧缓冲区的值之间。只有使用指定的透明alpha组分值的帧缓冲区配置才能被考虑。其缺省值是**GLX_DONT_CARE**。

除非**GLX_TRANSPARENT_TYPE**被包含在*attrib_list*中并且被指定为**GLX_TRANSPARENT_RGB**, 否则, 该属性将被忽略。

当发现不止一个GLX帧缓冲区与指定的属性匹配时, 它将返回一个匹配的配置表。该表将依据下面的优先规则进行排序, 这些规则采用升序(例如, 如果一个较低优先数的规则所选定的配置将由较高优先数的规则排序):

- (1) **GLX_CONFIG_CAVEAT**, 这时的优先次序是**GLX_NONE**、**GLX_SLOW_CONFIG**和**GLX_NON_CONFORMANT_CONFIG**。
- (2) **RGBA**颜色组分的总数目(**GLX_RED_SIZE**, **GLX_GREEN_SIZE**、**GLX_BLUE_SIZE**和**GLX_ALPHA_SIZE**的加和)越大, 它具有的二进制位的数目就越高。对于一种特定的颜色组分, 如果*attrib_list*中所要求的二进制位的数目是0或**GLX_DONT_CARE**, 则该组分中二进制位的数目将不予考虑。
- (3) 较小的**GLX_BUFFER_SIZE**。
- (4) 单缓冲区配置(**GLX_DOUBLEBUFFER**设置成**False**)优先于双缓冲区配置。
- (5) 较小的**GLX_AUX_BUFFERS**。
- (6) 较大的**GLX_DEPTH_SIZE**。
- (7) 较小的**GLX_STENCIL_SIZE**。

(8) 累积缓冲区颜色组分的总数目（**GLX_ACCUM_RED_SIZE**、**GLX_ACCUM_GREEN_SIZE**、**GLX_ACCUM_BLUE_SIZE**和**GLX_ACCUM_ALPHA_SIZE**的加和）越大，它具有的二进制位的数目就越高。对于一种特定的颜色组分，如果**attrib_list**中所要求的二进制位的数目是0或**GLX_DONT_CARE**，则该组分中二进制位的数目将不予考虑。

(9) **GLX_X_VISUAL_TYPE**，这时的优先次序是**GLX_TRUE_COLOR**、**GLX_DIRECT_COLOR**、**GLX_PSEUDOCOLOR**、**GLX_STATIC_COLOR**、**GLX_GRAY_SCALE**、**GLX_STATIC_GRAY**。

- 示例：

```
attrib_list= { GLX_RENDER_TYPE, GLX_RGBA_BIT, GLX_RED_SIZE, 4, GLX_GREEN_SIZE, 4, GLX_BLUE_SIZE, 4, None };
```

指定一个帧缓冲区配置，它支持RGBA绘制，位于法线缓冲区中，并且没有一个覆盖或衬垫缓冲区。返回的视觉环境支持红、绿和蓝组分至少各四位，但可能没有alpha组分位。它不支持立体显示。它可能有一个或多个辅助颜色缓冲区、一个后缓冲区、一个深度缓冲区、一个模板缓冲区或一个累积缓冲区，也可能没有这些缓冲区。

- 注意：

例程**glXChooseFBConfig()**仅在GLX 1.3以上的版本中才有效。

如果GLX的版本是1.1或1.0，则要求GL的版本必须是1.0。如果GLX的版本是1.2，则要求GL的版本必须是1.1。如果GLX的版本是1.3，则要求GL的版本必须是1.2。

例程**glXGetFBConfigs()**和**glXGetFBConfigAttrib()**可以用来实现算法选取，而不像例程**glXChooseFBConfig()**所实现的一般功能。调用例程**glXChooseFBConfig()**可以返回一个特定屏幕上的所有帧缓冲区配置，或者用一组特定的属性返回所有帧缓冲区的配置。但注意：这两种方式只能选择其中一种。接下来调用例程**glGetFBConfigAttrib()**来返回帧缓冲区配置的另外属性，然后在它们中间进行选择。

虽然这一作法并不被禁止，但GLX强烈建议用户不要使用例程**glXChooseFBConfig()**来选择算法。因此，客户端的数据库的版本变化可能会改变改变的结果。

- 出错提示：

当**attrib_list**中出现了未定义的GLX属性，或**screen**无效，或**dpy**不支持GLX扩展时；都将返回NULL。

- 请参阅：

glXGetFBConfigAttrib(), **glXGetFBConfigs()**, **glXGetVisualFromFBConfig()**

glXCopyContext

- 名称：

glXCopyContext()

- 功能：

将一个绘图环境中的状态复制到另一个环境中。

- C描述：

```
void glXCopyContext( Display *dpy,
              GLXContext src,
              GLXContext dst,
              unsigned long mask )
```

- 参数说明：

dpy 指定到X服务器的连接。
src 指定源环境。
dst 指定目标环境。
mask *k*指定*src*状态的哪些部分将被复制到*dst*中。

- 说明：

例程glXCopyContext()将把*src*中选定的一组状态变量复制到*dst*中。参数*mask*表示哪一组状态变量将被复制。参数*mask*中包含了与送入GL命令glPushAttrib()中的符号同名的二进制或操作。单一符号常量**GL_ALL_ATTRIB_BITS**可以用来复制绘图状态的最大的可能部分。

只有当*src*和*dst*所指定的绘制操作共享一个地址空间时复制操作才能进行。当两个绘图环境都间接地使用同一个服务器，或它们都被一个进程直接共有时，它们将共享一个地址空间。值得注意的是间接的情况下正在调用的线程不必共享一个地址空间，这时只要它们关联的绘图环境共享一个地址空间就可以了。

并非所有的GL状态值都能被复制。例如，像素的打包和解包状态、绘图模式状态、以及选择和反馈状态都不能被复制。能被复制的状态都是可以被GL命令glPushAttrib()所处理的状态。

如果*src*是正在调用的线程的当前绘图环境，例程glXCopyContext()将执行一个隐含的glFlush()操作。

- 注意：

一个“进程”就是一个单独的运行环境，它将在一个单独的地址空间中执行，并且包含一个或多个线程。

一个“线程”就是一组共享一个单一地址空间的子进程中的一个子进程，但它保留单独的程序计数器、堆栈空间以及其他相关的全局数据。如果一个“线程”是它的子进程组的唯一成员，则该“线程”相当于一个“进程”。

- 出错提示：

当绘图环境*src*和*dst*没有共享一个地址空间，或它们不是为同一个屏幕所创建时产生**BadMatch**提示。

如果调用例程glXCopyContext()时，*dst*对所有线程（包括正在调用的线程）都是当前的，则产生**BadAccess**提示。

如果*src*是当前的绘图环境，但当前的可绘区域是一个无效的窗口时，则产生**GLXBadCurrentWindow**提示。

当*src*或*dst*都是一个无效的GLX环境时，产生**GLXBadContext**提示。

- 请参阅：

glPushAttrib(), **glXCreateContext()**, **glXIsDirect()**

- **glXCreateNewContext**

- 名称:

glXCreateNewContext()

- 功能:

建立一个新的GLX绘图环境。

- C描述:

```
GLXContext glXCreateNewContext( Display *dpy
```

```
                GLXFBConfig config,
```

```
                int render_type,
```

```
                GLXContext share_list,
```

```
                Bool direct )
```

- 参数说明:

dpy 指定到X服务器的连接。

config 采用期望的属性来为绘图环境指定GLXFBConfig结构。

render_type 指定将要创建的绘图环境类型。它必须是**GLX_RGBA_TYPE**或**GLX_COLOR_INDEX_TYPE**中的一个。

share_list 指定用来共享显示列表的绘图环境。**NULL**表示不共享。

direct 指定绘制操作是直接连接到图形系统(**True**)还是通过X服务器连接到图形系统(**False**)。

- 说明:

例程**glXCreateNewContext()**建立一个GLX绘图环境并返回它的处理结果。这个环境可以用来绘入GLX窗口、像素映射或像素缓冲区。如果该例程建立绘制环境失败，则返回**NULL**。

如果*render_type*是**GLX_RGBA_TYPE**，则建立一个支持**RGB A**绘图的环境。如果*render_type*是**GLX_COLOR_INDEX_TYPE**，则建立一个支持颜色索引绘图的环境。

如果*render_type*不是**NULL**，则所有显示列表索引和定义将被环境*render_type*和新建立的环境共享。一个单独的显示列表空间可以被任意数目的环境所共享。然而，所有共享一个显示列表空间的绘图环境都必须存在于相同的地址空间中。如果两个绘图环境都间接地使用相同的服务器或者它们被一个进程所直接共有，则它们可以共享一个地址空间。值得注意的是间接的情况下正在调用的线程不必共享一个地址空间，这时只要它们关联的绘图环境共享一个地址空间就可以了。

当*direct*是**True**时，如果设备支持直接绘图，到X服务器的连接是局部的，并且允许一个直接绘图环境，则可以建立一个直接绘图环境。(当*direct*是**True**时，一个设备也可以返回一个间接环境。)如果*direct*是**False**，则总是建立一个通过X服务器绘制的绘图环境。在某些设备上，直接绘图有着性能优势。然而，直接绘图在一个单独进程之外是不能共享直接绘图环境的，而且它可能无法对GLX像素映射进行绘制。

- 注意:

例程**glXCreateNewContext()**仅在GLX 1.3以上的版本中才有效。

如果GLX的版本是1.1或1.0，则要求GL的版本必须是1.0。如果GLX的版本是1.2，则要求GL的版本必须是1.1。如果GLX的版本是1.3，则要求GL的版本必须是1.2。

- 出错提示：

如果客户端执行失败，则返回NULL提示。

如果*render_type*既不是一个GLX环境又不是NULL，则产生**GLXBadContext**提示。

如果*config*不是一个有效的GLXFBCConfig，则产生**GLXBadFBConfig**提示。

如果将要建立的环境不共享地址空间，或不共享由*render_type*指定的环境屏幕，则产生**BadMatch**提示。

如果服务器没有足够的资源来配置新的环境，则产生**BadAlloc**提示。

如果*config*不是一个有效的视觉环境（例如，如果一个特定的GLX设备不支持它时），则产生**BadValue**提示。

- 请参阅：

glXChooseFBConfig(), **glXCreateContext()**, **glXDestroyContext()**, **glXGetFBConfigs()**,
glXGetFBConfigAttrib(), **glXIsDirect()**, **glXMakeContextCurrent()**

• **glXCreatePbuffer**

- 名称：

glXCreatePbuffer()

- 功能：

建立一个屏幕外的绘图区域。

- C描述：

```
GLXPbuffer glXCreatePbuffer( Display *dpy,
                           GLXFBCConfig config,
                           const int *attrib_list )
```

- 参数说明：

dpy 指定到X服务器的连接。

config 采用所期望的属性来为窗口指定一个GLXFBCConfig结构。

attrib_list

指定属性数值对的列表。它必须用**None**或NULL来结束。可以接受的属性值有：**GLX_PBUFFER_WIDTH**、**GLX_PBUFFER_HEIGHT**、**GLX_PRESERVED_CONTENTS**和**GLX_LARGEST_PBUFFER**。

- 说明：

例程**glXCreatePbuffer()**建立了一个屏幕外的绘图区域并返回它的XID。任何根据*format*建立的GLX绘图环境都可以用来绘入这个窗口。请使用例程**glXMakeContextCurrent()**来将该绘图区域跟一个GLX绘图环境相连接。

对于每个GLXPbuffer，可接受的属性有：

GLX_PBUFFER_WIDTH

指定所需要的GLXPbuffer的像素宽度。其缺省值是0。

GLX PBUFFER HEIGHT

指定所需要的GLXPbuffer的像素高度。其缺省值是0。

GLX LARGEST PBUFFER

如果所需要的配置失败，它将指定所要获取的最大有效像素缓冲区。它所配置的像素缓冲区的宽度和高度将分别小于指定值**GLX_PBUFFER_WIDTH**和**GLX_PBUFFER_HEIGHT**。可以使用例程**glXQueryDrawable()**来恢复已配置的像素缓冲区的尺寸。其缺省值是**False**。

