

# OpenGL

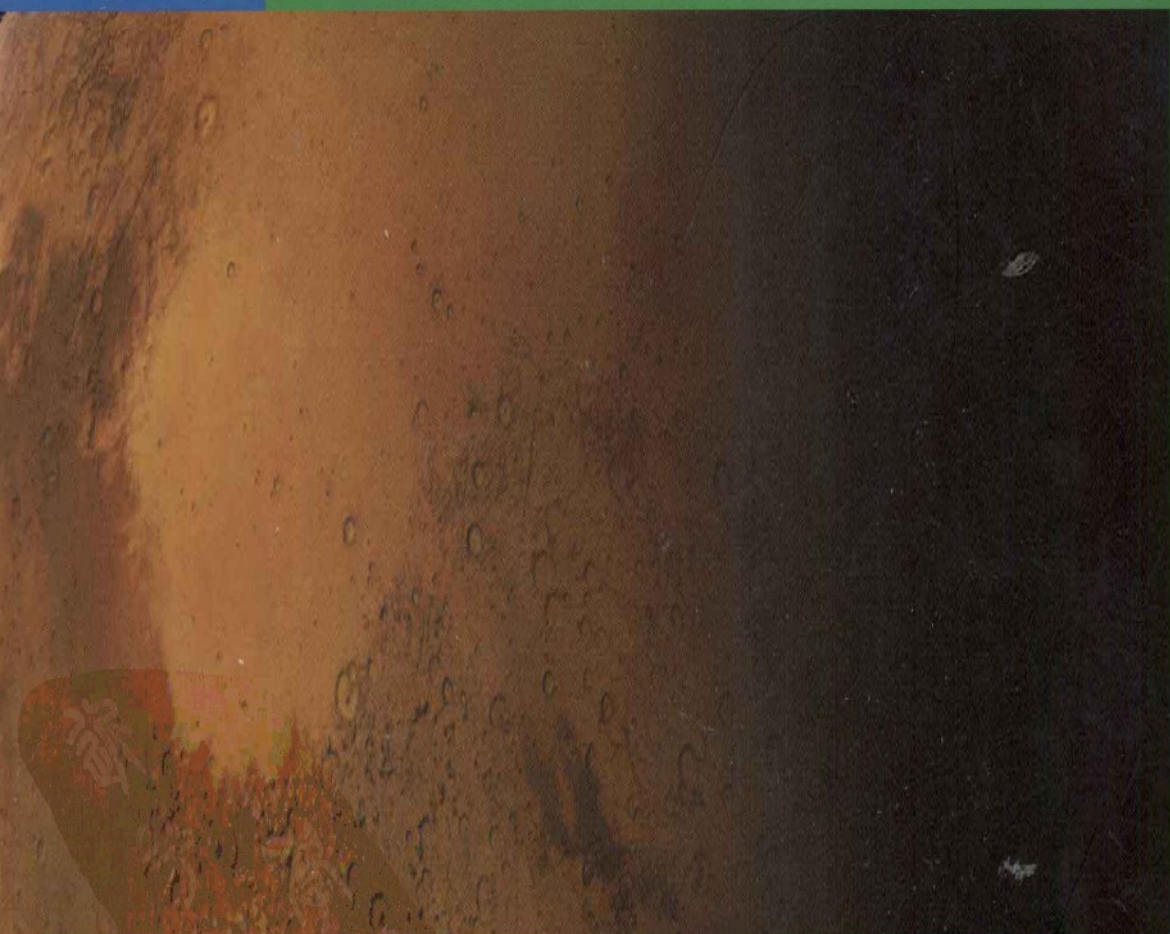
丰富 权威 实用

## 超级宝典 (第5版)

OpenGL<sup>®</sup> SuperBible Fifth Edition

[美] Richard S. Wright, Jr. Nicholas Haemel 著  
Graham Sellers Benjamin Lipchak

付飞 李艳辉 译



# OpenGL 超级宝典 (第5版)

对于使用实时计算机图形领域领先的 3D API——OpenGL 3.3 的程序员来说,本书无疑是权威的指导书、教程和参考资料。它从各个实践层面出发,向开发者详尽、全面地介绍了 OpenGL,对这种 API 及其相关的重要程序设计概念进行了清晰的阐述。读者能够在本书中获得对现代 OpenGL 开发,包括变换、纹理贴图、着色器、高级缓冲区、几何图形管理等内容的最新、最详尽的指导。本书第 5 版不仅针对最新的官方规范(3.3)进行了全面修订,还增加了 iPhone、iPod touch 和 iPad 环境下 OpenGL 应用的完整教程。主要包括:

- 以实践的形式介绍实时 3D 图形重点内容;
- OpenGL 3.3 中的渲染、变换和纹理贴图核心技术;
- 通过实例来引导读者编写自己的着色器;
- 多平台的 OpenGL: Windows (包括 Windows 7)、Mac OS X、GNU/Linux、UNIX 和嵌入式系统;
- iPhone、iPod touch 和 iPad 环境下的 OpenGL 程序设计:一步步地引导读者,并提供完整的示例程序。
- 高级缓冲区技术,包括使用浮点缓冲区和纹理进行全清晰度渲染;
- 片段操作:控制图形管线的终点;
- 高级着色器应用和几何图形管理;
- 基于官方 ARB (核心) OpenGL 3.3 手册页全面更新的 API 参考页;
- 本书原版配套网站 [www.starstonesoftware.com/OpenGL](http://www.starstonesoftware.com/OpenGL) 上提供了新的素材和示例代码。

## 作者简介:

**Richard S. Wright, Jr.** 是 Software Bisque 公司的资深软件工程师,在这家公司使用 OpenGL 开发多媒体宇航和天文软件。他曾经是 OpenGL ARB 在实时 3D 领域的代表人物,编写了大量基于 OpenGL 的游戏、科学与医学应用程序、数据库可视化工具和教育软件。

**Nicholas Haemel** 在 ATI 和 AMD 的 8 年中引导了 3D 图形硬件和软件结构设计,并对 OpenGL 的 3.0、3.1、3.2 和 3.3 标准作出了贡献。

**Graham Sellers** 是 AMD 的 OpenGL 小组的一位管理者,领导着一个 OpenGL 软件开发团队,致力于开发 AMD 的 OpenGL 驱动程序。他是 ARB 中的 AMD 代表,并对核心 OpenGL 3.2、3.3 和 4.0 规范作出了贡献。

**Benjamin Lipchak** 是苹果公司的软件工程主管,领导着一个致力于图形开发技术和标准测试程序的团队,并负责 iPhone 和 iPod touch 的 OpenGL ES 一致性测试。他曾经在 AMD 管理一个 OpenGL ES 驱动程序小组,并领导着 Khronos 的 OpenGL 生态系统小组,在那里他创建了 OpenGL SDK 和 OpenGL Pipeline 等刊物。

OpenGL 技术库的组成部分——OpenGL 开发者的官方知识资源。

OpenGL 技术库 (OpenGL Technical Library) 提供了 OpenGL 教程和参考书。程序员可以利用这个技术库通过实践来理解 OpenGL,并了解如何充分发掘它的潜力。这个库最初是由 SGI 开发的,在 OpenGL 体系结构审核委员会 (ARB, 一个负责引导 OpenGL 及其相关技术演变的行业机构,现已成为 Khronos 小组的一部分) 的管理下一直持续发展。

封面设计:胡萍丽

分类建议:计算机/程序开发/OpenGL

人民邮电出版社网址: [www.ptpress.com.cn](http://www.ptpress.com.cn)



ISBN 978-7-115-27456-4



ISBN 978-7-115-27456-4

定价: 108.00 元

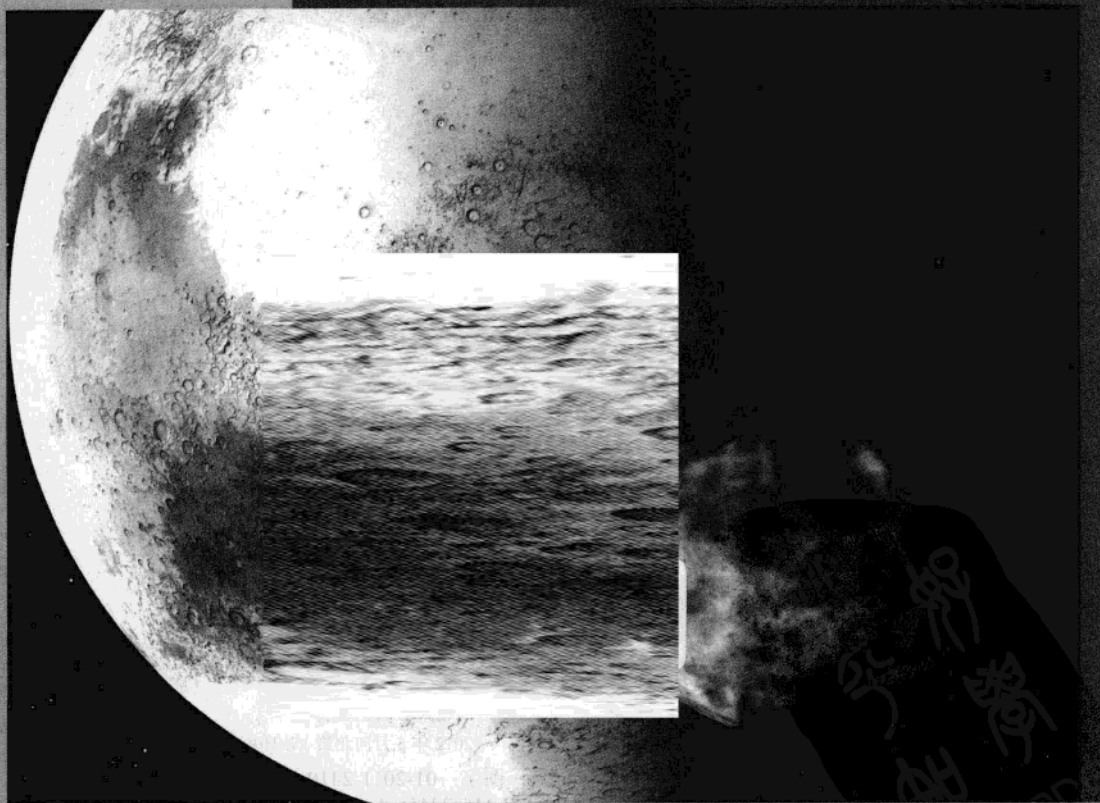
# OpenGL

## 超级宝典 (第5版)

OpenGL<sup>®</sup> SuperBible Fifth Edition

[美] Richard S. Wright, Jr. Nicholas Haemel 著  
Graham Sellers Benjamin Lipchak

付飞 李艳辉 译



人民邮电出版社

北京

## 图书在版编目 (C I P) 数据

OpenGL超级宝典 : 第5版 / (美) 赖特  
(Wright, R. S.) 等著, 付飞, 李艳辉译. — 北京 : 人  
民邮电出版社, 2012. 5  
ISBN 978-7-115-27456-4

I. ①O… II. ①赖… ②付… ③李… III. ①图形软  
件, OpenGL IV. ①TP391.41

中国版本图书馆CIP数据核字 (2012) 第056905号

## 版 权 声 明

Authorized translation from the English language edition, entitled OpenGL SuperBible Fourth Edition, 0321498828 by Richard S. Wright, Jr., published by Pearson Education, Inc, publishing as Addison Wesley Professional, Copyright © 2007 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD., and POSTS & TELECOMMUNICATIONS PRESS Copyright © 2007.

本书封面贴有 Pearson Education (培生教育出版集团) 激光防伪标签。无标签者不得销售。

## OpenGL 超级宝典 (第 5 版)

◆ 著 [美] Richard S. Wright, Jr. Nicholas Haemel  
Graham Sellers Benjamin Lipchak

译 付 飞 李艳辉

责任编辑 俞 彬

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号  
邮编 100061 电子邮件 315@ptpress.com.cn  
网址 <http://www.ptpress.com.cn>  
三河市海波印务有限公司印刷

◆ 开本: 787×1092 1/16

印张: 44 75

彩插: 4

字数: 1 259 千字

2012 年 5 月第 1 版

印数: 1~3 500 册

2012 年 5 月河北第 1 次印刷

著作权合同登记号 图字: 01-2011-2419 号

ISBN 978-7-115-27456-4

定价: 108.00 元

读者服务热线: (010)67132705 印装质量热线: (010)67129223

反盗版热线: (010)67171154

# 内容提要

本书是 OpenGL 及 3D 图形编程最好的入门指南，涵盖了使用最新版本的 OpenGL 进行编程所需要的主要知识。

全书分三部分，共 16 章，另有 3 个附录。第一部分包括第 1 章到第 7 章，介绍如何构建一个使用 OpenGL 的程序、如何设置 3D 渲染环境，以及如何创建基本对象和光线并对他们进行着色。然后深入研究如何使用 OpenGL，并向读者介绍 GLSL，以及如何创建自己的着色器。第二部分包括第 8 章到第 12 章，将进行更深入的研究，而懂得如何应用这些高级特性将使读者超越业余 3D 玩家的水平。这一部分不仅能够使我们掌握更多的可视化效果，同时也考虑了性能表现。第三部分包括第 13 章到第 16 章，着重介绍 OpenGL 如何支持和连接 Windows、Mac OS X、Linux 和掌上设备。附录部分给出了更多阅读建议、术语表和 API 参考介绍。

本书适合希望精通 OpenGL 以便对图形编程和 3D 图形知识进行扩展的程序员阅读，也可以帮助经验丰富的 OpenGL 程序员学习如何移植自己的应用程序。本书既可以作为学习 OpenGL 的教材，也可以作为随时查阅的参考手册。



# 序

在自然界中,有时候一片森林会长得过于茂密,以至于它的生态系统不堪承受自身的负担而濒临崩溃,这片森林可能会被闪电击中,这样往往会出现一个崭新的开始,并在以前的基础上展开一个新的蓝图。本书的第5版就经历了一次类似的根本性转变,我们已经彻底抛弃了第4版,只保留了一小部分独立内容,使这本书呈现了新的生机。

对于 OpenGL 来说也是如此,在 OpenGL 3.0 中,首次将“不推荐”(deprecated)这个词引入到规范中,对很多特性和功能都进行了删除标记,并强烈鼓励开发者转向新的技术,预期得到的是一个更加简洁清晰的 OpenGL 版本,可以看成是从开发包中抛弃了固定管线部分,然而,事情并不总是按照预期的情况发展。在上世纪 90 年代后期,OpenGL 陷入了一场现在所谓的“API 战争”(API Wars),这是由微软公司的 Direct 3D API 引起的,但是,开发者最终会选择更加优越的 API,而 OpenGL 拒绝消亡;相反,它变得更加繁荣,并且成为了实时 3D 图形渲染的世界性标准。开发者似乎又一次表明了态度,无论 ARB 中 OpenGL 的捍卫者如何努力,看来固定管线都拒绝消亡。现在,我们可以选择两种 OpenGL,即兼容版本(compatibility profile)和核心版本(core profile)。

出于这点考虑,由于本书的第4版覆盖了现在可以称为“经典 OpenGL 2.1 固定管线版本”的内容,在未来的几年内完全有理由继续受欢迎,固定功能管线拥有大量的传统优势。它的编程非常容易,并且目前能够在现代图形加速卡上进行完全的硬件加速,在未来一段时期内,很多性能要求不高和非图形专业的专家可能会非常推荐这种编程模型,只有时间能证明一切。

同时,我们还面临着在本书的第5版中该怎样做的问题,每隔6个月左右,OpenGL 就会进行一次升级,而在本书编写过程中我们就经历了两次升级,如果试图同时覆盖兼容版本和核心版本的内容,则会产生很多混淆之处,另外,很多更新、更现代的效果只能在着色器上实现。随着时间一年一年地推移,很多人还在使用的图形编程的固定管线模式看来是越来越过时了。因此,考虑到技术的进步,我们认为在第5版中最好专注于介绍核心版本,就我所知,这将是第一本这样做的图书。当我在佛塞大学(或译为全航大学,Full Sail University)讲授 OpenGL 时,我在试图寻找如何讲授不包含固定管线的 OpenGL 时遇到了巨大的挑战。我和所注意到的其他很多人都是将着色器编程作为固定管线的一个扩展来讲授的,我们怎样才能跳过在刚开始的时候非常麻烦的关于着色器编程的几章内容,而先开始使用 OpenGL 呢?有很多工具可以帮助我们在一个 IDE 类型环境中编写着色器,其中某些还将这种方式作为初始方式,这样做有它的好处,而我对此也并不排斥,但是,我更喜欢能够运行的实际程序,以便做一些有趣的事情,这就是我学习编程的方式。然后我可以将程序发送给我的母亲、女朋友、伙伴、老板等,向他们展示我的聪明才智。当我们开始掌握新技术时,这是一种强有力的反馈机制。

这是我的学习方法,也是本书(畅销的)前4版所采用的模式,在第5版我仍然不愿意放弃这种方式,希望你会喜欢我的努力成果。

——Richard S. Wright, Jr.

# 致 谢

感谢 Nick 和 Graham, 谢谢你们承担了本版中这么多的工作; 谢谢你们对我的代码进行调试, 及时发现那些愚蠢的错误而不至于被读者看到, 并且尽管在大多数时候都比我更更有才智, 却让我获得了大部分赞誉。谢谢你, Debra Williams Cauley, 在本书新的一版中再次给予我信任, 给予我帮助, 并且在我没能及时完成稿件时没有将催稿函发到我家——至少没有经常这样做。Songlin, 总有一天我会明白“你的”和“你是”、“它的”和“它是”之间的差别。谢谢你让我看起来好像真的通过了八年级, 没有使我觉得自己总是像一个笨蛋。Brian Collins 和 Chris “Xenon” Hanson 在审阅本书早期素材时同样做了大量工作, 并且发现了很多潜在的尴尬错误。我会永远感激你们, 随时来市里, 我还欠你们一顿啤酒。

感谢佛塞大学, 十多年来让我能够在继续“白天工作”的同时讲授 OpenGL——特别是 Rob Catto, 总是在我时常遇到阻碍时放我一马并伸出援手。感谢我在图形学系的好朋友和同伴 Wendy “Kitty” Jones、Kent Ward 和 Nick Bullock, 谢谢你们所有精神和物质上的帮助, 不期而至的泰国美食, 以及某些时候代替我完成我的工作。

特别感谢软件 Software Bisque (Steve、Tom、Daniel 和 Matt) 让我每天都可以使用 OpenGL 做一些“真实”的东西, 并且为我提供了人们梦寐以求的可能是最酷的白天(还有晚间)工作。我还要感谢我的家庭, LeeAnne、Sara、Stephen 和 Alex。你们忍受了太多的情绪波动、飘忽不定的关注点和毫无规律的工作日程, 更不用提在我需要的时候给我的那些激励了。

最后, 感谢苹果公司, 没有让我在每一次需要重启计算机来改变操作系统时都要等待“正在安装重要更新, 请不要关闭计算机电源”。感谢 AMD/ATI 为我提供了那么酷的新工具, 它们确实很有帮助。非常高兴看到你们竭尽全力地支持 OpenGL 标准。

—Richard S. Wright, Jr.

首先感谢 Richard, 让我参与创造又一个 OpenGL 出版物里程碑的荣誉之旅。如果没有你的付出和奉献, 计算机图形学的学生们就会失去学习 3D 图形的必要工具。和你一起长期共事来支持 3D 图形特别是 OpenGL 是我的荣幸。感谢 Graham 帮助我们将这一版带入最前沿的 OpenGL 3.3。你目光如炬, 帮助我避免了许多麻烦, 并使这本书一直保持着紧扣主题。谢谢你, Debra Williams Cauley, 让我们轻松回到出版流程, 并且引导我们这些外行人完成工作。你的耐心是特别的、无私的。感谢 Songlin 专业的目光, 对我粗陋的文字进行润色。

如果没有那些为数众多的反馈意见，这本书就不可能这样成功。特别感谢 AMD 的 Mark Young，你一丝不苟地审阅了我的所有书稿，并且提供了宝贵的反馈意见，而你并没有义务这样做。Brian Collins 和 Chris Hanson，你们对于确保素材质量一流和没有错误来说至关重要，感谢你们及时的反馈。

我还要感谢 AMD 和所有杰出的 OpenGL 开发者，你们提供了极大的支持和帮助，促使 OpenGL 3.3 在实际硬件上真正可用，使那些实例变得好用，也让我的工作得以进行。感谢 Mark Young、Mais Alnasser、Ken Wang、Jaakko Kontinen、Murat Balci、Bird Zhang、Zhihong Wang、Frank Li、Erick Zolnowski、Qun Lin、Jesse Zhou、Ethan Wei、Zhaohua Dong，以及其他许多人，你们都做出了杰出的工作。特别感谢 Khronos 小组和所有为保持 OpenGL 作为唯一的真正跨平台 3D API 的先进性、相关性和竞争力而努力的团体。

当然，如果没有家庭和朋友的支持，我就无法完成这项工作。谢谢我的妻子 Anna，这些年你容忍了我含糊混乱的技术术语，同时又在自己的领域治病救人，并且在制药方面有所建树。谢谢你的耐心和支持，没有你我就无法成功。

——Nicholas Haemel

感谢我的合作者 Richard 和 Nick，也感谢我们的出版人对我的信任，让我能够勉力地应付这样厚的一本书。和你们共事是一种享受，我希望这仅仅是一个开始。谢谢 Dave，谢谢你说“当然，我喜欢 Graham”。谢谢 Debra，谢谢你在必要时（必要的时候经常出现）敦促我交稿。感谢 Songlin 教会我如何正确地设置文本格式。从我们的技术审稿人 Brian Collins 和 Chris “Xenon” Hanson 那里得到的反馈使我受益匪浅，并且让我确信自己没有说什么愚蠢的话。

我还要衷心感谢 AMD 的同事们。特别地，我要感谢 Mark Young，他甚至在我的稿件送给技术审稿人之前并且还没有成型时就阅读了它们。Mark 还作了大量努力来更新本书附录中的 OpenGL 参考。这真的让我喜出望外——谢谢你！感谢 Jaakko、Murat 和其他在我们为本书实例进行调试时提供了建议和帮助的每一个人。我真的非常享受我们的头脑风暴，你们的参与极具价值，如果没有你们，我所做的一半工作都会不好用。感谢 Bill 和 Nick（再次感谢）帮助我联系 Khronos 和 ARB。Pierre，你是一位了不起的导师。感谢 Suki 使我能够脱离工作内容，来做那么多我想做的事，你为我提供了绝好的机会，我非常感激。

我要感谢一直以来帮助我的每个人：我在 Epson 的老同事，ARB 的同僚——感谢你们如此包容这个只出现了一天的家伙。这些人中很多都是我的竞争对手，我很庆幸我们能够在这么多事情上联手合作。

最后，我要深深地感谢我的家庭。Chris，你真棒，你给了我这么多，我爱你。Jeremy，你太棒了。妈妈、爸爸，谢谢你们创造了我！Barry 和 Phyllis，谢谢你们创造了 Chris！

——Graham Sellers

## 关于作者

**Richard S. Wright, Jr.** 拥有 15 年以上的 OpenGL 使用经验。他在美国佛罗里达州奥兰多附近的佛塞大学游戏设计学位课程 (game design degree program) 中教授编程, 至今已逾 10 年。现在, Richard 是 Software Bisque 公司的资深工程师, 是负责 Seeker (一款 3D 太阳系模拟软件) 的技术主管, 也是他们的全圆顶剧场天文馆 (full dome theater planetarium) 产品的产品经理。

多年前在洛克希德曼丁公司的 Real 3D 子公司时, Richard 是正规 OpenGL ARB 的成员, 并参与了 OpenGL 1.2 规范和一致性测试。从那时起, Richard 就开始致力于 Windows、Linux、Mac OS X 和各种移动平台上的多维数据库可视化、游戏开发、医学诊断可视化和天文空间模拟。

Richard 最早是在 1978 年上 8 年级时开始在纸质终端上学习编程的。在 16 岁时, 他的父母让他用割草得到的钱来买一台计算机而不是汽车, 过了不到一年, 他就卖出了他的第一款计算机程序 (并且这还是一个图形程序!)。当高中毕业时, 他的第一份工作是为当地消费教育公司教授编程和计算机文化。他在路易斯维尔大学 Speed 科学院学习电机工程和计算机科学, 并且在高年级上到一半时就完成了学业, 然后他的职业生涯到了最佳时期并来到了佛罗里达州。作为一个土生土长的肯塔基州路易斯维尔人, 他目前和他的妻子、3 个孩子一起居住在佛罗里达州的玛丽湖。在编程和躲避飓风之外的闲暇时间里, Richard 是一个热心的天文爱好者和摄影爱好者。Richard 还是一个 Mac 拥护者, 并以此为荣。

**Nicholas Haemel** 在 OpenGL 得到广泛接受之后不久就开始接触它了, 迄今已逾 12 年。他毕业于密尔瓦基工学院 (Milwaukee School of Engineering) 并获得计算机工程学位, 并且开始热爱嵌入式系统、计算机硬件以及让它们运行起来。刚一毕业, 他就在 ATI 的 3D 驱动小组中发挥了这些技能, 开发图形驱动和新的 GPU。

现在, Nick 是 AMD (高级微设备公司) OpenGL 小组的技术组成员, 并且已经成为驱动构架、设计和开发中的重要贡献者。他还领导着工作站 OpenGL 市场的所有行动和项目。在过去的 4 年中, Nick 还参加了体系结构审核委员会 (ARB, 现在是 Khronos 小组的一部分), 并参与了 OpenGL 3.0、3.1、3.2、3.3 和 4.0 规范和相关扩展及 GL 着色语言版本的定义。除了 OpenGL 之外, 他还为 OpenGL ES、WebGL 和 EGL 工作组作出了贡献。

Nick 的图形生涯开始于 9 岁那年, 也就是他第一次学习使用 Logo Writer 进行 2D 图形编程时。在说服父母购买了一台最新型的 286IBM 兼容机之后, 这台计算机马上就成了机械手和其他远程可编程设备的中心控制单元。20 年后, 这些被控制的设备则是指甲大小但却包含 20 亿个晶体管的 GPU。Nick 的兴趣还包括商业领导和管理, 特别是最近获得威斯康星大学麦迪逊分校 (University of Wisconsin-Madison)

的 MBA 之后更是如此。这所学校也是他现在的居住地。在致力于推进未来图形硬件发展的工作之余, Nick 喜欢作为竞技水手、登山队员、滑雪选手、自行车运动员和摄影师进行户外运动。

**Graham Sellers** 是一个典型的极客(geek)。在他 6 岁生日前夕, 家里拥有了第一台计算机( BBC Model B )。在父母通宵用计算机进行编程来播放“生日快乐”之后, 他就迷上了它, 并且决定要搞清楚它是如何工作的。接下来就是基础编程, 然后是装配语言。他第一次真正接触图形是在上世纪 90 年代早期的“demos”, 然后是 Glide, 最后是上世纪 90 年代晚期的 OpenGL。他拥有英国南安普顿大学的工程硕士学位。

现在, Graham 是 AMD 的 OpenGL 驱动小组中的一名经理。他在 ARB 中代表 AMD, 并对核心 OpenGL 规范和很多扩展作出了贡献。在此之前, 他曾经是爱普生的一位项目组长, 负责实现嵌入式产品的 OpenGL-ES 和 OpenVG 驱动。Graham 在计算机图形和图像处理领域拥有多项专利。在从事相关 OpenGL 工作之余的闲暇时间里, 他喜欢进行老视频游戏控制台的分解和反向工程( 仅仅是为了解它们如何工作, 以及他能利用它们做些什么 )。Graham 来自英格兰, 目前与他的妻子和儿子一起居住在佛罗里达州的奥兰多。



# 前言

欢迎阅读本书第 5 版。在过去十多年的时间里，我们致力于不仅为 OpenGL，也为通用 3D 图形编程提供全世界最好的入门指南。本书既是 OpenGL API 的全面参考书，也是一本能够教会读者如何使用这个强大的 API 来创建绝妙的 3D 可视化、游戏和其他所有类型图形的教程。本书从基础 3D 术语和概念开始，带领读者学习基本图元装配、变换、光照、纹理，并最终带领读者学习使用 OpenGL 着色语言的可编程图形管线的完整功能。

无论读者在 Windows、Mac OS X、Linux，还是在手持游戏设备上编程，本书都是开始学习 OpenGL，以及在读者特定平台上最大程度利用它的好工具。本书的主体是以 GLUT 或 FreeGLUT 工具箱为宿主的高度可移植的 C++ 代码。读者还可以在本书中看到特定操作系统的章节，这些章节展示如何将 OpenGL 写入本地窗口系统。在本书中，我们自始至终努力不去假设读者有多少 3D 图形编程预备知识，这样就促成了这本初学的程序员和经验丰富的程序员开始学习 OpenGL 时都能接受的教程。

## 第 5 版有哪些更新

本书前几版的读者可能会立刻注意到，这本书变薄了。这是怎么回事？在 OpenGL 3.0 中，某些特性被标记为“不鼓励使用的”，也就是说，这些特性在未来的 OpenGL 版本中可能会被删除。到目前为止，OpenGL 中还没有正式删去任何特性，这在很大程度上是迫于广大开发人员的压力。这样一来，我们在讲解 OpenGL 就有了两种思路，即包括所有最新功能和“不鼓励使用的”功能的完整框架，以及不包括任何“不鼓励使用的”功能的核心框架。鉴于标记“不鼓励使用的”功能的主要目的是推动 OpenGL 标准的发展，本版没有包含任何“不鼓励使用的”功能，而只专注于核心框架。这些核心框架是基于 OpenGL 3.3 的。

我们保留了已经证实非常受欢迎的书后参考资料，不过同样删去了所有“不鼓励使用的”功能。如果读者想实现最新的和具有前瞻性的 OpenGL 程序，那么从这一部分开始是非常好的选择。教程部分的各章内容 95%（或以上）都是新素材。我们不想采用基于“不鼓励使用的”功能的方式，所以就产生了采用全新素材的全新方式。这其中包括本版中关于操作系统特性的各章内容，这些内容几乎是完全重写的。

关于 OpenGL ES 的章节特别加入了在 iPhone 上使用 OpenGL ES 的内容，其中还包括了 iPod 和

iPad，并且本书前面的一些例子也引入了这些设备。这些新加入的内容是非常受欢迎的，因为在本书编写时，再没有一种主流 OpenGL ES 设备能够像它们这样为任何（使用 Mac）用户轻松使用。

在本版中，GLTools 库部分也被明显地加强了。书中归纳的存储着色器(stock shader)能帮助读者在真正开始研究编写自己的着色器前就尽快学会如何使用着色器。另外，本书归纳的轻量级 C++ 类允许对我们的几何批处理进行管理，并帮助我们创建和操作自己的矩阵堆栈。像过去的 GLU 库一样，这个库也应只被视为一套帮助例程，而不是一个完整的 OpenGL 应用框架。

## 本书的组织结构

本书共分为 3 部分。第一部分是 OpenGL/3D 图形学基本教程；第二部分涵盖了更加深入的 OpenGL 编程主题；第三部分则涵盖了一些操作系统特定性质，帮助读者在选定的平台上对 OpenGL 进行充分的利用。这 3 部分之后是 3 个附录，包含了其他 OpenGL 优秀参考资源和教程的汇总、一个简短的术语表，以及 OpenGL 核心框架的完整参考。

## 第一部分 基本概念

在这一部分，读者将学到如何构建一个使用 OpenGL 的程序，如何设置 3D 渲染环境，以及如何创建基本对象和光线并对他们进行着色。然后，我们将深入研究如何使用 OpenGL，并向读者介绍 GLSL，以及如何创建自己的着色器。这些章节是读者认识使用 OpenGL 进行 3D 图形编程很好的方式，并提供关于本书后面将讲到的更多高级性能的基础概念。

**第 1 章 3D 图形和 OpenGL 简介** 本章是一个针对 3D 图形新手的介绍性章节。它介绍了基本概念和一些通用词汇。

**第 2 章 入门指南** 在本章，我们向读者提供了关于 OpenGL 是什么、它从何而来以及将如何发展的应用知识。读者将编写自己的第一个使用 OpenGL 的程序，找出需要使用哪些头和库，学习如何设置环境，并发现一些通用惯例如何帮助读者记住 OpenGL 函数调用。本章还会介绍 OpenGL 状态机和错误处理机制。

**第 3 章 基础渲染** 在本章，我们展示 3D 图形编程的构造块。读者可以大致明白如何告诉计算机用几何图元、着色器在 OpenGL 中创建一个三维对象，建立统一样式和属性。读者也可以学到如何消除隐藏表面和使用模板缓冲区的方法，以及查询 OpenGL 驱动获得实现细节的各种不同方式。

**第 4 章 基础变换：初识向量/矩阵** 现在我们在虚拟世界创建三维形状，如何使它们移动？如何使自己移动？这些都在这一章学习。本章真正属于 OpenGL 的内容很少，但却澄清了读者继续深入学习所需要了解的概念。

**第 5 章 基础纹理** 纹理贴图在任何 3D 图形工具箱中都是最有用的特性之一。我们将学到如何将图像缠绕到多边形上，以及如何立即载入和形成多纹理。

**第 6 章 跳出“盒子”：非存储着色器** 现在我们已经具备 OpenGL 客户端程序设计基础，可以开始在服务器端如何使用 GLSL 编写着色器下功夫了。本章通过一些基于前面所学内容使用存储着色器创建的

例子来介绍这方面内容。

**第7章 纹理高级知识** 学习基础纹理后,在本章我们将学习立方体贴图、3D 纹理,并且只用纹理存储数据。我们还可以学习点精灵和一些其他类型的非可视纹理应用。

## 第二部分 深入探索

在本书的第二部分,我们将进行更深入的研究。这一部分关于 OpenGL 的内容更加令人振奋,而懂得如何应用这些高级特性将使读者超越业余 3D 玩家的水平。这一部分不仅能够使我们掌握更多的可视效果,而且其中很多内容还是性能导向的。

**第8章 缓冲区对象:存储尽在掌握** OpenGL 不再支持客户端的数据存储。在本章,读者将学习 OpenGL 中不同类型存储缓冲区的输入输出,包括如何在离屏缓冲区中渲染。

**第9章 高级缓冲区:超越基础水平** 本章将向读者展示如何进一步学习更高水平的缓冲区知识,并介绍一些非常有用但却并不典型的缓冲区格式。

**第10章 片断操作:管线的终点** 如果片段着色器的颜色、深度或其他数据变得不精确时,仍然需要进行很多处理。本章讨论对逐个片段的操作,包括非常有用的模板测试。

**第11章 高级着色器应用** 本章将帮助读者在着色器编程中引入几何着色器,此外还将介绍更多高级着色器管理和诸如统一块(uniform block)等应用模式。

**第12章 高级几何图形管理** 本章是第二部分的最后一章,介绍一些几何与渲染操作的高级管理方法与技巧。OpenGL 中有一些有用的特性可以用来优化大量几何图形的处理过程,以及消除事先不可见的几何图形。最后,实际上 OpenGL 中现在还内建了一些有用的时间特性。

## 第三部分 特定平台应用

本书的第三部分也是最后一部分,主要是关于各种操作系统接口如何带有和使用 OpenGL 的,而不是关于 OpenGL 本身的。这部分内容游离在“官方”OpenGL 规范之外,来了解 OpenGL 是如何支持和连接 Windows、Mac OS X、Linux 和掌上设备的,例如使用 OpenGL ES 2.0 的 iPhone。

**第13章 Windows 上的 OpenGL** 在本章,我们将学习如何编写真正的使用 OpenGL 的 Windows 程序。我们将学习 Microsoft 的“wiggly”函数,这个函数将 OpenGL 渲染代码与 Windows 设备环境结合起来。

**第14章 OS X 上的 OpenGL** 在本章,我们将学习如何在本地 Mac OS X 应用程序上使用 OpenGL。示例程序将为我们展示如何使用 Xcode 开发环境开始应用 GLUT、Carbon 或 Cocoa。

**第15章 Linux 上的 OpenGL** 本章讨论 GLX,一种用于通过 UNIX 和 Linux 上的 X Window 系统来支持 OpenGL 应用程序的 OpenGL 扩展。我们将学习如何创建和管理 OpenGL 环境,以及如何创建 OpenGL 绘图区域。

**第16章 OpenGL ES:移动设备上的 OpenGL** 本章完全是关于 OpenGL 如何进行精简以适应手持和嵌入式设备的。我们将了解删去了什么、新增了什么,以及如何在仿真环境下运行。本章甚至还将一个桌面示例程序移植到了 iPhone 上。

## 关于合作网站

这是本书第二次没有搭配 CD-ROM 发行。欢迎进入 Internet 时代！所有源代码都可以从支持网站在线下载。  
[www.starstonesoftware.com/OpenGL](http://www.starstonesoftware.com/OpenGL)

在这里可以找到所有示例程序的源代码，以及为 Developers Studio ( Windows ) 和 Xcode ( Mac OS X ) 预先建立的项目。对于 Linux 用户，我们也为项目的命令行建立制作了文件，甚至计划公布一些教程，所以请不时地进行核对，即使是在下载完所有源代码之后。



# 目 录

## 第一部分 基本概念

### 第 1 章 3D 图形和 OpenGL 简介 ..... 2

- 1.1 计算机图形的简单历史回顾 ..... 2
  - 1.1.1 进入电子时代 ..... 3
  - 1.1.2 走向 3D ..... 3
- 1.2 3D 图形技术和术语 ..... 6
  - 1.2.1 变换 ( Transformation ) 和投影 ( Projection ) ..... 6
  - 1.2.2 光栅化 ( Rasterization ) ..... 6
  - 1.2.3 着色 ..... 7
  - 1.2.4 纹理贴图 ..... 8
  - 1.2.5 混合 ..... 9
  - 1.2.6 将点连接起来 ..... 9
- 1.3 3D 图形的常见用途 ..... 9
  - 1.3.1 实时 3D ..... 10
  - 1.3.2 非实时 3D ..... 12
  - 1.3.3 着色器 ..... 12
- 1.4 3D 编程的基本原则 ..... 13
  - 1.4.1 并非工具包 ..... 13
  - 1.4.2 坐标系统 ..... 13
  - 1.4.3 投影: 从 3D 到 2D ..... 17
- 1.5 总结 ..... 19

### 第 2 章 入门指南 ..... 20

- 2.1 什么是 OpenGL? ..... 20
  - 2.1.1 标准的演化 ..... 21
  - 2.1.2 OpenGL 的未来 ..... 24
- 2.2 使用 OpenGL ..... 27
  - 2.2.1 支持阵容 ..... 28
  - 2.2.2 OpenGL API 特性 ..... 29

- 2.2.3 OpenGL 错误 ..... 31
- 2.2.4 确认版本 ..... 31
- 2.2.5 使用 glHint 获取线索 ..... 32
- 2.2.6 OpenGL 状态机 ..... 32
- 2.3 建立 Windows 项目 ..... 33
  - 2.3.1 包含路径 ..... 34
  - 2.3.2 创建项目 ..... 35
  - 2.3.3 添加文件 ..... 36
- 2.4 建立 Mac OS X 项目 ..... 38
  - 2.4.1 自定义创建设置 ..... 38
  - 2.4.2 创建新项目 ..... 39
  - 2.4.3 框架、头文件和库 ..... 41
- 2.5 第一个三角形 ..... 43
  - 2.5.1 要包含什么 ..... 45
  - 2.5.2 启动 GLUT ..... 45
  - 2.5.3 坐标系基础 ..... 47
  - 2.5.4 完成设置 ..... 50
  - 2.5.5 言归正传 ..... 52
- 2.6 加点儿活力! ..... 53
  - 2.6.1 特殊按键 ..... 53
  - 2.6.2 刷新显示 ..... 54
  - 2.6.3 简单的动画片 ..... 54
- 2.7 总结 ..... 55

### 第 3 章 基础渲染 ..... 56

- 3.1 基础图形管线 ..... 57
  - 3.1.1 客户机-服务器 ..... 57
  - 3.1.2 着色器 ..... 58
- 3.2 创建坐标系 ..... 60

3.2.1 正投影 .....	60	4.6.2 透视投影 .....	110
3.2.2 透视投影 .....	61	4.6.3 模型视图投影矩阵 .....	111
3.3 使用存储着色器 .....	61	4.7 变换管线 .....	113
3.3.1 属性 .....	62	4.7.1 使用矩阵堆栈 .....	114
3.3.2 Uniform 值 .....	62	4.7.2 管理管线 .....	115
3.4 将点连接起来 .....	64	4.7.3 加点调料 .....	118
3.4.1 点和线 .....	64	4.8 使用照相机和角色进行移动 .....	119
3.4.2 绘制 3D 三角形 .....	68	4.8.1 角色帧 .....	120
3.4.3 单独的三角形 .....	68	4.8.2 欧拉角: “卢克! 请使用帧” .....	121
3.4.4 一个简单批次容器 .....	72	4.8.3 照相机管理 .....	121
3.4.5 不希望出现的几何图形 .....	73	4.8.4 添加更多角色 .....	123
3.4.6 多边形偏移 .....	78	4.8.5 关于光线 .....	125
3.4.7 裁剪 .....	80	4.9 小结 .....	126
3.5 混合 .....	81		
3.5.1 组合颜色 .....	81	<b>第 5 章 基础纹理 .....</b>	<b>127</b>
3.5.2 改变混合方程式 .....	84	5.1 原始图像数据 .....	128
3.5.3 抗锯齿 .....	85	5.1.1 像素包装 .....	129
3.5.4 多重采样 .....	87	5.1.2 像素图 .....	130
3.6 小结 .....	89	5.1.3 包装的像素格式 .....	132
<b>第 4 章 基础变换: 初识向量/矩阵 .....</b>	<b>90</b>	5.1.4 保存像素 .....	133
4.1 本章是令人生畏的数学课吗 .....	90	5.1.5 读取像素 .....	134
4.2 3D 图形数学速成课 .....	91	5.2 载入纹理 .....	137
4.2.1 向量 .....	91	5.2.1 使用颜色缓冲区 .....	138
4.2.2 矩阵 .....	94	5.2.2 更新纹理 .....	138
4.3 理解变换 .....	95	5.2.3 纹理对象 .....	139
4.3.1 视觉坐标 .....	95	5.3 纹理应用 .....	140
4.3.2 视图变换 .....	96	5.3.1 纹理坐标 .....	140
4.3.3 模型变换 .....	96	5.3.2 纹理参数 .....	142
4.3.4 模型视图的二元性 .....	98	5.3.3 综合运用 .....	144
4.3.5 投影变换 .....	98	5.4 Mip 贴图 .....	148
4.3.6 视口变换 .....	99	5.4.1 Mip 贴图过滤 .....	149
4.4 模型视图矩阵 .....	99	5.4.2 生成 Mip 层 .....	150
4.4.1 矩阵构造 .....	100	5.4.3 活动的 Mip 贴图 .....	150
4.4.2 运用模型视图矩阵 .....	103	5.5 各向异性过滤 .....	158
4.5 更多对象 .....	105	5.6 纹理压缩 .....	160
4.5.1 使用三角形批次类 .....	105	5.6.1 压缩纹理 .....	160
4.5.2 创建一个球体 .....	106	5.6.2 加载压缩纹理 .....	161
4.5.3 创建一个花托 .....	106	5.6.3 最后一个示例 .....	162
4.5.4 创建一个圆柱或圆锥 .....	107	5.7 小结 .....	163
4.5.5 创建一个圆盘 .....	108		
4.6 投影矩阵 .....	108	<b>第 6 章 跳出“盒子”: 非存储着色器 .....</b>	<b>164</b>
4.6.1 正投影 .....	109	6.1 初识 OpenGL 着色语言 .....	164

6.5.4 卡通着色( Cell Shading )——将  
纹理单元作为光线 ..... 205

6.6 小结 ..... 207

7.1	矩形纹理 .....	208
7.1.1	加载矩形纹理 .....	209
7.1.2	使用矩形纹理 .....	209
7.2	立方体贴图 .....	212
7.2.1	加载立方体贴图 .....	212
7.2.2	创建天空盒 .....	213
7.2.3	创建反射 .....	215
7.3	多重纹理 .....	216
7.3.1	多重纹理坐标 .....	217
7.3.2	多重纹理示例 .....	217
7.4	点精灵 (点块纹理) .....	219
7.4.1	使用点 .....	220
7.4.2	点大小 .....	220
7.4.3	综合运用 .....	221
7.4.4	点参数 .....	223
7.4.5	异形点 .....	224
7.4.6	点的旋转 .....	225
7.5	纹理数组 .....	226
7.5.1	加载 2D 纹理数组 .....	226
7.5.2	纹理数组索引 .....	228
7.5.3	访问纹理数组 .....	228
7.6	纹理代理 .....	229
7.7	小结 .....	230

8.2.5	在帧缓冲区中复制数据.....	250
8.2.6	FBO 综合运用 .....	251
8.3	渲染到纹理.....	254
8.4	小结.....	259

9.1	获得数据 .....	260
9.1.1	映射缓冲区 .....	261
9.1.2	复制缓冲区 .....	262
9.2	控制像素着色器表现， 映射片段输出 .....	262

9.3 新一代硬件的新格式 .....	264	11.2.5 在几何着色器中生成 几何图形 .....	314
9.3.1 浮点——最终的真实精确 .....	264	11.2.6 在几何着色器中改变 图元类型 .....	317
9.3.2 多重采样 .....	276	11.2.7 由几何着色器引入的 新图元类型 .....	319
9.3.3 整数 .....	279	11.3 高级片段着色器 .....	321
9.3.4 sRGB .....	280	11.3.1 片段着色器中的后期处理—— 颜色校正 .....	322
9.3.5 纹理压缩 .....	281	11.3.2 片段着色器中的后期处理—— 卷积 .....	323
9.4 小结 .....	283	11.3.3 在片段着色器中生成 图像数据 .....	326
<b>第 10 章 片段操作：管线的终点 .....</b>	<b>284</b>	11.3.4 在片段着色器中丢弃工作 .....	328
10.1 裁剪——将几何图形剪切到 希望的大小 .....	285	11.3.5 逐片段控制深度 .....	329
10.2 多重采样 .....	285	11.4 更高级的着色器函数 .....	330
10.2.1 样本覆盖 .....	285	11.4.1 插值和存储限定符 .....	330
10.2.2 样本遮罩 .....	286	11.4.2 高级内建函数 .....	333
10.2.3 综合运用 .....	287	11.5 统一缓冲区对象 .....	334
10.3 模板操作 .....	290	11.5.1 建立统一块 .....	335
10.4 深度测试 .....	292	11.6 小结 .....	342
10.4.1 深度截取 .....	292		
10.5 进行混合 .....	293	<b>第 12 章 高级几何图形管理 .....</b>	<b>343</b>
10.5.1 混合方程式 .....	293	12.1 查询功能——收集 OpenGL 管线相关信息 .....	343
10.5.2 混合函数 .....	294	12.1.1 准备查询 .....	344
10.5.3 综合运用 .....	295	12.1.2 发出查询 .....	345
10.6 抖动 .....	296	12.1.3 取回查询结果 .....	345
10.7 逻辑操作 .....	297	12.1.4 使用查询结果 .....	346
10.8 遮罩输出 .....	298	12.1.5 让 OpenGL 决定 .....	349
10.8.1 颜色 .....	298	12.1.6 测量执行命令所需时间 .....	350
10.8.2 深度 .....	298	12.2 在 GPU 内存中存储数据 .....	352
10.8.3 模板 .....	298	12.2.1 使用缓冲区存储顶点数据 .....	353
10.8.4 用途 .....	299	12.2.2 在缓冲区中保存顶点索引 .....	356
10.9 小结 .....	299	12.3 使用顶点数组对象来组织缓冲区 .....	358
<b>第 11 章 高级着色器应用 .....</b>	<b>300</b>	12.4 高效地绘制大量几何图形 .....	359
11.1 高级顶点着色器 .....	300	12.4.1 组合绘制函数 .....	360
11.1.1 在顶点着色器中进行 物理模拟 .....	301	12.4.2 使用图元重启对几何图形进行 组合 .....	361
11.2 几何着色器 .....	306	12.4.3 实例渲染 .....	362
11.2.1 直通几何着色器 .....	306	12.4.4 自动获得数据 .....	367
11.2.2 在应用程序中使用几何 着色器 .....	308	12.5 存储变换的顶点——变换反馈 .....	371
11.2.3 在几何着色器中丢弃 几何图形 .....	311	12.5.1 变换反馈 .....	371
11.2.4 在几何着色器中修改 几何图形 .....	313		

12.5.2 关闭光栅化 .....	376
12.5.3 使用图元查询对顶点 进行计数 .....	376
12.5.4 使用图元查询的结果 .....	378
12.5.5 变换反馈的应用实例 .....	379

12.6 裁剪并确定绘制内容 .....	386
12.6.1 裁剪距离——自定义 裁剪空间 .....	387
12.7 在 OpenGL 开始绘制时进行同步 ..	389
12.8 小结 .....	392

### 第三部分 特定平台应用

#### 第 13 章 Windows 上的 OpenGL ..... 394

13.1 Windows 中的 OpenGL 实现 .....	395
13.1.1 微软的 OpenGL .....	395
13.1.2 现代图形驱动程序 .....	395
13.1.3 扩展 OpenGL .....	396
13.1.4 WGL 扩展 .....	398
13.2 基本窗口渲染 .....	399
13.2.1 GDI 设备环境 .....	399
13.2.2 像素格式 .....	400
13.2.3 OpenGL 渲染环境 .....	406
13.3 综合运用 .....	409
13.3.1 创建窗口 .....	410
13.4 全屏渲染 .....	414
13.5 双重缓冲 .....	415
13.5.1 消除视觉撕裂 .....	415
13.6 小结 .....	416

#### 第 14 章 OS X 上的 OpenGL ..... 417

14.1 OpenGL 在 Mac 上的 4 种接口 ..	417
14.2 在 OpenGL 中使用 Cocoa .....	418
14.2.1 创建一个 Cocoa 程序 .....	418
14.2.2 综合运用 .....	423
14.2.3 双缓冲还是单缓冲 .....	425
14.2.4 球体世界 .....	425
14.3 全屏渲染 .....	429
14.3.1 在 Cocoa 中进行全屏显示 ..	430
14.4 CGL .....	435
14.4.1 同步帧速率 .....	435
14.4.2 提高填充性能 .....	436
14.4.3 多线程 OpenGL .....	437
14.5 小结 .....	437

#### 第 15 章 Linux 上的 OpenGL ..... 438

15.1 基础知识 .....	438
-----------------	-----

15.1.1 简史 .....	439
15.1.2 什么是 X Window .....	439
15.2 入门讲解 .....	439
15.2.1 检查 OpenGL .....	440
15.2.2 设置 Mesa .....	440
15.2.3 设置 Mesa 硬件驱动程序 ..	441
15.2.4 设置 GLUT 和 GLEW .....	441
15.2.5 创建 OpenGL 应用程序 ...	442
15.3 GLX——X Window 的接口 .....	443
15.3.1 显示和 X Window .....	444
15.3.2 配置管理和显示效果 .....	444
15.3.3 窗口和渲染表面 .....	447
15.3.4 OpenGL 和 GLX 扩展 .....	448
15.3.5 环境管理 .....	448
15.3.6 同步 .....	451
15.3.7 GLX 查询 .....	452
15.3.8 综合运用 .....	453
15.4 小结 .....	455

#### 第 16 章 OpenGL ES: 移动设备上的 OpenGL ..... 456

16.1 精简的 OpenGL .....	456
16.1.1 ES 指什么 .....	457
16.1.2 历史概述 .....	457
16.2 版本选择 .....	458
16.2.1 ES 2.0 .....	459
16.3 ES 环境 .....	463
16.3.1 应用程序设计的注意事项 ..	463
16.3.2 有限环境的处理 .....	464
16.3.3 定点数学 .....	464
16.4 EGL: 新的窗口环境 .....	465
16.4.1 EGL 显示 .....	466
16.4.2 创建窗口 .....	467
16.4.3 环境管理 .....	470
16.4.4 呈现缓冲区和渲染同步 .....	471

16.4.5 更多关于 EGL 的内容 .....	472
16.5 处理嵌入式环境 .....	473
16.5.1 流行的操作系统 .....	473
16.5.2 供应商特定扩展 .....	473
16.5.3 个人玩家 .....	473
16.6 苹果掌上平台 .....	474
16.6.1 设置 iPhone 项目 .....	474
16.6.2 移植到 iPhone .....	477
16.7 小结 .....	483
<b>附录 A 更多阅读建议 .....</b>	<b>484</b>
<b>附录 B 词汇表 .....</b>	<b>486</b>
<b>附录 C (核心) OpenGL 3.3 参考 .....</b>	<b>489</b>



## 第一部分 基本概念

与读者以前听说的可能不同，OpenGL（对其他 3D 应用程序接口来说也是如此）3D 图形编程并不都是关于着色器的。不管是使用 C、C++、C#、JavaScript 等编程语言的哪一种，客户端都必须完成相当多的工作，来管理这些着色器，以及向它们馈送（feed）几何图形、变换矩阵和其他各种数据。

本书的第一部分是真正的教程——3D 图形编程教程，从基本原则开始讲述，当然这些内容都是基于实时 3D 图形渲染的行业标准 OpenGL 的。

着色器编程非常令人兴奋，但是作者并不打算将本书写成一本着色器编程书籍。实际上，如果没有如何管理场景，设置视点、模型和变换矩阵，以及载入纹理等知识，那么即使懂得如何编写优秀的着色器也难有所成……我想读者一定明白我的意思。

为了帮助读者上手，本书提供了一个小型的“存储着色器”库，它们能够完成大多数常规任务。读者甚至可能会发现，对于简单 3D 渲染来说，这些着色器已经能够满足所有需要了。但是，随着知识的增长，读者可能并不会满足于此。在进入第二部分之前，还为读者准备了 GLSL “快速开始”，因此读者无需等到完全掌握 OpenGL 应用程序接口就能开始发挥创造性了。

读者会发现网络世界（就像其他一些优秀图书一样）是一个巨大的知识库，充满了高级的、简单的和聪明的着色器代码，能够实现许多的目的。一旦有了足够的关于如何充分利用所有这些优秀着色器的应用知识，读者将会希望编写自己的着色器。为此，我们在附录 A 中向读者推荐了一些非常好的资源。

# 第 1 章 3D 图形和 OpenGL 简介

作者: Richard S. Wright, Jr.

## 本章内容

- ✦ 简单介绍计算机图形的历史
- ✦ 如何在 2D 屏幕上创建 3D 图形
- ✦ 基本的 3D 效果和术语
- ✦ 3D 坐标系统和视口的工作原理
- ✦ 什么是顶点以及如何使用
- ✦ 不同类型的 3D 投影

本书讲述的是 OpenGL，这是一种用于创建实时 3D 图像的编程接口。在讨论 OpenGL 究竟是什么以及它的工作方式之前，读者至少应该在一个比较高的层次上对实时 3D 图形有一个基本的了解。也许读者已经对实时 3D 的原则有了较深的理解，阅读本书的目的只是为了学习如何使用 OpenGL。如果是这样的话，那就太好了，请直接跳到第 2 章。但是，如果读者购买本书的原因是觉得这本书中的图片非常酷，并且希望学会如何在自己的 PC 中创建这样的图片，那么很可能需要从本章开始认真学习。

## 1.1 计算机图形的简单历史回顾

最早的计算机是由一行行的开关和灯组成的。技术人员和工程师需要工作几个小时、几天甚至几星期，对这些机器进行编程，并阅读它们的计算结果。发光灯泡所组成的各种模式向计算机用户传达有用的信息。此外，计算机或许还可能产生一些原始的打印信息。读者可以认为计算机图形的最初形式就是在一块面板上闪烁的灯（早期编写程序的程序员流传的一些故事似乎也印证了这个想法，他们所编写的程序除了追踪灯光、创建各种闪烁模式之外，就几乎没有其他用途了！）。

随着时间的变迁，这一切逐渐发生了变化。最初所谓的那些“思考机器”发展为完全可编程的设备，使用一种类似电传打字机的机制在纸卷上进行打印。数据可以有效地存储在磁带、磁盘上，甚至可以一行行地存储在打孔纸上，或者存储在一堆穿孔卡上。计算机图形诞生于计算机最初开始能够打印的日子。由于字母表中的每个字符都有固定的大小和形状，上世纪 70 年代的那些富有创造力的程序员仅仅利用星号 (\*) 就创建了许多艺术图案和图像。

### 1.1.1 进入电子时代

纸作为计算机的输出媒体非常实用，直到今天仍然是主要的输出媒体之一。激光打印机和彩色喷墨打印机已经取代了原始的 ASCII 艺术，它们具有鲜活的表现质量和近乎照片般的艺术再现能力。但是，作为常规的显示媒体，纸可能显得过于昂贵了。一直用纸作为输出媒体会浪费自然资源，尤其在绝大多数情况下，我们实际上并不需要对计算结果或数据库查询进行硬复制的输出。

作为计算机的一种辅助设备，阴极射线管 (CRT) 是一项震撼人心的技术。作为最初的计算机监视器，CRT 一开始只是一种显示 ASCII 文本的视频终端，就像最初的纸终端一样。但是，CRT 能够完美地绘制点和线，以及字母字符。不久，其他符号和图形陆续补充到字符终端。程序员使用计算机和监视器创建图形，作为文本或表格输出的补充。首先，一些用于绘制直线和曲线的算法被开发出来，并且公布于众。于是，计算机图形逐渐从一项业余爱好变成了一门科学。

最初，显示在这些终端上的计算机图形是二维的，或简称为 2D。人们用平面的直线、圆和多边形来创建各种各样的图形。图形可以用一种表格或数字无法实现的方式显示科学或统计数据。那些富有探索精神的程序员甚至创建了一些简单的街机游戏 (Arcade Game)，如 Lunar Lander 和 Pong。它们所使用的简单图形就是由各种线型所绘制的，并且每秒刷新数次 (重绘)。

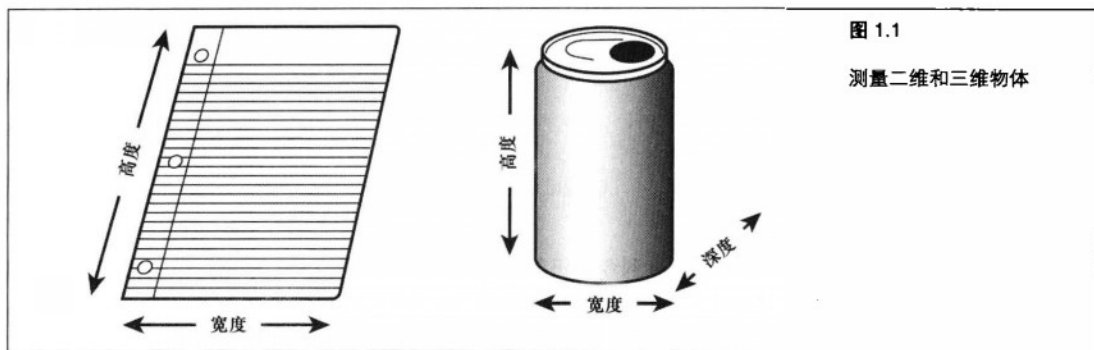
实时 (Real-Time) 这个术语最初应用于那些活动的计算机图形。在计算机科学中，这个术语更为广泛的含义是指计算机能够即时对输入进行处理。例如，人与人之间通过电话交谈就是一种实时活动。当你说话的时候，对方立即就能听到，并随即做出回答，然后又允许你立即做出响应，接下来循环往复。在现实中，由于电流的缘故，通话实际上存在延迟。但对通话者而言，这种延迟通常是感觉不到的。与电话相反，写信显然不是一种实时活动。

把实时这个术语应用到计算机图形中意味着计算机对诸如操作杆活动、击键等输入事件直接做出响应，产生动画或图像序列。实时计算机图形可以显示一种波形，由电子设备、数值读出器、交互性游戏和视觉模拟程序进行测量。

### 1.1.2 走向 3D

三维 (或 3D) 这个术语表示一个正在描述或显示的物体具有 3 个维度：宽度、高度和深度。例如，放在书桌上的一张纸 (上面画了一些图形或写了一些字) 是个二维物体，因为它没有可以令人感觉得到的深度。但是，放在它旁边的一罐苏打水却是个三维物体。这个苏打饮料罐又圆 (宽度和高度) 又长 (深度)。

根据观察点,我们可以把这个罐的任何一边作为宽度或高度,但这个罐头是个三维物体的本质并不会改变。图 1.1 所示显示了怎样测量这个罐头和这张纸的各个维度。

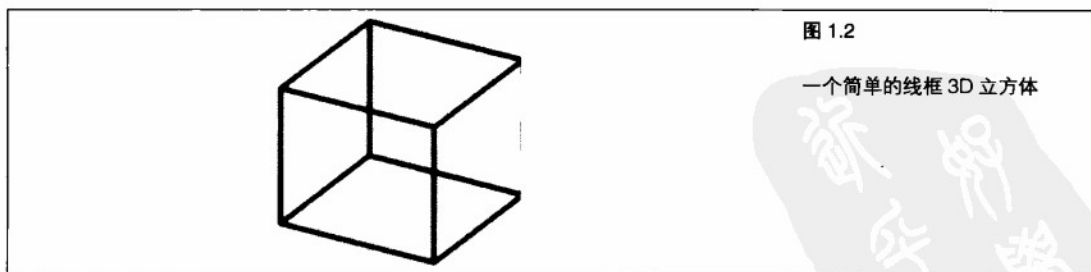


几个世纪以来,艺术家已经知道如何让一幅画看上去具有真实的深度。从本质上说,画本身是个二维物体,因为它只是用颜料在二维的画布上所创作的作品而已。类似地,计算机 3D 图形在实质上也是平面的,它只是在计算机屏幕上所显示的二维图像,但它可以提供深度(或第 3 维)的错觉。

$$2D + \text{透视} = 3D$$

毫无疑问,最初的计算机图形看上去类似于图 1.2 所示图形,我们可以看到 12 条线段组成了一个简单的三维立方体。使这个立方体看上去具有三维效果的是透视(Perspective),或线段之间的角度。正是它们产生了深度的幻觉。

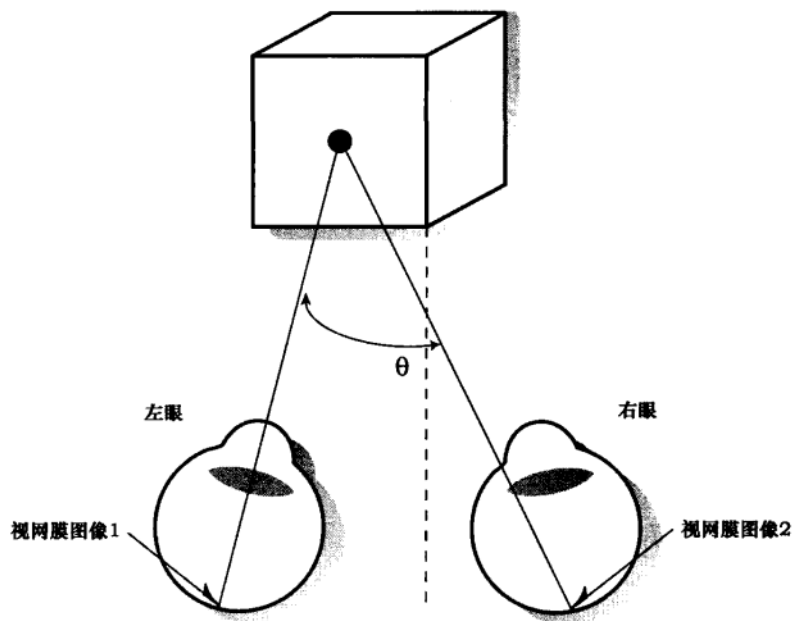
为了真正看到 3D 图像,实际上需要用两个眼睛观察一个物体,或者为每只眼睛分别提供这个物体的一幅独立而又唯一的图像。如图 1.3 所示,每只眼睛看到的都是一幅二维图像,非常类似于在每个视网膜(位于眼睛的后半部分)上显示了一幅临时照片。随后,大脑对这两幅略微不同的图像进行组合,在脑海中形成一幅单一的、合成的 3D 图片。



在图 1.3 中,随着物体逐渐变远,图像之间的角度也越来越小。我们可以通过加大两幅图像之间的角度来增强 3D 效果。立体单片(一种手持式立体镜片,在小时候可能玩过)和 3D 电影利用了这种效应,在每只眼睛上放置一块单独的镜片或者戴上一幅滤色眼镜,将两幅层次不一的图像分离。这些图像通常进行了过度增强,以取得某种戏剧性的效果或影院效果。图像卡上的快门玻璃和软件会在两只眼睛之间进行图像切换,在每只眼睛上显示变化的透视效果,从而产生了“真正”的立体 3D 体验。遗憾的是,许多人抱怨这个效果令他们头痛,或者令他们感到头晕目眩。

图 1.3

如何观察三维物体



计算机屏幕是在平面上显示平面图像,而不是通过不同的视角在两只眼睛上显示两幅图像。这样一来,绝大多数 3D 计算机图像实际上只是近似 3D。这种近似效果和多年来画家在绘画作品中实现视觉深度所使用的手法是一样的,就像人们用一只眼睛也能够观察到 3D 效果一样。

在生活中,读者可能已经注意到,当遮住一只眼睛时,所看到的世界并不会立即变成平面!当我们遮住一只眼睛时会发生什么呢?我们可能觉得自己看到的仍然是 3D 世界。但是,请尝试下面这个试验:把一只玻璃杯(或其他物体)放在一臂之外的地方,靠左手一侧(如果放得很近,这个戏法就不灵了),用右手遮住右眼,然后用左手去摸这只杯子(可能最好换用空的塑料杯)。注意,在摸到这个杯子之前,我们会觉得估计这个杯子的距离显得比较困难。现在,放开右眼,然后再去触摸这个杯子,我们就会发现这次判断杯子的距离要容易得多。现在,我们应该能够明白为什么只有一只眼睛的人在感知距离时常常会遇到困难了。

单凭透视本身就足以创建三维的外观。注意前面图 1.2 所显示的那个立方体。即使不进行着色,这个立方体仍然具有三维物体的外观。但是,如果长时间凝视这个立方体,就会发觉这个立方体的前后将会交换位置。由于图中缺少任何表面着色,大脑将会因此而产生混淆的感觉。这幅图不能提供足够的信息以帮助大脑确定它到底感知到了什么。我们遮住一只眼睛时所看到的世界没有突然看起来像平的,原因在于以二维形式观察时,很多 3D 世界的效果仍然存在。这些效果足以激发大脑辨别深度的能力。其中一个线索是由光线照射产生的表面着色,而另一个线索则是近处的物体看起来比远处的物体要大。这种透视效果称为透视缩短(Foreshortening)。这种效果加上颜色的改变、纹理、光照、着色以及各种不同的颜色强度共同组成了我们对三维图像的感知。

## 1.2 3D 图形技术和术语

本书的每一章都包含一个或多个示例程序用来演示这一章所讨论的编程技术。尽管本章有意避免了关于编程细节的讨论，但仍提供了一个示例程序向读者演示最低程度上需要熟悉的技术和术语，以帮助读者充分地利用本书。本章的示例程序叫做 BLOCK，读者可以从随书提供的示例程序集中的“Chapter 1”文件夹中找到它。

将数学和图形数据转换成 3D 空间图像的操作叫做渲染 (Rendering)。当这个术语作为动词使用时，指的是计算机创建三维图像时所经历的过程。它也作为名词使用，指的仅仅是最终的图像作品。这个术语在本书中会经常出现。现在我们来查看一下渲染过程中出现的其他一些属于和操作。

### 1.2.1 变换 (Transformation) 和投影 (Projection)

图 1.4 所示是 BLOCK 示例程序的原始输出结果，显示的是用线条绘制的一个放置在一张桌子或一个平面上的立方体。通过变换 (Transformation)，或者说旋转这些点，并在它们之间绘制线段，我们就能在平面的 2D 屏幕上创造出一个 3D 世界的错觉。早期的飞行模拟器所使用的技术不过如此。

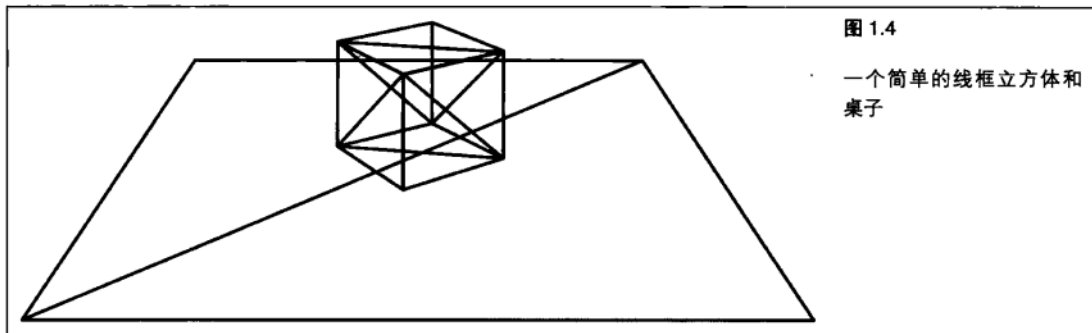


图 1.4

一个简单的线框立方体和桌子

这些点本身叫做顶点 (Vertices，单数为 Vertex)，它们能够通过一种称为变换矩阵 (Transformation Matrix) 的数学结构进行旋转 (本书第 4 章将详细讲解变换矩阵相关内容)。另外还有一种矩阵叫做投影矩阵 (Projection Matrix)，用于将 3D 坐标转换成二维屏幕坐标，实际的线条也将在二维屏幕坐标上进行绘制。

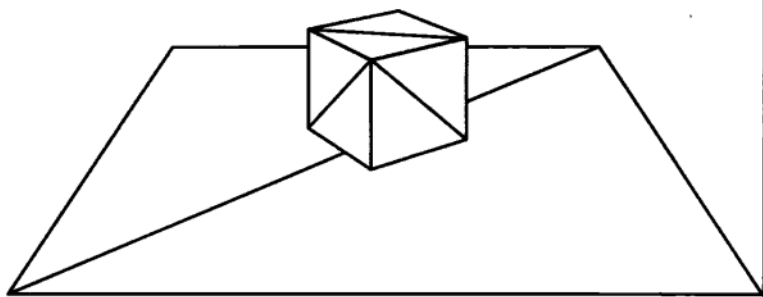
### 1.2.2 光栅化 (Rasterization)

实际绘制或填充每个定点之间的像素形成线段就叫做光栅化 (Rasterization)。我们可以通过隐藏表面消除 (Hidden Surface Removal) 来进一步澄清 3D 设计意图。图 1.5 所示显示了在 BLOCK 示例程序

第一次按空格键后的输出。虽然使用的仍然是点和线段，但是一个放置在桌面上的正方体的错觉却更加逼真了。

图 1.5

隐藏固体几何体的背面能够加强  
3D 错觉



虽然用线段绘图【也称做线框渲染 (Wireframe Rendering)】也有它的用处，但在大多数情况下我们并不是用线段，而是用实心三角形进行渲染。像线段一样，三角形和多边形也会被光栅化或填充。早期的图形硬件能够用纯色对三角形进行填充，但正如图 1.6 所示的那样，这样做并不能增强 3D 错觉。早期的游戏和模拟技术可能会在相邻的多边形上采用不同的纯色，这确实有所帮助，但却不能令人信服地对现实进行模拟。

图 1.6

使用纯色对几何图形  
进行填充的方法几乎  
没什么效果



### 1.2.3 着色

在图 1.7 (在运行 BLOCK 示例程序时再次按空格键) 中展示了着色 (Shading) 的效果。通过沿着表面 (在顶点之间) 改变颜色值，能够轻松创建光线照射在一个红色立方体上的效果。

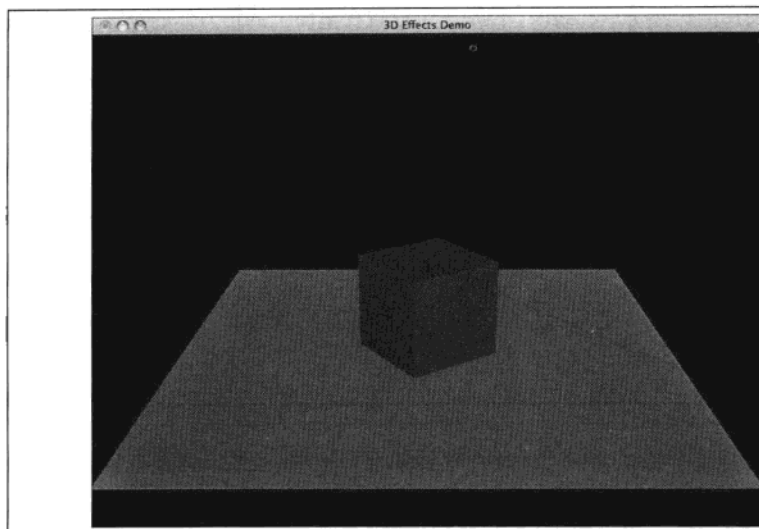


图 1.7

对表面进行着色能够创建  
光照错觉

光照和着色在 3D 图形专业领域占据了非常大的比重，并且有专门论述它们的书籍。另一方面，着色器（Shader）则是在图形硬件上执行的单独程序，用来处理顶点和执行光栅化任务。

## 1.2.4 纹理贴图

接下来要介绍的硬件技术进步是纹理贴图（Texture Mapping）。一个纹理不过是一幅用来贴到三角形或多边形上的图片。正如我们在图 1.8 中看到的，这些纹理将渲染提高到了一个崭新的层次。

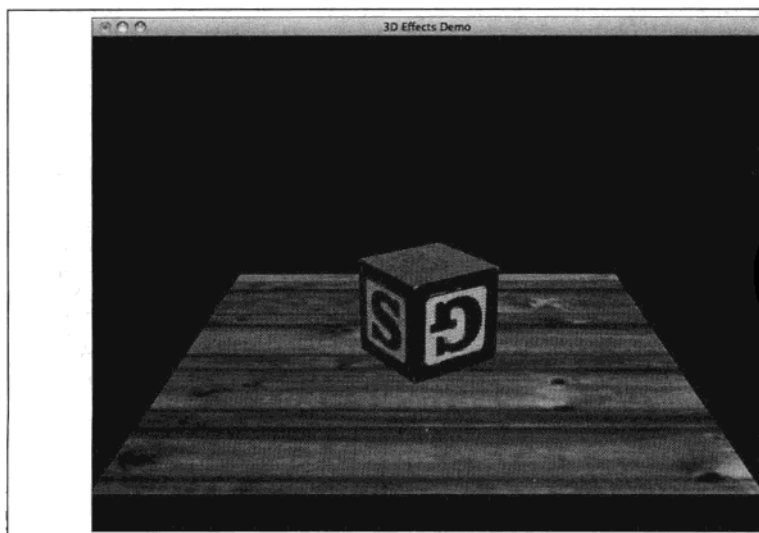


图 1.8

一个纹理堪比上千个  
三角形

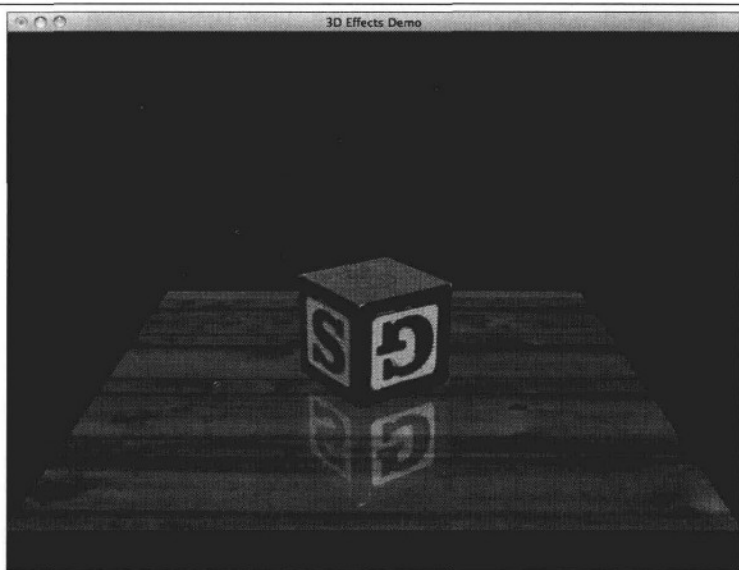
在如今的硬件上，纹理是快捷有效的，而一个纹理所能再现的表面如果用三角形来实现的话，可能需要几千甚至几百万个。

### 1.2.5 混合

最后,图 1.9 所示展示了混合 (Blending) 的效果。混合时我们能够将不同的颜色混在一起。我们首先上下颠倒地绘制这个立方体,然后再在它上面绘制地板并与它进行混合,在绘制正常方向的立方体,就能获得这种反射效果。我们确实“透过”地板看到了下面颠倒的立方体。大脑告诉我们,“哦……这是个倒影”。我们也可以应用混合使物体看起来透明。实际上,在图 1.9 中真正看到的倒立立方体其实是“透过”地板看到的。

图 1.9

使用混合来创建反射效果



### 1.2.6 将点连接起来

总而言之,上述内容大概就是所谓的计算机图形了。实心 3D 几何体无非是将顶点间的点连接起来,然后对三角形进行光栅化而使对象变得有实体。变换、着色、纹理与混合——我们在电影、电视游戏、医疗或商业应用中看到的任何计算机渲染场景都无非是灵活地运用这 4 种技术而产生的。

## 1.3 3D 图形的常见用途

在现代计算机应用程序中,三维图形具有广泛的应用。实时 3D 图形的应用范围包括交互式游戏和模拟以及数据的可视化显示(供科学、医学或商业应用)。高端 3D 图形在电影以及技术和教育出版物中也具有广泛的应用。

### 1.3.1 实时 3D

如前面所述, 实时 3D 图形是指活动的并与用户进行交互的图形。实时 3D 图形最早的用途之一是军事飞行模拟器。即使到了今天, 飞行模拟器仍然为许多业余爱好者所热衷。图 1.10 所示显示了一个流行的飞行模拟器的屏幕截图, 它使用 OpenGL 进行 3D 渲染 (www.x-plane.com)。



图 1.10

一个基于 OpenGL 的飞行模拟器, 由 x-plane.com 提供

在个人计算机领域, 3D 图形的应用几乎没有止境。目前最为常见的用途或许是计算机游戏。今天, 几乎所有发行的游戏都要求 PC 装有 3D 图形卡才能够运行。虽然 3D 图形在科学视觉和工程应用中也非常流行, 但价廉物美的 3D 硬件的大量涌现使得这些应用空前流行。商业应用程序也利用了 3D 硬件的新功能, 引入了越来越复杂的商业图形和数据库挖掘可视化技术。即使现代的 GUI 也受到了它的影响, 开始利用 3D 硬件的功能。例如, 新的 Macintosh OS X 使用 OpenGL 对所有窗口和控件进行渲染, 从而创建了功能强大、引人入胜的可视化界面。

图 1.11 至图 1.15 所示展示了现代个人计算机上无数 3D 应用中的一些例子。这些图像绝大多数都是用 OpenGL 实时渲染的。

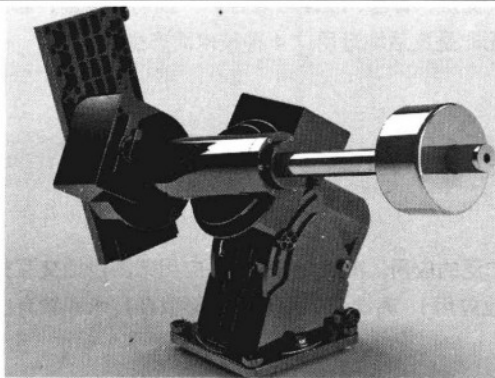


图 1.11

用于计算机辅助设计 (CAD) 的 3D 图形  
(图像由 Software Bisque 提供)

图 1.12

用于建筑或民用计划 ( 图像由  
Real 3D Inc.提供 ) 的 3D 图形



图 1.13

用于医学图像应用 ( Kitware 提供  
的 VolView ) 的 3D 图形

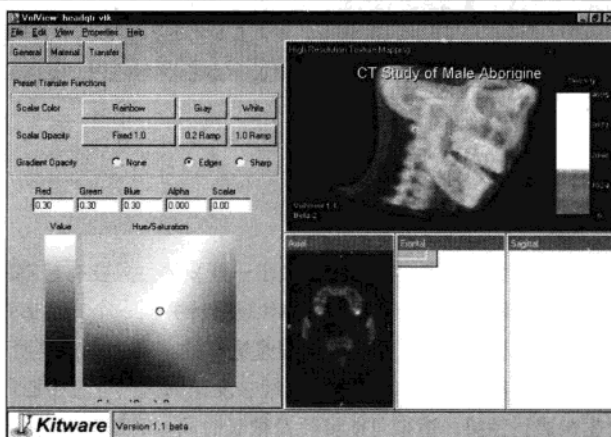


图 1.14

用于科学视觉的 3D 图形 ( 图像由  
Software Bisque 提供 )

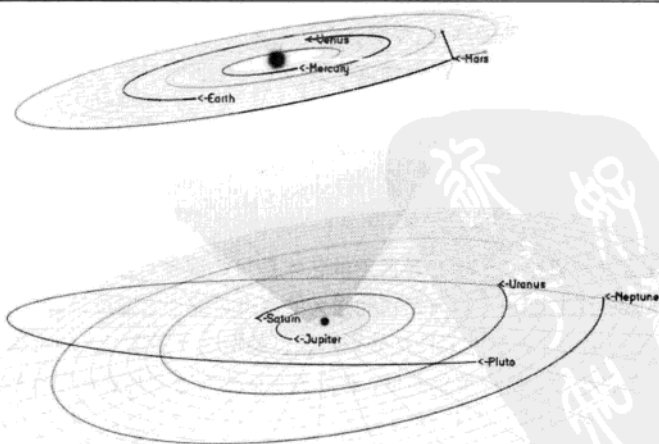




图 1.15

用于娱乐(取自 Outrage Entertainment Inc. 的 Descent 3)的 3D 图形

### 1.3.2 非实时 3D

在实时 3D 应用中,我们常常需要做出一些妥协。只要有足够的处理时间,我们可以创建更高质量的 3D 图形。在一般情况下,我们设计模型和场景,并用一个光线追踪器或扫描线渲染器来处理这些定义,产生高质量的 3D 图像。典型的处理过程是这样的:一个建模应用程序使用实时 3D 图形与艺术家进行交互,创建具体的内容;然后,它所创建的帧被发送到另一个应用程序(光线追踪器离线渲染器)或子程序,由它们对图像进行渲染,渲染可能要耗费很长时间。例如,在一台非常快速的计算机上,为一部电影(例如 toy Story 或 Shrek)渲染一个单独的帧可能需要耗费几个小时。渲染并保存成千上万个帧的过程生成了一个可以回放的动画序列。尽管这个动画序列在回放时看上去像是实时的,但它的内容却不是交互性的。因此,它并不是实时的,而是预渲染的。

### 1.3.3 着色器

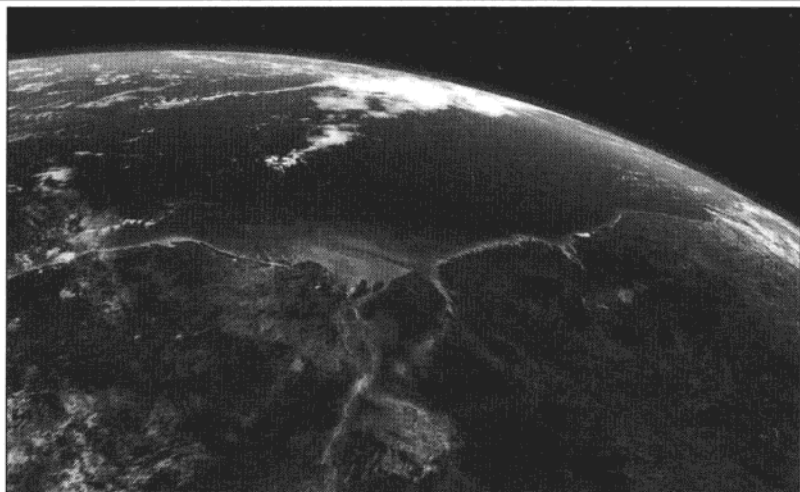
在实时计算机图形中,最前沿的艺术是可编程着色器(Programmable Shading)。今天的图形卡不再是低能的渲染芯片了,而是功能强大的高度可编程的渲染计算机。类似 CPU(中央处理单元)的术语 GPU 也应运而生,它代表图形处理单元,特指当今图形卡上的可编程芯片。它们是高度并行的,并且具有非常快的速度。同样重要的是,程序员可以进行重新配置图形卡的工作方式,几乎可以实现任何可以想像得到的特殊效果。

每年,基于着色器的图形硬件不断侵占传统上由高端光线追踪器和前面所提到的软件渲染工具所完成的任务。图 1.16 所示展示了 Software Bisque 的 Seeker 太阳系模拟器上的一幅地球图像。这个

应用程序使用了一个自定义的 OpenGL 着色器,以每秒 60 幅的速率生成了一幅逼真的地球动态图像。它还包括了大气效果、太阳在水中的倒影,甚至背景中的星星。本书插页的彩图 1 显示了这张图的彩色版本。

图 1.16

着色器可以实现前所未有的实时逼真性(图像由 Software Bisque, Inc.提供)



## 1.4 3D 编程的基本原则

现在,我们对实时 3D 的基本概念已经有了相当程度的认识。我们讨论了一些术语以及 PC 上的一些示例应用程序。那么,如何在自己的计算机上创建这样的图像呢?好吧,这正是本书剩余部分的任务所在。不过,读者还需要知道一些基础知识,这正是我们接下来将要讨论的。

### 1.4.1 并非工具包

OpenGL 基本上是一种底层渲染 API (应用程序接口)。我们不能告诉它“在什么地方绘制什么”——我们需要自己动手,通过载入三角形,应用必要的变换和正确的纹理、着色器并在必要时应用混合模式来组合一个模型。这使我们能够进行大量的底层控制。与使用高层工具包相比,使用 OpenGL 这样的底层 API 的动人之处在于,我们不能仅仅是重现许许多多的标准 3D 渲染算法,我们可以创造自己的算法,甚至可以发现一些新的捷径、性能技巧和艺术视觉技术。

### 1.4.2 坐标系统

现在,让我们考虑如何在三维中对物体进行描述。在指定一个物体的位置和大小之前,需要一个参考

帧对它进行测量和定位。当我们在一个简单的平面计算机屏幕上绘制点和线时，我们根据行和列指定一个位置。例如，标准的 VGA 屏幕从左向右为 640 个像素，自上而下为 480 个像素。为了在屏幕的中间指定一个位置，可以指定 (320, 240) 的点，也就是距离屏幕左边 320 个像素、距离屏幕顶端 240 个像素的位置。

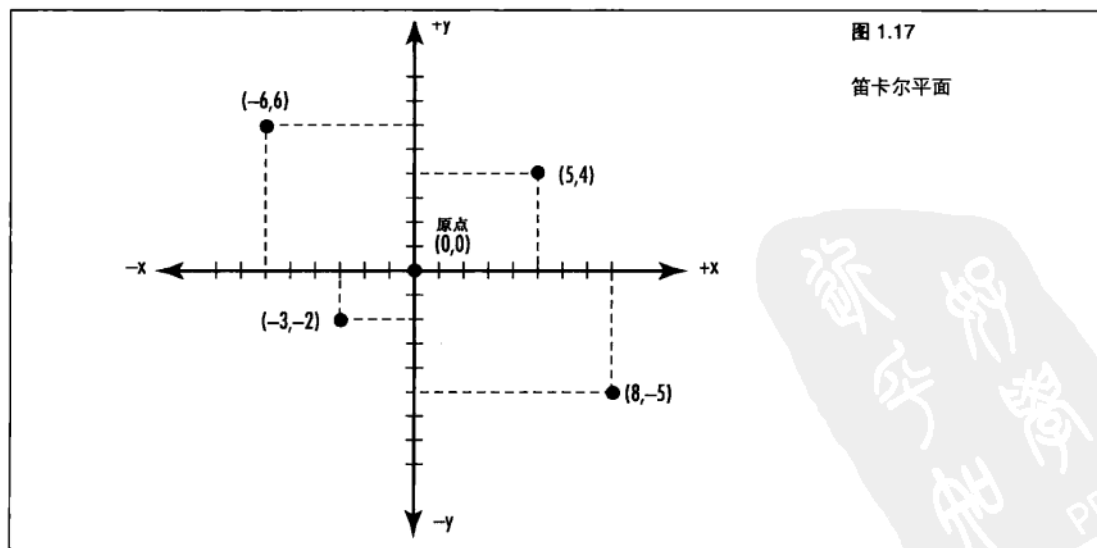
在 OpenGL 或几乎所有的 3D API 中创建一个用于绘图的窗口时，必须指定希望使用的坐标系统以及指定的坐标如何映射到实际的屏幕像素。首先，我们讨论在二维绘图中应该怎样做，然后把这个原则扩展到三维图形中。

## 2D 笛卡尔坐标

在二维绘图中，最为常用的坐标系统是笛卡尔坐标系统。笛卡尔坐标由一个  $x$  坐标和一个  $y$  坐标构成。 $x$  坐标测量水平方向的位置，而  $y$  坐标则测量垂直方向的位置。

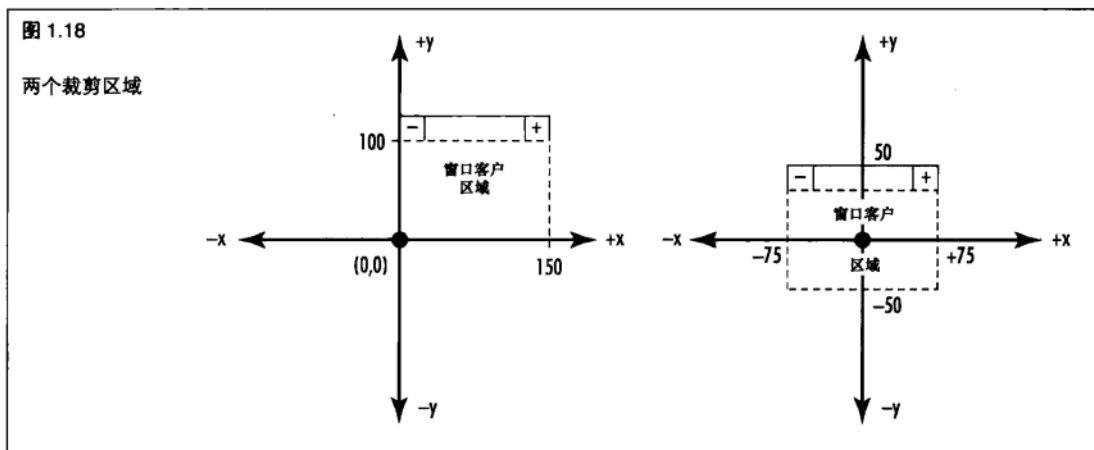
笛卡尔坐标系统的原点 (Origin) 是 ( $x=0$ ,  $y=0$ )。笛卡尔坐标用括号内的一个坐标对来表示，第一个是  $x$  坐标，第二个是  $y$  坐标，中间由一个逗号分隔。例如，原点就写成 (0, 0)。图 1.17 所示描述了二维的笛卡尔坐标系统。带刻度的  $x$  和  $y$  线被称为“轴”，可以从负无穷延伸到正无穷。这张图是我们在学校时经常使用的真实笛卡尔坐标系统。今天，当我们在绘图时指定坐标系统，不同的窗口映射模式可能会导致坐标的解释不一致。在本书后面章节，我们将看到如何使用不同的方式把真实的坐标空间映射到窗口坐标。

$x$  轴和  $y$  轴是垂直的 (直角相交)，它们共同定义了一个  $xy$  平面。简而言之，平面就是指一个扁平的表面。在任何坐标系统中，两条轴 (或两条线) 如果以直角相交，那么它们就定义了一个平面。很显然，如果一个系统只有两个轴，那么就只有一个平面可以用来绘图。



## 坐标裁剪

窗口是以像素为单位进行度量的。开始在窗口中绘制点、线和形状之前，必须告诉 OpenGL 如何把指定的坐标对翻译为屏幕坐标。我们可以通过指定占据窗口的笛卡尔空间区域完成这个任务，这个区域称为裁剪区域。在二维空间中，裁剪区域就是窗口内部最小和最大的  $x$  和  $y$  值。另一种方法是根据窗口指定原点的位置。图 1.22 所示显示了两种常见的裁剪区域。



在第一个例子中，也就是图 1.18 所示的左侧，窗口  $x$  坐标的范围自左向右为 0 到 +150， $y$  坐标的范围从上而下为 0 到 +100，屏幕正中的点用  $(75, 50)$  来表示。在第二个例子所示的裁剪区域中， $x$  坐标的范围自左向右为 -75 到 +75， $y$  坐标的范围从上而下为 -50 到 +50。在这个例子中，屏幕正中的点就是原点  $(0, 0)$ 。我们还可以使用 OpenGL 函数（或用于 GDI 绘图的普通 Windows 函数）上下反转或左右反转坐标系。事实上，在 Windows 窗口的默认映射中，坐标  $y$  的值始终为正，并且从上而下递增。这种默认的映射模式在自上而下绘制文本时非常有用，但在绘制图形时则显得不太方便。

### 视口：把绘图坐标映射到窗口坐标

裁剪区域的宽度和高度很少正好与窗口的宽度和高度（以像素为单位）相匹配。因此，坐标系统必须从逻辑笛卡尔坐标映射到物理屏幕像素坐标。这个映射是通过一种叫做视口（Viewport）的设置来指定的。视口就是窗口内部用于绘制裁剪区域的客户区域。视口简单地把裁剪区域映射到窗口中的一个区域。通常，视口被定义为整个窗口，但这并非严格必须的。例如，我们可能只希望在窗口的下半部分进行绘图。

图 1.19 所示是个很大的窗口，其大小为  $300 \times 200$  像素，它的视口被定义为整个用户区域。如果这个窗口的裁剪区域被设置为沿  $x$  轴 0 至 150，沿  $y$  轴 0 至 100，我们所看到这个窗口的逻辑坐标将被映射到一个更大的屏幕坐标系统中。逻辑坐标系统的每个增量将与窗口物理坐标系统（像素）的两个增量相匹配。

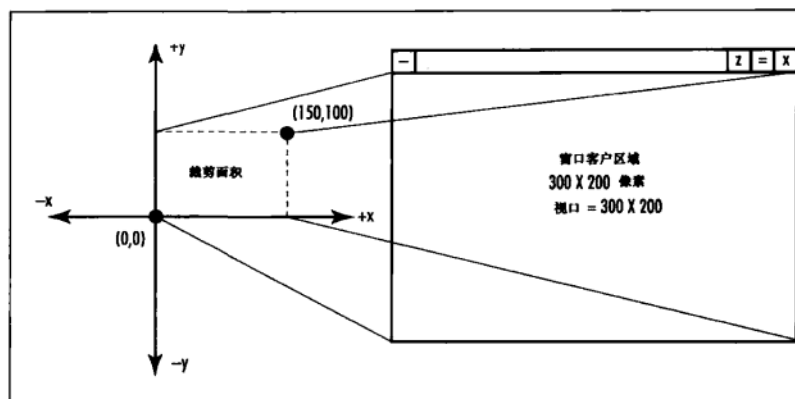


图 1.19

视口被定义为裁剪区域大小的两倍

与此形成对照的是，图 1.20 所示显示了一个与裁剪区域相匹配的视口。我们所看到的这个窗口仍然是  $300 \times 200$  像素。但是，现在可视区域将占据窗口的左下部分。

我们可以使用视口来缩小或放大窗口中的图像，也可以通过把视口设置为大于窗口的用户区域，从而只显示裁剪区域的一部分。

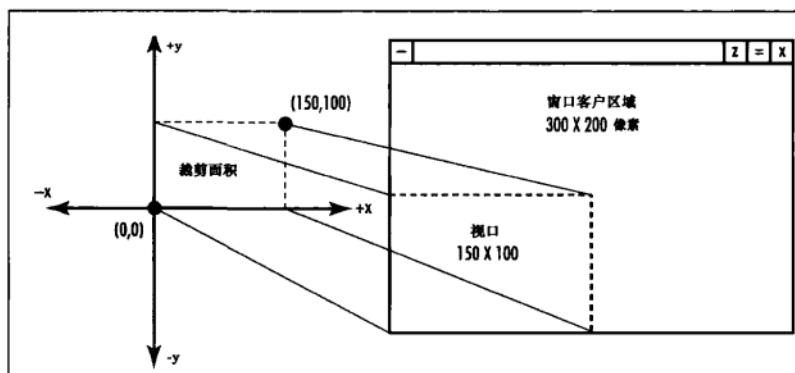


图 1.20

视口被定义为与裁剪区域相同的大小

## 顶点——空间中的一个位置

在 2D 和 3D 中，当我们绘制一个物体时，实际上都是用一些更小的称为图元 (Primitives) 的形状来组成这个物体。图元是一维或二维的实体或表面，如点、直线和多边形 (平面多边的形状)。在 3D 空间中，我们把图元组合在一起创建 3D 物体。例如一个三维立方体是由 6 个二维的正方形组成，每个正方形代表一个独立的面。正方形 (或其他任何图元) 的每个角称为顶点 (Vertex)。这些顶点就在 3D 空间中指定了一个特定的坐标。顶点其实也就是 2D 或 3D 空间中的一个坐标。创建实体 3D 几何图形其实不过就是一种连线游戏罢了。我们将在第 3 章讨论所有的 OpenGL 图元以及如何使用它们。

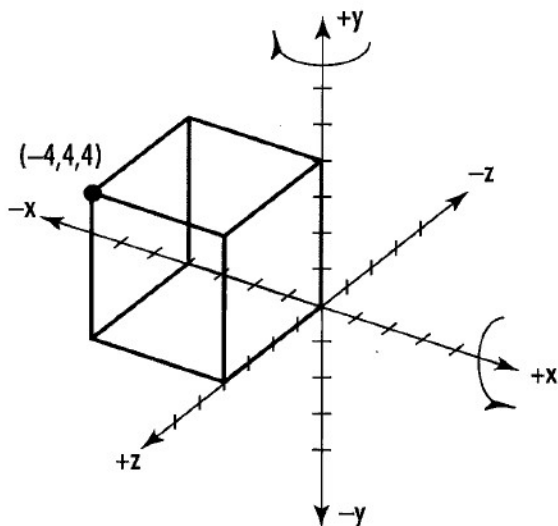
## 3D 笛卡尔坐标

现在，我们把二维坐标系统扩展到三维空间中，并增加深度分量。图 1.21 所示的笛卡尔坐标系统增

加了一个新的轴： $z$ 轴。 $z$ 轴同时垂直于 $x$ 轴和 $y$ 轴。它代表了一条从屏幕的中心朝向读者的直线（我们已经旋转了这个坐标系统的视角，把 $y$ 轴向左旋转，把 $x$ 轴向下和后旋转。否则， $z$ 轴将直接面向我们，无法看到）。现在，我们用3个坐标 $(x, y, z)$ 来指定三维空间中的一个位置。例如，图 1.21 所示显示了一个点 $(-4, 4, 4)$ 。

图 1.21

三维笛卡尔坐标



### 1.4.3 投影：从 3D 到 2D

我们已经知道如何在 3D 空间中使用笛卡尔坐标来表示位置。但是，不管我们觉得自己的眼睛所看到的三维图像有多么真实，屏幕上的像素实际上只是二维的。那么 OpenGL 是如何把这些笛卡尔坐标翻译为可以在屏幕上绘图的二维坐标呢？简而言之，答案就是“三角法和简单的矩阵操纵”。简单？事实上或许并非如此，但是如果我们花很长的篇幅来讨论其中的概念，我们很可能会失去很多对这些细节不感兴趣的读者。而且，如果读者已经忘了大学里所学习的线性代数，那么他们可能无法理解这种“简单”的技巧。在第 4 章，我们将对此稍作讨论。至于更深入的讨论，读者可以参考附录 A “更多阅读建议”中的参考部分。幸运的是，当我们使用 OpenGL 创建图形时，并不需要对数学有深入的理解。但是，我们在这方面的造诣越深，能够利用 OpenGL 所发挥的威力也就越大。

我们真正需要理解的第一个概念称为投影（Projection）。用于创建几何图形的 3D 坐标将投影到一个 2D 表面（窗口背景）。这有点像是在一块玻璃后面用一支黑色的笔描绘一些物体的外形。当物体消失或者移动玻璃时，我们仍然可以看到这个边缘带角的物体的外形，在图 1.22 中，背景中的一座房子被描绘到一块扁平的玻璃上。通过指定投影，我们可以指定在窗口中显示的视景物（Viewing Volume），并指定如何对它进行变换。

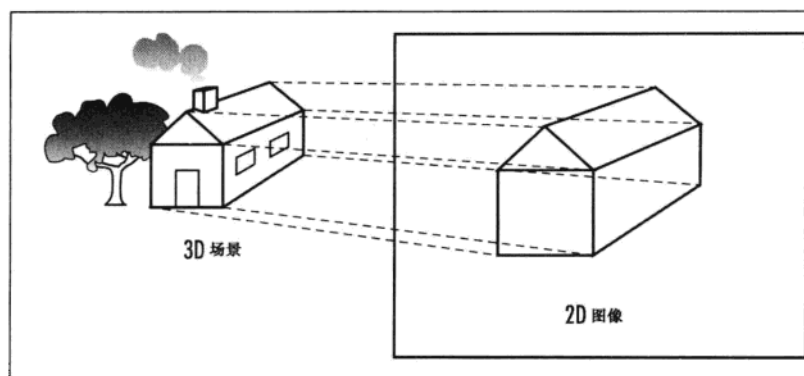


图 1.22

3D 图像投影到 2D 表面

## 正投影

在 OpenGL 中, 绝大多数情况下, 我们所关心的是两种主要类型的投影。第一种称为正投影 (Orthographic Projection) 或平行投影。使用这种投影时, 需要指定一个正方形或长方形的视景体。视景体之外的任何物体都不会被绘制。而且, 所有实际大小相同的物体在屏幕上都具有相同的大小, 不管它们是远是近。这种类型的投影 (如图 1.23 所示) 最常用于建筑设计、计算机辅助设计或 2D 图形中。此外, 在 3D 图形场景中, 我们也常常需要使用正投影, 在场景的顶部添加文本或者 2D 覆盖图。

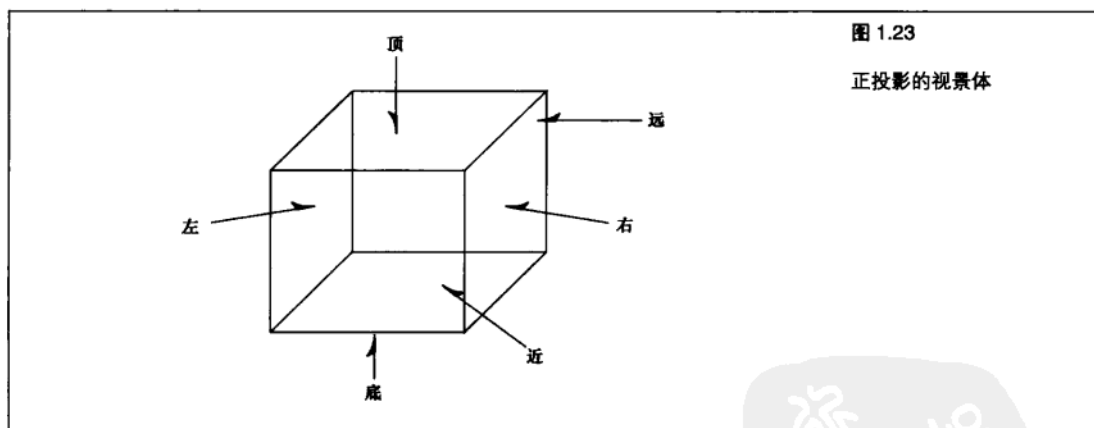


图 1.23

正投影的视景体

我们还可以在正投影中通过指定远、近、左、右、顶和底裁剪平面来指定视景体。在这个视景体中出现的物体和图形将被投影 (考虑它们的方向) 到一个在屏幕上出现的 2D 图像。

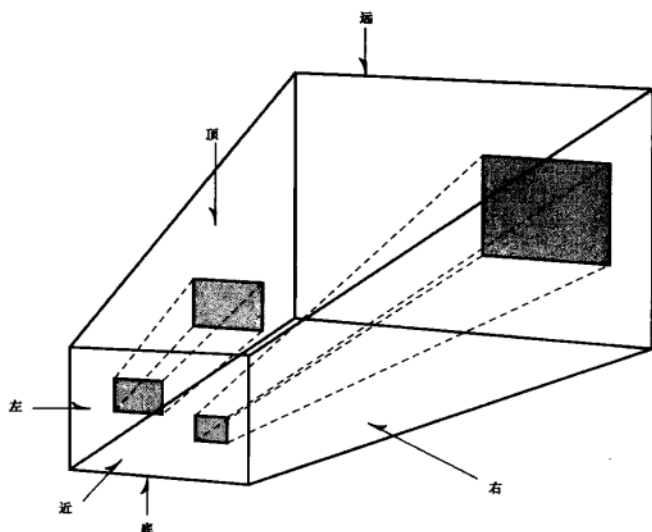
## 透视投影

第二种投影是透视投影 (Perspective Projection), 它更为常见。在这种投影中, 远处的物体看上去比近处的物体更小一些。它的视景体 (如图 1.24 所示) 看上去有点像一个顶部被削平的金字塔。剩下的这个形状称为平截头体 (Frustum)。靠近视景体前面的物体看上去比较接近它们的原始大小。但是,

当靠近视景体后部的物体被投影到视景体的前部时，它们看上去就显得比较小。在模拟和 3D 动画中，这种投影能够获得最大程度的逼真感。

图 1.24

透视投影的视景体（平截头体）



## 1.5 总结

在本章，我们介绍了 3D 图像的基础知识。读者可以看到，我们实际上需要一个物体的两幅图像（从不同角度）才能感知真正的三维空间。读者还可以发现，我们可以通过透视、隐藏直线消除、颜色、着色和其他技巧来创建深度幻觉。在 2D 和 3D 绘图中，我们引入了笛卡尔坐标系，并且学习了 OpenGL 所使用的两种方法把三维图像投影到二维屏幕。我们有意省略了 OpenGL 如何实际创建这些效果的细节。在接下来的章节中，我们将学习如何利用这些技巧，最大限度地发挥 OpenGL 的威力。在随书发行的示例代码中，读者可以找到一个程序，它显示了本章所讨论的一些 3D 效果。在 BLOCK 程序中，点击空格键可以把一个线框立方体逐渐完善为一个完整的带光照和纹理贴图的立方体，并且还带有光滑表面上的倒影。此时，我们可能还不理解这些代码，但它们做了一个强有力的展示。当读者学完本书以后可以重新回顾这个例子并对它加以改进，甚至可以自己重新编写。

## 第2章 入门指南

作者: Richard S. Wright, Jr.

### 本章内容

- ✦ OpenGL 的发展历程和未来趋势
- ✦ 扩展机制 (Extension Mechanism) 如何工作, 以及它为什么很重要
- ✦ 核心框架和“不鼓励使用”的功能
- ✦ 如何检测 OpenGL 编程错误
- ✦ 如何向 OpenGL 传递性能提示 (Hint)
- ✦ 如何获得一个基本项目并进入 Visual C++ 或 Xcode
- ✦ 如何使用在一个基本编程框架中使用 GLUT

既然我们已经介绍了 3D 图形的基本术语以及隐藏在 3D 图形背后的思路, 现在就可以进入正题了。在使用 OpenGL 之前, 我们需要讨论什么是 OpenGL 以及什么不是 OpenGL, 使读者能够同时了解这种 API 的功能和限制。本章概括地描述了 OpenGL 的操作, 并介绍了如何建立 3D 艺术品的渲染框架。

### 2.1 什么是 OpenGL?

严格地讲, OpenGL 被定义为“图形硬件的一种软件接口”。从本质上说, 它是一个 3D 图形和模型库, 具有高度的可移植性, 并且具有非常快的速度。使用 OpenGL, 可以创建优雅而漂亮的 3D 图像, 并且具有非常出色的视觉质量。使用 OpenGL 的最大优点是它的速度远远快于光线追踪器或软件渲染引擎。最初, 它使用 SGI (Silicon Graphics, Inc.) 精心开发和优化的算法。SGI 是一家久负盛名的公司, 在计算机图形和动画方面处于业界领先的地位。随着其他厂商不断奉献经验和智慧, 发展自己的高性能实现方案,

OpenGL 也得到了不断的发展。

OpenGL 并不像 C 和 C++ 那样的编程语言, 它更像一个 C 运行时的函数库, 提供了一些预先打包的功能。另一方面, OpenGL 规范包含 GLSL, 即 OpenGL 着色语言, 这实际上是一种非常类似于 C 语言的程序设计语言。但是, GLSL 并不会对应用程序流程和逻辑进行控制, 而是用于渲染操作。总体上看, 应用程序在 OpenGL 中编写情况并不像使用 OpenGL 那样多。事实上, 并不存在诸如“OpenGL 程序”之类的东西(如前所述, 着色器程序是个例外), 而是开发人员编写的程序“凑巧”使用了 OpenGL 作为应用程序编程接口(API)之一。就像我们可以使用 Windows API 来访问文件或 Internet 一样, 我们也可以使用 OpenGL 来创建实时的 3D 图形。

OpenGL 意在供那些专门为显示和处理 3D 图形而进行设计和优化的计算机硬件使用。纯软件的通用 OpenGL 实现也是可能的, Microsoft 所采用的实现方案就属于这种类型。使用纯软件的 OpenGL 实现, 渲染的速度可能会受到影响, 并且可能无法实现一些特殊的高级效果。但是, 使用软件实现意味着程序具有可以在范围极广的计算机系统上运行的潜力, 即使是那些并未安装全功能 3D 图形加速卡的系统。

OpenGL 具有多种用途, 范围涵盖从 CAD 工程和建筑应用程序到那些在恐怖电影中用于实现计算机生成鬼怪的建模程序等各种应用。将工业标准的 3D API 引入到拥有众多用户的操作系统如 Microsoft Windows 和 Macintosh OS X 产生的反响令人兴奋。随着硬件加速和高速 PC 微处理器越来越普及, 3D 图形已经成为消费者和商业应用程序的典型组成部分, 而不再局限于游戏和科学应用程序。

### 2.1.1 标准的演化

OpenGL 的前身是 SGI 公司的 IRIS GL。它最初是个 2D 图形函数库, 后来逐渐演化为由这家公司的高端 IRIS 图形工作站所使用的 3D 编程 API。这种计算机不仅是通用计算机, 它们具有专门经过优化的硬件, 用于显示复杂的图形。这种硬件提供了超快的矩阵变换能力(这是 3D 图形的先决条件)、硬件支持的深度缓冲以及一些其他功能。

但是在某些情况下, 出于支持陈旧系统的需要, 技术的发展常常受到束缚。IRIS GL 并不是一开始就设计为具有顶点风格的几何图形处理接口的。后来形势逐渐变得明朗, SGI 必须彻底改变才能继续发展。

OpenGL 就是 SGI 对 IRIS GL 的移植性进行改进和提高的结果。这个新的图形 API 不仅具有 GL 的功能, 而且是一个“开放”的标准。它的输入来自其他图形硬件厂商, 并且更容易应用到其他硬件平台和操作系统。OpenGL 是为了 3D 几何图形而完全重新设计的。

#### OpenGL ARB

如果只有某一家厂商控制标准, 那么这个标准就不是真正的开放式标准。SGI 当初的业务领域是高端的计算机图形。一旦已经处于某个行业的顶端, 你就会发现进一步成长的机会就很有有限了。SGI 认识到如果它能够做些事情来推动高端计算机图形硬件市场的成长, 那么这对公司也是件非常好的事情。一个得到众多厂商支持真正的开放式标准能使程序员更容易创建可以适应广泛平台的应用程序和内容。软件确实能够促进计算机的销售, 如果 SGI 希望卖出更多的计算机, 那么就需要在它的计算机上能够运行更多的

软件。其他厂商也意识到了这一点，这样 OpenGL 体系结构审核委员会（ARB, OpenGL Architecture Reiview Board）就诞生了。

SGI 最初控制了 OpenGL API 的许可，而 ARB 的创立者包括 SGI、DEC（Digital Equitment Corporation）、IBM、Intel 和 Microsoft。1992 年 7 月 1 日，OpenGL 规范 1.0 版正式出台。随着时间的推移，ARB 又陷于陆续增加一些新成员，其中有许多来自 PC 硬件社区。ARB 每隔 4 年召开一次会议，对规范进行维护和改善，并出台计划对 OpenGL 标准进行升级。

近年来，由于某些原因，SGI 的业务不断下滑，具体原因超出了本书的讨论范围。2006 年，实际上已经破产的 SGI 公司把对 OpenGL 标准的控制权从 ARB 移交给一个新的工作组：Khronos 小组（[www.khronos.org](http://www.khronos.org)）。Khronos 是一个由成员提供资金的行业协会，专注于开放媒体标准的创建和维护。大多数 ARB 成员已经成为 Khronos 的成员，因此这个变动并没有产生太大的波折。今天，Khronos 小组继续发展和升级 OpenGL 以及它的姊妹 API——OpenGL ES。我们将在第 16 章介绍 OpenGL ES。

OpenGL 以两种形式存在。第一种形式是“OpenGL 规范”，这个规范定义了行业标准，用非常完整和明确（这里用相似的词并不是偶然的）的术语描述了 OpenGL。它完整地定义了 OpenGL API、OpenGL 的整个状态机，以及各种特性是如何共同工作和运行的。然后像 ATI、NVIDA、Intel 或 Apple 这样的硬件厂商获取这个规范，并实现它。第二种形式就是 OpenGL 的实现，软件开发人员和顾客可以使它生成实时图形。例如，PC 上的软件驱动程序和图形卡就共同组成了一个 OpenGL 实现。

## · OpenGL 扩展机制

读者可能会认为，既然 OpenGL 是一种“标准”API，那么硬件提供商在竞争中只要考虑性能（可能还有视觉质量）因素就可以了。然而，3D 图形领域的竞争是非常激烈的，硬件提供商不仅在性能和质量方面拥有持久的创新力，在图形方法学和特效方面也是如此。OpenGL 允许提供商通过它的扩展机制进行创新。这种机制以两种方式运作。首先，提供商能够向 OpenGL API 中增加开发人员可用的新函数；其次，可以添加能够被已存在的 OpenGL 函数识别的标记（Token）或枚举（Enumerant）。

利用新的标记或枚举只需在项目中添加一个提供商支持的头文件（Header）。提供商必须在 OpenGL 工作组（Khronos 小组的一个下属机构）注册它们的扩展，以避免提供商使用其他提供商已经使用的值。有一个标准头文件 `glxext.h` 中包含了这些扩展，为我们提供了方便。

游戏需要为特定图形卡而进行重新编译的时代已经一去不返了。就像我们已经知道的，我们可以查到一个标识提供商和 OpenGL 驱动版本的字符串。确定一个扩展是否得到支持有两个步骤。首先，我们向 OpenGL 查询当前实现支持多少扩展。

```
GLint nNumExtensions;  
glGetIntegerv(GL_NUM_EXTENSIONS, &nNumExtensions);
```

然后，我们可以通过调用 `glGetStringi` 函数获取特定扩展的名称。这个函数将返回单个扩展的名称。例如，要在 Windows 中查询交换控制扩展，我们可以依次查阅所有扩展寻找我们需要的。一旦找到，我们要得到这个函数的函数指针并正确地调用它。

```
GLint nNum;
```

```

glGetIntegerv(GL_NUM_EXTENSIONS, &nNum);

for(GLint i = 0; i < nNum; i++)
    if(strcmp("WGL_EXT_swap_control", (const char *)glGetStringi(GL_EXTENSIONS, i)) ==
0)
    {
        wglSwapIntervalEXT =
            (PFNWGLSWAPINTERVAEXTPROC)wglGetProcAddress("wglSwapIntervalEXT");
        if(wglSwapIntervalEXT != NULL)
            wglSwapIntervalEXT(1);
    }

```

在 GLTools 库中有一个快捷工具包，稍后将对其进行讨论。

```
int gltIsExtSupported(const char *extension);
```

如果支持指定的扩展，那么这个函数返回 1；反之则返回 0。GLTools 库包含了一整套 OpenGL 帮助和实用工具函数，其中有很多在本书中自始至终都在使用。glttools.h 文件包含所有的函数的函数原型。

这个示例还展示了在 Windows 下如何获取一个指向新的 OpenGL 函数的指针。Windows 函数 wglGetProcAddress 返回一个指向 OpenGL 函数（扩展）名的指针。获取一个由于操作系统的不同而不同的扩展，这一主题在本书的第 3 部分将进行更详细的阐述。幸运的是，在 99% 的情况下我们使用一个叫做 GLEW 的快捷方式库就可以了，而我们将自动获取驱动程序支持功能的扩展函数指针。

## 这是谁的扩展

使用 OpenGL 扩展，我们可以在代码中规定代码路径来改善渲染表现和视觉质量，或者甚至可以添加只有特殊提供商的硬件才支持的特效。但是这些扩展是属于谁的？也就是说，哪家提供商创建一个特定扩展并为其提供技术支持？通常情况下我们只通过观察扩展名就能得出结论。每个扩展都有一个由 3 个字母组成的前缀，这个前缀标识了这个扩展的来源。表 2.1 展示了一些扩展识别的示例。

表 2.1 OpenGL 扩展识别示例

前 缀	提 供 商
SGI_	Silicon Graphics
ATI_	ATI Technologies
AMD_	Advanced Micro Devices
NV_	NVIDIA
IBM_	IBM
WGL_	Microsoft
EXT_	Cross-Vendor
ARB_	ARB Approved

一个提供商为另一个提供商的扩展提供支持的情况并不少见。例如，一些 NVIDIA 的扩展就在 ATI 硬件平台上得到支持并广泛流行。竞争提供商必须遵循原始提供商的规范（关于扩展如何工作的细节）。通常人们都认为扩展是好东西，扩展的 EXT\_前缀表明这个扩展（假定）是与特定提供商无关的，并且得

到众多实现的支持。

最后, 还有一些 ARB 承认的扩展。这些扩展的规范经过了 OpenGL ARB 的审核以及讨论。这些扩展常常意味着某些新技术或函数能够加入核心 OpenGL 规范前的最后步骤。

### 许可和一致

OpenGL 实现可以是软件库, 也就是对 OpenGL 函数调用做出响应、创建三维图像的软件函数库。OpenGL 实现也可以是一个用于完成三维图像渲染任务的硬件设备 (通常是显卡) 的驱动程序包。硬件实现比软件实现要快上许多倍。而且, 现在即使在廉价 PC 上这类硬件也已经非常普遍。

如果厂商希望创建并销售 OpenGL 实现, 首先必须从 Khronos 小组获得 OpenGL 许可。如果申请者是 PC 硬件厂商, Khronos 小组在批准许可时会顺带提供一个示例实现 (纯软件形式) 和一个设备驱动程序。然后, 厂商就可以据此创建经过优化的实现, 并且可以通过扩展提高产品的价值。在典型情况下, 厂商之间的竞争就是性能、图像质量和驱动程序稳定性的竞争。

此外, 厂商的实现必须通过 OpenGL 一致性测试。这些测试的设计目标就是保证实现方案是完整的 (包含所有必需的函数调用), 并且对于一组特定的函数, 这种实现所产生的 3D 渲染输出结果是可以接受的。

软件开发人员在使用 OpenGL 驱动程序时并不需要获得 OpenGL 许可或者支付任何费用。OpenGL 是得到操作系统原生支持的, 并且获得许可的驱动程序是由硬件厂商本身所提供的。

## 2.1.2 OpenGL 的未来

绝大多数公司认识到从长远而言, 竞争对于每家公司都是件好事, 因此它们都认可并支持行业标准, 甚至对行业标准作出贡献。Khronos 小组下属的体系结构审核委员会 (ARB) 如今已经非常壮大, 生气勃勃且充满活力。最近, OpenGL 规范的修订工作已经达到不到一年就能推出一个版本的速度。到本书编写时, OpenGL 最新的版本已经更新到了 3.3 和 4.0, 这两个版本都是在 2010 年游戏开发者大会上发布的。15 年以来, 加起来人们已经在各种类型的 OpenGL 技术、书籍、教程、示例代码和应用程序上花费了数百万年的人工。这种持续的动力将使 OpenGL 在可预见的未来保持广大应用程序和硬件平台首选 API 的地位。所有这些都使 OpenGL 获得良好的定位, 以充分利用未来 3D 图形技术创新成果。OpenGL 2.0 中加入的 OpenGL 着色语言, 使 OpenGL 显示出了长久的适应性, 能够满足不断发展的 3D 图形程序设计管线所带来的挑战。最后, OpenGL 是一种规范, 能够应用于各种各样程序设计范例。从 C/C++、Java 到 Visual Basic, 甚至 C# 这样的新兴程序设计语言, 现在都已经用来使用 OpenGL 创建 PC 游戏和应用程序。OpenGL 已经被广泛地接受和关注。

### OpenGL 与 Direct3D

就像政治上或宗教上的联盟一样, 对程序设计语言或 API 的选择经常在某种程度上是某些原因和情

感考虑。——“我就是这样成长的”或者“这就是最早学习的 API，我还是用它最顺手”。这当然是任何人会选择 Direct3D 而不选择 OpenGL 的唯一符合逻辑的原因。

如果读者是 3D 图形程序设计领域的新手，那么可能还不知道在两种互相竞争的标准 OpenGL 与 Direct3D 之间有一场“战争”。这是非常遗憾的，因为这两种标准都是可行的选择，而且它们都有各自的优点。人们经常将 OpenGL 与 DirectX 相比，这是不公平的。DirectX 是一个来自微软公司的游戏技术 SPI 族，其中包括 Direct3D 这种由微软公司为游戏程序涉及而开发的渲染 API。一个人可能更喜欢自己的汉堡而不是别人的牛排，但是拿别人的牛排和一家餐馆来比较是不公平的！实际上，大多数使用 OpenGL 的 Windows 游戏同时还使用 DirectX 的非渲染组件来更方便地进行声音的回放、游戏手柄控制和联网游戏等。

Direct3D 是微软公司专有的标准，被广泛地应用于 Windows 平台的游戏上，并且还有一些 Direct3D 的变体被用于 Xbox 游戏控制台平台和一些 Windows 移动设备上。在 Direct3D 发展的早期，这种 API 是非常不好用的，和 OpenGL 相比缺乏大量特性，并且存在一些固有的软件低效情况。微软公司采取了一些有争议的策略来帮助 Direct3D 成为 Windows 游戏程序设计的“标准”，这种情况持续了几年，被人们称为“API 战争”。在很多人看来，这场战争还在继续。公平地说，微软公司到现在为止已经与硬件提供商和软件提供商合作了十多年，现在 Direct3D 已经成为一种有用且有良好口碑的 API，并且在那些只对微软公司平台感兴趣的图形程序设计人员中非常流行。

但是，OpenGL 在 Windows 游戏开发人员中仍然非常流行，并且是那些制作非游戏 3D 应用程序（例如视觉模拟（Vis-sim）行业、内容创建工具、科学可视化和商业图形等）的软件开发人员的首选。在 OpenGL 和 Direct3D 之间“根据情感”的选择常常可以归结为喜欢或不喜欢 Direct3D 的面向对象 COM（组件对象模型）方法和 OpenGL 的状态机抽象化，或者仅仅是出于喜欢或不喜欢微软公司。在诸如 Mac OSX、iPhone、Linux（不仅仅是桌面系统，大多数手持智能电话设备使用的都是 UNIX 的变体）和 Sony 或任天堂游戏设备这样的非微软平台上，OpenGL 或者类似 OpenGL 的 API 则是实际上的标准。如果我们将整个 3D 图形产业看作一个整体，OpenGL 所占据的份额比 Direct3D 要大得多。

还有一些原因使我们可能选择 OpenGL 而不是 Direct3D。第一个原因是，OpenGL 是跨平台和可移植的，并且几乎现有的所有 3D 硬件设备都有对应的 OpenGL 驱动。如果读者对游戏感兴趣，那么就可以做一些市场调查。Windows 桌面系统在游戏产业中并没占据大部分份额。第二个原因是，OpenGL 是一个开放的标准，它能从所有领先的 3D 硬件提供商的知识和经验中受益。这些提供商必须合作，使 OpenGL 对开发人员而言是有吸引力和强大的，毕竟这些开发人员制作了促使人们购买他们硬件的软件。由于 OpenGL 是“到图形硬件的软件接口”，让软件提供商参与规范的演变是必要的。这就是人们可能选择 OpenGL 而不是 Direct3D 最后的也是最重要的原因——扩展机制。

扩展机制使硬件提供商不仅能在性能和图像质量方面进行竞争，同时还能在真正的技术革新上较高下。硬件提供商可以在硬件上加入新的特性，并且在他们愿意的时候通过 OpenGL 来公开这些新特性。他们不需要 ARB 的许可，不需要微软公司的许可，也不需要等待下一班的 OpenGL（或 Direct3D）发布。在 Direct3D 领域没有这些条条框框。微软公司通过代理来决定将什么加入 API 中，以及在某种程度上（有些情况应该说不公平的）影响硬件体系结构。最新和最优秀的硬件特性总是能够通过提供商 OpenGL 驱动程序和相关扩展来轻松获得应用。例如，当支持 DirectX 10 的硬件发布时，Windows 用户就需要使用 Windows Vista 才能享受应用了最新 DirectX 10 特性的游戏。但是，所有的新功能同时也在 OpenGL 上通

过扩展机制发布了，并且如果 Windows XP 用户的游戏使用 OpenGL，那么他们马上就能使用这些新功能，当然同时也能得到最新的硬件和驱动程序。许多年来，诸如 NVIDIA 或 ATI（现在是 AMD）这样的提供商都可以通过在 OpenGL 上编写的演示程序来展示他们最新的硬件革新技术，而他们也确实这样做了。仅此一点就使 OpenGL 在最新和最优秀的 3D 硬件新技术应用方面始终保持稍稍领先 Direct3D 的优势。

### “不鼓励使用”的功能

十多年来，OpenGL 标准通过在每个版本发布时加入新功能来不断演化着。新功能通常是通过扩展过程来进行审核的，在扩展过程中一些特性将作为提供商指定的或提供商联合扩展而添加进来，这些特性将被进一步完善并成为 ARB 扩展，最终将进入核心 API 规范中。在这个过程中从来没有哪些功能从 OpenGL 中被移除。这样就保证了对旧代码百分之百的向下兼容性，而且随着新硬件投入使用，现存应用程序只会运行得更快。开发人员也能够轻松地逐步升级代码，以在最新的渲染技术或性能强劲的新功能推出时充分利用它们，而不必重写已经完成或正在编写的代码。

然而，实际上这种过程也将到此为止。随着时间的流逝，GPU 和计算机体系结构已经发生了巨大的变革。15 年来一直保持性能上的权衡与工程上的妥协如今已再难适用。其结果就是，一些 OpenGL API 变得有些陈旧过时。很多提供商都首次开始寻求通过移除那些在现代代码中几乎不再使用，或性能远远低于最新技术的特性和功能来精简 OpenGL API 的规模。最终，ARB 决定以 OpenGL 3.0 为突破口，在 OpenGL 诞生以来首次抛弃一些负担，即过时的 OpenGL API。提供商仍然可以为了一些过时代码而继续支持 OpenGL 2.1 驱动程序，但定位于 OpenGL 3.0 或更新版本的最新应用程序应当抛弃旧的 API 函数和约定。这在当时看来确实是一个不错的决定。

### OpenGL 3.0

ARB 是由图形硬件提供商组成的，而提供商是有客户的，而且必须使这些客户满意。很多客户（软件开发人员）意识到这种将 OpenGL 2.1 作为古董束之高阁的模式实际上意味着一件事情，这就是这些驱动程序对于提供商来说将迅速降至低优先级，并且在新的硬件上不会很好地保存或更新，而这些软件开发者将被迫浪费在 OpenGL 上价值数百万美元的投资。最终他们在次序上达成了妥协，即在 OpenGL 3.0 中，不会真正移除任何功能，而是将这些功能标记为“不鼓励使用”。“不鼓励使用”的功能将仍然保留在驱动程序中，但它们是作为一种通告来提供的，通知软件提供商应该停止使用某些 OpenGL 特性并转向更新的和更现代的工作方式。据说在 OpenGL 3.1 中这些特性将被移除，或者说我们认为可能是这样。

### OpenGL 3.x

OpenGL 3.1 遇到了前所未有的苛刻要求。这种要求会让任何圆滑的政客都感到确实太过苛求。确实，所有“不鼓励使用”功能都从 OpenGL 核心规范中移除了，但是却引入了一个新的 OpenGL 扩展 GL\_ARB\_compatibility。很多正在寻求一个更合理 API 的软件提供商将这个扩展看作仅仅是“加入了所有我们承诺移除但却没有做到的‘不鼓励使用’ OpenGL 特性”。这意味着一个硬件提供商至少可以选择在 OpenGL 3.1 驱动程序中不包含任何“不鼓励使用”功能。但是，这种情况并没有发生。ARB 的

成员之一 NVIDIA 公开声明不会移除任何老旧功能。在某些种类应用程序（尤其是游戏）的开发人员开始指责这种行为的同时，我们应该客观地说，NVIDIA 或者其他硬件提供商还能怎么做呢？难道一个硬件提供商应该忽视它的客户而强制推行一种标准，仅仅是因为这家公司认为这样做最符合自己的利益吗？我们以前曾经遇到过这种情况。这样做几乎都得不到好的结果，而且没有人希望在 OpenGL 社区中发生这样的丑闻。

OpenGL 3.2 在这件事情上做得漂亮了些，它废除了这个扩展，取而代之地将 OpenGL 分成了核心框架和完整框架。核心框架规范将更加精简，并且不包含任何老旧的“不鼓励使用”功能。规范的一致性要求具有核心功能，但将兼容框架列为可选项。

事实上，那些“不鼓励使用”OpenGL 特性对于如今使用 OpenGL 的绝大多数开发人员来说要容易理解得多。这些特性中有很多可能会比最新的方法慢，但是它们更加容易使用，而且非常方便。工程师们都知道，在易用性、可实现性、可维护性和开发人员熟练程度，当然还有性能之间经常要做出权衡。性能并不是在所有类型的应用程序开发中都是绝对主要的考虑因素。兼容框架看起来还要存在相当长的时间。

### 只有核心

那么，这种情况会给我们带来什么影响呢？本书的第 4 版涵盖了 OpenGL 2.1 的内容，这个版本是允许选择使用着色器的固定管线的经典 OpenGL 实现。核心框架式则是它的最简形式，“只有着色器”。这样我们无论做什么，都需要编写一个着色器。这里没有内建的光照模式，没有方便的矩阵堆栈，没有简单的纹理应用程序，也没有轻松编写代码的立即模式来传输顶点数据。实际上，一些几何图元也被削减掉了。难怪众多开发人员并不急于让他们的代码“现代化”了。让事情变得更糟的是，迄今为止的大多数教程和图书都专注于展示如何从固定管线移植到着色器，似乎这是唯一的方式。这当然就意味着对于新入行的 OpenGL 程序员来说掌握 OpenGL 最简单的方式就是先从固定功能开始，然后再向着色器过渡。只是这并不是促进新的 OpenGL 核心框架应用的生产方式，也不是本书所采用的方式。

## 2.2 使用 OpenGL

OpenGL 是一种过程性而不是描述性的图形 API。事实上，程序员并不需要描述场景和它的外观，而是事先确定一些操作步骤，实现一定的外观或效果。这些步骤需要调用许多 OpenGL 命令。这些命令可以在三维空间中绘制各种图元，例如点、直线和三角形等。另外，OpenGL 还支持纹理贴图、混合、透明、动画以及其它许多特殊的效果和功能。关于这些如何实现的具体内容将在第 3 章详细介绍。本章主要关注如何建立并运行 OpenGL 项目。

OpenGL 并不包含任何负责窗口管理、用户交互或文件 I/O 的函数。每个宿主环境（例如 Mac OS X 或 Microsoft Windows）都提供了一些函数实现这些功能，并负责实现一些方法，向 OpenGL 移交窗口绘图的控制。我们无法使用类似“OpenGL 文件格式”这样的东西来表示模型或虚拟环境，因为它们并不存在。程序员构造这些环境以适合自己的高层需要，然后使用底层的 OpenGL 命令精心地对它们进行编程。

## 2.2.1 支持阵容

任何计算机程序都必须包含一些除渲染操作以外的其他东西才能使用。用户必须通过某种方式来使用键盘、鼠标、游戏手柄或其他一些输入机制来与程序进行互动。此外，必须打开并保持窗口（在大多数但并非全部操作系统中是如此），找到并载入文件等。C 和 C++ 都是良好的可移植程序设计语言，它们如今年在大多数平台上都适用。但是，在典型的程序中，编程语言要使用 API 来完成大量工作。OpenGL 就是一个 API 的例子，并且还是一个适用于大多数现代计算机平台的可移植 API。遗憾的是，与操作系统连接意味着与用户或在屏幕中的管理窗口的互动大多数情况下常常是由不可移植的操作系统特定 API 完成的。

### GLUT

首先出现的是 AUX，也就是 OpenGL 辅助函数库。AUX 函数库的目标帮助人们学习和编写 OpenGL 程序，而不必为任何平台特定环境的细枝末节而分神，不必顾虑所使用的是 UNIX、Windows 还是其他平台。如果使用 AUX，我们不是编写“最终”代码，它更像是一个预备阶段，对自己的想法进行测试。由于缺乏基本的 GUI 特性，这就限制了这个函数库在创建实用的应用程序方面的应用。

在跨平台的示例程序和演示程序中，AUX 渐渐为 GLUT 函数库所取代。GLUT 代表 OpenGL 实用工具箱（OpenGL utility toolkit，不要与标准的 GLU——OpenGL utility library，即 OpenGL 实用库混淆）。Mark Kilgard 在 SGI 时编写了 GLUT，把它作为 AUX 函数库的一个功能更强的替代品，并添加了一些 GUI 功能，至少使示例程序在 X Window 下显得更为实用。它的改进包括使用了弹出式菜单、增加了对其他窗口的管理，甚至提供了对操纵杆的支持。GLUT 并不是一个公众领域的产品，但它是免费的，并且可以自由地进行重新发布。GLUT 在绝大多数 UNIX 系统（包括 Linux）中都得到了支持，并且得到了 Max OS X 的本地支持，Apple 对这个函数库进行了维护和扩展。在 Windows 中，GLUT 的开发已经中断。由于 GLUT 最初并不是作为一种开放源代码的软件，因此一种新的 GLUT 实现 freeglut 已经崛起并取代了它的位置。在本书中，所有基于 GLUT 的 Windows 示例程序都利用了 freeglut 函数库。读者也可以通过本书的网站下载这个函数库。

在绝大多数情况下，本书使用 GLUT 作为编程框架。这出于两个目的。首先，它可以使本书面向更广的读者群。只要稍下功夫，有经验的 Windows、Linux 或 Mac 程序员应该很容易在编程环境中设置 GLUT，并且顺序地创建本书的绝大多数示例程序。第二个目的是使用 GLUT 可以使读者不必了解任何特定平台的基本 GUI 编程。尽管我们解释了一些基本的 GUI 概念，但本书并不是一本讲述 GUI 编程的书籍，而是专门讲述 OpenGL 的。把 OpenGL API 的范围限制在 GLUT，Windows/Mac/Linux 新手也更容易上手。

商业应用程序的所有功能不可能全部包含在 3D 绘图代码之中。虽然 GLUT 确实包含一些有限的 GUI 功能，但是非常简单和精简，就像 GUI 的工具包一样，因此我们不能依赖 GLUT 函数库来完成应用程序的所有任务。然而，GLUT 函数具有非常优秀的学习和演示功能，并且隐藏了像窗口创建和 OpenGL 环境初始化等平台特定的细节。即使是经验丰富的程序员，把 3D 图形集成到完整的应用程序之前，使用 GLUT 函数来整理 3D 图形代码也是非常方便的。

## GLEW

正如前面所提到的, OpenGL API 主要通过扩展机制来发展。这种扩展机制能够用来获得指向任何加入 OpenGL 1.0 之后任何版本核心的 OpenGL 函数的函数指针。有一个实现 OpenGL 3.3 API 完全存取的简单方法, 就是使用一个自动初始化所有新函数指针并包含所需类型定义、常量和枚举值的扩展加载库。不止一种这样的扩展加载库可供选择, 其中一种维护最好的开源库是 GLEW。通过驱动程序使用这种库来初始化全部可用的 OpenGL 功能并不太容易。我们需要在项目中添加一个单独的 C 源文件及头文件, 并且在程序启动时调用一个单独的初始化函数。稍后开始编写我们的第一个 OpenGL 程序时将讨论相关细节。为了使事情更简单, GLEW 被预先封装在了 GLTools 库中。实际上, GLTools 库就是基于 GLEW 库的。

## GLTools

每一位工匠都有一个工具箱, 里面装满了自己喜爱的工具, 程序员也是如此。有一些有用并且可重用的函数, 所有程序员在编写几乎所有 OpenGL 时都要用到它们。GLTools 是在本书第 3 版出现的。随着时间的流逝, 这个库已经逐渐发展起来, 并提供许多快捷方式和便捷的工具, 就像过去的 OpenGL 应用库 (GLU) 那样。GLTools 包含一个用于操作矩阵和向量的 3D 数学库, 并依靠 GLEW 获得 OpenGL 3.3 中用来产生和渲染一些简单 3D 对象的函数, 以及对视觉平截头体、相机类和变换矩阵进行管理的函数的充分支持。

## 2.2.2 OpenGL API 特性

OpenGL 是由一些充满智慧的人设计的, 他们拥有丰富的图形程序设计 API 设计经验。他们在函数命名和变量声明方法上采用了一些标准规则。API 简单清晰, 便于提供商进行扩展, 并且便于程序员记忆。OpenGL 试图尽可能地避免策略。这里的策略是指设计者做出的关于程序员如何使用 API 的假设。这使 OpenGL 能够保持灵活、强大和快速。我们只要灵活地运用 API 和着色语言, 就可以真正地发明一种全新的方法来渲染一个特效或场景。

这种理念为 OpenGL 的长寿和进化做出了贡献。即使如此, 随着时间的推移, 硬件性能出乎意料的发展和开发人员与硬件提供商的创造力在 OpenGL 经历了这些年的发展后还是给它带来了负面的影响。尽管如此, OpenGL 的基础 API 仍然显示出对新兴的不可预料特性的适应能力。只作很少甚至不作改动就能编译十年前源代码的能力对于众多应用程序开发人员来说是最重要的优势, 而 OpenGL 多年来一直坚持做到在加入新特性的同时尽量少地与旧代码发生冲突。现在, 我们有了更加简洁和现代化的 OpenGL, 可以重新开始这个过程了。

### 数据类型

为了使 OpenGL 代码更易于从一个平台移植到另一个平台, OpenGL 定义了数据类型。这些数据类型可以映射到所有平台上的特定最小格式。各种编译器和环境都有自己的规则来定义各种变量类型的大小

和内存布局，因此通过使用 OpenGL 定义的变量类型，可以使代码避免因类型在变量表示上不一致所带来的影响。表 2.2 列出了 OpenGL 数据类型和最小位宽。

表 2.2 OpenGL 变量类型和最小位宽

OpenGL 数据类型	最小位宽	描 述
GLboolean	1	布尔值，真或假
GLbyte	8	有符号 8 位整数
GLubyte	8	无符号 8 位整数
GLchar	8	字符串
GLshort	16	有符号 16 位整数
GLushort	16	无符号 16 位整数
GLhalf	16	半精度浮点值
GLint	32	有符号 32 位整数
GLuint	32	无符号 32 位整数
GLsizei	32	无符号 32 位整数
GLenum	32	无符号 32 位整数
GLfloat	32	32 位浮点数
GLclampf	32	[0, 1]范围内的 32 位浮点数
GLbitfield	32	32 位
GLdouble	64	64 位双精度数
GLclampd	64	[0, 1]范围内的 64 位双精度数
GLint64	64	有符号 64 位整数
GLuint64	64	无符号 64 位整数
GLsizeiptr	本地指针大小	无符号整数
GLintptr	本地指针大小	有符号整数
GLsync	本地指针大小	同步对象句柄

所有的数据类型都以 GL 开头，表示 OpenGL。函数后面是它们的最小位宽和相关描述，请注意它们并没有必要直接与 C 数据类型直接对应。OpenGL 规范要求这些数据类型所需的最小存储空间参见表 2.2。但是，虽然某些数值超出表中指出的范围是可能的，但只有大小在指定范围内的数值对 OpenGL 来说才是有意义的。请注意，有些类型前面还有个字母 u，表示这是一种无符号数据类型。例如，ubyte 表示无符号 byte 类型。在某些应用中，还有一些更具描述性的名称，就像 size 表示一个数值的长度或深度那样。例如，GLsizei 是一个 OpenGL 变量类型，表示整数形式的 size 参数。名称 Clamp 则是一种提示，表示这个值的范围将“截取”在 0.0 ~ 1.0 的范围内；GLboolean 变量表示真假条件；GLenum 表示枚举变量；GLbitfield 表示那些包含二进制位段的变量等。

OpenGL 并没有对指针和数组作特殊的考虑。我们可以像下面这样声明一个包含 10 个 GLshort 变量的数组。

```
GLshort shorts[10];
```

下面这行代码则声明了一个长度为 10 的指向 GLdouble 类型变量的指针数组。

```
GLdouble *doubles[10];
```

### 2.2.3 OpenGL 错误

在任何项目中,我们都希望编写出表现良好的程序,能够友好地响应用户,并且有一定程度的灵活性。使用 OpenGL 的图形程序也不例外,而且如果我们希望程序能够流畅地运行,就需要考虑程序可能出现的错误以及一些出乎意料的情况。OpenGL 提供了一种有用的机制,可以在代码中执行一种偶然健全性检查。这个功能是非常重要的。例如,单纯从代码的角度而言,要分辨程序的输出到底是“空间站自由度”还是“空间站融化的蜡笔”几乎是不可能的!

OpenGL 在内部保留了一组错误标志(共 4 个),其中每个标志代表一种不同类型的错误。当一个错误发生时,与这个错误相对应的标志就会被设置。为了观察哪些标志被设置,可以调用 glGetError 函数。

```
Glenum glGetError(void);
```

glGetError 函数返回表 2.3 所列出的其中一个值。如果被设置的标志不止一个,glGetError 仍然只返回一个唯一的值。当 glGetError 函数被调用时,这个值随后被清除,然后在 glGetError 再次被调用时将返回一个错误标志或 GL\_NO\_ERROR。通常情况下,我们需要在一个循环中调用 glGetError 函数,持续检查错误标志,直到返回值是 GL\_NO\_ERROR 为止。

表 2.3 OpenGL 错误代码

错误代码	描 述
GL_INVALID_ENUM	枚举参数超出范围
GL_INVALID_VALUE	数值参数超出范围
GL_INVALID_OPERATION	在当前的状态中操作非法
GL_OUT_OF_MEMORY	没有足够的内存来执行这条命令
GL_NO_ERROR	没有错误出现

如果一个错误是由于对 OpenGL 的非法调用所致,那么这条命令或函数调用将会被忽略。对此,我们可能会稍微感到安心。此时,唯一可能造成麻烦的是那些接受指向内存的指针作为参数的函数(如果指针无效,可能导致程序崩溃)。

### 2.2.4 确认版本

如前面所述,有时候我们希望利用一个特定实现中的一些已知行为。如果我们确实知道程序将运行于

一个特定提供商所生产的图形卡之上，就想依赖这个生产商特有的一些性能特征来强化程序。我们可能还希望限制这个特定厂商所提供驱动程序的最低版本号。为此，需要查询 OpenGL 的渲染引擎（OpenGL 驱动程序）的生产商和版本号。GL 函数库可以通过调用 `glGetString` 来返回与它们的版本号和生产商有关的特定信息。

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

这个函数返回一个静态的字符串，描述 GL 函数库中所请求的信息。附录 C 中列出了 `glGetString` 条目下所有合法的参数值，以及它们所代表的 GL 函数库的相关信息。

## 2.2.5 使用 `glHint` 获取线索

俗话说，给猫剥皮的方法不止一种。在 3D 图形算法中，情况也是如此。例如，为了追求高性能，我们常常需要做一些权衡。或者，如果视觉逼真度是最重要的因素，那么性能就会退居其次。一种 OpenGL 实现常常包含两种方法来执行一个特定的任务，一种是快速的方法，在性能上稍作妥协；另一种是慢速的方法，着重于改进视觉质量。`glHint` 函数允许我们指定偏重于视觉质量还是速度，以适应各种不同类型的操作需求。这个函数定义如下所示。

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

我们可以在 `target` 参数中指定希望进行修改的行为类型。附录 C 中的 `glHint` 条目下列出了这些值，其中还包括关于纹理压缩质量和抗锯齿准确性等的提示。`mode` 参数告诉 OpenGL 我们最为关心的什么，例如更快的渲染速度还是最好的输出质量，或者我们可能并不关心这些（只有在这种情况下我们才会使用默认行为）。但是，我们还是应该小心在意，因为所有的 OpenGL 实现都不要求必须在 `glHint` 函数的调用上保持一致。在 OpenGL 中，这是唯一一个行为完全依赖生产商的函数。

## 2.2.6 OpenGL 状态机

绘制 3D 图形是一项复杂的任务。在接下来的章节，我们将讨论许多 OpenGL 函数。对于一个特定的几何图形，有许多因素可能会影响它的绘制。对象是不是与背景混合？要不要进行正面或背面剔除？当前限制的是什么纹理？这样的问题数不胜数。

我们把这类变量的集合称为管线的状态。状态机是一个抽象的模型，表示一组状态变量的集合。每个状态变量可以有各种不同的值，或者只能可以打开或关闭等。当我们在 OpenGL 中进行绘图时，如果每次都要指定所有这些变量显然有点不切实际。反之，OpenGL 使用了一种状态模型（或称状态机）来追踪所有的 OpenGL 状态变量。当一个状态值被设置之后，它就一直保持这个状态，直到其他函数对它进行修改为止。许多状态只能简单地打开或关闭。例如，深度测试（参见第 3 章）就是要么打开、要么关闭。打开深度测试的几何绘图将会被检查以确保在进行渲染之前总会在任何位于它后面的对象前方。在深度测试关闭后进行的几何图形绘制（例如 2D 覆盖）则会在不进行深度比较的情况下进行绘制。

为了打开这些类型的状态变量，可以使用下面这个 OpenGL 函数。

```
void glEnable(GLenum capability);
```

我们可以使用下面这个对应的函数，把这些变量的状态设置为关闭。

```
void glDisable(GLenum capability);
```

以深度测试为例，可以使用下面这个函数调用打开深度测试。

```
glEnable(GL_DEPTH_TEST);
```

也可以使用下面这个函数调用关闭深度测试。

```
glDisable(GL_DEPTH_TEST);
```

如果希望对一个状态变量进行测试，以判断它是否已被打开，OpenGL 还提供了一种方便的机制。

```
GLboolean glIsEnabled(GLenum capability);
```

但是，并不是所有的状态变量都只是简单地打开或关闭。许多 OpenGL 函数专门用于设置变量的值，此后这些变量一直保持被设置时的值，直到再次被修改。我们在任何时候都可以查询这些变量的值。OpenGL 提供了一组查询函数，可以查询布尔型、整型、单精度浮点型和双精度浮点型变量的值。这 4 个函数的原型如下所示：

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

每个函数都会返回单个值，或者返回一个数组，把一些值存储到我们指定的地址中。附录 C 的参考部分列出了各种不同的参数（数量非常多）。现在，读者可能还无法理解其中的大多数参数，但是随着对本书学习的深入，读者将会逐渐开始欣赏 OpenGL 状态机的简洁和强大。

## 2.3 建立 Windows 项目

在微软公司的 Windows 系统上建立程序的方式有很多。在本书，我们将使用 Visual C++ 2008 速成版。这是微软公司提供的免费编译器，可以从网址 <http://www.microsoft.com/exPress> 上下载。在这个版本上创建的项目也能在这个开发环境的更新版本上使用。

正如前面已经提到的，本书所有的项目都是基于 GLEW、GLTools 和 freeglut 实用库（在 Windows 上）的。GLEW “内建”在 GLTools 中是因为 GLTools 需要 GLEW 来获得 OpenGL3.0 或更新版本的特性。而 freeglut 则是一个独立的库，它可以与其他 OpenGL 库配合使用，在我们要使用本地系统服务（参见第 13 章到第 16 章的内容）的情况下也可以独立使用。在开始创建我们的第一个新项目，或者重建本书中任何一个项目之前，我们要做的第一件事情就是将这些库的“include”文件夹添加到 Visual Studio 的包含搜索路径中。无论何时向编程目录中添加新的 SDK 或者库，都要这样做。如果读者以前还没做过这样的事，不要紧张，这只是小菜一碟，我们只要做一次就行，而不需要每次建立一个新项目时都做。

## 2.3.1 包含路径

打开 Visual C++, 从主菜单中选择 “Tools” (工具), 然后从弹出的下拉菜单底部选择 “Options” (选项)。“Options” 对话框如图 2.1 所示。展开 “Projects and Solutions” (项目与解决方案) 树节点, 并选择 “VC++ Directories” (VC++ 目录), 然后确保 “Show Directories For” (显示目录) 选项被设置为 “Include Files” (包含文件)。

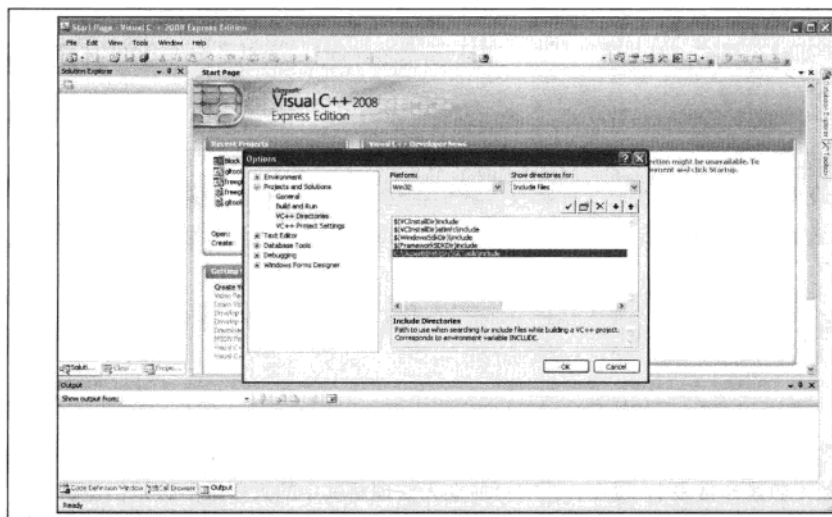


图 2.1

为包含文件添加一个  
搜索路径

下方的列表是一个组合框, 显示在我们向源代码中加入一个头文件时所有将被搜索的文件夹。我们需要为 GLTools 和 freelglut 添加 “include” (包含) 路径。选择列表中最末的空行, 在这一行单击两次, 这一行就会变成可编辑区域, 并且在右侧出现一个浏览按钮, 如图 2.2 所示。

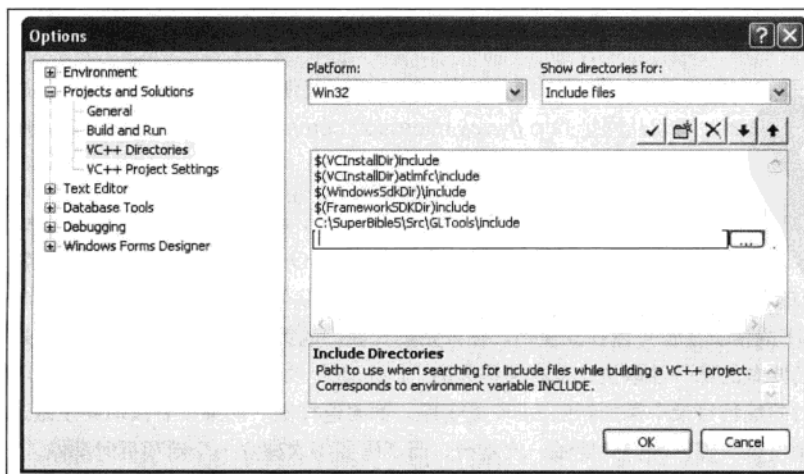
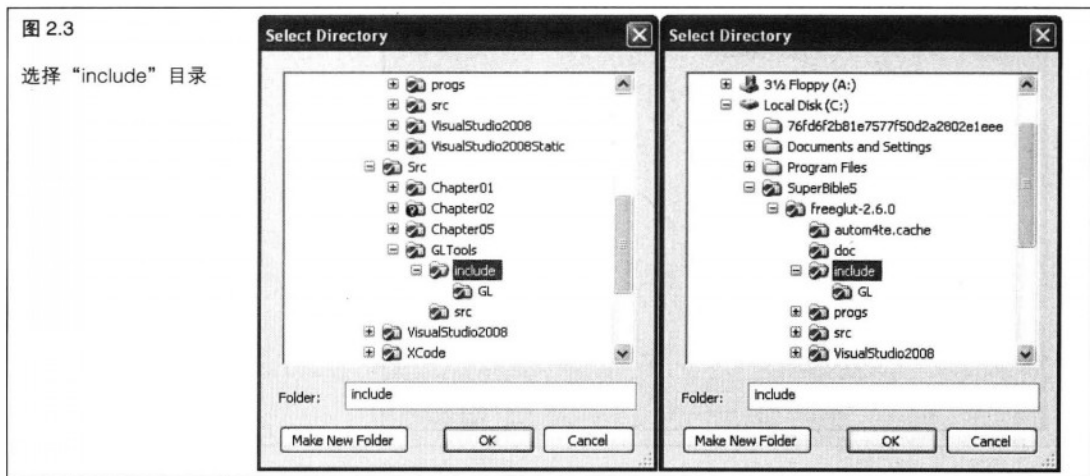


图 2.2

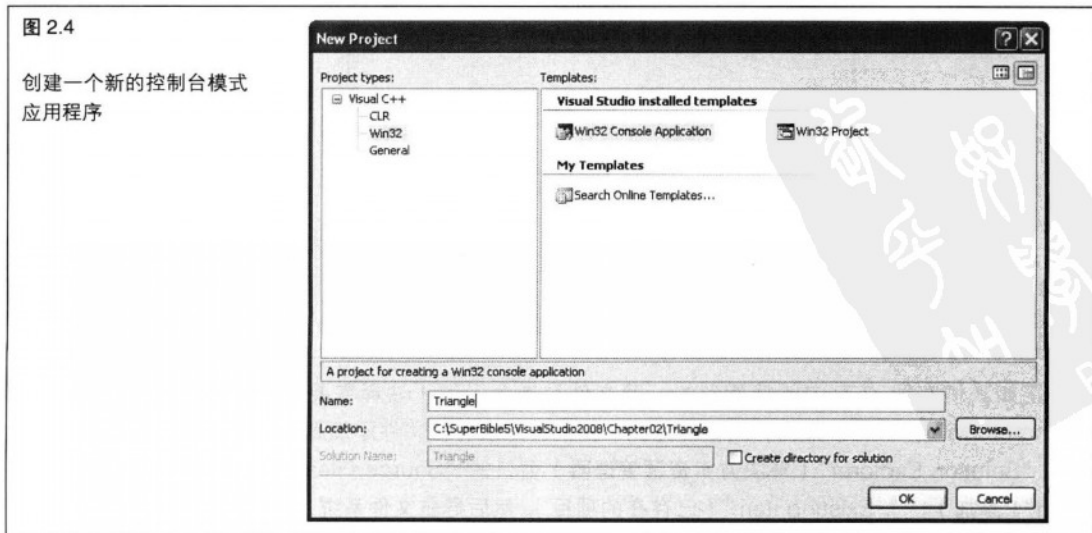
键入或浏览将要添加的路径

单击浏览按钮，就会出现一个文件浏览对话框。导航到 GLTools 下的“include”文件夹并选中它，如图 2.3 所示。用同样的方法为 freeglut 添加“include”路径。现在，我们已经将 Visual C++ 设置为包含 GLTools 和 freeglut 库了。现在可以开始创建我们的第一个项目了！



## 2.3.2 创建项目

如果还没有打开 Visual C++，那么现在就打开它，并在主菜单中选择“File”(文件)→“New Project”(新建项目)。基于 GLUT 的应用程序是 Win32 控制台模式的应用程序，所以要正确地进行选择，然后单击“OK”(确定)，如图 2.4 所示。随后出现的对话框如图 2.5 所示，选择“Empty Project”(空工程)，C++ 就会建立一个我们需要的项目，而不是它认为标准的基于控制台的项目。



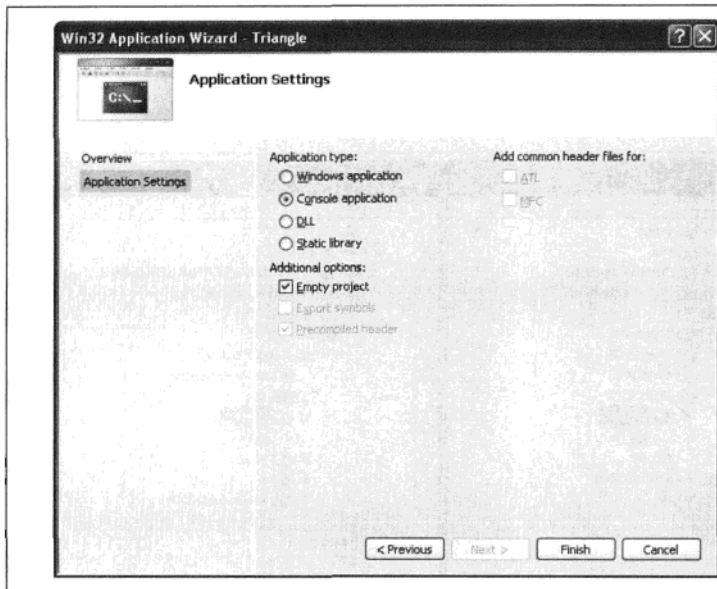


图 2.5

确保创建的是一个空项目

### 2.3.3 添加文件

现在可以开始创建主源文件了。在这个项目中，我们创建一个名叫“triangle.cpp”的 C++ 文件。要完成这一步，只需选择“File”（文件）→“New”（新建），再次在主菜单中选择“File”，选择一个新的 C++ 文件，如图 2.6 所示。

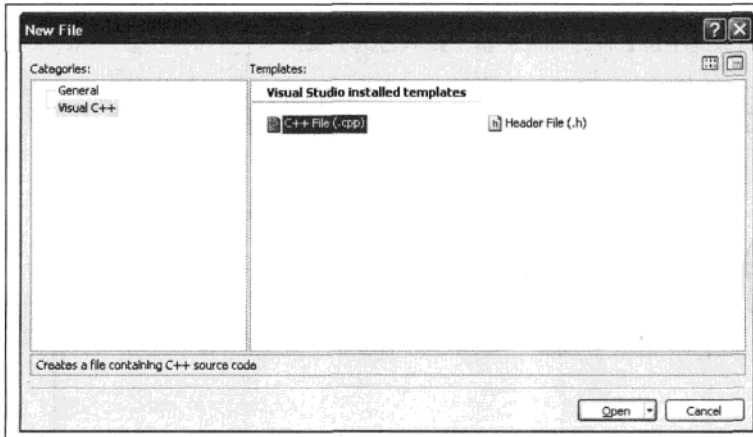


图 2.6

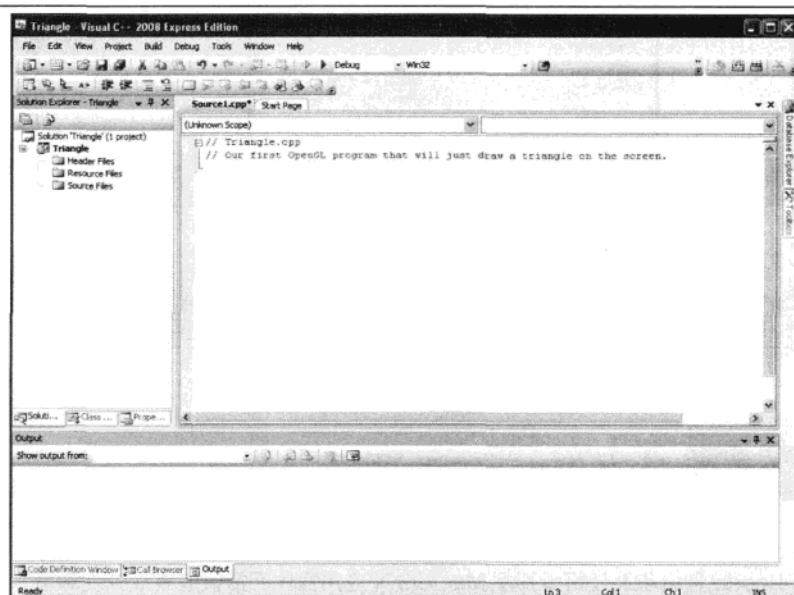
添加一个新的 C++ 源文件

正如我们在图 2.7 中能够看到的，源文件是未命名的（注释需要人工添加）。将文件命名为“triangle.cpp”并保存。然而，到目前为止，我们仍然没有把文件添加到项目中。要完成这一步，只需在“Solution Explorer”（解决方案资源管理器）窗口的“Source Files”文件夹上单击右键，选择“Add”（添加）→“Existing Item”（已存在的项目），然后导航文件系统，直到定位到“triangle.cpp”源文件。

我们差不多完成了。最后，需要将 GLTools 和 freeglut 库添加到项目中。将一个库添加到项目中，方法不止一种，我们使用最简单、最容易的方式来亲自操作一下。在项目名称上单击右键，并选择“Add”（添加）→“Existing Item”（已存在的项目），就像添加“triangle.cpp”源文件时一样。但是，这一次我们导航到“/Freeglut-2.6.0/VisualStudio2008Static/Release”文件夹，并选择“Freeglut\_static.lib”文件，如图 2.8 所示。

图 2.7

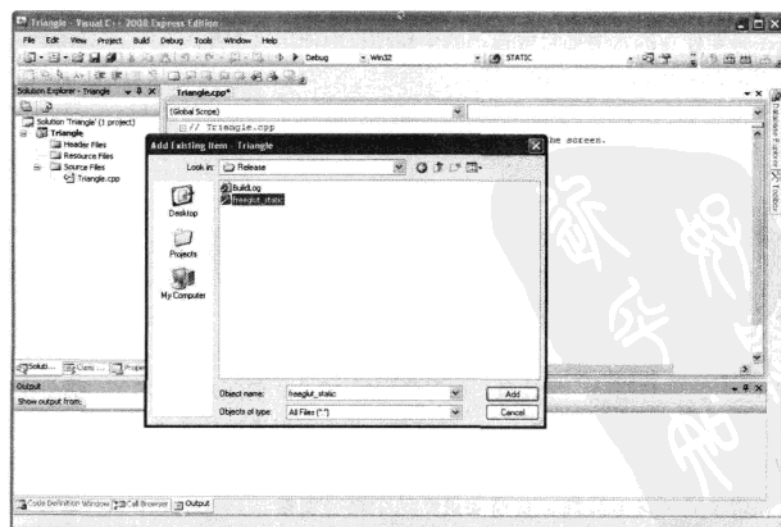
我们的未命名空源文件



用同样的方法添加 GLTools 库，位置在“VisualStudio2008/GLTools/Release folder”。图 2.9 所示是我们建立完成的项目，已经完全做好准备来开始我们的第一个 OpenGL 程序了！

图 2.8

选择 Freeglut\_static 库并添加到项目中



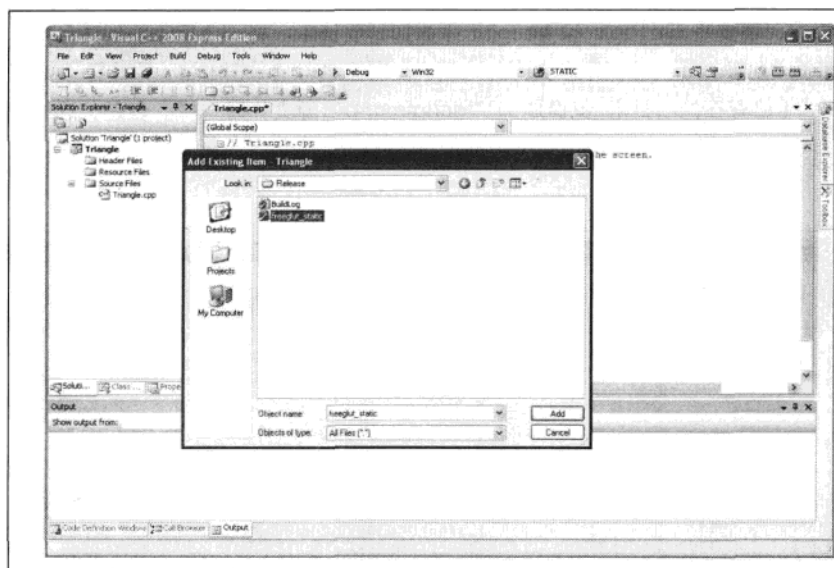


图 2.9

我们创建完成的 Visual C++ 项目

## 2.4 建立 Mac OS X 项目

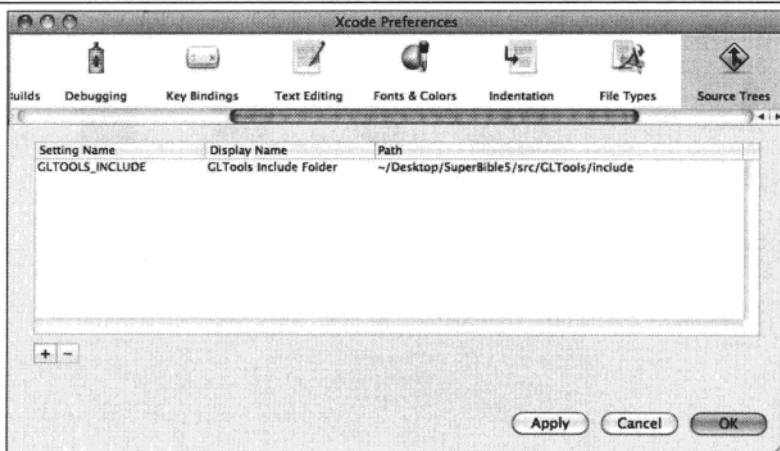
Xcode 是由苹果公司为开发 Mac OS X 应用程序而提供的集成开发环境 (IDE)。Xcode 是免费提供的, 并且可以在 OS X 安装盘或网址 <http://developer.apple.com> 找到它。在本书中所有的示例程序都建立了 Xcode 项目。正如前面已经提到的, 本书所有的 OpenGL 示例都使用 GLUT 和 GLTools 两个实用库。GLUT 库作为 OS X 的标准框架发行, 而 Xcode 知道如何使用 GLUT 而不需要任何特殊设置, 只要将框架添加到工程中就可以了。然而 GLTools 则是一种第三方库 (本书中也要加载), 我们需要配置 Xcode 才能在将要建立的本书项目或自己的全新项目中使用这个库。

### 2.4.1 自定义创建设置

最低限度上, 我们必须告诉 Xcode, GLTools 头文件在哪里, 以使本书包含的项目能够顺利编译 (请注意, 为 Snow Leopard 预构建的 32/64 位二进制文件也包含在了 Mac OS X 源代码中)。要完成这一步, 我们只需在 Xcode 代码树 (Source Tree) 设置中添加一个自定义设置。图 2.10 所示的对话框可以从 Xcode/Preference 菜单中进行访问。GLTOOLS\_INCLUDE 是必须添加的设置, 它是到 GLTools 的 “include” 文件的路径。请注意, 如果我们用 10.6 之前版本的 OS X, 那么我们还需要自己重建这个库, 或者在所有项目中包含进 GLTools 源文件。我们可以通过单击 “+” 按钮并直接在对话框显示的表格中键入来添加设置。在本例, SuperBible 文件是放置在操作系统桌面上的, 读者可能希望将他们放在其他位置。如果移动了这些文件, 就需要更新这个路径变量。

图 2.10

在 X code 中添加一个自定义设置

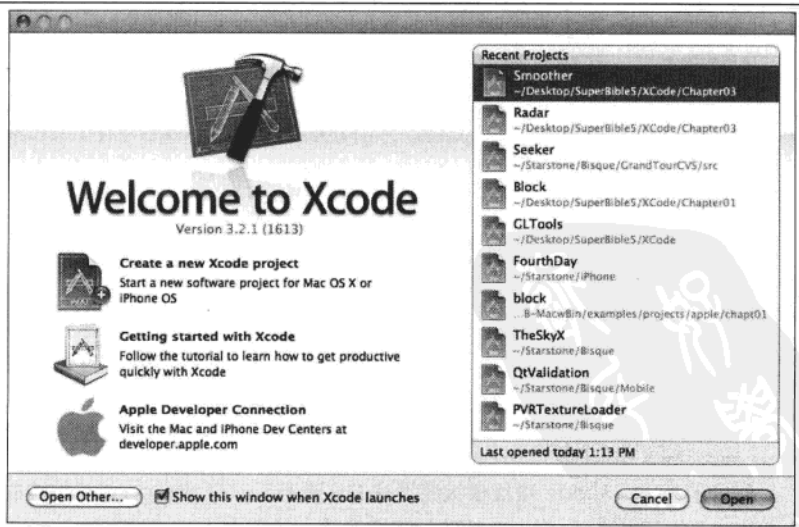


## 2.4.2 创建新项目

现在已经能够使包含的 Xcode 项目文件正常工作了。读者在学习过程中可能会希望自己创建这些示例, 或者对这些示例做一些改动使它们成为自己实验的一部分。让我们从头建立一个全新的工程吧。第一步先在 Xcode 主菜单中选择“File”(文件)→“New Project”(新建项目)。随后会出现如图 2.11 所示的对话框。在右侧我们可以看到任何我们可能会用到的最近的项目, 而在图 2.11 中可以看到作者编写本书时最近正在用的一些东西。在左边可以看到“Create a New Xcode Project”(创建一个新 Xcode 项目)按钮, 单击这个按钮来创建我们的第一个项目。

图 2.11

使用 Xcode 创建一个新项目



接下来, 在如图 2.12 所示的 Mac OS X 项目模板的列表中单击“Cocoa Application”图标。虽然我们将会使用 C++、GLUT, 但基本的编程框架实际上应用的却是 Cocoa, 这是最简单的开始方式。

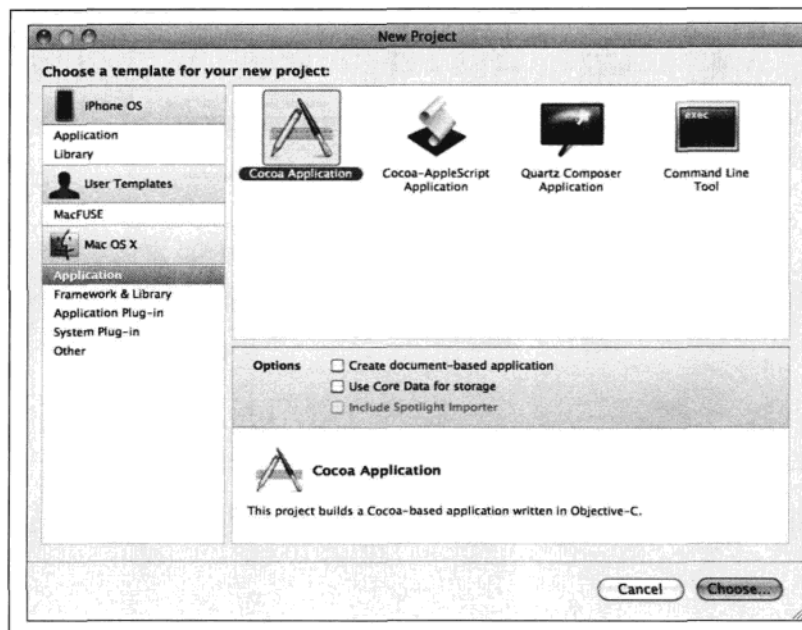


图 2.12

选择一个 Cocoa 应用程序

在示例中，我们将项目命名为“Triangle”。这将是我们的第一个示例程序。在 Xcode 窗口的左边，可以看到“Overview”（概览）面板，展开一些组，我们就能看到这些项目的结构，如图 2.13 所示。

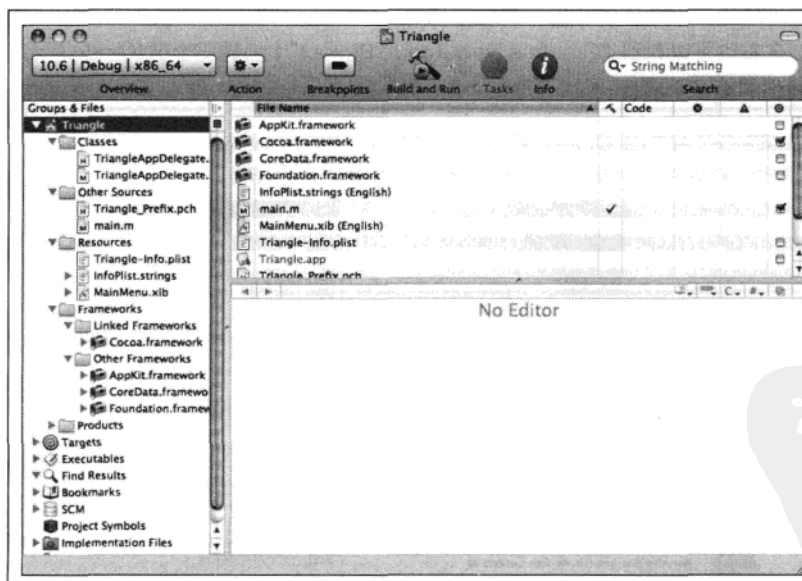


图 2.13

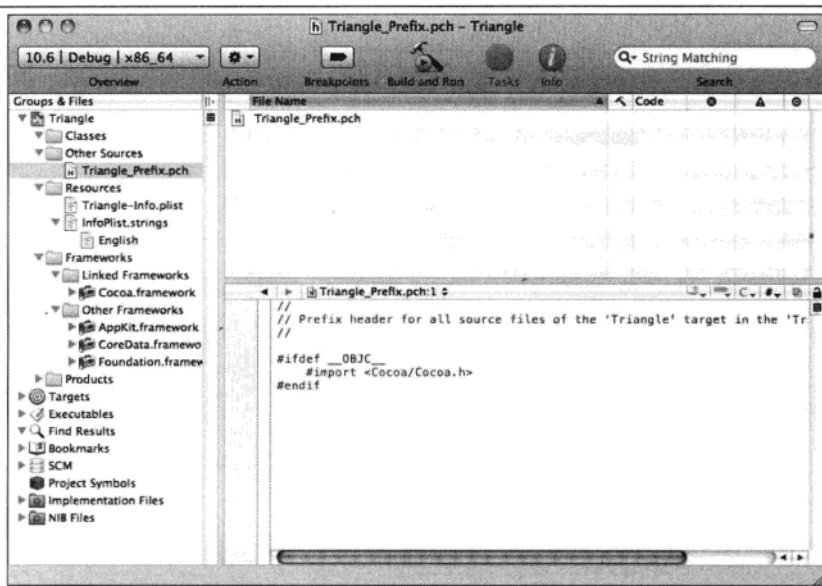
新 Cocoa 应用程序

到目前为止，Xcode 项目模板还没有做好进行 GLUT 或 OpenGL 编程的准备。我们要移除项目起始代码，并从自己的基于 GLUT 的简化程序开始。关于在 Mac 上使用 OpenGL 的更深入讨论，请参见第 14 章。至于基于 GLUT 的项目，需要删除“TriangleAppDelegate.\*”文件和“main.m”文件。我们可以通过在文件名上单击一次来加亮 Xcode 中的这些文件，然后按“Delete”键。继续单击确认对话框中的

“Also Move to Trash”（同时放入回收站）按钮来完全删除这些文件。我们还可以删除“Resource”（资源）组下的“MainMenu.xib”文件。最终清理过的 Xcode 项目如图 2.14 所示，不用管“Triangle.Prefix.pch”文件。

图 2.14

删除了默认代码的新项目



### 2.4.3 框架、头文件和库

接下来我们需要将 GLUT 和 OpenGL 框架添加到项目中。右键单击“Frameworks”（框架）文件夹组，并选择“Add”（添加）→“Existing Item”（已存在的项目）。在接下来的对话框中选择 GLUT 和 OpenGL 框架。这个对话框的外观不断地变化，但无论怎么变化我们都应该看到一个可用框架列表。单击其中一个加亮它，然后按住“command”键单击并选择多个框架。对于项目而言，只需要 GLUT 和 OpenGL。

最后，我们需要添加 GLTools 库。GLTools 是一个静态库而不是框架，所以需要不同的方式来添加。完成这项工作有很多方法，而且我确信会有人写信告诉我他们的方法更好。如果读者还不知道如何做，那么就可能会更喜欢我的而不是自己的方法了！

首先，我们必须将 GLTools 头文件路径添加到头文件搜索路径设置中。要完成这一步，只需右键单击“Groups&Files”下的项目名称，然后选择“Get Info”（获取信息）。确保“Configuration”（配置）设置被设为“All Configurations”（所有配置），如图 2.15 所示。向下滚动“Search Paths”分类，并在“Header Search Paths”区域单击，键入“\$(GLTOOLS\_INCLUDE)”。当按“Enter”（回车）键或改变当前区域时，就会自动改变为我们刚刚为这个变量所设置的值。

最后，我们需要将 LibGLTools.a 文件添加到项目中。要完成这一步，只需拖曳 LibGLTools.a 文件（在随书发布的源代码文件的/Xcode/Xcode/GLTools 文件夹中）并放到 Xcode 中的“Frameworks”（框架）文件夹中；还可以右键单击“Frameworks”并选择“Add”（添加）→“Existing Item”（已存在的项目），

然后手动导航到 GLTools.a 文件并选择它。拖曳的方法通常要简单得多！

要使我们的 Xcode 项目准备就绪，以上就是全部所需的工作了。现在我们需要做的就只有添加 C++ 源文件并开始编译了！右键单击“Other Sources”文件夹并选择 Add”（添加）→ “New File”（新文件），随后出现的对话框如图 2.16 所示。

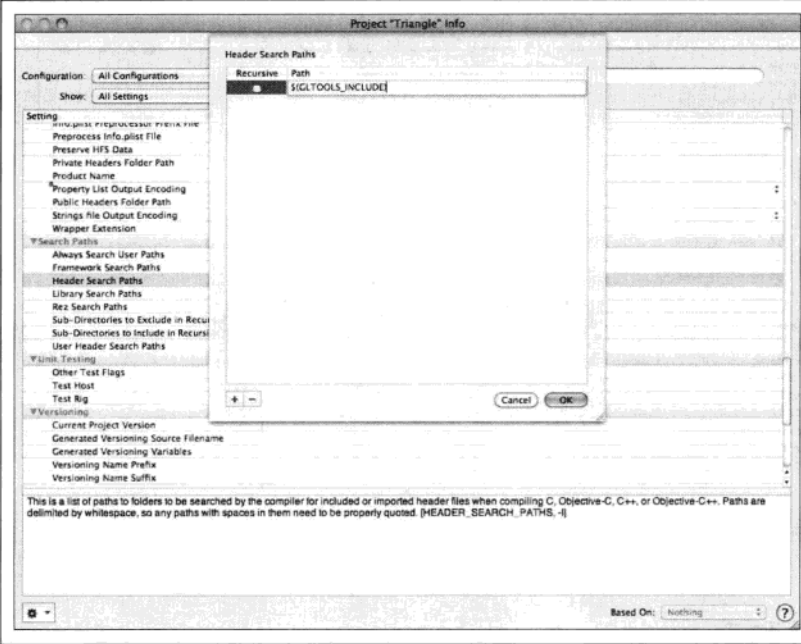


图 2.15

将 GLTools 的 include 路径添加到项目中

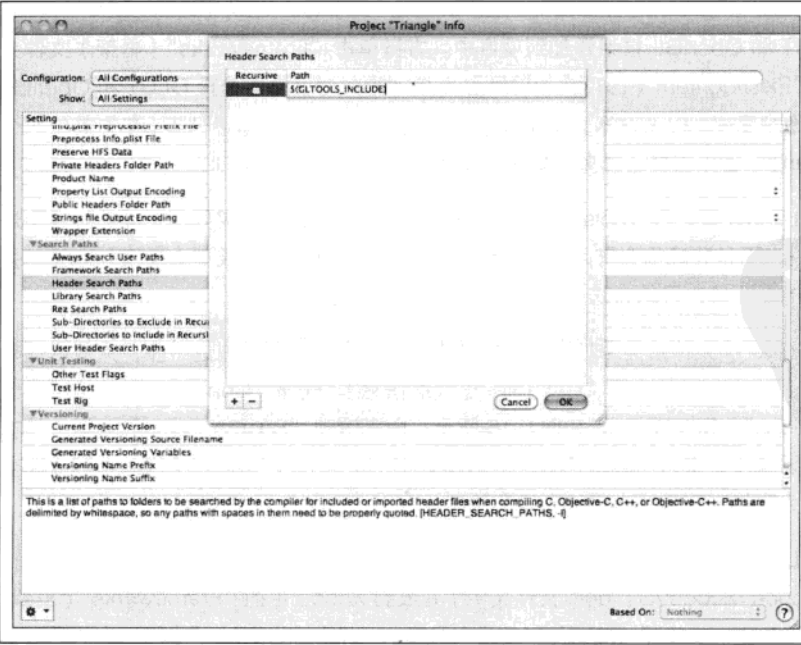


图 2.16

将新的 C++源文件添加到项目中

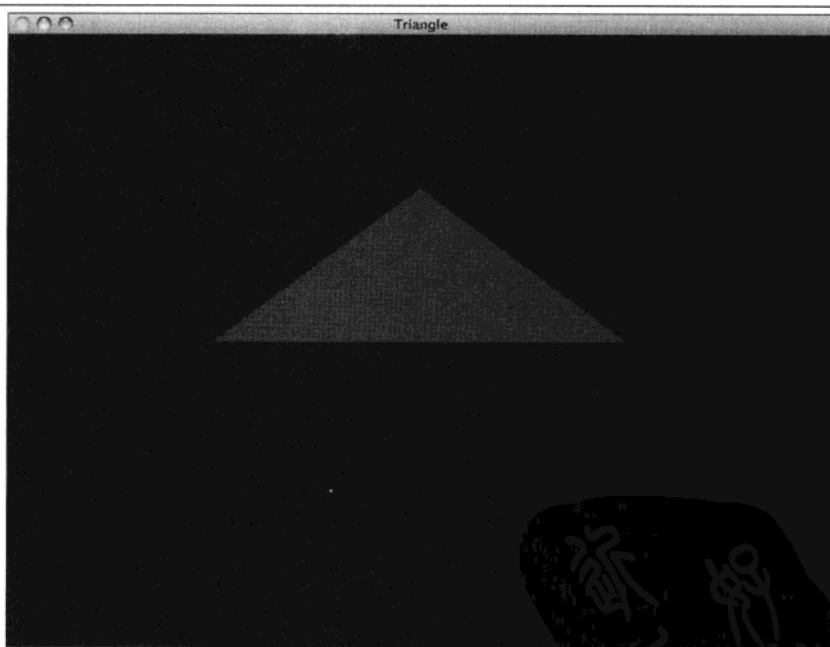
如图 2.16 所示, 选择 “C++ File”, 并在接下来出现的对话框中将程序文件命名为 “Triangle.cpp” 并保存。继续取消选择 “Also Create Triangle.h”, 这是因为既然我们并不是真的要建立一个 C++ 类, 那么主程序文件就不需要一个文件头。要在 Xcode 中创建一个项目, 需要按 “command” 键+ “B” 键, 而如果要建立并运行我们的项目, 则需要按 “command” 键+ “R” 键。当然, 到目前为止我们还没有任何源代码。让我们开始第一个 OpenGL 程序吧!