GLX PRESERVED CONTENTS

当发生资源冲突时，它用来确定是否保存像素缓冲区中的内容。如果它被设置成**False**，则像素缓冲区中的内容可能会随时丢失；如果是**True**，或根本就没有在*attrib_list*中指定，则像素缓冲区中的内容将被保存（这一过程就像是将帧缓冲区中的内容拷贝进主系统的内存中一样）。这两种情况都允许用户登录（当像素缓冲区的内容被保存或破坏时，请使用例程glXSelectEvent()来接收产生的像素缓冲区的冲突事件）。

- 注意 •

例程glXCreatePbuffer()仅在GLX 1.3以上的版本中才有效。

如果GLX的版本是1.1或1.0，则要求GL的版本必须是1.0。如果GLX的版本是1.2，则要求GL的版本必须是1.1。如果GLX的版本是1.3，则要求GL的版本必须是1.2。

GLXPbuffers是由帧缓冲区资源配置而成的，当将它们使用完成后应该考虑将它们解除配置。

- ### • 出錯提示 •

如果用来配置GLXPbuffer的资源不足，则产生BadAlloc提示。

如果config不是一个有效的GLXFBConfig，则产生**GLXBadFBConfig**提示。

如果*config*不支持对像素缓冲区的绘图（比如**GLX_DRAWABLE_TYPE**中不包含**GLX_PBUFFER_BIT**），则产生**BadMatch**提示。

- 请参阅 •

`glXChooseFBConfig()`, `glXCreatePixmap()`, `glXMakeContextCurrent()`, `glXSelectEvent()`

• glXCreatePixmap

- ### • 名称 •

glXCreatePixman()

- 功能 •

建立一个屏幕外的绘图区域

iFC 描述

GlxXPixmap **gIXCreatePixmap**(**Display** **display*,

GL_YERConfig config

```
Pixmap pixmap,
const int *attrib_list)
```

• 参数说明：

dpy 指定到X服务器的连接。

config 用所期望的属性来为窗口指定一个GLXFBConfig结构。

pixmap 指定用作绘图区域的X像素映射。

attrib_list 当前并不使用。它必须被设置成NULL或是一个空表（例如第一个元素为None的表）。

• 说明：

例程glXCreatePixmap()建立了一个屏幕外的绘图区域并返回它的XID。任何根据*format*建立的GLX绘图环境都可以用来绘入这个窗口。请使用例程glXMakeContextCurrent()来将该绘图区域跟一个GLX绘图环境相连接。

• 注意：

例程glXCreatePixmap()仅在GLX1.3以上的版本中才有效。

如果GLX的版本是1.1或1.0，则要求GL的版本必须是1.0。如果GLX的版本是1.2，则要求GL的版本必须是1.1。如果GLX的版本是1.3，则要求GL的版本必须是1.2。

• 出错提示：

如果*pixmap*不是由与*config*相对应的一个视觉环境所建立，则产生BadMatch提示。

如果*config*不支持向窗口的绘图（例如GLX_DRAWABLE_TYPE中不包含GLX_PIXMAP_BIT），则产生BadMatch提示。

如果*pixmap*不是一个有效的像素映射的XID，则产生BadPixmap提示。

如果已经有一个GLXFBConfig与*pixmap*相关联，则产生BadAlloc提示。

如果X服务器不能配置一个新的GLX窗口，则产生BadAlloc提示。

如果*config*不是一个有效的GLXFBConfig，则产生GLXBadFBConfig提示。

• 请参阅：

glXChooseFBConfig(), glXCreateGLXPixmap(), glXDestroyPixmap(), glXMakeContextCurrent()

glXCreateWindow

• 名称：

glXCreateWindow()

• 功能：

建立一个屏幕上的绘图区域。

• C描述：

```
GLXWindow glXCreateWindow( Display *dpy
                           GLXFBConfig config,
                           GLXWindow win,
```

```
const int *attrib_list)
```

- 参数说明：

dpy 指定到X服务器的连接。

config 用所期望的属性来为窗口指定一个GLXFBConfig结构。

win 指定用作绘图区域的X窗口。

attrib_list

当前并不使用。它必须被设置成NULL或是一个空表（例如第一个元素为**None**的表）。

- 说明：

例程**glXCreateWindow()**从一个已经存在的X窗口中建立一个屏幕上的绘图区域，这个已经存在的X窗口是由一个同*config*相匹配的视觉环境建立的。GLXWindow的XID将被返回。任何根据*format*建立的GLX绘图环境都可以用来绘入这个窗口。请使用例程**glXMakeContextCurrent()**来将该绘图区域跟一个GLX绘图环境相关联。

- 注意：

例程**glXCreateWindow()**仅在GLX 1.3以上的版本中才有效。

如果GLX的版本是1.1或1.0，则要求GL的版本必须是1.0。如果GLX的版本是1.2，则要求GL的版本必须是1.1。如果GLX的版本是1.3，则要求GL的版本必须是1.2。

- 出错提示：

如果*win*，不是由与*config*相对应建立的一个视觉环境所建立，则产生**BadMatch**提示。

如果*config*不支持向窗口的绘图（例如**GLX_DRAWABLE_TYPE**中不包含**GLX_WINDOW_BIT**），则产生**BadMatch**提示。

如果*win*不是一个有效的窗口XID，则产生**BadWindow**提示。

如果已经有一个GLXFBConfig与*win*相关联，则产生**BadAlloc**提示。

如果X服务器不能配置一个新的GLX窗口，则产生**BadAlloc**提示。

如果*config*不是一个有效的GLXFBConfig，则产生**GLXBadFBConfig**提示。

- 请参阅：

glXChooseFBConfig(), **glXCreateGLXPixmap()**, **glXDestroyWindow()**, **glXMakeContextCurrent()**

glXDestroyContext

- 名称：

glXDestroyContext()

- 功能：

删除一个GLX环境。

- C描述：

```
void glXDestroyContext( Display *dpy,
                        GLXContext ctx )
```

- 参数说明：

dpy 指定到X服务器的连接。

ctx 指定要删除的GLX环境。

- 说明：

如果对任何线程而言GLX绘图环境*ctx*都不是当前的，则例程**glXDestroyContext()**将立刻删除它。否则，要一直等到它对于任何线程都不是当前时才能被删除。对于上面的两种情况，都将立刻释放被*ctx*所利用的资源的ID。

- 出错提示：

如果*ctx*不是一个有效的GLX环境，则产生**GLXBadContext**提示。

- 请参阅：

glXCreateContext(), **glXCreateNewContext()**, **glXMakeCurrent()**

glXDestroyPbuffer

- 名称：

glXDestroyPbuffer()

- 功能：

删除一个屏幕外的绘图区域。

- C描述：

```
void glXDestroyPbuffer( Display *dpy,
                        GLXPbuffer pbuf )
```

- 参数说明：

dpy 指定到X服务器的连接。

pbuf 指定要删除的GLXPbuffer。

- 说明：

例程**glXDestroyPbuffer()**将删除一个由**glXCreatePbuffer()**所建立的GLXPbuffer。

- 注意：

例程**glXDestroyPbuffer()**仅在GLX 1.3以上的版本中才有效。

如果GLX的版本是1.1或1.0，则要求GL的版本必须是1.0。如果GLX的版本是1.2，则要求GL的版本必须是1.1。如果GLX的版本是1.3，则要求GL的版本必须是1.2。

- 出错提示：

如果*pbuf*不是一个有效的GLXPbuffer，则产生**GLXBadPbuffer**提示。

- 请参阅：

glXChooseFBConfig(), **glXCreatePbuffer()**

glXDestroyPixmap

- 名称：

glXDestroyPixmap()

- 功能：

删除一个屏幕外的绘图区域。

- C 描述：

```
void glXDestroyPixmap( Display *dpy,
                      GLXPixmap pixmap )
```

- 参数说明：

dpy 指定到X服务器的连接。

pixmap 指定要删除的GLXPixmap。

- 说明：

例程glXDestroyPixmap()将删除一个由glXCreatePixmap()所建立的GLXPixmap。

- 注意：

例程glXDestroyPixmap()仅在GLX 1.3以上的版本中才有效。

如果GLX的版本是1.1或1.0，则要求GL的版本必须是1.0。如果GLX的版本是1.2，则要求GL的版本必须是1.1。如果GLX的版本是1.3，则要求GL的版本必须是1.2。

- 出错提示：

如果*pixmap*不是一个有效的GLXPixmap，则产生GLXBadPixmap提示。

- 请参阅：

glXChooseFBConfig(), glXCreatePixmap(), glXDestroyGLXPixmap()

glXDestroyWindow

- 名称：

glXDestroyWindow()

- 功能：

删除一个屏幕上的绘图区域。

- C 描述：

```
void glXDestroyWindow( Display *dpy,
                      GLXWindow win )
```

- 参数说明：

dpy 指定到X服务器的连接。

win 指定要删除的GLXWindow。

- 说明：

例程glXDestroyWindow()将删除一个由glXCreateWindow()所建立的GLXWindow。

- 注意：

例程glXDestroyWindow()仅在GLX 1.3以上的版本中才有效。

如果GLX的版本是1.1或1.0，则要求GL的版本必须是1.0。如果GLX的版本是1.2，则要求GL的版本必须是1.1。如果GLX的版本是1.3，则要求GL的版本必须是1.2。

- 出错提示：

如果*win*不是一个有效的GLXPixmap，则产生GLXBadWindow提示。

- 请参阅：

glXChooseFBConfig(), **glXCreateWindow()**

• **glXGetClientString**

- 名称：

glXGetClientString()

- 功能：

返回一个描述客户端状况的字符串。

- C描述：

```
const char * glXGetClientString( Display *dpy,
                                int name )
```

- 参数说明：

dpy 指定到X服务器的连接。

name 指定一个要返回的字符串。它可以是**GLX_VENDOR**、**GLX_VERSION**或**GLX_EXTENSIONS**。

- 说明：

例程**glXGetClientString()**返回一个用来描述某些客户端数据库的字符串。参数*name*可能的值有**GLX_VENDOR**, **GLX_VERSION**和**GLX_EXTENSIONS**。如果*name*没有被设置成这些值，则例程**glXGetClientString()**返回NULL。卖方字符串的格式和内容由设备确定。

扩展信息的字符串是空结束的，它包含一个用空格隔开的扩展名清单。（该扩展名中不能含有空格。）如果没有向OLX的扩展，则返回空字符串。

版本字符串的格式如下：

<major_version.minor_version><space><vendor-specific info>

其中版本号的主要和次要部分的长度都是任意的。卖方所指定的信息是可选项。但是，如果有该项，则它的格式和内容将由设备指定。

- 注意：

例程**glXGetClientString()**仅在GLX 1.1以上的版本中才有效。

如果GLX的版本是1.1或1.0，则要求GL的版本必须是1.0。如果GLX的版本是1.2，则要求GL的版本必须是1.1。如果GLX的版本是1.3，则要求GL的版本必须是1.2。

例程**glXGetClientString()**仅返回由客户端所支持的GLX扩展的信息，如果要获得有关服务器端所支持的OL扩展的信息，请调用例程**glGetString()**。

- 请参阅：

glXQueryVersion(), **glXQueryExtensionsString()**, **glxQueryServerString()**

• **glXGetCurrentContext**

- 名称：

glXGetCurrentContext()

- 功能：

返回当前环境。

- C描述：

GLXContext glXGetCurrentContext(void)

- 说明：

例程**glXGetCurrentContext()**将返回由**glXMakeCurrent()**所指定的当前环境。如果没有当前环境，则返回NULL。

例程**glXGetCurrentContext()**返回的是客户端的信息，它不循环访问服务器。

- 请参阅：

glXCreateContext(), glXGetCurrentDisplay(), glXGetCurrentDrawable(), glXMakeCurrent()

◆ **glXGetCurrentDisplay**

- 名称：

glXGetCurrentDisplay()

- 功能：

为当前环境提供一个显示。

- C描述：

GLXDisplay glXGetCurrentDisplay(void)

- 说明：

例程**glXGetCurrentDisplay()**将为当前环境提供一个显示。如果没有当前环境，则返回NULL。

例程**glXGetCurrentDisplay()**返回的是客户端的信息，它不循环访问服务器，因此也不冲掉任何未决的事件。

- 注意：

例程**glXGetCurrentDisplay()**仅被GLX 1.2以上的版本支持。

- 请参阅：

glXGetCurrentContext(), glXGetCurrentDrawable(), glXQueryVersion(), glXQueryExtensionsString()

◆ **glXGetCurrentDrawable**

- 名称：

glXGetCurrentDrawable()

- 功能：

返回当前可绘区域。

- C描述：

GLXDrawable glXGetCurrentDrawable(void)

- 说明：

例程glXGetCurrentDrawable()将返回由glXMakeCurrent()所指定的当前可绘区域。如果没有当前环境，则返回None。

例程glXGetCurrentDrawable()返回的是客户端的信息，它不循环访问服务器。

- 请参阅：

glXCreateGLXPixmap(), glXGetCurrentContext(), glXGetCurrentDisplay(),
glXGetCurrentReadDrawable(), glXMakeCurrent()

• glXGetCurrentReadDrawable

- 名称：

glXGetCurrentReadDrawable()

- 功能：

返回当前可绘区域。

- C描述：

GLXDrawable glXGetCurrentReadDrawable(void)

- 说明：

例程glXGetCurrentReadDrawable()将返回由例程glXMakeContextCurrent()的参数read所指定的当前可读绘图环境。如果没有当前可绘区域，则返回None。

例程glXGetCurrentReadDrawable()返回的是客户端的信息，它不循环访问服务器。

- 注意：

例程glXGetCurrentReadDrawable()仅被GLX 1.3以上的版本支持。

- 请参阅：

glXGetCurrentContext(), glXGetCurrentDisplay(), glXGetCurrentDrawable(),
glXMakeContextCurrent()

• glXGetFBConfigAttrib

- 名称：

glXGetFBConfigAttrib()

- 功能：

返回一个GLX帧缓冲区配置的信息。

- C描述：

```
int glXGetFBConfigAttrib( Display *dpy,
                           GLXFBConfig config,
                           int attribute,
                           int *value )
```

- 参数说明：

dpy 指定到X服务区的连接。

config 指定所要查询的GLX帧缓冲区配置。

attribute 指定要返回的属性。

value 返回所需的值。

- 说明：

例程glXGetFBConfigAttrib()将*value*设置成根据*config*而建立的GLX可绘区域的*attribute*值。当操作失败时，例程glXGetFBConfigAttrib()将返回一个出错代码；否则，返回Success。

参数*attribute*是下列各值之一：

GLX_FBCONFIG_ID 所给出的GLXFBCConfig的XID。

GLX_BUFFER_SIZE 它表示每个颜色缓冲区中二进制位的数目。如果帧缓冲区配置支持RGBA环境，则**GLX_BUFFER_SIZE**是**GLX_RED_SIZE**、**GLX_GREEN_SIZE**、**GLX_BLUE_SIZE**和**GLX_ALPHA_SIZE**之和。如果帧缓冲区配置仅支持颜色索引环境，则**GLX_BUFFER_SIZE**表示颜色索引的尺寸。

GLX_LEVEL 它表示配置的帧缓冲区层数。0层代表默认的帧缓冲区。正的层数表示覆盖在默认缓冲区上面的帧缓冲区，负的层数表示衬垫在默认缓冲区下面的帧缓冲区。

GLX_DOUBLEBUFFER

如果颜色缓冲区存在可交换的前/后缓冲区对，它返回True；否则，返回False。

GLX_STEREO 如果颜色缓冲区存在左/右缓冲区对，它返回True；否则，返回False。

GLX_AUX_BUFFERS

它代表有效的辅助颜色缓冲区的数目。0表示不存在辅助颜色缓冲区。

GLX_RED_SIZE

它代表存储于每个颜色缓冲区的红组分中二进制位的数目。如果帧缓冲区配置不支持RGBA环境，它将是未定义的。

GLX_GREEN_SIZE

它代表存储于每个颜色缓冲区的绿组分中二进制位的数目。如果帧缓冲区配置不支持RGBA环境，它将是未定义的。

GLX_BLUE_SIZE

它代表存储于每个颜色缓冲区的蓝组分中二进制位的数目。如果帧缓冲区配置不支持RGBA环境，它将是未定义的。

GLX_ALPHA_SIZE

它代表存储于每个颜色缓冲区的alpha组分中二进制位的数目。如果帧缓冲区配置不支持RGBA环境，它将是未定义的。

GLX_DEPTH_SIZE

它代表深度缓冲区中二进制位的数目。

GLX_STENCIL_SIZE

它代表模板缓冲区中二进制位的数目。

GLX_ACCUM_RED_SIZE

它代表存储于累积缓冲区的红组分中二进制位的数目。

GLX_ACCUM_GREEN_SIZE

它代表存储于累积缓冲区的绿组分中二进制位的数目。

GLX_ACCUM_BLUE_SIZE

它代表存储于累积缓冲区的蓝组分中二进制位的数目。

GLX_ACCUM_ALPHA_SIZE

它代表存储于累积缓冲区的alpha组分中二进制位的数目。

GLX_RENDER_TYPE

它代表一个屏蔽值，这个屏蔽值表示对帧缓冲区配置而言哪种GLX环境可以成为当前的GLX环境。有效的位值有**GLX_RGBA_BIT**和**GLX_COLOR_INDEX_BIT**。

GLX_DRAWABLE_TYPE

它代表一个屏蔽值，这个屏蔽值表示帧缓冲区配置支持哪类绘图类型。有效的位值有**GLX_WINDOW_BIT**、**GLX_PBUFFER_BIT**、**MAP_BIT**和**GLX_PBUFFER_BIT**。

GLX_X_RENDERABLE

如果由帧缓冲区配置所建立的绘图环境可以被X绘入，则返回True。

GLX_VISUAL_ID

它代表相应视觉环境的XID；或者当没有相应的视觉环境时（例如，当**GLX_X_RENDERABLE**是False，或**GLX_DRAWABLE_TYPE**没有**GLX_WINDOW_BIT**的位设置），它将返回0。

GLX_X_VISUAL_TYPE

它代表相应的视觉环境的类型。如果没有相应的视觉环境存在，（例如，当**GLX_X_RENDERABLE**是False，或**GLX_DRAWABLE_TYPE**没有**GLX_WINDOW_BIT**的位设置）；它的返回值将是下面各值之一：**GLX_TRUE_COLOR**、**GLX_DIRECT_COLOR**、**GLX_PSEUDO_COLOR**、**GLX_STATIC_COLOR**、**GLX_GRAY_SCALE**或**GLX_STATIC_GRAY**或**GLX_NONE**。

GLX_CONFIG_CAVEAT

它将是**GLX_NONE**、**GLX_SLOW_CONFIG**或**GLX_NON_CONFORMANT_CONFIG**之一。它们分别表示没有警告说明的帧缓冲区配置，某些方面比其他帧缓冲区配置运行速度慢的帧缓冲区配置或某些方面存在不一致性的帧缓冲区配置。

GLX_TRANSPARENT_TYPE

它将是**GLX_NONE**、**GLX_TRANSPARENT_RGB**或**GLX_TRANSPARENT_INDEX**之一。它们分别表示不透明的帧缓冲区配置，对于某些特定的红、绿和蓝值透明的帧缓冲区配置或对于某些特定的颜色索引值透明的帧缓冲区配置。

GLX_TRANSPARENT_INDEX_VALUE

它将返回在0和索引的最大帧缓冲区的值之间的整数值，表示帧缓冲区配置的透明索引值。如果**GLX_TRANSPARENT_TYPE**

不是**GLX_TRANSPARENT_INDEX**, 它将是未定义的。

GLX_TRANSPARENT_RED_VALUE

它将返回在0和红组分的最大帧缓冲区的值之间的整数值, 表示帧缓冲区配置的透明红组分值。如果**GLX_TRANSPARENT_TYPE**不是**GLX_TRANSPARENT_RGB**, 它将是未定义的。

GLX_TRANSPARENT_GREEN_VALUE

它将返回在0和绿组分的最大帧缓冲区值之间的整数值, 表示帧缓冲区配置的透明绿组分值。如果**GLX_TRANSPARENT_TYPE**不是**GLX_TRANSPARENT_RGB**, 它将是未定义的。

GLX_TRANSPARENT_BLUE_VALUE

它将返回在0和蓝组分的最大帧缓冲区值之间的整数值, 表示帧缓冲区配置的透明蓝组分值。如果**GLX_TRANSPARENT_TYPE**不是**GLX_TRANSPARENT_RGB**, 它将是未定义的。

GLX_TRANSPARENT_ALPHA_VALUE

它将返回在0和alpha组分的最大帧缓冲区的值之间的整数值, 表示帧缓冲区配置的透明alpha组分值。如果**GLX_TRANSPARENT_TYPE**不是**GLX_TRANSPARENT_RGB**, 它将是未定义的。

GLX_MAX_PBUFFER_WIDTH

可以为glXCreateGLXPbuffer()指定的最大宽度。

GLX_MAX_PBUFFER_HEIGHT

可以为glXCreateGLXPbuffer()指定的最大高度。

GLX_MAX_PBUFFER_PIXELS

一个像素缓冲区的最大像素数目(宽度和高度的乘积)。值得注意的是这个值可能会小于**GLX_MAX_PBUFFER_WIDTH**与**GLX_MAX_PBUFFER_HEIGHT**的乘积。同时, 该值是静态的, 并且假设不再有其他的像素缓冲区或X资源来竞争帧缓冲区内存。因此, 它可能会导致无法配置一个由**GLX_MAX_PBUFFER_PIXELS**给定尺寸的像素缓冲区。

应用程序应该选用最接近它们所需求的帧缓冲区配置。用不必要的缓冲区建立窗口、GLX像素映射或GLX像素缓冲区将导致绘图速度的降低, 就像资源配置不足一样。

- 注意:

例程glXGetFBConfigAttrib()只有在GLX 1.3以上的版本中才有效。

如果GLX的版本是1.1或1.0, 则要求GL的版本必须是1.0。如果GLX的版本是1.2, 则要求GL的版本必须是1.1。如果GLX的版本是1.3, 则要求GL的版本必须是1.2。

- 出错提示:

如果*dpy*不支持GLX扩展，则产生**GLX_NO_EXTENSION**提示。

如果*attribute*不是一个有效的GLX属性，则产生**GLX_BAD_ATTRIBUTE**提示。

- 请参阅：

glXGetFBConfigs(), **glXChooseFBConfig()**, **glXGetVisualFromFBConfig()**, **glXGetConfig()**

• **glXGetFBConfigs**

- 名称：

glXGetFBConfigs()

- 功能：

列出一个给定屏幕的所有GLX帧缓冲区配置。

- C描述：

```
GLXFBConfig *glXGetFBConfigs( Display *dpy,
                               int screen,
                               int *nelements )
```

- 参数说明：

dpy 指定到X服务区的连接。

screen 指定屏幕的数目。

nelements

返回所需的GLXFBConfig数目。

- 说明：

例程**glXGetFBConfigs()**将返回一个对于由*screen*指定的屏幕有效的全部GLXFBConfigs的列表。用例程**glXGetFBConfigAttrib()**来从一个指定的GLXFBConfig中获取属性值。

- 注意：

例程**glXGetFBConfigs()**只有在GLX 1.3以上的版本中才有效。

如果GLX的版本是1.1或1.0，则要求GL的版本必须是1.0。如果GLX的版本是1.2，则要求GL的版本必须是1.1。如果GLX的版本是1.3，则要求GL的版本必须是1.2。

- 请参阅：

glXGetFBConfigAttrib(), **glXGetVisualFromFBConfig()**, **glXChooseFBConfig()**

• **glXGetSelectedEvent**

- 名称：

glXGetSelectedEvent()

- 功能：

返回为一个窗口或GLX像素缓冲区所选定的GLX事件。

- C描述：

```
void glXGetSelectedEvent( Display *dpy,
                           GLXDrawable draw,
                           unsigned long *event_mask)
```