## 2.5 第一个三角形

现在我们已经打好了基础, 终于可以开始编写代码了! 我们的第一个示例程序仅仅是在蓝色的背景上绘制一个红色的三角形。这乍看起来似乎没什么挑战性, 但它实践了所有必要的步骤, 并创建了一个完整的演示框架供本书以后使用。在创建过程中, 我们能够学习 GLUT, 并使用第一个 GLTools 帮助程序和类。我们的三角形程序如图 2.17 所示, 而程序清单 2.1 则完整地列出了我们的第一个程序。接下来我们将一行一行地讨论它。

图 2.17

第一个 OpenGL 程序的输出结果



程序清单 2.1 简单绘制一个三角形

```
// Triangle.cpp
// Our first OpenGL program that will just draw a triangle on the screen.

#include <GLTools.h>           // OpenGL toolkit
#include <GLShaderManager.h>   // Shader Manager Class

#ifdef __APPLE__
#include <glut/glut.h>         // OS X version of GLUT
```

```

#else
#define FREEGLUT_STATIC
#include <GL/glut.h> // Windows FreeGlut equivalent
#endif

GLBatch triangleBatch;
GLShaderManager shaderManager;

/////////////////////////////////////////////////////////////////
62 CHAPTER 2 Getting Started
04_0321712617_ch02.qxd 6/21/10 11:35 AM Page 62
// Window has changed size, or has just been created. In either case, we need
// to use the window dimensions to set the viewport and the projection matrix.
void ChangeSize(int w, int h)
{
    glViewport(0, 0, w, h);
}

/////////////////////////////////////////////////////////////////
// This function does any needed initialization on the rendering context.
// This is the first opportunity to do any OpenGL related tasks.
void SetupRC()
{
    // Blue background
    glClearColor(0.0f, 0.0f, 1.0f, 1.0f );

    shaderManager.InitializeStockShaders();

    // Load up a triangle
    GLfloat vVerts[] = { -0.5f, 0.0f, 0.0f,
                        0.5f, 0.0f, 0.0f,
                        0.0f, 0.5f, 0.0f };

    triangleBatch.Begin(GL_TRIANGLES, 3);
    triangleBatch.CopyVertexData3f(vVerts);
    triangleBatch.End();
}

/////////////////////////////////////////////////////////////////
// Called to draw scene
void RenderScene(void)
{
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

    GLfloat vRed[] = { 1.0f, 0.0f, 0.0f, 1.0f };
    shaderManager.UseStockShader(GLT_SHADER_IDENTITY, vRed);
    triangleBatch.Draw();

    // Perform the buffer swap to display the back buffer
    glutSwapBuffers();
}

/////////////////////////////////////////////////////////////////
// Main entry point for GLUT based programs
int main(int argc, char* argv[])
{
    gltSetWorkingDirectory(argv[0]);

```

```

glutInit(&argc, argv);
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH | GLUT_STENCIL);
glutInitWindowSize(800, 600);
glutCreateWindow("Triangle");
glutReshapeFunc(ChangeSize);
glutDisplayFunc(RenderScene);

GLenum err = glewInit();
if (GLEW_OK != err) {
    fprintf(stderr, "GLEW Error: %s\n", glewGetErrorString(err));
    return 1;
}

SetupRC();

glutMainLoop();
return 0;
}

```

## 2.5.1 要包含什么

在开始编写任何 C++ (或者只是 C) 程序之前, 都要先将要用到的函数和类定义的头文件包含进来。为了达到目的, 最低限度也要包含如下头文件。

```

#include <GLTools.h>          // OpenGL toolkit
#include <GLShaderManager.h>  // Shader Manager Class

#ifdef __APPLE__
#include <glut/glut.h>        // OS X version of GLUT
#else
#define FREEGLUT_STATIC
#include <GL/glut.h>          // Windows/Linux FreeGlut equivalent
#endif

```

GLTools.h 头文件中包含了大部分 GLTools 中类似 C 语言的独立函数, 而每个 GLTools 的 C++ 类则有自己的头文件。GLShaderManager.h 移入了 GLTools 着色器管理器 (Shader Manager) 类。没有着色器, 我们就不能在 OpenGL (核心框架) 中进行着色。着色器管理器不仅允许我们创建并管理着色器, 还提供一组“存储着色器” (Stock Shader), 它们能够进行一些初步和基本的渲染操作。在第 3 章中我们将详细讨论这部分内容。

根据应用程序是否是在 Mac 上创建的, GLUT 将采取不同的处理方式。在 Windows 和 Linux 上, 我们使用 freeglut 的静态库版本, 这就需要在它前面添加 FREEGLUT\_STATIC 处理器宏。

## 2.5.2 启动 GLUT

下面我们直接跳到程序清单的最后一个函数, 即所有 C 程序的入口点, 这里才是程序处理实际开始的地方。

```
////////////////////////////////////  
// Main entry point for GLUT based programs  
int main(int argc, char* argv[])  
{  
    glutSetWorkingDirectory(argv[0]);
```

控制台模式的 C 语言和 C++ 程序总是从 “main” 函数开始处理。对于有经验的 Windows 迷来说，在本例中找不到 WinMain 函数是十分奇怪的。这是因为我们是以一个控制台模式的应用程序开始的，所以没必要从创建窗口和消息循环开始。在 Win32 中，我们可以从控制台应用程序建立图形窗口，就像我们可以从 GUI 应用程序建立控制台窗口一样。GLUT 库隐藏了这些细节（请记住，GLUT 库就是设计用来隐藏这些平台相关细节的）。

GLTools 函数 glutSetWorkingDirectory 用来设置当前工作目录。实际上在 Windows 中是不必要的，因为工作目录默认就是与程序的可执行执行程序相同的目录。但是在 Mac OS X 中，这个程序将当前工作文件夹改为应用程序捆绑包（Application Bundle）中的/Resource 文件夹。GLUT 的优先设定自动进行了这种设置，但是这种方法更加安全，也总是奏效的，即使其他程序改变了这项设置时也是如此。这在我们以后想要载入纹理文件或模型数据时会派上用场。

接下来，我们将进行一些基于 GLUT 的标准设置。首先要调用 glutInit 函数，这个函数只是传输命令行参数并初始化 GLUT 库。

```
glutInit(&argc,argv);
```

然后我们必须告诉 GLUT 库，在创建窗口时要使用哪种类型的显示模式。

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH | GLUT_STENCIL);
```

这里的标志告诉它要使用双缓冲窗口（GLUT\_DOUBLE）和 RGBA 颜色模式（GLUT\_RGBA）。双缓冲窗口（）是指绘图命令实际上是在离屏缓冲区执行的，然后迅速转换成窗口视图。这种方式经常用来生成动画效果，本章稍后将做演示。GLUT\_DEPTH 位标志将一个深度缓冲区分配为显示的一部分，因此我们能够执行深度测试，同样，GLUT\_STENCIL 确保我们也会有一个可用的模板缓冲区。深度和模板测试后面都会讲到。

接下来，我们要告诉 GLUT 窗口的大小，并继续创建以 “Triangle” 为标题窗口。

```
glutInitWindowSize(800, 600);  
glutCreateWindow("Triangle");
```

GLUT 内部运行一个本地消息循环，拦截适当的消息，然后调用我们为不同时间注册的回调函数。与使用真正的系统特定框架相比有一定的局限性，但是大大简化了组织并运行一个程序的过程，并且支持一个演示框架的最低限度的事件。在这里，我们必须为窗口改变大小而设置一个回调函数，以便能够设置视点，还要注册一个函数以包含 OpenGL 渲染代码。

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

ChangeSize 和 RenderScene 函数很快就会讲到，但是在开始运行主消息循环之前，还要解决两件事情。第一件事情就是初始化 GLEW 库。重新调用 GLEW 库初始化 OpenGL 驱动程序中所有丢失的入口点，以确保 OpenGL API 对我们来说完全可用。调用 glewInit 一次就能完成这一步，在试图做任何渲染

之前，还要检查确定驱动程序的初始化过程中没有出现任何问题。

```
Glenum err = glewInit();
if (GLEW_OK != err) {
    fprintf(stderr, "GLEW Error: %s\n", glewGetErrorString(err));
    return 1;
}
```

最后一项准备工作就是调用 SetupRC。

```
SetupRC();
```

实际上这个函数对 GLUT 没有什么影响，但是在实际开始渲染之前，我们在这里进行任何 OpenGL 初始化都非常方便。这里的 RC 代表渲染环境（Rendering Context），这是一个运行中的 OpenGL 状态机的句柄。在任何 OpenGL 函数起作用之前必须创建一个渲染环境，而 GLUT 在我们第一次创建窗口时就完成了这项工作。关于特定操作系统的章节（第 13 章到第 16 章）中会研究这方面的更多细节。纵观全书，我们将在这里进行预加载纹理，建立几何图形、渲染器等工作。

最后，我们可以开始主消息循环并结束 main 函数了。

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

GlutMainLoop 函数被调用之后，在主窗口被关闭之前都不会返回，并且一个应用程序中只需调用一次。这个函数负责处理所有操作系统特定的消息、按键动作等，直到我们关闭程序为止。它还能确保我们注册的这些回调函数被正确地调用。

### 2.5.3 坐标系基础

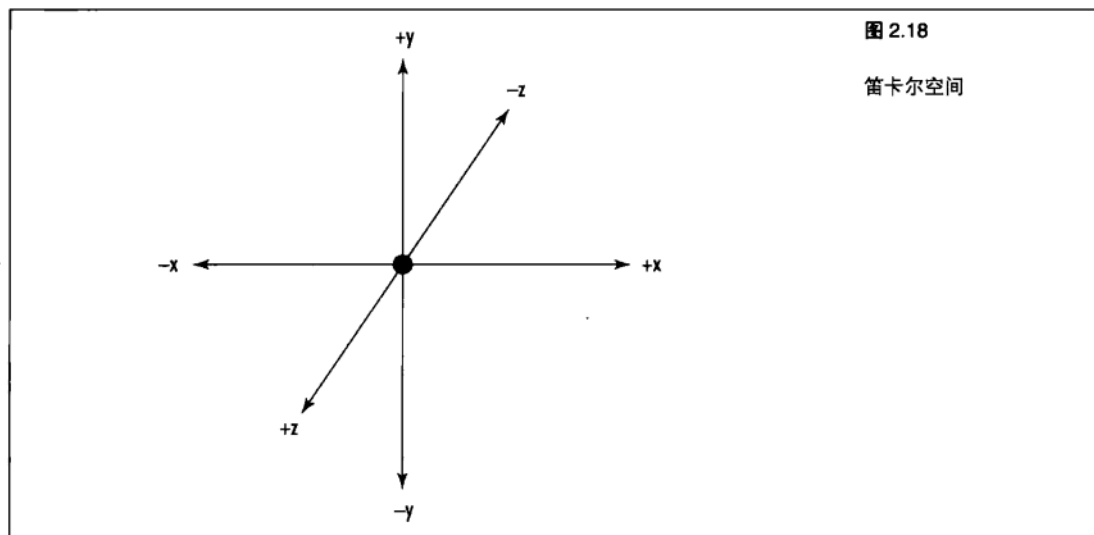
在早期的所有窗口化环境中，用户可以在任何时候改变窗口的大小和维度。甚至在编写一个总是运行在全屏模式下的游戏时，窗口仍然认为至少改变了一次窗口大小——就是在窗口创建时。在进行这些改变时，窗口通常会根据新的维度相应重绘它的内容。有时候，我们可能想要在一个小的窗口中截取绘图内容，或者在原始的大窗口中显示完整的绘图内容。为了达到目的，我们常常希望缩放绘图内容来适应窗口，不论绘图内容和窗口的大小如何。这样，一个很小的窗口可以显示一个完整但是很小的绘图内容，而一个很大的窗口也可以显示相似但是更大的绘图内容。

在第一章，我们讨论了视口和视景物是如何影响 2D 和 3D 绘图内容在计算机屏幕上 2D 窗口上的坐标范围和缩放的。现在，我们来讨论 OpenGL 中的视口和裁减区域坐标（Clipping Volume Coordinate）。在某种程度上，设置坐标系是绘制对象并将它们显示到我们希望的屏幕位置的先决条件！

尽管我们所绘制的图形是一个 2D 的平面三角形，但它实际是在一个 3D 坐标空间中绘制的。在本章中，我们将使用默认的笛卡尔坐标系，这个坐标系统在  $x$ 、 $y$  和  $z$  方向上从 -1 到 +1 延伸。 $x$  是坐标系的横坐标轴， $y$  是纵坐标轴，而  $z$  轴正方向从屏幕向外指向使用者。坐标 (0, 0, 0) 则位于屏幕的正中。在第 4 章，我们将更细致地讨论关于建立代替坐标系的问题。为了达到目的，我们在  $z=0$  的  $xy$  平面上绘制三角形。我们的视角是从  $z$  轴的正半轴看去，所看到的是  $z=0$  情况下的三角形（如果读者对这方面的

内容感到困惑，可以回顾第1章的相关资料)。

图 2.18 所示是基本笛卡尔坐标系的外观。许多绘图和图形库都使用窗口坐标(像素)来完成绘制命令。使用实数浮点(这看上去有点随意)坐标系统进行渲染，这常常令许多新手非常不习惯。不过，在创建了几个程序之后，读者很快就会对此习以为常。



## 定义视口

由于在不同环境下窗口的大小变化的检测和处理方式也不同，GLUT 库为此专门提供了 `glutReshapeFunc` 函数，这个函数注册了一个回调，供 GLUT 库在窗口维度改变时调用。我们传递到 `glutReshapeFunc` 的函数原形如下。

```
void ChangeSize(GLsizei w, GLsizei h);
```

我们选择 `ChangeSize` 作为这个函数的描述性名称，并且在以后的示例中也会使用这个名称。

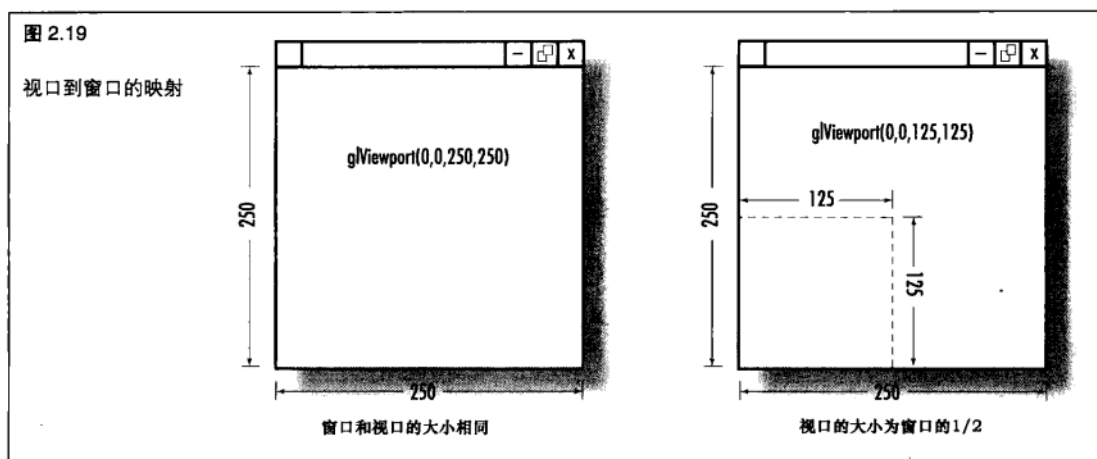
```
void ChangeSize(int w, int h)
{
    glViewport(0, 0, w, h);
}
```

`ChangeSize` 函数在窗口大小改变时接受新的宽度和高度。我们可以使用这个信息，在 OpenGL 函数 `glViewport` 的帮助下修改从目的坐标系到屏幕坐标系上的映射。要理解视口分辨率，让我们更仔细地观察 `ChangeSize` 函数中调用带有窗口的新宽度和高度的 `glViewport` 函数的部分。`glViewport` 函数定义如下。

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

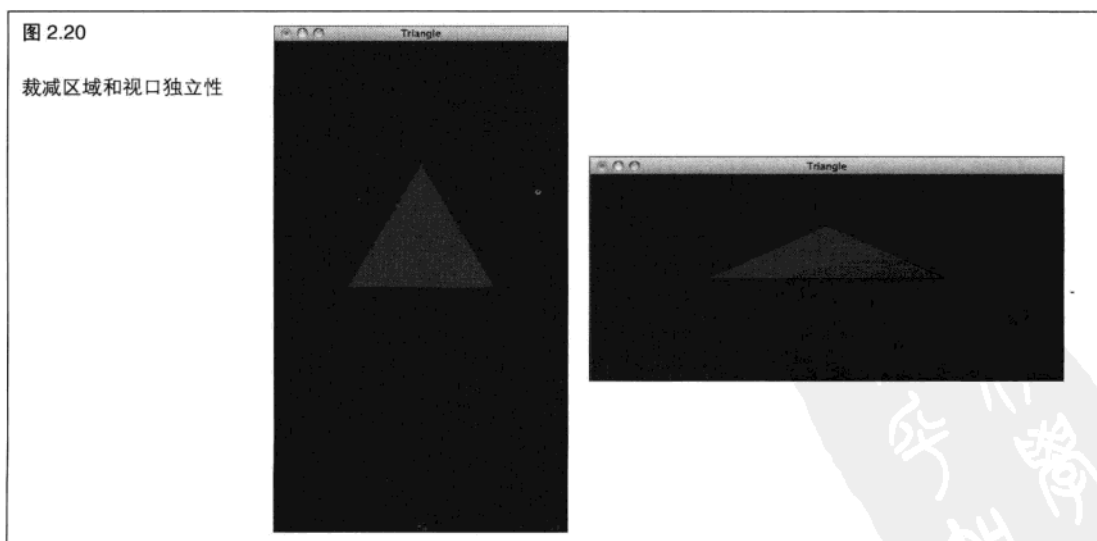
其中 `x` 参数和 `y` 参数代表窗口中视口的左下角坐标，而宽度和高度参数是用像素表示的。通常 `x` 和 `y` 都为 0，但是我们可以使用视口在窗口中的不同区域渲染多个图形。视口以实际屏幕坐标定义了窗口中的

区域，OpenGL 可以在这个区域中进行绘图（如图 2.19 所示）的裁剪区域被映射到新的视口。如果指定了一个比窗口坐标更小的视口，渲染区域就会缩小，如图 2.19 所示。



### 从笛卡尔坐标系到像素

在开始将几何图形光栅化（实际绘制）到屏幕上时，OpenGL 负责笛卡尔坐标系和窗口像素间的映射。我们要牢记一点，就是改变视口并不会改变基础坐标系。由于我们采用的是默认从-1 到+1 的映射，为三角形改变窗口大小会产生一些有趣的结果，如图 2.20 所示。



在图 2.20 左侧图中，我们可以看到+1 到-1 的范围在垂直方向是如何比水平方向延伸得更多的，而在右侧图中，则可以看到相反的效果。我们要先了解更多内容，然后才能考虑如何改变坐标系以响应窗口大小的变化，就像前面说过的，我们会在第 4 章完整地做完这项工作。

## 2.5.4 完成设置

在开始 main 函数中的 GLUT 主循环之前,我们先调用 SetupRC 函数。这时我们要为程序作一些一次性的设置。首先要做的就是通过以下调用来设置背景颜色。

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

这个函数设置用来进行窗口清除的颜色,它的函数原型如下所示。

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

在大多数 OpenGL 实现下, GLclampf 都被定义为一个浮点值。每个参数都包含最终颜色所要求的这种颜色分量的权值。这个函数不会立即清除背景,而是设置在以后颜色缓冲区被清除(可能是重复的)时使用的颜色。

### RGB 颜色空间

在 OpenGL 中,某种颜色是由红、绿、蓝和 Alpha(用于表示透明度)分量混合而成的。每种分量值的范围为从 0.0 至 1.0。这类似于在 Windows 中使用 RGB 宏创建 COLORREF 值的方式。不同的是,在 Windows 中, COLORREF 值的每种颜色成分范围在 0 至 255 之间,总共可以产生  $256 \times 256 \times 256$ (超过 1600 万)种颜色。在 OpenGL 中,每种成分的值可以是 0 至 1 之间任何有效的浮点值,因此理论上可以产生的颜色数量是无限的。从现实的角度讲,在绝大多数设备中,颜色值的输出限制在 24 位(1600 万种颜色)。

很自然,OpenGL 接受这个颜色值,并在内部把它转换为能够与可用的视频硬件准确匹配的最接近颜色。表 2.2 列出了一些常见的颜色以及它们的分量值。我们可以在任何与颜色相关的 OpenGL 函数中使用这些值。

表 2.4 一些常见的组合颜色

组合颜色	红色分量	绿色分量	蓝色分量
Black (黑)	0.0	0.0	0.0
Red (红)	1.0	0.0	0.0
Green (绿)	0.0	1.0	0.0
Yellow (黄)	1.0	1.0	0.0
Blue (蓝)	0.0	0.0	1.0
Magenta (洋红)	1.0	0.0	1.0
Cyan (青)	0.0	1.0	1.0
Dark gray (深灰)	0.25	0.25	0.25
Light gray (浅灰)	0.75	0.75	0.75
Brown (褐)	0.60	0.40	0.12

续表

组合颜色	红色分量	绿色分量	蓝色分量
Pumpkin orange (南瓜橙)	0.98	0.625	0.12
Pastel pink (粉红)	0.98	0.04	0.7
Barney purple (巴尼紫)	0.60	0.40	0.70
White (白)	1.0	1.0	1.0

`glClearColor` 的最后一个参数是 `alpha` 分量，它用来进行混合，并且可以产生一些特殊的效果，例如透明。透明是指一个物体允许光线穿过它。假定我们希望创建一块染成红色的玻璃，并且它的后面正好有一束蓝色的光。这道蓝光就会影响这块玻璃上的红色（蓝+红=紫）。我们可以用 `alpha` 成分值生成一种半透明的红色，使它看上去像是一块玻璃，它后面的物体也能够显示。这种类型的效果并不是仅仅靠使用 `alpha` 值就行了。在第 3 章，我们将详细讨论这个话题。在这之前，可以一直把 `alpha` 值设置为 1。

## 存储着色器

没有着色器，在 OpenGL 核心框架中就无法进行任何渲染。在第 6 章“跳出“盒子”：非存储着色器”中，我们将讨论如何编写着色器，以及如何编译和链接它们从而水它们可用。在那之前，我们先使用一些简单的存储着色器，它们可以用着色器管理器进行管理。我们要在源文件的开头部分声明一个着色器管理器的实例，如下所示：

```
GLShaderManager shaderManager;
```

我们也可以在第 3 章来熟悉这些着色器，并学习如何使用它们。但是着色器管理器需要编译和链接它自己的着色器，所以我们必须在 OpenGL 初始化时调用 `InitializeStockShaders` 方法。

```
shaderManager.InitializeStockShaders();
```

## 指定顶点

接下来我们要做的是设置三角形。在 OpenGL 中，三角形是一种“图元”类型，是一种基本的 3D 绘图元素。在第 3 章，我们会非常详细地讨论在 OpenGL 将会用到的所有 7 种图元。但在这里，我们只要了解一个三角形图元就是空间中的一系列组成一个三角形的顶点或点就可以了。我们通过将这些顶点放进一个单个浮点数组来指定它们。这个数组命名为 `vVerts`，其中包含所有 3 个顶点的 `x`、`y`、`z` 笛卡尔坐标对。请注意我们将所有 3 个点的 `z` 坐标都设为 0。

```
// Load up a triangle
GLfloat vVerts[] = { -0.5f, 0.0f, 0.0f,
                    0.5f, 0.0f, 0.0f,
                    0.0f, 0.5f, 0.0f };
```

在本书中有两章会讲解关于提交一个批次的顶点用于渲染的内容，即第 3 章和第 12 章，其中第 12 章将涉及更多的底层细节。一个简单的 GLTool 封装类 (Wrapper Class) 会将三角形顶点批次进行封装，

而我们则在源文件顶部附近声明一个这个 GLBatch 类的实例。

```
GLBatch triangleBatch;
```

在我们的设置函数中，下列代码建立了一个三角形的批次，仅包含 3 个顶点。在第 3 章我们将对此做进一步的讨论。

```
triangleBatch.Begin(GL_TRIANGLES, 3);  
triangleBatch.CopyVertexData3f(vVerts);  
triangleBatch.End();
```

## 2.5.5 言归正传

最后，我们终于可以真正开始渲染了！前面我们将清除颜色设为蓝色，现在我们需要执行一个函数真正进行清除。

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

glClear 函数清除一个或一组特定的缓冲区。缓冲区是一块存储图像信息的存储空间。红色、绿色、蓝色和 alpha 分量通常一起作为颜色缓冲区或像素缓冲区引用。

在 OpenGL 中有不止一种缓冲区（颜色缓冲区、深度缓冲区和模版缓冲区）供使用。在本书后面的内容中将详细介绍这些缓冲区。在前面的示例中，我们使用按位或（Bitwise OR）操作来同时清除所有这 3 种缓冲区。在接下来的几章中，我们真正需要理解的是，颜色缓冲区是显示图像在内部存储的地方，以及通过 glClear 清除缓冲区会将屏幕上最后绘制的内容删除。我们还会看到术语“帧缓冲区”（Framebuffer），指的是所有这些缓冲区一起串联工作。

下面的 3 行代码将真正执行操作，这也是整个第 3 章的主要课题！我们设置一组浮点数来表示红色（其 alpha 值设为 1.0），并将它传递到存储着色器，即 GLT\_SHADER\_IDENTITY 着色器。这个着色器只是使用指定颜色以默认笛卡尔坐标系在屏幕上渲染几何图形。

```
GLfloat vRed[] = { 1.0f, 0.0f, 0.0f, 1.0f };  
shaderManager.UseStockShader(GLT_SHADER_IDENTITY, vRed);  
triangleBatch.Draw();
```

GLBatch 的 Draw 方法指示将几何图形提交到着色器，然后……啊哈！——红色三角形……哦，差不多是。还有最后一个细节。当设置 OpenGL 窗口时，我们指定要一个双缓冲区渲染环境。这就意味着将在后台缓冲区进行渲染，然后在结束时交换到前台。这种形式能够防止观察者看到可能伴随着动画帧与动画帧之间闪烁的渲染过程。缓冲区交换将以平台特定的方式进行，但是 GLUT 有一个单独的函数调用可以完成这项工作。

```
glutSwapBuffers();
```

现在我们可以鞠躬谢幕了。我们已经用 OpenGL 渲染了第一个三角形。

## 2.6 加点儿活力!

现在我们已经了解如何使 GLUT 完成一个图形演示框架所能完成的最重要工作，也就是在屏幕上渲染图形了。我们可以再加入一点小功能，使它能让用户对渲染进行一些互动，例如通过按箭头键移动图形。一点动画效果就能使图形演示变得有活力。示例程序的“Move”就能做到这一点。它在窗口的正中绘制了一个正方形（实际上我们使用了另一个图元，这次是 `GL_TRIANGLE_FAN`）。在按箭头键时，正方形将上下或左右移动。分别用向上箭头和向下箭头中的哪一个来实现这两种运动则由读者决定。

### 2.6.1 特殊按键

GLUT 还提供了另一个回调函数，即 `glutSpecialFunc`。它注册了一个能够在按一个特殊按键时被调用的函数。在 GLUT 的语法中，特殊按键是指功能键或者方向键（上、下、左、右箭头键，page up/down 键等）中的一个。在主函数中加入下面的代码行，来注册 `SpecialKeys` 回调函数。

```
glutSpecialFunc(SpecialKeys);
```

它在按键时接受一个相应的按键编码，以及在使用鼠标时光标的  $x$  和  $y$  坐标位置（像素形式）。

在“Move”示例程序中，我们将顶点存储在一个全局（对于这个模型来说）数组中，这样我们就能够在按键时相应修正正方形的位置了。程序清单 2.2 展示了 `SpecialKeys` 函数的完整代码，这里我们还进行了碰撞检测，这样正方形就不会移出窗口范围了。请注意，我们可以轻松地更新批次位置，只需复制新的顶点数据即可。

```
squareBatch.CopyVertexData3f(vVerts);
```

**程序清单 2.2 用箭头键操纵正方形在屏幕范围内移动**

```
// Respond to arrow keys by moving the camera frame of reference
void SpecialKeys(int key, int x, int y)
{
    GLfloat stepSize = 0.025f;

    GLfloat blockX = vVerts[0]; // Upper left X
    GLfloat blockY = vVerts[7]; // Upper left Y

    if(key == GLUT_KEY_UP)
        blockY += stepSize;

    if(key == GLUT_KEY_DOWN)
        blockY -= stepSize;

    if(key == GLUT_KEY_LEFT)
        blockX -= stepSize;

    if(key == GLUT_KEY_RIGHT)
        blockX += stepSize;
```

```
// Collision detection
if(blockX < -1.0f) blockX = -1.0f;
if(blockX > (1.0f - blockSize * 2)) blockX = 1.0f - blockSize * 2;;
if(blockY < -1.0f + blockSize * 2) blockY = -1.0f + blockSize * 2;
if(blockY > 1.0f) blockY = 1.0f;

// Recalculate vertex positions
vVerts[0] = blockX;
vVerts[1] = blockY - blockSize*2;

vVerts[3] = blockX + blockSize*2;
vVerts[4] = blockY - blockSize*2;

vVerts[6] = blockX + blockSize*2;
vVerts[7] = blockY;

vVerts[9] = blockX;
vVerts[10] = blockY;

squareBatch.CopyVertexData3f(vVerts);

glutPostRedisplay();
}
```

## 2.6.2 刷新显示

SpecialKeys 函数的最后一行代码用来告诉 GLUT 需要更新窗口内容。

```
glutPostRedisplay();
```

默认情况下，在窗口创建、改变大小或者需要重绘时，GLUT 通过调用 RenderScene 函数来更新窗口。只要窗口发生最小化、恢复、最大化、覆盖或重新显示等变化，就会发生更新。我们可以人工调用 glutPostRedisplay 来告诉 GLUT 发生了某些改变，应该对场景进行渲染了。不过，用后面将要介绍的方法来完成这项工作尤其方便。

## 2.6.3 简单的动画片

在“Move”示例中，当我们按箭头键时，更新了几何图形位置，然后调用 glutPostRedisplay 函数激活屏幕刷新动作。如果我们将 glutPostRedisplay 函数调用在 RenderScene 函数末尾将会发生什么呢？如果读者想到的是得到一个持续自动刷新的程序，那么恭喜，答对了。但是不要担心，这并不是一个无限循环。重绘消息实际是一条传递到一个内部消息循环中的消息，在屏幕刷新的间隔中，也会发生其他窗口事件。这就是说，我们仍然可以检测按键动作、鼠标移动、改变窗口大小和程序结束等动作。

程序清单 2.3 显示了我们在“Move”示例程序中的 RenderScene 函数经过修改后的代码。

程序清单 2.3

```
////////////////////////////////////
// Called to draw scene
void RenderScene(void)
```

```

{
// Clear the window with current clearing color
glClearColor(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

GLfloat vRed[] = { 1.0f, 0.0f, 0.0f, 1.0f };
shaderManager.UseStockShader(GLT_SHADER_IDENTITY, vRed);
squareBatch.Draw();

// Flush drawing commands
glutSwapBuffers();

BounceFunction();
glutPostRedisplay(); // Redraw
}

```

## 2.7 总结

在本章，我们讨论了许多基础知识。我们介绍了 OpenGL，并简单地介绍了它的历史。另外，还介绍了 OpenGL 工具箱（GLUT），并讨论了使用 OpenGL 编写程序的基础知识。利用 GLUT，我们还展示了创建窗口并使用 OpenGL 命令在窗口中进行绘图的最简便方法，并学习了如何使用 GLUT 函数库创建一个可以改变大小的窗口，并创建了一个简单的动画示例程序。此外，我们还介绍了使用 OpenGL 进行绘图的过程——合成和选择颜色、清除屏幕、绘制三角形和矩形并在窗口帧中设置视口。我们还讨论了各种 OpenGL 数据类型以及生成 OpenGL 程序所需要的头文件，并引导读者分别在 Visual Studio（Windows 系统）和 Xcode（Mac OS X）系统中创建项目。

OpenGL 状态机奠定了我们以后用到的几乎所有操作的基础；扩展机制使我们能够访问自己所使用的硬件驱动程序支持的所有 OpenGL 特性，而不必考虑自己所使用的是什么开发工具。我们还学习了如何检查 OpenGL 错误，确保程序中并未出现任何非法的状态改变或渲染命令。稍微再熟悉一些代码以后，读者就可以掌握进一步学习所需的一些知识了。



## 第 3 章 基础渲染

作者: Richard S. Wright, Jr.

### 本章内容

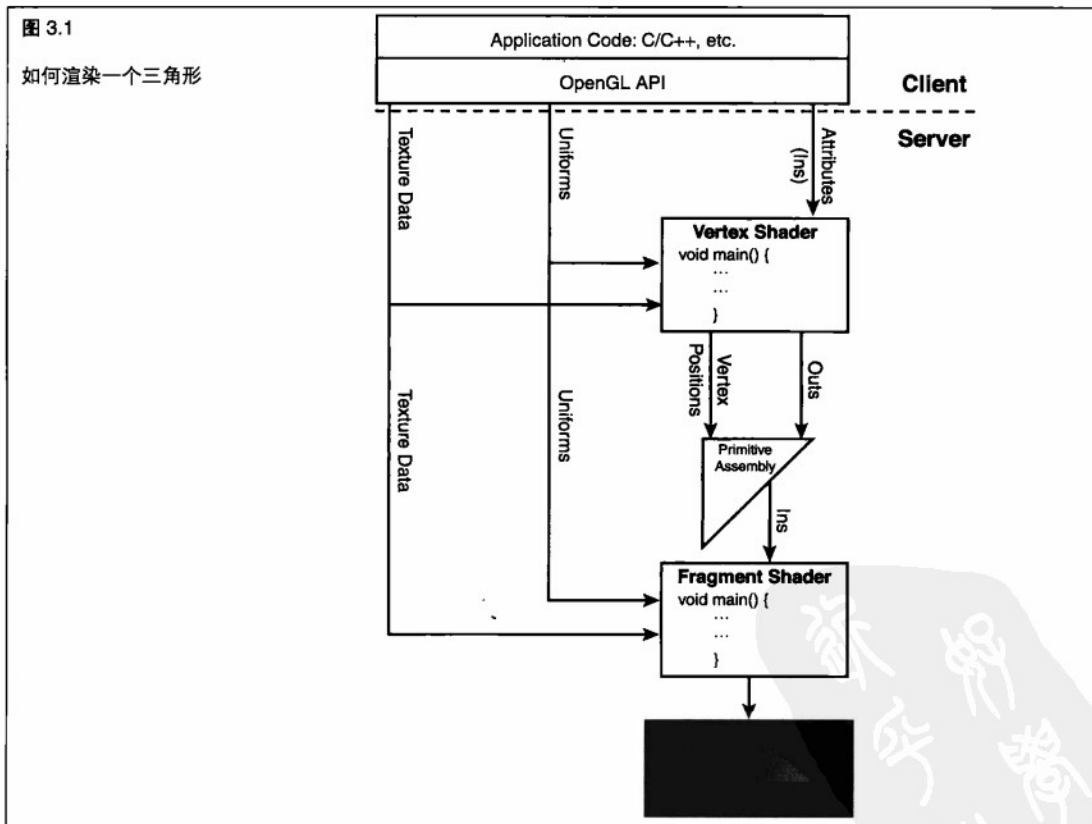
- ✦ OpenGL 渲染基础架构
- ✦ 如何使用 7 种 OpenGL 几何图元
- ✦ 如何使用存储着色器
- ✦ 如何使用 uniform 值和属性
- ✦ 如何使用 GLBatch 帮助类传递几何图形
- ✦ 如何执行深度测试和背面消除
- ✦ 如何绘制透明或混合几何图形
- ✦ 如何绘制抗锯齿点、线和多边形

如果读者曾经学习过化学课(即使没学过化学也可能知道),应该知道所有的物质都是由原子组成的,而且所有的原子只由 3 样东西组成:质子、中子和电子。我们在生活中所接触过的所有材料和物质,从玫瑰花瓣到沙滩上的沙子,都是由这 3 种基本成分组成的,只是在排列上有所不同而已。尽管对于小学三四年级文化程度的人而言,这样的解释未免显得简单,但它说明了一个放之四海而皆准的道理:可以用一些简单的建筑块创建高度复杂和漂亮优雅的结构。

这个道理和 OpenGL 的联系是显而易见的。OpenGL 所创建的物体和场景也是由更小、更简单的形状所组成的,并且是按照各种各样独一无二的方式进行排列和组合的。本章将讨论组成 3D 物体的这些建筑块,称为图元(Primitives)。在 OpenGL 中,所有的图元都是一维、二维或三维的物体,从简单的点和线直到复杂的多边形都是如此。在本章中,读者将学习用这些简单的形状在三维空间中绘制各种物体时所需的各种知识。

## 3.1 基础图形管线

OpenGL 中的图元只不过是顶点的集合以预定义的方式结合在一起罢了。例如，一个单独的点就是一个图元，它只需要一个顶点。三角形则是另外一个例子，它是由 3 个顶点组成的图元。在我们讨论不同种类的图元之前，先来看看一个图元是如何由读独立的顶点组合而成的。基础渲染管线接受 3 个顶点并将它们转换成一个三角形。它还可能应用颜色、一个或多个纹理并且移动它们的位置。这种管线也是可编程的，实际上我们编写了两个程序，图形硬件执行它们来处理顶点数据并在屏幕上填充像素【我们称它们为片段（复数），因为实际上在同一个像素上的片段可能会有不止一个】。为了便于理解 OpenGL 上的这种基础处理工作，让我们先来看看 OpenGL 渲染管线的一个简化版本，如图 3.1 所示。



### 3.1.1 客户机-服务器

首先请注意我们将管线分成了两部分。上半部分是客户端，而下半部分则是服务器端。这种基本的客户机-服务器设计是在管线的客户端在功能上从服务器端分离出来时开始采用的。就 OpenGL 而言，

客户端是存储在 CPU 存储器中的，并且在应用程序中执行，或者在主系统内存的驱动程序中执行。驱动程序将渲染命令与数据组合起来，并发送到服务器执行。在一台典型的桌面计算机上，服务器会跨越一些系统总线，实际上，它就是图形加速卡上的硬件和内存。

服务器和客户机在功能上也是异步的，也就是说它们是各自独立的软件块或硬件块，或者软硬件都有。为了获得最佳性能，我们希望两方面都尽可能不停地工作。客户机不断地将数据块和命令块组合在一起并送入缓冲区，然后这些缓冲区会发送到服务器执行。服务器将执行这些缓冲区的内容，与此同时客户端又做好了发送下一个用于渲染的数据或信息的准备。如果服务器停止工作等待客户机，或者客户机停止工作来等待服务器做好接受更多命令和数据的准备，我们就把这种情况称为管线停滞。管线停滞是注重性能的程序员的噩梦，我们真的不希望 CPU 或者 GPU 无所事事地等待工作。

### 3.1.2 着色器

图 3.1 所示的两个最大的框图代表顶点着色器和片段着色器。着色器是使用 GLSL（我们将在第 6 章讨论 GLSL 编程）编写的程序。GLSL 看起来与 C 语言非常类似，实际上 GLSL 语言的程序甚至是以我们熟悉的 main 函数开始的。这些着色器必须从源代码中编译和链接到一起（这一点仍然和 C、C++ 程序非常类似）才能使用。最终准备就绪的着色器程序随后在第一阶段构成顶点着色器，在第二阶段构成片段着色器。请注意，我们目前讨论的是简化的方式。实际上还有一种几何着色器可以（选择性地）安排在两者之间，就像用来将数据来回传递的所有类型的反馈机制一样。还有一些传递片段的处理特性，诸如混合、模板和深度测试，我们将在后面进行讨论。

顶点着色器处理从客户机输入的数据，应用变换，或者进行其他类型的数学运算来计算光照效果、位移、颜色值，等等。为了渲染一个共有 3 个顶点的三角形，顶点着色器将执行 3 次，也就是为每个顶点执行一次。在目前的硬件上有多个执行单元同时运行，这就意味着所有这 3 个顶点都可以同时进行处理。今天的图形处理器属于大规模并行计算机。不要将它们和 CPU 相比而被时钟速度蒙蔽，它们比图形操作要快上几个数量级。

现在，3 个顶点都做好了光栅化的准备。图 3.1 所示的图元组合（Primitive Assembly）框图意在说明 3 个顶点已经组合在一起，而三角形已经逐个片段地进行了光栅化。每个片段都通过执行片段着色器而进行了填充，片段着色器会输出我们将在屏幕上看到的最终颜色值。再强调一次，今天的硬件是大规模并行运算的，同时执行上百个甚至更多的这种片段程序并不困难。

当然我们必须首先为这些着色器提供数据，否则什么也做不成。有 3 种向 OpenGL 着色器传递渲染数据方法可供程序员选择，即属性、uniform 值和纹理。

#### 属性

所谓属性就是一个对每个顶点都要作改变的数据元素。实际上，顶点位置本身就是一个属性。属性值可以是浮点数、整数或布尔数据。属性总是以四维向量的形式进行内部存储的，即使我们不会使用到所有 4 个分量。例如，一个顶点位置可能存储为一个  $x$  值、一个  $y$  值和一个  $z$  值，将占用 4 个分量中的 3 个。OpenGL

会将第 4 个分量 ( $w$  分量, 如果读者想知道的话) 设为 1。实际上, 如果我们只在  $xy$  平面 (忽略  $z$ ) 中绘图, 那么第 3 个分量将会自动设为 0, 而第 4 个分量依旧设为 1。在我们创建的任何属性中都会进行这种默认为, 而不仅仅是在顶点位置中, 所以在我们不会用到所有 4 个可用分量时, 一定要多加注意。除了顶点的空间位置之外, 还有其他一些可能要逐个顶点修改的属性, 包括纹理坐标、颜色值和用于光照计算的表面法线。不过, 在顶点程序中, 属性可以代表我们想要任何意义——这些都在我们掌握之中。

属性会从本地客户机内存中复制存储在图形硬件中 (这种情况的可能性最大) 的一个缓冲区上。这些属性只供顶点着色器使用, 对于片段着色器来说没什么意义。还要声明一点, 这些属性对每个顶点都要做改变, 并不意味着它们的值不能重复, 而只是说明对于每个顶点都有一个实际存储值。当然, 通常情况下它们都是不同的, 但是也有可能会有整个数组都是同一个值的情况。但是这种情况是非常浪费的, 而且如果我们需要在某个批次中都是同一个值的数据元素, 还有更好的解决方案。

## Uniform 值

所谓属性就是一个对每个顶点都要做改变的数据元素。实际上, 顶点位置本身就是一个属性。属性是一种对于整个批次的属性都取统一值的单个值, 也就是说, 它是不变的。我们通常设置完 Uniform 变量就紧接着发出渲染一个图元批次的命令。Uniform 变量实际上可以无次数限制地使用, 我们可以设置一个应用于整个表面的单个颜色值, 还可以设置一个时间值, 在每次渲染某种类型的顶点动画时修改它 (请注意, 这里的 Uniform 变量在每个批次改变一次, 而不是每个顶点改变一次)。Uniform 变量一个最常见的应用是在顶点渲染中设置变换矩阵 (第 4 章将介绍)。

Uniform 值在本质上像属性一样, 可以是浮点值、整数或布尔值, 但和属性不同的是, 顶点着色器和片段着色器中都可以有 Uniform 变量。Uniform 变量可以既可以是标量类型, 也可以是矢量类型, 我们也可以使用 Uniform 矩阵。从技术上说, 我们也可以使用属性矩阵, 矩阵中每一列对应 4 个分量中的一个, 但是通常我们不这么做。在第 5 章, 我们甚至还会讨论一些特殊的 Uniform 值设置函数来处理这些问题。

## 纹理

我们可以传递到着色器的第三种数据类型是纹理数据。现在试图细致地讨论如何处理纹理并将其传递到着色器的细节还为时过早, 但是我们在第一章中已经大致了解了什么是纹理。从顶点着色器和片段着色器中都可以对纹理值进行采样和筛选。典型情况下, 片段着色器对一个纹理进行采样, 并在一个三角形的表面上应用图形数据。但是, 纹理数据的作用并不仅仅是表现图形。很多图形文件格式都是以无符号字节 (每个颜色通道 8 位) 形式对颜色分量进行存储的, 但是我们仍然可以设置浮点纹理。这就是说, 任何大型浮点数据块 (例如消耗资源很大的函数的大型查询表) 都可以通过这种方式传递给着色器。

## 输出

图 3.1 所示图表中的第 4 种数据类型是输出 (Out)。输出数据是作为一个阶段着色器的输出定义的, 而在后续阶段的着色器则是作为输入 (In) 定义的。输出类型的数据可以简单地从一个阶段传递到下一个阶段, 也可以以不同的方式插入。客户端的代码接触不到这些内部变量, 但是它们在顶点着色器和片段着

色器中（还可能包括可选的几何着色器）都进行了声明。顶点着色器为输出变量分配一个值，这个值是常量，也可以在图元被光栅化时插入到顶点之间。片段着色器对应的同名输入值接受这个常量或插入值。在第6章我们将更详细地讨论相关细节。

## 3.2 创建坐标系

在第一章，我们介绍了3D图形中常用的两种投影（正投影和透视投影）。这些投影，或者说坐标系类型，实际上只是一种特定的 $4 \times 4$ 变换矩阵。这些矩阵和其他一些类型的矩阵将在下一章进行讨论，因此我们在这里不急于在这些细节上过分纠缠。总而言之，我们需要这3种矩阵类型中的一种来在适当的坐标系中渲染几何图形。如果我们不采用这些矩阵中的一种，则会默认获得一个坐标范围在-1.0到+1.0之间的正投影。我们在第二章的示例程序中应用了这种坐标系，而且这非常有用的，因为所有这3个示例程序都是2D的。不过，在本章，我们将更进一步。

Math3d库是GLTools包含的函数的一部分，它为我们构建了不同种类的矩阵，我们将会在第4章学习如何运用它们。在本章，我们使用GLFrustum类来作为投影矩阵的容器。

### 3.2.1 正投影

我们通常在2D绘图中使用正投影，并在我们的几何图形中将 $z$ 坐标设为0.0。但是， $z$ 轴可以延伸到任何我们想要的长度。图3.2所示是一个正投影的例子，在所有3个轴（ $x$ 、 $y$ 和 $z$ ）中，它们的范围都是从-100至+100。这个视景体（经常被这样叫）将包括所有的几何图形。如果我们指定了视景体外的几何图形，那么它将被裁剪掉，也就是说，它将被沿着视景体的边界进行剪切。在正投影中，所有在这个空间范围内的所有东西都会被显示在屏幕上，而不存在照相机或视点坐标系的概念。我们通过调用GLFrustum方法来完成上述工作。

```
GLFrustum::SetOrthographic(GLfloat xMin, GLfloat xMax, GLfloat yMin,  
                           GLfloat yMax, GLfloat zMin, GLfloat zMax);
```

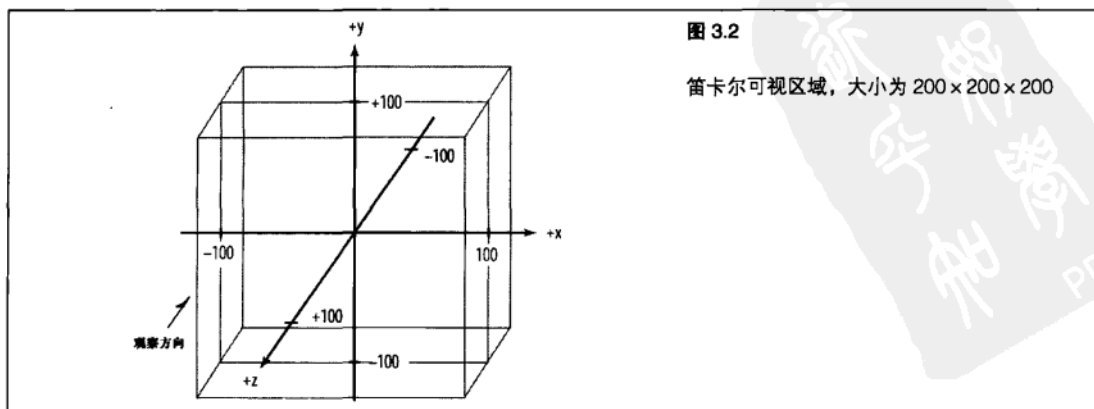
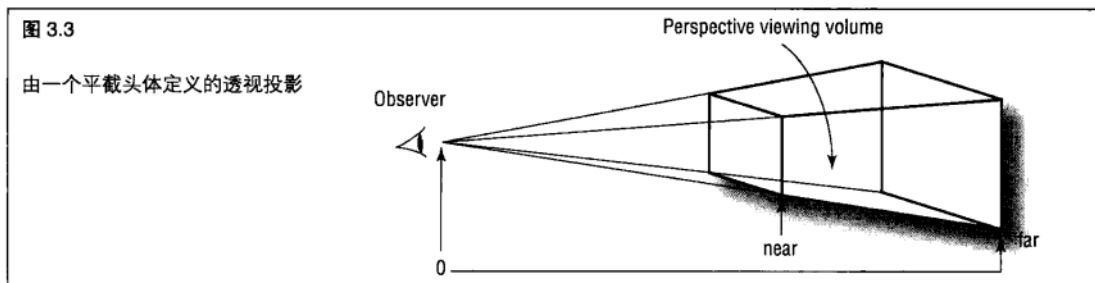


图 3.2

笛卡尔可视区域，大小为 $200 \times 200 \times 200$

## 3.2.2 透视投影

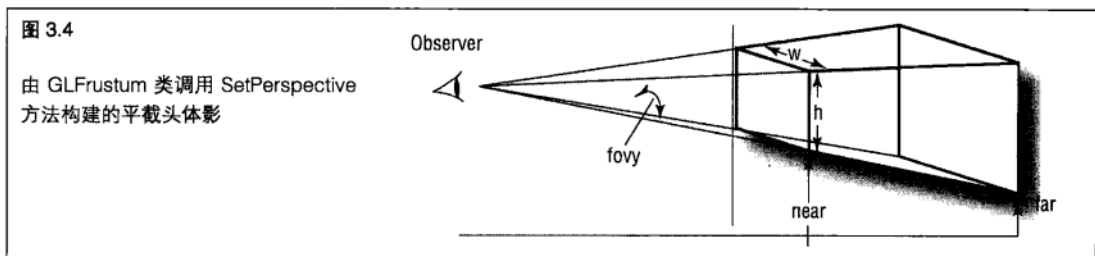
透视投影会进行透视除法对距离观察者很远的对象进行缩短和收缩。在投影到屏幕之后，视景体背面与视景体正面的宽度测量标准不同。这样，逻辑尺寸相同的对象绘制在视景体的前面比绘制在视景体的背面显得更大。图 3.3 所示是一个叫做平截头体（frustum）的几何体的示例。平截头体是一个金字塔形被截短之后的形状，它的观察方向是从金字塔的尖端到宽阔端，观察者的视点与金字塔的尖端拉开一定距离。



GLFrustum 类通过调用 SetPerspective 方法为我们构建一个平截头体。

```
GLFrustum::SetPerspective(float fFov, float fAspect, float fNear, float fFar);
```

其中的参数分别在垂直方向上的视场角度，窗口的宽度与高度的纵横比，以及到近裁剪面和远裁剪面之间的距离（如图 3.4 所示）。我们用宽度除以高度就能得到窗口或视口的纵横比。



## 3.3 使用存储着色器

在 OpenGL 核心框架中，并没有提供任何内建渲染管线，在提交一个几何图形进行渲染之前，必须制定一个着色器。这在学习图形程序设计中带来了类似于鸡和蛋的问题（是先有鸡还是先有蛋）。我们认为最好的方式是提供一系列在本书第一部分可以使用的存储着色器。这些存储着色器由 GLTools 的 C++ 类 GLShaderManager 进行管理，它们能够满足进行通常渲染的基本要求。要求不高的程序员甚至可能发现，这些存储着色器已经足以满足他们所有的要求。但是，随着时间的推移，除了最漫不经心的业余程序员外，所有人都不太可能一直满足于此，这种情况大概会在我们开始学习第 6 章时发生。

GLShaderManager 在使用前必须进行初始化，我们可以在所有示例程序源文件顶部附近找到 GLShaderManager 和调用 InitializeStockShaders 的例子。

```
shaderManager.InitializeStockShaders();
```

### 3.3.1 属性

OpenGL 支持多达 16 种可以为每个顶点设置的不同类型参数。这些参数编号为从 0 到 15，并且可以与顶点着色器中的任何指定变量相关联（第 6 章将介绍如何进行这项工作）。存储着色器为每个变量都使用一致的内部变量命名规则和相同的属性槽（Attribute Slot）。表 3.3 列出了这些属性。

表 3.1 GLShaderManager 预定义的标识符

标识符	描述
GLT_ATTRIBUTE_VERTEX	3 分量 (x, y, z) 顶点位置
GLT_ATTRIBUTE_COLOR	4 分量 (r, g, b, a) 颜色值
GLT_ATTRIBUTE_NORMAL	3 分量 (x, y, z) 表面法线
GLT_ATTRIBUTE_TEXTURE0	第一对 2 分量 (s, t) 纹理坐标
GLT_ATTRIBUTE_TEXTURE1	第二对 2 分量 (s, t) 纹理坐标

### 3.3.2 Uniform 值

要对几何图形进行渲染，我们需要为对象递交属性矩阵，但首先要绑定到我们想要使用的着色器程序上，并提供程序的 Uniform 值。GLShaderManager 类可以（暂时）为我们完成这项工作。UseStockShader 函数会选择一个存储着色器并提供这个着色器的 Uniform 值，这些工作通过一次函数调用就能完成。

```
GLShaderManager::UseStockShader(GLenum shader, .....);
```

在 C 语言（或 C++ 语言）中，..... 表示函数接受一个可变的参数数量。就这个函数本身而言，它根据我们选择的着色器从堆栈中提取正确的参数，这些参数就是特定着色器要求的 Uniform 值。后面我们将逐个讨论这些着色器。虽然现在还看不出这些 Uniform 值有什么意义，但当学习完本书的介绍部分后就会明白了。我们应该将这部分内容做重点标记，因为我们会发现自己将不时地回过头来参考这些内容。

#### 单位（Identity）着色器

单位（Identity）着色器只是简单地使用默认的笛卡尔坐标系（在所有坐标轴上的坐标范围都是 -1.0 ~ 1.0）。所有片段都应用同一种颜色，几何图形为实心 and 未渲染的。这种着色器只使用一个属性 GLT\_ATTRIBUTE\_VERTEX。vColor 参数包含了要求的颜色。

```
GLShaderManager::UseStockShader(GLT_SHADER_IDENTITY, GLfloat vColor[4]);
```

## 平面着色器

平面 (Flat) 着色器将统一着色器进行了扩展, 允许为几何图形变换指定一个  $4 \times 4$  变换矩阵。典型情况下这是一种左乘模型视图矩阵和投影矩阵, 经常被称作“模型视图投影矩阵”(这部分内容在第 4 章将进一步介绍)。这种着色器只使用一个属性 GLT\_ATTRIBUTE\_VERTEX。

```
GLShaderManager::UseStockShader(GLT_SHADER_FLAT, GLfloat mvp[16], GLfloat
vColor[4]);
```

## 上色 (Shaded) 着色器

这种着色器唯一的 Uniform 值就是在几何图形中应用的变换矩阵。GLT\_ATTRIBUTE\_VERTEX 和 GLT\_ATTRIBUTE\_COLOR 在这种着色器中都会使用。颜色值将被平滑地插入顶点之间 (称为平滑着色)。

```
GLShaderManager::UseStockShader(GLT_SHADER_SHADED, GLfloat mvp[16]);
```

## 默认光源着色器

这种着色器创造出一种错觉, 类似于由位于观察者位置的单漫射光所产生的效果。从本质上讲, 这种着色器使对象产生阴影和光照的效果。这里需要模型视图矩阵、投影矩阵和作为基本色的颜色值等 Uniform 值。所需的属性有 GLT\_ATTRIBUTE\_VERTEX 和 GLT\_ATTRIBUTE\_NORMAL。大多数光照着色器都需要正规矩阵 (normal matrix) 作为 Uniform 值。着色器从模型视图矩阵中推导出了正规矩阵——很方便, 但是效率不太高。在性能敏感的应用程序中, 我们要充分考虑这一点。

```
GLShaderManager::UseStockShader(GLT_SHADER_DEFAULT_LIGHT, GLfloat mvMatrix[16],
GLfloat pMatrix[16], GLfloat vColor[4]);
```

## 点光源着色器

点光源着色器和默认光源着色器很相似, 但是光源位置可能是特定的。这种着色器接受 4 个 Uniform 值, 即模型视图矩阵、投影矩阵、视点坐标系中的光源位置和对象的基本漫反射颜色。所需的属性有 GLT\_ATTRIBUTE\_VERTEX 和 GLT\_ATTRIBUTE\_NORMAL。

```
GLShaderManager::UseStockShader(GLT_SHADER_POINT_LIGHT_DIFF, GLfloat mvMatrix[16],
GLfloat pMatrix[16], GLfloat vLightPos[3], GLfloat vColor[4]);
```

## 纹理替换矩阵

着色器通过给定的模型视图投影矩阵, 使用绑定到 nTextureUnit 指定的纹理单元的纹理对几何图形进行变换。片段颜色是直接从纹理样本中直接获取的。所需的属性有 GLT\_ATTRIBUTE\_VERTEX 和 GLT\_ATTRIBUTE\_NORMAL。

```
GLShaderManager::UseStockShader(GLT_SHADER_TEXTURE_REPLACE,  
                                GLfloat mvpMatrix[16],GLint nTextureUnit);
```

### 纹理调整着色器

这种着色器将一个基本色乘以一个取自纹理单元 `nTextureUnit` 的纹理。所需的属性有 `GLT_ATTRIBUTE_VERTEX` 和 `GLT_ATTRIBUTE_TEXTURE0`。

```
GLShaderManager::UseStockShader(GLT_SHADER_TEXTURE_MODULATE, GLfloat mvpMatrix[16],  
                                GLfloat vColor, GLint nTextureUnit);
```

### 纹理光源着色器

这种着色器将一个纹理通过漫反射照明计算进行调整（相乘），光线在视觉空间中的位置是给定的。这种着色器接受 5 个 Uniform 值，即模型视图矩阵、投影矩阵、视觉空间中的光源位置、几何图形的基本色和将要使用的纹理单元。

所需的属性有 `GLT_ATTRIBUTE_VERTEX`、`GLT_ATTRIBUTE_NORMAL` 和 `GLT_ATTRIBUTE_TEXTURE0`。

```
GLShaderManager::UseStockShader(GLT_SHADER_TEXTURE_POINT_LIGHT_DIFF,  
                                GLfloat mvMatrix, GLfloat pMatrix[16], GLfloat vLightPos[3],  
                                GLfloat vBaseColor[4], GLint nTextureUnit);
```

## 3.4 将点连接起来

第一次（如果以前学过的话）学习在任何计算机系统中绘制任何类型的 2D 图形时，很可能都会从绘制像素开始。像素是计算机屏幕上显示的最小元素，在彩色系统中这个像素可以是许多可用颜色中的任意一种。以下是最简单的计算机图形：在计算机屏幕上绘制一个点，并将它设置一个特定的颜色。在这个简单概念的基础上，使用我们最喜欢的计算机语言来创建线、多边形、圆和其他形状与图形——甚至可以是一个 GUI。

然而，使用 OpenGL 在计算机屏幕上进行绘图则完全不同。

我们关心的不是物理屏幕坐标和像素，而是视景体中的位置坐标。将这些点、线和三角形从创建的 3D 空间投影到计算机屏幕上的 2D 图形则是着色器程序和光栅化硬件所要完成的工作。

### 3.4.1 点和线

我们将从 7 个由定义的几何图元来开始绘制实心几何图形。

这些图元将在一个包含给定图元的所有顶点和相关属性的单个批次中进行渲染。实质上，在一个给定的批

次中的所有定点都会用于组成这些图元中的一个。表 3.2 列出了这 7 个图元并简要说明它们的特性。

表 3.2                      OpenGL 几何图元	
图 元	描 述
GL_POINTS	每个顶点在屏幕上都是一个单独的点
GL_LINES	每一对顶点定义了一个线段
GL_LINE_STRIP	一个从第一个顶点依次经过每个后续顶点而绘制的线条
GL_LINE_LOOP	和 GL_LINE_STRIP 相同，但最后一个顶点和第一个顶点也连接了起来
GL_TRIANGLES	每 3 个顶点定义了一个新的三角形
GL_TRIANGLE_STRIP	共用一个条带（strip）上的顶点的一组三角形
GL_TRIANGLE_FAN	以一个圆点为中心呈扇形排列，共用相邻顶点的一组三角形

示例程序展示了以上每个图元的渲染效果图。运行示例程序，并按空格键，程序将依次从 GL\_POINTS 到 GL\_TRIANGLE\_STRIP 进行演示。我们还可以使用方向键来沿 x 轴和 y 轴旋转这些渲染效果图。

点

点是最简单的图元，每个特定的顶点在屏幕上都仅仅是一个单独的点。默认情况下，点的大小为一个像素。我们可以通过调用 glPointSize 改变默认点的大小。

```
void glPointSize(GLfloat size);

glPointSize 函数接受一个参数，这个参数指定绘制点的近似尺寸（用像素表示）。然而，并不是所有点的大小都能够支持，而我们则应确认指定的点大小是可用的。使用下面的函数可以获得点大小的范围，以及它们之间的最小间隔。

GLfloat sizes[2]; //存储支持的点大小范围
GLfloat step;     //存储支持的点大小增量

//获取支持的点大小范围和步长（增量）
glGetFloatv(GL_POINT_SIZE_RANGE,sizes);
glGetFloatv(GL_POINT_SIZE_GRANULARITY,&step);
```

在这里，大小值数组由两个元素组成，其中包含了 glPointSize 的最小和最大可用值。此外，变量 step 保存了相邻点大小值之间所允许的最小步长。OpenGL 规范只需要支持一个点大小，即 1.0，但大多数实现都支持很大范围内的各种点大小。例如，我们可以找到从 0.5 到 256.0 之间，最小步长为 0.125 的各种点大小。

指定一个允许范围之外的点大小并不会被认为是一个错误。相反，在这种情况下将根据哪个值离指定值最近来使用所允许的最大值或最小值代替。

在默认情况下，点大小和其他几何图形不同，它并不会受到透视除法的影响。

也就是说，当它们离视点更远时，它们看上去并不会变得更小；如果它们离视点更近，它们看上去也不会变得更大一些。另外，点总是正方形的像素，即便使用 glPointSize 增加点的大小，情况也不会发生

变化。我们只能得到更大的正方形！为了获得圆点，我们必须在抗锯齿模式下绘制点（本章稍后将进一步介绍）。

还可以通过使用程序点大小模式来设置点大小。

```
glEnable(GL_PROGRAM_POINT_SIZE);
```

这种模式下允许我们通过编程在顶点着色器或几何着色器中设置点大小，这两种着色器超出了本章所涉及的范围。为了描述的完整，我们还要说明一下，着色器内建变量为 `gl_PointSize`，并且在着色器源代码中，只要如下设置即可。

```
gl_PointSize=5.0;
```

图 3.5 所示是图元示例程序的初始输出。其中点大小被设为 4.0，并且使用了一个顶点数组来组成一个大致类似于佛罗里达州的形状。

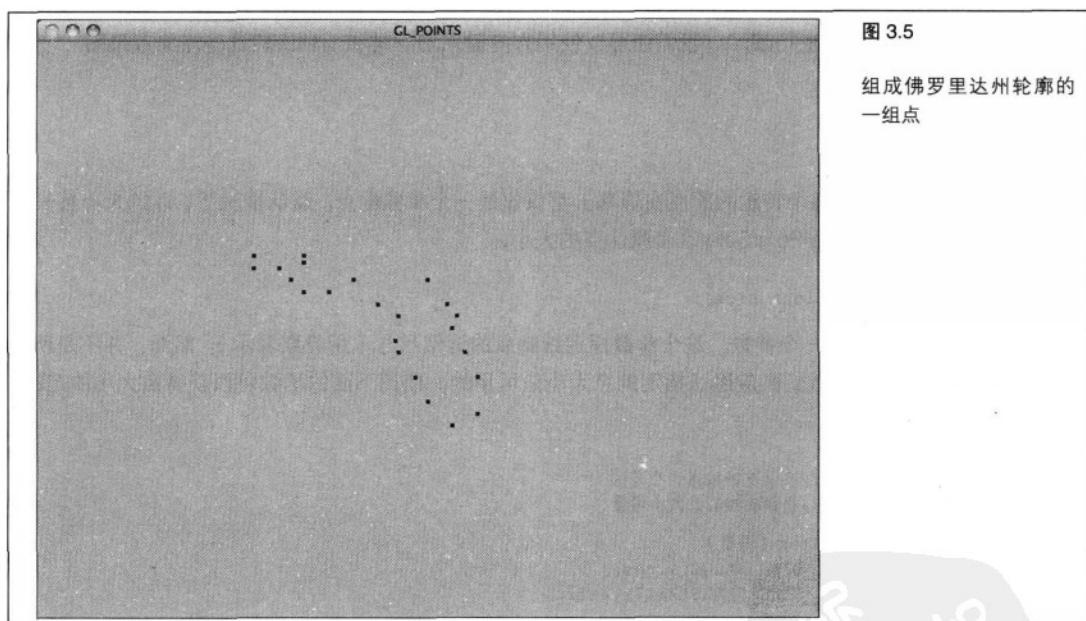


图 3.5

组成佛罗里达州轮廓的一组点

## 线

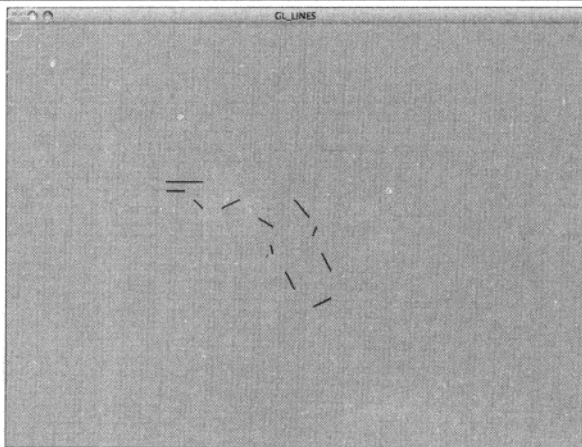
比点更进一步的是独立的线段。一条线段是在两个顶点之间绘制的，所以一批线段应该包括偶数个顶点，每个顶点都是线段的端点。如果我们使用在前面图形中使用过的同一组点来绘制一系列线段，那么沿着佛罗里达轮廓的每两个点都会组成一个新的线段。这种方式当然会在每个单独的线段之间留下缺口，如图 3.6 所示。

默认情况下，线段的宽度为一个像素。改变线段宽度的唯一方式是使用函数 `glLineWidth`。

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

图 3.6

独立线段，每一条都由两个顶点组成



### 线带

线带 (line strip) 连续地从一个顶点到下一个顶点绘制线段，以形成一个真正连接点的线条。为了用独立的线段围绕佛罗里达州的轮廓形成一个连续的线条，每个连接顶点都会被选定两次。其中一次是作为一条线段的终点，另一次则是作为下一条线段的起点。

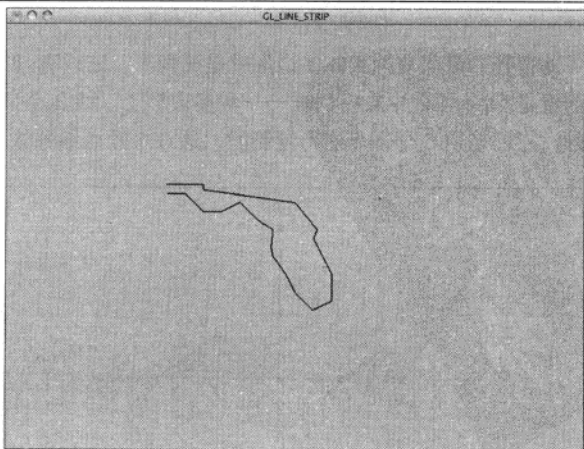
移动所有这些额外的数据并对同一个点进行两次转换，对于 GPU 的带宽和时钟来说都是一种浪费。如图 3.7 所示，再次显示了同一组点，这次是作为 `GL_LINE_STRIP` 绘制的。

### 线环

线环 (line loop) 是线带的一种简单扩展，在线带的基础上额外增加了一条连接着一批次中最后一个点和第一个点的线段。这样只是提供了一个顶点的存储，但是在我们试图将一个环或基于线段的图形进行闭合时就显得非常方便了。图 3.8 所示是我们在 “Primitives” (图元) 示例程序中所做的佛罗里达州轮廓渲染的必然结果。

图 3.7

一个线带只是从头到尾依次连接各个顶点。



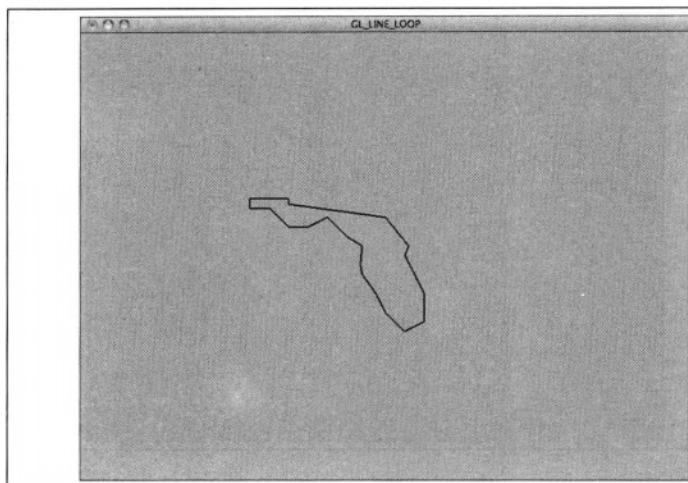


图 3.8

运用线环来将最后的缺口闭合

### 3.4.2 绘制 3D 三角形

前面我们已经了解如何绘制点和线，甚至使用 `GL_LINE_LOOP` 来绘制一些封闭的多边形了。运用这些图元，我们可以轻松地绘制任何可能存在的三维图形。例如，我们可以绘制 6 个正方形，并将它们进行排列，形成一个立方体的表面。

然而，读者可能已经发现，用这些图元创建的任何图形都没有填充任何颜色，说到底，我们只是画了一些线条而已。实际上，将 6 个正方形进行排列只是形成了一个线框立方体，而不是实心的立方体。为了绘制实体表面，我们需要的不仅仅是点和线——还需要多边形。一个多边形就是一个封闭的图形，它可能用颜色或纹理数据进行填充，也可能不进行填充，在 OpenGL 中，它是所有实体对象构建的基础。

### 3.4.3 单独的三角形

可能存在的最简单的实体多边形就是三角形，它只有 3 个边。光栅化硬件最欢迎三角形，而现在三角形已经是 OpenGL 中支持的唯一一种多边形了。每 3 个顶点定义一个新的三角形。图 3.9 所示是两个三角形，它们是用 6 个顶点进行绘制的，这 6 个顶点编号依次为  $V_0$  到  $V_5$ 。

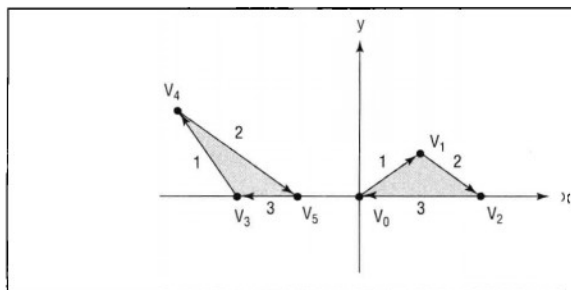


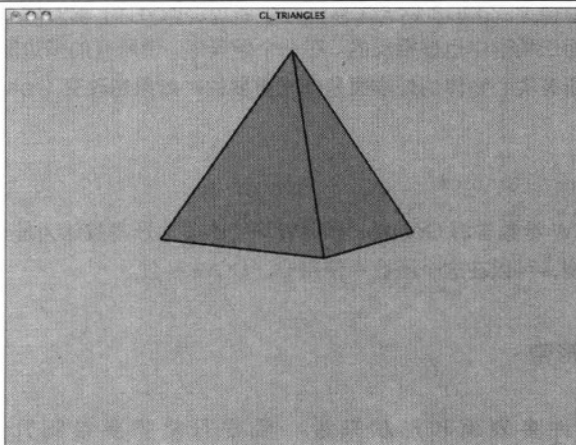
图 3.9

使用 `GL_TRIANGLES` 绘制两个三角形

图 3.10 所示是图元示例程序的下一步输出。在这里，我们不再仅仅绘制一个三角形，而是绘制 4 个组成金字塔形的三角形。我们可以使用方向键来旋转金字塔，从不同角度进行观察。这个金字塔是没有底面的，所以我們也可以看到它的内部。

图 3.10

用三角形组成一个四边金字塔



请注意，在这个示例程序中（包括此处和后续部分），我们创建了由黑色线条组成的绿色三角形轮廓。因为没有阴影或纹理与这些三角形交叉，这使得这些独立的三角形更加突出。这种黑色的轮廓并不是图元的自然固有特性，这是在绘制实心几何图形之后，再在实心图形上绘制同一个几何图形的黑色线框版本而得到的。稍后在 `glPolygonOffset` 部分我们将更加详细地展示这项工作是如何完成的。

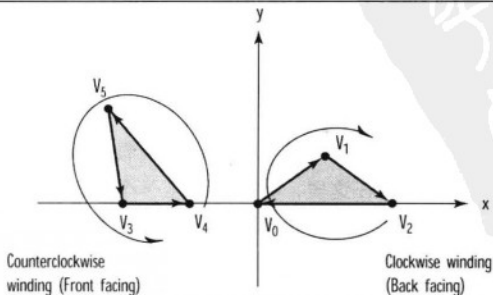
### 环绕

图 3.9 所示详细展示了任何三角形都遵循的重要特征。请注意连接顶点的线段上所标示的箭头。在绘制第一个三角形时，线条将按照从  $V_0$  到  $V_1$ ，再到  $V_2$ ，最后回到  $V_0$  的顺序来绘制一个闭合的三角形。这个路径是按照顶点被指定的顺序的，在本例从读者的观察角度看是沿着顺时针方向，这种方向特性也体现在了第二个三角形中。

这种顺序与方向结合起来指定顶点的方式称为环绕。图 3.9 所示的三角形就被称作是顺时针环绕的，因为它们确实是沿顺时针方向进行绘制的。如果我们将左边三角形的  $V_4$  和  $V_5$  的位置进行交换，我们就得到了逆时针环绕。图 3.11 所示是两个三角形，它们的缠绕方向相反。

图 3.11

两个缠绕方向相反的三角形



在默认情况下, OpenGL 认为具有逆时针方向环绕的多边形是正面的。这意味着图 3.11 的左侧是三角形的正面, 而右侧是三角形的背面。

为什么这个问题非常重要呢? 我们很快就会看到。我们常常希望为一个多边形的正面和背面分别设置不同的物理特征。我们可以完全隐藏一个多边形的背面, 或者给它设置一种不同的颜色和反射属性。纹理图像在背面三角形中也是相反的。在一个场景中, 使所有的多边形保持环绕方向的一致, 并使用正面多边形来绘制所有实心物体的外表面是非常重要的。如果想改变 OpenGL 的这个默认行为, 可以调用下面这个函数。

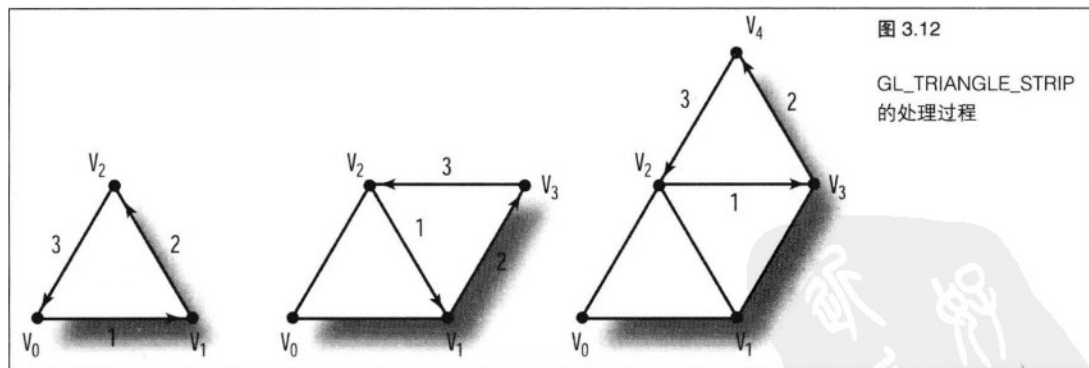
```
glFrontFace(GL_CW);
```

GL\_CW 参数告诉 OpenGL 顺时针环绕的多边形将被认为是正面的。为了把多边形的正面重新恢复为逆时针环绕, 可以在这个函数中使用 GL\_CCW 参数。

### 三角形带

对于许多表面和形状来说, 我们可能需要绘制几个相连的三角形。我们可以使用 GL\_TRIANGLE\_STRIP 图元绘制一串相连的三角形, 从而节省大量的时间。

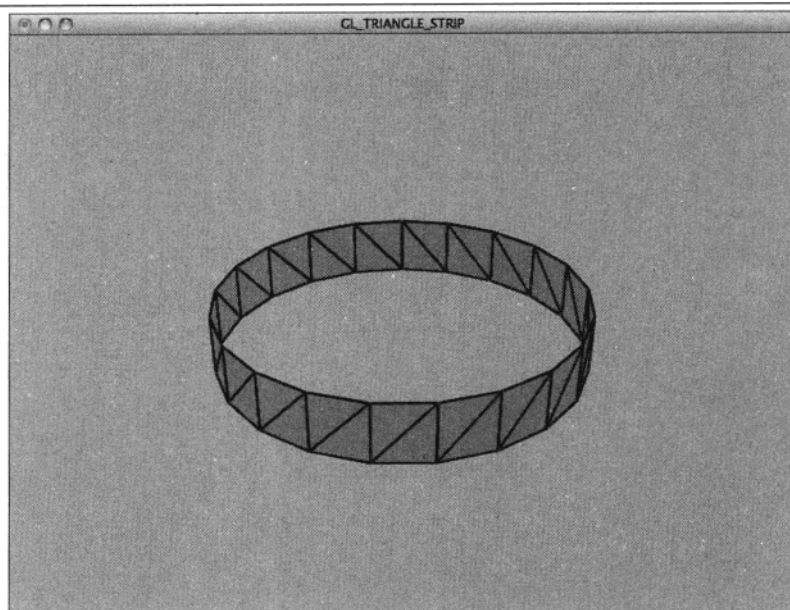
图 3.12 所示显示了由一组 5 个顶点 (标号从  $V_0$  至  $V_4$ ) 所指定的 3 个三角形组成的三角形带的绘制过程。在此, 读者可以看到这些顶点不一定是按照它们的指定顺序进行遍历的, 这是为了保持每个三角形的环绕方向 (逆时针)。它的绘制模式是  $V_0$ 、 $V_1$ 、 $V_2$ , 接着是  $V_2$ 、 $V_1$ 、 $V_3$ , 然后是  $V_2$ 、 $V_3$ 、 $V_4$ , 以此类推。



使用三角形带而不是分别指定每个三角形, 这样做有两个优点。首先, 用前 3 个顶点指定第 1 个三角形之后, 对于接下来的每个三角形, 只需要再指定 1 个顶点。需要绘制大量的三角形时, 采用这种方法可以节省大量的程序代码和数据存储空间。第二个优点是提高运算性能和节省带宽。更少的顶点意味着数据从内存传输到图形卡的速度更快, 并且顶点着色器需要进行处理的次数也更少。图 3.13 所示是图元示例程序的下一步输出。在这里, 我们绘制了一个由三角形带形成的圆环。

图 3.13

一个由三角形带形成的  
圆环



### 三角形扇

除了三角形带之外,还可以使用 `GL_TRIANGLE_FAN` 创建一组围绕一个中心点的相连三角形。图 3.14 所示是通过指定 4 个顶点所产生的包括 3 个三角形的三角形扇。第一个顶点  $V_0$  构成了扇形的原点。用前 3 个顶点指定了最初的三角形之后,后续每个顶点都和原点 ( $V_0$ ) 以及之前紧挨着它的那个顶点 ( $V_{n-1}$ ) 形成了接下来的那个三角形。

图 3.14

`GL_TRIANGLE_FAN`  
的处理过程

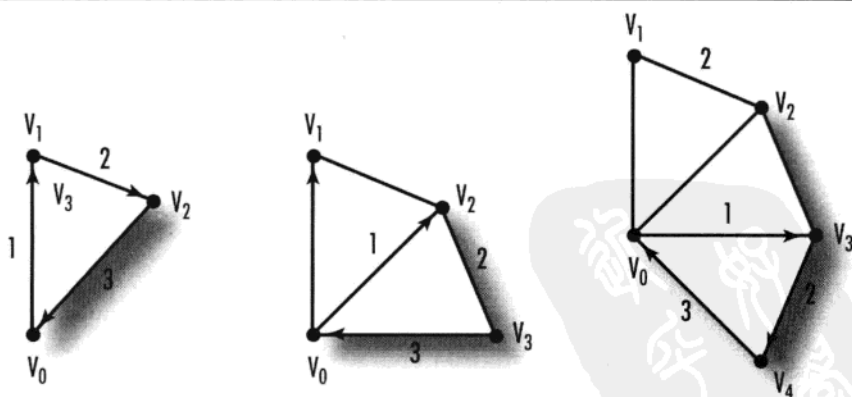


图 3.15 所示是图元示例程序的最后一步输出。我们已经围绕三角形扇的原点绘制了 6 个三角形,将这个原点稍微升高一点,以使它有一点深度。不要忘了,我们可以使用方向键将图形进行旋转。

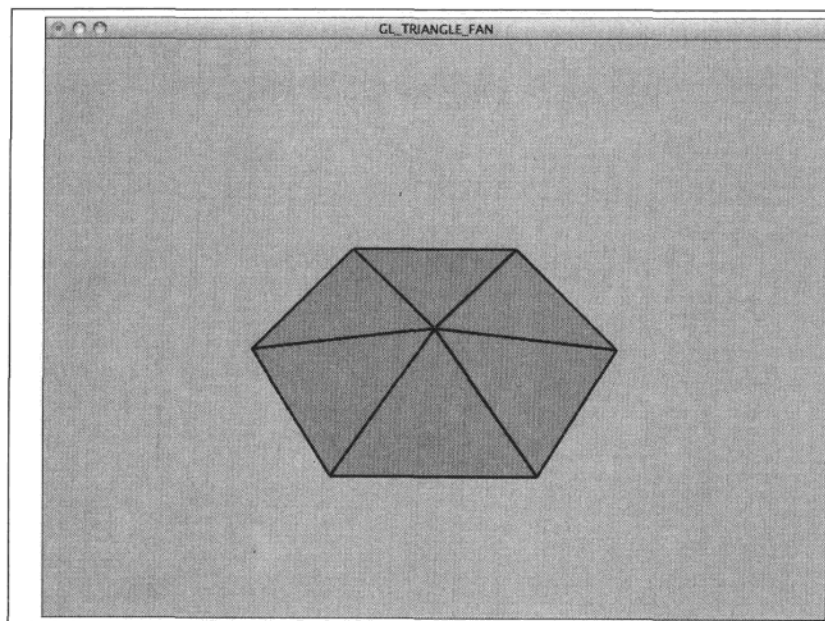


图 3.15

图元示例程序  
GL\_TRIANGLE\_FAN  
的最后一步输出。

### 3.4.4 一个简单批次容器

GLTools 库中包含一个简单的容器类，叫做 GBatch。这个类可以作为表 3.2 列出的 7 种图元简单批次的容器使用，而且它知道在使用 GLShaderManager 支持的任意存储着色器时如何对图元进行渲染。使用 GBatch 类非常简单。首先对批次进行初始化，告诉这个类它代表哪种图元，其中包括的顶点数，以及（可选）一组或两组纹理坐标。

```
void GBatch::Begin(GLenum primitive, GLuint nVerts, GLuint nTextureUnits=0);
```

然后，至少要复制一个由 3 分量（x, y, z）顶点组成的数组。

```
void GBatch::CopyVertexData3f(GLfloat *vVerts);
```

我们还可以选择复制表面法线、颜色和纹理坐标。

```
void GBatch::CopyNormalData3f(GLfloat *vNorms);
```

```
void GBatch::CopyColorData4f(GLfloat *vColors);
```

```
void GBatch::CopyTexCoordData2f(GLfloat *vTexCoords, GLuint uiTextureLayer);
```

在完成上述工作之后，我们可以调用 End 来表明已经完成了数据复制工作，并且将设置内部标记，以通知这个类包含哪些属性。

```
void GBatch::End(void);
```

实际上，可以在任何我们想要的时候来进行复制，只要不改变类的大小即可。而一旦调用 End 函数，就不能再增加新的属性了（也就是说我们现在也不能确定是否有表面法线了）。

关于提交属性的 OpenGL 实际内部运行机制实际上比这要复杂得多。然而, GLBatch 类只是一个便利类 (convenience class), 就像使用 GLUT 一样方便, 所以我们在做好充分准备之前不需要担心操作系统特性。

现在我们来快速浏览一下如何使用这个类渲染一个单个三角形。在第 2 章的三角形示例 (这是最简单的例子了) 中, 我们在源文件开头部分声明了 GLBatch 类的一个实例。

```
GLBatch triangleBatch;
```

然后在函数中, 我们用 3 个顶点设定了一个三角形。

```
// Load up a triangle
GLfloat vVerts[]={ -0.5f, 0.0f, 0.0f,
                   0.5f, 0.0f, 0.0f,
                   0.0f, 0.5f, 0.0f };

triangleBatch.Begin(GL_TRIANGLES, 3);
triangleBatch.CopyVertexData3f(vVerts);
triangleBatch.End();
```

最后, 在 RenderScene 函数中, 我们选择了适当的存储着色器并调用 Draw 函数。

```
GLfloat vRed[]={ 1.0f, 0.0f, 0.0f, 1.0f };
shaderManager.UseStockShader(GLT_SHADER_IDENTITY, vRed);
triangleBatch.Draw();
```

虽然 GLBatch 类提供了一种包含和提交几何图形的快捷方式, 但这并不代表它拥有 OpenGL 在图形处理方面的全部范围和能力。在第 12 章中, 我们将更加详细地讨论这方面内容。我们没有更早在本书中介绍这部分内容, 仅仅是因为我们希望读者能够尽可能快地开始进行实际渲染。进行实际操作是最好的学习方式之一, 而在还没有再屏幕上看到任何东西时就进行过多的解释则可能是非常令人沮丧的, 也更容易发现错误。

### 3.4.5 不希望出现的几何图形

在默认情况下, 我们所渲染的每个点、线或三角形都会在屏幕上进行光栅化, 并且会按照在组合图元批次时指定的顺序进行排列, 这在某些情况下会产生问题。其中一个可能出现的问题是, 如果我们绘制一个由很多个三角形组成的实体对象, 那么第一个绘制的三角形可能会被后面绘制的三角形覆盖。例如, 假设我们绘制一个由很多三角形组成的花托 (一种形状像面包圈的物体)。其中一些三角形在花托的背面, 而另一些则在花托的正面。我们看不到背面——至少我们应该是看不到的 (不考虑透明几何体的特殊情况)。根据我们的意图, 三角形绘制的顺序很可能会一团糟。

图 3.16 所示是 GeoTest (几何图形测试程序的缩写) 示例程序的输出, 花托稍微旋转了一点 (我们可以通过方向键来操作)。

对于这个问题, 一个可能的解决办法是, 对这些三角形进行排序, 并且首先渲染那些较远的三角形, 再在它们上方渲染那些较近的三角形。这种方式称为“油画法” (painters algorithm), 这种方法在计算机图形处理中是非常低效的, 主要原因有两个。其中一个原因是, 我们必须对任何发生几何图形重叠地方的

每个像素进行两次写操作，而在存储其中进行写操作会使速度变慢。另外一个原因是，对独立的三角形进行排序的开销会过高。我们还有更好的办法。

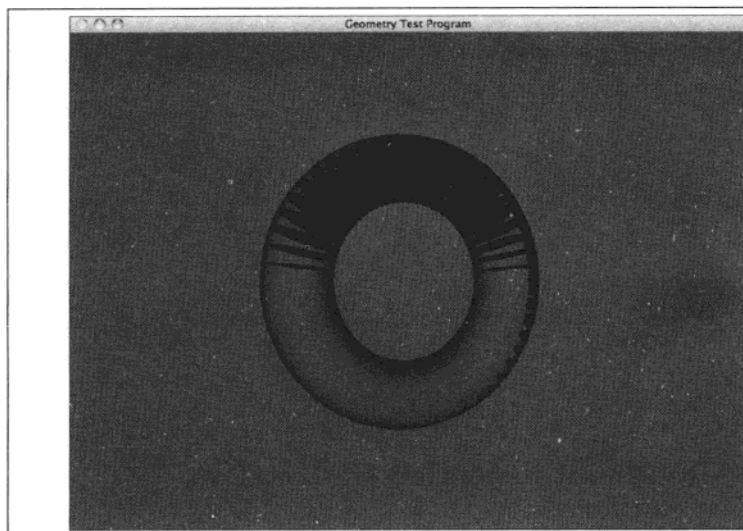


图 3.16

一个某些远端三角形绘制在近端三角形之上的对象

### 正面和背面剔除

对正面和背面三角形进行区分的原因之一就是为了进行剔除。背面剔除能够极大地提高性能，并且修正一些如图 3.16 所示之类的问题。右键单击 GeoTest 示例程序，并选择“Toggle Cull Backface”（开启背面剔除）菜单选项。图 3.17 所示为示例程序的输出。

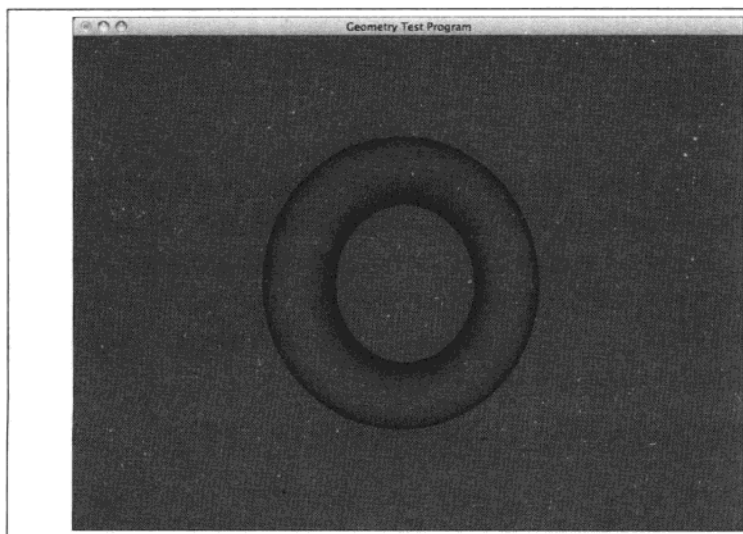


图 3.17

正确渲染的对象，消除了背面三角形

这种方式是非常高效的，在渲染的图元装配阶段就整体抛弃了一些三角形，并且没有执行任何不恰当的光栅化操作。一般表面剔除按如下方式开启。

```
glEnable(GL_CULL_FACE);
```

相应地，按照如下方式关闭。

```
glDisable(GL_CULL_FACE);
```

请注意，我们并没有指明剔除的是正面还是背面。这是由另外一个函数 `glCullFace` 控制的。

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

`mode` 参数的可用值为 `GL_FRONT`、`GL_BACK` 或 `GL_FRONT_AND_BACK`。这样，要消除不透明物体的内部几何图形就需要两行代码。

```
glCullFace(GL_BACK);
glEnable(GL_CULL_FACE);
```

在某些情况下，剔除实体几何体的正面也是非常有用的，例如在需要显示某些图形内部渲染的时候。在渲染透明对象时（马上就要说到混合了），我们经常会对一个对象进行两次渲染，其中一次开启透明并剔除正面，第二次则消除背面。这样就在渲染正面之前渲染了背面，这也是渲染透明物体的需要。

## 深度测试

深度测试是另外一种高效消除隐藏表面的技术。它的概念很简单：在绘制一个像素时，将一个值（称为  $z$  值）分配给它，这个值表示它到观察者的距离。然后，当另外一个像素需要在屏幕上的同样位置进行绘制时，新像素的  $z$  值将与已经存储的像素的  $z$  值进行比较。如果新像素的  $z$  值比较大，那么它距离观察者就比较近，这样就在原来的像素上面，所以原来的像素就会被新的像素覆盖。如果新像素的  $z$  值更低，那么它就必须位于原来像素的后面，不能遮住原来的像素。在内部，这个任务是通过深度缓冲区实现的，它存储了屏幕上每个像素的深度值。本书绝大多数示例程序使用了深度测试。

我们在使用 GLUT 设置 OpenGL 窗口的时候，应该请求一个深度缓冲区。例如，我们可以按照如下方式申请一个颜色缓冲区和一个深度缓冲区。

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
```

要启用深度测试，只需调用

```
glEnable(GL_DEPTH_TEST);
```

如果没有深度缓冲区，那么启用深度测试的命令将被忽略。

深度测试在绘制多个对象时能够进一步解决性能问题。就算背面剔除能够消除位于对象背面的三角形，那么如果是重叠的独立对象又该怎么办呢？我们之前提到过油画法，这种方法是根据一种油画使用的技术而得名的。我们只要先简单地绘制背景，再在上面绘制较近的对象。这样做可能只要在画布上进行次数不多的绘制（在手工绘制时更加有用），但对于图形硬件来说，这样做会导致在同一个片段区域重复进行绘制，而每一次绘制都会产生性能开销。如果开销过大则导致光栅化过程变慢，我们将这种方式称为“填充受限”。但是将油画法颠倒过来使用，实际上将会加速填充性能。首先绘制那些离观察者较近的对象，然后再绘制那些较远的对象。深度测试将消除那些应该被已存在像素覆盖的像素，这将节省可观的存储器带宽。

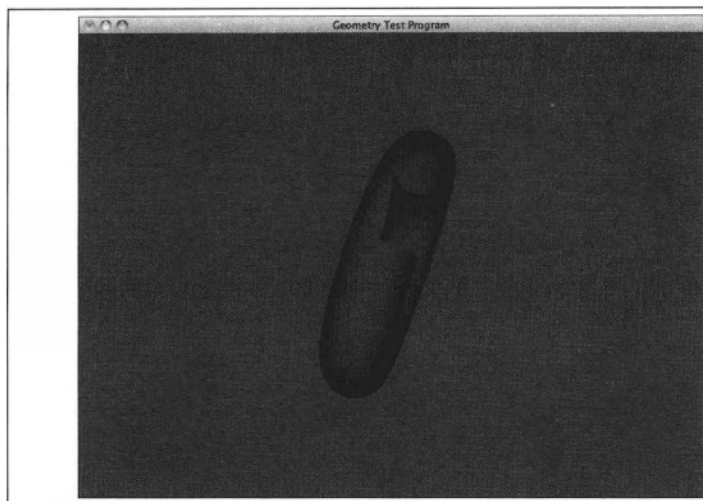


图 3.18

自身重叠的对象在不使用深度测试的情况下可能会出现这个问题

右键单击窗口，并在弹出菜单中选择“Toggle Depth Test”（开启深度测试）。这样就会调用 `glEnable` 开启深度测试，图 3.19 所示为正确渲染的对象。

相对于观察者的位置来排列对象并不困难，但是如果是自身重叠的对象又该怎么办呢？现在回到 `DepthTest`（深度测试）示例程序，我们可以调整（再重复一遍，使用方向键来完成）花托，直到它的一部分与离我们更近却恰好先进行绘制的另一部分重叠。图 3.18 所示为我们得到的效果。

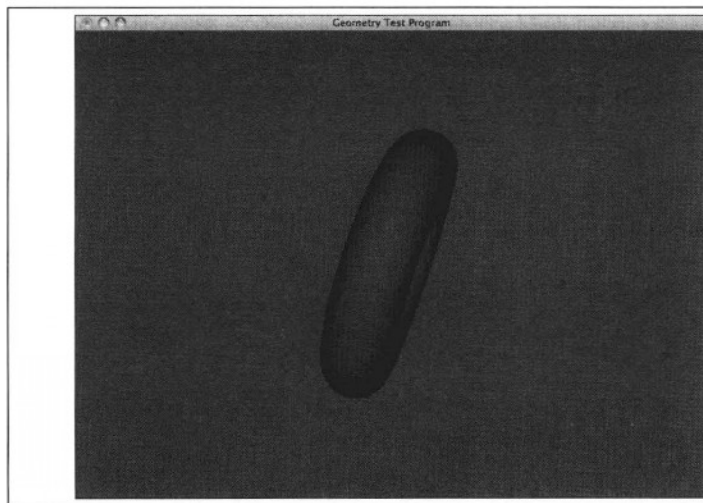


图 3.19

花托的正确深度测试

### 多边形模式

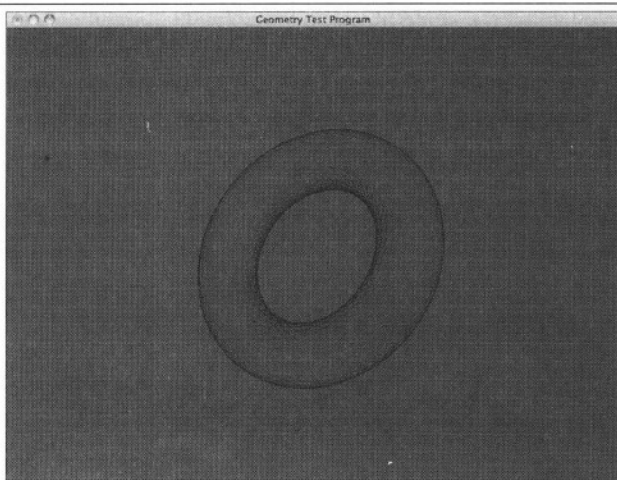
多边形（含三角形）不一定是实心的。在默认情况下，多边形是作为实心图形绘制的，但我们可以通过将多边形指定为显示轮廓或只有点（只显示顶点）来改变这种行为。函数 `glPolygonMode` 允许将多边形渲染成实体、轮廓或只有点。此外，我们可以在多边形的两面都应用这个渲染模式，也可以只在正面或背面应用。

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

和表面剔除一样，face 参数的可用值为 GL\_FRONT、GL\_BACK 或 GL\_FRONT\_AND\_BACK。而 mode 参数的可用值为 GL\_FILL (默认值)、GL\_LINE 或 GL\_POINT。图 3.20 所示为当选择 “Set Line Mode” (设置线模式) 时 GeoTest 的输出。

图 3.20

线框模式下的花托



通过调用 glPolygonMode 将多边形正面和背面设为线框模式，就能实现这种线框渲染。

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

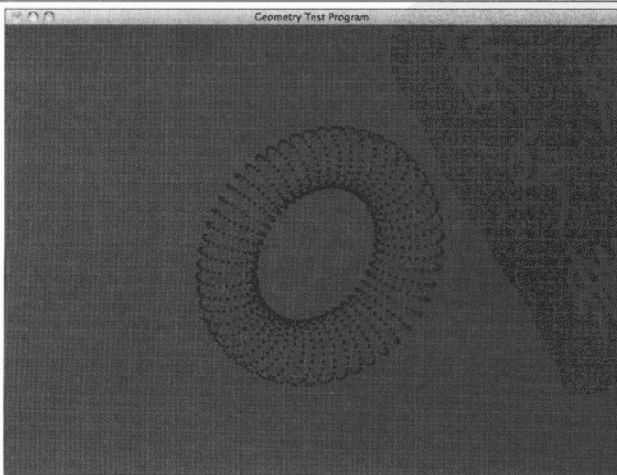
将花托绘制为点云也很容易实现。在 GeoTest 的上下文菜单中选择 “Set Point Mode” (设置点模式) 就能将多边形模式进行如下设置。

```
glPolygonMode(GL_FRONT_AND_BACK, GL_POINT);
```

图 3.21 所示为只将顶点绘制为点的花托。在本例，我们通过调用参数为 5.0 的 glPointSize 将点设置的更大了。

图 3.21

绘制为点云的花托



### 3.4.6 多边形偏移

虽然深度测试能够实现真实视觉并提高性能,但有时也会带来一些小麻烦,我们可能需要稍微“蒙骗”它一下。这种情况发生在有意要将两个几何图形绘制到同一位置时。这听起来很奇怪,但只要考虑两种情况就会明白。有时候,我们可能想要绘制一架大型飞机,然后在飞机上绘制一个较小的但却与飞机在同一物理空间的图形,这就叫做贴花(decaling)。例如我们可能会在飞机上绘制一个星形图案来进行一些设计。在这种情况下,星形图案的深度值将会与绘制原来飞机的深度缓冲区中的值相同,或者几乎相同。这样将导致片段深度测试不可预料地通过或失败,而使飞机看起来一团糟,这种情况通常称为z-fighting(z冲突)。

另外一种情况(很容易用当前的例子来演示)发生在我们想要绘制实心几何图形但又要突出它的边时。在以前介绍的示例程序 Primitives 中,三角形、三角形扇和三角形带都绘制成绿色,但都用黑线显示出互相独立的三角形。这并不是默认的行为,我们需要做一些特殊处理来达到这种效果。在默认情况下,绘制一个三角形带将会使这个圆环看起来如图 3.22 所示。

为了看到三角形的边,我们需要像前面部分展示的那样使用 `glPolygonMode` 来绘制条带。在线框模式中绘制粗黑线的结果只是生成如图 3.23 所示的线框。

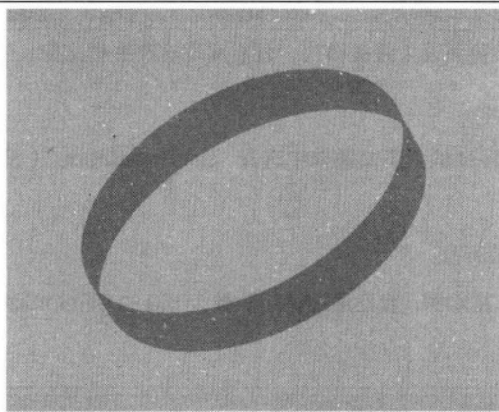


图 3.22

一个边缘没有突出显示的三角形带

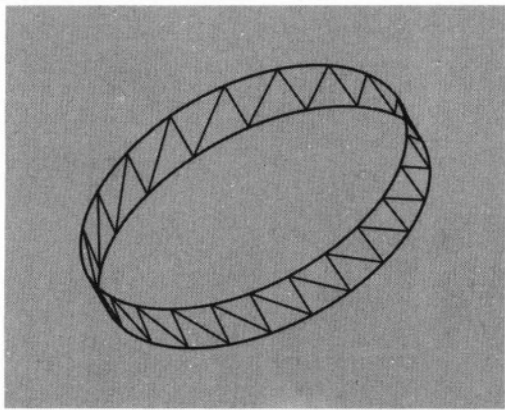


图 3.23

一个只有边缘的三角形带

问题在于，如果我们在实体条带的同一位置绘制线框，就会遇到前面提到的 z-fighting (z 冲突) 问题。我们可能会想，是不是可以通过第二次绘制时在 z 方向稍稍做一点偏移来解决这个问题。这样做确实能够达到目的，但我们必须小心确保只能沿着 z 轴向镜头方向移动，而且要移动得足够多以使深度测试产生偏移，同时又不能移动太多而导致几何图层之间产生缝隙。我们还有更好的办法。

下面介绍的 `glPolygonOffset` 函数使我们可以调节片段的深度值，这样就能使深度值产生偏移而并不实际改变 3D 空间中的物理位置。

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

应用到片段上的总偏移可以通过下面的方程式表示。

$$\text{Depth Offset} = (\text{DZ} \times \text{factor}) + (r \times \text{units})$$

其中 DZ 是深度值 (z 值) 相对于多边形屏幕区域的变化量，而 r 则是使深度缓冲区值产生变化的最小值。并没有一个硬性规定能够找到一个万无一失的值，我们具体运用的时候可能还需要试验一下。

负值将使 z 值距离我们更近，而正值则会将它们移动得更远。对于 Primitives 示例程序来说，我们将 factor 和 units 参数的值都设置为 -1.0。

除了使用 `glPolygonOffset` 设置偏移值之外，还必须启用多变量单独偏移来填充几何图形 (`GL_POLYGON_OFFSET_FILL`)、线 (`GL_POLYGON_OFFSET_LINE`) 和点 (`GL_POLYGON_OFFSET_POINT`)。程序清单 3.1 显示了 Primitives 示例程序中的一个函数，这个函数绘制了一批绿色的几何图形，并在上面绘制了它们的黑色线框版本。请注意，我们为了使显示效果更好，在轮廓上使用了更粗的抗锯齿线。接下来关于混合的内容中，我们将进一步讨论抗锯齿。

程序清单 3.1 用来绘制一个绿色带有黑色线框版本的图元批次的函数

```
void DrawWireFramedBatch(GLBatch* pBatch)
{
    // Draw the batch solid green
    shaderManager.UseStockShader(GLT_SHADER_FLAT,
        transformPipeline.GetModelViewProjectionMatrix(), vGreen);
    pBatch->Draw();

    // Draw black outline
    glPolygonOffset(-1.0f, -1.0f);    // Shift depth values
    glEnable(GL_POLYGON_OFFSET_LINE);
    // Draw lines antialiased
    glEnable(GL_LINE_SMOOTH);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

    // Draw black wireframe version of geometry
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    glLineWidth(2.5f);
    shaderManager.UseStockShader(GLT_SHADER_FLAT,
        transformPipeline.GetModelViewProjectionMatrix(), vBlack);
    pBatch->Draw();

    // Put everything back the way we found it
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    glDisable(GL_POLYGON_OFFSET_LINE);
    glLineWidth(1.0f);
    glDisable(GL_BLEND);
}
```

```
glDisable(GL_LINE_SMOOTH);
}
```

图 3.24 所示显示了这两次处理是如何叠加到一起的。

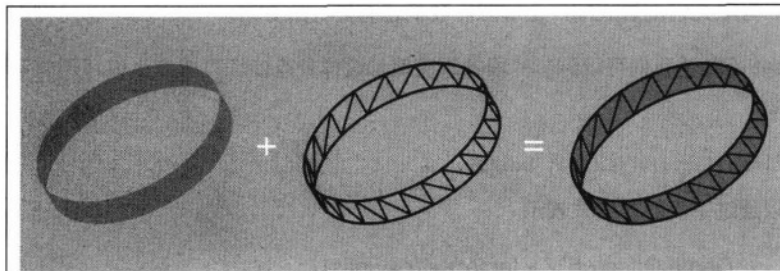


图 3.24

对实体和线框绘图进行的“装配”渲染

### 3.4.7 裁剪

另外一种提高渲染性能的方法是只刷新屏幕上发生变化的部分。我们可能还需要将 OpenGL 渲染限制在窗口中一个较小的矩形区域中。OpenGL 允许我们在将要进行渲染的窗口中指定一个裁剪框。在默认情况下，裁剪框与窗口同样大小，并且不会进行裁剪测试。我们可以使用几乎处处都要用到的 `glEnable` 函数开启裁剪测试。

```
glEnable(GL_SCISSOR_TEST);
```

当然，我们还可以使用相应的 `glDisable` 函数再次关闭裁剪测试。在执行渲染的窗口中，被称为裁剪框的矩形是使用如下函数来指定窗口坐标（像素）的。

```
void glScissor(GLint x, GLint y, GLsizei width, GLsizei height);
```

其中，`x` 和 `y` 参数指定了裁剪框的左下角，而宽度和高度则分别为裁剪框的相应尺寸。程序清单 3.2 显示了 Scissor（裁剪）示例程序的渲染代码。这个程序对颜色缓冲区进行了 3 次清除操作，每一次进行清除前都指定了一个较小的裁剪框。这样做的结果就得到了一组重叠的彩色三角形，如图 3.25 所示。

程序清单 3.2 使用裁剪框来渲染一组三角形

```
void RenderScene(void)
{
    // Clear blue window
    glClearColor(0.0f, 0.0f, 1.0f, 0.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    // Now set scissor to smaller red sub region
    glClearColor(1.0f, 0.0f, 0.0f, 0.0f);
    glScissor(100, 100, 600, 400);
    glEnable(GL_SCISSOR_TEST);
    glClear(GL_COLOR_BUFFER_BIT);

    // Finally, an even smaller green rectangle
    glClearColor(0.0f, 1.0f, 0.0f, 0.0f);
    glScissor(200, 200, 400, 200);
    glClear(GL_COLOR_BUFFER_BIT);
}
```

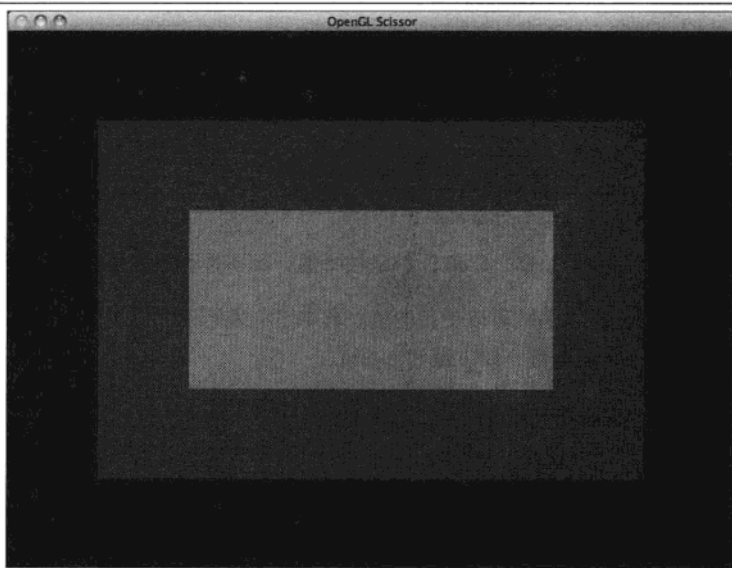


```
// Turn scissor back off for next render
glDisable(GL_SCISSOR_TEST);

glutSwapBuffers();
}
```

图 3.25

缩小裁剪框



## 3.5 混合

我们已经了解，通常情况下 OpenGL 渲染时会把颜色值放在颜色缓冲区中。我们还知道，每个片断的深度值也是放在深度缓冲区中的。当深度测试被关闭（禁用）时，新的颜色值简单地覆盖颜色缓冲区中已经存在的其他值。当深度测试被打开（启用）时，新的颜色片段只有当它们比原来的值更接近邻近的裁剪平面时才会替换原来的颜色片段。在正常情况下，任何绘制操作不是被完全丢弃，就是完全覆盖原来的颜色值，这取决于深度测试的结果。如果打开了 OpenGL 的混合功能，那么下层的颜色值就不会被清除。

```
glEnable(GL_BLEND);
```

在打开混合功能的情况下，新的颜色会与已经存在的颜色值在颜色缓冲区中进行组合。这些颜色的组合方式不同会导致很多不同的特殊效果。

### 3.5.1 组合颜色

首先，我们必须介绍一下新输入颜色值和已经存在颜色值的正式术语。已经存储在颜色缓冲区中的颜色值叫做目标颜色，这个颜色值包含了单独的红、绿、蓝成分以及一个可选的 alpha 值。作为当前渲染命令的结果进入颜色缓冲区的颜色值称为源颜色，它可能与目标颜色进行交互，也可能不与之进行交互。源

颜色也可以包含 3 种或 4 种颜色成分（红、绿、蓝和可选的 alpha 成分）。请注意，任意情况下只要我们忽略一个 alpha 值，OpenGL 都会将它设为 1.0。

当混和功能被启用时，源颜色和目标颜色的组合方式是由混合方程式控制的。在默认情况下，混合方程式如下所示。

$$C_f = (C_s * S) + (C_d * D)$$

其中， $C_f$  是最终计算产生的颜色， $C_s$  是源颜色， $C_d$  则是目标颜色， $S$  和  $D$  分别是源和目标混合因子。这些混合因子是用下面这个函数进行设置的。

```
glBlendFunc(GLenum S, GLenum D);
```

正如我们所看到的那样， $S$  和  $D$  都是枚举值，而不是可以直接指定的实际值。

表 3.3 列出了混合函数可以使用的值。其中下标表示源、目标和颜色（对于混合颜色后面再讨论）。 $R$ 、 $G$ 、 $B$  和  $A$  分别代表红、绿、蓝和 alpha。

表 3.3 OpenGL 混合因子

函 数	RGB 混合因子	Alpha 混合因子
GL_ZERO	(0,0,0)	0
GL_ONE	(1,1,1)	1
GL_SRC_COLOR	( $R_s, G_s, B_s$ )	$A_s$
GL_ONE_MINUS_SRC_COLOR	$(1, 1, 1) - (R_s, G_s, B_s)$	$1 - A_s$
GL_DST_COLOR	( $R_d, G_d, B_d$ )	$A_d$
GL_ONE_MINUS_DST_COLOR	$(1, 1, 1) - (R_d, G_d, B_d)$	$1 - A_d$
GL_SRC_ALPHA	( $A_s, A_s, A_s$ )	$A_s$
GL_ONE_MINUS_SRC_ALPHA	$(1, 1, 1) - (A_s, A_s, A_s)$	$1 - A_s$
GL_DST_ALPHA	( $A_d, A_d, A_d$ )	$A_d$
GL_ONE_MINUS_DST_ALPHA	$(1, 1, 1) - (A_d, A_d, A_d)$	$1 - A_d$
GL_CONSTANT_COLOR	( $R_c, G_c, B_c$ )	$A_c$
GL_ONE_MINUS_CONSTANT_COLOR	$(1, 1, 1) - (R_c, G_c, B_c)$	$1 - A_c$
GL_CONSTANT_ALPHA	( $A_c, A_c, A_c$ )	$A_c$
GL_ONE_MINUS_CONSTANT_ALPHA	$(1, 1, 1) - (A_c, A_c, A_c)$	$1 - A_c$
GL_SRC_ALPHA_SATURATE	( $f, f, f$ )*	1

\*其中  $f = \min(A_s, 1 - A_d)$

请记住，颜色是用浮点数表示的。因此，对它们进行加减甚至乘法都是完全合法的。表 3.3 看上去可能有点令人困惑，因此让我们通过一个常见的混合函数组合来举例说明。

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

这个函数告诉 OpenGL 接受源颜色并将这个颜色（RGB 值）与 alpha 值相乘，然后把这个结果加上

目标颜色乘以“1 减去源颜色的 alpha 值”的结果。例如，如果颜色缓冲区中已经有一种红色 (1.0f, 0.0f, 0.0f, 0.0f)，这是目标颜色 (Cd)。如果在这上面用一种 alpha 值为 0.6 的蓝色 (0.0f, 0.0f, 1.0f, 0.6f) 画了一些什么东西，就可以像下面这样计算最终颜色。

$C_d = \text{目标颜色} = (1.0f, 0.0f, 0.0f, 0.0f)$

$C_s = \text{源颜色} = (0.0f, 0.0f, 1.0f, 0.6f)$

$S = \text{源 alpha 值} = 0.6$

$D = 1 \text{ 减去源 alpha 值} = 1.0 - 0.6 = 0.4$

现在，下面这个方程式

$$C_f = (C_s * S) + (C_d * D)$$

等价于

$$C_f = (\text{Blue} * 0.6) + (\text{Red} * 0.4)$$

最终的颜色是原先的红色 (目标颜色) 与后来的蓝色 (源颜色) 进行缩放后的组合。源颜色的 alpha 值越高，添加的源颜色成分就越多，目标颜色所保留的成分就越少。

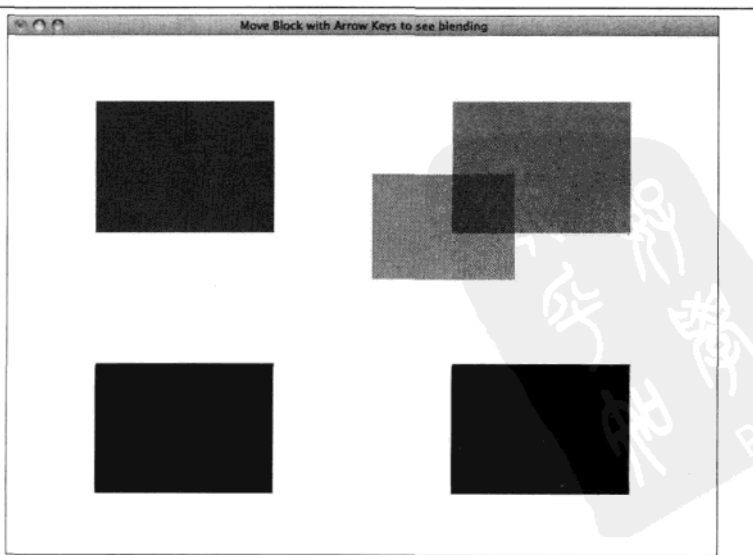
这个混合函数经常用于实现在其他一些不透明的物体前面绘制一个透明物体的效果。但是，这种技巧需要首先绘制一个或多个背景物体，然后再在上面绘制经过混合的透明物体。

例如，在 Blending 示例程序中，我们将使用透明度来实现可以在白色背景上来回移动的半透明红色矩形的幻觉。

在窗口中还有红色、蓝色、绿色和黑色的矩形，我们可以使用光标键 (方向键) 来移动半透明矩形覆盖其他颜色。这个程序的输出结果如图 2.26 所示。

图 3.26

可移动  
红色矩形与背景颜色  
进行混合



这个程序是在第2章的 Move 示例程序基础上编写的。但是，在这个例子中，背景是白色的，而且我们还在固定的位置绘制了4个其他彩色矩形。红色透明矩形在绘制时开启了混合，红色的 alpha 值设置为0.5。

```
GLfloat vRed[]={ 1.0f, 0.0f, 0.0f, 0.5f };
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
shaderManager.UseStockShader(GLT_SHADER_IDENTITY, vRed);
squareBatch.Draw();
glDisable(GL_BLEND);
```

有趣的是，我们将会发现，白色只是使红色变淡，黑色只是使红色变深，而将红色与红色混合得到的还是红色。

## 3.5.2 改变混合方程式

我们前面已经介绍的混合方程式

$$C_f = (C_s * S) + (C_d * D)$$

是默认混合方程式。实际上，我们可以从5个不同的混合方程式中进行选择，表3.4列出了这些方程式，我们可以通过下面的函数进行选择。

```
void glBlendEquation(GLenum mode);
```

表 3.4 可用的混合方程式模式

模 式	函 数
GL_FUNC_ADD	$C_f = (C_s * S) + (C_d * D)$
GL_FUNC_SUBTRACT	$C_f = (C_s * S) - (C_d * D)$
GL_FUNC_REVERSE_SUBTRACT	$C_f = (C_d * D) - (C_s * S)$
GL_MIN	$C_f = \min(C_s, C_d)$
GL_MAX	$C_f = \max(C_s, C_d)$

除了 glBlendFunc 之外，还可以利用下面的函数更加灵活地进行选择。

```
void glBlendFuncSeparate(GLenum srcRGB, GLenum dstRGB, GLenum srcAlpha, GLenum
dstAlpha);
```

其中 glBlendFunc 函数指定了源和目标 RGBA 值的混合函数，而 glBlendFuncSeparate 函数则允许为 RGB 和 alpha 成分单独指定混合函数。

最后，如表3.4所示，GL\_CONSTANT\_COLOR、GL\_ONE\_MINUS\_CONSTANT\_COLOR、GL\_CONSTANT\_ALPHA 和 GL\_ONE\_MINUS\_CONSTANT\_ALPHA 值都允许在混合方程式中引入一个常量混合颜色。这个常量混合颜色初始为黑色(0.0f, 0.0f, 0.0f, 0.0f)，但可以用下面这个函数对它进行修改。

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

```
GLclampf alpha);
```

### 3.5.3 抗锯齿

OpenGL 混合功能的另一个用途是抗锯齿。在绝大多数情况下，一个独立的渲染片段将会映射到计算机屏幕上的一个像素。

这些像素是正方形的（或者说近似正方形的），通常可以相当清楚地看到两种颜色的分界。它们常常被称为锯齿，会吸引眼睛的注意力，让人觉得图像是不自然的。这种锯齿现象会彻底地暴露出这个图像是由计算机生成的！对于许多渲染任务，我们要求达到尽可能的逼真，尤其是在游戏、模拟和艺术创作中。图 3.27 所示显示了 SMOOTHER 示例程序的输出。在图 3.28 中，我们放大了一条线段和一些点，以显示它们的锯齿状边缘。

图 3.27

SMOOTHER 程序的输出

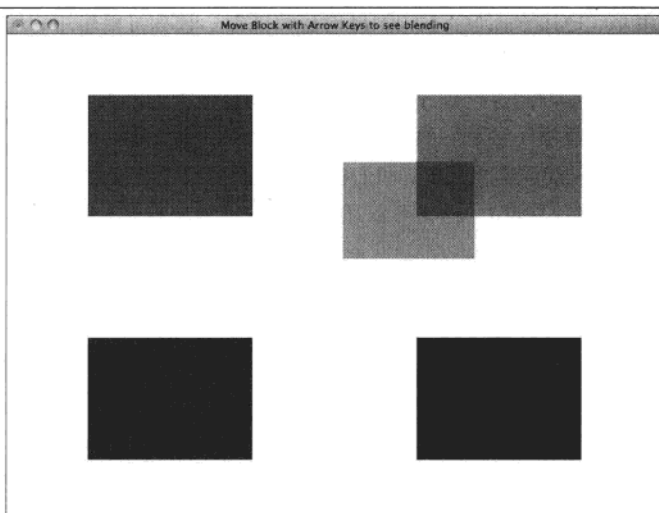
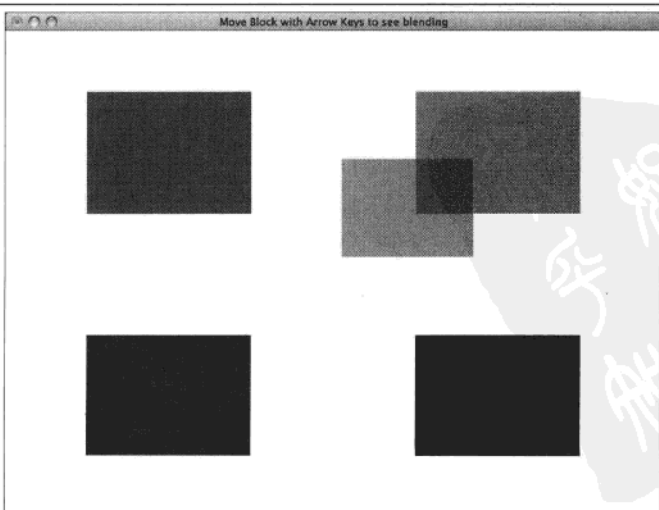


图 3.28

更近距离地观察锯齿



为了消除图元之间的锯齿状边缘, OpenGL 使用混合功能来混合片段的颜色, 也就是把像素的目标颜色与周围像素的颜色进行混合。从本质上说, 在任何图元的边缘上, 像素颜色会稍微延伸到相邻的像素。

开启抗锯齿功能非常简单。首先, 我们必须启用混合功能, 并像前一节实现透明一样设置混合函数。

```
glEnable(GL_BLEND);  
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

我们还需要确保把混合方程式设置为 GL\_ADD, 不过由于这是默认的设置, 也是最为常见的混合方程式, 因此我们就不在这里显示这个步骤了。在启用混合功能并选择正确的混合函数以及混合方程式之后, 可以选择调用 glEnable 函数对点、直线和 (或) 多边形 (任何实心图元) 进行抗锯齿处理。

```
glEnable(GL_POINT_SMOOTH);    // Smooth out points  
glEnable(GL_LINE_SMOOTH);     // Smooth out lines  
glEnable(GL_POLYGON_SMOOTH);  // Smooth out polygon edges
```

在使用 GL\_POLYGON\_SMOOTH 时应该注意。读者可能想用它使实心几何图元的边缘变得平滑。但是要实现这个目的, 还需要其他乏味的规则。例如, 重叠的几何图形需要一种不同的混合模式, 并可能需要对场景从前到后进行排序。我们对此不必钻研过深, 因为对实心物体进行抗锯齿处理并不常用, 在很大程度上已经被一种更好的对 3D 几何图形平滑边缘的称为多重采样的方法所代替。下一节将讨论这个特性。如果不使用多重采样, 我们在重叠的抗锯齿直线时仍然可能遇到这种重叠的几何图形问题。例如, 对于线框模型, 通常可以通过禁用深度测试避免直线交叉部分的深度人工痕迹。

程序清单 3.3 显示了 Smoother 程序的代码, 它对一个弹出式菜单作出响应, 允许用户在抗锯齿和非抗锯齿渲染模式间进行切换。

当这个程序在启用了抗锯齿功能之后运行时, 点和直线将渲染得更为平滑。图 3.29 所示是部分线段和点的放大, 它显示了和图 3.27 所示相同的区域, 但现在这些锯齿状边缘平滑了许多。

程序清单 3.3 在抗锯齿和正常渲染模式间切换

```
////////////////////////////////////  
//对菜单选择作出反应, 正确地重置标志  
void ProcessMenu(int value)  
{  
    switch(value)  
    {  
        case 1:  
            //打开抗锯齿, 并给出关于尽可能进行最佳的处理的提示  
            glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);  
            glEnable(GL_BLEND);  
            glEnable(GL_POINT_SMOOTH);  
            glHint(GL_POINT_SMOOTH_HINT, GL_NICEST);  
            glEnable(GL_LINE_SMOOTH);  
            glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);  
            glEnable(GL_POLYGON_SMOOTH);  
            glHint(GL_POLYGON_SMOOTH_HINT, GL_NICEST);  
            break;  
  
        case 2:  
            //关闭混合和所有的平滑处理  
            glDisable(GL_BLEND);  
            glDisable(GL_LINE_SMOOTH);  
            glDisable(GL_POINT_SMOOTH);  
            glDisable(GL_POLYGON_SMOOTH);
```

```

        break;

    default:
        break;
}

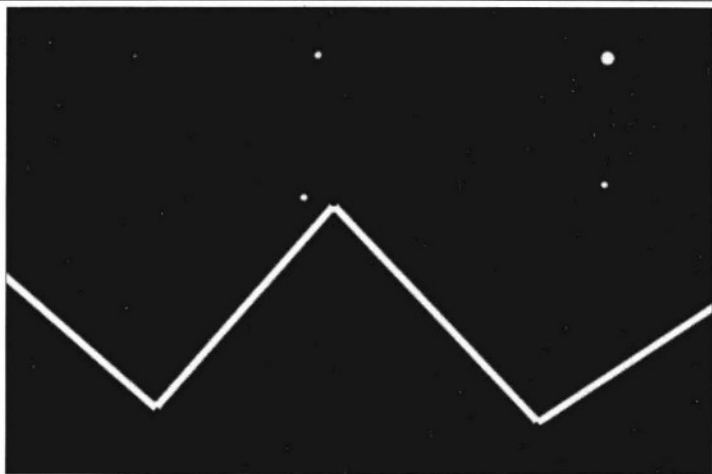
//触发重绘
glutPostRedisplay();
}

```

尤其要注意的是，这里调用了第 2 章曾经讨论过的 `glHint` 函数。有许多算法和方法可以实现抗锯齿处理的图元。任何特定的 OpenGL 实现都可以选择其中一种方法，甚至有可能支持两种方法！我们可以询问 OpenGL 是否支持多种抗锯齿算法，并根据需要选择速度最快的算法（`GL_FASTEST`）或者效果最佳的算法（`GL_NICEST`）。

图 3.29

锯齿消失了



### 3.5.4 多重采样

抗锯齿处理的最大优点之一就是能够使多边形的边缘更为平滑，使渲染效果显得更为自然和逼真。点和直线的平滑处理是得到广泛支持的，但遗憾的是多边形的平滑处理并没有在所有的平台上都得到实现。即使在可以使用 `GL_POLYGON_SMOOTH` 的时候，对整个场景进行抗锯齿处理并没有想像中的那么方便。这是因为抗锯齿处理是基于混合操作的，这就需从前到后对所有的图元进行排序，这是非常麻烦的！

OpenGL 新增了一个特性，称为多重采样（`multisampling`），可以用来解决这个问题。如果读者所使用的 OpenGL 实现支持这个特性（这是个 OpenGL 1.3 特性），已经包含了颜色、深度和模版值的帧缓冲区就会添加一个额外的缓冲区。所有的图元在每个像素上都进行了多次采样，其结果就存储在这个缓冲区中。每次当这个像素进行更新时，这些采样值进行解析，以产生一个单独的值。因此，从程序员的角度而言，它就像是自动的，属于“幕后发生的事情”。很自然，这种处理会带来额外的内存和处理器开销，有可能对性能造成影响。因此，有些 OpenGL 实现可能并不支持多渲染环境中的多重采样。

为了进行多重采样，首先必须获得一个支持多重采样帧缓冲区的渲染环境。这在不同的平台中可能各

不相同。但是，GLUT 提供了一个位段（GLUT\_MULTISAMPLE），允许请求这种帧缓冲区（在本书第 3 部分操作系统特定的章节时详述）。例如，为了请求一个多重采样、完全颜色、带深度的双缓冲帧缓冲区，可以调用：

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

可以使用 glEnable/glDisable 组合（使用 GL\_MULTISAMPLE 标记）打开或关闭多重采样：

```
glEnable(GL_MULTISAMPLE);
```

或

```
glDisable(GL_MULTISAMPLE);
```

关于多重采样还需要注意的一个地方是当它被启用时，点、直线和多边形的平滑特性都将被忽略（如果这些特性被启用的话）。这意味着在使用多重采样时，就不能同时使用点和直线的平滑处理。在一种特定的 OpenGL 实现中，点和直线如果采用平滑处理可能会比使用多重采样效果更好。因此，当绘制点和直线时，可以关闭多重采样，在绘制其他实心几何图形时再打开多重采样。下面这段伪码大致展示了这种做法。

```
glDisable(GL_MULTISAMPLE);
glEnable(GL_POINT_SMOOTH);

// Draw some smooth points
// .....
glDisable(GL_POINT_SMOOTH);
glEnable(GL_MULTISAMPLE);
```

当然，如果没有多重采样缓冲区，OpenGL 就当做 GL\_MULTISAMPLE 是被禁用的。

### 状态排序

打开或关闭不同的 OpenGL 特性将会修改驱动程序的内部状态，这种状态的改变可能会对渲染的性能造成影响。对性能非常敏感的程序员常常会不辞辛苦地对所有绘图命令进行排序，这样需要相同状态的几何图形就可以在一起绘制。这种状态排序是在游戏中常用的提高速度的方法之一。

多重采样缓冲区在默认情况下使用片断的 RGB 值，并不包括颜色的 alpha 成分。我们可以通过调用 glEnable（使用下面 3 个值之一）来修改这个行为。

- GL\_SAMPLE\_ALPHA\_TO\_COVERAGE——使用 alpha 值
- GL\_SAMPLE\_ALPHA\_TO\_ON——将 alpha 值设为 1 并使用它
- GL\_SAMPLE\_COVERAGE——使用 glSampleCoverage 所设置的值

当启用 GL\_SAMPLE\_COVERAGE 时，glSampleConverage 函数允许指定一个特定的值，它是与片断覆盖值进行按位与操作的结果。

```
void glSampleCoverage(GLclampf value, GLboolean invert);
```

这种对多重采样操作的优化并不是严格由 OpenGL 规范所规定的，其确切的结果可能因不同的 OpenGL 实现而异。

## 3.6 小结

在本章，我们介绍了大量背景知识。实际上，如果读者是 OpenGL 或者 3D 图形编程方面的新手，那么这部分内容很可能是本书中最重要的基础知识。本章以揭示今天的可编程硬件如何运用着色器进行渲染开篇，介绍了如何建立 3D 坐标空间、将顶点和属性组织到图元批次中，以及使用正确的着色器和 uniform 值对它们进行渲染。

正面和背面环绕与表面剔除是大量图形渲染算法的重要组成部分，也是性能优化工作的重要组成部分。

我们了解到，深度测试几乎是大多数 3D 场景的先决条件，甚至了解了如何使用它来加速填充性能，或者通过对我们需要与其他几何图形一致的片段增加一个微小偏移来“欺骗”它。大量特效和技术都使用了混合，在下一章我们将再次对这个主题进行讨论，并展示更加深入地使用混合来创建一个简单的反射效果。最后，我们还了解了抗锯齿和多重采样能够使计算机生成的图形质量得到巨大的提高。

我们鼓励读者亲身实践一下在本章中学到的内容。在学习本书其他内容之前，我们要利用想象力创建一些 3D 对象。这样我们就有了一些自己的示例可以操作，并且在学习和探索本书中新技术时对它们进行加强。在下一章，我们将真正地使我们的对象活跃起来！



## 第4章 基础变换：初识向量/矩阵

作者：Richard S. Wright, Jr.

### 本章内容

- ✦ 什么是向量，以及为什么要了解它
- ✦ 什么是矩阵，以及为什么要更认真地了解它
- ✦ 我们如何使用矩阵和向量来移动几何图形
- ✦ OpenGL 对于模型视图和投影矩阵的约定
- ✦ 什么是照相机，以及如何应用它的转换
- ✦ 如何将一个点光源位置转换到视点坐标系

在第3章，我们学习了如何绘制3D点、线和三角形。为了将一系列图形转换到连续的场景，我们必须将它们相对于其他图形和观察者进行排列。在本章，我们开始学习在坐标系中移动图形和对象。确定对象位置和方向的能力对于任何3D图形编程人员来说都是非常重要的。正如我们将要看到的，围绕着原点来描述对象的维度，再将对象变换到需要的位置实际上是非常方便的。

### 4.1 本章是令人生畏的数学课吗

在大多数3D图形编程书籍中，本章的内容应该是枯燥的数学知识。

但是，不要紧张，因为和一些教科书相比，我们采用了更加容易接受的方式来阐述这些原则。

对象和坐标变换的关键是 OpenGL 程序员常用的两个矩阵。为了帮助读者熟悉这两个矩阵，本章在两种极端的计算机图形学理念之间进行了平衡。其中一个极端是“在阅读本章之前应该复习一下线性代数的内容”。而另一个极端则是“学习3D图形不必通晓那些复杂的数学公式”。但是我们对这两者都不完全赞同。

事实上,即使我们不懂得那些高深的 3D 图形数学知识,也不会造成太大的妨碍,就像我们不需要懂得任何关于汽车结构和内燃机方面的知识也能每天开汽车一样。但是,我们最好对汽车足够了解,以便能够意识到需要时常更换机油,定期向油箱加油,以及在轮胎花纹磨光时要更换轮胎。这些知识使我们成为一位可靠(还有安全!)的车主。同样,如果想要成为一位可靠和有能力的 OpenGL 程序员,也要遵循同样的标准。至少需要理解那些基础知识,才知道能做什么,以及哪些工具适合我们要做的工作。如果是初学者,我们将会发现,经过一段时间的实践,就会渐渐理解矩阵和向量,并且培养出一种更为直观(和强大)的能力,能够在实践中充分利用本章所介绍的这些概念。

因此,即使我们还没有能力在脑海中默算出两个矩阵的乘法,也要明白矩阵是什么,以及这些矩阵对于 OpenGL 处理来说意味着什么。但是,在清理线性代数的老课本(每个人都有,对吧?)之前,不要紧张,GLTools 库中有一个组建叫做 Math3d,其中包含了大量好用的与 OpenGL 一致的 3D 数学例程和数据类型。虽然我们不必亲自进行所有的矩阵和向量操作,我们仍然知道它们是什么,以及如何应用它们。看,这样我们就二者兼得了。

## 4.2 3D 图形数学速成课

关于 3D 图形数学的书籍数不胜数,我们发现了一些非常好的,附录 A 中列出了这些书籍。我们并不会做出一副要讲完所有需要了解的重要问题的架势,甚至并不试图讲完所有应该了解的问题。在本章,我们只涉及真正需要了解的。如果已经是数学高手,那么就应该跳过这一部分,立即开始学习模型视图矩阵部分。这并不只是因为您已经知道了我们将要讲解的内容,还因为大多数数学高手会因为没有提供足够的空间来讨论它们喜爱的齐次坐标空间特性而感到不快。

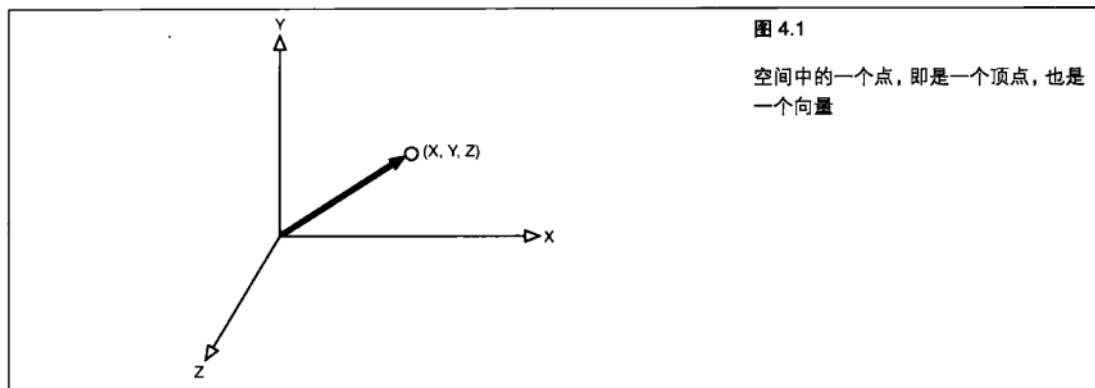
想象一下一个我们必须从一大群鳄鱼逃脱出来的电视真人秀吧。我们到底要知道多少 3D 数学才能生存?这就是接下来两部分的内容——3D 数学生存技能。鳄鱼才不在乎我们是不是真正懂得齐次坐标空间呢。

### 4.2.1 向量

在第 1 章和第 2 章,我们已经介绍了顶点和 3D 笛卡尔坐标。基本上,一个顶点就是 XYZ 坐标空间上的一个位置,而在空间中给定的一个位置恰恰是由一个且只由一个单独的 XYZ 三元组定义的。然而,一组 XYZ 值还能表示一个向量(实际上,从纯粹的数学思维上讲一个顶点其实同时也是一个向量……这里我们只讨论主要问题)。在使用向量来操纵 3D 几何图形时,向量可能就变成了最重要的基本概念了(没有之一)。这 3 个值(X、Y 和 Z)组合起来表示两个重要的值——一个方向和一个数量。

如图 4.1 所示为空间中(任意选择)的一个点,以及空间中从坐标系原点到这个点坐标的一条带箭头的线段。在拼接三角形时,这个点可以视为一个顶点,而这个带箭头的线段则可以视为一个向量。一个向量首先是空间中从原点指向这个点的方向。在中,我们总是使用向量来表示带方向的量。例如,X 轴就是向量(1, 0, 0)。在 X 方向为+1,而在 Y 方向和 Z 方向则为 0。

我们还是用一个向量来指出我们所要的方向，例如，照相机指向哪个方向，或者我们要沿着哪个方向远离鳄鱼！



向量能够代表的第二个量就是数量。一个向量的数量就是这个向量的长度。对于我们的  $x$  轴向量  $(1, 0, 0)$  来说，向量的长度就是 1。我们把长度为 1 的向量称为单位向量 (unit vector)。如果一个向量不是单位向量，而我们将它缩放到 1，就叫做标准化 (normalization)。将一个向量进行标准化就是将其长度缩放为 1。在我们只想表示一个方向而不表示数量时，单位向量就非常重要了。数量也可能是非常重要的。例如，它能够告诉我们需要在指定的方向上移动多远——我们需要离开鳄鱼多远。

math3d 库有两个数据类型，能够表示一个三维或四维向量：M3DVector3f 可以表示一个三维向量  $(X, Y, Z)$ ，而 M3DVector4f 则可以表示一个四维向量  $(X, Y, Z, W)$ 。典型情况下  $W$  坐标设为 1.0。 $X$ 、 $Y$  和  $Z$  值通过除以  $W$  来进行缩放，而除以 1.0 则本质上不改变  $XYZ$  的值。要将它们定义成数组，只需如下操作。

```
typedef float M3DVector3f[3];
typedef float M3DVector4f[4];
```

声明一个三分量向量只需如下操作。

```
M3DVector3f vVector;
```

类似地，我们可以声明并初始化一个四分量向量。

```
M3DVector4f vVertex = { 0.0f, 0.0f, 1.0f, 1.0f };
```

现在我们声明一个三分量顶点数组，例如为了生成一个三角形：

```
M3DVector3f vVerts[] = { -0.5f, 0.0f, 0.0f,
                          0.5f, 0.0f, 0.0f,
                          0.0f, 0.5f, 0.0f };
```

在这里，我们要注意不要过分忽视第 4 个分量  $W$ 。在绝大多数情况下我们通过顶点位置指定几何图形时，我们只想存储和发送给 OpenGL 三分量的顶点。对于许多方向性顶点例如一个表面法线（用于进行光照计算）来说，一个三分量的向量也是足够的。但是，我们很快就要开始研究矩阵，并且进行 3D 顶点变换，这时我们就必须用它乘以一个  $4 \times 4$  变换矩阵了。这里的规则是，我们必须用一个四分量向量乘以一个  $4 \times 4$  矩阵，如果我们试图使用一个三分量向量和一个矩阵相乘……鳄鱼会吃了我们！稍后将更加

详细地解释。基本上，如果我们要在向量上进行自己的矩阵操作，那么在很多情况下很可能会希望使用四分量向量。

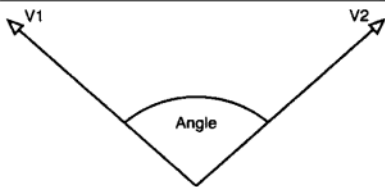
## 点乘

向量可以进行加法、减法运算，也可以简单地通过加法、减法进行缩放，或者对 XYZ 分量单独进行缩放。然而还有一种有趣又有用的操作称为点乘（dot product），这种操作只能在两个向量之间进行。

两个（三分量）单位向量之间的点乘运算将得到一个标量（只有一个值），它表示两个向量之间的夹角。要进行这种运算，这两个向量必须为单位长度，而返回的结果将在  $-1.0$  到  $+1.0$  范围之内。这个数字实际上就是这两个向量之间夹角的余弦值。在漫射光计算中，表面法向量和指向光源的向量之间大量进行着这种运算。在第 6 章，我们甚至会在着色器代码中进行这种运算。图 4.2 所示为两个向量  $V1$  和  $V2$ ，以及它们之间角度的表示方法。

图 4.2

点乘运算返回两个向量之间的夹角



math3d 库中也包含一些有用的函数使用点乘操作。

首先，我们可以使用 `m3dDotProduct3` 函数来实际获得两个向量之间的点乘结果。

```
float m3dDotProduct3(const M3DVector3f u, const M3DVector3f v);
```

实际点乘结果是一个在  $-1$  和  $+1$  之间的值，它代表这两个单位向量之间夹角的余弦值。

`m3dGetAngleBetweenVectors3` 是一个更高级一点的函数，实际上返回这个角的弧度值。

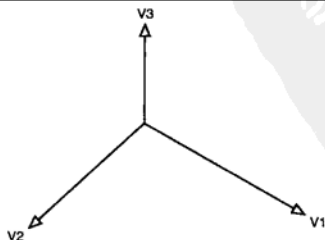
```
float m3dGetAngleBetweenVectors3(const M3DVector3f u, const M3DVector3f v);
```

## 叉乘

另一种在两个向量之间进行的有用的数学操作是叉乘（cross product）。两个向量之间叉乘所得的结果是另外一个向量，这个新向量与原来两个向量定义的平面垂直。要进行叉乘，这两个向量都不必为单位向量。图 4.3 所示为两个向量  $V1$  和  $V2$ ，以及它们的叉乘结果  $V3$ 。

图 4.3

叉乘运算的结果返回一个新的向量，这个新向量与原来两个向量垂直。



在 math3d 库中也有一个函数 `m3dCrossProduct3` 对两个向量进行叉乘并返回运算得到的结果向量。

```
void m3dCrossProduct3(M3DVector3f result, const M3DVector3f u,
                     const M3DVector3f v);
```

和点乘不同，在进行叉乘时向量的顺序是非常重要的。如图 4.3 所示， $V_3$  是  $V_2$  和  $V_1$  进行叉乘得到的结果。如果调换  $V_1$  和  $V_2$  的顺序，那么结果得到  $V_3$  的将会指向与原来相反的方向。从寻找三角形表面法线到构造变换矩阵，关于叉乘的应用数不胜数。

## 4.2.2 矩阵

矩阵 (matrix, 也是电影《黑客帝国》的英文原名) 不仅是好莱坞电影三部曲的名字, 它还是一种功能非常强大的数学工具, 它大大简化了求解变量之间有复杂关系的方程或方程组的过程。其中一个具有普遍性的例子和图形程序设计人员密切相关, 就是坐标变换。例如, 如果在空间中有一个点, 由  $x$ ,  $y$  和  $z$  坐标定义, 将它围绕任意点沿任意方向旋转一定角度后, 我们需要知道这个点现在的位置, 就要用到矩阵。这是为什么呢? 因为新的  $x$  坐标不仅与原来的  $x$  坐标和其他旋转参数有关, 还与  $y$  和  $z$  坐标值有关。这种变量与解之间的相关性就是矩阵最擅长解决的问题对于有数学背景的电影《黑客帝国》的影迷来说, 矩阵 (matrix) 确实是一个合适的标题。

在数学上, 矩阵只不过是一组排列在统一的行和列中的数字而已——用程序设计的语言来说就是一个二维数组。一个矩阵不必是正方形的, 但是矩阵中每一行 (或每一列) 的元素数都必须和其他行 (或列) 的元素数相同。图 4.4 所示为一些矩阵的示例。它们并没提供什么特别的信息, 只是演示一下矩阵结构。请注意, 一个矩阵只有一行或者一列也是合法的。只有一行或者一列数字可以更简单地称为向量, 就像之前讨论过的一样。实际上, 就像马上就要看到的, 我们可以将矩阵看作一组列向量。

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix} \begin{bmatrix} 0 & 42 \\ 1.5 & 0.877 \\ 2 & 14 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

图 4.4

三个矩阵的示例

矩阵和向量是两个重要的术语, 我们在 3D 图形编程文献中经常遇到。在与这些量打交道时, 我们还会遇到“标量”这个术语。一个标量就是一个普通的单独数据, 用来表示大小和特定量。

矩阵之间可以进行乘法和加法, 也能与向量或者标量相乘。用一个点 (向量) 乘以一个矩阵 (一次变换) 结果得到一个新的变换点 (向量)。矩阵变换实际上并不难理解, 但是刚开始的时候可能会显得令人畏惧。由于理解矩阵变换是许多 3D 任务的基础, 我们还是应该努力熟悉它们。幸运的是, 只有一点点了解就足以使我们能够使用 OpenGL 做一些奇妙的事了。久而久之, 随着经验和学习 (参见附录 A) 的积累, 我们就可以掌握这种数学工具了。

与此同时, 就像前面讲的向量一样, 我们将会发现 math3d 库中有许多有用的矩阵函数和特性。GLTools 源代码文件夹中的 `math3d.h` 和 `math3d.cpp` 中还提供了这个库的源代码。这个 3D 数学库大大简化了本章和下一章中的很多工作。这个库还有一个“有用的”特性, 就是它缺乏特别聪明和高度优化的代码! 这就使这个库具有了高度可移植性, 并且非常容易理解。我们还会发现, 这是一种和 OpenGL 非常类似的 API。

在我们将要进行 3D 程序设计工作时，我们将使用的几乎全部是两种维度的矩阵，即  $3 \times 3$  和  $4 \times 4$  矩阵。在 math3d 库中也有这两种维度的矩阵数据类型。