- 参数说明：

dpy 指定到X服务器的连接。

draw 指定一个GLX可绘区域。它必须是一个GLX像素缓冲区或一个窗口。

event_mask

返回由*draw*所选定的事件。

- 说明：

例程glXGetSelectedEvent()将把由*draw*所选定的事件返回到*event_mask*中。

- 注意：

例程glXGetSelectedEvent()只有在GLX 1.3以上的版本中才有效。

如果GLX的版本是1.1或1.0，则要求GL的版本必须是1.0。如果GLX的版本是1.2，则要求GL的版本必须是1.1。如果GLX的版本是1.3，则要求GL的版本必须是1.2。

- 出错提示：

如果*dpy*不是一个有效的窗口或不是一个有效的GLX像素缓冲区，则产生**GLXBadDrawable**提示。

- 请参阅：

glXSelectEvent(), glXCreatePbuffer()

glXGetVisualFromFBConfig

- 名称：

glXGetVisualFromFBConfig()

- 功能：

返回一个与帧缓冲区配置相关联的视觉环境。

- C描述：

```
XVisualInfo *glXGetVisualFromFBConfig( Display *dpy,
                                         GLXFBConfig config )
```

- 参数说明：

dpy 指定到X服务器的连接。

config 指定GLX帧缓冲区的配置。

- 说明：

如果*config*是一个有效的GLX帧缓冲区配置并且它有一个相关联的X视觉环境，则返回用来描述该视觉环境的信息。否则，返回NULL。请使用Xfree()释放返回的数据。

- 注意：

例程glXGetVisualFromFBConfig()只有在GLX 1.3以上的版本中才有效。

如果GLX的版本是1.1或1.0，则要求GL的版本必须是1.0。如果GLX的版本是1.2，则要求GL的版本必须是1.1。如果GLX的版本是1.3，则要求GL的版本必须是1.2。

XvisualInfo在Xutil.h中定义。它是一种包含*visual*、*visualID*、*screen*和*depth*元素的结构。

- 出错提示：

如果*config*不是一个有效的GLXFBCConfig，则返回NULL。

- 请参阅：

glXGetFBConfigAttrib(), **glXChooseFBConfig()**, **glXChooseVisual()**, **glXGetConfig()**

- **glXIntro**

- 名称：

glXIntro()

- 功能：

在X窗口系统中引入OpenGL。

- 简介：

OpenGL(其他章节中称其为GL)是一个高性能的3D导向的绘图工具。通过GLX扩展，它可以被用于X窗口系统中。如果要查询一个X服务器是否支持GLX扩展以及支持什么版本，请使用例程**glXQueryExtension()**和**glXQueryVersion()**。

GLX通过扩展X服务器的一个视觉环境子集而使它有效地支持OpenGL绘图。通过这些视觉环境而建立的可绘区域也可以用核心的X绘图工具或其他任何一个兼容所有核心的X视觉环境的X扩展来进行绘制工作。

GLX用附加的缓冲区来扩展了一个可绘制的标准的颜色缓冲区。这些附加缓冲区包括后缓冲区和辅助缓冲区、一个深度缓冲区、一个模板缓冲区和一个颜色累积缓冲区。每个支持OpenGL的X视觉环境都包括了上面所列的一些或全部缓冲区。

GLX支持向三种类型的可绘区域绘制：窗口、像素映射和像素缓冲区。GLX窗口和像素映射是X资源，它接受核心的X绘图的能力同使用OpenGL绘图的能力一样优秀。GLX像素缓冲区是GLX独有的资源，它不能接受核心的X绘图。

为了使用OpenGL向一个GLX可绘区域绘图，你必须确定适当的GLXFBCConfig来支持你的应用所要求的绘图特性。例程**glXChooseFBConfig()**用来返回一个同所要求的属性相匹配的GLXFBCConfig，如果没有发现相应的匹配，它将返回NULL。你也可以通过调用例程**glXGetFBConfigs()**来获取一个服务器所支持的GLXFBCConfig完整清单。如果你要查询一个特定的GLXFBCConfig属性，请调用例程**glXGetFBConfigAttrib()**。

对于GLX窗口和像素映射来说，首先应该用相匹配的视觉环境建立一个适当的X可绘区域（分别用**XCreateWindow()**和**XCreatePixmap()**）。你可以调用例程**glXGetVisualFromFBConfig()**来获取建立X可绘区域所必需的XVisualInfo结构。对于像素缓冲区，则不要求基础X可绘区域。

如果要从一个X窗口建立一个GLX窗口，请调用例程**glXCreateWindow()**。类似地，可以用例程**glXCreatePixmap()**建立一个像素映射，用例程**glXCreatePbuffer()**建立一个像素缓冲区。用例程**glXDestroyWindow()**, **glXDestroyPixmap()**和**glXDestroyPbuffer()**来释放以前分配的资源。

每个GLX的环境都要求将一个OpenOL绘图环境同一个GLX资源连接在一起。每个GLX资源和绘图环境都必须含有兼容的GLXFBCConfigs。请调用例程**glXCreateNewContext()**来建立一个GLX环境。你可以用例程**glXMakeContextCurrent()**将一个环境同一个GLX可绘区域连接起来。这时，这个环境/可绘区域对就变成了当前环境和当前可绘区域。它们将被所有的OpenGL绘图命

令使用直到用不同的参数调用glXMakeContextCurrent()为止。

核心的X命令和OpenGL命令都可以用来对可绘区域进行操作，然而，X命令和OpenGL命令流是不同步的。通过调用例程glXWaitGL()、glXWaitX()、XSync()和XFlush()来明确地指定它们的同步性。

- 示例：

下面是一个小型的示例，其目的是用来建立一个RGBA格式的、与使用GLX1.3命令的OpenGL相兼容的X窗口。当程序执行后，该窗口将被刷新成黄颜色。该程序进行最小错误检测；所有返回值都将被检测。

```
#include <stdio.h>
#include <stdlib.h>
#include <GL/gl.h>
#include <GL/glx.h>

int singleBufferAttributes[] = {
    GLX_DRAWABLE_TYPE, GLX_WINDOW_BIT,
    GLX_RENDER_TYPE,   GLX_RGBA_BIT,
    GLX_RED_SIZE,      1, /* Request a single buffered color buffer */
    GLX_GREEN_SIZE,    1, /* with the maximum number of color bits */
    GLX_BLUE_SIZE,     1, /* for each component */
    None
};

int doubleBufferAttributes[] = {
    GLX_DRAWABLE_TYPE, GLX_WINDOW_BIT,
    GLX_RENDER_TYPE,   GLX_RGBA_BIT,
    GLX_DOUBLEBUFFER,  True, /* Request a double-buffered color buffer with */
    GLX_RED_SIZE,      1, /* the maximum number of bits per component */
    GLX_GREEN_SIZE,    1,
    GLX_BLUE_SIZE,     1,
    None
};

static Bool WaitForNotify( Display *dpy, XEvent *event, XPointer arg ) {
    return (event->type == MapNotify) && (event->xmap.window == (Window) arg);
}

int main( int argc, char *argv[] )
{
    Display          *dpy;
    Window           xWin;
    XEvent           event;
    XVisualInfo      *vInfo;
    XSetWindowAttributes swa;
    GLXFBConfig      *fbConfigs;
```

```

GLXContext      context;
GLXWindow       glxWin;
int             swaMask;
int             numReturned;
int             swapFlag = True;

/* Open a connection to the X server */
dpy = XOpenDisplay( NULL );
if ( dpy == NULL ) {
    printf( "Unable to open a connection to the X server0" );
    exit( EXIT_FAILURE );
}

/* Request a suitable framebuffer configuration - try for a double
** buffered configuration first */
fbConfigs = glXChooseFBConfig( dpy, DefaultScreen(dpy),
                               doubleBufferAttributes, &numReturned );

if ( fbConfigs == NULL ) ( /* no double buffered configs available */
    fbConfigs = glXChooseFBConfig( dpy, DefaultScreen(dpy),
                                  singleBufferAttributes, &numReturned );
    swapFlag = False;
)

/* Create an X colormap and window with a visual matching the first
** returned framebuffer config */
vInfo = glXGetVisualFromFBConfig( dpy, fbConfigs[0] );

swa.border_pixel = 0;
swa.event_mask = StructureNotifyMask;
swa.colormap = XCreateColormap( dpy, RootWindow(dpy, vInfo->screen),
                               vInfo->visual, AllocNone );

swaMask = CWBorderPixel | CWColormap | CWEEventMask;
xWin = XCreateWindow( dpy, RootWindow(dpy, vInfo->screen), 0, 0, 256, 256,
                      0, vInfo->depth, InputOutput, vInfo->visual,
                      swaMask, &swa );

/* Create a GLX context for OpenGL rendering */
context = glXCreateNewContext( dpy, fbConfigs[0], GLX_RGBA_TYPE,
                             NULL, True );

/* Create a GLX window to associate the frame buffer configuration
** with the created X window */
glxWin = glXCreateWindow( dpy, fbConfigs[0], xWin, NULL );

/* Map the window to the screen, and wait for it to appear */

```

```

XMapWindow( dpy, (xWin );
XIfEvent( dpy, &event, WaitForNotify, (XPointer) xWin );

/* Bind the GLX context to the Window */
glXMakeContextCurrent( dpy, glxWin, glxWin, context );

/* OpenGL rendering ... */
glClearColor( 1.0, 1.0, 0.0, 1.0 );
glClear( GL_COLOR_BUFFER_BIT );

glFlush();

if ( swapFlag )
    glXSwapBuffers( dpy, glxWin );

sleep( 10 );
exit( EXIT_SUCCESS );
}

```

- 注意：

必须建立一个X颜色映射并将它送给**XCreateWindow()**。

在OpenGL命令执行之前，必须建立一个GLX环境并将它同一个GLX可绘区域相连接。如果当前没有环境/可绘区域对，执行OpenGL命令将成为未定义的行为。

所谓的“曝光事件”是指所有与指定窗口相连接的缓冲区均遭到了破坏，需要重画。即使有一些系统中的一些视觉环境的某些缓冲区（如深度缓冲区）可能不需要重画；但对于编写程序，假定这些缓冲区将不被破坏也是不正确的。

GLX命令使用的是**XVisualInfo**结构而不是视觉环境的指针或直接的visualIDs。**XVisualInfo** 结构包含*visual*, *visualID*, *screen*和*depth*元素，就像其他X指定的信息。

使用GLX扩展

所有被支持的GLX扩展都将在glx.h中有一个相应的定义，并且在例程**glXQueryExtensionsString()** 所返回的扩展字符串中有一个代号。例如，当支持**EXT_visual_info**扩展时，这该代号将被定义在glx.h中，并且出现在由例程**glXQueryExtensionsString()**所返回的扩展字符串中。在glx.h中的定义可以在编译程序时用来确定与一个扩展相应的程序调用是否存在与该程序库中。

OpenGL自身也能被扩展，有关细节请参考**glIntro()**。

GLX 1.1、GLX 1.2和GLX

现在所支持的版本是GLX 1.3，它向下兼容GLX 1.1和GLX 1.2。它引进的新功能（即所谓的OLXFBCConfigs）替代了GLX 1.2的功能。但它仍然支持GLX 1.2的命令，只是在新应用程序开发时不推荐使用这些命令。

GLX 1.3对应OpenGL1.2，并引进了如下的新例程：**glXGetFBConfigs()**、**glXGetFBConfigAttrib()**、**glXGetVisualFromFBConfig()**、**glXCreateWindow()**、**glXDestroyWindow()**、**glXCreatePixmap()**、**glXDestroyPixmap()**、**glXCreatePbuffer()**、**glXDestroyPbuffer()**、**glXQueryDrawable()**、**glXCreateNewContext()**、**glXMakeContextCurrent()**、**glXGetCurrentRead-**

Drawable()、glXGetCurrentDisplay()、glXQueryContext()、glXSelectEvent()、glXGetSelectedEvent()。

GLX 1.2对应OpenGL 1.1，并引进了如下的新例程：glXGetCurrentDisplay()。

GLX 1.1对应OpenGL 1.0，并引进了如下的新例程：glXQueryExtensionsString()、glXQueryServerString()、glXGetClientString()。