```
typedef float M3DMatrix33f[9];
typedef float M3DMatrix44f[16];
```

许多矩阵库都定义了一个二维矩阵作为 C 语言中的二维数组。

OpenGL 约定中拒绝了这个传统并使用了一个一维数组。这样做的原因是，OpenGL 使用一种叫做 Column-Major（以列为主的）矩阵排序的矩阵约定。我们马上将会进一步讨论这个问题，不过，讨论在数学上通过矩阵能够做的所有事情有点太抽象了，不适合我们的口味。让我们先来解释一下将要试图完成的工作，然后再来展示矩阵是如何实现它的。

## 4.3 理解变换

我们想一想就会知道，大多数 3D 图形其实并不是真正 3D 的。我们使用 3D 概念和术语来描述物体，然后这些 3D 数据被“压扁”到一个 2D 的计算机屏幕上。这种将 3D 数据被“压扁”成 2D 数据的处理过程叫做投影（projection），我们在第一章中已经介绍了正投影和透视投影。在我们想要描述投影中出现的变换（正交变换或透视变换）的类型时，我们都会涉及投影，但投影只是 OpenGL 中发生的变换中的一种而已。变换还允许我们旋转对象、移动对象，甚至对它们进行伸展、收缩和扭曲。

在我们指定顶点和这些顶点出现在屏幕上之间的这段时间里，可能会发生 3 种类型的几何变换：视图变换、模型变换和投影变换。在这一部分，我们研究了每一种变换类型的原则，在表 4.1 中进行了概述。

表 4.1 OpenGL 变换术语概览

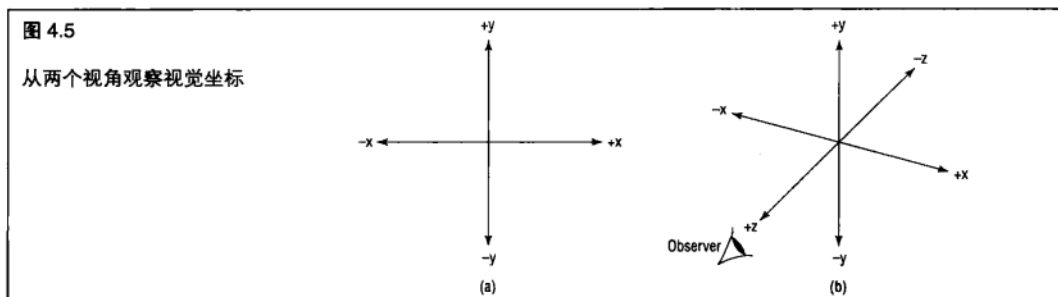
变 换	应 用
视图	指定观察者或照相机的位置
模型	在场景中移动物体
模型视图	描述试图和模型变换的二元性
投影	改变视图体的大小或重新设置它的形状
视口	这是一种伪变换，只是对窗口上的最终输出进行缩放

### 4.3.1 视觉坐标

视觉坐标是一个贯穿本章的重要概念。视觉坐标是相对于观察者的视角而言的，无论可能进行何种变换，我们都可以将它们视为“绝对的”屏幕坐标。这样，视觉坐标就表示一个虚拟的固定坐标系，它通常作为参考坐标系使用。本章讨论的所有变换都是根据它们相对于视觉坐标系的效果来描述的。

图 4.5 所示从两个不同视点显示了视觉坐标系。在左边的图（a 图）中，视觉坐标系是以场景的观察

者的角度（也就是垂直于显示器的方向）。在右边的图（b图）中，视觉坐标系稍稍进行了旋转，这样就可以更好地观察 $z$ 轴的位置关系了。从观察者的角度来看， $x$ 轴和 $y$ 轴的正方向分别指向右方和上方。 $z$ 轴的正方向从原点指向使用者，而 $z$ 轴的负方向则从观察者指向屏幕内部。



当我们利用 OpenGL 进行 3D 绘制时，就会使用笛卡尔坐标系。如果不进行任何变换，那么使用的坐标系将与刚刚描述的视觉坐标系相同。

### 4.3.2 视图变换

视图变换是应用到场景中的第一种变换。它用来确定场景中的有利位置。在默认情况下，透视投影中的观察点位于原点  $(0,0,0)$ ，并沿着  $z$  轴的负方向进行观察（向显示器内部“看进去”）。观察点相对于视觉坐标系进行移动，来提供特定的有利位置。当观察点位于原点  $(0,0,0)$  时，就像在透视投影中一样，绘制在  $z$  坐标为正的位置的物体则位于观察者背后。

然而在正投影中，观察者被认为是在  $z$  轴正方向无穷远的位置，能够看到视景物中的任何东西。

视图变换允许我们把观察点放在所希望的任何位置，并允许在任何方向上观察场景。确定视图变换就像在场景中放置照相机并让它指向某个方向。

从大局上考虑，在应用任何其他模型变换之前，必须先应用视图变换。这样做是因为，对于视觉坐标系而言，视图变换移动了当前的工作坐标系。所有后续变换随后都会基于新调整的坐标系进行。然后，在实际开始考虑如何进行这些变换时，就会更容易地看到这些变换是如何实现的了。

### 4.3.3 模型变换

模型变换用于操纵模型和其中的特定对象。这些变换将对象移动到需要的位置，然后再对它们进行旋转和缩放。

图 4.6 所示说明了我们将在对象上应用得最普遍的模型变换中的 3 种。图 4.6 (a) 所示展示了平移变换，图中的对象沿着给定的轴进行移动；图 4.6 (b) 所示展示了旋转变换，图中的对象围绕着一坐标轴进行旋转；图 4.6 (c) 所示展示了缩放效果，途中对象的大小进行了指定数量的放大或缩小。缩放可以是不均匀的（不同的维度可以进行不同程度的缩放），所以我们可以使用缩放来对对象进行伸展或收缩。

场景或对象的最终外观可能在很大程度上取决于应用的模型变换顺序。在平移和旋转上尤其如此。

图 4.7 (a) 所示展示了一个正方形逐渐变换的过程，它先围绕  $z$  轴进行旋转，然后再沿着变换后得到的新  $x$  轴进行平移。在图 4.7 (b) 中，同样的正方形首先沿着  $x$  轴进行平移，然后再围绕  $z$  轴进行旋转。

两者最终外观上有所不同是因为每次变换都与最后的变换结果相关。在图 4.7 (a) 中，正方形是围绕原始坐标轴旋转的。而在图 4.7 (b) 中，在正方形已经进行平移之后，旋转是围绕新变换的坐标轴进行的。

图 4.6

模型变换

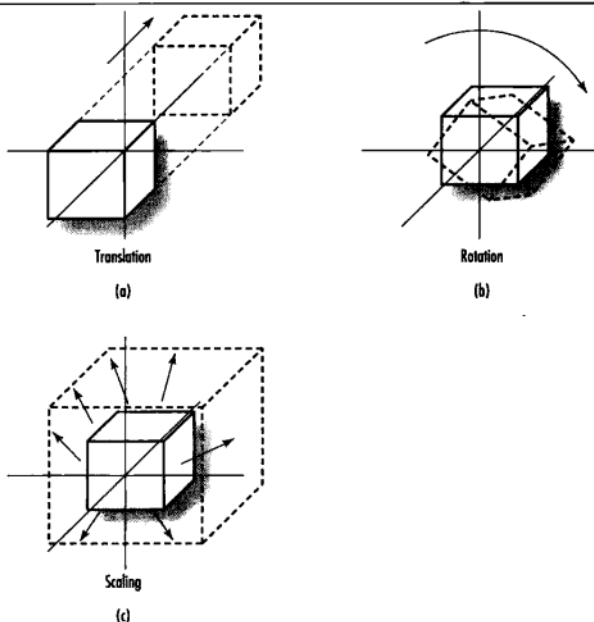
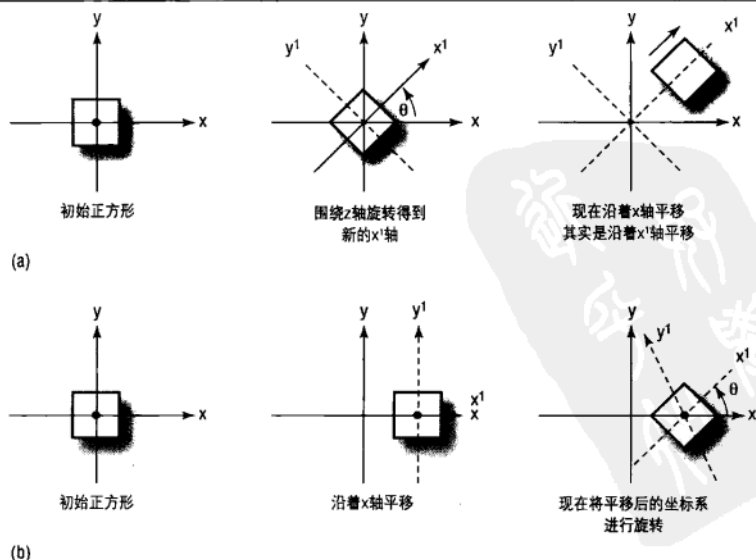


图 4.7

模型变换：先旋转后平移  
与先平移后旋转

### 4.3.4 模型视图的二元性

实际上，视图和模型变换按照它们内部效果和对场景的最终外观来说是一样的。将这两者区分纯粹是为了程序员的方便。将对象向后移动和将参考坐标系向前移动在视觉上没有区别，如图 4.8 所示，其效果是相同的。（我们可能亲身体验过这种效果，当我们坐在车里，看到后面的车超过我们时，会感到自己的车正在后退。）视图变换和模型变换一样，都应用在整个场景中，在场景中的对象常常在进行视图变换后单独进行模型变换。术语“模型视图”是指这两种变换在变换管线中进行组合，成为一个单独的矩阵，即模型视图矩阵。

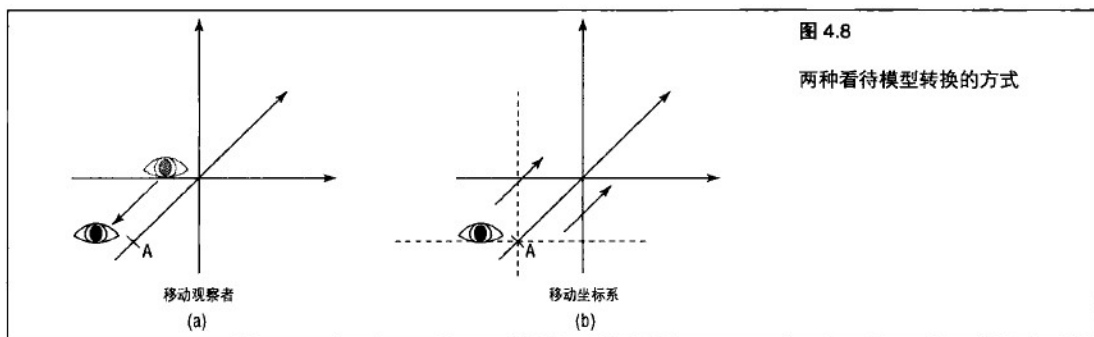


图 4.8

两种看待模型转换的方式

因此，视图变换并没有特别之处。从本质上说，它只是在绘制对象之前应用到一个虚拟对象（观察者）之上的一种模型变换。正如我们将要看到的，在将更多对象加入场景中的同时，还会不断指定新的变换。按照惯例，初始变换是所有其他变换参考的基础。

### 4.3.5 投影变换

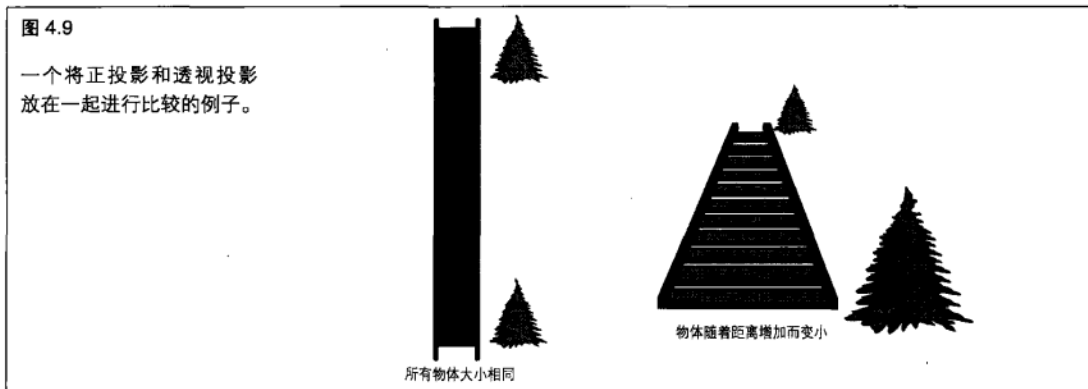
投影变换将在模型视图变换之后应用到顶点上。这种投影实际上定义了视景体并创建了裁剪平面。

裁剪平面是 3D 空间中的平面方程式，OpenGL 用它来确定几何图形对于观察者来说是否可见。更具体地说，投影变换指定一个完成的场景（所有模型变换都已完成）是如何投影到屏幕上的最终图像。在本章后面部分，我们将学习更多关于两种投影——正投影和透视投影的内容。

在正投影（或者说平行投影）中，所有多边形都是精确地按照指定的相对大小来在屏幕上绘制的。线和多边形使用平行线来直接映射到 2D 屏幕上，这就意味着，无论某个物体的位置有多远，它都会按照同样的大小来进行绘制，仅仅是平贴到屏幕上而已。典型情况下，这种投影用于渲染二维图像，例如蓝图或者是文本或屏幕菜单等二维图形。

透视投影所显示的场景与现实生活中更加接近，而不是一张蓝图。透视投影的特点就是透视缩短（foreshortening），这种特性使得远处的物体看起来比近处同样大小的物体更小一些。3D 空间中应该是平行的线可能在观察者看来不总是平行的。例如，对于铁轨来说，两根铁轨是平行的，但是在使用透视投影的情况下，它们看起来将在远处的某一点汇聚在一起。

透视投影的优势在于，我们不必弄清楚线在哪里相交或远处的物体到底有多销。我们需要做的仅仅是指定适用模型视图变换的场景，然后应用透视投影矩阵。线性代数将帮助我们完成一切。图 4.9 所示在两个不同场景中比较了正投影和透视投影。



正投影大多在进行 2D 绘制使用，这种情况下我们希望点和绘制单元精确一致。我们可能会在原理图设计、文本或者 2D 图形应用中使用正投影。在渲染对象的深度与它们到视点的距离相比非常小时，我们也可以使用正投影来进行 3D 渲染。透视投影应用在渲染包含广阔的空间或者需要应用透视缩短的物体的场景时使用。在大多数情况下，透视投影都应用在 3D 图形中。实际上，使用正投影的 3D 对象看起来可能会有些让人不舒服。

### 4.3.6 视口变换

当所有这些都讲述并完成后，就得到了一个场景的二维投影，它将被映射到屏幕上某处的窗口上。这种到物理窗口标的映射是我们最后要做的变换，称为视口变换。通常，颜色缓冲区和窗口像素之间存在一一对应关系，但情况也并非一定如此。在某些情况下，视口变换会将“规范化”设备坐标重新映射到窗口坐标上。幸运的是，我们不必为这些事操心，图形硬件会为我们做好这些。

## 4.4 模型视图矩阵

模型视图矩阵式一个  $4 \times 4$  矩阵，它表示一个变换后的坐标系，我们可以用来放置对象和确定对象的方向。我们为图元提供的顶点将作为一个单列矩阵（也就是一个向量）的形式来使用，并乘以一个模型视图矩阵来获得一个相对于视觉坐标系的经过变换的新坐标。

如图 4.10 所示，一个包含单个顶点数据的矩阵乘以模型视图矩阵后得到新的视觉坐标。顶点数据实际上是 4 个元素，其中包含一个附加值  $W$ ，它表示一个缩放因子。默认情况下这个值被设置为 1.0，而我们很少去改动它。

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \begin{bmatrix} 4 \times 4 \\ M \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ w_0 \end{bmatrix}$$

图 4.10

一个在单个顶点上应用模型视图变换的矩阵方程

将一个顶点乘以一个矩阵来对它进行变换。这到底是如何做到的呢？

### 4.4.1 矩阵构造

正如我们以前提到过的，OpenGL 并不是将一个  $4 \times 4$  矩阵表示为一个浮点值的二维数组，而是将它表示为一个由 16 个浮点值组成的单个数组。这种方式与许多数学库都不同，这些数学库都使用二维数组的方式。

例如，OpenGL 会采用的是下面两个例子中的第一个。

```
GLfloat matrix[16];           // Nice OpenGL friendly matrix
GLfloat matrix[4][4];        // Popular, but not as efficient for OpenGL
```

OpenGL 也可以使用第二种选择，但第一种是更加有效的表示方式，其中的原因很快就会清楚。这 16 个元素表示  $4 \times 4$  矩阵，如图 4.11 所示。当这些数组元素一个接一个地遍历矩阵列中的列时，我们称这种方式为列优先排序。在存储器中，这种  $4 \times 4$  的二维数组形式（即上述代码的第二种选项）将以行优先顺序呈现。在数学中，这两种形式的矩阵互为转置矩阵。

$$\begin{bmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{bmatrix}$$

图 4.11

列优先矩阵排序

真正的奥妙在于，这 16 个值表示了空间中的一个特定位置，以及相对于视觉坐标系（回忆一下我们以前讲过的那个固定的、没改变过的坐标系）的 3 个轴上的方向。解释这些数字一点也不困难。这 4 列中每一列都代表一个由 4 个元素组成的向量。为了保持本书简洁的风格，我们将注意力集中在前 3 列中的向量的前 3 个元素上。第 4 列向量包含变换后的坐标系原点的  $x$ 、 $y$  和  $z$  值。

前 3 列的前 3 个元素只是方向向量，它们表示空间中  $x$  轴、 $y$  轴和  $z$  轴上的方向（在这里用向量来表示一个方向）。对于大多数应用来说，这 3 个向量相互之间总是成  $90^\circ$  角，并且通常为单位长度（除非我们还应用了缩放或裁剪）。这种情况的数学术语叫做标准正交（向量为单位长度）或者正交（向量不是单位长度）。图 4.12 所示为  $4 \times 4$  转换矩阵，其中的列向量进行了特别标注。请注意，矩阵的最后一行都为 0，只有最有一个元素为 1。

$$\begin{array}{c} \text{方向} \\ \text{向量} \\ \text{X轴} \\ \downarrow \\ x_x \\ x_y \\ x_z \\ 0 \end{array} \quad \begin{array}{c} \text{方向} \\ \text{向量} \\ \text{Y轴} \\ \downarrow \\ y_x \\ y_y \\ y_z \\ 0 \end{array} \quad \begin{array}{c} \text{方向} \\ \text{向量} \\ \text{Z轴} \\ \downarrow \\ z_x \\ z_y \\ z_z \\ 0 \end{array} \quad \begin{array}{c} \text{变换后} \\ \text{位置} \\ \downarrow \\ t_x \\ t_y \\ t_z \\ 1 \end{array}$$

图 4.12

一个  $4 \times 4$  矩阵是如何在 3D 空间中表示一个位置和方向的

最奇妙的是，如果有一个包含一个不同坐标系的位置和方向的  $4 \times 4$  矩阵，然后用一个表示原来坐标系的向量（表示为一个列矩阵或向量）乘以这个矩阵，得到的结果是一个转换到新坐标系下的新向量。这就意味着，空间中任何位置和任何想要的方向都可以由一个  $4 \times 4$  矩阵唯一确定，并且如果用一个对象的所有向量乘以这个矩阵，那么我们就将整个对象变换到了空间中的给定位置和方向！

## 单位矩阵

有一些重要类型的变换矩阵，在我们开始尝试使用它们之前，首先要熟悉它们。第一个就是单位矩阵。将一个向量乘以一个单位矩阵，就相当于用这个向量乘以 1，不会发生任何改变。

如图 4.13 所示，单位矩阵中除了对角线上的一组元素之外，其他元素均为 0。

图 4.13

将一个向量乘以一个单位矩阵得到的结果还是原来的矩阵

$$\begin{bmatrix} 8.0 \\ 4.5 \\ -2.0 \\ 1.0 \end{bmatrix} \begin{bmatrix} 1.0 & 0 & 0 & 0 \\ 0 & 1.0 & 0 & 0 \\ 0 & 0 & 1.0 & 0 \\ 0 & 0 & 0 & 1.0 \end{bmatrix} = \begin{bmatrix} 8.0 \\ 4.5 \\ -2.0 \\ 1.0 \end{bmatrix}$$

使用单位矩阵绘制的对象不会发生变换，他们还在原点（最后一列），并且  $x$  轴、 $y$  轴和  $z$  轴与视觉坐标中一样。

我们可以在 OpenGL 中这样生成一个单位矩阵：

```
GLfloat m[] = { 1.0f, 0.0f, 0.0f, 0.0f, // X Column
                0.0f, 1.0f, 0.0f, 0.0f, // Y Column
                0.0f, 0.0f, 1.0f, 0.0f, // Z Column
                0.0f, 0.0f, 0.0f, 1.0f }; // Translation
```

或者使用 math3d 的 M3DMatrix44f 类型：

```
M3DMatrix44f m = { 1.0f, 0.0f, 0.0f, 0.0f, // X Column
                   0.0f, 1.0f, 0.0f, 0.0f, // Y Column
                   0.0f, 0.0f, 1.0f, 0.0f, // Z Column
                   0.0f, 0.0f, 0.0f, 1.0f }; // Translation
```

在 math3d 库中还有一个快捷函数 m3dLoadIdentity44，这个函数初始化一个空单位矩阵。

```
void m3dLoadIdentity44(M3DMatrix44f m);
```

我们可以回忆一下，在本书中使用的第一个存储（顶点）着色器就叫做单位着色器。这个着色器完全不对顶点做任何改变，而是将这些顶点绘制在默认坐标系中，并且不在这些顶点上应用任何矩阵。我们可以将这些顶点乘以单位矩阵，但是这是一种浪费和毫无意义的操作。

## 平移

一个平移矩阵仅仅是将我们的顶点沿着 3 个坐标轴中的一个或多个进行平移。

图 4.14 所示是一个将正方形沿着  $y$  轴正方向平移 10 个单位长度的例子。

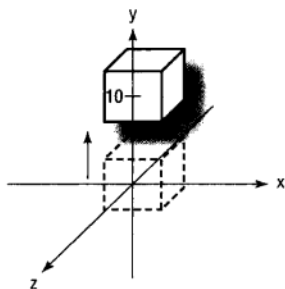


图 4.14

将正方形沿着 y 轴正方向平移 10 个单位长度

我们可以调用 math3d 库中的 `m3dTranslationMatrix44` 函数来使用变换矩阵。

```
void m3dTranslationMatrix44(M3DMatrix44f m, float x, float y, float z);
```

### 旋转

为了将一个对象沿着 3 个坐标轴中的一个或者任意向量进行旋转，需要找到一个旋转矩阵，又有一个 math3d 函数来帮助我们了。

```
m3dRotationMatrix44(M3DMatrix44f m, float angle, float x, float y, float z);
```

在这里，我们围绕一个由  $x$ 、 $y$  和  $z$  变量指定的向量来进行旋转。旋转的角度沿逆时针方向按照弧度计算，由变量 `angle` 指定。在最简单的情况下，旋转只是围绕坐标系的一个坐标轴（ $x$ 、 $y$  或  $z$ ）进行旋转。

我们还可以围绕任意一个由  $x$ 、 $y$  和  $z$  变量指定的向量来进行旋转。要表示这个旋转轴，我们只要绘制一条从原点到由  $(x, y, z)$  表示的点的线段就可以了。举例来说，下面的代码创建一个旋转矩阵，可以使顶点沿着任意由  $(1, 1, 1)$  指定的轴旋转  $45^\circ$ ，如图 4.15 所示。

```
M3DMatrix44f m;  
m3dRotationMatrix(m3dDegToRad(45.0), 1.0f, 1.0f, 1.0f);
```

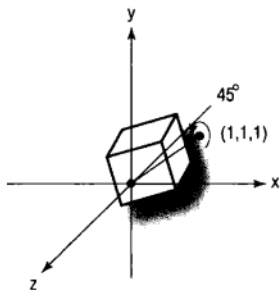


图 4.15

一个正方形围绕任意轴进行旋转

请注意在这个例子中 math3d 宏 `m3dDegToRad` 的使用。这个宏将角度值转换为弧度值，因为和计算机不同，大多数程序员比较习惯以角度为单位进行思考。使用这个宏来代替内联函数（对于 C++ 的忠实拥护者来说肯定听说过这个词）的一个好处是，如果这个值是一个硬编码（硬编码是指将可变变量用一个固定值来代替的方法——译者注）的文字，转换将发生在编译时，这样角度和弧度之间的转换就不会产生运行时损失。

## 缩放

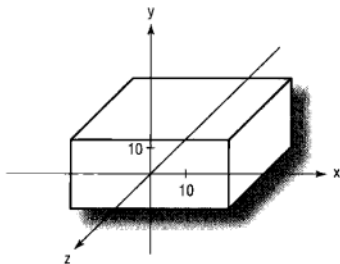
我们要讲的最后一个矩阵是缩放矩阵。缩放矩阵可以沿着 3 个坐标轴方向按照指定因子放大或缩小所有顶点，以改变对象的大小，使用 math3d 库创建一个缩放矩阵，方法与创建平移或旋转矩阵的方法类似。

```
M3DMatrix44f m;
void m3dScaleMatrix44(M3DMatrix44f m, float xScale, float yScale, float zScale);
```

缩放不一定是一致的，我们可以在不同的方向同时使用它来进行伸展和压缩。例如，一个  $10 \times 10 \times 10$  的正方体可以在 x 方向和 z 方向放大到原来的两倍，如图 4.16 所示。

图 4.16

立方体的不一致缩放



## 综合变换

我们很少会只进行这些变换类型中的一种。实际上，我们总是想同时进行这些变换。为了将对象移动到想要的位置，我们可能需要先将它平移到指定位置，然后再旋转到想要的方向。由于  $4 \times 4$  变换矩阵包含一个位置和一个方向，我们可能会想到，一个矩阵就可以完成这两种转换。我们是对的！

将两种变换加在一起很简单，只需将两个矩阵相乘。结果得到的矩阵包含结合到一起的转换，都在一个矩阵中了。“加”这个词在数学中当然是指加法，但实际上我们不会把这两个矩阵“加”到一起，它们是相乘的。为了解决这个术语上的冲突，我们经常使用术语“连接”（concatenate）来表示两种变换以这种方式结合。不过在矩阵乘法中有一个小陷阱需要注意，就是运算的顺序是有影响的。例如，用一个旋转矩阵乘以一个平移矩阵，与用一个平移矩阵乘以一个旋转矩阵是不同的。图 4.7 所示对这一点进行了讨论和演示。

math3d 库函数 m3dMatrixMultiply44 用来将两个矩阵相乘并返回运算结果。

```
void m3dMatrixMultiply44(M3DMatrix44f product,
                        const M3DMatrix44f a, const M3DMatrix44f b);
```

现在让我们来看一个将这些变换叠加到一起的具体例子。

### 4.4.2 运用模型视图矩阵

在第 2 章的 Move 示例程序中，我们让一个红色的方块随着按动方向键而在窗口中移动。这里使用的是强制性的方法，即更新三角形扇的坐标，然后重建图元批次。能够达到同样效果更好的方法是，一次性创建批次

(通常是以原点为中心), 然后在渲染这个批次时对顶点应用一个矩阵(实际上就是模型视图矩阵)。在原来的程序中, 我们使用了单位着色器, 这个着色器对顶点不做任何变换, 它只是传递这些顶点并在默认的笛卡尔坐标系中对它们进行渲染。另一个存储着色器, 即平面着色器, 接受  $4 \times 4$  变换矩阵作为它的参数之一。

```
GLShaderManager::UseStockShader(GLT_SHADER_FLAT, M3DMatrix44f m, GLfloat vColor[4]);
```

这个着色器在对图元进行渲染之前用每个向量乘以矩阵  $m$ 。在针对本章内容修改过的 Move 示例程序中, 我们使用两个变量  $yPos$  和  $xPos$  来记录正方形的位置。现在可以方便地创建一个变换矩阵了。

```
m3dTranslationMatrix44(mTranslationMatrix, xPos, yPos, 0.0f);
```

然后, 这个变换矩阵就可以在绘制对象之前被发送到着色器了, 如下所示。

```
shaderManager.UseStockShader(GLT_SHADER_FLAT, mTranslationMatrix, vRed);
squareBatch.Draw();
```

为了让事情更加有趣(同时也是为了演示一个重要问题), 我们在移动这个正方形的同时还对它进行旋转。在  $xy$  平面中旋转这个正方形也包括围绕  $z$  轴旋转。程序清单 4.1 展示了 Move 示例程序中的整个 RenderScene 函数。

程序清单 4.1 在屏幕上先平移然后旋转正方形的代码

```
void RenderScene(void)
{
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

    GLfloat vRed[] = { 1.0f, 0.0f, 0.0f, 1.0f };

    M3DMatrix44f mFinalTransform, mTranslationMatrix, mRotationMatrix;

    // Just Translate
    m3dTranslationMatrix44(mTranslationMatrix, xPos, yPos, 0.0f);

    // Rotate 5 degrees every time we redraw
    static float yRot = 0.0f;
    yRot += 5.0f;
    m3dRotationMatrix44(mRotationMatrix, m3dDegToRad(yRot), 0.0f, 0.0f, 1.0f);

    m3dMatrixMultiply44(mFinalTransform, mTranslationMatrix, mRotationMatrix);

    shaderManager.UseStockShader(GLT_SHADER_FLAT, mFinalTransform, vRed);
    squareBatch.Draw();

    // Perform the buffer swap
    glutSwapBuffers();
}
```

请注意我们是如何分别创建平移矩阵  $mTranslationMatrix$  和旋转矩阵  $mRotationMatrix$  的。然后将它们相乘, 以创建最终的变换矩阵。

```
m3dMatrixMultiply44(mFinalTransform, mTranslationMatrix, mRotationMatrix);
```

平面着色器只接受一个矩阵变量, 然后它会用这些顶点乘以这个矩阵。这个“模型视图”矩阵通过在默认坐标系中平移这些顶点来使我们的正方形在屏幕上移动, 我们可以回忆一下, 在这个坐标系中所有 3 个坐标轴范围都在  $-1$  和  $+1$  之间。然而, 这个简单的坐标系并不是总能满足我们的需要, 而且在更大的坐

标空间中考虑我们的对象会更加方便。那么就可能会有另外一格矩阵能够允许我们将任何我们想要的坐标空间缩放至-1 到+1 的范围内。确实，这就是第二种类型的矩阵变换，称为投影，很快就会介绍相关内容。

## 4.5 更多对象

对于演示的目的来说，正方形和三角形很快就变得有点乏味了。在更进一步学习之前，让我们先介绍一些 GLTools 库中内建的存储对象。我们可以回忆一下 GLBatch 类，这个类的目的是为了解决容纳一个顶点列表并将它们作为一个特定类型的图元批次来进行渲染。在这里我们将介绍一种新的类，即 GLTriangleBatch 类。这个类是专门作为三角形的容器的。每个顶点都可以有一个表面法线，以进行光照计算和纹理坐标。GLTriangleBatch 类确切的内部实现所使用的技术在第 12 章以后才会介绍。就现在而言，我们知道它们将三角形以更加高效的方式（索引顶点数组）进行组织，并且实际上将多边形存储在图形卡（使用定点缓冲区对象）上就够了。

### 4.5.1 使用三角形批次类

建立自己的三角形批次对象是一件非常简单的事。

首先，我们需要为对象创建一个事件。

```
GLTriangleBatch myCoolObject;
```

然后通知容器最多打算使用的顶点数，以开始创建网格。

```
myCoolObject.BeginMesh(200); // 200 verts in my cool object.
```

现在来添加三角形。AddTriangle 成员函数接受一个包含 3 个顶点的数组，一个包含 3 个法线的数组，以及一个包含 3 个纹理坐标的数组。

```
void GLTriangleBatch::AddTriangle(M3DVector3f verts[3], M3DVector3f vNorms[3],
                                   M3DVector2f vTexCoords[3])
```

不要担心会出现重复的顶点数据（读者可能会认为三角形带或三角形扇效率会更高）。在我们每一次添加一个顶点时，GLTriangleBatch 类都会搜索重复值并对我们的批次进行优化。实际上，对于非常大的批次来说，我们可能会发现这种操作在每次添加一个新三角形时都会越来越明显地降低速度。

当我们添加完三角形时，调用 End。

```
myCoolObject.End();
```

现在，我们只要选择想要的存储着色器并调用 Draw 函数。

```
myCoolObject.Draw();
```

我们当然也可以在自己的着色器中使用这些对象，我们将在第 6 章中介绍这方面的约定。

库包含很多实用函数，它们可以将一个对象填充到一个 `GLTriangleBatch` 类中。示例程序 `Objects` 随着按下空格键而重复进行这个过程，并且使用与我们在第3章的 `Primitives` 示例程序中使用过的相同的线框技术来对它们进行渲染。我们还可以使用方向键来沿  $x$  轴和  $y$  轴旋转每一个对象来更细致地观察它们。下面让我们一个一个地了解它们。

## 4.5.2 创建一个球体

许多“人工的”对象都使用一种基本形状，即简单的球形。`gltMakeSphere` 函数引用一个三角形批次、球的半径和组成球体的片段及其堆叠数量。

```
void gltMakeSphere(GLTriangleBatch& sphereBatch, GLfloat fRadius,
                  GLint iSlices, GLint iStacks);
```

图 4.17 所示显示了 `Objects` 示例程序中球体的输出结果。球体的半径意义非常明显，而 `iSlices` 和 `iStacks` 参数则需要进行一点解释。我们可以将球体想象成围绕成球形的一系列三角形带。参数 `iStacks` 是这些从球体底部堆叠到顶部的三角形带的数量。而 `iSlices` 参数则是围绕着球体排列的三角形对数。典型情况下一个对称性较好的球体的片段数量是堆叠数量的 2 倍。想一想为什么会是这样——围绕球体一周是  $360^\circ$ ，而从底部到顶部只有  $180^\circ$ （ $360^\circ$  的一半）。另外需要注意的一点是，这些球体都是围绕  $z$  轴的，这样  $+z$  就是球体的顶点，而  $-z$  则是球体的底。

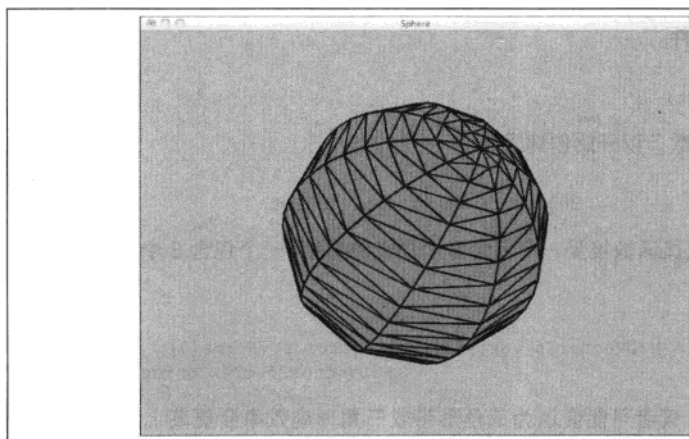


图 4.17

一个球体

## 4.5.3 创建一个花托

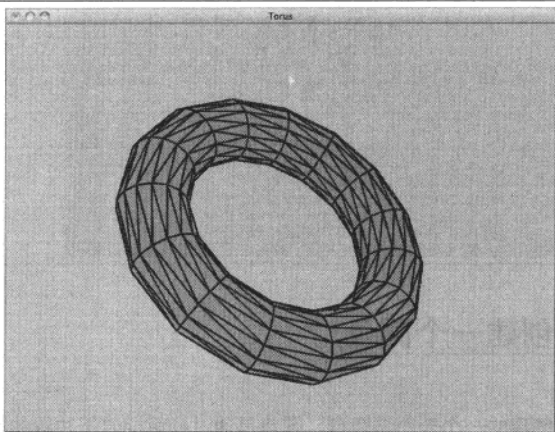
另外一个有趣并且有用的对象就是花托。花托是一种环状的像面包圈一样的物体，如图 4.18 所示。用来创建花托的 `GLTools` 函数是 `gltMakeTorus`。

```
void gltMakeTorus(GLTriangleBatch& torusBatch, GLfloat majorRadius, GLfloat
minorRadius, GLint numMajor, GLint numMinor);
```

其中 `majorRadius` 是花托中从中心到外边缘的半径,而 `minorRadius` 则是到内边缘的半径。`numMajor` 和 `numMinor` 参数的作用与球体中的 `iSlices` 和 `iStacks` 参数类似,他们是沿着主半径和内部较小半径的细分单元的数量。

图 4.18

一个花托



#### 4.5.4 创建一个圆柱或圆锥

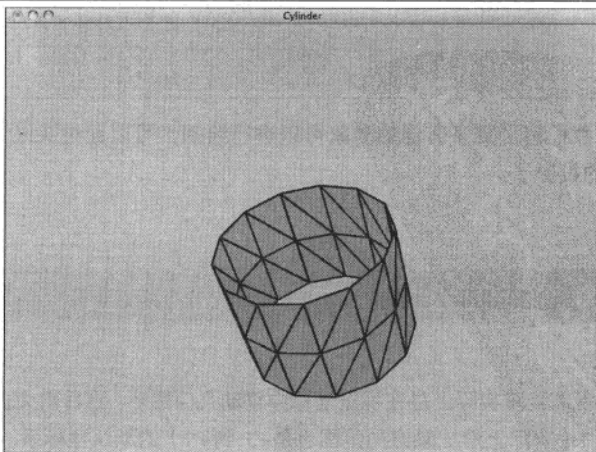
`glcMakeCylinder` 函数可以创建一个空心圆柱体。

```
void glcMakeCylinder(GLTriangleBatch& cylinderBatch, GLfloat baseRadius, GLfloat
topRadius, GLfloat fLength, GLint numSlices, GLint numStacks);
```

圆柱体从 0 开始向 z 轴正方向延伸,我们既可以指定底部半径,也可以指定顶部半径。图 4.19 展示了一个上下半径相等的圆柱体,而图 4.20 所示则展示了一端半径设置为 0 的圆柱体。这实际上是形成了一个锥体,但是我们还可以用同样简单的方式来创建一个漏斗形状。参数 `numSlices` 代表围绕 z 轴的三角形对的数量,而参数 `numStacks` 则代表从圆柱体底部堆叠到顶部圆环的数量。

图 4.19

一个两端半径相等的圆柱体



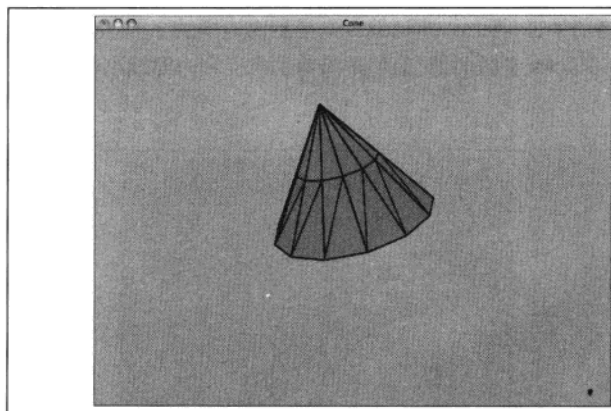


图 4.20

一个一端半径设置为 0 的圆柱体，也就是一个圆锥

### 4.5.5 创建一个圆盘

最后要创建的一个表面是圆盘。圆盘是通过分解成若干片段的三角形带绘制而成的。我们可以指定一个内部半径来创建一个类似垫圈的形状，也可以让这个值保持为 0 来创建一个实心圆盘。glcMakeDisk 函数，到目前为止看起来还是一个很熟悉的 API，它用圆盘形状来填充一个 GLTriangleBatch，如图 4.21 所示。

```
void glcMakeDisk(GLTriangleBatch& diskBatch, GLfloat innerRadius, GLfloat
outerRadius, GLint nSlices, GLint nStacks);
```

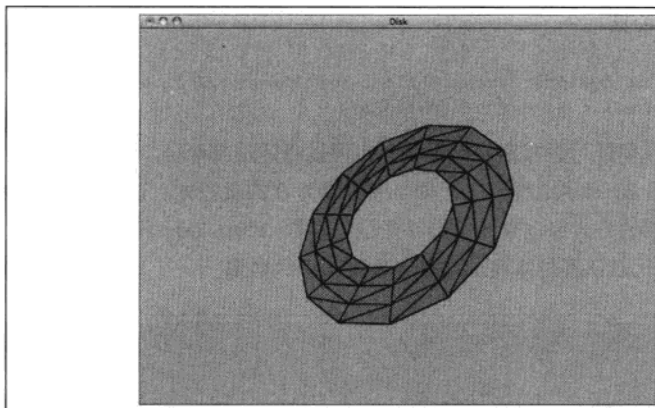


图 4.21

设置了内部和外部半径的圆盘

现在我们有更多有趣的对象可以进行绘制，可以回过头来讨论一下如何创建替换坐标系，或者说 3D 场景的投影了。

## 4.6 投影矩阵

模型视图矩阵实际上是在视觉坐标中移动几何图形。到目前为止，我们已经在屏幕或窗口上使用了范围为 -1 到 +1（实际上沿  $z$  轴方向范围也是 -1 到 +1）的默认坐标系。那么如果我们希望有不同的坐标系呢？

好吧, 事实是, 这个小小的坐标范围确实是硬件唯一能够接受的。那么使用不同坐标系的技巧就是, 将我们想要的坐标系变换到这个单位立方体中。我们使用一个新的矩阵来完成这项工作, 就是投影矩阵。接下来的两个示例程序 Orthographic (正交) 和 Perspective (透视), 将不会从源代码的角度进行详细的解释。这两个示例有助于我们弄清正投影和透视投影两者之间的区别。这两个互动示例可以使我们比较容易地看清透视是如何扭曲一个物体的外观的。如果可能的话, 我们应该在阅读下面两段内容的同时运行这些示例。

### 4.6.1 正投影

到目前为止我们在本书大部分内容中使用的正投影的所有面都是正方形的。前面、背面、顶面、地面、左面和右面的逻辑宽度都是相等的。这样就产生了一个平行投影, 这种投影在绘制从远处观察不产生任何透视缩短的特定物体时非常有用。这对于文本或者建筑绘图等我们用来在屏幕上精确地显示大小和尺寸的场所来说再好不过了。

图 4.22 显示了本章子目录下附带的源代码中 Orthographic 示例程序的输出结果。为了生成这个中空

图 4.22

一个用正投影显示的中空方管

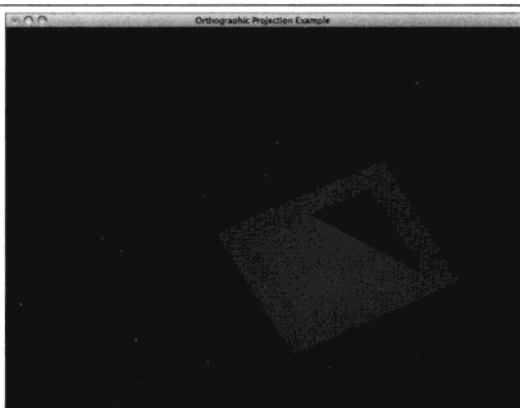
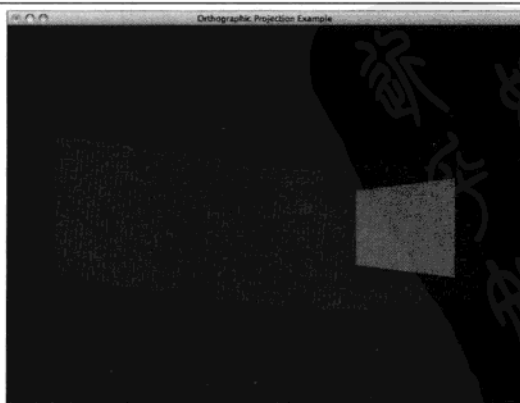


图 4.23

方管的侧视图, 显示它的长度



在图 4.24 中，我们会正面看向方管口。因为这个方管不会在远处汇聚在一点，所以这不完全是现实生活中一个这样的方管看起来的实际外观。为了增加一些透视效果，必须使用透视投影。

回忆一下第 3 章的内容，我们可以使用 `math3d` 库或 `GLFrustum` 类来创建一个正投影矩阵。

```
GLFrustum::SetOrthographic(GLfloat xMin, GLfloat xMax, GLfloat yMin, GLfloat yMax,  
    GLfloat zMin, GLfloat zMax);
```

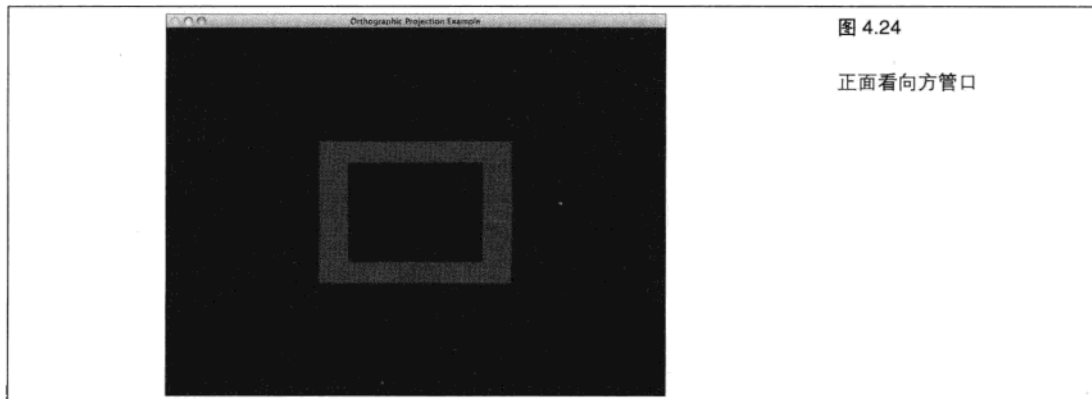


图 4.24

正面看向方管口

## 4.6.2 透视投影

我们可以回忆一下，在第 3 章中我们已经对透视投影进行了一些讨论。这一章的图 3.3 所示和图 3.4 所示展示了我们称为视景体的几何体。视景体是一个从窄端看向宽端的金字塔中截取的一部分。

作为第 3 章内容的回顾，我们使用 `GLFrustum` 类来设置透视投影。

```
GLFrustum::SetPerspective(float fFov, float fAspect, float fNear, float fFar);
```

`SetPerspective` 函数的参数是一个从顶点方向看去的视场角度（用角度值表示）、宽度和高度的比值（宽高比）和从近剪切面到远剪切面的距离（如图 3.4 所示）。用窗口或者视口的宽度（`w`）除以高度（`h`），我们就得到了宽高比的值。`GLFrustum` 类基于这些参数构建合适的  $4 \times 4$  投影矩阵，这个矩阵随后将成为我们整体变换管线的一部分。透视缩短为我们原来的方管正投影增加了现实感（如图 4.25、图 4.26 和图 4.27 所示）。我们所做的唯一实质性改变就是转换为透视投影。

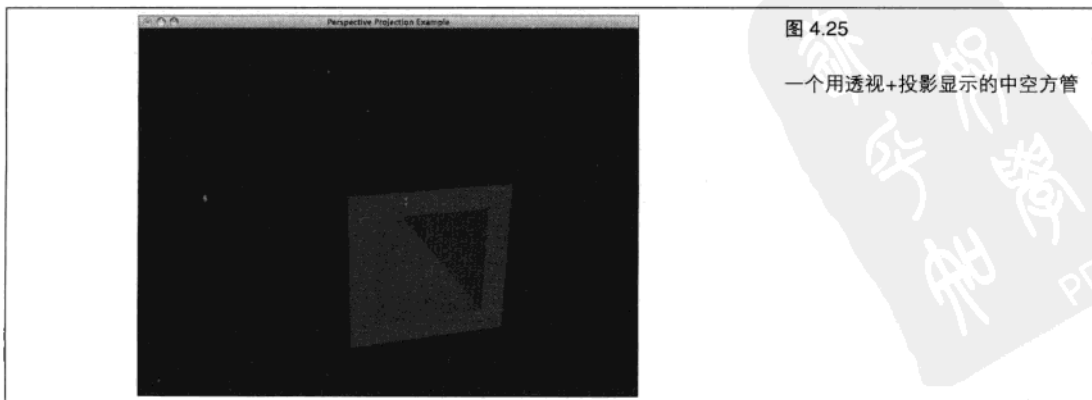


图 4.25

一个用透视+投影显示的中空方管

图 4.26

带有透视缩短效果的侧视图

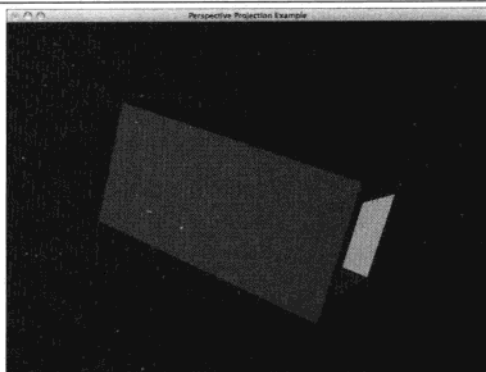
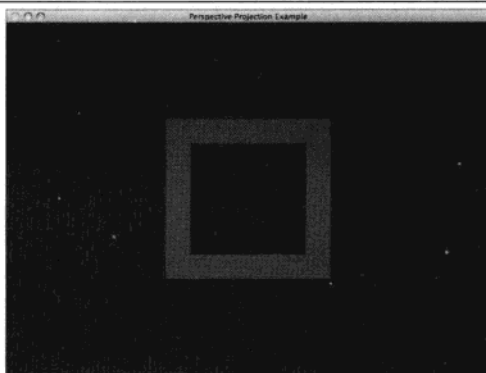


图 4.27

正面看向方管口，增加了透视图效果

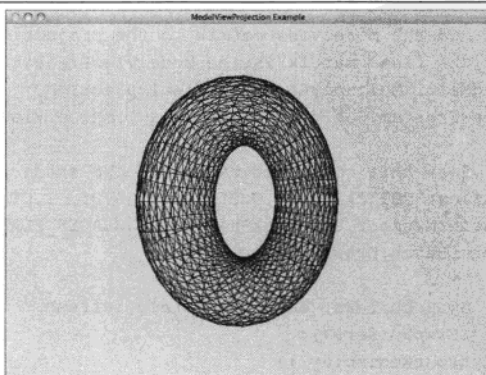


### 4.6.3 模型视图投影矩阵

下面让我们了解一下如何将所有这些内容结合起来。ModelviewProjection（模型视图投影矩阵）示例程序绘制了一个在屏幕中间旋转的线框花托。图 4.28 所示是这个示例程序输出的一个画面。我们使用 GLFrustum 类的一个叫做 viewFrustum 的实例来为渲染设置一个透视投影矩阵。程序清单 4.2 中的 ChangeSize 函数展示了我们是如何设置视口和投影矩阵的。

图 4.28

一个旋转的花托，由 ModelviewProjection（模型视图投影）矩阵进行变换



## 程序清单 4.2 ModelViewProjection 示例程序的矩阵操作

```

// Global view frustum class
GLFrustum      viewFrustum;

.....

// Set up the viewport and the projection matrix
void ChangeSize(int w, int h)
{
    // Prevent a divide by zero
    if(h == 0)
        h = 1;

    // Set Viewport to window dimensions
    glViewport(0, 0, w, h);

    viewFrustum.SetPerspective(35.0f, float(w)/float(h), 1.0f, 1000.0f);
}

// Called to draw scene
void RenderScene(void)
{
    // Set up time based animation
    static CStopWatch rotTimer;
    float yRot = rotTimer.GetElapsedSeconds() * 60.0f;

    // Clear the window and the depth buffer
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Matrix Variables
    M3DMatrix44f mTranslate, mRotate, mModelview, mModelViewProjection;

    // Create a translation matrix to move the torus back and into sight
    m3dTranslationMatrix44(mTranslate, 0.0f, 0.0f, -2.5f);

    // Create a rotation matrix based on the current value of yRot
    m3dRotationMatrix44(mRotate, m3dDegToRad(yRot), 0.0f, 1.0f, 0.0f);

    // Add the rotation to the translation, store the result in mModelView
    m3dMatrixMultiply44(mModelview, mTranslate, mRotate);

    // Add the modelview matrix to the projection matrix,
    // the final matrix is the ModelViewProjection matrix.
    m3dMatrixMultiply44(mModelViewProjection,
        viewFrustum.GetProjectionMatrix(), mModelview);

    // Pass this completed matrix to the shader, and render the torus
    GLfloat vBlack[] = { 0.0f, 0.0f, 0.0f, 1.0f };
    shaderManager.UseStockShader(GLT_SHADER_FLAT, mModelViewProjection, vBlack);
    torusBatch.Draw();

    // Swap buffers, and immediately refresh
    glutSwapBuffers();
    glutPostRedisplay();
}

```

在函数中，我们创建了 4 个  $4 \times 4$  矩阵变量。mTranslate 变量保存初始变换，这时将花托沿着 z 轴负方向移动 2.5 个单位长度。

```
m3dTranslationMatrix44(mTranslate, 0.0f, 0.0f, -2.5f);
```

然后创建一个旋转矩阵，并将它保存在 mRotate 中。

```
m3dRotationMatrix44(mRotate, m3dDegToRad(yRot), 0.0f, 1.0f, 0.0f);
```

请注意我们是如何使用 CStopWatch 类（GLTools 库的一个组成部分）来基于经过的时间长短设置旋转速度的。一开始将旋转速度设在了每秒  $60^\circ$ 。我们应该总是根据经过的时间来设置动画率，而不是采用单纯的基于帧的方式。例如，像下面这样编写动画代码是很有吸引力的。

```
static GLfloat yRot = 0;
yRot += 1.0f;
m3dRotationMatrix44(mRotate, m3dDegToRad(yRot), 0.0f, 1.0f, 0.0f);
```

这样的代码会使对象在帧速率低时旋转得很慢，而在帧速率高时旋转得很快，所以程序员会倾向于改变加到 yRot 的数字，直到动画看起来正常为止（真是简单粗糙的编程方式！）。然而，随着机器、驱动程序版本等因素的改变，帧速率也将随之改变，这将在不同的机器上产生不可预料的动画速率。但是，时间则是以恒定速度流动的，不管帧速率如何。更高的帧速率应该会产生更平滑的动画，而不是更快的动画。

现在让我们回到变换花托的任务上来——我们要进行的下一步工作是通过进行一次矩阵乘法操作来同时添加平移和旋转。请记住，操作的顺序非常重要，而在本例中先进行平移，然后再进行旋转。

```
m3dMatrixMultiply44(mModelview, mTranslate, mRotate);
```

现在，这个花托应该呈现在我们面前，并在正确的位置旋转了，至少对于模型视图矩阵来说是这样的。不过，不要忘记为了想要的坐标系而设置了一个透视投影矩阵。现在，我们需要将这个坐标系缩减到单元正方体范围，通过用投影矩阵乘以模型视图矩阵来完成这项工作。再次强调，操作的顺序非常重要！

```
m3dMatrixMultiply44(mModelViewProjection, viewFrustum.GetProjectionMatrix(),
    mModelview);
```

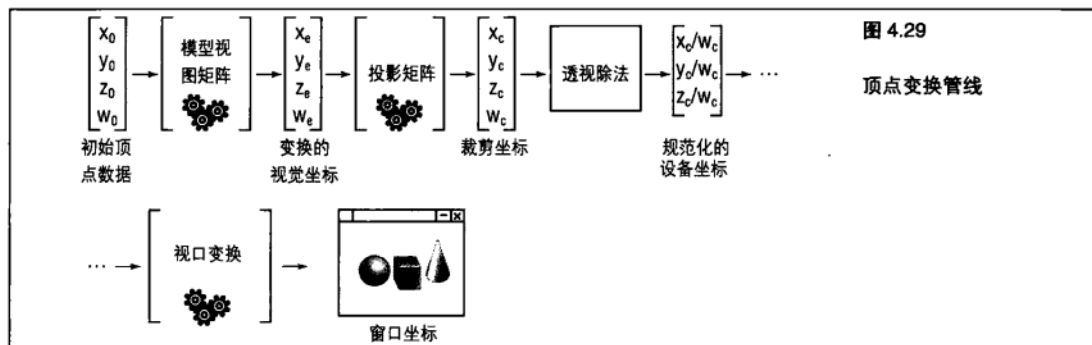
结果得到的矩阵 mModelViewProjection 包含所有的变换和到屏幕的投影的串联形式。这只是简单的魔术而已！或许读者现在已经想要开始研读附录 A 中列出的参考书了。最后一步是将我们的矩阵传递到平面着色器并提交花托属性。平面着色器的工作只是使用提供的矩阵来对顶点进行转换（这是通过向量与矩阵的乘法来完成的），并且使用指定的颜色对几何图形进行着色以得到实心几何图形，在本例中使用的是黑色。

```
shaderManager.UseStockShader(GLT_SHADER_FLAT, mModelViewProjection, vBlack);
torusBatch.Draw();
```

## 4.7 变换管线

现在我们已经了解了如何用模型视图矩阵和投影矩阵在屏幕上显示想要的东西，下面让我们从整体上了解一下变换管线。图 4.29 所示是处理过程的流程图。

首先，我们的顶点将被视为一个  $1 \times 4$  矩阵，其中前 3 个值为  $x$ 、 $y$  和  $z$  坐标。第 4 个数字是一个缩放因子，如果需要的话我们可以手动进行设置。这就是  $w$  坐标，通常在默认情况下为 1.0，我们很少会真正去直接修改这个值。然后，顶点将乘以模型视图矩阵，生成变换的视觉坐标。随后，视觉坐标再乘以投影矩阵，生成裁剪坐标。裁剪坐标值位于我们前面提到的  $\pm 1.0$  单位坐标系内。将有效地将所有位于这个裁剪空间之外的数据消除掉。裁剪坐标后再除以  $w$  坐标，生成规范化的设备坐标。其中  $w$  值可能会被投影矩阵或模型视图矩阵修改，这取决于所发生的变换。透视除法将作为图元装配过程的一部分进行。最后，坐标三元组将通过视口变换被映射到 2D 平面上。这项操作也是由一个矩阵来表示的，但不能直接指定或者修改这个矩阵。OpenGL 将在内部根据指定的 `glViewport` 值来设置这个矩阵。



### 4.7.1 使用矩阵堆栈

因为矩阵乘法是 3D 图形中如此重要的组成部分，所以几乎所有程序员的工具箱中都包含了一系列函数类，用来创建和操作矩阵乘法。实际上，`math3d` 库就包含了类型丰富的函数来进行这些工作。

在分层方式中，一个或多个对象会相对于另一个对象进行绘制，在这种方式中经常会应用到变换。这样，就会需要大量由用户代码进行构造和管理的矩阵在 3D 空间中建立复杂的场景。

习惯上，我们会使用一个矩阵堆栈来帮助完成这些工作，而 `GLTools` 库则会在 `math3d` 矩阵函数顶部建立实用类。这个类称为 `GLMatrixStack`。熟悉兼容版本中现在已经“不推荐”的 OpenGL 矩阵堆栈的读者会对这个类感到很熟悉。

这个类的构造函数允许指定堆栈的最大深度，默认的堆栈深度为 64。这个矩阵堆栈在初始化时已经在堆栈中包含了单位矩阵。

```
GLMatrixStack::GLMatrixStack(int iStackSize = 64);
```

我们可以通过调用在顶部载入这个单位矩阵。

```
void GLMatrixStack::LoadIdentity(void);
```

或者可以在堆栈的顶部载入任何矩阵。

```
void GLMatrixStack::LoadMatrix(const M3DMatrix44f m);
```

此外，我们可以用一个矩阵乘以矩阵堆栈的顶部矩阵，相乘得到的结果随后将存储在堆栈的顶部。

```
void GLMatrixStack::MultMatrix(const M3DMatrix44f);
```

最后，只要用 `GetMatrix` 函数就可以获得矩阵堆栈顶部的值，这个函数可以进行两次重载，以适应 `GLShaderManager` 的使用，或者仅仅是获得顶部矩阵的副本。

```
const M3DMatrix44f& GLMatrixStack::GetMatrix(void);
void GLMatrixStack::GetMatrix(M3DMatrix44f mMatrix);
```

## 压栈与出栈

一个矩阵的真正价值在于通过压栈操作存储一个状态，然后通过出栈操作恢复这个状态。通过 `GLMatrixStack` 类，我们可以使用 `PushMatrix` 函数将矩阵压入堆栈来存储当前矩阵值。

这样做实际上是复制了当前矩阵值，并将新的值放入了堆栈的顶部。类似地，`PopMatrix` 将移除顶部矩阵，并恢复它下面的值。以上每种情况都有几次重载。

```
void GLMatrixStack::PushMatrix(void);
void PushMatrix(const M3DMatrix44f mMatrix);
void PushMatrix(GLFrame& frame);

void GLMatrixStack::PopMatrix(void);
```

除了将当前矩阵压入堆栈之外，通过 `M3DMatrix44f` 数据类型或者 `GLFrame` 类（很快我们要更详细地介绍 `GLFrame` 类），还可以将任意一个矩阵压入到堆栈的顶部。

## 仿射变换

`GLMatrixStack` 类也内建了对创建旋转、平移和缩放矩阵的支持。相应的函数列出如下。

```
void MatrixStack::Rotate(GLfloat angle, GLfloat x, GLfloat y, GLfloat z);
void MatrixStack::Translate(GLfloat x, GLfloat y, GLfloat z);
void MatrixStack::Scale(GLfloat x, GLfloat y, GLfloat z);
```

这些函数与它们对应的低阶 `math3d` 函数的运作类似，但有一点不同。`Rotate` 函数接受角度值而不是弧度值，以更加接近目前已经“不推荐”的 OpenGL 函数 `glRotate`。所有这 3 个函数都可以创建恰当的矩阵，然后用这个矩阵乘以矩阵堆栈顶部的元素，实际上就是对当前矩阵添加变换（我们应该还记得将矩阵相乘来添加变换的方式）。

## 4.7.2 管理管线

读者可能会猜想，为模型视图矩阵和投影矩阵都建立一个矩阵堆栈会有很多优势。我们还经常需要检索这两种矩阵并将它们相乘以得到模型视图投影矩阵。另一种有用的矩阵就是正规矩阵，它用来进行光照计算，并且可以从模型视图矩阵推导出来。另外一种实用类 `GLGeometryTransform` 为我们跟踪记录这两种矩阵堆栈，并快速检索模型视图投影矩阵的顶部或正规矩阵堆栈的顶部。

下面让我们了解一下在 `SphereWorld` 示例程序中如何综合使用所有这些类。在本章，`SphereWorld`

示例程序将进行几次修改，而最开始的时候它只是在绿色网格背景上显示一个旋转线框模式的花托，如图 4.30 所示。

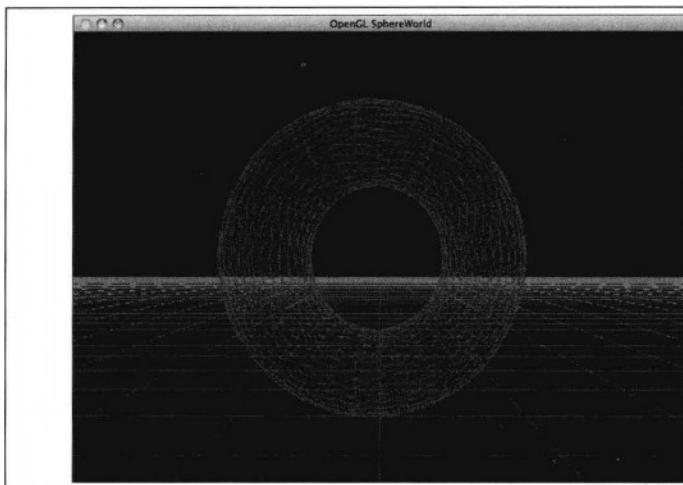


图 4.30

最初的 SphereWorld 示例程序

在 SphereWorld 示例程序源代码的顶部，为模型视图矩阵和投影矩阵声明了 GLMatrixStack 的一个实例。我们使用 GLFrustum 类构造投影矩阵，最后用 GLGeometryTransform 类的一个实例来管理矩阵堆栈。

```
GLMatrixStack      modelViewMatrix;      //模型视图矩阵
GLMatrixStack      projectionMatrix;     //投影矩阵
GLFrustum          viewFrustum;         //视景体
GLGeometryTransform transformPipeline;   //几何变换管线
```

程序清单 4.3 展示了最初的 SphereWorld 示例程序中的 ChangeSize 和 RenderScene 函数。

程序清单 4.3 最初的 SphereWorld 变换

```
////////////////////////////////////
// 屏幕改变大小或初始化
void ChangeSize(int nWidth, int nHeight)
{
    glViewport(0, 0, nWidth, nHeight);

    //创建投影矩阵，并将它载入到投影矩阵堆栈中
    viewFrustum.SetPerspective(35.0f, float(nWidth)/float(nHeight), 1.0f, 100.0f);
    projectionMatrix.LoadMatrix(viewFrustum.GetProjectionMatrix());

    // 设置变换管线以使用两个矩阵堆栈
    transformPipeline.SetMatrixStacks(modelViewMatrix, projectionMatrix);
}

// 进行调用以绘制场景
void RenderScene(void)
{
    //颜色值
    static GLfloat vFloorColor[] = { 0.0f, 1.0f, 0.0f, 1.0f};
    static GLfloat vTorusColor[] = { 1.0f, 0.0f, 0.0f, 1.0f };

    // 基于时间的动画
    static CstopWatch rotTimer;
```

```

float yRot = rotTimer.GetElapsedSeconds() * 60.0f;

// 清除颜色缓冲区和深度缓冲区
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// 保存当前的模型视图矩阵 (单位矩阵)
modelViewMatrix.PushMatrix();

// 绘制背景
shaderManager.UseStockShader(GLT_SHADER_FLAT,
                             transformPipeline.GetModelViewProjectionMatrix(),
                             vFloorColor);

floorBatch.Draw();

// 绘制旋转的花托
modelViewMatrix.Translate(0.0f, 0.0f, -2.5f);
modelViewMatrix.Rotate(yRot, 0.0f, 1.0f, 0.0f);
shaderManager.UseStockShader(GLT_SHADER_FLAT,
                             transformPipeline.GetModelViewProjectionMatrix(),
                             vTorusColor);

torusBatch.Draw();

// 保存以前的模型视图矩阵 (单位矩阵)
modelViewMatrix.PopMatrix();

// 进行缓冲区交换
glutSwapBuffers();

// 通知 GLUT 再进行一次同样操作
glutPostRedisplay();
}

```

在 `ChangeSize` 函数中, 我们对透视投影进行设置。因为在这里被告知窗口的大小 (或这些大小值是否改变), 所以将这些代码放在这里是合适的。`GLFrustum` 类的 `viewFrustum` 实例会为我们设置投影矩阵, 然后将它载入到投影矩阵对象 `projectionMatrix`。

```

// 创建投影矩阵, 并将它载入到投影矩阵堆栈中
viewFrustum.SetPerspective(35.0f, float(nWidth)/float(nHeight), 1.0f, 100.0f);
projectionMatrix.LoadMatrix(viewFrustum.GetProjectionMatrix());

```

在这里我们要做的最后一件事情是初始化 `GLGeometryTransform` 的实例 `transformPipeline`, 通过将它的内部指针设置为指向模型视图矩阵堆栈和投影矩阵堆栈实例来完成这项任务。

```
transformPipeline.SetMatrixStacks(modelViewMatrix, projectionMatrix);
```

我们确实也可以在 `SetupRC` 函数中完成这项工作, 但是在窗口大小改变时重新设置它们并没有坏处, 而且这样可以一次性完成矩阵和管线的设置。

接下来, 在 `RenderScene` 函数中, 我们开始渲染多边形, 首先保存模型视图矩阵, 这个矩阵已经被默认设置为单位矩阵。

```

// 保存当前的模型视图矩阵 (单位矩阵)
modelViewMatrix.PushMatrix();

```

这样做看起来似乎没什么意义, 因为我们接下来要做的事情是绘制背景, 而背景完全没有进行任何转换。在开始传递渲染时保存矩阵状态, 然后在结束时使用相应的 `PopMatrix` 恢复它, 是一种很好的做法。这样就不必在每一次渲染时都重载单位矩阵了, 何况在添加照相机时, 为了组织目的, 它很快就会派上用场。

现在，代码终于可以将花托移动到位了。首先，调用 `Translate` 将一个平移矩阵应用到矩阵堆栈顶部。这样就将花托从原点（我们所在的位置）移开，以便我们能看到它。然后使用 `Rotate` 进行旋转。参数 `Rotate` 使花托围绕  $y$  轴进行旋转，而 `yRot` 则来自从上一帧以来经过的时间值。

```
//绘制旋转的花托
modelViewMatrix.Translate(0.0f, 0.0f, -2.5f);
modelViewMatrix.Rotate(yRot, 0.0f, 1.0f, 0.0f);
shaderManager.UseStockShader(GLT_SHADER_FLAT,
                             transformPipeline.GetModelViewProjectionMatrix(),
                             vTorusColor);

torusBatch.Draw();
```

随后，最后的矩阵将作为一个 `Uniform` 值传递到着色器，而花托批次则被提交来渲染对象。与其获取当前模型视图矩阵和投影矩阵再将它们相乘，现在可以简单地从 `transformPipeline` 获得串联矩阵。

这样可以使代码更加清晰，减少混乱，并且容易阅读。这个变换矩阵仍然在堆栈的顶部，所以我们调用 `PopMatrix` 来移出它并恢复单位矩阵。

```
modelViewMatrix.PopMatrix();
```

### 4.7.3 加点调料

`SphereWorld` 示例程序非常简单——只有一个固定的几何图形（地板）和一个变换（旋转）的对象。到目前为止，看起来我们进行了大量工作，得到的真正收益却很少。让我们向 `SphereWorld` 示例程序添加一些其他东西，看看它是如何适应新的变换管线的。

图 4.31 所示为 `SphereWorld2` 示例程序的输出。在 `SphereWorld2` 中，我们添加了蓝色球体，这次它以花托为中心旋转。下面让我们来看一看做了哪些改变，参见程序清单 4.4。在这个程序清单中，我们添加了行数，以方便讨论所发生的过程。

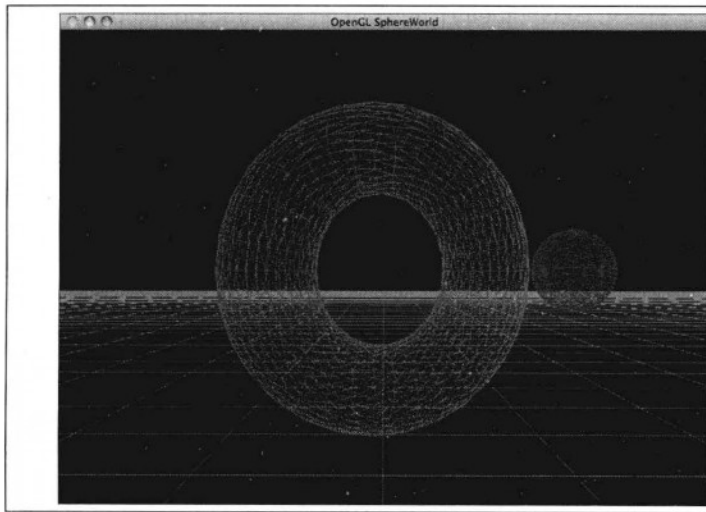


图 4.31

`SphereWorld2` 中添加了一个球体

程序清单 4.4 在 SphereWorld2 中添加一个球体

```

1 // 保存当前的模型视图矩阵（单位矩阵）
2 modelViewMatrix.PushMatrix();
3
4 //绘制背景
5 shaderManager.UseStockShader(GLT_SHADER_FLAT,
6 transformPipeline.GetModelViewProjectionMatrix(),
7 vFloorColor);
8 floorBatch.Draw();
9
10 //绘制旋转的花托
11 modelViewMatrix.Translate(0.0f, 0.0f, -2.5f);
12
13 //保存平移
14 modelViewMatrix.PushMatrix();
15
16 // 应用旋转并绘制花托
17 modelViewMatrix.Rotate(yRot, 0.0f, 1.0f, 0.0f);
18 shaderManager.UseStockShader(GLT_SHADER_FLAT,
19 transformPipeline.GetModelViewProjectionMatrix(),
20 vTorusColor);
21 torusBatch.Draw();
22 modelViewMatrix.PopMatrix(); // “消除” 以前的旋转
23
24 //应用另一个旋转，然后进行评议，然后再绘制球体
25 modelViewMatrix.Rotate(yRot * -2.0f, 0.0f, 1.0f, 0.0f);
26 modelViewMatrix.Translate(0.8f, 0.0f, 0.0f);
27 shaderManager.UseStockShader(GLT_SHADER_FLAT,
28 transformPipeline.GetModelViewProjectionMatrix(),
29 vSphereColor);
30 sphereBatch.Draw();
31
32 //保存以前的模型视图矩阵（单位矩阵）
33 modelViewMatrix.PopMatrix();

```

在本例，我们在第 14 行添加了一个新的 PushMatrix，就在进行平移操作将视场向后移动并远离我们之后。然后，就像第一个程序一样，继续在第 17 行进行旋转操作，最终对花托进行渲染。我们在第 22 行调用了 PopMatrix。这样就恢复了在第 14 行保存的矩阵；实际上，对于矩阵堆栈而言，现在旋转还没有发生。如果使用这个矩阵渲染了球体，那么它将在旋转的花托中心静止不动，因为它将简单地沿着  $z$  轴负方向平移 2.5 个单位长度。试一试吧！然而为了达到目的，我们要应用不同的旋转，围绕  $y$  轴向着相反的方向，并且为了看起来更有趣，我们将速度加倍了。进行旋转后，紧接着在第 26 行进行了平移，将球体沿  $x$  轴移出。得到的总效果就是球体围绕着花托旋转。我们要做的最后一件事就是在第 33 行最后一次调用 PopMatrix 来恢复矩阵堆栈顶部的单位矩阵。现在我们很可能开始明白保存和恢复变换矩阵非常有用了。但是等一等……它会变得更好！

## 4.8 使用照相机和角色进行移动

为了在 3D 场景中表示任何对象的位置和方向，我们可以使用一个  $4 \times 4$  矩阵表示它的变换。但是，直接操纵矩阵仍然显得有点笨拙，因此程序员总是想方设法用更简洁的方式表示空间中的坐标和方向。固定的

物体（如地形，或者 SphereWorld 中的地板）通常不进行变换，它们的顶点通常准确地指定几何图形在空间中应该怎样进行绘制。在场景中移动的物体通常称为角色（Actor），就像舞台上的演员一样。

角色有它们自己的变换，而且角色的变换常常不仅与全局坐标系（视觉坐标系）有关，也与其他角色有关。每个有自己的变换角色都被称为有自己的参考帧，或者本地对象坐标系。在本地和全局坐标系之间进行转换常常是非常有用的，而且对于许多非渲染相关的几何图形测试来说也是如此。

### 4.8.1 角色帧

我们可以用一种简单灵活的方式表示参考帧，就是使用一个数据结构（或者 C++ 中的类），这个类中包含空间中的一个位置、一个指向前方的向量和一个指向上方的向量。使用这些量，我们可以在空间中唯一确定一个给定的位置和方向。下面是一个数据结构的例子 GLFrame，取自 GLTools library，它利用了 math3d 库，并将所有这些信息一起进行了存储。

```
class GLFrame
{
    protected:
        M3DVector3f vLocation;
        M3DVector3f vUp;
        M3DVector3f vForward;

    public:
        .....
};
```

使用一个像这样的帧来表示一个对象的位置和方向是一种强大的机制。首先，我们可以使用这些数据直接创建一个  $4 \times 4$  变换矩阵。让我们回过头来看一下图 4.12，其中向上的向量成为矩阵的  $y$  列，向前的向量作为矩阵的  $z$  列，而位置则作为平移列向量。这其中唯独缺少  $x$  列向量，由于我们知道这 3 个轴都是单位长度，并且是相互垂直（正交）的，我们可以通过计算  $y$  向量和  $z$  向量叉乘的结果来获得  $x$  列向量。程序清单 4.5 展示了 GLFrame 的 GetMatrix 方法，它正是用来完成这项工作的。

程序清单 4.5 从一个帧导出  $4 \times 4$  矩阵的代码

```
////////////////////////////////////
// 从一个参考帧导出  $4 \times 4$  矩阵的代码
void GLFrame::GetMatrix(M3DMatrix44f mMatrix, bool bRotationOnly = false)
{
    // 计算向右的向量 (x)，并将它放在矩阵中的右边
    M3DVector3f vXAxis;
    m3dCrossProduct(vXAxis, vUp, vForward);

    // 设置矩阵列，并没有填充第四个值
    m3dSetMatrixColumn44(matrix, vXAxis, 0);
    matrix[3] = 0.0f;

    // y 列
    m3dSetMatrixColumn44(matrix, vUp, 1);
    matrix[7] = 0.0f;

    // z 列
    m3dSetMatrixColumn44(matrix, vForward, 2);
    matrix[11] = 0.0f;
```

```
//平移 (已经完成)
if(bRotationOnly == true)
{
    matrix[12] = 0.0f;
    matrix[13] = 0.0f;
    matrix[14] = 0.0f;
}
else
    m3dSetMatrixColumn44(matrix, vOrigin, 3);

matrix[15] = 1.0f;
}
```

类包含 3 次重用, 甚至允许我们使用 GLFrame 类来代替一个完整的矩阵。

```
void GLMatrixStack::LoadMatrix(GLFrame& frame);
void GLMatrixStack::MultMatrix(GLFrame& frame);
void GLMatrixStack::PushMatrix(GLFrame& frame);
```

## 4.8.2 欧拉角: “卢克! 请使用帧”

某些图像编程书籍会推荐一种甚至更加简单的机制来存储一个对象的位置和方向, 即欧拉角 (Euler angles)。欧拉角所需要的空间更少, 因为实际上只存储一个物体的位置以及 3 个角度 (表示沿 x 轴、y 轴和 z 轴的旋转, 有时称为 yaw、pitch 和 roll)。这种结构可以用来表示一架飞机的位置和方向。

```
struct EULER {
    M3DVector3f vPosition;
    GLfloat fRoll;
    GLfloat fPitch;
    GLfloat fYaw;
};
```

欧拉角稍微有点难以理解, 有时会被业内人士称为“油滑的角”。它的第一个问题是一个给定的位置和方向可以用一组以上的欧拉角来表示。当试图判断物体是如何从一个方向平滑地移动到另一个方向时, 有多组角度就可能会导致问题。有时候, 还会遇到第二个问题, 称为“万向节锁”。这个问题可能会导致围绕一个轴进行旋转。最后, 当简单地向前移动视线时, 计算新的欧拉角显得非常麻烦。或者, 如果我们想绕一个本地轴旋转, 并计算新的欧拉角时, 也会觉得这个任务比较繁琐。

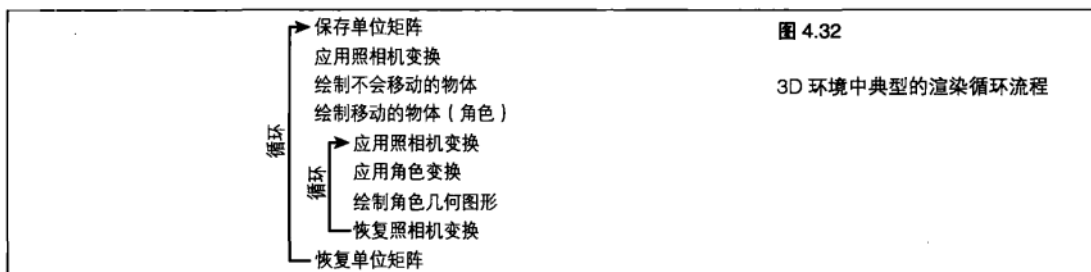
目前, 有些文献试图用一种称为四元组 (quaternions) 的数学工具来解决欧拉角问题。四元组本身就很难理解, 而且它并不能解决通过使用前面介绍过的参考帧方法无法解决的任何欧拉角问题。的确, 四元组要优于欧拉角很多, 但是四元组关于帧的变量并没有太大的说服力。我们已经承诺本书不会在数学问题上钻得太深, 因此不打算对不同系统的优劣进行辩论。但是还是应该说明, 四元组和线性代数 (矩阵) 之间的争议已经持续了上百年, 远远早于它们在计算机图形学上的应用!

## 4.8.3 照相机管理

OpenGL 中其实并不存在像照相机变换这样的东西。我们用照相机作为一种有用的比喻, 帮助我们在某些类型的 3D 环境中管理观察点。如果把照相机想像成一种物体, 它在空间中具有一些位置和—

些特定的方向，就会发现当前的参考帧系统在 3D 环境中既可以用角色表示，也可以用照相机表示。

为了应用照相机变换，我们使用照相机的角色变换并对它进行反转，这样向后移动照相机就相当于向前移动整个场景。类似地，向左旋转相当于把整个场景向右旋转。为了渲染一个特定的场景，通常使用图 4.32 所示的描述方法。



类包含一个 GLFrame 函数，这个函数用来检索条件适合的照相机矩阵。

```
void GetCameraMatrix(M3DMatrix44f m, bool bRotationOnly = false);
```

在这里我们做了一些变通，所以只能获得照相机的旋转变换。这里的 C++ 默认参数允许忽略这一点，除非对这种特性有特别的需要。例如，一种在身临其境的环境中经常应用的技术就是天空盒 (sky box)。所谓的天空盒只是一个带有天空图片的大盒子。我们和周围的环境都在这个盒子中进行渲染。随着我们的移动，天空盒也应该跟着移动（只是进行旋转），但是我们并不希望能够走到天空的尽头。我们应该只在天空盒上应用照相机变换的旋转分量，而场景中其他所有东西都应该通过完整的照相机变换进行变换。

让我们在 SphereWorld 示例程序中添加一个照相机，这样我们可以在总体上更好地理解这种机制是如何运作的。程序清单 4.6 展示了 SphereWorld2 的重要部分，其中包括通过方向键进行移动的能力。

程序清单 4.6 在 SphereWorld2 中添加一个照相机

```
GLFrame          cameraFrame; // 全局照相机实例

//移动 zhoxiangji 参考帧来对方向键作出响应
void SpecialKeys(int key, int x, int y)
{
    float linear = 0.1f;
    float angular = float(m3dDegToRad(5.0f));

    if(key == GLUT_KEY_UP)
        cameraFrame.MoveForward(linear);

    if(key == GLUT_KEY_DOWN)
        cameraFrame.MoveForward(-linear);

    if(key == GLUT_KEY_LEFT)
        cameraFrame.RotateWorld(angular, 0.0f, 1.0f, 0.0f);

    if(key == GLUT_KEY_RIGHT)
        cameraFrame.RotateWorld(-angular, 0.0f, 1.0f, 0.0f);
}

// 进行调用以绘制场景
void RenderScene(void)
{
    //颜色值
    static GLfloat vFloorColor[] = { 0.0f, 1.0f, 0.0f, 1.0f };
    static GLfloat vTorusColor[] = { 1.0f, 0.0f, 0.0f, 1.0f };
}
```

```

static GLfloat vSphereColor[] = { 0.0f, 0.0f, 1.0f, 1.0f };

// 基于时间的动画
static CStopWatch rotTimer;
float yRot = rotTimer.GetElapsedSeconds() * 60.0f;

// 清除颜色缓冲区和深度缓冲区
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// 保存当前的模型视图矩阵（单位矩阵）
modelViewMatrix.PushMatrix();

M3DMatrix44f mCamera;
cameraFrame.GetCameraMatrix(mCamera);
modelViewMatrix.PushMatrix(mCamera);

// 绘制背景
shaderManager.UseStockShader(GLT_SHADER_FLAT,
    transformPipeline.GetModelViewProjectionMatrix(),
    vFloorColor);
floorBatch.Draw();

// 绘制旋转的花托
modelViewMatrix.Translate(0.0f, 0.0f, -2.5f);

// 保存平移
modelViewMatrix.PushMatrix();

    // 应用旋转并绘制花托
    modelViewMatrix.Rotate(yRot, 0.0f, 1.0f, 0.0f);
    shaderManager.UseStockShader(GLT_SHADER_FLAT,
        transformPipeline.GetModelViewProjectionMatrix(),
        vTorusColor);
    torusBatch.Draw();
modelViewMatrix.PopMatrix(); // 消除以前的旋转

// 应用另一个旋转，然后进行平移，然后再绘制球体
modelViewMatrix.Rotate(yRot * -2.0f, 0.0f, 1.0f, 0.0f);
modelViewMatrix.Translate(0.8f, 0.0f, 0.0f);
shaderManager.UseStockShader(GLT_SHADER_FLAT,
    transformPipeline.GetModelViewProjectionMatrix(),
    vSphereColor);
sphereBatch.Draw();

// 保存以前的模型视图矩阵（单位矩阵）
modelViewMatrix.PopMatrix();
modelViewMatrix.PopMatrix();

// 进行缓冲区交换
glutSwapBuffers();

// 通知 GLUT 再进行一次同样操作
glutPostRedisplay();
}

```

只要方向键被按下，就要调用 `SpecialKeys` 函数。在照相机对象 `cameraFrame` 上调用 `GLFrame` 类的成员函数，以对按向上和向下的方向键（向前和向后移动）以及向左和向右的方向键（侧向旋转）作出响应。

#### 4.8.4 添加更多角色

`GLFrame` 类创建了一个很好的照相机类，而它对于角色来说也非常有用。我们经常会有很多对象分

散在各处，而我们要分别在空间中管理每个对象的位置和方向。GLFrame 类为这些位置和方向提供了一个非常好的容器。比如说 SphereWorld3，我们想要添加 50 个四处漂浮的随机球体。它终于可以真正成为“SphereWorld”（球体世界）了。我们将引导读者了解为了在场景中添加这些球体所做的改变，而不是列出一个长长的清单。首先，我们需要一个球体位置的列表。

```
#define NUM_SPHERES 50
GLFrame spheres[NUM_SPHERES];
```

请注意，我们并不需要 50 个实际的球体，只要将同一个球体绘制 50 次，每次都绘制在一个不同的位置。在 SetupRC 中，我们在场景中的随机位置对球体进行了初始化。

```
//随机放置球体
for(int i = 0; i < NUM_SPHERES; i++) {
    GLfloat x = ((GLfloat)(rand() % 400) - 200) * 0.1f;
    GLfloat z = ((GLfloat)(rand() % 400) - 200) * 0.1f;
    spheres[i].SetOrigin(x, 0.0f, z);
}
```

在 y 方向，我们将球体设在 0.0 的位置。这使它们看起来是漂浮在眼睛的高度。最后，在 RenderScene 函数中，这段简单的代码对所有的球体在正确的位置进行了渲染。

```
floorBatch.Draw();

for(int i = 0; i < NUM_SPHERES; i++) {
    modelViewMatrix.PushMatrix();
    modelViewMatrix.MultMatrix(spheres[i]);
    shaderManager.UseStockShader(GLT_SHADER_FLAT,
        transformPipeline.GetModelViewProjectionMatrix(),
        vSphereColor);

    sphereBatch.Draw();
    modelViewMatrix.PopMatrix();
}
```

将这段代码放置在绘制地板（前面已经展示过），但其实可以把它放在任何位置，只要是在将照相机变换应用在模型视图矩阵堆栈的顶端之后就可以。图 4.33 所示为 SphereWorld3 示例程序的输出。

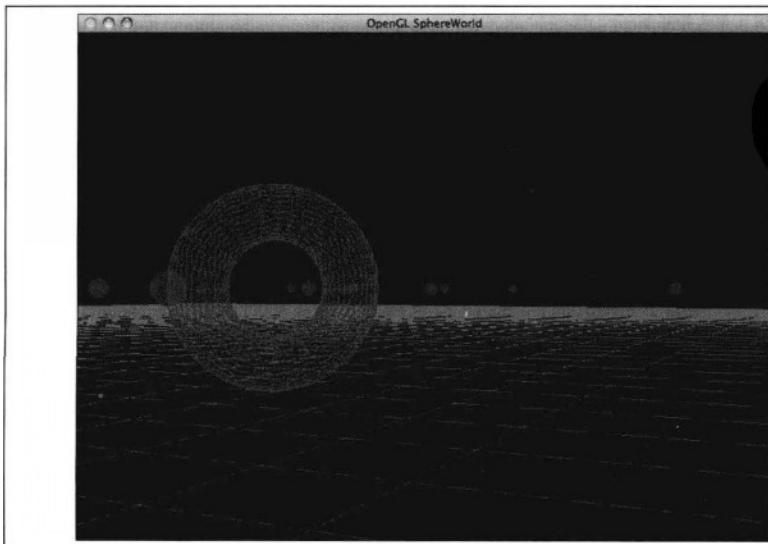


图 4.33

添加了照相机和球体的  
SphereWorld 示例程序

### 4.8.5 关于光线

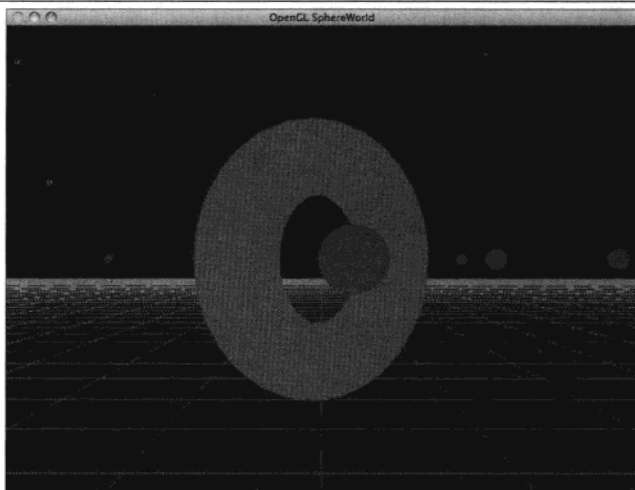
变换几何图形当然很不错，但是在本章结束前我们还要最后讨论一件事，就是变换光线。对于几何图形变换来说，典型情况下我们会设置变换矩阵，将它们传递到着色器，然后让硬件完成所有顶点的变换。对于光源来说，典型情况下我们的做法会有些不同。

光源位置也需要转换到视觉坐标系，但是传递到着色器的矩阵变换是几何图形，而不是光线。像点光源这样的固定光源不会移动——或者会移动？请记住关于照相机的比喻，整个世界实际上是相对于照相机移动的，其中也包括光源。

让我们再向 SphereWorld 进行一次增补。在 SphereWorld4 中，我们添加了一个单独的点光源。到目前为止，我们已经渲染了线框模式的 SphereWorld。图 4.34 所示为当取消调用 SetupRC 函数中的 `glPolygonMode` 时，SphereWorld 的外观。

图 4.34

未进行任何着色的 SphereWorld 示例程序



将一个固定光源位置变换到视觉坐标相对简单，而在每个场景中只需进行一次。下面介绍我们在 SphereWorld4 中是如何完成这项任务的。

```
//将光源位置变换到视觉坐标系
M3DVector4f vLightPos = { 0.0f, 10.0f, 5.0f, 1.0f };
M3DVector4f vLightEyePos;
m3dTransformVector4(vLightEyePos, vLightPos, mCamera);
```

光源位置的全局坐标存储在 `vLightPos` 变量中，其中包含了光源位置的  $x$  坐标、 $y$  坐标、 $z$  坐标和  $w$  坐标。我们必须保留  $w$  坐标（而且它必须为 1.0），因为无法用一个只有 3 个分量的向量去乘以一个  $4 \times 4$  矩阵。使用原来获得的照相机矩阵 `mCamera`，可以使用 `math3d` 库函数 `m3dTransformVector4` 对光源位置进行变换。我们在第 3 章已经介绍了存储着色器，现在终于有机会看一看点光源存储着色器的应用了。

例如，要渲染其中一个蓝色球体，使用正确的着色器并传递 `uniform` 值，如下所示。

```
shaderManager.UseStockShader(GLT_SHADER_POINT_LIGHT_DIFF,  
    transformPipeline.GetModelViewMatrix(),  
    transformPipeline.GetProjectionMatrix(),  
    vLightEyePos, vSphereColor);
```

很多光照射着色器还使用一个正规矩阵。经过一定操作，正规矩阵可以从模型视图矩阵推导出来，这个着色器就完成这项工作。不过这并不是最理想的方式，后面在第 6 章我们开始编写自己的着色器时，将更详细地讨论。就现在来说，这个简单的点着色器就够用了，我们可以在图 4.35 中看到 SphereWorld 示例程序在本章的最终版本。

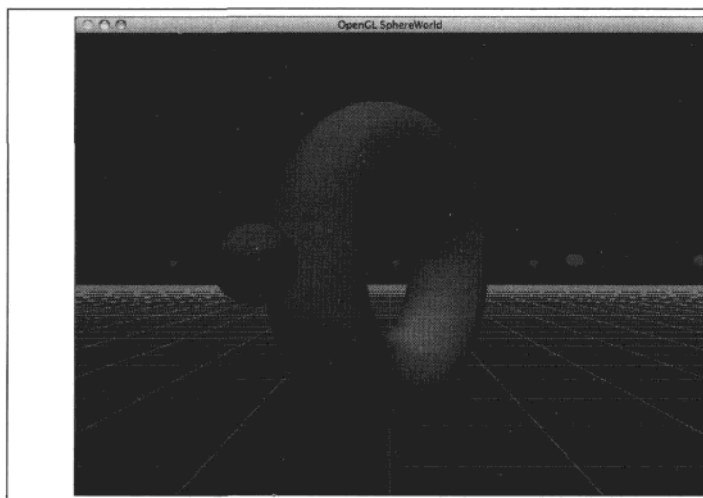


图 4.35

添加了点光源的 SphereWorld 示例程序

## 4.9 小结

在本章，我们学习了一些对于使用 OpenGL 建立 3D 场景来说非常重要的概念。即使我们还不能在脑海中把矩阵认识得很清楚，至少现在了解了矩阵是什么，以及如何用它们进行各种变换。我们还学习了如何操作模型视图矩阵和投影矩阵堆栈将对象放置到场景中，并确定它们在屏幕上看起来是什么样子。本章还介绍了功能强大的参考帧的概念，我们也看到了操纵帧并对它们进行变换是多么的容易。

最后，我们开始对贯穿本书的 GLTools 和 math3d 库进行更多的应用。这些库都是用可移植的 C++ 代码编写的，并为我们提供了一个方便的工具包，其中包括了可以和 OpenGL 一起使用的综合数学例程和帮助例程。意外的是，我们在这一整章都没有讲述一个新的 OpenGL 函数调用。

是的，这本来是一章数学内容，如果我们认为数学就是关于公式运算的话，那么我们甚至意识不到这一章是关于数学的。相关的向量、矩阵和应用程序对于使用进行 3D 对象和场景渲染来说是非常关键的。即使使用不同的 3D 数学库，或者甚至用自己的方式而不使用这些库，仍然会发现自己还是要沿用本章为了操纵几何图形和 3D 场景而制定的模式。好了，让我们继续前进，做一些有趣的事吧！

## 第5章 基础纹理

作者: Richard S. Wright, Jr.

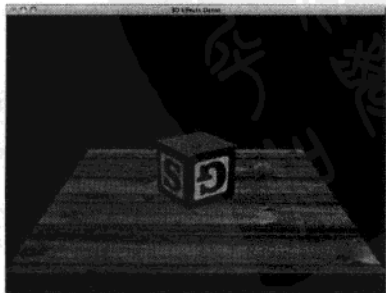
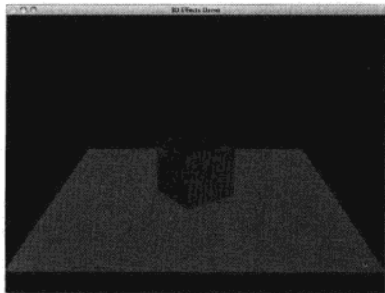
## 本章内容

任 务	使用的函数
载入纹理图像	glTexImage/glTexSubImage
设置纹理贴图参数	glTexParameter
管理多重纹理	glGenTextures/glDeleteTextures/glBindTexture
生成 Mip 贴图	glGenerateMipmap
使用各向异性过滤	glGetFloatv/glTexParameter
载入压缩纹理	glCompressedTexImage/glCompressedTexSubImage

到现在为止,我们已经对点、线和三角形进行了渲染,也看到了如何能够通过计算颜色值对它们的表面进行着色,以及在它们之间进行插值操作来模拟光照效果。这一切都非常不错,并且在大量的 3D 应用程序细分市场中,这些就已经是需要了解的全部内容了。但是,为了达到更加现实的效果,还有一种非常棒的捷径,这就是纹理贴图(texture mapping)。纹理只是一种能够应用到场景中的三角形上的图像数据,它通过经过过滤的纹理单元(texel,相当于基于纹理的像素)填充到实心区域。如图 5.1 所示,少量纹理文件就能为 3D 渲染增添令人兴奋的效果。

图 5.1

进行了纹理贴图和没有进行纹理贴图的几何图形形成了鲜明的对比



不过,正如我们在第7章即将看到的,纹理远远不止是图像数据那么简单,它是大多数现代3D渲染算法的一个关键因素。

## 5.1 原始图像数据

刚开始的时候,我们只有一些位图,并且那时它们已经……足够好了。最早的电子计算机显示器是单色的(即只有一种颜色),一般是绿色或者琥珀色,每个像素都只能是两种状态中的一种:开启或关闭。早期的计算机图形非常简单,图像数据都是由位图(一系列表示开启和关闭像素值的0和1)表示的。在位图中,每个内存块中的每个位都与屏幕上某一个像素的状态一一对应。图5.2所示是用位图表示的一匹马的图像。虽然只用了两种颜色(黑色和白色的点),但仍然能够清楚地表现出一匹马的外观。让我们将这幅图片和图5.3中显示的同一匹马的灰度图对比一下。在这个由像素组成的矩形【很多老前辈仍然称之为像素图(pixelmap)】中,每个像素都显示了256种不同深度的灰色中的一种。

术语“位图”(bitmap)经常用在饱含灰度或全彩色数据的图像中。在Windows平台上,位图这个术语的这种用法尤为常见,它总是和.BMP(bitmap)文件扩展名联系在一起,.BMP这种文件类型命名得很不好。很多人可能会对此进行争辩,严格地讲,这是对术语明显的误用。在本书中,我们不会将像素数据成为位图。

彩图2所示再次展示了这两幅图像,并且增加了一个全彩色的RGB版本。



图 5.2

一匹马真正的位图



图 5.3

一匹马的像素图(像素矩形)

### 5.1.1 像素包装

图像数据在内存中很少以紧密包装的形式存在。在许多硬件平台上,出于性能上的考虑,一幅图像的每一行都应该从一种特定的字节对齐地址开始。绝大多数编译器会自动把变量和缓冲区放置在一个针对该架构对齐优化的地址上。

在默认情况下,OpenGL 采用 4 个字节的对齐方式,这种方式适合于很多目前正在使用的系统。很多程序员会简单地将图像宽度值乘以高度值,再乘以每个像素的字节数,这样就错误地判断了存储一个图像所需的存储器数量。例如,如果我们有一幅 RGB 图像,包含 3 个分量(一个红色分量、一个绿色分量和一个蓝色分量),每个分量都存储在一个字节中(每个颜色通道 8 位,实际上这是非常典型的情况),那么,如果图像的宽度为 199 个像素,图像的每一行需要多少存储空间呢?读者可能会想,好吧,只要用 199 乘以 3(3 种颜色通道各需要一个字节),结果应该是图像数据的每一行需要 597 个字节。这也许是对的。但是,如果您是一位优秀的程序员,那么可能会非常讨厌这个数字!如果硬件本身的体系结构是 4 字节排列的(大部分是这样的),那么图像每一行的末尾都将有额外的 3 个空字节进行填充(这就使每一行都有了 600 字节),而这只是为了使每一行的存储器地址从一个能够被 4 整除的地址开始。

然而,很多时候这个问题都能够自己解决,尤其是在坚持二次幂纹理(稍后将做进一步讨论)时,但是我们还是应该多加小心,因为出现这样的小纰漏很可能在后面的工作中导致某些很难发现的与内存相关的奇怪 bug,这可能会让我们焦头烂额。虽然这看起来可能像是一种存储空间的浪费,但是这种排列能够让大多数 CPU 更高效地获取数据块。

许多未经压缩的图像文件格式也都遵循这种惯例。前面提到过的 Windows 中的 BMP 文件格式的像素数据使用 4 字节排列;然而 Targa(.TGA)文件格式则是 1 个字节排列的……这样不会浪费空间。为什么内存分配意图对于 OpenGL 来说这样重要?这是因为,在我们向 OpenGL 提交图像数据或从 OpenGL 获取图像数据时,OpenGL 需要知道我们想要在内存中对数据进行怎样的包装或解包装操作。

我们可以使用下列函数改变或者恢复像素的存储方式。

```
void glPixelStorei(GLenum pname, GLint param);
void glPixelStoref(GLenum pname, GLfloat param);
```

举例来说,如果我们想要改成紧密包装像素数据,应该像下面这样调用函数。

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

其中 GL\_UNPACK\_ALIGNMENT 指定 OpenGL 如何从数据缓冲区中解包图像数据。

类似地,我们可以使用 GL\_PACK\_ALIGNMENT 来告诉 OpenGL 如何将像素缓冲区中读取并放置到一个用户指定的内存缓冲区的数据进行包装。表 5.1 列出了这个函数支持的像素存储模式的完整列表,并且在附录 C 中对更多细节进行了讨论。

表 5.1

glPixelStore 参数

参 数 名	类 型	初 始 值
GL_PACK_SWAP_BYTES	GLboolean	GL_FALSE
GL_UNPACK_SWAP_BYTES	GLboolean	GL_FALSE
GL_PACK_LSB_FIRST	GLboolean	GL_FALSE
GL_UNPACK_LSB_FIRST	GLboolean	GL_FALSE
GL_PACK_ROW_LENGTH	GLint	0
GL_UNPACK_ROW_LENGTH	GLint	0
GL_PACK_SKIP_ROWS	GLint	0
GL_UNPACK_SKIP_ROWS	GLint	0
GL_PACK_SKIP_PIXELS	GLint	0
GL_UNPACK_SKIP_PIXELS	GLint	0
GL_PACK_ALIGNMENT	GLint	4
GL_UNPACK_ALIGNMENT	GLint	4
GL_PACK_IMAGE_HEIGHT	GLint	0
GL_UNPACK_IMAGE_HEIGHT	GLint	0
GL_PACK_SKIP_IMAGES	GLint	0
GL_UNPACK_SKIP_IMAGES	GLint	0

## 5.1.2 像素图

在当今全彩色计算机系统中，更加有趣并且更加实用一些的是像素图（pixmap）。像素图在内存布局上与位图非常相似，但是每个像素将需要一个以上的存储位来表示。每个像素的附加位允许存储强度（intensity，有时被称为亮度，即 luminance 值）或者颜色分量值。在 OpenGL 核心版本中，我们无法直接将一个像素图绘制到颜色缓冲区中，但是可以使用下面的函数将颜色缓冲区的内容作为像素图直接读取。

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

我们将 x 和 y 值指定为矩形左下角的窗口坐标，然后再指定矩形的 width 和 height 值（像素形式）。如果颜色缓冲区存储的数据与我们要求的不同，OpenGL 将负责进行必要的转换。

这种能力可能会非常有用。指向图像数据的指针 \*pixels 必须是合法的，并且必须包含足够的存储空间来存储转换后的图像数据，否则我们可能就会遇到严重的内存运行时异常。我们还要注意，如果指定的窗口坐标超出了允许范围，那么只能获得实际 OpenGL 帧缓冲区内像素的数据。

函数 glReadPixels 的第 4 个变量是 format，这个变量指定 pixels 指向的数据元素的颜色布局，并且可以采用表 5.2 中列出常量中的一个。

表 5.2

OpenGL 像素格式

常 量	描 述
GL_RGB	按照红、绿、蓝顺序排列的颜色
GL_RGBA	按照红、绿、蓝、Alpha 顺序排列的颜色
GL_BGR	按照蓝、绿、红顺序排列的颜色
GL_BGRA	按照蓝、绿、红、Alpha 顺序排列的颜色
GL_RED	每个像素只包含一个红色分量
GL_GREEN	每个像素只包含一个绿色分量
GL_BLUE	每个像素只包含一个蓝色分量
GL_RG	每个像素依次包含一个红色和一个绿色分量
GL_RED_INTEGER	每个像素包含一个整数形式的红色分量
GL_GREEN_INTEGER	每个像素包含一个整数形式的绿色分量
GL_BLUE_INTEGER	每个像素包含一个整数形式的蓝色分量
GL_RG_INTEGER	每个像素依次包含一个整数形式的红色和一个整数形式的绿色分量
GL_RGB_INTEGER	每个像素依次包含整数形式的红色、绿色和蓝色分量
GL_RGBA_INTEGER	每个像素依次包含整数形式的红色、绿色、蓝色和 Alpha 分量
GL_BGR_INTEGER	每个像素依次包含整数形式的蓝色、绿色和红色分量
GL_BGRA_INTEGER	每个像素依次包含整数形式的蓝色、绿色、红色和 Alpha 分量
GL_STENCIL_INDEX	每个像素只包含一个模板值
GL_DEPTH_COMPONENT	每个像素只包含一个深度值
GL_DEPTH_STENCIL	每个像素包含一个深度值和一个模板值

最后 3 个格式 GL\_STENCIL\_INDEX、GL\_DEPTH\_COMPONENT 和 GL\_DEPTH\_STENCIL 用于对模板缓冲区和深度缓冲区直接进行读写。参数 type 解释参数 \*pixels 指向的数据，它告诉 OpenGL 使用缓冲区中的什么数据类型来存储颜色分量。表 5.3 列出了可以使用的值。

表 5.3

像素数据的数据类型

常 量	描 述
GL_UNSIGNED_BYTE	每种颜色分量都是一个 8 位无符号整数
GL_BYTE	8 位有符号整数
GL_UNSIGNED_SHORT	16 位无符号整数
GL_SHORT	16 位有符号整数
GL_UNSIGNED_INT	32 位无符号整数
GL_INT	32 位有符号整数
GL_FLOAT	单精度浮点数
GL_HALF_FLOAT	半精度浮点数
GL_UNSIGNED_BYTE_3_2_2	包装的 RGB 值
GL_UNSIGNED_BYTE_2_3_3_REV	包装的 RGB 值
GL_UNSIGNED_SHORT_5_6_5	包装的 RGB 值
GL_UNSIGNED_SHORT_5_6_5_REV	包装的 RGB 值

续表

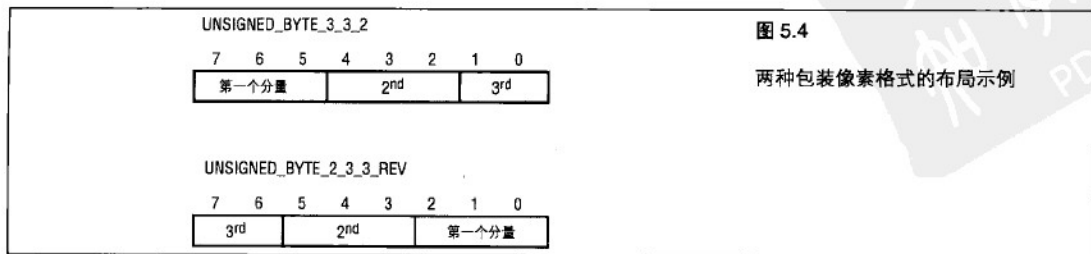
常 量	描 述
GL_UNSIGNED_SHORT_4_4_4_4	包装的 RGBA 值
GL_UNSIGNED_SHORT_4_4_4_4_REV	包装的 RGBA 值
GL_UNSIGNED_SHORT_5_5_5_1	包装的 RGBA 值
GL_UNSIGNED_SHORT_1_5_5_5_REV	包装的 RGBA 值
GL_UNSIGNED_INT_8_8_8_8	包装的 RGBA 值
GL_UNSIGNED_INT_8_8_8_8_REV	包装的 RGBA 值
GL_UNSIGNED_INT_10_10_10_2	包装的 RGBA 值
GL_UNSIGNED_INT_2_10_10_10_REV	包装的 RGBA 值
GL_UNSIGNED_INT_24_8	包装的 RGBA 值
GL_UNSIGNED_INT_10F_11F_11F_REV	包装的 RGBA 值
GL_FLOAT_32_UNSIGNED_INT_24_8_REV	包装的 RGBA 值

有必要指出, glReadPixels 从图形硬件中复制数据, 通常通过总线传输到系统内存。在这种情况下, 应用程序将被阻塞, 直到内存传输完成。此外, 如果我们指定一个与图形硬件的本地排列不同的像素布局, 那么在数据进行重定格式时将产生额外的性能开销。

### 5.1.3 包装的像素格式

表 5.3 中列出的包装格式在 OpenGL 1.2 ( 以及更新版本 ) 中是作为一种手段出现的, 为的是允许图形数据以更多的压缩形式进行存储, 以便与更广泛的颜色图形硬件相匹配。如果包装像素数据的种类更少, 那么显示硬件的设计就能够节省内存空间, 或者更快地进行操作。这些包装像素格式在某些 PC 硬件中仍然能够找到, 并且可能会在未来的硬件平台中继续发挥作用。

包装的像素格式将颜色数据压缩到了尽可能少的存储位中, 每个颜色通道的位数显示在常量中。例如, 格式为第一个分量提供 3 位存储空间, 为第二个分量也提供 3 位存储空间, 而为第 3 个分量则提供了两位存储空间。请记住, 指定的分量 ( 红、绿、蓝和 Alpha ) 的排列顺序仍然是根据 format 参数确定的。这些分量从高位 ( 最大位, 简称 MSB ) 到低位 ( 最小位, 简称 LSB ) 进行排列。GL\_UNSIGNED\_BYTE\_2\_3\_3\_REV 没有采用这种排序方式, 而是将最后的分量放置在了前两位, 诸如此类。图 5.4 所示以图形的方式展示了这两种排列方式的位布局。所有其他的包装格式都可以用同样的方式解释。



这些格式和数据类型参数也在大量其他图像和纹理相关函数中使用，后面我们会再次提到这些表格。默认情况下，对于 `glReadPixels` 函数来说，读取操作在双缓冲区渲染环境下将在后台缓冲区进行，而在单缓冲区渲染环境下则在前台缓冲区进行。我们可以用下面的函数改变这些像素操作的源。

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

模式参数可以取 `GL_FRONT`、`GL_BACK`、`GL_LEFT`、`GL_RIGHT`、`GL_FRONT_LEFT`、`GL_FRONT_RIGHT`、`GL_BACK_LEFT`、`GL_BACK_RIGHT` 或者甚至是 `GL_NONE` 中的任意一个。

### 5.1.4 保存像素

对于如何对像素数据进行一些有用的操作，我们已经讲解得不少了。`GLTools` 库中的 `gltWriteTGA` 函数从前台颜色缓冲区中读取颜色数据，并将这些数据存储到一个 Targa 文件格式的图像文件中。能够将当前的 OpenGL 渲染保存到一个标准图像文件格式中，这可能会非常有用。程序清单 5.1 列出了完整的 `gltWriteTGA` 函数。

程序清单 5.1 用 `gltWriteTGA` 函数来将屏幕图像保存为一个 Targa 文件

```
////////////////////////////////////
//捕获当前视口并将它保存为一个 targa 文件。
//确保在调用这个函数之前，在双缓冲区环境下调用 SwapBuffers，而在单缓冲区环境下调用 glFinish。
//如果出现错误则返回 0，如果没有出现错误则返回 1。
GLint gltWriteTGA(const char *szFileName)
{
    FILE *pFile;                // 文件指针
    TGAHEADER tgaHeader;        // TGA 文件头
    unsigned long lImageSize;    // 图像的大小，用字节表示
    GLbyte *pBits = NULL;       // 指向位的指针
    GLint iViewport[4];          // 以像素表示的视口
    GLenum lastBuffer;           // 存储当前的读取缓冲区设置
    // 获取视口大小
    glGetIntegerv(GL_VIEWPORT, iViewport);

    // 图像应该多大 (targa 文件将被紧密包装)
    lImageSize = iViewport[2] * 3 * iViewport[3];

    // 分配块。如果这种操作不起作用则返回
    pBits = (GLbyte *)malloc(lImageSize);
    if(pBits == NULL)
        return 0;

    // 从颜色缓冲区读取位
    glPixelStorei(GL_PACK_ALIGNMENT, 1);
    glPixelStorei(GL_PACK_ROW_LENGTH, 0);
    glPixelStorei(GL_PACK_SKIP_ROWS, 0);
    glPixelStorei(GL_PACK_SKIP_PIXELS, 0);

    // 获取当前读取缓冲区设置并进行保存。
    // 切换到前台缓冲区并进行读取操作。最后恢复读取缓冲区状态。
    glGetIntegerv(GL_READ_BUFFER, &lastBuffer);
    glReadBuffer(GL_FRONT);
    glReadPixels(0, 0, iViewport[2], iViewport[3], GL_BGR,
                 GL_UNSIGNED_BYTE, pBits);
    glReadBuffer(lastBuffer);
}
```

```
// 初始化 Targa 头
tgaHeader.identsize = 0;
tgaHeader.colorMapType = 0;
tgaHeader.imageType = 2;
tgaHeader.colorMapStart = 0;
tgaHeader.colorMapLength = 0;
tgaHeader.colorMapBits = 0;
tgaHeader.xstart = 0;
tgaHeader.ystart = 0;
tgaHeader.width = iViewport[2];
tgaHeader.height = iViewport[3];
tgaHeader.bits = 24;
tgaHeader.descriptor = 0;

// 为大小字节存储顺序问题而进行字节交换
#ifdef __APPLE__
    LITTLE_ENDIAN_WORD(&tgaHeader.colorMapStart);
    LITTLE_ENDIAN_WORD(&tgaHeader.colorMapLength);
    LITTLE_ENDIAN_WORD(&tgaHeader.xstart);
    LITTLE_ENDIAN_WORD(&tgaHeader.ystart);
    LITTLE_ENDIAN_WORD(&tgaHeader.width);
    LITTLE_ENDIAN_WORD(&tgaHeader.height);
#endif

// 尝试打开文件
pFile = fopen(szFileName, "wb");
if(pFile == NULL)
{
    free(pBits); // 释放缓冲区并返回错误
    return 0;
}

// 写入文件头
fwrite(&tgaHeader, sizeof(TGAHEADER), 1, pFile);

// 写入图像数据
fwrite(pBits, lImageSize, 1, pFile);

// 释放临时缓冲区并关闭文件
free(pBits);
fclose(pFile);

// 成功了!
return 1;
}
```

### 5.1.5 读取像素

Targa 图像格式是一种方便而且容易使用的图像格式,并且它既支持简单颜色图像,也支持带有 Alpha 值的图像。在本书中,我们会一直使用这种格式来进行纹理操作,现在我们先展示一下用于从磁盘中载入 Targa 文件的函数。

```
GLbyte *gltReadTGABits(const char *szFileName, GLint *iWidth, GLint *iHeight,
                       GLint *iComponents, GLenum *eFormat);
```

第一个参数是即将载入的 Targa 文件的文件名(如果有必要的话则会附加路径)。Targa 图像格式是

得到广泛支持的通用图像文件格式，它与 JPEG 文件不同，JPEG 文件（通常）以未经压缩的格式存储图像。glReadTGABits 函数用来打开文件，然后读入文件头并进行语法分析，以确定文件的宽度、高度和数据格式。分量的数量可以是一个、3 个或 4 个，分别为亮度、RGB 或 RGBA 图像。最终的参数是一个指向 GLenum 的指针，它接受图像相应的 GLenum 图像格式。如果函数调用成功，那么它就会返回一个新定位到直接从文件中读取的图像数据的指针（使用 malloc）。如果没有找到文件，或者出现其他错误，函数则会返回 NULL。程序清单 5.2 列出了完整的 glReadTGABits 函数。

程序清单 5.2 读取 Targa 文件以备 OpenGL 使用的函数

```

////////////////////////////////////
// 进行内存定位并载入 targa 位。返回指向新的缓冲区、纹理的高度和宽度，以及 OpenGL 数据格式。
// 结束时在缓冲区调用 free()
// 只支持 targa，只能是 8 位、24 位或 32 位色，没有调色板和 RLE 编码。
GLbyte *glReadTGABits(const char *szFileName, GLint *iWidth, GLint *iHeight,
                      GLint *iComponents, GLenum *eFormat)
{
    FILE          *pFile;           // 文件指针
    TGAHEADER      tgaHeader;        // TGA 文件头
    unsigned long  lImageSize;       // 图像的大小，用字节表示
    short          sDepth;           // 像素深度
    GLbyte         *pBits = NULL;    // 指向位的指针

    // 默认/失败值
    *iWidth = 0;
    *iHeight = 0;
    *eFormat = GL_RGB;
    *iComponents = GL_RGB;

    // 尝试打开文件
    pFile = fopen(szFileName, "rb");
    if(pFile == NULL)
        return NULL;

    // 读入文件头（二进制）
    fread(&tgaHeader, 18/* sizeof(TGAHEADER)*/, 1, pFile);

    // 为大小字节存储顺序问题而进行字节交换
#ifdef __APPLE__
    LITTLE_ENDIAN_WORD(&tgaHeader.colorMapStart);
    LITTLE_ENDIAN_WORD(&tgaHeader.colorMapLength);
    LITTLE_ENDIAN_WORD(&tgaHeader.xstart);
    LITTLE_ENDIAN_WORD(&tgaHeader.ystart);
    LITTLE_ENDIAN_WORD(&tgaHeader.width);
    LITTLE_ENDIAN_WORD(&tgaHeader.height);
#endif

    // 获取纹理的宽度、高度和深度
    *iWidth = tgaHeader.width;
    *iHeight = tgaHeader.height;
    sDepth = tgaHeader.bits / 8;

    // 这里进行一些有效性检验。非常简单，我们只要懂得或者说关心 8 位、24 位或 32 位 targa。
    if(tgaHeader.bits != 8 && tgaHeader.bits != 24 && tgaHeader.bits != 32)
        return NULL;

    // 计算图像缓冲区的大小
    lImageSize = tgaHeader.width * tgaHeader.height * sDepth;

    // 进行内存定位并进行成功检验。

```

```

pBits = (GLbyte*)malloc(lImageSize * sizeof(GLbyte));
if(pBits == NULL)
    return NULL;

// 读入位
// 检查读取错误。这项操作应该发现 RLE 或者其他我们不想识别的奇怪格式。
if(fread(pBits, lImageSize, 1, pFile) != 1)
{
    free(pBits);
    return NULL;
}

// 设置希望的 OpenGL 格式
switch(sDepth)
{
#ifdef OPENGLES
    case 3: // 最可能的情况
        *eFormat = GL_BGR;
        *iComponents = GL_RGB;
        break;
#endif
#ifdef WIN32
    case 3: // 最可能的情况
        *eFormat = GL_BGR;
        *iComponents = GL_RGB;
        break;
#endif
#ifdef linux
    case 3: // 最可能的情况
        *eFormat = GL_BGR;
        *iComponents = GL_RGB;
        break;
#endif
    case 4:
        *eFormat = GL_BGRA;
        *iComponents = GL_RGBA;
        break;
    case 1:
        *eFormat = GL_LUMINANCE;
        *iComponents = GL_LUMINANCE;
        break;
    default: // RGB
        // 如果实在 iPhone 上, TGA 为 BGR, 并且 iPhone 不支持没有 Alpha 的 BGR, 但是它支持 RGB, 所以只要
        // 将红色和蓝色调整一下就能满足要求。
        // 但是为了加快 iPhone 的载入速度, 请保存带有 Alpha 的 TGA。
#ifdef OPENGLES
        for(int i = 0; i < lImageSize; i+=3)
        {
            GLbyte temp = pBits[i];
            pBits[i] = pBits[i+2];
            pBits[i+2] = temp;
        }
#endif
        break;
}

// 文件操作完成
fclose(pFile);

// 返回指向图像数据的指针
return pBits;
}

```

读者可能已经发现,分量的数量并没有被设置为整数 1、3 或 4,而是设置为 GL\_LUMINANCE8、GL\_RGB8 和 GL\_RGBA8。OpenGL 识别这些特殊的常量是为了在操作图像数据时保持完整的内部精度。

例如,出于性能方面的原因,一些 OpenGL 实现可能会在内部对一个 24 位颜色进行向下取样而得到一个 16 位颜色。这种情况在某些显示输出精度只有 16 位的实现中载入更高位数深度的图像时尤为普遍。这些实现请求这些常量存储和使用图像数据,以完全支持它们每通道 8 位的颜色深度。

## 5.2 载入纹理

在几何图形中应用纹理贴图时,第一个必要步骤就是将纹理载入内存。一旦被载入,这些纹理就会成为当前纹理状态(稍候将进一步介绍相关内容)的一部分。有 3 个 OpenGL 函数最经常用来从存储器缓冲区中载入(比如说,从一个磁盘文件中读取)纹理数据。

```
void glTexImage1D(GLenum target, GLint level, GLint internalformat,
                  GLsizei width, GLint border,
                  GLenum format, GLenum type, void *data);

void glTexImage2D(GLenum target, GLint level, GLint internalformat,
                  GLsizei width, GLsizei height, GLint border,
                  GLenum format, GLenum type, void *data);

void glTexImage3D(GLenum target, GLint level, GLint internalformat,
                  GLsizei width, GLsizei height, GLsizei depth, GLint border,
                  GLenum format, GLenum type, void *data);
```

这 3 个函数确实非常冗长,它们通知 OpenGL 所需要知道的与如何解释数据参数指向的纹理数据有关的所有信息。

关于这些函数,我们应该知道的第一件事就是,它们实际上是由同一个函数 `glTexImage` 派生出来的。OpenGL 支持一维、二维和三维纹理贴图,并使用相应的函数来载入这些纹理并将它们设置为当前纹理。OpenGL 还支持立方图纹理,我们将把相关内容留到第 7 章讨论。我们还应该注意,OpenGL 会在调用这些函数中的一个时从 `data` 中复制纹理信息。这种数据复制可能会有很大的开销,稍后我们将讨论几种有助于减轻这个问题的方法。

这些函数中的 `target` 变量应分别为 `GL_TEXTURE_1D`、`GL_TEXTURE_2D` 或 `GL_TEXTURE_3D`。我们也可以指定代理纹理(Proxy Texture),方法是指定 `GL_PROXY_TEXTURE_1D`、`GL_PROXY_TEXTURE_2D` 或 `GL_PROXY_TEXTURE_3D`,并使用 `glGetTexParameter` 函数提取代理查询的结果。代理纹理和其他一些有趣的纹理对象将在第 7 章进行介绍。

`Level` 参数指定了这些函数所加载的 mip 贴图层次。我们将在稍后介绍 mip 贴图。因此就目前来说,对于非 mip 贴图的纹理(可以把它当作旧式的普通纹理贴图),我们总是可以把这个参数设置为 0。

接下来,我们必须指定纹理数据的 `internalformat` 参数。这个信息会告诉 OpenGL 我们希望在每个纹理单元中存储多少颜色成分,并在可能的情况下说明这些成分的存储大小,以及是否希望对纹理进行压缩。

表 5.4 列出了这个函数最为常用的一些值。完整的列表见附录 C。

表 5.4 最为常用的纹理内部格式

常 量	含 义
GL_ALPHA	按照 alpha 值存储纹理单元
GL_LUMINANCE	按照亮度值存储纹理单元
GL_LUMINANCE_ALPHA	按照亮度值和 alpha 值存储纹理单元
GL_RGB	按照红、绿、蓝成分存储纹理单元
GL_RGBA	按照红、绿、蓝和 alpha 成分存储纹理单元

width、height 和 depth 参数（在取值合适的时候）指定了被加载纹理的宽度、高度和深度。注意，这些值必须是 2 的整数次方（1、2、4、8、16、32、64 等），这一点非常重要。纹理贴图并不要求是立方体（3 个维度都相等），但是一个纹理在加载时如果使用了非 2 的整数次幂的值，在较老的 OpenGL 实现中，将会导致纹理贴图被隐式地禁用。尽管 OpenGL 2.0（以及更新的版本）允许使用非 2 的整数次幂的纹理，但无法保证它们在底层的硬件中能够实现足够的速度。出于这个原因，许多追求性能的开发人员仍然避免使用非 2 的整数次幂的纹理。

border 参数允许我们为纹理贴图指定一个边界宽度。纹理边界允许我们通过对边界处的纹理单元进行额外的设置，来对它的宽度、高度或深度进行扩展。在稍后将要进行的对纹理过滤的讨论中，纹理边界扮演了一个非常重要的角色。不过就目前来讲，我们可以把这个值设置为 0。

最后 3 个参数 format、type 和 data 和用于把图像数据放入颜色缓冲区的 glDrawPixels 函数的对应参数相同。为了方便起见，我们在表 5.2 和 5.3 中列出了 format 和 type 参数的合法常量值。

## 5.2.1 使用颜色缓冲区

一维和二维纹理也可以从颜色缓冲区加载数据。我们可以从颜色缓冲区读取一幅图像，并通过下面这两个函数将它作为一个新的纹理使用。

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

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

这两个函数的操作类似于 glTexImage，但在这里 x 和 y 在颜色缓冲区中指定了开始读取纹理数据的位置。源缓冲区是通过 glReadBuffer 函数设置的。请注意，并不存在 glCopyTexImage3D，因为我们无法从 2D 颜色缓冲区获取体积数据。

## 5.2.2 更新纹理

在时间敏感的场所如游戏或模拟应用程序中，重复加载新纹理可能会成为性能瓶颈。如果我们不再需要某个已加载的纹理，它可以被全部替换，也可以被替换掉一部分。替换一个纹理图像常常要比直接使用

glTexImage 重新加载一个新纹理快得多。用于完成这个任务的函数是 glTexSubImage, 它同样具有 3 个变型。

```
void glTexSubImage1D(GLenum target, GLint level,
                    GLint xOffset,
                    GLsizei width,
                    GLenum format, GLenum type, const GLvoid *data);

void glTexSubImage2D(GLenum target, GLint level,
                    GLint xOffset, GLint yOffset,
                    GLsizei width, GLsizei height,
                    GLenum format, GLenum type, const GLvoid *data);

void glTexSubImage3D(GLenum target, GLint level,
                    GLint xOffset, GLint yOffset, GLint zOffset,
                    GLsizei width, GLsizei height, GLsizei depth,
                    GLenum format, GLenum type, const GLvoid *data);
```

绝大部分参数都与 glTexImage 函数所使用的参数准确地对应。xOffset、yOffset 和 zOffset 参数指定了在原来的纹理贴图中开始替换纹理数据的偏移量。width、height 和 depth 参数指定了“插入”到原来那个纹理中的新纹理的宽度、高度和深度。

最后一组函数允许我们从颜色缓冲区读取纹理, 并插入或替换原来纹理的一部分。下面这些函数都是 glCopyTexSubImage 函数的变型, 它们都用于完成这个任务。

```
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);
```

读者可能已经注意到, 这里并没有列出 glCopyTexImage3D 函数。这是因为颜色缓冲区是 2D 的, 不存在一种对应方法来将一幅 2D 彩色图像作为一个 3D 纹理的来源。但是, 我们可以使用 glCopyTexSubImage3D 函数, 在一个三维纹理中使用颜色缓冲区的数据来设置它的一个纹理单元平面。

## 5.2.3 纹理对象

到目前为止, 我们已经看到了几种加载纹理的方式和一些替换纹理的方法。很多年前我们就见过只能支持一种纹理的硬件, 在这种情况下 OpenGL 就发展出一种管理多重纹理并在它们之间进行转换的方法。纹理图像本身就是所谓的纹理状态的一部分。纹理状态包含了纹理图像本身和一组纹理参数, 这些参数控制过滤和纹理坐标的行为。使用 glTexParameter 函数设置这些纹理状态参数的相关内容随后将进行讨论。不过, 首先让我们了解一下如何加载和管理几种不同的纹理。

像 glTexImage 和 glTexSubImage 这样的函数调用所耗费的内存特别多, 并且可能需要重新对这些

数据进行格式化以匹配一些内部表示方式。

在纹理之间进行切换或者重新加载不同的纹理图像可能会是开销很大的操作。纹理对象允许我们一次加载一个以上的纹理状态（包括纹理图像），以及在它们之间进行快速切换。纹理状态是由当前绑定的纹理对象维护的，而纹理对象是由一个无符号整数标识的。我们可以用下面这个函数分配一些纹理对象。

```
void glGenTextures(GLsizei n, GLuint *textures);
```

在这个函数中，我们可以指定纹理对象的数量和一个指针，这个指针指向一个无符号整型数组（由纹理对象标识符填充）。我们可以把它们看成是不同的可用纹理状态的句柄。为了“绑定”其中一种纹理状态，可以调用下面这个函数。

```
void glBindTexture(GLenum target, GLuint texture);
```

target 参数必须是 GL\_TEXTURE\_1D、GL\_TEXTURE\_2D 或 GL\_TEXTURE\_3D，而 texture 参数则是需要绑定的特定纹理对象。此后，所有的纹理加载和纹理参数设置只影响当前绑定的纹理对象。为了删除纹理对象，可以调用下面这个函数。

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

这个函数的参数和 glGenTextures 函数的参数具有相同的含义。我们并不需要同时产生和删除所有的纹理对象。多次调用 glGenTextures 所带来的额外开销很小。多次调用 glDeleteTextures 可能会造成一些延迟，但这种情况只有在销毁大量的纹理内存时才会发生。

我们可以使用下面这个函数对纹理对象名（或句柄）进行测试，以判断它们是否有效。

```
GLboolean glIsTexture(GLuint texture);
```

如果这个整数是一个以前已经分配的纹理对象名，那么这个函数就返回 GL\_TRUE，否则它就返回 GL\_FALSE。

## 5.3 纹理应用

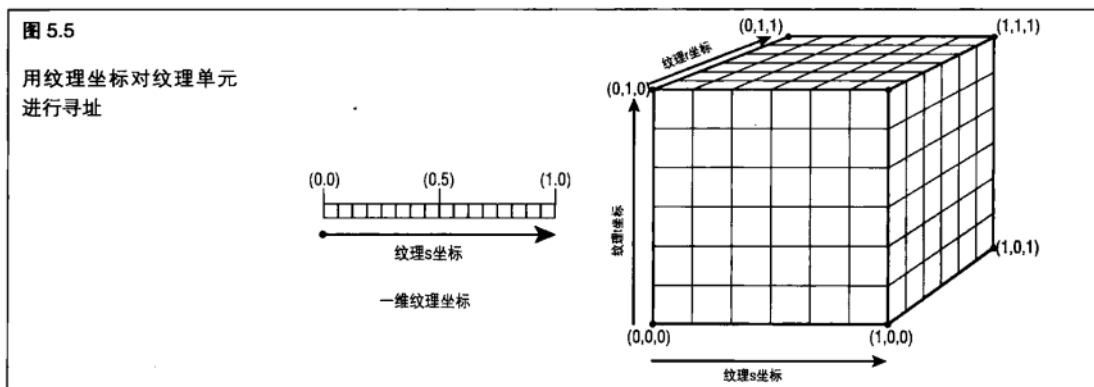
加载纹理只是在几何图形上应用纹理的第一步。最低限度我们必须同时提供纹理坐标，并设置纹理坐标环绕模式和纹理过滤。最后，我们可以选择对纹理进行 Mip 贴图，以提高纹理贴图性能和/或视觉质量。当然，在这整个过程中，我们都在假定着色器在做“正确的事情”。在本章中，我们坚持使用 2D 纹理示例并使用存储着色器。在下一章中，当我们开始编写着色器时，就会看到如何从着色器的层次上应用纹理了。不过就目前来说，我们只要关注客户端的纹理贴图技术就可以了。

### 5.3.1 纹理坐标

总体上讲，我们是通过为每个顶点指定一个纹理坐标而直接在几何图形上进行纹理贴图的。纹理坐标要么是指定为着色器的一个属性，要么通过算法计算出来。纹理贴图纹理单元是作为一个更加抽象（经

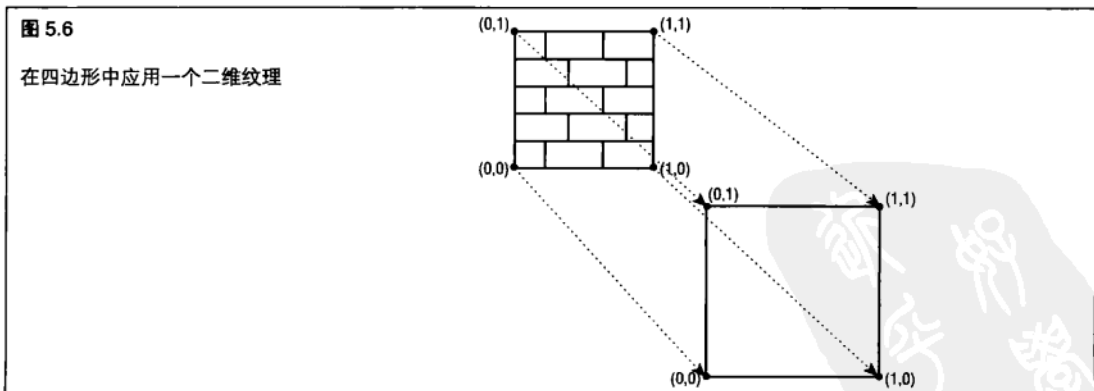
常是浮点值)的纹理坐标,而不是作为内存位置(在像素图中则是这样)进行寻址的。典型情况下,纹理坐标是作为 0.0 到 1.0 范围内的浮点值指定的。纹理坐标命名为  $s$ 、 $t$ 、 $r$  和  $q$  (与顶点坐标  $x$ 、 $y$ 、 $z$  和  $w$  相类似),支持从一维到三维的纹理坐标,并且可以选择一种对坐标进行缩放的方法。

图 5.5 所示显示了一维、二维和三维纹理,以及根据它们的纹理单元排列纹理坐标的方式。



因为不存在四分量的纹理,所以读者可能会奇怪, $q$  坐标是做什么用的。 $q$  坐标对应几何坐标  $w$ 。这是一个缩放因子,作用于其他纹理坐标。也就是说,我们实际所使用的纹理坐标是  $s/q$ 、 $t/q$  和  $r/q$ 。在默认情况下, $q$  设置为 1.0。虽然这看起来似乎有些随意,但对于阴影贴图 (shadow mapping) 之类一些高级纹理坐标生成算法来说确实非常有用。

一个纹理坐标会在每个顶点上应用一个纹理(没错,确实有一种同时应用一个以上纹理的方法)。然后,OpenGL 根据需要对纹理进行放大或缩小,将纹理贴图到几何图形上。(放大或缩小是使用当前的纹理过滤器实现的,我们将在稍后讨论这个话题)。图 5.6 所示显示了一个例子,一个二维纹理被贴图到几何图形上的一个正方形(可能是三角形扇)上。注意这个纹理的各个角对应于这个四边形的各个角。



然而,我们很少能够碰到上面这种情况,也就是一个四边形纹理能够严丝合缝地贴图到一个正方体的几何图形上。为了帮助读者更好地理解纹理坐标,我们提供了另一个例子,如图 5.7 所示。这幅图也显示了一个正方形的纹理图像,但现在这个几何图形是个三角形。叠加在这个纹理贴图上的扩展到这个三角形各个顶点在贴图位置上位置的纹理坐标。

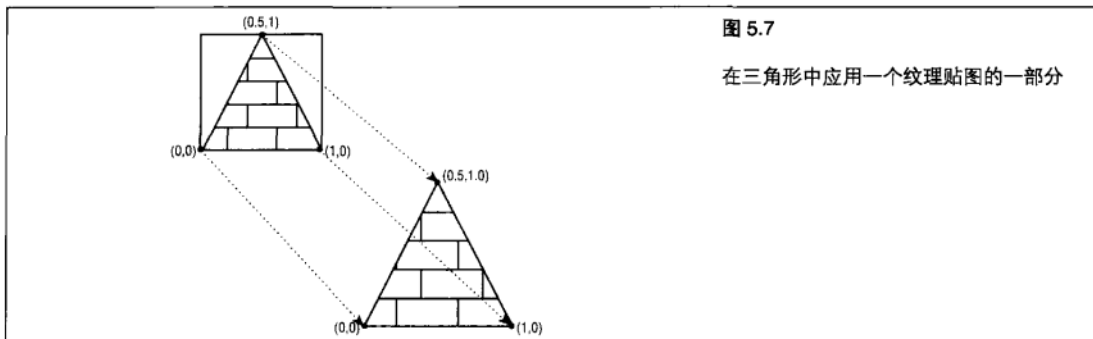


图 5.7

在三角形中应用一个纹理贴图的一部分

## 5.3.2 纹理参数

和将一幅图片贴在三角形的一面相比，纹理贴图需要更多的工作。很多参数的应用都会影响渲染的规则和纹理贴图的行为。这些纹理参数都是通过 `glTexParameter` 函数的变量来进行设置的。

```
void glTexParameterf(GLenum target, GLenum pname, GLfloat param);
void glTexParameteri(GLenum target, GLenum pname, GLint param);
void glTexParameterfv(GLenum target, GLenum pname, GLfloat *params);
void glTexParameteriv(GLenum target, GLenum pname, GLint *params);
```

第一个参数 `target` 指定这些参数将要应用在哪个纹理模式上，它可以是 `GL_TEXTURE_1D`、`GL_TEXTURE_2D` 或 `GL_TEXTURE_3D`（关于后两个值将在第 7 章做进一步讨论）。第二个参数 `pname` 指定了需要设置哪个纹理参数，而最后一个参数 `param` 或 `params` 用于设置特定的纹理参数的值。

### 基本过滤

纹理图像中的纹理单元和屏幕上的像素几乎从来不会形成一对一对应关系。如果程序员足够细心，确实可以实现这个效果，但这需要在对几何图形进行纹理贴图时进行精心的计划，使出现在屏幕上的纹理单元和像素能够对齐。（实际上在用 OpenGL 进行图像处理应用时经常做这项工作）。因此，当纹理应用于几何图形的表面时，纹理图像不是被拉伸就是被收缩。根据几何图形的方向，一个特定的纹理甚至可能会在贴到一些物体表面的同时就开始进行拉伸和收缩。

根据一个拉伸或收缩的纹理贴图计算颜色片段的过程称为纹理过滤（Texture Filtering）。使用 OpenGL 的纹理参数函数，可以同时设置放大和缩小过滤器。这两种过滤器的参数名分别是 `GL_TEXTURE_MAG_FILTER` 和 `GL_TEXTURE_MIN_FILTER`。就目前来说，我们可以为它们从两种基本的纹理过滤器 `GL_NEAREST` 和 `GL_LINEAR` 中进行选择，它们分别对应于最邻近过滤和线性过滤。确保总是为 `GL_TEXTURE_MIN_FILTER` 选择这两种过滤器中的一种，因为默认的过滤器不适用于 Mip 贴图（参见“Mip 贴图”一节）。

最邻近过滤是我们能够选择的最简单、最快速的过滤方法。

纹理坐标总是根据纹理图像的纹理单元进行求值和绘图的。不管纹理坐标位于哪个纹理单元，这个纹理单元的颜色就作为这个片段的纹理颜色。最邻近过滤最显著的特征就是当纹理被拉伸到特别大时所出现

的大片斑驳状像素。图 5.8 所示显示了最邻近过滤的一个例子。我们可以使用下面这两个函数，为放大和缩小过滤器设置纹理过滤器（用于 GL\_TEXTURE\_2D）。

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

和最邻近过滤相比，线性过滤还需要更多的工作，但它所实现的效果往往值得付出这些额外的开销。在当今的高速硬件上，线性过滤所带来的额外开销几乎可以忽略不计。

线性过滤并不是把最邻近的纹理单元应用到纹理坐标中，而是把这个纹理坐标周围的纹理单元的加权平均值应用到这个纹理坐标上（线性插值）。为了让这个插值的片段与纹理单元的颜色准确匹配，纹理坐标需要准确地落在纹理单元的中心。线性过滤最显著的特征就是当纹理被拉伸时所出现的“失真”图形。但是，和最邻近过滤模式下所呈现的斑驳状像素块相比，这种“失真”更接近真实，没有那种人工操作的痕迹。图 5.9 所示显示了一个与图 5.8 形成对照的例子。我们可以使用下面这几行代码，简单地设置线性过滤（用于 GL\_TEXTURE\_2D）。

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

图 5.8

近距离观察最邻近过滤

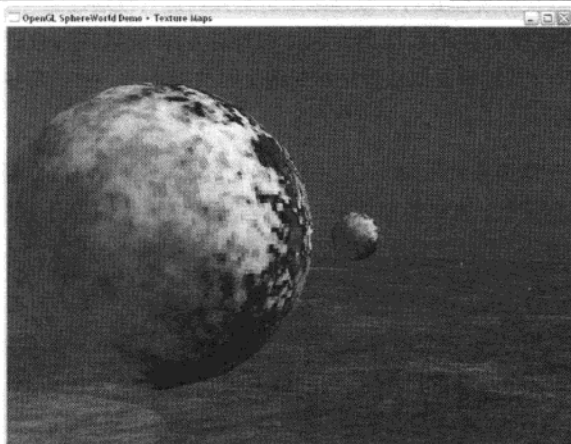


图 5.9

近距离观察线性过滤（彩图 3 并排显示了最邻近过滤和线性过滤）



## 纹理环绕

在正常情况下，我们在 0.0 到 1.0 的范围之内指定纹理坐标，使它与纹理贴图上的纹理单元形成映射关系。如果纹理坐标落在这个范围之外，OpenGL 则根据当前纹理环绕模式（Wrapping Mode）处理这个问题。我们可以调用 `glTexParameterf` 函数（并分别使用 `GL_TEXTURE_WRAP_S`、`GL_TEXTURE_WRAP_T` 或 `GL_TEXTURE_WRAP_R` 作为参数），为每个坐标分别设置环绕模式。然后，我们可以把环绕模式设置为下面几个值之一：`GL_REPEAT`、`GL_CLAMP`、`GL_CLAMP_TO_EDGE` 或 `GL_CLAMP_TO_BORDER`。

在 `GL_REPEAT` 环绕模式下，OpenGL 在纹理坐标值超过 1.0 的方向上对纹理进行重复。这种纹理重复对每个整型纹理坐标都适用。如果我们需要把一个小型的平铺式纹理应用到大型几何图形的表面，这种模式就会非常有用。设计良好的无缝纹理可以导致大型纹理看上去也是无缝的，但是这个效果所付出的代价是它需要小得多的纹理图像。其他模式并不进行重复，而是像它们的名字所提示的那样，进行“截取”。

如果环绕模式仅有的意义就在于是否对纹理进行重复，那么只需要两种环绕模式就够了：重复和截取。但是，纹理环绕模式对于纹理贴图边缘如何进行纹理过滤有着非常大的影响。在 `GL_NEAREST` 过滤模式中，环绕模式并不起作用，因为纹理坐标总是对齐到纹理贴图中一些特定的纹理单元。但是，`GL_LINEAR` 过滤则需要取纹理坐标周围像素的平均值，对于那些位于纹理贴图边缘的纹理单元，这样就会出现一些问题。

如果纹理环绕模式设置为 `GL_REPEAT`，这个问题就可以非常简单地得到处理。纹理采样简单地从接下来的行或列提取。在重复模式中，这相当于环绕到纹理的另一边。对于沿物体环绕并与另一边吻合的纹理（例如球体），这种模式是相当完美的。

截取型的纹理环绕模式提供了一些选项来处理纹理边缘。对于 `GL_CLAMP`，所需的纹理单元取自纹理边界或 `TEXTURE_BORDER_COLOR`（用 `glTexParameterfv` 函数进行设置）。`GL_CLAMP_TO_EDGE` 环绕模式强制对范围之外的纹理坐标沿着合法的纹理单元的最后一行或者最后一列进行采样。最后，`GL_CLAMP_TO_BORDER` 环绕模式在纹理坐标在 0.0 到 1.0 的范围之外时只使用边界纹理单元。边界纹理单元是作为围绕基本图像的额外的行和列，并与基本纹理图像一起加载的。

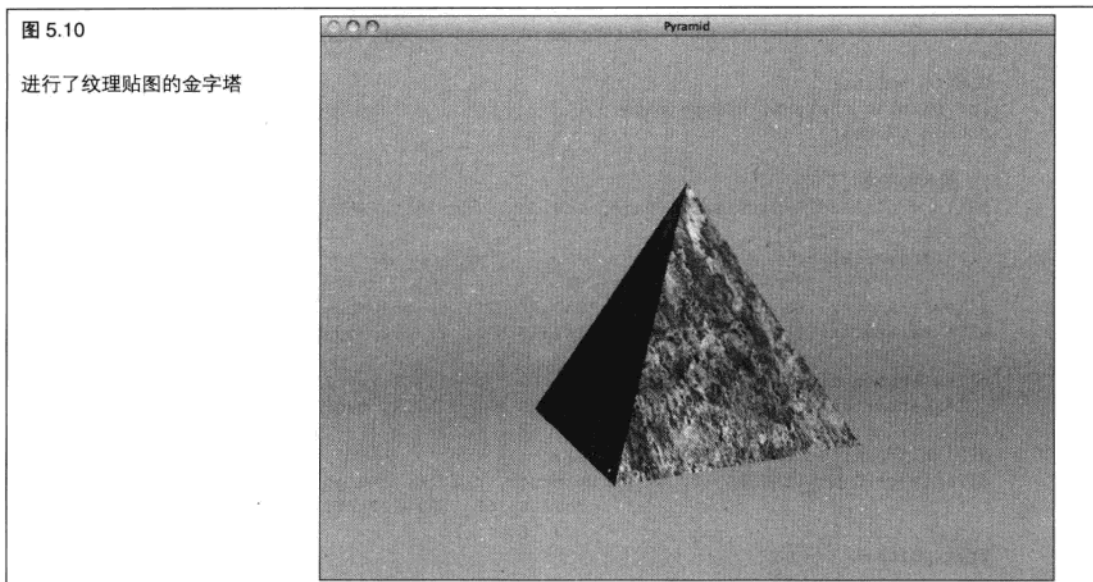
截取模式的一个典型应用就是在必须对一块大型区域进行纹理处理时。此时如果使用单个纹理，它将会由于过于庞大而无法装入到内存中，或者它可能会加载到单个纹理图像中。在这种情况下，这块区域被切割成更小的平铺在一起的“瓷砖”。此时，如果不使用像 `GL_CLAMP_TO_EDGE` 这样的环绕模式，就会导致各块“瓷砖”之间存在明显的缝隙痕迹。

在极少的情况下，如果这种做法仍然不能满足要求，可以求助于纹理边界纹理单元。

### 5.3.3 综合运用

我们已经了解了很多纹理贴图的特性和需要，但还没有接触具体的示例程序。下面来看一个完整的示例程序 Pyramid（金字塔），这个程序绘制了一个金字塔并在上面应用了类似图 5.6 和图 5.7 所示的纹理。

图 5.10 所示显示了本章中第一个示例程序的输出结果。



### 加载纹理

我们要做的第一步工作就是加载纹理 stone.tga。我们通过 SetupRC 函数完成这项工作，如下所示。

```
glGenTextures(1, &textureID);
glBindTexture(GL_TEXTURE_2D, textureID);
LoadTGATexture("stone.tga", GL_LINEAR, GL_LINEAR, GL_CLAMP_TO_EDGE);
```

在原文件（ pyramid.cpp ）的顶部声明变量 textureID，如下所示。

```
GLuint textureID;
```

函数 glGenTextures 将分配一个纹理对象，并将它放置在这个变量中。我们使用 textureID 值来识别单个纹理，并调用 glBindTexture 对这个纹理状态进行初始绑定。函数 glGenTextures 只保留一个纹理对象 ID，这的确是一个轻量级的函数。新的纹理状态实际上在我们第一次调用 glBindTexture 之前都不会进行创建和初始化。与此相对应，我们已经使用过了在程序结束时用于删除纹理对象的 ShutdownRC 函数。

```
glDeleteTextures(1, &textureID);
```

实际加载纹理图像和设置纹理状态是通过函数 LoadTGATexture 完成的，其原型如下所示。

```
bool LoadTGATexture(const char *szFileName, GLenum minFilter, GLenum magFilter,
                    GLenum wrapMode);
```

这个函数接受图像文件的文件名，需要的缩小和放大过滤器，以及纹理坐标环绕模式为参数。它将完整地设置纹理状态，并且因为它被放置在调用 glBindTexture 之后，所以就成为了由 textureID 标识的纹理对象的一部分。程序清单 5.3 完整地展示了 LoadTGATexture 函数。虽然这个函数不是 GLTools 的一部分，我们还是会在几个示例程序中使用到它。

## 程序清单 5.3 完整的纹理加载函数

// 加载一个 TGA 作为 2D 纹理 对状态进行完全初始化

```
bool LoadTGATexture(const char *szFileName, GLenum minFilter,
                    GLenum magFilter, GLenum wrapMode)
{
    GLbyte *pBits;
    int nWidth, nHeight, nComponents;
    GLenum eFormat;

    // 读入纹理位
    pBits = gltReadTGABits(szFileName, &nWidth, &nHeight, &nComponents, &eFormat);
    if(pBits == NULL)
        return false;

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, wrapMode);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, wrapMode);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, minFilter);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, magFilter);

    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    glTexImage2D(GL_TEXTURE_2D, 0, nComponents, nWidth, nHeight, 0,
                eFormat, GL_UNSIGNED_BYTE, pBits);

    free(pBits);

    if(minFilter == GL_LINEAR_MIPMAP_LINEAR ||
       minFilter == GL_LINEAR_MIPMAP_NEAREST ||
       minFilter == GL_NEAREST_MIPMAP_LINEAR ||
       minFilter == GL_NEAREST_MIPMAP_NEAREST)
        glGenerateMipmap(GL_TEXTURE_2D);

    return true;
}
```

还有一件事我们没有讨论，这就是调用 `glGenerateMipmap` 和基于 Mip 贴图的过滤器。Mip 贴图将在后面的内容中讲述，在实际使用这个特性之前我们暂时不管它。

### 指定纹理坐标

在 `SetupRC` 中加载纹理之后，我们调用函数 `MakePyramid`，并传递 `GLBatch` 的一个叫做 `pyramidBatch` 的实例。

```
MakePyramid(pyramidBatch);
```

这个函数手动构建了一个由独立三角形组成的金字塔，并将它们放置到 `pyramidBatch` 容器类中。我们在这里并不打算列出这个函数的完整代码，但是对于这里发生的一些有趣的事情，我们还是需要说明一下。前面使用 `GLBatch` 类时，我们用 `CopyVertexData` 函数一次性将整整一个数组的数据复制到了批次中。`GLBatch` 类还包含一些函数，允许我们每次一个顶点建立一个批次。这看起来有点像兼容版本中那种古老的、现在已经不推荐的立即模式（`immediate mode`），这肯定会引起一些争议。毫无疑问，真正的立即模式是组建一个顶点批次最慢的方式，但是它可以是方便的，并且可以使手动构建几何图形得到简化。`GLBatch` 类的这个特性并不是模仿真正的立即模式，因此，如果读者已经对立即模式非常熟悉的话，就要忘掉大部分所知的内容，因为我们的应用已经经过了简化。

下面让我们了解一下如何开始组建三角形批次。

```
pyramidBatch.Begin(GL_TRIANGLES, 18, 1);
```

这样就开始了批次，就像我们开始任何一个 GLBatch 一样。请注意这里的最后一个参数是 1。这就意味着在这个批次中将应用一个纹理。我们使用默认参数的 C++ 特性，而如果保持这个参数关闭状态的话，那么它将自动设置为 0。实际上我们可以一次性应用多个纹理，在第 7 章我们将讨论如何进行这项工作。

现在让我们来看一下，如何将三角形的前两个顶点添加到金字塔的底部。

```
//金字塔的底部
pyramidBatch.Normal3f(0.0f, -1.0f, 0.0f);
pyramidBatch.MultiTexCoord2f(0, 0.0f, 0.0f);
pyramidBatch.Vertex3f(-1.0f, -1.0f, -1.0f);
pyramidBatch.Normal3f(0.0f, -1.0f, 0.0f);
pyramidBatch.MultiTexCoord2f(0, 1.0f, 0.0f);
pyramidBatch.Vertex3f(1.0f, -1.0f, -1.0f);
```

Normal3f 方法向批次中添加了一个表面法线。MultiTexCoord2f 添加了一个纹理坐标，最后，Vertex3f 添加了顶点的位置。对于不应用到老式立即模式的 GLBatch 来说，有一个重要的原则是，如果我们为任何顶点指定了法线或纹理坐标，那么就必须为每个顶点进行同样的指定。这样就损失了老方法的一些灵活性，但是确实让它运行得快了一些。Normal3f 和 Vertex3f 函数都是不言自明的，但是 MultiTexCoord2f 则有 3 个参数，并且第一个参数是一个整数。

```
void GLBatch::MultiTexCoord2f(GLuint texture, GLclampf s, GLclampf t);
```

在这里，除了纹理坐标之外，我们通过 texture 指定纹理层次。在学习第 7 章的多重纹理相关内容之前，对于存储着色器来说，总要把它设置为 0。

对于数学推导和手动建模的几何图形来说，这种设置顶点数据的方式可能是非常方便的，并且可以精简代码，这里我们将展示如何为金字塔的一个面来计算表面法线，然后再在所有 3 个顶点上使用它。

```
//金字塔的前面
m3dFindNormal(n, vApex, vFrontLeft, vFrontRight);
pyramidBatch.Normal3fv(n);
pyramidBatch.MultiTexCoord2f(0, 0.5f, 1.0f);
pyramidBatch.Vertex3fv(vApex); // 顶部

pyramidBatch.Normal3fv(n);
pyramidBatch.MultiTexCoord2f(0, 0.0f, 0.0f);
pyramidBatch.Vertex3fv(vFrontLeft); // 左前角

pyramidBatch.Normal3fv(n);
pyramidBatch.MultiTexCoord2f(0, 1.0f, 0.0f);
pyramidBatch.Vertex3fv(vFrontRight); // 右前角
```

表面法线是有方向的向量，它代表表面（或者顶点）面对的方向。这在大多数光照模式下都是必须的。我们将在下一章通过一个实例着色器来讨论这部分内容。

请记住，以每次一个元素的方式复制大量数据，无异于用每次一茶叶水的方式填满一个游泳池。在性能敏感的环境中，我们不应该采用这种方式。通常启动开销可以忽略不计，这就好办了。不过，如果几何

图形是动态的，而我们又经常要改变它们，那么这就可能是移动大量几何图形数据的最差方式了。

最后，我们来看一看如何在示例程序中实际渲染金字塔。请注意我们必须再一次绑定纹理对象 textureID。

```
glBindTexture(GL_TEXTURE_2D, textureID);
shaderManager.UseStockShader(GLT_SHADER_TEXTURE_POINT_LIGHT_DIFF,
    transformPipeline.GetModelViewMatrix(),
    transformPipeline.GetProjectionMatrix(),
    vLightPos, vWhite, 0);

pyramidBatch.Draw();
```

严格地讲，对纹理进行绑定并不是必须的，因为在项目中只有一个纹理，而在加载纹理时已经对它进行了绑定。然而这样的情况并不多见，所以我们要注意，在提交一个几何图形批次的时候需要绑定到我们想要使用的纹理。实际上绑定也可以放置在着色器之后，只要它在几何图形提交之前进行绑定，那么这个纹理就会是使用过的那个。

对于本例来说，我们也使用一个新的存储着色器 GLT\_SHADER\_TEXTURE\_POINT\_LIGHT\_DIFF。这个着色器将在我们的场景中设置一个点光源，使用指定的颜色来对几何图形进行着色，在本例中为 vWhite，然后将它乘以纹理颜色。结果得到经过着色和纹理贴图的金字塔，如图 5.10 所示。

## 5.4 Mip 贴图

Mip 贴图是一种功能强大的纹理技巧，它不仅可以提高渲染性能，而且可以改善场景的显示质量。它使用标准纹理贴图处理两个常见的问题，从而实现上述目标。第一个问题是一种称为闪烁（Scintillation，即锯齿假影）的效果。当屏幕上被渲染物体的表面与它所应用的纹理图像相比显得非常小时，就会出现这种效果。闪烁可以被看成是某种类型的闪光，当纹理图像的采样区域的移动幅度与它在屏幕上的大小相比显得不成比例时，就会发生这种现象。当照相机或物体处于运动状态时，我们很容易看到闪烁的负面效果。

第二个问题更多地与性能有关，但它的原因和闪烁相同。也就是说，问题的根源在于它必须加载大量的纹理内存并对它们进行过滤处理，但屏幕上实际显示的只是很少的一部分片段。纹理越大，这个问题所造成的性能影响也就越为明显。

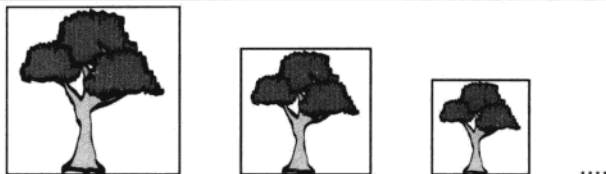
当然，我们可以用一种非常简单的方法解决这两个问题，就是使用更小的纹理图像。但是，这种解决方法又产生了一个新问题，就是当一个物体更靠近观察者时，它必须渲染得比原来更大一些。这样，那个较小的纹理图像不得不进行拉伸，结果形成了视觉效果很差的模糊或斑驳状的纹理化物体。

这两个问题的解决方案就是使用 Mip 贴图。Mip 贴图取自拉丁文短语“multum in parvo”，意思是“一个小地方有许多东西”。从本质上说，我们不是把单个图像加载到纹理状态中，而是把一系列从最大到最小的图像加载到单个“Mip 贴图”纹理状态。然后，OpenGL 使用一组新的过滤模式，为一个特定的几何图形选择具有最佳过滤效果的纹理。在付出一些额外内存的代价之后（可能还有一些额外的处理任务），不仅可以消除闪烁现象，而且可以大大降低对远处物体进行纹理贴图时所需要的内存及处理开销。同时，在需要的时候，它还可以维护一组具有更高分辨率的可用纹理。

Mip 贴图纹理由一系列纹理图像组成, 每个图像大小在每个轴的方向上都缩小一半, 或者说是原来图像像素总数的四分之一。图 5.11 所示显示了这些场景。Mip 贴图层并不一定是正方形的, 但每个图像的大小都依次减半, 直到最后一个图像的大小是  $1 \times 1$  的纹理单元为止。当其中一个维度的大小到达 1 时, 接下来的减半处理就只发生在其他维度上了。使用一组正方形 (即各个维度的大小相等) 的 Mip 贴图所要求的内存比不使用 Mip 贴图要多出三分之一。

图 5.11

一系列 Mip 贴图图像



Mip 贴图层是通过 `glTexImage` 函数加载的。现在轮到 `level` 参数发挥它的作用了, 因为它指定了图像数据用于哪个 Mip 层。第一层是 0, 接着是 1、2, 然后依此类推。如果 Mip 贴图未被使用, 那么就只有第 0 层会被加载。在默认情况下, 为了使用 Mip 贴图, 所有的 Mip 层都必须加载。但是, 我们可以用 `GL_TEXTURE_BASE_LEVEL` 和 `GL_TEXTURE_MAX_LEVEL` 纹理参数特别设置需要使用的基层和最大层。例如, 如果想指定只加载从第 0 层至第 4 层, 可以像下面这样调用 `glTexParameter` 函数两次。

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL, 0);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, 4);
```

尽管 `GL_TEXTURE_BASE_LEVEL` 和 `GL_TEXTURE_MAX_LEVEL` 控制哪些 Mip 层将被加载, 但我们仍然可以使用 `GL_TEXTURE_MIN_LOD` 和 `GL_TEXTURE_MAX_LOD` 参数限制已加载的 Mip 层的使用范围。

### 5.4.1 Mip 贴图过滤

Mip 贴图在两个基本的纹理过滤模式 `GL_NEAREST` 和 `GL_LINEAR` 上添加了一个新的变化, 这是通过向 Mip 贴图过滤模式提供 4 种不同变化实现的。表 5.5 列出了这些变化。

表 5.5

经过 Mip 贴图的纹理过滤

常 量	描 述
<code>GL_NEAREST</code>	在 Mip 基层上执行最邻近过滤
<code>GL_LINEAR</code>	在 Mip 基层上执行线性过滤
<code>GL_NEAREST_MIPMAP_NEAREST</code>	选择最邻近 Mip 层, 并执行最邻近过滤
<code>GL_NEAREST_MIPMAP_LINEAR</code>	在 Mip 层之间执行线性插补, 并执行最邻近过滤
<code>GL_LINEAR_MIPMAP_NEAREST</code>	选择最邻近 Mip 层, 并执行线性过滤
<code>GL_LINEAR_MIPMAP_LINEAR</code>	在 Mip 层之间执行线性插补, 并执行线性过滤, 又称三线性 Mip 贴图

仅仅使用 `glTexImage` 函数加载 Mip 层并不能启用 Mip 贴图功能。如果纹理过滤设置为 `GL_LINEAR` 或 `GL_NEAREST`, 那么就只有纹理贴图基层会被使用, 其他所有加载的 Mip 层都将被忽略。我们必须指定其中一个 Mip 贴图过滤器, 这样才能使用所有已加载的 Mip 层。这个常量具有

GL\_FILTER\_MIPMAP\_SELECTOR 的形式，其中 FILTER 指定了被选择的 Mip 层将要使用的纹理过滤器，SELECTOR 则指定了如何选择 Mip 层。例如，GL\_NEAREST 选择最接近匹配的 Mip 层。

如果选择 GL\_LINEAR，它就会在两个最邻近的 Mip 层之间执行线性插值，其结果又由被选择的纹理过滤器进行过滤。如果选择了其中一种 Mip 贴图过滤模式，但不加载 Mip 层，那么这将导致无效的纹理状态。不要这样做。

应该选择哪种过滤器取决于具体的应用以及希望实现的性能要求。例如，GL\_NEAREST\_MIPMAP\_NEAREST 具有非常好的性能，并且闪烁现象也非常弱，但最邻近过滤在视觉效果上常常难以令人满意。GL\_LINEAR\_MIPMAP\_NEAREST 常用于对游戏进行加速，因为它使用了更高质量的线性过滤器。但是，它需要在不同大小的可用 Mip 层之间进行快速选择（最邻近过滤）。

使用最邻近模式作为 Mip 贴图选择器（前面的两个例子都是如此）可能会导致难以令人满意的视觉效果。通过一个倾斜的观察角度，常常可以看到物体表面从一个 Mip 层到另一个 Mip 层的转变。我们可以看到一条扭曲的线段，或者从一个细节层次到另一个细节层次之间的急剧转变。GL\_LINEAR\_MIPMAP\_LINEAR 和 GL\_NEAREST\_MIPMAP\_LINEAR 过滤器在 Mip 层之间执行一些额外的插值，以消除它们之间的过渡痕迹，但它需要相当可观的额外处理开销。

GL\_LINEAR\_MIPMAP\_LINEAR 过滤器通常又称为三线性的 Mip 贴图，直到最近为止它还是纹理过滤的黄金准则（具有最高的精度）。最近，各向异性的纹理过滤（后面将进行讨论）逐渐在 OpenGL 硬件上流行，但它进一步增加了纹理贴图的开销（对性能的影响）。

## 5.4.2 生成 Mip 层

如前所述，和仅仅加载基本的纹理图像相比，Mip 贴图所需要的纹理内存要多出大约三分之一。它还要求所有更小的基本纹理图像都可以进行加载。有时候，这可能会带来不便，因为程序员或软件的终端用户并不一定需要这些更小的图像。对纹理的 Mip 层进行预先计算会得到最好的结果，同时让我们生成纹理是非常方便的，也是比较普遍的方式。一旦我们通过 glGenerateMipmap 函数加载了第 0 层，就可以为纹理生成所有的 Mip 层了。

```
void glGenerateMipmap(GLenum target);
```

目标参数可以是 GL\_TEXTURE\_1D、GL\_TEXTURE\_2D、GL\_TEXTURE\_3D、GL\_TEXTURE\_CUBE\_MAP、GL\_TEXTURE\_1D\_ARRAY 或 GL\_TEXTURE\_2D\_ARRAY（最后 3 个值将在第 7 章进行讨论）。用于创建最小纹理的过滤器的质量在各种实现上千差万别。此外，在运行过程中生成 Mip 贴图通常比加载预建的 Mip 贴图要慢，这在性能很关键的应用中也是需要考虑的。为了得到高视觉质量（同时也具有高度一致性），我们应该加载自己预先生成的 Mip 贴图。

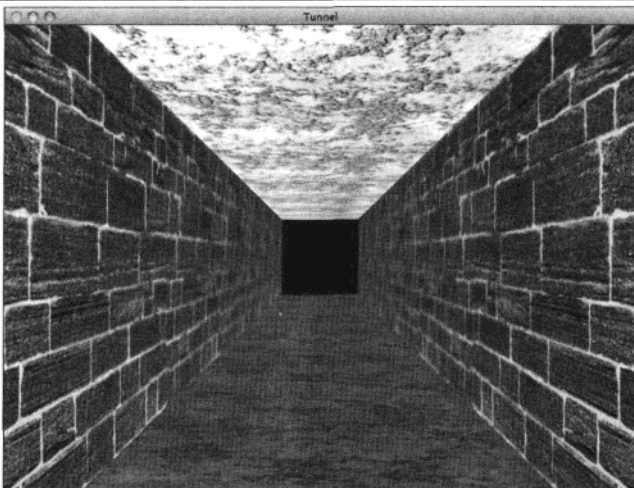
## 5.4.3 活动的 Mip 贴图

示例程序 Tunnel 显示了到目前为止本章已经讨论的所有主题，并在视觉上演示了不同的过滤器和

Mip 贴图模式。这个示例程序在启动时加载 3 个纹理，然后在它们之间进行切换，来对一个隧道进行渲染。这个隧道在地面和天花板上使用不同材料的砖墙图案。Tunnel 示例程序的输出结果如图 5.12 所示。

图 5.12

用 3 种不同的纹理和 Mip 贴图进行渲染的隧道（同样显示在了彩图 4 中）



TUNNEL 示例程序还显示了 Mip 贴图以及使用不同 Mip 贴图的纹理过滤模式。我们可以按上下方向键在隧道中前后移动观察点，也可以使用上下文菜单（右击菜单）在 6 种不同的过滤模式中进行选择，以观察它们是如何影响渲染图像的。这个程序的完整源代码如程序清单 5.4 所示。

程序清单 5.4 TUNNEL 示例程序的源代码

```
// Tunnel.cpp
// 演示 Mip 贴图以及使用纹理对象
// OpenGL 超级宝典
// Richard S. Wright Jr.
#include <GLTools.h>
#include <GLShaderManager.h>
#include <GLFrustum.h>
#include <GLBatch.h>
#include <GLFrame.h>
#include <GLMatrixStack.h>
#include <GLGeometryTransform.h>

#ifdef __APPLE__
#include <glut/glut.h>
#else
#define FREEGLUT_STATIC
#include <gl/glut.h>
#endif

GLShaderManager      shaderManager;          //着色器管理器
GLMatrixStack        modelViewMatrix;        //模型视图矩阵
GLMatrixStack        projectionMatrix;        //投影矩阵
GLFrustum            viewFrustum;            //视景体
GLGeometryTransform  transformPipeline;      //几何变换管线

GLBatch              floorBatch;
GLBatch              ceilingBatch;
GLBatch              leftWallBatch;
```

```

GLBatch      rightWallBatch;

GLfloat      viewZ = -65.0f;

// 纹理对象
#define TEXTURE_BRICK    0
#define TEXTURE_FLOOR    1
#define TEXTURE_CEILING  2
#define TEXTURE_COUNT    3
GLuint textures[TEXTURE_COUNT];
const char *szTextureFiles[TEXTURE_COUNT] = { "brick.tga",
                                              "floor.tga", "ceiling.tga" };

////////////////////////////////////
// 为每个纹理对象改变纹理过滤器
void ProcessMenu(int value)
{
    GLint iLoop;

    for(iLoop = 0; iLoop < TEXTURE_COUNT; iLoop++)
    {
        glBindTexture(GL_TEXTURE_2D, textures[iLoop]);

        switch(value)
        {
            case 0:
                glTexParameteri(GL_TEXTURE_2D,
                                GL_TEXTURE_MIN_FILTER, GL_NEAREST);
                break;

            case 1:
                glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

                break;

            case 2:
                glTexParameteri(GL_TEXTURE_2D,
                                GL_TEXTURE_MIN_FILTER, GL_NEAREST_MIPMAP_NEAREST);
                break;

            case 3:
                glTexParameteri(GL_TEXTURE_2D,
                                GL_TEXTURE_MIN_FILTER, GL_NEAREST_MIPMAP_LINEAR);
                break;

            case 4:
                glTexParameteri(GL_TEXTURE_2D,
                                GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
                break;

            case 5:
                glTexParameteri(GL_TEXTURE_2D,
                                GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
                break;
        }
    }
}

```

```

//触发重绘
glutPostRedisplay();
}

////////////////////////////////////
// 这个函数能够在渲染环境中进行任何需要的初始化。它在这里设置并初始化纹理对象。
void SetupRC()
{
    GLbyte *pBytes;
    GLint iWidth, iHeight, iComponents;
    GLenum eFormat;
    GLint iLoop;

    // 黑色背景
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

    shaderManager.InitializeStockShaders();

    // 加载纹理
    glGenTextures(TEXTURE_COUNT, textures);
    for(iLoop = 0; iLoop < TEXTURE_COUNT; iLoop++)
    {
        // 绑定下一个纹理对象
        glBindTexture(GL_TEXTURE_2D, textures[iLoop]);

        //加载纹理, 设置过滤器和环绕模式
        pBytes = gltReadTGABits(szTextureFiles[iLoop], &iWidth, &iHeight,
                                &iComponents, &eFormat);

        //加载纹理, 设置过滤器和环绕模式
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
        glTexImage2D(GL_TEXTURE_2D, 0, iComponents, iWidth, iHeight,
                    0, eFormat, GL_UNSIGNED_BYTE, pBytes);
        glGenerateMipmap(GL_TEXTURE_2D);
        // 不再需要初始的纹理数据
        free(pBytes);
    }

    // 建立几何图形
    GLfloat z;
    floorBatch.Begin(GL_TRIANGLE_STRIP, 28, 1);
    for(z = 60.0f; z >= 0.0f; z -= 10.0f)
    {
        floorBatch.MultiTexCoord2f(0, 0.0f, 0.0f);
        floorBatch.Vertex3f(-10.0f, -10.0f, z);

        floorBatch.MultiTexCoord2f(0, 1.0f, 0.0f);
        floorBatch.Vertex3f(10.0f, -10.0f, z);

        floorBatch.MultiTexCoord2f(0, 0.0f, 1.0f);
        floorBatch.Vertex3f(-10.0f, -10.0f, z - 10.0f);
    }
}

```

```
        floorBatch.MultiTexCoord2f(0, 1.0f, 1.0f);
        floorBatch.Vertex3f(10.0f, -10.0f, z - 10.0f);
    }
    floorBatch.End();

    ceilingBatch.Begin(GL_TRIANGLE_STRIP, 28, 1);
    for(z = 60.0f; z >= 0.0f; z -=10.0f)
    {
        ceilingBatch.MultiTexCoord2f(0, 0.0f, 1.0f);
        ceilingBatch.Vertex3f(-10.0f, 10.0f, z - 10.0f);

        ceilingBatch.MultiTexCoord2f(0, 1.0f, 1.0f);
        ceilingBatch.Vertex3f(10.0f, 10.0f, z - 10.0f);

        ceilingBatch.MultiTexCoord2f(0, 0.0f, 0.0f);
        ceilingBatch.Vertex3f(-10.0f, 10.0f, z);

        ceilingBatch.MultiTexCoord2f(0, 1.0f, 0.0f);
        ceilingBatch.Vertex3f(10.0f, 10.0f, z);
    }
    ceilingBatch.End();

    leftWallBatch.Begin(GL_TRIANGLE_STRIP, 28, 1);
    for(z = 60.0f; z >= 0.0f; z -=10.0f)
    {
        leftWallBatch.MultiTexCoord2f(0, 0.0f, 0.0f);
        leftWallBatch.Vertex3f(-10.0f, -10.0f, z);

        leftWallBatch.MultiTexCoord2f(0, 0.0f, 1.0f);
        leftWallBatch.Vertex3f(-10.0f, 10.0f, z);

        leftWallBatch.MultiTexCoord2f(0, 1.0f, 0.0f);
        leftWallBatch.Vertex3f(-10.0f, -10.0f, z - 10.0f);

        leftWallBatch.MultiTexCoord2f(0, 1.0f, 1.0f);
        leftWallBatch.Vertex3f(-10.0f, 10.0f, z - 10.0f);
    }
    leftWallBatch.End();

    rightWallBatch.Begin(GL_TRIANGLE_STRIP, 28, 1);
    for(z = 60.0f; z >= 0.0f; z -=10.0f)
    {
        rightWallBatch.MultiTexCoord2f(0, 0.0f, 0.0f);
        rightWallBatch.Vertex3f(10.0f, -10.0f, z);

        rightWallBatch.MultiTexCoord2f(0, 0.0f, 1.0f);
        rightWallBatch.Vertex3f(10.0f, 10.0f, z);

        rightWallBatch.MultiTexCoord2f(0, 1.0f, 0.0f);
        rightWallBatch.Vertex3f(10.0f, -10.0f, z - 10.0f);

        rightWallBatch.MultiTexCoord2f(0, 1.0f, 1.0f);
        rightWallBatch.Vertex3f(10.0f, 10.0f, z - 10.0f);
    }
    rightWallBatch.End();
}
```

```

////////////////////////////////////
// 关闭渲染环境。删除纹理对象即可。
void ShutdownRC(void)
{
    glDeleteTextures(TEXTURE_COUNT, textures);
}

////////////////////////////////////
// 前后移动视口来对方向键作出响应
void SpecialKeys(int key, int x, int y)
{
    if(key == GLUT_KEY_UP)
        viewZ += 0.5f;

    if(key == GLUT_KEY_DOWN)
        viewZ -= 0.5f;

    // 更新窗口
    glutPostRedisplay();
}

////////////////////////////////////
// 改变视景体和视口。在改变窗口大小时调用
void ChangeSize(int w, int h)
{
    GLfloat fAspect;

    // 防止对 0 进行除法操作
    if(h == 0)
        h = 1;

    // 将视口设置为窗口大小
    glViewport(0, 0, w, h);

    fAspect = (GLfloat)w/(GLfloat)h;

    // 生成透视投影
    viewFrustum.SetPerspective(80.0f, fAspect, 1.0, 120.0);
    projectionMatrix.LoadMatrix(viewFrustum.GetProjectionMatrix());
    transformPipeline.SetMatrixStacks(modelViewMatrix, projectionMatrix);
}

////////////////////////////////////
// 进行调用以绘制场景
void RenderScene(void)
{
    // 用当前清除色清除窗口
    glClear(GL_COLOR_BUFFER_BIT);

    modelViewMatrix.PushMatrix();
        modelViewMatrix.Translate(0.0f, 0.0f, viewZ);

```

```

        shaderManager.UseStockShader(GLT_SHADER_TEXTURE_REPLACE,
                                     transformPipeline.GetModelViewProjectionMatrix(),
                                     0);

        glBindTexture(GL_TEXTURE_2D, textures[TEXTURE_FLOOR]);
        floorBatch.Draw();

        glBindTexture(GL_TEXTURE_2D, textures[TEXTURE_CEILING]);
        ceilingBatch.Draw();

        glBindTexture(GL_TEXTURE_2D, textures[TEXTURE_BRICK]);
        leftWallBatch.Draw();
        rightWallBatch.Draw();

        modelViewMatrix.PopMatrix();

        // 缓冲区交换
        glutSwapBuffers();
    }

    //////////////////////////////////////
    // 程序入口点
    int main(int argc, char *argv[])
    {
        gltSetWorkingDirectory(argv[0]);

        // 标准初始化
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
        glutInitWindowSize(800, 600);
        glutCreateWindow("Tunnel");
        glutReshapeFunc(ChangeSize);
        glutSpecialFunc(SpecialKeys);
        glutDisplayFunc(RenderScene);

        // 添加菜单入口以改变过滤器
        glutCreateMenu(ProcessMenu);
        glutAddMenuEntry("GL_NEAREST", 0);
        glutAddMenuEntry("GL_LINEAR", 1);
        glutAddMenuEntry("GL_NEAREST_MIPMAP_NEAREST", 2);
        glutAddMenuEntry("GL_NEAREST_MIPMAP_LINEAR", 3);
        glutAddMenuEntry("GL_LINEAR_MIPMAP_NEAREST", 4);
        glutAddMenuEntry("GL_LINEAR_MIPMAP_LINEAR", 5);

        glutAttachMenu(GLUT_RIGHT_BUTTON);

        GLenum err = glewInit();
        if (GLEW_OK != err) {
            fprintf(stderr, "GLEW Error: %s\n", glewGetErrorString(err));
            return 1;
        }

        // 启动, 循环, 关闭
        SetupRC();
        glutMainLoop();
    }

```

```

ShutdownRC();

return 0;
}

```

在这个例子中，我们首先为 3 个纹理对象创建标识符。textures 数组将包含 3 个整数，它们可以用宏 TEXTURE\_BRICK、TEXTURE\_FLOOR 和 TEXTURE\_CEILING 进行访问。为了增加灵活性，我们还创建了一个宏，定义了将要加载的纹理的最大数量，并且创建了一个字符串数组，包含了纹理贴图文件的名称。

```

//纹理对象
#define TEXTURE_BRICK 0
#define TEXTURE_FLOOR 1
#define TEXTURE_CEILING 2
#define TEXTURE_COUNT 3
GLuint textures[TEXTURE_COUNT];
const char *szTextureFiles[TEXTURE_COUNT]=
    {"brick.tga", "floor.tga", "ceiling.tga"};

```

纹理对象是在 SetupRC 函数中分配的。

```
glGenTextures(TEXTURE_COUNT, textures);
```

然后是一个简单的循环，依次绑定每个纹理对象，并加载包含纹理图像和纹理参数的纹理状态。

```

for(iLoop = 0; iLoop < TEXTURE_COUNT; iLoop++)
{
    // 绑定到下一个纹理对象
    glBindTexture(GL_TEXTURE_2D, textures[iLoop]);

    //加载纹理，设置过滤器和环绕模式
    pBytes = gltReadTGABits(szTextureFiles[iLoop], &iWidth, &iHeight,
                           &iComponents, &eFormat);

    //加载纹理，设置过滤器和环绕模式
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexImage2D(GL_TEXTURE_2D, 0, iComponents, iWidth, iHeight,
                0, eFormat, GL_UNSIGNED_BYTE, pBytes);
    glGenerateMipmap(GL_TEXTURE_2D);

    // 不再需要初始的纹理数据了
    free(pBytes);
}

```

在这 3 个纹理对象都进行了初始化之后，我们就可以很方便地在渲染时在它们之间进行切换，以更改纹理。

```

glBindTexture(GL_TEXTURE_2D, textures[TEXTURE_FLOOR]);
floorBatch.Draw();

glBindTexture(GL_TEXTURE_2D, textures[TEXTURE_CEILING]);
ceilingBatch.Draw();
....
....