调用例程glQueryVersion()可以确定运行时哪个GLX版本是有效的。该函数返回连接所支持的版本。这样，如果返回的是1.3，则说明客户端和服务器端支持的版本都是GLX 1.3。你也可以在编译时检查GLX的版本：如果支持的是GLX 1.1，则将把GLX_VERSION_1_1定义在glx.h中；如果支持的是GLX 1.2，则将把GLX_VERSION_1_2定义在glx.h中；如果支持的是GLX 1.3，则将把GLX_VERSION_1_3定义在glx.h中。

- 请参阅：

glIntro(), glFinish(), glFlush(), glXChooseVisual(), glXCopyContext(), glXCreateContext(), glXCreateGLXPixmap(), glXDestroyContext(), glXGetClientString(), glXGetConfig(), glXIsDirect(), glXMakeCurrent(), glXQueryExtension(), glXQueryExtensionsString(), glXQueryServerString(), glXQueryVersion(), glXSwapBuffers(), glXUseXFont(), glXWaitGL(), glXWaitX(), glXGetFBConfig(), glXGetFBConfigAttrib(), glXGetVisualFromFBConfig(), glXCreateWindow(), glXDestroyWindow(), glXCreatePixmap(), glXDestroyPixmap(), glXCreatePbuffer(), glXDestroyPbuffer(), glXQueryDrawable(), glXCreateNewContext(), glXMakeContextCurrent(), glXGetCurrentReadDrawable(), glXGetCurrentDisplay(), glXQueryContext(), glXSelectEvent(), glXGetSelectedEvent(), XCreateColormap(), XCreateWindow(), XSync()

• **glXIsDirect**

- 名称：

glXIsDirect()

- 功能：

表示是否启动直接绘图模式。

- C描述：

```
Bool glXIsDirect( Display *dpy,
                  GLXContext ctx )
```

- 参数说明：

dpy 指定到X服务器的连接。

ctx 指定要查询的GLX环境。

- 说明：

如果*ctx*是一个直接绘图环境，例程glXIsDirect()返回True；否则，返回False。直接绘图环境从正在执行的进程的地址空间向绘图系统直接传送命令，而绕过了X服务器。间接绘图环境将把所有绘图命令都送到X服务器。

- 出错提示：

如果`ctx`不是一个有效的GLX环境，则产生**GLXBadContext**提示。

- 请参阅：

glXCreateContext()

glXMakeContextCurrent

- 名称：

glXMakeContextCurrent()

- 功能：

将一个GLX环境同一个GLX可绘区域相关联。

- C描述：

```
Bool glXMakeContextCurrent( Display *display,
                           GLXDrawable draw,
                           GLXDrawable read,
                           GLXContext ctx )
```

- 参数说明：

`display` 指定到X服务器的连接。

`draw` 指定要绘入的GLX可绘区域。它必须是一个代表GLXWindow、GLXPixmap或GLXPbuffer的XID。

`read` 指定一个将从中读取的GLX可绘区域。它必须是一个代表GLXWindow、GLXPixmap或GLXPbuffer的XID。

`ctx` 指定与`read`和`ctx`相关联的GLX环境。

- 说明：

例程**glXMakeContextCurrent()**将`ctx`连接到当前的绘图线程和`draw`和`read`的GLX可绘区域。参数`draw`和`read`可以是同一个可绘区域。

参数`draw`可用于所有的OpenGL操作，除了以下几种情况：

(1) 基于**GL_READ_BUFFER**的值而读取的任何像素数据。值得注意的是用**GL_READ_BUFFER**的值进行的累积操作，如果`draw`和`read`不相同，该操作将被禁止；

(2) 任何由**glReadPixels()**或**glCopyPixels()**所返回的深度值；

(3) 任何由**glReadPixels()**或**glCopyPixels()**所返回的模板值。

从`draw`中取得帧缓冲区的值。

如果当前的绘图线程含有一个当前的绘图环境，则该环境将被`ctx`冲掉并替换。

当`ctx`第1次被设置成当前环境时，其视口和裁剪尺寸被设置成`draw`可绘区域的尺寸。当`ctx`被再次设置成当前环境时，视口和裁剪尺寸将不再被修改。

如果要释放当前的环境但又不分配新的环境，请调用例程**glXMakeContextCurrent()**，这时要将`draw`和`read`设置成**None**，将`ctx`设置成**NULL**。

如果例程**glXMakeContextCurrent()**调用成功，它将返回**True**；否则，返回**False**。如果返

回**False**，则以前的当前绘图环境和可绘区域（如果有）将保持不变。

- 注意：

例程**glXMakeContextCurrent()**只有在GLX 1.3以上的版本中才有效。

如果GLX的版本是1.1或1.0，则要求GL的版本必须是1.0。如果GLX的版本是1.2，则要求GL的版本必须是1.1。如果GLX的版本是1.3，则要求GL的版本必须是1.2。

- 出错提示：

如果*draw*和*read*不兼容，则产生**BadMatch**提示。

如果*ctx*是其他线程的当前环境，则产生**BadAccess**提示。

如果存在一个当前绘制环境，并且它的绘图模式是**GL_FEEDBACK**或**GL_SELECT**，则产生**GLXBadContextState**提示。

如果*ctx*不是一个有效的GLX绘图环境，则产生**GLXBadContext**提示。

如果*draw*或*read*不是一个有效的GLX可绘区域，则产生**GLXBadDrawable**提示。

如果基础的X窗口对于*draw*或*read*不再有效，则产生**GLXBadWindow**提示。

如果正在调用的线程的以前环境中还有未冲掉的GL命令并且以前的可绘区域已经不再有效，则产生**GLXBadCurrentDrawable**提示。

如果X服务器没有足够的资源来配置缓冲区，则产生**BadAlloc**提示。

如果出现下面情况，则产生**BadMatch**提示：

(1) 参数*draw*和*read*不能同时适合帧缓冲区存储器；

(2) 参数*draw*或*read*是一个GLXPixmap而*ctx*却是一个直接绘图环境；

(3) 参数*draw*或*read*是一个GLXPixmap而*ctx*早已被连接到一个GLXWindow或GLXPbuffer；

(4) 参数*draw*和*read*是一个GLXWindow或GLXPbuffer而*ctx*早已被连接到一个GLXPixmap。

- 请参阅：

glXCreateNewContext(), **glXCreateWindow()**, **glXCreatePixmap()**, **glXCreatePbuffer()**,
glXDestroyContext(), **glXGetCurrentContext()**, **glXGetCurrentDisplay()**, **glXGetCurrent-Drawable()**, **glXGetCurrentReadDrawable()**, **glXMakeCurrent()**

glXMakeCurrent

- 名称：

glXMakeCurrent()

- 功能：

将一个GLX环境连接到一个窗口或一个GLX像素映射。

- C描述：

```
Bool glXMakeCurrent( Display *dpy
                      GLXDrawable drawable,
                      GLXContext ctx )
```

- 参数说明：

dpy 指定到X服务器的连接。

drawable 指定一个GLX的可绘区域。它必须是一个X窗口ID或一个GLX像素映射ID。

ctx 指定一个GLX绘图环境。它将被连接到*drawable*。

- 说明：

例程glXMakeCurrent()做两件事：它将*ctx*设置成正在调用的线程的当前GLX绘图环境，并同时替换以前的当前环境（如果存在以前的当前环境）；它将*ctx*连接到一个GLX可绘区域（可以是一个窗口或一个GLX像素映射）。其结果是，随后的GL绘图调用将用绘图环境*ctx*来修正GLX的可绘区域*drawable*（用于读取和写入）。由于例程glXMakeCurrent()总是用*ctx*替换当前的绘图环境，所以每个线程都只有一个当前环境。

如果以前环境中有未完成的命令，则执行本例程之前要先冲掉（限时完成）这些命令。

当*ctx*第1次被设置成某一线程的当前环境时，它的视口将被设置成*drawable*的满尺寸。随后用*ctx*调用例程glXMakeCurrent()的任何线程都不再影响*ctx*的视口。

如果要释放当前的环境但又不分配新的环境，请调用例程glXMakeCurrent()，这时要将*drawable*设置成None，将*ctx*设置成NULL。

如果例程glXMakeCurrent()调用成功，它将返回True；否则，返回False。如果返回False，则以前的当前绘图环境和可绘区域（如果有）将保持不变。

- 注意：

一个“进程”就是一个单独的运行环境，它将在一个单一地址空间中执行，并且包含一个或多个线程。

一个“线程”就是一组共享一个单一地址空间的子进程中的一个子进程，但它含有单独的程序计数器、堆栈空间以及其他相关的全局数据。一个“线程”如果是它的子进程组的唯一成员，则该“线程”相当于一个“进程”。

- 出错提示：

如果*drawable*不是被*ctx*相同的X屏幕和视觉环境建立，或者当*drawable*是None或*ctx*是NULL，则产生BadMatch提示。

如果调用例程glXMakeCurrent()时，环境*ctx*正好是其他线程的当前环境，则产生BadAccess提示。

如果*drawable*不是一个有效的GLX可绘区域，则产生GLXBadDrawable提示。

当*ctx*不是一个有效的GLX环境时产生GLXBadContext提示。

当例程glXMakeCurrent()在函数对glBegin()/glEnd()之间执行时，产生GLXBadContextState提示。

如果正在调用的线程的当前绘图环境是GL绘图状态GL_FEEDBACK或GL_SELECT时，也将产生GLXBadContextState提示。

如果以前环境还有未完成的GL命令并且当前的可绘区域是一个已经无效的窗口时，产生GLXBadCurrentWindow提示。

如果直到调用例程glXMakeCurrent()时才发现已经没有足够的资源来完成相关配置，而使得服务器推迟配置辅助缓冲区，则产生BadAlloc提示。

- 请参阅：

glXCreateContext(), **glXCreateGLXPixmap()**, **glXGetCurrentContext()**, **glXGetCurrentDisplay()**, **glXGetCurrentDrawable()**, **glXGetCurrentReadDrawable()**, **glXMakeContextCurrent()**

• **glXQueryContext**

- 名称:

glXQueryContext()

- 功能:

查询环境信息。

- C描述:

```
int glXQueryContext( Display *dpy
                      GLXContext ctx,
                      int attribute,
                      int *value )
```

- 参数说明:

dpy 指定到X服务器的连接。

ctx 指定一个GLX绘图环境。

attribute

指定将要返回的环境参数。它必须是**GLX_FBCONFIG_ID**、**GLX_RENDER_TYPE**或**GLX_SCREEN**之一。

value 它包含*attribute*的返回值。

- 说明:

例程**glXQueryContext()**将根据*ctx*的情况而将*value*设置成*attribute*的值。参数*attribute*可以是下列各值之一:

GLX_FBCONFIG_ID 返回与*ctx*相关联的GLXFBCConfig的XID。

GLX_RENDER_TYPE 返回*ctx*所支持的绘图类型。

GLX_SCREEN 返回与*ctx*相关联的屏幕数。

当参数*attribute*不是一个有效的GLX环境参数时，返回**GLX_BAD_ATTRIBUTE**；否则，返回**Success**。

调用本例程将引起一个对服务器进行循环访问。

- 注意:

例程**glXQueryContext()**只有在GLX1.3以上的版本中才有效。

如果GLX的版本是1.1或1.0，则要求GL的版本必须是1.0。如果GLX的版本是1.2，则要求GL的版本必须是1.1。如果GLX的版本是1.3，则要求GL的版本必须是1.2。

- 出错提示:

当参数*ctx*不是一个有效的环境时产生**GLXBadContext**提示。

- 请参阅:

glXCreateNewContext(), **glXGetCurrentContext()**, **glXQueryVersion()**, **glXQueryExtensionsString()**

• **glXQueryExtension**

- 名称：

glXQueryExtension()

- 功能：

表明是否支持GLX扩展。

- C描述：

```
Bool glXQueryExtension( Display *dpy,
                        int *errorBase,
                        int *eventBase)
```

- 参数说明：

dpy 指定到X服务器的连接。

errorBase 返回GLX服务器扩展的基准出错代码。

eventBase 返回GLX服务器扩展的基准事件代码。

- 说明：

如果*dpy*连接的X服务器支持GLX扩展，则例程**glXQueryExtension()**返回**True**；否则，返回**False**。当返回值是**True**时，参数*errorBase*和*eventBase*返回GLX扩展的出错基准和事件基准。这些值加上出错常量值和事件常量值才可以确定实际的出错值和事件值。否则，当返回值是**False**时，参数*errorBase*和*eventBase*不改变。

当参数*errorBase*和*eventBase*被指定为NULL时，它们将不返回值。

- 请参阅：

glXQueryVersion()

• **glXQueryExtensionsString**

- 名称：

glXQueryExtensionsString()

- 功能：

返回一个所支持的扩展清单。

- C描述：

```
const char * glXQueryExtensionsString( Display *dpy,
                                       int screen )
```

- 参数说明：

dpy 指定到X服务器的连接。

screen 指定屏幕的数目。

- 说明：

例程**glXQueryExtensionsString()**返回一个指针，这个指针指向用来描述在连接中支持哪些

GLX扩展的字符串。该字符串是以空结束的，并且包含一个用空格分隔的扩展名清单。(但扩展名本身并不包含空格。)如果没有对于GLX的扩展，则返回空字符串。

- 注意：

例程`glXQueryExtensionsString()`只有在GLX 1.1以上的版本中才有效。

例程`glXQueryExtensionsString()`仅返回有关GLX扩展的信息。调用例程`glGetString()`可以得到一个GL扩展清单。

- 请参阅：

`glGetString()`, `glXQueryVersion()`, `glXQueryServerString()`, `glXGetClientString()`

:glXQueryServerString

- 名称：

`glXQueryServerString()`

- 功能：

返回一个描述服务器状况的字符串。

- C描述：

```
const char * glXQueryServerString( Display *dpy,
                                  int screen,
                                  int name )
```

- 参数说明：

dpy 指定到X服务器的连接。

screen 指定屏幕的数目。

name 指定一个要返回的字符串。它可以是**GLX_VENDOR**、**GLX_VERSION**或**GLX_EXTENSIONS**。

- 说明：

函数`glXQueryServerString()`返回一个指针，这个指针指向用来描述服务器的某些GLX扩展情况的一个静态的以空结束的字符串。参数*name*的可能值以及字符串的格式与例程`glXGetClientString()`相同。如果*name*没有被设置成一个认可的值，则返回**NULL**。

- 注意：

例程`glXQueryServerString()`只有在GLX 1.1以上的版本中才有用。

如果GLX的版本是1.1或1.0，则要求GL的版本必须是1.0。如果GLX的版本是1.2，则要求GL的版本必须是1.1。如果GLX的版本是1.3，则要求GL的版本必须是1.2。

例程`glXQueryServerString()`仅返回由服务器端所支持的有关GLX扩展的信息。调用函数`glGetString()`可以得到一个GL扩展清单。调用例程`glXGetClientString()`可以得到一个客户端支持的GLX扩展清单。

- 请参阅：

`glXQueryVersion()`, `glXGetClientString()`, `glXQueryExtensionsString()`

:glXQueryVersion

- 名称：

glXQueryVersion()

- 功能：

返回GLX扩展的版本号。

- C描述：

```
Bool glXQueryVersion( Display *dpy,
                      int *major,
                      int *minor )
```

- 参数说明：

dpy 指定到X服务器的连接。

major 返回GLX服务器扩展的主版本号。

minor 返回GLX服务器扩展的次版本号。

- 说明：

例程glXQueryVersion()返回与连接*dpy*相关联的服务器所实现的GLX扩展的主版本号和次版本号。主版本号相同的实现是向上兼容的，也就是说拥有高的次版本号的实现是拥有低的次版本号的实现的父集。

当参数*major*和*minor*被指定为NULL时，它们将不返回数值。

- 出错提示：

如果例程glXQueryVersion()执行失败，它将返回False；否则，返回True。

当返回值是False时，参数*major*和*minor*将不被更新。

- 请参阅：

glXQueryExtension()

:glXSelectEvent

- 名称：

glXSelectEvent()

- 功能：

为一个窗口或一个GLX像素缓冲区选择GLX事件。

- C描述：

```
void glXSelectEvent( Display *dpy,
                     GLXDrawable draw,
                     unsigned long event_mask )
```

- 参数说明：

dpy 指定到X服务器的连接。

draw 指定一个GLX可绘区域。它必须是一个GLX像素缓冲区或一个窗口。

event_mask

指定返回给*draw*的事件。

- 说明：

例程glXSelectEvent()将为一个GLX像素缓冲区或一个窗口设置GLX事件屏蔽。调用例程glXSelectEvent()将覆盖原来由客户为*draw*设置的事件屏蔽。值得注意的是它并不影响其他客户端为*draw*所选定的事件屏蔽，这是因为每个绘入*draw*的客户端都含有各自独立的*draw*的事件屏蔽。

当然，这里只能选取一个GLX事件，即**GLX_PBUFFER_CLOBBER_MASK**。当一个**GLX_PBUFFER_CLOBBER_MASK**事件发生时，客户端将返回如下数据：

```
typdef struct { l1.
    int event_type;      /* GLX_DAMAGED or GLX_SAVED */
    int draw_type;       /* GLX_WINDOW or GLX_PBUFFER */
    unsigned long serial; /* # of last request processed by server */
    Bool send_event;    /* true if this came for SendEvent request */
    Display *display;   /* display the event was read from */
    GLXDrawable drawable; /* i.d. of Drawable */
    unsigned int buffer_mask; /* mask indicating affectedbuffers */
    int x, y;
    int width, height;
    int count;           /* if nonzero, at least this many more */
} GLXPbufferClobberEvent;
```

在*buffer_mask*中所使用的有效的位屏蔽有：

center; lb lb l1. _ Bitmask	Corresponding Buffer _
GLX_FRONT_LEFT_BUFFER_BIT	Front left color buffer
GLX_FRONT_RIGHT_BUFFER_BIT	Front right color buffer
GLX_BACK_LEFT_BUFFER_BIT	Back left color buffer
GLX_BACK_RIGHT_BUFFER_BIT	Back right color buffer
GLX_AUX_BUFFERS_BIT	Auxillary buffer
GLX_DEPTH_BUFFER_BIT	Depth buffer
GLX_STENCIL_BUFFER_BIT	Stencil buffer
GLX_ACCUM_BUFFER_BIT	Accumulation buffer

每一个单独的X服务器操作都可能导致几个缓冲区重创事件发生。（例如，当一个GLX像素缓冲区被破坏时，它将导致多个缓冲区重创事件发生）。每个事件都将指定一个受X服务器操作所影响的GLX可绘区域。字段*buffer_mask*表示哪些颜色缓冲区和辅助缓冲区被影响。所有由一个X服务器动作所引发的缓冲区重创事件将被保证在一个事件队列中是连续的。该重创事件发生的条件及*event_type*变量由GLX可绘区域的类型决定。

当*buffer_mask*中设置了**GLX_AUX_BUFFERS_BIT**时，则*aux_buffer*将被设置成表示哪个缓冲区将被影响。如果有不止一个辅助缓冲区受到了影响，则附加的事件将作为同一个连续事件组的一部分发生。每一个附加事件的都只在*buffer_mask*中设置**GLX_AUX_BUFFERS_BIT**，并将字段*aux_buffer*设置成适当的值。对于非立体的可绘区域而言，**GLX_FRONT_LEFT_BUFFER_BIT**和**GLX_BACK_LEFT_BUFFER_BIT**分别用来指定前后颜色缓冲区。

对于需要保存的GLX像素缓冲区，只要GLX像素缓冲区的内容被移出了屏幕外的存储区，一个类型为**GLX_SAVED**的缓冲区重创事件就会发生。这个（些）事件描述了GLX像素缓冲区的哪些部分受到了影响。接收各种缓冲区重创事件的客户端将根据不同的事件而作出不同的存储动作。客户

端还需要考虑释放GLX像素缓冲区的资源，以便保护系统免受资源不足而造成的损害。

对于不需要保护的GLXPbuffer，只要GLX像素缓冲区的一部分变得无效，一个类型为**GLX_DAMAGED**的缓冲区重创事件就会发生。这时客户可能希望重新生成已失效的像素缓冲区部分。

对窗口来说，只要一个与它相关的辅助缓冲区遭到重创或被移出了屏幕外的存储区，一个类型为**GLX_SAVED**的缓冲区重创事件就会发生。该事件包含的信息表示哪些颜色缓冲区和辅助缓冲区及这些缓冲区的哪些部分受到了影响。

- 注意：

例程glXSelectEvent()只有在GLX 1.3以上的版本中才有效。

如果GLX的版本是1.1或1.0，则要求GL的版本必须是1.0。如果GLX的版本是1.2，则要求GL的版本必须是1.1。如果GLX的版本是1.3，则要求GL的版本必须是1.2。

- 出错提示：

如果draw不是一个有效的窗口或一个有效的GLX像素缓冲区，则产生**GLXBadDrawable**提示。

- 相关数据的获取：

glXGetSelectedEvent()

- 请参阅：

glXCreatePbuffer()

glXSwapBuffers

- 名称：

glXSwapBuffers()

- 功能：

交换前后缓冲区。

- C描述：

```
void glXSwapBuffers( Display *dpy,
                     GLXDrawable drawable )
```

- 参数说明：

dpy 指定到X服务器的连接。

drawable 指定将被交换缓冲区的可绘区域。

- 说明：

例程glXSwapBuffers()将用*drawable*的后缓冲区中的内容替换其前缓冲区中的内容。这时后缓冲区中的内容将未定义。这一更新动作通常在监视器纵向折回时才发生，而不是调用例程glXSwapBuffers()后就立刻发生。

例程glXSwapBuffers()返回前首先执行了一个隐含的glFlush()例程。调用例程glXSwapBuffers()后可能会立刻发布OpenGL命令，但它们要等到前后缓冲区交换完成后才会被执行。

如果建立的*drawable*不是一个双缓冲的视觉环境，则例程glXSwapBuffers()将无效，但这时

并不产生错误。

- 注意：

交换操作执行之后后缓冲区中的内容将变成未定义的。值得注意的是该命令同样适用于像素缓冲区和窗口。

所有的GLX绘图环境的前缓冲区和后缓冲区的概念是相同的。因此，当有多个客户绘入同一个双缓冲区窗口时，在其中一个客户发布交换缓冲区的命令之前所有的绘制都必须已经完成。客户负责这一同步的实现。这一功能的典型实现方式是首先执行glFinish()，然后在交换前通过共享内存中的一个信号量将它们汇聚在一起。

- 出错提示：

如果`drawable`不是一个有效的GLX可绘区域，则产生**GLXBadDrawable**提示。

如果`dpy`和`drawable`分别是与正在调用的线程的当前环境相关联的显示器和可绘区域，而`drawable`等同于一个无效窗口，则产生**GLXBadCurrentWindow**提示。

- 请参阅：

`glFlush()`

• `glXUseXFont`

- 名称：

`'glXUseXFont()`

- 功能：

从一个X字体建立位图显示列表。

- C描述：

```
void glXUseXFont( Font font,
```

```
                      int first,
                      int count,
                      int listBase )
```

- 参数说明：

`font` 从将要使用的字母图示符中指定字体。

`first` 指定将要使用的第一位图示符的索引。

`count` 指定将要使用的图示符的数目。

`listBase` 指定将要生成的第一个显示列表的索引。

- 说明：

例程`glXUseXFont()`用来生成从`listBase`到`listBase + count - 1`共`count`个显示列表，每个显示列表都包含一个单独的`glBitmap()`命令。显示列表`listBase+i`的`glBitmap()`命令的参数是从图示符`first+i`中获取。位图参数`xorig`、`yorig`、`width`和`height`分别从字体尺寸`descent-1`、`-lbearing`、`rbearing-lbearing`和`ascent+descent`中计算得到。参数`xmove`由图示符的`width`尺寸得到，参数`ymove`被设置成0。然后，图示符的图形被转换成`glBitmap()`的适当格式。