```

最后,当程序终止时,我们只需要删除纹理对象,以完成最后的清理任务。

```
////////////////////////////////////  
//关闭渲染环境。删除纹理对象即可  
void ShutdownRC(void)  
{  
    glDeleteTextures(TEXTURE_COUNT, textures);  
}
```

同时还需要注意,当 TUNNEL 示例程序设置 Mip 贴图纹理过滤器时,它被选择为只用于缩小过滤器。

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
```

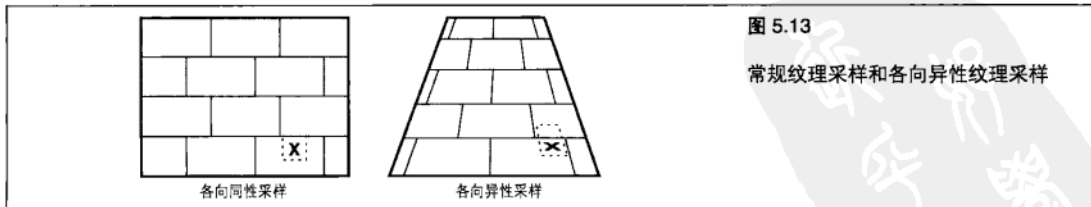
这是最为典型的情况,因为在 OpenGL 选择了最大可用的 Mip 层之后,就没有更大的 Mip 层可供选择了。从本质上说,这相当于一旦传递了一个门限值,实际使用的就是最大纹理图像,不再有其他 Mip 贴图层可供选择。

## 5.5 各向异性过滤

各向异性纹理过滤 (Anisotropic texture filtering) 并不是 OpenGL 核心规范的一部分,但它是一种得到广泛支持的扩展,可以极大地提高纹理过滤操作的质量。我们在本章前面内容中讲述了纹理贴图,并学习了两种基本的纹理过滤: 最邻近过滤 (GL\_NEAREST) 和线性过滤 (GL\_LINEAR)。当一个纹理贴图被过滤时,OpenGL 使用纹理坐标来判断一个特定的几何片段将落在纹理贴图的什么地方。然后,紧邻这个位置的纹理单元使用 GL\_NEAREST 或 GL\_LINEAR 过滤操作进行采样。

当几何图形进行纹理贴图时,如果它的观察方向与观察点恰好垂直,那么这个过程是相当完美的,如图 5.13 的左侧所示。但是,当我们从一个角度倾斜地观察这个几何图形时,对周围纹理单元进行常规采样将导致一些纹理信息的丢失(看上去显得模糊)。

更为逼真和准确的采样应该是沿着包含纹理的平面方向进行延伸。它的结果如图 5.13 的右侧所示。如果我们在进行纹理过滤时考虑了观察角度,那么这种过滤方法就称为各向异性过滤。



我们还可以把各向异性过滤应用到所有的基本纹理过滤或 Mip 贴图纹理过滤模型中。应用各向异性过滤需要 3 个步骤。首先,必须确认这种扩展是得到支持的。为此,可以查询扩展字符串 GL\_EXT\_texture\_filter\_anisotropic。我们可以使用 glTools 函数 gltExtensionSupported 完成这个任务。

```
if(gltIsExtSupported("GL_EXT_texture_filter_anisotropic"))  
    //设置标志,表示这个扩展受到支持
```

在确认这个扩展得到支持之后，就可以查询得到支持的各向异性过滤的最大数量。为此，可以调用 `glGetFloatv` 函数，并以 `GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT` 为参数。

```
GLfloat fLargest;
....
....
glGetFloatv(GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT, &fLargest);
```

各向异性过滤所应用的数量越大，沿最大变化方向（沿最强的观察点）所采样的纹理单元就越多。值 1.0 表示常规的纹理过滤（称为各向同性过滤）。请记住，各向异性过滤并不是不需要付出代价的。额外的工作，包括其它纹理单元，有时候可能会对性能造成相当大的影响。不过，在现代的硬件上，应用这个特性对速度造成的影响并不大，目前它已经成为流行游戏、动画和模拟程序的一个标准特性。

最后，我们可以用 `glTexParameter` 函数以及 `GL_TEXTURE_MAX_ANISOTROPY_EXT` 常量设置想要应用各向异性过滤的数量。例如，使用前面的代码，如果想应用最大数量的各向异性过滤，可以像下面这样调用 `glTexParameter` 函数。

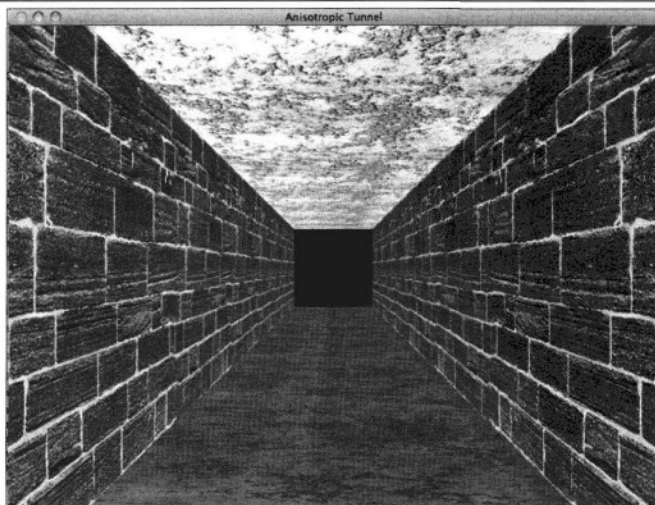
```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY_EXT, fLargest);
```

各向异性过滤是以每个纹理对象为基础进行应用的，就像标准过滤参数一样。示例程序 `Anisotropic` 提供了各向异性纹理过滤的一个实际应用例子。这个程序显示一个用墙、地面和天花板几何图形组成的隧道，这实际上是前面的 `Tunnel` 示例程序的一个活动版本。我们可以使用方向键在隧道内部向前或向后移动观察点（或隧道）。右键单击鼠标会弹出一个菜单，允许我们选择各种纹理过滤器，并允许打开或关闭各向异性过滤。图 5.12 所示显示了使用三线性过滤的 Mip 贴图隧道。注意在远处，物体（特别是砖块）的图案是如何变得模糊的。

我们可以将图 5.12 与图 5.14 进行比较，后者启用了各向异性过滤（彩插中的彩图 4 并列显示了这些图像）。现在，砖块之间的灰泥清晰可见，即使到了隧道的终端依然如此。事实上，各向异性的过滤还可以极大地弱化 `GL_LINEAR_MIPMAP_NEAREST` 和 `GL_NEAREST_MIPMAP_NEAREST` 类型的 Mip 贴图过滤器所存在的 Mip 贴图过渡图案。

图 5.14

使用各向异性过滤的 `Anisotropic` 隧道例子（同样显示在了彩图 4 中）



## 5.6 纹理压缩

纹理贴图可以在 3D 渲染场景中增加令人难以置信的逼真性，而它在顶点处理上所需要付出的代价非常之少。但是，使用纹理存在一个缺点，就是它们需要大量的内存来存储和处理纹理。早期对纹理压缩的尝试就是简单地把纹理作为 JPG 文件存储，在调用 `glTexImage` 之前进行加载时对它进行解压。这种做法可以节省磁盘空间并减少在网络（如 Internet）上传输图像所需要的时间，但对于缓解加载到图形硬件内存的纹理图像的存储需求则没有多大帮助。

在 1.3 版本中，OpenGL 添加了对纹理压缩的本地支持。早期版本的 OpenGL 也可以通过相同名称的扩展函数支持纹理压缩。我们可以使用 `GL_ARB_texture_compression` 字符串测试这个扩展是否得到支持。

OpenGL 硬件对纹理压缩的支持远远不止是简单地允许加载经过压缩的纹理。在绝大多数实现中，纹理数据甚至在图形硬件内存中仍然保持压缩状态。这就允许我们在较小的内存中加载更多的纹理，从而显著地改善纹理处理的性能，这是由于在纹理过滤时减少了纹理交换（移动纹理）并使用了更少内存的原因。

### 5.6.1 压缩纹理

为了利用 OpenGL 对压缩纹理的支持，纹理数据一开始并不需要进行压缩。我们可以在加载一幅纹理图像时请求 OpenGL 对它进行压缩，这是通过在 `glTexImage` 函数中把 `internalFormat` 参数设置为表 5.6 中的任意一个值实现的。

表 5.6

通用压缩纹理格式

压缩格式	基本内部格式
<code>GL_COMPRESSED_RGB</code>	<code>GL_RGB</code>
<code>GL_COMPRESSED_RGBA</code>	<code>GL_RGBA</code>
<code>GL_COMPRESSED_SRGB</code>	<code>GL_RGB</code>
<code>GL_COMPRESSED_SRGB_ALPHA</code>	<code>GL_RGBA</code>
<code>GL_COMPRESSED_RED</code>	<code>GL_RED</code>
<code>GL_COMPRESSED_RG</code>	<code>GL_RG(Red Green)</code>

除了这些通用压缩格式之外，OpenGL 3.2 中还加入了一些特定的压缩格式，即 `GL_COMPRESSED_SIGNED_RED_RGTC1`、`GL_COMPRESSED_RG_RGTC2` 和 `GL_COMPRESSED_SIGNED_RG_RGTC2`。它们用于各种单颜色通道和双颜色通道压缩纹理。实际上，它们代替了兼容版本中 `GL_LUMINANCE` 和 `GL_LUMINANCE_ALPHA` 的功能。

通过这种方式进行图像压缩增加了纹理加载的开销，但却能够通过更有效地使用纹理存储空间来增加纹理性能。如果由于某些原因而无法对纹理进行压缩，OpenGL 就会使用上表所列出的基本内部格式，并加载未经压缩的纹理。

当我们试图按照这种方式加载并压缩一个纹理时，可以使用 `glGetTexLevelParameteriv` 函数（以 `GL_TEXTURE_COMPRESSED` 为参数）来判断这个纹理是否被成功压缩。

```
GLint compFlag;
.....
glGetTexLevelParameteriv(GL_TEXTURE_2D, 0, GL_TEXTURE_COMPRESSED, &compFlag);
```

`glGetTexLevelParameteriv` 函数可以接受几个新的参数名，它们都和压缩纹理有关。表 5.7 列出了这些参数。

表 5.7 用 `glGetTexLevelParameter` 函数提取的压缩纹理格式

参 数	返 回
<code>GL_TEXTURE_COMPRESSED</code>	如果纹理被压缩，返回 1，否则返回 0
<code>GL_TEXTURE_COMPRESSED_IMAGE_SIZE</code>	压缩后的纹理的大小（以字节为单位）
<code>GL_TEXTURE_INTERNAL_FORMAT</code>	所使用的压缩格式
<code>GL_NUM_COMPRESSED_TEXTURE_FORMATS</code>	受支持的压缩纹理格式的数量
<code>GL_COMPRESSED_TEXTURE_FORMATS</code>	一个包含了一些常量值的数组，每个常量值对应于一种受支持的压缩纹理格式
<code>GL_TEXTURE_COMPRESSION_HINT</code>	纹理压缩提示的值（ <code>GL_NICEST/GL_FASTEST</code> ）

当纹理使用表 5.6 所列出的值进行压缩之后，OpenGL 会选择最适当的纹理压缩格式。我们可以使用 `glHint` 指定希望 OpenGL 根据最快速度还是最佳质量算法来选择压缩格式。

```
glHint(GL_TEXTURE_COMPRESSION_HINT, GL_FASTEST);
glHint(GL_TEXTURE_COMPRESSION_HINT, GL_NICEST);
glHint(GL_TEXTURE_COMPRESSION_HINT, GL_DONT_CARE);
```

具体的压缩格式因实现而异，可以使用 `GL_NUM_COMPRESSED_TEXTURE_FORMATS` 和 `GL_COMPRESSED_TEXTURE_FORMATS` 为参数查询压缩格式的数量以及相关值的列表。为了检查一组特定的压缩纹理格式是否得到支持，需要检查一个与这些格式有关的特定扩展。例如，几乎所有的桌面应用都支持 `GL_EXT_texture_compression_s3tc` 纹理压缩格式。如果这个扩展是得到支持的，那么表 5.8 所列出的所有压缩纹理格式都是得到支持的，但它们只适用于二维纹理。

表 5.8 `GL_EXT_texture_compression_s3tc` 的压缩格式

格 式	描 述
<code>GL_COMPRESSED_RGB_S3TC_DXT1</code>	RGB 数据被压缩，alpha 值始终是 1.0
<code>GL_COMPRESSED_RGBA_S3TC_DXT1</code>	RGB 数据被压缩，alpha 值是 1.0 或 0.0
<code>GL_COMPRESSED_RGBA_S3TC_DXT3</code>	RGB 数据被压缩，alpha 值用 4 位存储
<code>GL_COMPRESSED_RGBA_S3TC_DXT5</code>	RGB 数据被压缩，alpha 值是一些 8 位值的加权平均值

## 5.6.2 加载压缩纹理

使用前面所介绍的函数，可以让 OpenGL 用一种本地支持的格式对纹理进行压缩，用 `glGetCompressedTexImage` 函数（相当于未压缩纹理的 `glTexImage` 函数）提取经过压缩的数据并把它

保存到磁盘中。在后续的纹理加载中，可以使用原始压缩数据，从而极大地提高纹理的加载速度。但是，还要注意有些生产商在加载纹理以优化纹理存储或过滤操作方面稍稍存在一些欺骗性。因此，这个技巧只有在完全遵循标准的硬件实现中才是可行的。为了加载预先经过压缩的纹理数据，可以使用下列函数之一。

```
void glCompressedTexImage1D(GLenum target, GLint level,
                           GLenum internalFormat,
                           GLsizei width,
                           GLint border, GLsizei imageSize, void *data);

void glCompressedTexImage2D(GLenum target, GLint level, GLenum internalFormat,
                           GLsizei width, GLsizei height,
                           GLint border, GLsizei imageSize, void *data);

void glCompressedTexImage3D(GLenum target, GLint level,
                           GLenum internalFormat,
                           GLsizei width, GLsizei height, GLsizei depth,
                           GLint border, GLsizei imageSize, GLvoid *data);
```

这些函数实际上与前一章的 `glTexImage` 函数等同，仅有的区别是这些函数的 `internalFormat` 参数必须指定一种受到支持的压缩纹理图像格式。如果我们所使用的 OpenGL 实现支持 `GL_EXT_texture_compression_s3tc` 扩展，那么这个参数所取的值可以是表 5.8 所列出的值之一。另外还有一组对应的 `glCompressedTexSubImage` 函数，用于更新部分或全部已经加载的纹理，它们与前一章 `glTexSubImage` 的功能相当。

纹理压缩是一种非常流行的纹理特性。较小的纹理所占据的空间更小、通过网络传输的速度更快、从磁盘加载的速度更快、复制到图形内存也更加快速，并且允许更多的纹理加载到硬件中，而且使纹理的启用更加快速！但是，不要忘了，和生活中的很多事情一样，天下没有免费的午餐。使用纹理压缩也要付出一些代价。例如，`GL_EXT_texture_compression_s3tc` 方法是通过从每个纹理单元提取颜色数据进行工作的。

对于有些纹理，这样可能会导致图像质量的损失（尤其是那些包含了平滑颜色过渡的纹理）。在其他一些时候，具有大量细节的纹理在视觉上几乎与未压缩的源纹理相同，颜色的变化几乎无法察觉。纹理压缩方法的选择（或选择不使用纹理压缩）可能极大地依赖于底层图像的本质。

### 5.6.3 最后一个示例

本章介绍的最后一个程序是 `SphereWorld`（球体世界），这是利用我们到目前为止所介绍的技术所创建的一个代表作。如图 5.15 所示，`SphereWorld` 示例程序中添加了漂浮的球体，这些球体盘旋在反光的大理石地板上方。这种反光效果只不过是烟雾和镜面的小把戏——首先渲染反转的场景，然后在上面对大理石地板，再使用一个小的 Alpha 值与背景进行混合。然后再迅速上下颠倒地渲染场景——一个简单的反射效果。我们还可以使用方向键进行移动。在这个示例程序中，唯一真正的新东西就是对 `LoadTGA` 函数所做的小小改变，在这里我们将内部格式参数改为了通用压缩纹理格式 `GL_COMPRESSED_RGB`。

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_COMPRESSED_RGB, nWidth, nHeight, 0,
             eFormat, GL_UNSIGNED_BYTE, pBits);
```

图 5.15

添加了反光大理石地板的完整  
SphereWorld 示例程序



## 5.7 小结

本章内容实际上是纹理贴图的初级教程——它是第一堂课,也是我们在学习更多中级和高级技术之前需要掌握的基础。

从现在开始,我们已经做好了开始建立有趣的 3D 场景和在表面上应用纹理的准备。我们已经学习了如何从图像数据中加载纹理并将其传递到 OpenGL, 以及通过纹理坐标在几何图形上应用纹理。我们已经了解了如何设置不同的图像过滤器, 以及这些图像过滤器的功能, 如何使用硬件来生成 Mip 贴图, 并且使用它们来提高渲染性能和视觉质量。我们还学习了更加高级的过滤选项, 即各向异性过滤, 并且通过 Tunnel 和 Anisotropic 两个示例程序了解了这些选项所带来的戏剧性变化。

最后, 我们还介绍了纹理压缩, 不但讲解了如何在运行过程中压缩一个纹理, 还讲解了如何直接加载一个预压缩的纹理。

实际上, 到这里我们已经揭示了所有关于 3D 图形是什么的内容。我们只是将点来回平移、用图元连接点、使用计算得到的颜色值或者从某个图形文件中采样得到的纹理单元来填充图元内部。有时候我们也会混合使用这些结果。当事情进展到如何组合和渲染任何我们能够想象到的 3D 场景时, 我们就确实找到感觉了。对于本书剩下的内容来说, 实际上我们将要学习的还是前几章中介绍过的这些主题, 不过是越来越深入地进行研究。做好准备, 来迎接第 6 章吧!

## 第6章 跳出“盒子”：非存储着色器

作者：Richard S. Wright, Jr.

### 本章内容

- ✦ 如何通过 GLBatch 编写自己的着色器
- ✦ 介绍不同的 GLSL 变量类型
- ✦ 介绍 GLSL 的内建函数
- ✦ 如何使用自己的光照明着色器
- ✦ 如何在 GLSL 中使用一维和二维纹理
- ✦ 如何雕琢独立的片段

在第 3 章，我们第一次介绍了着色器和它们的使用方法。如果跳过这部分直接进行着色器编程，那么我们确实需要先复习一下这一章，并且确保能够懂得如何使用这些属性和 Uniform 值，以及如何将它们从客户端代码传递到着色器中。在第 3 章，我们把精力完全放在了客户端上，并且使用了一些预建的存储着色器，这些存储着色器执行一些例行程序和典型的渲染操作。在本章中，我们将进一步研究客户端操作，但最终将开始学习如何编写自己的着色器，即服务器端的着色器应用：着色器编程和着色语言。

### 6.1 初识 OpenGL 着色语言

OpenGL 着色语言 (GLSL) 看上去很像 C 语言，它由 OpenGL 实现进行编译和连接，并且（经常是）完全在图形硬件中运行。

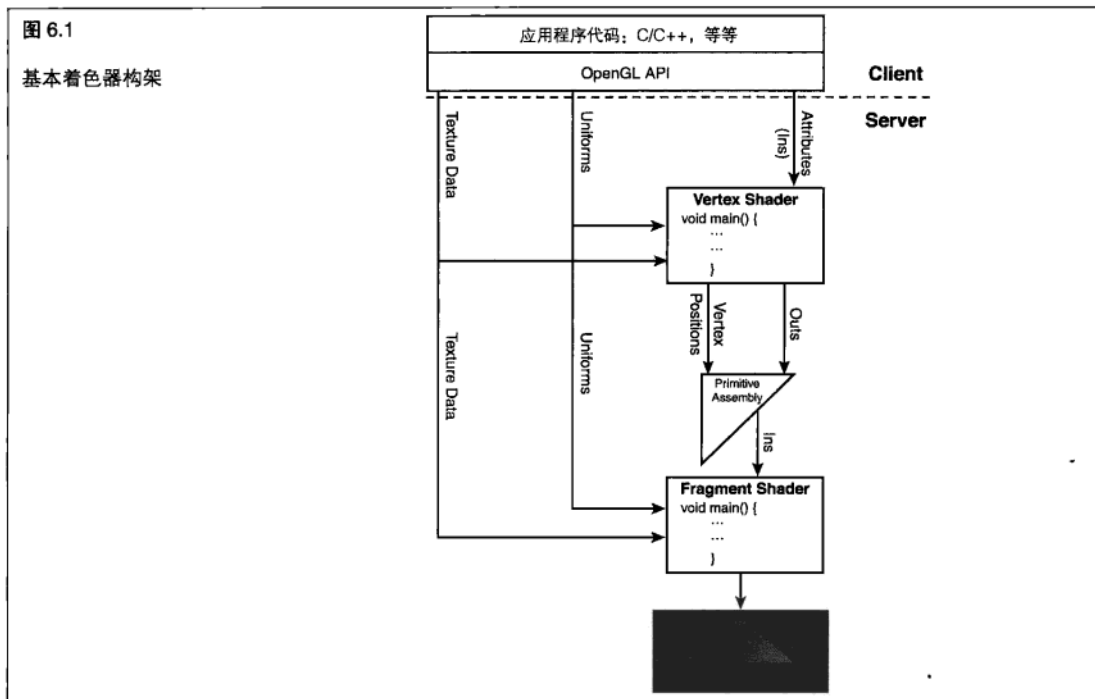
我们把第 3 章中的图 3.1 作为图 6.1，展示一下基本着色器架构。

正如大家所看到的，我们至少需要两个着色器：一个顶点着色器、一个片段着色器。还有一种可选的

着色器称为几何着色器，将在第 11 章详细介绍。我们可以在 3 种方式中选择一种向顶点着色器传递数据：一是参数，是对每个顶点而言的；二是统一值，是针对整个顶点数据批次的常量（所以是一致的）；最后是在第 5 章学到的，也可以加载和使用纹理数据。我们可以为片段着色器设置统一值和纹理数据。将顶点属性发送到片段着色器毫无意义，因为片段着色器只是用来在图元进行光栅化后对片段（最基本的是像素）进行填充。不过，每个顶点数据都可以通过顶点程序传递到片段着色器。但是在这种情况下，这些数据可能是常量（每个片段都是同样的值），或者这些值也可以用不同的方式在图元表面进行插值。

图 6.1

基本着色器构架



着色器程序看起来确实和 C 语言非常类似，它们从入口点 `main` 函数开始，并且使用同样的字符集和注释约定，以及很多相同的处理指令。我们可以在 OpenGL 着色语言规范（OpenGL Shading Language Specification）中找到一个完整的语言规范。附录 A 中列出了一些网址，可以帮助我们找到这些文档，另外还有其他一些非常好的参考文献和补充教程。为了达到目的，我们假定读者已经达到了所需的条件，即已经熟悉了 C/C++ 语言，这样才能从一个 C/C++ 程序员的角度来专注 OpenGL 着色语言。

### 6.1.1 变量和数据类型

要学习 OpenGL 着色语言，从讨论我们可用的数据类型开始是个不错的选择。我们可用的数据类型只有 4 种：整数（包括有符号整数和无符号整数）、浮点数（对于 OpenGL 3.3 来说只能使用单精度浮点数）和布尔值（`bool`）。在 OpenGL 着色语言中没有指针，并且没有任何类型的字符串或字符。函数可以返回这些类型中的任何一种，但也可以声明为 `void`，不过同样不允许 `void` 指针。这些数据类型在 OpenGL 着色语言中的应用与它们在 C/C++ 中一样。

```
bool bDone = false; // 布尔值, 真(true)或假(false)
int iValue = 42; // 有符号整数
uint uiValue = 3929u; // 无符号整数
float fValue = 42.0f; // 浮点值
```

## 向量类型

OpenGL 着色语言有一个独特而令人兴奋的特性(和 C/C++相比),就是可以使用向量数据类型。所有这 4 种基本数据类型都可以存储在二维、三维或者四维向量中。表 6.1 中列出了向量数据类型的完整列表。

表 6.1 OpenGL 着色语言向量数据类型

类 型	描 述
vec2,vec3,vec4	2 分量、3 分量和 4 分量浮点向量
ivec2,ivec3,ivec4	2 分量、3 分量和 4 分量整数向量
uvec2,uvec3,uvec4	2 分量、3 分量和 4 分量无符号整数向量
bvec2,bvec3,bvec4	2 分量、3 分量和 4 分量布尔向量

一个向量数据类型可以像任何其他类型的变量一样进行声明,例如,我们可以用一个 4 分量的浮点向量声明一个顶点位置,如下所示。

```
vec4 vVertexPos;
```

我们也可以用一个构造函数初始化一个向量。

```
vec4 vVertexPos = vec4(39.0f, 10.0f, 0.0f, 1.0f);
```

请注意,不要把这个构造函数与 C++类构造函数混淆。OpenGL 着色语言的向量数据类型并不是类,它们有自己的内建数据类型。向量可以被赋值给另一个向量,可以相加,也可以用一个标量(非向量类型)进行缩放。

```
vVertexPos = vOldPos + vOffset;
vVertexPos = vNewPos;
vVertexPos += vec4(1.0f, 1.0f, 0.0f, 0.0f);
vVertexPos *= 5.0f;
```

OpenGL 着色语言的另一个独特的性质是对一个向量的独立元素进行寻址的方式。如果读者熟悉 C/C++中的 union (联合体)结构就会知道,向量与联合体非常类似。

我们用点号来确定多达 4 个向量元素的地址,但是可以使用下列 3 组标识符中的任意一组:xyzw、rgba 或 stpq。典型情况下,可以在引用向量类型数据时使用 xyzw 组标识符。

```
vVertexPos.x = 3.0f;
vVertexPos.xy = vec2(3.0f, 5.0f);
vVertexPos.xyz = vNewPos.xyz;
```

然后,在进行颜色操作时使用 rgba。

```
vOutputColor.r = 1.0f;
vOutputColor.rgb = vec3(1.0f, 1.0f, 0.5f, 1.0f);
```

最后,在进行纹理坐标操作时,使用 stpq。

```
vTexCoord.st = vec2(1.0f, 0.0f);
```

选择使用哪一组标识符对于 OpenGL 着色语言来说是完全随意的,例如我们可以简单地进行如下操作。

```
vTexCoord.st = vVertex.st;
```

不过,我们不能将不同的组混合到一个向量访问中,如下所示。

```
vTexCoord.st = vVertex.xt; // 对 x 和 t 进行混合是不允许的!
```

向量数据类型还支持 swizzle (调换) 操作。Swizzle (调换) 操作是指将两个或两个以上的向量元素进行交换。例如,如果要将颜色数据从 RGB 顺序转换到 BGR 顺序,只要用下面这行代码就能做到。

```
vNewColor.bgra = vOldColor.rgba;
```

向量数据类型并不只是 OpenGL 着色语言的本地数据类型,也是硬件的本地数据类型。它们的速度很快,而且能一次性完成对所有分量的操作。例如,下列操作:

```
vVertex.x = vOtherVertex.x + 5.0f;
vVertex.y = vOtherVertex.y + 4.0f;
vVertex.z = vOtherVertex.z + 1.0f;
```

在我们使用本地向量符号时会快得多。

```
vVertex.xyz = vOtherVertex.xyz + vec3(5.0f, 4.0f, 1.0f);
```

## 矩阵类型

除了向量数据类型之外,OpenGL 着色语言还支持多种矩阵类型。不过,和向量类型不同,所有矩阵类型都只支持浮点数——抱歉,不支持整数或者布尔值矩阵。表 6.2 列出了支持的矩阵类型。

表 6.2 OpenGL 着色语言矩阵数据类型

类 型	描 述
mat2, mat2x2	两行两列
mat3, mat3x3	三行三列
mat4, mat4x4	四行四列
mat2x3	三行两列
mat2x4	四行两列
mat3x2	两行三列
mat3x4	四行三列
mat4x2	两行四列
mat4x3	三行四列

实际上在 OpenGL 着色语言中,一个矩阵就是一个由向量组成的数组——实际上是列向量(在这里我们可以复习一下第 4 章中介绍过的列优先向量)。例如,为了设置一个 4x4 矩阵的最后一列,我们可以编写类似下面这样的代码。

```
mModelView[3] = vec4(0.0f, 0.0f, 0.0f, 1.0f);
```

相应地，可以按照如下方式恢复一个矩阵的最后一列。

```
vec4 vTranslation = mModelView[3];
```

或者甚至可以进行更加细致的查询。

```
vec3 vTranslation = mModelView[3].xyz;
```

矩阵也可以乘以向量，这种运算通常在通过模型视图投影矩阵来对一个向量进行变换时使用，就像下面这样。

```
vec4 vVertex;
mat4 mvpMatrix;
.....
.....
vOutPos = mvpMatrix * vVertex;
```

同样，就像向量一样，矩阵数据类型也有它们自己的构造函数。例如，为了对一个矩阵的最后一列进行硬编码，我们可以编写类似下面这样的代码。

```
mat4 vTransform = mat4(1.0f, 0.0f, 0.0f, 0.0f,
                       0.0f, 1.0f, 0.0f, 0.0f,
                       0.0f, 0.0f, 1.0f, 0.0f,
                       0.0f, 0.0f, 0.0f, 1.0f);
```

在这种情况下，我们会采用单位矩阵作为变换矩阵，还可以使用一个更快的矩阵构造器，它只将单个值填入矩阵的对角线。

```
mat4 vTransform = mat4(1.0f);
```

## 6.1.2 存储限定符

着色器变量声明也可以选择指定一个存储限定符。限定符用于将变量标记为输入变量（in 或 uniform）、输出变量（out）或常量（const）。输入变量接受来自 OpenGL 客户端（通过 C/C++ 提交的属性）或者以前的着色器阶段（例如从顶点着色器传递到片段着色器）。输出变量是指在任何着色器阶段进行写入的变量，我们希望后续的着色器阶段能看到这些变量，例如，从顶点着色器传递到片段着色器，或者用片段着色器写入最终片段颜色。表 6.3 列出了主要的变量限定符。

表 6.3

变量存储限定符

限定符	描述
<none>	只是普通的本地变量，外部不可见，外部不可访问
const	一个编译时常量，或者说是一个对函数来说为只读的参数
in	一个从以前的阶段传递过来的变量
in centroid	一个从以前的阶段传递过来的变量，使用质心插值
out	传递到下一个处理阶段或者在一个函数中指定一个返回值
out centroid	传递到下一个处理阶段，使用质心插值
inout	一个读/写变量，只能用于局部函数参数
uniform	一个从客户端代码传递过来的变量，在顶点之间不做改变

有一个变量限定符 `inout` 只能在一个函数中声明一个参数时使用。因为 OpenGL 着色语言并不支持指针(或者引用), 所以这是将一个值传递到一个函数并且允许这个函数修改并返回同一个变量值的唯一方法。

例如, 下面这个函数声明:

```
int CalculateSomething(float fTime, float fStepSize, inout float fVariance);
```

将会返回一个整数(可能是一个通过/失败标记), 但是也可以修改 `fVariance` 变量的值, 并且调用代码也可以从变量中读取新值。在 C/C++ 中, 为了允许对参数进行修改, 我们也可以用指针这种方式声明这个函数。

```
int CalculateSomething(float fTime, float fStepSize, float* fVariance);
```

除非正在对一个多重采样缓冲区进行渲染, 否则 `centroid` 限定符不会起任何作用。在一个单采样缓冲区中, 插值操作总是从像素的中心开始的。对于多重采样, 在使用 `centroid` 限定词时, 插补值将会被选中, 因此它会落到图元和像素中。更多关于多重采样如何工作的细节, 请参见第 9 章。

在默认情况下, 参数将在两个着色器阶段之间以一种透视正确的方法进行插补。我们可以通过 `noperspective` 关键词来指定一个非透视插值, 或者甚至可以通过 `flat` 关键词而不进行插值。我们还可以选择使用 `smooth` 关键词来声明, 这个变量是以一种透视正确的方法进行插补的, 但是这实际上已经是默认设置了。下面列出了一些示例声明。

```
smooth out vec3 vSmoothValue;
flat out vec3 vFlatColor;
noperspective float vLinearlySmoothed;
```

### 6.1.3 真正的着色器

最后, 让我们来看看能够完成一些有用工作的真正着色器。`GLShaderManager` 类有一个存储着色器, 叫做单位着色器。这种着色器不会对几何图形进行转换, 而是使用单一的颜色来绘制图元。可能这看起来有点太简单了, 让我们更进一步展示一下。我们也可以为每个顶点使用不同的颜色值来对一个类似三角形的图元进行着色。程序清单 6.1 展示了顶点着色器, 程序清单 6.2 则展示了片段着色器。

程序清单 6.1 ShadedIdentity 着色器顶点程序

```
// ShadedIdentity 着色器
// 顶点着色器
// Richard S. Wright Jr.
// OpenGL 超级宝典
#version 330
in vec4 vVertex; // 顶点位置属性
in vec4 vColor; // 顶点颜色属性
out vec4 vVaryingColor; // 传递到片段着色器的颜色值
void main(void)
{
    vVaryingColor = vColor; // 简单复制颜色值
    gl_Position = vVertex; // 简单传递顶点位置
}
```

## 程序清单 6.2 ShadedIdentity 着色器片段程序

```
// ShadedIdentity 着色器
// 片段着色器
// Richard S. Wright Jr.
// OpenGL 超级宝典
#version 330
out vec4 vFragColor; // 将要进行光栅化的片段颜色
in vec4 vVaryingColor; // 从顶点阶段得到的颜色
void main(void)
{
    vFragColor = vVaryingColor; // 对片段进行颜色插值
}
```

## OpenGL 着色语言版本

每个着色器的第一行非命令行都是指定版本。

```
#version 330
```

上面这一行就指定了这个着色器要求的 OpenGL 着色语言的最低版本为 3.3。如果 OpenGL 驱动不支持 3.3 版，那么着色器将不会编译。OpenGL 3.2 引入了 GLSL (OpenGL 着色语言) 1.5 版，OpenGL 3.1 引入了 GLSL 1.4 版，而 OpenGL 3.0 则引入了 GLSL 1.3 版。听起来有点困惑？这很正常。在 OpenGL 3.3 中，ARB 决定使 GLSL 版本号与从 3.3 版开始的所有 OpenGL 主版本保持同步。实际上 OpenGL 4.0 规范是与 3.3 版同时发布的，而与 OpenGL 4.0 对应的着色语言版本也是 4.0。在着色器中要求 4.0 版本，如下所示。

```
#version 400
```

如果在 GLTools 源代码中检查存储着色器，那么不会找到这样的版本信息。GLTools 的目的在于在兼容版本中运行并且使用 GLSL 1.1 中更老的约定。实际上 GLTools 在 2.1 版的 OpenGL 驱动中运行得非常好。不要忘记，GLTools 只是为了提供一些“帮助”，以及作为使用 OpenGL 的一个起点而已。

## 属性声明

属性是由 C/C++ 客户端 OpenGL 代码逐个顶点进行指定的。在顶点着色器中，这些属性只是简单地声明为 in。

```
in vec4 vVertex;
in vec4 vColor;
```

在这里，我们声明了两个导入属性，即一个 4 分量顶点位置和一个 4 分量顶点颜色值。示例程序 ShadedTriangle (渲染后的三角形) 特别使用了这个着色器，我们利用 GLBatch 类来设置 3 个顶点位置和 3 个颜色值。关于 GLBatch 类是如何将这 3 个值传递到着色器的相关内容，将稍后介绍。我们可以回忆一下第 3 章的内容，在 GLSL 中每个顶点程序都最多可以有 16 个属性。

同样，每个属性也总是一个 4 分量向量，即使不使用所有的 4 分量时也是如此。

例如，如果我们只是指定了一个 float 值作为参数，那么在内部仍将占据 4 个浮点值的空间。

另外一件要记住的事情是，标记为 `in` 的变量是只读的。对变量名进行重用，在着色器中进行一些中间计算看起来比较聪明，但是如果试图这样做的话，那么驱动中的 GLSL 编译器就会产生一个错误。

## 声明输出

接下来我们将为顶点程序声明一个输出变量，同样是 4 分量浮点向量。

```
out vec4 vVaryingColor;
```

这个变量将成为将要传送到片段着色器的顶点的颜色值。实际上，这个变量必须为片段着色器声明为一个 `in` 变量，否则在我们试图将着色器编译和连接到一起时，就会得到一个连接错误。

当在一个顶点着色器中将一个值声明为 `out`，并在片段着色器中将其声明为 `in` 时，这个片段着色器接受的变量值为一个插补值。在默认情况下，这些工作将以一种正确透视的方式进行，并且在变量之前指定另一个额外的限定符 `smooth`，以确保完成了这些工作。我们还可以指定 `flat` 声明不应进行任何插值，或者指定 `noperspective` 来声明在各个值之间进行直接线性插值 (straight linear interpolation)。当使用 `flat` 时，还有必要进行一些额外的考虑。

## 顶点动作

最后，我们来看一下顶点程序的主体，它在批次中将为每个顶点执行一次。

```
void main(void)
{
    vVaryingColor = vColor;
    gl_Position = vVertex;
}
```

这确实非常简单。我们将输入颜色属性分配给即将发出的插补值，并且未经变换直接将输入的顶点值分配给 `gl_Position`。

变量 `gl_Position` 是一个预定义的内建 4 分量向量，它包含顶点着色器要求的一个输出。输入 `gl_Position` 的值是几何图形阶段用来装配图元的。请记住，既然我们没有进行任何附加的变换，顶点将只映射到所有 3 个坐标范围都在  $\pm 1.0$  之间的笛卡尔坐标上。

## 片段三角形

现在，我们将注意力转移到片段程序上。在渲染一个图元（例如一个三角形）时，一旦 3 个顶点由顶点程序进行了处理，那么它们将组装成一个三角形，而这个三角形将由硬件进行光栅化。图形硬件确定独立片段属于屏幕上（或者更精确地，在颜色缓冲区中）的什么位置，并且为三角形中的每个片段（如果不进行任何多重采样的话则只是一个点）执行片段程序的一个实例。

片段程序的最终输出颜色是一个 4 分量浮点向量，我们如下声明这个向量。

```
out vec4 vFragColor;
```

如果片段程序只有一个输出，那么它在内部将分配为“输出 0”。这是片段着色器的第一个输出，并且将传输到由 `glDrawBuffers` 设置的缓冲区目标，默认情况下为 `GL_BACK`，即黑色缓冲区（对于双重缓冲区环境来说是这样的）。实际颜色缓冲区常常并不包含 4 个浮点分量，这样输出值就会映射到目标缓冲区的范围内。例如，大多数情况下，这可能只是 4 个无符号字节分量（每个分量值都在 0 到 255 之间）。我们还可以使用 `ivec4` 来输出整型值，而这些值也会被映射到颜色缓冲区的范围内。输出除颜色值以外的更多值是可能的，也可以一次性写入多重缓冲区，但是这些内容已经远远超出了本章涉及的范围。

输入片段着色器是经过平滑插值的颜色值，由顶点程序上游传入。这只是作为一个 `in` 变量进行声明的。

```
in vec4 vVaryingColor;
```

最后，片段着色器的主体甚至会比顶点着色器更加简单，它只是将平滑插值的颜色值直接分配给片段颜色。

```
void main(void)
{
    vFragColor = vVaryingColor;
}
```

图 6.2 所示显示了这个着色器运行的最终输出。

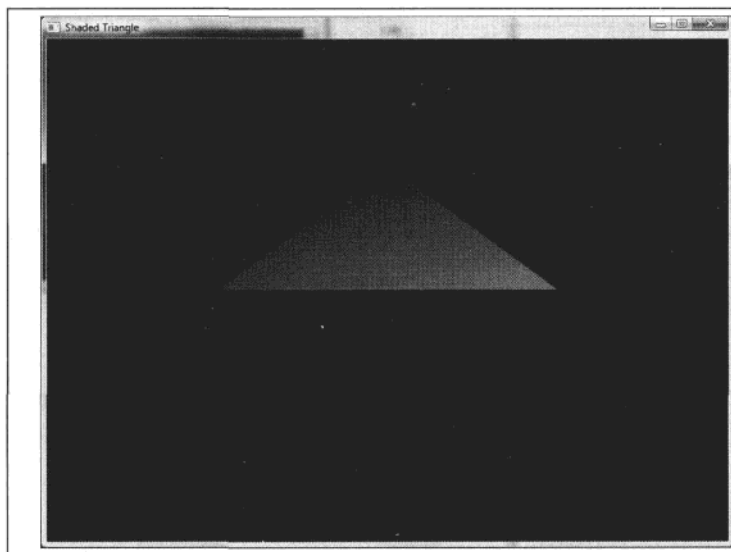


图 6.2

ShadedTriangle 程序的输出

### 6.1.4 编译、绑定和连接

现在我们已经看到了一个简单着色器的使用情况，接下来需要讨论的是在 OpenGL 中一个着色器是如何实际编译和连接以供使用的。着色器源代码被传递给驱动程序，然后进行编译，最后进行连接，就像我们要对所有 C/C++ 程序做的一样。此外，着色器中的属性名需要绑定到由 GLSL 提供的 16 个预分配属性槽中的某一个上。在整个过程中，我们可以检查错误，甚至可以接收从驱动程序传回的关于试图建立着色器时为什么会失败的诊断信息。

OpenGL API 不支持任何类型的文件 I/O 操作。着色器的源代码采用什么样的方式，由程序员根据哪种方式对应用程序有利来进行选择。最简单的方式是将着色器存储在 ASCII 纯文本文件中。这样要使用典型的文件系统函数从磁盘加载文本文件就是一件很简单的事情了。在示例中采用的就是这种方法，而且顶点着色器采用扩展名.vp，而片段着色器则采用扩展名.fp。另一种方式是将文本作为硬编码的字符数组存储在 C/C++ 源代码中。

不过，这样在进行编辑时比较麻烦，而且使程序更加自我封闭（selfcontained），从而使修改着色器和阅读源代码更加麻烦。当然，我们也可以通过算法生成着色器源代码，或者从数据库中恢复，又或者从某种加密数据文件中恢复。这些选择可能在移植一个应用程序时对我们非常有吸引力，但是对于学习或者仅仅是开发调试的目的来说，没有什么比纯文本文件更合适的了。

GLTools 函数 `gltLoadShaderPairWithAttributes` 对于加载和初始化着色器来说确实是真正的“重型升降机”。程序清单 6.3 列出了完整的程序，而且通过这个函数，我们介绍了加载一个着色器的所有必要元素。

程序清单 6.3 `gltLoadShaderPairWithAttributes` 函数

```

////////////////////////////////////
// 加载一对着色器，进行编译并连接到一起
// 为每个着色器指定完整的源文本。
// 在着色器名之后，指定参数数量，然后指定索引和每个参数的参数名。

GLuint gltLoadShaderPairWithAttributes(const char *szVertexProg,
                                       const char *szFragmentProg,.....)
{
    // 临时着色器对象
    GLuint hVertexShader;
    GLuint hFragmentShader;
    GLuint hReturn = 0;
    GLint testVal;
    // 创建着色器对象
    hVertexShader = glCreateShader(GL_VERTEX_SHADER);
    hFragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
    // 加载它们。如果失败则进行清除并返回 null
    // 顶点程序
    if(gltLoadShaderFile(szVertexProg, hVertexShader) == false)
    {
        glDeleteShader(hVertexShader);
        glDeleteShader(hFragmentShader);
        cout << "The shader at " << szVertexProg
              << " could not be found.\n";
        return (GLuint)NULL;
    }
    // 片段程序
    if(gltLoadShaderFile(szFragmentProg, hFragmentShader) == false)
    {
        glDeleteShader(hVertexShader);
        glDeleteShader(hFragmentShader);

        cout << "The shader at " << szFragmentProg
              << " could not be found.\n";
        return (GLuint)NULL;
    }
    // 对两者进行编译
    glCompileShader(hVertexShader);
    glCompileShader(hFragmentShader);
}

```

```

// 在顶点着色器中检查错误
glGetShaderiv(hVertexShader, GL_COMPILE_STATUS, &testVal);
if(testVal == GL_FALSE)
{
    char infoLog[1024];
    glGetShaderInfoLog(hVertexShader, 1024, NULL, infoLog);
    cout << "The shader at " << szVertexProg
    << " failed to compile with the following error:\n"
    << infoLog << "\n";
    glDeleteShader(hVertexShader);
    glDeleteShader(hFragmentShader);
    return (GLuint)NULL;
}

// 在片段着色器中检查错误
glGetShaderiv(hFragmentShader, GL_COMPILE_STATUS, &testVal);
if(testVal == GL_FALSE)
{
    char infoLog[1024];
    glGetShaderInfoLog(hFragmentShader, 1024, NULL, infoLog);
    cout << "The shader at " << hFragmentShader
    << " failed to compile with the following error:\n"
    << infoLog << "\n";
    glDeleteShader(hVertexShader);
    glDeleteShader(hFragmentShader);
    return (GLuint)NULL;
}

// 创建最终的程序对象，并连接着色器
hReturn = glCreateProgram();
glAttachShader(hReturn, hVertexShader);
glAttachShader(hReturn, hFragmentShader);
// 现在，我们需要将参数名绑定到它们指定的参数位置列表上
va_list attributeList;
va_start(attributeList, szFragmentProg);
// 重复迭代这个参数列表
char *szNextArg;
int iArgCount = va_arg(attributeList, int); // 参数数量
for(int i = 0; i < iArgCount; i++)
{
    int index = va_arg(attributeList, int);
    szNextArg = va_arg(attributeList, char*);
    glBindAttribLocation(hReturn, index, szNextArg);
}
va_end(attributeList);
// 尝试连接
glLinkProgram(hReturn);
// 这些都不再需要了
glDeleteShader(hVertexShader);
glDeleteShader(hFragmentShader);
// 确认连接也有效
glGetProgramiv(hReturn, GL_LINK_STATUS, &testVal);
if(testVal == GL_FALSE)
{
    char infoLog[1024];
    glGetProgramInfoLog(hReturn, 1024, NULL, infoLog);
    cout << "The program " << hReturn
    << " failed to link with the following error:\n"
    << infoLog << "\n";
    glDeleteProgram(hReturn);
    return (GLuint)NULL;
}

```

解  
答  
PDG

```
// All done, return our ready to use shader program
return hReturn;
}
```

## 指定属性

函数原型获取了顶点程序文件的名称、片段程序文件的名称和指定属性的参数的变量数量。

```
GLuint gltLoadShaderPairWithAttributes(const char *szVertexProg,
                                       const char *szFragmentProg,.....);
```

如果我们以前没见过接受可变参数列表的函数声明,那么参数列表末尾的 ..... 可能看起来像一个打字错误。另外一些接受可变参数列表的例子还有诸如 printf 或 sprintf 这样的函数。但是,对于这个函数来说,第一个附加的参数是顶点程序包含属性的数量。在这之后是一个对应于第一个属性的基于 0 的索引,以及作为一个字符数组的属性名。然后,只要有必要,属性槽的数量和名称将进行足够多次数的重复。例如,为了加载一个带有顶点位置和表面法线属性的着色器,我们可能像下面这样调用 gltLoadShaderPairWithAttributes。

```
hShader = gltLoadShaderPairWithAttributes("vertexProg.vp",
                                           "fragmentProg.fp", 2, 0, "vVertexPos", 1, "vNormal");
```

对于两个属性位置进行 0 和 1 的选择是任意性的,只要这个值在 0~15 范围之内。我们也可以选择 7 和 13。不过,GLTools 类 GLBatch 和 GLTriangleBatch 则使用一系列一致的属性位置,由 typedef 指定,如下所示。

```
typedef enum GLT_SHADER_ATTRIBUTE { GLT_ATTRIBUTE_VERTEX = 0,
                                     GLT_ATTRIBUTE_COLOR, GLT_ATTRIBUTE_NORMAL,
                                     GLT_ATTRIBUTE_TEXTURE0, GLT_ATTRIBUTE_TEXTURE1,
                                     GLT_ATTRIBUTE_TEXTURE2, GLT_ATTRIBUTE_TEXTURE3,
                                     GLT_ATTRIBUTE_LAST};
```

如果使用这些属性位置标识符,我们就可以开始和 GLShaderManager 类中支持的存储着色器一起使用自己的着色器了。这也意味着,在第 12 章更加详细地学习顶点属性提交相关内容之前,我们可以继续使用 GLBatch 和 GLTriangleBatch 类来提交几何图形。

## 设置源代码

我们的首要任务是创建两个着色器对象,分别对应顶点着色器和片段着色器。

```
hVertexShader = glCreateShader(GL_VERTEX_SHADER);
hFragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
```

然后,可以使用这两个着色器 ID 来加载着色器源代码。我们略去了 gltLoadShaderFile 函数无关紧要的细节,因为这些代码中大部分都是用来以指定的文件名从磁盘中加载着色器文本的。不过,一旦这些任务完成,下面的代码就会将着色器源文件送入着色器对象。还要注意的,我们要进行两次这项工作——一次是为了顶点着色器,另一次是为了片段着色器。

```
GLchar *fsStringPtr[1];
```

```
fsStringPtr[0] = (GLchar *)szShaderSrc;  
glShaderSource(shader, 1, (const GLchar **)fsStringPtr, NULL);
```

szShaderSrc 变量只是一个简单的字符指针，它指向着色器的完整代码，而 shader 则是我们加载的着色器对象的对象 ID。

## 编译着色器

编译着色器是一项简单的一次性工作。

```
glCompileShader(hVertexShader);  
glCompileShader(hFragmentShader);
```

每个 OpenGL 实现都有一个内建的 GLSL 编译器，这个编译器由硬件提供商提供。这就是说，任何指定的提供商都应该最有资格为他们自己的硬件建立编译器。当然，就像 C/C++ 代码一样，很多因素都会阻碍我们的 GLSL 着色器进行编译，例如语法错误、实现的故障等。为了检查失败，我们使用以 GL\_COMPILE\_STATUS 标记为参数的 glGetShader 函数。

```
glGetShaderiv(hVertexShader, GL_COMPILE_STATUS, &testVal);
```

如果返回的 testVal 值为 GL\_FALSE，那就说明源代码编译失败了。如果我们能够在实现中得到的只有简单的通过或者失败信息，那么编写着色器确实会非常困难。在编译失败时，我们可以通过 glGetShaderInfoLog 函数来检查着色器消息日志，看看是什么地方出了问题。在目前的例子中，我们将错误信息显示在控制台，清除着色器对象，并返回 NULL。

```
if(testVal == GL_FALSE)  
{  
    char infoLog[1024];  
    glGetShaderInfoLog(hVertexShader, 1024, NULL, infoLog);  
    cout << "The shader at " << szVertexProg  
    << " failed to compile with the following error:\n"  
    << infoLog << "\n";  
    glDeleteShader(hVertexShader);  
    glDeleteShader(hFragmentShader);  
    return (GLuint)NULL;  
}
```

## 进行连接和绑定

对 GLSL 源代码进行编译之后，我们就完成了一半工作，但是还必须做一些其他工作，才能对它们进行连接。首先，我们要创建最终的着色器程序对象，并将顶点着色器和片段着色器与它绑定到一起。

```
hReturn = glCreateProgram();  
glAttachShader(hReturn, hVertexShader);  
glAttachShader(hReturn, hFragmentShader);
```

现在，着色器已经做好了连接的准备。不过，在连接着色器程序之前，我们还要做一件重要的工作，就是将属性变量名绑定到指定的数字属性位置。函数 glBindAttribLocation 将为我们完成这项工作，它的原型如下所示。

```
void glBindAttribLocation(GLuint shaderProg, GLuint attribLocation,
                          const GLchar *szAttributeName);
```

它接受我们正在讨论的着色器的标识符、将要进行绑定的属性位置和属性变量的名称。例如，在存储着色器中，我们采用了一个约定，即顶点位置属性总是使用变量名 `vVertex`，而属性位置总是使用 `GLT_ATTRIBUTE_VERTEX` 值（值 0）。我们也可以很简单地重复这个过程。

```
glBindAttribLocation(hShader, GLT_ATTRIBUTE_VERTEX, "vVertex");
```

在进行连接之前，必须按照这种方式对属性位置进行连接。在此处代码中，我们遍历了可变参数列表，只需简单地为每个需要进行绑定的属性重复调用这个函数。

```
// 重复迭代这个参数列表
char *szNextArg;
int iArgCount = va_arg(attributeList, int); // 属性数量
for(int i = 0; i < iArgCount; i++)
{
    int index = va_arg(attributeList, int);
    szNextArg = va_arg(attributeList, char*);
    glBindAttribLocation(hReturn, index, szNextArg);
}
va_end(attributeList);
```

## 连接着色器

最后，终于到了对着色器进行连接的时候，在这之后也可以丢弃顶点着色器对象和片段着色器对象。

```
glLinkProgram(hReturn);
// 这些都不再需要了
glDeleteShader(hVertexShader);
glDeleteShader(hFragmentShader);
```

和编译一样，很多原因能够导致连接失败。例如，如果我们在顶点程序中声明一个 `out` 变量，但在片段着色器中声明这个变量；或者我们在片段着色器中声明了这个变量，但是两个声明的类型不同。这样，在返回之前，我们检查一个错误并显示诊断信息，正如在编译时所做的一样。

现在，OpenGL GLSL 着色器已经 100%准备好了。我们已经创建了一个着色器程序，还要说明的是，当不再使用它之后（可能是在程序终止时），需要使用如下函数删除它。

```
void glDeleteProgram(GLuint program);
```

### 6.1.5 使用着色器

要使用 GLSL 着色器，必须使用 `glUseProgram` 函数选定它，如下所示。

```
glUseProgram(myShaderProgram);
```

这样就将着色器设置为活动的，现在顶点着色器和片段着色器会处理所有提交的几何图形。在提交顶点属性之前，要对 Uniform 值和纹理进行设置，稍后我们将对此进行介绍。不过，提交顶点属性是一个很

大的主题——大到实际上足以单独占用一章的篇幅来介绍它，这个主题将在第 12 章进行更详细的介绍。不过现在我们可以使用 GLBatch 和 GLTriangleBatch 类来管理几何图形。

在本章的第一个示例程序 ShadedTriangle 中，我们将三角形加载到 GLBatch 的一个叫做 triangleBatch 的实例中，这个实例使用最简单的坐标系（我们称之为“单位”坐标系）。

```
// 加载一个三角形
GLfloat vVerts[] = { -0.5f, 0.0f, 0.0f,
                     0.5f, 0.0f, 0.0f,
                     0.0f, 0.5f, 0.0f };
GLfloat vColors[] = { 1.0f, 0.0f, 0.0f, 1.0f,
                     0.0f, 1.0f, 0.0f, 1.0f,
                     0.0f, 0.0f, 1.0f, 1.0f };

triangleBatch.Begin(GL_TRIANGLES, 3);
triangleBatch.CopyVertexData3f(vVerts);
triangleBatch.CopyColorData4f(vColors);
triangleBatch.End();

myIdentityShader = gltLoadShaderPairWithAttributes("ShadedIdentity.vp",
                                                  "ShadedIdentity.fp", 2, GLT_ATTRIBUTE_VERTEX, "vVertex",
                                                  GLT_ATTRIBUTE_COLOR, "vColor");
```

我们还为每个顶点设置了不同的颜色，分别为红色、绿色和蓝色。最后，使用前面介绍过的 gltLoadShaderPairWithAttributes 函数加载着色器。请注意我们是如何使两组属性（顶点值和颜色值）与提供给 GLBatch 类的数据组相匹配的。

提交批次来进行渲染现在已经是一项简单的工作了，只要选择着色器并使用 GLBatch 类来提交顶点属性。

```
glUseProgram(myIdentityShader);
triangleBatch.Draw();
```

所有这些努力的成果，即最后渲染过的三角形，如图 6.2 所示。

### 6.1.6 Provoking Vertex

ShadedTriangle 示例很好地演示了如何在顶点之间进行平滑的插值。每个顶点都有一个不同的颜色值，我们在图 6.2 所示的三角形中看到的实际上是颜色空间中由这 3 个颜色坐标所表示的平面颜色值。很酷，是吗？不过，我们还可以从一个着色器阶段传递到下一个着色器阶段的变量设置为 flat。如果有一个值对于整个批次来说都必须是常数，那么最好像第 3 章讨论的那样使用一个 Uniform 值。不过，有时候有一个对整个图元（例如三角形）表面来说是唯一的，但是对每一个三角形都需要进行改变的值，还是非常有用的。在使用统一值时发送大量三角形，每个批次用一个三角形作为示例，这样做将会非常低效。由此引入了 flat 存储标识符，在 ShadedTriangle.vp 着色器中声明了输出平滑着色器颜色值，如下所示。

```
out vec4 vVaryingColor;
```

但是，如果将它声明为 flat（并且不要忘记，片段着色器中相应的 in 变量也必须声明为 flat），就像下面介绍的那样，结果得到的三角形将会是实心蓝色的。

```
flat out vec4 vFlatColor;
```

当一个图元的每个顶点都有一个不同的平面着色变量值时，只有其中一个顶点可以“平面地”应用。默认的约定是使用为图元的最后一个顶点指定的值。在本例中，三角形的 3 个顶点中最后一个使用的颜色值是蓝色。这个约定叫做“provoking vertex”，我们可以用下面的函数来对它从最后一个顶点到第一个顶点进行修改。

```
void glProvokingVertex(GLenum provokeMode);
```

provokeMode 的合法值为 GL\_FIRST\_VERTEX\_CONVENTION 和 GL\_LAST\_VERTEX\_CONVENTION (默认值)。

示例程序 ProvokingVertex 演示了它的使用情况。实际上这是 ShadedTriangle 程序的一个稍加修改的版本。按空格键可以切换这个约定，而三角形则在实心蓝色和实心红色之间来回切换。

## 6.2 着色器统一值

属性是每个顶点位置、表面法和纹理坐标等都需要的，而统一值则用于为整个图元批次向保持不变的（统一（uniform）的）着色器传递数据。对于顶点着色器来说，可能最普遍的统一值就是变换矩阵了。以前我们用内建支持存储着色器和它们所需统一值的 GLShaderManager 类来完成这项工作。现在我们已经开始编写自己的着色器了，这就需要能够设置自己的统一值，而不只是矩阵值了；任何着色器变量都可以指定为一个统一值，而统一值可以在 3 个着色器阶段中的任何一个阶段中（尽管本章我们只讨论顶点着色器和片段着色器）。创建一个统一值非常简单，只需在变量声明开始时放置一个 uniform 关键词。

```
uniform float fTime;
uniform int iIndex;
uniform vec4 vColorValue;
uniform mat4 mvpMatrix;
```

统一值不能被标记为 in 或 out，它们也不能在着色器阶段之间进行插值（虽然可以将它们复制到经过插值的变量中），并且它们总是只读的。

### 6.2.1 寻找统一值

在一个着色器进行编译和连接之后，我们必须在着色器中寻找统一值位置。这项工作可以使用 glGetUniformLocation 函数来完成。

```
GLint glGetUniformLocation(GLuint shaderID, const GLchar* varName);
```

这个函数返回一个有符号的整数，代表在 shaderID 指定的着色器中由 varName 命名的变量的位置。例如，为了获取一个名为 vColorValue 的统一值的位置，我们应该进行如下操作。

```
GLint iLocation = glGetUniformLocation(myShader, "vColorValue");
```

着色器变量名是区分大小写的，如果 `glGetUniformLocation` 的返回值是 -1，就说明统一值的名称在着色器中不能被定位。

我们应当牢记，即使着色器编译正确，如果不是在着色器中直接使用，那么一个统一值名称仍然可能在这个着色器中“消失”。我们不必担心统一变量会被优化掉，但是如果声明了一个统一值而不去使用它，那么编译器将会丢弃它。

## 6.2.2 设置标量和向量统一值

一个单独的标量和向量数据类型可以在 `glUniform` 函数中使用下面的变量进行设置。

```
void glUniform1f(GLint location, GLfloat v0);
void glUniform2f(GLint location, GLfloat v0, GLfloat v1);
void glUniform3f(GLint location, GLfloat v0, GLfloat v1, GLfloat v2);
void glUniform4f(GLint location, GLfloat v0, GLfloat v1, GLfloat v2,
                  GLfloat v3);

void glUniform1i(GLint location, GLint v0);
void glUniform2i(GLint location, GLint v0, GLint v1);
void glUniform3i(GLint location, GLint v0, GLint v1, GLint v2);
void glUniform4i(GLint location, GLint v0, GLint v1, GLint v2, GLint v3);
```

例如，考虑在一个着色器中声明的 4 个变量，如下所示。

```
uniform float fTime;
uniform int iIndex;
uniform vec4 vColorValue;
uniform bool bSomeFlag;
```

为了在着色器中寻找并设置这些值，我们的 C/C++ 代码可能会像下面这样。

```
GLint locTime, locIndex, locColor, locFlag;
locTime = glGetUniformLocation(myShader, "fTime");
locIndex = glGetUniformLocation(myShader, "iIndex");
locColor = glGetUniformLocation(myShader, "vColorValue");
locFlag = glGetUniformLocation(myShader, "bSomeFlag");
.....
.....
glUseProgram(myShader);
glUniform1f(locTime, 45.2f);
glUniform1i(locIndex, 42);
glUniform4f(locColor, 1.0f, 0.0f, 0.0f, 1.0f);
glUniform1i(locFlag, GL_FALSE);
```

请注意，我们是在使用一个整数版本的 `glUniform` 来传递一个 `bool` 值。布尔值也可以作为浮点值进行传递，0.0 代表假，1.0 则代表真。

## 6.2.3 设置统一数组

`glUniform` 函数还接受一个指针，假定指向一个数值数组。

```

void glUniform1fv(GLint location, GLuint count, GLfloat* v);
void glUniform2fv(GLint location, GLuint count, GLfloat* v);
void glUniform3fv(GLint location, GLuint count, GLfloat* v);
void glUniform4fv(GLint location, GLuint count, GLfloat* v);
void glUniform1iv(GLint location, GLuint count, GLint* v);
void glUniform2iv(GLint location, GLuint count, GLint* v);
void glUniform3iv(GLint location, GLuint count, GLint* v);
void glUniform4iv(GLint location, GLuint count, GLint* v);

```

在这里，count 值代表每个含有 x 个分量的数组中有多少个元素，其中 x 是位于函数名末尾的数字。例如，如果我们有一个 4 分量的统一值。

```
uniform vec4 vColor;
```

在 C/C++ 中，我们可以将它表示为一个浮点数组。

```
GLfloat vColor[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
```

但是，这是一个包含 4 个值的单个数组，所以会像下面这样将它传递到着色器中。

```
glUniform4fv(iColorLocation, 1, vColor);
```

另一方面，如果我们在着色器中有一个颜色值数组 uniform vec4 vColors[2];，那么在 C/C++ 中，我们可以像下面这样表示和传递一个数组。

```

GLfloat vColors[2][4] = {{ 1.0f, 1.0f, 1.0f, 1.0f },
                          { 1.0f, 0.0f, 0.0f, 1.0f }};
.....

```

```
glUniform4fv(iColorLocation, 2, vColors);
```

在最简单的情况下，我们可以像下面这样设置一个单个浮点统一值。

```

GLfloat fValue = 45.2f;
glUniform1fv(iLocation, 1, &fValue);

```

## 6.2.4 设置统一矩阵

最后，我们来看一看如何设置一个矩阵统一值。着色器矩阵数据类型只有浮点类型，这样变量就减少了很多。下面的函数分别用来加载一个 2x2、3x3 和 4x4 矩阵。

```

glUniformMatrix2fv(GLint location, GLuint count, GLboolean transpose,
                   const GLfloat *m);
glUniformMatrix3fv(GLint location, GLuint count, GLboolean transpose,
                   const GLfloat *m);
glUniformMatrix4fv(GLint location, GLuint count, GLboolean transpose,
                   const GLfloat *m);

```

变量 count 代表指针参数 m 中存储的矩阵数量（没错，我们可以使用矩阵数组！）。如果矩阵已经按照列优先排序（OpenGL 推荐的方式）进行存储，布尔值标记 transpose 将被设为 GL\_TRUE。将这个值设置为 GL\_FALSE 会导致这个矩阵在复制到着色器中时发生变换。

如果我们正在使用一个采用行优先矩阵布局的矩阵库（例如，Direct3D 就使用行优先排序），那么这一点将非常有用。

## 6.2.5 平面着色器

现在让我们来看一个使用统一值的示例着色器。在存储着色器中，有一个平面着色器，它的功能就是将几何图形进行变换并将其设置为一个单色。它使用的顶点属性只有顶点位置。它还需要使用两个统一值、一个变换矩阵和一个颜色值。

FlatShader 示例程序仅仅是加载一个旋转的花托并将它的颜色设置为蓝色。

我们通过 `glPolygonMode` 函数将它在线框模式下进行渲染，这样就能看到确实存在 3D 几何图形。到目前为止的绝大多数 OpenGL 客户端代码都并不重要，所以我们并不列出整个程序。不过，程序清单 6.4 和程序清单 6.5 列出了完整的着色器程序。

程序清单 6.4 平面着色器顶点程序

```
// 平面着色器
// 顶点着色器
// 作者: Richard S. Wright, Jr.
// OpenGL 超级宝典
#version 330
// 变换矩阵
uniform mat4 mvpMatrix;
// 输入每个顶点
in vec4 vVertex;
void main(void)
{
    // 基本上就是这样，对几何图形进行变换
    gl_Position = mvpMatrix * vVertex;
}
```

程序清单 6.5 平面着色器片段程序

```
// 平面着色器
// 片段着色器
// 作者: Richard S. Wright, Jr.
// OpenGL 超级宝典
#version 130
// 将几何图形设为实心的
uniform vec4 vColorValue;
// 输出片段颜色
out vec4 vFragColor;
void main(void)
{
    gl_FragColor = vColorValue;
}
```

在程序清单 6.4 中列出的定点程序中有一个统一值，即连接变换矩阵。

```
uniform mat4 mvpMatrix;
```

这个着色器执行的唯一一项几何图形处理就是使用 `ModelviewProjection` 矩阵对顶点进行变换。正如我们所看到的，用一个矩阵数据类型乘以一个向量数据类型，在 GLSL 中是非常自然的。

```
gl_Position = mvpMatrix * vVertex;
```

在程序清单 6.5 中列出的片段着色器中，仍然只有一个统一值，即一个将要应用在光栅化片段上的 4

分量颜色值。

```
uniform vec4 vColorValue;
```

在客户端，FlatShader 示例程序加载这两个着色器文件并获得 SetupRC 函数中两个统一值的索引。

```
GLuint flatShader;
GLint locMP;
GLint locColor;
.....
.....
flatShader = glLoadShaderPairWithAttributes("FlatShader.vp", "FlatShader.fp",
                                           1, GLT_ATTRIBUTE_VERTEX, "vVertex");
locMVP = glGetUniformLocation(flatShader, "mvpMatrix");
locColor = glGetUniformLocation(flatShader, "vColorValue");
```

程序清单 6.6 列出了完整的 RenderScene 函数。它只是在合适的位置渲染一个旋转的花托（请记住我们还将多边形模式设置成了 GL\_LINE）。在选择平面着色器之后，集合图形颜色的统一值和变换矩阵的统一值将在花托对象上调用 Draw 函数之前进行设置。最终的输出结果如图 6.3 所示。

程序清单 6.6 使用新的平面着色器

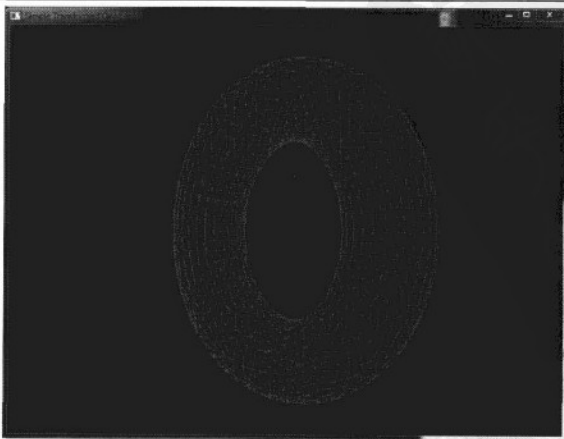
// 进行调用以绘制场景

```
void RenderScene(void)
{
    static CStopWatch rotTimer;
    // 清除窗口和深度缓冲区
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    modelViewMatrix.PushMatrix(viewFrame);
    modelViewMatrix.Rotate(rotTimer.GetElapsedSeconds() * 10.0f,
                          0.0f, 1.0f, 0.0f);

    GLfloat vColor[] = { 0.1f, 0.1f, 1.f, 1.0f };
    glUseProgram(flatShader);
    glUniform4fv(locColor, 1, vColor);
    glUniformMatrix4fv(locMVP, 1, GL_FALSE,
                      transformPipeline.GetModelViewProjectionMatrix());
    torusBatch.Draw();
    modelViewMatrix.PopMatrix();
    glutSwapBuffers();
    glutPostRedisplay();
}
```

图 6.3

FlatShader 程序的输出



## 6.3 内建函数

几乎所有高级编程语言都附带一个标准函数选项。在 C/C++ 中，我们有标准 C 运行时库、标准 I/O 函数等。GLSL 也包含很多有用的内建函数，其中大多数都是在一个标量值或者同时在整个向量上执行数学运算。这些内建函数中有一些是一般性的，另外一些则是选择性的，这是由它们在典型图形渲染算法中的适用性决定的。下面几个表格中列出的函数几乎是原样从 GLSL 语言规范中照搬的。

### 6.3.1 三角函数

表 6.4 列出了 GLSL 支持的三角函数。这些函数是为了 float、vec2、vec3 和 vec4 数据类型而定义的。这里我们用 anyFloat 来表示这 4 种浮点数据类型中的任意一种。

表 6.4 三角函数

函 数	描 述
anyFloat radians(anyFloat degrees)	将角度值转化为弧度值
anyFloat degrees(anyFloat radians)	将弧度值转化为角度值
anyFloat sin(anyFloat angle)	三角正弦
anyFloat cos(anyFloat angle)	三角余弦
anyFloat tan(anyFloat angle)	三角正切
anyFloat asin(anyFloat x)	反正弦
anyFloat acos(anyFloat x)	反余弦
anyFloat atan(anyFloat y, anyFloat x)	$y/x$ 的反正切
anyFloat atan(anyFloat y_over_x)	$y\_over\_x$ 的反正切
anyFloat sinh(anyFloat x)	双曲正弦
anyFloat cosh(anyFloat x)	双曲余弦
anyFloat tanh(anyFloat x)	双曲正切
anyFloat asinh(anyFloat x)	反双曲正弦
anyFloat acosh(anyFloat x)	反双曲余弦
anyFloat atanh(anyFloat x)	反双曲正切

### 6.3.2 指数函数

和三角函数一样，指数函数也是针对浮点数据类型（浮点数和浮点向量）的。表 6.5 列出了完整的指数函数清单。

表 6.5

指数函数

函 数	描 述
anyFloat pow(anyFloat x, anyFloat y)	x 的 y 次方
anyFloat exp(anyFloat x)	x 的自然指数
anyFloat log(anyFloat x)	x 的自然对数
anyFloat exp2(anyFloat x)	2 的 x 次方
anyFloat log2(anyFloat angle)	以 2 为底的 x 的自然对数
anyFloat sqrt(anyFloat x)	x 的平方根
anyFloat inversesqrt(anyFloat x)	x 的逆平方根

### 6.3.3 几何函数

GLSL 中还包括许多通用几何函数。这些函数中有一些采用特殊参数类型（例如叉乘），其他函数则接受任何浮点向量类型（vec2、vec3 和 vec4），这里我们统称为 vec。表 6.6 列出了这些函数。

表 6.6

几何函数

函 数	描 述
float length(vec2/vec3/vec4 x)	返回 x 向量的长度
float distance(vec p0, vec p1)	返回 p0 和 p1 之间的距离
float dot(vec x, vec y)	返回 x 和 y 的点乘结果
vec3 cross(vec3 x, vec3 y)	返回 x 和 y 的叉乘结果
vec normalize(vec x)	返回和 x 方向相同的单位长度向量
vec faceforward(vec N, vec I, vec nRef)	如果 $\text{dot}(\text{nRef}, \text{I}) < 0$ 则返回 N，否则返回 N
vec reflect(vec I, vec N)	返回入射向量 I 的反射方向和表面方向 N
vec refract(vec I, vec N, float eta)	返回入射向量 I 的反射方向、表面方向 N 和折射指数比 eta

### 6.3.4 矩阵函数

许多矩阵操作都是使用常规数学运算符进行的。不过还有一些有用的矩阵函数，表 6.7 列出了这些函数。这些函数中的每一个都是特殊函数，并且接受特殊的参数数据类型。

表 6.7

矩阵函数

函 数	描 述
mat matrixCompMult(mat x, mat y)	逐个分量地将两个矩阵相乘。这与线性代数的矩阵乘法不同
mat2 outerProduct(vec2 c, vec2 r)	返回一个矩阵，这个矩阵是指定的两个向量的外积（叉乘积）
mat3 outerProduct(vec3 c, vec3 r)	

续表

函 数	描 述
mat4 outerProduct(vec4 c, vec4 r)	返回一个矩阵，这个矩阵是指定的两个向量的外积（叉乘积）
mat2x3 outerProduct(vec3 c, vec2 r)	
mat3x2 outerProduct(vec2 c, vec3 r)	
mat2x4 outerProduct(vec4 c, vec2 r)	
mat4x2 outerProduct(vec2 c, vec4 r)	
mat3x4 outerProduct(vec4 c, vec3 r)	
mat4x3 outerProduct(vec3 c, vec4 r)	
mat2 transpose(mat2 m)	返回一个矩阵，这个矩阵是指定矩阵的转置矩阵
mat3 transpose(mat3 m)	
mat4 transpose(mat4 m)	
mat2x3 transpose(mat3x2 m)	
mat3x2 transpose(mat2x3 m)	
mat2x4 transpose(mat4x2 m)	
mat4x2 transpose(mat2x4 m)	
mat3x4 transpose(mat4x3 m)	返回一个矩阵，这个矩阵是指定矩阵的行列式
mat4x3 transpose(mat3x4 m)	
float determinant(mat2 m)	
float determinant(mat3 m)	返回一个矩阵，这个矩阵是指定矩阵的逆矩阵
float determinant(mat4 m)	
mat2 inverse(mat2 m)	
mat3 inverse(mat3 m)	
mat4 inverse(mat4 m)	

### 6.3.5 向量相关函数

标量值可以使用标准比较运算符（<, <=, >, >=, ++, !=）进行比较。

而对于向量之间的比较，就要使用表 6.8 列出的这些函数。所有这些函数都返回一个布尔向量，这些布尔向量都有相同的维数作为参数。

表 6.8

向量相关函数

函 数	描 述
bvec lessThan(vec x, vec y)	逐个分量地返回 $x < y$ 的结果
bvec lessThan(ivec x, ivec y)	
bvec lessThan(uvec x, uvec y)	
bvec lessThanEqual(vec x, vec y)	逐个分量地返回 $x \leq y$ 的结果
bvec lessThanEqual(ivec x, ivec y)	

续表

函 数	描 述
bvec lessThanEqual(uvec x, uvec y)	逐个分量地返回 $x \leq y$ 的结果
bvec greaterThan(vec x, vec y)	逐个分量地返回 $x > y$ 的结果
bvec greaterThan(ivec x, ivec y)	
bvec greaterThan(uvec x, uvec y)	
bvec greaterThanEqual(vec x, vec y)	
bvec greaterThanEqual(ivec x, ivec y)	逐个分量地返回 $x \geq y$ 的结果
bvec greaterThanEqual(uvec x, uvec y)	
bvec equal(vec x, vec y)	
bvec equal(ivec x, ivec y)	逐个分量地返回 $x == y$ 的结果
bvec equal(uvec x, uvec y)	
bvec equal(bvec x, bvec y)	
bvec notEqual(vec x, vec y)	
bvec notEqual(ivec x, ivec y)	逐个分量地返回 $x != y$ 的结果
bvec notEqual(uvec x, uvec y)	
bvec notEqual(bvec x, bvec y)	
bool any(bvec x)	如果 $x$ 的任意分量为真，则返回真
bool all(bvec x)	如果 $x$ 的所有分量都为真，则返回真
bvec not(bvec x)	返回 $x$ 的逐个分量的补集

### 6.3.6 常用函数

最后，我们列出通用函数的列表。所有这些函数都能用于标量和向量数据类型的运算，并且也返回标量和向量数据类型（参见表 6.9）。

表 6.9

矩阵函数 通用函数

函 数	描 述
anyFloat abs(anyFloat x)	返回 $x$ 的绝对值
anyInt abs(anyInt x)	
anyFloat sign(anyFloat x)	返回 1.0 或 -1.0，取决于 $x$
anyInt sign(anyInt x)	
anyFloat floor(anyFloat x)	返回不大于 $x$ 的最小整数
anyFloat trunc(anyFloat x)	返回不大于 $x$ 的最接近的整数
anyFloat round(anyFloat x)	返回最接近 $x$ 的整数的值。如果是小数部分为 0.5 则可能取任意一个方向的整数（根据具体实现而定）
anyFloat roundEven(anyFloat x)	返回最接近 $x$ 的整数的值。如果是小数部分为 0.5 则取最接近的偶数
anyFloat ceil(anyFloat x)	返回大于 $x$ 的最接近它的整数值

续表

函 数	描 述
<code>anyFloat fract(anyFloat x)</code>	返回 x 的小数部分
<code>anyFloat mod(anyFloat x, float y)</code>	返回 x 对 y 取余得到的模数
<code>anyFloat mod(anyFloat x, anyFloat y)</code>	
<code>anyFloat modf(anyFloat x, out anyFloat i)</code>	返回 x 的小数部分，并将 i 设为余下的整数部分的值
<code>anyFloat min(anyFloat x, anyFloat y)</code>	返回 x 和 y 中较小的一个
<code>anyFloat min(anyFloat x, float y)</code>	
<code>anyInt min(anyInt x, anyInt y)</code>	
<code>anyInt min(anyInt x, int y)</code>	
<code>anyUInt min(anyUInt x, anyUInt y)</code>	
<code>anyUInt min(anyUInt x, uint y)</code>	
<code>anyFloat max(anyFloat x, anyFloat y)</code>	返回 x 和 y 中较大的一个
<code>anyFloat max(anyFloat x, float y)</code>	
<code>anyInt max(anyInt x, anyInt y)</code>	
<code>anyInt max(anyInt x, int y)</code>	
<code>anyUInt max(anyUInt x, anyUInt y)</code>	
<code>anyUInt max(anyUInt x, uint y)</code>	
<code>anyFloat clamp(anyFloat x,                   anyFloat minVal,                   anyFloat maxVal)</code>	返回缩放到 minVal 到 maxVal 范围内的 x
<code>anyFloat clamp(anyFloat x,                   float minVal,                   float maxVal);</code>	
<code>anyInt clamp(anyInt x,               anyInt minVal,               anyInt maxVal)</code>	
<code>anyInt clamp(anyInt x,               int minVal,               int maxVal)</code>	
<code>anyUInt clamp(anyUInt x,               anyUInt minVal,               anyUInt maxVal);</code>	
<code>anyUInt clamp(anyUInt x,               uint minVal,               uint maxVal)</code>	

续表

函 数	描 述
anyFloat mix(anyFloat x, anyFloat y, anyFloat a)	返回 x 和 y 的线性混合, a 从 0 到 1 变化
anyFloat mix(anyFloat x, anyFloat y, float a)	
anyFloat mix(anyFloat x, anyFloat y, anyBool a)	在 a 为假时返回 x 的各个分量, 而在 a 为真时返回 y 的各个分量
anyFloat step(anyFloat edge, anyFloat x)	如果 x 小于 edge 则返回 0.0 或 1.0, 否则返回 1.0
anyFloat step(float edge, anyFloat x)	
anyFloat smoothstep(anyFloat edge0, anyFloat edge1, anyFloat x)	如果 $x \leq \text{edge0}$ 则返回 0.0, 如果 $x \geq \text{edge1}$ 则返回 1.0, 如果在两者之间则在 0.0 和 1.0 之间取一个平滑的 Hermite 插值
anyFloat smoothStep(float edge0, float edge1, anyFloat x)	
anyBool isnan(anyFloat x)	Returns true if x is Nan
anyBool isinf(anyFloat x)	如果 x 为正无穷大或负无穷大, 则返回真
anyInt floatBitsToInt(anyFloat x)	将一个浮点值转换成整数值
anyUInt floatBitsToUInt(anyFloat x)	
anyFloat intBitsToFloat(anyInt x)	将一个整数值转换成浮点值
anyFloat uintBitsToFloat(anyUInt x)	

## 6.4 模拟光线

现在我们已经有了使用 GLSL 的良好基础, 可以开始编写一些复杂的着色器了。模拟光线 (Simulating Light) 是计算机图形学的基本技术之一, 这种技术并不是特别复杂, 所以它可以很好地演示着色器编程技术。模拟光线、照明和材质特性本身就值得用一本书的篇幅来讲解, 并且确实已经出现了很多这样的书籍。在这里我们了解一下计算机照明的基础, 并使用 GLSL 来实现它们。这些简单的方法是更多高级技术的基础。

### 6.4.1 简单漫射光

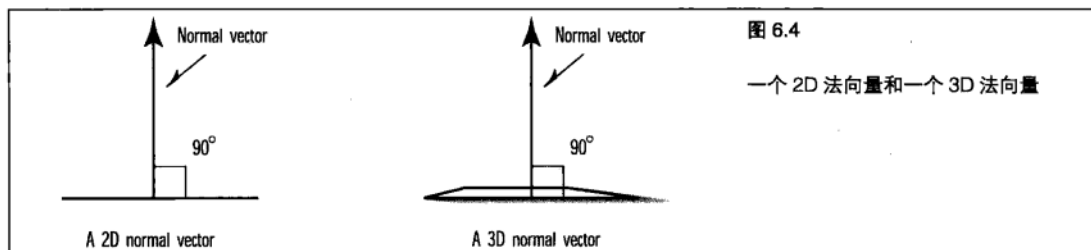
3D 图形中应用最普遍的光线类型是漫射光 (diffuse light)。漫射光是一种经过平面反射的定向光,

其强度与光线在表面上的入射角成正比。这样，如果光线直接射向表面的话，物体的表面就要比光线以一个很大的角度倾斜着射向表面的时候亮度高。实际上这就是很多通过照亮物体表面产生阴影（或者改变颜色）的光照模型的漫射光分量。

要确定一个指定顶点上光线的强度，我们需要两个向量。第一个向量就是光源的方向。某些光照技术只提供指向光源的向量，我们称之为定向光（directional light），因为对于所有顶点来说指向光源的都是同一个向量。这种方式在光源距离被照亮的物体非常远（或者无穷远）是非常适用的。现在我们思考一下阳光照在足球场上所有运动员身上的情景。太阳光照在足球场上不同位置时，它的角度不会有明显的变化。在另一种情况下，如果比赛在晚上进行，那么我们可以明显地观察到一个高架灯在运动员或者其他物体在球场上移动时的照明效果了。如果照明代码提供的是光源的位置，那么我们必须要在着色器中用经过变换的（视觉坐标）光源位置减去顶点位置，来确定指向光源的向量。

## 表面法线

我们在漫射光中（后面我们将看到，实际上不只是在漫射光中）需要的第二个向量是表面法线。经过某个假想平面（或者我们的三角形）上方的顶点，并与这个平面（或三角形）成直角的一条线段，这条线就称为法向量。术语法向量（normal vector）可能听起来像《星际旅行》（Star Trek）中的船员，但是这个词只代表一条垂直于一个真实的或者假想的表面的线。一个向量就是一条指向某个方向的线，“法向”只是一些知识分子表示“垂直”（以  $90^\circ$  角相交）的另一种说法。总之，一个法向量就是一条指向一个与多边形的前面成  $90^\circ$  角方向的线。图 6.4 所示是 2D 和 3D 法向量的示例。



可能读者已经产生了疑问，为什么要为每个顶点指定一个法向量？为什么不能简单地为一个多边形指定一个法向，并且在每个顶点上都使用它？可以这样做，但是，某些时候我们并不希望每个法向都与多边形的表面精确地垂直。读者可能已经注意到，很多表面都不是平的！我们可以将这些表面近似地看成是平的，但是结果会得到一个锯齿状的或者多面的表面。我们可以通过“调整”表面法线使平面多边形表面平滑，从而得到平滑表面的错觉。例如在一个球体上，每个顶点的表面法线都是与球体的实际表面精确垂直的，而不是与用来渲染球体的那些三角形垂直。

## 顶点照明

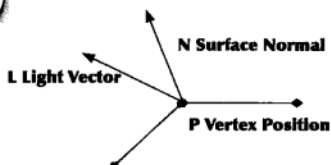
图 6.5 所示为漫射光需要的两个向量。顶点上光的强度通过接受到光源的向量和表面法线的向量点乘积来计算。这两个向量也需要是单位长度的，而点乘积将会返回一个  $+1.0$  到  $-1.0$  之间的值。当表面法线和光照向量指向同一个方向时，将出现一个值为  $1.0$  的点乘积，而当两个向量指向相反的方向时则返回

-1.0。当两个向量互相成  $90^\circ$  时，返回的点乘积值为 0.0。这个+1.0 和 -1.0 之间的值实际上就是这两个向量之间夹角的余弦值。读者可能会猜想，正值意味着这个光线落在顶点上，这个值越大（越接近 1）则光照效果越强，越接近 0（或者小于 0）则光照效果越弱。

图 6.5

漫射光的基本向量

Light Source



我们可以用点乘积的值与顶点的一个颜色值相乘，得到一个基于顶点光线强度的光照颜色值。在顶点之间对这些颜色值进行平滑的着色，有时候被称作顶点照明（vertex lighting），或者背景着色（Gouraud shading）。在 GLSL 中，点乘积的部分非常简单，通常只是如下所示。

```
float intensity = dot(vSurfaceNormal, vLightDirection);
```

## 6.4.2 点光源漫反射着色器

下面让我们了解一下示例程序 DiffuseLight（漫反射光）。这个程序在一个蓝色球体上演示了一个简单的漫反射光线着色器。这个程序使用一个点光源，所以我们还可以看到如何在着色器中确定这些内容的。当然，使用一个定向光源会更加简单，因为我们已经提供了这个向量，但是将它作为一个练习留给读者。程序清单 6.7 展示了 DiffuseLight.vp 顶点着色器的完整代码。

程序清单 6.7 漫反射光线顶点着色器

```
// 简单漫射光着色器
// 顶点着色器
// 作者: Richard S. Wright, Jr.
// OpenGL 超级宝典
#version 330
// 输入每个顶点.....位置和法向
in vec4 vVertex;
in vec3 vNormal;
// 设置每个批次
uniform vec4 diffuseColor;
uniform vec3 vLightPosition;
uniform mat4 mvpMatrix;
uniform mat4 mvMatrix;
uniform mat3 normalMatrix;
// 片段程序颜色
smooth out vec4 vVaryingColor;
void main(void)
{
    // 获取表面法线的视觉坐标
    vec3 vEyeNormal = normalMatrix * vNormal;
    // 获取顶点位置的视觉坐标
    vec4 vPosition4 = mvMatrix * vVertex;
    vec3 vPosition3 = vPosition4.xyz / vPosition4.w;
    // 获取到光源的向量
```

```

vec3 vLightDir = normalize(vLightPosition - vPosition3);
// 从点乘得到漫反射强度
float diff = max(0.0, dot(vEyeNormal, vLightDir));
// 用强度乘以漫反射颜色, 将 alpha 值设为 1.0
vVaryingColor.xyz = diff * diffuseColor.xyz;
vVaryingColor.a = 1.0;
// 不要忘记对多边形进行变换
gl_Position = mvpMatrix * vVertex;
}

```

着色器只指定了两个属性：顶点位置 `vVertex` 和表面法线 `vNormal`。另一方面，这个着色器需要 5 个统一值。

```

uniform vec4 diffuseColor;
uniform vec3 vLightPosition;
uniform mat4 mvpMatrix;
uniform mat4 mvMatrix;
uniform mat3 normalMatrix;

```

`diffuseColor` 包含球体的颜色，`vLightPosition` 是光源位置的视觉坐标，`mpvMatrix` 是模型视图投影矩阵，`mvMatrix` 是模型视图矩阵。我们在使用存储着色器（虽然是在客户端）之前已经了解了这些。新的变化是  $3 \times 3$  的 `normalMatrix`。

典型情况下，表面法线作为一个顶点属性提交。不过，表面法线必须进行旋转以使它的方向在视觉空间之内。我们也不能用它乘以模型视图矩阵来完成这项工作，因为模型视图矩阵也包含一个变换，这个变换在我们进行计算时也会影响向量的方向。取而代之地，我们经常传递一个法向矩阵（`normal matrix`）作为一个统一值，这个值只包含模型视图矩阵的旋转分量。对我们来说非常幸运的是，我们已经开始使用的 `GLTransformationPipeline` 类有一个 `GetNormalMatrix` 函数返回这个值。

这样一来获取法向视觉坐标就是一个简单的矩阵乘法了。

```
vec3 vEyeNormal = normalMatrix * vNormal;
```

在主程序之外，我们还声明了一个平滑的经过着色的颜色值 `vVaryingColor`。

```
smooth out vec4 vVaryingColor;
```

除了对几何图形进行变换之外，这就是顶点着色器唯一的输出了。片段程序变得不太重要了，它只是将这个输入值分配给输出片段颜色而已。

```
vFragColor = vVaryingColor;
```

由于我们传递的是光源位置而不是到光源的方向，必须将顶点位置变换到视觉坐标，并且用光线位置减去它。

```

vec4 vPosition4 = mvMatrix * vVertex;
vec3 vPosition3 = vPosition4.xyz / vPosition4.w;
// 获取到光源的向量
vec3 vLightDir = normalize(vLightPosition - vPosition3);

```

顶点的视觉坐标不能乘以一个包含投影的矩阵，因此我们必须有一个单独的模型视图矩阵供这个着色器使用。

在这里 `w` 坐标应该现身了。在变换矩阵包含任何缩放量的情况下，进行这种除法运算就非常重要了

(回过头来参考第 4 章, 来看这为什么对我们来说是重要的或者是不重要的)。

向量棒极了, 不是吗? 为了使向量指向光源, 只要将这两个向量相减并对结果进行标准化即可。现在我们可以使用点乘来确定这个顶点上光的强度了。还要注意, 我们是如何使用 GLSL 函数 `max` 将强度值限定在 0 和 1 之间的。

```
float diff = max(0.0, dot(vEyeNormal, vLightDir));
```

光照计算的最后一部分就是将表面颜色和光线强度相乘。在本例中, 我们只对 `rgb` 分量进行运算, 而 `alpha` 则不受光照影响。

```
vVaryingColor.rgb = diff * diffuseColor.rgb;
vVaryingColor.a = diffuseColor.a;
```

程序清单 6.8 展示了 `DiffuseLight` 示例程序中的 `SetupRC` 和 `RenderScene` 函数。

程序清单 6.8 `DiffuseLight` 示例程序中的设置和渲染代码

// 这个函数能够在渲染环境中进行任何需要的初始化

```
void SetupRC(void)
{
    // 背景
    glClearColor(0.3f, 0.3f, 0.3f, 1.0f);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);
    shaderManager.InitializeStockShaders();
    viewFrame.MoveForward(4.0f);
    // 创建球体
    gltMakeSphere(sphereBatch, 1.0f, 26, 13);
    diffuseLightShader = gltLoadShaderPairWithAttributes(
        "DiffuseLight.vp", "DiffuseLight.fp", 2,
        GLT_ATTRIBUTE_VERTEX, "vVertex",
        GLT_ATTRIBUTE_NORMAL, "vNormal");
    locColor = glGetUniformLocation(diffuseLightShader, "diffuseColor");
    locLight = glGetUniformLocation(diffuseLightShader, "vLightPosition");
    locMVP = glGetUniformLocation(diffuseLightShader, "mvpMatrix");
    locMV = glGetUniformLocation(diffuseLightShader, "mvMatrix");
    locNM = glGetUniformLocation(diffuseLightShader, "normalMatrix");
}
// 进行调用以绘制场景
void RenderScene(void)
{
    static CStopWatch rotTimer;
    // 清除窗口和深度缓冲区
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    modelViewMatrix.PushMatrix(viewFrame);
    modelViewMatrix.Rotate(rotTimer.GetElapsedSeconds() * 10.0f,
        0.0f, 1.0f, 0.0f);

    GLfloat vEyeLight[] = { -100.0f, 100.0f, 100.0f };
    GLfloat vDiffuseColor[] = { 0.0f, 0.0f, 1.0f, 1.0f };
    glUseProgram(diffuseLightShader);
    glUniform4fv(locColor, 1, vDiffuseColor);
    glUniform3fv(locLight, 1, vEyeLight);
    glUniformMatrix4fv(locMVP, 1, GL_FALSE,
        transformPipeline.GetModelViewProjectionMatrix());
    glUniformMatrix4fv(locMV, 1, GL_FALSE,
        transformPipeline.GetModelViewMatrix());
    glUniformMatrix3fv(locNM, 1, GL_FALSE,
```

```
transformPipeline.GetNormalMatrix();  
sphereBatch.Draw();  
modelViewMatrix.PopMatrix();  
glutSwapBuffers();  
glutPostRedisplay();  
}
```

这是我们第一次大量使用非存储着色器，可以看到对 `glUniform` 函数的 5 次独立的调用，这个函数用来在渲染函数中设置这个着色器。

有一个经常出现的错误（尤其是在习惯了老式固定管线的程序员身上）是，在设置着色器统一值之后和对几何图形进行渲染之前作进一步的修改。请记住，`glUniform` 函数并不会将一个对这些数据的引用复制到着色器；这个函数会将实际数据复制到着色器中。这也提供了一个机会，可以清除一些不经常改变的用于统一值的程序调用。经过漫反射着色的球体最终输出结果如图 6.6 所示。

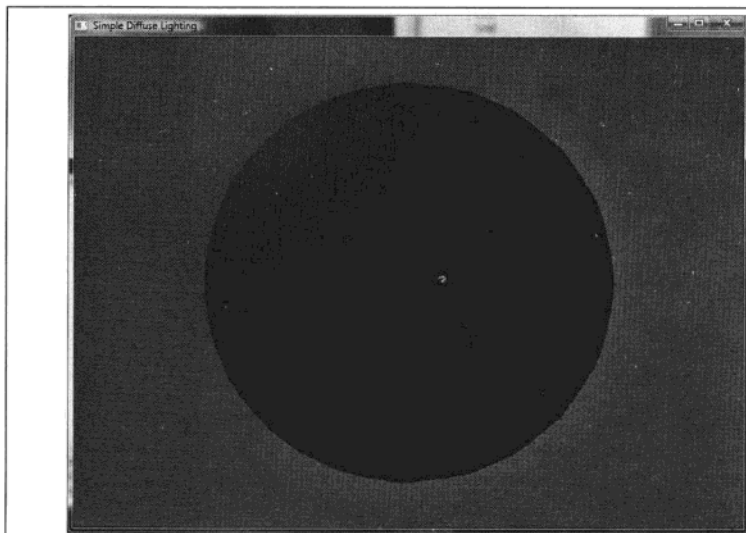


图 6.6

DiffuseLight 示例程序

### 6.4.3 ADS 光照模型

ADS 光照模型是一种最常见的光照模型，尤其对于那些熟悉现在已经“不推荐”的固定功能管线的程序员来说。ADS 代表环境光（Ambient）、漫射光（Diffuse）和镜面光（Specular）。它遵循一个简单的原则，即物体有 3 种材质属性，环境光反射、漫反射和镜面反射。这些属性都是分配的颜色值，更明亮的颜色代表更高的反射量。光源也有这 3 种相同的属性，同样也是分配的颜色值，表示光源的亮度。这样，最终的顶点颜色值就是光照和材质的这 3 个属性互相影响的总和。

#### 环境光

环境光并不来自任何特定的方向。它来自某个光源，但光线却是在房间或场景中四处反射，没有方向可言。

由环境光所照射的物体在所有方向的表面都是均匀照亮的。我们可以把环境光看成是应用到每个光源的全局“照明”因子。这种光照分量确实非常接近环境中源自光源的散射光。

为了计算一个环境光源对最终顶点颜色的影响，环境光材质的性质由环境光的值来度量（就是将这两个颜色值相乘），这个值产生对环境颜色的影响。在 GLSL 着色器中，我们应该像下面这样编写代码。

```
uniform vec3 vAmbientMaterial;
uniform vec3 vAmbientLight;
vec3 vAmbientColor = vAmbientMaterial * vAmbientLight;
```

## 漫射光

漫射光是光源的定向分量，也是我们前面的示例光照着色器的主题。在 ADS 光照模式下，漫反射材质和光照值相乘，就像环境光分量一样。不过，这个值随后将由表面法线和光照向量的点乘积（漫反射强度）进行缩放。在着色器中，代码应该如下所示。

```
uniform vec3 vDiffuseMaterial;
uniform vec3 vDiffuseLight;
float fDotProduct = max(0.0, dot(vNormal, vLightDir));
vec3 vDiffuseColor = vDiffuseMaterial * vDiffuseLight * fDotProduct;
```

请注意，我们并不是简单地接受这两个向量的点乘积，而是还要使用 GLSL 函数 max。点乘积也可能是一个负数，而我们确实不能接受负的光线值或颜色值。任何小于 0 的值都会用 0 代替。

## 镜面光

和散射光一样，镜面光也具有很强的方向性，但它的反射角度很锐利，只沿一个特定的方向反射。高强度的镜面光（实际上是现实世界的材料属性）趋向于在它所照射的表面上形成一个亮点，称为镜面亮点。由于它的高度方向性本质，根据观察者位置的不同，镜面光甚至有可能看不到。聚光灯和太阳都是产生很强的镜面光的例子，不过它们当然必须是照射在一个“光亮的”物体上才行。

对镜面材质和光照颜色在颜色上的影响由一个值来进行缩放，这个值需要更多的计算，到目前为止进行过的计算是不够的。首先我们必须找到被表面法线反射的向量和反向的光线向量。随后这两个向量的点乘积将取“反光度”（shininess）次幂。反光度数值越大，结果得到镜面反射的高亮区越小。如下所示的着色器框架代码可以完成这项工作。

```
uniform vec3 vSpecularMaterial;
uniform vec3 vSpecularLight;
float shininess = 128.0;
vec3 vReflection = reflect(-vLightDir, vEyeNormal);
float EyeReflectionAngle = max(0.0, dot(vEyeNormal, vReflection));
fSpec = pow(EyeReflectionAngle, shininess);
vec3 vSpecularColor = vSpecularLight * vSpecularMaterial * fSpec;
```

和其他参数一样，反光度参数也可以是统一值。传统上（从固定管线时代开始），最高的镜面指数（specular power）被设置为 128，大于这个数字的值其效果将逐渐减弱。

## ADS 着色器

在我们前 3 个例子的基础上，顶点最终的颜色可以像下面这样进行计算。

```
vVertexColor = vAmbientColor + vDiffuseColor + vSpecularColor;
```

示例程序 AD SGouraud 就是这样一个着色器，不过我们进行了一些简化。我们仅仅为环境光、漫射光和镜面材料传递了一个单独的颜色值，而并没有传递独立的材质和光照颜色/强度。我们可以将它看作将材质性质左乘光照颜色。如果我们不是为每个顶点改变材质特性，那么这样做就可以提供一种简单的优化。示例程序名称中加上“Gouraud”部分是因为我们逐个顶点地计算了光照值，然后为了进行着色而在顶点之间使用了颜色空间插值。程序清单 6.9 列出了完整的顶点着色器。

程序清单 6.9 AD SGouraud 着色器顶点程序

```
// ADS 点光照着色器
// 顶点着色器
// 作者: Richard S. Wright, Jr.
// OpenGL 超级宝典
#version 330
// 输入每个顶点.....位置和法向
in vec4 vVertex;
in vec3 vNormal;
// 设置每个批次
uniform vec4 ambientColor;
uniform vec4 diffuseColor;
uniform vec4 specularColor;
uniform vec3 vLightPosition;
uniform mat4 mvpMatrix;
uniform mat4 mvMatrix;
uniform mat3 normalMatrix;
// 片段程序颜色
smooth out vec4 vVaryingColor;
void main(void)
{
    // 获取表面法线的视觉坐标
    vec3 vEyeNormal = normalMatrix * vNormal;
    // 获取顶点位置的视觉坐标
    vec4 vPosition4 = mvMatrix * vVertex;
    vec3 vPosition3 = vPosition4.xyz / vPosition4.w;
    // 获取到光源的向量
    vec3 vLightDir = normalize(vLightPosition - vPosition3);
    // 从点乘得到漫反射强度
    float diff = max(0.0, dot(vEyeNormal, vLightDir));
    // 用强度乘以漫反射颜色，将 alpha 值设为 1.0
    vVaryingColor = diff * diffuseColor;
    // 添加环境光
    vVaryingColor += ambientColor;
    // 镜面光
    vec3 vReflection = normalize(reflect(-vLightDir, vEyeNormal));
    float spec = max(0.0, dot(vEyeNormal, vReflection));
    if(diff != 0) {
        float fSpec = pow(spec, 128.0);
        vVaryingColor.rgb += vec3(fSpec, fSpec, fSpec);
    }
    // 不要忘记对多边形进行变换
    gl_Position = mvpMatrix * vVertex;
}
```

我们并不打算列出整个片段着色器，因为它所做的所有工作只是将输入的 `vVaryingColor` 分配给片段颜色而已。

```
vFragColor = vVaryingColor;
```

对于一个给定的三角形来说，只有 3 个顶点，另外还有许多片段填充这个三角形。这使得顶点光照和 Gouraud 着色非常高效，因为所有计算对每个顶点只进行一次。图 6.7 所示显示了 AD SGouraud 示例程序的输出。

图 6.7

基于每个顶点的光照  
(Gouraud 着色)



#### 6.4.4 Phong 着色

图 6.7 所示已经清楚地呈现了 Gouraud 着色的一个缺点。请注意镜面高亮部分的星光模式。在一个静止的图片上，这种形式还可以当作一种有意为之的艺术效果勉强过关。但是，在活动的示例程序中，旋转这个球体时显示一种特有的闪烁则有些烦人，并且一般说来是不可取的。这种现象是由三角形之间的不连续造成的，这种不连续则是由于颜色值在空间中进行的是线性插值而导致的。这些亮线实际上是相互独立的三角形之间的缝隙。有一种方法可以减弱这种现象，即在几何图形上使用更多的顶点。另一种更高品质的方法叫做 Phong 着色。在 Phong 着色时，我们并不在顶点之间进行颜色值插值，而是在顶点之间进行表面法线插值。图 6.8 所示为 AD SPhong 示例程序的输出（在彩图 5 中并列显示了图 6.7 和图 6.8）。

当然有一点我们需要权衡，就是现在在片段程序中所做的工作量大大提高，因为片段着色器的执行次数将比顶点程序的执行次数多得多。

基本的代码和 AD SGouraud 示例程序相同，但是这一次我们要对着色器代码进行较大的调整。程序清单 6.10 展示了新的顶点程序。



图 6.8

基于每个像素的光照  
(Phong 着色)

#### 程序清单 6.10 ADSPhong 顶点着色器

```
// ADS 点光照着色器
// 顶点着色器
// 作者: Richard S. Wright, Jr.
// OpenGL 超级宝典
#version 330
// 输入每个顶点.....位置和法向
in vec4 vVertex;
in vec3 vNormal;
uniform mat4 mvpMatrix;
uniform mat4 mvMatrix;
uniform mat3 normalMatrix;
uniform vec3 vLightPosition;
// 片段程序颜色
smooth out vec3 vVaryingNormal;
smooth out vec3 vVaryingLightDir;
void main(void)
{
    // 获取表面法线的视觉坐标
    vVaryingNormal = normalMatrix * vNormal;
    // 获取顶点位置的视觉坐标
    vec4 vPosition4 = mvMatrix * vVertex;
    vec3 vPosition3 = vPosition4.xyz / vPosition4.w;
    // 获取到光源的向量
    vVaryingLightDir = normalize(vLightPosition - vPosition3);
    // 不要忘记对多边形进行变换
    gl_Position = mvpMatrix * vVertex;
}
```

所有光照计算都是基于表面法线和光线方向向量进行的。

我们并不会为每一个顶点都传递一个经过计算的颜色值，取而代之的是传递两个向量。

```
smooth out vec3 vVaryingNormal;
smooth out vec3 vVaryingLightDir;
```

现在片段着色器要做的工作比以前多得多了，这个着色器如图 6.11 所示。

程序清单 6.11 ADSPong 片段着色器

```
// ADS 点光照着色器
// 片段着色器
// 作者: Richard S. Wright, Jr.
// OpenGL 超级宝典
#version 330
out vec4 vFragColor;
uniform vec4 ambientColor;
uniform vec4 diffuseColor;
uniform vec4 specularColor;
in vec3 vVaryingNormal;
in vec3 vVaryingLightDir;
void main(void)
{
    // 从点乘得到漫反射强度
    float diff = max(0.0, dot(normalize(vVaryingNormal),
                             normalize(vVaryingLightDir)));
    // 用强度乘以漫反射颜色，将 alpha 值设为 1.0
    vFragColor = diff * diffuseColor;
    // 添加环境光
    vFragColor += ambientColor;
    // 镜面光
    vec3 vReflection = normalize(reflect(-normalize(vVaryingLightDir),
                                         normalize(vVaryingNormal)));
    float spec = max(0.0, dot(normalize(vVaryingNormal), vReflection));
    // 如果漫射光为 0，那么就不必考虑指数函数了
    if(diff != 0) {
        float fSpec = pow(spec, 128.0);
        vFragColor.rgb += vec3(fSpec, fSpec, fSpec);
    }
}
```

在如今的硬件平台上，经常会使用类似 Phong 着色这样的高质量渲染选项。这些渲染的视觉质量非常高，而性能则常常会受到一些影响。在性能较低的硬件（例如嵌入式设备）上，或者在已经选择了很多其他开销很大的选项的场景中，Gouraud 着色仍然是最好的选择。一个着色器性能优化的常规原则是，将尽可能多的处理过程移出片段着色器而放入顶点着色器。在这个例子中，我们可以了解到这样做的原因。

## 6.5 访问纹理

从着色器访问纹理贴图是非常简单的。纹理坐标将作为属性传递到我们的顶点着色器。在片段着色器中，这些属性通常是在顶点之间进行平滑插值的。片段着色器使用这些插值纹理坐标来对纹理进行采样（sample）。当前绑定的纹理对象已经针对 Mip 贴图/非 Mip 贴图、过滤模式和环绕模式等进行了设置。经过采样和过滤的纹理颜色将作为 RGBA 颜色值返回，我们可以直接将它写入片段，或者也可以将它与其他颜色计算相结合。在第 7 章，我们将更加深入地学习在 GLSL 中和 GLSL 之外使用纹理。而就目前来说，我们至少要先了解一些基础知识。

### 6.5.1 只有纹理单元

TexturedTriangle 示例程序展示了使用纹理的最简单的着色器。它的目标很简单，即绘制一个三角形并给它添加一个纹理。图 6.9 所示可以看到它的效果。

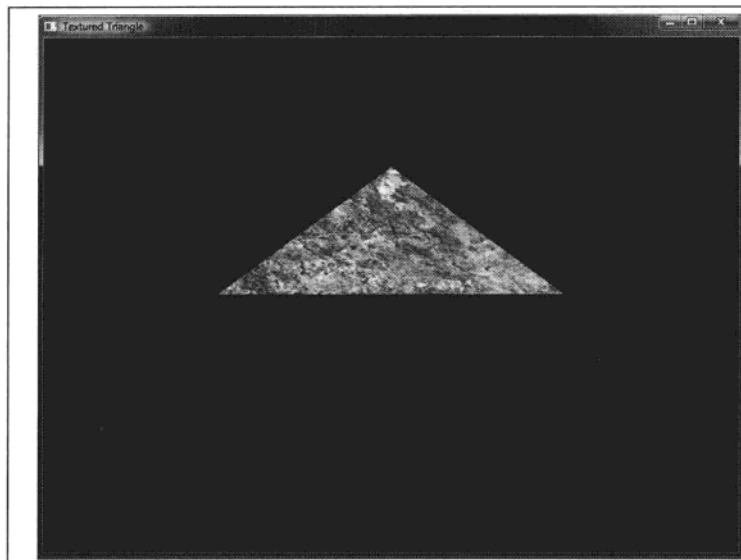


图 6.9

Textured Triangle (纹理三角形)  
示例程序

在客户端，渲染三角形的 C/C++ 代码非常普通，而设置一个三角形纹理坐标的相关工作在学习存储着色器时都已经做过了。程序清单 6.12 列出了接受顶点属性的顶点程序。

程序清单 6.12 TexturedTriangle 顶点程序

```
// TexturedIdentity 着色器
// 顶点着色器
// 作者: Richard S. Wright, Jr.
// OpenGL 超级宝典
#version 330
in vec4 vVertex;
in vec2 vTexCoords;
smooth out vec2 vVaryingTexCoords;
void main(void)
{
    vVaryingTexCoords = vTexCoords;
    gl_Position = vVertex;
}
```

这个顶点程序非常简短，它的核心部分就是包含这个顶点的 s 和 t 纹理坐标的输入顶点属性 vTexCoords，以及输出变量 vVaryingTexCoords。用纹理坐标在三角形表面进行插值所需要的就是这些了。

程序清单 6.13 列出的片段程序也很简短，它包含一些我们到目前还没有介绍的新内容。

程序清单 6.13 TexturedTriangle 片段程序

```
// TexturedIdentity 着色器
// 片段着色器
```

```
// 作者: Richard S. Wright, Jr.
// OpenGL 超级宝典
#version 330
uniform sampler2D colorMap;
out vec4 vFragColor;
in vec4 vVaryingTexCoords;
void main(void)
{
    vFragColor = texture(colorMap, vVaryingTexCoords.st);
}
```

在程序的顶部附近，出现了一个新的变量类型 sampler2D。

```
uniform sampler2D colorMap;
```

一个采样器 (sampler) 实际上就是一个整数 (我们使用 glUniform1i 来设置它的值)，它代表我们将要采样的纹理所绑定的纹理单元。sampler2D 中的“2D”表明这是一个 2D 纹理，我们也可以使用 1D、3D 或者其他类型的采样器 (这些采样器在下一章都会介绍)。就目前来说，我们总是将这个值设为 0，来指示纹理单元 0。在第 5 章，我们介绍了用纹理对象作为一种管理任何数量的不同纹理状态的手段，并且使用了 glBindTexture 函数在不同的纹理对象之间进行选择。

所有这些纹理绑定实际上都是绑定到默认纹理单元——纹理单元 0 的。实际上有很多纹理单元，而且每个纹理单元都有自己的纹理对象与它进行绑定。同时使用一个以上的纹理可以实现许多非常酷的效果，这是一种强大的技术，在下一章我们将进行介绍。

设置采样器统一值并在客户端代码中对三角形进行渲染是非常简单的。

```
glUseProgram(myTexturedIdentityShader);
glBindTexture(GL_TEXTURE_2D, textureID);
GLint iTextureUniform = glGetUniformLocation(myTexturedIdentityShader,
                                                "colorMap");

glUniform1i(iTextureUniform, 0);
triangleBatch.Draw();
```

在这个着色器中，我们调用纹理贴图内建函数 texture 来使用插值纹理坐标对纹理进行采样，并将颜色值直接分配给片段颜色。

```
vFragColor = texture(colorMap, vVaryingTexCoords.st);
```

## 6.5.2 照亮纹理单元

现在我们了解了如何对一个纹理进行采样，让我们用这些经过过滤的纹理单元值做一些更有趣的事情。例如，在 ADSP Phong 着色器中添加一个纹理。

在所有的光照着色器中，我们实际上是将基本色值和光线的强度相乘，这可以是逐个顶点进行的，也可以是逐个像素进行的。我们将修改过的 ADSP Phong 着色器称为 ADSTexture，它会对纹理进行采样，然后用纹理颜色值乘以光线强度。示例程序 LitTexture 的输出结果如图 6.10 所示。我们要特别注意球体左上角良好的镜面高光。

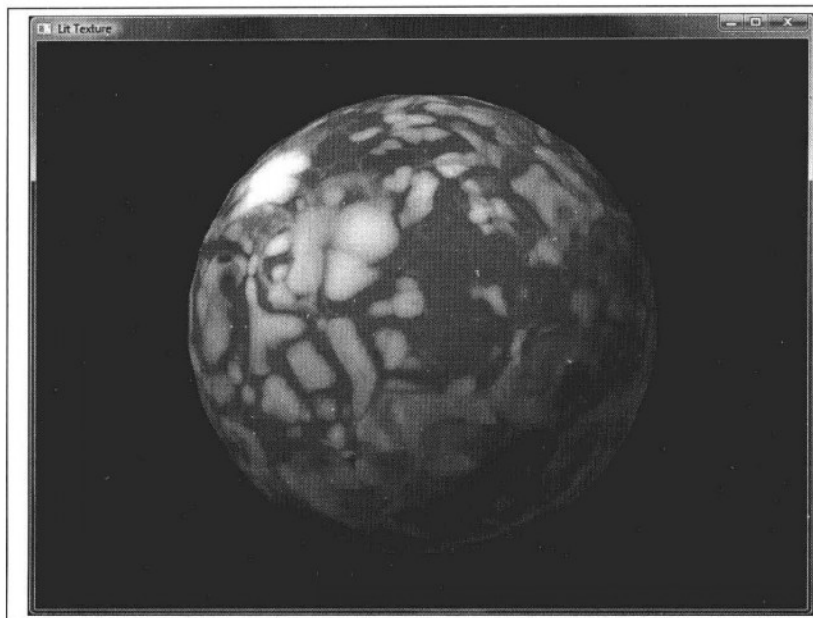


图 6.10

在 LitTexture 中将光线和纹理进行结合

这个白色镜面高光提示了一个我们在进行纹理表面光照时必须特别考虑的问题。环境光和漫射光的总和有可能会很亮，以至于看起来就像纯白色，而纯白色在颜色空间中都为 1。将一个纹理颜色和白色相乘，只会得到和原来一样明亮的纹理颜色值。这就意味着，用一个纹理颜色乘以一个有效的光照值来表现一个白色镜面上的高光是不可能的——至少看起来是这样的。

事实上，我们的光照计算（包括镜面高光）的结果，每个颜色通道的值经常是稍微大于 1.0 的。这意味着，至少使颜色过饱和，以及获得一个白色镜面高光是不可能的。不过，正确的方法应该是将环境光和漫射光强度的和与纹理颜色相乘，然后再加上镜面光部分。程序清单 6.14 展示了我们是如何修改 ADSPHONG 片段着色器来完成这项工作的。

程序清单 6.14 ADSTexture 片段程序

```
// ADS 点光照着色器
// 片段着色器
// 作者: Richard S. Wright, Jr.
// OpenGL 超级宝典
#version 330
out vec4 vFragColor;
uniform vec4 ambientColor;
uniform vec4 diffuseColor;
uniform vec4 specularColor;
uniform sampler2D colorMap;
smooth in vec3 vVaryingNormal;
smooth in vec3 vVaryingLightDir;
smooth in vec2 vTexCoords;
void main(void)
{
    // 从点乘积得到漫反射强度
    float diff = max(0.0, dot(normalize(vVaryingNormal),
                             normalize(vVaryingLightDir)));
    // 用强度乘以漫反射颜色，将 alpha 值设为 1.0
```

```

vFragColor = diff * diffuseColor;
// 添加环境光
vFragColor += ambientColor;
// Modulate in the texture
vFragColor *= texture(colorMap, vTexCoords);
// 镜面光
vec3 vReflection = normalize(reflect(-normalize(vVaryingLightDir),
                                   normalize(vVaryingNormal)));
float spec = max(0.0, dot(normalize(vVaryingNormal), vReflection));
if(diff != 0) {
    float fSpec = pow(spec, 128.0);
    vFragColor.rgb += vec3(fSpec, fSpec, fSpec);
}
}

```

### 6.5.3 丢弃片段

片段着色器设有取消处理过程而不写入任何片段颜色（或者是深度、模板等）值的选项。声明 `discard` 只会使片段程序停止运行，这个声明的一个常规用途就是执行 `alpha` 测试。普通的混合操作需要从颜色缓冲区进行一次读取、两次乘法（至少）、对颜色进行一次求和，然后将得到的值写回颜色缓冲区。如果 `alpha` 为 0，或者非常接近 0，那么片段实际上是不可见的。绘制不可见的东西在性能上实在是糟糕的选择！更不用说这样会在深度缓冲区创建一个不可见的模式，从而导致深度测试异常了。`alpha` 测试只是检查一些阈值，并且在 `alpha` 值低于这个值时完全丢弃这个片段。例如，要测试一个 `alpha` 值是不是小于 0.1，我们可以如下操作。

```

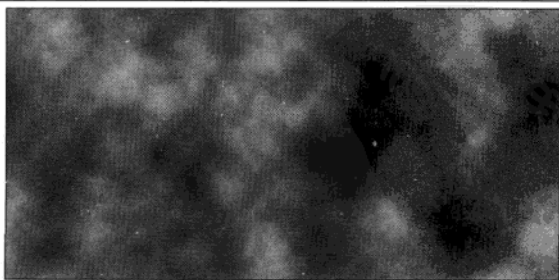
if(vColorValue.a < 0.1f)
    discard;

```

有一种很酷、很生动的效果会使用这种特性，这就是侵蚀着色器（erosion shader）。侵蚀着色器可以使几何图形看起来像是经过了岁月的侵蚀。使用 `discard` 声明，我们可以逐个像素地控制哪个片段会进行绘制，哪个片段不会。示例程序 `Dissolve` 就能实现这样一种效果。让我们从一个带有适当噪点或者云模式的纹理开始。这些纹理可以在大多数照片编辑软件包中很容易地创建出来。在这个示例中，我们使用如图 6.11 所示的纹理。

图 6.11

我们用来实现侵蚀效果的云纹理



在客户端代码中，我们设置了一个基于时间的 Uniform 值，它的取值范围为 1.0 到 0.0，覆盖 10 秒的范围。我们的目标是让对象（一个绿色花托）在 10 秒范围内“消散”。我们通过对云纹理进行采样，并将一个颜色分量与倒计时变量进行比较，当颜色值大于阈值时则完全丢弃片段。程序清单 6.15 展示了

这个片段着色器的完整代码。

程序清单 6.15 Dissolve 片段程序

```
// 带有侵蚀效果多的 ADS 点光照着色器
// 片段着色器
// 作者: Richard S. Wright, Jr.
// OpenGL 超级宝典
#version 330
out vec4 vFragColor;
uniform vec4 ambientColor;
uniform vec4 diffuseColor;
uniform vec4 specularColor;
uniform sampler2D cloudTexture;
uniform float dissolveFactor;
smooth in vec3 vVaryingNormal;
smooth in vec3 vVaryingLightDir;
smooth in vec2 vVaryingTexCoord;
void main(void)
{
    vec4 vCloudSample = texture2D(cloudTexture, vVaryingTexCoord);
    if(vCloudSample.r < dissolveFactor)
        discard;
    // 从点乘积得到漫反射强度
    float diff = max(0.0, dot(normalize(vVaryingNormal),
                             normalize(vVaryingLightDir)));
    // 用强度乘以漫反射颜色, 将 alpha 值设为 1.0
    vFragColor = diff * diffuseColor;
    // 添加环境光
    vFragColor += ambientColor;
    // 镜面光
    vec3 vReflection = normalize(reflect(-normalize(vVaryingLightDir),
                                           normalize(vVaryingNormal)));
    float spec = max(0.0, dot(normalize(vVaryingNormal), vReflection));
    if(diff != 0) {
        float fSpec = pow(spec, 128.0);
        vFragColor.rgb += vec3(fSpec, fSpec, fSpec);
    }
}
```

实际上这还是 ADSPhong 光照片段程序的一个修改版本。这种消散的效果只是简单地加进这个着色器。首先, 我们需要为采样器和倒计时器提供统一值。

```
uniform sampler2D cloudTexture;
uniform float dissolveFactor;
```

然后对纹理进行取样, 并确定红色值(考虑到这是一个灰度图像, 我们随机选择颜色值)是否低于倒计时值, 最终完全丢弃这个片段。

```
vec4 vCloudSample = texture(cloudTexture, vVaryingTexCoord);
if(vCloudSample.r < dissolveFactor)
    discard;
```

还要注意我们在片段着色器中很早就进行这项工作。如果这个片段不会被绘制, 那么执行这种开销很大的计算过程就毫无意义的。示例程序 Dissolve is 的输出结果(至少是整个动画的一个帧)如图 6.12 所示。

图 6.12

Dissolve 示例程序的输出



### 6.5.4 卡通着色 (Cell Shading) ——将纹理单元作为光线

在这一章和上一章所有的纹理贴图示例中，使用的都是 2D 纹理。一般来说，二维纹理是最简单和最容易理解的。

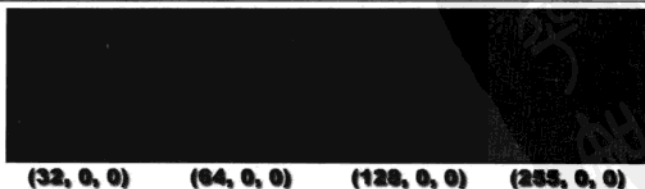
很多人都能够直觉地想象将一个 2D 图片放到 2D 或者 3D 几何图形的一个面上的过程。下面让我们了解一下一维纹理贴图示例，它通常应用在计算机游戏中对一个几何图形进行渲染，使其在屏幕上看起来像一个动画片。

卡通着色 (英文为 Toon shading, 也经常称为 cell shading) 将一个一维纹理贴图作为查询表, 使用纹理贴图图中的纯色 (使用 GL\_NEAREST) 填充几何图形。

基本的思路是, 使用漫射光照强度 (视觉空间表面法线和光线方向向量的点乘积) 作为纹理坐标添加到一个包含逐渐变亮颜色表的一维纹理中。图 6.13 所示就是一个这样的纹理, 包含 4 个逐渐变亮的红色纹理单元 (定义为 RGBunsigned byte 颜色分量)。

图 6.13

一个一维颜色查询表



回想一下, 漫射光点乘积的值是从没有强度的 0.0 到最高强度的 1.0 之间变化的。方便之处在于, 这种方式可以很好地映射到一维纹理坐标范围。

加载这个一维纹理非常简单，如下所示。

```
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_1D, texture);
GLubyte textureData[4][3] = { 32, 0, 0,
                               64, 0, 0,
                               128, 0, 0,
                               255, 0, 0};
glTexImage1D(GL_TEXTURE_1D, 0, GL_RGB, 4, 0, GL_RGB,
              GL_UNSIGNED_BYTE, textureData);
glTexParameterf(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameterf(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameterf(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
```

上面的代码是从 ToonShader 示例程序中摘录的，这个示例程序渲染了一个旋转的花托，并应用了卡通着色效果。虽然用来创建花托的 GLTriangleBatch 提供了一系列二维纹理坐标，但在顶点程序中还是忽略了它们，程序清单 6.16 展示了这个顶点程序。

程序清单 6.16 卡通顶点程序

```
// 动画光照着色器
// 顶点着色器
// 作者: Richard S. Wright, Jr.
// OpenGL 超级宝典
#version 330
// 输入每个顶点.....位置和法向
in vec4 vVertex;
in vec3 vNormal;
smooth out float textureCoordinate;
uniform vec3 vLightPosition;
uniform mat4 mvpMatrix;
uniform mat4 mvMatrix;
uniform mat3 normalMatrix;
void main(void)
{
    // 获取表面法线的视觉坐标
    vec3 vEyeNormal = normalMatrix * vNormal;
    // 获取顶点位置的视觉坐标
    vec4 vPosition4 = mvMatrix * vVertex;
    vec3 vPosition3 = vPosition4.xyz / vPosition4.w;
    // 获取到光源的向量
    vec3 vLightDir = normalize(vLightPosition - vPosition3);
    // 从点乘积得到漫反射强度
    textureCoordinate = max(0.0, dot(vEyeNormal, vLightDir));
    // 不要忘记对多边形进行变换
    gl_Position = mvpMatrix * vVertex;
}
```

除了经过变换的几何图形位置之外，这个着色器唯一的输出就是一个插值纹理坐标 textureCoordinate，这个坐标表示一个单独的 float。这个漫射光照分量的计算与 DiffuseLight 示例几乎是相同的。

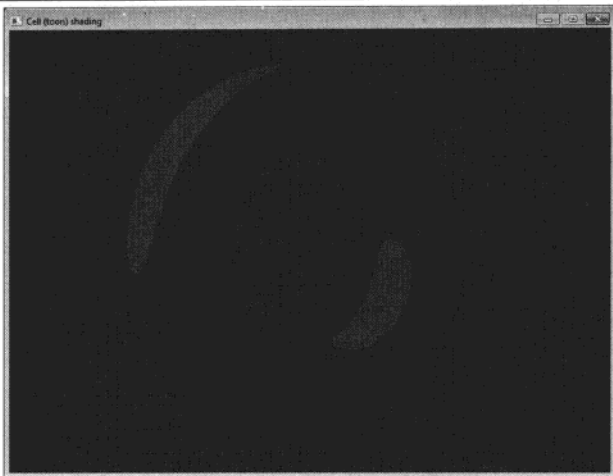
我们的卡通着色器的片段程序只是对一维纹理进行采样，并将它的值写入帧缓冲区片段。

```
vFragColor = texture(colorTable, textureCoordinate);
```

得到的输出结果如图 6.14 所示。彩图 6 展示了红色斜面纹理和卡通着色花托。

图 6.14

卡通着色花托



## 6.6 小结

在本章，我们摆脱了前面 5 章一直使用的存储着色器的束缚。现在，GLTools 中 GLBatch 类型的类提供了一个简单的方法来传递最典型的顶点属性，而我们已经看到，将着色器属性连接到这些类上非常简单。我们已经介绍过，GLSL 与 C/C++ 非常相似，并介绍了可以使用的内建函数，以及如何编写自己的函数。我们已经学习了两种光照模型以及使用着色器实现它们，讨论了在顶点着色器和片段着色器中进行大开销计算的利弊，并且开始了解如何在着色器中访问纹理数据。

我们不只是学习将 2D 纹理映射到纹理几何图形表面，还学习了如何将纹理作为“数据”使用。我们可以将纹理当作查询表值使用以通过 discard 声明来消除几何图形，还可以当作一维颜色表使用来实现动画着色。

我们可以用 GLSL 完成很多工作，在本章只进行了表面的“刮擦”。在后续的章节中，我们将学习更多关于 GLSL 的内容，并学习更加激动人心的渲染技术，同时学习更多关于 OpenGL API 和着色语言的内容。还等什么呢？我们已经懂得足够多的知识，可以开始进行实践了。我们可以修改前面介绍过的着色器，还可以构建自己的着色器！

## 第7章 纹理高级知识

作者: Richard S. Wright, Jr.

### 本章内容

- ✦ 如何使用矩形纹理
- ✦ 如何使用称为立方图的六面体纹理
- ✦ 如何同时使用多个纹理
- ✦ 如何使用点精灵
- ✦ 如何使用纹理数组使着色器能够访问更多的纹理
- ✦ 如何通过代理向驱动程序查询纹理支持信息

在第5章,我们了解了纹理贴图与OpenGL之间的联系。首先我们了解了一些基础:加载2D图像文件、使用纹理坐标,以及不同的环绕模式等。现在我们可以在这个知识库中进行更深、更广的研究了。我们很快就能看到,纹理数据可以采用更多的形式,而不仅仅是从磁盘上加载2D图像文件,有时甚至可以不包含任何形式的视觉数据或者图片的纹理!最后,我们会看到在某些时候纹理甚至不是真正存在的,而是由片段程序凭空创造出来的。

### 7.1 矩形纹理

首先,我们继续以在纹理中使用图像文件的经验为基础,创建我们实际希望以某种形式进行显示的图像。第5章主要是关于使用纹理目标 `GL_TEXTURE_2D` 的2D图像的,我们还在前一章了解了如何将 `GL_TEXTURE_1D` 作为颜色查询表使用来进行动画着色。对于一维、二维和三维纹理(回忆一下 `GL_TEXTURE_3D`)来说,我们在典型情况下将纹理映射到几何图形上,其纹理坐标经过标准化,取值范围为0.0到1.0。我们可以超出这个范围,并使用各种纹理坐标环绕模式确定这个纹理是否以不同的方式进行重复,或者被截取到纹理图像的边缘。

对于二维纹理图像来说，另一个有用的选项是纹理目标 `GL_TEXTURE_RECTANGLE`。这个纹理目标模式的工作方式和 `GL_TEXTURE_2D` 非常相似，但有几点不同。首先，它们不能进行 Mip 贴图，这就意味着我们只能加载 `glTexImage2D` 的第 0 层；第二，纹理坐标不是标准化的，这就意味着纹理坐标实际上是对像素寻址，而不是从 0.0 到 1.0 的范围覆盖图像的。纹理坐标(5, 19) 实际上是图像中从左起 6 个像素，以及从上面起 20 个像素（请记住，程序员是从 0 开始数的！）。

此外，纹理坐标不能重复，并且不支持纹理压缩。

相对于使用纹理来获得 3D 模型表面特征，这种方式对于许多 OpenGL 用来处理和提交图像数据的应用程序来说更加方便，对于纹理矩形的硬件支持也比对通常的 2D 纹理贴图更简单，并且更快、效率更高。

### 7.1.1 加载矩形纹理

程序清单 7.1 用 `glWriteTGA` 函数将屏幕图像保存为一个 Targa 文件。这与我们以前在 `GL_TEXTURE_2D` 纹理中应用过的 `LoadTGATexture` 非常相似。最显著的变化当然就是现在所有的纹理函数都使用 `GL_TEXTURE_RECTANGLE` 而不是 `GL_TEXTURE_2D` 作为它们的第一个参数。任何用于矩形纹理的纹理函数都会发生这种变化。我们还删除了用于检查 Mip 贴图纹理过滤器的代码，因为它们已经不被支持了；我们必须使用 `GL_NEAREST` 或 `GL_LINEAR` 过滤器模式。

我们原样保留了 `wrapMode` 参数，但是 `GL_REPEAT` 和 `GL_REPEAT_MIRRORED` 环绕模式在矩形纹理中也不被支持了。

程序清单 7.1 加载矩形纹理

```
bool LoadTGATextureRect(const char *szFileName, GLenum minFilter,
                        GLenum magFilter, GLenum wrapMode)
{
    GLbyte *pBits;
    int nWidth, nHeight, nComponents;
    GLenum eFormat;
    // 读入纹理位
    pBits = gltReadTGABits(szFileName, &nWidth, &nHeight,
                          &nComponents, &eFormat);

    if(pBits == NULL)
        return false;
    glTexParameteri(GL_TEXTURE_RECTANGLE, GL_TEXTURE_WRAP_S, wrapMode);
    glTexParameteri(GL_TEXTURE_RECTANGLE, GL_TEXTURE_WRAP_T, wrapMode);
    glTexParameteri(GL_TEXTURE_RECTANGLE, GL_TEXTURE_MIN_FILTER, minFilter);
    glTexParameteri(GL_TEXTURE_RECTANGLE, GL_TEXTURE_MAG_FILTER, magFilter);
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    glTexImage2D(GL_TEXTURE_RECTANGLE, 0, nComponents, nWidth, nHeight, 0,
                eFormat, GL_UNSIGNED_BYTE, pBits);

    free(pBits);
    return true;
}
```

### 7.1.2 使用矩形纹理

图 7.1 所示是一个 OpenGL 标志，在第一个示例程序会使用它。

这是一个带有 alpha 通道的 Targa 文件，我们将把它放在活动的 SphereWorld 示例的屏幕前方。这个图像的宽为 300 像素，而高为 155 像素。



图 7.1

OpenGL 标志纹理

将这个文件作为矩形纹理进行加载，看起来和加载以前的 2D 纹理文件很相似。我们还要注意在 `glBindTexture` 函数调用过程中 `GL_TEXTURE_RECTANGLE` 的改变。

```
glBindTexture(GL_TEXTURE_RECTANGLE, uiTextures[3]);  
LoadTGATextureRect("OpenGL-Logo.tga", GL_NEAREST, GL_NEAREST,  
                    GL_CLAMP_TO_EDGE);
```

TextureRect 示例程序的目标就是在屏幕的右下角显示 OpenGL 标志。在屏幕空间中进行 2D 绘制时，普遍的做法是创建一个与屏幕大小相匹配的正投影矩阵。我们选择让坐标系与屏幕上的纹理匹配，但是将原点(0, 0) 设置在了左下角而不是左上角。

这样就能保证绘制坐标都干净整齐地落在笛卡尔坐标系第一象限中了。

对这个投影矩阵的设置请参见下面这段代码。

```
M3DMatrix44f mScreenSpace;  
m3dMakeOrthographicMatrix(mScreenSpace, 0.0f, 800.0f, 0.0f, 600.0f,  
                           -1.0f, 1.0f);
```

我们使用 `GLBatch` 类和一个三角形扇在将要显示 OpenGL 标志的位置创建矩形。请注意从 0.0 到标志宽度或者高度范围内的纹理坐标是如何指定的。

```
int x = 500;  
int y = 155;  
int width = 300;  
int height = 155;  
logoBatch.Begin(GL_TRIANGLE_FAN, 4, 1);  
  
// 左上角  
logoBatch.MultiTexCoord2f(0, 0.0f, height);  
logoBatch.Vertex3f(x, y, 0.0f);  
  
// 左下角  
logoBatch.MultiTexCoord2f(0, 0.0f, 0.0f);  
logoBatch.Vertex3f(x, y - height, 0.0f);  
  
// 右下角  
logoBatch.MultiTexCoord2f(0, width, 0.0f);  
logoBatch.Vertex3f(x + width, y - height, 0.0f);  
  
// 右上角  
logoBatch.MultiTexCoord2f(0, width, height);  
logoBatch.Vertex3f(x + width, y, 0.0f);  
  
logoBatch.End();
```

现在我们有了一个顶点和纹理坐标批次，可以开始进行渲染了。首先需要有一个能够使用矩形纹理的纹

理贴图着色器。这又是一个很普通的对于2D纹理着色器的修改版本,而我们只需要将采样器从sampler2D类型改变成samplerRect类型。程序清单7.2展示了这个采样程序中的片段着色器。

程序清单 7.2 TextureRect (纹理矩形) 示例程序的片段着色器

```
// 矩形纹理 (替换) 着色器
// 片段着色器
// 作者: Richard S. Wright, Jr.
// OpenGL 超级宝典
#version 330

out vec4 vFragColor;

uniform samplerRect rectangleImage;

smooth in vec2 vVaryingTexCoord;

void main(void)
{
    vFragColor = texture(rectangleImage, vVaryingTexCoord);
}
```

最后,我们对常规的 SphereWorld 输出界面顶部的标志进行渲染。为了完成这项工作,我们再次开启混合并关闭深度测试。另一方面,因为改变了坐标系,所以可以很容易地获得基本3D场景的深度值,以防2D图像被不正确地进行渲染。

```
// 开启混合, 关闭深度测试
glEnable(GL_BLEND);
glDisable(GL_DEPTH_TEST);

glUseProgram(rectReplaceShader);
glUniform1i(locRectTexture, 0);
glUniformMatrix4fv(locRectMVP, 1, GL_FALSE, mScreenSpace);
glBindTexture(GL_TEXTURE_RECTANGLE, uiTextures[3]);
logoBatch.Draw();

// 恢复关闭混合并开启深度测试的状态
glDisable(GL_BLEND);
glEnable(GL_DEPTH_TEST);
```

最终输出如图7.2所示,彩图7也展示了这个图形。

图 7.2

最终在3D场景中显示的  
纹理矩形



## 7.2 立方体贴图

立方体贴图是作为一个单独的纹理对象看待的，但是它由组成立方体 6 个面的 6 个正方形（没错，它们必须是正方形！）的 2D 图像组成的。立方体贴图的应用范围包括 3D 光线贴图、反射和高精度环境贴图。

图 7.3 所示是组成立方体的 6 个正方形的布局，我们在 Cubemap 示例程序会使用它。

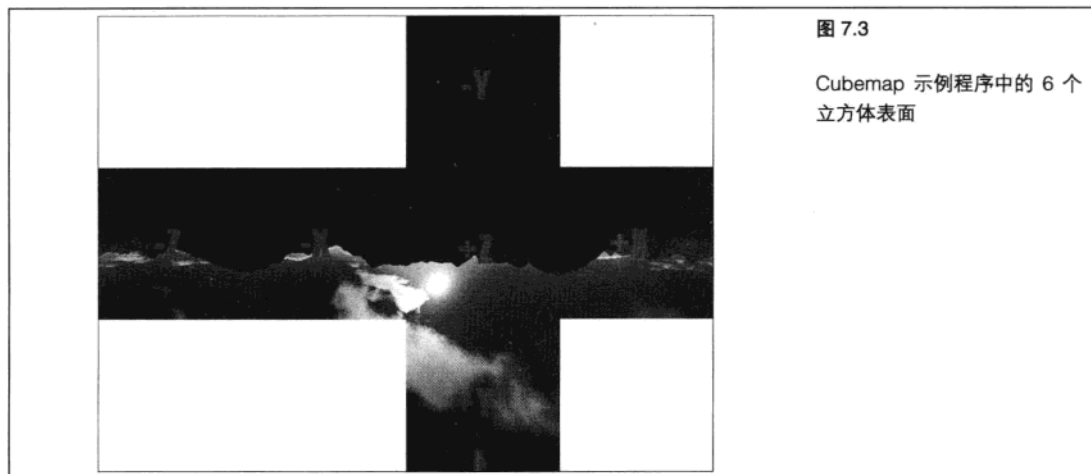


图 7.3

Cubemap 示例程序中的 6 个立方体表面

这 6 个“瓷砖”从 6 个不同的方向（X、Y 和 Z 轴的正、负方向）展示了场景的全貌。实质上，一个立方体贴图是投影到一个对象上的，就像这个立方体贴图是包围着这个对象一样。

### 7.2.1 加载立方体贴图

立方体贴图新增了以下 6 个值，这些值可以传递到 `glTexImage2D`。

`GL_TEXTURE_CUBE_MAP_POSITIVE_X`、`GL_TEXTURE_CUBE_MAP_NEGATIVE_X`、`GL_TEXTURE_CUBE_MAP_POSITIVE_Y`、`GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`、`GL_TEXTURE_CUBE_MAP_POSITIVE_Z` 和 `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`。

这些常量展示了包围被贴图物体立方体表面的场景坐标方向。例如，要加载  $x$  轴正方向的贴图，就可以使用下面的函数。

```
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X, 0, GL_RGBA, iWidth, iHeight,
             0, GL_RGBA, GL_UNSIGNED_BYTE,
             pImage);
```

为了更进一步完善这个示例，请看下面这段取自示例程序的代码片段。这里，我们在一个数组中存储了 6 个立方体贴图表面的名称和标识符，然后使用一个循环将这 6 个图像加载到一个单独的纹理对象中。

```

const char *szCubeFaces[6] = { "pos_x.tga", "neg_x.tga", "pos_y.tga",
                                "neg_y.tga", "pos_z.tga", "neg_z.tga" };

GLenum cube[6] = { GL_TEXTURE_CUBE_MAP_POSITIVE_X,
                   GL_TEXTURE_CUBE_MAP_NEGATIVE_X,
                   GL_TEXTURE_CUBE_MAP_POSITIVE_Y,
                   GL_TEXTURE_CUBE_MAP_NEGATIVE_Y,
                   GL_TEXTURE_CUBE_MAP_POSITIVE_Z,
                   GL_TEXTURE_CUBE_MAP_NEGATIVE_Z };

...
...
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER,
                 GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S,
                 GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T,
                 GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R,
                 GL_CLAMP_TO_EDGE);

GLbyte *pBytes;
GLint iWidth, iHeight, iComponents;
GLenum eFormat;

// 加载立方体贴图图像
for(i = 0; i < 6; i++)
{
    // 加载这个纹理贴图
    (GL_TEXTURE_CUBE_MAP, GL_GENERATE_MIPMAP, GL_TRUE);
    pBytes = gltLoadTGABits(szCubeFaces[i], &iWidth, &iHeight,
                           &iComponents, &eFormat);
    glTexImage2D(cube[i], 0, iComponents, iWidth, iHeight, 0, eFormat,
                 GL_UNSIGNED_BYTE, pBytes);
    free(pBytes);
}
glGenerateMipmap(GL_TEXTURE_CUBE_MAP);

```

这个立方体贴图的纹理坐标乍一看会让人觉得有点奇怪。和真正的 3D 纹理不同，S、T 和 R 纹理坐标表示的是一个从纹理贴图的中心出发的有符号向量。这个向量与立方体贴图的 6 个面中的一个相交。然后，围绕这个交点的纹理向量将被采样，从纹理中创建过滤颜色值。

## 7.2.2 创建天空盒

立方体贴图最普遍的用法就是创建一个反映它周围景象的对象。

Cubemap 示例程序使用的 6 个图像由 The Game Creators, Ltd. ([www.thegamecreators.com](http://www.thegamecreators.com)) 提供。这个立方体贴图被应用到一个球体上，创建了镜面表面的外观。同样的立方体贴图也应用在天空盒上，这个天空盒创建了反射后的背景。

所谓的天空盒只不过是一个带有天空图片的大盒子。另一种观点是将它看成一个贴在大盒子上的天空图片！够简单了吧。一个有效的天空盒包含 6 个图像，这 6 个图像包括从我们场景中心沿着 6 个方向轴所看到的场景。如果这听起来像一个立方体贴图，那么恭喜，说明大家的注意力非常集中！在 Cubemap 示例程

序中，我们在场景中绘制了一个很大的盒子，立方体贴图纹理就应用在这个立方体的 6 个面上。天空盒使用 GLTools 函数 `gltMakeCube` 进行绘制，这个函数所做的只是用组成一个指定半径的立方体的三角形来填充 GLBatch 容器。在这个示例中，我们选择一个在每个方向到原点距离都为 20 个单位长度的立方体。

```
gltMakeCube(cubeBatch, 20.0f);
```

这个函数将 2D 纹理坐标分配给 `GLT_ATTRIBUTE_TEXTURE0` 属性槽，这样在立方体的每个表面上都会应用一个 2D 图像。不过，这样做并不能满足我们对于立方体贴图的需求，我们需要的是代表一个向量的 3D 纹理坐标，沿着这个向量在立方体贴图上进行纹理单元采样。GLBatch 类只支持 2D 纹理，所以它在盒子的外面是无法使用的。解决办法是编写一个自定义顶点着色器，用它为我们计算纹理坐标。实际上，这样只是简单地指定立方体的每个角在顶点空间中也是一个从立方体的中心指向这个位置的向量。我们要做的就是对这个向量进行标准化，而我们已经有了一个现成的立方体贴图纹理坐标。

天空盒着色器顶点程序的源代码如程序清单 7.3 所示。这个示例程序的唯一目的就是通过对模型视图投影矩阵将顶点位置进行变换，并从原来的顶点位置获得一个纹理坐标。

程序清单 7.3 立方体贴图顶点着色器

```
// 天空盒着色器
// 顶点着色器
// 作者: Richard S. Wright, Jr.
// OpenGL 超级宝典
#version 330

// 输入每个顶点...只输入位置
in vec4 vVertex;

uniform mat4 mvpMatrix; // 变换矩阵
// 片段程序的纹理坐标
varying vec3 vVaryingTexCoord;

void main(void)
{
    // 传递纹理坐标
    vVaryingTexCoord = normalize(vVertex.xyz);

    // 不要忘记对几何图形进行变换
    gl_Position = mvpMatrix * vVertex;
}
```

程序清单 7.4 提供的片段程序接受 3 分量纹理坐标，并在这个位置对立方体贴图进行采样。请注意，对于立方体贴图来说，采样器类型为 `samplerCube`。

程序清单 7.4 立方体贴图片段着色器

```
// 天空盒着色器
// 片段着色器
// 作者: Richard S. Wright, Jr.
// OpenGL 超级宝典
#version 330

out vec4 vFragColor;

uniform samplerCube cubeMap;

varying vec3 vVaryingTexCoord;

void main(void)
{
    vFragColor = texture(cubeMap, vVaryingTexCoord);
}
```

关于天空盒最后要注意的一点是，当我们在立方体贴图上使用 Mip 贴图时，沿着两个面结合的边缘常常会出现缝隙（实际上在立方体贴图的其他应用中也会出现）。OpenGL 内部会调整自己的过滤规则，在启用 `GL_TEXTURE_CUBE_MAP_SEAMLESS` 时帮助消除这些缝隙，如下所示。

```
glEnable(GL_TEXTURE_CUBE_MAP_SEAMLESS);
```

### 7.2.3 创建反射

对天空盒进行渲染非常简单，创建反射也只是更复杂一点点而已。

首先必须使用表面法线和指向顶点的向量在着色器中创建一个视觉坐标系中的反射向量。另外，为了获得一个真实的反射，还要考虑照相机的方向。从 `GLFrame` 类中提取照相机的旋转矩阵并进行转置。然后将其作为统一值，与另一个变换矩阵（用来对前述的反射向量进行旋转，这个反射向量实际上就是立方体贴图纹理坐标）一起提供给着色器。

如果不对纹理坐标进行旋转，那么当照相机在场景中移动时，立方体贴图将不能正确地反射围绕它的天空盒。

程序清单 7.5 列出了 `Reflection.vp` 顶点着色器程序的源代码。相应的片段着色器代码与天空盒着色器的片段着色器代码基本相同，它只是使用插值立方体贴图纹理坐标对立方体贴图进行采样并将其应用到片段上。

程序清单 7.5 反射顶点着色器

```
// 反射着色器
// 顶点着色器
// 作者: Richard S. Wright, Jr.
// OpenGL 超级宝典
#version 330

// 输入每个顶点...位置和法向
in vec4 vVertex;
in vec3 vNormal;

uniform mat4 mvpMatrix;
uniform mat4 mvMatrix;
uniform mat3 normalMatrix;
uniform mat4 mInverseCamera;

// 片段程序的纹理坐标
smooth out vec3 vVaryingTexCoord;

void main(void)
{
    // 视觉空间中的法线
    vec3 vEyeNormal = normalMatrix * vNormal;

    // 视觉空间中的顶点位置
    vec4 vVert4 = mvMatrix * vVertex;
    vec3 vEyeVertex = normalize(vVert4.xyz / vVert4.w);

    // 获取反射向量
    vec4 vCoords = vec4(reflect(vEyeVertex, vEyeNormal), 1.0);

    // 通过反转的照相机进行旋转
```

```

vCoords = mInverseCamera * vCoords;
vVaryingTexCoord.xyz = normalize(vCoords.xyz);

// 不要忘记对几何图形进行变换
gl_Position = mvpMatrix * vVertex;
}

```

图 7.4 所示显示了 Cubemap 示例程序的输出。请注意天空和周围的地形是如何正确地通过球体表面反射的。围绕球体移动照相机（通过使用方向键实现）也可以展示背景和天空视图是如何通过球体进行正确的反射的。



图 7.4

Cubemap 示例程序的输出（彩图 8 也展示了这个图像）

## 7.3 多重纹理

以前接触的纹理贴图都是将一个单独的纹理加载到纹理对象上。当我们想要使用这个纹理时，将它绑定到选定的纹理对象上，然后将片段着色器中的单个统一值设置为……0。为什么是 0 呢？因为 0 是我们将要绑定到的纹理单元的索引。现在的 OpenGL 允许我们将独立的纹理对象绑定到一些可用的纹理单元上，从而提供了将两个或更多纹理同时应用到几何图形上的能力。我们可以对实现进行查询，来查看支持的纹理单元数量，如下所示。

```

GLint iUnits;
glGetIntegerv(GL_MAX_TEXTURE_UNITS, &iUnits);

```

默认情况下，第一个纹理单元为活动的纹理单元。所有纹理绑定操作都会影响当前活动的纹理单元。我们可以通过调用以纹理单元标识符为变量的 `glActiveTexture` 来改变当前纹理单元。例如，要切换到第二个纹理单元并将其绑定到指定的纹理上，我们应该进行下面的工作。

```

glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, textureID);

```

在使用多个纹理进行渲染的时候，记住那个纹理单元当前是活动的，这一点非常重要。这些纹理单元在纹理的维数上也是未知的，它们可以是一维、二维或者三维纹理或立方体贴图，也可能是纹理矩形。

### 7.3.1 多重纹理坐标

通过对纹理坐标进行插值，我们可以将纹理应用到几何图形上。没有什么能够阻止我们为了任意数量的纹理单元或者层次（有时候它们会被这样称呼）使用一组纹理坐标。我们还可以计算纹理坐标，就像在前面的示例中为天空盒所做的一样，或者可以为每个纹理提供独立的一组纹理坐标；毕竟这只不过是在我们的批次中增加一组属性而已。这些内容没什么特别的。

默认情况下，GLBatch 类不会以一个属性数组的形式提供任何纹理坐标。不过，在调用以 `nTextureUnits` 为参数的 `Begin` 函数时，我们最多可以指定 4 组纹理坐标。

```
void GLBatch::Begin(GLenum primitive, GLuint nVerts,
                   GLuint nTextureUnits = 0);
```

有两个函数可以提供纹理坐标。第一个函数是 `CopyTexCoordData2f`，它的速度是最快的，因为它会一次复制整个一组纹理坐标。

```
void GLBatch::CopyTexCoordData2f(M3DVector2f *vTexCoords,
                                GLuint uiTextureLayer);
```

第二个函数则是使用较慢的每次一个顶点的接口，与立即模式类似。

我们可以通过两种方式指定一个二维纹理坐标，每次指定一个。

```
void GLBatch::MultiTexCoord2f(GLuint texture, GLclampf s, GLclampf t);
void GLBatch::MultiTexCoord2fv(GLuint texture, M3DVector2f vTexCoord);
```

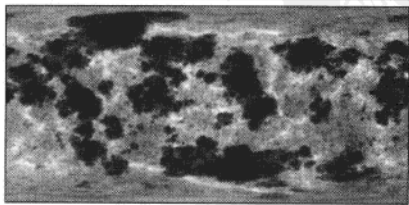
### 7.3.2 多重纹理示例

多重纹理可以以几乎无限多的方式进行组合。有种类繁多的技术采用在一个着色器中一次使用两个或多个纹理的方式。我们仍然向读者推荐附录 A，在这里我们可以了解关于更多 3D 技术的资源。

为了演示将多重纹理进行结合是多么简单，我们最后再创建一个立方体贴图示例，在场景中间的镜面球体上添加一些“晦暗”的成分。“晦暗”纹理如图 7.5 所示。

图 7.5

晦暗纹理示例程序



我们将这个二维纹理绑定到纹理单元 `GL_TEXTURE1` 上，然后用取自晦暗纹理的纹理颜色乘以立方体贴图纹理的颜色。在晦暗纹理较深的地方，反射会变暗，而在晦暗纹理较浅或者接近白色的地方，它对反射纹理几乎没有影响。这个效果的输出结果如图 7.6 所示。



图 7.6

Multitexture 示例程序的输出 (彩图 9 也展示了这个图像)

这项操作的客户端部分非常简单。现在已知 `tarnishTexture` 是包含晦暗纹理的纹理对象，而 `cubeTexture` 是包含立方体贴图的纹理对象名称，下面的代码会将这两个纹理进行绑定，其中每个纹理都绑定到自己的纹理单元上。

```
// 将纹理绑定到自己的纹理单元上
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, tarnishTexture);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_CUBE_MAP, cubeTexture);
```

回忆一下，以前的球体批次包含一组二维纹理坐标，这组坐标在立方体贴图示例程序中并没有使用。现在我们修改一下着色器代码，将这些纹理坐标用在晦暗纹理上，同时继续对立方体贴图纹理坐标进行计算。我们只讨论新增加的这 3 行着色器代码，而不再重新列出整个顶点程序。首先，为晦暗纹理将要使用的二维为纹理坐标添加属性。

```
in vec2 vTexCoords;
```

当然，随后我们需要对它们进行插值，所以设置一组坐标，这组坐标能够在顶点之间平滑地进行插值。

```
smooth out vec2 vTarnishCoords;
```

最后，我们只要将这些属性分配给插值变量即可。

```
vTarnishCoords = vTexCoords.st;
```

这实在不算什么，对于普通的纹理贴图来说，我们通常不会做这些工作。最大的改变发生在片段程序中，程序清单 7.6 完整地列出了片段程序的代码。

程序清单 7.6 支持多重纹理的反射着色器

```
// 支持多重纹理的反射着色器
// 片段着色器
// 作者: Richard S. Wright, Jr.
// OpenGL 超级宝典
#version 330
```

```

out vec4 vFragColor;

uniform samplerCube cubeMap;
uniform sampler2D tarnishMap;

smooth in vec3 vVaryingTexCoord;
smooth in vec2 vTarnishCoords;

void main(void)
{
    vFragColor = texture(cubeMap, vVaryingTexCoord.stp);
    vFragColor *= texture(tarnishMap, vTarnishCoords);
}

```

请注意现在有两个采样器，即 samplerCube 类型的 cubeMap 和 sampler2D 类型的 tarnishMap。这两个纹理使用各自的纹理坐标进行采样，而结果得到的过滤颜色值只是简单地相乘，就会得到最终的片段颜色。

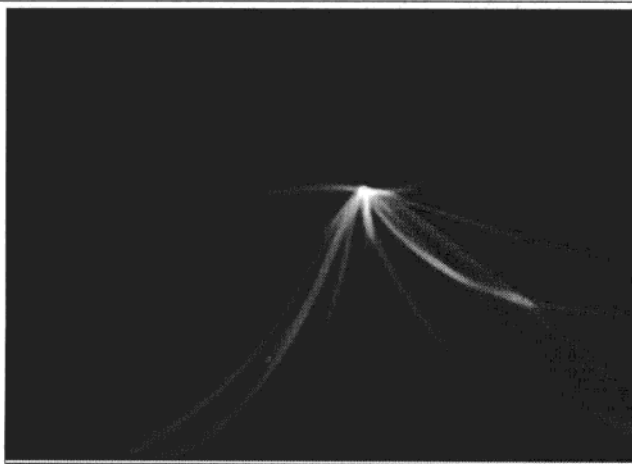
## 7.4 点精灵（点块纹理）

点精灵（Point Sprite，又译为点块纹理）是 OpenGL 1.5 及更新版本所支持的一个激动人心的特性。尽管 OpenGL 一直支持对点进行纹理贴图，但在 1.5 版本之前这意味着将单个纹理坐标应用于整个点。较大的经过纹理贴图的点就是经过过滤的单个纹理单元的放大版本。使用点精灵（点精灵在 OpenGL 3.0 及更新版本中已成为默认的点渲染模式），我们可以通过绘制一个 3D 点将一个 2D 纹理图像显示在屏幕的任意位置上。

点精灵最常见的应用可能就要算微粒系统了。我们可以用点来表示在屏幕上移动的大量微粒，来产生一些视觉效果。但是，把这些点表示为很小的重叠 2D 图像可以戏剧性地流动着动画细丝。例如，图 7.7 显示了 Macintosh（苹果电脑）上一个众所周知的屏幕保护程序，它是由微粒效果所驱动的。

图 7.7

一个屏幕保护程序所显示的  
微粒效果



在点精灵之前，为了实现这种类型的效果，需要在屏幕上绘制大量纹理四边形（或者三角形带）。这可以通过对每个多边形执行开销巨大的旋转来实现，以确保它面对着照相机，或者也可以在 2D 正交投影

下绘制所有的微粒。点精灵允许我们通过发送单个 3D 顶点，渲染一个完美对齐的纹理 2D 多边形。点精灵是 OpenGL 中一种强大而又高效的特性，它所需要的带宽只有为四边形发送 4 个顶点所需带宽的四分之一，并且不需要客户端的矩阵逻辑来保持 3D 四边形与照相机的对齐。

### 7.4.1 使用点

点精灵非常容易使用。在客户端，我们需要做的只是简单地绑定一个 2D 纹理（不要忘记为纹理单元设置恰当的统一值！），因为现在点精灵已经是默认的点光栅化模式，只有一种情况下例外，就是在开启点平滑（point smoothing）的时候。

我们不能同时使用点精灵和抗锯齿点。在片段程序中，有一个内建变量 `gl_PointCoord`，这是一个二分量向量，在顶点上对纹理坐标进行插值。程序清单 7.7 展示了 PointSprites 示例程序中的片段着色器。

程序清单 7.7 在片段着色器中渲染一个点精灵

```
// SpaceFlight 色器
// 片段着色器
// 作者: Richard S. Wright, Jr.
// OpenGL 超级宝典
#version 330

out vec4 vFragColor;

in vec4 vStarColor;

uniform sampler2D starImage;

void main(void)
{
    vFragColor = texture(starImage, gl_PointCoord) * vStarColor;
}
```

所以对于点精灵来说，我们不需要将纹理坐标作为一个属性进行传递。

既然一个点就是一个单独的顶点，我们就不能以任何其他方式在点表面上进行插值了。当然，如果我们无论如何都要提供一个纹理坐标，或者以自定义的方式进行插值，那么也没有什么能够阻止我们。

### 7.4.2 点大小

有两种方式可以设置点大小。第一种方式是 `glPointSize` 函数。

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

这个函数为锯齿点和抗锯齿点设置点的直径，以像素为单位。在第 3 章已经介绍了这个函数，以及如何确定可用的点大小范围。我们也可以在顶点着色器中用程序设置点大小。首先启用点大小模式。

```
glEnable(GL_PROGRAM_POINT_SIZE);
```

然后，在我们的顶点程序中，可以设置一个内建变量 `gl_PointSize`，这个变量确定了点的最终光栅化

大小。这种方式的一种常见用途就是根据点的距离来确定它的大小。当我们使用 `glPointSize` 函数设置点的大小时，它们不受透视除法的影响，而是将所有的点设置为同样大小，无论它们有多远。

下面的方程常常用来实现基于距离的点大小变化。

$$size = \sqrt{\left( \frac{1}{a + b*d + c*d^2} \right)}$$

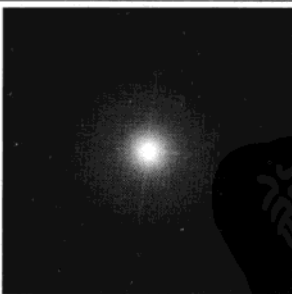
其中  $d$  是从这个点到观察点的距离，而  $a$ 、 $b$  和  $c$  则是二次方程的参数。我们可以将它们存储为统一值，并用应用程序来对它们进行更新，或者如果我们已经想好了一组特定的参数，也可以在顶点着色器中将它们设为常量。例如，如果我们想设定一个常量大小值，那么就将  $a$  设为非 0 值，而将  $b$  和  $c$  设为 0。如果  $a$  和  $c$  都设为 0 而  $b$  设为非 0 值，那么点大小将随着距离的变化而线性变化。类似地，如果  $a$  和  $b$  都设为 0 而  $c$  设为非 0 值，那么点大小将随着距离的变化而呈平方关系变化。

### 7.4.3 综合运用

现在让我们来看一个示例程序，其中运用了我们到目前为止讨论过的点精灵特性。PointSprite (点精灵) 示例程序创建了一个动画星空，它看起来就像我们在星空中向前飞行一样。这种效果通过在视野前方放置随机点，然后将一个时间值作为统一值传递给顶点而实现的。这个时间值用来移动这个点的位置，从而使这些点随着时间的推移向我们靠近，然后在它们在接近视景体背面的邻近剪切面时进行循环。此外，我们还会对星星的大小进行缩放，这样它们在开始的时候很小，但随着越来越接近我们的视野而越来越大。这样做得到的结果非常真实……现在我们唯一需要的就是一些天文馆或者太空电影的音乐了！图 7.8 所示展示了我们在点上应用的星星纹理。这只是一个 Targa 文件，我们使用与到目前为止加载任何其他 2D 纹理相同的方法来加载它；也可以在点上进行 Mip 贴图，而且因为它们的范围可能会从非常小到非常大，所以这样做确实是个好主意。

图 7.8

星星纹理贴图 (同样显示在了彩图 10 中)



我们并不打算介绍设置星空效果的所有细节，因为这些内容基本上都属于例行公事。如果想要看看随机数是如何选择的，我们可以查看一下源代码。更重要的是 `RenderScene` (渲染场景) 函数中代码的实际渲染。

```
glClear(GL_COLOR_BUFFER_BIT);

// 开启附加混合
glEnable(GL_BLEND);
```

```

glBlendFunc(GL_ONE, GL_ONE);

// 让顶点程序决定点大小
glEnable(GL_PROGRAM_POINT_SIZE);

// 绑定到着色器上, 设置 Uniform 值
glUseProgram(starFieldShader);
glUniformMatrix4fv(locMVP, 1, GL_FALSE, viewFrustum.GetProjectionMatrix());
glUniform1i(locTexture, 0);

// fTime 从 0.0 变化到 999.0 并进行循环
float fTime = timer.GetElapsedSeconds() * 10.0f;
fTime = fmod(fTime, 999.0f);
glUniform1f(locTimeStamp, fTime);

// 绘制星星
starsBatch.Draw();

```

读者可能一上来就发现了, 我们没有对深度缓冲区进行清除。这是因为要使用附加混合来对行星与背景进行混合。因为纹理暗部是黑色的 (在颜色空间中为 0), 所以我们只要将颜色相加就可以了, 就像我们所看到的一样。使用 alpha 透明度会要求按照深度对星星进行排序, 而我们完全可以避免这种开销。在开启点大小程序模式之后, 我们将绑定到着色器上, 并设置统一值。这里令人感兴趣的是, 我们有一个计时器, 它驱动星星的循环 z 坐标, 所以只从 0 计数到 999。程序清单 7.8 展示了顶点着色器的源代码, 其中也有一些有趣的特性。

程序清单 7.8 星空效果的顶点着色器

```

// SpaceFlight 着色器
// 顶点着色器
// 作者: Richard S. Wright, Jr.
// OpenGL 超级宝典
#version 330

// 输入每个顶点的位置和法向
in vec4 vVertex;
in vec4 vColor;
uniform mat4 mvpMatrix;
uniform float timeStamp;

out vec4 vStarColor;

void main(void)
{
    vec4 vNewVertex = vVertex;
    vStarColor = vColor;

    // 由运行时进行偏移, 使它移近
    vNewVertex.z += timeStamp;

    // 如果超出范围则进行调整
    if(vNewVertex.z > -1.0)
        vNewVertex.z -= 999.0;

    // 自定义大小调整
    gl_PointSize = 30.0 + (vNewVertex.z / sqrt(-vNewVertex.z));

    // 如果它们非常小的话, 则让它们逐渐消失
    if(gl_PointSize < 4.0)
        vStarColor = smoothstep(0.0, 4.0, gl_PointSize) * vStarColor;
}

```

```
// 不要忘记对几何图形进行变换
gl_Position = mvpMatrix * vNewVertex;
}
```

顶点的  $z$  位置通过时间戳统一值进行偏移, 星星向我们靠近的动画效果就是这样实现的。我们需要检查位置, 而当它们到达临近剪切面时, 只要将它们的位置进行循环回到远端剪切面即可。我们使用一个平方根倒数函数来使星星在接近我们的过程中越来越大, 并用变量设置最终的大小。如果星星太小, 有时候会变得闪烁, 所以最后再进行一项检查, 当点的大小小于 4.0 时, 使这个点的颜色逐渐变暗, 从而使它们在视野中消失, 而不是在远端剪切面附近忽隐忽现。最终的输出结果如图 7.9 所示。

图 7.9

使用点精灵飞越太空 (同样显示在彩图 10 中)

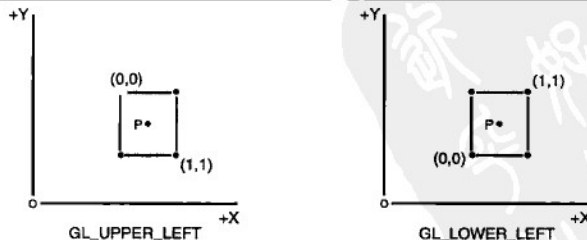


## 7.4.4 点参数

通过 `glPointParameter` 函数, 我们可以对点精灵 (实际上还包括一般的点) 的几个特性进行微调。图 7.10 所示为应用到一个点精灵上的纹理的原点 (0,0) 的两个可能位置。

图 7.10

一个点精灵上的纹理的两个可能方向



将 `GL_POINT_SPRITE_COORD_ORIGIN` 参数设置为 `GL_LOWER_LEFT`, 可以将纹理坐标系的原点放置在点的左下角。

```
glPointParameteri(GL_POINT_SPRITE_COORD_ORIGIN, GL_LOWER_LEFT);
```

点精灵的默认方向为 `GL_UPPER_LEFT`。

另外一个与纹理无关的点参数也可以用来设置 alpha 值,使点可以通过将 alpha 与到观察点的距离进行混合而逐渐消失。要了解这些参数的相关细节,请参见附录 C 中的 `glPointParameter` 函数入口。

### 7.4.5 异形点

除了以 `gl_PointCoord` 为纹理坐标应用纹理之外,我们还可以用点精灵来完成一些其他工作。`gl_FragCoord` 就是另外一个内建变量。

事实上,在任何其他图元进行渲染时,`gl_FragCoord` 会包含当前片段的屏幕空间坐标。这样,这个坐标的  $x$  和  $y$  分量在这个点区域的不同位置也不相同。然而, $z$  和  $w$  分量都是常量,因为这个点是作为一个平面进行渲染的,这个平面与近端面 and 远端面平行。

我们可以使用 `gl_PointCoord` 来完成很多工作,而不仅仅是纹理坐标。

例如,我们可以在片段着色器中使用 `discard` 关键字来丢弃位于我们想要的点形状范围之外的片段,从而创建出非正方形的点。

下面的片段着色器代码能够生成圆形的点。

```
vec2 p = gl_PointCoord * 2.0 - vec2(1.0);  
if (dot(p, p) > 1.0)  
    discard;
```

或者还可以生成有趣的花朵形状。

```
vec2 temp = gl_PointCoord * 2.0 - vec2(1.0);  
if (dot(temp, temp) > sin(atan(temp.y, temp.x) * 5.0))  
    discard;
```

这些都是简单的代码段,允许我们渲染任意形状的点。图 7.11 所示展示了我们可以用这种方式生成的一些其他有趣图形的例子。

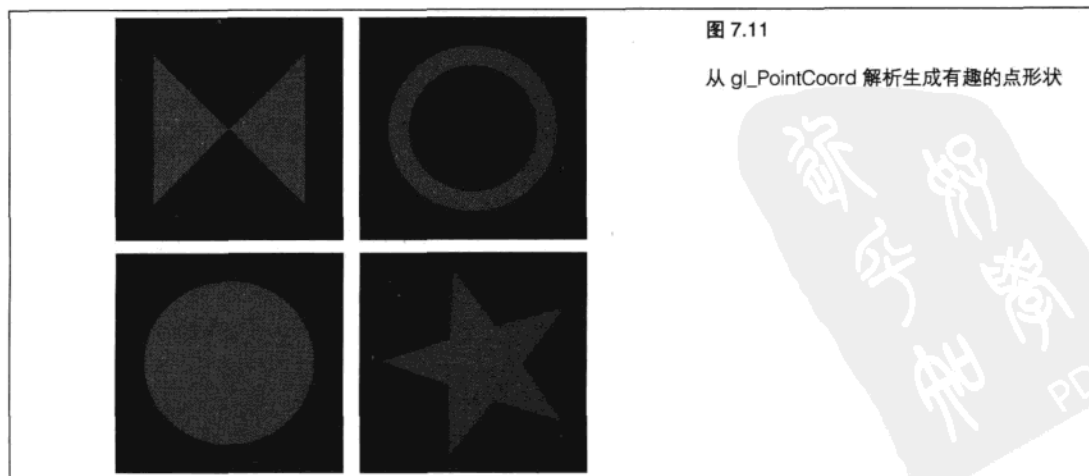


图 7.11

从 `gl_PointCoord` 解析生成有趣的点形状

### 7.4.6 点的旋转

因为 OpenGL 中的点是作为按轴对齐的正方形而进行渲染的, 对点精灵进行旋转必须通过修改用于读取点精灵纹理的纹理坐标来完成。要完成这项工作, 我们只需在片段着色器中创建一个 2D 旋转矩阵, 并用它乘以 `gl_PointCoord` 使它围绕  $z$  轴进行旋转。旋转的角度可以从顶点着色器或者几何着色器中作为一个插值变量传递到片段着色器。变量的值可以在顶点着色器或几何着色器中依次进行计算, 或者也可以通过一个顶点属性提供。程序清单 7.9 展示了一个更加复杂一点的点精灵片段着色器, 它允许点围绕着自己的中心进行旋转。

程序清单 7.9 自转点精灵片段着色器

```
#version 330

uniform sampler2D sprite_texture;

in float angle;

out vec4 color;

void main(void)
{
    const float sin_theta = sin(angle);
    const float cos_theta = cos(angle);
    const mat2 rotation_matrix = mat2(cos_theta, sin_theta,
                                      -sin_theta, cos_theta);
    const vec2 pt = gl_PointCoord - vec2(0.5);
    color = texture(sprite_texture, rotation_matrix * pt + vec2(0.5));
}
```

这个示例允许我们创建旋转的点精灵。不过, `angle` 的值当然不能在点精灵中的片段之间进行改变。这就是说, 对于点中的每个片段来说, 旋转矩阵也是一个常量。这样, 比起为每个片段分别计算旋转矩阵, 在顶点着色器中进行旋转矩阵运算, 然后将它作为一个 `mat2` 变量传递给片段着色器的效率要高得多。

这里有一个更新过的顶点着色器和片段着色器, 允许我们绘制旋转的点精灵。首先, 程序清单 7.10 列出了这个顶点着色器。

程序清单 7.10 旋转点精灵顶点着色器

```
#version 330

uniform matrix mvp;

in vec4 position;
in float angle;

out mat2 rotation_matrix;

void main(void)
{
    const float sin_theta = sin(angle);
    const float cos_theta = cos(angle);
    rotation_matrix = mat2(cos_theta, sin_theta,
                          -sin_theta, cos_theta);
    gl_Position = mvp * position;
```

```

}

```

其次，程序清单 7.11 列出了这个片段着色器。

程序清单 7.11 旋转点精灵片段着色器

```

#version 330

uniform sampler2D sprite_texture;

in mat2 rotation_matrix;

out vec4 color;

void main(void)
{
    const vec2 pt = gl_PointCoord - vec2(0.5);
    color = texture(sprite_texture, rotation_matrix * pt + vec2(0.5));
}

```

就像我们所看到的，开销很大的正弦和余弦函数从片段着色器中转移到了顶点着色器中。如果点很大的话，这一对着色器的执行效果将会比以前那种靠“蛮力”在片段着色器中进行旋转矩阵计算的方式好得多。

## 7.5 纹理数组

在本章前面部分我们讨论了可以通过不同的纹理单元一次性访问几个纹理的情况。这项功能非常强大也非常有用，着色器可以一次性访问几个纹理对象。实际上我们可以更进一步使用称为纹理数组的特性。通过纹理数组，我们可以将几个 2D 图像加载到一个单独的纹理对象中。在一个纹理中添加多个图像已经不是什么新概念了。这种情况在 Mip 贴图中已经出现过，每个 Mip 层次都是一个不同的图像，而在立方体贴图中，立方体的每个面都有它们自己的图像，甚至是自己的一组 Mip 层次。但是，在纹理数组中，我们可以将整个数组的纹理图像绑定到一个纹理对象上，然后在着色器中对它们进行检索，这样就大大增加了着色器可用的纹理数据的数量。

### 7.5.1 加载 2D 纹理数组

为了演示纹理数组，我们回过头来再看一看第 3 章中的 Smoother 示例程序。在这个程序中，我们用线绘制了一个格式化的 2D 山脉、大大小小的星星，以及一个代表月亮的白色圆形。对于 TextureArray(纹理数组)示例程序来说，我们使用新的点精灵特性对星星进行一点美化，并且使用一个 2D 纹理数组(它们实际上也是 1D 纹理数组!)显示一个月亮图像的动画序列。我们提供了 29 个不同的月亮图像，编号从 moon00.tga 到 moon28.tga，并把它们加载到一个单独的纹理对象中，然后设置一个表示经过时间的统一值，并且每经过一秒就切换到数组中的下一个月亮图像。经过 30 秒后，就能看到一个月亮在每个月中循环变化的动画了。

纹理数组添加了两个新的纹理对象作为大多数纹理管理函数的有效参数，它们是 GL\_TEXTURE\_1D\_ARRAY 和 GL\_TEXTURE\_2D\_ARRAY。对于这些二维月亮图像来说，就像任何其他

纹理一样将它们创建和绑定到纹理上，不同的只是在这里改变了 target 参数。

```
GLuint moonTexture;
.....
.....
glGenTextures(1, &moonTexture);
glBindTexture(GL_TEXTURE_2D_ARRAY, moonTexture);
```

纹理参数、环绕模式和过滤器的情况也是如此。

```
glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

到目前为止只是改变了纹理目标参数，这是非常直观的。实际上现在加载纹理数据是一个小小的飞跃；对于 2D 纹理数组来说，我们使用 glTexImage3D 函数。

```
void glTexImage3D(GLenum target, GLint level, GLint internalformat,
                 GLsizei width, GLsizei height, GLsizei depth, GLint border,
                 GLenum format, GLenum type, void *data);
```

对于 target 参数来说，我们仍然使用 GL\_TEXTURE\_2D\_ARRAY，并且参数代表 2D 图像的“切片”或数组索引。关于这个方程的使用，一个优点是它可以一次性加载整个 2D 图像数组。使用这个方程的一个缺点是，它需要一次性加载一个 2D 图像数组——这并不总是那么方便的，特别是当有 29 个独立的月亮图像需要加载时。

完成这项工作还有一个简单的变通方法，这是所有 glTexImageXD 函数（我们还没有讲到）都有的特性。如果我们将 NULL 作为最后一个参数（这样就没有任何纹理数据需要复制了），OpenGL 会保留纹理存储空间，但会将它们保持为未初始化状态。

这样，我们随后就可以使用 glTexSubImageXD 函数族来更新纹理了（所有这些函数都在第 5 章介绍过了）。为了达到我们的目的，需要保留 29 个 64 x 64 的 RGBA 图像，所以代码应如下所示。

```
glTexImage3D(GL_TEXTURE_2D_ARRAY, 0, GL_RGBA, 64, 64, 29, 0,
             GL_BGRA, GL_UNSIGNED_BYTE, NULL);
```

然后需要加载其他的图像，每次加载一个。我们设置一个循环，用来根据循环索引为每个文件创建文件名，并使用 glTexSubImage3D 函数加载图像，每次加载一个切片。

```
for(int i = 0; i < 29; i++) {
    char cFile[32];
    sprintf(cFile, "moon%02d.tga", i);

    GLbyte *pBits;
    int nWidth, nHeight, nComponents;
    GLenum eFormat;

    // 读入纹理位
    pBits = gltReadTGABits(cFile, &nWidth, &nHeight, &nComponents, &eFormat);
    glTexSubImage3D(GL_TEXTURE_2D_ARRAY, 0, 0, 0, i, nWidth, nHeight,
                   1, GL_BGRA, GL_UNSIGNED_BYTE, pBits);

    free(pBits);
}
```

## 7.5.2 纹理数组索引

现在我们的纹理数组已经加载并做好准备,可以使用了。在对月亮进行渲染之前绑定到这个纹理对象,现在我们可以用一个采样器来访问整个月亮图像数组了。当然,我们需要一种方式与着色器进行通信,来确定要使用哪个图像。我们设置一个计时器,它随着秒数的变化而进行循环。在对月亮(只不过是一个三角形扇)进行渲染之前,下面的代码会在顶点着色器中设置恰当的统一值。

```
// fTime 从 0.0 变化到 28.0 并进行循环
float fTime = timer.GetElapsedSeconds();
fTime = fmod(fTime, 28.0f);
glUniform1f(locTimeStamp, fTime);

moonBatch.Draw();
```

在着色器中,已经有了一个接受纹理坐标的属性,而我们只要将  $s$  和  $t$  坐标复制到 `vec3` 变量 `vMoonCoords` 中就可以了。纹理坐标  $p$  来自包含经过时间(不要忘记,我们的循环实际上是从 0 到 28 的)的统一值,而第 3 个纹理坐标维度值随后将在片段着色器中使用。程序清单 7.12 列出了这个顶点着色器。

程序清单 7.12 纹理数组示例的顶点着色器

```
// MoonShader
// 顶点着色器
// 作者: Richard S. Wright, Jr.
// OpenGL 超级宝典
#version 330
in vec4 vTexCoords;

uniform mat4 mvpMatrix;
uniform float fTime;

smooth out vec3 vMoonCoords;

void main(void)
{
    vMoonCoords.st = vTexCoords.st;
    vMoonCoords.p = fTime;

    gl_Position = mvpMatrix * vVertex;
}
```

## 7.5.3 访问纹理数组

在片段着色器(如程序清单 7.13 所示)中,有一个新类型的 2D 纹理数组采样器 `sampler2DArray`,使用 `texture2DArray` 函数对这个纹理进行采样,并且传递一个 3 分量纹理坐标。这个纹理坐标的前两个分量  $s$  和  $t$ (参见变量 `vMoonCoords`)用作典型的二维纹理坐标。第 3 个分量  $p$  实际上是纹理数组的一个整型索引。回忆一下,我们在顶点程序中对此进行过设置,它的值从 0 到 28 变化,每秒变化一个整数。最后得到的结果是一个动画图像,每隔一秒变化一次。

程序清单 7.13 纹理数组示例的片段着色器

```
// MoonShader
// 片段着色器
// 作者: Richard S. Wright, Jr.
// OpenGL 超级宝典
#version 330

out vec4 vFragColor;

uniform sampler2DArray moonImage;

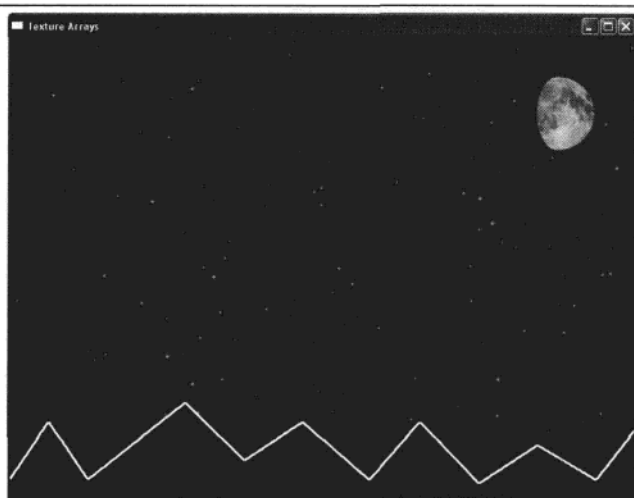
smooth in vec3 vMoonCoords;

void main(void)
{
    vFragColor = texture2DArray(moonImage, vMoonCoords.stp);
}
```

TextureArray 示例程序的最终输出结果如图 7.12 所示。

图 7.12

使用纹理数组的月亮动画图像



## 7.6 纹理代理

纹理内存是一种既重要而又有限的资源，开发者需要特别注意这一点。纹理数据有很多不同的大小和数据类型，并且在某些情况下，能够弄清纹理是如何通过一个特定实现进行管理和存储是非常有用的，更不用说我们甚至还可以加载特定的纹理了。通常我们只是加载大小适中的纹理，并且如果在开发过程中它们能够在屏幕上显示，那么我们就有了很大的信心，确信它们也会在终端用户的屏幕上显示出来。

这有点像俄罗斯轮盘赌，并且对于高质量的商业软件来说，这是一种糟糕的编程实践。我们可以做一项简单的测试，来弄清当前实现能够支持最大的纹理大小。

```
GLint maxSize;
glGetIntegerv(GL_MAX_TEXTURE_SIZE, &maxSize);
```

这样就给出了一个一维或二维纹理贴图（我们也可以为那些相应的纹理坐标类型使用 `GL_MAX_3D_TEXTURE_SIZE` 和 `GL_MAX_CUBE_MAP_TEXTURE_SIZE`）最大宽度或者最大高度的下限。如果 `maxSize` 返回 2048，那就说明支持 2048 x 2048 的 2D 纹理。不过其实也有可能支持 2048 x 4096 的纹理大小，而 2048 这个值只是说明不支持 4096 x 4096 而已。

要弄清是否支持某种特定的纹理大小和格式，我们可以使用一个纹理代理（texture proxy）。纹理代理是一种“伪造”的或者“代替”的纹理，不占用内存空间（更不要想试图将它应用到任何几何图形上了！），但是也可以是一种加载纹理的有效尝试。

要创建一个纹理代理，我们可以使用现在大家已经非常熟悉的 `glTexImage2D` 函数。

```
glTexImage2D(GL_PROXY_TEXTURE_2D, level, internalFormat,
             width, height, border, format, type, NULL);
```

纹理代理也可以用于纹理目标 `GL_PROXY_TEXTURE_1D`、`GL_PROXY_TEXTURE_3D` 和 `GL_PROXY_TEXTURE_CUBE_MAP`。请注意，我们将 `NULL` 作为最后一个参数，在典型情况下这个参数用作指向纹理数据的指针。再次强调，纹理代理并不实际创建一个真正的纹理。我们一旦创建了纹理代理，就可以使用 `glGetTexLevelParameter` 对所有纹理值进行查询。如果 OpenGL 内部对数据进行重新格式化，我们可以（举例来说）查询 `GL_TEXTURE_INTERNAL_FORMAT` 值，看一看实际值到底是什么。如果纹理根本不能被加载，那么这个函数进行的查询将返回 0。例如，为了查明是否能够真正加载一个 2048 x 4096 的 BGRA 纹理，我们可以创建类似下面这样的代理。

```
glTexImage2D(GL_PROXY_TEXTURE_2D, 0, GL_RGBA, 2048, 4096,
             0, GL_BGRA, GL_UNSIGNED_BYTE, NULL);
```

然后查看是否支持相应的高度 4096。

```
void glGetTexLevelParameter(GL_PROXY_TEXTURE_2D, 0,
                           GL_TEXTURE_HEIGHT, &height);
```

我们可以查询关于当前加载纹理的所有类型的信息，无论是真正的纹理还是纹理代理。这个函数的完整列表参见附录 C。

## 7.7 小结

在本章，我们继续探索了 OpenGL 纹理技术，了解了两种新类型，即矩形纹理和立方体贴图的纹理目标。矩形纹理是方便和高效的，尤其是在有图像需求并且不是特定渲染 3D 模型的时候。立方体贴图是一种强大的方法，能够创建三维光线贴图或基于图像的反射。接下来我们了解多重纹理，这种技术是许多特效和技术的基础。

我们了解了如何在一个点的表面应用纹理，以及这对于创建奇妙的微粒系统来说是多么有用。我们还学习了如何利用纹理数组在一个单独的对象上加载大量纹理。最后，我们讨论了纹理代理，以及如何容易地对 OpenGL 实现进行查询，以获得关于纹理数据在其内部的表示形式，或者甚至是一个纹理能否进行内部存储的信息。纹理贴图是 OpenGL 中的一个很重要的主题，到现在为止我们应该已经很好地掌握了它的基本原则，这些原则是更高级技术的基础。

## 第二部分 深入探索

现在是进行更进一步学习的时候了。如果读者是按顺序阅读本书的话，那么应该不仅对 OpenGL 有了基本的了解，而且对基本的 3 D 图形编程原则也有了一定的了解。实际上，在第一部分“基本概念”中，几乎已经涵盖了我們学习如何创建实时交互 3 D 图形所需的所有知识。坐标系、顶点变换、图元装配、纹理贴图、基本着色器操作和编程——这些都是游戏、模拟、可视化和无数消费及商业应用程序中 3 D 图形效果的基础。

在这一部分内容中，我们开始接触 OpenGL API 的一些更加高级的特性。我们不再停留在基础阶段，现在应该去探索当 OpenGL 应用在更加复杂的渲染问题上时到底能有多灵活、多强大。不过，并不是所有问题都是简单地关于渲染效果的，许多 OpenGL 特性都是针对性能问题的。

相对于 API 和方法的变化，第一部分中的原则几十年来一直保持不变。在第二部分，我们要真正开始探索艺术王国和图形硬件的未来发展会将我们带向何处了。

尽情享受吧！

## 第 8 章 缓冲区对象：存储尽在掌握

作者：Nicholas Haemel

## 本章内容

任 务	使用的函数
创建和使用缓冲区对象	glGenBuffers/glBindBuffer/glBufferData
创建、绑定和使用帧缓冲区对象	glGenFramebuffers/glBindFramebuffer
创建、绑定和使用渲染缓冲区对象	glGenRenderbuffers/glBindRenderbuffer
使用纹理缓冲区对象	glTexBuffer
为渲染缓冲区分配存储空间并	glRenderbufferStorage/
连接到帧缓冲区	glFramebufferRenderbuffer
将纹理连接到帧缓冲区对象	glFramebufferTexture2D
设置多重颜色输出	glDrawBuffers

到目前为止，我们已经学习了 OpenGL 基础、如何指定几何图形、什么是着色器、如何使用纹理等。现在可以揭开我们应用程序的“盖子”，介绍更加快速灵活地进行渲染和移动数据的方法了。我们还将学习离屏渲染，以及如何创建和控制自己的帧缓冲区。

缓冲区对象是一个强大的概念，它允许应用程序迅速方便地将数据从一个渲染管线移动到另一个渲染管线，以及从一个对象绑定到另一个对象。我们的数据最终摆脱了强类型对象（strongly typed object）的束缚！我们不仅可以把数据移动到合适的位置，还可以在无需 CPU 介入的情况下完成这项工作。

帧缓冲区对象使我们获得了对像素的真正控制。我们不再受我们的环境所绑定到的操作系统窗口的限制。实际上，我们现在可以离屏对想要的任意数量的缓冲区进行渲染。不仅如此，我们还可以使用最适合我们需要的任意大小和格式的表面。现在片段着色器对哪些像素去哪里有了最终的控制权。

## 8.1 缓冲区

OpenGL 3.2 并没有创建上百种不同类型的对象让开发者忙于记录哪个是哪个，而是对大多数保存数据的对象进行了归纳。现在可以分配任意多个我们需要的缓冲区，然后再决定之后如何使用它们。缓冲区有很多不同的用途，它们能够保存顶点数据、像素数据、纹理数据、着色器处理的输入，或者不同着色器阶段的输出。

缓冲区保存在 GPU 内存中，它们提供高速和高效的访问。在 OpenGL 有缓冲区对象之前，应用程序只有有限的选择可以在 GPU 中存储数据。另外，在 GPU 中更新数据常常需要重新加载整个对象。在系统内存和 GPU 内存之间来回移动数据可能是一个缓慢的过程。

首先让我们来看一看处理缓冲区对象的基础知识。随后，我们将了解对缓冲区中的数据进行访问的更高级方式，以及如何使用它们来达到不同的目的。

### 8.1.1 创建自己的缓冲区

创建一个新缓冲区非常简单，只需调用 `glGenBuffers` 来为我们所需任何数量的新缓冲区创建名称。实际缓冲区对象将在第一次使用时创建。

```
GLuint  pixBuffObjs[1];
glGenBuffers(1, pixBuffObjs);
```

一旦有了新缓冲区的名称，我们就能将这个名称进行绑定来使用缓冲区了。

在 OpenGL 中有许多不同的绑定点（binding point），每个绑定点都允许我们为了不同的目的而使用某个缓冲区。我们可以将每个结合点或绑定点看作一个在同一时刻只能结合一个对象的槽。表 8.1 列出了这些绑定点。稍后我们将更详细地讨论如何使用这些绑定。

表 8.1 缓冲区对象绑定点

名 称	描 述
GL_ARRAY_BUFFER	数组缓冲区存储颜色、位置、纹理坐标等顶点属性，或者其他自定义属性
GL_COPY_READ_BUFFER	缓冲区用作通过 <code>glCopyBufferSubData</code> 进行复制的数据源
GL_COPY_WRITE_BUFFER	缓冲区用作通过 <code>glCopyBufferSubData</code> 进行复制的目标
GL_ELEMENT_ARRAY_BUFFER	索引数组缓冲区用于保存 <code>glDrawElements</code> 、 <code>glDrawRangeElements</code> 和 <code>glDrawElementsInstanced</code> 的索引
GL_PIXEL_PACK_BUFFER	<code>glReadPixels</code> 之类像素包装操作的目标缓冲区
GL_PIXEL_UNPACK_BUFFER	<code>glTexImage1D</code> 、 <code>glTexImage2D</code> 、 <code>glTexImage3D</code> 、 <code>glTexSubImage1D</code> 、 <code>glTexSubImage2D</code> 和 <code>glTexSubImage3D</code> 之类纹理更新函数的源缓冲区
GL_TEXTURE_BUFFER	着色器可以通过纹理单元拾取来访问的缓冲区
GL_TRANSFORM_FEEDBACK_BUFFER	变换反馈顶点着色器（transform feedback vertex shader）写入的缓冲区
GL_UNIFORM_BUFFER	着色器能够访问的 Uniform 值

要绑定一个缓冲区以备使用，我们可以以这个缓冲区名称为参数，以表 8.1 中列出的缓冲区为目标来调用 `glBindBuffer`。接下来我们将新的缓冲区绑定到像素包装缓冲区绑定点，这样就能使用 `glReadPixels` 将像素数据复制到缓冲区中了。

```
glBindBuffer(GL_PIXEL_PACK_BUFFER, pixBuffObjs[0]);
```

要从一个绑定中对一个缓冲解除绑定，我们可以再次调用以 0 为缓冲区名称、目标与上述调用相同的 `glBindBuffer`。我们还可以只是将另外一个合法的缓冲区绑定到同一个目标上。

当我们用完一个缓冲区之后，这个缓冲区需要进行清除，就像所有其他 OpenGL 对象一样，通过调用 `glDeleteBuffers` 来删除它。按照一般的做法，在进行删除之前，我们要确保缓冲区没有被绑定到任何绑定点。

```
glDeleteBuffers(1, pixBuffObjs);
```

## 8.1.2 填充缓冲区

创建和删除缓冲实际上是同一件事。但是，我们如何将合法的数据传递到缓冲区来使用它？将数据填充到缓冲区有很多种方法，其中一些更加复杂的我们将在后续章节中介绍。我们可以使用 `glBufferData` 函数来简单地将数据直接上传到任何类型的缓冲区中。

```
glBindBuffer(GL_PIXEL_PACK_BUFFER, pixBuffObjs[0]);  
glBufferData(GL_PIXEL_PACK_BUFFER, pixelDataSize, pixelData, GL_DYNAMIC_COPY);  
glBindBuffer(GL_PIXEL_PACK_BUFFER, 0);
```

在调用 `glBufferData` 之前，我们必须将要使用的缓冲区进行绑定。对 `glBufferData` 使用的目标与我们为第一个参数绑定缓冲区时使用的目标相同。第二个参数是我们将要上传的数据大小，以字节（byte）为单位，而第 3 个参数则是将要被上传的数据本身。请注意，如果我们想要分配一个特定大小的缓冲区却不需要立即对它进行填充，那么这个指针也可能是 `NULL`。`glBufferData` 的第 4 个参数用来告诉 OpenGL 我们打算如何使用缓冲区。

为使用方式选择正确的值需要一些小技巧。表 8.2 列出了可能使用方式的选项。使用方式的值实际上只是一个性能提示，来帮助 OpenGL 驱动程序在正确的位置分配内存。例如，如果应用程序需要频繁地进行读取，那么某些能够很容易被 CPU 访问的内存将会是非常好的选择。而另外一些内存可能不能直接被 CPU 访问，但是可以被 GPU 很快地访问。提前将使用计划告诉 OpenGL 驱动程序，这样缓冲区就可以被分配在一个更便于我们使用的位置了。

表 8.2 缓冲区对象使用方式

缓冲区使用方式	描 述
GL_STREAM_DRAW	缓冲区的内容将由应用程序进行一次设置，并且经常用于绘制
GL_STREAM_READ	缓冲区的内容将作为一条 OpenGL 命令的输出来进行一次设置，并且经常用于绘制
GL_STREAM_COPY	缓冲区的内容将作为一条 OpenGL 命令的输出来进行一次设置，并且不经常用于绘制或复制到其他图像
GL_STATIC_DRAW	缓冲区的内容将由应用程序进行一次设置，并且经常用于绘制或复制到其他图像

续表

缓冲区使用方式	描 述
GL_STATIC_READ	缓冲区的内容将作为一条 OpenGL 命令的输出来进行一次设置, 并且由应用程序进行多次查询
GL_STATIC_COPY	缓冲区的内容将作为一条 OpenGL 命令的输出来进行一次设置, 并且经常用于绘制或复制到其他图像
GL_DYNAMIC_DRAW	缓冲区的内容将会经常由应用程序进行更新, 并且经常用于绘制或复制到其他图像
GL_DYNAMIC_READ	缓冲区的内容将作为 OpenGL 命令的输出经常进行更新, 并且由应用程序进行多次查询
GL_DYNAMIC_COPY	缓冲区的内容将作为 OpenGL 命令的输出经常进行更新, 并且经常用于绘制或复制到其他图像

在我们不确定缓冲区的用途时, 对于通常的缓冲区使用方式或条件来说, 使用 GL\_DYNAMIC\_DRAW 是一个比较安全的值。我们总是可以再次调用 glBufferData 对缓冲区重新进行填充, 还可以改变使用方式的提示。但是如果我们真的再次调用了 glBufferData, 那么缓冲区中原来的所有数据都将被删除。

我们可以使用 glBufferSubData 对已经存在的缓冲区中的一部分进行更新, 而不会导致缓冲区其他部分的内容变为无效。

```
void glBufferSubData(GLenum target, intptr offset, sizeiptr size, const void *data);
```

glBufferSubData 的大多数参数和 glBufferData 的相应参数相同。新的 offset 参数允许我们从除开头部分以外的其他位置开始更新。我们也不能改变缓冲区的使用方式, 因为内存已经被分配了。

### 8.1.3 像素缓冲区对象

图形方面的许多最新和最重要的进步都与那些以更快和更高效的方式完成某些原有操作的新方法有关。在存储像素/纹理单元方面, 像素缓冲区对象与纹理缓冲区对象非常相似。和所有缓冲区对象一样, 它们都存储在 GPU 内存中。我们可以访问和填充像素缓冲区对象 (或者缩写为 PBO), 方法和任何其他缓冲区对象类型一样。实际上, 只有在绑定到一个 PBO 缓冲区绑定点时, 一个缓冲区才真正成为一个像素缓冲区对象。

第一个像素缓冲区对象绑定点是 GL\_PIXEL\_PACK\_BUFFER。当一个像素缓冲区对象被绑定到这个目标上时, 任何读取像素的 OpenGL 操作都会从像素缓冲区对象中获得它们的数据。这些操作包括 glReadPixels、glGetTexImage 和 glGetCompressedTexImage。通常这些操作会从一个帧缓冲区或纹理中抽取数据, 并将它们读回到客户端内存中。当一个像素缓冲区对象被绑定到包装缓冲区时, 像素数据在 GPU 内存中的像素缓冲区对象中的任务就结束了, 而并不会下载到客户端。

第二个 PBO 绑定点是 GL\_PIXEL\_UNPACK\_BUFFER。当一个像素缓冲区对象被绑定到这个目标时, 任何绘制像素的 OpenGL 操作都会向一个绑定的像素缓冲区对象中放入它们的数据。

glTexImage\*D、glTexSubImage\*D、glCompressedTexImage\*D 和 glCompressedTexSubImage\*D 就是这样的操作。这些操作将数据和纹理从本地 CPU 内存中读取到帧缓冲区中。但是当一个像素缓冲区

对象作为解包缓冲区被绑定时，会使读取操作成为 GPU 内存中而不是 CPU 内存中的 PBO。

为什么要干涉这些像素缓冲区对象 (PBO) 呢？毕竟没有它们我们也可以在 GPU 中读、写和移动像素。对初学者来说，任何从 PBO 中读取或写入 PBO 中的调用或者任何缓冲区对象都用管线进行处理。这就意味着 GPU 不需要完成所有其他工作，只要对数据复制进行初始化，等待复制完成，然后继续运行就行了。

因为缓冲区对象没有这样的顺序问题，它们在处理需要经常访问、修改或更新像素数据有着巨大的优势。下面来看一些例子。

- **流纹理更新**——在某些情况下，应用程序可能会需要在每一帧中都对一个纹理进行更新。也许需要根据用户输入来改变它，或者也可能会想要流视频。像素缓冲区对象允许应用程序改变纹理数据，而不必先下载然后再重新上传整个表面。
- **渲染顶点数据**——因为缓冲区对象是通用的数据存储，应用程序可以很容易地使用同一个缓冲区来达到不同的目的。例如，一个应用程序可以将顶点数据写入一个颜色缓冲区，然后再将这些数据复制到 PBO。一旦完成，缓冲区就能够作为一个顶点缓冲区进行绑定，并且用来绘制新的几何图形。这种方式展示了灵活性，允许我们对新的顶点数据“染色”！
- **异步调用 glReadPixels**——应用程序经常需要从屏幕上剥离像素，执行一些操作，然后或者保存它们，或者使用它们重新进行绘制。遗憾的是，将像素数据读取到 CPU 内存中需要 GPU 完成正在进行的其他所有工作，然后执行复制后，才能开始其他工作或者返回实际调用。如果未来的绘制调用要依赖读取或多次读取的结果呢？如果我们试图保持 GPU 忙于绘制所有 3D 图形，使用 glReadPixels 可以真正地解决问题。像素缓冲区对象 (PBO) 派上用场了。因为读取操作是通过管线进行的，所以对 glReadPixels 的调用能够立即返回。我们甚至可以进行多次调用，使用不同的缓冲区对象来读取不同的区域。

像素缓冲区对象是一个很好的容器，可以暂时存储 GPU 本地像素数据，但是请记住，在使用它们之前需要先为它们分配存储空间。和其他所有缓冲区对象一样，要调用 glBufferData 为这个缓冲区分配内存空间并用数据填充。但是，我们没有必要提供数据，为数据指针传递 NULL 可以简单地分配内存而不进行填充。如果我们试图填充存储空间之前不对它进行分配，那么 OpenGL 将抛出一个错误。

```
glBufferData(GL_PIXEL_PACK_BUFFER, pixelDataSize, pixelData, GL_DYNAMIC_COPY);
```

像素缓冲区经常用来存储来自一个渲染目标的 2D 图像、纹理或其他数据源。但是缓冲区对象本身是一维的，它们本质上没有宽度或高度。在为 2D 图像分配存储空间时，我们可以只是将宽度与高度相乘，再与像素大小相乘。对于存储像素数据没有其他需要补充的了，但是缓冲区可以比给定的数据组所需的大小更大。

实际上，如果我们计划为多种数据大小使用相同的 PBO，最好马上关闭对数据大小上限的设定，而不是频繁地对它进行重新设定。

所有对 glBufferData 的调用都和其他的绘制调用一起通过管线进行处理。这就意味着，OpenGL 实现不需要等待所有活动停止，就可以将新数据发送到 GPU。这一点在某些情况下可能尤其重要。

设想一下我们在玩喜爱的游戏时，在所有这些情况下都必须等待几分钟才能进入下一关，而这其中就有一部分时间是用来上传大量新的纹理数据的情况吧。或者想象一下在我们（在游戏中）进入一个新房间

总要停顿一下，等待纹理数据更新的情况。PBO 能够以一种不拖延其他工作的方式在必要时提供纹理数据，这样可以帮助解决一些这类问题。

### 从缓冲区中读取像素数据

当绘制内容在屏幕上显示时，我们可能会需要在这些像素彻底消失之前再次取回。这样做的原因之一是检查实际的渲染情况，以便确定接下来在屏幕上如何进行渲染。另外一个原因则是在应用到后续帧的效果中使用前面帧的像素。无论是什么原因，`glReadPixels` 函数都能发挥作用。

这个函数从当前启用的读取缓冲区的特定位置获取像素，然后将它们复制到 CPU 内存中。

```
void* data = (void*)malloc(pixelDataSize);
glReadBuffer(GL_BACK_LEFT);
glReadPixels(0, 0, GetWidth(), GetHeight(), GL_RGB, GL_UNSIGNED_BYTE, pixelData);
```

当我们向客户端内存进行写入时，整个管线经常需要被清空，以保证所有会影响我们将要进行读取的绘制工作能够完成。这对于应用程序性能可能会有很大的冲击。但是，值得庆幸的是，我们可以使用缓冲区对象来克服这种性能问题。在调用 `glReadPixels` 并将 `glReadPixels` 调用中的数据指针设为 1 之前，我们可以将一个缓冲区对象绑定到 `GL_PIXEL_PACK_BUFFER` 上。这样就能够将像素重定向到 GPU 中的一个缓冲区中，并且避免了复制到客户端内存上可能带来的性能问题。

```
glReadBuffer(GL_BACK_LEFT);
glBindBuffer(GL_PIXEL_PACK_BUFFER, pixBuffObjs[0]);
glReadPixels(0, 0, GetWidth(), GetHeight(), GL_RGB, GL_UNSIGNED_BYTE, NULL);
```

在我们的第一个示例应用程序中，这两种方法都得到了应用。

### 使用 PBO

在应用程序中添加 PBO 可能很简单，但却能带来明显的性能提升。本章的第一个示例程序能够完成几项工作，但是更重要的是它能够演示 PBO 的真正效率。

运动模糊 (Motion blur) 是一种特效，它有助于显示出一个场景中哪些对象是运动的，以及它们运动得有多快。我们可能已经在电影、电视或视频中见过这些模糊的对象了。当一个物体以一个相对于单帧快门速度来说过高的速度经过照相机时，这一帧和相邻帧的图片上就会沿着运动的方向出现一片模糊的痕迹。在照相机相对于一个对象或者整个场景快速移动时也会出现同样的效果。想象一下在高速公路上行驶的汽车中一位乘客从汽车中向侧面拍摄照片的场景吧。

在 OpenGL 中有很多复杂的方法可以用来创建这样的效果。有一个应用程序可以在一个缓冲区中进行多次渲染，将快速移动的对象稍稍进行偏移并将得到的结果混合到一起。另一个选择是为一个物体的图像沿着运动方向对纹理单元像素数据进行多次采样，然后将采样结果混合到一起。甚至还有更加复杂的方法，使用深度缓冲区数据对更加靠近照相机的对象应用更加戏剧性的模糊效果。

对于示例应用程序来说，我们使用另一种简单的方法，将前面帧的结果进行存储并与当前帧混合到一起。为了创建可视的运动模糊，这个程序将会存储最后的 5 帧图像。这个程序既可以使用老式的方式将

数据复制到 CPU 中再复制出来，也可以使用更快的 PBO 方式。

首先，在程序清单 8.1 中，我们对必要的纹理和 PBO 进行了初始化。

程序清单 8.1 为 pix\_buffs 示例程序设置 PBO 和纹理

```
// 创建模糊纹理
glGenTextures(6, blurTextures);

// 分配一个像素缓冲区来对纹理和 PBO 进行初始化
pixelDataSize = GetWidth()*GetHeight()*3*sizeof(unsigned byte);
void* data = (void*)malloc(pixelDataSize);
memset(data, 0x00, pixelDataSize);

// 为模糊特效设置 6 个纹理单元
// 初始化纹理数据
for (int i=0; i<6;i++)
{
    glActiveTexture(GL_TEXTURE1+i);
    glBindTexture(GL_TEXTURE_2D, blurTextures[i]);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, GetWidth(), GetHeight(), 0, GL_RGB,
                GL_UNSIGNED_BYTE, data);
}

// 为像素的复制分配空间，以免我们在每次进行绘制时都调用函数原形
glGenBuffers(1, pixBuffObjs);
glBindBuffer(GL_PIXEL_PACK_BUFFER, pixBuffObjs[0]);
glBufferData(GL_PIXEL_PACK_BUFFER, pixelDataSize, pixelData, GL_DYNAMIC_COPY);
glBindBuffer(GL_PIXEL_PACK_BUFFER, 0);
```

当所有的资源都设置完成后，场景将被渲染到后台缓冲区，就好像没发生什么特别的事情一样。得到的结果将被复制到一个用于模糊特效的纹理中，而不是简单地调用交换。在传统的方式中，这些工作是通过调用 `glReadPixels` 来获取像素数据，然后调用 `glTexImage2D` 将像素数据复制到一个纹理对象上完成。数据的纹理目标将在 6 个模糊纹理之间进行轮换。如果上一次使用的是纹理 3，那么接下来用的将是纹理 4。这就意味着纹理 4 将包含来自这一帧的数据，纹理 3 包含来自上一帧的数据，而纹理 2 则包含来自再前面一个帧的数据，以此类推。在使用最后一个纹理之后，当前帧的目标将再次进行循环。最后 6 个帧的纹理数据在这个“纹理环缓冲区”（texture ring buffer）中总是按顺序排列并且是可用的。

PBO 方式则稍有不同。在这种方式下，我们不是将数据复制回 CPU，而是将 PBO 绑定到 `GL_PIXEL_PACK_BUFFER`，而在调用 `glReadPixels` 时，这些像素将被重新定向到 PBO 而不是再次回到 CPU。然后，同一个缓冲区将从 `GL_PIXEL_PACK_BUFFER` 绑定点上解除绑定，然后再绑定到 `GL_PIXEL_UNPACK_BUFFER`。在接下来调用 `glTexImage2D` 时，缓冲区中的像素数据将被加载到纹理上，它们不会离开 GPU，也不会与其他命令一起保持在管线中。

我们可以在程序清单 8.2 中看到这个处理过程。最后，这个环形缓冲区将进行更新，指向下一个模糊纹理。在程序运行时可以按 P 键来在两种方式之间切换。

程序清单 8.2 在对场景进行渲染之后，将结果复制到最近的纹理对象。

```
if (bUsePBOPath)
{
    // 首先将 PBO 绑定为包装缓冲区
```

```

// 然后直接将像素读取到 PBO
glBindBuffer(GL_PIXEL_PACK_BUFFER, pixBuffObjs[0]);
glReadPixels(0, 0, GetWidth(), GetHeight(), GL_RGB,
             GL_UNSIGNED_BYTE, NULL);
glBindBuffer(GL_PIXEL_PACK_BUFFER, 0);

// 接下来将 PBO 绑定为解包缓冲区
// 然后直接将像素读取到 PBO, 然后将像素放入纹理中
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pixBuffObjs[0]);

// 为新的模糊设置纹理, 每一帧都有增加
glActiveTexture(GL_TEXTURE0+GetBlurTarget0());
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB8, GetWidth(), GetHeight(),
             0, GL_RGB, GL_UNSIGNED_BYTE, NULL);
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);
}
else
{
    // 获取屏幕像素并复制到客户端内存
    glReadPixels(0, 0, GetWidth(), GetHeight(), GL_RGB,
                GL_UNSIGNED_BYTE, pixelData);
    // 将像素从客户端内存中移到纹理中
    // 为新的模糊设置纹理单元, 每一帧都有增加
    glActiveTexture(GL_TEXTURE0+GetBlurTarget0());
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB8, GetWidth(), GetHeight(),
                0, GL_RGB, GL_UNSIGNED_BYTE, pixelData);
}

// 使用模糊着色器和所有模糊纹理进行全屏绘制
projectionMatrix.PushMatrix();
projectionMatrix.LoadIdentity();
projectionMatrix.LoadMatrix(orthoMatrix);
modelViewMatrix.PushMatrix();
modelViewMatrix.LoadIdentity();
glDisable(GL_DEPTH_TEST);
SetupBlurProg(); // 这个程序将所有纹理一起进行模糊
screenQuad.Draw();
glEnable(GL_DEPTH_TEST);
modelViewMatrix.PopMatrix();
projectionMatrix.PopMatrix();

// 为下一帧转向下一个模糊纹理
AdvanceBlurTarget();

```

为了进行实际的模糊操作, 片段着色器将从所有 6 个纹理中进行采样并对结果进行平均。已经给定所有纹理大小都相同, 并且需要与其他纹理进行排列, 因此片段着色器只需要一组纹理坐标。程序清单 8.3 显示了完成所有 6 个纹理采样的着色器代码。这个着色器用于对屏幕对齐的四边形进行渲染, 这些四边形是通过建立一个基于窗口宽度和高度的正模型视图投影矩阵进行设置的。正交矩阵创建一个变换, 将坐标直接映射到屏幕空间。我们在几何图形  $x$  坐标上每增加一个单位, 就要向屏幕右方多移动一个像素。在  $y$  方向向上移动一个单位, 相当于在屏幕上向上移动一个像素。结果得到的是 2D 渲染, 几何图形的坐标也是屏幕上的像素位置。

程序清单 8.3 片段着色器 blur.fs

```

// blur.fs
// 输出 4 个纹理经过加权和混合的结果
//
#version 150
in vec2 vTexCoord;

```

```

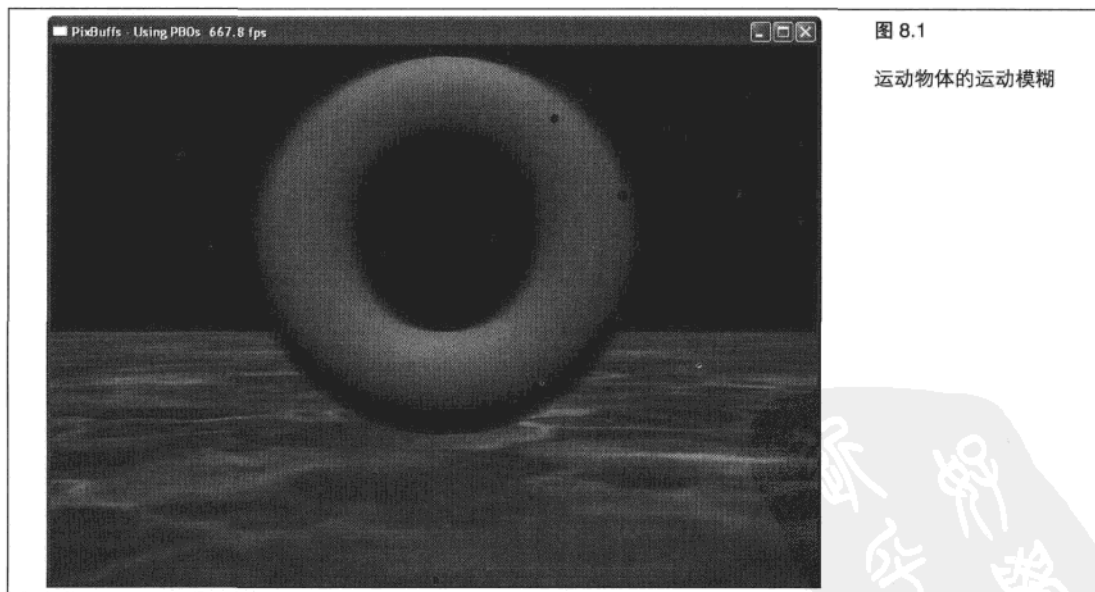
uniform sampler2D textureUnit0;
uniform sampler2D textureUnit1;
uniform sampler2D textureUnit2;
uniform sampler2D textureUnit3;
uniform sampler2D textureUnit4;
uniform sampler2D textureUnit5;

void main(void)
{
    // 0 是最新的图像，而 5 是最老的
    vec4 blur0 = texture(textureUnit0, vTexCoord);
    vec4 blur1 = texture(textureUnit1, vTexCoord);
    vec4 blur2 = texture(textureUnit2, vTexCoord);
    vec4 blur3 = texture(textureUnit3, vTexCoord);
    vec4 blur4 = texture(textureUnit4, vTexCoord);
    vec4 blur5 = texture(textureUnit5, vTexCoord);

    vec4 summedBlur = blur0 + blur1 + blur2 + blur3 + blur4 + blur5;
    gl_FragColor = summedBlur/6.0;
}

```

在第一次启用 `pix_buffs` 时，程序将使用客户端内存路径来加载混合纹理。在对象从一端到另一端移动时，请注意混合是如何只出现在运动的轴上的。我们可以在图 8.1 所示看到这种效果，并且在彩图 11 中也进行了展示。按 P 键可以开启和关闭 PBO 路径。我们可以使用数字键盘上的+键和-键来对物体的运动进行加速和减速。在速度改变时，注意运动模糊的量是如何随之改变的。



这个程序的速度显示在了标题栏上。请注意客户端复制和 PBO 复制之间的性能差异——差异非常大！在较慢的系统中，PBO 的路径几乎比客户端内存路径快 6 倍。我们多希望程序运行速度能够快 6 倍啊！请注意将数据进行移动是如何帮助实现这一点的。巨大的性能提升是缓冲区对象现在成为 OpenGL 程序的重要部分的原因之一。

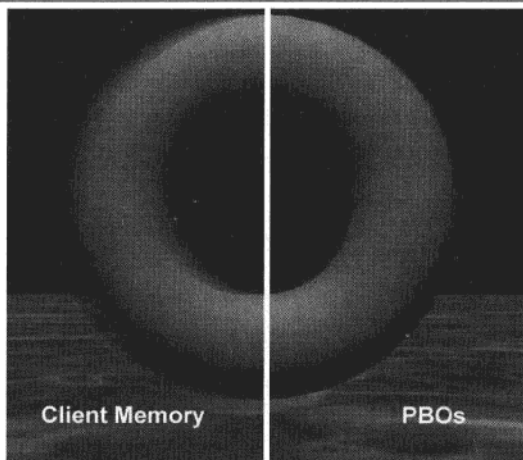
当切换到 PBO 路径时，模糊的量减少了。发生这种情况是因为采样程序使用最后 5 帧来创建混合的

输出，无论程序的运行速度多快。在使用 PBO 时，最后 5 帧看起来互相靠近了很多（因为更快的渲染速度允许更高的帧速率），也就产生了更少的模糊。让我们看一看图 8.2，可以尝试修改程序，为 PBQ 路径创建更多的模糊，或者修改程序使模糊无论在选择什么路径时都是相同的。

另外一种很好的实践是尝试不同的应用运动模糊的方法，或者在对帧纹理进行组合时使用加权值。

图 8.2

GPU 复制和客户端复制之间  
模糊的不同



#### 8.1.4 缓冲区对象

我们已经了解了某些缓冲区绑定目标（如 `GL_PIXEL_PACK_BUFFER` 和 `GL_COPY_READ_BUFFER`）是如何应用于从一个缓冲区（当它在 GPU 中时）中更新或获取数据的。其他缓冲区绑定诸如 `GL_TEXTURE_BUFFER`、`GL_TRANSFORM_FEEDBACK_BUFFER` 和 `GL_UNIFORM_BUFFER` 允许缓冲区在渲染管线中直接进行使用。这些绑定点中有一些将在随后几章中进行探索，但是现在，让我们先来看看缓冲区对象是如何在纹理上直接使用的。

一个纹理包含两个主要组成部分：纹理采样状态和包含纹理值的数据缓冲区。现在我们也可以将一个缓冲区对象绑定到 `GL_TEXTURE_BUFFER` 缓冲区中一个纹理的绑定。读者可能会问：“为什么还要进行另外的纹理绑定？”这是一个很好的问题！纹理缓冲区也称为 texBO 或 TBO，允许我们完成一些传统纹理不能完成的工作。首先，纹理缓冲区能够直接填充来自其他渲染结果（例如变换反馈、像素读取操作或顶点数据）的数据。这样就节省了不少时间，因为应用程序能够直接从着色器以前的渲染调用中获取像素数据。

texBO 的另一个特性是宽松的大小限制。纹理缓冲区与传统的一维纹理相似，但要更大。OpenGL 规范所规定的纹理缓冲区大小的最大值比 1D 纹理大 64 倍，但是在某些实现中纹理缓冲区的大小可能要大几万倍！

那么有了这些 texBO 我们能做什么呢？对于初学者来说，所有类型的着色器数学运算都曾经非常困难（如果不是不可能的话）。TexBO 向着色器提供了大量对多种不同格式和类型的数据访问，允许着色器以通常是预留 CPU 的方式进行操作。纹理缓冲区能够用来提供对片段着色器和顶点着色器中

的顶点数组的访问。这在着色器需要关于临近几何图形的信息以作出运行时决策和计算的情况下可能会非常有用。但是为了做到这一点，我们常常需要将 texBO 大小作为一个统一值传递到着色器中。

纹理缓冲区是作为普通的缓冲区来创建的，当它被绑定到一个纹理或者 GL\_TEXTURE\_BUFFER 绑定点时会成为真正的纹理缓冲区。

```
glBindBuffer(GL_TEXTURE_BUFFER, texBO[0]);
glBufferData(GL_TEXTURE_BUFFER, sizeof(float)*count, fileData,
             GL_STATIC_DRAW);
```

但是 texBO 必须绑定到一个纹理单元上才能真正变得有用。要将一个 texBO 绑定到一个纹理上，可以调用 glTexBuffer，但首先要确保要使用的纹理已经进行了绑定。

```
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_BUFFER, texBOTexture);
glTexBuffer(GL_TEXTURE_BUFFER, GL_R32F, texBO[0]);
```

虽然纹理缓冲区对象看起来和操作起来都很像普通纹理，但是还有一些重要的不同点。纹理缓冲区不能在着色器中用普通的采样器——也就是 sampler1D 和 sampler2D 进行访问。取而代之的是，我们必须使用一个新的采样器，叫做 samplerBuffer。因为采样器类型不同，用来从纹理缓冲区中获取值的采样函数也不相同。我们可以使用 texelFetch 从纹理缓冲区进行读取。

```
uniform samplerBuffer lumCurveSampler;
void main(void) {
    ....
    int offset = int(vColor.r * (1024-1));
    lumFactor.r = texelFetch(lumCurveSampler, offset ).r;
}
```

当着色器在一个纹理缓冲区中进行查询时，它必须使用一个基于整数的非标准化索引。诸如 texture 这样的传统采样函数接受的坐标范围为从 0.0 到 1.0，但是 texBO 查询函数 texelFetch 则接受一个从 0 到缓冲区大小值的整数索引。如果纹理查询坐标已经进行了标准化，那么我们可以通过乘以 texBO 的大值减去 1 的结果，然后再将得到的结果转化为整数的方式转换成索引。

## 8.2 帧缓冲区对象，摆脱窗口的限制

在大多数人提起 3D 渲染时，首先想到的第一件事就是一个 3D 游戏或者计算机辅助设计程序的屏幕输出内容，毕竟观看互动的 3D 输出结果正是大多数使用者所追求的。但是，除了渲染到窗口或者渲染到全屏之外，OpenGL 还允许我们去做了很多事情。一个 OpenGL 窗口的表面长期以来一直被称作“帧缓冲区”。但是现在 OpenGL 将绘制帧缓冲区到一个对象所需要的状态进行了封装，称为帧缓冲区对象(FBO)。

默认的帧缓冲区对象是与创建的 OpenGL 窗口相关联的，并且在一个新的环境被绑定时自动进行绑定。我们可以创建多个帧缓冲区对象，也叫做 FBO，并且直接渲染一个 FBO 而不是窗口。使用这种离屏渲染，应用程序就可以执行许多不同种类的渲染算法了，例如阴影贴图、应用辐射着色(radiosity)、反射、后期处理和许多其他特效。另外，帧缓冲区对象并不受窗口大小的限制，它可以包含多个颜色缓冲区。我们甚至可以将纹理绑定到一个 FBO 上，这就意味着可以直接渲染到一个纹理中。

虽然帧缓冲区的名称中包含一个“缓冲区”字眼，但是其实他们根本不是缓冲区。实际上，并不存在与一个帧缓冲区对象相关联的真正内存存储空间。相反，帧缓冲区对象是一种容器，它可以保存其他确实有内存存储并且可以进行渲染的对象，例如纹理或渲染缓冲区。采用这种方式，帧缓冲区对象能够在保存 OpenGL 管线的输出时将需要的状态和表面绑定到一起。

## 8.2.1 如何使用 FBO

创建和设置一个新的 FBO 非常简单，但是别忘了 FBO 只是一个图像对象的容器。所以在需要先添加图像，才能渲染到一个 FBO。一旦一个 FBO 被创建、设置和绑定，大多数 OpenGL 操作就将像是在渲染到一个窗口一样执行，但是输出结果将存储在绑定到 FBO 的图像中。

### 创建新的 FBO

要创建 FBO，首先要生成 FBO 缓冲区名称。我们可以同时生成任意数量的名称。

```
GLuint fboName;
glGenFramebuffers(1, &fboName);
```

然后再绑定一个新的 FBO 来修改和使用它。

```
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fboName);
```

同一时间只有一个 FBO 可以绑定用来进行绘制，并且同一时间只有一个 FBO 可以绑定来进行读取。在绑定一个帧缓冲区时，glBindFramebuffer 的第一个参数既可以是 GL\_DRAW\_FRAMEBUFFER，也可以是 GL\_READ\_FRAMEBUFFER。这就意味着我们可以使用一个帧缓冲区进行读取，而使用另一个不同的缓冲区进行绘制。在本章的第一个程序中，我们就看到了一个例子。将名称 0 绑定到任意一个 FBO 目标，都会将当前的缓冲区解除绑定，并再次绑定到默认 FBO。一旦默认 FBO 被绑定，那么读取和写入就都再次绑定到了窗口的帧缓冲区。

### 销毁 FBO

在使用完 FBO，或者在退出前进行清除时，要删除 FBO。

```
glDeleteFramebuffers(1, &fboName);
```

## 8.2.2 渲染缓冲区对象

现在我们可以与 FBO 进行交互了，我们需要将一些东西放到 FBO 中。渲染缓冲区对象，或者简称 RBO，是一种图像表面，它是专门为了绑定到 FBO 而设计的。一个渲染缓冲区对象可以是一个颜色表面、模板表面或者深度 / 模板组合表面。我们可以为给定的 FBO 挑选需要的任意 RBO 组合。实际上，我们

甚至可以一次性绘制很多个颜色缓冲区！创建 RBO 和创建 FBO 以及大多数其他 OpenGL 对象非常类似。

```
glGenRenderbuffers(3, renderBufferNames);
```

和 FBO 类似，RBO 需要先进行绑定才能修改。绑定渲染缓冲区的唯一合法目标是 GL\_RENDERBUFFER。

```
glBindRenderbuffer(GL_RENDERBUFFER, renderBufferNames[0]);
```

现在 RBO 已经绑定，需要分配支持 RBO 的内存空间。RBO 在创建时是没有初始存储的。没有存储，我们就没有任何东西可渲染。

首先，我们要确定应用程序需要什么。然后，选择合适的格式，使其与缓冲区的用途相匹配。大多数合法的纹理格式同时也是合法的渲染缓冲区格式。另外，我们可以创建包含一个模板格式的渲染缓冲区存储。纹理可以有一个 DEPTH\_STENCIL 组合格式，但不只是一个模板格式。

```
glBindRenderbuffer(GL_RENDERBUFFER, renderBufferNames[0]);  
glRenderbufferStorage(GL_RENDERBUFFER, GL_RGBA8, screenWidth, screenHeight);  
glBindRenderbuffer(GL_RENDERBUFFER, depthBufferName);  
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT32, screenWidth,  
                      screenHeight);
```

在前面的示例中，RBO 存储被分配了与示例程序窗口相同的大小。但是，渲染缓冲区的大小其实不必与窗口相同。我们可以通过调用以 GL\_MAX\_RENDERBUFFER\_SIZE 为参数的 glGetIntegerv 来查询 OpenGL 实现所支持的最大维度，宽度和高度值必须小于这个最大值。创建存储的唯一有效目标是 GL\_RENDERBUFFER。

我们还可以使用一个类似的函数 glRenderbufferStorageMultisample 来创建多重采样的渲染缓冲区存储，这个函数接受一个附加的采样变量。这样做的好处是，可以在任何像素显示在屏幕上之前进行离屏多重采样。

## 绑定 RBO

一旦我们为 FBO 创建了所有的渲染表面，就到了将它们衔接起来的时候了。一个帧缓冲区有多个绑定定点可以进行绑定：一个深度绑定、一个模板绑定、以及多个颜色绑定。我们可以使用 glGetIntegerv 查询 GL\_MAX\_COLOR\_ATTACHMENTS，来查出一次可以绑定多少颜色缓冲区。在示例应用程序中，一次就使用了一个深度缓冲区和 3 个颜色缓冲区。在试图绑定一个渲染缓冲区之前，要确保 FBO 已经被绑定。

```
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fboName);  
glFramebufferRenderbuffer(GL_DRAW_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,  
                           GL_RENDERBUFFER, depthBufferName);  
glFramebufferRenderbuffer(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,  
                           GL_RENDERBUFFER, renderBufferNames[0]);  
glFramebufferRenderbuffer(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT1,  
                           GL_RENDERBUFFER, renderBufferNames[1]);  
glFramebufferRenderbuffer(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT2,  
                           GL_RENDERBUFFER, renderBufferNames[2]);
```

第一个参数可以是 GL\_DRAW\_FRAMEBUFFER 或 GL\_READ\_FRAMEBUFFER，这取决于我们将 FBO 绑定到了哪里，然后再指定绑定。第 3 个参数总是 GL\_RENDERBUFFER，而最后一个参数则是

将要使用的着色器缓冲区名称。

如果我们以 0 为名称调用了 `glFramebufferRenderbuffer`，那么无论绑定到当前 FBO 指定绑定点的是何种缓冲区，它都将被解除绑定。有一个特殊的绑定点 `GL_DEPTH_STENCIL_ATTACHMENT`，它允许我们将同一个缓冲区同时绑定到深度绑定点和模板绑定点。要使用这个绑定点，我们需要创建一个带有内部 `GL_DEPTH_STENCIL` 格式的 RBO。

除非我们突然出现了什么疯狂的想法，否则就无法改变默认帧缓冲区的绑定。同样也不能将默认帧缓冲区的一个表面绑定到用户生成的帧缓冲区上。

## RBO 大小

帧缓冲区对象在衔接上有着惊人的灵活性。我们可以将不同颜色格式的渲染缓冲区绑定到同一个帧缓冲区上。实际上，我们甚至可以将不同大小的 RBO 绑定到同一个帧缓冲区上。如果我们的 RBO 确实大小不同，那么只能渲染到一个大小为最小缓冲区的矩形上。这种灵活性比我们想象中更加有用。例如，深度缓冲区可能占据一定的空间。如果我们有多个 FBO 或多个缓冲区需要进行深度测试，那么可以创建一个深度缓冲区，并在所有 FBO 或者渲染传递中使用它，在每次使用的间隔清除深度值。我们要做的就是确保分配一个深度格式 RBO，它要足够大，足以容纳最大的 FBO 配置。

## 8.2.3 绘制缓冲区

现在我们已经知道了如何将整个批次的渲染缓冲区绑定到一个帧缓冲区，最好确保它们都能够立即使用！要获得对渲染缓冲区的访问，有两个重要步骤。第一步是确保片段着色器设置正确，第二步是确保输出被引导到正确的位置。

### 着色器输出

为了将颜色输出到多个缓冲区，着色器必须配置为写入多重颜色输出（write multiple color outputs）。更好的情况是，写入到每个缓冲区的值可以是不同的，否则还有什么意义呢？从着色器写入颜色输出的一种方法是写入到名为 `gl_FragData[n]` 的内建输出中。但是我们不能在同一个着色器中使用 `gl_FragData[n]` 和 `gl_FragColor`，`n` 的值是着色器的输出索引。程序清单 8.4 完整地列出了第一个采样程序的片段着色器。这里使用了 3 个颜色输出，其中每个输出都使用了一种各不相同的着色技术。

程序清单 8.4 `fbo_drawbuffers` 的片段着色器 `multibuffer.fs`

```
// multibuffer.fs
// 输出到 3 个缓冲区: normal color, grayscale 和 luminance adjusted color
#version 150

in vec4 vFragColor;
in vec2 vTexCoord;

uniform sampler2D textureUnit0;
uniform int bUseTexture;
```

```

uniform samplerBuffer lumCurveSampler;

void main(void) {
    vec4 vColor;
    vec4 lumFactor;
    if (bUseTexture != 0)
        vColor = texture(textureUnit0, vTexCoord);
    else
        vColor = vFragColor;

    // 对第一个缓冲区进行原样输出
    gl_FragData[0] = vColor;

    // 对第二个缓冲区进行灰度输出
    float grey = dot(vColor.rgb, vec3(0.3, 0.59, 0.11));
    gl_FragData[1] = vec4(grey, grey, grey, 1.0f);

    // 对输入颜色进行截取，确保它的值在 0.0 和 1.0 之间
    vColor = clamp(vColor, 0.0f, 1.0f);

    int offset = int(vColor.r * (1024 - 1));
    lumFactor.r = texelFetch(lumCurveSampler, offset).r;

    offset = int(vColor.g * (1024 - 1));
    lumFactor.g = texelFetch(lumCurveSampler, offset).r;

    offset = int(vColor.b * (1024 - 1));
    lumFactor.b = texelFetch(lumCurveSampler, offset).r;

    lumFactor.a = 1.0f;
    gl_FragData[2] = lumFactor;
}

```

## 缓冲区映射

现在大家已经了解了着色器将要写入什么样的输出，我们需要告诉 OpenGL 要输出到哪里。我们已经看到了多重缓冲区是如何绑定到一个 FBO 的，以及着色器是如何写入到不同的输出索引的。OpenGL 允许一个应用程序通过为每个缓冲区指定颜色绑定来将着色器输出映射到不同的 FBO 缓冲区。

默认的行为是一个单独的颜色输出将被发送到颜色绑定 0。如果不告诉 OpenGL 如何处理着色器输出，那么只有第一个输出被路由通过，即使我们有多着色器输出和多个颜色缓冲区绑定到帧缓冲区对象上也是如此。

我们可以通过调用 `glDrawBuffers` 来对着色器输出进行路由（route）。这将覆盖以前所有的映射，即使指定的映射比上一次少时也是如此。

```

GLenum fboBufs[] = { GL_COLOR_ATTACHMENT0,
                     GL_COLOR_ATTACHMENT1,
                     GL_COLOR_ATTACHMENT2 };
glDrawBuffers(3, fboBufs);

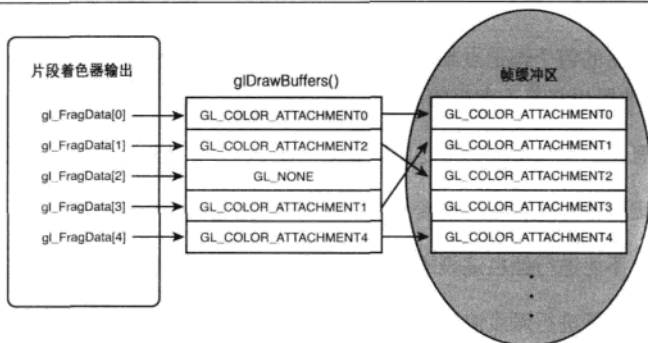
```

第二个参数是指向一个 `GLenum` 数组的指针，这个数组指定了着色器输出索引值将要路由到的颜色绑定。图 8.3 所示展示了着色器输出是如何映射到实际缓冲区的。数组的索引会被传递到着色器输出索引对应的 `glDrawBuffers`。大多数情况下我们可能只想要一个一对一的映射，其中着色器输出的索引与颜色

绑定的偏移量相等。

图 8.3

将着色器输出映射到实际缓冲区



我们仍然要在调用 `glDrawBuffers` 之前确保 FBO 已经被绑定。如果在一个由用户创建的 FBO 被绑定的情况下使用 `glDrawBuffers`，那么合法的缓冲区目标为从 `GL_COLOR_ATTACHMENT0` 到“1 减去最大值”的结果，或者为 `GL_NONE`。但是如果默认的 FBO 被绑定，那么可以使用与窗口相关联的颜色缓冲区名称，最普遍的是 `GL_BACK_LEFT`。请注意，无论使用的 FBO 是何种类型，在数组中除 `GL_NONE` 以外的值最多只能使用一次。如果默认帧缓冲区被绑定，或者着色器程序写入到 `gl_FragColor`，那么我们传递到 `glDrawBuffers` 的所有缓冲区都会获得相同的颜色。不要忘记在使用完 FBO 之后恢复绘制缓冲区的设置，否则将会产生 GL 错误。

```
GLenum windowBuff [] = { GL_FRONT_LEFT };
glDrawBuffers(1, windowBuff);
```

当然，我们没有理由一定要将来自 `gl_FragData[0]` 的颜色输出映射到 `GL_COLOR_ATTACHMENT0`。我们可以以我们希望的方式进行混合，或者如果不需要片段着色器的某个输出的话，可以在绘制缓冲区列表设置一个到 `GL_NONE` 的入口。在如图 8.3 所示的示例映射中，第一个着色器输出被路由到第一个 FBO 颜色缓冲区绑定，而第二个着色器输出则被路由到第 3 个颜色缓冲区绑定。第 3 个着色器没有被路由到任何缓冲区，而第 4 个颜色缓冲区没有接受任何着色器输出。我们可以通过 `glDrawBuffers` 设置一个映射数量的限制值。我们可以通过调用以 `GL_MAX_DRAW_BUFFERS` 为参数的 `glGetIntegerv` 查询支持的最大映射数。

使用 `glDrawBuffers` 来选择着色器将要写入的缓冲区对于读取缓冲区绑定来说没有任何影响。我们可以通过调用 `glReadBuffer` 设置读取缓冲区，其值与 `glDrawBuffers` 中使用的值相同。

## 8.2.4 帧缓冲区的完整性

在使用完帧缓冲区对象之后，还有最后一个重要的主题。我们对设置 FBO 的方式很满意并不意味着 OpenGL 实现已经做好了进行渲染的准备。查明我们的 FBO 是否已经正确设置，并且实现可以通过某种方式来使用它的唯一方法就是检查帧缓冲区的完整性。

帧缓冲区的完整性在概念上与纹理的完整性类似。如果一个纹理没有包含所有的大小、格式等都正确的指定 Mip 贴图层次，那么这个纹理就是不完整的，也就无法使用。完整性有两种类型：绑定完整性和整个帧缓冲区的完整性。

## 绑定完整性

一个 FBO 的每个绑定点都必须符合某种被认为是完整的标准。如果任意一个绑定点是不完整的，那么整个帧缓冲区也将是不完整的。部分导致绑定不完整的情况如下。

- 没有任何图像关联到绑定的对象。
- 绑定图像的宽度或者高度为 0。
- 一个不能进行颜色渲染的格式被绑定到一个颜色绑定上。
- 一个不能进行深度渲染的格式被绑定到一个深度绑定上。
- 一个不能进行模板渲染的格式被绑定到一个模板绑定上。

## 帧缓冲区的整体完整性

不只是每个绑定点都必须有效并且符合特定标准，帧缓冲区对象作为一个整体也必须是完整的。默认的帧缓冲区（如果存在的话）总是完整的。整个帧缓冲区不完整的情况如下。

- 没有任何图像被绑定到帧缓冲区。
- `glDrawBuffers` 被映射到一个没有任何图像进行绑定的 FBO 绑定。
- 内部格式的组合不被支持。

## 检查帧缓冲区

当我们认为已经完成了 FBO 设置时，可以调用：

```
GLenum fboStatus = glCheckFramebufferStatus(GL_DRAW_FRAMEBUFFER);
```

来检查它是否完整。

如果 `glCheckFramebufferStatus` 返回 `GL_FRAMEBUFFER_COMPLETE`，那么一切正常，我们可以使用 FBO 了。返回值 `glCheckFramebufferStatus` 则会提示哪里出了问题，导致帧缓冲区不完整。表 8.3 描述了所有可能的返回条件和它们的意义。

表 8.3 帧缓冲区完整性返回值

返回值	描述
GL_FRAMEBUFFER_UNDEFINED	当前绑定 FBO 为 0，但不存在默认帧缓冲区
GL_FRAMEBUFFER_COMPLETE	一个有用户定义的 FBO 被绑定并且是完整的，可以进行渲染
GL_FRAMEBUFFER_INCOMPLETE_ATTACHMENT	为了进行渲染而启用的某个缓冲区不完整
GL_FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT	没有任何缓冲区被绑定到 FBO

续表

返回值	描 述
GL_FRAMEBUFFER_INCOMPLETE_DRAW_BUFFER	没有任何缓冲区被绑定到为了进行渲染而启用的某个缓冲区绑定
GL_FRAMEBUFFER_INCOMPLETE_READ_BUFFER	没有任何缓冲区被绑定到为了进行读取而启用的某个缓冲区绑定
GL_FRAMEBUFFER_UNSUPPORTED	内部格式的组合不被支持
GL_FRAMEBUFFER_INCOMPLETE_MULTISAMPLE	采样数或者为所有缓冲区设置的 TEXTURE_FIXED_SAMPLE_LOCATIONS 值不匹配
GL_FRAMEBUFFER_INCOMPLETE_LAYER_TARGETS	颜色绑定不都是层次纹理，或者没有都绑定到同一个目标

这些返回值中有很多在进行排错调试时都很有用，但在应用程序发布之后用处就不大了。无论如何，第一个示例应用程序还是进行检查来确保不出现上述那些情况框。在使用 FBO 的应用程序中进行这些检查是值得的，可以确保我们的用例不会超出某些与实现相关的限制。

下面的代码就是一个这方面的例子。

```

GLenum fboStatus = glCheckFramebufferStatus(GL_DRAW_FRAMEBUFFER);
if(fboStatus != GL_FRAMEBUFFER_COMPLETE)
{
    switch (fboStatus)
    {
    case GL_FRAMEBUFFER_UNDEFINED:
        // 哎呀，没有窗口？
        break;
    case GL_FRAMEBUFFER_INCOMPLETE_ATTACHMENT:
        // 检查每个绑定的状态
        break;
    case GL_FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT:
        // 将至少一个缓冲区绑定到 FBO
        break;
    case GL_FRAMEBUFFER_INCOMPLETE_DRAW_BUFFER:
        // 检查所有通过
        // glDrawBuffers 启用的绑定在 FBO 中都存在
    case GL_FRAMEBUFFER_INCOMPLETE_READ_BUFFER:
        // 检查所有通过 glReadBuffer 指定的缓冲区在 FBO 中都存在
        break;
    case GL_FRAMEBUFFER_UNSUPPORTED:
        // 重新考虑用于绑定缓冲区的格式
        break;
    case GL_FRAMEBUFFER_INCOMPLETE_MULTISAMPLE:
        // 确保每个绑定的采样数量相同
        break;
    case GL_FRAMEBUFFER_INCOMPLETE_LAYER_TARGETS:
        // 确保每个绑定的层次数量相同
        break;
    }
}

```

如果我们在一个不完整的 FBO 被绑定时试图执行任何从帧缓冲区进行读取或者写入帧缓冲区的命令，那么这个命令将在抛出一个 GL\_INVALID\_FRAMEBUFFER\_OPERATION 错误后返回，这个错误可以通过调用 glGetError 获取。

## 读取帧缓冲区也需要是完整的

在前面的示例中，我们测试了绑定到缓冲区绑定点 `GL_DRAW_FRAMEBUFFER` 的 FBO。但是绑定到 `GL_READ_FRAMEBUFFER` 的帧缓冲区也必须是绑定完整和帧缓冲区整体完整的，这样才能进行读取。因为同一时间只能启用一个读取缓冲区，这样确保一个 FBO 对于读取操作是完整的就相对简单一些。

### 8.2.5 在帧缓冲区中复制数据

渲染到这些离屏缓冲区很棒，但是最终还要利用这些结果做一些有用的工作。传统上图形 API 允许一个应用程序将像素或者缓冲区数据读回到系统内存中，也提供了将它们写回屏幕的方法。当这些方法起作用时，它们需要从 GPU 中将数据复制到 CPU 内存，然后再返回来将它复制回去。这样是非常低效的！现在有一种方法，使用 `blit` 命令可以将像素数据从一点移动到另外一个点。`blit` 是一个术语，它代表直接、有效的 bit 级数据/内存复制。关于这个术语的由来有很多种说法，但最可信的说法是 bit 级图像传输（Bit-Level-Image-Transfer）或块传输（Block-Transfer）。不管 `blit` 这个术语在语言上的由来如何，实际的操作是相同的。执行这些复制是简单的，使用的函数如下所示。

```
void glBlitFramebuffer(GLint srcX0, GLint srcY0, GLint srcX1, GLint srcY1,
                       GLint dstX0, GLint dstY0, GLint dstX1, GLint dstY1,
                       GLbitfield mask, GLenum filter);
```

虽然这个函数的名称里包含“blit”字眼，但是它不只是能够完成简单的逐位复制而已。实际上，它更像一种自动的纹理操作。复制的源是通过调用 `glReadBuffer` 而指定的读取缓冲区，而复制的区域则是由以 `(srcX0, srcY0)` 和 `(srcX1, srcY1)` 为顶点的矩形所定义的区域。类似地，复制的目标是通过调用 `glDrawBuffer` 而指定的当前绘制缓冲区，复制到的区域则是由以 `(dstX0, dstY0)` 和 `(dstX1, dstY1)` 为顶点的矩形所定义的区域。因为源和目标区域的矩形不需要是同样大小的，所以可以使用这个函数对被复制的像素进行缩放。如果我们将读取和绘制缓冲区设置为同一个 FBO，并将同一个 FBO 绑定到了 `GL_DRAW_FRAMEBUFFER` 绑定和 `GL_READ_FRAMEBUFFER` 绑定，那么我们甚至可以将数据从一个缓冲区的一部分复制到另一个缓冲区。

蒙板参数（`mask` argument）可以是 `GL_DEPTH_BUFFER_BIT`、`GL_STENCIL_BUFFER_BIT` 或 `GL_COLOR_BUFFER_BIT` 中的任何一个或者全部。过滤器可以是 `GL_LINEAR` 或 `GL_NEAREST`，但是如果复制深度或模板数据时则必须是 `GL_LINEAR`。这些过滤器的行为和进行渲染时是一样的。对于例子来说，我们只复制颜色数据，可以使用线性过滤器。

```
GLint width = 800;
GLint height = 600;
GLenum fboBufs[] = { GL_COLOR_ATTACHMENT0 };
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fboName);
glBindFramebuffer(GL_READ_FRAMEBUFFER, fboName);
glDrawBuffers(1, fboBufs);
glReadBuffer(GL_COLOR_ATTACHMENT0);
glBlitFramebuffer(0, 0, width, height,
                  (width * 0.8), (height * 0.8), width, height,
                  GL_COLOR_BUFFER_BIT, GL_LINEAR );
```

假定前面代码中绑定到 FBO 绑定点的 RBO 的宽度和高度分别为 800 和 600。这些代码创建了整个缓冲区的一个缩小为总大小 20% 的副本, 并将它放置在右上角。

## 8.2.6 FBO 综合运用

我们的第二个示例应用程序将 FBO、RBO、texBO、帧缓冲区和更多其他内容结合在一起。这个模型非常简单, 但是其中所有渲染都是使用一个片段着色器 (参见程序清单 8.4) 一次性完成的。为了捕捉所有这些输出, 我们使用一个带有一个深度缓冲区和 3 个颜色缓冲区的 FBO。程序清单 8.5 进行了这些设置。

程序清单 8.5 创建并设置一个包含 4 个绑定的 FBO

```
// 创建新的 FBO
glGenFramebuffers(1, &fboName);

// 创建深度渲染缓冲区
glGenRenderbuffers(1, &depthBufferName);
glBindRenderbuffer(GL_RENDERBUFFER, depthBufferName);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT32, screenWidth,
                      screenHeight);

// 创建 3 个颜色缓冲区
glGenRenderbuffers(3, renderBufferNames);
glBindRenderbuffer(GL_RENDERBUFFER, renderBufferNames[0]);
glRenderbufferStorage(GL_RENDERBUFFER, GL_RGBA8, screenWidth, screenHeight);
glBindRenderbuffer(GL_RENDERBUFFER, renderBufferNames[1]);
glRenderbufferStorage(GL_RENDERBUFFER, GL_RGBA8, screenWidth, screenHeight);
glBindRenderbuffer(GL_RENDERBUFFER, renderBufferNames[2]);
glRenderbufferStorage(GL_RENDERBUFFER, GL_RGBA8, screenWidth, screenHeight);

// 将所有 4 个渲染缓冲区绑定到 FBO
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fboName);
glFramebufferRenderbuffer(GL_DRAW_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                          GL_RENDERBUFFER, depthBufferName);
glFramebufferRenderbuffer(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                          GL_RENDERBUFFER, renderBufferNames[0]);
glFramebufferRenderbuffer(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT1,
                          GL_RENDERBUFFER, renderBufferNames[1]);
glFramebufferRenderbuffer(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT2,
                          GL_RENDERBUFFER, renderBufferNames[2]);

// 为处理过程设置缓冲区
processProg = gltLoadShaderPairWithAttributes("multibuffer.vs",
                                              "multibuffer.fs", 3, GLT_ATTRIBUTE_VERTEX,
                                              "vVertex", GLT_ATTRIBUTE_NORMAL,
                                              "vNormal", GLT_ATTRIBUTE_TEXTURE0, "vTexCoord0");

// 创建 3 个新的缓冲区对象
glGenBuffers(3, texBO);
glGenTextures(1, &texBOTexture);

int count = 0;
float* fileData = 0;

// 加载第一个类似正切曲线的 texBO, 1024 个值
fileData = LoadFloatData("LumTan.data", &count);
if (count > 0)
```

```

{
    glBindBuffer(GL_TEXTURE_BUFFER_ARB, texBO[0]);
    glBufferData(GL_TEXTURE_BUFFER_ARB, sizeof(float)*count,
                 fileData, GL_STATIC_DRAW);
    delete fileData;
}

// 加载第二个类似正弦曲线的 texBO, 1024 个值
fileData = LoadFloatData("LumSin.data", &count);
if (count > 0)
{
    glBindBuffer(GL_TEXTURE_BUFFER_ARB, texBO[1]);
    glBufferData(GL_TEXTURE_BUFFER_ARB, sizeof(float)*count,
                 fileData, GL_STATIC_DRAW);
    delete fileData;
}

// 加载第 3 个线性曲线的 texBO, 1024 个值
fileData = LoadFloatData("LumLinear.data", &count);
if (count > 0)
{
    glBindBuffer(GL_TEXTURE_BUFFER_ARB, texBO[2]);
    glBufferData(GL_TEXTURE_BUFFER_ARB, sizeof(float)*count,
                 fileData, GL_STATIC_DRAW);
    delete fileData;
}

// 首先加载正切曲线
glBindBuffer(GL_TEXTURE_BUFFER_ARB, 0);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_BUFFER_ARB, texBOTexture);
glTexBuffer(GL_TEXTURE_BUFFER_ARB, GL_R32F, texBO[0]);
glActiveTexture(GL_TEXTURE0);

// 重置帧着色器绑定
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);

```

程序清单 8.5 的第一部分对 FBO 和 RBO 进行了设置，并将它们组合在了一起。接下来我们使用 GLTools 库来创建着色器和程序，并对它们进行编译和连接。然后我们创建了 3 个缓冲区对象，并用来自离线文件的浮点数据对它们进行填充。这些文件中包含的数据是一些偏置曲线。其中一个为正弦偏置，一个是正切偏置，另外一个则是线性偏置。这些曲线都相互对照着绘制在了图 8.4 中，以便进行比较。

在纹理缓冲区被创建和加载之后，默认的帧缓冲区对象再次进行了重新绑定。

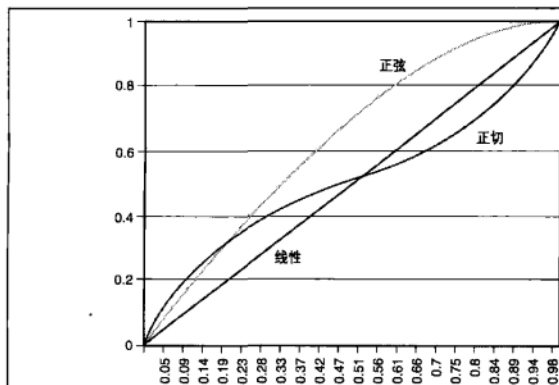


图 8.4

fbo\_drawbuffers 示例程序中偏置曲线的对比

程序清单 8.6 中展示的代码对使用多个着色器目标进行着色的程序所必需的全部着色器状态进行了设置。纹理缓冲区已经进行了加载，但是由于纹理缓冲区对象使用一个纹理单元来获取值，所以纹理缓冲区采样器 `lumCurveSampler` 必须设置到纹理缓冲区进行加载的纹理单元。

程序清单 8.6 为程序设置 OpenGL 状态

```
glUseProgram(processProg);
// 为顶点程序设置矩阵
glUniformMatrix4fv(glGetUniformLocation(processProg, "mvMatrix"),
    1, GL_FALSE, transformPipeline.GetModelViewMatrix());
glUniformMatrix4fv(glGetUniformLocation(processProg, "pMatrix"),
    1, GL_FALSE, transformPipeline.GetProjectionMatrix());

// 设置光源位置
glUniform3fv(glGetUniformLocation(processProg, "vLightPos"), 1, vLightPos);

// 为渲染的像素设置顶点颜色
glUniform4fv(glGetUniformLocation(processProg, "vColor"), 1, vColor);

// 为 texBO 获取设置纹理单元
glUniform1i(glGetUniformLocation(processProg, "lumCurveSampler"), 1);

// 如果几何图形已经进行了纹理贴图，则设置纹理单元
if(textureUnit != -1)
{
    glUniform1i(glGetUniformLocation(processProg, "bUseTexture"), 1);
    glUniform1i(glGetUniformLocation(processProg, "textureUnit0"),
        textureUnit);
}
else
{
    glUniform1i(glGetUniformLocation(processProg, "bUseTexture"), 0);
}
```

采样程序最后一个有趣的部分就是设置 FBO，指定绘制到哪个缓冲区，然后渲染场景。在程序清单 8.7 中，应用程序创建的 FBO 被绑定，而绘制缓冲区则被设置到前 3 个颜色绑定上。接下来将清除缓冲区，绑定处理程序，渲染场景。当渲染完成时，得到的结果通过 3 次调用 `glBlitFramebuffer` 来显示在窗口中。读取缓冲区则在每次调用时设置到适当的 FBO 绑定上。请注意 3 个输出是如何同时可见的，如图 8.5 所示，彩图 12 中也展示了这个图形。

程序清单 8.7 对 FBO 执行渲染，并复制到屏幕

```
GGLenum fboBufs[] = { GL_COLOR_ATTACHMENT0,
    GL_COLOR_ATTACHMENT1,
    GL_COLOR_ATTACHMENT2 };

glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fboName);
glDrawBuffers(3, fboBufs);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

UseProcessProgram(vLightTransformed, vFloorColor, 0);

floorBatch.Draw();
DrawWorld(yRot);

// 直接绘制到窗口
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
glDrawBuffers(1, windowBuff);
glViewport(0, 0, GetWidth(), GetHeight());
```

```
// 源缓冲区从帧缓冲区对象进行读取
glBindFramebuffer(GL_READ_FRAMEBUFFER, fboName);

// 将灰度输出复制到屏幕的左半边
glReadBuffer(GL_COLOR_ATTACHMENT1);
glBlitFramebuffer(0, 0, GetWidth()/2, GetHeight(),
                  0, 0, GetWidth()/2, GetHeight(),
                  GL_COLOR_BUFFER_BIT, GL_NEAREST);

// 将亮度调整颜色复制到屏幕的右半边
glReadBuffer(GL_COLOR_ATTACHMENT2);
glBlitFramebuffer(GetWidth()/2, 0, GetWidth(), GetHeight(),
                  GetWidth()/2, 0, GetWidth(), GetHeight(),
                  GL_COLOR_BUFFER_BIT, GL_NEAREST);

// 将原始图像复制到屏幕的右上方
glReadBuffer(GL_COLOR_ATTACHMENT0);
glBlitFramebuffer(0, 0, GetWidth(), GetHeight(),
                  (int)(GetWidth()*0.8), (int)(GetHeight()*0.8),
                  GetWidth(), GetHeight(),
                  GL_COLOR_BUFFER_BIT, GL_LINEAR );
```

按 F3、F4 和 F5 键可以切换应用到窗口右边的亮度曲线。处理着色器接受最终的颜色，使用纹理缓冲区对象的大小对颜色值进行缩放，然后查询新的 R、G 和 B 值，再将这些值存储到它们的颜色输出中。我们可以试着改变生成这些曲线的数据，增加自己的曲线，或者在每个颜色通道上应用不同的因子。

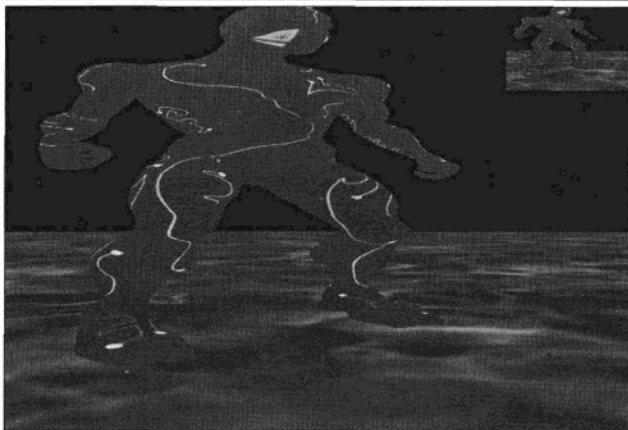


图 8.5

多个缓冲区同时进行绘制

可以按 F2 键在“向 FBO 渲染多个输出”和“直接渲染到屏幕上”这两者之间进行切换。不要忘记，绘制表面大小是根据绑定的表面而定的。就 fbo\_drawbuffers 采样程序的情况来说，渲染缓冲区的大小和窗口一致。所以无论它绘制到何处，缓冲区的大小都是相同的。即使在本例中并不是必需的，我们也要记住还必须通过调用 glViewport 来绘制到整个缓冲区（也不要过度绘制！），以改变视口大小。

## 8.3 渲染到纹理

好了，从传统的窗口渲染开始，我们已经走过了很长一段路程。FBO 是一种进行离屏渲染的灵活工具。

但是, RBO 本身也确实存在一些限制。实际上, 它们实在是只在直接绑定到 FBO 上时才是真正有用的。这就意味着获取数据需要一个副本, 而这正是我们想避免的。幸运的是, 我们不一定非要使用 RBO。

取而代之, 我们可以将一个纹理直接绑定到一个 FBO 绑定点上。因为各种纹理的情况不尽相同, 所以有 3 个入口点用来将纹理绑定到一个帧缓冲区绑定点上。

```
void glFramebufferTexture1D(GLenum target, GLenum attachment,
                           GLenum textarget, GLuint texture, GLint level);

void glFramebufferTexture2D(GLenum target, GLenum attachment,
                           GLenum textarget, GLuint texture, GLint level);

void glFramebufferTexture3D(GLenum target, GLenum attachment,
                           GLenum textarget, GLuint texture, GLint level,
                           GLint layer);
```

目标既可以是 GL\_DRAW\_FRAMEBUFFER, 也可以是 GL\_READ\_FRAMEBUFFER, 这与渲染缓冲区的情况相同。第二个参数指定 FBO 绑定点, 它可以是 GL\_DEPTH\_ATTACHMENT、GL\_STENCIL\_ATTACHMENT, 也可以是各个 GL\_COLOR\_ATTACHMENTn 值中的任意一个, 这一点也与渲染缓冲区的情况类似。对于大多数纹理来说, 第 3 个参数是相应的纹理类型, 但是对于立方体贴图来说, 我们需要传递表面的目标。接下来要给出纹理的名称, 然后是纹理要绑定到的 Mip 贴图层次。对于 glFramebufferTexture3D, 我们还必须指定 3D 纹理将要使用的层次。一维纹理只能通过 glFramebufferTexture1D 进行绑定, 而 glFramebufferTexture3D 则只能用于三维纹理。在二维纹理、矩形纹理和立方体贴图纹理中则使用 glFramebufferTexture2D。

在第 3 个示例程序 fbo\_textures 中, 我们使用一个绑定到 FBO 来为场景创建镜面效果的纹理。首先, 像第一个示例程序中一样设置 FBO。但是, 这一次程序会将一个纹理绑定到 FBO 上, 正如程序清单 8.8 所示。

程序清单 8.8 设置一个包含纹理绑定点的 FBO

```
// 创建并绑定一个 FBO
glGenFramebuffers(1, &fboName);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fboName);

// 创建深度渲染缓冲区
glGenRenderbuffers(1, &depthBufferName);
glBindRenderbuffer(GL_RENDERBUFFER, depthBufferName);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT32, 800, 800);

// 创建反射纹理
glGenTextures(1, &mirrorTexture);
glBindTexture(GL_TEXTURE_2D, mirrorTexture);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, 800, 800, 0, GL_RGBA, GL_FLOAT,
             NULL);

// 将纹理绑定到第一个颜色绑定点和深度 RBO
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                      GL_TEXTURE_2D, mirrorTexture, 0);
glFramebufferRenderbuffer(GL_DRAW_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                         GL_RENDERBUFFER, depthBufferName);
```

所有对新 FBO 的渲染都以与 RBO 绑定相同的方式完成。但是, 还有一些事情需要我们留意。因为现在绑定到 FBO 的纹理也可以用于渲染, 那么就有可能创建一个渲染循环。着色器可能会从一个纹理中获取纹理单元, 然后将最终的着色结果写回同一个纹理, 有可能会覆盖到同一个位置上。这样就可能导

致未定义的结果，这可能是一个非常难以查出的问题。作为一般的规则，我们最好确保绑定到 FBO 的纹理和被写入的纹理不要绑定到任何纹理单元上。

绑定到 FBO 的每个纹理的状态也会影响 FBO 的完整性。一个纹理图像表面的大小和格式可以在通过进行 `glTexImage2D` 之类的调用绑定到 FBO 的同时异步地进行修改。如果修改一个作为渲染目标的纹理图像表面，那么就应该确保帧缓冲区仍然可以通过调用 `glCheckFramebufferStatus` 进行渲染。我们可以绑定一个纹理的任何 Mip 贴图层次，只要在绑定纹理时指定它即可。如果我们还计划为纹理贴图使用 Mip 贴图，那么还要为刚刚进行渲染的纹理生成 Mip 链的剩余部分。这些工作并不是自动完成的，但是可以为我们想要更新的纹理类型调用 `glGenerateMipmap`，这样就使用基层的内容更新了所有基层以上的层次。

这种镜面效果并不神秘，如图 8.6 所示（同样显示在了彩图 13 中）。但是，FBO 能够获得真实的反射效果，这种效果用任何其他方法几乎都是不可能实现的。在本书前面的内容中，我们已经使用 alpha 混合和图像倒置模拟了大理石地板上的反射效果。这种 alpha 倒置效果在总是反射同一个角度

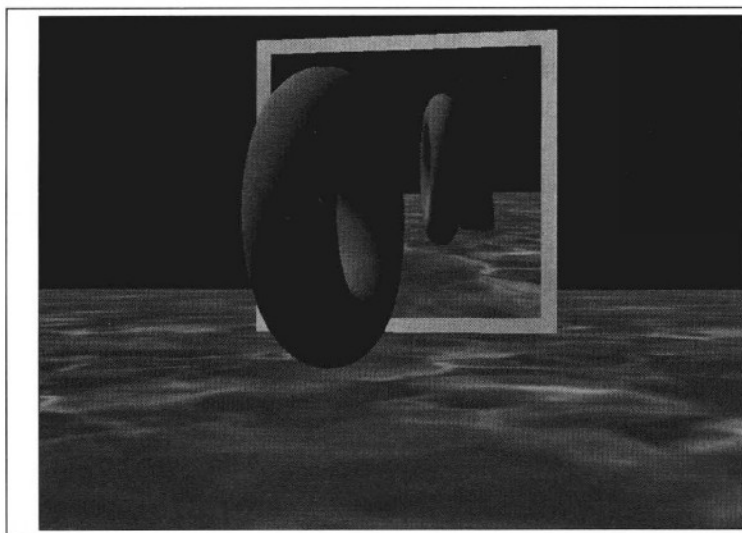


图 8.6

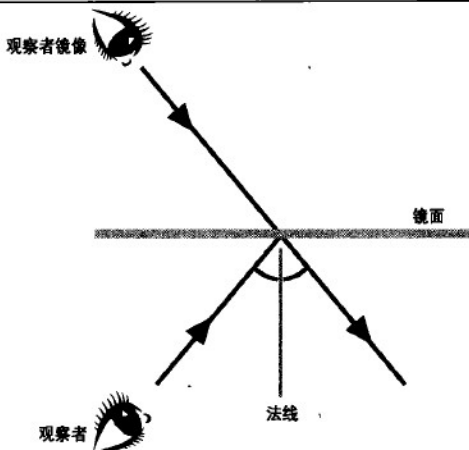
使用 FBO 和纹理创建准确的反射

的情况下使用时非常好。但是在大理石地板的边缘，在角度改变时，或者沿着一个陡坡向上走时就不那么好用了。在有障碍物（可能是一个盒子或者凳子）遮挡了反射路线时，这种方式也很难奏效。另外，使用 alpha 倒置方法可能还会导致深度测试问题。

使用一个 FBO，我们首先要将视角变换到镜子的视角上。这就意味着整个场景和所有内容都将从镜子的位置进行渲染。但是，镜子到底是“看向”哪个方向的呢？从镜子表面直接向外吗？这样就意味着当观察者相对于镜子移动时，反射也不会随之改变。那么回到观察者视角又如何呢？那就意味着我们总会看到自己的镜像，无论站在哪里都是如此，这样虽然接近，但还不是正确的结果。我们可以绘制一个从观察者位置指向镜面中心的向量，而镜面则应该是沿着这个向量通过垂直于镜面的法向量反射得到的向量向外看的。我们可以想象一下，自己站在镜子后面沿着反射的方向看镜子显示的情景。如图 8.7 所示，我们能找到一些关于这种效果如何实现的认识。不要忘记，这里的入射角并不是观察者/照相机所看向的方向，而是在观察者位置和镜面中心画一条线而得到的角度。

图 8.7

根据反射确定镜面的视角



一旦我们找到了镜子的位置，并且计算出镜面观察的角度，我们就可以从镜面的位置和视角来渲染场景了。程序清单 8.9 展示了这个场景是如何按照镜面的视角来绘制的。在 `fbo_textures` 示例程序中，我们使用 GLTools 资源中的一个 GLFrame 对象根据位置、向上的向量和观察方向产生对模型视图矩阵的调整。然后，模型视图矩阵将在  $x$  方向进行反转，来模拟反射——在镜子中所有物体都是反向的。作为奖励，观察者将在反射图像中被绘制成一个蓝色的锥体，以帮助我们想象照相机的位置。

程序清单 8.9 从镜面的视角进行绘制

```
// 设置镜面帧（照相机）的位置
vMirrorPos[0] = 0.0;
vMirrorPos[1] = 0.1f;
// 观察位置实际上是在镜面后的
mirrorFrame.SetOrigin(vMirrorPos);

// 计算镜面帧（照相机）的方向
// 因为我们知道镜面相对于原点的位置
// 通过向观察者的向量——原点添加镜面偏置来寻找方向向量
vMirrorForward[0] = vCameraPos[0];
vMirrorForward[1] = vCameraPos[1];
vMirrorForward[2] = (vCameraPos[2] + 5);
m3dNormalizeVector3(vMirrorForward);
mirrorFrame.SetForwardVector(vMirrorForward);

// 首先从镜面的视角进行渲染
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fboName);
glDrawBuffers(1, fboBufs);
glViewport(0, 0, mirrorTexWidth, mirrorTexHeight);

// 从镜像照相机的视角绘制场景
modelViewMatrix.PushMatrix();
M3DMatrix44f mMirrorView;
mirrorFrame.GetCameraMatrix(mMirrorView);
modelViewMatrix.MultMatrix(mMirrorView);

// 为了达到反射效果而对照相机进行水平反转
modelViewMatrix.Scale(-1.0f, 1.0f, 1.0f);
glBindTexture(GL_TEXTURE_2D, textures[0]); // Marble
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
shaderManager.UseStockShader(GLT_SHADER_TEXTURE_MODULATE,
    transformPipeline.GetModelViewProjectionMatrix(),
```

```

        vWhite, 0);
floorBatch.Draw();
DrawWorld(yRot);

// 现在绘制一个圆筒来代表观察者
M3DVector4f vLightTransformed;
modelViewMatrix.GetMatrix(mMirrorView);
m3dTransformVector4(vLightTransformed, vLightPos, mMirrorView);
modelViewMatrix.Translate(vCameraPos[0], vCameraPos[1]-0.8f,
                        vCameraPos[2]-1.0f);
modelViewMatrix.Rotate(-90.0f, 1.0f, 0.0f, 0.0f);

shaderManager.UseStockShader(GLT_SHADER_POINT_LIGHT_DIFF,
                            modelViewMatrix.GetMatrix(),
                            transformPipeline.GetProjectionMatrix(),
                            vLightTransformed, vBlue, 0);

cylinderBatch.Draw();
modelViewMatrix.PopMatrix();

```

接下来程序清单 8.10 中的代码将对场景进行绘制。帧缓冲区、视点和绘制缓冲区都恢复成进行窗口渲染的默认设置。然后再次绘制场景，这一次从观察者/照相机的视角进行绘制。上述工作完成之后，就可以进行镜面本身的绘制了。在应用包含镜面图像的纹理之前，示例程序会先确认观察者在镜面的哪一边。为了避免在镜面的背后出现不应出现的反射，程序会进行检查，确认观察者是在镜面的前面还是后面。如果观察者在后面，那么镜面将会绘制成黑色。

**程序清单 8.10 绘制场景的其他部分，包括镜面**

```

// 重置 FBO。再次从真实的照相机视角来绘制场景
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
glDrawBuffers(1, windowBuff);
glViewport(0, 0, screenWidth, screenHeight);
modelViewMatrix.PushMatrix();
M3DMatrix44f mCamera;
cameraFrame.GetCameraMatrix(mCamera);
modelViewMatrix.MultMatrix(mCamera);

glBindTexture(GL_TEXTURE_2D, textures[0]); // Marble
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
shaderManager.UseStockShader(GLT_SHADER_TEXTURE_MODULATE,
                            transformPipeline.GetModelViewProjectionMatrix(),
                            vWhite, 0);

floorBatch.Draw();
DrawWorld(yRot);

// 现在绘制镜面表面
modelViewMatrix.PushMatrix();
modelViewMatrix.Translate(0.0f, -0.4f, -5.0f);
if(vCameraPos[2] > -5.0)
{
    glBindTexture(GL_TEXTURE_2D, mirrorTexture); // Reflection
    shaderManager.UseStockShader(GLT_SHADER_TEXTURE_REPLACE,
                                transformPipeline.GetModelViewProjectionMatrix(), 0);
}
else
{
    // 如果观察者在镜面后面，那么镜面将会绘制成黑色
    shaderManager.UseStockShader(GLT_SHADER_FLAT,
                                transformPipeline.GetModelViewProjectionMatrix(), vBlack);
}

```

```
mirrorBatch.Draw();
shaderManager.UseStockShader(GLT_SHADER_FLAT,
    transformPipeline.GetModelViewProjectionMatrix(), vGrey);
mirrorBorderBatch.Draw();
modelViewMatrix.PopMatrix();
```

这种方法还有一些限制因素。其中一个因素是，整个场景将会进行两次绘制，如果在镜面后面或者视野之外有很多物体，这在性能上就可能成为一个问题。在第 12 章，我们将学习封闭查询（occlusion query），以及如何做到只绘制将要看到的内容。另一个问题是，在示例程序中观察者在镜面中的位置取得是镜面中心，这样做是为了使数学计算变得简单。但这样做的结果并不是非常真实的。要生成更加准确的反射，一种可能的方法是使用镜面上距离照相机最近的点。作为练习，我们可以试着去修改应用程序，使镜面也可以在场景中旋转。在这种情况下，我们就要根据照相机相对于镜面的位置和镜面相对于照相机的角度来重新计算入射角了。

## 8.4 小结

在本章，我们在管理 OpenGL 内存和缓冲区的方式上作出了较大的改变。使用像素缓冲区对象，可以将数据在 GPU 和管线数据加载之间来回移动，来完成纹理更新之类的操作。我们可以使用纹理缓冲区对象将任意数据绑定到纹理单元，然后在着色器中获取这些数据。

帧缓冲区对象与渲染缓冲区对象配合使用，而纹理则为离屏渲染带来了全新的可能性。现在，我们可以直接绑定和渲染到任何像素表面，而不会影响到屏幕上的内容。我们还学到了如何使用着色器来同时对大量颜色表面进行绘制。



## 第9章 高级缓冲区：超越基础水平

作者：Nicholas Haemel

## 本章内容

任 务	使用的函数
使用自定义片段着色器输出绑定	<code>glBindFragDataLocation / glBindFragDataLocationIndexed</code>
更新已存在的缓冲区	<code>glMapBuffer/glMapBufferRange/glUnmapBuffer</code>
加载压缩纹理	<code>glCompressedTexSubImage2D</code>
在缓冲区对象之间进行数据复制	<code>glCopyBufferSubData</code>
渲染到多个纹理	<code>glRenderBufferStorageMultisample</code>
在多个缓冲区中获取样本	<code>glGetMultisamplefv</code>
使用浮点缓冲区进行完整意义上的渲染	

现在帧缓冲区已经有些过时了，我们可以利用 FBO、纹理和缓冲区对象带来的灵活性来真正提升 OpenGL 管线表现。到目前为止，大部分工作都是针对传统的 8 位纹理和渲染缓冲区的，甚至是深度缓冲区也将所有值映射到 24 位或 32 位物理定点范围（physical fixed-point range）。

新的数据格式带来了新的局面，允许应用程序存储片段着色器的实际输出而不会丢失精度。有趣的事还不止这些。

OpenGL 还提供了许多访问和更新 GPU 中缓冲区的方法，这些方法无需中止渲染。

## 9.1 获得数据

本章的大部分篇幅都用于介绍所有新的数据格式和使用方法。但是在介绍它们之前，先来回顾一下第 8 章学到的一些内容，并且介绍一些用来访问那些将要进行性能优化的缓冲区的重要方法。

### 9.1.1 映射缓冲区

在前一章，我们使用 `glBufferData` 来上传缓冲区对象，从而对缓冲区进行填充。但是，如果在缓冲区已经加载到 GPU 之后还需要对缓冲区内容进行修改或更新的话，又该怎么办呢？好吧，这就是 `glMapBuffer` 和 `glMapBufferRange` 的作用了。当调用 `glMapBufferRange` 时，OpenGL 就会提供一个指向内存的指针，我们可以用这个指针直接读取或更新某个缓冲区中的数据。我们需要做的所有工作就是告诉实现我们打算对这些数据做什么。在 GPU 已经对缓冲区进行写入，而我们又希望将结果返回 CPU 时，可以选择只从映射的缓冲区中进行读取。或者在我们所做的改变已经在 GPU 内存存储的缓冲区中体现出来的情况下，也可以将缓冲区映射为写入状态。我们选择的映射类型将对性能产生影响，因此在只需要从某个缓冲区进行读取时，应当试图避免将这个缓冲区映射为写入状态。同样地，当只需要向一个缓冲区中写入数据时也不要将它映射为读取状态。表 9.1 列出了映射缓冲区可能的位字段（bitfield）值。

表 9.1 映射缓冲区访问类型

访问标记	用 途
GL_MAP_READ_BIT	返回可能用于读取缓冲区的指针
GL_MAP_WRITE_BIT	返回可能用于修改缓冲区的指针
GL_MAP_INVALIDATE_RANGE_BIT	表示 OpenGL 可以丢弃映射范围内以前的内容。范围内的数据将成为位定义的，直到应用程序进行更新
GL_MAP_INVALIDATE_BUFFER_BIT	表示 OpenGL 可以丢弃整个缓冲区中以前的内容。缓冲区内的数据将成为未定义的，直到应用程序进行更新
GL_MAP_FLUSH_EXPLICIT_BIT	以 GL_MAP_WRITE_BIT 方式使用这个位需要一个应用程序明确地对通过调用 <code>glFlushMappedBufferRange</code> 更新的范围进行清理。如果这个位没有被指定，那么整个缓冲区将在调用 <code>glUnmapBuffer</code> 时被清理
GL_MAP_UNSYNCHRONIZED_BIT	告诉 OpenGL 在进行映射之前不要试图对任何挂起的 GPU 写入这个缓冲区的操作进行同步

在完成映射缓冲区的更新之后，调用 `glUnmapBuffer` 来告诉 OpenGL 已经完成了这项工作。

```

Glint accessFlags = GL_MAP_WRITE_BIT | GL_MAP_INVALIDATE_RANGE_BIT |
                    GL_MAP_FLUSH_EXPLICIT_BIT;
Glint offset = 32 * 100;
Glint length = 32 * 48;
GLvoid *bufferData = glMapBufferRange(GL_TEXTURE_BUFFER, offset, length, access-
Flags);
// 在这里更新缓冲区
....
glUnmapBuffer(GL_TEXTURE_BUFFER);

```

如果我们设置了 `GL_MAP_FLUSH_EXPLICIT_FLAG`，那么就需要告诉 OpenGL 我们想要清理缓冲区的哪些部分，或者在对缓冲区解除映射之前通过调用 `glFlushMappedBufferRange` 更新了哪些部分。只需要，我们就可以任意多次地为更新的范围调用 `glFlushMappedBufferRange`。

```

GLvoid glFlushMappedBufferRange(GLenum target, intptr offset, sizeiptr length);

```

这里要使用与缓冲区绑定到相同的 `target`。 `offset` 和 `length` 参数用来标记缓冲区的哪部分进行改变。

我们还可以调用 `glMapBuffer` 代替 `glMapBufferRange` 来映射整个缓冲区。

```
GLvoid *bufferData = glMapBuffer(GL_TEXTURE_BUFFER, accessFlags);
```

在本书剩下的内容中会大量使用 `glMapBuffer` 和 `glMapBufferRange` 加载和更新 GPU 数据。

## 9.1.2 复制缓冲区

当数据被发送到 GPU 之后，我们完全有可能希望在缓冲区之间共享这些数据，或者从一个缓冲区中将结果复制到另一个缓冲区。幸运的是，OpenGL 也提供了一种使用起来很简单的方法，可以用来完成这项工作。`glCopyBufferSubData` 使得我们可以指定相关的缓冲区，以及使用的大小和偏置。

```
glCopyBufferSubData(GL_COPY_READ_BUFFER, GL_COPY_WRITE_BUFFER, readStart, writeStart, size);
```

我们可以从绑定到表 8.1 列出的任何绑定点的缓冲区中复制，也可以复制到这些缓冲区中。但是，既然缓冲区绑定点在同一时间只能绑定一个缓冲区，我们就无法在两个都绑定到 `GL_TEXTURE_BUFFER` 上的缓冲区之间进行复制了。OpenGL 的创造者也想到了这一点！还记得我们在第 8 章中第一次看到，但是到现在为止还没使用过的 `GL_COPY_READ_BUFFER` 和 `GL_COPY_WRITE_BUFFER` 吗？没错，这些绑定点就是为了将数据从一个缓冲区复制到另一个缓冲区而特别加入的。我们可以将读取和写入缓冲区绑定到这些绑定点，而不会影响到其他任何缓冲区绑定。然后再将偏置添加到每个缓冲区，并指定大小。

要确保我们从中读取和要写入到的范围保持在缓冲区的大小范围内，否则复制将会失败。`glCopyBufferSubData` 可以在许多巧妙的算法中使用。其中一种普遍的用法，就是为一个应用程序创建第二个 OpenGL 环境下的线程用于进行数据加载。在这里，`glCopyBufferSubData` 用来在主环境中更新几何图形数据非常方便，而不需要主线程中断渲染。

## 9.2 控制像素着色器表现，映射片段输出

在第 8 章，我们学习了如何将多个缓冲区对象连接到一个帧缓冲区中，以及对同一个片段着色器的许多不同输出进行渲染。为了完成这项工作，着色器可以写入到名为 `gl_FragData[n]` 的内建着色器输出，而不是写入到 `gl_FragColor`。

虽然我们仍然可以使用这两种输出中的任意一种来编译一个 GLSL 1.50 着色器，但这两种方式都已经不推荐使用了。这就意味着未来的 OpenGL 版本将会删除它们，而我们最好使用“更新更先进”的方法来写入着色器颜色输出。

使用内建着色器输出是 2006 年的做法了！老方法的一个问题就是，我们可以写入 `gl_FragData` 或 `gl_FragColor`，但是不能两者都写入。另外，如果着色器要渲染多个输出，那么它还必须包含硬编码的索引。还有，我们怎样才能保持跟踪并从逻辑上理解通过多个着色器写入到 `gl_FragData[7]` 的内容呢？我们可以定义着色器输出，而不是设置一个内建颜色输出索引的值。对于颜色输出，可以在片段着色器中将输出声明为 `out vec4`。我们重写第 8 章绘制缓冲区示例程序的输出来使用自定义位置。

```
out vec4 oStraightColor;
out vec4 oGreyscale;
out vec4 oLumAdjColor;
```

然后, 在对程序进行连接之前, 告诉 OpenGL 我们想要使用 `glBindFragDataLocation` 将输出映射到哪里。只要指定每个输出要映射到的索引即可。

```
glBindFragDataLocation(processProg, 0, "oStraightColor");
glBindFragDataLocation(processProg, 1, "oGreyscale");
glBindFragDataLocation(processProg, 2, "oLumAdjColor");
glLinkProgram(processProg);
```

我们还可以编译着色器, 将程序连接到一起, 然后指定输出的位置。请记住, 在使用程序之前要将其重新进行连接, 这样对输出位置的设置才能生效。现在着色器数据已经完成了配置, 每一种颜色都写入到唯一的索引。请记住, 我们不能将一个输出分配给多个索引。程序清单 9.1 完整地列出了第 8 章的绘制缓冲区示例程序中的片段着色器。这里声明了 3 个颜色输出, 其中每个输出都使用了一种各不相同的着色技术。

程序清单 9.1 fbo\_drawbuffers 的片段着色器 `multibuffer_frag_location.fs`

```
#version 150
// multibuffer_frag_location.fs
// 输出到 3 个缓冲区: normal color, grayscale 和 luminance adjusted color

in vec4 vFragColor;
in vec2 vTex;
uniform sampler2D textureUnit0;
uniform int bUseTexture;
uniform samplerBuffer lumCurveSampler;

out vec4 oStraightColor;
out vec4 oGrayscale;
out vec4 oLumAdjColor;

void main(void) {
    vec4 vColor;
    vec4 lumFactor;

    if (bUseTexture != 0)
        vColor = texture(textureUnit0, vTex);
    else
        vColor = vFragColor;

    // 对第一个缓冲区进行原样输出
    oStraightColor = vColor;

    // 对第二个缓冲区进行灰度输出
    float grey = dot(vColor.rgb, vec3(0.3, 0.59, 0.11));
    oGrayscale = vec4(grey, grey, grey, 1.0f);

    // 对输入颜色进行截取, 确保它的值在 0.0 和 1.0 之间
    vColor = clamp(vColor, 0.0f, 1.0f);

    int offset = int(vColor.r * 1024);
    oLumAdjColor.r = texelFetch(lumCurveSampler, offset).r;

    offset = int(vColor.g * 1024);
    oLumAdjColor.g = texelFetch(lumCurveSampler, offset).r;
```

```
offset = int(vColor.b * 1024);  
oLumAdjColor.b = texelFetch(lumCurveSampler, offset).r;  
oLumAdjColor.a = 1.0f;  
}
```

使用 `glBindFragDataLocation` 有很多优点。我们可以在着色器中为输出使用有实际含义的逻辑名，还可以在多个着色器中使用同样的名称，并在运行时将这个名称映射到正确的逻辑缓冲区索引。

我们将在第 10 章更加深入地了解应用程序如何使用混合。在 OpenGL 3.3 中，某些混合方程式需要一个着色器为每个片段输出两个不同的颜色。我们可以使用 `glBindFragDataLocationIndexed` 完成这项工作。

```
glBindFragDataLocationIndexed(program, colorNumber, index, outputName);
```

这个函数的行为和 `glBindFragDataLocation` 类似。在 OpenGL 3.3 中 `index` 参数有两个可能的索引值。如果选择 0，那么这个颜色将作为第一个输入颜色使用，就像我们已经使用了 `glBindFragDataLocation` 一样；如果选择 1，那么这个颜色将作为第二个输入颜色用于混合。

## 9.3 新一代硬件的新格式

在过去的几年中，OpenGL 取得进展的一种方式就是为大量新数据格式和数据类型增加本地支持。OpenGL 标准的撰写者不断地在 3D 应用程序开发中增加灵活性——最初是图形管线中可以完全自定义的部分，后来是灵活的缓冲区应用，而现在又增加了灵活的数据格式。

一开始，这种想法可能看起来很麻烦，或者无关紧要。但是任何花费时间试图将它们所有的颜色数据压缩到 8bit 之内的人都会赞同这个想法。进入 OpenGL 渲染管线的大部分数据都来自某些其他应用程序或工具。大多数游戏的顶点和纹理数据都来自诸如 Maya 或 3DS Max 这样的艺术创作工具。

CAD 程序使用复杂的引擎来根据用户输入和文件格式生成 3D 表面。因为顶点、纹理和相关数据可能会非常大、非常复杂，所以将所有这些数据从各种不同的源转换到一个小程序集合范围内几乎是不可能的。

现在大多数通用格式和许多非通用格式都得到了本地支持，所以对于 OpenGL 来说进行转换常常是不必要的。

### 9.3.1 浮点——最终的真正精确

增加浮点格式是最重要的增强中的一项。虽然在 OpenGL 管线内部经常使用浮点数据，但是源和目标却经常是定点数据，这使其精度大大降低。这样做的结果就是，管线的许多部分都曾经将所有值截取到 0 和 1 之间，这样它们就可以以定点格式进行存储了。在 OpenGL 3.2 中仍然允许我们将片段着色器的输出进行截取，而在 OpenGL 3.3 中就将截取操作整个删除了。

传递到一个顶点着色器的数据类型取决于我们，但是它典型情况下都会被声明为 `vec4`，或者一个包含 4 个浮点值的向量。类似地，当我们在一个顶点着色器中将变量声明为 `out` 或 `varying` 时，要决定顶点

着色器将写入何种输出。随后这些输出将在几何图形上进行光栅化，并传递到片段着色器。决定在整个管线中的颜色上使用的数据类型方面，我们拥有完全的控制权，虽然通常情况下使用的都是浮点数。现在，我们对数据从顶点数组到最终输出的整个过程采用的数据类型和传递方式都拥有完全的控制权。

这太棒了！现在，颜色和阴影可以使用  $1.18 \times 10^{-38}$  到  $3.4 \times 10^{38}$  之间的所有值，而不再只是 256 个值了！（负的颜色值没有意义）但是，如果我们绘制到一个每种颜色只有 8 位的窗口，会发生什么？遗憾的是，在这种情况下输出将被截取到 0 到 1 之间的范围内，然后再映射到一个定点值。这可不是好玩儿的！在有人发明能够懂得和显示浮点数据的显示器或监视器之前，我们仍然要受最终输出设备的限制。

但是这并不意味着浮点渲染没有用。恰恰相反！我们仍然可以采用全浮点精度渲染到纹理和渲染缓冲区。不仅如此，我们还可以对浮点数据如何映射到定点输出格式拥有完全的控制权。这对最终的结果有着重大影响，通常称为高动态范围（High Dynamic Range，缩写为 HDR）。

### 使用浮点格式

对应用程序进行升级使用浮点缓冲区比我们想象的要简单。实际上，我们甚至不需要调用任何新的函数。取而代之的是，可以在创建缓冲区时使用两个新的标记 GL\_RGBA16F 和 GL\_RGBA32F。我们可以在为 RBO（渲染缓冲区对象）创建存储或者分配纹理时使用它们。

```
glRenderbufferStorage(GL_RENDERBUFFER, GL_RGBA16F, nWidth, nHeight);
glRenderbufferStorage(GL_RENDERBUFFER, GL_RGBA32F, nWidth, nHeight);

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F, texWidth, texHeight, 0, GL_RGBA,
             GL_FLOAT, texels);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, texWidth, texHeight, 0, GL_RGBA,
             GL_FLOAT, texels);
```

除了更加传统的 RGBA 格式之外，表 9.2 列出了允许用于创建浮点渲染缓冲区的其他格式。纹理更加开放，并且可以采用更多的格式来创建，但是其中只有两种是浮点格式。还记得我们以前说过 OpenGL 变得更加灵活，能让许多不同的应用程序更容易地工作吗？有了这么多可用的浮点格式，应用程序就可以直接使用数据存储格式而不需要进行第一次转换了，而这种转换可能会非常耗费时间。

表 9.2 浮点渲染缓冲区和纹理格式

渲染缓冲区	纹 理
GL_RGBA32F	GL_RGBA32F
GL_RGBA16F	GL_RGBA16F
GL_R11_G11_B10F	GL_R11_G11_B10F
GL_RG32F	GL_RG32F
GL_RG16F	GL_RG16F
GL_R32F	GL_R32F
GL_R16F	GL_R16F
	GL_RGB32F
	GL_RGB16F

## HDR

许多现代游戏应用程序都使用浮点渲染来生成所有我们希望看到的漂亮视觉效果。如果能够产生诸如光源泛光 (light bloom)、镜头光晕 (lens flare)、光线反射 (light reflection)、光线折射 (light refraction)、朦胧光线 (crepuscular ray)、以及尘埃和云雾等非均匀介质效果,那么现实级的效果是可以实现的,而这些效果如果不使用浮点值通常是无法实现的。到浮点缓冲区的 HDR 渲染能够使明亮的区域真正地明亮起来,使阴影部分保持非常阴暗,同时又使我们仍然能够看清这两种区域中的细节。

毕竟,人的眼睛拥有不可思议的能力,可以感知非常高的对比度,远远超出了目前显示器的性能。为了简单起见,我们使用已经生成的 HDR 图像,而不再在示例程序中重新绘制一个包含许多几何图形和光照的复杂场景来展示 HDR 的效果了。第一个示例程序 `hdr_imaging` 加载了使用一种叫做 OpenEXR 的文件格式的 HDR (浮点) 图像。工业光学魔术公司 (Industrial Light and Magic, 简称工业光魔) 开发了 OpenEXR, 将它作为一种工具来帮助存储所有高保真图像处理所需的图像数据。我们可以将 OpenEXR 图像看作是一个由照相机在不同的曝光等级下拍摄的多幅图像的组合。低曝光能捕捉到场景中高光区域的细节,而高曝光则可以捕捉到场景中阴暗区域的细节。图 9.1 所示 (同样显示在了彩图 14 中) 展示了一个场景的 3 个视图,这个场景中的前景中有一颗树,背景则是明亮的视场。左边图像以非常低的曝光度进行渲染,即使在非常明亮的情况下,也能显示视场中所有的细节;中间图像开始显示前景,包括树干和最近的树上的树叶;右边图像则清晰地显示了树前面的地面细节,我们甚至可以看到树根部空洞中的景象!这几幅树的图片展示了存储在一幅单个图像中不可思议的丰富细节和范围。OpenEXR 中带有示例图像,我们可以用它们来演示 HDR 渲染。



图 9.1

一个 OpenEXR HDR 图像的不同视图。左边为最低曝光,右边为最高曝光

在一个单个图像中存储如此多细节的唯一方式就是使用浮点数据。在 OpenGL 中渲染的任何场景,特别是当它包含非常明亮或者非常阴暗的区域时,如果保留真正的颜色输出,而不是将其截取到 0.0 到 1.0 之间然后再分成仅有的 256 个可能值的话,看起来会真实得多。

### 使用 OpenEXR

因为 OpenEXR 是一种自定义的数据格式,所以我们不能使用常规的文件访问方式对这种数据进行读取和解释执行。幸运的是工业光魔已经为我们提供了完成所有繁重工作所必需的库。通过包含一些 OpenEXR 头文件并与 OpenEXR lib 文件进行连接,就可以使用这些已经创建好的工具加载数据了。OpenEXR 将所有对 EXR 文件的访问看作文件中包含数据的“窗口”或“视图”。在应用程序中,首先将我们想要打开的文件名称传递给构造函数,从而创建一个 `RGBAPrimitive` 对象。接下来,创建一个

Box2i 对象并用来来自一个 dataWindow 调用的强类型数据对其进行填充, 从而获取 OpenEXR 图像的宽度和高度。然后这些宽度和高度就可以用来创建包含 RGBA 数据的像素的 2D 数组了。

```
Array2D<Rgba> pixels;
Box2i dw = file.dataWindow();
texWidth = dw.max.x - dw.min.x + 1;
texHeight = dw.max.y - dw.min.y + 1;
pixels.resizeErase (texHeight, texWidth);
```

在文件打开并且有了存储数据的位置之后, 需要告诉 RgbaInputFile 对象我们想要通过调用 setFrameBuffer 将数据放到哪里, 然后通过调用 readPixels 来读取实际数据。

```
file.setFrameBuffer (&pixels[0][0] - dw.min.x - dw.min.y * texWidth, 1, texWidth);
file.readPixels (dw.min.y, dw.max.y);
```

现在已经有了数据, 可以将它加载到一个纹理上了。但是, 数据首先必须是一种 OpenGL 能够理解的格式。这些数据必须复制到一个浮点数组。

```
GLfloat* texels = (GLfloat*)malloc(texWidth * texHeight * 3 * sizeof(GLfloat));
GLfloat* pTex = texels;
// 将 OpenEXR 复制到本地缓冲区, 准备加载到纹理
for (unsigned int v = 0; v < texHeight; v++)
{
    for (unsigned int u = 0; u < texWidth; u++)
    {
        Imf::Rgba texel = pixels[texHeight - v - 1][u];
        pTex[0] = texel.r;
        pTex[1] = texel.g;
        pTex[2] = texel.b;
        pTex += 3;
    }
}
```

然后, 将浮点数组加载到指定的纹理对象。

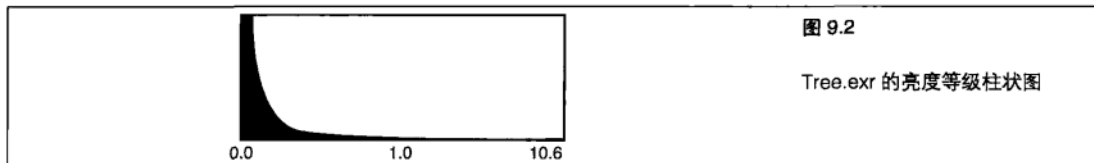
```
// 绑定纹理, 加载图像, 设置 tex 状态
glBindTexture(GL_TEXTURE_2D, textureName);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB16F, texWidth, texHeight, 0, GL_RGB, GL_FLOAT,
texels);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
free(texels);
```

好了! 现在 HDR 图像数据已经加载到 OpenGL 纹理图像上, 做好了使用准备。

## 色调映射

现在我们已经看到了使用浮点渲染的一些优点, 那么如何使用这些数据创建仍然要使用从 0 到 255 的值进行显示的动态图像呢? 色调映射 (Tone Mapping) 就是将颜色数据从一组颜色映射到另一组颜色, 或者从一个颜色空间映射到另一个颜色空间的操作。因为我们不能直接显示浮点数据, 所以这些数据必须将色调映射到一个能够被显示的颜色空间。

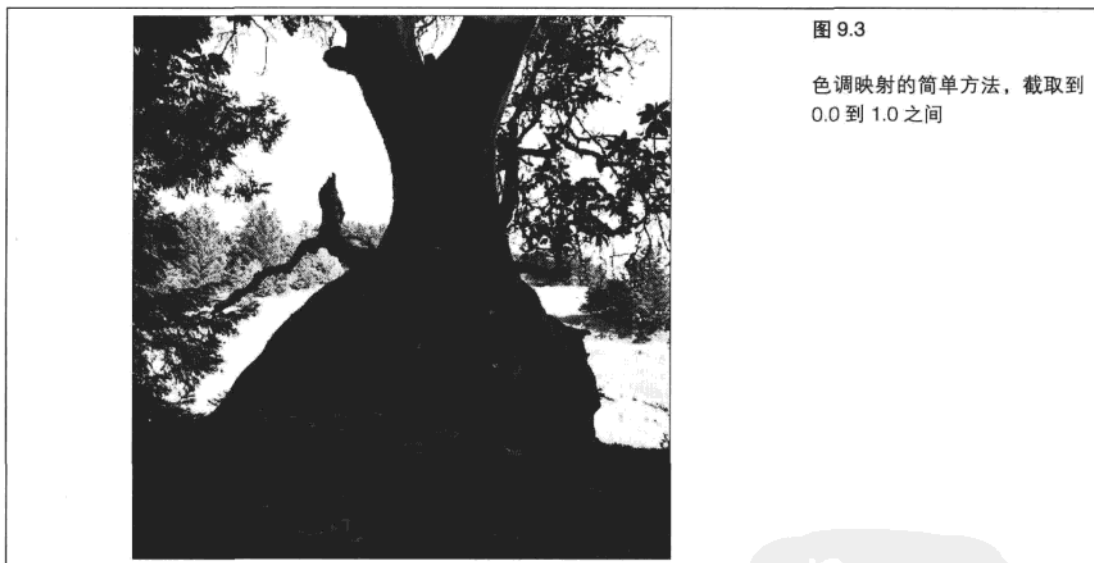
第一个示例程序 `hdr_imaging` 使用 3 种方法将高清晰度的输出映射到低清晰度的屏幕上。第一种方法通过按 1 键来启用，这是一种简单朴素的方法，直接将浮点纹理进行纹理贴图到屏幕上。图 9.2 所示的柱状图展示了 0 到 1 之间的大部分图像数据，但是很多重要的高光都远高于 1.0。实际上，这幅图片的最高亮度等级是 9.16！



结果，图像被截取，所有的高光区域看起来都是白色的。

另外，因为数据的主要部分都在这个范围的 4 分之一部分之内，或者说在直接映射到 8 位时是在 0 到 63 之间，所以它们都被混合到一起，看起来是黑色的。

图 9.3 所示显示了输出结果，其中的明亮部分几乎是白色的，而阴暗部分则接近黑色。



示例程序中的第二种方法是改变图像的“曝光”程度，与照相机改变对环境的曝光类似。按 2 键可以进入这种模式。每个曝光等级都提供了一个到纹理数据的稍有不同的窗口。低曝光能够显示场景中高亮区域的细节，而高曝光则使我们可以看到场景中阴暗区域的细节，但却冲掉了高亮部分。这种情况与图 9.1 所示类似，在图 9.1 所示左边的图像为低曝光，右边的图像则为高曝光。

对于我们的色调映射传递来说，`hdr_imaging` 示例程序从一个浮点纹理中进行读取，并写入到一个帧缓冲区对象，在这个缓冲区对象中一个 8 位纹理被绑定到了第一个渲染目标。这样就允许从 HDR 到 LDR（低动态范围）的变换在逐个像素的基础上进行，这种方式减少了当一个纹理单元在明亮部分和阴暗部分之间进行插值时出现的修饰痕迹。一旦 LDR 图像生成，它就会作为一个纹理直接绘制到屏幕上。程序清单 9.2 展示了 FBO 和纹理的设置，以及进行转换的渲染传递。

程序清单 9.2 将 HDR 的内容渲染到一个 FBO，然后再渲染到窗口

```
// 创建并绑定一个 FBO
glGenFramebuffers(1, &fboName);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fboName);

// 创建 FBO 纹理
glGenTextures(1, fboTextures);
glBindTexture(GL_TEXTURE_2D, fboTextures[0]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB8, hdrTexturesWidth[curHDRTex],
             hdrTexturesHeight[curHDRTex], 0, GL_RGBA, GL_FLOAT, NULL);
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                      GL_TEXTURE_2D, fboTextures[0], 0);

....

// 设置一个或多个 HDR 纹理
glActiveTexture(GL_TEXTURE0);
glGenTextures(1, hdrTextures);
glBindTexture(GL_TEXTURE_2D, hdrTextures[curHDRTex]);

// 从 EXR 文件中加载 HDR 图像
LoadOpenEXRImage("Tree.exr", hdrTextures[curHDRTex],
                 hdrTexturesWidth[curHDRTex], hdrTexturesHeight[curHDRTex]);

....

// 首先以完整的 FBO 分辨率绘制 FBO

// 将 FBO 绑定到 8 位绑定
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fboName);
glViewport(0, 0, hdrTexturesWidth[curHDRTex], hdrTexturesHeight[curHDRTex]);
glClear(GL_COLOR_BUFFER_BIT);

// 将纹理和 HDR 图像进行绑定
glBindTexture(GL_TEXTURE_2D, hdrTextures[curHDRTex]);

// 渲染传递，通过使用选定的程序下采样 8 位
projectionMatrix.LoadMatrix(fboOrthoMatrix);
SetupHDRProg();
fboQuad.Draw();

// 然后将结果的到的图像绘制到屏幕，保留图像比例
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
glViewport(0, 0, screenWidth, screenHeight);
glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);

// 将 8 位纹理和 HDR 图像进行绑定
glBindTexture(GL_TEXTURE_2D, fboTextures[0]);

// 绘制确定大小的屏幕，成比例的四边形，采用 8 位纹理
projectionMatrix.LoadMatrix(orthoMatrix);
SetupStraightTexProg();
screenQuad.Draw();
```

程序清单 9.2 中的代码看起来与我们以前看到过的其他采样程序类似。奇妙之处在于完成实际转换的片段着色器。程序清单 9.3 中包含了执行基于曝光进行转换的片段着色器的源代码。如果程序在可变曝光模式下，我们可以使用方向键上和下来调整曝光。这个程序的曝光范围为 0.01 到 20.0。请注意随着曝光等级的变化，图像中不同位置的细节层次是如何变化的。

程序清单 9.3 用于进行从 HDR 到 LDR 转换的 `hdr_exposure.fs` 片段着色器

```
#version 150
// hdr_exposure.fs
// 根据指定曝光将浮点纹理缩放到 0.0 - 1.0 范围内
//

in vec2 vTexCoord;

uniform sampler2D textureUnit0;
uniform float exposure;

out vec4 oColor;

void main(void)
{
    // 从 HDR 纹理中获取
    vec4 vColor = texture(textureUnit0, vTexCoord);

    // 将曝光应用到这个纹理单元
    oColor = 1.0 - exp2(-vColor * exposure);
    oColor.a = 1.0f;
}
```

第一个示例程序中使用的最后一个色调映射着色器根据场景中不同部分的相对亮度对曝光等级进行动态调整。首先，这个着色器需要知道当前进行色调映射的纹理单元附近区域的相对亮度。着色器通过当前纹理单元中心进行一个  $5 \times 5$  矩阵的采样来完成这项工作。所有环绕的样本随后进行加权，并相加在一起。相加得到的最终颜色被转换成一个亮度值。示例程序使用一个查询表将亮度转换成曝光，然后这个曝光值被用于将 HDR 纹理单元转换成 LDR 值。程序清单 9.4 展示了自适应 HDR 着色器。

程序清单 9.4 用于从 HDR 到 LDR 转换中自适应曝光水平的 `hdr_adaptive` 片段着色器

```
#version 150
// hdr_adaptive.fs
//
//

in vec2 vTex;

uniform sampler2D textureUnit0;
uniform sampler1D textureUnit1;
uniform vec2 tc_offset[25];

out vec4 oColor;

void main(void)
{
    vec4 hdrSample[25];
    for (int i = 0; i < 25; i++)
    {
        // 围绕当前纹理单元执行 25 次查询
        hdrSample[i] = texture(textureUnit0, vTex.st + tc_offset[i]);
    }
    // 计算区域加权颜色
    vec4 vColor = hdrSample[12];
    vec4 kernelcolor = {
        (1.0 * (hdrSample[0] + hdrSample[4] + hdrSample[20] + hdrSample[24])) +
        (4.0 * (hdrSample[1] + hdrSample[3] + hdrSample[5] + hdrSample[9] +
        hdrSample[15] + hdrSample[19] + hdrSample[21] + hdrSample[23])) +
        (7.0 * (hdrSample[2] + hdrSample[10] + hdrSample[14] + hdrSample[22])) +

```

```

    (16.0 * (hdrSample[6] + hdrSample[8] + hdrSample[16] + hdrSample[18])) +
    (26.0 * (hdrSample[7] + hdrSample[11] + hdrSample[13] + hdrSample[17])) +
    (41.0 * hdrSample[12])
    ) / 273.0;

// 为整个过滤器核心计算亮度
float kernelLuminance = dot(kernelcolor.rgb, vec3(0.3, 0.59, 0.11));

// 查询相应的曝光
float exposure = texture(textureUnit1, kernelLuminance/2.0).r;
exposure = clamp(exposure, 0.02f, 20.0f);

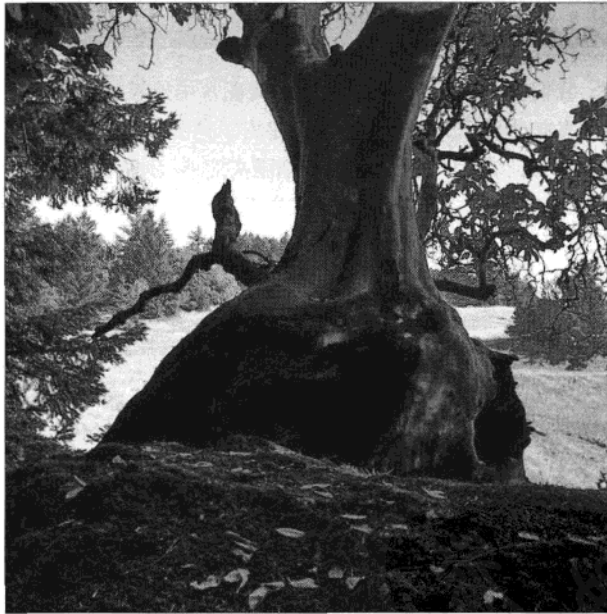
// 将曝光应用到这个纹理单元
oColor = 1.0 - exp2(-vColor * exposure);
oColor.a = 1.0f;
}

```

在为一个图像使用一种曝光时，我们可以考虑整个范围并使用平均值来进行调整以获得最佳效果。在明亮区域和阴暗区域中，使用这种方法仍然会丢失大量细节。自适应片段着色器使用的查询表为图像的明亮区域和阴暗区域带来了细节。这个查询表使用一种类似对数的缩放方式，将亮度值映射到曝光等级上。我们可以修改这个表，来增加或者减小使用曝光的范围，以及最终结果在不同的动态范围内得到的细节数量。

图 9.4

自适应色调映射能够在明亮区域和阴暗区域带来细节  
(同样显示在了彩图 15 中)



这里采用的过滤器核心和查询表方法只是众多方法中的一种。作为练习，我们可以试着修改片段着色器，对曝光进行编程计算。

我们还可以使用 HDR 纹理的更低 Mip 贴图层次来确定附近区域的亮度。

好了，现在我们已经知道如何对 OpenEXR 文件进行图像处理了，但是在典型的 OpenGL 程序中这样做有什么好处呢？好处很多！OpenEXR 图像只是任意的照明 OpenGL 场景的代替品而已。现在很多 OpenGL 游戏和应用程序都将 HDR 场景和其他内容渲染到浮点缓冲区，然后再将结果在屏幕上显示。我们可以使用刚学到的用于在 HDR 渲染的方法，生成更多逼真的照明环境，并显示每个帧的动态范围和细节。

## 向场景添加泛光

HDR 渲染所涉及的不仅仅是浮点缓冲区，实际上，这只是个开始。

有了这种额外的精度，所有类型的效果都可能达到。现在我们已经开始了解浮点缓冲区能够做什么，我们再增加一种效果，让直接光照的场景更加真实。读者可能注意到了，太阳或者明亮的灯具有时候可能会吞没树枝或者其他位于我们和光源之间的物体。这种效果就叫做光源泛光。图 9.5 所示（也可以参见彩图 16）展示了光源泛光是如何影响室内场景的。

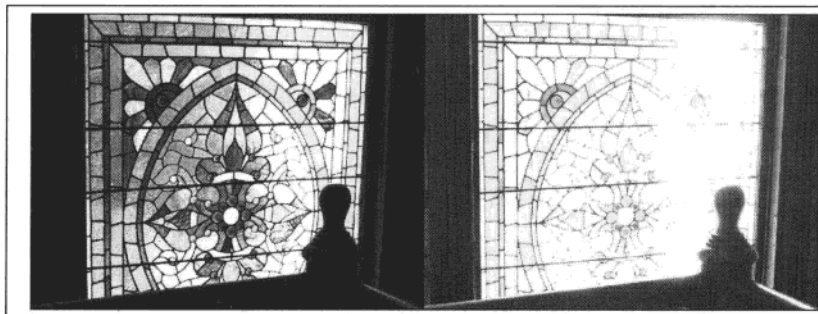


图 9.5

左边的图片展示了彩色玻璃的细节；右边图片的光源泛光中更加明亮的曝光遮盖了彩色玻璃

请注意，我们在图 9.5 所示左边的低曝光图片中能够看到所有的细节。右边图片的曝光要强得多，彩色玻璃中的线条被光源泛光覆盖了。甚至是右下角的木头柱子看起来也变小了，因为它也被泛光覆盖了。通过向场景中添加泛光，可以在特定区域增强亮度感。我们可以模拟这种由明亮的光源所产生的泛光效果。虽然我们也可以使用 8 位精度的缓冲区来实现这种效果，但在 HDR 场景中使用浮点缓冲区效果要好得多。

第一步要绘制到我们在 HDR 中的场景。对于 `hdr_bloom` 示例程序来说，我们设置了一个绑定了两个浮点纹理的 FBO。场景会像通常一样渲染到第一个绑定的纹理。

但是第二个绑定的纹理则只接受视野中的明亮区域。为了提高效率，示例程序 `hdr_bloom` 从一个着色器中通过一次传递对两个纹理进行填充，如程序清单 9.5 所示。明亮区域的数据用来生成泛光效果。泛光等级可以通过一个统一值进行调整。为了对明亮区域进行过滤，首先要对所有低于截断值的片段应用截断（cutoff），从而将它们归零。然后剩下的片段将被缩放到 0.0 到 0.1 之间，对应亮度等级 0.0 到 0.5，任何高于 0.5 的值都会被截取到 1。

程序清单 9.5 `tex_replace` 片段着色器，将亮度数据输出到一个单独的缓冲区

```
#version 150
// tex_replace.fs
// 使用纹理替换输出一个颜色值
//

varying vec2 vTexCoord;
uniform sampler2D textureUnit0;
uniform vec4 vColor;
out vec4 oColor;
out vec4 oBright;

void main(void)
{
    const float bloomLimit = 1.0;
```

```

oColor = vColor*texture(textureUnit0, vTexCoord);
oColor.a = 1.0;

vec3 brightColor = max(vColor.rgb - vec3(bloomLimit), vec3(0.0));
float bright = dot(brightColor, vec3(1.0));
bright = smoothstep(0.0, 0.5, bright);
oBright.rgb = mix(vec3(0.0), vColor.rgb, bright).rgb;
oBright.a = 1.0;
}

```

在完成以上工作之后，结果得到的亮度等级（在 0.0 到 0.5 之间）将与原来的颜色进行混合。这就是说，混合操作的结果将是一个介于 (0, 0, 0) 和原始颜色之间的颜色值，具体值根据亮度值而定。最后，亮度传递缓冲区将由只出现在屏幕中明亮区域的 0.0 之外的值进行填充，这些值以浮点格式进行存储。

现在场景已经进行了渲染，但要完成亮度传递，我们还有一些工作要做。亮度数据必须进行模糊，才能实现泛光效果。为了完成这项工作，我们创建 4 个浮点纹理，其中第一个纹理的宽度和高度是屏幕的三分之一，而每个后续的纹理都是前一个纹理的三分之一。我们通过执行一个原始图像的高斯模糊（Gaussian blur）来对第一个纹理进行渲染。第二个纹理通过对第一个纹理进行高斯模糊而进行绘制，所有 4 个纹理依此类推。示例程序使用程序清单 9.6 中列出的着色器来应用模糊，这个着色器将在输入的纹理上运用一个 5 x 5 卷积内核（convolution kernel），并将最接近的 24 个纹理单元的结果进行组合以创建一个新值。

程序清单 9.6 模糊片段着色器，在输入的纹理上应用一个 5 x 5 卷积内核

```

#version 150
// blur.fs
// 使用输入纹理的高斯模糊输出一个颜色值
//

in vec4 vFragColor;
in vec2 vTexCoords;

uniform sampler2D textureUnit0;
uniform vec2 tc_offset[25];

out vec4 oColor;

void main(void)
{
    vec4 sample[25];
    for (int i = 0; i < 25; i++)
    {
        sample[i] = texture(textureUnit0, vTexCoords.st + tc_offset[i]);
    }

    // 1 4 7 4 1
    // 4 16 26 16 4
    // 7 26 41 26 7 / 273
    // 4 16 26 16 4
    // 1 4 7 4 1

    oColor = (
        (1.0 * (sample[0] + sample[4] + sample[20] + sample[24])) +
        (4.0 * (sample[1] + sample[3] + sample[5] + sample[9] +
        sample[15] + sample[19] + sample[21] + sample[23])) +
        (7.0 * (sample[2] + sample[10] + sample[14] + sample[22])) +

```

```
        (16.0 * (sample[6] + sample[8] + sample[16] + sample[18])) +  
        (26.0 * (sample[7] + sample[11] + sample[13] + sample[17])) +  
        (41.0 * sample[12])  
    ) / 273.0;  
}
```

在模糊传递完成后，模糊结果将与场景中的完整颜色纹理进行组合，以生成最终结果。在程序清单 9.7 中，我们能从 5 个纹理中看到最终的着色器示例效果：完整彩色纹理、亮度传递和亮度传递的 4 个渐进模糊版本。亮度传递和模糊结果相加到一起，以形成泛光效果，再乘以一个用户控制的统一值。我们可以按键盘上向右的方向键放大泛光效果，也可以按向左的方向键进行缩小。最终的 HDR 颜色结果随后将进行曝光计算，这些计算我们应该已经在上一个示例程序中很熟悉了。

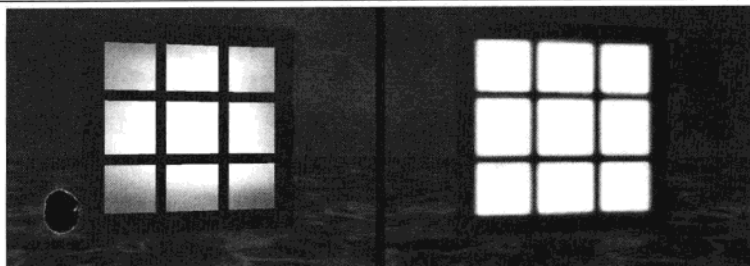
程序清单 9.7 hdr\_exposure 片段着色器，向场景中添加泛光效果

```
#version 150  
// hdr_exposure.fs  
// 从浮点模糊纹理中应用模糊效果  
// 根据指定曝光将浮点纹理场景缩放到 0.0 - 1.0 范围内  
//  
  
in vec2 vTexCoord;  
  
uniform sampler2D origImage;  
uniform sampler2D brightImage;  
uniform sampler2D blur1;  
uniform sampler2D blur2;  
uniform sampler2D blur3;  
uniform sampler2D blur4;  
  
uniform float exposure;  
uniform float bloomLevel;  
  
out vec4 oColor;  
out vec4 oBright;  
  
void main(void)  
{  
    // 从 HDR 和模糊纹理中获取  
    vec4 vBaseImage = texture(origImage, vTexCoord);  
    vec4 vBrightPass = texture(brightImage, vTexCoord);  
    vec4 vBlurColor1 = texture(blur1, vTexCoord);  
    vec4 vBlurColor2 = texture(blur2, vTexCoord);  
    vec4 vBlurColor3 = texture(blur3, vTexCoord);  
    vec4 vBlurColor4 = texture(blur4, vTexCoord);  
    vec4 vBloom = vBrightPass +  
        vBlurColor1 +  
        vBlurColor2 +  
        vBlurColor3 +  
        vBlurColor4;  
  
    vec4 vColor = vBaseImage + bloomLevel * vBloom;  
  
    // 将曝光应用到这个纹理单元  
    vColor = 1.0 - exp2(-vColor * exposure);  
    oColor = vColor;  
    oColor.a = 1.0f;  
}
```

程序清单 9.7 中展示的曝光着色器用来将一个屏幕大小的纹理四边形绘制到窗口。好了！我们可以调整泛光效果，以达到理想的状态。图 9.6 所示展示了低泛光等级和高泛光等级的 hdr\_bloom 示例程序。尝试使用一个单个 Mip 贴图纹理而不是一组 4 个混合纹理来对程序进行修改。

图 9.6

hdr\_bloom 程序。左边显示了没有泛光的效果；右边则显示了过度曝光效果（同样显示在了彩图 17 中）



### 浮点深度缓冲区

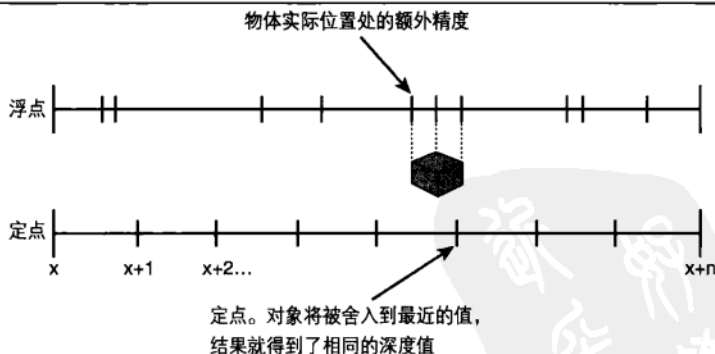
我们不仅可以对颜色数据使用浮点格式，还可以对深度数据使用浮点格式！典型情况下深度缓冲区的深度是 24 位，为我们提供了 16 777 216 种可能的深度值，这些深度值通常会缩放到 0.0 到 1.0 之间。坦白地说，这看起来好像已经很多了！但使用定点数据格式作为缓冲区存在更大的问题，就是每次增加深度值都是增加同样的大小。我们可能会有一个批次的几何图形，它们都落在同一个深度值上，这是因为我们对几何图形进行了分层，或者因为我们最终只使用了整个深度范围中一个很小的部分。这种情况可能导致 z 冲突、低分辨率或其他不确定行为。

浮点深度缓冲区可以帮助我们解决一些类似的问题。在这种情况下我们不必再受限于固定增量，写入到浮点深度缓冲区的临近几何图形拥有的回旋余地要大得多。如图 9.7 所示，我们就能了解这种效果是如何实现的。这种附加的精度可能在很多情况下都有帮助，特别是在处理阴影区时。

但是，虽然浮点深度缓冲区非常好用，最好也不要随处使用它们。和其他定点格式相比，浮点缓冲区可能会占用更多的空间，也可能导致 GPU 的读取和写入速度变慢。对于许多传统应用来说，固定精度通常就足够了。

图 9.7

定点深度缓冲区和浮点深度缓冲区的精度对比



即使在浮点深度缓冲区中，如果几何图形距离太近的话，还是可能遇到 z 冲突问题。但是既然浮点深度缓冲区并不只局限于从 0.0 到 1.0 的范围内，那么有一种方式可以帮助我们避免浮点深度缓冲区中的 z 冲突问题，这就是将几何图形进行伸展。我们可以超越典型的 0.0–1.0 范围，充分利用浮点存储方式。

OpenGL 为我们提供了可以在其中进行渲染的裁剪区（clip volume）。落在裁剪区外面的几何图形将被“裁剪掉”，而不会进行光栅化。为了渲染到显示的窗口中，裁剪区通常由这个窗口的顶部、右侧、

底部和左侧边缘构成。另外还有一个近端面 and 远端面。如果一个对象太近或者太远，那么它将不会进行绘制。远端面能够帮助我们避免对太远的几何图形进行绘制，这些几何图形甚至比一个像素还要小。但是还有一些情况（例如阴影区）下，我们需要所有深度数据，无论多远或者多近。要做到这一点，可以通过调用 `glDisable(GL_DEPTH_CLAMP)` 来关闭深度裁剪，这样就会忽略近端和远端裁剪面上的所有裁剪。深度裁剪在默认情况下是关闭的。

### 9.3.2 多重采样

有时候一次采样是不够的！在很多情况下，我们以一个与屏幕空间垂直方向或平行方向成很小角度的方向绘制直线时，会生成一个锯齿状的边缘，因为这个边缘只穿过了有限的几行或者几列像素。以我们在图 9.6 中看到的 `hdr_bloom` 示例程序中的窗口顶部为例。窗口的顶部已经被光栅化，窗口从第  $x$  行像素开始，但是它最终持续延伸到第  $x+1$  行。这种称为锯齿的效果是不美观的、不真实的，也是我们不愿意看到的。

多重采样会为每个像素位置在稍有不同的位置生成几个片段，这些片段称为亚像素（subpixel）。这些亚像素还必须先进行“解析”，然后才能显示。对一个多重采样缓冲区进行解析，就是将所有亚像素一起进行平均，以确定最终的像素颜色。以如图 9.8 所示的线为例，对于右边直线上的每个像素来说，某些样本将落到边缘上，而另外一些则不会。随后当所有亚像素一起进行平均化时，边缘两边的像素颜色会显示出这条边缘中多大一部分穿过了这些像素。这样我们得到的就是一个非常平滑的过渡，而不再是参差不齐的锯齿状边缘。采用同样的方式，不靠近边缘的像素也可以通过多重采样得到加强，最终的颜色会比单独采样达到的最好结果更加接近真实情况。

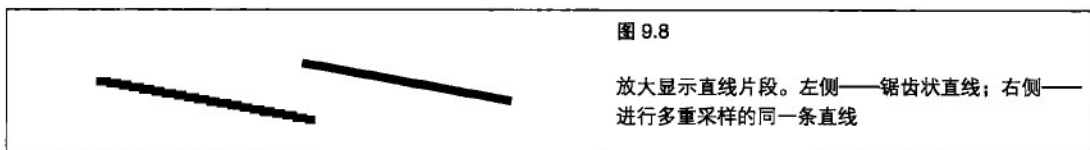


图 9.8

放大显示直线片段。左侧——锯齿状直线；右侧——进行多重采样的同一条直线

亚像素的位置并不是有规律分布的。取而代之的是，亚像素位置是在像素区域进行伪随机（pseudorandomly）分布的。这种方式增强了多重采样的抗锯齿效果。图 9.9 所示展示了 2x、4x 和 8x 抗锯齿中亚像素的可能位置。

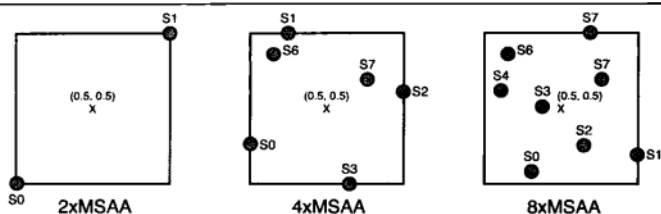
我们可以通过调用 `glGetMultisamplefv` 获取每个亚像素的实际位置。首先，调用带有 `GL_SAMPLES` 枚数值的 `glGetIntegerv` 查出当前帧缓冲区一共有多少样本。

```
// 获取每个多重采样亚像素的位置
int sampleCount = 0;
glGetIntegerv(GL_SAMPLES, &sampleCount);

float positions[64]; // 至少足以处理 32 个样本
for(int i=0; i < sampleCount; i++)
{
    glGetMultisamplefv(GL_SAMPLE_POSITION, i, &positions[i*2]);
}
```

图 9.9

2x、4x 和 8x 多重采样抗锯齿  
中样本的可能位置



OpenGL 允许我们使用几种支持多重采样的表面。第一种就是窗口表面本身。我们将在第 13 章、第 14 章和第 15 章学习如何创建多重采样窗口，这些章节对操作系统相关的窗口管理细节进行了讨论。第二种就是渲染缓冲区。我们可以使用 `glRenderBufferStorageMultisample` 为一个多重采样的 RBO 创建存储。类似地，我们还可以通过使用 `glTexImage2DMultisample` 和 `glTexImage3DMultisample` 创建多重采样纹理。

```
glGenTextures(1, hdrTextures);
glBindTexture(GL_TEXTURE_2D_MULTISAMPLE, hdrTextures[0]);
glTexImage2DMultisample(GL_TEXTURE_2D_MULTISAMPLE, 8, GL_RGB16F,
    screenWidth, screenHeight, GL_FALSE);
```

这里需要注意的重要部分是 MSAA 纹理的新纹理目标。当绑定一个 MSAA 纹理时，必须使用 `GL_TEXTURE_2D_MULTISAMPLE` 代替 `GL_TEXTURE_2D`。对于多重采样数组，可以使用 `GL_TEXTURE_2D_MULTISAMPLE_ARRAY` 目标。

现在我们已经了解如何创建多重采样表面了，还需要了解关于在着色器中访问多重采样纹理的几件事。过去的 `sampler2D` 无法完成这项工作，取而代之的是，我们要声明一个新的采样器，叫做 `sampler2DMS`。然后，在片段着色器中，我们可以使用 `texelFetch()` 为一个片段获取任何给定的样本。`texelFetch()` 采样函数要求我们指定一个整数片段位置，而不是提供 0.0 到 1.0 之间的纹理坐标。GLSL 为我们提供了一个叫做 `textureSize()` 的函数，它可以弄清着色器中多重采样纹理的大小。

这样，我们就能将标准化纹理坐标转换成整数纹理坐标了。

```
// 从 HDR 和模糊纹理中获取
iTmp = textureSize(origImage);
tmp = floor(iTmp * vTex);
vec4 vBaseImage = texelFetch(origImage, ivec2(tmp), sampleNumber);
```

对多重采样缓冲区的解析接受所有样本，并创建一个最终的输出值。

典型的解析函数可能会将每个样本的所有颜色加在一起，然后再除以样本数量。我们能够访问一个多重采样纹理的每一个样本，这使得我们可以创建自定义解析函数，而不必依赖默认函数。这在 HDR 渲染中尤其有用，因为我们可以每个亚像素上应用色调映射，然后再执行一个解析操作，而不是只将所有东西加在一起丧失多重采样的所有优势。在 `hdr_msaa` 示例程序中，我们使用一个熟悉的设置演示使用多重采样缓冲区带来的优势。

首先，创建 MSAA 纹理，并将它绑定到 MSAA 纹理目标。然后，在程序清单 9.8 中，获取每个亚像素的位置，然后使用它们计算到中心的距离。这种信息用来创建加权值，这种加权值将在片段着色器中进行自定义着色器解析时被应用到每个亚像素上。

程序清单 9.8 亚像素距离计算，纹理缓冲区对象编程

```
// 将距离值相加，以充分利用加权计算
for(int i=1; i<8; i++)
```

```

{
    float totalWeight = 0.0f;
    for(int j=0; j<=i; j++)
    {
        totalWeight += invertedSampleDistances[j];
    }

    // 转而获取用于每个样本的因子，所有样本的权重的和总为 1.0
    float perSampleFactor = 1 / totalWeight;
    for(int j=0; j<=i; j++)
    {
        sampleWeights[i][j] = invertedSampleDistances[j] * perSampleFactor;
    }
}

// 设置一个纹理缓冲区对象来保存样本加权值
glGenBuffers(1, &sampleWeightBuf);
glBindBuffer(GL_TEXTURE_BUFFER_ARB, sampleWeightBuf);
glBufferData(GL_TEXTURE_BUFFER_ARB, sizeof(float)*8, sampleWeights,
             GL_DYNAMIC_DRAW);
glBindBuffer(GL_TEXTURE_BUFFER_ARB, 0);

```

现在到了设置程序和对象状态的时候了。我们可以将场景渲染到 MSSA FBO，就像什么都没发生一样。GPU 会自动生成所有亚像素，并在每个亚像素上调用片段着色器。当 FBO 中的场景完成时，程序将运行一个解析着色器进行色调映射并对多重采样缓冲区进行解析，以在窗口中进行显示。因为场景中的一部分是进行 HDR 渲染的，所以色调映射是一个非常重要的步骤。实际上，如果我们没有在着色器中进行解析，那么 HDR 值将严重歪曲硬件解析。这样做得到的结果就是，图像看起来比我们使用只有一个样本的缓冲区时锯齿更严重。

程序清单 9.9 展示了解析着色器。这里进行了两种不同类型的解析。第一种解析是简单的平均化，即将所有样本相加到一起，然后再除以样本数量。这个值将存储在 `vColor`。另一种解析是通过每一个样本乘以一个加权值完成，这个加权值根据样本位置得到。这个值存储在 `vWeightedColor` 中。使用者可以决定显示哪种解析，通过按 `W` 键选择加权解析，或者按 `Q` 键选择直接平均解析。

程序清单 9.9 在 `hdr_exposure.fs` 中执行 MSAA 解析和色调映射操作

```

#version 150
// hdr_exposure.fs
// 根据指定曝光将浮点纹理缩放到 0.0 - 1.0 范围内
// 根据输入样本计数对多重采样缓冲区进行解析
//

in vec2 vTexCoord;

uniform sampler2DMS origImage;
uniform samplerBuffer sampleWeightSampler;
uniform int sampleCount; // 0-based, 0=1sample, 1=2samples 等
uniform int useWeightedResolve; // 0=false, 1=true
uniform float exposure;

out vec4 oColor;

// 在一个独立的函数中进行所有色调映射
vec4 toneMap(vec4 vHdrColor)
{
    vec4 vLdrColor;
    vLdrColor = 1.0 - exp2(-vHdrColor * exposure);
}

```

```

    vLdrColor.a = 1.0f;
    return vLdrColor;
}

void main(void)
{
    // 计算整数纹理坐标
    vec2 tmp = floor(textureSize(origImage) * vTexCoord);

    // 查找加权颜色和未加权颜色
    vec4 vColor = vec4(0.0, 0.0, 0.0, 1.0);
    vec4 vWeightedColor = vec4(0.0, 0.0, 0.0, 1.0);

    for (int i = 0; i <= sampleCount; i++)
    {
        // 从 texBo 获取这个样本的加权值, 根据样本的数量而变化
        float weight = texelFetchBuffer(sampleWeightSampler, i).r;
        // 在进行加权值前对 HDR 纹理单元进行色调映射
        vec4 sample = toneMap(texelFetch(origImage, ivec2(tmp), i));
        vWeightedColor += sample * weight;
        vColor += sample;
    }

    // 现在, 确定将要执行的解析类型
    oColor = vWeightedColor;

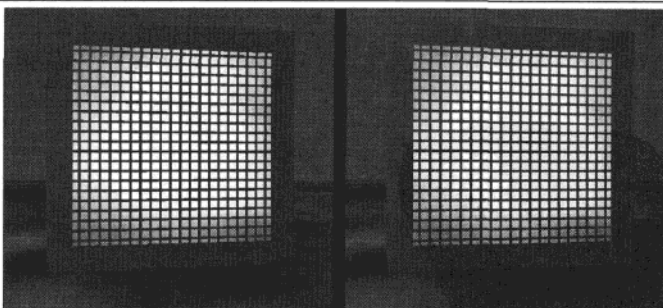
    // 如果用户选择非加权解析, 则输出平均加权值
    if (useWeightedResolve != 0)
    {
        oColor = vColor / (sampleCount+1);
    }
    oColor.a = 1.0f;
}

```

用户还可以对用于执行解析的样本数量进行配置。我们可以使用从 1 到 8 的数字键来选择解析中使用的样本数量。图 9.10 所示 (也可以参见彩图 18) 展示了单个采样和 8 次采样之间的区别。使用多重缓冲区能够对场景质量产生重大的影响。

图 9.10

多重采样的结果。左边的图像有 1 个样本, 而右边的图像有 8 个样本



### 9.3.3 整数

GLSL 开始的时候使用许多特有的数据类型, 这些数据类型并不严格遵守 IEEE 对浮点数的标准和 CPU 中常用数据类型的其他工业标准。但是最近这些格式进行了很大程度的标准化。我们不仅可以在着色

器中使用浮点数据并希望他们的表现和在 CPU 中相同，还可以使用其他数据类型，例如整数和无符号整数。

同时还出现了新的纹理格式，以满足着色器的迫切需求。我们可以使用大量不同的整数格式创建纹理，这些格式既有有符号的，也有无符号的。它们的使用范围从一个通道（只有 R）到所有 4 个通道（RGBA），可以包含的精度从每通道 8 位到每通道 32 位。几乎所有整数纹理所需要的格式都存在。除了纹理之外，RBO 也可以通过整数格式创建。我们可以对一个绑定到 PBO 的基于整数的缓冲区进行清除，对于整数缓冲区可以调用 `glClearBufferiv`，而对于无符号整数缓冲区则可以调用 `glClearBufferuiv`。

为什么说整数和无符号整数格式很重要呢？首先，我们现在可以通过纹理和统一值将整数发送到着色器，这些整数可以是大于 255 的。着色器还允许我们访问一个按位进行的方法中整数纹理中的数据，允许我们以任何希望的方式进行封装。在对大量已经渲染的几何图形（例如由大量树木组成的森林）进行修改时，整数缓冲区和纹理可以用于检索或选择。为了访问整数纹理的独立纹理单元，可以在 GLSL 着色器中使用 `texelFetch` 命令。

这个示例函数接受一个整数向量来指定一个纹理中的一个位置，所以我们可以确定获得的是哪个纹理单元，还可以将整数数据绑定到纹理缓冲区对象。

增加整数格式最重要的原因就是，这些整数格式在着色器中创造了真正灵活和并行的计算环境。我们可以加载浮点格式或整数格式的任意纹理数据，并且可以执行需要的任何计算，例如物理计算、图像处理、建模以及各种需要高度并行处理的计算相关工作。这种通用计算（General Purpose Computing）常常称为 GPGPU，它确实为我们打开了一扇大门，使 GPU 能够在创纪录的短时间内完成大量工作。

### 9.3.4 sRGB

RGB 颜色空间足以满足需求的时代已经一去不复返了。现在我们可以使用超级 RGB 了！实际上它的名称是 sRGB，但是它比我们在过去的 20 年中已经习惯的传统 RGB 颜色空间要强大得多。即使 RGB 在计算机图形中是最通用的颜色空间，还是有各种因素会制约它到底还能走多远。

人类眼睛的灵敏度比 RGB 颜色空间中所能显示的灵敏度要高得多。

RGB 颜色空间使用从 0 到 1 的值，但是对最终结果应用了一种线性伽玛渐变，扩展了它能够显示的颜色范围。sRGB 光谱中更深的颜色使用一个接近 2.2 的伽玛值，但是在更亮的区域使用一个达到 2.4 的扩展伽玛值。这就意味着 sRGB 格式有一个内建的扩展动态范围。sRGB 最初是为了在图像和照片处理中用来帮助在办公室和暗房这样的典型观察环境中更好地映射和显示颜色而创建的。

当我们在 OpenGL 中使用一个 sRGB 纹理时，在纹理被采样时 sRGB 格式将被转换成 RGB 格式。但是只有 RGB 分量会被转换，而 alpha 分量则保持不变。

每个分量都会根据下列规则而独立进行转换。

如果纹理单元值小于或等于 0.04045，那么实现将使用如下方式进行转换。

$$\text{Sample} = \text{Texel} / 12.92$$

如果纹理单元值大于 0.04045，样本将进行如下转换。

$$\text{Sample} = ((\text{Texel} + 0.055) / 1.055)^{2.4}$$

渲染缓冲区还支持 sRGB 存储格式，尤其是必须支持 GL\_SRGB8\_ALPHA8 格式。这意味着我们可以将带有内部 sRGB 格式的 RBO 和纹理绑定到一个帧缓冲区对象上，然后再对它们进行渲染。因为我们刚刚讨论了 sRGB 格式是怎样非线性的，所以读者可能也不希望对 sRGB FBO 的写入是线性的，因为这将使整个目标化为泡影。好消息是，OpenGL 能够在调用 glEnable(GL\_FRAMEBUFFER\_SRGB)时自动将着色器输出的线性颜色值转换成 sRGB 值。请记住，这种方法只适用于包含一个 sRGB 表面的绑定。我们可以调用带有 GL\_FRAMEBUFFER\_ATTACHMENT\_COLOR\_ENCODING 值的 glGetFramebufferAttachmentParameteriv 来查询绑定的表面是否是 sRGB。

如果是 sRGB 表面则返回 GL\_SRGB，而其他表面则返回 GL\_LINEAR。这种对于片段颜色 (fc) 的变换遵循表 9.3 中列出的方程式。

表 9.3 颜色输出转换为 sRGB 遵循的变换方程式

片段值	变换方程
fc <= 0.0	0.0
0.0 < fc < 0.0031308	12.92 * fc
0.0031308 < fc < 1.0	1.055 * fc 0.41666 - 0.055
fc > 1.0	1.0

9.3.5 纹理压缩

新的格式使 OpenGL 更加有用、更加灵活，而纹理压缩则甚至在现代 GPU 可以应用吉比特级内存的条件下也非常有帮助。纹理可能会占用数量惊人的空间！某些现代游戏在特定的级别下能够轻易地使用 1 吉比特的纹理数据。这实在是很大的数据量！我们要把它放在哪里呢？对于创建丰富的、真实的和令人印象深刻的场景来说，纹理单元是一个重要的组成部分，但是如果无法将所有数据加载到 GPU 的话，那么渲染将会非常慢（如果不是不可能完成的话）。解决大量纹理数据的存储和使用的一种方法是将这些数据进行压缩。

OpenGL 实现至少会支持表 9.4 中列出的压缩方案，其中一个方案是 RGTC（红-绿纹理压缩）。RGTC 格式将一个纹理图像分解成 4 x 4 纹理单元块，使用一系列的代码将独立的通道压缩到这个块中。这种压缩模式只适用于一个或两个通道的有符号和无符号纹理。我们不需要担心具体的压缩方案，除非打算编写一个压缩程序。我们只要注意使用 RGTC 所节省的空间是 50%就可以了。

表 9.4 本地 OpenGL 纹理压缩格式

格式	类型
GL_COMPRESSED_RED	Generic
GL_COMPRESSED_RG	Generic
GL_COMPRESSED_RGB	Generic

续表

格 式	类 型
GL_COMPRESSED_RGBA	Generic
GL_COMPRESSED_SRGB	Generic
GL_COMPRESSED_SRGB_ALPHA	Generic
GL_COMPRESSED_RED_RGTC1	RGTC
GL_COMPRESSED_SIGNED_RED_RGTC1	RGTC
GL_COMPRESSED_RG_RGTC1	RGTC
GL_COMPRESSED_SIGNED_RG_RGTC1	RGTC

表 9.4 中列出的前 6 种格式是通用的，允许 OpenGL 驱动程序决定使用哪种压缩机制。这就意味着驱动程序能够使用最适合当前情况的格式。美中不足的是，这种方式是与实现相关的，并且不能移植。

实现可能还会支持其他压缩格式，例如 ETC1 和 S3TC。

我们应该首先查询 OpenGL 没有要求的格式的适用性，然后再尝试使用它们。最好的方式就是查询对相关扩展名的支持，后面在第 13 章、第 14 章和第 15 章，我们将学习更多相关内容。

### 使用压缩

我们可以在加载一个纹理的时候要求 OpenGL 对它进行压缩。我们所要做的只是将内部格式请求为一种压缩格式。OpenGL 将接受未经压缩的数据，并在加载纹理图像时对数据进行转换。在压缩纹理和未经压缩纹理的使用方式上没有什么区别。GPU 会在从纹理上进行采样时处理这种转换。很多用于创建纹理和其他图像的图像工具允许我们将数据直接保存为压缩格式。

一旦使用某种非通用压缩内部格式加载了一个纹理，我们就可以通过调用 `glGetCompressedTexImage` 获取经过压缩的图像。只要挑选我们感兴趣的纹理目标和 Mip 贴图层次就好。因为我们可能并不清楚图像是如何压缩的或者使用的是什么格式，所以应该检查图像大小来确保我们有足够的空间可以容纳整个表面。