使用例程`glXUseXFont()`将比直接访问X字体和生成一个显示列表效率更高，这是因为它所

建立的服务器端的显示列表不需要循环访问图示符数据，还因为服务器可以选择延迟每个位图的建立，即直到某个位图将被访问时服务器才建立它。

对于所有要求被建立且未在*font*中被定义的图示符，都将被建立一个空的显示列表。如果这里没有当前的GLX环境，则例程glXUseXFont()将被忽略。

- 出错提示：

如果*font*不是一个有效字体，则产生**BadFont**提示。

如果当前的GLX环境处于显示列表的结构模式中，则产生**GLXBadContextState**提示。

如果与正在执行的线程中的当前环境相关的可绘区域是一个无效的窗口，则产生**GLX-BadCurrentWindow**提示。

- 请参阅：

glBitmap(), **glXMakeCurrent()**

◆ glXWaitGL

- 名称：

glXWaitGL()

- 功能：

首先完成GL执行，然后再调用后面的X命令。

- C描述：

```
void glXWaitGL( void )
```

- 说明：

确保在例程glXWaitGL()之前已调用的GL绘制命令将比它后面所调用的X绘图命令先执行。尽管用glFinish()也能得到相同的结果，但例程glXWaitGL()不需要循环访问服务器，因此它对于客户端和服务器端不是同一台机器的情况将更有效。

如果没有当前GLX环境，例程glXWaitGL()将被忽略。

- 注意：

例程glXWaitGL()有时会，但有时又不会冲掉X流。

- 出错提示：

如果与正在调用的线程的当前环境相关联的可绘区域是一个无效的窗口，则产生**GLX-BadCurrentWindow**提示。

- 请参阅：

glFinish(), **glFlush()**, **glXWaitX()**, **XSync()**

◆ glXWaitX

- 名称：

glXWaitX()

- 功能：

首先完成X执行，然后再调用后面的GL命令。

- C描述：

```
void glXWaitX( void )
```

- 说明：

确保例程glXWaitX()之前已调用的X绘制命令将比它后面所调用的GL绘图命令先执行。尽管用XSync()也能得到相同的结果，但例程glXWaitX()不需要循环访问服务器，因此它对于客户端和服务器端不是同一台机器的情况将更有效。

如果没有当前GLX环境，例程glXWaitX()将被忽略。

- 注意：

例程glXWaitX()有时会，但有时又不会冲掉GL流。

- 出错提示：

如果与正在执行的线程的当前环境相关联的可绘区域是一个无效的窗口，则产生**GLX-BadCurrentWindow**提示。

- 请参阅：

glFinish(), **glFlush()**, **glXWaitGL()**, **XSync()**

■ glXChooseVisual

- 名称：

glXChooseVisual()

- 功能：

返回一个与指定属性相匹配的视觉环境。

- C描述：

```
XVisualInfo * glXChooseVisual( Display *dpy,
                               int screen,
                               int *attribList )
```

- 参数说明：

dpy 指定到X服务器的连接。

screen 指定屏幕的数目。

attribList

指定一个boolean属性和整型属性/值对的列表。最后一个属性必须是**None**。

- 说明：

例程glXChooseVisual()将返回一个指向XVisualInfo结构的指针，该结构用来描述与最小指定值相匹配的最佳视觉环境。返回视觉环境的boolean型的GLX属性将与指定的值相匹配，整型GLX属性将等于或大于所指定的最小值。如果其他的属性是相同的，则TrueColor和PseudoColor视觉环境将分别优先于DirectColor和StaticColor视觉环境。如果没有合适的视觉环境存在，则返回NULL。请用XFree()来释放由该函数返回的数据。

除**GLX_USE_GL**的缺省值是**True**外，其他的boolean型GLX属性的缺省值都是**False**。所有整型GLX属性的缺省值都是0。缺省值将由*attribList*中的值所替代。包含于*attribList*中的boolean属性被认为是**True**。整型和枚举型的属性后都紧跟着相应的期望值或最小值。列表必须用**None**

作为结束。

各种GLX视觉环境属性的说明如下：

GLX_USE_GL 忽略。只有用GLX来绘制视觉环境时才考虑它。

GLX_BUFFER_SIZE

后面必须跟一个非负的整数，它表示所期望的颜色索引缓冲区的尺寸。它将选出预先指定尺寸的索引缓冲区中最小的一个。当处于**GLX_RGBA**时忽略该属性。

GLX_LEVEL

后面必须跟一个整数的缓冲区级别的规格。这个规格是非常重要的。0级缓冲区表示显示器的主帧缓冲区。1级缓冲区表示第一个覆盖帧缓冲区，2级缓冲区表示第二个覆盖帧缓冲区，…依此类推。负数级别的缓冲区表示基础帧缓冲区。

GLX_RGBA

当提供该属性时，它只考虑TrueColor和DirectColor视觉环境。否则，只考虑PseudoColor和StaticColor视觉环境。

GLX_DOUBLEBUFFER

当提供该属性时，它只考虑双缓冲区的视觉环境。否则，只考虑单缓冲区的视觉环境。

GLX_STEREO

当提供该属性时，它只考虑立体缓冲区的视觉环境。否则，只考虑单原子缓冲区的视觉环境。

GLX_AUX_BUFFERS

后面必须跟一个非负的整数，它表示所期望的辅助缓冲区的个数。它将选出含有辅助缓冲区的最小数目等于或大于指定数目的视觉环境。

GLX_RED_SIZE

后面必须跟一个非负的最小尺寸规格。如果这个值是0，它将选出最小的有效红组分缓冲区。否则，它将选出最小尺寸的有效红组分缓冲区中最大的一个红组分缓冲区。

GLX_GREEN_SIZE

后面必须跟一个非负的最小尺寸规格。如果这个值是0，它将选出最小的有效绿组分缓冲区。否则，它将选出最小尺寸的有效绿组分缓冲区中最大的一个绿组分缓冲区。

GLX_BLUE_SIZE

后面必须跟一个非负的最小尺寸规格。如果这个值是0，它将选出最小的有效蓝组分缓冲区。否则，它将选出最小尺寸的有效蓝组分缓冲区中最大的一个蓝组分缓冲区。

GLX_ALPHA_SIZE

后面必须跟一个非负的最小尺寸规格。如果这个值是0，它将选出最小的有效alpha组分缓冲区。否则，它将选出最小尺寸的有效alpha组分缓冲区中最大的一个alpha组分缓冲区。

GLX_DEPTH_SIZE

后面必须跟一个非负的最小尺寸规格。如果这个值是0，它将选出没有

深度缓冲区的视觉环境。否则，它将选出最小尺寸的有效深度缓冲区中最大的一个深度缓冲区。

GLX_STENCIL_SIZE

后面必须跟一个非负的整数，它表示所期望的模板位面的数目。它将选出指定尺寸的模板缓冲区中最小的一个模板缓冲区。如果期望值是0，那么它将选出没有模板缓冲区的视觉环境。

GLX_ACCUM_RED_SIZE

后面必须跟一个非负的最小尺寸规格。如果这个值是0，它将选出没有红组分累积缓冲区的视觉环境。否则，它将选出最小尺寸的红组分累积缓冲区中最大的一个红组分累积缓冲区。

GLX_ACCUM_GREEN_SIZE

后面必须跟一个非负的最小尺寸规格。如果这个值是0，它将选出没有绿组分累积缓冲区的视觉环境。否则，它将选出最小尺寸的绿组分累积缓冲区中最大的一个绿组分累积缓冲区。

GLX_ACCUM_BLUE_SIZE

后面必须跟一个非负的最小尺寸规格。如果这个值是0，它将选出没有蓝组分累积缓冲区的视觉环境。否则，它将选出最小尺寸的蓝组分累积缓冲区中最大的一个蓝组分累积缓冲区。

GLX_ACCUM_ALPHA_SIZE

后面必须跟一个非负的最小尺寸规格。如果这个值是0，它将选出没有alpha组分累积缓冲区的视觉环境。否则，它将选出最小尺寸的alpha组分累积缓冲区中最大的一个alpha组分累积缓冲区。

- **示例：**

```
attribList={GLX_RGBA, GLX_RED_SIZE, 4, GLX_GREEN_SIZE, 4,
           GLX_BLUE_SIZE, 4, None};
```

从普通帧缓冲区中指定一个单缓冲区的RGB视觉环境，而不是在覆盖或基础缓冲区中指定。返回的视觉环境支持红、绿和蓝组分至少各四位，但可能没有alpha组分位。它不能支持颜色索引模式，也不支持双缓冲区或立体显示。它可以有也可以没有一个或多个辅助颜色缓冲区、一个深度缓冲区、一个模板缓冲区或一个累积缓冲区。

- **注意：**

XVisualInfo在**Xutil.h**中定义。它是一种包含**visual**，**visualID**，**screen**和**depth**元素的结构。

例程**glXChooseVisual()**仅使用**XGetVisualInfo()**和**glXGetConfig()**来实现一种客户端应用。调用这两个例程可以实现算法选取，而不像例程**glXChooseVisual()**所实现的一般功能。

虽然这一做法并不被禁止，但建议用户不要使用例程**glXChooseVisual()**来选择算法。因此，客户端数据库的版本变化时可能会导致选择的变化。

只有选取支持**GLXPixmaps**的视觉环境时，才可使用直接滤波器。当一个视觉环境的**GLX_BUFFER_SIZE**是一个X服务器所支持的像素映射深度时，它将支持**GLXPixmaps**。

- 出错提示：

当`attribList`中出现了未定义的GLX属性时将返回**NULL**。

- 请参阅：

`glXCreateContext()`, `glXGetConfig()`

`glXCreateContext`

- 名称：

`glXCreateContext()`

- 功能：

建立一个新的GLX绘图环境。

- C描述：

```
GLXContext glXCreateContext( Display *dpy
                           XVisualInfo *vis,
                           GLXContext shareList,
                           Bool direct )
```

- 参数说明：

`dpy` 指定到X服务器的连接。

`vis` 指定一个视觉环境，它将用来定义绘图环境可用到的帧缓冲区资源。它是一个指向`XVisualInfo`结构的指针，而不是一个视觉环境ID或指向`Visual`的指针。

`shareList` 指定用来共享显示列表的环境。**NULL**表示不共享。

`direct` 指定绘制操作是通过直接连接的图形系统(**True**)还是通过X服务器(**False**)。

- 说明：

例程`glXCreateContext()`建立一个GLX绘图环境并返回它的处理结果。这个环境可以用来绘入窗口和GLX像素映射中。如果该例程建立绘制环境失败，则返回**NULL**。

当`direct`是**True**时，如果具体实现支持直接绘图，到X服务器的连接是局部的，并且一个直接绘图环境是允许的，则建立一个直接绘图环境。(当`direct`是**True**时，一个实现也可以返回一个间接环境。)如果`direct`是**False**，则总是建立一个通过X服务器绘制的绘图环境。直接绘图在某些设备上是优先支持的。然而，在一个进程之外是不能共享直接绘图环境的，而且它们可能无法向GLX像素映射绘制。

如果`shareList`不是**NULL**，则所有显示列表索引和定义将被环境`shareList`和新建立的环境共享。一个单独的显示列表空间可以被任意数目的环境所共享。然而，所有共享一个显示列表空间的绘图环境都必须存在于相同的地址空间中。如果两个绘图环境都间接地使用相同的服务器或者它们被一个进程所直接拥有，则它们可以共享一个地址空间。值得注意的是在间接的情况下正在调用的线程不必共享一个地址空间，这时只要它们所关联的绘图环境共享一个地址空间就可以了。

在GL 1.1以上的版本中，除对象0之外的所有其他纹理对象将被共享显示列表的任何环境所共享。

- 注意：

XVisualInfo在**Xutil.h**中定义。它是一种包含*visual*、*visualID*、*screen*和*depth*元素的结构。

一个“进程”就是一个单独的运行环境，它将在一个单一的地址空间中执行，并且包含一个或多个线程。

一个“线程”就是一组共享一个单一地址空间的子进程中的一子进程，但它含有单独的程序计数器、堆栈空间以及其他有关的全局数据。一个“线程”如果是它的子进程组的唯一的成员，则该“线程”相当于一个“进程”。