```
Glint imageSize = 0;
glGetTexParameteri(GL_TEXTURE_2D, TEXTURE_COMPRESSED_IMAGE_SIZE, &imageSize);
void *data = malloc(imageSize);
glGetCompressedTexImage(GL_TEXTURE_2D, 0, data);
```

要加载压缩纹理图像，我们可以使用专用纹理加载函数：`glCompressedTexImage1D`、`glCompressedTexImage2D` 和 `glCompressedTexImage3D`。

使用这些函数的方式与使用 `glTexImage1D`、`glTexImage2D` 等的方式相同。

我们还可以通过 `glCompressedTexSubImage1D`、`glCompressedTexSubImage2D` 或 `glCompressedTexSubImage3D` 来更新压缩纹理图像。

### 共享指数

虽然在真正意义上的场景中，共享指数在技术上并不是一种压缩格式，但它们确实允许我们使用浮点

纹理数据时节省存储空间。

共享指数对整个纹理单元使用同样的指数值，而不是为 R、G 和 B 值各保存一个指数。每个值的小数和指数部分都会以整数形式进行存储，然后在纹理采样时将它们组合到一起。

对于 GL\_RGB9\_E5 格式，有 9 位用于存储每种颜色，5 位用于存储所有通道的通用指数。这种格式将 3 个浮点值封装到 32 位中，节省了 67% 的空间！为了利用共享指数，我们可以直接以这种格式从内容创建工具中获取数据，或者编写一个能够将浮点 RGB 值压缩到共享指数格式的转换器。

## 9.4 小结

OpenGL 的功能远远超出了我们最早在第 8 章中看到的基本帧缓冲区访问的范围。在本章，我们学习了片段着色器的逻辑名是如何使用的，以及这些名称如何映射到输出索引。然后体验了浮点缓冲区的强大，以及它们是如何加强场景的真实性。

浮点深度缓冲区在提供额外精度上发挥了作用，而多重采样缓冲区则使得片段着色器能够直接进行样本级访问。

我们着实花费了一些时间来了解多重采样抗锯齿，以及如何使用它提高我们的输出质量。我们学习了如何在片段着色器中直接对多重采样缓冲区的每个样本进行访问。我们还了解了整数纹理是如何用于提供着色器功能更简单的外部索引，以及如何在着色器单元中开启通用计算。最后，我们探索了以 sRGB 格式存储纹理和缓冲区数据、压缩和共享指数的新方法，它们提供了更加真实的颜色映射，并有助于节省存储空间。



## 第 10 章 片段操作：管线的终点

作者：Nicholas Haemel

## 本章内容

任 务	使用的函数
对多重采样渲染进行微调	glSampleMask, glSampleCoverage
创建并使用模板样式	glStencilFuncSeparate, glStencilOpSeparate, glClearStencil
将颜色和 alpha 输出绑定到一起	glBlendFuncSeparate, glBlendOpSeparate, glBlendColor
使用逻辑操作	glLogicOp
对最终输出进行遮罩操作	glColorMask, glColorMaski, glDepthMask, glStencilMask, glStencilMaskSeparate

在学过前 9 章的内容之后，我们应该可以非常熟练地使用顶点和片段着色器来根据几何图形生成输出了。但是当片段着色器完成之后会发生什么呢？所有这些片段都去了哪里呢？实际情况是，这些片段在最终到达缓冲区或者窗口中的目的地之前，还要经历几个步骤。

本章将学习 OpenGL 管线中的最后几个步骤，即逐片段操作。

我们从整个过程的第一步——裁剪测试开始，对一个虚拟片段进行多重采样操作、模板测试、深度缓冲区测试、混合、抖动和逻辑操作等操作。图 10.1 所示显示了一个片段在所有阶段都启用时所遵循的路径。

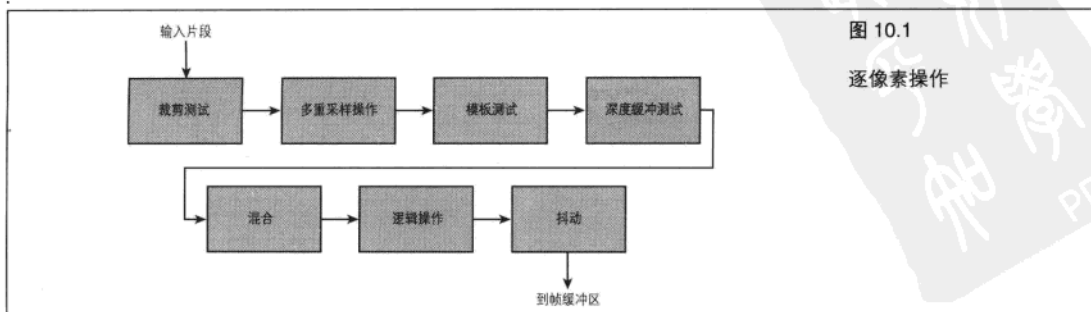


图 10.1  
逐像素操作

## 10.1 裁剪——将几何图形剪切到希望的大小

将这些片段发送到最终目的地的第一步是,决定是否将它们放在一个从可渲染区域中裁剪出来的区域中。裁剪操作是在窗口坐标中执行的,这就意味着所有输入片段都有一个位于(0, 0) 和 (width, height) 之间的窗口坐标,其中 width 和 height 是窗口的维度。

应用程序可以定义一个剪切面,它会对几何图形的某些部分进行裁剪。这种操作是通过指定 x 值的一对最大值和最小值,以及 y 值的一对最大值和最小值来实现的。我们可以通过调用 glScissor 来设置一个裁剪区域。

```
void glScissor(GLint left, GLint bottom, GLsizei width, GLsizei height);
```

必须通过调用 glEnable(GL\_SCISSOR\_TEST)启用裁剪测试,才能启用裁剪。如果片段的窗口坐标落在由裁剪定义的区域中,那么片段将继续留在管道中。否则就会被丢弃。

另一种表达这种操作的方法是通过两个使用传递到 glScissor 的值的方程式。如果  $\text{left} \leq \text{xw} < (\text{left} + \text{width})$  并且  $\text{bottom} \leq \text{yw} < (\text{bottom} + \text{height})$ , 那么测试通过。

我们在第 3 章已经学习了裁剪测试。如果对裁剪操作的记忆已经有些模糊了的话,可以回过头再复习一下。

## 10.2 多重采样

执行了裁剪操作之后,在管线中要进行的下一步工作就是多重采样。我们在第 9 章已经对多重采样有了初步的了解。现在让我们进一步了解如何对多重采样的具体细节进行控制。

请记住,多重采样阶段可以为任何给定像素生成多重子样本,这在一个像素恰好落在线或多边形边缘时可能会尤其有用。一个缓冲区的样本数是在进行分配时确定的。对于窗口表面来说,我们必须在选择一个像素格式或配置时指定采样数。对于帧缓冲区来说,我们可以在创建绑定到帧缓冲区的纹理和渲染缓冲区时选择样本数。请注意,所有帧缓冲区绑定的样本数都必须是相同的。

在第 9 章,我们还学习了如何通过调用 glGetMultisamplefv 获取像素中每个亚像素样本的位置。现在让我们了解一下如何对这些亚像素进行控制。

这里有两个可以控制的步骤,能够影响多重采样的处理方式:修改覆盖值和对样本进行掩码操作。

### 10.2.1 样本覆盖

这里的“覆盖”是指一个亚像素“覆盖”了多大区域。我们可以将一个片段的 alpha 值直接转换成一个覆盖值,来确定帧缓冲区的多少个样本将被这个片段更新。我们通过调用 glEnable(GL\_SAMPLE\_ALPHA\_TO\_COVERAGE)来完成这项工作。

一个片段的覆盖值用于确定将写入多少子样本。举例来说，一个 alpha 值为 0.4 的片段应该生成一个 40% 的覆盖值。对于一个 8 样本的 MSAA 缓冲区来说，这些像素中将有 3 个将被写入。

因为这个 alpha 值已经用于确定应该写入多少个子样本，那么将这些子样本与同一个 alpha 值进行混合就没有意义了。毕竟，使用“alpha 覆盖”是一种进行混合的方法。为了帮助防止这些亚像素在启用混合的时候也被进行混合，我们可以通过调用 `glEnable(GL_SAMPLE_ALPHA_TO_ONE)` 将这些样本的 alpha 值强制设定为 1。

使用“alpha 覆盖”的方法相对于简单的混合来说有几个优点。在对多重采样缓冲区进行渲染时，alpha 混合通常会同等地应用在整个像素上。使用“alpha 覆盖”方法，alpha 遮罩的边缘将是抗锯齿的，将产生更加自然和平滑的结果。这在绘制灌木丛、树木或者刷子上部分 alpha 透明的浓密刷毛时尤其有用。

OpenGL 还允许我们通过调用 `glSampleCoverage` 手动设置样本覆盖。我们要为一个在应用“alpha 覆盖”遮罩后出现的像素手动应用一个覆盖值。为了使这一步生效，必须通过调用 `glEnable(GL_SAMPLE_COVERAGE)` 启用样本覆盖。

```
glSampleCoverage(clampf value, Boolean invert)
```

传递到 value 参数的覆盖值可以在 0 到 1 之间。如果结果得到的遮罩应该进行反转，那么 invert 参数将被标记到 OpenGL。例如，如果绘制两颗重叠的树，其中一棵覆盖 60%，而另一棵覆盖 40%，那么我们将希望对其中一个覆盖值进行反转以确保两次绘制调用不会使用同一个遮罩。

```
glSampleCoverage(0.5, GL_FALSE);  
// 绘制第一组几何图形  
.....  
glSampleCoverage(0.5, GL_TRUE);  
// 绘制第二组几何图形  
.....
```

## 10.2.2 样本遮罩

多重采样阶段的最后一种可配置选项是样本遮罩。这一步允许我们使用 `glSampleMaski` 函数将特定样本屏蔽掉。和前面的阶段不同，我们可以确切地指定将要关闭的样本。请记住，“alpha 覆盖”方法和样本覆盖会影响哪些样本将在到达这一阶段之前被启用。这就意味着这一阶段的样本遮罩并不保证样本将会被启用。

```
glSampleMaski(GLuint maskNumber, GLbitfield mask);
```

从本质上说，mask 参数是一个 32 位的像素样本遮罩，其中 0 位映射到样本 0，1 位映射到样本 1，依此类推。我们可以使用 maskNumber 来对超出前 32 位的位进行寻址，其中每个增加的遮罩值代表另外增加的 32 位。

我们可以查询 `GL_MAX_SAMPLE_MASK_WORDS` 来检查到底能够支持多少个遮罩。在撰写本文时，实现只支持一个字，考虑到没有哪个实现支持每个像素多于 32 个样本，这种设置是有道理的。

还用另一种方法可以修改样本遮罩。我们可以写入到一个片段着色器中内建的 `gl_SampleMask[]` 输出数组，来设置着色器中的遮罩。

### 10.2.3 综合运用

这一章的示例程序 oit 绘制了一些半透明对象，形状就像一个彩色的玻璃风铃。当在 OpenGL 中绘制几个半透明表面时，简单地将它们混合到一起会产生错误的结果。让我们想象一下，如果绘制一个 alpha 值为 0.5 的对象，然后试图在它后面绘制另一个对象，其 alpha 值也为 0.5。如果我们将深度测试保持为启用状态，那么作为深度测试失败的结果，后面的对象将被简单地丢弃；如果深度测试被关闭，那么后面的对象将简单地绘制在前面对象的前方，就像后面的对象是在前面一样。在本章后面内容中，我们将深入地讨论更多关于混合的细节。

为了克服这种混合缺陷，我们需要使用顺序无关透明度 (Order Independent Transparency)，简称 OIT。大多数对透明几何图形进行正确渲染的算法都要按照深度对将要进行渲染的对象进行排序，首先对距离最远的对象进行渲染。这样做是非常复杂和耗费时间的。更糟糕的是，在很多情况下这样做并不能得到正确的排序。

为了解决这个问题，我们使用样本遮罩将每次渲染传递存储到多重采样缓冲区的独立样本中。在场景进行渲染之后，这种解析操作会为每个像素将所有样本以正确的顺序组合在一起。现在就开始做吧。

第一步将所有几何图形绘制到一个多重采样帧缓冲区。程序清单 10.1 中的代码将对部分几何图形进行绘制。所有不透明的对象都被进行遮罩到样本 0，每个半透明物体都使用样本遮罩来渲染到唯一的样本。

程序清单 10.1 设置样本遮罩状态

```
glSampleMaski(0, 0x01);
glEnable(GL_SAMPLE_MASK);

....

glBindTexture(GL_TEXTURE_2D, textures[1]);
shaderManager.UseStockShader(GLT_SHADER_TEXTURE_REPLACE,
    transformPipeline.GetModelViewProjectionMatrix(), 0);
bckgrndCylBatch.Draw();

....

modelViewMatrix.Translate(0.0f, 0.8f, 0.0f);
modelViewMatrix.PushMatrix();
modelViewMatrix.Translate(-0.3f, 0.f, 0.0f);
modelViewMatrix.Scale(0.40, 0.8, 0.40);
modelViewMatrix.Rotate(50.0, 0.0, 10.0, 0.0);
glSampleMaski(0, 0x02);
shaderManager.UseStockShader(GLT_SHADER_FLAT,
    transformPipeline.GetModelViewProjectionMatrix(), vLtYellow);
glasslBatch.Draw();
modelViewMatrix.PopMatrix();

modelViewMatrix.PushMatrix();
modelViewMatrix.Translate(0.4f, 0.0f, 0.0f);
modelViewMatrix.Scale(0.5, 0.8, 1.0);
modelViewMatrix.Rotate(-20.0, 0.0, 1.0, 0.0);
glSampleMaski(0, 0x04);
shaderManager.UseStockShader(GLT_SHADER_FLAT,
    transformPipeline.GetModelViewProjectionMatrix(), vLtGreen);
```

```
glass2Batch.Draw();
modelViewMatrix.PopMatrix();
```

.....

当所有的表面都各自绘制到唯一的样本位置时，它们必须进行组合。但是，使用常规的多重采样解析是做不到的！因此，我们使用程序清单 10.2 中的自定义解析着色器来代替。每个样本的颜色值和深度值首先被提取到一个数组中，然后进行分析以确定片段的颜色。

程序清单 10.2 按深度解析多个层次

```
#version 150
```

```
// oitResolve.fs
//
```

```
in vec2 vTexCoord;
```

```
uniform sampler2DMS origImage;
uniform sampler2DMS origDepth;
```

```
out vec4 oColor;
```

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

```
    const int sampleCount = 8;
```

```
    vec4 vColor[sampleCount];
    float vDepth[sampleCount];
    int vSurfOrder[sampleCount];
    int i = 0;
```

```
    // 计算非标准化的纹理坐标
```

```
    vec2 tmp = floor(textureSize(origDepth) * vTexCoord);
```

```
    // 首先获取样本数据并对表面顺序进行初始化
```

```
    for (i = 0; i < sampleCount; i++)
```

```
    {
```

```
        vSurfOrder[i] = i;
```

```
        vColor[i] = texelFetch(origImage, ivec2(tmp), i);
```

```
        vDepth[i] = texelFetch(origDepth, ivec2(tmp), i).r;
```

```
    }
```

```
    // 对深度值进行排序，最大值在前面，最小值在后面
```

// 必须对数组进行 (size^2-size) 次遍历，或者如果在任何一次遍历显示所有样本都以正确的顺序排列的情况下可以提前退出

```
    for (int j = 0; j < sampleCount; j++)
```

```
    {
```

```
        bool bFinished = true;
```

```
        for (i = 0; i < (sampleCount-1); i++)
```

```
        {
```

```
            float temp1 = vDepth[vSurfOrder[i]];
```

```
            float temp2 = vDepth[vSurfOrder[i+1]];
```

```
            if (temp2 < temp1)
```

```
            {
```

```
                // 交换值
```

```
                int tempIndex = vSurfOrder[i];
```

```
                vSurfOrder[i] = vSurfOrder[i+1];
```

```
                vSurfOrder[i+1] = tempIndex;
```

```
                bFinished = false;
```

```

    }
}

if (bFinished)
    j = 8; // 完成了。可以提前退出!
}

// 现在, 按照从前到后的顺序将所有颜色相加应用 alpha 值
bool bFoundFirstColor = false;
vec4 summedColor = vec4(0.0, 0.0, 0.0, 0.0);

for (i = (sampleCount-1); i >= 0; i--)
{
    int surfIndex = vSurfOrder[i];
    if(vColor[surfIndex].a > 0.001)
    {
        if (bFoundFirstColor == false)
        {
            // 在第一种颜色上应用 100%
            summedColor = vColor[surfIndex];
            bFoundFirstColor = true;
        }
        else
        {
            // 应用 alpha 值与使用 glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA); 时相同的颜色
            summedColor.rgb =
                (summedColor.rgb * (1 - vColor[surfIndex].a)) +
                (vColor[surfIndex].rgb * vColor[surfIndex].a);
        }
    }
}

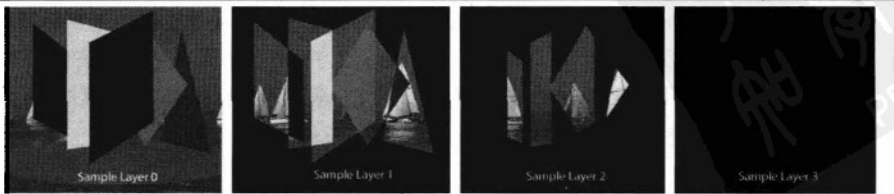
oColor = summedColor;
oColor.a = 1.0f;
}

```

为了使透明度正确地工作, 每一块几何图形的颜色都必须从后向前应用。要做到这一点, 我们需要弄清楚哪个几何图形覆盖在另外的哪个几何图形上。这就意味着这些深度值必须进行解析和存储。我们将排序的结果存储在 `vSurfOrder` 数组中, 准备在下一步使用。这个数组保存了指向样本数组的指针。索引 0 指向最近的样本, 而索引 1 则指向除此之外下一个最近的样本, 以此类推。对于只有一个或两个层次被绘制的位置, 所有其他样本的颜色值和 `alpha` 值都为 0。图 10.2 所示为排序的结果。在最左边是 `vSurfOrder[0]` 指向的最近的样本, 第二个是 `vSurfOrder[1]` 指向的除上述样本之外最近的样本, 以此类推。请注意, 样本 0 包含了主要背景, 因为没有什么与背景重叠。这样, 背景就是最近的, `vSurfOrder` 中只有一个样本与之相关。对于这个应用程序来说, 在任意给定的区域中最多只有 4 个重叠的几何图形。

图 10.2

按照深度进行排序的样本。最近的样本在左侧, 最远的在右侧



现在我们已经知道每个样本的顺序, 并由此知道了每个几何图形的顺序, 我们可以正确地为每个几何图形应用透明度了。整个处理过程都可以通过在包含透明度数据的多重采样缓冲区中运行一个解析着色器

来完成。要执行解析，着色器要在颜色数组 `vColor` 中根据 `vSurfOrder` 指定的顺序查询每个样本。然后，每种颜色都会应用到每个像素的总颜色中。新的顶层颜色将和它的 `alpha` 值相乘，然后再将已经存在的总颜色与“1 减去输入 `alpha` 值”得到的结果相乘，再将两个乘积相加。

这种操作完全是在硬件中进行的，它为每个像素创建了透明颜色。最终的输出结果如图 10.3 所示（也可以参见彩图 19）。这些半透明的玻璃状物体进行渲染的顺序并不重要，最终结果会进行正确的混合。我们可以按键盘上向左的方向键和向右的方向键对 `oit` 程序中的场景进行旋转。请注意最接近我们的物体总是显示在顶部，即使在它们的渲染顺序并不改变的情况下也是如此。

我们也可以通过绑定到一个 FBO 的独立颜色缓冲区来使用相似的方法，但是很多实现都受绑定缓冲区数量的限制。使用多重采样缓冲区来完成顺序无关的透明度提供了简单的缓冲区访问，并且不会对正常的渲染过程产生明显的干扰。但是，这种方法还有一些限制因素。首先，只有数量有限的透明或交叉几何图形能够被渲染。在一个多重采样缓冲区只有这么多的样本。这就意味着包含成百上千可能重叠对象的复杂透明几何图形已经超出了这种方法的处理能力。另外，我们无法在为 `oit` 使用一个多重采样缓冲区的同时再进行多重采样渲染。但对于简单的 `oit` 情况来说，这种方法很好用，并不需要对几何图形进行预存储。

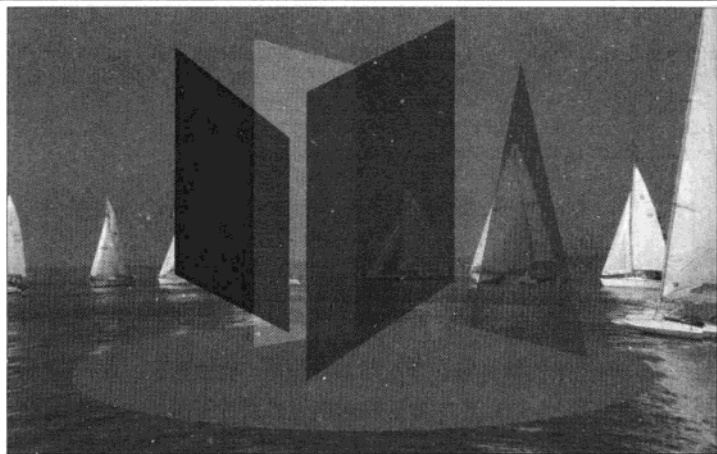


图 10.3

顺序无关深度的最终结果

## 10.3 模板操作

片段管线中的下一个步骤是模板测试。我们可以将模板测试想象成在一块硬纸板上裁剪下一个图案，然后再用这个镂空部分在墙上喷涂出这个图案。喷涂只能接触到墙壁上硬纸板被裁减掉的位置。如果我们有一个支持模板缓冲区的像素格式，就可以用类似的方式将我们绘制的图形在帧缓冲区中进行遮罩操作了。我们可以通过调用 `glEnable(GL_STENCIL_TEST)` 来启用模板操作。大多数模板缓冲区包含 8 位，但是一些配置可能会支持更少的位数。

我们的绘制命令可以对模板缓冲区产生直接的影响，而模板缓冲区的值可以对我们绘制的像素产生直接的影响。为了控制与模板缓冲区的相互影响，OpenGL 提供了两个命令：`glStencilFuncSeparate` 和 `glStencilOpSeparate`。OpenGL 允许我们分别对前向几何图形和背向几何图形分别设置它们。

```
void glStencilFuncSeparate(GLenum face, GLenum func, GLint ref, GLuint mask);
void glStencilOpSeparate(GLenum face, GLenum sfail, GLenum dpfail, GLenum dppass);
```

首先看一下 `glStencilFuncSeparate`，它控制着模板测试通过或失败的条件。我们可以为 `face` 传递 `GL_FRONT`、`GL_BACK` 或 `GL_FRONT_AND_BACK`，表示哪个几何图形将受到影响。`func` 的值可以是表 10.1 中的任何值。这些值描述在什么情况下几何图形将通过模板测试。`ref` 值是一个参考值，它用来计算通过/失败的结果，而 `mask` 则允许我们控制参考值的哪些位和缓冲区进行比较。

表 10.1 模板函数

函 数	通过条件
<code>GL_NEVER</code>	总是不通过测试
<code>GL_ALWAYS</code>	总是通过测试
<code>GL_LESS</code>	参考值小于缓冲区值
<code>GL_LEQUAL</code>	参考值小于或等于缓冲区值
<code>GL_EQUAL</code>	参考值等于缓冲区值
<code>GL_GEQUAL</code>	参考值大于或等于缓冲区值
<code>GL_GREATER</code>	参考值大于缓冲区值
<code>GL_NOTEQUAL</code>	参考值不等于缓冲区值

下一个步骤是通过调用 `glStencilOpSeparate` 告诉 OpenGL 在模板测试通过或者失败时要做什么。这个函数接受 4 个参数，其中第一个参数指定哪个面将受到影响，后面的 3 个参数控制在模板测试执行之后会发生什么，它们可以是表 10.2 中的任意值。第二个参数 `sfail` 是在模板测试失败的情况下采取的动作。`dpfail` 参数指定在深度缓冲区测试失败的情况下采取的动作，而最后一个参数 `dppass` 则指定在深度缓冲区测试通过的情况下采取的动作。

表 10.2 模板操作

操 作	结 果
<code>GL_KEEP</code>	不要修改模板缓冲区
<code>GL_ZERO</code>	将模板缓冲区的值设为 0
<code>GL_REPLACE</code>	用参考值替换模板值
<code>GL_INCR</code>	增加模板的饱和度值
<code>GL_DECR</code>	降低模板的饱和度值
<code>GL_INVERT</code>	对模板值进行按位取反
<code>GL_INCR_WRAP</code>	增加模板的非饱和度值
<code>GL_DECR_WRAP</code>	降低模板的非饱和度值

那么这到底是如何实现的呢？下面让我们来看看一个典型应用的简单示例，参见程序清单 10.3。第一步是通过 `glClearStencil` 设置模板清除值，然后调用带有模板缓冲区位的清除操作，从而将模板缓冲区清除为 0。接下来将绘制一个窗口边框，它包含诸如玩家分数和统计信息之类的细节。

调用 `glStencilFuncSeparate` 将模板测试设置为当参考值为 1 时总是通过，然后告诉 OpenGL 只在调用 `glStencilOpSeparate` 进行深度测试时通过，然后对边界几何图形进行渲染时，才替换模板缓冲区中的

值。这样就将边框区域的像素设置为 1，而帧缓冲区中其他像素仍然为 0。

接下来对模板状态进行设置，使模板测试只在模板缓冲区值为 0 时才能通过，然后对场景的其他部分进行渲染。这样将导致所有将要覆盖刚刚绘制边框的像素不能通过模板测试，也就不能绘制到帧缓冲区了。程序清单 10.3 展示了一个如何使用模板的示例。

程序清单 10.3 模板缓冲区使用示例，模板边框装饰

```
// 将模板缓冲区清除为 0
glClearStencil(0);
glClear(GL_STENCIL_BUFFER_BIT);

// 设置模板状态来进行边框渲染
glStencilFuncSeparate(GL_FRONT, GL_ALWAYS, 1, 0xff);
glStencilOpSeparate(GL_FRONT, GL_KEEP, GL_ZERO, GL_REPLACE);

// 渲染边框装饰

... ..

// 现在边框装饰像素有了模板值 1
// 所有其他像素的模板值都为 0

// 为常规渲染设置模板状态，
// 如果像素要覆盖边框则不能通过
glStencilFuncSeparate(GL_FRONT_AND_BACK, GL_LESS, 1, 0xff);
glStencilOpSeparate(GL_FRONT, GL_KEEP, GL_KEEP, GL_KEEP);

// 对场景的其他部分进行渲染，不会对覆盖模板边框的内容进行渲染

... ..
```

还有另外两个模板函数：`glStencilFunc` 和 `glStencilOp`。它们与 `face` 参数设置为 `GL_FRONT_AND_BACK` 的 `glStencilFuncSeparate` 和 `glStencilOpSeparate` 的行为方式相同。

## 10.4 深度测试

在模板操作完成之后，如果深度测试启用，硬件将对一个片段的深度值进行测试。如果深度写入被启用，而片段通过了深度测试，那么深度缓冲区将会对片段的新深度值进行更新。如果深度测试失败，那么片段将被销毁，而不会传递到片段操作的其他阶段。我们在本书的整个内容中都在使用深度缓冲区和深度测试对它们的操作应该很熟悉！可以回顾一下第 3 章来复习相关内容。

### 10.4.1 深度截取

还有一种与深度测试相关的功能称为深度截取（depth clamping）。

在默认情况下，深度截取是关闭的，但是可以通过调用 `glEnable(GL_DEPTH_CLAMP)` 将其开启。如果开启了深度截取，那么输入像素的深度将在深度测试执行之前被截取到近端和远端剪切面。

在防止几何图形被裁剪到裁剪区域时，深度截取可能会非常有用。阴影区域渲染就是一个应用实例。在对阴影区域进行渲染时，我们希望尽可能多地沿着  $z$  轴方向保留几何图形。要做到这一点，可以启用深度截取，用它来避免比远端剪切面还要远和比近端剪切面还要近的数据被裁剪掉。

## 10.5 进行混合

如果一个片段通过了深度测试，那么它就会被传递到混合阶段。混合操作允许我们将输入的源颜色与已经存在于颜色缓冲区中的颜色，或者其他使用众多支持的混合方程之一的常量进行组合。混合只能在定点和浮点格式中进行，不能将诸如 `GL_RGB_16I` 或者 `GL_RGB32I` 这样的整数格式进行混合。同样，如果绘制到的缓冲区是定点格式的，那么输入源颜色将在进行任何混合操作之前被截取到 0.0–1.0 范围内。混合是以每个绘制缓冲区为基础进行控制的，通过调用 `glEnablei(GL_BLEND, bufferIndex)` 启用。就像使用 `glDrawBuffers` 一样，缓冲区索引可以是 `GL_DRAW_BUFFER0`、`GL_DRAW_BUFFER1` 等。如果默认 FBO 被绑定，那么混合将在所有启用的缓冲区执行。

### 10.5.1 混合方程式

混合可以进行高度的自定义。要考虑的第一个方面是我们希望如何将像素值（源）与帧缓冲区颜色（目标）相结合。如果使用 `glBlendEquationSeparate`，那么可以为 RGB 值和 alpha 值选择单独的操作；如果使用 `glBlendEquation`，那么可以为 RGB 值和 alpha 值使用相同的方程式。

表 10.3 列出了可用的混合方程式。混合操作就按照源和目标颜色都为浮点数的情况执行。

```
glBlendEquation(GLenum mode);
glBlendEquationSeparate(GLenum modeRGB, GLenum modeAlpha);
```

表 10.3 混合方程式

混合方程式	RGB	Alpha
<code>GL_FUNC_ADD</code>	$R = R_s * S_r + R_d * D_r$ $G = G_s * S_g + G_d * D_g$ $B = B_s * S_b + B_d * D_b$	$A = A_s * S_a + A_d * D_a$
<code>GL_FUNC_SUBTRACT</code>	$R = R_s * S_r - R_d * D_r$ $G = G_s * S_g - G_d * D_g$ $B = B_s * S_b - B_d * D_b$	$A = A_s * S_a - A_d * D_a$
<code>GL_FUNC_REVERSE_SUBTRACT</code>	$R = R_d * D_r - R_s * S_r$ $G = G_d * D_g - G_s * S_g$ $B = B_d * D_b - B_s * S_b$	$A = A_d * D_a - A_s * S_a$
<code>GL_MIN</code>	$R = \min(R_s, R_d)$ $G = \min(G_s, G_d)$ $B = \min(B_s, B_d)$	$A = \min(A_s, A_d)$

续表

混合方程式	RGB	Alpha
GL_MAX	$R = \max(R_s, R_d)$ $G = \max(G_s, G_d)$ $B = \max(B_s, B_d)$	$A = \max(A_s, A_d)$

## 10.5.2 混合函数

现在我们已经选择了一个方程式来对源和目标颜色进行结合，必须对混合方程式中要使用的因数进行设置。这项工作可以通过调用包含我们想要使用因子的 `glBlendFunc` 和 `glBlendFuncSeparate` 来完成。就像 `glBlendEquation` 一样，我们既可以为 RGB 和 alpha 分别设置函数，也可以使用一个命令将它们设置为同一个值。

```
glBlendFuncSeparate(GLenum srcRGB, GLenum dstRGB, GLenum srcAlpha, GLenum
dstAlpha);
glBlendFunc(GLenum src, GLenum dst);
```

表 10.4 列出了这些调用的可能取值。请注意那些需要进行加法或减法的函数会以向量的形式执行这些操作。其中某些还需要一个可以通过调用 `glBlendColor` 进行设置的常量。

```
glBlendColor(clampf red, clampf green, clampf blue, clampf alpha);
```

表 10.4 混合函数

混合函数	RGB	Alpha
GL_ZERO	(0, 0, 0)	0
GL_ONE	(1, 1, 1)	1
GL_SRC_COLOR	( $R_{so}, G_{so}, B_{so}$ )	$A_{so}$
GL_ONE_MINUS_SRC_COLOR	$(1, 1, 1) - (R_{so}, G_{so}, B_{so})$	$1 - A_{so}$
GL_DST_COLOR	( $R_d, G_d, B_d$ )	$A_d$
GL_ONE_MINUS_DST_COLOR	$(1, 1, 1) - (R_d, G_d, B_d)$	$1 - A_d$
GL_SRC_ALPHA	( $A_{so}, A_{so}, A_{so}$ )	$A_{so}$
GL_ONE_MINUS_SRC_ALPHA	$(1, 1, 1) - (A_{so}, A_{so}, A_{so})$	$1 - A_{so}$
GL_DST_ALPHA	( $A_d, A_d, A_d$ )	$A_d$
GL_ONE_MINUS_DST_ALPHA	$(1, 1, 1) - (A_d, A_d, A_d)$	$1 - A_d$
GL_CONSTANT_COLOR	( $R_c, G_c, B_c$ )	$A_c$
GL_ONE_MINUS_CONSTANT_COLOR	$(1, 1, 1) - (R_c, G_c, B_c)$	$1 - A_c$
GL_CONSTANT_ALPHA	( $A_c, A_c, A_c$ )	$A_c$
GL_ONE_MINUS_CONSTANT_ALPHA	$(1, 1, 1) - (A_c, A_c, A_c)$	$1 - A_c$
GL_ALPHA_SATURATE	$(f, f, f)$ $f = \min(A_{so}, 1 - A_d)$	1

续表

混合函数	RGB	Alpha
GL_SRC1_COLOR	$(R_{s1}, G_{s1}, B_{s1})$	$A_{s1}$
GL_ONE_MINUS_SRC1_COLOR	$(1, 1, 1) - (R_{s1}, G_{s1}, B_{s1})$	$1 - A_{s1}$
GL_SRC1_ALPHA	$(A_{s1}, A_{s1}, A_{s1})$	$A_{s1}$
GL_ONE_MINUS_SRC1_ALPHA	$(1, 1, 1) - (A_{s1}, A_{s1}, A_{s1})$	$1 - A_{s1}$

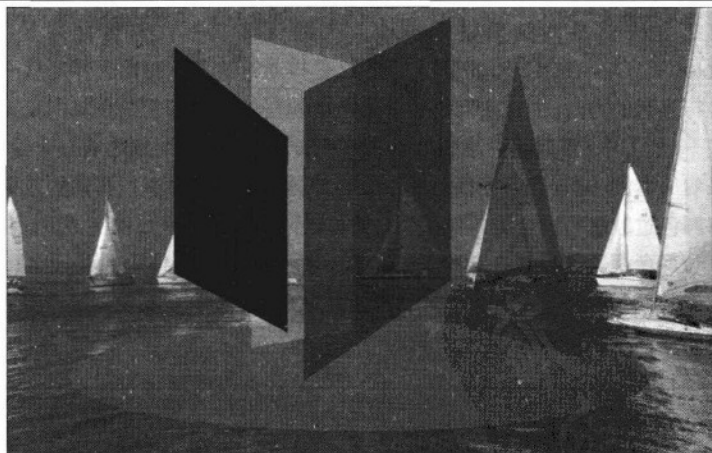
读者可能已经注意到了, 表 10.4 中的某些因子使用源颜色 0, 而其他一些则使用源颜色 1。着色器能够通过使用 `glBindFragDataLocationIndexed` 进行输出设置, 为给定的颜色绑定输出一个以上的最终颜色。使用两个输出的方法是使用正确的混合因子将这些颜色混合到一起。通过查询 `GL_MAX_DUAL_SOURCE_DRAW_BUFFERS` 的值, 可以看到所支持的双输出缓冲区数量。

### 10.5.3 综合运用

我们前面看到的 `oit` 示例程序也可以用来进行简单的混合。按键盘上的 B 键就可以切换到混合模式。每个玻璃状的几何图形都是分别进行绘制的, 并与背景进行混合。我们可以通过按键盘上从 1 到 7 的数字键在几种预设混合模式中进行选择, 来创建几种不同的效果。检查图 10.4 来看一看使用最常见的混合函数组之一, `GL_SRC_ALPHA` 或 `GL_SRC_ONE_MINUS_ALPHA` 所得到的结果。

图 10.4

混合透明玻璃



如果选择了混合模式, 那么程序会启用混合模式, 然后设置正确的混合参数 (参见程序清单 10.4)。OpenGL 会完成剩余的工作, 将增加的每个几何图形与帧缓冲区在其进行绘制时混合。最后一步是对多重采样 FBO 进行解析, 在屏幕上显示最终结果。

程序清单 10.4 模板缓冲区使用示例, 模板边框装饰

```
// 设置混合状态
glEnable(GL_BLEND);
switch (blendMode)
{
case 1:
```

```
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);  
    break;  
case 2:  
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_DST_ALPHA);  
    break;  
.....
```

请注意，在进行渲染顺序相关问题的混合时，与前面看到的顺序无关的透明度示例有所不同。如果我们想对存储程序做出一些改进，那么可以尝试修改采样模式。我们可以设置不同的混合函数和不同的混合方程式，来看一看它们的效果。

## 10.6 抖动

好了！像素几乎已经到了管线的终点。在进行混合之后，像素数据仍然以一组浮点数的形式表示。但是，除非帧缓冲区是一个浮点缓冲区，否则像素数据都必须先进行转换才能存储。例如，大多数窗口可渲染格式只支持每通道 8 位。这就意味着 GPU 必须对最终颜色输出进行转换才能存储它。

根据抖动开启或者关闭，这种转换可以以两种方式进行。第一种方式是，这些结果可以简单地映射到可以表示的最大正颜色。例如，如果一个特定像素的 R 值为 0.3222，而窗口格式为 GL\_RGB\_8，那么 GPU 可以将它映射到 256 中的 82，或者 256 中的 83。如果抖动关闭，那么 GPU 将自动选择 83。我们可以调用 `glDisable(GL_DITHER)` 来设置这种行为。

第二种方式是对结果进行抖动。在默认情况下，深度截取是关闭的，但是可以通过调用 `glEnable(GL_DEPTH_CLAMP)` 开启。什么是抖动？这是一种用硬件对从一种可表示的颜色到下一步骤的过渡进行混合的方法。通过在两种相邻颜色中的任何一种都无法在其中真正表现出来的区域中将两种颜色混合在一起，GPU 可以将过渡的边缘进行柔和化，而不是突然地从一个颜色层次改变到另一个颜色层次。我们来看一看图 10.5 所示图像。图中上半部分显示了没有进行抖动的效果，而下半部分则演示了抖动是如何将颜色过渡进行混合的。有几个公式可以计算出抖动是如何完成的。但是，基本上如果底层颜色对于一个 8 位颜色缓冲区来说在 82 和 83 之间，那么每种颜色使用的百分比都与它们和 82 与 83 的接近程度成正比。由每个供应商决定各自抖动算法的方式毫无意义。在使用某个颜色缓冲区格式时，某些实现可能会选择简单地直接进入下一个渲染。

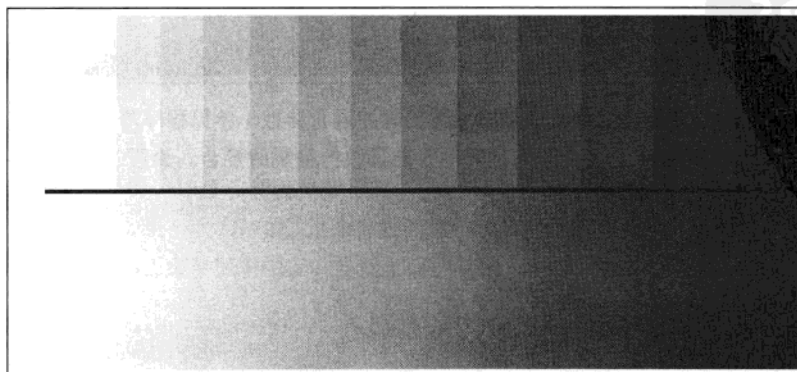


图 10.5

抖动将颜色过渡进行混合。  
上半部分是未经混合的，下  
半部分是经过噪声混合的

抖动可能会带来很大的方便。在对象逐渐地进行平滑着色时，它可以排除条纹问题。最棒的是，我们甚至不需要为它费心。抖动在默认情况下是开启的，它能够使渲染更加漂亮自然。

## 10.7 逻辑操作

如果像素颜色与帧缓冲区的格式和位深度相同，那么还有两个步骤可以对最终结果产生影响。第一个步骤允许我们在进行传递之前对像素颜色应用一个逻辑操作。在启用它时，混合的效果将会被忽略。

逻辑操作不会影响浮点缓冲区。我们可以通过调用：

```
glEnable(GL_COLOR_LOGIC_OP)
```

来启用逻辑操作。

逻辑操作使用输入像素的值和已经存在的帧缓冲区的值来计算一个最终值。我们可以选择用于计算最终值的操作，表 10.5 列出了可能的选项。我们可以将选择的逻辑操作传递到 glLogicOp。

```
glLogicOp(GLenum op);
```

表 10.5

逻辑操作

操 作	结 果
GL_ZERO	将所有值设为 0
GL_AND	源&目标
GL_AND_REVERSE	源&~目标
GL_COPY	源
GL_AND_INVERTED	~(源&目标)
GL_NOOP	目标
GL_XOR	源^目标
GL_OR	源   目标
GL_NOR	~(源   目标)
GL_EQUIV	~(源 ^ 目标)
GL_INVERT	~目标
GL_OR_REVERSE	源   ~目标
GL_COPY_INVERTED	~源
GL_OR_INVERTED	~源   目标
GL_NAND	~(源&目标)
GL_ZERO	将所有值设为 1

逻辑操作将分别应用到每个颜色通道。将源和目标进行组合的操作是在颜色值上按位执行的。逻辑操

作在今天的图形应用程序中并不是普遍应用的，但是它们仍然部分地保留在了 OpenGL 中，这是因为在通常的 GPU 中仍然支持这些功能。

## 10.8 遮罩输出

在一个片段着色器进行写操作之前可以最后对其进行的调整之一就是遮罩。到目前为止，我们已经了解，一个片段着色器可以写入 3 种不同类型的数据：颜色、深度和模板数据。类似地，我们可以对每种数据的结果分别进行操作。

### 10.8.1 颜色

我们可以使用 `glColorMask` 和 `glColorMaski` 对颜色写入进行遮罩，或者防止颜色写入发生。

我们不需要同时对所有颜色通道进行遮罩。举例来说，我们可以选择对红色和绿色通道进行遮罩，同时允许写入蓝色通道。我们可以为一个通道传递 `GL_TRUE`，以允许对这个通道进行写入，或者传递 `GL_FALSE` 将这些写入操作屏蔽掉。第一个函数 `glColorMask` 允许我们对所有当前启用渲染的缓冲区进行遮罩，而第二个函数 `glColorMaski` 则允许我们为特定的颜色缓冲区设置遮罩。

```
glColorMask(writeR, writeG, writeB);  
glColorMaski(colorBufIndex, writeR, writeG, writeB);
```

### 10.8.2 深度

对深度缓冲区的写入可以以类似的方式进行遮罩。`glDepthMask` 也接受一个布尔值，如果这个值为 `GL_TRUE` 的话则开启写入，如果为 `GL_FALSE` 的话则关闭写入。

```
glDepthMask(GL_FALSE);
```

### 10.8.3 模板

模板缓冲区也可以进行遮罩。大家可能已经猜出来了，我们用来对模板缓冲区进行遮罩的函数叫做 `glStencilMask`。但是和其他函数不同，我们对屏蔽掉的对象有更精细的控制。模板遮罩函数接受一个位段，而不只接受一个布尔值。这个位段的最小部分映射到模板缓冲区的同一位数上。如果一个遮罩位被设为 1，那么模板缓冲区中相应的位就可以被更新。但如果遮罩位为 0，那么相应的模板位就不会被写入。

```
GLuint mask = 0x0007;  
glStencilMask(mask);  
glStencilMaskSeparate(GL_BACK, mask);
```

在前面的示例中，对 `glStencilMask` 的第一次调用将模板缓冲区的最低 3 位设置为可以写入，同时其他位保持为不能写入。第二次调用 `glStencilMaskSeparate`，允许我们为前向和背向的图元分别设置遮罩。

### 10.8.4 用途

写入遮罩对于很多操作来说都会非常有用。例如，如果我们想要用深度信息填充一个阴影区域，就可以将所有颜色写入屏蔽掉，因为这时只有深度信息才是重要的。或者在我们想要直接向屏幕空间绘制一个贴花的情况下，可以关闭深度写入，以防止深度数据被污染。遮罩的关键点是，我们可以对它们进行设置，并立即调用正常渲染路径，这种路径可能会设置必要的缓冲区状态并输出所有我们通常会用到的颜色、深度和模板数据，而不需要了解任何遮罩状态。我们不必改变着色器使其不要写入某些值，也不必将某组缓冲区解除绑定，或者改变已经启用的绘制缓冲区。我们可以完全忽略掉渲染途径的其余部分，并且仍然能生成正确的结果。

## 10.9 小结

在本章，我们学习了 OpenGL 管线的终点。

第一步是进行裁剪。接下来，我们学习了如何对多重采样进行精细的控制，来调整样本覆盖或应用样本遮罩。然后，模板操作对允许哪个片段继续在管线中处理进行控制。在这之后，深度缓冲区上场了，进行测试来检查一个片段是否落在了已经进行渲染的几何图形的后方。接下来，上面处理结果得到的片段将根据用户控制的函数和方程式与深度缓冲区进行混合。然后，前面得到的结果再进行抖动，生成平滑的颜色过渡。最后，对上述结果进行遮罩，以使深度、模板和颜色操作不能应用在可应用的地方。

在 oit 示例程序中，我们学习了如何直接与一个多重采样缓冲区的独立样本进行互动。我们用样本遮罩接口对半透明对象进行混合，而无需考虑它们的绘制顺序。oit 示例程序还演示了混合是如何起作用的，并演示了几种常用混合模式。



## 第 11 章 高级着色器应用

作者: Graham Sellers

## 本章内容

任 务	使用的特性
使用一个变换反馈着色器	<code>glBindBuffer</code>
将一个变换过的顶点存储到缓冲区	<code>glBeginTransformFeedback</code> , <code>glEndTransformFeedback</code>
使用一个几何着色器	<code>glCreateShader(GL_GEOMETRY_SHADER)</code>
创建、绑定和使用统一块	<code>glGetUniformBlockIndex</code> , <code>glUniformBlockBinding</code>
使用索引绑定点	<code>glBindBufferBase</code> , <code>glBindBufferRange</code>
控制插值和存储	布局限定符

在本章,我们将了解更多的高级着色器主题,这些内容将允许我们使用可编程图形硬件,以实现简单多边形渲染之外的更多功能。我们提供了一个详细的示例,使用 GPU 通过变换反馈进行数据再循环,从而实现物理模拟。我们介绍了一个全新的着色器阶段——几何着色器,它可以对整个图元进行处理,甚至可以凭空生成新的图元。

我们还讨论了如何使用片段着色器来执行高级逐像素操作,包括图像处理和生成贴花。

本章引入了布局限定符,它允许我们控制存储、插值和其他影响着色器输入输出的参数。我们还介绍了在片段着色器中丢弃工作的方法。

在本章最后,我们将能够编写复杂的着色器,复杂到可能会使我们因为使用着色器统一值而感到厌烦!我们介绍了统一缓冲区对象,它允许我们在不同的程序对象之间共享大量的统一值。

## 11.1 高级顶点着色器

到目前为止,我们已经在将顶点从对象空间转换到场景或视觉空间的过程中使用过顶点着色器

了。我们可能会将它看作一种只进行简单几何图形变换的单输入、单输出着色器阶段。但是，顶点着色器其实非常强大——在大多数现代硬件中它已经能够访问片段着色器所能访问的所有资源了。

它可以在那些本质上并非必需几何图形的工作中使用。和变换反馈（将在第 12 章进行详细讨论）相结合，顶点着色器能够在一个循环中将结果进行循环传递，并在每一次循环过程中进行迭代和更新。这些数据不必是位置，顶点着色器的结果也不必进行直接渲染。这一部分包含几个示例，它们不太明显地应用了顶点着色器。

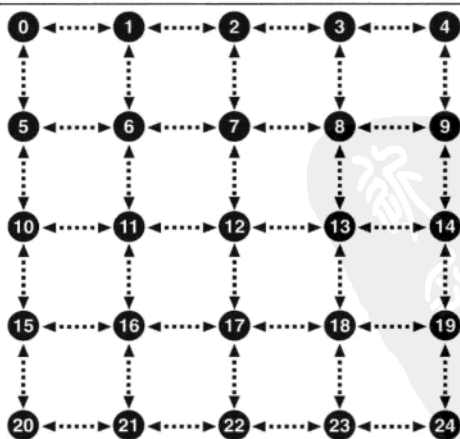
### 11.1.1 在顶点着色器中进行物理模拟

在本例中，我们建立了一个弹簧和物块组成的网状物的物理模拟。这里的每个顶点代表一个重物，与其他 4 个相邻的重物通过弹性绳进行连接。这个示例对顶点进行反复迭代，用一个顶点着色器对每一个顶点进行处理。这个示例中应用了大量高级特性。我们使用一个纹理缓冲区对象（TBO）来保存顶点位置数据，以及一个常规属性数组。同一个缓冲区将被绑定到这个 TBO 上，以及与输入到顶点着色器的位置相关联的 VBO（顶点缓冲区对象）上。这样就允许我们对系统中其他顶点的当前位置随意进行访问了。我们还使用一个整数顶点属性来保存相邻顶点的索引。此外，我们还使用变换反馈来在每次数学迭代之间存储每个物块的位置和速度。

对于每个顶点来说，我们需要一个位置、一个速度和一个质量值。我们可以将位置和质量包装到一个顶点数组中，而将速度包装到另一个数组中。在位置数组中，每个元素实际上都是一个 `vec4`，其中  $x$ 、 $y$  和  $z$  是顶点的三维坐标，而  $w$  则包含顶点的重量。速度数组则可以是一个简单的 `vec3` 数组。另外，我们使用一个 `ivec4` 数组来存储关于将重物连接到一起的弹簧的信息。每一个顶点都对应一个 `ivec4`，其中 4 个分量中的每一个都包含到与弹簧另一端相连的顶点的索引。我们称它们为连接向量（connection vector）。这就意味着我们可以将一个物块与其他 4 个物块相连。为了表示没有连接的情况，我们将连接元素指向同一个顶点本身（如图 11.1 所示）。

图 11.1

弹簧物块系统中的顶点连接



以顶点 12 为例，它有一个 `ivec4` 向量与它相关联，这个向量包含 `<7, 13, 17, 11>`——即与它相连接顶点的索引。类似地，顶点 13 的连接向量则包含 `<8, 14, 18, 12>`。在顶点 12 和顶点 13 之间有一个双向

连接。位于网状物边缘的顶点上的弹簧并不是都连接到了其他顶点。因此，顶点 14 的连接向量包含  $\langle 9, 14, 19, 13 \rangle$ 。请注意，这里的  $y$  分量返回来指向了顶点 14 本身，表示这里没有弹簧。

除了连接到的顶点的索引（用一个自我引用表示没有连接）之外，我们还定义了一些特殊值来表示其他意义。索引 -1 用来表示顶点固定在原地不动。无论受到什么力的作用，它的位置都不会被更新。这样就允许我们对系统中某些顶点的位置进行固定了。如果连接向量的  $x$  分量为 -1，那么更新顶点位置和速度的计算就会被跳过。

在每个顶点上，我们的顶点着色器使用常规顶点属性来运行和获取它本身的位置和连接向量。然后它将通过使用连接向量（同样也是常规顶点属性）的元素对 TBO 进行检索，从而对与它连接的顶点的当前位置进行查询。对于每个连接的顶点来说，它可以计算出这些顶点到它的距离，从而计算出它们之间虚拟弹簧的伸长量。有了这个伸长量，就可以计算出弹簧施加在它上面的力，从而计算出对这个顶点物块产生的加速度，继而得到下一次迭代中使用的新位置的速度。这听起来好像很复杂，其实并非如此——这只是一些牛顿力学和胡克定律的内容罢了。

胡克定律的内容是：

$$F = -kx$$

$F$  为弹簧产生的力， $k$  是弹簧的弹性系数（代表弹簧有多硬），而  $x$  则为弹簧的伸长量。弹簧的伸长量是相对于它的自然长度（不受外力拉伸或压缩情况下的长度）而言的。

对于我们的系统来说，我们将所有弹簧的自然长度设为相同值，并将它存储在一个统一值中。

弹簧的任何伸展都会产生一个正的  $x$  值，而弹簧的任何压缩则都会产生一个负的  $x$  值。弹簧的瞬时长度只不过是它从一端到另一端的向量的长度——这正是我们要在顶点着色器中进行计算的。我们通过将表示力大小的标量  $F$  乘以弹簧的方向来为这个力添加一个方向。在这里我们引入变量  $d$ ，它就是弹簧的标准化方向。

$$\vec{F} = dF$$

这样，我们就得到了由于弹簧的伸长或压缩而作用在物块上的力。如果我们要简单地在物块上应用这个力，那么这个系统将会产生摆动，并且由于数值上存在误差，最终将会变得不稳定。所有真实的弹簧系统都会由于摩擦力的作用而产生能量损失，这种效果可以通过在力学方程中加入阻尼来进行建模。由于阻尼而产生的力就由这个方程式确定

$$\vec{F}_d = -c\vec{v}$$

其中  $c$  代表阻尼系数。在理想状态下，我们应该为每个弹簧计算阻尼产生的力，但是对于这个简单的系统来说，有一个基于这个物块速度的力就够了。我们还在每个时间步长时使用初始速度来近似地代替这个方程中要用到的连续微分。在着色器中，我们通过计算阻尼力，然后将每个连接到这个物块的弹簧所施加给这个物块的力进行累加，从而对  $F$  进行初始化。

最后，我们可以将重力简单地视为在每个物块上增加一个作用力，从而将重力应用到这个系统。重力是一种不变的力，通常是沿着向下的方向作用的。我们只要将它加到作用在物块上的初始作用力上就可以了。

$$F_{total} = G - d\vec{k}x - c\vec{v}$$

一旦我们得到了合力，我们就可以简单地应用牛顿力学定律了。首先，牛顿第二定律让我们能够计算出物块的加速度。

$$\vec{F} = m\vec{a}$$

$$\vec{a} = \frac{\vec{F}}{m}$$

在这里， $F$ 代表我们刚刚使用重力、阻尼系数和胡克定律计算出来的合力； $m$ 是顶点的质量（保存在位置属性的  $w$  分量中）；而  $a$  则是最终得到的加速度结果。给定初始速度（我们从其他属性数组中获得），我们可以将它代入下面的运动方程式中，来计算我们的最终速度，以及在一定时间内会移动的距离。

$$\vec{v} = \vec{u} + \vec{a}t$$

$$\vec{s} = \vec{u} + \frac{\vec{a}t^2}{2}$$

其中  $u$  是初始速度（从速度属性数组中读取）， $v$  是最终速度， $t$  是时间步长（由应用程序提供），而  $s$  则为移动的距离。

不要忘记， $a$ 、 $u$ 、 $v$  和  $s$  都是向量。现在我们要做的只剩下编写着色器并将它连接到一个应用程序上了。程序清单 11.1 展示了这个顶点着色器。

程序清单 11.1 弹簧物块系统的顶点着色器

```
#version 330
precision highp float;

// 这个输入向量在 xyz 中包含了顶点位置，而在 w 中则包含了顶点的质量
in vec4 position_mass;
// 这是顶点的当前速度
in vec3 velocity;
// 这是我们的连接向量
in ivec4 connection;

// 这是一个 TBO，它将与 position_mass 输入属性被绑定到同一个缓冲区
uniform samplerBuffer tex_position_mass;

// 顶点着色器的输出与输入相同，只是包装在了一个接口模块中
out Vertex
{
    vec4 position_mass;
    vec3 velocity;
} vertex;

// 一个用来保存时间步长的 Uniform 值。应用程序可以对它进行更新
uniform float t;

// 全局弹性系数
uniform float k;

// 全局阻尼常数
uniform float c;
// 重力
const vec3 gravity = vec3(0.0, -0.03, 0.0);
```

```

// 弹簧的自然长度
uniform float rest_length;

// 模型视图投影矩阵
uniform mat4 mvp;

void main(void)
{
    vec3 p = position_mass.xyz; // p 可以是位置
    float m = position_mass.w; // m 是顶点的质量
    vec3 u = velocity; // u 是初始速度
    vec3 F; // F 是物块上受到的力
    vec3 v = u; // v 是最终速度
    vec3 s = vec3(0.0); // s 是在这一步长内的位移

    // 检查这个顶点是否是“固定”顶点
    if (connection[0] != -1) {
        // 使用重力和阻尼来对 F 进行初始化
        F = gravity - c * u;

        for (int i = 0; i < 4; i++) {
            if (connection[i] != gl_VertexID) {
                // q 是另一个顶点的位置
                // 不考虑其他顶点的质量
                vec3 q = texture(tex_position_mass, connection[i]).xyz;
                vec3 d = q - p;
                float x = length(d);
                F += -k * (1.0 - x) * normalize(d);
            }
        }

        // 由力所产生的加速度
        float a = F / m;
        // 位移
        s = u * t + 0.5 * a * t * t;
        // 最终速度
        v = u + a * t;
    }

    // 对输出进行写操作
    vertex.position_mass = vec4(p + s, m);
    vertex.velocity = v;

    // 更新 gl_Position 以便能够对点进行渲染
    gl_Position = mvp * vec4(p + s, 1.0);
}

```

没那么难，是吧？ 我们还需要构建缓冲区来保存位置、速度和连接信息。我们需要对位置和速度信息进行双重缓冲，以便能够一次性从一组缓冲区内进行读取，并写入到另一组缓冲区中，然后进行缓冲区交换，以便数据能够在一个缓冲区和另一个缓冲区之间来回移动。连接信息在每次传递过程中都保持不变，所以它将是常量。要完成这些工作，我们需要使用两对 VBO 和一对 VAO（顶点数组对象）。有一组位置和速度属性绑定到第一个 VAO，随着通用连接信息指向第一对 VBO。另一组位置和速度属性绑定到另一个 VAO，同样随着通用连接信息指向第二对 VBO。我们总共需要 5 个 VBO——两个缓冲区用来保存位置，两个缓冲区用来保存速度，一个缓冲区则包含连接向量。

除了 VBO 之外，我们还需要两个 TBO。我们将每个缓冲区都同时作为一个 VBO 和一个 TBO 使用。

这看起来似乎有些奇怪，但是在 OpenGL 中是完全合理的——毕竟，我们只是在同一个缓冲区中通过两种不同的方法进行读取而已。为了对此进行设置，我们生成两个纹理，并将它们绑定到 `GL_BUFFER_TEXTURE` 绑定点，并且使用 `glTexBuffer` 将这些缓冲区连接到它们之上，这部分知识将在本书后面的内容进行讲解。在我们绑定顶点数组对象 A 时，还绑定了纹理 A。在绑定顶点数组对象 B 时，我们还绑定了纹理 B。这样，同一个数据就会同时出现在 `position` 顶点属性和 `tex_position samplerBuffer` 缓冲区纹理中。

完成这些设置的代码并不复杂，但却有些重复。完整实现的代码可以在本书的网站中找到。示例应用程序包含创建和初始化缓冲区、执行双缓冲和对结果进行可视化的代码。这个应用程序将两个顶点固定在了相应位置，所以整个系统不会掉落到屏幕的底部。一旦我们将所有这些缓冲区进行连接，我们就可以在系统中通过调用一次 `glDrawArrays` 来模拟一个时间步长了。系统中的每一个节点都有一个单独的 `GL_POINT` 图元来表示。如果我们对模型视图投影矩阵（存储在统一值 `mvp` 中）进行初始化并让系统开始运行，我们就会看到类似于图 11.2 所示的结果。

图 11.2 所示的图像并不是特别有趣，但是它确实演示了我们的模拟是在正常地运行着。为了使视觉效果更加吸引人，我们可以将点的大小设置得更大，还可以采用另一个使用 `glDrawElements` 和 `GL_LINES` 图元的索引化绘制来对节点间的连接进行可视化。请注意，同一个顶点位置可以作为这第二个传递的输入来使用，但是我们需要构建另外一个带有 `GL_ELEMENT_ARRAY` 绑定的缓冲区来使用，其中包含到每个弹簧末端的顶点的索引。示例程序中同样执行了这个附加步骤。图 11.3 所示为最终结果。

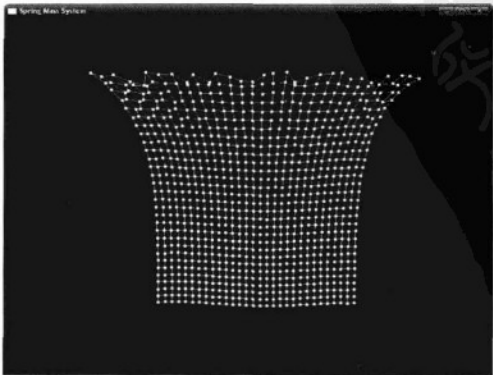
图 11.2

对由弹簧连接的点进行模拟



图 11.3

弹簧物块系统中的可视化弹簧



当然,物理模拟(以及由它产生的顶点数据)可以用于任何情况。如果我们不想在屏幕上绘制这些点,可以启用 `GL_RASTERIZER_DISCARD`,正如下一章将要介绍的。

## 11.2 几何着色器

几何着色器是一种新的着色器类型,最初是以 OpenGL 扩展的形式引入的,然后又成为了 OpenGL 核心规范(即 OpenGL 3.2)的一部分。与其他着色器类型相比,几何着色器的与众不同之处在于,它一次性对整个图元(三角形、线或点)进行处理,并且实际上可以改变 OpenGL 管线中的数据量。一个顶点着色器一次可以处理一个顶点,它无法对其他顶点的信息进行访问,并且它是严格单进单出的。这就是说,它不能生成新的顶点,并且它不能阻止 OpenGL 对这个顶点进行进一步的处理。类似地,片段着色器一次只能处理单个片段,无法访问其他片段的数据,不能创建新的片段,并且只能通过丢弃片段来销毁它们。而另一方面,一个几何着色器可以对一个图元(新的 `GL_TRIANGLES_ADJACENCY` 和 `GL_TRIANGLE_STRIP_ADJACENCY` 图元模式可以支持多达 6 个)中的所有顶点进行访问,可以改变一个图元的类型,甚至可以创建和销毁图元。

几何着色器与顶点着色器及片段着色器的另一个区别是,几何着色器是 OpenGL 管线中的可选部分。只将一个顶点着色器和片段着色器连接到一个程序对象上是完全合法的,并且一直到现在为止这是我们还使用 OpenGL 的唯一方式。在没有出现几何着色器时,OpenGL 管线就像平常一样操作;顶点着色器的输出在进行渲染的图元上进行插值,并直接传递到片段着色器。但是,在加入几何着色器之后,顶点着色器的输出就成为几何着色器的输入,而几何着色器的输出则进行插值并传递到片段着色器。几何着色器可以对顶点着色器的输出进行进一步的处理,并且如果它产生新图元(称为放大(amplification)),可以在创建每个图元时对它们应用不同的变换。

### 11.2.1 直通几何着色器

和顶点着色器与片段着色器一样,几何着色器也是用 GLSL 编写的,并且它们并没有什么神秘之处。稍后我们将对此进行全面的解释,而程序清单 11.2 则完整地展示了一个简单的几何着色器。

程序清单 11.2 一个简单的几何着色器的源代码

```
#version 330

precision highp float;

layout (triangles) in;
layout (triangle_strip) out;
layout (max_vertices = 3) out;

void main(void)
{
    int i;

    for (i = 0; i < gl_in.length(); i++) {
```

```

        gl_Position = gl_in[i].gl_Position;
        EmitVertex();
    }
    EndPrimitive();
}

```

这是一个非常简单的直通几何着色器，它将输入直接进行输出，而没有进行任何修改。这看起来像是一个顶点着色器，但是这里还有几点不同之处，我们读几行代码就会非常清楚了。就像任何顶点着色器或片段着色器一样，代码中的前几行只是设置了着色器的版本号（330）和精度。接下来的几行代码是几何着色器特有的第一项内容。程序清单 11.3 再次展示了这些代码。

程序清单 11.3 几何着色器布局限定符

```

#version 330

precision highp float;

layout (triangles) in;
layout (triangle_strip) out;
layout (max_vertices = 3) out;

```

这段代码使用 layout 限定符设置了输入和输出图元模式。在这个特定着色器中，我们使用 triangles 作为输入，而 triangle\_strip 则作为输出。其他图元类型以及 layout 限定符，稍后将会讲解。对于几何着色器的输出来说，我们不仅指定图元类型，还要指定希望着色器生成的顶点数量（通过 max\_vertices 限定符）。在本例中，这个着色器生成独立的三角形（作为一个非常短的三角形带生成），所以在这里我们指定为 3 个顶点。

接下来是 main() 函数，它看起来与顶点着色器和片段着色器仍然非常相似。这个着色器包含一个循环，而这个循环运行的次数取决于内建数组 gl\_in 的长度。这又是一个几何着色器特有的变量。因为这个几何着色器要对输入图元的所有顶点进行访问，所以它的输入必须声明为一个数组。由顶点着色器（例如 gl\_Position）写入的所有内建变量都保存在一个结构体中，而一个由这些结构体组成的数组将通过一个名为 gl\_in 的变量提交给几何着色器。

gl\_in 数组的长度由输入图元模式决定，而因为在本例中的特定着色器中，输入图元模式为三角形，所以 gl\_in 的大小为 3。程序清单 11.4 重新列出了内循环。

程序清单 11.4 gl\_in 元素的迭代

```

for (i = 0; i < gl_in.length(); i++) {
    gl_Position = gl_in[i].gl_Position;
    EmitVertex();
}

```

在循环中，我们通过简单地将 gl\_in[] 的元素复制到几何着色器的输出来生成顶点。一个几何着色器的输出与顶点着色器的输出相似。在这里，我们将写入 gl\_Position，就像在顶点着色器中要做的一样。

在完成所有新顶点属性的设置之后，我们调用 EmitVertex()。这是一个内建函数，它是几何着色器特有的函数，用来通知这个着色器我们已经完成了对这个顶点的工作，它应该将所有这些信息进行存储，并准备开始设置下一个顶点了。

最后，在循环执行完之后，还要调用另外一个几何着色器特有的函数 EndPrimitive()。EndPrimitive() 通知着色器，我们已经完成了为当前图元生成顶点的工作，可以开始处理下一个图元了。我们指定

triangle\_strip 为我们着色器的输出，因此，如果我们继续调用 EmitVertex()3 次以上，OpenGL 将继续向三角形带中添加三角形。

如果我们需要几何着色器生成单独的、独立的三角形或多个不进行连接的三角形带（回想一下，几何着色器可以创建新的几何图形，或者放大几何图形），我们应该在它们之间调用 EndPrimitive()来标记它们的边界。如果我们不在着色器的某处调用 EndPrimitive()，那么图元将自动在着色器末尾结束。

## 11.2.2 在应用程序中使用几何着色器

和其他着色器类型一样，几何着色器通过调用 glCreateShader 函数进行创建，并使用 GL\_GEOMETRY\_SHADER 作为着色器类型，如下所示。

```
glCreateShader(GL_GEOMETRY_SHADER);
```

一旦着色器被创建，就可以像其他任何着色器对象一样使用了。我们通过调用 glShaderSource 来将着色器源代码提交 OpenGL，使用 glCompileShader 函数来对着色器进行编译，并通过调用 glAttachShader 函数来将它连接到一个程序对象。然后程序将像平常一样使用 glLinkProgram 函数进行连接。

现在我们有了一个程序对象，并且有一个几何着色器连接到它上面，当我们使用类似 glDrawArrays 这样的函数绘制几何图形时，顶点着色器将为每个顶点运行一次，几何着色器也将为每个图元（点、线或三角形）运行一次，而片段着色器则将为每个片段运行一次。在我们将几何图形发送到 OpenGL 时所使用的图元模式，必须与几何着色器的输入图元模式相匹配。例如，如果几何着色器的输入图元模式是点，那么我们在调用 glDrawArrays 时也只能用 GL\_POINTS。

如果几何着色器的输入图元模式是三角形，那么我们在调用 glDrawArrays 时可以使用 GL\_TRIANGLES、GL\_TRIANGLE\_STRIP 或 GL\_TRIANGLE\_FAN。表 11.1 完整地列出了几何着色器输入图元模式及其允许的几何图形类型。

表 11.1 几何着色器输入模式允许的绘制模式

几何着色器输入模式	允许的绘制模式
points	GL_POINTS
lines	GL_LINES, GL_LINE_LOOP, GL_LINE_STRIP
triangles	GL_TRIANGLES, GL_TRIANGLE_FAN, GL_TRIANGLE_STRIP
lines_adjacency	GL_LINES_ADJACENCY
triangles_adjacency	GL_TRIANGLES_ADJACENCY

输入几何图形类型在几何着色器的程序体中使用 layout 限定符来指定。输入布局限定符的一般形式是：

```
layout (primitive_type) in;
```

这样就指定了 primitive\_type 为几何着色器将要处理的输入几何图形类型，而 primitive\_type 必须是支持的几何图形模式中的一种，即 points、lines、triangles、lines\_adjacency 或 triangles\_adjacency。几何

着色器为每个图元运行一次。这就意味着，对于 GL\_POINTS 它将为每个点运行一次；对于 GL\_LINES、GL\_LINE\_STRIP 和 GL\_LINE\_LOOP 它将为每条线运行一次；而对于 GL\_TRIANGLES、GL\_TRIANGLE\_STRIP 和 GL\_TRIANGLE\_FAN 它则将为每个三角形运行一次。几何着色器的输入将以数组的形式表示，这个数组包含组成输入图元的所有顶点。

预定义的输入存储在一个叫做 gl\_in[] 的内建数组中，这是一个由结构体组成的数组，程序清单 11.5 展示了它是如何定义的。

程序清单 11.5 gl\_in[] 的定义

```
in gl_PerVertex
{
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
} gl_in[];
```

这个结构体的成员都是写入顶点着色器中的内建变量：gl\_Position、gl\_PointSize 和 gl\_ClipDistance[]。现在我们应该已经对 gl\_Position 和 gl\_PointSize 非常熟悉了，而 gl\_ClipDistance 则将在第 12 章进行解释。这些变量在顶点着色器中以全局变量的形式出现，但是它们的值最终则是作为结构体成员出现在几何着色器中。顶点着色器所写入的其他变量在几何着色器中也会成为数组形式。在独立 varying 变量（varying）的情况下，顶点着色器的输出会像通常一样进行声明，而几何着色器的输入则会有一个相似的声明，除非它们是数组。考虑一个顶点着色器将输出进行如下定义的情况。

```
out vec4 color;
out vec3 normal;
```

那么相应的几何着色器输入则应如下所示。

```
in vec4 color[];
in vec3 normal[];
```

请注意，varying 变量 color 和 normal 在几何着色器中都成为了数组形式。如果我们有大量数据要从顶点着色器传递到几何着色器，那么将这些从顶点着色器传递到几何着色器的逐顶点数据包装成一个接口块（interface block）可能会非常方便。在这种情况下，我们的顶点着色器将进行如下定义。

```
out VertexData
{
    vec4 color;
    vec3 normal;
} vertex;
```

那么相应的几何着色器输入则应如下所示。

```
in VertexData
{
    vec4 color;
    vec3 normal;
    // 这里可以插入更多的逐顶点属性
} vertex[];
```

有了这些声明，我们就可以使用 vertex[n].color 等对几何着色器中的逐顶点数据进行访问了。几何着色器中输入数组的长度取决于将要处理的图元类型。例如，点是由单个顶点构成的，那么这个数组将只包

含一个单独的元素；而三角形则是由 3 个顶点构成的，这样数组将包含 3 个元素。如果我们编写的着色器特别设计用来处理特定的图元类型，那么我们可以显式地指定输出数组的大小，这样就会使编译时的错误检查的工作量比较少。否则的话，我们可以让输入图元类型布局限定符来自动指定数组的大小。表 11.2 列出了输入图元模式和输入数组最终大小的完整映射关系。

表 11.2 几何着色器输入数组的大小

输入图元类型	输入数组大小
points	1
lines	2
triangles	3
lines_adjacency	4
triangles_adjacency	6

我们还需要指定几何着色器将要生成的图元类型。

同样，它也是使用一个布局限定符确定的，如下所示。

```
layout (primitive_type) out;
```

这与输入几何图形类型布局限定符相似，唯一的不同就是我们使用 out 关键词来声明着色器的输出。允许的几何着色器输出几何图形类型为 points、line\_strip 和 triangle\_strip。

请注意，几何着色器只支持输出条带图元类型（不包括点——显然不存在点带这种东西）。

最后我们还必须使用一个布局限定符来对几何着色器进行配置。因为几何着色器能够为每个顶点产生一个不同数量的数据，所以必须通过指定几何着色器预计要生成的顶点最大数量来告诉 OpenGL 要为所有这些数据分配多少存储空间。我们使用下列布局限定符完成这项工作。

```
layout (max_vertices = n) out;
```

这样就将几何着色器可能产生的顶点数的最大值设为 n 了。因为 OpenGL 可能会分配缓冲区空间来存储每个顶点的中间结果，所以这个最大值应该为能够让应用程序正确运行的可能的最小数量。例如，如果我们计划接收点并一次生成一条线，那么我们就可以将它设为 2，这样设置是安全的。这样就为着色器硬件提供了最好的条件，使其得以最快地运行。如果我们要对输入的几何图形进行大量的分格化，那么我们就可能希望将它设为一个大得多的数字，虽然这样可能会让我们付出一些性能上的代价。一个几何着色器所能生成的顶点数量的上限值取决于 OpenGL 实现。可以保证这个值最小为 256，但是绝对的最大值要通过调用以 GL\_MAX\_GEOMETRY\_OUTPUT\_VERTICES 为参数的 glGetIntegerv 来查询。

我们还可以通过使用逗号进行分隔的方式用一条语句声明一个以上的布局限定符，如下所示。

```
layout (triangle_strip, max_vertices = n) out;
```

有了这些布局限定符，一个样板#version 声明，以及一个空的 main()函数，我们就应该能够生成可以进行编译和连接，但是什么事情也不做的几何着色器了。实际上，它会丢弃我们发送给它的任何几何图形，而应用程序不会进行任何绘制。我们需要引入两个重要的函数：EmitVertex() 和 EndPrimitive()。如果我们不调用这两个函数，就不能绘制任何东西。

EmitVertex 通知几何着色器我们已经完成了这个顶点所有信息的填充。设置顶点的工作与顶点着色器非常相似。我们需要写入内建变量 `gl_Position`。这是为了设置由几何着色器生成顶点的裁剪空间坐标，就像在顶点着色器中一样。我们想要从几何着色器传递到片段着色器的任何其他属性都可以在一个接口块中进行声明，或者也可以在几何着色器中作为全局变量进行声明。在调用 `EmitVertex` 时，几何着色器会将当前所有输出变量中的值进行存储，并使用它们生成一个新的顶点。在一个几何着色器中，我们可以对 `EmitVertex` 进行任意次数的调用，直到达到在 `max_vertices` 布局限定符中指定的限制为止。每一次调用我们都会向输出变量中放入新值，来生成新的顶点。

关于 `EmitVertex`，有一点很重要，我们一定要注意，就是它会将任意输出变量（例如 `gl_Position`）的值都变成未定义的。因此，举例来说，如果我们希望用一种单色来输出一个三角形，那么需要在每个顶点都写入这种颜色，否则就会得到未定义的结果。

`EmitPrimitive` 表示我们已经完成了将顶点附加到图元末端的工作。不要忘记，几何着色器只支持条带图元类型（`line_strip` 和 `triangle_strip`）。

如果输出图元类型为 `triangle_strip`，并且调用 `EmitVertex` 的次数大于 3 次，那么几何着色器将在一个条带中生成多个三角形。类似地，如果输出图元类型为 `line_strip`，并且调用 `EmitVertex` 的次数大于两次，那么就会得到多条线段。

在几何着色器中，`EndPrimitive` 指的是条带。这就意味着如果我们想要绘制独立的线或三角形，就必须在每两个或 3 个顶点之后调用 `EndPrimitive`。我们还可以通过在多次调用 `EndPrimitive` 之间，对 `EmitVertex` 进行多次调用，来绘制多重条带。

关于在几何着色器中调用 `EmitVertex` 和 `EndPrimitive`，还有最后一件事情需要注意，就是如果还没有生成足够的顶点来构成一个图元（例如，我们要生成 `triangle_strips`，而在两个顶点之后就调用 `EndPrimitive`），那么不会产生任何图元，而已经生成的顶点也将简单地被丢弃。

### 11.2.3 在几何着色器中丢弃几何图形

程序中的几何着色器会为每个图元运行一次。要对这个图元做什么，完全由我们决定。`EmitVertex` 和 `EndPrimitive` 两个函数允许我们通过编程将新的顶点附加到三角形带或线带上，并开始新的条带。我们可以任意多次地调用它们（直到达到我们实现所定义的最大值）。我们还可以完全不调用它们，这样就能够将几何图形裁剪掉，也可以丢弃图元。如果几何着色器正在运行而我们从没有为特定几何图形调用 `EmitVertex`，那么就不会进行任何绘制。为了阐明这一点，我们可以使用一个自定义背面剔除例程，对几何图形进行剔除，就像是从空间中的任意一个点进行观察一样。

首先，设置着色器版本和精度，声明着色器接受三角形并生成三角形带。背面剔除对于线和点来说并没有太大的意义。我们还定义了一个统一值，它将保存我们在空间中的自定义视点。程序清单 11.6 显示了这些代码。

程序清单 11.6 配置自定义剔除几何着色器

```
#version 330
precision highp float;
```

```
// 输入为三角形，输出为三角形带 因为我们要建立一个单入单出着色器来为每个输入产生一个三角形，所以在这里
max_vertices 可以为 3
layout (triangles) in;
layout (triangle_strip, max_vertices=3) out;

// Uniform 变量将保存我们的自定义视点
uniform vec3 viewpoint;
```

现在，在 main() 函数中，我们需要为三角形找到表面法线。它只是三角形平面内任意两个向量的叉乘积而已——在这里我们可以使用三角形边缘。程序清单 11.7 显示了这些工作是如何完成的。

程序清单 11.7 在几何着色器中找出表面法线

```
// 在输入三角形所在平面上计算两个向量
vec3 ab = gl_in[1].gl_Position.xyz - gl_in[0].gl_Position.xyz;
vec3 ac = gl_in[2].gl_Position.xyz - gl_in[0].gl_Position.xyz;
vec3 normal = normalize(cross(ab, ac));
```

现在我们已经有了法线，可以确定它是要面向还是背向用户定义视点了。要完成这项工作，我们需要将法线变换到与视点相同的坐标空间，也就是整个场景空间。假定我们有一个统一值格式的模型视图矩阵，将法线与此矩阵相乘即可。为了使结果更加精确，我们应该将这个向量与此模型视图矩阵左上角的 3x3 子矩阵的转置矩阵的逆矩阵相乘。这就是通常所说的正规矩阵，我们可以自由地实现它，并将它放在它自己的统一值中，如果我们想这样做的话。但是，如果我们的模型视图矩阵只包含平移、统一值缩放（不进行裁剪）和旋转的话，那么我们可以直接使用它。不要忘记，法线是一个 3 元素的向量，而模型视图矩阵则是一个 4x4 矩阵。我们需要先将法线扩展成一个 4 元素的向量，然后才能将它们相乘。然后我们可以接受刚才得到的 4 元素向量与从视点到三角形上任意点的向量的点乘积。

如果得到的点乘积的符号是负值，这就说明法线方向是背向观察者的，而三角形应该被剔除。如果这个符号为正，三角形的法线将指向观察者，我们应该让三角形继续通过。程序清单 11.8 列出了进行表面法线变换、执行点乘操作和测试结果符号正负的代码。

程序清单 11.8 在一个几何着色器中对几何图形进行有条件的输出

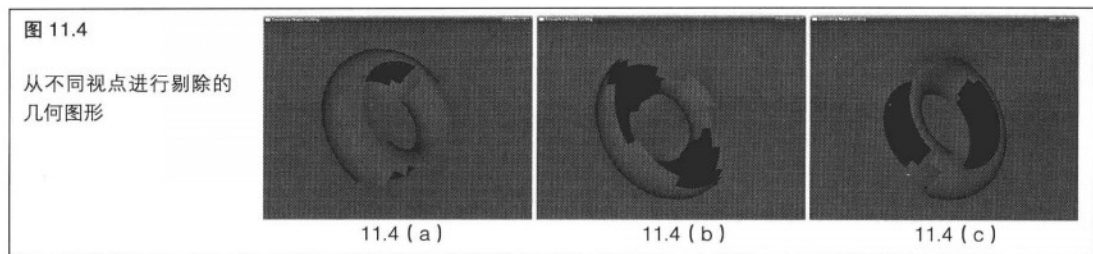
```
// 计算变换的表面法线和观察方向的向量
vec3 transformed_normal = (vec4(normal, 0.0) * modelview_matrix).xyz;
vec3 vt = normalize(gl_in[0].gl_Position.xyz - viewpoint);
// 接受法线和观察方向的点乘积
float d = dot(vt, normal);

// 只有在图元的点乘积符号为正时才能输出
if (d > 0.0) {
    for (int i = 0; i < 3; i++) {
        gl_Position = gl_in[i].gl_Position;
        EmitVertex();
    }
    EndPrimitive();
}
```

在程序清单 11.8 中，如果点乘积为正，我们将输入顶点复制到几何着色器的输出，并为每个顶点调用 EmitVertex；如果点乘积为负，那么我们什么都不做就可以了。这样就会导致输入的三角形一起被丢弃，不会进行任何绘制。

在这个特定的示例中，我们最多为每个输入几何着色器的三角形生成一个三角形输出。虽然几何着色器的输出是一个三角形带，但是我们的条带只包含一个三角形。这样，就不是一定需要调用 EndPrimitive 了。我们在这里保留它只是出于完整性考虑。

图 11.4 所示展示了包含这个着色器程序的一组截屏图像。



在图 11.4 的每个截图中，虚拟观察者都移动到了不同的位置。就像我们能够看到的，模型的不同部分被几何着色器剔除了。我们并不指望这个示例会特别有用，但是它确实演示了一个几何着色器根据应用程序定义的条件来进行几何图形剔除的能力。

### 11.2.4 在几何着色器中修改几何图形

前面的示例不是丢弃几何图形就是不加修改地让它通过。我们还可以在顶点通过几何着色器的时候对它们进行修改，来创建新的衍生图形。即使我们的几何着色器是一一对应地传递顶点的（也就是说，不进行任何放大或消除），我们仍然可以做一些事情，这些事情是在其他情况下单独使用第一个顶点着色器时不可能做到的。举例来说，如果输入的几何图形是以三角形带或三角形扇的形式出现的，结果得到的几何图形将包含公共顶点和公共边。使用顶点着色器移动公用顶点将会移动共用这个顶点的所有三角形。这样，单独使用顶点着色器来将初始几何图形中的两个共用同一个边的三角形分离就不可能了。但是，这对于几何着色器来说就不成问题了。

让我们来考虑一个接受三角形并产生三角形带的几何着色器。接受三角形的几何着色器的输入是独立的三角形，无论它们最初是来自 `glDrawArrays` 函数调用还是来自 `glDrawElements` 函数调用，或者图元类型是 `GL_TRIANGLES`、`GL_TRIANGLE_STRIP` 还是 `GL_TRIANGLE_FAN`。

除非几何着色器输出 3 个以上顶点，否则结果就是独立的、没有连接的三角形。

在接下来的例子中，我们通过让所有三角形沿着它们的表面法线运动，来使一个模型“爆炸”。原始模型是按照独立三角形进行绘制的，还是按照三角形带或三角形扇绘制的，这并不重要。和前一个示例一样，输入为三角形，输出为三角形带，并且由于我们不对几何图形进行放大或缩减，所以几何着色器生成的最大顶点数为 3。程序清单 11.9 列出了进行这些设置的代码。

程序清单 11.9 设置“爆炸”几何着色器

```
#version 330
precision highp float;

// 输入为三角形，输出为三角形带。因为我们要建立一个单入单出着色器来为每个输入产生一个三角形，所以在这里
// max_vertices 可以为 3
layout (triangles) in;
layout (triangle_strip, max_vertices=3) out;
```

为了将三角形向外“发射”，我们需要计算每个三角形的表面法线。我们可以通过三角形平面内两个向量——三角形的两个边的叉乘积来求出表面法线。为了完成这项任务，我们可以重用程序清单 11.7 中的代码。

现在我们有三角形的表面法线，可以沿着这个法线将顶点射出一定量，这个量由应用程序控制。这个量可以保存在一个统一值（我们称之为 `explode_factor`）中，并且由应用程序进行更新。程序清单 11.10 显示了这些简单的代码。

程序清单 11.10 将一个表面沿着它的法线射出

```
for (int i = 0; i < 3; i++) {
    gl_Position = gl_in[i].gl_Position + vec4(explode_factor * normal, 0.0);
}
```

在一个模型上运行这个几何着色器的结果如图 11.5 所示。这个模型被解体，而独立的三角形显现了出来。

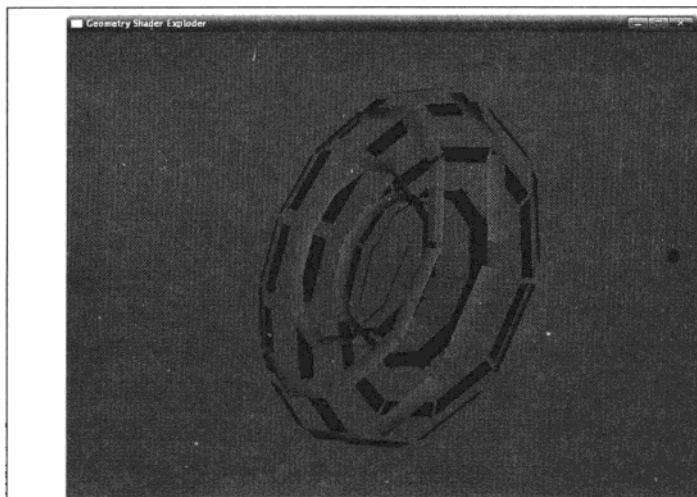


图 11.5

使用几何着色器让一个模型爆炸

## 11.2.5 在几何着色器中生成几何图形

就像完全不要求我们调用 `EmitVertex` 和 `EndPrimitive` 一样，如果不想生成任何来自几何着色器的几何图形的话，也可以对 `EmitVertex` 和 `EndPrimitive` 进行我们需要的任意次数的调用，来产生新的几何图形。这就是说，只要没有达到在几何着色器开头部分声明的输出顶点最大数量，我们就可以继续进行调用。这种功能可以用在创建输入的多个副本或将输入分解成小块の場合，这些是下一节将讨论的主题。输入到着色器的是一个以原点为中心的立方体。立方体的每个面都是由一对三角形组成的，这对三角形共用一条边，这条边就是立方体表面上的对角线。我们通过对角线的中点产生新顶点来对输入的三角形进行分格化，然后移动结果得到所有顶点，以使它们到原点的距离相等。这样立方体就转变成了一个大致上的球体。

因为几何着色器是在对象空间（我们说过，立方体的顶点是围绕着原点这个中心的）中操作的，我们不需要进行任何坐标变换，而是要在生成新顶点之后在几何着色器中进行变换。要完成这项工作，我们需要一个简单的直通顶点着色器。程序清单 11.11 展示了一个非常简单的直通顶点着色器的代码。

程序清单 11.11 一个简单的直通顶点着色器

```
#version 330
precision highp float;

in vec4 position;
```

```
void main(void)
{
    gl_Position = position;
}
```

这个着色器只是将顶点位置传递到几何着色器。如果我们有与顶点相关的其他属性，例如纹理坐标或法线，那么同样需要将它们通过顶点着色器传递到几何着色器。

就像前面的例子一样，我们接受三角形作为几何着色器的输入，并产生一个三角形带。我们在每个三角形之后都将三角形带进行解体，这样就能产生单独的、独立的三角形。在本例中，我们为每个输入三角形生成两个输出三角形。我们需要将最大输出顶点数声明为 6——两个三角形乘以 3 个顶点。我们还需要在几何着色器中声明一个统一矩阵来保存模型视图变换矩阵，因为要在生成顶点之后进行这些变换。程序清单 11.12 显示了这些代码。

程序清单 11.12 设置“分格器”几何着色器

```
#version 330
precision highp float;
layout (triangles) in;
layout (triangle_strip, max_vertices=6) out;

// 一个用来保存模型视图投影矩阵的统一值
uniform mat4 mvp;
```

要确保我们知道哪条边是对角线，可以在程序中使用索引，并使用 `glDrawElements` 绘制立方体。这样就可以总是让第一个顶点作为三角形的顶点，而第二个三角形和第 3 个三角形之间的边作为立方体表面的对角线。对于每个生成的三角形，可以使用与输入三角形相同的第一个顶点。然后，对于生成的两个三角形，我们可以使用生成的顶点和其他输入的顶点中的一个。

首先，将输入的顶点坐标进行标准化，这样就使这些坐标与原点的距离都相等了，因为在这种情况下从原点到任意顶点的向量长度都为 1。如果初始立方体的顶点坐标已经进行了标准化，那么我们就没必要再进行标准化了，但是这就使这个立方体的边长为单位长度——这对于存储几何图形来说是非常普遍的。我们在这里还将结果得到的顶点坐标与模型视图投影矩阵相乘。程序清单 11.13 显示了这些代码。

程序清单 11.13 设置“分格器”几何着色器

```
// 将输入的顶点移动到一个半径为 1 的圆的表面
vec3 a = normalize(gl_in[0].gl_Position.xyz);
vec3 b = normalize(gl_in[1].gl_Position.xyz);
vec3 c = normalize(gl_in[2].gl_Position.xyz);

// 在 b 和 c 的中点生成新的顶点。请注意对顶点进行标准化就意味着我们不需要再除以 2 来取平均了
vec3 d = normalize(b + c);

// 现在将生成的顶点转换到场景空间
a = a * mvp;
b = b * mvp;
c = c * mvp;
d = d * mvp;
```

现在，`a` 成为了三角形的顶点，而 `d` 则是生成的顶点。`bc` 边是立方体面的对角线，也是要将这个面分成两半的边。我们将要输出的两个三角形将是 `abd` 和 `adc`。为了生成两个输出三角形，我们需要设置顶点，为每个顶点调用 `EmitVertex`，然后在每个三角形之间调用 `EndPrimitive` 重新开始三角形带。程序清单 11.14 显示了这些代码。

## 程序清单 11.14 输出分格化的顶点

```
// 生成第一个三角形 abd
gl_Position = a;
EmitVertex();
gl_Position = b;
EmitVertex();
gl_Position = d;
EmitVertex();
EndPrimitive();

// 生成第二个三角形 adc
gl_Position = a;
EmitVertex();
gl_Position = d;
EmitVertex();
gl_Position = c;
EmitVertex();
EndPrimitive();
```

在本例中，我们为每个输入三角形生成两个独立的三角形。但是，这两个三角形实际上共用了 ad 边，并且通过以正确的顺序对这些顶点进行输出，它们可以用一个三角形带来表示，这个三角形带正是几何着色器设计上要输出的。新的三角形生成代码如程序清单 11.15 所示。

## 程序清单 11.15 使用三角形带进行分格化

```
gl_Position = b;
EmitVertex();
gl_Position = d;
EmitVertex();
gl_Position = a;
EmitVertex();
gl_Position = c;
EmitVertex();
EndPrimitive();
```

请注意我们只对 EmitVertex 进行了 4 次而不是 6 次调用，而且删除了一个 EndPrimitive 调用。对于这个几何着色器，我们可以将 max\_vertices 减少到 4，这样可以使程序运行得更快。即使将程序渲染的三角形的数量加倍，在使用一个短条带时，也只是将要处理的顶点数增加了三分之一。

图 11.6 所示展示了基于我们的简单几何着色器的分格化程序的一组截屏图像。

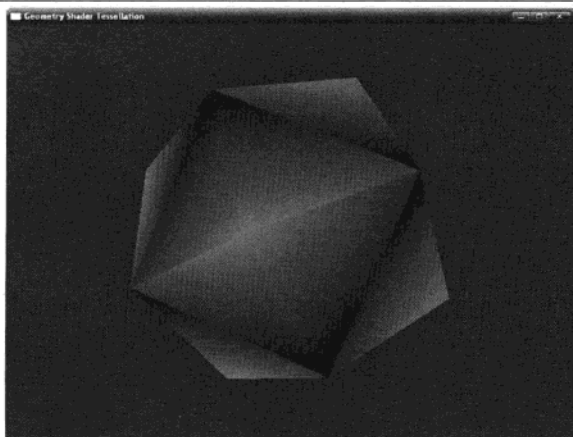


图 11.6

使用几何着色器的基本分格化

有一点需要注意，在复杂的分格化中使用几何着色器可能得不到最优的性能。如果需要进行比本例更复杂的分格化，最好使用一个 OpenGL 扩展进行硬件分格化。对这些扩展的全面讲解超出了本书涉及的范围。

### 11.2.6 在几何着色器中改变图元类型

到现在为止，我们讲解的所有几何着色器都接受三角形为输入，并且生成三角形带为输出，这样并没有改变几何图形的类型。但是，几何着色器还能够输入和输出不同类型的几何图形。例如，我们可以将点转换成三角形，或者将三角形转换成点。在下一个示例中，我们将把几何图形的类型从三角形转换成线。对于每个输入着色器的顶点，我们获取顶点法线并将它表示为一条线。我们还要获取表面法线，并将它表示为另一条线。这样就可以显示模型的法线了，包括每个顶点和每个面的法线。不过，要注意的是，如果绘制原始模型顶部的法线，就需要将所有东西进行两次绘制——一次是使用几何着色器来将法线可视化，而另一次则是显示模型而不使用几何着色器。我们无法从一个几何着色器中混合输出两种不同的图元。

对于几何着色器，除了 `gl_in` 结构体的成员之外，还需要每个顶点的法线，它们还要直通过顶点着色器。程序清单 11.16 列出了对程序清单 11.11 的代码进行改进而得到的升级版直通顶点着色器。

程序清单 11.16 一个包含法线的直通顶点着色器

```
#version 330
precision highp float;

in vec4 position;
in vec3 normal;

out Vertex
{
    vec3 normal;
} vertex;

void main(void)
{
    gl_Position = position;
    vertex.normal = normal;
}
```

这样就将 `position` 属性直接传递到了 `gl_Position` 内建变量中，并将法线放到了一个输出块中。

程序清单 11.17 列出了对这个几何着色器进行设置的代码。在本例中，我们为接受三角形而生成线带，每个线带只有一条线。因为我们为显示的每条法线输出了一个单独的线，所以我们为每个消耗的顶点生成两个顶点，还要为表面法线多生成两个顶点。这样，我们为每个输入的三角形所输出的最大顶点数就是 8 个。为了与在顶点着色器中声明的顶点输出块相匹配，我们还需要在几何着色器中声明一个相应的输入接口块。因为我们要在几何着色器中进行从对象空间到场景空间的变换，所以要声明一个名为 `mvp` 的 `mat4` 统一值来表示模型视图投影矩阵。这样做是必要的，以便我们可以将顶点位置与它的法线保持在同一个坐标系中，直到生成表示这条线的新顶点。

程序清单 11.17 设置“法线可视化”几何着色器

```
#version 330
precision highp float;
```

```
layout (triangles) in;  
layout (line_strip) out;  
layout (max_vertices = 8) out;  
  
in Vertex  
{  
    vec3 normal;  
} vertex[];  
  
// 用来保存模型视图投影矩阵的统一值  
uniform mat4 mvp;  
  
// 用来保存可视化法线长度的统一值  
uniform float normal_length;
```

每个输入顶点都会变换到最终位置，并且从几何着色器中输出，然后沿着它的法线取代输入顶点并将其变换到最终位置而产生第二个顶点。这样就将所有法线的长度变成 1，但是允许在模型视图投影矩阵中进行编码的任何缩放随着模型一起应用到它们之上。我们将法线与应用程序支持的统一值 `normal_length` 相乘，允许它们进行缩放，从而与模型相匹配。程序清单 11.18 列出了内循环。

程序清单 11.18 在几何着色器中由法线生成线

```
for (int i = 0; i < gl_in.length(); i++) {  
    gl_Position = mvp * gl_in[i].gl_Position;  
    EmitVertex();  
    gl_Position = mvp * vec4(gl_in[i].gl_Position.xyz +  
                             vertex[i].normal * normal_length, 1.0);  
    EmitVertex();  
    EndPrimitive();  
}
```

这样就在每个顶点生成了一个短的线片段，指向法线方向。

现在我们需要生成表面法线。要完成这项工作，我们需要挑选一个合适的位置，从这个位置来绘制法线，然后我们需要在几何着色器中计算表面法线本身，我们就是要沿着这个法线来绘制线的。

就像前面程序清单 11.7 中的示例一样，我们使用三角形两条边的一个叉乘积找出表面法线。要为此条线选定一个起点，我们选择三角形的形心，它的坐标只是简单地取输入顶点坐标的平均值。程序清单 11.19 显示了这个着色器的代码。

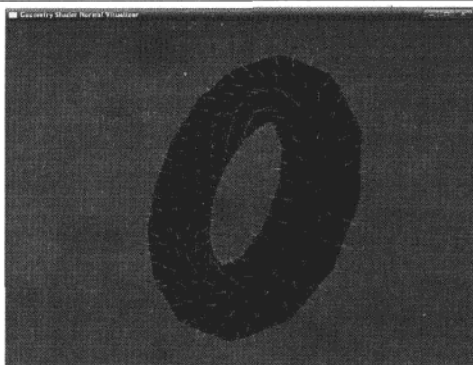
程序清单 11.19 在几何着色器中绘制表面法线

```
vec4 centroid = (gl_in[0].gl_Position +  
                gl_in[1].gl_Position +  
                gl_in[2].gl_Position) / 3.0;  
vec3 face_normal = normalize(cross(gl_in[1].gl_Position.xyz -  
                                   gl_in[0].gl_Position.xyz,  
                                   gl_in[2].gl_Position.xyz -  
                                   gl_in[0].gl_Position.xyz));  
gl_Position = centroid * mvp;  
  
EmitVertex();  
gl_Position = (centroid + vec4(face_normal * normal_length, 0.0)) * mvp;  
EmitVertex();  
EndPrimitive();
```

现在，当我们对模型进行渲染时，就会得到如图 11.7 所示的图像。

图 11.7

使用几何着色器显示一个模型的法线



### 11.2.7 由几何着色器引入的新图元类型

随几何图形引入了 4 种新的图元类型，它们是 `GL_LINES_ADJACENCY`、`GL_LINE_STRIP_ADJACENCY`、`GL_TRIANGLES_ADJACENCY` 和 `GL_TRIANGLE_STRIP_ADJACENCY`。这些图元类型确实是在一个几何着色器活动的情况下进行渲染时才有用的。

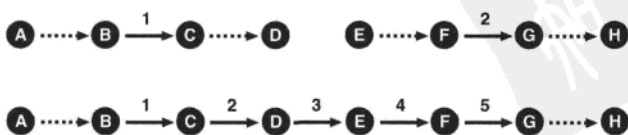
在使用新的邻接图元类型 (adjacency primitive type) 时，对于传递到几何着色器的每条线或每个三角形来说，它不仅访问定义这个图元的顶点，还可以访问与它正在处理的图元相邻的图元的顶点。

当使用 `GL_LINES_ADJACENCY` 进行渲染时，每个线片段要从启用的属性数组中消耗 4 个顶点。中间的两个顶点组成这条线；第一个和最后一个顶点则被视为邻接顶点。这样，输入到着色器的就是一个 4 元素的数组。实际上，因为几何着色器的输入和输出类型不需要关联，`GL_LINES_ADJACENCY` 可以看作是一种将广义的 4 顶点图元发送到几何着色器的方法。几何着色器可以自由地将它们转换成任何它想转换的东西。例如，我们的几何着色器可以将每一组 4 个顶点转换成一个由两个三角形组成的三角形带。这样就允许我们使用 `GL_LINES_ADJACENCY` 图元来渲染四边形。不过，我们应当注意，如果在没有活动几何着色器的情况下使用 `GL_LINES_ADJACENCY` 进行绘制，那么就会使用每组 4 个顶点中最靠中间的两个顶点来绘制常规的线。而靠外侧的两个顶点将被丢弃，顶点着色器根本就不会对它们进行任何处理。

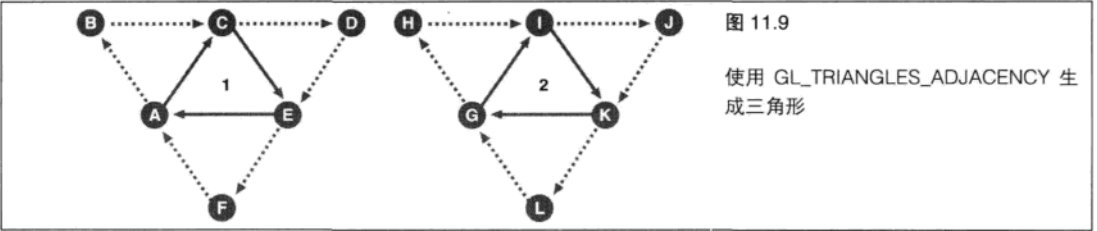
使用 `GL_LINE_STRIP_ADJACENCY` 也会产生类似的效果。不同之处在于，这时整个条带将会看作是一个图元，而在每一端都带有一个附加的顶点。如果使用 `GL_LINES_ADJACENCY` 将 8 个顶点发送到 OpenGL，那么几何着色器将运行两次，然而如果使用 `GL_LINE_STRIP_ADJACENCY` 发送同样一批顶点时，几何着色器则会运行 5 次。图 11.8 所示清楚地展示了这一点。最上面一行的 8 个顶点以 `GL_LINES_ADJACENCY` 图元模式发送到 OpenGL。几何着色器每次为 4 个顶点运行两次——ABCD 和 EFGH。在第二行，同样的 8 个顶点以 `GL_LINE_STRIP_ADJACENCY` 图元模式发送到 OpenGL。这一次，几何着色器运行 5 次——ABCD、BCDE，以此类推直到 EFGH。在这种情况下，如果没有出现几何着色器的话，那么实线箭头就被渲染。

图 11.8

使用 `GL_LINES_ADJACENCY` 和 `GL_LINE_STRIP_ADJACENCY` 生成的线

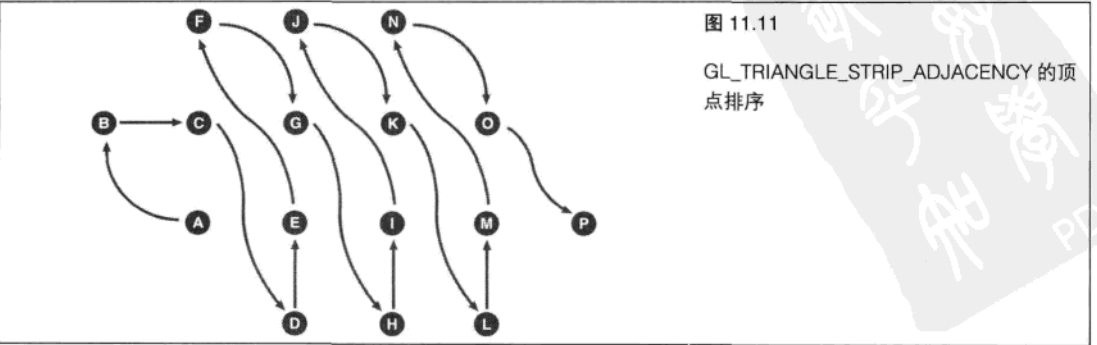
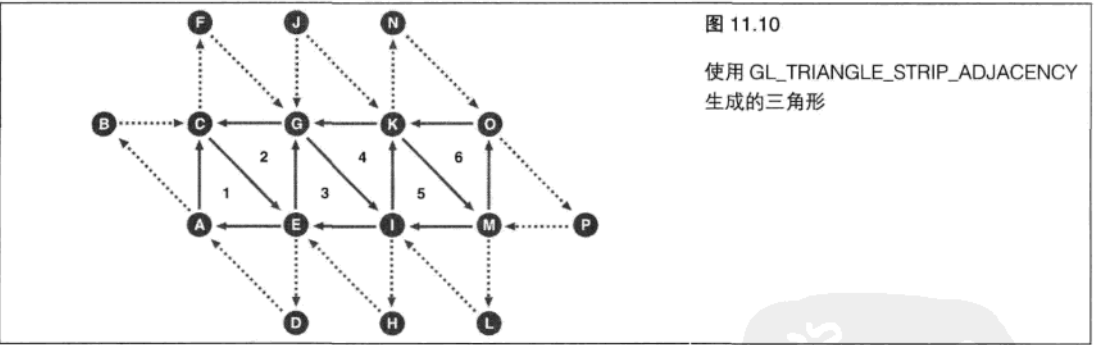


GL\_TRIANGLES\_ADJACENCY 图元模式的使用与 GL\_LINES\_ADJACENCY 模式类似。为启用的属性数组中每一组 6 个顶点都要向几何着色器传递一个三角形。第 1 个、第 3 个和第 5 个顶点被认为是组成真正三角形的顶点，而第 2 个、第 4 个和第 6 个顶点则被认为是在三角形的顶点之间。这就是说，输入到几何着色器的是一个 6 个元素的数组。和以前一样，我们可以使用几何着色器对这些顶点进行我们想要做的任何操作，GL\_TRIANGLES\_ADJACENCY 是一种很好的将任意 6 个顶点的图元传递到几何着色器的方法。图 11.9 所示展示了这一点。



这些图元类型中的最后一种就是 GL\_TRIANGLE\_STRIP\_ADJACENCY，或许这也是最复杂的一种（或者说是最难理解的一种）。这种图元表示一个三角形带，每隔一个顶点来选用一个顶点（第 1 个、第 3 个、第 7 个、第 9 个……）组成这个三角形带。它们中间的顶点为邻接顶点。图 11.10 所示演示了这种规则。在图中，从顶点 A 到顶点 P 代表发送到 OpenGL 的 16 个顶点。每隔一个顶点选用一个顶点（A、C、E、G、I 等）生成一个三角形带，而夹在这些选用顶点之间的顶点（B、D、F、H、J 等）则为邻接顶点。

对于位于三角形带始端和末端的三角形来说，还有一些特殊情况，但是一旦三角形带开始，顶点就会进入一种常规模式，这一点在图 11.11 中表现得更加清晰。



关于 `GL_TRIANGLE_STRIP_ADJACENCY` 的排序规则，在 OpenGL 规范中已经做了详细的说明——特别是，对于那些特殊情况也进行了提示。如果想要使用这种图元类型，我们建议读者阅读规范中的相关部分。

## 11.3 高级片段着色器

片段着色器是管线中一个功能强大的阶段。典型情况下，图形硬件从纹理中进行读取和写入到帧缓冲区时会使用最大的内存带宽。但是，我们并不只限于在片段着色器中进行视觉数据的处理。

如果我们要进行数据密集型的操作，那么片段着色器可能是最适合的地方了。到目前为止，我们已经利用纹理贴图、法线贴图等技术，使用片段着色器来模拟了材质和表面。片段着色器会在由图元生成的每个片段上执行。但是，除了对渲染对象的表面进行模拟之外，我们还可以使用片段着色器做更多的工作。

在这一部分，我们将介绍片段着色器的更高级应用。对于这些示例中的大多数来说，我们使用单独的一对三角形作为输入几何图形覆盖整个屏幕。这通常被称为全屏四边形，因为这是一个覆盖了整个屏幕的四边形。首先，我们回顾一下使用片段着色器来应用模糊和颜色校正与增强等后期处理特效。然后，我们来展示只使用片段着色器就能够生成的整个场景。

在下面的几个例子中，我们使用同一个简单的直通顶点着色器。这个着色器所做的工作只是将输入坐标传递到输出位置，并将它复制到一个纹理坐标，以便片段着色器可以对它进行访问。程序清单 11.20 显示了这些代码。

程序清单 11.20 全屏四边形直通顶点着色器

```
#version 330
precision highp float;

in vec2 position;

out Fragment
{
    vec2 tex_coord;
} fragment;

void main(void)
{
    gl_Position = vec4(position, 0.5, 1.0);
    // 这样就生成了一个范围从(-1.0, -1.0)到(1.0, 1.0)的纹理坐标
    fragment.tex_coord = position;
    // 或者，我们也可以使用下面这行代码将我们的纹理坐标范围设置成从(0.0, 0.0)到(1.0, 1.0)
    fragment.tex_coord = position * 0.5 + vec2(0.5, 0.5);
}
```

这个着色器的输入是一个单独的 `vec2` 属性 `position`，它是从坐标直接传递过来的，代表四边形的角，如程序清单 11.21 所示。

程序清单 11.21 全屏四边形几何图形

```
const GLfloat quad_coords[] =
{
    -1.0f, -1.0f,
```

```
1.0f, -1.0f,  
-1.0f, 1.0f,  
1.0f, 1.0f  
};
```

使用这些坐标并将它们作为一个 `GL_TRIANGLE_STRIP` 进行绘制，一个四边形可以通过调用一次 `glDrawArrays` 进行渲染，覆盖整个视口。顶点着色器输出的 `fragment.tex_coord` 的范围可以从  $(-1.0, -1.0)$  到  $(1.0, 1.0)$ ，也可以是从  $(0.0, 0.0)$  到  $(1.0, 1.0)$ ，这取决于哪一行被取消注释，就像着色器的注释中所解释的一样。这些示例中有一些要求范围为  $-1.0$  到  $1.0$ ，也有一些要求范围为  $0.0$  到  $1.0$ 。

### 11.3.1 片段着色器中的后期处理——颜色校正

在本例中，我们假定在一个纹理中有一个输入图像。这可能是一个预生成的图像，或者也可以是将场景渲染到一个绑定到帧缓冲区对象（FBO）的纹理而得到的结果。要学习更多关于 FBO 和纹理渲染的知识，可以参考第 8 章。在本章，我们从纹理中读取单个纹理单元，对存储到其中的颜色进行变换，并从片段着色器中对其进行输出向用户显示。在我们从输入纹理中进行采样时，需要纹理坐标的范围为从  $(0.0, 0.0)$  到  $(1.0, 1.0)$ ，所以需要启用直通顶点着色器的这个变体。

要应用颜色校正，我们要使用一个矩阵来对每个片段进行变换。通过把这个矩阵放到一个统一值中，应用程序可以在运行时对这个矩阵进行更新，以生成不同的效果。程序清单 11.22 列出了对片段着色器进行设置的代码。

程序清单 11.22 设置颜色校正片段着色器

```
#version 330  
precision highp float;  
  
// 这是用于从虚拟顶点着色器传递纹理坐标的接口块  
in Fragment  
{  
    vec2 texcoord;  
} fragment;  
  
// 这个统一值包含了用于进行颜色校正的矩阵  
uniform mat4 color_matrix;  
  
// 代表我们输入图像的采样器  
uniform sampler2D tex_input_image;  
  
// 最终颜色  
out vec4 final_color;
```

现在我们有片着色器设置的输入，可以继续进行颜色校正着色了。每个输出片段都是从源图像的一个纹理单元直接生成的。我们可以通过使用一个变换矩阵在输入颜色中应用一个广义投影变换。存储在输入纹理中的颜色是 RGB 格式的。我们需要通过将 alpha 通道设置为 1.0 来将它扩展成一个齐次向量，就像 OpenGL 位置一样。然后，我们可以用它乘以变换矩阵，并除以 w 坐标。着色器程序体真的非常简单——只有几行代码，如程序清单 11.23 所示。

程序清单 11.23 颜色校正片着色器的程序主体

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

```

// 从着色器中读取输入颜色，并将它转换成一个齐次向量
vec4 input_color = vec4(texture(tex_input_image,
                                fragment.tex_coord).rgb, 1.0);
// 使用我们的颜色变换矩阵来对它进行转换
vec4 transformed_color = color_matrix * input_color;
// 这样我们就在最终颜色上生成一个“透视”变换了
final_color = transformed_color / transformed_color.a;
}

```

使用这个着色器，我们可以获取一个图像，并将它变换到其他颜色空间，修改亮度，或者调整图像的颜色平衡。图 11.12 所示给出了可以用来在图像上应用有趣效果的矩阵的几个示例。

图 11.12	$\begin{pmatrix} 0.5 & 0.4 & 0.2 & 0.0 \\ 0.4 & 0.3 & 0.2 & 0.0 \\ 0.3 & 0.3 & 0.2 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix}$	$\begin{pmatrix} 0.3 & 0.6 & 0.1 & 0.0 \\ 0.3 & 0.6 & 0.1 & 0.0 \\ 0.3 & 0.6 & 0.1 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix}$	$\begin{pmatrix} 0.0 & 1.0 & 0.0 & 0.0 \\ 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix}$
示例颜色处理矩阵	深褐色调	灰度	交换红色和绿色

这些都是些非常简单的矩阵，所以很容易就能了解它们是怎么运行的。深褐色调本质上是将所有颜色通道一起进行平均，在红色通道上应用轻微的偏置，以使结果得到的图像获得褐色图像中典型的深褐色调。灰度矩阵从输入颜色中获得一个加权平均值——因为矩阵的每一行都是相同的，输出图像中的每个颜色通道都和其他的通道相同。

但是，输入的绿色通道加权比其他通道更高一点，因为人眼对于绿色比其他颜色更加敏感。这个矩阵模拟了视觉系统。红-绿交换矩阵是由简单的 1 和 0 组成的。结果是，绿色通道对于红色输出的加权值为 0，而红色输入对于绿色输出的加权值也为 0。

### 11.3.2 片段着色器中的后期处理——卷积

在颜色校正示例中，每个输出片段都是从单个输入纹理单元中产生的。在本例中，我们可以将着色器的输入扩展成多重输入纹理单元。这样就能允许我们将来自多个纹理的数据进行组合并实现过滤操作了。

我们可以在图像上将一个可分离内核使用两次来实现卷积。可分离内核能够分解成水平向量分量和垂直向量分量，这样就可以求得它们的外积（叉乘积），生成一个二维内核矩阵了。高斯过滤（Gaussian Filter）就是这方面的一个例子，它可以用来生成平滑的模糊效果。

这个示例还是使用了一个全屏四边形作为输入几何图形，并使用直通顶点着色器的变体生成(0.0, 0.0)和(1.0, 1.0)之间的纹理坐标。我们使用一个 TBO（纹理缓冲区对象）来保存这些过滤系数。TBO 是在第 8 章引入的。为了逐步得到输出图像，我们提供一个统一值 `tc_scale`，它为每个参数指定替代输入纹理坐标多远。这样从本质上说就允许我们对我们的过滤内核相对输入图像进行缩放了。通过将它的 *x* 或 *y* 分量设置成 0，我们可以分别垂直或水平地遍历图像。

通过将 *x* 和 *y* 都设置为非 0 值，我们也可以以任意角度来经过输入图像。只要每次通过的向量是正

交的, 过滤器就仍然是可分离的。程序清单 11.24 列出了卷积着色器的输入声明。

程序清单 11.24 卷积片段程序的输入

```
#version 330
precision highp float;

// 来自顶点着色器的输入接口块
in Fragment
{
    vec2 tex_coord;
} fragment;

// 代表我们输入图像的采样器
uniform sampler2D tex_input_image;

// 保存我们过滤参数的 TBO
uniform samplerBuffer tbo_coefficient;

// 将整数缩放到纹理坐标的统一值
uniform vec2 tc_scale;

// 最终输出颜色
out vec4 output_color;
```

过滤器的大小可以由 TBO 的大小决定, 后者可以通过调用 textureSize 函数来查询。这就意味着我们不需要显式地告诉着色器过滤器的大小, 甚至可以通过在水平和垂直方向使用不同大小的 TBO 来使用一个非四边形的过滤器。着色器是以输出片段为中心的。如果我们的过滤器不以输出片段为中心, 那么每次传递都会将图像进行水平或垂直移动。为了执行过滤, 我们在输入图像上的纹理单元进行循环, 用一个来自 TBO 的样本对每个纹理单元进行加权。

着色器的程序体非常简单, 如程序清单 11.25 所示。

程序清单 11.25 可分离的卷积片段着色器

```
int filter_size = textureSize(tbo_coefficient);
vec2 color = vec4(0.0);
vec2 tc_offset;
float coefficient;
for (int i = 0; i < filter_size; i++) {
    coefficient = texelFetch(tbo_vertical_coefficient, i).r;
    tc_offset = float(i - filter_size / 2) * tc_scale;
    color += coefficient * texture(tex_input_image,
                                   fragment.tex_coord + tc_offset);
}

output_color = color;
```

这个着色器的主应用程序非常简单。整个实现都已经包含在本书网站所提供的源代码中了。应用程序将在图像上执行两次。其中第一次使用输入图像作为一个纹理, 并将其渲染到一个绑定到 FBO 的纹理上。第二次使用以前在第一次执行时写入的纹理作为一个输入并渲染到帧缓冲区。

给定图 11.13 所示的输入图像, 使用高斯过滤器进行卷积所得到的结果如图 11.14 所示。

这个图像是使用表 11.3 中的过滤器内核生成的。请注意这个内核的所有加权值加起来等于 1。如果相加的结果不为 1, 就会导致输出图像变得比输入图像更亮或更暗。这个内核同时也是对称的 (高斯内核的一个性质)。如果不是这样, 就会导致输出图像相对于输入发生水平或垂直方向的平移。

表 11.3

高斯模糊示例的过滤器加权值

0.015625	0.09375	0.234375	0.3125	0.234375	0.09375	0.015625
----------	---------	----------	--------	----------	---------	----------

图 11.13

卷积示例程序的输入



图 11.14

在图像上应用高斯模糊的结果



另一个可分离过滤器的示例就是索贝尔边缘检测器 (Sobel edge detector)。图 11.15 所示展示了图 11.13 在应用了索贝尔边缘检测器之后的情况。

图 11.15

在图像上应用索贝尔边缘检测器的结果



表 11.4 给出了索贝尔运算符的加权值。请注意索贝尔过滤器分成了两个不同的内核，每次使用一个。索贝尔运算符在每一次执行时也在水平方向或垂直方向进行边缘检测，它还检测图像过渡，这就意味着我们需要获取结果的强度才能显示结果。这个示例应用程序就能完成这项工作。

表 11.4 索贝尔边缘检测的分离过滤器加权值			
Pass 1	1.0	2.0	1.0
Pass 2	1.0	0.0	-1.0

### 11.3.3 在片段着色器中生成图像数据

在前两个使用片段着色器来执行后期处理的例子中，我们是从一个预渲染图像开始的，它或者是一个由应用程序支持的纹理，或者是使用 OpenGL 渲染到一个纹理的结果。在下一个例子中，我们渲染一个茹利亚集（Julia set），只根据纹理坐标生成图像数据。茹利亚集与曼德勃罗特集（Mandelbrot set）——像球茎形状的分形有关。曼德勃罗特集是通过对方程

$$z_n = z_{n-1}^2 + c$$

进行迭代，直到  $z$  的数值超过一个阈值，并计算迭代的次数而得到的。如果  $z$  的数值在允许的迭代次数内不会超过阈值，那么这个点将被确定为在曼德勃罗特集中，并使用某种默认的颜色进行着色。如果  $z$  的值在允许的迭代次数内超过阈值，那么这个点就在曼德勃罗特集之外。曼德勃罗特集通常的可视化使用一个在点被确定在集合之外时进行计数的迭代函数对点进行上色。曼德勃罗特集与茹利亚集之间的主要区别在于  $z$  和  $c$  的初始条件不同。

在渲染曼德勃罗特集时， $z$  设为  $0+0i$ ， $c$  设为进行迭代的点的坐标。在渲染曼德勃罗特集时， $z$  设为进行迭代的点的坐标，而  $c$  则设为一个应用程序指定的常量。这样，当只有一个曼德勃罗特集时，就有无限多个茹利亚集——对于每个可能的  $c$  值都有一个。正因为这样，茹利亚集可以通过参数进行控制，甚至可以动画化。就像在前面的例子中一样，我们在每个片段通过绘制全屏四边形来调用这个着色器。

让我们使用一个只包含纹理坐标的输入块对片段着色器进行设置，还需要一个统一值来保存  $c$  的值。我们使用一个带有颜色过渡的一维纹理在结果得到的茹利亚图像上应用有趣的颜色。当对一个落到集合之外的点进行迭代时，使用迭代计数对这个纹理进行检索，从而对输出片段进行着色。最后，还定义了一个统一值，它包含我们想要执行的最大迭代次数。这样就允许应用程序在性能和结果得到图像的细节层次之间进行平衡。程序清单 11.26 展示了茹利亚渲染程序的片段着色器的设置。

```
程序清单 11.26 设置茹利亚集渲染程序
#version 330
precision highp float;

in Fragment
{
    vec2 tex_coord;
} fragment;
```

```
// 这里是我们的 c 值
uniform vec2 c;

// 这是颜色过渡纹理
uniform sampler1D tex_gradient;

// 这是我们在考虑集合之外的点之前将要执行的最大迭代数
uniform int max_iterations;

// 这个片段着色器的输出颜色
out vec4 output_color;
```

现在有了着色器的输入，我们已经做好了准备，可以开始渲染茹利亚集了。

$c$  的值是从应用程序提供的统一值中获取的。 $z$  的初始值是从顶点着色器提供的输入纹理坐标中获取的。程序清单 11.27 列出了内循环。

程序清单 11.27 茹利亚渲染程序的内循环

```
int iterations = 0;
vec2 z = fragment.tex_coords;
const float threshold_squared = 4.0;

while (iterations < max_iterations && dot(z, z) < threshold_squared)
{
    vec2 z_squared;
    z_squared.x = z.x * z.x - z.y * z.y;
    z_squared.y = 2.0 * z.x * z.y;
    z = z_squared + c;
    iterations++;
}
```

循环在发生两种情况之一时结束——或者我们达到迭代允许的最大值 ( $iterations = max\_iterations$ )，或者  $z$  的值通过了阈值。请注意，在这个着色器中，我们将  $z$  的值的平方（使用 `dot` 函数得到）与阈值的平方（统一值 `threshold_squared`）进行比较。这两种操作是等价的，但是这样做可以避免在着色器中出现平方根，从而提高了性能。如果在循环末尾 `iterations` 等于 `max_iterations`，我们就会知道已经超出了阈值，而这个点在集合中——我们将它设成黑色。或者，点在超出阈值时离开集合，而我们则可以相应地对这个点着色。

如果要这样做，我们可以查出使用的是整个允许阈值的哪部分，并用它对过渡纹理进行查询。程序清单 11.28 展示了相应的代码。

程序清单 11.28 使用过渡纹理对茹利亚集进行上色

```
if (iterations == max_iterations) {
    output_color = vec4(0.0, 0.0, 0.0, 0.0);
} else {
    output_color = texture(tex_gradient,
                           float(iterations) / float(max_iterations));
}
```

现在所剩的工作只有支持过渡纹理并为  $c$  设置一个正确的值了。对于应用程序来说，我们在每一帧都对  $c$  进行更新。通过跟踪记录我们渲染过的帧数，我们可以对分形进行动画化。图 11.16 所示展示了程序生成的茹利亚集动画的几个帧。（彩插中的彩图 3 提供了另一个例子）

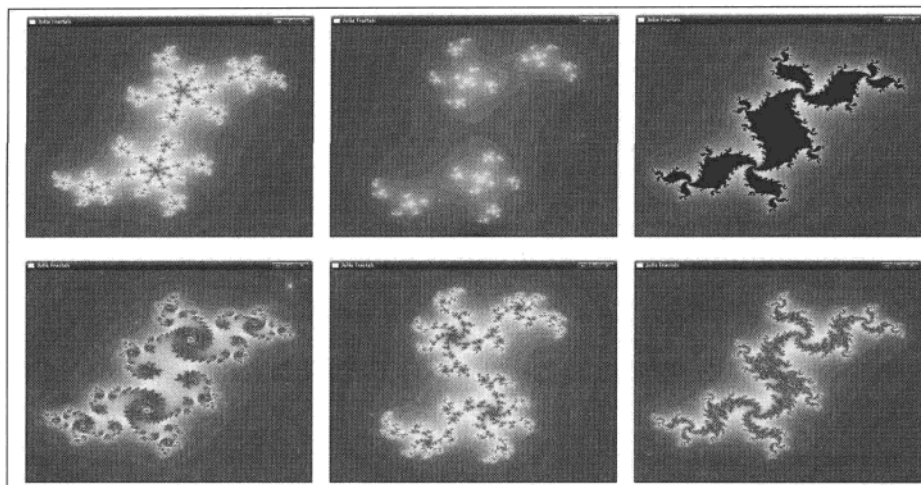


图 11.16  
茹利亚集动画  
的几个帧

### 11.3.4 在片段着色器中丢弃工作

片段着色器是一种强大的工具，它可以帮助我们计算渲染像素的颜色。渲染的形状取决于渲染的几何图形。如果想要绘制形状更加细致的图形，就需要向 OpenGL 发送更多的几何图形。我们可以在部分透明的几何图形上使用 alpha 混合，但是到目前为止，还不能使用片段着色器控制形状。如果片段着色器决定某些对象是完全透明的，OpenGL 仍然会对深度缓冲区和模板缓冲区进行写入，甚至在它优化掉对颜色缓冲区的写入时也是如此。

对于片段着色器来说，通知 OpenGL 丢弃一起进行渲染的像素是可能的。这项工作可以通过使用 discard 关键字来完成。如果片段着色器执行了 discard 关键字，那么着色器的结果将会被丢弃，并且不会对任何输出缓冲区进行写入，包括深度、模板或者颜色缓冲区绑定。程序清单 11.29 中的例子展示了如何在片段着色器中实现 alpha 测试，这样就会允许在输出几何图形上裁剪出空洞。

程序清单 11.29 简单的 alpha 测试片段着色器

```
#version 330
precision highp float;
uniform sampler2D my_texture;

in Fragment
{
    vec2 texture_coord;
} fragment;

out vec4 color_out;

void main(void)
{
    vec4 color = texture(my_texture, fragment.texture_coord);
    if (color.a < 0.1)
        discard;
    color_out = color;
}
```

在本例，如果存储在一个纹理中的 alpha 值小于某个阈值（在本例为 0.1），就会执行 discard 关键字，那么片段着色器的结果就不会被写入到任何绑定的缓冲区中。如果 alpha 值大于或等于这个阈值，那么着色器将继续进行，而结果得到的颜色也将被写入。不要忘记，alpha 混合是在片段着色器之后由一个附加阶段执行的。着色器所要做的全部工作就是将颜色（包括它的 alpha 分量）写入到输出变量，而固定功能的混合阶段则负责将它混合到帧缓冲区时所需的计算。

片段着色器可以为了它所选择的任何原因执行 discard 关键字。除了将它建立在纹理的正确性基础之上，它还可以通过对输入变量进行分析生成条件。如果将一个纹理作为决定因子使用，那么结果得到图像的细节层次就会取决于纹理的分辨率。如果这个决定是经过分析得到的，那么结果得到图像的细节则只取决于片段着色器的精度。例如，如果片段确定在集合内，那么我们可以修改茹利亚渲染器，执行 discard 关键字，正如程序清单 11.30 中的代码片段所示。

程序清单 11.30 修改茹利亚渲染程序，应用 discard 关键字

```
if (iterations == max_iterations)
    discard;

output_color = texture(tex_gradient,
                      float(iterations) / float(max_iterations));
```

现在，当我们对分形进行渲染时，在它上面像素落在集合内的位置就会出现一个空洞，只有当像素离开集合时才会被着色。当放大茹利亚集（或者增加显示器的分辨率）时，空洞的边缘将会呈现更多的细节——这是分形的典型特性。我们可以在某些前面渲染过的几何图形上绘制茹利亚集，那么这些图形就可以透过这个空洞被观察到了。

### 11.3.5 逐片段控制深度

除了为片段着色器定义的输出变量之外，也可以在特殊的 `gl_FragDepth` 内建变量中写入和更新深度值。如果片段着色器不写入这个变量，由 OpenGL 生成的插值深度将作为片段的深度值来使用。片段着色器可以为 `gl_FragDepth` 计算一个全新的值，或者也可以使用 `gl_FragCoord.z` 值来控制深度值。接下来，OpenGL 既可以用这个新值进行深度测试，也可以将它作为写入到深度缓冲区的值。

我们可以使用这种功能对深度缓冲区中的值进行轻微的扰动，并创建物理上波动起伏的表面。当附加的几何图形进行渲染并随后进行深度测试时，它将针对这些扰动值进行测试。不过，在使用这些特性时要多加注意。如果片段着色器没有被写入 `gl_FragDepth` 变量，那么在片段着色器运行之前，OpenGL 就会知道深度的最终值是多少。

这样，在运行片段着色器之前执行深度测试就成为了大多数现代图形硬件所做的一项非常普遍的优化。如果片段的深度测试失败，那么 OpenGL 将不会运行片段着色器。但是，如果着色器确实写入了 `gl_FragDepth`，那么 OpenGL 就不知道这个片段是不是会通过深度测试了，或者说在片段着色器运行之前不会知道。这样，它就必须总是运行片段着色器，并在着色器运行后执行深度测试。这样会导致性能大大降低。如果这对于我们试图实现的算法的正确功能来说是绝对必要的话，那么为了获得最佳性能，我们也只在片段着色器中写入 `gl_FragDepth`。

## 11.4 更高级的着色器函数

现在我们已经学习了一些在 OpenGL 中可以实现的有趣的东西。有一些更加高级的特性需要在各着色器阶段之间进行协作，或者不适合任何特定的着色器阶段。在这里我们将介绍其中的一部分。

### 11.4.1 插值和存储限定符

在前面的几章内容，我们已经接触了存储限定符，已经了解了如何使用 flat 存储限定符来关闭插值，并要求 OpenGL 在图元上执行平面着色。

还有两种存储限定符可以对插值进行控制，我们可以使用它们进行高级渲染。它们就是 centroid 限定符和 noperspective 限定符，现在就来快速地了解一下。

#### 质心采样

centroid 存储限定符控制 OpenGL 将片段着色器的输入插值到一个像素的什么地方。它只在渲染到多重采样表面的情况下应用，多重采样前面已经介绍过了。我们可以像指定其他任何存储限定符那样指定 centroid 存储限定符。要创建一个带有 centroid 存储限定符的变量，首先要在顶点着色器（或者几何着色器）中声明带有 centroid 关键字的输出变量。

```
centroid out vec2 tex_coord;
```

然后，在片段着色器中声明同一个带有 centroid 关键字的输入变量。

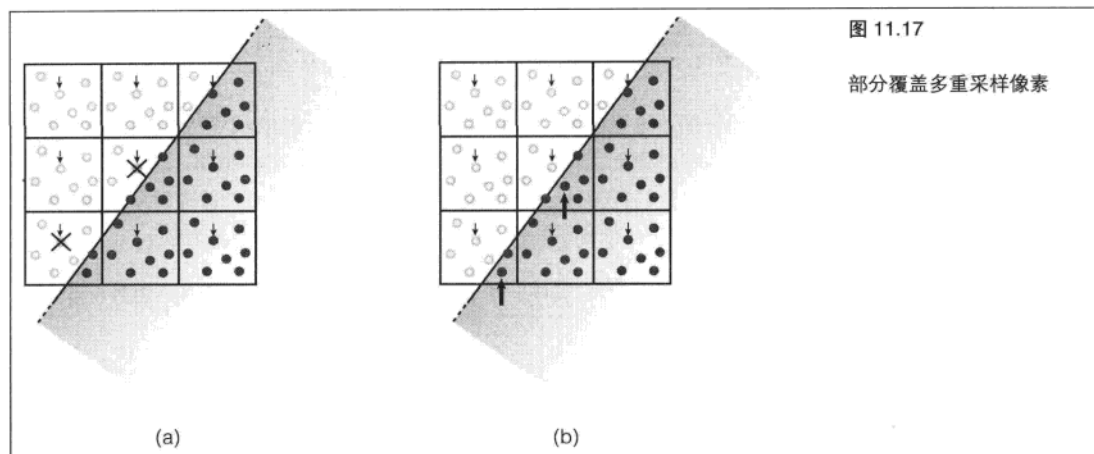
```
centroid in vec2 tex_coord;
```

现在 tex\_coord 变量被定义为使用 centroid 存储限定符。如果我们有一个单采样绘制缓冲区，并不会有任何不同，而到达片段着色器的变量将会被插值到像素中心。当我们渲染到一个多重采样绘制缓冲区时，质心采样就变得有用了。根据 OpenGL 规范，当质心采样没有被指定时（默认情况下是这样），片段着色器变量将会被插值到像素中的任何位置。

当在一个大三角形的中间时，这一点其实并不重要。当我们对恰好位于三角形边缘的像素（也就是说，三角形的一个边穿过这个像素）进行着色时，就变得非常重要了。图 11.17 所示展示了 OpenGL 如何从一个三角形中进行采样的示例。

让我们来看一看图 11.17(a)。它显示了一个穿过几个像素的三角形边。图中的实心点代表被三角形覆盖的样本，空心点则代表没有被三角形覆盖的样本。OpenGL 选择了将变量插值到距离像素中心最近的样本，这些样本通过一个箭头指示。

对于左上角的像素来说，这样很好——它们完全没有被覆盖，而片段着色器也不会在这些像素上运行。



类似地，位于右下角的像素则完全被覆盖。片段着色器将会运行，在哪个样本上运行并不太重要。但是，在沿着三角形边缘的像素上，就会出现这个问题。因为 OpenGL 已经选择距离像素中心最近的样本作为它的插值点，我们的变量实际上可能会被插值到位于三角形外的一个点上！这些样本是由一个 X 标记的。比如说，想象一下如果我们使用这个变量对一个纹理进行采样，将会发生什么。如果这个纹理进行对齐，使它的边缘与三角形的边缘相匹配，那么纹理坐标将会落到纹理之外。在最好的情况下，我们将获得一个稍微有点不太正确的图像。在最坏的情况下，将会产生明显的痕迹。

如果我们声明带有 centroid 存储限定符的变量，那么 OpenGL 规范对此也有说明：“这些值必须被插值到一个同时位于像素中和被渲染的图元中的点上，或者被插值到像素的一个落到图元中的样本上”。

这就意味着 OpenGL 为每个像素选择了一个样本，这个样本一定是在一个所有变量将要插值到其上的三角形中。我们可以放心地在片段着色器中为了任何目的使用这个变量，并且知道它们是合法的，而且不会指向三角形之外的点。

现在让我们来看一看图 11.17(b)。对于完全覆盖的像素，OpenGL 仍然选择将变量插值到距离像素中心最近的样本。但是，对于那些部分覆盖的像素来说，OpenGL 则会选择位于三角形内的其他样本（通过大一些的箭头指出）。这就意味着提供给片段着色器的输入是合法的，并且指向位于三角形内部的点。我们可以使用它们来在一个纹理中进行采样，或者在一个其结果只定义在特定范围内，并且已知可以得到有意义的结果的函数中使用它们。

你可能会想，如果使用 centroid 存储限定符就能保证在片段着色器中得到正确的结果，而不使用它可能就意味着变量将会落在图元之外，那么为什么不一直开启质心采样呢？其实，使用质心采样还存在一些不利的因素。其中最重要的就是 OpenGL 能够向片段着色器提供输入的过渡（或者微分）。对于具体的实现来说可能情况有所不同，但多数都使用离散微分，在相邻像素的相同变量之间获取差值。这种方式在变量被插值到每个像素的同一位置时非常适合。在这种情况下，选择哪个采样位置并不重要，样本总是会恰好间隔一个像素。但是，在为一个输入开启质心采样时，相邻像素的值实际上可能会在这些像素中被插值到不同的位置。这就意味着这些样本的间隔将不是恰好一个像素，而提供给片段着色器的离散微分可能会是不准确的。

如果在片段着色器中需要精确地过渡，那么最好不要使用质心采样。

## 使用质心采样来执行边缘检测

质心采样的一种有趣应用就是硬件加速边缘检测。我们刚刚学习了使用 centroid 存储限定符来确保我们的变量被插值到一个确定落在进行渲染的图元之内的点上。为了实现这一点, OpenGL 选择了一个它确定会落在三角形之内的样本, 来对这些变量进行评估, 而 OpenGL 在这个像素被完全覆盖时所选择的样本和在没有使用 centroid 存储限定符时所选择的样本可能会不同。我们可以充分利用这些知识。

为了从中提取边缘信息, 我们声明两个变量, 其中一个带有 centroid 存储限定符, 而另外一个则没有, 并且在顶点着色器中为它们指定同一个值。

只要这个值对于每个顶点来说是不同的, 那么它的具体值就不重要。经过变换的顶点位置的  $x$  和  $y$  分量可能是一个很好的选择, 因为我们知道对于任何实际可见的三角形来说, 每个顶点的值都是不同的。

```
out vec2 maybe_outside;
```

为我们提供了非质心变量, 可以插值到三角形之外的一个点上, 而

```
centroid out vec2 certainly_inside;
```

则为我们提供了已知在三角形之内的质心采样变量。

在这个片段着色器中, 我们可以将两个变量的值进行比较。如果这个像素完全被三角形遮盖, 那么 OpenGL 就会为这两个变量都使用同一个值。但是, 如果像素只是部分地被三角形遮盖, 那么 OpenGL 会为 maybe\_outside 采样使用它的一般选择, 并为 certainly\_inside 挑选一个确定会落在三角形之内的样本。这样选择的样本就可能与为 maybe\_outside 选择的样本不同, 并且这就意味着这两个变量的值可能会不同。现在我们可以对这两个值进行比较来确定是否在图元的边缘上。

```
bool may_be_on_edge = any(maybe_outside != certainly_inside);
```

这种方法并不是十分简单。即使一个像素是在三角形的边缘上, 它也可能覆盖 OpenGL 初始选择的样本, 这样对于 maybe\_outside 和 certainly\_inside 来说仍然会得到同样的值。然而, 这样产生的边缘像素最多。

为了使用这些信息, 我们可以将这个值写入一个绑定到帧缓冲区的纹理, 然后使用这个纹理进行进一步的处理。另一种选择是只绘制到模板缓冲区, 将模板参考值设为 1, 禁用模板测试, 然后将模板操作设为 GL\_REPLACE。当遇到边缘时, 让片段着色器继续运行。当遇到不在边缘上的像素时, 在着色器中使用 discard 关键字防止像素被写入到模板缓冲区。这样做的结果是, 模板缓冲区在场景中的边缘处是一些 1, 而在没有边缘的地方都是 0。接下来, 我们可以使用一个大开销的片段着色器渲染一个全屏四边形, 这个片段着色器只应用在通过开启模板测试, 将模板函数设置为 GL\_EQUAL, 并保留参考值为 1 来表示几何图形边缘的像素上。这种着色器能够实现前面描述过的图像处理操作。例如, 使用一个卷积操作 (如本章前面部分所演示的) 应用高斯模糊, 可以将场景中多边形的边缘变得平滑, 允许应用程序执行抗锯齿。

## 无透视校正的插值

就像我们已经学过的一样, OpenGL 使用变量的值在图元 (例如三角形) 的表面进行插值, 并为片段着色器的每一次调用提供一个新值。在默认条件下, 插值在三角形的空间中是线性的。这就意味着, 如

果直着面对这个三角形看去，这些变量在它表面上的步长是相等的。但是，OpenGL 会在逐像素步进时在屏幕空间进行插值。直接面向三角形观察的情况非常少见，这样透视缩短就意味着从像素到像素的每次步进都不是常量——也就是说，它们在屏幕空间中不是线性的。OpenGL 使用透视校正（perspective-correct）插值对此进行校正。

要做到这一点，它会使用在屏幕空间中呈线性的值进行插值，并使用它们来获取在每个像素上变量的实际值。

让我们考虑一个纹理坐标  $uv$ ，它将被插值到一个三角形上。 $u$  和  $v$  在屏幕空间中都不是线性的。但是（由于某些超出这部分知识范围的数学因素）， $u/w$  和  $v/w$  在屏幕空间是线性的，正如  $1/w$ （ $w$  是片段坐标的第 4 个分量）一样。这样，OpenGL 实际上是在每个像素上

$$\frac{u}{w}, \frac{v}{w}, \text{ and } \frac{1}{w}$$

进行插值的，它用  $1/w$  来找出  $w$ ，然后再用  $u/w$  和  $v/w$  分别乘以  $w$  来找出  $u$  和  $v$ 。这样就为片段着色器的每个实例的插值提供了透视校正。

一般来说，这正是我们想要的。但是，有时候这也可能不是我们所希望的。

如果我们实际上希望无论图元方向如何都在屏幕空间中进行插值，可以使用 `noperspective` 存储限定符。

例如，在顶点着色器中使用：

```
noperspective out vec2 texcoord;
```

而在片段着色器中使用：

```
noperspective in vec2 texcoord;
```

使用透视校正和屏幕空间线性（非透视）渲染的结果分别如图 11.18 (a) 和图 11.18 (b) 所示。

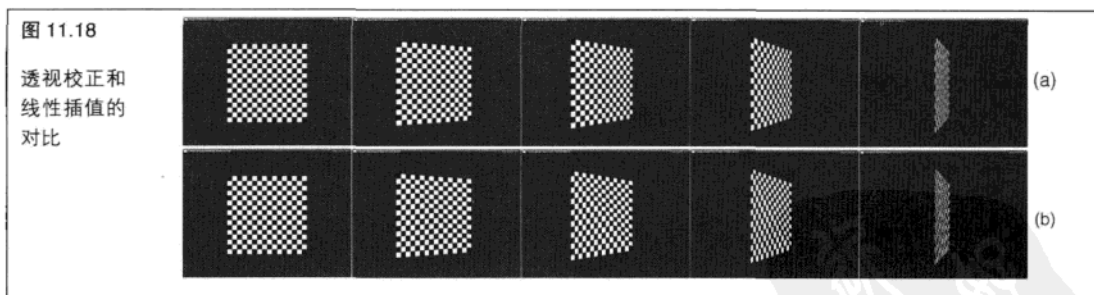


图 11.18 (a) 所示展示了应用到一对三角形上的透视校正插值，这对三角形相对于观察者的角度是变化的。同时，图 11.18 (b) 所示则展示了 `noperspective` 存储限定符是如何影响纹理坐标的插值的。随着这对三角形的移动，相对于观察者的角度也越来越斜，纹理也变得越来越斜。

### 11.4.2 高级内建函数

GL 着色语言（GLSL）和 C 语言非常相近。诸如 `for`、`while` 和 `do` 这样的循环结构实际上和 C 语言

中的定义是相同的，而像 `if-else` 声明、`switch` 声明和 `?` 操作符这样的条件结构也是如此。

C 标准库中的许多标准数学函数在 GLSL 中也可以使用，其中包括类似 `sin`、`cos` 和 `tan` 这样的三角函数，类似 `abs`、`floor` 和 `ceil` 这样的数学函数，类似 `pow`、`exp` 和 `log` 这样的指数函数，以及类似 `+`、`-` 和 `==` 这样的内建操作符。如果读者使用 C 语言或其他类似 C 语言的经验，那么就会对这些函数非常熟悉。但是，既然 GLSL 是为了对图元进行操作而设计的，就还有一些内建函数用来对向量和矩阵类型进行操作，而它们常常能够在硬件中应用得很理想。在大多数情况下，这些函数可以用于任何着色器阶段。

除了这些标准数学函数之外，GLSL 还提供一些实用函数，例如 `clamp`（将一个值截取到两个值之间的范围内）、`mix`（执行线性插值），以及 `step` 和 `smoothstep`（根据两个值的输入在它们之间创建过渡）。

同样，浮点值和整数值之间的位强制类型转换也可以使用 `intBitsToFloat` 和 `floatBitsToInt` 函数和它们的无符号变体执行。

GLSL 向量函数包括 `dot` 和 `cross`，就像名字提示的那样，它们分别执行一个点乘和一个叉乘操作。另外，函数 `outerProduct` 可以用来提取两个向量的外积。`distance` 和 `length` 函数分别计算两点之间的距离和一个向量的长度，可以使用 `normalize` 函数对一个向量进行标准化。还有一些更加复杂的情况，`reflect` 和 `refract` 函数提供内建的、潜在优化的反射和折射方程实现，它们通常在光照和路径追踪算法中使用。

更高级的矩阵相关函数包括 `transpose`、`determinant` 和 `inverse`，分别进行矩阵转置、求行列式和求逆矩阵操作。此外，`matrixCompMult` 函数在两个矩阵之间执行一个逐分量相乘的操作。这个函数是十分必要的，因为默认乘法操作符（`*`）应用在矩阵中执行的是矩阵与矩阵的乘法。

因为关系运算符（例如 `>`、`!=` 和 `<=`）是设计用来生成单个标量布尔值结果的，所以这些比较运算符的向量版本就作为内建函数来提供，返回布尔向量。它们包括 `lessThan`、`notEqual` 和 `lessThanEqual` 等。为了在结果得到的布尔向量上执行集合操作，我们还可以使用 `any`（在任一变量为真时返回真）和 `all`（在所有变量为真时返回真）函数。

布尔向量不能作为一个 `if` 语句的表达式来提供，所以我们需要使用 `any` 或 `all` 将向量转化为标量才能在条件语句中使用。

## 11.5 统一缓冲区对象

到目前为止，我们所写的着色器已经变得很复杂了。其中有一些还需要很多常量数据，而将这些数据传递到着色器的方法就是使用统一值。如果在一个应用程序中有很多着色器，那么就需要为这些着色器中的每一个设置统一值，这就意味着要很多次调用不同的 `glUniform` 函数。我们还需要跟踪统一值的变化。有些变化会改变所有对象，有些变化则每次改变一个帧，而另一些则只需要为整个应用程序初始化一次。这就是说，我们要么需要为位于应用程序不同位置的不同的统一集合进行更新，使它们维护起来更加复杂，要么以性能为代价总是同时更新所有的统一值。

为了减轻所有 `glUniform` 调用的开销，使更新一组大量的统一值更加简单，同时也为了更容易地在

不同程序之间共享一组统一值，OpenGL 允许将一组统一值组合到一个统一块中，并在一个缓冲区对象中存储整个块。缓冲区对象与我们以前使用过的其他缓冲区对象类似。我们可以通过改变缓冲区绑定，或者覆盖一个绑定缓冲区的内容来快速地设置整个一组统一值。我们还可以在改变程序时保持缓冲区绑定，而新的程序将看到当前的统一值设置。这项功能叫做统一缓冲区对象（uniform buffer object），缩写为 UBO。实际上，我们到目前为止使用的统一值都在默认块中。一个着色器中全局声明的任何统一值最终都会在默认统一块中。我们不能在一个统一缓冲区对象中保存默认块，需要创建一个或多个统一块并对其命名。

要声明一组存储在缓冲区对象中的统一值，需要在着色器中使用一个命名的统一块。这看起来很像一个接口块，但是它使用的是 `uniform` 关键字。程序清单 11.31 展示了着色器中相应的代码。

程序清单 11.31 示例统一块声明

```
uniform TransformBlock
{
    float scale; // 应用到各处全局缩放
    vec3 translation; // 在 X、Y 和 Z 方向平移
    float rotation[3]; // 围绕 X、Y 和 Z 轴旋转
    mat4 projection_matrix; // 在进行缩放和旋转之后应用的广义投影矩阵
} transform;
```

这段代码声明了一个名为 `TransformBlock` 的统一块，还声明了一个名为 `transform` 块的实例。在着色器内部，我们可以使用它的实例名 `transform` 在块中引用这个成员。但是，为了设置将要用来自返回块的缓冲区对象中的数据，我们需要知道块的一个成员的位置，并且为此我们需要块的名称 `TransformBlock`。如果我们需要块的多个实例，每个实例都有自己的缓冲区，那么可以将 `transform` 设为一个数组。

这个块的成员在每个块中将会有相同的位置，但是现在我们在着色器中将能够引用这个块的几个实例了。当我们想要用数据填充一个块时，在这个块中查询成员的位置就非常重要了，这一点在接下来的内容中将进行解释。

## 11.5.1 建立统一块

在着色器中通过指定统一块进行访问的数据可以在缓冲区对象中进行存储。一般说来，使用 `glBufferData` 或 `glMapBuffer` 这样的函数对缓冲区对象进行数据填充是应用程序的工作。那么，现在的问题就是，缓冲区中的数据应该是什么样的？这里实际上有两种可能性，无论选择哪种都是进行权衡的结果。第一种情况是让 OpenGL 决定将数据放在哪里。这样可以生成效率最高的缓冲区，但是这就意味着应用程序需要确定将数据放在哪里，以便 OpenGL 能够读取它们。这样做可能会非常不方便，所以第二种方法中数据采用一种标准的、受到认可的布局。这就意味着应用程序只需将数据复制到缓冲区并在块中为成员分配位置就可以了——我们甚至可以将数据提前存储在磁盘上，并直接将它们读取到一个使用 `glMapBuffer` 进行映射的缓冲区就可以了。既然这种布局是标准的，适合所有图形硬件和驱动，那么它对于其中的任何一种都不是最优的。这是因为在块的各种成员之间保留了一些空闲的空间，使缓冲区比需要的大。我们可能要为换取这种方便而付出一些性能上的代价。我们在这里对这两种方法都进行了介绍，读者需要决定哪一种方法最适合应用程序。

数据存储的统一缓冲区中的第一种布局是共享布局（shared layout）。如果没有明确地对 OpenGL

提出其他要求的话,那么这就是默认的布局了。在共享布局的情况下,缓冲区中的数据将由 OpenGL 决定如何布置对于运行时性能和着色器访问来说才是最佳的。这样经常可以使着色器获得更高的性能,但需要应用程序做更多的工作。这种布局被称为共享布局的原因就是,当 OpenGL 在缓冲区中对数据进行排列时,这种排列在共用相同的统一块声明的多个程序和着色器之间是相同的。例如,着色器编译器仍然会为统一块的成员保留空间,即使在着色器并不使用它们时也是如此。这样就允许我们在任何程序上使用同一个缓冲区对象了。要使用共享布局,应用程序必须在缓冲区对象中确定统一块成员的位置。

一个统一块的每个成员都有一个索引,用于对它进行引用,来查询它的大小和在块中的位置。为了获取一个统一块中一个成员的索引,我们可以调用:

```
void glGetUniformIndices(GLuint program, GLsizei uniformCount, const GLchar **  
uniformNames, GLuint * uniformIndices);
```

这看起来似乎非常复杂,但其实却并非如此。这个函数允许我们通过 OpenGL 中进行一次调用来获取一组大量统一值的索引——甚至可能是一个程序中所有的统一值,即使它们是不同的块中的成员。它接受对我们想要进行索引的统一值的数量的一个计数值 (uniformCount), 一个统一值名称的数组 (uniformNames), 并将它们的索引放在一个数组 (uniformIndices) 中。程序清单 11.32 包含一个示例,告诉我们如何检索前面声明的 TransformBlock 中成员的索引。

程序清单 11.32 对统一块成员的索引进行检索

```
const GLchar * uniformNames[4] =  
{  
    "TransformBlock.scale",  
    "TransformBlock.translation",  
    "TransformBlock.rotation",  
    "TransformBlock.projection_matrix"  
};  
GLuint uniformIndices[4];  
  
glGetUniformIndices(program, 4, uniformNames, uniformIndices);
```

在这个代码运行之后,就得到了 uniformIndices 数组中统一块的 3 个成员索引。现在有了索引,就可以使用它们来查找缓冲区中块成员的位置了。为了完成这项工作,我们可以调用:

```
void glGetActiveUniformsiv(GLuint program, GLsizei uniformCount, const GLuint *  
uniformIndices, GLenum pname, GLint * params);
```

这个函数能够提供很多关于指定统一块成员的信息。

我们感兴趣的信息是缓冲区中成员的偏置量、数组步长 (为了进行 TransformBlock.rotation) 和矩阵步长 (为了进行 TransformBlock.projection\_matrix)。这些值告诉我们要将数据放在缓冲区的什么地方,以便着色器能够访问它们。我们可以将 pname 分别设置为 GL\_UNIFORM\_OFFSET、GL\_UNIFORM\_ARRAY\_STRIDE 和 GL\_UNIFORM\_MATRIX\_STRIDE 来从 OpenGL 中查询这些值。程序清单 11.33 展示了相应的代码。

程序清单 11.33 对统一块成员的信息进行检索

```
GLint uniformOffsets[4];  
GLint arrayStrides[4];  
GLint matrixStrides[4];  
glGetActiveUniformsiv(program, 4, uniformIndices,  
    GL_UNIFORM_OFFSET, uniformOffsets);
```

```
glGetActiveUniformsiv(program, 4, uniformIndices,
                      GL_UNIFORM_ARRAY_STRIDE, arrayStrides);
glGetActiveUniformsiv(program, 4, uniformIndices,
                      GL_UNIFORM_MATRIX_STRIDE, matrixStrides);
```

程序清单 11.33 中的代码执行之后, uniformOffsets 中就包含了 TransformBlock 块中成员的偏置量, arrayStrides 中包含了数组成员(目前只有 rotation)的步长, 而 matrixStrides 中则包含了矩阵成员(只有 projection\_matrix)的步长。

我们可以查询的其他关于统一块成员的信息包括统一值的数据类型, 它在内存中消耗的空间大小(以字节为单位), 以及与这个块中数组和矩阵相关的布局信息。我们需要其中一些信息来对一个类型更加复杂的缓冲区对象进行初始化, 即使是我们自己编写的这些着色器并且已经知道了这些成员的大小和类型。表 11.5 列出了 pname 能够接受的值和对应的返回信息。

表 11.5 通过查询统一值参数

Pname 的值	返回的信息
GL_UNIFORM_TYPE	统一值的数据类型
GL_UNIFORM_SIZE	数组的大小, 以 GL_UNIFORM_TYPE 指定的单位表示 如果统一值不是数组, 那么这个值总为 1
GL_UNIFORM_NAME_LENGTH	统一值名称的长度, 以字符为单位
GL_UNIFORM_BLOCK_INDEX	统一值作为其中一个成员的块的索引
GL_UNIFORM_OFFSET	块(或者更加精确地说, 返回块的缓冲区)中统一值的偏置
GL_UNIFORM_ARRAY_STRIDE	一个数组中连续元素之间间隔的字节数 如果统一值不是数组, 那么这个值总为 0
GL_UNIFORM_MATRIX_STRIDE	列优先矩阵中的每一列或行优先矩阵中的每一行的第一个元素之间间隔的字节数。如果统一值不是数组, 那么这个值总为 0
GL_UNIFORM_IS_ROW_MAJOR	输出数组中每个元素的值, 如果统一值是一个行优先矩阵则为 1, 或者如果统一值是一个列优先矩阵则为 0

如果我们感兴趣的统一值类型是诸如 int、float、bool 这样的简单类型, 或者甚至是这些类型的向量(vec4 等), 那么我们所需要的就只是它的偏移量了。一旦知道了缓冲区中的统一值位置, 就可以将偏移值传递到 glBufferSubData 在正确的位置加载数据, 或者也可以在代码中直接使用偏移量在内存中装配缓冲区了。在这里, 我们对后一种选择进行了演示, 因为它强化了统一值存储在内存中的思想, 就像纹理或顶点信息一样。这也意味着更少的 OpenGL 调用, 这样做有时会带来更高的性能。对于这些例子来说, 在应用程序的内存中装配数据, 然后再使用 glBufferData 将它加载到一个缓冲区中。我们也可以选择使用 glMapBuffer 来获取一个指向缓冲区内存的指针, 并将数据直接装配到这个缓冲区中。

让我们从设置 TransformBlock 块中最简单的统一值——scale 开始吧。这个统一值是一个单独的 float(浮点类型), 它存储在 uniformIndices 数组的第一个元素中。程序清单 11.34 展示了如何设置这个浮点类型的值。

程序清单 11.34 在一个统一块中设置单个 float

```
// 为我们的缓冲区分配一些内存(后面不要忘记释放它们)
unsigned char * buffer = (unsigned char *)malloc(4096);

// 我们知道是在块中的位, 所以我们可以相应地对我们的缓冲区指针进行偏置来进行存储
*({float *}(buffer + uniformOffsets[0])) = 3.0f;
```

接下来我们就可以为 TransformBlock.translation 进行初始化数据了。这是一个 vec3，也就是说它包含 3 个浮点值，它们紧密地包装在内存中。为了对其进行更新，需要找出这个向量的第一个元素的位置，并从那个位置开始将 3 个连续的浮点值存储到内存中。程序清单 11.35 显示了这些代码。

程序清单 11.35 对统一块成员的索引进行检索

```
// 将 3 个连续的 GLfloat 值放入内存来更新 vec3.
*((float *) (buffer + uniformOffsets[1])) = 1.0f;
*((float *) (buffer + uniformOffsets[1] + sizeof(GLfloat))) = 2.0f;
*((float *) (buffer + uniformOffsets[1] + 2 * sizeof(GLfloat))) = 3.0f;
```

现在，让我们来对付 rotation 数组。在这里我们也可以使用一个 vec3，但是为了达到这个示例的目的，我们使用一个 3 个元素的数组来演示 GL\_UNIFORM\_ARRAY\_STRIDE 参数的应用。在使用共享布局时，数组是作为一个元素序列单独定义的，这个序列的步长以字节为单位，由具体的实现来定义。这就意味着我们必须将元素放到由 GL\_UNIFORM\_OFFSET 和 GL\_UNIFORM\_ARRAY\_STRIDE 定义的缓冲区中，正如程序清单 11.36 中的代码片段所示。

程序清单 11.36 为统一块中的一个数组指定数据

```
// TransformBlock.rotations[0]是在缓冲区的 uniformOffsets[1]位
// 数组的每个元素都在多个 arrayStrides[1]位之后
const GLfloat rotations[] = { 30.0f, 40.0f, 60.0f };
unsigned int offset = uniformOffsets[2];

for (int n = 0; n < 3; n++) {
    *((float *) (buffer + offset)) = rotations[n];
    offset += arrayStrides[2];
}
```

最后，我们为 TransformBlock.projection\_matrix 设置这些数据。统一块中矩阵的行为与向量数组非常相似。对于列优先的矩阵（默认情况下如此）来说，矩阵的每一列都会被当作一个向量来对待，它的长度就是矩阵的高度。类似地，行优先的矩阵也会被当作一个向量数组来对待，其中每一行就是这个数组中的一个元素。就像普通的数组一样，矩阵中每一列（或每一行）的初始偏置是由一个由实现定义的量来决定的。

它可以通过 glGetActiveUniformsiv 的 GL\_UNIFORM\_MATRIX\_STRIDE 参数来查询。矩阵的每一列都可以使用与用来初始化 vec3 TransformBlock.translation 的代码相似的代码进行初始化。程序清单 11.37 列出了这些设置代码。

程序清单 11.37 在一个统一块中设置一个矩阵

```
// TransformBlock.projection_matrix 的第一列位于缓冲区中的 uniformOffsets[2]位
// 这些列以 matrixStride[2] bytes 的步长放置，实际上他们都是一些 vec4
// 这就是源矩阵——还记得吧，它是列优先的，所以
const GLfloat matrix[] =
{
    1.0f, 2.0f, 3.0f, 4.0f,
    9.0f, 8.0f, 7.0f, 6.0f,
    2.0f, 4.0f, 6.0f, 8.0f,
    1.0f, 3.0f, 5.0f, 7.0f
};

for (int i = 0; i < 4; i++)
{
    GLuint offset = uniformOffsets[2] + matrixStride[2] * i;
    for (j = 0; j < 4; j++) {
        *((float *) (buffer + offset)) = matrix[i * 4 + j];
    }
}
```

```

        offset += sizeof(GLfloat);
    }
}

```

这种查询偏置和步长的方法适用于任何布局。对于共享布局来说，这是唯一的选择。但是，这种方法有些不太方便，并且就像我们能够看到的，需要编写很多代码将数据以正确的方式在缓冲区进行布局。作为一种选择，我们也可以使用 standard（标准）布局。标准布局允许我们根据 OpenGL 提供的一组为各种数据类型指定大小和排列的规则决定将数据放在缓冲区的什么位置。这些规则在所有 OpenGL 实现中都是通用的，这样我们不需要进行任何查询就可以使用它们了（不过我们还是应该查询偏置和步长，以确保结果是正确的）。

为了告诉 OpenGL 我们想要使用标准布局，需要声明带有布局限定符的统一块。TransformBlock 的一个带有布局限定符的重复声明 std140 如程序清单 11.38 所示。

程序清单 11.38 对统一块成员的索引进行检索

```

layout(std140) uniform TransformBlock
{
    float scale;           // 应用到各处的全局缩放
    vec3 translation;      // 在 X、Y 和 Z 方向平移
    float rotation[3];     // 围绕 X、Y 和 Z 轴旋转
    mat4 projection_matrix; // 在进行缩放和旋转之后应用的广义投影矩阵
} transform;

```

一旦一个统一块被声明为使用标准布局，或者说 std140 布局，这个块的每个成员就会在缓冲区以遵循一组规则确定的偏置占用预先定义好的空间大小。这些规则概述如下。

任何在缓冲区中占用 N 个字节的类型都会在这个缓冲区中的一个 N 个字节的分界处开始。这就意味着诸如 int、float 和 bool 这样的标准 GLSL 类型（它们的大小都定义为 32 位或者 4 个字节）在 4 个字节的倍数的位置开始。这些类型长度为 2 的向量总是在一个 2N 字节分界处开始。举例来说，这就意味着一个在内存中长度为 8 字节的 vec2，总是在一个 8 字节分界处开始。3 元素和 4 元素向量总是在一个 4N 字节的分界处开始，所以 vec3 和 vec4 类型会在 16 字节分界处开始。一个标量数组或者向量类型数组（例如 ints 或 vec3s）总是在以同样规则定义的分界处开始，但是会进行上舍入与 vec4 看齐。特别是，这意味着除了 vec4（以及 Nx4 矩阵）之外的任何数组都不会进行紧密的包装，但是这样就会在每个元素之间出现一个缺口。矩阵本质上是被当作向量的数组来对待的，而矩阵的数组则被当作很长的向量数组对待。最后，结构体和结构体的数组都需要进行额外的包装：整个结构体从它最大的成员所要求的分界点开始，并上舍入到 vec4 的大小。

我们必须特别要注意 std140 布局和包装规则之间的区别，而随后经常紧接着的就是选择的编译器。特别是，统一块中的一个数组没有必要进行紧密的包装。举例来说，这就意味着我们无法在一个统一块中创建一个 floats 的数组，并简单地从一个 C 语言的 floats 数组中将数据复制到其中，因为来自 C 数组的数据将进行包装，而统一块中的数据则不会进行包装。

这些听起来都很复杂，但是逻辑性很好，经过很好的定义，并且允许大范围的图形硬件高效地实现统一缓冲区对象。回到 TransformBlock 示例，可以使用这些规则来确定缓冲区中块的成员的偏置。程序清单 11.39 展示了一个统一块及其成员偏置一起进行声明的示例。

程序清单 11.39 带有偏置的统一块示例

```

layout(std140) uniform TransformBlock
{

```

// 成员	基本排列	偏置	排列偏置
float scale;	// 4	0	0
vec3 translation;	// 16	4	16
float rotation[3];	// 16	28	32 {rotation[0]}
	//		48 {rotation[1]}
	//		64 {rotation[2]}
mat4 projection_matrix;	// 16	80	80 {column 0}
	//		96 {column 1}
	//		112 {column 2}
	//		128 {column 3}

```

} transform;

```

在 ARB\_uniform\_buffer\_object 扩展规范中有一个关于各种类型排列的完整示例。

现在我们已经对缓冲区进行了填充，可以将它绑定到程序中的一个统一块了。在进行这些操作之前，还需要对这个统一块的索引进行检索。程序中每一个统一块都有一个由编译器分配的索引。有一个固定的统一块最大值，可以用于单个程序，还有一个可以在任何给定的着色器阶段使用的最大值。我们可以通过调用带有 GL\_MAX\_UNIFORM\_BUFFERS 参数的 glGetIntegerv 来查找这些限制（每个程序的总限制），用 GL\_MAX\_VERTEX\_UNIFORM\_BUFFERS、GL\_MAX\_GEOMETRY\_UNIFORM\_BUFFERS 或 GL\_MAX\_FRAGMENT\_UNIFORM\_BUFFERS 分别查找顶点着色器、几何着色器和片段着色器的限制。为了获取程序中一个统一块的索引，我们可以调用：