有时可能无法向一个带直接绘图环境的GLX像素映射绘制。

- 出错提示：

如果客户端执行失败，则产生**NULL**提示。

如果将要建立的环境不共享一个地址空间或者由*shareList*指定的环境屏幕，则产生**BadMatch**提示。

如果*vis*不是一个有效的视觉环境（例如，如果一个特定的GLX实现不支持它），则产生**BadValue**提示。

如果*shareList*既不是一个GLX环境又不是**NULL**，则产生**GLXBadContext**提示。

如果服务器没有足够的资源来配置新的环境，则产生**BadAlloc**提示。

- 请参阅：

glXDestroyContext(), **glXGetConfig()**, **glXIsDirect()**, **glXMakeCurrent()**

glXCreateGLXPixmap

- 名称：

glXCreateGLXPixmap()

- 功能：

建立一个屏幕外的GLX绘图区域。

- C描述：

```
GLXPixmap glXCreateGLXPixmap( Display *dpy
                               XVisualInfo *vis,
                               Pixmap pixmap )
```

- 参数说明：

dpy 指定到X服务器的连接。

vis 指定一个视觉环境，它将用来定义绘图区的结构。它是一个指向**XVisualInfo**结构的指针，而不是一个视觉环境ID或指向**Visual**的指针。

pixmap 指定将用作屏幕外绘图区域的前左颜色缓冲区的X像素映射。

- 说明：

例程**glXCreateGLXPixmap()**建立一个屏幕外的绘图区域并返回它的XID。任何根据*vis*建立的GLX绘图环境都可以用来向这个屏幕外的区域绘制。请使用例程**glXMakeCurrent()**来将该绘图区域跟一个GLX绘图环境相连接。

由*pixmap*指定的X像素映射将被用作最后的屏幕外绘图区域的前左缓冲区。除前左缓冲区之外，其他由*vis*所指定的缓冲区（包括颜色缓冲区）建立后将没有外部视觉环境名称。它将支持带双缓冲的GLX像素映射。然而，例程glXSwapBuffers()将被这些像素映射所忽略。

某些实现不支持带直接绘图环境的GLX像素映射。

- 注意：

XVisualInfo在**Xutil.h**中定义。它是一种包含*visual*, *visualID*, *screen*和*depth*元素的结构。

- 出错提示：

如果*pixmap*的深度值同核心的X11提供给*vis*的深度值不匹配，或者*pixmap*没有由*vis*创建相同屏幕，则产生**BadMatch**提示。

如果*vis*不是一个有效的XVisualInfo指针（例如，如果一个特定的GLX实现不支持该视觉环境），则产生**BadValue**提示。

如果*pixmap*不是一个有效的*pixmap*，则产生**BadPixmap**提示。

如果服务器不能配置GLX像素映射，则产生**BadAlloc**提示。

- 请参阅：

glXCreateContext(), **glXCreatePixmap()**, **glXDestroyGLXPixmap()**, **glXIsDirect()**,
glXMakeCurrent()

- **glXDestroyGLXPixmap**

- 名称：

glXDestroyGLXPixmap()

- 功能：

删除一个GLX像素映射。

- C描述：

```
void glXDestroyGLXPixmap( Display *dpy,
                           GLXPixmap pix )
```

- 参数说明：

dpy 指定到X服务器的连接。

pix 指定要删除的GLX像素映射。

- 说明：

如果对任何客户而言，GLX像素映射*pix*都不是当前的，则例程glXDestroyGLXPixmap()将立刻删除它。否则，要等到*pix*对于任何客户都不是当前的时候才删除它。上面的两种情况都将立刻释放资源的ID。

- 出错提示：

如果*pix*不是一个有效的GLX像素映射，则产生**GLXBadPixmap**提示。

- 请参阅：

glXCreateGLXPixmap(), **glXDestroyPixmap()**, **glXMakeCurrent()**

- **glXGetConfig**

- 名称：

glXGetConfig()

- 功能：

返回GLX视觉环境的信息。

- C描述：

```
int glXGetConfig( Display *dpy,
                   XVisualInfo *vis,
                   int attrib,
                   int *value )
```

- 参数说明：

dpy 指定到X服务器的连接。

vis 指定所要查询的视觉环境。它是一个指向XVisualInfo结构的指针，而不是一个视觉环境ID或指向Visual的指针。

attrib 指定要返回的视觉环境属性。

value 返回所需的值。

- 说明：

例程glXGetConfig()将*value*设置为根据*vis*所建立的窗口或GLX像素映射的*attrib*值。当由于某种原因导致操作失败时，它将返回一个出错代码；否则，它返回0。

参数*attrib*是下列各值之一：

GLX_USE_GL	如果该视觉环境支持OpenGL绘图，则返回 True ；否则，返回 False 。
GLX_BUFFER_SIZE	它表示每个颜色缓冲区二进制位的数目。对于RGBA的视觉环境， GLX_BUFFER_SIZE 是 GLX_RED_SIZE 、 GLX_GREEN_SIZE 、 GLX_BLUE_SIZE 和 GLX_ALPHA_SIZE 之和。对于颜色索引的视觉环境， GLX_BUFFER_SIZE 表示颜色索引的尺寸。
GLX_LEVEL	它表示视觉环境的帧缓冲区的层数。0层代表默认的帧缓冲区。正的层数表示覆盖于默认缓冲区上面的帧缓冲区，负的层数表示衬垫于默认缓冲区下面的帧缓冲区。
GLX_RGBA	如果颜色缓冲区中存放的是红、绿、蓝和alpha值，它返回 True ；如果存放的是颜色索引值，它返回 False 。
GLX_DOUBLEBUFFER	如果颜色缓冲区中存在可交换的前/后缓冲区对，它返回 True ；否则，返回 False 。
GLX_STEREO	如果颜色缓冲区中存在左/右缓冲区对，它返回 True ；否则，返回 False 。
GLX_AUX_BUFFERS	

	它代表有效的辅助颜色缓冲区的个数。0表示没有辅助颜色缓冲区存在。
GLX_RED_SIZE	它代表存储于每个颜色缓冲区的红组分中二进制位的数目。如果 GLX_RGBA 是 False , 它将是未定义的。
GLX_GREEN_SIZE	它代表存储于每个颜色缓冲区的绿组分中二进制位的数目。如果 GLX_RGBA 是 False , 它将是未定义的。
GLX_BLUE_SIZE	它代表存储于每个颜色缓冲区的蓝组分中二进制位的数目。如果 GLX_RGBA 是 False , 它将是未定义的。
GLX_ALPHA_SIZE	它代表存储于每个颜色缓冲区的alpha组分中二进制位的数目。如果 GLX_RGBA 是 False , 它将是未定义的。
GLX_DEPTH_SIZE	它代表深度缓冲区中二进制位的数目。
GLX_STENCIL_SIZE	它代表模板缓冲区中二进制位的数目。
GLX_ACCUM_RED_SIZE	它代表存储于累积缓冲区的红组分中二进制位的数目。
GLX_ACCUM_GREEN_SIZE	它代表存储于累积缓冲区的绿组分中二进制位的数目。
GLX_ACCUM_BLUE_SIZE	它代表存储于累积缓冲区的蓝组分中二进制位的数目。
GLX_ACCUM_ALPHA_SIZE	它代表存储于累积缓冲区的alpha组分中二进制位的数目。

X协议允许每个像素用不同数目的二进制位来说明一个单独的视觉环境的ID。窗口或GLX像素映射将用OpenGL进行绘图。然而，它们必须用**GLX_BUFFER_SIZE**的一个颜色缓冲区深度来说明。

尽管一个GLX实现能输出许多支持GL绘图的视觉环境，但它必须至少支持一个**RGBA**视觉环境。这个视觉环境必须至少有一个颜色缓冲区、一个至少1位的模板缓冲区、一个至少12位的深度缓冲区和一个累积缓冲区。在这个视觉环境中，alpha的位面是任意的。但是它的颜色缓冲区的尺寸至少应该等于0层中所支持的最深的**TrueColor**、**DirectColor**、**PseudoColor**或**StaticColor**视觉环境，并且在0层中它必须使自身有效。

另外，如果X服务器在帧缓冲区0层中输出一个**PseudoColor**或**StaticColor**视觉环境，则在该层中同样也需要一个颜色索引视觉环境。这个颜色索引视觉环境至少应该含有一个颜色缓冲区、一个至少1位的模板缓冲区和一个至少12位的深度缓冲区。这个视觉环境必须同0层中所支持的最深**PseudoColor**或**StaticColor**视觉环境具有同样多的颜色位面。

应用程序中最好选用最接近它们需求的视觉环境。用不必要的缓冲区建立窗口或GLX像素映射将导致绘图速度的降低，就象资源配置不足一样。

- 注意：

XVisualInfo在**Xutil.h**中定义。它是一种包含*visual*、*visualID*、*screen*和*depth*元素的结构。

- 出错提示：

如果*dp*y不支持GLX扩展，则产生**GLX_NO_EXTENSION**提示。

如果*vis*的屏幕没有同一个屏幕相关联，则产生**GLX_BAD_SCREEN**提示。

如果*attrib*不是一个有效的GLX属性，则产生**GLX_BAD_ATTRIBUTE**提示。

如果*vis*不支持GLX和除**GLX_USE_GL**之外的其他所需的属性，则产生**GLX_BAD_VISUAL**提示。

- 请参阅：

glXChooseVisual(), **glXCreateContext()**

[G e n e r a l I n f o r m a t i o n]

书名 = OpenGL 参考手册

作者 =

页数 = 481

S S号 = 10331451

出版日期 =

[封面页](#)
[书名页](#)
[版权页](#)
[前言页](#)
[目录页](#)

第1章 OpenGL简介

- 1.1 OpenGL基础**
 - 1.1.1 OpenGL图元及命令
 - 1.1.2 OpenGL是一种过程语言
 - 1.1.3 OpenGL的执行模式

1.2 基本OpenGL操作

第2章 命令和例程概述

- 2.1 OpenGL处理流程**
 - 2.1.1 顶点
 - 2.1.2 ARB绘图子集
 - 2.1.3 片断
- 2.2 其他OpenGL命令**
 - 2.2.1 使用求值器
 - 2.2.2 执行选择和反馈
 - 2.2.3 显示列表的使用
 - 2.2.4 模式和运行的管理
 - 2.2.5 获取状态信息
- 2.3 OpenGL实用库**
 - 2.3.1 生成纹理操作所需的图形
 - 2.3.2 坐标转换
 - 2.3.3 多边形的镶嵌分块
 - 2.3.4 绘制球体、圆柱和圆盘
 - 2.3.5 NURBS曲线和曲面
 - 2.3.6 错误处理
- 2.4 对X窗口系统的OpenGL扩展**
 - 2.4.1 初始化
 - 2.4.2 控制绘制操作

第3章 命令和例程一览

- 3.1 注释**
- 3.2 OpenGL命令**
 - 3.2.1 图元
 - 3.2.2 顶点数组
 - 3.2.3 坐标转换
 - 3.2.4 着色与光照
 - 3.2.5 剪切
 - 3.2.6 光栅化
 - 3.2.7 像素操作
 - 3.2.8 纹理
 - 3.2.9 雾
 - 3.2.10 帧缓冲区操作
 - 3.2.11 求值器
 - 3.2.12 选择与反馈
 - 3.2.13 显示列表
 - 3.2.14 模式与执行
 - 3.2.15 状态查询
- 3.3 ARB扩展**
 - 3.3.1 多重纹理
 - 3.3.2 绘图子集
- 3.4 GLU例程**
 - 3.4.1 纹理图像
 - 3.4.2 坐标转换
 - 3.4.3 多边形镶嵌分块
 - 3.4.4 二次对象
 - 3.4.5 NURBS曲线和曲面
 - 3.4.6 状态查询
- 3.5 GLX例程**

3 . 5 . 1 初始话

3 . 5 . 2 控制绘图操作

第4章 定义的常量及相关命令

第5章 O p e n G L 参考说明

第6章 G L U 参考说明

第7章 G L X 参考说明

附录页