```
GLuint glGetUniformBlockIndex(GLuint program, const GLchar * uniformBlockName);
```

它将返回指定统一块的索引。在示例统一块声明中，uniformBlockName 为“TransformBlock”。这里有一组缓冲区绑定点，可以将一个缓冲区绑定到它们，来为统一块提供数据。将一个缓冲区绑定到一个统一块上，实际上是一个分为两步的处理过程。统一块被分配了绑定点，然后缓冲区可以被绑定到这些绑定点上，将缓冲区和统一块匹配起来。这样，不同的程序可以在不改变缓冲区绑定的情况下来回切换，而固定的统一值集合可以自动地被新的程序访问。将它与默认块中统一值的值进行比较，这些值就是每个程序相关的状态。即使两个程序包含同样名称的统一值，它们的值也必须分别为每个程序进行设置，并且在活动的程序改变时也将随之改变。

要将一个绑定点分配给一个统一缓冲区，可以调用

```
void glUniformBlockBinding(GLuint program, GLuint uniformBlockIndex, GLuint uniformBlockBinding);
```

其中 program 是我们改变的统一块所在的程序。uniformBlockIndex 则是我们将一个绑定点分配到统一块的索引，刚才已经通过调用 glGetUniformBlockIndex 来检索它了。uniformBlockBinding 是统一块绑定点的索引。一个 OpenGL 的实现提供一个固定的绑定点最大数量，我们可以通过调用以 GL\_MAX\_UNIFORM\_BUFFER\_BINDINGS 为参数的 glGetIntegerv 来查询这个限制。

一旦将绑定点分配给程序中的统一块，就可以将缓冲区绑定到这些相同的绑定点来使缓冲区中的数据出现在这些统一块中。我们通过调用：

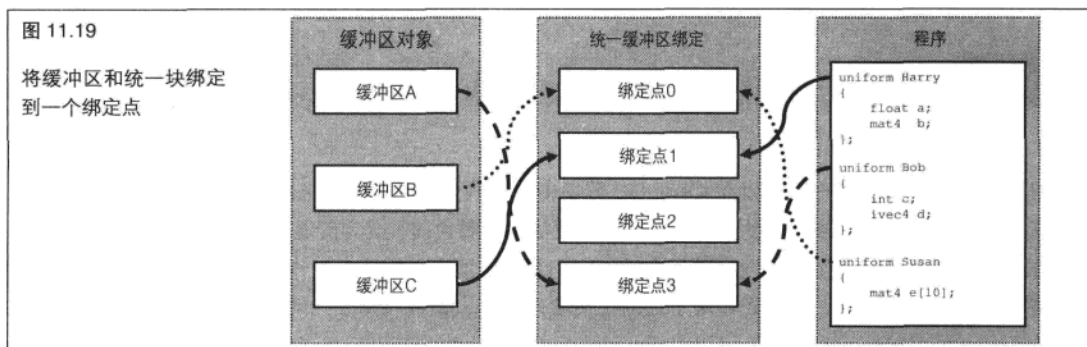
```
glBindBufferBase(GL_UNIFORM_BUFFER, index, buffer);
```

来完成这项工作。

在这里，GL\_UNIFORM\_BUFFER 告诉 OpenGL 我们将一个缓冲区绑定到了统一缓冲区的一个绑定

点上, index 是绑定点的索引, 并且应该和我们在调用 glUniformBlockBinding 时的 uniformBlockBinding 中所指定的相匹配。buffer 是我们想要绑定的缓冲区对象。请注意 index 并不是统一块的索引 ( glUniformBlockBinding 中的 uniformBlockIndex ), 而是统一缓冲区绑定点的索引。这是一个普遍出现的错误, 并且很容易被忽视。

图 11.19 所示说明了这种绑定点与统一块索引之间的混合和匹配。



在图 11.19 中, 有一个包含 3 个统一块 ( Harry、Bob 和 Susan ) 和 3 个缓冲区对象 ( A、B 和 C ) 的程序。Harry 被分配给绑定点 1, 缓冲区 C 被绑定到绑定点 1, 所以 Harry 的数据就来自缓冲区 C。类似地, Bob 被分配给绑定点 3, 而缓冲区 A 被绑定到绑定点 3, 所以 Bob 的数据就来自缓冲区 A。最后, Susan 被分配给绑定点 0, 缓冲区 B 被绑定到绑定点 0, 所以 Susan 的数据就来自缓冲区 B。请注意绑定点 2 并没有被使用。这没有关系, 我们可以将一个缓冲区绑定到这里, 但是程序不会使用它。

程序清单 11.40 列出了对此进行设置的代码, 这些代码非常简单。

程序清单 11.40 为统一块指定绑定

```

// 使用 glGetUniformBlockIndex 来获取统一块的索引
GLuint harry_index = glGetUniformBlockIndex(program, "Harry");
GLuint bob_index = glGetUniformBlockIndex(program, "Bob");
GLuint susan_index = glGetUniformBlockIndex(program, "Susan");

// 使用统一块的索引来为它们分配缓冲区绑定
glUniformBlockBinding(program, harry_index, 1);
glUniformBlockBinding(program, bob_index, 3);
glUniformBlockBinding(program, susan_index, 0);

// 将缓冲区绑定到绑定点
// 绑定 0, 缓冲区 B 和 Susan 的数据进行绑定
glBindBufferBase(GL_UNIFORM_BUFFER, 0, buffer_b);
// 绑定 1, 缓冲区 C 和 Harry 的数据进行绑定
glBindBufferBase(GL_UNIFORM_BUFFER, 1, buffer_c);
// 请注意我们跳过了绑定 2
// 绑定 3, 将缓冲区 C 和 Harry 的数据进行绑定
glBindBufferBase(GL_UNIFORM_BUFFER, 3, buffer_a);
  
```

统一块的一种普遍用法就是从瞬时状态分离出稳定状态。通过使用一个标准约定为所有程序设置绑定, 我们可以在改变程序时让缓冲区保持绑定。例如, 如果有一些相对固定的状态——比如说投影矩阵、视口大小和一些其他东西, 它们每一帧改变一次, 或者更少——那么我们可以将这个信息保留在一个绑定到绑定点 0 的缓冲区中。然后, 如果将所有程序将固定状态的绑定设为 0, 那么无论何时使用 glUseProgram

切换程序，这些统一值都会保留在缓冲区中以备使用。

现在，比如说我们有一个片段着色器模拟某种材质（例如布或金属），就可以将材质的参数放到其他缓冲区中。在我们对这种材质进行着色的程序中，将包含这种材质参数的统一块绑定到绑定 1 上。每个对象都会维护一个包含它的表面参数的缓冲区对象。在对每个对象进行渲染时，它使用通用材质着色器，并将它的参数缓冲区绑定到缓冲区绑定 1。

## 11.6 小结

在本章，我们学习了一些可以在 OpenGL 程序中使用的更加高级的着色器技术。这里介绍的主题中有很多都允许我们编写更加高效的应用程序，生成更简短的着色器，或者实现高级渲染技术。我们了解了使用顶点着色器来完成除了简单地将顶点变换到它们最终位置之外的更多工作。本章引入了一个全新的着色器单元——几何着色器，同时还学习了如何使用它来创建、销毁和修改几何图元。

我们甚至还看到了，可以在几何图形在 OpenGL 管线中进行传递时改变它的类型。除了简单地对模型表面上的像素进行着色之外，片段着色器还有更多的应用。我们可以用它对一个已经进行预渲染的图像上应用后期处理效果，甚至在一个片段着色器中生成整个图像。

本章还介绍了 GLSL 语言的许多更高级特性，可以利用这些特性在着色器中实现有趣的效果和算法。我们知道了可以在缓冲区对象中存储统一值，也可以在切换程序时保持这些缓冲区的绑定。这样大大减少了我们保持常用统一值在程序之间及时更新所需要做的工作。



## 第 12 章 高级几何图形管理

作者: Graham Sellers

## 本章内容

任 务	使用的特性
在顶点缓冲区中管理数据	glVertexAttribPointer
绘制大量几何图形	glMultiDrawArrays, glPrimitiveRestartIndex, glDrawArraysInstanced, glVertexAttribDivisor
存储顶点和几何着色器的结果	glBeginTransformFeedback, glEndTransformFeedback
获取关于 OpenGL 所做工作的信息	glGenQueries, glBeginQuery, glEndQuery, glGetQueryiv
两个或更多 OpenGL 环境同步	glFenceSync, glWaitSync, glClientWaitSync
控制如何对几何图形进行剪切	gl_ClipDistance[], GL_DEPTH_CLAMP

在本章, 我们介绍一些与几何图形管理相关的更高级 OpenGL 特性。其中包括查询已经进行了哪些渲染, 以及从 OpenGL 中获取关于它处理的几何图形数量的信息。本章还介绍了一种存储渲染中间结果以备后续使用的方法, 以及如何对两个 OpenGL 环境进行同步, 以便其中一个环境能够使用另一个环境产生的数据。

我们会学到如何在图形加速卡内存中管理几何图形数据, 以及如何控制 OpenGL 处理批量图元, 例如三角形扇和线带。我们还会学到如何使一个 OpenGL 应用程序将大量几何图形渲染到图形加速卡。

这些技术中有许多都是设计用来提高性能和 GPU 能够处理的任务数量上限的。但是, 也有一些让我们能够使用 GPU 来应用新的、有趣的技术, 而这些技术是用其他方法不能实现的。

## 12.1 查询功能——收集 OpenGL 管线相关信息

我们都希望询问 OpenGL 是否绘制了调用的函数所要的结果, 这看起来是一个奇怪的问题。我们刚

刚调用了一长串的 OpenGL 函数，还将很多几何图形发送到 OpenGL 管线，那么当然要绘制出什么才对。好吧，不要忘记，即使是应该落在屏幕界限之内的几何图形实际上也可能并没有对任何像素进行改变。发生这种情况有很多原因，其中包括三角形由于背面剔除而被丢弃，或者片段没有通过深度测试或被片段着色器丢弃。了解任意像素是否成功显示在了屏幕上，或者甚至是了解到底有多少像素成功显示在了屏幕上可能会非常有用。举例来说，让我们来设想在一个游戏中，屏幕上有很多人物或物体，游戏引擎可能需要知道玩家是否能看到其他一些对象，例如敌人、奖励物品或其他玩家。我们当然也可以根据游戏几何图形资源来构建一个复杂的视线测试。但是，如果向 GPU 查询它是不是实际绘制了对象的任何部分，则要简单得多。

我们是通过遮挡查询来向 GPU 查询这些问题的。这个名称有些误导性，因为它确实更是一种可见查询。答案是 0，或者如果没有绘制任何像素且非 0 则为 false，又或者在绘制了一些像素的情况下为 true。所以这个问题实际上是“这是可视的吗？”，而不是“这被遮挡了吗？”。也许它应该被称为“可视查询”。在任何情况下，一个查询都是代表一个问题的 OpenGL 对象。有几种类型的查询对象，它们代表了所有不同类型的问题，而一个遮挡查询代表的问题是“我们绘制了什么吗？”

## 12.1.1 准备查询

回想一下当初在学校的时候老师要求我们在提问之前先举手。这样很像是排队保留提问的位置——老师不知道我们要提的问题是什么，但是她知道我们有问题要问。OpenGL 也是如此。在我们可以提出问题之前，必须保留一个点，以便 OpenGL 知道我们要问问题了。OpenGL 中的问题由查询对象（query object）表示，就像 OpenGL 中的任何其他对象一样，查询对象必须被保留或生成。要完成这项工作，我们可以调用 `glGenQueries`，传递想要保留的查询数量，以及想要将查询对象放置到其中变量（或数组）的地址。

```
void glGenQueries(GLsizei n, GLuint *ids);
```

这个函数为我们保留了一些查询对象，并告诉我们它们的名字，以便随后可以引用它们。我们可以一次性生成所需的所有查询对象。

```
GLuint one_query;  
GLuint ten_queries[10];  
glGenQueries(1, &one_query);  
glGenQueries(10, ten_queries);
```

在本例中，对 `glGenQueries` 的第一次调用生成了一个简单的查询对象，并在变量 `one_query` 中返回了它的名称。对 `glGenQueries` 的第二次调用生成了 10 个查询对象，并在变量 `one_query` 中返回了 10 个名称。这样，总共生成 11 个查询对象，而 OpenGL 则保留了 11 个唯一的名称来表示它们。OpenGL 不能为我们创建一个查询的情况很难出现，但是仍然是有可能的，在这种情况下它会返回 0 作为查询的名称。一个编写良好的应用程序总是会检查 `glGenQueries` 是否为每个需要的查询对象返回了一个非 0 值。如果出现失败情况，OpenGL 就会跟踪原因，我们可以通过调用 `glGetError` 进行查询。

每个查询对象都会在 OpenGL 中保留一个很小但是可以度量的资源。这些资源必须返回 OpenGL，因为如果不返回的话，OpenGL 可能会为了进行查询而耗尽空间，这样后面就无法为应用程序生成更多

资源了。要将资源返回 OpenGL，可以调用 `glDeleteQueries`。

```
void glDeleteQueries(GLsizei n, const GLuint *ids);
```

`glGenQueries` 的使用也非常类似——它接受要删除查询对象的数量和一个保存它们名称的变量或数组。

```
glDeleteQueries(10, ten_queries);
glDeleteQueries(1, &one_query);
```

在查询被删除之后，它们实际上就彻底消失了。查询的名称不能再次使用，除非通过另一次调用重新将它们分配给我们。

### 12.1.2 发出查询

一旦使用 `glGenQueries` 保留了点，我们就可以提问了。OpenGL 并不会自动报出跟踪它所绘制的像素数量。它必须进行计数，并且必须在开始计数时得到通知。我们可以使用 `glBeginQuery` 进行这项工作。`glBeginQuery` 函数接受两个参数。

第一个参数是我们想要问的问题，而第二个参数则是我们以前保留的查询对象的名称。

```
glBeginQuery(GL_SAMPLES_PASSED, one_query);
```

`GL_SAMPLES_PASSED` 代表我们提出的问题：“有多少样本通过了深度测试？”在这里，OpenGL 对样本进行计数，因为我们可能要渲染到一个多重采样的显示格式，而在那种情况，每个像素可以有一个以上的样本。对于一个普通的单采样格式来说，每个像素有一个样本，这样就有了一个从样本到像素——对应的映射。每一个样本通过了深度测试（也就是说它以前没有被片段着色器丢弃），OpenGL 就进行一次计数。它将进行所有渲染的样本相加在一起，并且将结果保存在为查询对象保留的一部分空间中。

现在 OpenGL 在进行样本（或像素）计数，我们可以像往常一样进行渲染，而 OpenGL 则保持跟踪所有作为结果而生成的像素，我们所渲染的任何东西都会计入总数中。当我们希望 OpenGL 将从我们告诉它开始计数时起所渲染的所有对象都加到一起时，我们可以通过调用 `glEndQuery` 告诉 OpenGL 停止计数。

```
glEndQuery(GL_SAMPLES_PASSED);
```

这样就告诉 OpenGL 停止对传递到深度测试的样本进行计数，并让他们通过片段着色器而不被丢弃。在 `glBeginQuery` 调用和 `glEndQuery` 调用之间进行的所有绘制命令所生成的所有像素都会被加到一起。

### 12.1.3 取回查询结果

现在绘制命令所产生的像素已经被计数，我们需要从 OpenGL 中取回它们。这要通过调用

```
glGetQueryObjectuiv(the_query, GL_QUERY_RESULT, &result);
```

来完成。

这样可以命令 OpenGL 将与查询对象相关的计数放到变量中。如果在最后的 `glBeginQuery` 和

glEndQuery 调用之间进行的所有绘制命令没有产生任何像素作为结果,那么 result 将为 0。如果在屏幕上实际进行了任何渲染,那么 result 将包含所写入的像素数。通过在一次 glBeginQuery 和 glEndQuery 调用之间渲染一个对象,然后再检查 result 是否为 0,我们可以确定对象是否可见。

由于 OpenGL 是作为管线进行操作的,它可能会有很多绘制命令连续排列着等待处理。有一种情况可能发生,即所有绘制命令在最后一次调用 glEndQuery 之前就完成了产生像素的工作。实际上,有一些命令可能甚至根本就没有被执行。在这种情况下,glGetQueryObjectuiv 会让 OpenGL 进行等待,直到 glBeginQuery 和 glEndQuery 之间的所有绘制工作都完成,而它已经准备好返回一个准确的计数。

如果我们计划使用一个查询对象作为一种性能优化,那么这当然不是我们希望看到的。所有这些短暂的延迟都会叠加在一起,并且最终使应用程序变慢!好的消息是,我们可以向 OpenGL 进行查询,它是否完成了任何可能影响查询结果的工作并得到了我们所要的结果。这要通过调用

```
glGetQueryObjectuiv(the_query, GL_QUERY_RESULT_AVAILABLE, &result);
```

来完成。

如果查询对象的结果并不是立即可用的,并且又试图取回这个结果,这样就会导致应用程序必须等待 OpenGL 结束正在进行的工作,而 result 则变为 GL\_FALSE。如果 OpenGL 已经准备好并得到了结果,那么 result 就会为 GL\_TRUE。这就意味着从 OpenGL 取回结果将不会导致任何延迟。

现在我们可以等待 OpenGL 准备好向我们提供像素计数时做一些有用的工作,或者可以根据结果对我们来说是否用来作出决定。例如,如果要跳过某些渲染,并且 result 为 0,那么就可以选择无论如何都继续对它进行渲染,而不是等待查询的结果。

## 12.1.4 使用查询结果

现在有了这些查询结果,我们能用它做些什么呢?遮挡查询有一种普遍的用法,就是通过避免进行不必要的工作来对一个应用程序的性能进行优化。让我们来考虑一个外观非常精细的对象。这个对象有着许多三角形,并且可能会有一个复杂的片段着色器,其中包括大量纹理查询和密集的数学运算。这里可能会有许多顶点属性和纹理,这样仅仅是为了做好绘制对象的准备,应用程序就需要做大量的工作。对这个对象进行渲染是非常消耗资源的。这个对象也有可能永远都不会在场景中可见。它有可能被其他东西覆盖,也可能会整个离开屏幕。如果能够提前知道这些,并且在它永远都不会被用户看到的情况下完全不对它进行绘制就好了。

要做到这一点,遮挡查询就是一种好方法。这种方法可以接受复杂的、代价高昂的对象,并产生一个精度低得多的版本。通常情况下,一个简单的边框就可以完成这项工作。开始一个遮挡查询,对边框进行渲染,然后结束遮挡查询并取回结果。如果这个对象的边框没有哪个部分生成任何像素,那么这个对象细节丰富的版本将不会被看到,它也就没有必要发送到 OpenGL 了。

当然,我们可能并不真正希望在最终的场景中看到这个边框。有很多方法可以保证 OpenGL 不会真的绘制边框。最简单的办法可能就是通过为所有参数都传递 GL\_FALSE 来使用 glColorMask 关闭对颜色缓冲区的写入。

程序清单 12.1 展示了一个简单的示例，演示了如何使用 `glGetQueryObjectuiv` 从一个查询对象中取回结果。

程序清单 12.1 从一个查询对象中获取结果

```
glBeginQuery(GL_SAMPLES_PASSED, the_query);
RenderSimplifiedObject(object);
glEndQuery(GL_SAMPLES_PASSED);
glGetQueryObjectuiv(the_query, GL_QUERY_RESULT, &the_result);
if (the_result != 0)
    RenderRealObject(object);
```

函数 `RenderSimplifiedObject` 可以渲染对象的一个低精度版本，而 `RenderRealObject` 则会渲染对象的所有细节。现在，`RenderRealObject` 只在 `RenderSimplifiedObject` 至少生成一个像素的情况下才会被调用。

请记住，调用 `glGetQueryObjectuiv` 会导致应用程序在得到查询结果之前必须进行等待。如果 `RenderSimplifiedObject` 进行的渲染很简单的话，更容易出现这种情况——这正是本例要讨论的。如果我们只是想要知道跳过某些渲染是否是安全，那么可以先查询对象是否可用，并且在查询结果不可用（也就是说对象可能是可见的，也可能是隐藏的），或者可用并且非 0（也就是说对象确定可见）的情况下，渲染更加复杂的对象。

程序清单 12.2 演示了怎样在查询实际计数之前确定一个查询对象结果是否准备好了，这就可以根据查询结果的可用性和值来做出决定了。

程序清单 12.2 查询遮挡查询结果是否准备好

```
glBeginQuery(GL_SAMPLES_PASSED, the_query);
RenderSimplifiedObject(object);
glEndQuery(GL_SAMPLES_PASSED);
glGetQueryObjectuiv(the_query, GL_QUERY_RESULT_AVAILABLE, &the_result);
if (the_result != 0)
    glGetQueryObjectuiv(the_query, GL_QUERY_RESULT, &the_result);
else
    the_result = 1;
if (the_result != 0)
    RenderRealObject(object);
```

在这个新的示例中，我们确定结果是否是可用的，如果可用则从 OpenGL 取回这个结果；如果是不可用的，就在结果中放入一个计数 1，这样对象的复杂版本就将被渲染。

我们也可以同时有多个活动的遮挡查询。使用多个查询对象，对于应用程序来说是另一种避免等待 OpenGL 的方法。

OpenGL 在同一时间只能进行计数并将结果累加到一个查询对象中，但是它可以管理几个查询对象，并且连续执行很多查询。我们可以对示例进行扩展，并使用多个遮挡查询来渲染多个对象。如果有一个包含 10 个要渲染对象的数组，其中每个对象都有一个简化形式，那么就可以对提供的示例进行重写，正如程序清单 12.3 所示。

程序清单 12.3 简单的应用程序侧条件渲染

```
int n;
for (n = 0; n < 10; n++) {
    glBeginQuery(GL_SAMPLES_PASSED, ten_queries[n]);
    RenderSimplifiedObject(&object[n]);
    glEndQuery(GL_SAMPLES_PASSED);
}
```

```
for (n = 0; n < 10; n++) {  
    glGetQueryObjectiv(ten_queries[n], GL_QUERY_RESULT, &the_result);  
    if (the_result != 0)  
        RenderRealObject(&object[n]);  
}
```

正如前面讨论过的，OpenGL 采用的是管线模式，可以同时进行很多操作。如果我们绘制的是一些简单的东西，例如一个边框，那么它就有可能没有到达管线的终点，而是在需要查询结果时就进行渲染。这就意味着当调用 `glGetQueryObjectiv` 时，应用程序可能必须等待一会儿，等待 OpenGL 完成关于边框的工作，然后为我们提供结果，我们可以根据这个结果进行下一步工作。

在下一个示例中，我们在请求第一个查询的结果之前渲染了 10 个边框。这就意味着 OpenGL 的管线可以被填充，并且它可能有很多工作要做，从而更有可能在我们请求第一个查询结果之前完成第一个边框的工作。简而言之，我们给 OpenGL 用来完成我们要求它做的工作的时间越多，它就越能够获得查询的结果，而应用程序需要等待结果的可能性就越小。某些复杂的应用程序将这一点表现到了极致，并且使用前一帧的查询结果来为新的一帧做决定。

最后，我们将这两种技术都放到一个示例中，程序清单 12.4 列出了这些代码。

程序清单 12.4 在查询结果不可用时进行渲染

```
int n;  
for (n = 0; n < 10; n++) {  
    glBeginQuery(GL_SAMPLES_PASSED, ten_queries[n]);  
    RenderSimplifiedObject(&object[n]);  
    glEndQuery(GL_SAMPLES_PASSED);  
}  
for (n = 0; n < 10; n++) {  
    glGetQueryObjectiv(ten_queries[n],  
                      GL_QUERY_RESULT_AVAILABLE,  
                      &the_result);  
    if (the_result != 0)  
        glGetQueryObjectiv(ten_queries[n],  
                          GL_QUERY_RESULT,  
                          &the_result);  
    else  
        the_result = 1;  
    if (the_result != 0)  
        RenderRealObject(&object[n]);  
}
```

因为通过 `RenderRealObject` 发送到 OpenGL 的任务数量远远大于通过 `RenderSimplifiedObject` 发送的数量，所以等到请求第 2 次、第 3 次、第 4 次和更多查询对象的结果时，会有越来越多的任务被发送到 OpenGL 管线，而查询结果完成的可能性就更大了。在合理的范围内，场景越复杂，使用的查询对象越多，我们获得性能提升的可能性就越大。

如果我们不关心查询结果的实际值，例如在前面的示例中我们只关心结果是不是 0，那么根据使用图形硬件和驱动程序，我们可以使用一个额外的查询类型更快地得到结果。我们可以使用特殊的 `GL_ANY_SAMPLES_PASSED` 查询，而不是 `GL_SAMPLES_PASSED` 查询。这种查询的结果是严格的布尔形式。也就是说，这个结果要么是真要么是假，要么是 0 要么非 0。这种方式在某些硬件上运行更快的原因是，只要第一个像素被渲染，OpenGL 就会知道查询的结果为真。这样，它就可以停止像素计数了。它还可以在得到结果时立即将它返回给我们，即使它还没有完成对发送到遮挡查询中的几何图形的渲染。

如果 OpenGL 实现支持的话，可以在类似的算法中直接用 `GL_ANY_SAMPLES_PASSED` 代替 `GL_SAMPLES_PASSED`，然后就能看到应用程序中的性能提升了。

### 12.1.5 让 OpenGL 决定

前面的示例展示了如何要求 OpenGL 进行像素计数，以及如何从 OpenGL 中获得结果并送到应用程序，以便它能够决定下一步要做什么。但是，应用程序并不真正关心结果的 actual 值。它只是利用这个结果来决定是否要向 OpenGL 发送更多的任务，或者对它用来进行渲染的方法做更多的改变。这些结果必须从 OpenGL 被返回到应用程序，可能是通过 CPU 总线，或者在使用一个远程渲染系统时甚至要通过网络连接来传送，而这仅仅是为了让应用程序可以决定是否向 OpenGL 发送更多的任务。这种方式会导致延时，并且会降低性能，有时候会比使用前面使用查询所获得的潜在优势影响更大。

如果我们能够将所有渲染命令发送到 OpenGL，并告诉它只有在查询结果表明应该执行的时候再执行它们，效果就会好得多。这种方式就叫做推断 (predication)，并且幸运的是，它可以通过一种叫做条件渲染 (conditional rendering) 的技术来实现。

条件渲染允许我们将一个 OpenGL 函数调用序列进行包装并与一个查询对象和一个内容为 “ignore all of this if the result stored in the query object is zero (如果存储在查询对象中的结果为 0 则忽略所有这些内容)” 的消息一起发送到 OpenGL 中。要标记这个调用序列开始，可以使用

```
glBeginConditionalRender(the_query, GL_QUERY_WAIT);
```

而要标记这个序列结束则使用

```
glEndConditionalRender();
```

如果查询对象的结果为 0 (与使用 `glGetQueryObjectiv` 所能取回的是同一个值)，那么在 `glBeginConditionalRender` 和 `glEndConditionalRender` 之间的任何调用都会被忽略。这就意味着查询的实际结果不需要发送回应用程序，图形硬件就能够为我们决定是否进行渲染了。将前面的示例进行修改来使用条件渲染，代码如程序清单 12.5 所示。

程序清单 12.5 基本条件渲染示例

```
// 要求 OpenGL 对在遮挡查询的开始和结束之间渲染的样本进行计数
glBeginQuery(GL_SAMPLES_PASSED, the_query);
RenderSimplifiedObject(object);
glEndQuery(GL_SAMPLES_PASSED);
// 只有在遮挡查询表明有些内容被渲染时才执行下面的几条命令
glBeginConditionalRender(the_query, GL_QUERY_WAIT);
RenderRealObject(object);
glEndConditionalRender();
```

`RenderSimplifiedObject` 和 `RenderRealObject` 两个函数分别是我们假想示例应用程序的简单渲染版本 (例如可能只有边框而已) 和更复杂的版本。

现在要注意，我们从没有调用过 `glGetQueryResultuiv`，也从没有从 OpenGL 中读取任何信息 (例如查询对象的结果)。这种方式对于远程渲染来说非常有利，因为在这种情况下结果必须经过网络的传输才能最终到达应用程序。

读者可能已经注意到了传递到 `glBeginConditionalRender` 的 `GL_QUERY_WAIT` 参数。读者可能会好奇，如果应用程序不需要等待结果完成的话，那么这样做又是为什么呢？正如前面提过的，OpenGL 是作为管线来操作的，而在调用 `glBeginConditionalRender` 之前，或者在从 `RenderRealObject` 中调用第一个绘制函数之前，OpenGL 可能还没有完成对 `RenderSimplifiedObject` 的处理。在这种情况下，OpenGL 要么可以等待从 `RenderSimplifiedObject` 进行的所有调用都到达管线的终点，然后再决定是否要执行应用程序发送的命令；要么也可以在结果没有及时完成的情况下继续对 `RenderRealObject` 进行处理。要告诉 OpenGL 如果结果不可用的话，不要等待，而是继续开始进行渲染，我们需要调用：

```
glBeginConditionalRender(the_query, GL_QUERY_NO_WAIT);
```

这样就告诉 OpenGL “如果查询的结果还不可用，就不要等它们了，无论如何都继续渲染吧”，这在使用遮挡查询来提升性能时是最好的做法，等待遮挡查询的结果可能会用光以前使用它们所节省的时间。这样，使用 `GL_QUERY_NO_WAIT` 标记在本质上说就是在结果及时完成的情况下允许遮挡查询作为一个优化使用，而在结果没有完成的情况下则当作没有使用它们一样。使用 `GL_QUERY_NO_WAIT` 和在前面的示例中使用 `GL_QUERY_RESULT_AVAILABLE` 非常类似。

但是，不要忘记，如果使用 `GL_QUERY_NO_WAIT`，那么实际渲染的几何图形就会以这些命令是否有助于那些执行完成的查询对象为依据。这可能取决于运行应用程序的机器的性能，从而在每次运行时都会不同。我们应该确保程序的结果不会依赖于进行渲染的第二组几何图形（除非这是我们所希望的）。如果情况确实如此，那么程序可能最终会在一个比较快的系统和一个比较慢的系统中产生不同的输出结果。

当然，我们也可以使用多个条件渲染的查询对象，这样就产生了一个使用这一部分介绍的所有技术的最终综合示例，如程序清单 12.6 所示。

程序清单 12.6 一个更加完善的条件渲染示例

```
// 渲染 10 个对象的简化版本，每个对象都带有自己的遮挡查询
int n;
for (n = 0; n < 10; n++) {
    glBeginQuery(GL_SAMPLES_PASSED, ten_queries[n]);
    RenderSimplifiedObject(&object[n]);
    glEndQuery(GL_SAMPLES_PASSED);
}
// 渲染这些对象的更复杂版本，如果遮挡查询结果可用并为 0 的话则跳过它们

for (n = 0; n < 10; n++) {
    glBeginConditionalRender(ten_queries[n], GL_QUERY_NO_WAIT);
    RenderRealObject(&object[n]);
    glEndConditionalRender();
}
```

在本例中，首先渲染了 10 个对象的简化版本，每个对象都带有遮挡查询。一旦这些对象的简化版本被渲染，那么这些对象的更复杂版本就会根据这些遮挡查询的结果进行条件遮挡。如果这些对象的简化版本不可见，就会跳过这些复杂版本的处理，这就有可能使性能得到提高。

## 12.1.6 测量执行命令所需时间

我们可以使用一个更深入的查询类型来判断渲染要花费多长时间，这就是定时器查询（timer query）。

我们可以通过传递 `GL_TIME_ELAPSED` 查询类作为 `glBeginQuery` 和 `glEndQuery` 的 `target` 参数来使用定时器查询。

当调用 `glGetQueryObjectiv` 从查询对象中获取结果时，得到的结果是在调用 `glBeginQuery` 和 `glEndQuery` 之间所经过的纳秒数。这实际上是处理位于 `glBeginQuery` 和 `glEndQuery` 命令之间的所有命令所花费的时间量。

举例来说，我们可以使用它来查询场景中开销最大的是哪一部分。现在让我们来考虑程序清单 12.7 中显示的代码。

程序清单 12.7 使用计时器查询的计时器操作

```
// 声明我们的变量
GLuint queries[3];    // 将要使用的 3 个查询对象
GLuint world_time;    // 绘制整个场景所需的时间
GLuint objects_time;  // 在场景中绘制对象所需的时间
GLuint HUD_time;      // 绘制 HUD 和其他 UI 元素所需的时间

// 创建 3 个查询对象
glGenQueries(3, queries);
// 开始第一个查询
glBeginQuery(GL_TIME_ELAPSED, queries[0]);
// 对整个场景进行渲染
RenderWorld();
// 停止第一个查询并开始第二个查询...
// 请注意，我们还没有从查询中读取这个值
glEndQuery(GL_TIME_ELAPSED);
glBeginQuery(GL_TIME_ELAPSED, queries[1]);
// 在整个场景中对对象进行渲染
RenderObjects();
// 停止第二个查询并开始第 3 个查询
glEndQuery(GL_TIME_ELAPSED);
glBeginQuery(GL_TIME_ELAPSED, queries[2]);
// 渲染 HUD
RenderHUD();
// 停止最后一个查询
glEndQuery(GL_TIME_ELAPSED);
// 现在，可以从第 3 个查询中取回结果了。现在进行到这里，希望 RenderWorld() 已经通过管线，并且结果已经完成
glGetQueryObjectiv(queries[0], GL_QUERY_RESULT, &world_time);
glGetQueryObjectiv(queries[1], GL_QUERY_RESULT, &objects_time);
glGetQueryObjectiv(queries[2], GL_QUERY_RESULT, &HUD_time);
// 完成了。world_time、objects_time 和 hud_time 包含了所需要的值
// 最后进行清除工作
glDeleteQueries(3, queries);
```

在这些代码执行后，`world_time`、`objects_time` 和 `HUD_time` 将分别包含对整个场景进行渲染所花费的纳秒数、整个场景中的所有对象和平视显示器（HUD）。我们可以使用它来确定图形硬件的哪个时间段被用来对场景中的每个元素进行渲染。在开发过程中对代码进行分析非常有用——我们可以查出应用程序中哪一部分最消耗资源，从而也就知道了要对哪里重点投入精力进行优化。我们还可以在运行时使用它改变应用程序的行为，以试图获得图形子系统可能的最佳性能。例如，我们可以根据 `objects_time` 的相对值增加或减少场景中对象的数量，还可以根据图形硬件的能力，针对场景的元素在复杂性更高或更低的着色器之间进行切换。

如果我们只想知道在程序进行两次动作之间经过了多长时间（根据 OpenGL），可以使用

glQueryCounter, 其原型为:

```
void glQueryCounter(GLuint id, GLenum target);
```

我们需要将 id 设置为 GL\_TIMESTAMP, 而 target 则设置为我们早先创建的一个查询对象的名称。这个函数将查询直接放到了 OpenGL 管线中, 当这个查询到达管线的终点时, OpenGL 会将视角下的当前时间记录到查询对象。时间 0 并没有真正的定义——它只是代表过去的一些未定义时间。为了更加有效地使用它, 应用程序需要获取多个时间戳之间的差值。为了使用 glQueryCounter 实现前面的例子, 可以编写如程序清单 12.8 所示的代码。

程序清单 12.8 使用 glQueryCounter 进行计时操作

```
// 声明我们的变量
GLuint queries[4]; // 现在需要 4 个查询对象
GLuint start_time; // 应用程序的开始时间
GLuint world_time; // 绘制整个场景所花费的时间
GLuint objects_time; // 在场景中绘制对象所花费的时间
GLuint HUD_time; // 绘制 HUD 和其他 UI 元素所需的时间

// 创建 4 个查询对象
glGenQueries(4, queries);
// 获取开始时间
glQueryCounter(GL_TIMESTAMP, queries[0]);
// 对整个场景进行渲染
RenderWorld();
// 在 RenderWorld 完成之后获取时间
glQueryCounter(GL_TIMESTAMP, queries[1]);
// 在整个场景中对对象进行渲染
RenderObjects();
// 在 RenderObjects 完成之后获取时间
glQueryCounter(GL_TIMESTAMP, queries[2]);
// 渲染 HUD
RenderHUD();
// 在所有工作完成之后获取时间
glQueryCounter(GL_TIMESTAMP, queries[3]);
// 从 3 个查询中获取结果, 并将它们相减获得差值
glGetQueryObjectuiv(queries[0], GL_QUERY_RESULT, &start_time);
glGetQueryObjectuiv(queries[1], GL_QUERY_RESULT, &world_time);
glGetQueryObjectuiv(queries[2], GL_QUERY_RESULT, &objects_time);
glGetQueryObjectuiv(queries[3], GL_QUERY_RESULT, &HUD_time);
HUD_time -= objects_time;
objects_time -= world_time;
world_time -= start_time;
// 完成了。world_time、objects_time 和 hud_time 包含了我们所需要的值
// 最后进行清除工作
glDeleteQueries(4, queries);
```

就像我们所能看到的, 这个示例中的代码与前面在程序清单 12.7 中展示的代码差别并不太大。我们需要创建 4 个查询对象, 而不是 3 个, 最后需要将这些结果相减以得到时间差。但是, 我们不需要成对调用 glBeginQuery 和 glEndQuery, 这就意味着我们对 OpenGL 调用的总量减少了。

## 12.2 在 GPU 内存中存储数据

到目前为止, 我们使用的所有几何图形 (顶点、颜色、法线和其他顶点属性数据) 都是由 GLTools

库管理的。当调用诸如 `GLBatch::CopyVertexData` 或 `GLBatch::CopyNormalData` 这样的函数时，指定的指针是一个指向内存中包含顶点坐标、颜色、法线和其他我们想要渲染的数据的真正指针。如果每次调用 `glDrawArrays`、`glDrawElements` 或者其他一些需要顶点数据的函数时，这些信息是从一个带有本地 GPU 的高性能系统中的应用程序内存中获取的，这可能就意味着数据将从应用程序的内存（绑定到 CPU）中通过将 CPU 连接到 GPU（通常是通过 PCI-Express 接口）的总线传递到 GPU 本地内存，以便可以对其进行操作。这样做会花费很多时间，以至于大大降低应用程序的运行速度。在远程系统中，数据可能会通过一个网络连接被传送到服务器进行渲染。这样做对于性能来说是灾难性的。

当 GPU 访问本地内存（例如与视频卡物理绑定）时，可能只需要进行几次访问，甚至可能比访问系统内存中同样数据快几倍。在远程渲染系统的情况下，访问本地 GPU 内存可能比通过网络连接发送这些数据快几万倍。如果对于每个帧来说，要进行渲染的数据基本上相同，或者如果在单个帧中对同样数据的多个副本进行渲染，那么一次将这些数据复制到 GPU 的本地内存中，再很多次重复使用这个副本是非常有利的。

为了允许这种情况发生，GLTools 中的各种类会对 GPU 本地内存中的缓冲区进行管理，并且隐藏这些操作的复杂性。但是，实际上管理这些缓冲区也不是特别困难。当我们开始编写更加复杂的应用程序，而这些应用程序除了简单的位置、颜色和法向量之外，还需要其他数据时，最终将会需要做这些工作。

在这一部分内容，我们会学习到如何确保 GPU 所需的顶点数据和其他信息是可用的，并且存储在内存中。要完成这项工作，可以使用包含应用程序所提供数据的缓冲区对象。我们还会学习如何管理这些对象，如何告诉 OpenGL 我们将会使用它来做什么，以及如何在 GPU 内存中以最好的方式保存数据。

## 12.2.1 使用缓冲区存储顶点数据

在 OpenGL 中，我们可以在一个缓冲区对象中保存诸如位置、颜色等顶点属性数据，或其他任何顶点着色器所需的信息。缓冲区对象是表示数据存储的 OpenGL 对象，在本书前面的内容中已经介绍过了。在这里，我们使用一个 OpenGL 缓冲区作为一个顶点缓冲区对象（VBO）。VBO 是一个用来表示顶点数据存储的缓冲区对象。数据可以存储在这些缓冲区中，并且带有一些提示，告诉 OpenGL 我们计划如何使用它，而 OpenGL 则可以使用这些提示来决定怎样处理这些数据。如果这些数据将要被使用不止一次，那么 OpenGL 一定会将它复制到绑定在图形卡上的快速内存中。

由于一个复杂的应用程序可能会需要几个 VBO 和许多顶点属性，所以 OpenGL 有一个可用的叫做 VAO（顶点数组对象，vertex array object）的特殊容器对象，用来管理所有这些状态。下一节将更详细地讨论 VAO。但是，由于不存在默认的 VAO，我们就需要创建并绑定一个 VAO，然后才能使用这一部分的任何代码。下面这些代码就应该足以完成这些工作了。

```
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
```

这些代码会创建并绑定单个 VAO。这个 VAO 可以在应用程序运行过程中保持绑定，而我们则能够使用和控制顶点缓冲区。要创建一个或多个缓冲区对象，可以调用

```
glGenBuffers(1, &one_buffer);
```

或

```
glGenBuffers(10, ten_buffers);
```

要将顶点数据保存到一个缓冲区或从缓冲区中取回，它必须被绑定到 `GL_ARRAY_BUFFER` 绑定点。我们通过调用

```
glBindBuffer(GL_ARRAY_BUFFER, one_buffer);
```

来完成这项工作。

一旦进行绑定，我们可以使用很多函数来控制缓冲区对象，这些函数需要一个缓冲区绑定作为参数。`glBufferData`、`glBufferSubData`、`glMapBuffer` 和 `glCopyBuffer` 就是这些函数的例子。

在调用 `glVertexAttribPointer` 时，属性指针的值不会作为一个指向内存中数据的真实物理指针进行解释。这个指针实际上是作为一个在进行调用时绑定到 `GL_ARRAY_BUFFER` 绑定点的到缓冲区对象的偏置来进行解释的。同样，当前绑定的缓冲区的记录也在当前 VAO 中产生，并用于这个属性。这就是说，`glVertexAttribPointer` 不仅告诉 OpenGL 能够找到一个顶点属性数据的缓冲区偏置，它还告诉 OpenGL 哪个缓冲区包含了这些数据。

这样我们就可以通过为每个属性调用 `glBindBuffer`，接着调用 `glVertexAttribPointer` 来同时使用多重缓冲区了——每个属性对应一个缓冲区。我们还可以通过交叉存取在单个缓冲区中存储几个不同的属性。要完成这项工作，可以调用 `glVertexArrayPointer`，并将步长参数设置为同类型属性之间的距离（以位为单位）。最后，因为每个顶点属性都有参数组，包括偏置、步长和缓冲区绑定，所以也可以使用一个交叉存取和独立缓冲区的组合。举例来说，一个单独的模型可以在一个缓冲区中进行位置和法线的交叉存取，而在另一个独立的缓冲区中存储纹理坐标。这样就允许我们只通过为纹理坐标顶点属性改变缓冲区绑定，就可以在同一个模型上使用不同纹理坐标的不同纹理。

程序清单 12.9 展示的示例创建了单个缓冲区，将它绑定到 `GL_ARRAY_BUFFER` 绑定点，存入一些数据，然后设置一个顶点属性指针来引用这个缓冲区。这里将一个大数据块放入了缓冲区（data 数组），而且它占用了整个缓冲区。

程序清单 12.9 对单个 VBO 进行定位和初始化

```
// 这个变量将会保存缓冲区的名称
GLuint my_buffer;
// 这个数组包含要用来初始化这个缓冲区的数据
// 通常情况下，这些数据实际上是存储在一个文件中的，而不是存储在一个原始的 C 数组中
static const GLfloat data[] = { 1.0f, 2.0f, 3.0f, 4.0f, ... };
// 创建一个缓冲区
glGenBuffers(1, &my_buffer);
// 一个行为良好的应用程序会在这里检查缓冲区创建是否成功只要将它进行绑定，希望得到最好的结果
glBindBuffer(GL_ARRAY_BUFFER, my_buffer);
// 将数据放入缓冲区之前，不会为缓冲区分配任何存储空间。下面的操作将“数据”数组的内容复制到了这个缓冲区中
glBufferData(GL_ARRAY_BUFFER, sizeof(data), data, GL_STATIC_DRAW);
// 现在，设置顶点属性指针。它的位置为 0（这一点我们已经知道了），大小为 4（顶点着色器的属性被声明为 vec4），我们有没经过标准化的浮点数据
// 步长为 0，因为在本例中数据进行了紧密的包装。最后，请注意我们传递 0 作为指向数据的指针。这样做是合法的，因为它将被解释为一个到“”的偏置，而数据确实是从偏置 0 开始的
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, (const GLvoid *)0);
```

程序清单 12.10 展示的下一个示例创建了单个缓冲区，在其中不同的位置存入了一些数据，然后设置几个顶点属性指针指向这些数据的偏置。这个示例演示了如何使用一个缓冲区来保存几个独立的属性，并同时为每个属性保存所有数据。

程序清单 12.10 使用单个 VBO 来保存多个顶点属性

```
// 下面是我们将要使用的新数据:
static const GLfloat positions[] = { /* many floating point vec4s */ };
static const GLfloat colors[] = { /* more floating point vec4s */ };
static const GLfloat normals[] = { /* a bunch of floating point vec3s */ };
// 假定我们已经像前面的例子中那样创建并绑定了一个缓冲区
// 现在，要指定大小为数据分配空间
// 但是使用 NULL 作为指向数据的指针
glBufferData(GL_ARRAY_BUFFER,
             sizeof(positions) + sizeof(colors) + sizeof(normals),
             NULL, GL_STATIC_DRAW);
// 现在我们可以将单独的数组复制到这个大缓冲区中了
glBufferSubData(GL_ARRAY_BUFFER, 0,
                sizeof(positions), positions);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(positions),
                sizeof(colors), colors);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(positions) + sizeof(colors),
                sizeof(normals), normals);
// 现在这个缓冲区包含了 3 个大数据块中的 3 个属性的数据，我们可以一个接一个地将顶点属性指针设置到这个数据在这个缓冲区中的偏置。
// 首先是位置:
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0,
                      (const GLvoid *)0);
// 然后是颜色:
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0,
                      (const GLvoid *)sizeof(positions));
// 然后是法线:
glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, 0,
                      (const GLvoid *) (sizeof(positions) + sizeof(colors)));
```

在程序清单 12.11 中给出的最终示例中，用单个缓冲区来保存交叉存取的属性数据。数据被声明为一个 C 结构体，并被直接复制到缓冲区中。glVertexAttribPointer 的 stride 参数用来告诉 OpenGL，内存中属性之间的间隔是多少位。这是一个交叉存取的示例。单个顶点的所有这些属性最终在缓冲区中是一个接一个紧接着放置的。

程序清单 12.11 使用单个 VBO 来保存交叉存取属性

```
// VERTEX 结构体包含单个顶点的位置、颜色和法线，在内存中包装在一起
struct VERTEX_t
{
    vec4 position;
    vec4 color;
    vec3 normal;
};
typedef struct VERTEX_t VERTEX;
// 假定这里有一些顶点数据的扩展数组
extern VERTEX vertices[];

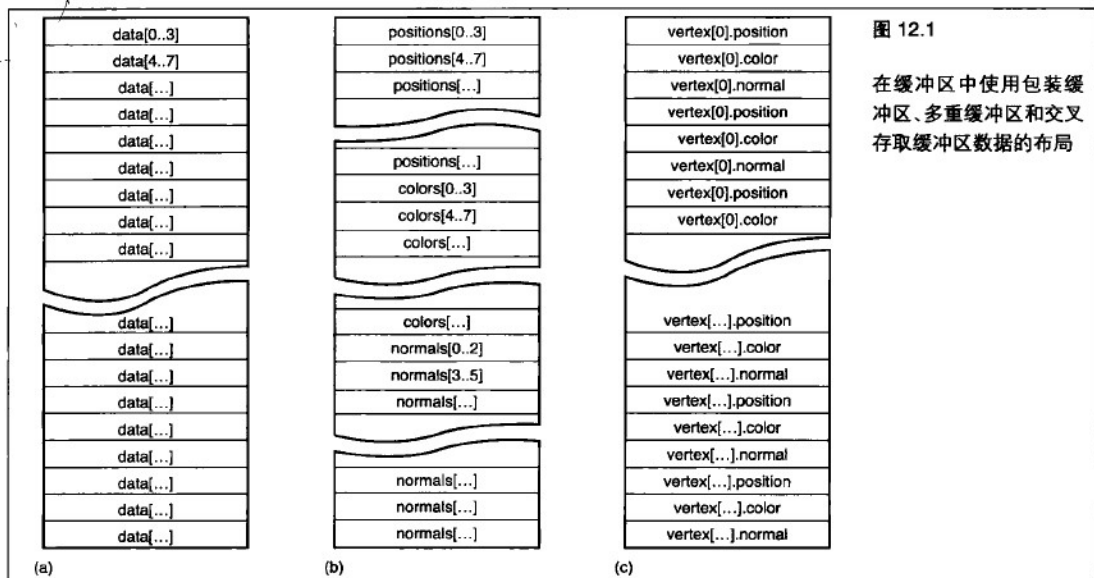
// 现在可以将所有顶点数据复制到这个大缓冲区中了
glBufferData(GL_ARRAY_BUFFER, vertex_count * sizeof(VERTEX),
             vertices, GL_STATIC_DRAW);
// 现在每个顶点属性都来自同一个缓冲区了
// 步长参数是指从一个顶点到下一个顶点之间的距离，以位为单位——也就是 (VERTEX) 的大小，而数据在缓冲区中的位置仅仅是这个结构体中元素的偏置
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, sizeof(VERTEX),
```

```

        (const GLvoid *)OFFSETOF(VERTEX, position));
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, sizeof(VERTEX),
        (const GLvoid *)OFFSETOF(VERTEX, color));
glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, sizeof(VERTEX),
        (const GLvoid *)OFFSETOF(VERTEX, normal));

```

这个数据在缓冲区中的输出布局如图 12.1 所示。在(a)中，数据仅仅是复制到缓冲区中，并且就像在应用程序内存中一样出现在 GPU 内存中；在(b)中，一些属性数组被连续地放置到缓冲区中；在(c)中，每个顶点的独立属性都会一起进行交叉存取。



和某些其他的 OpenGL 对象不同，这里没有默认缓冲区对象。这就意味着我们必须先创建一个顶点缓冲区对象并对其进行绑定，然后才能调用 glVertexAttribPointer。OpenGL 保留一个命名为 0 的缓冲区对象，代表“没有缓冲区”。这样，要对一个缓冲区解除绑定而不指定新的缓冲区来代替它，只需将名称 0 绑定到 GL\_ARRAY\_BUFFER 绑定点。

```
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

如果这些数据经常被应用程序改变，那么在应用程序的内存空间保存数据可能就会非常有用。但是，为此使用一个 VBO 仍然是必要的，因为 OpenGL 并不支持直接从系统内存读取数据。如果在调用 glBufferData 时指定 GL\_STREAM\_DRAW 的使用模式，OpenGL 知道这些数据很可能只会使用一次，而应用程序的行为和性能应该与在应用程序内存中保存数据时相同。在任何情况下，即使能够在应用程序内存中保存数据，OpenGL 驱动很可能也会在内部执行一个相似的操作，并最终在使用这些数据之前将它们复制到 GPU 内存的一个暂存区域中。

## 12.2.2 在缓冲区中保存顶点索引

到目前为止，我们只讨论了 GL\_ARRAY\_BUFFER 绑定，还有一个相关的缓冲区绑定，即

GL\_ELEMENT\_ARRAY\_BUFFER 绑定。元素数组缓冲区是一个存储顶点索引的缓冲区，供 glDrawElements 和 glDrawRangeElements 这样的函数使用。对于 GL\_ELEMENT\_ARRAY\_BUFFER 绑定来说，没有与 glVertexAttribPointer 等价的函数。举例来说，这就是说没有 glElementPointer 函数。我们可以使用 GL\_ELEMENT\_ARRAY\_BUFFER 绑定，就像任何其他为了对它进行分配或者将数据放入其中（例如使用 glBufferData 或 glBufferSubData）而进行的绑定。下面来看一下 glDrawElements 的函数原型。

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

第一个参数 indices 是元素数组缓冲区中第一个索引的偏置。

请记住，在调用 glVertexAttribPointer 时，最后一个参数 pointer 将被解释为到这个缓冲区的一个偏置，它被绑定到 GL\_ARRAY\_BUFFER 绑定点。对于 glDrawElements 的 indices 参数来说同样如此，而缓冲区则绑定到 GL\_ELEMENT\_ARRAY\_BUFFER 绑定点。如果在调用 glDrawElements 时，一个非 0 缓冲区被绑定到 GL\_ELEMENT\_ARRAY\_BUFFER 绑定点，那么 indices 将被解释为一个到缓冲区的偏置，而这些将要被绘制顶点的索引将从这个缓冲区中获取。

如果没有缓冲区被绑定到 GL\_ELEMENT\_ARRAY\_BUFFER 绑定点，那么 glDrawElements 将不会做任何事。如果没有元素缓冲区，那么就没有顶点索引的存储会被绘制。正如 OpenGL 不支持从应用程序内存中读取顶点属性数据一样，它也不能读取顶点索引数据。这样，为了使用 glDrawElements 函数，我们就必须有一个缓冲区绑定到 GL\_ELEMENT\_ARRAY\_BUFFER 绑定点。

还有一个 glDrawElements 的更高级版本，允许我们使用同一个索引，但是要为每次调用使用不同的顶点数据。glDrawElementsBaseVertex 函数允许我们指定一个偏置，这个偏置将在用于从顶点缓冲区中读取数据之前添加到每个顶点索引上。它的原型为：

```
void glDrawElementsBaseVertex(GLenum mode, GLsizei count, GLenum type, GLvoid
*indices, GLint basevertex);
```

让我们考虑一个比较复杂的模型，它使用几个顶点缓冲区——比如说，一个用来存储位置数据，一个用来存储法线，两个或 3 个用来存储纹理坐标，并且可能有另外几个用来存储对模型进行渲染时可能用到的其他数据。如果模型是动画的，那么我们就需要几个帧，每个帧都有它们自己的这些数据的完整集合。在每个帧中，位置是改变的，法线是变化的，但是这些顶点的索引将保持不变。在这里我们有几个选择。我们可以为每个帧使用单独的一组 VBO。这样做可能需要进行很多设置，但却是一个可行的解决方案。我们可以为对象的每个帧使用一个单独的 VAO（在后面部分将更加详细地讨论），它将存储所有绑定。另一种选择是将所有数据存储在一个单独的大缓冲区中，其中的数据一帧接一帧地进行包装。这样做会要求我们为每一帧多次调用 glVertexAttribPointer。同样，我们可以为每一帧将这些信息存储在一个独立的 VAO 中。

glDrawElementsBaseVertex 函数是一个可选项，它允许我们指定到缓冲区的相对于元素缓冲区中索引的偏置。举例来说，如果我们有一个包含 1000 个顶点的模型，那么第一个帧就从偏置 0 开始，第二帧从偏置 1000 开始，第 3 个从 2000 开始，依次类推。将偏置传递到 glDrawElementsBaseVertex 命令的操作比起重新绑定 VBO 或 VAO 来，或者比起处理几个 glVertexAttribPointer 调用来说要简单得多（从 OpenGL 的角度来看）。实际上，对于每一个 glDrawElements 变体来说，都有一个等价的版本，它接受一个 basevertex 参数：glDrawElementsBaseVertex、glDrawRangeElementsBaseVertex、glDrawElementsInstancedBaseVertex 或 glMultiDrawElementsBaseVertex。OpenGL 规范中比较详细地介绍了它们。

## 12.3 使用顶点数组对象来组织缓冲区

我们刚刚已经了解了一下顶点缓冲区对象。每个顶点属性都有一个缓冲区中的偏置，以及一组其他状态，例如数据类型和步长。每个顶点属性也都有一个关联缓冲区，它对于每个属性来说可以是不同的。调用 `glVertexAttribPointer` 可以对所有这些状态进行设置，包括属性的缓冲区绑定。如果我们有一个非常复杂的场景，其中有几个对象，并且每个对象的顶点数据都保存在这些顶点的 VBO 中，那么每个对象都会有数量可观的状态。如果这个应用程序编写良好，那么对这些对象中的一个进行绘制应该像单次调用一个诸如 `glDrawElements` 和 `glDrawArrays` 这样的函数一样简单。

即使对象之间数据的布局是相同的（对于很多应用程序来说可能都会如此），并且数据的偏置都相同（例如，可能所有数据都从偏置 0 开始），那么为每个顶点属性调用 `glVertexAttribPointer` 就仍然是必要的。比如说，对于一个有 8 个顶点属性的对象来说，这就意味着至少调用 `glBindBuffer` 一次（如果顶点属性在相互独立的缓冲区中的话，很可能会调用多达 8 次），并且调用 `glVertexAttribPointer` 8 次。

如果我们使用的是索引顶点，那么还需要绑定 `GL_ELEMENT_ARRAY_BUFFER`。所有这些都是为了对 `glDrawElements` 进行单次调用所做的准备。这里需要设置许多状态，驱动程序必须进行多次错误检查，而应用程序则需要关心许多信息。

为了帮助组织所有这些信息，OpenGL 提供了一个对象，叫做顶点数组对象（VAO）。VAO 是一个容器，它将所有可以由 `glVertexAttribPointer` 和其他一些函数进行设置的状态包装到一起。在使用一个 VAO 时，所有通过一次 `glVertexAttribPointer` 调用来指定的状态都会被存储到当前的 VAO 中。在 OpenGL 中不存在默认 VAO。这就意味着我们必须先创建一个 VAO 并对其进行绑定，然后才能指定顶点指针。对于一个简单的应用程序来说，创建单个 VAO，对其进行绑定，并且在应用程序的生命周期中保持其绑定（就像我们在前面介绍 VBO 时所做的）可能就足够了。但是，一个应用程序可以创建所需的任意多个 VAO，并使用它们来管理所有这些数组状态。在使用特定一组顶点属性进行绘制时，只要对包含这组状态的 VAO 进行绑定并开始绘制就可以了。这样做允许一个场景中的每个对象通过创建一个 VAO 来保存它的状态，并在进行绘制前将它进行绑定，来管理顶点缓冲区。这样，这个对象就不会干扰场景中任何其他对象的顶点数组状态了。

为了创建一个或多个 VAO，可以调用：

```
void glGenVertexArrays(GLsizei n, GLuint *arrays);
```

就像大多数其他 OpenGL 对象一样，VAO 也是通过表示为无符号整数的名称来引用的。函数 `glGenVertexArrays` 创建 `n` 个顶点数组并将它们的名称放到数组 `arrays` 中。如果由于某些原因，`glGenVertexArrays` 不能成功地为一个 VAO 进行分配，那么就会返回 0 作为它的名称。一个编写良好的应用程序应该总是在试图使用结果之前检查这个条件。和其他缓冲区对象一样，OpenGL 保留 VAO 的名称 0，代表“没有 VAO”。同样，在没有 VAO 被绑定时，`glVertexAttribPointer` 将不会工作，并且如果调用它的话，将生成一个错误。为了删除 VAO，可以调用：

```
void glDeleteVertexArrays(GLsizei n, GLuint *arrays);
```

这个函数会删除名称保存在 arrays 中的这 n 个 VAO。对于应用程序来说，最后进行清除非常重要。如果 arrays 有一个元素包含名称 0，那么它将被忽略。这就意味着我们可以安全地将一个以前由 glGenVertexArrays 进行写入的数组传递到 glDeleteVertexArrays，而无需担心这些名称中是否会有一些为 0（比如说是由于在执行 glGenVertexArrays 时的一个错误而出现的）。为了开始使用 VAO，可以调用：

```
void glBindVertexArray(GLuint array);
```

这样就会将 array 设为当前 VAO。当一个新的 VAO 第一次被绑定时，它会包含所有默认状态，这些状态会出现在一个新创建的环境中。从现在开始，我们无论在什么时候调用一个访问顶点数组状态的函数，它都会访问当前绑定的 VBO 中所包含的状态。这其中包含设置状态的函数，例如 glVertexAttribPointer；隐式地使用这个状态的函数，例如 glDrawArrays 或 glDrawElements；以及显式地读取数组状态的函数，例如 glGetIntegerv。

现在我们有了一个 VAO，可以在它上面设置我们所需任意数量的状态了。我们可以对 glVertexAttribPointer 进行所需任意次数的调用，而这些状态将被存储在这个 VAO 中。如果调用 glBindBuffer，然后接着调用 glVertexAttribPointer，那么缓冲区绑定也将被存储在 VAO 中。但是，要注意的是，当与顶点属性相关联的缓冲区绑定被存储在 VAO 中时，绑定一个新的 VAO 并不能改变当前的缓冲区绑定。也就是说，当前绑定缓冲区的实际状态并没有保存在 VAO 中。回到本节开始时的例子——那个有很多顶点属性的对象，每个属性都有不同的状态和缓冲区绑定，我们可以使用 VAO 显著地提高它的性能。

我们可以不在即将绘制对象之前再对 glBindBuffer 和 glVertexAttribPointer 进行很多次调用，而是在初始化时进行这项工作。在对象创建时，它可以生成一个 VAO，使用 glBindVertexArray 对它进行绑定，并且对所有顶点数组状态进行设置，就像它即将渲染自己一样。在进行初始化之后返回 OpenGL，这样就没有 VBO 会通过调用

```
glBindVertexArray(0);
```

进行绑定了。

现在，当对象即将被渲染时，再次调用带有对象的 VBO 的 glBindVertexArray，然后再调用诸如 glDrawArrays 这样的渲染函数。这样，渲染一个包含许多存储在一个不同参数的 VBO 集合中的顶点的完整对象就可以变得非常简单，只需调用两个函数——例如 glBindVertexArray 和 glDrawElements。这样对于分层库（layered library）、场景图（scene graph）管理和可能想要在不干扰当前 OpenGL 状态的情况下进行渲染的中间件来说也是非常有利的。如果这个环境的正常行为就是没有绑定的 VAO，那么每个对象都会绑定自己的 VAO，对自己进行渲染，然后绑定 VAO 0，并进行全面的重新设置。

## 12.4 高效地绘制大量几何图形

到目前为止，我们已经了解了如何将数据块发送到 OpenGL 来使用诸如 glDrawArrays 这样的函数进行渲染了。我们可以通过单次调用这个函数来发送大量顶点——如果必要的话，可以是数百万个到

OpenGL。但是，如果几何图形已经在一个大连续块中良好地进行排列的话，那么这就是唯一的用处了。在任何复杂的应用程序中，都会有很多不同的、互不相关的对象。很可能会有一个场景或某种背景，而它们中的每一个都可能需要几次调用某个绘制函数。一个复杂的应用程序在每一帧中都对 OpenGL 提供的各种绘制函数进行几千次甚至几十万次调用，这种情况并不罕见。在本节，我们会了解一些方法，使用这些方法，能够通过对 OpenGL 进行很少的几次调用来绘制大量独立的几何图形片段。

### 12.4.1 组合绘制函数

如果我们在单个应用程序中有很多几何图形要发送到 OpenGL，那么就很可能采用一种首选的绘制方法。举例来说，可能会使用 `glDrawArrays` 或 `glDrawElements`。如果要将所有对象的所有顶点数据打包到单个缓冲区中，代码中就应该包含一个循环，类似下面这样。

```
for (int i = 0; i < num_objects; i++) {
    glDrawArrays(GL_TRIANGLES,
                 object[i]->first_vertex,
                 object[i]->vertex_count);
}
```

这样可能会产生很多对 OpenGL 的调用，每一个调用都会伴随着一些系统开销。如果场景中有大量对象，每个对象都有一些相关的少量三角形，那么这些对 `glDrawArrays` 的调用中每一个的开销将会开始累积，从而对应用程序性能产生负面影响。在这种情况下，有两个函数可能会有所帮助，它们分别是

```
void glMultiDrawArrays(GLenum mode, GLint *first, GLsizei *count, GLsizei primcount);
```

和

```
void glMultiDrawElements(GLenum mode, GLsizei *count, GLenum type, GLvoid **indices,
                          GLsizei primcount);
```

这两个函数会对前面的代码进行相似的操作。它们的行为就像其 non-Multi 版本被调用 `primcount` 次一样。对于 `glMultiDrawArrays` 来说，`first` 和 `count` 都是数组。同样，对于 `glMultiDrawElements` 来说，`count` 和 `indices` 也都是数组。这就允许 OpenGL 可以一次性完成它的所有设置，一次性检查所有参数的正确性，并且如果驱动程序支持的话，就会发送单个命令到图形硬件。这样可以允许很多与调用 OpenGL 函数相关的系统开销以 `glMultiDraw` 函数所取代函数调用的次数进行分摊。

通过重写这个示例，我们可以看到只有到 `glMultiDrawArrays` 的一个函数调用可以用来取代这些对 `glDrawArrays` 的大量（可能是数千）调用。程序清单 12.12 显示了新版本的代码。虽然这里代码更多了，但是这里有一些到 OpenGL 的调用，而这些调用通常会带来更好的性能。

程序清单 12.12 简单的 `glMultiDrawArrays` 示例

```
// 这些数组假定足够大，足以容纳足够的数据来在场景中表现所有的对象
GLint first[];
GLsizei count[];

// 建立我们的第一个顶点的列表和顶点计数
for (int i = 0; i < num_objects; i++) {
    first[i] = object[i]->first_vertex;
    count[i] = object[i]->vertex_count;
}
```

```
// 现在调用一次 glDrawArrays
glMultiDrawArrays(GL_TRIANGLES, first, count, num_objects);
```

如果对象的列表不变（或者不经常改变），就可以预先创建 `first` 和 `count` 数组，并从示例中完全删除 `for` 循环。举例来说，如果有一个简单的游戏，在其中一关有一些敌人和一些奖励，那么我们可能只需要在玩家消灭一个敌人或收集一个奖励时更新 `first` 和 `count` 数组。

## 12.4.2 使用图元重启对几何图形进行组合

有很多工具可以用来对几何图形进行“条带化”。这些工具的思想就是通过接受“三角形汤”（triangle soup）——大批非连接的三角形，并且试图将它合并为一组三角形带，从而使性能得到提升。这样做是有效果的，因为独立的三角形每一个都会接受 3 个顶点才能进行表示，而一个三角形带则将三角形带中表示每个三角形所需的顶点数减少到 1 个（不包括三角形带中的第一个三角形）。通过将几何图形从三角形汤转换为三角形带，需要处理的几何图形数据更少了，而系统也会运行得更快。如果这种工具取得了很好的效果，并且产生了数量较少的长三角形带，每个三角形带包含许多三角形，那么基本上就可以说这种方法很奏效了。

对于这种类型的算法有很多研究，而一种新方法的成功与否是通过将一些众所周知的模型通过新的“条带生成器”来进行传递，并将由这种工具生成的条带数量和长度与目前最先进的条带生成器所生成的进行比较而进行判断的。

尽管已经进行了这些研究，但现实情况是，一个“三角形汤”可以通过单次调用 `glDrawArrays` 或 `glDrawElements` 进行渲染，但是只要还没有使用即将介绍的功能，那么一组三角形带的渲染就需要单独对 OpenGL 进行多次调用。这就意味着在一个使用条带化几何图形的程序中会有更多的函数调用，并且如果这个条带化应用程序没有获得较好的效果，或者如果模型没有很好地完成条带化，那么这可能抵消掉前面使用条带所获得的所有性能提升。甚至是 `glMultiDrawArrays` 和 `glMultiDrawElements` 这样的函数也不总能有所帮助，因为图形硬件可能不会直接实现这些函数，这样 OpenGL 基本上无论如何都必须将它们转换成多个 `glDrawArrays` 调用了。

图元重启（primitive restart）是几乎得到最新图形硬件普遍支持的特性，并且是 OpenGL 的一部分。图元重启应用在 `GL_TRIANGLE_STRIP`、`GL_TRIANGLE_FAN`、`GL_LINE_STRIP` 和 `GL_LINE_LOOP` 几何图形类型中。这种方法在一个条带（或者扇、环）结束和另一个开始时通知 OpenGL。要在几何图形中指出一个条带结束、下一个条带开始的位置，就要在元素数组中放置一个作为保留值的特殊标志。由于 OpenGL 或者从元素数组中获取顶点索引，或者在内部生成它们，在 `glDrawArrays` 这样的非索引绘制命令情况下，它会检查这个特殊索引值，并且在遇到它时结束当前条带并在下一个顶点开始一个新的条带。在默认情况下，这种模式是关闭的，但是可以通过调用

```
glEnable(GL_PRIMITIVE_RESTART);
```

开启，而通过调用

```
glDisable(GL_PRIMITIVE_RESTART);
```

关闭。

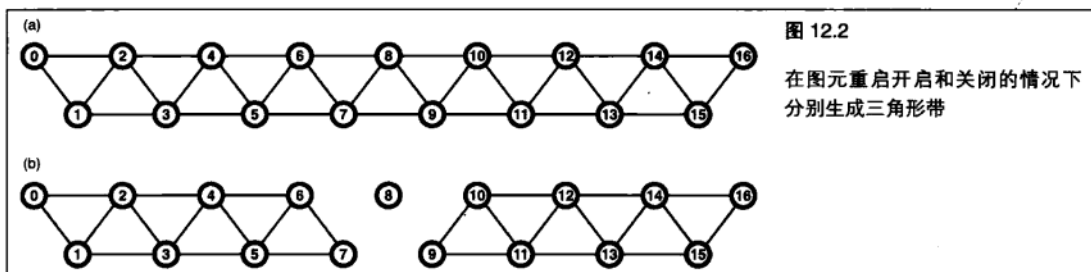
在图元重启模式开启时, OpenGL 会在获取或生成它们时开始关注它们, 并在遇到它时停止当前的条带并开始一个新的条带。要设置 OpenGL 应该关注的索引, 可以调用

```
glPrimitiveRestartIndex(index);
```

OpenGL 关注由 index 指定的值, 并使用它作为图元重启的标志。因为这个标志是一个顶点索引, 所以图元重启在诸如 `glDrawElements` 这样的索引绘制函数上能够得到最佳应用。举例来说, 我们仍然可以在 `glDrawArrays` 上使用图元重启。在这种情况下, OpenGL 可能最终会在内部生成重启索引, 而当这项工作完成时, 会重启图元。举例来说, 如果我们将重启索引设置为 10, 然后使用 `GL_TRIANGLE_STRIP` 模式绘制 20 个顶点, 那么我们将得到两个独立的条带。

图元重启索引的默认值为 0。因为一个真实的顶点的索引将会包含在模型中, 所以在使用图元重启模式时将重启索引设置为一个新值是个好主意。0xFFFFFFFF 是一个非常合适的值, 因为我们几乎可以确定它不会作为一个顶点的有效索引使用。很多条带化工具都可以选择创建独立的条带, 或者创建单个带有重启索引的条带。条带化工具可能会使用一个预定义索引, 或者在创建模型的条带化版本时输出它使用的索引 (例如一个比模型中的顶点数还要大的索引)。我们需要了解这一点, 并使用 `glPrimitiveRestartIndex` 函数对它进行设置, 以便在应用程序中使用这个工具的输出。

图元重启特性的图示如图 12.2 所示。



在图 12.2 中, 显示了一个三角形带, 其顶点由它们的索引表示。在 (a) 中, 这个三角形带由 17 个顶点组成, 它们在单个连接的三角形带中一共产生了 15 个三角形。通过开启图元重启模式并将图元重启索引设置为 8, 顶点 8 被 OpenGL 视为特殊的重启标志, 而三角形带则会在顶点 7 处结束, 如 (b) 所示。顶点 8 的实际位置将被忽略, 这是因为 OpenGL 不会将它视为一个真实顶点的索引。下一个进行处理的顶点 (顶点 9) 将成为新的三角形带的开始位置。所以, 在仍然向 OpenGL 传递 17 个顶点的情况下, 得到的结果将会是两个独立的三角形带, 其中一个绘制 8 个三角形, 而另一个则绘制 6 个三角形。

### 12.4.3 实例渲染

可能在很多情况下我们都会希望多次绘制同一个对象。

想象一个太空舰队, 或者一片草地。这时可能会有基本相同的几何图形集合的数千个副本, 在每个实例之间只有很小的改动。

在这种情况下, 一个简单的应用程序可能对草地中每一片独立的草进行循环, 并分别对它们进行绘制,

为每一棵草调用一次 `glDrawArrays`，并且可能在每一次迭代中更新一组着色器统一值。假定每一棵草都是由一个包含 4 个三角形的三角形带组成的，那么代码可能与程序清单 12.13 所示类似。

程序清单 12.13 多次绘制同一个几何图形

```
glBindVertexArray(grass_vao);
for (int n = 0; n < number_of_blades_of_grass; n++) {
    SetupGrassBladeParameters();
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 6);
}
```

草地一共有多少棵草？`number_of_blades_of_grass` 的值是多少？可能是几千，也可能是几百万。每一棵草都应该占据屏幕上很小的一块区域，而显示草的顶点数量也会非常少。我们的图形加速卡并不是真的要做很多工作来渲染单独的一片草，而系统则很可能会将它的大部分时间用在向 OpenGL 发送命令上，而不是用在实际进行绘制上。OpenGL 通过实例渲染（instanced rendering）来对此进行处理，这是一种绘制同一个几何图形的大量副本的方法。

实例渲染是 OpenGL 提供了一种指定通过单次函数调用来绘制同一个几何图形的多个副本的方法。这种功能是通过实例渲染函数来实现的，例如

```
void glDrawArraysInstanced(GLenum mode, GLint first, GLsizei count, GLsizei primcount);
```

和

```
void glDrawElementsInstanced(GLenum mode, GLsizei count, GLenum type, const void *
indices, GLsizei primcount);
```

这两个函数的行为很像 `glDrawArrays` 和 `glDrawElements`，除了它们是告诉 OpenGL 来渲染几何图形的 `primcount` 个副本以外。在这些函数的常规、非实例版本中，每个函数最前面的参数（对于 `glDrawArraysInstanced` 来说是 `mode`、`first` 和 `count`，而对于 `glDrawElementsInstanced` 来说则是 `indices`）都代表同样的含义。当调用这些函数中的一个时，OpenGL 会做好任何需要的准备工作，来对几何图形（例如将顶点数据复制到图形加速卡内存中等）只进行一次绘制，然后对同样的顶点进行多次绘制。

如果这些函数所做的所有工作就只是将同样顶点的多个副本发送到 OpenGL，就像 `glDrawArrays` 和 `glDrawElements` 已经在一个紧凑的循环中被调用一样，那么它们就不会非常有用。使实例渲染可用并且非常强大的因素之一，是 GLSL 中的一种特殊的内建变量，叫做 `gl_InstanceID`。`gl_InstanceID` 变量就好像一个整型统一值一样出现在 GLSL 中。当顶点的第一个副本被发送到 OpenGL 时，`gl_InstanceID` 为 0。随后，它将为几何图形的每一个副本递增一次，最终达到 `primcount - 1`。因为 `gl_InstanceID` 是一个整数，这里有一个实际的上限值，即我们可以通过调用 `glDrawArraysInstanced` 或 `glDrawElementsInstanced` 一次渲染 200 万个实例，但是这对于绝大多数应用程序来说都应该足够了。如果需要渲染几何图形的超过 200 万个副本，那么应用程序就很可能运行得非常缓慢，而我们将不会看到一个由于将渲染分解成包含 10 亿个顶点的块而导致的巨大性能损失。

`glDrawArraysInstanced` 函数所进行的操作基本上就像执行了程序清单 12.14 中列出的代码一样。

程序清单 12.14 描述 `glDrawArraysInstanced` 的行为的伪代码

```
// 在所有（即 primcount 个）这些实例上执行循环
for (int n = 0; n < primcount; n++) {
    // 设置 gl_InstanceID Uniform——在这里 gl_InstanceID 是一个 C 变量，它保存“虚拟”gl_InstanceID
    Uniform 的位置
    glUniformli(gl_InstanceID, n);
```

```
// 现在, 当我们调用 glDrawArrays 时, 着色器中的 gl_InstanceID 变量将会包含这个被渲染的实例的索引  
glDrawArrays(mode, first, count);  
}
```

类似地, `glDrawElementsInstanced` 函数的操作与程序清单 12.15 中列出的代码类似。

程序清单 12.15 描述 `glDrawElementsInstanced` 的行为的伪代码

```
for (int n = 0; n < primcount; n++) {  
    // 设置 gl_InstanceID 的值  
    glUniform1i(gl_InstanceID, n);  
    // 对 glDrawElements 进行一次常规调用  
    glDrawElements(mode, count, type, indices);  
}
```

当然, `gl_InstanceID` 并不是一个真正的统一值, 我们也不能通过调用 `glGetUniformLocation` 为它分配一个位置。`gl_InstanceID` 的值由 OpenGL 进行管理, 并且由硬件生成的可能性非常高, 也就是说在性能方面基本上可以自由使用。实例渲染的强大来自于对这个变量和实例数组 (instanced arrays, 稍候将进行解释) 的富有想象力的应用。

`gl_InstanceID` 的值可以直接作为着色器函数的一个参数使用, 或者用于诸如纹理或统一数组这样的数据的索引。回到我们的草地示例, 让我们来搞清楚要怎样利用 `gl_InstanceID` 来使我们的草地变得不是只在一个点上长出几千棵同样的草。我们的每颗草都是由包含 4 个三角形的小三角形带组成的, 每个三角形带中有 6 个顶点。让它们都变得各不相同可能会非常麻烦。但是, 通过一些着色器技巧, 我们可以使每一棵草看起来都有足够的不同, 这样就能生成一个有趣的输出了。在这里, 我们就不完整地列出这个着色器的代码了 (上一章中有大量更高级的着色器示例), 但是我们会了解几个如何使用 `gl_InstanceID` 向场景中添加变化的办法。

首先, 我们需要每一棵草都有不同的位置; 否则, 它们就都会绘制在另外一棵上面。让我们以差不多均匀的方式来对这些草进行排列。如果要进行渲染的草的数量是 2 的幂, 那么可以使用 `gl_InstanceID` 的一半位数来表示一棵草的  $x$  坐标, 而  $y$  坐标则用来表示  $z$  坐标 (草地是在一个  $x$ - $z$  平面上, 而  $y$  为高度)。在本例中, 我们渲染了  $2^{20}$  个, 或者说略多于 100 万棵草 (实际上是 1 048 576 颗草, 但谁会去数呢? )。通过使用最低的 10 位 (第 9 位到第 0 位) 作为  $x$  坐标, 而使用最高的 10 位 (第 19 位到第 10 位) 作为  $z$  坐标, 我们就有了一个草地的统一网格。下面让我们看一下图 12.3 所示图像, 来看看到目前为止的成果。

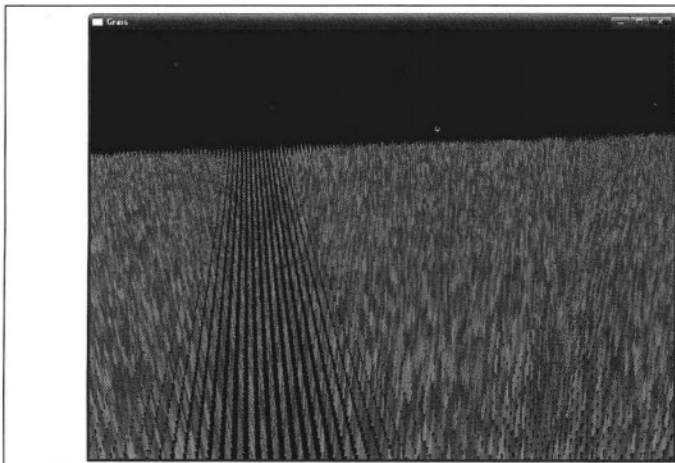


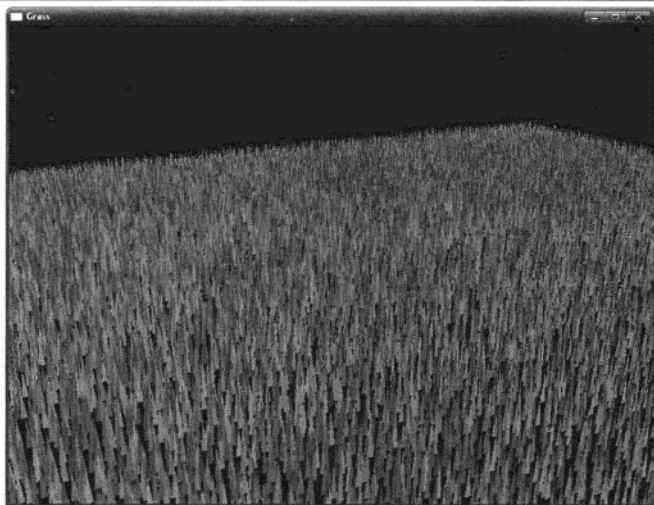
图 12.3

实例草地的第一次尝试

统一草地网格看起来有点太清晰了，就好像一个特别用心的管理员手工栽种了每一棵草一样。我们真正要做的是将这些草在网格方框中进行一定随机量的移位，这样就可以使这片草地看起来不那么呆板了。生成随机值的一种简单方法是，用一个种子值乘以一个很大的数，然后取生成结果的所有位的子集，并用它作为一个函数的输入。在这里，我们并不追求一个完美的分布，所以这种简单的生成器足可以胜任。通常，在使用这种算法时，我们可以重复使用这个种子值，作为随机数生成器下一次迭代的输入。但是，在这种情况下，在一个伪随机序列（pseudo-random sequence）中的 `gl_InstanceID` 之后，我们真正生成接下来的几个数字时，可以直接使用 `gl_InstanceID`。在对我们的伪随机函数进行仅仅两次迭代之后，我们就能得到一个合理的随机分布了。因为我们需要在  $x$  和  $y$  方向上都进行移位，所以从 `gl_InstanceID` 生成两个连续的随机数字，并使用它们对平面上的草进行移位。图 12.4 所示显示了目前获得的效果。

图 12.4

进行轻微扰动的草地



到现在为止，草地是均匀分布的，其中每一棵草的位置都进行了随机的扰动。但是，所有的草看起来都是相同的。（实际上，我们使用同样的随机数生成器来为每颗草分配了稍微有所不同的颜色，以使它们能够显示出各自的形状。）我们可以在这片草地上应用一些变化，来使每一棵草看起来稍微有所不同。这可能是我们希望进行控制的，所以使用一个纹理来保存关于这些草的信息。

对于每一棵草来说，我们都有一个  $x$  坐标和一个  $z$  坐标，这是通过从 `gl_InstanceID` 直接生成一个网格坐标，然后生成一个随机数字并对  $x$ - $z$  平面中的草进行移位而得到的。这个坐标对可以作为一个坐标，来查询一个 2D 纹理中的纹理单元，而我们可以其中放置任何希望放置的东西。让我们使用这种纹理来控制草的长度，可以在这个纹理中（使用红色通道）放置一个长度参数，然后将草的几何图形每个顶点的  $y$  坐标乘以这个参数生成更长或更短的草。在纹理中 0 值将会产生非常短的（或者不存在的）草，而为 1 的值则会产生最大长度的草。现在，我们可以设计一个纹理，其中每个纹理单元表示草地中某个区域草的长度。为什么不绘制几个麦田圈呢？这个纹理可以采用 `GL_LINEAR` 采样进行采样，甚至可以使用 Mip 贴图。

现在，这些草在这片草地上均匀地分布着，并且我们在不同的区域对草的长度进行了控制。但是，这些草仍然是相互进行缩放的副本。我们还可以引入一些更多的变化。接下来，让每棵草围绕它的轴根据纹理中的其他参数进行旋转。我们使用纹理的绿色通道存储角度，这棵草就应该沿着  $y$  轴旋转这样的角度，其中 0 代表没有进行旋转，而 1 则代表旋转了  $360^\circ$ 。我们仍然只在顶点着色器中进行一次纹理获取，而

`gl_InstanceID` 仍然是这个着色器的唯一输入。到这里事情就开始有眉目了，来看一看图 12.5 所示图像。

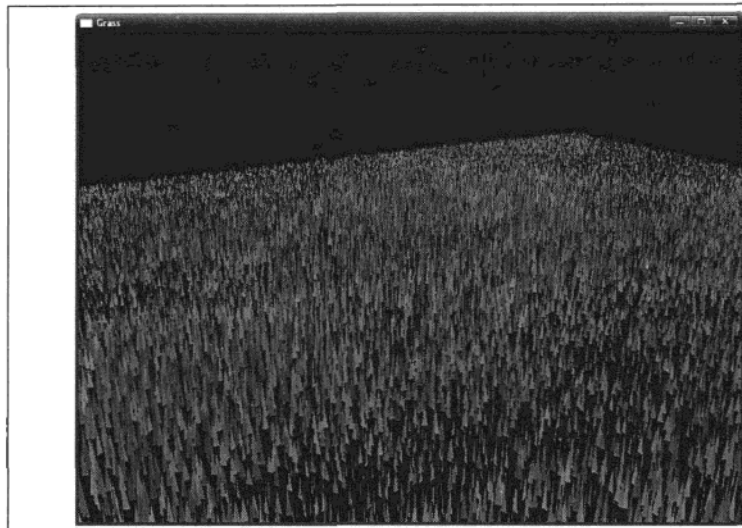


图 12.5

控制草的长度和方向

草地看起来还是有些乏味。这些草只是直立在那里不动。真正的草是在风中摇摆的，而当有东西滚过时则会被压平。我们需要这些草进行弯曲，并且要对此进行控制。为什么不使用参数纹理的另外一个通道（蓝色通道）来控制一个弯曲因子呢？我们可以使用它作为另外一个角度值，并且在我们应用绿色通道中的旋转之前将草围绕  $x$  轴进行旋转。这样就使我们能够根据纹理中的参数对草进行弯曲了。使用 0 来表示没有进行任何弯曲（草是直立的），而使用 1 来表示完全平躺的草。通常情况下，这些草应该是轻轻摆动的，所以这个参数应该是一个比较小的值。当这些草平躺时，这个值会大得多。

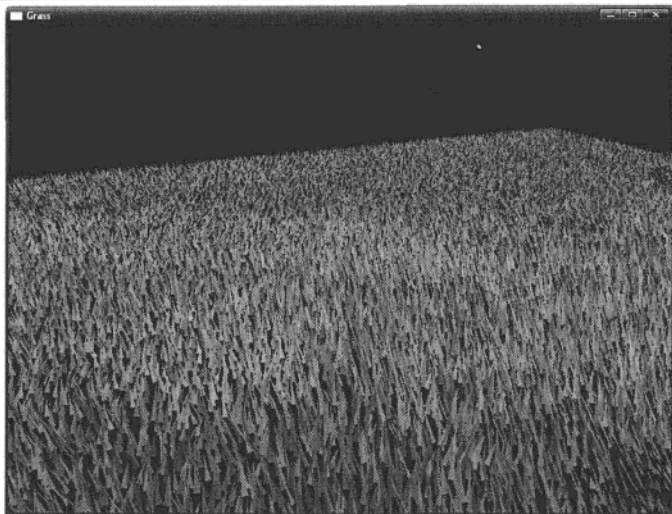
最后，我们可以控制草的颜色。在一个很大的纹理中只存储草的颜色，这看起来似乎是符合逻辑的。举例来说，如果我们要绘制的是一个有很多线、标志或广告的运动场，那么这样做可能是个好主意，但是在本例中，如果这些草都是深浅不一的绿色，那么这样做是相当浪费的。为了取代这种方式，让我们在一个 1D 纹理中为草地设置一个调色板，并使用参数纹理中最后的通道（alpha 通道）存储到这个调色板的索引。这个调色板的一端可以从一种毫无生气的枯草黄色开始，直到另一端的一种繁茂的深绿色为止。现在，从参数纹理中和所有其他参数一起读取 alpha 通道，并用它对 1D 纹理进行索引——这是一种依赖性的纹理获取。最终的草地结果如图 12.6 所示（同样显示在彩插中的彩图 21 中）。

现在，最终的草地包含数百万棵草，它们是均匀分布的，并且由应用程序控制长度、“平度”、弯曲方向，或者摇摆和颜色。请记住，着色器的唯一输入是 `gl_InstanceID`，它将每一棵草与其他草区分开来，发送到 OpenGL 的几何图形总共只有 6 个点，而绘制草地中所有这些草所需要的代码只调用一次 `glDrawArraysInstanced`。

参数纹理可以使用线性纹理进行读取，以提供草地区域之间的平滑过渡，而分辨率则可以非常低。如果我们希望草地在风中摇摆，或者像是被一群动物踩过一样，可以通过对每一帧或每两帧进行更新，并在对草进行渲染之前上传一个新版本，使其产生动画效果。因为 `gl_InstanceID` 被用来生成随机数字，在将它传递到随机数字生成器之前向它添加一个偏置，就可以使用同一个着色器生成一个不同但却预先定义好的“随机”草地块了。

图 12.6

最终的草地结果



#### 12.4.4 自动获得数据

当调用 `glDrawArraysInstanced` 或 `glDrawElementsInstanced` 时，着色器中的内建变量 `gl_InstanceID` 将用来告诉正在处理的是哪一个实例，而对于进行渲染的每一个几何图形新实例，都会累加 1。甚至是在没有使用任何实例绘制函数时，它实际上也是可用的——在这种情况下它只是为 0 而已。这就意味着我们可以在实例化渲染和非实例化渲染中使用同一个着色器。

我们可以使用 `gl_InstanceID` 对长度与进行渲染的实例数相同的数组进行索引。举例来说，我们可以使用它来查询一个纹理中的纹理单元，或者对一个统一数组进行索引。然而实际上，我们要做的是将数组看作一个“实例化的属性”。这就是说，为进行渲染的每一个实例读取这个属性的新值。OpenGL 能够使用一个称为“实例化数组”（instanced array）的特性自动将这个数据送入着色器中。为了使用实例化数组，我们要像通常一样为着色器声明一个输入。这个输入属性将会有有一个索引，我们可以在调用类似 `glVertexAttribPointer` 这样的函数时使用这个索引。通常情况下，每一个顶点的顶点属性都会被读取，而一个新的值将会送入着色器。但是，为了使 OpenGL 为每个实例都从这个数组中读取属性，我们可以调用

```
void glVertexAttribDivisor(GLuint index, GLuint divisor);
```

将这个属性的索引传递到这个函数的 `index`，并将 `divisor` 设为我们想要在每次从数组中读取新值之间进行传递的实例数。如果 `divisor` 为 0，那么这个数组就成为一个常规顶点属性，为每个顶点读取一个新值。但是，如果 `divisor` 是非 0 的，那么就会每隔几个实例从这个数组中读取一个新数据。举例来说，如果将 `divisor` 设为 1，那么我们就对每个实例都从这个数组中获得一个新值；如果将 `divisor` 设为 2，那么我们就对每两个实例都从这个数组中获得一个新值，依次类推。我们可以对 `divisor` 进行混合搭配，为每个属性设置不同的值。

使用这项功能的一个例子就是在我们想要绘制一组颜色不同的对象时。考虑程序清单 12.16 中的简单顶点着色器。

程序清单 12.16 各顶点颜色不同的简单顶点着色器

```
#version 150

precision highp float;

in vec4 position;
in vec4 color;

out Fragment
{
    vec4 color;
} fragment;

uniform mat4 mvp;

void main(void)
{
    gl_Position = mvp * position;
    fragment.color = color;
}
```

通常情况下, color 属性会为每个顶点读取一次, 这样每个顶点的颜色最终都会各不相同。应用程序会提供一个颜色数组, 颜色与模型中的顶点数相同。对象的每个实例也不可能都有不同的颜色, 因为着色器并不知道任何关于实例化的信息。如果调用

```
glVertexAttribDivisor(index_of_color, 1);
```

我们可以将 color 设为一个实例数组, 这里的 index\_of\_color 是 color 属性绑定到的槽的索引。

现在, 每个实例都将从顶点数组中获取 color 的一个新值。特定实例中的每一个顶点将会接受同一个值作为颜色值, 结果是对象的每个实例都将渲染成不同的颜色。为 color 保存数据的顶点数组的大小只要和我们想要渲染的索引数相同就可以了。如果我们增加 divisor 的值, 那么从数组中读取新值的频率就会越来越低。如果 divisor 为 2, 那么每两个实例就会出现一个新的颜色值; 如果 divisor 为 3, 那么颜色将会每 3 个实例更新一次, 以此类推。

如果我们使用这个简单的着色器对几何图形进行渲染, 那么每个实例都将绘制在其他实例上。我们需要修改每个实例的位置, 以便能够看到它们。为此, 我们可以使用另一个实例。程序清单 12.17 所示显示了对程序清单 12.16 中的顶点着色器进行简单修改后的结果。

程序清单 12.17 简单的实例顶点着色器

```
#version 150

precision highp float;

in vec4 position;
in vec4 instance_color;
in vec4 instance_position;

out Fragment
{
    vec4 color;
} fragment;

uniform mat4 mvp;

void main(void)
```

```

{
    gl_Position = mvp * (position + instance_position);
    fragment.color = instance_color;
}

```

现在,我们不但有了逐个顶点的位置,同时也有了逐个实例的位置。在与模型视图投影矩阵相乘之前,我们可以将它们在顶点着色器中相加到一起。

我们可以通过再次调用

```
glVertexAttribDivisor(index_of_instance_position, 1);
```

将 instance\_position 输入属性设置为一个实例数组。

和前面类似,这里的 index\_of\_instance\_position 是 instance\_position 属性绑定到的位置的索引。任何类型的输入属性都可以通过使用 glVertexAttribDivisor 设置为实例。这个例子非常简单,只使用了一次平移(这个值保存在 instance\_position 中)。一个更加高级的应用程序可以使用矩阵顶点属性,或者将一些平移矩阵包装到统一值中,并向实例数组中传递矩阵加权值。应用程序可以用这种方式渲染一个由士兵组成的军队,每个士兵都有不同的姿势;或者渲染一个星际舰队,每艘太空船都向不同的方向飞行。

现在,让我们将这个简单的着色器连接到真正的程序中。首先,加载着色器,并且在连接程序之前像通常一样设置属性位置,如程序清单 12.18 所示。

程序清单 12.18 设置实例属性

```

instancingProg = gltLoadShaderPair("instancing.vs", "instancing.fs");
glBindAttribLocation(instancingProg, 0, "position");
glBindAttribLocation(instancingProg, 1, "instance_color");
glBindAttribLocation(instancingProg, 2, "instance_position");
glLinkProgram(instancingProg);

```

在程序清单 12.19 中,我们声明一些数据,并将它们加载到一个顶点缓冲区(绑定到一个顶点数组对象)中。

程序清单 12.19 为实例渲染做好准备

```

static const GLfloat square_vertices[] =
{
    -1.0f, -1.0f, 0.0f, 1.0f,
    1.0f, -1.0f, 0.0f, 1.0f,
    1.0f, 1.0f, 0.0f, 1.0f,
    -1.0f, 1.0f, 0.0f, 1.0f
};

static const GLfloat instance_colors[] =
{
    1.0f, 0.0f, 0.0f, 1.0f,
    0.0f, 1.0f, 0.0f, 1.0f,
    0.0f, 0.0f, 1.0f, 1.0f,
    1.0f, 1.0f, 0.0f, 1.0f
};

static const GLfloat instance_positions[] =
{
    -2.0f, -2.0f, 0.0f, 0.0f,
    2.0f, -2.0f, 0.0f, 0.0f,
    2.0f, 2.0f, 0.0f, 0.0f,
    -2.0f, 2.0f, 0.0f, 0.0f
}

```

```
};

GLuint offset = 0;

glGenVertexArrays(1, &square_vao);
glGenBuffers(1, &square_vbo);
glBindVertexArray(square_vao);
glBindBuffer(GL_ARRAY_BUFFER, square_vbo);
glBufferData(GL_ARRAY_BUFFER,
             sizeof(square_vertices) +
             sizeof(instance_colors) +
             sizeof(instance_positions), NULL, GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, offset,
               sizeof(square_vertices),
               square_vertices);
offset += sizeof(square_vertices);
glBufferSubData(GL_ARRAY_BUFFER, offset,
               sizeof(instance_colors), instance_colors);
offset += sizeof(instance_colors);
glBufferSubData(GL_ARRAY_BUFFER, offset,
               sizeof(instance_positions), instance_positions);
offset += sizeof(instance_positions);

glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0,
                    (GLvoid *)sizeof(square_vertices));
glVertexAttribPointer(2, 4, GL_FLOAT, GL_FALSE, 0,
                    (GLvoid *) (sizeof(square_vertices) +
                                sizeof(instance_colors)));

glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
glEnableVertexAttribArray(2);
```

现在要做的事情就只剩下为 `instance_color` 和 `instance_position` 属性数组设置顶点属性公约数 (divisor) 了。

```
glVertexAttribDivisor(1, 1);
glVertexAttribDivisor(2, 1);
```

现在我们绘制放在顶点着色器中的几何图形的 4 个实例。每个实例包含 4 个顶点，每个顶点都有自己的位置。每个实例中的同一个顶点都有相同的位置。但是，同一个实例中所有顶点都会对应同一个 `instance_color` 值和 `instance_position` 值，而这两个参数对于每个实例都会分配一个新值。我们的渲染循环和下面代码类似。

```
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
glClear(GL_COLOR_BUFFER_BIT);

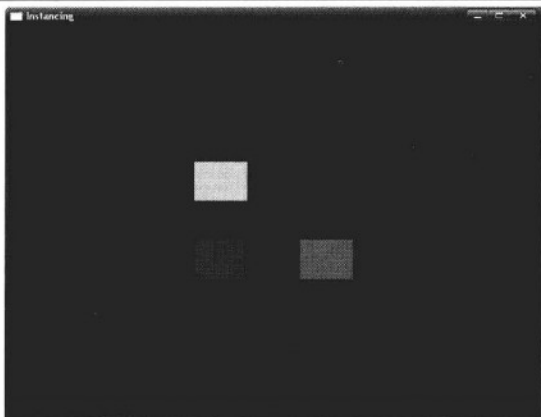
glUseProgram(instancingProg);
glBindVertexArray(square_vao);
glDrawArraysInstanced(GL_TRIANGLE_FAN, 0, 4, 4);
```

得到的结果如图 12.7 所示。

在图中，我们可以看到已经渲染了 4 个矩形。这些矩形中的每一个都有不同的位置，并且每一个的颜色也都不同。这种方式可以扩展到数千个甚至数百万个实例，而现代图形硬件应该能够对此处理，而不会产生任何问题。

图 12.7

实例渲染的结果



## 12.5 存储变换的顶点——变换反馈

在 OpenGL 中，我们可以将顶点着色器或者几何着色器的结果存储到一个缓冲区对象中。这种特性就是变换反馈（transform feedback）。在使用变换反馈时，从顶点着色器或几何着色器输出的一个特定属性集就会被写入到缓冲区中。在不存在任何几何着色器时（我们应该还记得，几何着色器是可选的），这些数据来自顶点着色器。当出现一个几何着色器时，几何着色器所产生的顶点就会被记录下来。用来捕获顶点着色器和几何着色器输出的缓冲区就称为变换反馈缓冲区（transform feedback buffer）。一旦数据被放置到一个使用变换反馈的缓冲区中，就能够使用一个类似 `glGetBufferSubData` 这样的函数将它读取回来，或者通过使用 `glMapBuffer` 将它映射到应用程序地址空间从而直接对它进行读取，它也可以用作后续绘制命令的数据源。

### 12.5.1 变换反馈

变换反馈是 OpenGL 的一种特殊模式，它允许将一个顶点着色器或几何着色器的结果保存到一个缓冲区中。一旦这些信息出现在这个缓冲区中，就可以作为更多绘制命令的一个顶点数据源使用了。顶点着色器或几何着色器的任意输出属性都可以被存储到这个缓冲区中。但是，我们不能同时对顶点着色器和几何着色器的输出进行记录。如果几何着色器是活动的，那么只有几何着色器的输出可以访问。如果我们需要来自顶点着色器的原始数据，就需要将这些数据不加修改地通过几何着色器进行传递。

变换反馈位置的图示如图 12.8 所示。

正如我们所看到的，变换反馈缓冲区位于几何图形着色输出和顶点装配阶段之间。既然几何着色器是可选的，那么如果它不存在的话，这些数据实际上就来自顶点着色器——这是通过虚线表示的。

虽然这个图解展示了向顶点装配阶段提供数据的变换反馈缓冲区，但是它只对所创建的反馈循环（因此出现了术语“变换反馈”）进行了图示。虽然 OpenGL 允许我们将同一个缓冲区同时作为变换反馈缓冲

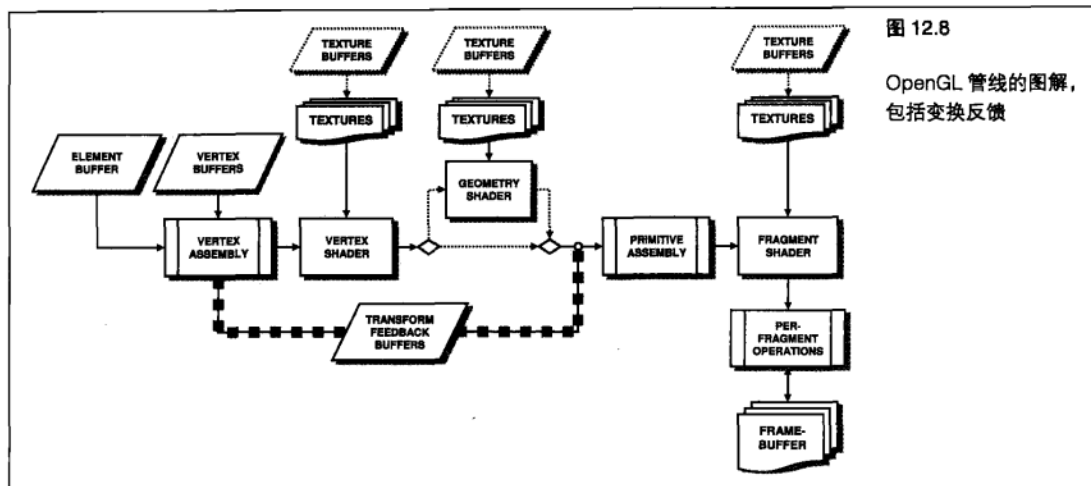


图 12.8

OpenGL 管线的图解，  
包括变换反馈

区和顶点缓冲区，但是如果这样做的话，得到的结果将是未定义的，而我们肯定无法得到想要的结果。将要在变换反馈模式中被记录的顶点属性或 varying 变量的集合，是使用

```
void glTransformFeedbackVaryings(GLuint program, GLsizei count, const GLchar **
varyings, GLenum bufferMode);
```

指定的。

glTransformFeedbackVaryings 的第一个参数是程序对象的名称。变换反馈变化状态是逐个程序对象进行维护的。这就意味着不同的程序可以存储不同的顶点属性组，即使在它们之中使用的是相同的顶点着色器或几何着色器时也是如此。第二个参数是要记录的 varying 变量的数量，同时也是第 3 个参数中给出地址的数组的长度。第 3 个参数只是一个由 C 语言风格的字符串组成的数组，它给出了要记录的 varying 的名称。它们是顶点着色器或几何着色器中的 out 变量的名称。最后一个参数则指定这些 varying 变量的存储模式。这个参数必须为 GL\_SEPARATE\_ATTRIBS 或 GL\_INTERLEAVED\_ATTRIBS。如果 bufferMode 是 GL\_INTERLEAVED\_ATTRIBS 的话，那么这些 varying 变量将被一个接一个地记录到单独的缓冲区中。而如果 bufferMode 是 GL\_SEPARATE\_ATTRIBS 的话，那么这些 varying 变量将被记录到每个 varying 变量的缓冲区中。

让我们来考虑下面的代码段，它声明了输出的 varying 变量。

```
out vec4 vs_position_out;
out vec4 vs_color_out;
out vec3 vs_normal_out;
out vec3 vs_binormal_out;
out vec3 vs_tangent_out;
```

为了指定 vs\_position\_out、vs\_color\_out 等 varying 变量应该被写入到单个交叉存取变换反馈缓冲区中，我们可以在应用程序中使用如下代码。

```
static const char * varying_names[] =
{
    "vs_position_out",
    "vs_color_out",
    "vs_normal_out",
    "vs_binormal_out",
    "vs_tangent_out",
};
```

```

    "vs_binormal_out",
    "vs_tangent_out"
};
glTransformFeedbackVaryings(program, 5, varying_names,
                             GL_INTERLEAVED_ATTRIBS);

```

并不是顶点着色器或几何着色器的所有输出都需要被存储到变换反馈缓冲区中。我们可以将顶点着色器输出的一个子集保存到变换反馈缓冲区中，并将更多的输出发送到片段缓冲区进行插值。

类似地，也可以将顶点着色器的一些输出保存到一个没有被片段着色器使用的变换反馈缓冲区中。因此，可能已经被看作非活动的（因为它们没有被片段着色器使用）顶点着色器的输出，有可能由于它们被存储在一个变换反馈缓冲区中而变成活动的。这样，在通过调用 `glTransformFeedbackVaryings` 来指定一组新的变换反馈 `varying` 变量之后，有必要使用

```
glLinkProgram(program);
```

对程序对象进行连接。

一旦这些变换反馈 `varying` 变量被指定，并且程序被连接，那么它就可以像通常一样使用了。在实际捕获任何东西之前，我们需要将一个缓冲区对象进行绑定，以将其作为变换反馈缓冲区。在将变换反馈模式指定为 `GL_INTERLEAVED_ATTRIBS` 之后，所有存储的顶点属性都会一个接一个地写入单个缓冲区中。我们通过调用

```
glBindBuffer(GL_TRANSFORM_FEEDBACK_BUFFER, buffer);
```

指定这个缓冲区。

在这里，`GL_TRANSFORM_FEEDBACK_BUFFER` 会告诉 OpenGL，我们想要将一个用来存储顶点着色器或几何着色器结果的缓冲区绑定到 `GL_TRANSFORM_FEEDBACK_BUFFER` 绑定点。第 2 个参数是前面通过调用 `glGenBuffers` 创建的缓冲区对象的名称。

任何数据都必须先在缓冲区中为其分配空间，然后才能写入缓冲区。我们通过调用

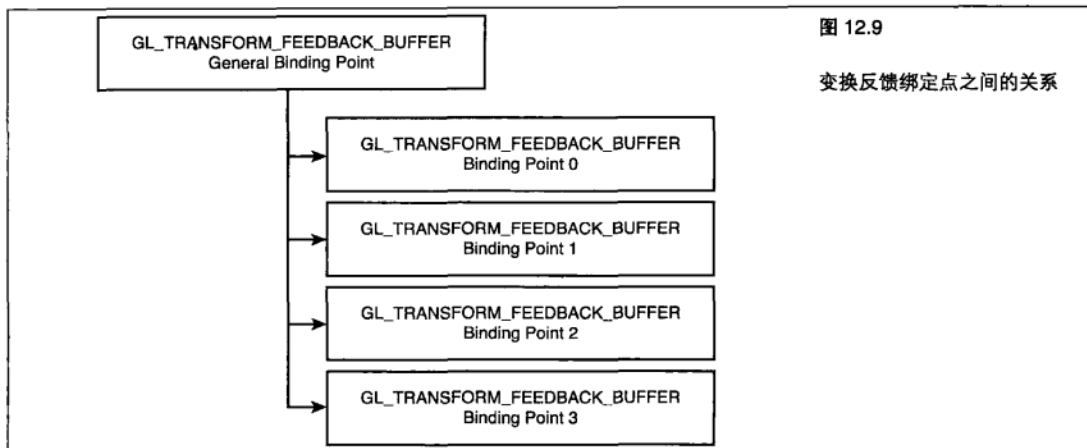
```
glBufferData(GL_TRANSFORM_FEEDBACK_BUFFER, size, NULL, GL_DYNAMIC_COPY);
```

来分配空间而不指定数据。

其中第一个参数是将其为其分配空间的缓冲区。为了达到绑定一个缓冲区并为其分配空间的目的，我们可以使用任何想要使用的缓冲区绑定。但是，OpenGL 可能会根据第一个进行绑定的绑定点来推断这个缓冲区将要用来做什么，这样一来，尤其当这是一个新缓冲区时，`GL_TRANSFORM_FEEDBACK_BUFFER` 绑定点将是一个好的选择。`size` 参数指定我们想要分配多少空间，以字节为单位。这取决于应用程序的需要，但在变换反馈时，如果将要放入缓冲区的数据生成得太多，那么超出的部分将会被丢弃。`NULL` 会告诉 OpenGL 还没有给出任何数据，我们只是想要分配空间以备以后使用。最后一个参数 `usage` 会提示 OpenGL，我们将要用这个缓冲区做什么。

`usage` 有很多可能的值，但是对于一个变换反馈缓冲区来说，`GL_DYNAMIC_COPY` 可能是一个很好的选择。`DYNAMIC` 部分告诉 OpenGL，这些数据很可能会经常改变，但是在每次更新之间很可能只使用很少的几次。`COPY` 部分说明我们想要通过 OpenGL 功能（例如变换反馈）来对缓冲区中的数据进行更新，然后将这些数据返回 OpenGL，以供其他操作（例如绘制操作）使用。前面第 8 章中介绍了更多关

于缓冲区使用的信息。要指定变换反馈数据将要写入到哪个缓冲区，我们需要将一个缓冲区绑定到一个索引变换反馈绑定点上。实际上有多个 `GL_TRANSFORM_FEEDBACK_BUFFER` 绑定点可以达到这个目的，它们在概念上是独立的，但是都与通用 `GL_TRANSFORM_FEEDBACK_BUFFER` 绑定点相关。图 12.9 所示显示了相关图示。



我们通过调用

```
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, index, buffer);
```

将一个缓冲区绑定到任意索引绑定点。

和前面一样，`GL_TRANSFORM_FEEDBACK_BUFFER` 告诉 OpenGL 我们要绑定一个缓冲区对象来存储变换反馈的结果，而最后一个参数 `buffer` 则是将要绑定到的缓冲区对象的名称。附加参数 `index` 是 `GL_TRANSFORM_FEEDBACK_BUFFER` 绑定点的索引。有一件重要的事情需要注意，就是无法直接对任何由 `glBindBufferBase` 通过类似 `glBufferData` 或 `glCopyBuffer` 这样的函数提供的额外绑定直接进行寻址。

但是，当调用 `glBindBufferBase` 时，它实际上会将这个缓冲区绑定到索引绑定点和通用绑定点。这样，如果在调用 `glBindBufferBase` 之后就访问通用绑定点，我们可以使用额外的绑定点在缓冲区中分配空间。

`glBindBufferBase` 还有一个稍为高级一些的版本，即 `glBindBufferRange`，其原型为：

```
void glBindBufferRange(GLenum target, GLuint index, GLuint buffer, GLintptr offset, GLsizeiptr size);
```

尽管 `glBindBuffer` 和 `glBindBufferBase` 只能将整个缓冲区同时绑定到一个绑定点，但是 `glBindBufferRange` 函数则允许我们将一个缓冲区的一部分绑定到一个索引绑定点。前 3 个参数 (`target`、`index` 和 `buffer`) 与 `glBindBufferBase` 中的相应参数含义相同。`offset` 和 `size` 参数分别用来指定缓冲区中我们想要进行绑定部分的开始位置和长度。我们可以同时将同一个缓冲区中的不同部分绑定到几个不同的索引绑定点。这样就使我们可以使用 `GL_SEPARATE_ATTRIBS` 模式中使用变换反馈将输出顶点的每个属性写入单个缓冲区的独立部分中。如果应用程序将所有属性包装到单个顶点缓冲区中，并使用

glVertexAttribPointer 指定缓冲区中的非 0 偏置，就使我们能够将变换反馈的输出与顶点着色器的输入进行匹配了。

如果我们指定所有属性都应该通过在 glTransformFeedbackVaryings 中使用 GL\_INTERLEAVED\_ATTRIBS 参数记录到单个变换反馈缓冲区中，那么这些数据将会被写入绑定到第一个 GL\_TRANSFORM\_FEEDBACK\_BUFFER 绑定点（index 为 0）的缓冲区中。但是，如果我们指定变换反馈模式为 GL\_SEPARATE\_ATTRIBS，那么顶点着色器的每个输出都将被记录到自己的独立缓冲区（或者是缓冲区中的一个部分，如果我们使用 glBindBufferRange 的话）中。在这种情况下，我们需要将多个缓冲区或缓冲区部分进行绑定，以将其作为变换反馈缓冲区。参数必须在 0 和“一个使用变换反馈模式能够记录到独立缓冲区中的 varying 变量最大数量减去 1 所得到的值”之间。这个限制取决于图形硬件和驱动程序，并且可以通过调用以 GL\_MAX\_TRANSFORM\_FEEDBACK\_SEPARATE\_ATTRIBS 为参数的 glGetIntegerv 来查询。

这个限制也应用在 glTransformFeedbackVaryings 的 count 参数上。在 GL\_INTERLEAVED\_ATTRIBS 模式下，对于能够写入变换反馈缓冲区的独立 varying 变量的数量并没有上限，但是对于能够写入一个缓冲区的分量数量则存在一个最大值。举例来说，在使用变换反馈时，可以写入的 vec3s 要比可以写入的 vec4s 更多。这个限制还是取决于图形硬件，并且可以通过调用以 GL\_MAX\_TRANSFORM\_FEEDBACK\_INTERLEAVED\_COMPONENTS 为参数的 glGetIntegerv 来查询。

在将另一组属性写入到另一个缓冲区中的同时，将一组输出 varying 变量以交叉存取的方式写入一个缓冲区是不可能的。当变换反馈激活时，输出 varying 变量要么都以交叉反馈的形式存储到一个缓冲区，要么进行包装后存储到几个不同的缓冲区或不同的缓冲区部分中。这样，如果我们想要使用变换反馈为后续传递生成顶点数据，那么我们就需要在计划输入顶点布局时考虑这一点。总体上说，比起通过变换反馈进行写入的方式，顶点着色器在读取顶点数据的方式上更加灵活。

一旦要接受变换反馈结果的缓冲区被绑定，变换反馈模式就会通过调用

```
void glBeginTransformFeedback(GLenum primitiveMode);
```

来激活。

现在，顶点无论何时通过一个顶点着色器或几何着色器，后续着色器的输出 varying 变量都将写入到变换反馈缓冲区。函数的 primitiveMode 参数告诉 OpenGL，几何图形将会是什么类型的。可接受的参数有 GL\_POINTS、GL\_LINES 或 GL\_TRIANGLES。当调用 glDrawArrays 或其他 OpenGL 绘制函数时，基本几何类型必须与我们指定的变换反馈图元模式相匹配，或者必须有一个几何着色器输出正确的图元类型。例如，如果 primitiveMode 为 GL\_TRIANGLES，那么必须调用以 GL\_TRIANGLES、GL\_TRIANGLE\_STRIP 或 GL\_TRIANGLE\_FAN 为参数的 glDrawArrays，或者必须有一个生成 GL\_TRIANGLE\_STRIP 的几何着色器。变换反馈图元模式到绘制类型的映射如表 12.1 所示。

表 12.1

图元模式的值

图元模式的值	允许绘制的类型
GL_POINTS	GL_POINTS
GL_LINES GL_LINES,	GL_LINE_STRIP, GL_LINE_LOOP
GL_TRIANGLES	GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN

在变换反馈模式推出之前，或者在为变换反馈缓冲区分配的空间耗尽之前，顶点会记录到变换反馈缓冲区中。要退出变换反馈模式，可以调用

```
glEndTransformFeedback();
```

所有出现在一次对 `glBeginTransformFeedback` 和 `glEndTransformFeedback` 的调用之间的渲染都会导致数据被写入当前绑定的变换反馈缓冲区中。每次在 `glBeginTransformFeedback` 被调用时，OpenGL 就开始在为了变换反馈而进行绑定的缓冲区的开始位置写入数据，覆盖那里可能已经存在的旧数据。在变换反馈缓冲区激活，并且在调用 `glBeginTransformFeedback` 和 `glEndTransformFeedback` 之间禁止改变变换反馈状态时，需要多加注意。举例来说，在变换反馈模式激活时，我们不可能改变变换反馈缓冲区绑定，也不可能对任何变换反馈缓冲区重新设定大小或重新进行分配。

## 12.5.2 关闭光栅化

到目前为止，我们已经了解，变换反馈是一种在 OpenGL 进行渲染时保存顶点着色器或几何着色器中间结果的机制。但是，如果并不想实际绘制任何东西呢？如果只想使用变换反馈本身而不改变屏幕内容呢？如果我们使用顶点着色器进行除了几何图形处理以外的计算（例如物理模拟），就很可能想要这么做。我们可以通过关闭光栅化来使用变换反馈达到我们的目的。也就是说，顶点着色器和几何着色器仍然会运行，所以变换反馈也会工作，但是在这之后 OpenGL 管线将会被删掉，所以片段着色器根本不会运行。这样，举例来说，这种方式就比仅仅创建一个丢弃所有信息或通过 `glColorMask` 关闭颜色写入的片段着色器的效率高得多了。要关闭光栅化，实际上需要告诉 OpenGL，它应该通过调用

```
glEnable(GL_RASTERIZER_DISCARD);
```

来丢弃所有光栅化。

要重新开启光栅化，只需调用

```
glDisable(GL_RASTERIZER_DISCARD);
```

在启用 `GL_RASTERIZER_DISCARD` 时，顶点着色器和几何着色器所生成的任何东西都不会创建任何片段，而片段着色器则根本不会运行。如果关闭光栅化，并且不是用变换反馈模式，那么 OpenGL 管线实质上就会关闭。

## 12.5.3 使用图元查询对顶点进行计数

在存在一个顶点着色器，但不存在几何着色器时，顶点着色器的输出将被记录，而存储到变换反馈缓冲区的顶点数量则与发送到 OpenGL 的顶点数量相等，除非某个变换缓冲区中的可用空间被耗尽。如果存在一个几何着色器，那么这个着色器可能会创建或丢弃顶点，这样写入到变换反馈缓冲区的顶点数量就可能与发送到 OpenGL 的顶点数量不同。OpenGL 能够通过查询对象跟踪写入到变换反馈缓冲区中的顶点数量。应用程序随后能够使用这些信息对产生的数据进行绘制，或者了解从变换反馈缓冲区中读回的数据

据量，以及是否应该保存这些数据。

在本章前面讲解遮挡查询的内容时，已经介绍了查询对象。前面已经说过，我们可以向 OpenGL 询问很多问题。生成的图元数量和实际写入到变换反馈缓冲区中的图元数量都可以进行查询。

和以前一样，可以调用

```
glGenQueries(1, &one_query);
```

生成一个查询对象，或者调用

```
glGenQueries(10, ten_queries);
```

生成几个查询对象。

现在已经创建了查询对象，可以通过启用相应的查询类型要求 OpenGL 通过进行 GL\_PRIMITIVES\_GENERATED 或 GL\_TRANSFORM\_FEEDBACK\_PRIMITIVES\_WRITTEN 查询在图元生成时对它们进行计数了。

要开始任意一种查询，可以调用

```
glBeginQuery(GL_PRIMITIVES_GENERATED, one_query);
```

或

```
glBeginQuery(GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN, one_query);
```

在调用一次以 GL\_PRIMITIVES\_GENERATED 或 GL\_TRANSFORM\_FEEDBACK\_PRIMITIVES\_WRITTEN 为参数的 glBeginQuery 之后，OpenGL 会对由顶点着色器或几何着色器产生的图元数量保持跟踪，或者对实际写入到变换反馈缓冲区的图元数量保持跟踪，直到使用

```
glEndQuery(GL_PRIMITIVES_GENERATED);
```

或

```
glEndQuery(GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN);
```

停止查询为止。

查询的结果可以通过调用以 GL\_QUERY\_RESULT 和查询对象的名称为参数的 glGetQueryObjectiv 进行读取。和其他 OpenGL 查询一样，得到的结果可能不会立即可用，这是由于 OpenGL 的管线本质所决定的。要查询这些结果是否可用，可以调用以 GL\_QUERY\_RESULT\_AVAILABLE 为参数的 glGetQueryObjectiv。更多关于查询对象的信息请参考本章前面“12.1 查询功能——收集 OpenGL 管线相关信息”部分的内容。

在 GL\_PRIMITIVES\_GENERATED 和 GL\_TRANSFORM\_FEEDBACK\_PRIMITIVES\_WRITTEN 查询之间有两处微妙的差别。首先，GL\_PRIMITIVES\_GENERATED 查询对几何着色器发出的图元进行计数，而 GL\_TRANSFORM\_FEEDBACK\_PRIMITIVES\_WRITTEN 查询只对成功写入到变换反馈缓冲区的图元进行计数。由几何着色器生成的图元数量可能比发送到 OpenGL 的几何图元数量更多或更少，这取决于它在做什么样的工作。通常，这两种查询的结果应该是相同的，但是如果在变换反馈缓冲区内没有足够空间的

话, 那么 `GL_PRIMITIVES_GENERATED` 将保持计数, 而 `GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN` 会停止计数。

通过同时运行这两种查询并且对结果进行比较, 可以检查应用程序所产生的所有图元是否都被捕获到了变换反馈缓冲区中。如果它们相等, 那就说明所有图元都被成功写入。如果它们不同, 则说明我们用来进行变换反馈的缓冲区可能太小了。

第二个不同点是, `GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN` 只在变换反馈缓冲区活动时才有意义。这就是为什么它的名称中包含 `TRANSFORM_FEEDBACK`, 而 `GL_PRIMITIVES_GENERATED` 的名称中却不包含的原因。如果在变换反馈缓冲区不活动时运行一个 `GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN` 查询, 那么结果将会为 0。

但是, `GL_PRIMITIVES_GENERATED` 查询则可以在任何时间使用, 并且将得到由 OpenGL 生成的图元数量的可信值。我们可以使用它来查询几何着色器生成或丢弃了多少顶点。

## 12.5.4 使用图元查询的结果

现在我们得到了顶点着色器或几何着色器的结果, 这个结果就保存在一个缓冲区中。我们还通过使用一个查询对象了解了这个缓冲区中有多少数据。现在可以使用这些结果进行更进一步的渲染了。我们应该还记得, 顶点着色器或几何着色器的结果会通过变换反馈放置到一个缓冲区中。只要将这个缓冲区绑定到一个 `GL_TRANSFORM_FEEDBACK_BUFFER` 绑定点上, 它就会成为一个变换反馈缓冲区。但是, OpenGL 中的缓冲区是通用数据块, 并且可以应用于其他目的。

通常情况下, 运行一次渲染传递在一个变换反馈缓冲区中产生这些数据后, 我们会将这个缓冲区对象绑定到 `GL_ARRAY_BUFFER` 绑定点, 以便它能够作为顶点缓冲区使用。如果我们使用的是一个可能产生数据总量未知的几何着色器, 就需要使用一个 `GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN` 查询来查出第二次传递要对多少顶点进行渲染。程序清单 12.20 展示了相关代码的大致内容。

程序清单 12.20 写入一个变换反馈缓冲区的绘制数据

```
// 我们一共有两个缓冲区, 即 buffer1 和 buffer2
// 首先, 我们将 buffer1 作为绘制操作 (GL_ARRAY_BUFFER) 的数据源进行绑定
// 而 buffer2 则作为变换反馈 (GL_TRANSFORM_FEEDBACK_BUFFER) 的目标进行绑定
glBindBuffer(GL_ARRAY_BUFFER, buffer1);
glBindBuffer(GL_TRANSFORM_FEEDBACK_BUFFER, buffer2);
// 现在, 我们需要运行一个查询来对写入到变换反馈缓冲区的顶点进行计数
glBeginQuery(GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN, q);
// 好了, 开始变换反馈.....
glBeginTransformFeedback(GL_POINTS);
// 进行一些绘制来将数据放入变换反馈缓冲区
DrawSomePoints();
// 完成变换反馈
glEndTransformFeedback();
// 结束查询并取回结果
glEndQuery(GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN);
glGetQueryObjectiv(q, GL_QUERY_RESULT, &vertices_to_render);
// 现在我们将 buffer2 (它刚刚作为一个变换反馈缓冲区使用) 作为一个顶点缓冲区进行绑定, 并从中渲染更多点
glBindBuffer(GL_ARRAY_BUFFER, buffer2);
glDrawArrays(GL_POINTS, 0, vertices_to_render);
```

## 12.5.5 变换反馈的应用实例

这里有两个例子，展示我们可以如何使用一个变换反馈缓冲区中的数据。但是请记住，OpenGL 是非常灵活的，而变换缓冲区可以有无数潜在的应用方式。

### 存储中间结果

变换反馈的第一种实例用法是对中间结果进行存储。我们已经对实例渲染有所了解。让我们来考虑一个算法，它执行一组逐个实例操作，然后要求返回每个顶点的这些结果。现在，假定我们想要使用实例渲染来渲染这个对象的很多副本。我们可以设置一个顶点着色器，它使用一些实例数组和一些常规的逐顶点属性作为其输入。必须为这个对象的每一个副本执行所有这些逐实例计算，即使它们每次都会产生同样的结果。

这种算法可以分成两次进行，而不是编写一个庞大的顶点着色器一次性完成所有计算。编写第一个顶点着色器来计算通用的逐实例结果，并将它们作为一组输出 varying 变量写入到一个变换反馈缓冲区。现在这个着色器可以为每个实例运行一次。

接下来，编写第二个顶点着色器，执行其余计算（这些计算对于这个对象的每个副本来说是不同的），并使用一个实例数组读取这些逐实例属性，从而将它们与来自第一个顶点着色器的中间结果进行结合。

现在我们有了一对着色器，可以将第一个着色器为每个实例运行一次（使用一个常规的 `glDrawArrays` 命令），然后使用第二个着色器实际渲染这个对象的每一个副本。第一个着色器（逐顶点运行的那个）应该在关闭光栅化（通过启用前面讨论过的 `GL_RASTERIZER_DISCARD`）的情况下运行。这样就能在变换反馈缓冲区中产生中间结果，而不需要实际进行任何渲染。现在，重新开启光栅化，然后使用第二个着色器，并调用一个类似 `glDrawArraysInstancedv` 这样的实例渲染函数，对这个对象的所有独立副本进行渲染。

### 迭代或递归算法

很多算法都是迭代的，其结果从一步到下一步进行循环。物理模拟是这种算法的一种主要用途，而变换反馈则是创建后续传递中重用数据的一种理想方法。因为变换反馈将数据写入缓冲区所使用的格式允许这些缓冲区以后作为顶点缓冲区进行绑定，所以在数据每次传递之间不需要进行变换或复制。所需要的只是一个简单的双重缓冲区方案。

粒子系统模拟是循环算法的一个好例子。在进行模拟的每一个步骤中，每个粒子都有一个位置值和一个体积值必须要进行更新。它们可能还会有一些固定不变的参数，例如质量、颜色或任何数量的其他属性。要使用变换反馈生成一个简单的粒子系统，每个粒子都应该可以表示为一个顶点，而它的属性则存储在顶点缓冲区中。一个顶点着色器可以进行构造，为系统中的粒子计算一个更新的位置值和速度值。那些在粒子系统进行迭代时保持不变的粒子参数可以存储在一个顶点缓冲区中，最好使用 `GL_STATIC_DRAW` 应用模式进行分配。那些在每次分配之间进行改变的参数应该进行双重缓冲。我们用一个缓冲区作为顶点缓冲区，同时作为参数的源，以用来对粒子系统进行渲染。第二个缓冲区作为一

个变换反馈缓冲区进行绑定, 并且对由顶点着色器写入其中的参数进行更新。在每次迭代之间, 这两个缓冲区会进行交换。

当粒子系统进行渲染时, 一个时间步长会被传递到顶点着色器中, 它表示从上一次更新以来经过了多长时间。顶点着色器计算由于粒子的质量而作用在粒子上力 (即重力) 的近似值、输入速率 (空气阻力), 以及其他任何对应用程序来说重要的因子; 综合这个时间步长中的粒子速度; 然后生成一个新的位置和速度。

为了将这些例子作为点进行渲染, 可以使用一个以 `GL_POINTS` 为图元类型的类似于 `glDrawArrays` 这样的命令。我们可能会希望只使用变换反馈缓冲区对粒子位置进行更新, 但要在每个粒子处进行一些更加复杂的绘制 (例如一个球体或一艘太空船)。要做到这一点, 我们可以启用 `GL_RASTERIZER_DISCARD` 在更新阶段关闭光栅化, 然后使用位置数据作为第二次传递的输入, 将这些点转换成更加复杂的几何图形组, 在屏幕上进行渲染。

### 更加深入的变换反馈示例——集群 (Flocking)

我们将这两个例子组合起来, 创建一个集群算法 (flocking algorithm) 的实现。集群算法通过更新独立于所有其他成员的个体成员属性, 显示了一个庞大群组中的突发性行为。这种类型的行为在自然界中经常可以看到, 这样的例子包括一大群蜜蜂、一大群鸟类和一大群鱼, 它们看起来都是统一地进行运动, 即使这个群组的成员并不直接进行沟通。由一个个体所做出的决定只取决于它对群组中其他成员的感知力。但是, 对于任何特定决定所产生的结果, 在各个成员之间并没有进行任何协作。这就意味着每个群组成员的新属性可以并行计算——这对于 GPU 实现来说是非常理想的。

为了对前面提到的两个方法 (存储中间结果和迭代算法) 都进行演示, 我们通过一对顶点着色器来实现集群算法。我们将这个群组的每个成员都表示为单个顶点。每个顶点都有一个位置值和一个速度值, 它们由第一个顶点着色器进行更新。得到的结果通过变换反馈来写入到一个缓冲区。随后这个缓冲区会作为一个顶点缓冲区进行绑定, 并作为第二个着色器的一个实例输入来使用。这个集群中的每个成员都是第二次绘制中的一个实例。第二个顶点着色器负责将一个网点 (可能是一个鸟的模型) 变换到在第一个顶点着色器中计算出的位置和方向。然后算法将进行迭代, 再次从第一个着色器开始, 并重用上一次过程中计算出的位置和速率。所有数据都不会离开图形加速卡的内存, 而 CPU 则与这里的任何计算无关。

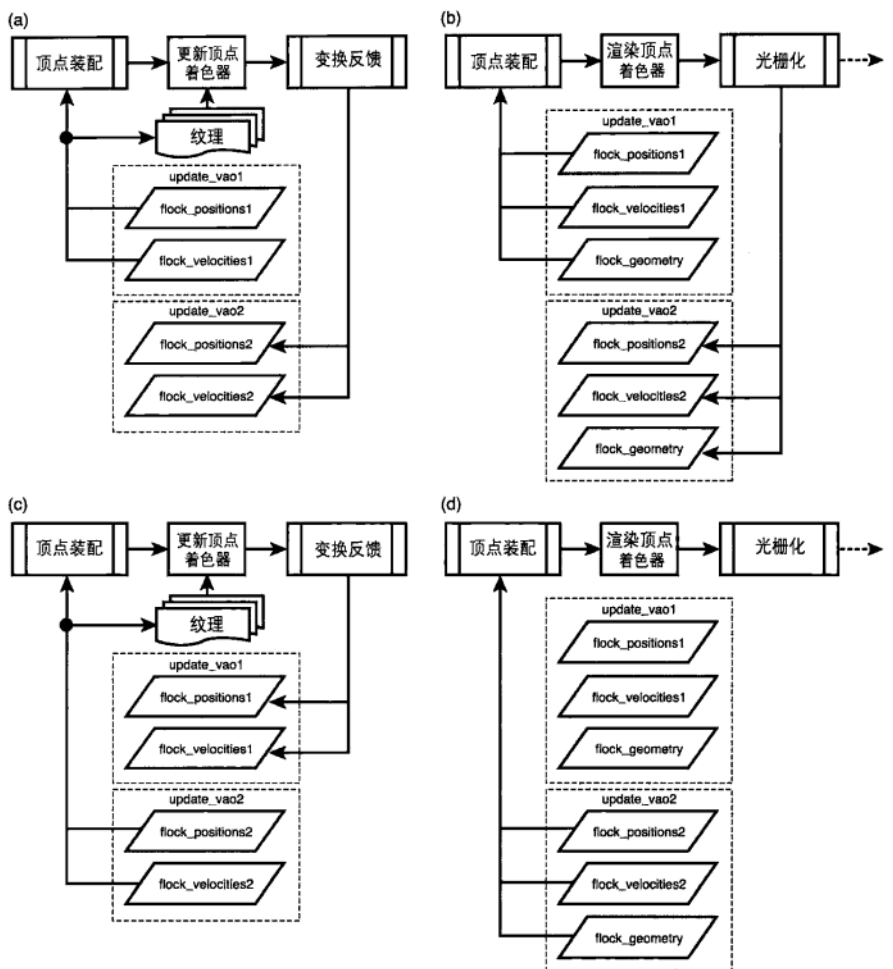
在这个例子中, 我们需要的数据结构是一组 VAO, 用来表示每次迭代过程中的顶点数组状态, 以及一组 VBO, 用来保存群组中成员的位置和速率, 以及用来表示它们的模型的顶点数据。群组位置和速率需要进行双重缓冲, 因为无法使用变换反馈来对同一个缓冲区同时进行读取和写入。同时, 因为群组的每个成员 (即顶点) 都需要对群组中所有其他成员的当前位置和速率进行访问, 所以我们将位置和速率缓冲区同时绑定到一对纹理缓冲区对象 (TBO) 上。这样, 顶点缓冲区就可以随意地对 TBO 进行读取, 来访问其他顶点的属性了。前面在第 8 章已经介绍了 TBO。

图 12.10 所示对这个算法的过程进行了图示。

在 (a) 中, 我们对一个偶数帧进行了更新。第一组位置和速率缓冲区作为顶点着色器的输入进行绑定, 而第二组位置和速率缓冲区则作为变换反馈缓冲区进行绑定。请注意, 这里我们还使用第一组位置和速率缓冲区来支持顶点着色器使用的纹理 (实际上就是 TBO)。

图 12.10

迭代集群算法中的各个阶段



接下来，在 (b) 中，我们使用同一组缓冲区作为输入进行渲染，这和在更新过程中一样。我们在更新过程和渲染过程中使用同一组缓冲区作为输入，所以渲染过程对更新过程并没有任何依赖性。这就意味着 OpenGL 可能能够在更新过程结束前开始执行渲染过程。这一组位置和速度缓冲区现在已经进行实例化，而附加的几何图形缓冲区则用来提供顶点位置数据。

在 (c) 中，我们开始处理下一帧。这些缓冲区进行了交换——第二组缓冲区现在是顶点着色器的输入，而第一组缓冲区则使用变换反馈进行写入。

最后，在 (d) 中，我们开始渲染这个奇数帧。第二组缓冲区用作顶点着色器的输入。不过请注意，flock\_geometry 缓冲区既是 render\_vao1 的成员，也是 render\_vao2 的成员，因为这两个过程中都会使用同样的数据，所以不需要有两个副本。

程序清单 12.21 列出了进行所有这些设置的代码。这些内容并不是特别复杂的，但是其中有很多重复部分，这使它变得很冗长。这个程序清单包含初始化程序块，其中部分内容出于简洁考虑而进行了省略(省略的部分在注释中用\*\*\* 表示)。

## 程序清单 12.21 为集群算法实例进行数据结构初始化

```
// 创建四个 VAO——即 update_vao1、update_vao2、render_vao1 和 render_vao2
// 的确，我们可以使用一个数组，但是为了达到本例的目的，这样做更加清晰
glGenVertexArrays(1, &update_vao1);
// *** 用同样方法创建 update_vao2、render_vao1 和 render_vao2

// 创建缓冲区对象。稍后将对它们进行绑定和初始化
glGenBuffers(1, &flock_positions1);
// *** 用同样方法创建 flock_positions2、flock_velocities1、flock_velocities2 和 flock_geometry

// 设置 VAO 和缓冲区——首先是 update_vao1
glBindVertexArray(update_vao1);
glBindBuffer(GL_ARRAY_BUFFER, flock_positions1);
// *** 在这里将一些初始位置放入 flock_positions1
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, flock_velocities1);
// *** 用 0 对进行初始化（例如 glBufferData(... NULL)、glMapBuffer、memset）
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(1);

// 接下来是 update_vao2
// *** 这基本上和 update_vao1 相同，
// *** 除非我们不需要 flock_positions2 或 flock_velocities2 的初始数据
// *** 因为它们将在第一次处理过程中被写入。
// *** 但是，我们确实需要使用 glBufferData(... NULL) 来对它们进行分配

// 现在轮到那些渲染 VAO 了——首先是 render_vao1
// 我们将同样的 flock_positions1 和 flock_positions2 缓冲区绑定到这个 VAO 上
// 但是这一次，它们是实例数组 我们还将作为一个常规顶点数组进行绑定
glBindVertexArray(render_vao1);
glBindBuffer(GL_ARRAY_BUFFER, flock_positions1);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(0);
glVertexAttribDivisor(0, 1);
glBindBuffer(GL_ARRAY_BUFFER, flock_velocities1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(1);
glVertexAttribDivisor(1, 1);
glBindBuffer(GL_ARRAY_BUFFER, flock_geometry);
glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(2);

// 设置 render_vao2
// *** 这看起来和设置 render_vao1 差不多，只不过我们使用的是 flock_positions2 和 flock_velocities2
// *** 但是，请注意，我们仍然对 flock_geometry 进行绑定，因为它在每次迭代之间并不进行改变

// 最后来设置 TBO
glGenTextures(1, &position_texture1);
glBindTexture(GL_TEXTURE_BUFFER, position_texture1);
glBindBuffer(GL_TEXTURE_BUFFER, flock_positions1);
// *** 用同样的方法为 flock_velocities1、flock_position2 和 flock_velocities2 各创建一个缓冲区纹理
```

一旦我们完成了缓冲区的设置，就需要对着色器进行编译，然后在一个程序中将它们连接到一起。在程序进行连接之前，我们需要将顶点着色器中的属性绑定到正确的位置，以使它们与我们设置的顶点数组相匹配。

我们还需要告诉 OpenGL，我们想要将哪些 varying 变量写入变换反馈缓冲区中。程序清单 12.22 显示了这些顶点属性和变换反馈 varying 变量是如何进行初始化的。

## 程序清单 12.22 为集群算法实例进行属性和变换反馈初始化

```
// *** 假定我们创建了我们的顶点着色器和片段着色器，对它们进行编译并将它们绑定到我们的程序对象上
// 首先，我们要设置更新程序中的属性
glBindAttribLocation(update_program, 0, "position");
glBindAttribLocation(update_program, 1, "velocity");
// 现在轮到渲染程序了。
// 前两个属性实际上与更新程序写入的这两个属性相同。
// 第三个属性是几何图形中顶点的位置
glBindAttribLocation(render_program, 0, "instance_position");
glBindAttribLocation(render_program, 1, "instance_velocity");
glBindAttribLocation(render_program, 2, "geometry_position");
// 现在我们来设置变换反馈 varying 变量:
static const char * tf_varyings[] = { "position_out", "velocity_out";
                                       GL_SEPARATE_ATTRIBS);
// 现在，所有设置都已经完成，所以我们可以继续连接我们的程序对象了
glLinkProgram(update_program);
glLinkProgram(render_program);
```

现在我们需要一个渲染循环来对集群位置进行更新，并绘制集群中的成员。实际上这非常简单，现在我们已经将数据封装到了 VAO 中。程序清单 12.23 列出了渲染循环。

## 程序清单 12.23 集群算法的渲染循环

```
// 将更新程序设为当前程序
glUseProgram(update_program);
// 我们使用一组缓冲区作为着色器输入，而其他缓冲区则作为变换反馈缓冲区来保存着色器输出
// 在帧进行交替时，我们将对这两个缓冲区进行交换
if (frame_index & 1) {
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, position_buffer1);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 1, velocity_buffer1);
    glBindVertexArray(update_vao2);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_BUFFER, position_texture2);
    glActiveTexture(GL_TEXTURE1);
    glBindTexture(GL_TEXTURE_BUFFER, velocity_texture2);
} else {
    // *** 这里还是和前面相同，只是使用作为 position_buffer2 和 velocity_buffer2 变换反馈缓冲区
    // *** 而 update_vao1、position_texture1 和 velocity_texture1 则作为输入
}
// 关闭光栅化（开启光栅化丢弃）
glEnable(GL_RASTERIZER_DISCARD);
// 开始变换反馈——记录更新位置
glBeginTransformFeedback(GL_POINTS);
// 绘制数组——为集群的每个成员绘制一个点
glDrawArrays(GL_POINTS, 0, flock_size);
// 完成变换反馈
glEndTransformFeedback(GL_POINTS);
// 好了，现在我们要进行所有绘制工作了。现在我们需要重新开启光栅化
glDisable(GL_RASTERIZER_DISCARD);
// 使用渲染程序
glUseProgram(render_program);
if (frame_index & 1) {
    glBindVertexArray(render_vao2);
} else {
    glBindVertexArray(render_vao1);
}
// 进行一次实例绘制——每个成员都是一个实例
// 通过“update_program”更新的上一帧数据现在是 render_program 中的一个实例数组了
glDrawArraysInstanced(GL_TRIANGLES, 0, 50, flock_size);
frame_index++;
```

这些差不多就是程序端最有趣的部分了。下面我们来看一下着色器端的情况。集群算法是通过为群组的每一个成员应用一组规则，来决定向哪个方向行进。每个规则都将群组成员的当前属性和其他成员的属性视为是正在进行更新的个体所感知到的。这些规则中大部分都需要访问其他成员的位置和速率，所以 update\_program 使用一对 TBO 从包含这些信息的缓冲区中进行读取。程序清单 12.24 展示了更新顶点着色器的开始部分。

程序清单 12.24 为集群算法实例进行属性和变换反馈初始化

```
#version 150

precision highp float;

// 这些是输入属性
in vec3 position;
in vec3 velocity;

// 这些将被写入到变换反馈缓冲区
out vec3 position_out;
out vec3 velocity_out;

// 这些是作为位置和速率映射到同一个缓冲区的 TBO
uniform samplerBuffer tex_position;
uniform samplerBuffer tex_velocity;

// 集群中成员的数量
uniform int flock_size;

// 在模拟中使用的参数
uniform Parameters
{
    // *** 将所有模拟参数放在这里
};
```

程序的主体非常简单。我们只要读取集群中其他成员的位置和速率，轮流应用这些数据，计算出结果向量，并输出一个更新的位置和速率。程序清单 12.25 显示了完成这些工作的代码。

程序清单 12.25 集群更新顶点着色器的程序主体

```
void main(void)
{
    vec3 other_position;
    vec3 other_velocity;
    vec3 acceleration = vec3(0.0);
    int i;

    for (i = 0; i < flock_size; i++) {
        other_position = texelFetch(tex_position, i).xyz;
        other_velocity = texelFetch(tex_velocity, i).xyz;
        acceleration += rule1(position, velocity,
                               other_position, other_velocity);
        acceleration += rule2(position, velocity,
                               other_position, other_velocity);
        // *** 诸如此类.....我们可以应用我们想要的任意数量的规则
        // *** 要生成一个令人信服的模拟效果，3 个或 4 个就足够了
    }

    position_out = position + velocity;
    velocity_out = velocity + acceleration / float(flock_size);
}
```

现在我们必须定义这些规则，使用的规则如下所示。

- 各个成员试图避免和其他成员发生碰撞。它们需要与其他成员保持一定距离。
- 各个成员试图与周围其他成员沿着相同方向飞行。
- 各个成员试图保持与群组中的其他成员在一起。它们将会飞向集群的中心。

程序清单 12.26 显示了实现第一个规则的着色器代码。如果我们与其他成员的距离比预想的要近，那么只需向远离这个成员的方向移动。

程序清单 12.26 集群的第一条规则

```
vec3 rule1(vec3 my_position, vec3 my_velocity,
           vec3 their_position, vec3 their_velocity)
{
    vec3 d = my_position - their_position;
    if (dot(d, d) < parameters.closest_allowed_position)
        return d * parameters.rule1_weight;
    return vec3(0.0);
}
```

程序清单 12.27 显示了实现第二个规则的着色器代码。它返回一个由从一个成员到另一个成员距离的平方倒数进行加权的速度改变值。

程序清单 12.27 集群的第二条规则

```
vec3 rule2(vec3 my_position, vec3 my_velocity,
           vec3 their_position, vec3 their_velocity)
{
    vec3 dv = (their_velocity - my_velocity);
    return parameters.rule2_weight *
        dv / (dot(my_position, their_position) + 1.0);
}
```

将这些规则和其他我们想要实现的规则综合起来，完成程序的更新部分。现在我们需要生成第二个顶点着色器——它负责对集群进行渲染。它将这个位置和速度数据作为实例数组使用，并将一个固定的顶点集合变换到一个根据独立成员的位置和速度得出的位置值。程序清单 12.28 展示了这个着色器的输入。

程序清单 12.28 声明集群渲染顶点着色器的输入

```
#version 150

precision highp float;

// 这些是实例数组
in vec3 instance_position;
in vec3 instance_velocity;

// 常规几何数组
in vec3 position;
```

我们着色器的程序体（在程序清单 12.29 中给出）只是为了特定实例将由 position 表示的网点变换到正确的方向和位置。

程序清单 12.29 集群顶点着色器程序体

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

```
// rotate_to_match是一个函数，它将一个点（位置）围绕原点进行旋转，以匹配一个方向向量
(instance_velocity)
vec3 local_position = rotate_to_match(position, instance_velocity);
gl_Position = mvp * vec4(instance_position + local_position, 1.0);
}
```

好了！在这里我们并不打算介绍 rotate\_to\_match，因为这些内容超出了本例的范围。在本书的网站中，我们可以找到一个完整的实现，以及本例的其余部分代码。当然，如果我们想要渲染一些比平面的白色粒子更加有趣的东西，那么最终的渲染顶点着色器可能会变得更加复杂。它还需要包含一些额外的逻辑，以试图保持集群成员是直立的（并且让它们停止围绕它们的轴旋转）。在这里，我们也并不打算介绍这个片段着色器，因为它和实例化或变换反馈都没什么关系。

## 12.6 裁剪并确定绘制内容

当我们将几何图形发送到 OpenGL 时，它会被顶点着色器和几何着色器从输入（对象）坐标空间变换到裁剪坐标空间。OpenGL 就是在这个时候执行裁剪来决定哪些顶点落在视口内，以及哪些顶点落在视口之外。

要完成这项工作，OpenGL 会将 3D 空间分成 6 个“半空间”（half space），它们是由裁剪区的边界定义的。这些半空间是由所谓的左裁剪面、右裁剪面、上裁剪面、下裁剪面、近端裁剪面和远端裁剪面定义的。在每个顶点经过裁剪阶段时，OpenGL 会计算出从这个顶点到这些平面中每一个平面有符号的距离值。这个距离的绝对值并不重要——重要的只是它的符号。如果这个顶点到某个平面的距离是正的，那么这个顶点就位于这个平面的内部（如果站在视景体的中间并看向这个平面的话，能够看到这个顶点，而不会被平面遮挡）。如果这个距离是负的，那么这个顶点就位于这个平面的外侧。如果这个距离恰好是 0，那么这个顶点就恰好位于这个平面上。现在，OpenGL 只要检查一下这个顶点到这 6 个平面距离值的正负，就可以很快确定一个顶点是否在视景体的内部或外部了，而通过综合几个顶点的结果，就可以确定一个大块的几何图形是否可见。

如果单个三角形的所有顶点都位于任意单个平面的外部——也就是说，这个三角形的所有顶点到同一个平面的距离都是负的，那么这个三角形就完全在视景体之外，并且可以简单地丢弃。

类似地，如果一个三角形的任意一个顶点到任何一个平面的距离都不为负，那么这个三角形就完全在视景体之内，从而就是可见的。

只有当一个三角形跨越一个平面时，才需要做更进一步的工作。这种情况就意味着，这个三角形是部分可见的。OpenGL 的不同实现会以不同的方式处理这些情况。其中某些实现会使用一个类似 Sutherland-Hodgman 这样的裁剪算法将这个三角形分解成几个更小的三角形。而另一些实现则会将整个三角形进行简单的光栅化，并强制丢弃最终落在视口外部的片段。

这 6 个平面在裁剪空间中组成了一个长方体形状，它看起来像是在更大的 3D 空间中的一个盒子。当它被变换到窗口坐标时，可能会经历一次透视变换并成为一个平截头体（frustum）。这就是所谓的视景体（view frustum）。

### 12.6.1 裁剪距离——自定义裁剪空间

除了到这 6 个组成视景体的标准裁剪平面的这 6 个距离之外，应用程序还可以使用一组额外的距离，它们可以写入顶点或几何着色器中。这些裁剪距离可以通过内建变量 `gl_ClipDistance[]` 写入顶点着色器，这个内建变量是一个浮点值数组。能够支持的裁剪距离数量取决于具体 OpenGL 实现。这些距离会完全按照内建裁剪距离进行解释。如果一个着色器编写者想要使用用户定义的裁剪距离，那么它们将由应用程序通过调用

```
glEnable(GL_CLIP_DISTANCE0 + n);
```

来启用。

在这里，`n` 是要启用裁剪距离的索引。`GL_CLIP_DISTANCE1`、`GL_CLIP_DISTANCE2` 等，一直到 `GL_CLIP_DISTANCE5`，这些标记通常在标准 OpenGL 头文件中都进行了定义。但是，`n` 的最大值是由实现定义的，并且可以通过调用以 `GL_MAX_CLIP_DISTANCES` 为标记的 `glGetIntegerv` 来找到。我们可以通过调用带有同样标记的 `glDisable` 来禁用这个用户定义裁剪距离。如果一个特定索引中的用户定义裁剪距离没有被启用，那么在这个索引中写入到 `gl_ClipDistance[]` 的值就会被忽略。

和内建裁剪平面一样，写入到 `gl_ClipDistance[]` 数组的距离标记也用来确定一个顶点是否在用户定义的裁剪区域的内部或外部。如果一个三角形的每个顶点的所有这些距离的符号都为负，那么这个三角形就会被裁剪掉。如果这个三角形确定是部分可见的，那么这个裁剪距离就会在这个三角形上进行线性插值，并确定每个像素的可视性。这样，渲染结果将是对逐个顶点距离函数的线性近似，由顶点着色器进行估算。这样就允许一个顶点着色器对一个任意的平面集合进行几何图形裁剪了（一个点到平面的距离可以通过一个简单的点乘运算来得到）。

`gl_ClipDistance[]` 数组也可以作为片段着色器的一个输入使用。在 `gl_ClipDistance[]` 的任何元素中，为负值的片段都会被裁剪掉，永远不会到达片段着色器。但是，任何在 `gl_ClipDistance[]` 中只有正值的片段都会被传递到片段着色器，这样这个值就会被读取，并且可以被这个着色器用来达到任何目的。这项功能的一个应用范例，就是通过减少一个片段中决定裁剪距离的 `alpha` 值并使其接近于 0，来使这个片段逐渐消失。这样就可以使被顶点着色器用一个平面来进行裁剪的大块图元逐渐消失，或者由片段着色器进行抗锯齿而不是生成一个生硬的裁剪边缘。请注意，如果组成一个图元（点、线或三角形）的所有顶点都被同一个平面裁剪的话，那么整个图元就会被消除，这一点非常重要。

这样做看起来很有道理，并且对于常规多边形网格来说，其行为也是符合预期的。但是，在使用点和线时，就要小心了。在使用点的情况下，我们可以通过将参数设置为一个大于 1.0 的值，用一个覆盖多个像素的单个顶点渲染一个点。当 `gl_PointSize` 很大时，就会围绕这个顶点渲染一个很大的点。这就意味着，如果有一个很大的点缓慢地向屏幕的边缘移动，并且最终出离这个边缘，那么在这个点的中心出离视景体，而表示这个点的顶点被裁减掉时，这个点就会突然消失，类似地，OpenGL 也可以渲染很粗的线。如果要绘制一条线的两个顶点都在裁剪平面外部，但是某些其他部分应该可见，那么在这种情况下不会进行任何绘制。如果我们对此不多加小心的话，就可能出现奇怪的不自然痕迹。

左裁剪面、右裁剪面、上裁剪面和下裁剪面在现实世界中都有对应元素——也就是我们的视野边界。

在现实中,视野并不是严格的矩形。它更像一个边界模糊的椭圆形。但是在实践中,视口的边界(例如显示器的边缘)定义了一个硬性的限制。类似地,近端裁剪面也大致与眼睛平面相对应。任何位于近端裁剪面后面的东西实际上是位于我们后面的,这样我们就无法看到它了,但是在真实世界中却没有相应的等价物。那么远端裁剪面又如何呢?在现实世界中根本不存在远端裁剪面的等价物。光线在没有遇到遮挡的情况下可以传播到无限远处。那么为什么还需要一个远端裁剪面呢?OpenGL 用一个位于 0 到 1 之间有限大小的数字来表示每个片段的深度。一个深度值为 0 的片段会与近端裁剪面相交(如果是真实世界的话,它会进入观察者的眼睛里面),而一个深度值为 1 的片段则位于可以表示的最大深度,但并不是无限远的。如果要消除远端裁剪平面,从而绘制位于任何距离的物体,就需要在深度缓冲区中存储任意大的数字——而这是不可能的。为了克服这个问题,OpenGL 可以选择关闭近端裁剪面和远端裁剪面的裁剪,代之以只生成 0 到 1 之间的深度值。这就意味着,任何突出到近端裁剪面之后或远端裁剪面之外的几何图形实际上都会被投影到相应的裁剪面上。

要开启深度截取(同时关闭在近端裁剪面和远端裁剪面上的裁剪),调用

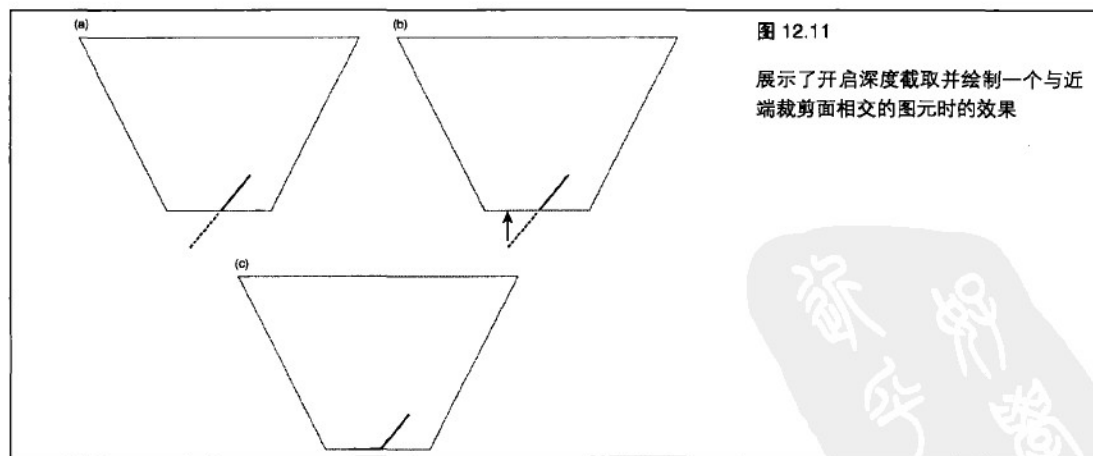
```
glEnable(GL_DEPTH_CLAMP);
```

而要关闭深度截取,则需要调用

```
glDisable(GL_DEPTH_CLAMP);
```

当然,这些只会影响 OpenGL 的内建近端裁剪面和远端裁剪面的裁剪计算。如果需要的话,我们仍然可以在我们的顶点着色器中使用一个用户定义的裁剪距离,来模拟一个深度值大于 1 的深度平面。

图 12.11 所示展示了开启深度截取并绘制一个与近端裁剪面相交的图元时的效果。

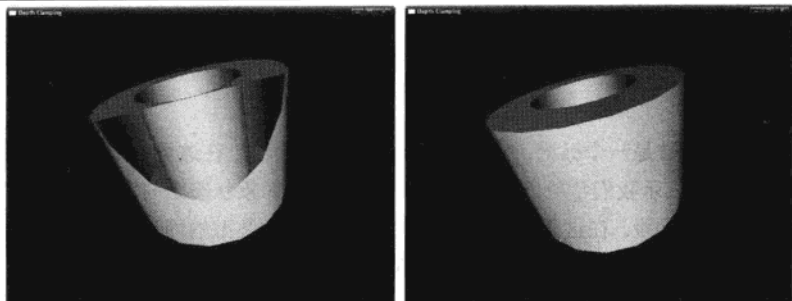


在两个维度中进行演示会直观得多,所以在图 12.11 (a)中显示了一个视景体,就好像我们直接看向它一样。图中的黑线表示会被近端裁剪面裁剪的图元,而虚线则表示图元中被裁剪掉的部分。当深度截取启用时,并不会对图元进行裁剪,而是将位于 0 到 1 范围之外的深度值截取到这个范围之内,这样就有效地将这个图元投影到了近端裁剪面(或者是远端裁剪面,如果图元会被它裁剪的话)。

图 12.11 (b)所示显示了这种投影。实际得到的渲染结果如图 12.11 (c)所示。图中的黑线代表最终被写入到深度缓冲区的值。图 12.12 所示展示了这些效果在一个实际应用程序中是如何体现的。

图 12.12

一个被裁剪对象在开启  
和关闭深度裁剪的情况  
下看起来的样子



在图 12.12 左侧的图片中，几何图形与观察者的距离如此之近，它的一部分已经被近端裁剪面进行裁剪了。结果是，这个多边形应该在近端裁剪面后面的部分就没有被绘制，这样就在模型上造成了一个大洞。我们可以通过这个大洞直接看到模型的另一面，而这个图像则看起来非常不对劲。在图 12.12 右侧的图片中则开启了深度截取。正如我们所看到的，在左侧图中丢失的几何图形又回来了，并且填充了对象中出现的大洞。这时在深度缓冲区中的值在技术上是说不正确的，但是这并没有导致视觉效果异常，并且得到的画面比起左侧图像来说要好得多。

## 12.7 在 OpenGL 开始绘制时进行同步

在一个高级应用程序中，OpenGL 的操作顺序和系统的管线本质可能会非常重要。这方面的例子包括那些包含多个环境和多个线程的应用程序，或者那些在 OpenGL 和类似 OpenCL 这样的其他 API 之间共享数据的应用程序。在某些情况下，确定发送到 OpenGL 的命令是否已经完成，以及这些命令的结果是否已经准备就绪，可能是非常必要的。OpenGL 包含两个命令，它们会强制开始或强制结束到目前为止已经发出命令的处理过程。它们分别是

```
glFlush();
```

和

```
glFinish();
```

这两个命令有着微妙的区别。第一个命令 `glFlush` 保证到目前为止发出的任何命令至少会放入 OpenGL 管线的始端，并且最终会被执行。而另一方面，`glFinish` 则实际上是保证所有发出的命令都被完整执行，而 OpenGL 管线则是空的。问题在于，`glFlush` 并不会告诉我们任何关于这些已发出命令执行状态的信息——我们只知道它们最终会被执行，而 `glFinish` 则保证所有 OpenGL 命令都被执行之后，它将清空 OpenGL 管线，导致泡沫，降低性能，有时甚至是极大地降低性能。

有时候，了解到某些时刻为止 OpenGL 是不是完成了命令的执行可能会非常重要。例如，当我们在两个环境之间，或者在 OpenGL 和 OpenCL 之间共享数据时，这一点就非常重要了。这种类型的同步是由所谓的同步对象（sync object）来进行管理的。和其他 OpenGL 对象一样，它们必须在使用之前先进行创建，并且在使用完并不再需要之后进行销毁。同步对象有两种可能的状态，即标记状态和未标记状态。它们在一开始时是未标记状态，而当某些特定事件发生时，它们会转换成标记状态。由什么事件来触发它们从未标记状态到标记状态的转变则取决于它们的类型。我们感兴趣的同步对象类型叫做围栏

同步 (fence sync), 可以通过调用

```
GLsync glFenceSync(GL_SYNC_GPU_COMMANDS_COMPLETE, 0);
```

来创建。

第一个参数是一个标记, 指定我们将要等待的事件。在这里, GL\_SYNC\_GPU\_COMMANDS\_COMPLETE 表示我们希望 GPU 在设置标记的同步对象状态之前完成管线中所有命令的处理。第二个参数是一个标志字段, 在这里为 0, 这是因为没有与这种类型的同步对象相关的标记。glFenceSync 函数返回一个新的 GLsync 对象。在围栏同步创建之后, 它马上 (以未标记状态) 进入 OpenGL 管线, 并与其他命令一起进行处理, 而不会使 OpenGL 产生延迟, 也不会消耗大量资源。当它到达管线终点时, 会像其他命令一样“执行”, 而这样就会将它的状态设为标记状态。由于 OpenGL 的排序特性, 我们由此可知, 任何在 glFenceSync 调用之前发出的 OpenGL 命令都已经完成, 即使在 glFenceSync 之后发出的命令可能还没有达到管线终点。

一旦同步对象被创建 (并且由此进入 OpenGL 管线), 就可以对这些状态进行查询, 以弄清它是否已经到达了管线的终点, 并且可以要求 OpenGL 等待它变成标记状态后再返回应用程序。

要确定同步对象是否已经变成标记状态, 可以调用

```
glGetSynciv(sync, GL_SYNC_STATUS, sizeof(GLint), NULL, &result);
```

当 glGetSynciv 返回时, 如果同步对象是在标记状态下, 那么结果 (是一个 GLint) 将包含 GL\_SIGNALED, 否则将包含 GL\_UNSIGNALED。这样, 应用程序就可以检验同步对象的状态, 并可能利用这些信息在 GPU 忙于处理以前的命令时做一些有用的工作。举例来说, 让我们考虑程序清单 12.30 中显示的代码。

程序清单 12.30 在等待一个同步对象时所做的工作

```
GLint result = GL_UNSIGNALED;
glGetSynciv(sync, GL_SYNC_STATUS, sizeof(GLint), NULL, &result);
while (result != GL_SIGNALED) {
    DoSomeUsefulWork();
    glGetSynciv(sync, GL_SYNC_STATUS, sizeof(GLint), NULL, &result);
}
```

这段代码是循环的, 它在每次迭代过程中完成少量的有用工作, 直到同步对象变为标记状态。如果这个应用程序要在每一帧开始时创建一个同步对象, 那么这个应用程序可以等待来自两个帧之前的同步对象, 并且根据 GPU 处理来自那一帧的命令所需的时间来完成相应的工作量。这样, 应用程序就可以在 CPU 完成的工作量 (例如混合的声音效果的数量, 或者进行的物理模拟迭代次数) 和 GPU 的速度之间进行平衡了。

要让 OpenGL 实际地等待一个同步对象变成标记状态 (并且由此在同步完成前等待管线中的命令), 我们可以使用两个函数:

```
glClientWaitSync(sync, GL_SYNC_FLUSH_COMMANDS_BIT, timeout);
```

或

```
glWaitSync(sync, 0, GL_TIMEOUT_IGNORED);
```

这两个函数的第一个参数都是由 glFenceSync 返回的同步对象的名称。而这两个函数的第二个参数和第 3 个函数的名称都相同, 但是必须进行不同的设置。

对于 `glClientWaitSync` 来说, 第二个参数是一个位段, 它指定函数的附加行为。 `GL_SYNC_FLUSH_COMMANDS_BIT` 告诉 `glClientWaitSync` 确保同步对象在开始等待它变成标记状态之前已经进入管线。如果没有这个位, 就有可能出现 OpenGL 等待一个还没有发送到管线中的同步对象的情况, 最终会导致应用程序永远等待并挂起。设置这个位是个好主意, 除非我们有充分的理由不这样做。第 3 个参数是一个需要等待的超时值, 以纳秒为单位。如果同步对象在这些时间内没有变成标记状态, 那么将返回一个状态代码来对此进行标记。

`glClientWaitSync` 可以返回 4 个可能的状态代码。表 12.2 对此进行了总结。

表 12.2 `glClientWaitSync` 的可能返回值

ClientWaitSync 的返回状态	含 义
<code>GL_ALREADY_SIGNALED</code>	在调用 <code>glClientWaitSync</code> 时同步对象已经为标记状态, 所以函数立即返回
<code>GL_TIMEOUT_EXPIRED</code>	已经达到超时参数所制定的超时值, 意味着在允许时间内同步对象不会变成标记状态
<code>GL_CONDITION_SATISFIED</code>	同步对象在允许时间内变成标记状态(但是在 <code>glClientWaitSync</code> 调用时还不是标记状态)
<code>GL_WAIT_FAILED</code>	出现一个错误(例如同步并不是一个有效同步对象), 而用户应该检查 <code>glGetError()</code> 的结果已获得更多信息

关于超时值, 还有两件事需要注意。首先, 当度量单位为纳秒时, 在 OpenGL 中就没有精度要求了。如果我们指定想要等待一纳秒, 那么 OpenGL 可能会将这个值舍入到下一毫秒或更长时间之后。第二, 我们指定一个大小为 0 的超时值, 如果在调用时同步对象为标记状态, 那么 `glClientWaitSync` 将会返回 `GL_ALREADY_SIGNALED`, 否则将会返回 `GL_TIMEOUT_EXPIRED`。它不会返回 `GL_CONDITION_SATISFIED`。

对于 `glWaitSync` 来说, 这个行为稍有不同。实际上应用程序不会等待同步对象变为标记状态, 只有 GPU 会等待。这样, `glWaitSync` 将会立即返回应用程序。这样就会使第二个参数和第 3 个参数变得有些不相关了。因为应用程序并不等待函数返回, 所以就没有挂起的危险, 而且这样就不需要 `GL_SYNC_FLUSH_COMMANDS_BIT` 了, 并且如果进行指定的话, 实际上会导致一个错误。实际上超时也是与实现相关的, 并且会明确指定特定超时值 `GL_TIMEOUT_IGNORED`。如果有兴趣的话, 我们还可以通过调用以 `GL_MAX_SERVER_WAIT_TIMEOUT` 为参数的 `glGetInteger64v` 找出我们的实现所使用的超时值。

读者可能会疑惑, “为什么要求 GPU 等待一个同步对象到达管线的终点?” 毕竟, 同步对象在到达管线终点时将变为标记状态, 这样, 如果我们等待它到达管线终点的话, 那么它当然将是标记状态的。这样的话, `glWaitSync` 不是没有做任何事吗? 如果我们仅仅考虑只使用单个 OpenGL 环境而不使用其他 API 的简单应用程序, 这样说是对的。但是, 当我们使用多个 OpenGL 环境时, 同步对象的强大之处就显现出来了。同步对象可以在 OpenGL 环境之间共享, 也可以在诸如 OpenGL 这样的兼容 API 之间共享。这就是说, 通过在一个环境中调用 `glFenceSync` 来创建的同步对象, 可以在另一个环境中通过调用 `glWaitSync` (或者 `glClientWaitSync`) 进行等待。

考虑下面这种情况。我们可以要求一个 OpenGL 环境来推迟某些渲染, 直到其他环境完成某些操作为止, 这样就使两个环境之间可以进行同步了。一个应用程序可以有两个线程和两个环境(还可以更多, 如果我们想要的话)。如果我们在每个环境中创建一个同步对象, 然后在每个环境中使用任意 `glClientWaitSync`

来等待来自其他环境的同步对象,当所有函数都返回时,所有这些环境都会与另一个同步。结合由操作系统提供的线程同步图元(例如信号量(emaphore)),我们可以让多个窗口的渲染保持同步。

在两个环境之间进行共享的一个缓冲区就是这类应用的一个示例。第一个环境使用变换反馈写入缓冲区,而第二个环境则要对变换反馈的结果进行绘制。第一个环境将使用变换反馈模式进行绘制。在调用 `glEndTransformFeedback` 之后,它会立即调用 `glFenceSync`。现在,应用程序会将第二个环境设为当前环境,并调用 `glWaitSync` 等待同步对象变成标记状态。然后,它可以向 OpenGL 发出更多命令(在新的环境中),而驱动程序将对它们进行排序,做好执行准备。

只有当 GPU 在第一个环境中完成了向变换反馈缓冲区记录数据的工作时,它才会在第二个环境中使用这些数据对这些命令进行处理。

在 OpenGL 这样的 API 中,还有一些扩展和其他功能,允许对缓冲区进行同步写入。我们可以使用 `glWaitSync` 来要求一个 GPU 进行等待,直到一个缓冲区中的数据通过在生成这些数据的环境中创建一个同步对象,然后等待这个同步对象在将要使用这些数据的环境中变成标记状态,从而变得有效。

同步对象只能从未标记状态变换到标记状态,并不存在将一个同步对象从标记状态变换到未标记状态的机制,即使是手动也不可以。这是因为对一个同步对象进行手动变换可能会导致竞争条件,并且可能使应用程序挂起。

考虑一个同步对象被创建,到达管线终点,并且转换成标记状态,然后应用程序再将它设置回未标记状态的情况。如果另一个线程试图等待这个同步对象,但是却在应用程序已经将同步对象设置回未标记状态之后才开始等待的话,那么它将一直等待。这样,每个同步对象代表一个一次性事件,而每一次需要进行同步时,必须通过调用 `glFenceSync` 创建一个新同步对象。虽然在使用完对象时对它们进行删除这样的清除工作总是非常重要的,但对于同步对象来说尤其重要,因为我们可能会在每一帧创建很多新的对象。为了删除同步对象,可以调用

```
glDeleteSync(sync);
```

这样就删除了同步对象。这种情况可能不会立即发生;任何等待同步对象变成标记状态的线程在各自的超时时间都仍将继续等待,而一旦没有任何线程等待这个对象的话,这个对象实际上将被删除。这样,调用 `glWaitSync`, 然后再调用 `glDeleteSync` 就是完全合法的,即使同步对象仍然在 OpenGL 管线中。

## 12.8 小结

在本章,我们学习了如何管理大量顶点和其他数据,控制 OpenGL 如何访问这些数据,以及获取 OpenGL 使用这些数据的信息。我们了解了如何存储算法在一次执行过程中生成的数据,并在另一次执行过程中对它进行重用。我们已经有了对多个环境进行同步的工具,所以由一个环境产生的数据可以被其他环境使用。我们了解了绘制一组几何图形大量副本的方法,也学习了如何为 OpenGL 提供数据,以供这些副本中的每一个使用。我们还知道了如何对 OpenGL 管线的操作进行计时,这就使我们能够对应用程序进行怎样的渲染做出明智的决定,以提高性能表现。

## 第三部分 特定平台应用

现在，让我们来讨论 OpenGL 的可移植性。尽管 OpenGL 本身就已经完全是图形硬件的抽象了，它与平台无关，但 OpenGL 总需要与操作系统或窗口系统进行交互。在每个平台中，总是存在一些不可移植的绑定函数，把 OpenGL 连接到本地窗口或显示系统。此外，这些实现总会有一些特定平台的注意事项和特性。本书的这一部分将讨论这些接口。

今天，最流行的 4 种 OpenGL 平台无疑是 Windows、Mac OS X、UNIX 和无数利用 OpenGL 的一个子集——OpenGL ES 的手持式系统。本书将分别用一个特定的章节讨论在每个平台上使用 OpenGL 的特殊之处。OpenGL 现在是最流行的 3D 图形 API，它的足迹几乎遍及每个应用领域或 3D 硬件所出现的每个平台。

OpenGL 随处可见，算算看吧。

## 第 13 章 Windows 上的 OpenGL

作者: Nicholas Haemel

## 本章内容

任 务	使用的函数
请求并选择一个	wglChoosePixelFormatARB/
OpenGL 像素格式	SetPixelFormat
创建和使用 OpenGL	wglCreateContextAttribsARB/
渲染环境	wglDeleteContext/wglMakeCurrent
在 Windows 中使用双缓冲	SwapBuffers

正如我们所看到的, OpenGL 是一种强大的 API。它的低级特性使应用程序开发者获得了所有控制权。另外, OpenGL 核心代码可以在多种不同平台和操作系统之间进行移植。因为每种操作系统都有不同的窗口管理方法, 所以每种操作系统都有一个不同的层次来帮助应用程序与 OpenGL 通过接口进行连接。这样能够帮助驱动程序实现了解应该为任意具体实例使用哪些类型的缓冲区、颜色格式和其他特性。

在微软的 Windows 桌面操作系统(笔记本、便携式计算机、台式机、服务器等)中, 使用了一组特别依赖于所使用 API 的函数, 称为 WGL (Windows-GLE)。WGL 函数的前面带有 wgl 前缀, 代表这些函数是为了 Windows 和 OpenGL 之间的接口而编写的。由于这个前缀, 它们有时候会被称为 wiggle 函数。从现在开始, 我们使用真正的 wgl 函数直接与 Windows 和 OpenGL 驱动程序进行接口连接, 而不再使用 GLUT 库。对于简单应用程序的准备和运行来说, GLUT 非常好, 但是需要付出减少控制权和灵活性的代价。

在本章, 我们学习如何使用 wgl 探索一个系统的性能、创建和管理窗口, 以及处理适用的系统信息。本章内容是逐步进行介绍的, 同时我们会建立一个 OpenGL 示范程序, 这个程序为 Windows 特定的 OpenGL 支持提供了一个框架。

到目前为止, 本书还不需要 3D 图形或 OpenGL 方面的预备知识或者经验。但是对于本章来说, 我们假定读者至少具有入门级的 Windows 编程知识。否则, 我们就得写一本相当于本书两倍厚度的书了。

我们就得花费更多的时间解释为 Windows 编写程序的相关细节,而只能用更少的时间讲解 OpenGL 编程。有很多优秀的图书和资源,详细地讲解了编写 Windows 应用程序的相关细节。

## 13.1 Windows 中的 OpenGL 实现

OpenGL 进入 Win32 平台是在 Windows NT 3.5 发布之后。它是作为 Windows 95 的一种增强功能在稍后发布的,随后成为 Windows 95 操作系统(OSR2)的一部分。

现在,OpenGL 在所有的 Win32 平台(Windows XP、Vista、Win 7、Server 2003、Server 2008 等)上都是一种本地 API,它的功能是从 `opengl32.dll` 导出的。许多层次的 OpenGL 硬件都可以被 Windows 平台所使用,包括从部分 OpenGL 功能在软件中实现的芯片组到入门级视频卡,再到速度极快的工作站级视频卡。应当注意的是,应用程序可以在这些平台中的任何一个运行。

### 13.1.1 微软的 OpenGL

微软当前发布了一种通用 OpenGL 软件实现,作为操作系统的默认版本。如果在一个系统中不存在 3D 硬件,或者没有安装正确的硬件驱动,那么我们得到的就是 OpenGL 实现的微软版本。在很多年的时间中,微软都没有对 OpenGL 投入任何工作,在大多数微软的操作系统中所支持的 OpenGL 版本还是 1.0 或 1.1。

对于大多数现代 3D 应用程序来说,这样是不够的。另外,一个软件实现的处理速度常常不够快,无法支持任何有价值的图形。基于这些原因,很多 OpenGL 应用程序都会检查所支持的 OpenGL 版本,并且在 OpenGL 的更新版本没有得到支持的情况下不会运行。

### 13.1.2 现代图形驱动程序

可安装客户端驱动程序(ICD, Installable Client Driver)最初是为 Windows NT 提供的硬件驱动程序。ICD 必须使用一个特定硬件和为它编写的软件组合来实现完整的 OpenGL 管线。对于供应商来说,从头开始创建一个 ICD 是一项庞大的工程。

ICD 介入并处理微软的 OpenGL 实现。微软会将连接到 `opengl32.dll` 的应用程序自动传递到一个已经为 OpenGL 调用安装的 ICD 驱动。因为存在一个通用接口,驱动程序和应用程序就不必进行编译才能在系统中利用 OpenGL 硬件,即使在它发生改变时也是如此。ICD 实际上是显示驱动程序的一部分,它并不影响已经存在的 `opengl32.dll` 系统 DLL(动态链接库)。这个驱动程序模型为供应商提供大量的机会来对驱动程序和硬件的组合进行优化。

所有主要硬件供应商目前都使用 ICD 模型。如果一个给定的硬件单元不能本地支持 OpenGL 的某些部分,那么 ICD 就必须实现这些缺失的功能。通过这种方式,所有 ICD 驱动程序应该都能支持由这个驱动程序输出的 OpenGL 版本的整个特性集了。

因为 OpenGL 的 `opengl32.dll` 部分会调用属于操作系统的堆栈，所以应用程序和驱动程序必须使用随给定操作系统发行的库。因为微软的软件实现只支持 OpenGL 1.0 或 1.1，所以 `opengl32.dll` 的入口点也只支持同样版本的 OpenGL。随着 OpenGL 的发展和演变，以及新功能的加入，这种情况就使我们陷入了进退两难的窘境。自从 OpenGL 1.1 发布之后的 18 年，我们已经走过了漫长的道路。

因为一个显示驱动程序不能对 `opengl32.dll` 进行修改来为当前版本增加新的特性，所以 OpenGL 就需要一种方法，使应用程序能够对那些并非由 `opengl32.dll` 输出的部分进行访问。这是通过扩展机制和允许应用程序获得任何所支持接口的入口点地址完成的。

这种方式不仅对 OpenGL 的新版本有效，硬件供应商还可以使用这种机制来扩展 OpenGL 的特性集，正如我们稍后将看到的。

## Vista 和 Windows 7 上的 OpenGL

OpenGL 在 Vista 和 Windows 7 上的使用方式与在早期操作系统上的使用方式大体相同。操作系统仍然包含一个 `opengl32.dll` 版本，而应用程序也以大致相同的方式对 OpenGL 函数进行调用。但是，在这些新的操作系统中，桌面组件被用来创建用户所看到的最终图像。在以前的操作系统中，每个窗口都会渲染到它所包含的桌面像素上。但是在 Vista 和 Windows 7 中，每个窗口都会渲染到一个表面上，这个表面将会转交给操作系统的一个新组件，称为桌面窗口管理器（DWM，Desktop Window Manager）。

每个窗口表面都会“出现”在 DWM 中，而 DWM 则直接与图形内核驱动程序通过接口相连接。DWM 接受所有来自每个正在运行的 2D 和 3D 应用程序的窗口，并使用 GPU 将它们与桌面组件进行组合，以创建一个最终图像，也就是使用者看到的图像。这种新机制将每个窗口着色器表面进行分离，并允许操作系统利用高级 GPU 性能来提供漂亮的混合及 3D 效果。

`opengl32.dll` 在 Vista 和 Windows 7 上的版本仍然只支持 OpenGL 1.1。但是，微软已经实现了一个从 OpenGL 到 D3D 的仿真器，支持 1.4 版本的 OpenGL。这个实现看起来很像是一个 ICD，但它只在没有安装真正的 ICD 时才会出现。至于在 Vista 最初发行时，是无法手动开启这个实现的。只有少量游戏（由微软选择）可以通过一定的办法看到这个实现。

就像 XP 一样，Vista 并没有在发行媒介上附带 ICD 驱动程序。但是，当用户从厂商的网站下载一个新的显示驱动程序之后，就可以得到真正的基于 ICD 的驱动程序，并且对窗口游戏和全屏游戏都提供了完全的 OpenGL 支持。

### 13.1.3 扩展 OpenGL

在我们开始了解对 wgl 的复杂使用之前，先来看一看如何对 OpenGL 和 wgl 的核心功能进行扩展。因为核心 `OpenGL32.dll` 只提供了最少量的入口点，所以我们需要了解如何在新的函数中获取它们，以真正使用 wgl 和 OpenGL。我们会学习处理扩展的两种方式：直接使用接口或者让 GLEW 库帮助我们进行处理。

扩展就是指对一个 OpenGL 核心版本增加的任何东西。在 OpenGL 网站上的 OpenGL 扩展注册（OpenGL extension registry）中列出了这些扩展。这些扩展是作为不同的规范来编写的。也就是说，这

些扩展的文本描述了如果要支持这个扩展的话, OpenGL 核心规范必须进行什么样的改变。

一共有 3 种主要的扩展类别, 即供应商扩展、EXT 扩展和 ARB 扩展。供应商扩展是在供应商的硬件上编写和实现的。代表特定供应商的首字母通常会作为扩展名的一部分使用——“AMD”代表“高级微型设备 (Advanced Micro Devices)”, 而“NV”则代表 NVIDIA。可能会有不止一个供应商支持一个特定供应商扩展, 特别是在这个扩展得到广泛支持的情况下。EXT 扩展是由两个或多个供应商一起编写的。它们经常是在最初的时候作为供应商特定扩展出现的。ARB 扩展是 OpenGL 一个正式组成部分, 因为它们是由 OpenGL 的管理机构——体系结构审核委员会 (ARB) 批准的。这些扩展通常被所有主要硬件供应商所支持, 在开始时也是作为供应商扩展或 EXT 扩展出现的。

这种扩展过程乍一看可能颇有些令人困惑。现在已经有了数百种扩展! 但是新的 OpenGL 版本常常会根据程序员认为有用的扩展进行构建。这样, 每个扩展都会有自己的归宿。那些经过考验的扩展会进入核心, 而那些用处不大的则不会被考虑。这种“自然选择”的处理过程有助于确保只有最有用和最重要的新特性能够进入 OpenGL 核心版本。

## 使用扩展

在第 2 章, 我们已经学习了如何查询在给定的系统中哪些扩展是可用的。扩展可以对 OpenGL 功能产生很多不同的影响。它们可以简单地解除某些目前存在的限制; 可以引入新的枚举值, 用于设置状态之类的操作; 还可以向 API 中添加整个新函数。唯一需要特别注意的, 就是那些应用程序必须使用新入口点的情况。

在 Windows 平台上, 我们不会对 OpenGL 驱动程序直接进行访问。OpenGL 3.2 中的 OpenGL 函数调用是通过 OpenGL32.dll 库进行的, 这个函数曾经是 OpenGL 1.1 的一部分。

因为这个 DLL 只能理解 OpenGL 1.1 入口点 (函数名), 所以 OpenGL 驱动程序为我们提供了一种方法, 来获得驱动程序直接支持的所有新 OpenGL 函数。Windows OpenGL 实现提供了一个名为 `wglGetProcAddress` 的函数, 它允许我们对指向一个由驱动程序提供的 OpenGL 函数的指针进行检索。

```
PROC wglGetProcAddress(LPSTR lpzProc);
```

这个函数接受一个 OpenGL 函数或扩展的名称为参数, 并返回一个函数指针, 我们可以用它直接调用这个函数。为此, 必须知道这个函数的原型, 这样就可以创建一个指向它的指针, 并在以后调用这个函数。

扩展的数量非常多, 尤其当添加更新的 OpenGL 核心功能和供应商特定的扩展时。如果要对所有 OpenGL 扩展作完整的介绍, 可能需要整整一本书 (如果还不需要一本百科词典的话)。如果有时间的话, 可以看一看扩展注册, 本书的附录 A 中提供了相应链接。

幸运的是, 我们只要使用下面这两个头文件就可以在程序中访问绝大多数 OpenGL 扩展。

```
#include <wglext.h>
#include <glxext.h>
```

这些文件可以在 OpenGL 扩展注册网站上找到, 但绝大多数图形卡厂商也对它们进行了维护 (参见它们的开发人员支持网站)。对于本书展示的例子来说, 我们使用 GLEW 版本, 它包含在

\\src\\GLTools\\include\\GL\\目录中。wglext.h 头文件包含了一些 Windows 特有的扩展, glext.h 头文件则包含了标准 OpenGL 扩展和许多厂商特定的 OpenGL 扩展。

### 13.1.4 WGL 扩展

我们也可以使用一些 Windows 特有的 WGL 扩展。我们可以像访问其他扩展一样访问 WGL 扩展的入口点, 即使用 wglGetProcAddress 函数。但是, 这里存在一个重要的例外。一般情况下, 在众多 WGL 扩展中, 只有两种可以通过 glGetString(GL\_EXTENSIONS)来获取相关信息。一种是前面提到的交换间隔扩展(允许我们用垂直回溯同步缓冲区交换), 另一种是 WGL\_ARB\_extensions\_string 扩展。后者提供了另一个入口点, 专门用于查询 WGL 扩展。ARB 扩展字符串函数的原型如下。

```
const char *wglGetExtensionsStringARB(HDC hdc);
```

这个函数获取 WGL 扩展列表的方式和 glGetString 相同。使用 wglext.h 头文件, 可以获取一个指向这个函数的指针, 如下所示。

```
PFNWGLGETEXTENSIONSSTRINGARBPROC *wglGetExtensionsStringARB;  
wglGetExtensionsStringARB = (PFNWGLGETEXTENSIONSSTRINGARBPROC)  
    wglGetProcAddress("wglGetExtensionsStringARB");
```

glGetString 返回 WGL\_ARB\_extensions\_string 标识符, 但开发人员常常忽略对这个值的检查, 而只是简单地使用入口点, 如前面的代码段所示。

在绝大多数 OpenGL 扩展中, 这种方法是安全的。但是我们也应该认识到, 严格地讲, “我们可能在直线之外着色”。有些厂商导出一些“试验性”的扩展, 并且这些扩展可能并没有得到官方的支持, 或者如果我们跳过扩展字符串检查的话, 这些函数可能不会正确地运行。另外, 可能多个扩展使用相同的函数。如果只测试函数的可用性, 就无法知道自己想要使用的某种特定扩展是否得到支持。

#### 综合运用

大多数普通的开发者都会很快地厌倦于总是必须在程序开始时查询新的函数指针。还有一种更快的方法, 实际上, 到目前为止我们在本书的所有示例中使用的都是这种快捷方式。GLEW (GL Extension Wrangler) 库和为本书发布的源代码一起包含在 GLTools 目录中, 要想自动访问驱动程序支持的所有函数指针非常简单, 只需在项目中添加 glew.c, 并且在头文件列表的顶部添加 glew.h。然后, 在应用程序开始进行任何 OpenGL 调用之前先调用 glewInit()。OpenGL 1.1 以上版本的扩展和核心特性的所有函数指针都将自动进行设置。如果函数失败, 它会返回一个错误, 扩展指针则可能不会被初始化。

```
GLenum err = glewInit();  
if (GLEW_OK != err)  
{  
    /* Problem: glewInit failed, something is seriously wrong. */  
    fprintf(stderr, "Error: %s\n", glewGetErrorString(err));  
}
```

使用 GLEW 就不再需要 Windows 中的特殊驱动程序为了对所有 OpenGL 功能进行访问而执行专门

的初始化了。但是，这样并不能避免对驱动程序当前支持的是哪个 OpenGL 版本进行检查。如果在驱动程序中并不存在某个入口点，那么指向这个入口点的函数指针将为 NULL，而调用这个函数则将使程序崩溃。这对于本地系统来说也许不算什么，但是我们肯定不希望移植给朋友或客户的程序就因为他们的系统使用了比较老的硬件而崩溃。虽然我们在本书使用的是 GLEW 和相关的示例程序，但是其他工具也提供了类似的扩展加载支持。

## 13.2 基本窗口渲染

现在让我们回过头使用 wgl 设置应用程序。通常使用的 GLUT 库只提供了一个窗口，而 OpenGL 函数调用总是在这个窗口中生成输出（不在这个窗口中还能在哪儿呢？）。但是，实际 Windows 应用程序经常会有不止一个窗口。实际上，对话框、控件甚至是菜单都是基础级的窗口；而一个有用的程序只包含一个窗口的情况几乎是不可能发生的（好吧，可能游戏是一种重要的例外！）。GLUT 也要求使用控制函数 `glutMainLoop()`。这样对于简单的应用程序来说可能应用得很好，但是对于库来说，或者当代码不控制主事件循环时，就无法应付了。让我们了解一下更加灵活的窗口和环境管理方式吧。

### 13.2.1 GDI 设备环境

有许多方法可以用来在微软操作系统的窗口中进行绘图。最为古老并且受到最广泛支持的是 Windows GDI（图形设备接口）。GDI 后来经过更新而发布了 GDI+。严格地讲，GDI 是个 2D 绘图接口，在 Windows Vista 出现之前得到了广泛的硬件加速支持。尽管 Vista 和 Windows 7 仍然可以使用 GDI，但它不再以同样的方式进行硬件加速。更先进的高层绘图技术基于 .NET 框架，称为 Windows Presentation Foundation（WPF）。Windows XP 也可以通过下载使用 WPF。

在过去的几年中，有些 2D API 一度出现又逐渐消失，其中包括 Direct3D 的一些变体。在 Vista 中，新的底层渲染接口称为 Windows Graphics Foundation（WGF），它在本质上只是 Direct3D 的版本 10。

GDI 是一种对所有版本的 Windows（甚至包括 Windows Mobile）都通用的本地渲染 API。这很幸运，因为我们在所有版本的 Windows（除了 Windows Mobile，Microsoft 并未对它提供 OpenGL 的本地支持）上都是用 GDI 对 OpenGL 进行初始化并与 OpenGL 进行交互的。在 Vista 和 Windows 7 中，GDI 不再是硬件加速，但这并没有关系，因为我们实际上根本不会用 GDI 进行任何绘图操作（至少在使用 OpenGL 时是这样）。

当我们使用 GDI 时，每个窗口都具有一个实际接受图形输出的设备环境，并且每个 GDI 函数都接受一个设备环境为参数，表示这个函数将作用于哪个窗口。我们可以使用多个设备环境，但每个窗口只能有一个设备环境。

读者可能会得出一个结论，认为 OpenGL 的工作方式应该与此类似，但是请记住 GDI 是 Windows 特有的。OpenGL 的设计目标就是实现在各种环境和硬件平台上的完全可移植性。（它并不是为 Windows

量身定做的!) 如果向 OpenGL 函数中添加一个设备环境作为参数, 那么在任何 Windows 之外的环境中, 这种 OpenGL 代码就会变得毫无用处。

但是, OpenGL 也有一个环境标识符, 称为渲染环境 (Rendering Context)。OpenGL 渲染环境在许多方面都和 GDI 的设备环境非常相似, 因为正是渲染环境负责记住当前的颜色、状态设置等信息, 就像 Windows 的设备环境需要记住当前的画刷或画笔颜色一样。

## 13.2.2 像素格式

在 Windows 中, 设备环境的概念对于 3D 图形的支持十分有限, 因为它的设计目标就是用于 2D 图形应用程序。在 Windows 中, 我们为一个特定的窗口申请一个设备环境标识符。设备环境的本质取决于设备的本质。

如果桌面设置为 16 位的颜色, 那么 Windows 所提供的设备环境就只知道和理解 16 位颜色。例如, 我们不能告诉 Windows, 一个窗口是 16 位颜色的, 另一个窗口则是 32 位颜色的。程序员无法控制一个窗口设备环境的本质特征。

任何实现 3D 图形渲染的窗口或设备应该具备的特征远远不止颜色深度那么简单。直到现在, 我们都是让 GLUT 负责这些细节的。

当我们初始化 GLUT 时, 必须告诉它需要什么缓冲区 (双颜色缓冲区或单颜色缓冲区, 以及深度、模版和 alpha 缓冲区)。

在 OpenGL 可以对窗口进行渲染之前, 首先必须根据渲染需要对窗口进行配置。渲染使用单缓冲还是双缓冲? 是否需要深度缓冲区? 是否需要模版或目标 alpha? 我们需要哪个版本的 OpenGL? 当我们为窗口设置了这些参数之后, 以后就无法再对它们进行修改了。为了从一个只有深度缓冲区和颜色缓冲区的窗口切换到一个只有模版和颜色缓冲区的窗口, 必须首先销毁第一个窗口, 然后再根据自己所需要的特征重新创建一个新窗口。

Windows 上的 OpenGL 使用像素格式将所有这些信息封装到分组功能中。我们需要寻找一种特性和功能都与应用程序的需要相匹配的像素格式。然后, 这种像素格式会被用来创建一个 OpenGL 渲染环境。我们可以选择使用两种方法来寻找像素格式。第一种方法是由 OpenGL 直接提供的, 这种机制功能更强, 也更受推荐。第二种方法使用原始的 Windows 接口, 这种方法在 Windows 刚开始支持 OpenGL 的时候就出现了。

### 查询像素格式的新方法

一个窗口的像素格式可以通过一个以 1 为基准的整数索引值进行指定。实现可以输出要从中进行选择的像素格式数量。Windows 对 OpenGL 的接口并没有随着 OpenGL 的发展而发展。结果就是, OpenGL 中新增的特性就无法使用传统的 Windows 函数进行访问了。

幸运的是, OpenGL 增加了一种方式, 可以获得这些新特性。这种新机制还提供了高级搜索功能,

可以节省我们用来为应用程序寻找正确像素格式的时间。

现在我们可以使用第一个可能也是最重要的一个 wgl 扩展了。WGL\_ARB\_pixel\_format 扩展提供了一个机制，使我们可以检查和选择像素格式特性，这些特性超越了 Windows 所能访问的范围。例如，我们可以使用这种扩展来寻找一种支持多重采样渲染的像素格式。

这种扩展定义了大量可以与环境属性相联系的属性，如表 13.1 所示。我们可以使用函数来寻找能够满足我们要求的像素格式。

```
BOOL wglChoosePixelFormatARB(HDC hdc, const int *piAttribIList,
                             const float *pfAttribFList, UINT nMaxFormats,
                             const int *piFormats, UINT *nNumFormats);
```

要注意这个函数的“ARB”后缀，这非常重要。wglChoosePixelFormatARB 与 wglChoosePixelFormat 是不同的，我们应该总是使用 wglChoosePixelFormatARB。

还要注意，我们必须先创建一个 OpenGL 环境，然后才能对这个扩展进行设置并调用 wglChoosePixelFormatARB。要完成这项工作，我们可以创建一个虚拟环境，这个环境在找到所需像素格式时就会马上删除。

这里有许多属性需要处理。第一个参数 hdc 是要使用这种像素格式的窗口的设备环境。第二个参数和第 3 个参数则用来指定正在搜索的属性。这两个参数都是属性和数值对的列表。piAttribIList 是一个整数值列表，而 pfAttribFList 则是一个浮点值列表。某些属性定义成浮点数比定义成整数要好，它们都以 null 结束。

要使用这些属性，我们要创建一个只包含一种类型的数组，然后将第一个索引设置成我们想要指定的第一个属性的值。将第二个索引设置成需要的最小值。重复这个过程，将第 3 个索引设置成第二个属性值，以此类推。当我们添加了所有的属性后，就要在数组的末尾添加一个 null。某些属性会要求精确的匹配，例如 WGL\_DRAW\_TO\_WINDOW\_ARB 和 WGL\_SWAP\_METHOD。我们指定的某些属性值是一个最低值，例如 WGL\_COLOR\_BITS\_ARB 和 WGL\_ALPHA\_BITS\_ARB。

我们必须分配另外的一个数组来保存搜索结果。然后，再将结果数组的大小传递到 nMaxFormats，并将一个指向这个整数数组的指针传递到 piFormats。

写入到结果数组中的格式的实际数量会被传回到 nNumFormats 参数中。通常情况下这也是查找到格式的数量，但是如果数组太小，并且 nNumFormats 与 nMaxFormats 相同，那么驱动程序就找到了更多匹配的格式，结果数组中已经容纳不下这些格式了。如果我们没有在 piAttribIList 或 pfAttribFList 中指定一个属性，那么这个函数在寻找匹配项时就会忽略它，这里不会使用任何默认设置。如果将 null 传递到 piAttribIList 和 pfAttribFList，那么我们将获得所有支持的格式。

wglChoosePixelFormatARB 在 piFormats 属性中返回的结果会以“最佳”匹配格式存储到这个列表的开始位置。“最佳”匹配是由实现定义的，并且与设备有关。通常情况下，选择实现认为是最佳匹配的格式是有利的，只要它们能够满足我们应用程序的要求。

在大多数查询中，某些属性是必需的，这样才能使结果得到的像素格式变得有用。

大多数程序应该会要求 WGL\_SUPPORT\_OPENGL\_ARB、WGL\_DRAW\_TO\_WINDOW\_ARB 和

WGL\_ACCELERATION\_ARB 属性。下一节将更详细地讨论这些属性。所有这些信息看起来可能有些让人困惑,但实际上寻找一种像素格式比看起来要容易。程序清单 13.1 展示了一个如何选择像素格式的示例。

程序清单 13.1 寻找一个像素格式来满足我们的需要

```
int nPixCount = 0;

// 指定我们关心的重要属性
int pixAttribs[] = {
    WGL_SUPPORT_OPENGL_ARB, 1, // 必须支持 OGL 渲染
    WGL_DRAW_TO_WINDOW_ARB, 1, // 能够运行一个窗口的像素格式
    WGL_RED_BITS_ARB, 8, // 窗口中的 8 位红色精度
    WGL_GREEN_BITS_ARB, 8, // 窗口中的 8 位绿色精度
    WGL_BLUE_BITS_ARB, 8, // 窗口中的 8 位蓝色精度
    WGL_DEPTH_BITS_ARB, 16, // 窗口的 16 位深度精度
    WGL_ACCELERATION_ARB,
    WGL_FULL_ACCELERATION_ARB, // 必须为硬件加速
    WGL_PIXEL_TYPE_ARB,
    WGL_TYPE_RGBA_ARB, // 像素格式必须为 RGBA 类型
    0; // 以 NULL 结束

// 要求 OpenGL 寻找与我们的属性相匹配的最佳格式
// 只取回一种格式
wglChoosePixelFormatARB(g_hDC, &pixAttribs[0], NULL, 1, &nPixelFormat,
    (UINT*)&nPixCount);

if(nPixelFormat == -1)
{
    // 无法找到格式,可能没有安装硬件或驱动程序
    g_hDC = 0;
    g_hDC = 0;
    bRetVal = false;
    printf("!!! An error occurred trying to find a pixel format with the requested
    attribs.\n");
}
```

## 像素格式属性

在应用程序已经选择了一种像素格式之后,或者当查看整个列表时,就能使用 wglGetPixelFormatAttribvARB 和 wglGetPixelFormatAttribfvARB 函数获取一个像素格式的任何特定属性的信息了。

```
BOOL wglGetPixelFormatAttribvARB(HDC hdc, int iPixelFormat,
    int iLayerPlane, UINT nAttributes,
    const int *piAttributes, int *piValues);
BOOL wglGetPixelFormatAttribfvARB(HDC hdc, int iPixelFormat,
    int iLayerPlane, UINT nAttributes,
    const int *piAttributes, float *pfValues);
```

它们是同一个函数的两种变体,允许我们查询一个特定的像素格式索引值,并获取一个包含了这种像素格式的属性数据数组。第一个参数 hdc 是将要使用这种像素格式的窗口的设备环境,而第二个参数则是这种像素格式的索引值。iLayerPlane 参数指定了需要查询哪个层平面(如果是 Vista、Windows 7,或者我们所使用的 OpenGL 实现并不支持层平面,就把这个参数设置为 0)。下一个参数 nAttributes 指定了这种像素格式要查询多少个属性,而 piAttributes 数组则包含了需要查询的属性名的列表。表 13.1 列出了可以指定的属性。最后一个参数是个数组,它将由对应的像素格式属性填充。

表 13.1

像素格式属性

常 量	描 述
WGL_NUMBER_PIXEL_FORMATS_ARB	这种设备的像素格式数量
WGL_DRAW_TO_WINDOW_ARB	如果这种像素格式可以在一个窗口中使用, 设为非零值
WGL_DRAW_TO_BITMAP_ARB	如果这种像素格式可以在一个内存设备独立的位图 (DIB) 中使用, 设为非零值
WGL_DEPTH_BITS_ARB	深度缓冲区的位数
WGL_STENCIL_BITS_ARB	模板缓冲区的位数
WGL_ACCELERATION_ARB	表 13.2 中的一个值, 指定所使用的一种硬件驱动程序 (如果有的话)
WGL_NEED_PALETTE_ARB	如果需要调色板, 设为非零值
WGL_NEED_SYSTEM_PALETTE_ARB	如果硬件只有在 256 色模式色支持一个调色板, 设为非零值
WGL_SWAP_LAYER_BUFFERS_ARB	如果硬件支持交换层平面, 设为非零值
WGL_SWAP_METHOD_ARB	在双缓冲的像素格式中实现缓冲区交换的方法, 它是表 13.5 所列的其中一个值
WGL_NUMBER_OVERLAYS_ARB	上层平面的数量
WGL_NUMBER_UNDERLAYS_ARB	下层平面的数量
WGL_SAMPLES_ARB	每个像素中多重采样的样本数量, 默认值为 1
WGL_TRANSPARENT_ARB	如果支持透明, 设为非零值
WGL_TRANSPARENT_RED_VALUE_ARB	透明红色
WGL_TRANSPARENT_GREEN_VALUE_ARB	透明绿色
WGL_TRANSPARENT_BLUE_VALUE_ARB	透明蓝色
WGL_TRANSPARENT_ALPHA_VALUE_ARB	透明 alpha 颜色
WGL_SHARE_DEPTH_ARB	如果层平面与主平面共享一个深度缓冲区, 设为非零值
WGL_SHARE_STENCIL_ARB	如果层平面与主平面共享一个模板缓冲区, 设为非零值
WGL_SHARE_ACCUM_ARB	如果层平面与主平面共享一个累积缓冲区, 设为非零值
WGL_SUPPORT_GDI_ARB	如果支持 GDI 渲染 (只是前缓冲区), 设为非零值
WGL_SUPPORT_OPENGL_ARB	如果支持 OpenGL, 设为非零值
WGL_DOUBLE_BUFFER_ARB	如果是双缓冲, 设为非零值
WGL_STEREO_ARB	如果支持左右缓冲区, 设为非零值
WGL_PIXEL_TYPE_ARB	在 RGBA 颜色模式中, 使用 WGL_TYPE_RGBA_ARBW, 在颜色索引模式中, 使用 GL_TYPE_COLORINDEX_ARB
WGL_COLOR_BITS_ARB	颜色缓冲区中位平面的数量
WGL_RED_BITS_ARB	颜色缓冲区中红色位平面的数量
WGL_RED_SHIFT_ARB	红色位平面的移位置
WGL_GREEN_BITS_ARB	颜色缓冲区中绿色位平面的数量
WGL_GREEN_SHIFT_ARB	绿色位平面的移位置
WGL_BLUE_BITS_ARB	颜色缓冲区中蓝色位平面的数量
WGL_BLUE_SHIFT_ARB	蓝色位平面的移位置
WGL_ALPHA_BITS_ARB	颜色缓冲区中 alpha 位平面的数量
WGL_ALPHA_SHIFT_ARB	alpha 位平面的移位置

表 13.2 用于 WGL\_ACCELERATION\_ARB 的加速标志

常 量	描 述
WGL_NO_ACCELERATION_ARB	软件渲染，无加速
WGL_GENERIC_ACCELERATION_ARB	通过一个 MCD 驱动程序加速
WGL_FULL_ACCELERATION_ARB	通过一个 ICD 驱动程序加速

表 13.3 用于 WGL\_SWAP\_METHOD\_ARB 的缓冲区交换值

常 量	描 述
WGL_SWAP_EXCHANGE_ARB	交换前后缓冲区
WGL_SWAP_COPY_ARB	后缓冲区复制到前缓冲区
WGL_SWAP_UNDEFINED_ARB	后缓冲区复制到前缓冲区，但后缓冲区的内容在缓冲区交换之后处于未定义状态

但是，在这里（以及对于所有其他 OpenGL 扩展来说）也有所谓的“22 条军规”。我们必须有一个合法的 OpenGL 渲染环境，然后才能调用大多数 OpenGL 函数的 `glGetString` 或 `wglGetProcAddress`。这意味着我们必须首先创建一个临时窗口，设置一种像素格式（实际上我们可以蒙混过关，只指定像素格式 1，它将为第一个硬件加速格式），然后获取和使用指向其中一个 `wglGetPixelFormatAttribARB` 函数的指针。在程序的登录屏幕或展现给用户的初始选项对话框中完成这些任务可能会很方便。但是，我们不应该试图使用 Windows 的桌面，因为应用程序并不拥有它！

下面这个简单的例子查询一个属性，即受到支持的像素格式的数量，这样我们就知道需要观察多少种像素格式了：

```
int attrib[] = { WGL_NUMBER_PIXEL_FORMATS_ARB };
int nResults[1] = {0};
int pixFmt = 1;
wglGetPixelFormatAttribARB (hDC, pixFmt, 0, 1, attrib, nResults);
// nResults[0] 现在包含了被导出的像素格式的数量
```

关于更详细的展示如何查找一种特定像素格式（包括一种多重采样像素格式）的例子，请参看稍后的“`sphere_world_redux`”示例程序。

我们要理解，为 OpenGL 获取的所有入口点只对当前 OpenGL 环境是合法的，这一点很重要。如果我们删除一个环境并创建一个新的环境，就应该重新填充入口点。不同环境之间的入口点也可以不同，特别是当创建的环境支持 OpenGL 的不同版本，或者由多个图形加速卡驱动不同监视器时。

### 查询像素格式的老方法

Windows 还提供了几个函数，可以用来寻找 OpenGL 像素格式。但是这些方法非常有限，并且不能支持所有的格式或所有的属性。为了内容的完整性，我们也将展示如何使用它们。如果想要编写一个新的 OpenGL 程序，最好使用刚才描述的方法。很多最新的 OpenGL 特性（例如多重采样缓冲区）都无法通过老的像素格式选择模式来访问。

窗口的 3D 特征是一次性设置的，通常就在创建窗口之后进行，这些设置集合的名称就是像素格式。Windows 提供了一个称为 `PIXELFORMATDESCRIPTOR` 的结构，用于描述像素格式。这个结构的定义如下所示。

```

typedef struct tagPIXELFORMATDESCRIPTOR {
    WORD nSize;           // 这个结构的大小
    WORD nVersion;        // 结构的版本 (应该是 1)
    DWORD dwFlags;        // 像素缓冲区属性
    BYTE iPixelFormat;    // 像素数据的类型 (RGBA 或 颜色)
    BYTE cColorBits;      // 颜色缓冲区中颜色位平面的数量
    BYTE cRedBits;        // 用多少位表示红色
    BYTE cRedShift;       // 红色位的移位计数
    BYTE cGreenBits;      // 用多少位表示绿色
    BYTE cGreenShift;     // 绿色位的移位计数
    BYTE cBlueBits;       // 用多少位表示蓝色
    BYTE cBlueShift;      // 蓝色位的移位计数
    BYTE cAlphaBits;      // 用多少位表示目标 Alpha 值
    BYTE cAlphaShift;     // 目标 Alpha 值的移位计数
    BYTE cAccumBits;      // 用多少位表示累积缓冲区
    BYTE cAccumRedBits;   // 用多少位表示累积缓冲区中的红色
    BYTE cAccumGreenBits; // 用多少位表示累积缓冲区中的绿色
    BYTE cAccumBlueBits;  // 用多少位表示累积缓冲区中的蓝色
    BYTE cAccumAlphaBits; // 用多少位表示累积缓冲区中的 Alpha
    BYTE cDepthBits;      // 用多少位表示深度缓冲区
    BYTE cStencilBits;    // 用多少位表示模版缓冲区
    BYTE cAuxBuffers;     // 多少个辅助缓冲区
    BYTE iLayerType;      // 已过时—忽略
    BYTE bReserved;       // 上层或下层平面的数量
    DWORD dwLayerMask;    // 已过时—忽略
    DWORD dwVisibleMask;  // 下层平面的透明颜色
    DWORD dwDamageMask;   // 已过时—忽略
} PIXELFORMATDESCRIPTOR;

```

对于一个特定的 OpenGL 设备 (硬件或软件) 来说, 这些成员的值并不是随意的。一个特定的窗口只能使用有限数量的像素格式。像素格式是由 OpenGL 驱动程序或软件渲染器导出的。在老的软件实现中, 应用程序曾经使用 `ChoosePixelFormat` 从微软获得一种像素格式, 这是通过填充一个 `PIXELFORMATDESCRIPTOR` 并调用 `ChoosePixelFormat` 来完成的。虽然这个调用仍然可以使用, 但我们最好使用由 OpenGL 驱动程序自身实现的 `wglChoosePixelFormatARB`。

`wglChoosePixelFormatARB` 能够返回微软接口所不能返回的格式, 但是我们仍然在这里展示了 `ChoosePixelFormat`, 以备在老旧的应用程序中遇到它。

```

PIXELFORMATDESCRIPTOR pfd;
// 在这里填充 pfd
int nPfd = ChoosePixelFormat(g_hdc, &pfd);

```

### 枚举像素格式

读者可能已经注意到了, 在表 13.1 中有一个值能够通过 `wglGetPixelFormatAttribvARB` 和 `wglGetPixelFormatAttribfvARB` 进行查询, 这个值就是 `WGL_NUMBER_PIXEL_FORMATS_ARB`。我们可以对 `wglGetPixelFormatAttribvARB` 进行一次初始调用, 来获取格式总数, 然后使用这些信息来遍历整个列表并查询我们感兴趣的关于每个可用像素格式的信息。

```

GLint pfAttribCount[] = { WGL_NUMBER_PIXEL_FORMATS_ARB
};
GLint pfAttribList[] = { WGL_DRAW_TO_WINDOW_ARB,
                        WGL_ACCELERATION_ARB,

```

```

        WGL_SUPPORT_OPENGL_ARB,
        WGL_DOUBLE_BUFFER_ARB,
        WGL_DEPTH_BITS_ARB,
        WGL_STENCIL_BITS_ARB,
        WGL_RED_BITS_ARB,
        WGL_GREEN_BITS_ARB,
        WGL_BLUE_BITS_ARB,
        WGL_ALPHA_BITS_ARB
    };

    int nPixelFormatCount = 0;
    wglGetPixelFormatAttribivARB(g_hDC, 1, 0, 1, pfAttribCount,
                                &nPixelFormatCount);
    for (int i=0; i<nPixelFormatCount; i++)
    {
        GLint results[10];
        printf("Pixel format %d details:\n", nPixelFormatCount);
        wglGetPixelFormatAttribivARB(g_hDC, i, 0, 10, pfAttribList, results);
        printf(" Draw to Window = %d:\n", results[0]);
        printf(" HW Accelerated = %d:\n", results[1]);
        printf(" Supports OpenGL = %d:\n", results[2]);
        printf(" Double Buffered = %d:\n", results[3]);
        printf(" Depth Bits = %d:\n", results[4]);
        printf(" Stencil Bits = %d:\n", results[5]);
        printf(" Red Bits = %d:\n", results[6]);
        printf(" Green Bits = %d:\n", results[7]);
        printf(" Blue Bits = %d:\n", results[8]);
        printf(" Alpha Bits = %d:\n", results[9]);
    }

```

这段代码能够打印出一个像素格式清单，但是如果不希望使用由 `wglChoosePixelFormatARB` 提供的更加自动化的方式，我们也可以同样的方法来选择像素格式。

### 选择并设置一个像素格式

在通过 `wglChoosePixelFormatARB` 和 `wglGetPixelFormatAttribARB` 找到一个像素格式之后，就可以告诉 Windows 和 OpenGL 驱动程序我们想要使用哪种像素格式了。我们使用 `SetPixelFormat` 函数完成这项工作。

```

int nPixelFormat;
.....
static PIXELFORMATDESCRIPTOR pfd;
// 为设备环境设置像素格式
SetPixelFormat(hDC, nPixelFormat, &pfd);

```

`PIXELFORMATDESCRIPTOR` 的初始内容并不影响 `SetPixelFormat` 函数的功能。在 `hDC` 参数中传入窗口设备环境句柄，并在 `nPixelFormat` 参数中传入选择的像素格式。`SetPixelFormat` 只能被调用一次。要改变像素格式，窗口必须被销毁并重新创建。

## 13.2.3 OpenGL 渲染环境

一个典型的 Windows 应用程序可以由许多窗口组成。如果愿意，甚至可以为每个窗口设置不同的像素格式（使用每个窗口的设备环境）。当我们调用一条 OpenGL 命令时，驱动程序如何知道将输

出发送给哪个窗口呢？在前面的章节中，我们使用了 GLUT 框架，它提供了一种简单的窗口来显示 OpenGL 输出。请记住，在正常的基于 GDI 的 Windows 绘图中，每个窗口都有自己的设备环境。

为了实现核心 OpenGL 函数的可移植性，每种环境必须实现一些方法，在执行任何 OpenGL 命令之前指定一个当前的渲染窗口。就像 Windows GDI 函数使用窗口设备环境一样，OpenGL 环境也提供了一个称为渲染环境的概念。渲染环境能够记住 OpenGL 的设置和命令。

在过去的 15 年，曾经发布了很多不同的 OpenGL 版本，其中某些版本是不向下兼容的。因此，我们可以选择应用程序将使用的特定 OpenGL 版本。如果 OpenGL 不允许我们这样做，那么当一个 OpenGL 新版本发布，并且这个新版本与设计应用程序所针对的版本不兼容的时候，应用程序就可能无法工作了。我们可以通过调用另一个扩展 `wglCreateContextAttribsARB` 来创建一个 OpenGL 渲染环境。

```
HGLRC wglCreateContextAttribsARB(HDC hDC, HGLRC hShareContext, const int
*attribList);
```

OpenGL 渲染环境的数据类型是 HGLRC。如果一切顺利的话，就会返回新的环境句柄。OpenGL 能够在各个环境之间共享对象（纹理、FBO、顶点数组等）。如果想要在两个或更多环境之间共享对象，那就需要将一个已经创建的环境的环境句柄传入 `hShareContext` 参数。如果我们将 `NULL` 传入新的环境，那么其他已经存在的环境就不会与新的环境共享数据了。

参数是一个属性的数值对列表，我们可以在一个新环境中对其进行查询。首先，在这个数组中指定属性名称，然后指定属性值。`WGL_CONTEXT_MAJOR_VERSION_ARB` 和 `WGL_CONTEXT_MINOR_VERSION_ARB` 属性用来要求特定的 OpenGL 环境版本。如果应用程序是基于 OpenGL 3.3 编写的，那么我们就需要在主版本号中传入 3，在次版本号中也传入 3。类似地，如果应用程序更老，我们需要一个 OpenGL 3.0 环境，也可以对此进行要求。但是，OpenGL 驱动程序也可以返回任何百分之百向下兼容与我们所要求版本的更高版本。如果我们不指定一个 OpenGL 版本，或者如果我们要求 1.0 版，那么驱动程序可能将会创建一个 OpenGL 3.1 环境。具体的行为在各个供应商之间有所不同，最好的办法就是要求特定的 OpenGL 版本。对于新创建的应用程序来说，应该要求 OpenGL 3.3 或更新的版本。

我们只能创建一个 OpenGL 驱动程序支持版本下的环境。可以通过调用带有 `GL_VERSION` 枚举值的 `glGetString` 来查询能够支持的最新版本。

```
ubyte *verString = glGetString(GL_VERSION);
```

或者也可以通过 `glGetIntegerv` 命令查询版本，这个命令会将版本号作为整数部分返回。

```
int majorVer, minorVer;
glGetIntegerv(GL_MAJOR_VERSION, &majorVer);
glGetIntegerv(GL_MINOR_VERSION, &minorVer);
```

我们还可以通过 `attrib_list` 查询一些其他类型的属性。`WGL_CONTEXT_PROFILE_MASK_ARB` 属性后面是一个位段，其中包含 `WGL_CONTEXT_CORE_PROFILE_BIT_ARB` 或 `WGL_CONTEXT_COMPATIBILITY_PROFILE_BIT_ARB`，每一次只能使用一位。

设置 `WGL_CONTEXT_CORE_PROFILE_BIT_ARB` 位会导致驱动程序返回一个只包含核心功能的环境，而不会包含任何“不推荐”的 OpenGL 功能。如果是为了 OpenGL 的下次修订（“不推荐”的 OpenGL 功能可能会被删除）准备应用程序的话，那么使用这个位是一个好办法。设置 `WGL_CONTEXT_COMPATIBILITY_`

PROFILE\_BIT\_ARB 位会要求驱动程序创建一个能够兼容所有 OpenGL 老版本的环境。换句话说,就是不会删除任何“不推荐”的功能。设置这个位而创建的环境可能会比一个核心版本的环境运行速度慢,因为需要跟踪额外的状态和功能。

WGL\_CONTEXT\_FLAGS\_ARB 属性能够用来为环境创建设置其他标志。

这里唯一支持的标志就是 WGL\_CONTEXT\_DEBUG\_BIT。指定这个位所创建的环境能够在应用程序开发过程中为其提供可用的调试信息。至于提供什么样的信息,以及如何访问这些信息,就是由供应商指定的了。

如果系统中的 OpenGL 驱动程序不支持我们指定的任何属性,那么 wglCreateContextAttribsARB 将返回 null,并且会生成一个错误。如果向下兼容环境中主版本号和次版本号的组合不是一个有效的 OpenGL 版本号,就会抛出 WGL\_ERROR\_INVALID\_VERSION\_ARB 错误。

如果为 WGL\_CONTEXT\_PROFILE\_MASK\_ARB 指定的任何一位都不被支持,就会抛出 WGL\_ERROR\_INVALID\_PROFILE\_ARB 错误。

### 调试环境

使用一个调试环境,能够帮助我们确定应用程序在哪里出现了问题。在本书编写时,AMD 是唯一提供了调试环境的供应商,这个调试环境是通过一个称为 GL\_AMD\_debug\_context 的扩展提供的,这个扩展对开发者如何访问这些附加调试信息进行了定义。

这里提供了一个回调函数,允许一个应用程序设置一个中断或中断点,并且在出现错误时能够立即发现。这个扩展还允许一个应用程序选择特定错误类型来进行监视,并且支持多个严重性等级。

关于如何使用 GL\_AMD\_debug\_context 扩展的更多信息,可以查询扩展规范,它也包含在了 OpenGL 扩展注册中。

### 使用环境

为窗口创建一个渲染环境,这个渲染环境要与这个窗口兼容。我们可以在应用程序中创建一个以上的渲染环境——例如两个窗口分别使用不同的绘制模式等。但是,为了使 OpenGL 命令知道它们正在哪个窗口进行操作,每个线程在同一时间只能激活一个渲染环境。当渲染环境被激活时,它就成为了当前的 (current) 渲染环境。

当前渲染环境与设备环境相关,因而也与特定窗口相关。现在,OpenGL 已经知道应该直接渲染到哪个窗口中了。

我们甚至可以将一个 OpenGL 渲染环境从一个窗口移动到另一个窗口,但是每个窗口都必须采用相同的像素格式。我们可以调用 wglMakeCurrent 函数,来将一个渲染环境设置为当前渲染环境,并将它与一个特定窗口相关联。这个函数接受两个参数,即窗口的设备环境和 OpenGL 渲染环境。

```
void wglMakeCurrent(g_hDC, g_hRC);
```

我们可以调用带有 hDC 并且 hRC 值为 null 的 wglMakeCurrent 函数,来将一个渲染环境解除绑定。如果我们完成了渲染或者正在退出应用程序,就应该删除 OpenGL 环境来释放那些保留的资源。一旦一

个环境不再是当前环境，我们可以使用 `wglDeleteContext` 来销毁这个 OpenGL 环境。

```
bool wglDeleteContext(g_hRC);
```

请记住，每个线程在同一时间只能有一个当前渲染环境，这一点非常重要。但是，在不同的线程中可以同时存在两个不同的当前环境。我们甚至可以在多个环境中共享对象。在我们创建第二个环境时，要在第二次调用 `wglCreateContextAttribsARB` 时传入第一个环境的句柄。

```
GLint attribs[] = {WGL_CONTEXT_MAJOR_VERSION_ARB, 3,
                  WGL_CONTEXT_MINOR_VERSION_ARB, 3,
                  0};
HGLRC oglRC1 = wglCreateContextAttribsARB(g_hDC, 0, attribs);
HGLRC oglRC2 = wglCreateContextAttribsARB(g_hDC, oglRC1, attribs);
```

### 过去的环境

一个应用程序也可以使用 `wglCreateContext` 创建 OpenGL 环境。这个旧函数不允许我们确切地指定想要的参数。实际上，我们很难判断将会获得那个版本的 OpenGL。出于上述原因，我们最好为应用程序使用 `wglCreateContextAttribsARB`。但是，很多老的应用程序可能仍然使用老版本。

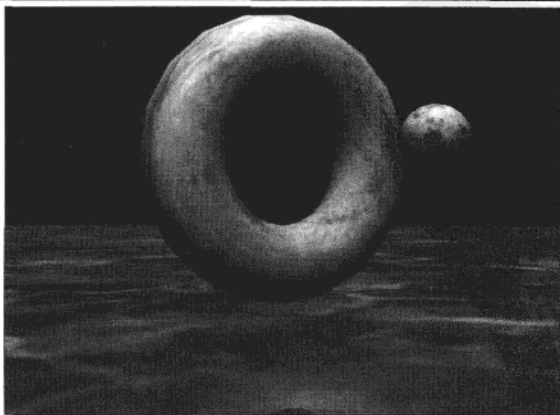
```
HGLRC wglCreateContext(g_hDC);
```

## 13.3 综合运用

在前面的内容中，已经介绍了很多背景知识。我们已经分别描述了这些拼图中的每一块，现在将它们拼到一起。除了查看所有 OpenGL 相关代码之外，还应该检查一些任何 Windows 程序要支持 OpenGL 都要满足的最低要求。我们在本节中的示例程序包括 `block_redux` 和 `sphere_world_redux`。`Block_redux` 看起来应该有些熟悉，因为它就是第 1 章的第一个基于 GLUT 的示例程序，而 `sphere_world_redux` 则在第 5 章出现过。现在这些程序都是成熟的 Windows 程序，完全是用 C++ 和 Win32 API 编写的。图 13.1 所示展示了 `sphere_world_redux` 的输出，现在使用了一个多重采样窗口。

图 13.1

`sphere_world_redux` 的输出，  
现在使用了一个多重采样窗口



### 13.3.1 创建窗口

所有基于 Windows 的低级 GUI 程序的入口点都是 main 函数。我们在第二个名为 SetupWindow 的函数中完成了所有窗口设置。程序清单 13.2 展示了第一个示例程序中 SetupWindow 和 main 函数的片段。

程序清单 13.2 GLRECT 示例程序的中 WinMain 函数

```
////////////////////////////////////
////////////////////////////////////
// 设置实际窗口和相关状态
// 创建窗口, 查找像素格式, 创建 OpenGL 环境
bool SetupWindow(int nWidth, int nHeight)
{
    // 初始化
    ....

    TCHAR szWindowName[50] = TEXT("GLRECT Redux");
    TCHAR szClassName[50] = TEXT("OGL_CLASS");
    // 设置窗口类
    g_windClass.lpszClassName = szClassName; // Set the name of the Class
    g_windClass.lpfnWndProc = (WNDPROC)WndProc;
    g_windClass.hInstance = g_hInstance; // Use this for the module handle

    // 选择默认鼠标光标
    g_windClass.hCursor = LoadCursor(NULL, IDC_ARROW);

    // 选择默认窗口图标
    g_windClass.hIcon = LoadIcon(NULL, IDI_WINLOGO);
    g_windClass.hbrBackground = NULL; // 没有背景
    g_windClass.lpszMenuName = NULL; // 这个窗口没有菜单

    // 为这个类设置样式, 特别是捕获窗口重绘、unique DC 和调整大小
    g_windClass.style = CS_HREDRAW | CS_OWNDC | CS_VREDRAW;
    g_windClass.cbClsExtra = 0; // 额外的类内存
    g_windClass.cbWndExtra = 0; // 额外的窗口内存

    // 注册新定义类
    if(!RegisterClass(&g_windClass))
        bRetVal = false;

    dwExtStyle = WS_EX_APPWINDOW | WS_EX_WINDOWEDGE;
    dwWindStyle = WS_OVERLAPPEDWINDOW;
    ShowCursor(TRUE);

    // 设置窗口宽度和高度
    g_windowRect.left = nWindowX;
    g_windowRect.right = nWindowX + nWidth;
    g_windowRect.top = nWindowY;
    g_windowRect.bottom = nWindowY + nHeight;
    AdjustWindowRectEx(&g_windowRect, dwWindStyle, FALSE, dwExtStyle);
    int nWindowWidth = g_windowRect.right - g_windowRect.left;
    int nWindowHeight = g_windowRect.bottom - g_windowRect.top;

    // 创建窗口
    g_hWnd = CreateWindowEx(dwExtStyle, // 扩展样式
```

```

        szClassName, // 类名
        szWindowName, // 窗口名
        dwWindStyle |
        WS_CLIPSIBLINGS |
        WS_CLIPCHILDREN, // 窗口样式
        nWindowX, // 窗口位置 x
        nWindowY, // 窗口位置 y
        nWindowWidth, // 高度
        nWindowHeight, // 宽度
        NULL, // 父窗口
        NULL, // 菜单
        g_hInstance, // 实例
        NULL); // 将它传递到 WM_CREATE

// 现在我们有了一个窗口, 设置像素格式描述符
g_hDC = GetDC(g_hWnd);

// 设置一个虚拟像素格式, 以便我们可以对 wgl 函数进行访问
SetPixelFormat( g_hDC, 1, &pfid);
// 创建 OGL 环境并将其设置为当前环境
g_hRC = wglCreateContext( g_hDC );
wglMakeCurrent( g_hDC, g_hRC );

if (g_hDC == 0 ||
    g_hRC == 0)
{
    bRetVal = false;
    printf("!!! An error occurred creating an OpenGL window.\n");
}
// 设置加载 OGL 函数指针的 GLEW
GLenum err = glewInit();
if (GLEW_OK != err)
{
    /* 问题: glewInit 失败, 出现某些严重错误。*/
    bRetVal = false;
    printf("Error: %s\n", glewGetErrorString(err));
}
const GLubyte *oglVersion = glGetString(GL_VERSION);
printf("This system supports OpenGL Version %s.\n", oglVersion);

// 现在已经对扩展进行了设置,
// 删除窗口并开始选择真正的格式
wglMakeCurrent(NULL, NULL);
wglDeleteContext(g_hRC);
ReleaseDC(g_hWnd, g_hDC);
DestroyWindow(g_hWnd);
// 再次创建窗口

... ..

int nPixCount = 0;

// 指定我们关心的重要属性
int pixAttribs[] = {
    WGL_SUPPORT_OPENGL_ARB, 1, // 必须支持 OGL 渲染
    WGL_DRAW_TO_WINDOW_ARB, 1, // 能够运行一个窗口的像素格式
    WGL_ACCELERATION_ARB, 1, // 必须为硬件加速
    WGL_COLOR_BITS_ARB, 24, // R、G 和 B 各 8 位精度
    WGL_DEPTH_BITS_ARB, 16, // 窗口的 16 位深度精度
    WGL_DOUBLE_BUFFER_ARB, GL_TRUE, // 双缓冲环境
    WGL_SAMPLE_BUFFERS_ARB, GL_TRUE, // 开启 MSAA

```

```

WGL_SAMPLES_ARB, 8, // 8x MSAA
WGL_PIXEL_TYPE_ARB, WGL_TYPE_RGBA_ARB, // 像素格式应该为 RGBA 类型
0 }; // 以 NULL 结束

// 要求 OpenGL 寻找与我们的属性相匹配的最佳格式
// 只取回一种格式
wglChoosePixelFormatARB(g_hDC, &pixAttribs[0], NULL, 1,
                        &nPixelFormat, (UINT*)&nPixCount);

if(nPixelFormat != -1)
{
    // 已经获取一个格式, 现在将它设为当前格式
    SetPixelFormat( g_hDC, nPixelFormat, &pf );
    GLint attribs[] = {WGL_CONTEXT_MAJOR_VERSION_ARB, 3,
                      WGL_CONTEXT_MINOR_VERSION_ARB, 3,
                      0 };
    g_hRC = wglCreateContextAttribsARB(g_hDC, 0, attribs);
    if (g_hRC == NULL)
    {
        // 处理错误 . . .
    }
    wglMakeCurrent( g_hDC, g_hRC );
}
ShowWindow( g_hWnd, SW_SHOW );
SetForegroundWindow( g_hWnd );
SetFocus( g_hWnd );
g_ContinueRendering = true;
return bRetVal;
}
// ////////////////////////////////////////
// 主程序函数, 在开始时调用
// 首先设置窗口和 OGL 状态, 然后进入渲染循环
int main(int argc, char* argv[])
{
    gltSetWorkingDirectory(argv[0]);
    if(SetupWindow(800, 600))
    {
        SetupRC();
        ChangeSize(800, 600);
        while (g_ContinueRendering)
        {
            mainLoop();
            Sleep(0);
        }
    }
    KillWindow();
    return 0;
}

```

这个程序清单几乎完全包含了标准设置代码。请注意这里为了设置窗口样式而包含了 CS\_OWNDC。指定这个标志会导致 Windows 特别为窗口分配一个 DC (设备环境)。我们需要一个设备环境, 能够支持 GDI 渲染和 OpenGL 双重缓冲翻页 (OpenGL double-buffered page flipping)。我们可以使用这个设备环境来引用特定窗口。

**首先, 我们需要一个设备环境**

首先, 我们需要一个 Windows 设备环境, 然后才能在窗口中使用 GDI 进行绘制。无论编写 OpenGL、

GDI 或者甚至是 DirectX 程序，我们都需要这个设备环境。

在 Windows 中进行的任何绘制（即使是在内存中绘制位图）都需要一个设备环境来对所绘制的特定对象进行标识。我们可以使用一个简单的函数调用来获取这个设备环境。

```
HDC hDC = GetDC(hWnd);
```

hDC 变量就是由窗口句柄 hWnd 标识的窗口的设备环境的句柄。我们为在这个窗口中进行绘制的所有 GDI 函数使用这个设备环境。在创建一个 OpenGL 渲染环境，将它设置为当前渲染环境，以及执行缓冲区交换时，也需要这个设备环境。我们可以调用另外一个同样使用这两个值的简单函数，来告诉 Windows 我们这个窗口不再需要这个设备环境了。

```
ReleaseDC(hWnd, hDC);
```

### 渲染环境的初始化

在窗口创建时，我们要做的第一件事就是获取设备环境（一定要牢记）并设置像素格式。

```
// 存储设备环境
g_hDC = GetDC(g_hWnd);

// 虚拟 pfd
PIXELFORMATDESCRIPTOR pfd;

// 设置像素格式
SetPixelFormat(g_hDC, nPixelFormat, &pfd);
```

然后，我们创建 OpenGL 渲染环境，并将它设置为当前环境。

```
// 创建渲染环境并将其设置为当前渲染环境
GLint attribs[] = { WGL_CONTEXT_MAJOR_VERSION_ARB, 3,
                   WGL_CONTEXT_MINOR_VERSION_ARB, 3, 0 };
g_hRC = wglCreateContextAttribsARB(g_hDC, 0, attribs);
wglMakeCurrent(hDC, hRC);
```

### 关闭渲染环境

当窗口过程接收到 WM\_DESTROY 消息时，或者确定不再使用它时，OpenGL 渲染环境必须被删除。在使用 wglDeleteContext 函数删除渲染环境之前，首先必须再次调用 wglMakeCurrent，但是这次是以 NULL 为 OpenGL 渲染环境的参数。

```
// 取消选择当前渲染环境并将其删除
wglMakeCurrent(g_hDC, NULL);
wglDeleteContext(g_hRC);
```

在删除渲染环境之前，应该先删除显示列表、纹理对象，以及其他 OpenGL 分配的内存。编写良好的程序会仔细地清除它们分配的所有对象和内存。如果对绑定到一个环境的对象清除失败，那么可能会导致内存泄露或其他副作用。

## 13.4 全屏渲染

窗口化的 OpenGL 应用程序虽然很好,但是如果应用程序不是全屏的话,就很难创建一个“沉浸式”游戏(immersive game)了!开发者们最普遍的问题之一,就是“我们怎样才能使用 OpenGL 进行全屏渲染呢?”实际上,如果读者已经阅读了这一章内容的话,就已经知道如何使用 OpenGL 进行全屏渲染了——就像渲染到其他任何窗口一样!这样,真正的问题就是“我们怎样才能创建一个占据了整个屏幕并且没有边框的窗口呢?”一旦创建了这样的窗口,在这个窗口中进行渲染就与在本书任何其他示例中的任何其他窗口进行渲染没有什么区别了。

即使严格地说这个问题并不是关于 OpenGL 的,广大读者也会有足够的兴趣,促使我们在这里针对这个主题做一些介绍。创建一个全屏窗口非常简单,几乎和创建一个以(0,0)为原点,与屏幕大小相等的常规窗口一样简单。我们还是使用了一个不同的窗口样式,因为不需要标题栏或边框,反正它们都是不可见的。程序清单 13.3 中的代码就能完成这项工作。

程序清单 13.3 设置一个全屏窗口

```
if(bUseFS)
{
    // 准备进行所需分辨率的模式设置
    DEVMODE dm;
    memset(&dm,0,sizeof(dm));
    dm.dmSize=sizeof(dm);
    dm.dmPelsWidth = nWidth;
    dm.dmPelsHeight = nHeight;
    dm.dmBitsPerPel = 32;
    dm.dmFields=DM_BITSPERPEL|DM_PELSWIDTH|DM_PELSHEIGHT;

    long error = ChangeDisplaySettings(&dm, CDS_FULLSCREEN);

    if (error != DISP_CHANGE_SUCCESSFUL)
    {
        // 哎呀,出问题了,要通知用户
        if (MessageBox(NULL, "Could not set fullscreen mode.\n"
            "Your video card may not support the requested mode.\n"
            "Use windowed mode instead?", g_szAppName,
            MB_YESNO|MB_ICONEXCLAMATION)==IDYES)
        {
            g_InFullScreen = false;
            dwExtStyle = WS_EX_APPWINDOW | WS_EX_WINDOWEDGE;
            dwWindStyle = WS_OVERLAPPEDWINDOW;
        }
        else
        {
            MessageBox(NULL, "Program will exit.",
                "ERROR", MB_OK|MB_ICONSTOP);
            return false;
        }
    }
}
else
{
    // 模式设置通过,进行全屏样式设置
    g_InFullScreen = true;
```

```

        dwExtStyle = WS_EX_APPWINDOW;
        dwWindStyle = WS_POPUP;
        ShowCursor(FALSE);
    }
}

AdjustWindowRectEx(&g_windowRect, dwWindStyle, FALSE, dwExtStyle);

// 再次创建窗口
.....

```

## 13.5 双重缓冲

在使用 `wglChoosePixelFormatARB` 查询一个像素格式时, `sphere_world_redux` 示例程序通过在属性列表中指定 `WGL_DOUBLE_BUFFER_ARB` 来请求一个双重缓冲像素格式。到目前为止, 我们已经接触了很多使用双重缓冲的示例程序。但是, 既然与如何分配像素格式以及如何对程序进行控制有关, 我们还是简单地回顾一下吧。在使用一个双重缓冲像素格式时, 就会分配两个大小与这个窗口相同的表面, 其中一个作为前台缓冲区使用, 而另一个则作为后台缓冲区使用。和第 8 章、第 9 章一样, 我们可以通过调用以 `GL_FRONT` 或 `GL_BACK` 为参数的 `glDrawBuffers` 绘制到这两个缓冲区。我们为什么要这样做呢? 双重缓冲使得 OpenGL 可以将我们的整个场景绘制到后台缓冲区中, 而不需要在屏幕上显示任何中间结果。这种方式可以为用户提供一种更加平滑和赏心悦目的视觉体验。

但是, 如果我们总是渲染到一个不可见的缓冲区中, 那么用户如何才能看到它们呢? 很简单, 只要在完成绘制并且缓冲区需要进行交换时告诉 OpenGL 就可以了。

只要调用以这个窗口的设备句柄为参数的 `SwapBuffers` 就能做到这一点。一旦进行了这项调用, 后台缓冲区将被显示, 而程序将会对一个新的后台缓冲区进行处理。

```

//进行缓冲区交换
SwapBuffers(g_hDC);

```

### 13.5.1 消除视觉撕裂

如果应用程序能够快速地进行绘制, 并且调用 `SwapBuffers` 的速度比显示器的刷新速度更快, 就会出现一个称为图像撕裂 (tearing) 的难看效果。如果应用程序在前面一帧完成扫描之前就调用 `SwapBuffers`, 那么使用这个应用程序就会看到包含一部分上一帧内容和一部分下一帧内容的图像。

有一个得到广泛支持的 `WGL_EXT_swap_control` 函数能够帮助我们! 我们可以告诉 OpenGL 在交换调用的间隔中至少允许出现多少视频帧或垂直同步 (V-Sync)。只要使用下面的函数设置间隔即可。

```

Bool wglSwapIntervalEXT(GLint interval);

```

如果将间隔 (interval) 设为 0, 那么对 `SwapBuffers` 的调用就不受限制了, 就像没有这个扩展一样;

如果将间隔设为 1, 那么显示器的每一次垂直刷新( 每一个视频帧 )就只允许进行一次 SwapBuffers 调用。要消除图像撕裂, 这正是我们想要的! 当应用程序等待交换完成时, 所有额外的 CPU 时间都可以用来做其他事情。

我们可以在 wglSwapIntervalEXT 中设置更长的间隔, 在两次交换之间等待等多的帧, 但是这样做会导致应用程序中出现明显的停顿。还有很多驱动程序不支持大于 1 的间隔, 并且会自动将间隔截取到 1。

## 13.6 小结

本章介绍了如何在 Windows 平台上使用 OpenGL。我们了解了可用于 Windows 的不同驱动程序模型和实现, 以及相关的注意事项。我们还学习了如何搜索、枚举和选择一个像素格式, 来获取我们想要的硬件加速或软件渲染支持类型。现在我们看到了代替 GLUT 框架的 Win32 程序基本框架, 可以编写真正的本地 Win32 和 Win64 应用程序代码了。我们还了解了如何为游戏或模拟类型的应用程序创建全屏窗口。

最后, 本章提供了一个 block\_redux 程序的源代码, 它帮助我们开始接触 Windows 上的 OpenGL 应用, 还有一个对 SphereWorld 进行了进一步扩展的全屏和多重采样版本, 即 sphere\_world\_redux 程序。这些程序演示了如何使用多种 Windows 特定属性和 WGL 扩展( 如果它们可用的话 )。



## 第 14 章 OS X 上的 OpenGL

作者: Richard S. Wright, Jr.

### 本章内容

- ✦ 使用 Cocoa 和 Interface Builder 创建 OpenGL 视图
- ✦ 创建优化的全屏 OpenGL 窗口
- ✦ 消除屏幕上的视觉撕裂
- ✦ 优化填充性能
- ✦ 启用 OpenGL 的多核心版本

OpenGL 是 Mac OS X 平台原生和首选的 3D 渲染 API。实际上, OpenGL 在操作系统的最底层为桌面、GUI 和 Mac OS X 的 2D 绘图 API 与图像合成引擎 (Quartz) 提供服务。OpenGL 在 Mac 平台上的重要性怎么形容都不过分。由于备受 Apple 的青睐 (在某种程度上与 Direct3D 备受 Microsoft 青睐的情况有些相似), 它得到了 Apple 巨大的支持和投资, 以持续地对这种 API 进行修改和扩展。

请记住, 本章并不是关于 OS X 编程的, 而是关于在 OS X 中使用 OpenGL 的。下面几节内容假定读者拥有一些 Mac 编程经验。您也许仍然能够继续阅读, 但这些内容将主要讲解如何使用 OpenGL, 而很少讲述如何进行 OS X 开发。

### 14.1 OpenGL 在 Mac 上的 4 种接口

Mac 上有 4 种 OpenGL 编程接口, 每一种都有其自身的特点、历史和用途。如何选择使用它们主要取决于我们想要如何在 Mac 上创建应用程序, 以及特定的渲染需求。如果我们全面了解 OS X 编程领域的话, 就会遇到所有这 4 种技术, 但是时至今日, 它们中的某些技术已经不再适用了。表 14.1 列出了这 4 种接口。

表 14.1

MacOS X 中的 OpenGL 接口

名 称	描 述
GLUT	为基于简单渲染的应用程序提供完整的框架 这种接口位于 NSOpenGL 的顶层
AGL	为使用 Carbon 作为框架的开发者提供 OpenGL 接口
NSOpenGL	为使用 Cocoa 面向对象框架的开发者提供 OpenGL 接口
CGL	是最底层的 OpenGL 接口, 适用于所有应用程序技术。AGL 和 NSOpenGL 都在 CGL 的上层

我们用这些接口在窗口或显示设备中进行 OpenGL 的配置。在这些工作完成之后, 剩下的工作就交给 OpenGL 了。我们在第 2 章介绍了如何对一个基于 GLUT 的程序进行设置, 并且前面章节的所有示例程序都是基于 GLUT 的, 所以在这里就没必要进一步讨论如何使用 GLUT 了。对于快速便捷的演示程序, 或者甚至是简单到不需要任何用户接口的应用程序来说, GLUT 是一个很好的选择。即使是在最新的 OS X 版本, 即 10.6 版 (Snow Leopard, 雪豹) 中, AGL 接口仍然得到支持 (这里的“支持”其实只表示“它还能用”)。但是, AGL 是一种只适用于 Carbon 的 API, 而 AGL 和 Carbon 都已经是“不推荐”的, 被看作过时的 API 了。Carbon 也没有随着 OS X 的变革被升级到 64 位, 因此我们在本书的这一版中不再介绍 AGL。CGL 是 Mac 可用的最底层 OpenGL 接口, 能够从任何 OpenGL 程序中直接进行调用。

但是, 由于操作系统架构的进步, 对于“雪豹”上全屏高性能渲染来说, 使用 CGL 已经没有太大必要了, 这一点在本章后面的内容将会看到。这样本章的重要目标就是基于 Cocoa 的 OpenGL 编程, 因为从大体上讲, 这是我们构建应用程序框架和 OpenGL 初始化的主要方式。

## 14.2 在 OpenGL 中使用 Cocoa

在 Mac OS X 中, 开发者可以选用很多种编程语言。Objective-C 是一种在 Mac 上非常流行的语言 (但在其他地方就不那么流行了)。对 Cocoa 最恰当的表述是, 它既是应用程序框架类的集合, 又是一种可视化编程范例。开发者做了很多关于 Interface Builder、设计用户界面、分配 properties 类, 甚至在事件之间建立连接的工作。

Objective-C 的类是 controls 类的子类, 或者是为了增加应用程序的功能性而从头开始创建的。幸运的是, OpenGL 是这个开发环境中最优秀的成员。

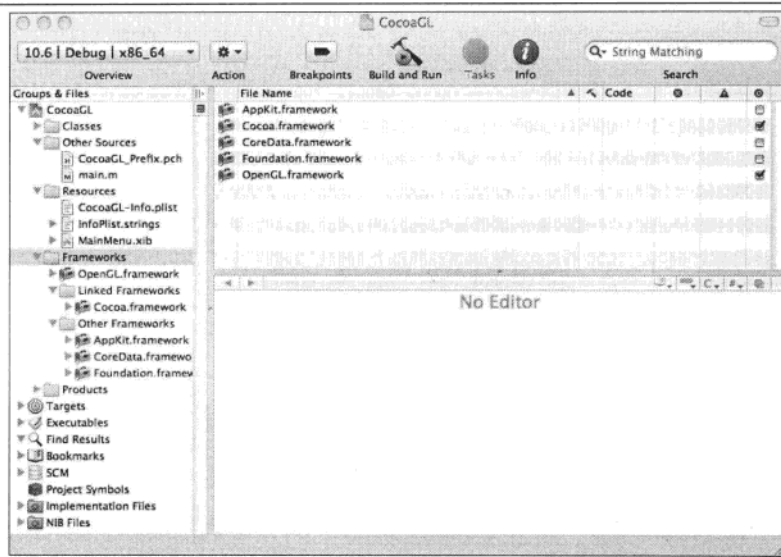
### 14.2.1 创建一个 Cocoa 程序

可以使用 Xcode 中的新项目助理来创建一个基于 Cocoa 的程序。在第 2 章中我们已经完成过这项工作, 当时我们是使用 Xcode 创建了我们的第一个基于 GLUT 的 OpenGL 程序。

但是这一次, 我们不再使用基于 GLUT 的代码来替换已生成的项目。图 14.1 所示展示了我们在添加了 OpenGL 框架 (但是这一次并没有添加 GLUT 框架!) 之后最新创建的 COCOAGL 项目。

图 14.1

初始 CocoaGL 项目

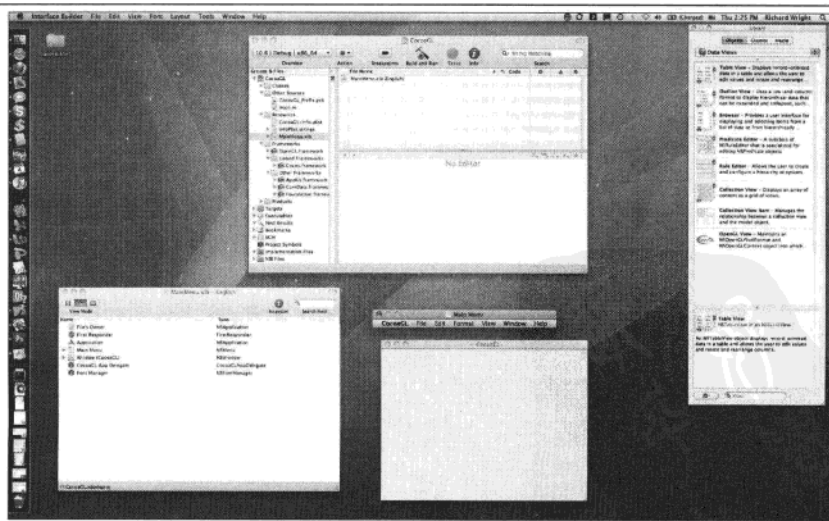


### 添加 OpenGL 视图

Cocoa 应用程序在一个 XIB 文件（过去的 NIB 的经过编译的版本。由于一些历史原因，NIB 代表 NEXTSTEP Interface Builder）中保存资源和 GUI 布局。双击 Resources 文件夹下的 MainMenu.xib 文件，这样将启动 Interface Builder 并打开主 xib 进行编辑。我们将会看到与图 14.2 所示相似的屏幕内容，其中的主窗口已经打开。

图 14.2

Interface Builder——准备好了



在 library 调板中，使用顶部的标签来选择类，然后向下滚动，直到看见 NSOpenGLView 为止。点击一个 NSOpenGLView 将其拖曳到主窗口中，并改变它的大小以填充大多数主窗口。我们还可以尝试改变主窗口的大小。在图 14.3 中可以看到，我们现在已经在窗口中央准备好了一个 NSOpenGLView。

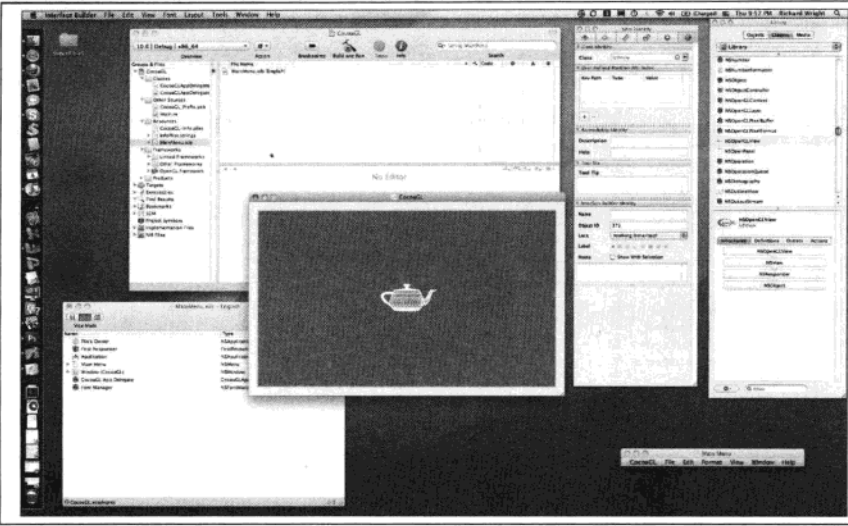


图 14.3  
一个非常基础的界面窗口

创建一个自定义 OpenGL 类

下一项工作是创建一个从 `NSOpenGLView` 中衍生出来的自定义类，并将它在窗口中与 OpenGL 视图关联起来。单击库窗口中的类标签，向下滚动到 `NSOpenGLView` 入口，并右键单击它，如图 14.4 所示，然后选择 `New Subclass`（新的子类）。

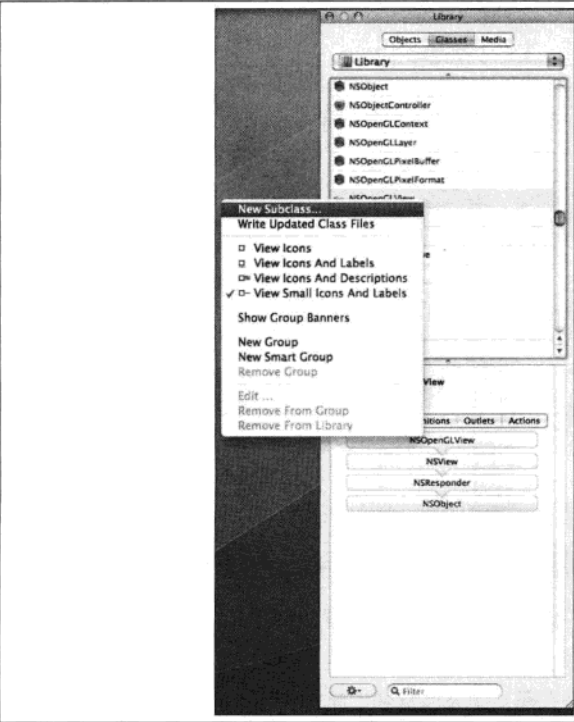
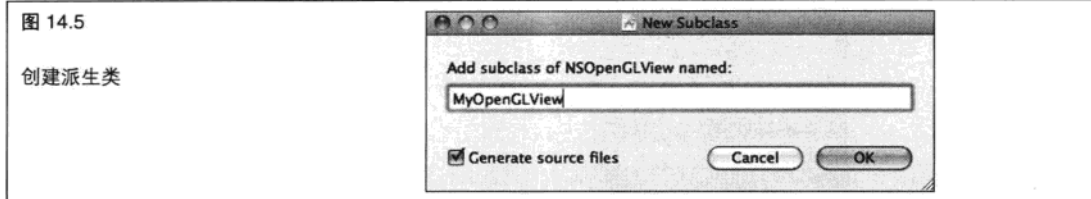
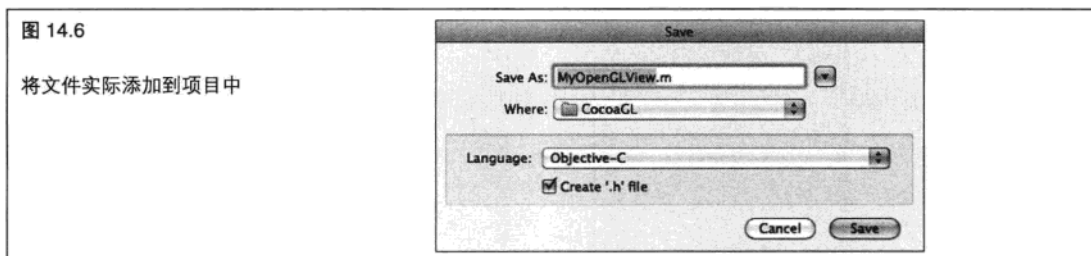


图 14.4  
创建 NSOpenGLView 子类

在出现的弹出窗口中命名子类。在这里，我们可以采用默认的名称 `MyOpenGLView`。我们要选中“Generate Source Files”复选框，如图 14.5 所示。

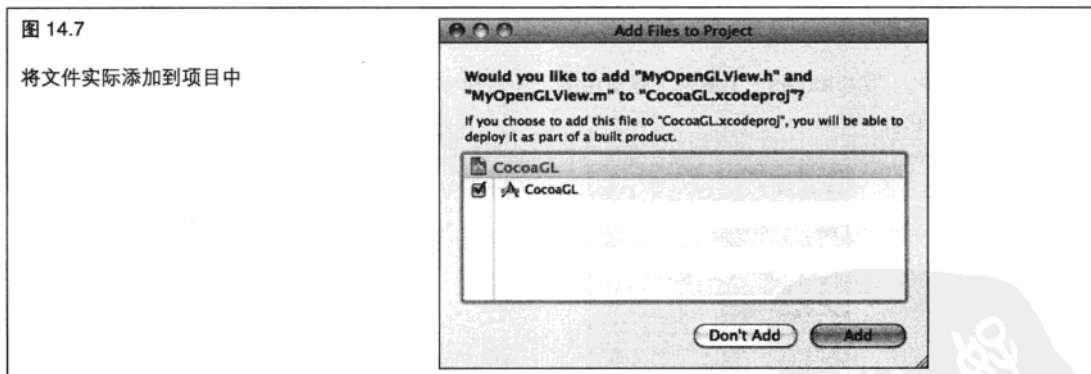


下一个弹出窗口会询问将要放置派生类文件的名称，如图 14.6 所示，确保选中“Create ‘.h’ File”复选框。



最后，Interface Builder 会询问我们是否要将这个类添加到项目中，如图 14.7 所示。选中项目名称旁边的复选框，然后单击“Add”按钮。

现在，我们有了一个真正的 Cocoa OpenGL 视图类，可以开始进行综合运用了。



## 综合运用

开始编写代码之前，在 Interface Builder 中我们还有两件事需要做。第一件事是，我们需要将 `NSOpenGLView` 窗口设置成与自定义 `MyOpenGLView` 类相连接。在 Interface Builder 中选择 `NSOpenGLView` 窗口，并在菜单中选择 Inspector。Inspector 窗口如图 14.8 所示，图中选择了 Identity 标签。在 class 组合框中，将 `NSOpenGLView` 改为 `MyOpenGLView`。

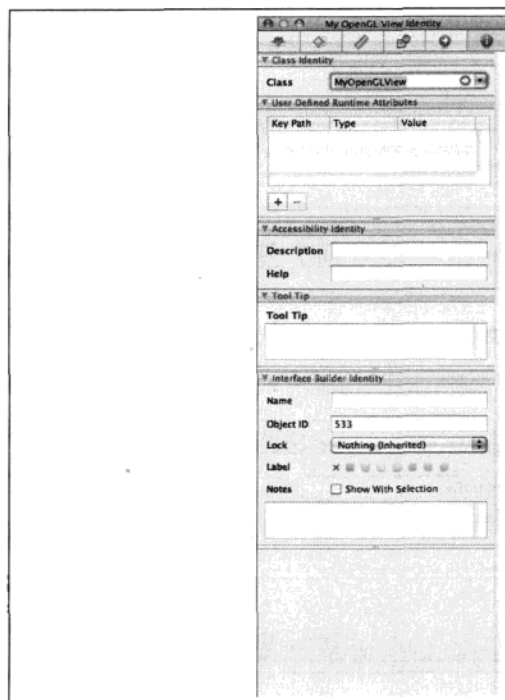


图 14.8

连接自定义 MyOpenGLView 类

第二件事情是，要对父窗口进行修改，以使它不再使用 One Shot 内存。这个标记在默认情况下是开启的，它告诉父窗口，在最小化到 Dock 工具条或者隐藏时可以删除子窗口对象。对于一个 OpenGL 窗口来说，这样做可能会导致消极的副作用，即破坏视图和 OpenGL 环境之间的连接，这样就可能阻止进一步的渲染操作进行显示。

在图 14.9 中，“Attributes”（属性）标签中的“One Shot”复选框被取消选择，单击主窗口的说明文字就可以看到它。

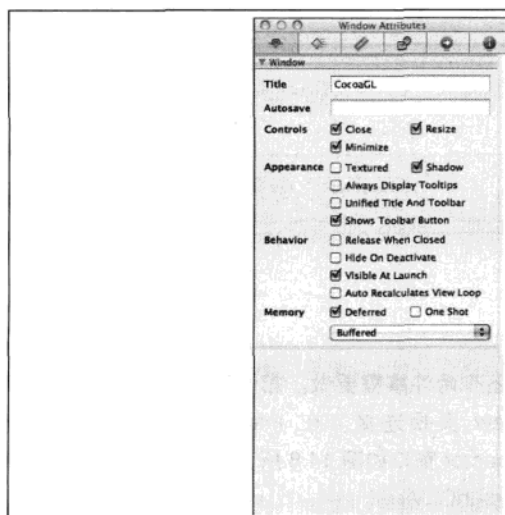


图 14.9

关闭“One Shot”内存属性

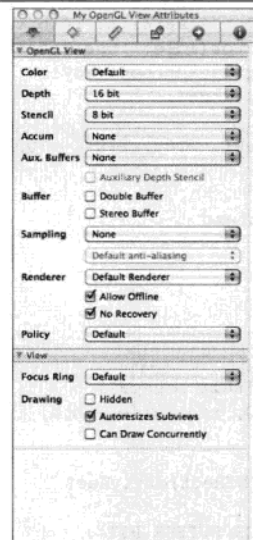
在这里我们可以进行一些选择,例如颜色、深度和模板缓冲区的位深和格式。我们还可以配置一个累积缓冲区,但是这个特性在核心版本中已经不推荐了,所以就不鼓励使用了(同时我们也有充分的理由不在本书对此进行介绍了)。我们还可以选择一个多重采样颜色缓冲区、立体缓冲区(左缓冲区和右缓冲区),甚至可以强制进行后备的软件渲染而不使用 OpenGL 硬件。

### 设置 OpenGL 视图属性

Interface Builder 还为我们提供了到一个 NSOpenGLView 控件中所有帧缓冲区属性的访问。单击控件本身,然后选择 Inspector (检查)窗口的属性标签。图 14.10 所示显示了大量选项。

图 14.10

OpenGL 视图属性窗口



## 14.2.2 综合运用

回到 Xcode 项目窗口,我们可以看到 MyOpenGLView 头文件和执行文件。这里包含 MyOpenGLView 类的定义,它是由 NSOpenGLView 派生出来的。Interface builder 已经将这个类连接到主窗口中的 OpenGL 视图,现在需要我们做的只是添加类框架和 OpenGL 渲染代码。

为新类编辑过的头文件很小,只包含动画中要用到的一个到 NSTimer 的成员指针。

```
#import <Cocoa/Cocoa.h>

@interface MyOpenGLView : NSOpenGLView {
    NSTimer *pTimer;
}

@end
```

在执行文件中,我们添加一个 idle 函数,还要执行每个 OpenGL 程序都需要的 4 个最高级 OpenGL 任务: prepareGL 用来进行 OpenGL 初始化, clearGLContext 用来进行 OpenGL 清理, reshape 用来进

行视口计算和窗口绑定，而最后的 `drawRect` 则用来执行渲染任务。程序清单 14.1 列出了最小 OpenGL 框架的完整源代码。

程序清单 14.1 一个 OpenGL 视图类的框架

```
#import "MyOpenGLView.h"

@implementation MyOpenGLView

- (void)idle:(NSTimer *)pTimer
{
    [self drawRect:[self bounds]];
}

- (void) prepareOpenGL
{
    pTimer = [NSTimer timerWithTimeInterval:(1.0/60.0) target:self
    selector:@selector(idle:) userInfo:nil repeats:YES];
    [[NSRunLoop currentRunLoop]addTimer:pTimer forMode:
    NSDefaultRunLoopMode];

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

- (void) clearGLContext
{
    // 进行清理工作
}

- (void) reshape
{
    NSRect rect = [self bounds];
    glViewport(0, 0, rect.size.width, rect.size.height);
}

- (void) drawRect:(NSRect)dirtyRect
{
    glClear(GL_COLOR_BUFFER_BIT);

    glFlush();
}

@end
```

CocoaGL 的输出结果如图 14.11 所示。正如我们所看到的，这只是一个空白的蓝色窗口而已，但是我们现在已经为下一个示例准备好一个完整的框架。

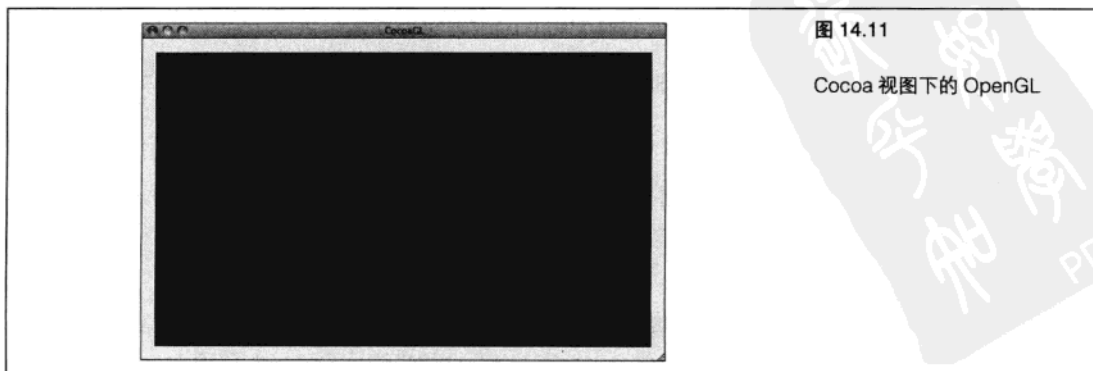


图 14.11

Cocoa 视图下的 OpenGL

### 14.2.3 双缓冲还是单缓冲

看到这里，读者可能会想象到在人行横道上急刹车时轮胎所发出的声音。

在程序清单 14.1 中，我们看到的是 `glFlush`，而不是某种缓冲区交换调用，对吧？确实如此，这让我们接触到 Mac OS X 上的 OpenGL 的一个微妙之处（同时也为平滑过渡到下一部分做好了准备）。

在 Mac OS X 中，整个桌面实际上是用 OpenGL 进行加速的。在任何时候进行 OpenGL 渲染，我们都会渲染到一个离屏缓冲区。缓冲区交换的作用只是通知操作系统，渲染工作已经准备好与桌面其他部分的渲染进行合成了。我们可以考虑将桌面合成引擎（desktop compositing engine）作为前台缓冲区（front buffer）。这样，在窗口化的 OpenGL 应用程序（这在 Cocoa 和现在已经“不推荐”的 Carbon 中都有应用）中，所有 OpenGL 窗口都被真正地进行单缓冲了。根据个人看法的不同，我们也可以说所有的 OpenGL 窗口都被真正地双缓冲了，因为桌面合成成为了前台缓冲区。不管选择哪种看法，都能让我们心里感到踏实。实际上，如果要执行 `glDrawBuffer(GL_FRONT)`，Mac 上的驱动程序将会进入三重缓冲模式（triple-buffered mode）！事实上，Mac 上所有 OpenGL 窗口都应该以单缓冲对待。缓冲区交换调用其实只是执行了一次 `glFlush`，除非我们正在全屏环境下。因为这个原因（还有很多其他原因—至少还有一个原因是，我们放弃了驱动程序本身对何时进行清理良好判断能力），在我们完成所有的 OpenGL 渲染之前，应当避免在 Cocoa 视图中调用 `glFlush`。

### 14.2.4 球体世界

为了对以前的示例进行补充，我们将第 5 章的 SphereWorld（球体世界）示例从 GLUT 框架移植到我们刚刚构建的 Cocoa 框架中。在开始时，我们创建一个新的 Xcode 项目并遵循对 CocoaGL 所执行的相同步骤，除了调用自定义视图类 SphereWorldView 之外。接下来，我们将 SphereWorld.cpp 文件和 3 个纹理文件复制到新项目的文件夹中。在这个项目中添加 SphereWorld.cpp，并将这 3 个纹理文件添加到 Xcode 中的/Resources。

#### GLTools 和 Objective-C++

在这个例子中，我们还要再次使用 C++ 库 GLTools。我们可以采用前面基于 GLUT 的示例中所使用同样的方法来添加 GLTools，这种方法已经在第 2 章逐个步骤地介绍过，所以在这里就不再重复讨论了。但是，我们确实有一些新工作需要做，就是使 C++ 代码能够在 Objective-C 中使用。而事实证明这是不可能的。相应的解决办法却出人意料地简单：将项目改为使用 Objective-C++！我们所要做的全部工作就是将 .m 文件重命名为 .mm，这样它们就成了 Objective-C++ 文件，而我们就可以在这些文件中使用 C++ 类了，就好像这些文件本身就是 C++ 文件一样。作为额外的奖励，所有 Cocoa Objective-C 类都可以用同样的方式处理。图 14.12 所示展示了 Xcode 项目到目前为止所发生的变化。现在可以对 SphereWorld.cpp 文件进行清理，并在 Cocoa 类中添加正确的函数调用了。



我们基于 Objective-C++ 和 GLTools 的项目

## 修剪球体世界

如果我们将 GLUT 框架代码从 SphereWorld.cpp 中移除,那么就只剩下 4 个需要从 Cocoa 框架中进行调用的函数了: SetupRC、ShutdownRC、ChangeSize 和 RenderScene。我们大概能够猜出它们的去向。移除 GLUT 的工作有些麻烦。首先,从原文件的顶部移除 GLUT 文件头。

```
#ifdef __APPLE__
#include <glut/glut.h>
#else
#define FREEGLUT_STATIC
#include <gl/glut.h>
#endif
```

然后，在 `RenderScene` 函数的末尾，执行一次缓冲区交换，并触发一次更新。我们已经不再需要缓冲区交换了，因为 Cocoa 框架现在负责这项工作，并且有一个定时器负责进行周期性更新。删除这两行，就接近完成了。

```
//进行缓冲区交换
glutSwapBuffers();

// 再来一次
glutPostRedisplay();
```

修剪的最后工作就是将主函数整个移除，并删除 SpecialKeys 回调，因为我们已经不再通过 GLUT 接收键盘输入了。

## 进行连接

在 C++ 模块中从一个 Objective-C++ 模块调用一个函数,其实和普通的 C++ 跨模块编程没什么区别。

程序清单 14.2 展示了 SphereWorldView 实现，其中声明了 SphereWorld 中的适当函数，并在框架中需要的地方进行了调用。我们还添加了 Cocoa 消息 keyDown 来捕捉键盘动作，并使用了 acceptFirstResponder 方法，这样这些键盘动作就能由 OpenGL 窗口进行了处理。这样一来就可以采用我们在基于 GLUT 的版本中使用的同样方式来移动照相机了。

程序清单 14.2 基于 Cocoa 的球体世界 (SphereWorld)

```
#include <GLTools.h>
#include <GLFrame.h>

#import "SphereWorldView.h"

void ChangeSize(int nWidth, int nHeight);
void RenderScene(void);
void ShutdownRC(void);
void SetupRC(void);

extern GLFrame cameraFrame; // 照相机帧

@implementation SphereWorldView
- (void)idle:(NSTimer *)pTimer
{
    [self drawRect:[self bounds]];
}
- (BOOL)acceptsFirstResponder
{
    [[self window] makeFirstResponder:self];
    return YES;
}
- (void)keyDown:(NSEvent*)event
{
    float linear = 0.1f;
    float angular = float(m3dDegToRad(5.0f));

    int key = (int)[[event characters] characterAtIndex:0];
    switch(key)
    {
        case NSUpArrowFunctionKey:
            cameraFrame.MoveForward(linear);
            break;
        case NSDownArrowFunctionKey:
            cameraFrame.MoveForward(-linear);
            break;
        case NSLeftArrowFunctionKey:
            cameraFrame.RotateWorld(angular, 0.0f, 1.0f, 0.0f);
            break;
        case NSRightArrowFunctionKey:
            cameraFrame.RotateWorld(-angular, 0.0f, 1.0f, 0.0f);
            break;
    }
}
- (void) prepareOpenGL
{
    pTimer = [NSTimer timerWithTimeInterval:(1.0/60.0) target:self
                                           selector:@selector(idle:) userInfo:nil
                                           repeats:YES];

    [[NSRunLoop currentRunLoop]addTimer:pTimer forMode: NSDefaultRunLoopMode];
}
```

```
        SetupRC();
    }

    - (void) clearGLContext
    {
        ShutdownRC();
    }

    - (void) reshape
    {
        NSRect rect = [self bounds];
        ChangeSize(rect.size.width, rect.size.height);
    }

    - (void) drawRect:(NSRect)dirtyRect
    {
        RenderScene();
        glFlush();
    }

@end
```

## 寻找纹理文件

就像在基于 GLUT 的示例中一样，我们将 SphereWorld 的纹理文件放入 Xcode 中的资源组。我们仍然必须重设当前的工作目录，以便文件函数能够找到它们。在 GLUT 示例中，我们将它放在主函数中。对于 Cocoa 来说，我们还是将它放在主函数中。下面列出了整个 main.mm 文件。

```
#include <GLTools.h>
#import <Cocoa/Cocoa.h>

int main(int argc, char *argv[])
{
    gltSetWorkingDirectory(argv[0]);
    return NSApplicationMain(argc, (const char **) argv);
}
```

我们需要在文件顶部添加 GLTools.h 头文件，然后调用现在应该已经很熟悉的 gltSetWorkingDirectory 函数就可以完成剩余工作了。

## GLEW 与 Cocoa

最后还有一件事需要我們进行处理。SphereWorld 使用了 GLTools，但是 GLTools 则使用了 GLEW 库，这个库引入了额外的 OpenGL 函数和扩展。使用 GLEW 有一个要求，就是 GLEW 头文件必须放置在实际系统 OpenGL 头文件 gl.h 之前。对于基于 GLUT 的程序来说，这从来都不是问题，因为 GLTools 引入了所有需要的头文件。当我们将 SphereWorld.cpp 添加到这个项目中，仍然没有问题。但是，一旦将 GLTools 添加到一个 .mm 文件就会出现编译错误。如果跟踪这些错误到这个头文件，就会发现它们是由于 glew.h 位于 gl.h 之后而引起的。即使进行搜索，也无法通过检查这个项目的源代码来找到这个错误。

事实上，Cocoa 本身就包含了 OpenGL 头文件。请记住，在 Mac 中 OpenGL 无处不在。在 Xcode

项目中有一个叫做 SphereWorld\_Prefix.pch 的文件是“前缀头文件”。这是一个预编译的头文件，它是自动添加到所有 Objective-C/C++ 模块中的。我们所需要的就是将 glew.h 头文件放到这个文件中。这没什么困难的——它只有 4 行代码，而这些就是我们所做的改变了！

```
#ifndef __OBJC__
#include <gl/glew.h>
#import <Cocoa/Cocoa.h>
#endif
```

最后，我们可以干净利落地进行构建了，并有了一个真正具有完全（好吧，是几乎完全）特性基于 Cocoa 的 OpenGL 程序。图 14.13 所示为最终作品。

图 14.13

基于 Cocoa 的应用程序中的 SphereWorld



## 14.3 全屏渲染

很多 OpenGL 应用程序都需要渲染到整个屏幕，而不是只在窗口的范围内。这些应用程序包括很多游戏、媒体播放器、kiosk 托管应用程序，以及其他专门类型的应用程序。完成这项工作的一种方法就是简单地创建一个大窗口，其大小与整个显示区域的大小相同。在 OS X 10.6（雪豹）之前，这并不是理想的方法，那时有必要使用 CGL 函数“捕获”显示器进行全屏渲染，以得到最佳的结果。

在“雪豹”中仍然支持这些 API，但它们已经不是必须的了，并且实际上 Apple 并不提倡屏幕捕捉技术。当渲染到一个全屏窗口时，我们会设置一个特殊环境标记，而 OS X 会自动尝试用老旧的屏幕捕捉技术采用的方式对渲染输出进行优化。但是，如果没有捕捉到显示器，也允许临界 UI 消息或其他窗口在全屏窗口中弹出。

以现在的标准来看，捕捉显示器就显得有些棘手了。现在甚至还有一种简单的方法，可以渲染到一个更小的后台缓冲区来提升填充性能，而无需改变显示器分辨率。让我们先来创建 SphereWorld 的一个全屏版本，即 SphereWorldFS。

### 14.3.1 在 Cocoa 中进行全屏显示

现在开始创建新版本的 SphereWorld，首先我们还是创建一个全新的 Xcode Cocoa 项目，命名为 SphereWorldFS。就像前面的示例一样，添加 OpenGL 框架，添加 GLTools 库，将 .m 文件重命名为 .mm，并且复制 SphereWorldView Cocoa 类、SphereWorld.cpp 渲染代码，以及添加到项目的/Resources 文件夹中的纹理文件。但是这一次，不再使用 Interface Builder。我们将手动创建并管理窗口。基于 Cocoa 的程序中的应用程序代理（application delegate），有一种称为 applicationDidFinishLaunching 的方法，在应用程序成功启动时就会调用这种方法。在新项目中，它被放在 SphereWorldFSAppDelegate.mm 文件中。

#### 选择一个像素格式

在 OpenGL 能够进行窗口初始化之前，我们首先必须选择合适的像素格式。像素格式描述了 3D 渲染的硬件缓冲区配置，诸如颜色缓冲区的深度值、模版缓冲区的大小，以及缓冲区是 onscreen（默认设置）还是 offscreen（离屏）的。像素格式由 AGL 数据类型描述。像素格式是由 Cocoa 数据类型 NSOpenGLPixelFormat 描述的。

为了选择满足需要的像素格式，我们首先要构造一个整型属性值数组。例如，下面的数组需要一个在目的缓冲区中有红、绿、蓝和透明度等分量的双缓冲像素格式，以及一个 16bit 的缓冲区，并且希望对像素格式进行加速，而不是软件 OpenGL 渲染。可能还会有其他属性，但从本质上说上面这些才是我们真正关心的。

```
NSOpenGLPixelFormatAttribute attrs[] = {

    // 设置我们的其他条件
    NSOpenGLPFAColorSize, 32,
    NSOpenGLPFADepthSize, 16,
    NSOpenGLPFADoubleBuffer,
    NSOpenGLFPAccelerated,
    0
};
```

请注意我们必须以 0 或 nil 作为数组的结尾。接下来，我们使用这个属性数组来分配像素格式。如果无法创建像素格式，那么分配例程将返回 nil，而我们应该进行一些正确的处理，因为一旦影响到 OpenGL 渲染，就会“游戏结束”了。

```
NSOpenGLPixelFormat* pixelFormat = [[NSOpenGLPixelFormat alloc]
                                   initWithAttributes:attrs];

if(pixelFormat == nil)
    NSLog(@"No valid matching OpenGL Pixel Format found");
```

大多数属性或者是一个布尔标记，或者包含一个整数值。只要出现布尔标记就会对属性进行设置，例如前面示例中的 NSOpenGLPFADoubleBuffer。另一方面，一个整数标记（例如 NSOpenGLPFADepthSize）后面则应该有一个整数值，这个值指定深度缓冲区所需要的位数。表 14.2 列出了可用的属性及其意义。

表 14.2

Cocoa 像素格式属性

属 性	意 义
NSOpenGLPFAAllRenderers	布尔值：允许所有可用的渲染程序
NSOpenGLPFADoubleBuffer	布尔值：必须为双缓冲
NSOpenGLPFAStereo	布尔值：必须为立体的
NSOpenGLPFAAuxBuffers	整数：辅助缓冲区的数量
NSOpenGLPFAColorSize	整数：颜色缓冲区（默认与屏幕匹配）的深度，以位为单位
NSOpenGLPFAAlphaSize	整数：颜色缓冲区中 alpha 的深度，以位为单位
NSOpenGLPFADepthSize	整数：深度缓冲区的深度，以位为单位
NSOpenGLPFAStencilSize	整数：模板缓冲区的深度，以位为单位
NSOpenGLPFAAccumSize	整数：累积缓冲区（在 OpenGL 3.x 中已经“不推荐”了）的深度，以位为单位
NSOpenGLPFAMinimumPolicy	布尔值：只考虑大于或等于指定深度的缓冲区
NSOpenGLPFAMaximumPolicy	布尔值：使用任何要求的缓冲区可用的最大深度值
NSOpenGLPFAOffScreen	布尔值：只使用可以离屏渲染的渲染程序
NSOpenGLPFAFullScreen	布尔值：只使用可以渲染到全屏环境的渲染程序。这个标记暗含 NSOpenGLPFASingleRenderer 标记
NSOpenGLPFASampleBuffers	整数：多重采样缓冲区的数量
NSOpenGLPFASamples	整数：每个多重采样缓冲区中的样本数量
NSOpenGLPFAAuxDepthStencil	独立属性：每个辅助缓冲区都有它自己的深度模板
NSOpenGLPFAColorFloat	布尔值：选择一个浮点颜色缓冲区
NSOpenGLPFAMultisample	布尔值：多重采样优先
NSOpenGLPFASupersample	布尔值：超级采样优先
NSOpenGLPFASampleAlpha	布尔值：更新多重采样 alpha 值
NSOpenGLPFARendererID	整数：使用这个整数指定的特定渲染程序
NSOpenGLPFASingleRenderer	布尔值：在单个监视器上强制进行单次渲染
NSOpenGLPFANoRecovery	布尔值：当资源耗尽时在单个环境中强制继续进行渲染。一般用处不大
NSOpenGLPFAAccelerated	布尔值：只选择一个硬件加速渲染程序
NSOpenGLPFAClosestPolicy	布尔值：选择与指定颜色缓冲区最接近的颜色缓冲区
NSOpenGLPFARobust	布尔值：只选择没有由于缺少资源而导致的失败模式的渲染程序。一般用处不大
NSOpenGLPFABackingStore	布尔值：只选择后台缓冲区与前台缓冲区大小相等的渲染程序。另外，保证在调用 flushBuffer 之后，后台缓冲区保持完整
NSOpenGLPFAMPSafe	布尔值：选择一个多处理器安全渲染程序
NSOpenGLPFAWindow	布尔值：只选择一个能够渲染到窗口的渲染程序
NSOpenGLPFAMultiScreen	布尔值：只选择一个可以驱动多个屏幕的渲染程序

续表

属 性	意 义
NSOpenGLPFACompliant	布尔值：只使用 OpenGL 兼容的渲染程序
NSOpenGLPFAScreenMask	整数：支持的物理屏幕的一个位遮罩（bit mask）
NSOpenGLPFAPixelBuffer	布尔值：允许渲染到一个像素缓冲区
NSOpenGLPFARemotePixelBuffer	布尔值：允许渲染到一个离线像素缓冲区
NSOpenGLPFAAllowOffLineRenderers	布尔值：允许离线渲染程序
NSOpenGLPFAAcceleratedCompute	布尔值：只选择支持 OpenGL 的渲染程序
NSOpenGLPFAVirtualScreenCount	整数：要求的虚拟屏幕数量

### 全屏应用程序核心

现在，让我们看一看全屏版本 SphereWorld 的主程序体大致是什么样子。程序清单 14.3 展示了 applicationDidFinishLaunching 实现的完整代码。

程序清单 14.3 创建并管理我们的全屏窗口

```
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification {  
  
    NSOpenGLPixelFormatAttribute attrs[] = {  
  
        NSOpenGLPFAFullScreen, 1, // 全屏环境  
  
        // 确定我们想要显示在哪个屏幕上（对于全屏环境来说我们必须这样做）  
        NSOpenGLPFAScreenMask,  
        CGDisplayIDToOpenGLDisplayMask(kCGDirectMainDisplay),  
  
        // 设置我们的其他条件  
        NSOpenGLPFAColorSize, 24,  
        NSOpenGLPFADepthSize, 16,  
        NSOpenGLPFADoubleBuffer,  
        NSOpenGLPFAAccelerated,  
        0  
    };  
  
    NSOpenGLPixelFormat* pixelFormat =  
        [[NSOpenGLPixelFormat alloc] initWithAttributes:attrs];  
    if(pixelFormat == nil)  
        NSLog(@"No valid matching OpenGL Pixel Format found");  
  
    NSRect mainDisplayRect = [[NSScreen mainScreen] frame];  
    NSWindow *pMainWindow =  
        [[NSWindow alloc] initWithContentRect: mainDisplayRect  
        styleMask:NSBorderlessWindowMask  
        backing:NSBackingStoreBuffered defer:YES];  
  
    [pMainWindow setLevel:NSMainMenuWindowLevel+1];  
    [pMainWindow setOpaque:YES];  
    [pMainWindow setHidesOnDeactivate:YES];  
  
    NSRect viewRect = NSMakeRect(0.0, 0.0,  
        mainDisplayRect.size.width, mainDisplayRect.size.height);  
    SphereWorldView *fullScreenView =
```

```

    [[SphereWorldView alloc] initWithFrame:viewRect
    pixelFormat: [ pixelFormat autorelease] ];
[pMainWindow setContentView: fullScreenView];
[pMainWindow makeKeyAndOrderFront:self];

// 隐藏光标
CGDisplayHideCursor (kCGDirectMainDisplay);

bool bQuit = false;
while(!bQuit) {
    // 检查并处理输入事件
    NSEvent *event;
    event = [NSApp nextEventMatchingMask:NSAnyEventMask
    untilDate:[NSDate distantPast]
    inMode:NSDefaultRunLoopMode dequeue:YES];
    if(event != nil)
        switch ([event type]) {
            case NSKeyDown:
                [fullScreenView keyDown:event];

                if((int)[[event characters]
                characterAtIndex:0] == 27) // ESC 退出
                    bQuit = true;
                break;

            case NSKeyUp:
                [fullScreenView keyUp:event];
                break;

            default:
                break;
        }

    [fullScreenView drawRect:viewRect];
}

// 再次显示光标
CGDisplayShowCursor(kCGDirectMainDisplay);

// 终止应用程序
[NSApp terminate:self];
}

```

我们所要做的第一项工作就是创建一个全屏像素格式。请注意 `NSOpenGLPFAScreenMask` 标记和相应 `NSOpenGLPFAScreenMask` 的使用。我们必须一起使用这两个标记来获取一个全屏环境下有效的像素格式，从而充分利用“雪豹”对于全屏窗口的优化能力所带来的好处。

第二项工作是，创建一个主窗口，其大小与当前桌面的大小相同。在这里我们使用 `NSBorderlessWindowMask` 消除标题、最小化按钮等。

```

NSRect mainDisplayRect = [[NSScreen mainScreen] frame];
NSWindow *pMainWindow =
    [[NSWindow alloc] initWithContentRect: mainDisplayRect
    styleMask:NSBorderlessWindowMask
    backing:NSBackingStoreBuffered defer:YES];

```

还有其他两项设置，对于真实的全屏体验来说也非常有用。我们将窗口级别设置为高于菜单栏，确保窗口不透明，并设置 `setHideOnDeactivate` 标记。这样做可以在进行切换离开这个窗口时将窗口进行隐藏，

然后再重新激活这个应用程序时自动恢复全屏状态。

```
[pMainWindow setLevel:NSMainMenuWindowLevel+1];  
[pMainWindow setOpaque:YES];  
[pMainWindow setHidesOnDeactivate:YES];
```

接下来我们根据前面构建的 SphereWorldView 类创建实际基于 OpenGL 的视图。这个视图被分配到主窗口，这个主窗口随后会被激活并进行显示。虽然 Interface Builder 为我们提供了一些像素格式的控制权，但是选择采用 initWithFrame 方式来创建 NSOpenGLView 类则为我们提供了最大限度的控制权和灵活性。

```
NSRect viewRect = NSMakeRect(0.0, 0.0,  
    mainDisplayRect.size.width, mainDisplayRect.size.height);  
SphereWorldView *fullScreenView =  
    [[SphereWorldView alloc] initWithFrame:viewRect  
    pixelFormat: [ pixelFormat autorelease] ];  
[pMainWindow setContentView: fullScreenView];  
[pMainWindow makeKeyAndOrderFront:self];
```

一旦我们创建了全屏窗口，通常就不需要显示鼠标光标了。Core GL ( CGL ) 方法 CGDisplayHideCursor 为我们隐藏了光标，直到应用程序终止或我们调用相应的 CGDisplayShowCursor 为止。

```
CGDisplayHideCursor(kCGDirectMainDisplay);
```

对于全屏窗口来说，我们对 Cocoa 事件循环本身进行处理。NSApp 方法 nextEventMatchingMask 能够检索最近发生的一个事件并将它从事件队列中删除。

如果是一个 NSKeyDown 事件，那么我们将它转发给 SphereWorldView 类，这个类会对方向键进行检查，以便进行照相机移动。在这里我们还会检查空格键，如果按下它的话就会将 bQuit 的值改为真，结束事件循环并最终结束应用程序。

```
[NSApp terminate:self];
```

## 改变视图类

在全屏模式下使用像素格式时，还要对 SphereWorldView 类进行一项额外的改变。在 drawRect 方法的末尾，我们调用 OpenGL 环境下的 flushBuffer 来代替 glFlush。

```
[[self openGLContext ]flushBuffer];
```

这样相当于为双缓冲渲染程序执行了一次缓冲区交换。因为我们不再是在“窗口”模式下，所以只是对命令缓冲区进行清理就不够了；我们还需要进行缓冲区交换，而这实际上已经发生了。

既然已经进入了全屏状态，那么我们同时还会进行一些其他有趣的工作。其中一项是，我们将 GLString 类从苹果的一个 OpenGL 演示（苹果自己的 CocosGL 演示）进行了移植，只使用新的 OpenGL 核心版本和 GLTools 中的存储着色器。我们在 SphereWorldView 类中使用它来计算和显示我们新的全屏 SphereWorld 的帧速率，它能够不受限制地以其最快速度运行。图 14.14 所示显示了 SphereWorld 以 199 帧/秒的速度运行时的情况。

图 14.14

显示帧速率的 SphereWorld  
示例程序



最后，我们还修改了运动的键盘处理，以使物体运动变得平滑。

在按下下一个运动键时，我们将这个键的标记设置为真，在释放这个键时则将这个标记设置为假，而不是在按下这个键时移动照相机。然后，我们在渲染函数中根据这些标记的状态进行移动。由于键盘消息所固有的延迟，这样做能够保持运动的平滑并减少抖动。让我们来亲身体验一下 SphereWorldFS 中的导航与原始的 SphereWorld 中的导航有什么区别吧！

## 14.4 CGL

正如我们已经说明的，核心 GL（常常被称为 CGL）是在 Mac OS X 中对 OpenGL 的最底层支持。它可以与表 14.1 中列出的任何其他 OpenGL 技术一起使用，而我們也可以在 SphereWorldFS 示例中使用一些有趣的函数。在这里，我们介绍在基于 Cocoa 的应用程序中使用 CGL 的一些快捷方便而又有用的技巧。其中一些在 Cocoa 中也有对应技巧，但是 CGL 版本在基于 GLUT 的 OpenGL 程序中，甚至是早在我们可能选择使用的更高层次第 3 方 C++ 框架中也适用。我们还可以单独使用 CGL 创建一个全屏环境并在需要时在其中进行渲染，但是就像我们刚刚看到的一样，在“雪豹”中这已经不是必要的了。

我们感兴趣的所有 CGL 函数都需要以当前 CGL 环境作为一个参数。在任何 OpenGL 应用程序中，我们都可以通过调用 CGLGetCurrentContext 来查询当前 CGL 环境。

```
CGLContextObj CGLGetCurrentContext(void);
```

### 14.4.1 同步帧速率

在以前的 SphereWorldFS 示例程序中，事件循环会以全速（每秒帧数最大）运行和渲染。因为每秒的帧数是代码执行速度的一种简单计算方法，所以这样做在对渲染或处理代码进行性能测试时非常有用。但是，在一个正式发布的应用程序中，这样做却有两个缺点。首先，除了对 GPU 过度使用之外，我们还

(至少!) 占用了 CPU 核心的全部循环。如果考虑典型情况下每秒进行 60 次显示刷新, 那么每秒显示多于 60 帧就没有真正的意义了。这样节省下的 GPU 性能就能够用于生成更加高级的渲染效果了, 而节省的 CPU 性能则可以用于提高其他应用程序处理性能, 或者可以为应用程序或游戏增加更多细节或特性。第二, 因为显示器每秒的刷新次数只有这么多, 所以每秒渲染的帧数超过显示器能够显示的帧数就会导致图像撕裂 (tearing)。当缓冲区交换发生在除了屏幕垂直回溯之外的任何一点时, 就会产生图像撕裂 (tearing)。从本质上说, 这种情况下在屏幕上会同时显示两个不同的帧。较早的一帧会占据显示器中当前帧刷新位置上方的区域, 而屏幕的底部则被新的缓冲区内内容所填充。当视图在场景中水平移动时这种现象看起来尤为严重。图 14.15 所示展示了撕裂的一个典型例子, 其中的显示器显示了两个不同的帧。



图 14.15

由于不同步的缓冲区交换导致的图像撕裂 (彩图 23 也展示了这个图像)

在一个双缓冲应用程序 (例如前面的全屏示例) 中, 交换间隔设置了应该在发生缓冲区交换之前发生的垂直回溯次数。将这个值设置为 1 会强制每次垂直回溯不多于 1 帧, 而将它设置为 2 则允许在两次缓冲区交换之间进行两次垂直回溯。例如, 如果交换间隔设置为 1, 而显示器刷新速率为 60 (典型情况), 那么最终得到的帧速率不会大于每秒 60 帧。对于交换间隔为 2 的情况, 最终的帧速率则不会大于每秒 30 帧, 以此类推。我们可以使用 CGL 函数 `CGLSetParameter` 来设置交换间隔。

```
GLint sync = 1;  
CGLSetParameter (CGLGetCurrentContext(), kCGLCPSwapInterval, &sync);
```

请注意, 这样做并没有将帧速率“固定”到与显示器的刷新速率相等。如果我们完成这项工作的渲染或 CPU 代码占用了过多时间, 那么得到的刷新速率可能会低于显示器的刷新速率。但是, 这种情况下仍然可以做到缓冲区交换只发生在两次刷新之间, 这样就消除了图像撕裂的问题。

## 14.4.2 提高填充性能

填充性能是指渲染中的性能开销, 特别是与向帧缓冲区中的像素写入数据所花费的时间相关。提高填充性能的一种简单方法是, 只渲染到一个小窗口, 或者在诸如游戏这样的全屏应用程序中将屏幕的分辨率改成更小的值。举例来说, 在“雪豹”之前的全屏 OpenGL 游戏中, 在开始运行、捕捉显示器等操作之

前改变屏幕分辨率的情况并不少见。现在我们不再需要显示器捕捉解决方案了，可以利用 CGL 的能力改变后台缓冲区的大小，而不是改变屏幕分辨率。将后台缓冲区大小改成小于前台缓冲区大小，还会使填充性能得到额外的提高，而不需改变显示模式。随后，在发生缓冲区交换时，后台缓冲区的内容将自动伸展填充整个显示器。

为了设置后台表面大小，我们将 CGL 参数 `kCGLCPSurfaceBackingSize` 设置成我们希望的整数维度。另外，我们必须使用 `CGLEnable` 来启用 `kCGLCESurfaceBackingSize` 特性。下面的代码展示了我们应该如何进行这项工作以获得希望的大小，即 `newWidth x newHeight`。

```
GLint dim[2] = {newWidth, newHeight};
CGLSetParameter(CGLGetCurrentContext(), kCGLCPSurfaceBackingSize, dim);
CGLEnable(CGLGetCurrentContext(), kCGLCESurfaceBackingSize);
```

### 14.4.3 多线程 OpenGL

在渲染数据到达硬件进行渲染之前，OpenGL 驱动程序会对这些数据进行大量处理。在 OS X 10.5 或之后的版本中，可以启用一个多线程 OpenGL 核心，将其中一些工作分散到其他线程中进行。在多核心系统中，这样做可以获得可观的性能提升。我们可以通过在 `kCGLCEMPEngine` 标记上调用 `CGLEnable` 启用这个特性。

```
CGLEnable(CGLGetCurrentContext(), kCGLCEMPEngine);
```

这样做并不总是能提高性能，实际上有些时候甚至还会降低性能！例如，如果 OpenGL 代码并没有受到 CPU 处理的制约，那么这样做就很可能只会对渲染性能产生很小的影响，甚至没有影响。再比如说，如果渲染代码调用了很多产生管线延迟的函数（`glGetFloatv`、`glGetIntegerv`、`glReadPixels` 等），那么这些函数极有可能会抵消这些潜在的优化效果。

## 14.5 小结

在本章，我们介绍了如何使用 OpenGL 以及 Interface Builder 和 Cocoa 建立本地 OS X 应用程序。GLUT 有自己的用处，我们介绍了如何在 Objective-C 中使用本地应用程序框架创建一个支持 OpenGL 的 Mac 应用程序。

我们还展示了最新版的 OS X 10.6（雪豹）中的新技术是如何使高性能全屏应用程序的创建比以往更加简单的。最后，我们了解了一些可以在 Mac 的最底层 OpenGL 接口——CGL 中使用的简单技巧。

对于 Macintosh 来说，OpenGL 是一种核心基础技术。对 OpenGL 以及应用程序如何能够与它进行本地交互有一些基本的理解，这对于任何 Mac OS X 开发者来说都是重要的技能。本章只浅显地介绍了一些主题，而这些主题实际上可以进行更加深入和复杂地研究。我们刻意地止步于这个“游泳池”的“浅水区”，以便能够快速地了解它们，并尽可能多地在这个奇妙的平台上体验 OpenGL。在附录 A 中，我们可以找到关于这个令人兴奋的主题的一些很好的补充资料。

## 第 15 章 Linux 上的 OpenGL

作者: Nicholas Haemel

## 本章内容

任 务	使用的函数
视觉管理	glXChooseVisual, glXChooseFBConfig
创建 GLX 窗口	glXCreateWindow
管理 OpenGL 绘图环境	glXCreateContextAttribsARB, glXDestroyContext, glXMakeCurrent
创建 OpenGL 窗口	glXCreateWindow
进行双缓冲绘图	glXSwapBuffers

OpenGL 的一大优点就是能够支持许多不同的平台。我们已经学习了在 Windows 和 Mac OS 上如何使用 OpenGL。现在, 我们开始学习在最流行的开源平台——Linux 上进行 3D 渲染。

在本章, 我们将了解 Linux 如何支持 OpenGL、如何选择 OpenGL 的特定版本、开发人员可以使用哪些接口以及如何设置应用程序。我们还将讨论 GLUT、环境管理以及如何在 X Window 中分配、渲染和处理窗口。

## 15.1 基础知识

差不多从 3D 渲染成为可能开始, OpenGL 就在各种版本的 Linux、UNIX 和类似平台上得到支持了。Linux 提供了几种方法来使用 OpenGL, 大多数主流图形硬件都提供了一些加速形式。

Mesa3D 是一种软件实现, 它并不依赖于硬件, 而且能够安装在大多数 X Server 配置上。

## 15.1.1 简史

在上世纪 80 年代早期, Silicon Graphics ( SGI ) 为了在工作站上进行 2D 和 3D 图形处理而引入了一种专有 API, 命名为 IRIS GL ( Integrated Raster Imaging System Graphics Language, 综合光栅成像系统图形语言 )。1992 年, SGI 对这个实现进行了修订, 并将它作为一种开放的行业标准发布, 这就是 OpenGL。1993 年, Brian Paul 开始进行一项工程, 创建一个纯软件的 OpenGL 实现, 名为 Mesa3D, 由此开启了不受特定硬件供应商约束而广泛支持 3D 渲染的大门。

如今正在使用的大多数计算机系统都能够进行某种类型的 3D 加速。现代 3D 硬件供应商都为最新版本的 OpenGL 提供了支持。目前, ATI/AMD 和 NVIDIA 都提供了支持 OpenGL 3.3 的驱动程序。到本书出版时, 最新版本的 Mesa ( 7.7 版 ) 能够支持 OpenGL 2.1。

## 15.1.2 什么是 X Window

X Window System ( X 窗口系统 ) 是一种图形用户接口, 它为使用者提供了比命令提示符更加直观的环境, 与 Microsoft Windows 和 Mac OS 类似。X Window 会话并不限于在本地系统上使用。例如, 我们可以从计算机开始一个 X Window 会话, 这个会话将对相隔半个国家之外的一台超级计算机进行访问。这样就能够使用远程计算机, 就像我们正坐在它面前一样。在 X Window 术语中, 计算机提供的用户显示服务被称为 X Window Server, 而运行实际应用程序的计算机则被称为客户端。这种叫法可能与通常我们所熟悉的服务器和客户端角色相反。

我们将在 X Window 内部运行 Linux OpenGL 应用程序。大多数 Linux 发布版本使用 X Window System 的 XFree86 实现。我们可以使用很多不同的桌面管理器, 例如 KDE 和 Gnome, 它们都在基本 X Window 软件上运行, 并为用户提供改变窗口大小、加载程序和其他基本操作的互动。

## 15.2 入门讲解

我们需要设置几个组件, 以使 OpenGL 应用程序能够编译和运行。

首先也是最明显的, 我们需要一个 Linux 系统。不同的 Linux 发布版本, 例如 OpenSUSE、Fedora 和 Ubuntu 都可以免费下载。

接下来, 强烈建议安装能够支持当前 OpenGL 版本的现代图形加速卡或带有图形芯片的系统, 支持并安装新的驱动程序也很重要。虽然运行一个 OpenGL 软件实现也是可能的, 但是这些软件实现可能不能完全支持 OpenGL 的所有特性, 而且运行起来非常慢。

我们还需要 OpenGL 和 GLX 的头文件和库。所有这些对于编译应用程序来说都是必要的。

## 15.2.1 检查 OpenGL

让我们快速了解一下如何能够确保系统支持 OpenGL。

如果做不到这一点,那么这一章的其他内容就没有意义了。尝试运行 `glxinfo` 命令,如下所示。

```
glxinfo |grep rendering
```

我们应该得到下面两种响应中的一种。

```
direct rendering: Yes
```

```
direct rendering: No
```

如果结果是 `yes`,这是个好消息!我们的硬件支持 3D 渲染;如果结果是 `no`,那么硬件就可能不支持 OpenGL,或者可能是没有为 OpenGL 安装驱动程序。如果没有得到硬件支持,那么尝试运行下面代码。

```
glxinfo |grep "OpenGL vendor"  
glxinfo |grep "OpenGL version"
```

这样就能打印出当前安装的 OpenGL 驱动程序信息,要特别注意大写字母!如果没有硬件驱动程序,但安装了 Mesa,就会显示 Mesa 驱动程序的信息。这样当前的 OpenGL 版本也能获得 Mesa 实现的支持。

如果 `glxinfo` 命令失败,或者没有可用的提供商/版本信息,那么 Linux 发布版本就不支持 OpenGL 渲染。在这里我们有几个选择。首先,可以安装 Mesa,或者安装一个支持 3D 渲染并且带有 Linux 下驱动程序的显卡。

大多数 Linux 发布版本使用几种软件包管理器中的一种(基于 RPM 或 deb 文件)管理安装的软件。如果 Linux 系统并没有安装 OpenGL 本身、OpenGL 硬件驱动程序或 OpenGL 开发头文件/库,那么我们就需要利用软件包管理器来获取和安装它们。类似 Mesa3D、GLUT 和 GLEW 这样的附加组件可能也可以在发布版本中以包的形式允许简便安装。但是,这些工具以包的形式发布的版本与那些直接从项目网站上下载的版本比起来可能相对过时一些。

## 15.2.2 设置 Mesa

我们可以从 Mesa3D 网站上下载最新版本的 Mesa,本书附录 A 中提供了相应链接。在那里我们可以获得 SourceForge 的下载链接。下载完成后将文件解包(Mesa 7.7 示例)。

```
gunzip MesaLib-7.7.tar.gz  
tar xf MesaLib-7.7.tar
```

接下来,需要对刚刚解包的源文件进行编译。进入刚从 tar 包创建的目录中并运行如下代码。

```
make linux-x86
```

为系统创建 Mesa 软件需要一些时间。创建完成后将生成一些库。现在,我们需要安装库和头文件,

以使操作系统和创建环境在必要的时候能够找到他们。

运行下列命令来完成这项安装工作。

```
make install
```

这个 library 和 include 位置通常位于下列目录中。

```
Libraries: /usr/X11R6/lib
Includes: /usr/include/
```

现在, 我们已经完成了 Mesa 的安装。如果想进一步了解 Mesa 的设置或安装工作, 请访问 Mesa3D 网站。

### 15.2.3 设置 Mesa 硬件驱动程序

如果我们有现代的图像硬件, 就会希望确保安装了最新的驱动程序。不同的硬件供应商支持 Linux 的驱动程序不同, ATI 和 NVIDIA 都提供了拥有专利的驱动程序包可供下载。安装过程通常非常简单, 只要运行下载之后的软件包并依照提示信息来做就可以了。在生产商的网站上能够找到特定的安装指南。某些硬件供应商可能还会提供它们的显示驱动程序的开源版本。

虽然一般来说能够获得驱动程序的源代码非常好, 但这些驱动程序通常很慢, 也不经常更新, 并且特性更少或者比它们相应的正常版本限制更多。值得注意的是, 某些发行版本可能会将驱动程序进行预包装。

这些可能是比较过时的, 而通常最简单的办法就是不要安装这些打包的版本, 而是安装最新的供应商驱动程序。

### 15.2.4 设置 GLUT 和 GLEW

本书前面内容已经对 GLUT 进行过介绍了。基本上讲, 它是一组非常实用的函数, 帮助我们创建 OpenGL 的接口, 使系统对用户更加友好, 并且负责处理许多无法看到的细节。用 GLUT 编写的 OpenGL 代码是平台独立的, 这使代码具有很好的可移植性。本书所采用的是 freeglut 版本, 它比 GLUT 的原始版本更新。

GLUT 和 freeglut 可以用于在 Linux 中进行下载和安装工作, 就像在其他操作系统中一样。因为这些代码可以在 Windows、Mac 和 Linux 上进行编译, 所以这样可以使任何使用 GLUT 的应用程序变得容易移植。对于应用程序的快速准备和运行来说, 这也是一种很好的方式, 因为它不需要窗口管理。

GLUT 并不支持与 X Server 的直接接口。这就意味着某些可以通过直接与操作系统或 X Server 进行通信而完成的工作, 在使用 GLUT 时将会更加困难, 甚至不可能完成了。

GLEW (GL extension wrangler) 在本书前面内容中也已经使用过了。GLEW 提供了一组工具, 帮助我们在 OpenGL 中加载扩展。类似于 GLUT, GLEW 也可以用于很多不同的操作系统和平台。通过使用 GLEW, 应用程序能够专注于 3D 渲染, 而不需要过多地关心如何在不同的平台上使用。

## 安装 GLUT

在系统中可能还没有安装 GLUT。即使如此，它也能够很容易地下载。然后进入 GLUT 目录并执行如下命令。

```
./mkmkfiles.imake  
make  
make install
```

第一条命令用于创建用来编译代码的 makefile 文件。Makefile 文件是为每个系统自定义创建的，因为不同系统中的不同资源可能会被分配到不同的位置。第二条命令会实际进行代码编译，而第 3 条命令则对结果进行安装。

要在应用程序中使用 GLUT，就需要将 GLUT 库添加到连接命令中。

```
-lglut      .
```

Mesa3D 还支持一个能够进行下载和安装的 GLUT 版本。

## 安装 GLEW

GLEW 非常容易理解，它包含在两个头文件和一个单独源文件中。本书发布的源代码的\GLTools 目录中提供了这些文件。

要想自动访问驱动程序支持的所有函数指针非常简单，只需在项目中添加 glew.c，并在头文件列表的顶部添加 glew.h。然后，在应用程序开始进行任何 OpenGL 调用之前，先调用 glewInit()。OpenGL 1.1 以上版本的扩展和核心特性的所有函数指针都将自动进行设置。如果 glewInit 函数失败，它会返回一个错误，扩展指针则可能不会被初始化。

## 15.2.5 创建 OpenGL 应用程序

现在，我们了解了全部设置过程，并且系统已经做好了运行和编译 OpenGL 程序的准备，现在让我们来看一看如何创建这些程序。如果读者已经进行过一些 Linux 相关工作，那么可能就已经对创建 makefile 文件比较熟悉了。

如果是这样的话，跳过这些内容就是了。

makefile 文件用于在 Linux 系统中对源代码进行编译和连接，并且创建可执行文件。makefile 文件中保存了编译器和连接器所需的指令，这些指令告诉它们在哪里可以找到文件，以及对这些文件进行什么样的处理。接下来是一个 makefile 文件的示例。对它进行一些修改和扩展，就可以用于我们的项目了。

```
LIBDIRS = -L/usr/X11R6/lib -L/usr/X11R6/lib64 -L/usr/local/lib  
INCDIRS = -I/usr/include -L/usr/local/include
```

```
CC = gcc
CFLAGS = $(COMPILERFLAGS) -g $(INCDIRS)
LIBS = -lX11 -lXi -lXmu -lglut -lGL -lGLU -lm

example : example.o
    $(CC) $(CFLAGS) -o example $(LIBDIRS) example.c $(LIBS)
clean:
    rm -f *.o
```

第一行代码创建一个变量，其中的内容为将要包含库的连接参数。这里使用的代码既对 64 位特定库的标准 lib 目录进行查询，也对 X11 的标准 lib 目录进行查询。

第二行代码列出了编译器在试图查找头文件时应该使用的 include 路径。CC = gcc 选择要使用的编译器。接下来的一行代码指定这个实例将要使用的编译标记。然后，LIBS = 选择所有需要连接到程序中的库。

最后，我们对为这个示例所指定的名为 example.c 的单个源文件进行编译和连接。最后一行代码负责对在处理过程中创建的中间对象进行清除。这个示例可以用来在脚本中代替我们的文件，其他文件也可以一起进行编译。网上还可以找到很多资源和教程，而在附录 A 中则有两个很好的 makefile 基础资料可以帮助读者入门。

## 15.3 GLX——X Window 的接口

X Window 提供了一个通用接口，叫做 GLX，它允许使用 OpenGL 的应用程序与 X Window 进行通信。这个接口与 Microsoft Windows 上的 WGL 和 Mac 上的 AGL 非常相似。GLX 有许多不同的版本，1.4 版则是最新的版本。GLX 1.4 与 GLX 1.3 版非常相似，但也进行了一些细微的修改。GLX 1.2 则是更老的版本，缺少很多新版本中的功能。出于以上原因，在应用程序中将使用 GLX 1.4。

我们可以再次使用 glxinfo 命令，来查找更多关于 GLX 安装的信息。

```
glxinfo |grep "glx vendor"
glxinfo |grep "glx version"
```

这些代码能够显示 X Window 的服务器和客户端组件的 GLX 信息。

我们可以使用的有效版本，是服务器版本和客户端版本中较老的一个。这样，如果客户端版本为 1.4 而服务器版本为 1.3，那么我们就只能使用 GLX 1.3 了。如果客户端或服务器驱动程序不支持 GLX 1.4，可以尝试对显示驱动程序进行升级，就像我们前面描述的那样。

我们也可以在程序内部调用 glXQueryVersion 来获取 GLX 版本信息。

```
Bool glXQueryVersion(Display * dpy, int *major, int *minor);
```

这项调用如下所示。

```
int majorVer, minorVer;
glXQueryVersion(dpy, majorVer, minorVer);
```

### 15.3.1 显示和 X Window

在我们更进一步研究使用 GLX 之前, 还需要几个前提条件, 以便理解 GLX 在 Linux 上是如何工作的。一个 OpenGL 应用程序是在 X Server 上的一个窗口中运行的。前面已经提到, X Window 支持各自在独立系统中运行的客户端和服务组件, 这样就从原理上使我们能够从其他地方运行桌面系统了。此外, 一个 X Server 可以支持多个活动的显示器, 甚至多个图形加速卡。

我们需要弄清 OpenGL 应用程序将在哪个显示器上执行, 然后才能创建一个窗口。显示器能够帮助 X Server 理解我们正在哪里进行渲染, 可以使用 `XOpenDisplay()` 函数获取当前显示器。

```
Display *dpy = XOpenDisplay(getenv("DISPLAY"));
```

这就为我们提供了一个指向默认显示器的显示器对象的指针, 稍后可以使用它来告诉 X Server 我们在哪里进行显示。在应用程序运行完成之后, 还需要使用 `XCLOSEDisplay()` 函数来关闭显示。这样就告诉 X Server 我们的工作已经完成了, 它可以关闭连接了。

```
XCLOSEDisplay(Display * display);
```

### 15.3.2 配置管理和显示效果

我们需要先了解要求什么样的特性, 然后才能创建窗口或 OpenGL 渲染环境。Linux 中的配置与 OpenGL ES 中的配置或 Windows 中的像素格式非常相似。一个配置就是一个由 X Window 或 OpenGL/GLX 驱动程序支持的属性组成的枚举集。一个实现通常支持很多窗口和渲染属性的组合, 从而支持大量配置。因为配置是由很多因素组合而成的, 所以它们处理起来可能会非常麻烦。

刚开始的时候, 我们可以使用 `glXGetFBConfigs` 接口获取所有能够支持的配置的信息。

```
GLXFBConfig *glXGetFBConfigs(Display * dpy, int screen, int *nelements);
```

这里使用通过调用 `XOpenDisplay` 获取的显示句柄。我们在这里可以使用默认屏幕为 `screen` 参数。当调用返回时, `nelements` 会告诉我们返回了多少配置。

在每个配置中不仅仅是它的索引, 每个配置都有一个唯一的属性集, 它代表着各配置的功能。表 15.1 列出了这些属性和它们的描述。

表 15.1

配置属性列表

类 型	描 述
GLX_BUFFER_SIZE	颜色缓冲区的总位数
GLX_RED_SIZE	颜色缓冲区的红色通道的位数
GLX_GREEN_SIZE	颜色缓冲区的绿色通道道的位数
GLX_BLUE_SIZE	颜色缓冲区的蓝色通道的位数
GLX_ALPHA_SIZE	颜色缓冲区的 alpha 通道的位数

续表

类 型	描 述
GLX_DEPTH_SIZE	深度缓冲区的位数
GLX_STENCIL_SIZE	模板缓冲区的位数
GLX_CONFIG_CAVEAT	设置为下面几个值之一：GLX_NONE、GLX_SLOW_CONFIG 或 GLX_NON_CONFORMANT_CONFIG。它们能够对于这个配置的潜在问题进行警告。低速配置可能是软件模拟的，因为它超过了硬件限制。不一致的配置将无法通过一致性测试
GLX_X_RENDERABLE	如果 X_Server 可以渲染到这个表面，那么它设置为 GLX_TRUE
GLX_VISUAL_ID	相关画面的 XID
GLX_X_VISUAL_TYPE	如果配置支持窗口渲染，那么它表示 X 画面的类型（相关画面存在）
GLX_DRAWABLE_TYPE	得到支持的有效画面对象。可以是 GLX_WINDOW_BIT、GLX_PIXMAP_BIT 或者 GLX_PBUFFER_BIT 中的任意几个或全部
GLX_RENDER_TYPE	这个字段表示可以绑定的环境类型。可以是 GLX_RGBA_BIT 或 GLX_COLOR_INDEX_BIT
GLX_FBCONFIG_ID	GLXFBConfig 的 XID
GLX_LEVEL	帧缓冲区的层次
GLX_DOUBLEBUFFER	如果颜色缓冲区是双缓冲的，那么它的值为 GLX_TRUE
GLX_STEREO	如果颜色缓冲区支持立体渲染，那么它的值为 GLX_TRUE
GLX_SAMPLE_BUFFERS	多重采样缓冲区的数量，必须是 0 或者 1
GLX_SAMPLES	多重采样缓冲区中每个像素的样本数量。如果 GLX_SAMPLE_BUFFERS 为 0，那么它也必须是 0
GLX_TRANSPARENT_TYPE	表示对透明的支持。它的值可以是 GLX_NONE、GLX_TRANSPARENT_RGB 或 GLX_TRANSPARENT_INDEX。如果支持透明，当一个像素的成分都等于各自的透明 RGB 值时，就会绘制一个透明像素
GLX_TRANSPARENT_RED_VALUE	一个帧缓冲区像素为透明时必须取这个红色值
GLX_TRANSPARENT_GREEN_VALUE	一个帧缓冲区像素为透明时必须取这个绿色值
GLX_TRANSPARENT_BLUE_VALUE	一个帧缓冲区像素为透明时必须取这个蓝色值
GLX_TRANSPARENT_ALPHA_VALUE	一个帧缓冲区像素为透明时必须取这个 alpha 值
GLX_TRANSPARENT_INDEX_VALUE	一个帧缓冲区像素为透明时必须取这个索引值，只适用于颜色索引配置
GLX_AUX_BUFFERS	能够支持的辅助缓冲区数量
GLX_ACCUM_RED_SIZE	辅助缓冲区红色通道的位数
GLX_ACCUM_GREEN_SIZE	辅助缓冲区绿色通道道的位数
GLX_ACCUM_BLUE_SIZE	辅助缓冲区蓝色通道道的位数
GLX_ACCUM_ALPHA_SIZE	辅助缓冲区 alpha 通道道的位数

我们可以使用 `glXGetFBConfigAttrib` 命令对任何配置进行查询，来找出这些属性的值。

```
int glXGetFBConfigAttrib(Display * dpy, GLXFBConfig config,
                        int attribute, int *value);
```

将这个配置参数设置为我们在查询中感兴趣的配置数量,而这个属性参数则设置为我们想要查询的属性。得到的结果将返回到 value 参数中。当 glXGetFBConfigAttrib 调用失败时,如果查询的属性不存在,那么它可能会返回 GLX\_BAD\_ATTRIBUTE 错误。

GLX 还提供了一种方法,能够获取一个符合一组规则的配置子集。这样有助于将整个集合缩小到只包含我们所关心的那些配置,这使我们能够更加容易地找到适合应用程序的配置。举例来说,如果我们要将一个应用程序渲染到一个窗口中,那么所选择的配置就要支持窗口渲染。

```
GLXFBConfig *glXChooseFBConfig(Display * dpy, int screen,  
                                const int *attrib_list, int *nelements);
```

将我们感兴趣的屏幕作为 screen 参数传入,并在进行配置匹配时指定那些所需的元素。这些工作是通过一个由参数和数值对组成的列表完成的,这个列表以一个 NULL 结束。这些属性与表 15.1 列出的配置属性相同。

```
attrib_list = {attribute1, attribute_value1,  
               attribute2, attribute_value2,  
               attribute3, attribute_value3,  
               0};
```

类似于 glXGetFBConfigs,与参数列表相匹配的配置数量会返回到 nelements 中。这个函数将返回一个指向匹配配置列表的指针。

不要忘记使用 XFree 对调用 glXChooseFBConfig 返回的内存进行清理。返回的所有配置都会符合我们在属性列表中设置的最低要求。

在创建一个配置时,我们可能想要特别注意一些关键属性。例如, GLX\_X\_RENDERABLE 应该为 GLX\_TRUE,以便我们能够使用 OpenGL 来执行渲染。如果我們是在窗口中进行渲染的话,那么 GLX\_DRAWABLE\_TYPE 就需要包含 GLX\_WINDOW\_BIT; GLX\_RENDER\_TYPE 应该为 GLX\_RGBA\_BIT,而 GLX\_CONFIG\_CAVEAT 则应该设置为 GLX\_NONE,否则至少不能对 GLX\_SLOW\_CONFIG 位进行设置。

在这之后,我们可能还想确保颜色通道、深度通道和模板通道符合最低要求。pBuffer、accumulation (累积) 和 transparency (透明度) 值并不经常使用。

对于那些我们没有指定的属性来说, glXChooseFBConfigs 命令可以隐式地将它们设置为默认值。GLX 规范中罗列了这些内容。排序机制能够使用属性优先级来对返回属性的列表进行自动排序。优先级最高的属性的顺序为 GLX\_CONFIG\_CAVEAT、颜色缓冲区的位深度、GLX\_BUFFER\_SIZE 和 GLX\_DOUBLEBUFFER。

如果一个配置为 GLX\_DRAWABLE\_TYPE 属性设置了 GLX\_WINDOW\_BIT,那么这个配置将具有一个相关联的 X 画面。这个画面可以使用下面的命令进行查询。

```
XVisualInfo *glXGetVisualFromFBConfig(Display * dpy, GLXFBConfig config);
```

如果不存在相关联的 X 画面,那么这个函数将返回 NULL。不要忘记用 XFree 释放这个函数所返回的内存。

我们不再讨论 pBuffer,因为它们已经“不推荐”了,并且硬件供应商可能也不再支持它们了。有许

多更加灵活的方法可以实现离屏渲染，或者在没有窗口的情况下进行渲染。像素图的情况也是如此，帧缓冲对象取代了这项功能。另外，我们也不再讨论颜色索引模式，因为它已“不推荐”了，并且大多数基于 PC 的实现已经不再支持它。

### 15.3.3 窗口和渲染表面

现在我们已经理清了一些概念，让我们来创建一个窗口吧。X 提供了一个 X Server 函数 `XCreateWindow`，我们可以用它来创建窗口。这个函数返回新的 X 窗口的句柄。这个命令需要一个父窗口，但是我们也可以使用 X 主窗口实现这个目的，并且读者应该已经熟悉这里所使用的 `Display` 参数了。另外，还需要告诉 X 我们希望的窗口大小，并用 *x*、*y* 位置和宽度/高度参数指定窗口的位置。

另外，还要用窗口类告诉 X 我们需要创建什么类型的窗口。它的值可以是下列三者之一：`InputOnly`、`InputOutput` 或 `CopyFromParent`。一个 `InputOnly` 窗口无法作为图形请求的源或目标使用，并且 `CopyFromParent` 值继承了父窗口建立时使用的值，所以 `InputOutput` 是最有用的。

`attributes` 和 `valuemask` 字段能帮我们告诉 X，这个窗口应该具有哪些类型的特征。`attributes` 字段保存这些值，而 `valuemask` 则告诉 X 应该注意哪些值。要了解关于属性的更多信息，请参考 X Server 文档。全部函数声明如下所示。

```
Window XCreateWindow(Display * dpy, Window parent, int x, int y,
                    unsigned int width, unsigned int height,
                    unsigned int border_width, int depth,
                    unsigned int class, Visual *visual,
                    unsigned long valuemask,
                    XSetWindowAttributes *attributes);
```

为创建窗口选择了合适的值并且调用 `XCreateWindow` 之后，将会返回新窗口的句柄。然后，这个窗口句柄就用来创建相应的 GLX 窗口。在创建 GLX 窗口时，我们使用的配置必须与创建 X Window 时的视觉相兼容。使用 `glXCreateWindow` 命令创建一个新的在屏 OpenGL 渲染区域与新创建的 X Window 相关联。

```
GLXWindow glXCreateWindow(Display * dpy, GLXFBConfig config,
                          Window win, const int *attrib_list);
```

到目前为止，我们已经熟悉了 `Display` 参数。我们可以采用在使用 `glXGetFBConfigs` 或 `glXChooseFBConfig` 的部分已经选择的配置。这个窗口句柄与 `XCreateWindow` 返回的句柄是相同的。`attrib_list` 目前并不支持任何参数，考虑到将来的扩展，我们应该传递 `NULL`。

如果这个配置与窗口视觉不兼容，或者这个配置不支持窗口渲染，或者窗口参数不合法，或者一个 `GLXFBConfig` 已经与窗口进行关联，或者 `GLXFBConfig` 不合法，又或者在生成窗口时出现了一般错误，`glXCreateWindow` 都会抛出一个错误并且会创建失败。我们还要记住，只有 GLX 1.3 或更新的版本才支持 `glXCreateWindow`，它在老版本中无法使用。读者可能还会记得，前面我们已经通过在一个终端上运行 `glxinfo | grep "glx version"` 来检查了 GLX 的版本。

一旦渲染完成，我们还必须清除所创建的窗口。要销毁 GLX 窗口，需要调用以在调用

glXCreateWindow 时返回的窗口句柄为参数的 glXDestroyWindow。

```
glXDestroyWindow(Display * dpy, GLXWindow window);
```

最后，销毁开始时创建的 X Window。我们可以使用与前面类似的 XDestroyWindow 命令并返回 X Window 句柄。

```
XDestroyWindow(Display * dpy, Window win);
```

## GLX 字符串

从 GLX 中可以查询字符串，以更多地了解我们的操作系统能做什么。最重要的字符串之一就是扩展字符串。这是一个列表，包含了当前 GLX 实现所支持的有扩展。要获取扩展字符串，可以使用

```
const char *glXQueryExtensionsString(Display *dpy, int screen);
```

返回的字符串，或者说字符数组，是一个由空格分隔的扩展名列表，这个数组以数值 0 结束。

我们还可以调用 glXGetClientString 或 glXQueryServerString 分别查询关于客户端库或服务器的信息。用下列枚举值中的一个作为名称变量值：GLX\_VENDOR、GLX\_VERSION 或 GLX\_EXTENSIONS。

```
const char *glXGetClientString(Display *dpy, int name);  
const char *glXQueryServerString(Display *dpy, int screen, int name);
```

## 15.3.4 OpenGL 和 GLX 扩展

在进一步学习之前，让我们先来了解一下如何在不创建一个全新 GLX 版本的情况下对 GLX 进行扩展。供应商能够为 GLX 和 OpenGL 编写新扩展，从而为应用程序增添新功能。这样就使得应用程序能够使用供应商特定的特性，或者在它们成为核心规范的一部分之前就可以使用了。

我们刚刚学习了如何通过调用 glXQueryExtensionString 获得 GLX 扩展的列表。在第 2 章，我们还学习了如何获取所有 OpenGL 扩展的列表。

在网上的 OpenGL 扩展库中能够找到关于新扩展的描述。在了解了哪些扩展可用以及它们是做什么的之后，我们就可以获取新的入口以使用这些扩展了。GLX 提供的 glXGetProcAddress 可以查询扩展的函数地址。

```
void (*glXGetProcAddress(const ubyte *procname))();
```

## 15.3.5 环境管理

一个环境就是一组与某个句柄相关联的 OpenGL 状态。一个环境必须被绑定到一个可绘制元素（例如一个窗口），以进行状态设置或进行渲染，可以创建多个环境，但是每一次只有一个环境能够绑定到可绘

制元素上。至少必须创建一个环境，应用程序才能进行渲染。

## 创建环境

`glXCreateNewContext` 命令是一种可以用来创建一个新环境的方法。

```
GLXContext glXCreateNewContext(Display * dpy, GLXFBConfig config,
                               int render_type, GLXContext share_list, bool direct);
```

如果创建成功，那么这个函数会返回一个环境句柄，我们可以用它来告诉 GLX 我们在进行渲染时想要使用哪个环境。我们用来创建这个环境的配置需要与想要在其中进行渲染的表面相兼容。在一般情况下，采用与创建 GLX 窗口时所使用的同一个配置是最简单的方式。

`render_type` 参数接受 `GLX_RGBA_TYPE` 或 `GLX_COLOR_INDEX_TYPE`。因为我们没有使用颜色索引模式，所以应该使用 `GLX_RGBA_TYPE`。大多数实现都已经不再支持颜色索引模式了。一般情况下，我们还应该使用 `NULL` 作为 `share_list` 参数的值。但是，如果一个应用程序有多个环境，并且想要共享诸如 textures、VBOs 和 FBOs 等对象，那么在创建第二个环境时可以传入第一个环境的句柄。这样会使这两个环境都使用同一个名字空间。将 `direct`（直接）参数设为 `TRUE` 会为 X Server 连接请求一个直接硬件环境，而设为 `FALSE` 则可能创建一个通过 X Server 进行渲染的环境。

如果创建失败的话，函数将会返回 `NULL`；否则，它会将环境初始化为默认的 OpenGL 状态。如果我们传递了一个无效的 `share_list` 句柄作为参数，或者这个配置是无效的，又或者系统的资源耗尽的话，那么这个函数将会抛出一个错误。

如果我们的实现支持 OpenGL 3.1，或者支持任何能够 100% 向下兼容 OpenGL 3.1 的更新环境，那么创建这个环境的 OpenGL 版本将达到 OpenGL 3.1。因为当调用 `glXCreateNewContext` 时并不能确定我们将要获得哪个版本的 OpenGL 环境，所以这并不是最好的办法。我们使用更新的版本 `glXCreateContextAttribsARB` 代替它。

在使用 `glXCreateContextAttribsARB` 之前，我们应该先检查确认 `GLX_ARB_create_context_profile` 字符串已经包含在了 GLX 扩展的列表中。然后，我们需要获取这个扩展的函数指针。在这之后，我们就做好了使用这种方法创建环境的准备。

```
GLint attribs[] = {
    GLX_CONTEXT_MAJOR_VERSION_ARB, 3,
    GLX_CONTEXT_MINOR_VERSION_ARB, 3,
    0 };
rcx->ctx = glXCreateContextAttribsARB(rcx->dpy, fbConfigs[0], 0,
                                     True, attribs);
glXMakeCurrent(rcx->dpy, rcx->win, rcx->ctx);
```

更新的 `glXCreateContextAttribsARB` 方法接受一个附加的参数，使我们能够准确地选择我们希望的环境。

```
GLXContext glXCreateContextAttribsARB(Display * dpy, GLXFBConfig config,
                                       int render_type, GLXContext share_list, bool direct,
                                       const int *attrib_list);
```

`attrib_list` 参数是一个属性的数值对列表，我们可以在一个新环境中对其进行查询。首先，在这个数

组中指定属性名称，然后指定属性值。GLX\_CONTEXT\_MAJOR\_VERSION\_ARB 和 GLX\_CONTEXT\_MINOR\_VERSION\_ARB 属性用来显式地要求特定的 OpenGL 环境版本。如果应用程序是基于 OpenGL 3.3 编写的，那么就要在主版本号中传入 3，在次版本号中也传入 3。类似地，如果应用程序更老，并且需要一个 OpenGL 3.0 环境，也可以对版本号进行要求。但是，OpenGL 驱动程序也可以返回任何百分之百向下兼容我们要求版本的更新版本。如果我们不指定一个 OpenGL 版本，或者我们要求 1.0 版，那么驱动程序可能将会创建一个 OpenGL 3.1 环境。具体的行为在各个供应商之间有所不同，最好的办法就是要求特定的 OpenGL 版本。

我们只能创建一个我们的 OpenGL 驱动程序支持的版本下的环境。可以通过调用带有 GL\_VERSION 枚举的 glGetString 查询能够支持的最新版本。

```
ubyte *verString = glGetString(GL_VERSION);
```

或者也可以通过 glGetIntegerv 命令查询版本，这个命令会将版本号作为整数分量返回。

```
int majorVer, minorVer;  
glGetIntegerv(GL_MAJOR_VERSION, &majorVer);  
glGetIntegerv(GL_MINOR_VERSION, &minorVer);
```

我们还可以通过 attrib\_list 查询一些其他类型的属性。GLX\_CONTEXT\_PROFILE\_MASK\_ARB 属性后面是一个位段，其中包含 GLX\_CONTEXT\_CORE\_PROFILE\_BIT\_ARB 或 GLX\_CONTEXT\_COMPATIBILITY\_PROFILE\_BIT\_ARB。

每一次只能使用一位。设置 GLX\_CONTEXT\_CORE\_PROFILE\_BIT\_ARB 位会导致驱动程序返回一个只包含核心功能的环境，而不会包含任何“不推荐”的 OpenGL 功能。如果是为了 OpenGL 的下一次修订（“不推荐”的 OpenGL 功能可能会被删除）而准备应用程序的话，那么使用这个位是一个好办法。设置 GLX\_CONTEXT\_COMPATIBILITY\_PROFILE\_BIT\_ARB 位会要求驱动程序创建一个能够兼容所有 OpenGL 老版本的环境。换句话说，就是不会删除任何“不推荐”的功能。设置这个位而创建的环境可能会比一个核心版本的环境运行速度慢，因为需要跟踪额外的状态和功能。

GLX\_CONTEXT\_FLAGS\_ARB 属性能够用来为环境创建设置其他标志。唯一支持的标志就是 GLX\_CONTEXT\_DEBUG\_BIT。指定这个位所创建的环境能够在应用程序开发过程中为其提供可用的调试信息。

至于提供什么样的信息，以及如何访问这些信息，就是供应商指定的了。如果系统中的 OpenGL 驱动程序不支持指定的任何属性，那么将会生成一个错误。如果向下兼容环境中主版本号和次版本号的组合不是一个有效的 OpenGL 版本号，那么就会抛出 GLXBadMatch 错误。如果为 GLX\_CONTEXT\_PROFILE\_MASK\_ARB 指定的任何一位都不被支持，那么就会抛出 GLXBadProfileARB 错误。

当不再使用一个环境之后，要销毁这个环境，以便实现可以释放所有相关的资源，这一点非常重要。我们使用 glXDestroyContext 命令对环境进行销毁。

```
glXDestroyContext(Display * dpy, GLXContext ctx);
```

如果这个环境当前还被绑定在某个线程上，那么直到这个环境不再是当前环境之后才会被销毁。如果我们传递一个无效的环境句柄，那么函数将抛出一个错误。

GLX 还提供了另外一个方便的特性，就是使用 glXCopyContext 将数据从一个环境复制到另一个环

境。我们需要传递源环境和目的环境，以及一个用于指定我们想要复制哪些 OpenGL 状态的遮罩。它们和可能传递到 `glPushAttrib/glPopAttrib` 中的枚举值相同。我们可以传递 `GL_ALL_ATTRIB_BITS` 来复制所有内容。客户端的状态是不能被复制的。

```
glXCopyContext(Display * dpy, GLXContext source, GLXContext dest, unsigned long
               mask);
```

在 GLX 中，直接环境就是支持直接渲染到一个本地 X Server 的环境。我们可以调用 `glXIsDirect` 查询一个已经存在的环境是不是直接环境。如果这个环境是直接渲染环境，那么就会返回 `true`。

```
glXIsDirect(Display * dpy, GLXContext ctx);
```

### 调试环境

使用一个调试环境，能够帮助我们确定应用程序在哪里出现了问题。在本书编写时，AMD 是唯一提供了调试环境的供应商，这个调试环境是通过一个称为 `GL_AMD_debug_context` 的扩展提供的，这个扩展对开发者如何访问这些附加调试信息进行了定义。

这里提供了一个回调函数，允许一个应用程序设置一个中断或中断点，并且在出现错误时能够立即发现。这个扩展还允许一个应用程序选择特定错误类型来进行监视，并且支持多个严重性等级。

关于如何使用 `GL_AMD_debug_context` 扩展的更多信息，可以查询扩展规范，它也包含在 OpenGL 扩展注册中。

### 使用环境

我们可以调用 `glXMakeContextCurrent` 来使用一个已经创建的环境。

```
glXMakeContextCurrent(Display * dpy, GLXDrawable draw, GLXDrawable read,
                     GLXContext ctx);
```

在大多数情况下，我们应当为一个环境的读写和绘制指定同一个可绘制元素。这就意味着将会在读取和绘制操作中使用同一个环境。如果在进行这项调用之前已经绑定了一个不同的环境，那么它将被清理并标记为非当前环境。如果传递的环境不是有效的，或者可绘制元素不是有效的，那么这个函数将会抛出一个错误。如果这个环境的配置与创建可绘制元素时使用的配置不兼容，它也会抛出一个错误。将可绘制元素的 `read` 和 `draw` 参数设为 `None`，并以 `NULL` 作为环境，可以对一个线程中的环境进行释放。如果没有传递 `None` 作为可绘制元素，那么 GLX 将会抛出一个错误。

## 15.3.6 同步

GLX 还有几个同步命令，它们与其他操作系统的同步命令相似。

```
void glXWaitGL(void);
```

调用 `glXWaitGL`，可以确保在允许处理 `glXWaitGL` 调用之后出现其他本地渲染之前，一个窗口的所有 GL 渲染都会结束。这样就使应用程序能够确保所有渲染以正确的顺序进行，而使渲染不会以错误的方式发生重叠或覆盖。

在某些实现中，对 `glXWaitGL` 的调用可能会立即返回，而不产生任何可见的渲染。一个实现可能会在较早进行的渲染完成之前就开始等待另一个渲染进行初始化。

```
void glXWaitX(void);
```

类似地，调用 `glXWaitGL` 可以确保在允许处理 `glXWaitGL` 调用之后出现的其他本地渲染完成之后，再进行在允许执行这个调用之后出现的任何渲染。

```
void glXSwapBuffers(Display *dpy, GLXDrawable draw);
```

在使用一个双缓冲配置时，调用 `glXSwapBuffers` 能够显示窗口后台缓冲区中的内容。这个调用还会在进行交换之前隐式地执行一次 `glFlush` 操作。另外，新后台缓冲区的内容将为未定义的。我们不应该在调用 `glXSwapBuffers` 之后就假定新的后台缓冲区内容和旧的后台缓冲区、前台缓冲区或任何其他已定义内容一样，以保留供应商之间的可移植性。如果可绘制元素是无效的，或者显示是无效的，那么 GLX 将会抛出一个错误。

### 15.3.7 GLX 查询

GLX 还允许我们查询一个环境的特定属性。使用 `glXQueryContext` 命令查询与环境相关的 `GLX_FBCONFIG_ID`、`GLX_RENDER_TYPE` 或 `GLX_SCREEN` 属性。

```
int glXQueryContext(Display * dpy, GLXContext ctx, int attribute, int *value);
```

在 GLX 中还有一些其他与环境相关的命令，它们的名称基本上都是自我描述的。

`glXGetCurrentReadDrawable` 会返回当前读取的可绘制元素句柄。

```
GLXDrawable glXGetCurrentReadDrawable(void);
```

另外，当前环境、可绘制元素和显示能够通过下列函数进行查询。

```
GLXContext glXGetCurrentContext(void);
GLXDrawable glXGetCurrentDrawable(void);
GLXDrawable glXGetCurrentReadDrawable(void);
Display glXGetCurrentDisplay(void);
```

GLX 中还有一些不太常见的组件，我们还没有对它们进行介绍。为了本书内容的完整，我们来快速地了解一下它们。我们可以使用 `glXQueryDrawable` 函数来从当前可绘制元素中查询特定状态。传递我们感兴趣的可执行元素和属性：`GLX_WIDTH`、`GLX_HEIGHT`、`GLX_PRESERVED_CONTENTS`、`GLX_LARGEST_PBUFFER` 或 `GLX_FBCONFIG_ID`。结果将返回到 `value` 字段。

```
void glXQueryDrawable(Display *dpy, GLXDrawable draw, int attribute, unsigned int *value);
```

另外还有一组用来创建、处理和删除位图和 `pBuffer` 的函数。在这里我们并不打算介绍它们，因为我

们已经不使用位图和 pBuffer 的函数了，也不推荐读者使用它们。

### 15.3.8 综合运用

现在，有趣的部分开始了！让我们综合运用这些 GLX 内容来创建应用程序，在程序中使用 GLX 来代替 GLUT 进行窗口的创建和维护。GLUT 非常适合用来创建快速、简单的应用程序，但是却无法对 GLX 环境进行非常精细的控制。

本章提供两个简单的程序。Block 与我们从第 1 章中看到的程序相同，但是这一次我们使用 GLX 来代替 GLUT。GLXBasics 是一个从头编写的应用程序，它使用 GLX，同时演示了对 GLX 回调的处理，包括如何对鼠标位置进行解释。第一步是打开一个到 X Server 的连接。

```
rcx->dpy = XOpenDisplay(NULL);
```

然后，让我们来检查支持的 GLX 版本，以确保能够支持接下来将使用的功能。

```
glXQueryVersion(rcx->dpy, &nMajorVer, &nMinorVer);
printf("Supported GLX version - %d.%d\n", nMajorVer, nMinorVer);

if(nMajorVer == 1 && nMinorVer < 3)
{
    printf("ERROR: GLX 1.3 or greater is necessary\n");
    XCloseDisplay(rcx->dpy);
    exit(0);
}
```

现在我们已经准备好了，寻找一个符合我们要求的配置。考虑到这个应用程序并不包含任何与帧缓冲区的复杂交互，我们在这里并未特别苛求。

```
GLXFBConfig *fbConfigs;
int numConfigs = 0;
static int fbAttribs[] = {
    GLX_RENDER_TYPE,    GLX_RGBA_BIT,
    GLX_X_RENDERABLE,   True,
    GLX_DRAWABLE_TYPE,  GLX_WINDOW_BIT,
    GLX_DOUBLEBUFFER,   True,
    GLX_RED_SIZE,        8,
    GLX_BLUE_SIZE,       8,
    GLX_GREEN_SIZE,      8,
    0 };
// 获取一个符合我们要求的新帧缓冲区配置
fbConfigs = glXChooseFBConfig(rcx->dpy, DefaultScreen(rcx->dpy),
                             fbAttribs, &numConfigs);
```

我们还需要一个视觉来创建 X Window。一旦有了一个配置，就可以从它这里获得相应的视觉了。

```
XVisualInfo *visualInfo;
visualInfo = glXGetVisualFromFBConfig(rcx->dpy, fbConfigs[0]);
```

在获得一个视觉之后，就可以用它来创建新的 X Window 了。在调用 XCreateWindow 之前，我们必须清楚想要这个窗口来做什么。挑选有用的事件，并将它们添加到事件遮罩中。对窗口遮罩也进行同样操作。设置我们需要的边界大小和位置。我们还需要为将要使用的窗口创建一个颜色映射，这时将窗口映射到显示上。

```
winAttribs.event_mask = ExposureMask | VisibilityChangeMask |
                        KeyPressMask | PointerMotionMask |
                        StructureNotifyMask ;

winAttribs.border_pixel = 0;
winAttribs.bit_gravity = StaticGravity;
winAttribs.colormap = XCreateColormap(rcx->dpy,
                                     RootWindow(rcx->dpy, visualInfo->screen),
                                     visualInfo->visual, AllocNone);
winmask = CWBorderPixel | CWBitGravity | CWEventMask | CWColormap;

rcx->win = XCreateWindow(rcx->dpy, DefaultRootWindow(rcx->dpy), 20, 20,
                        rcx->nWinWidth, rcx->nWinHeight, 0,
                        visualInfo->depth, InputOutput,
                        visualInfo->visual, winmask, &winAttribs);

XMapWindow(rcx->dpy, rcx->win);
```

好了！我们有了一个窗口！在进行渲染之前，还要完成几个步骤。

• 首先，创建一个环境并将它设置为当前环境。请记住，我们需要与用来创建窗口的视觉相对应的配置创建环境。

```
// 还要创建一个新的 GL 环境进行渲染
GLint attribs[] = {
    GLX_CONTEXT_MAJOR_VERSION_ARB, 3,
    GLX_CONTEXT_MINOR_VERSION_ARB, 3,
    0 };
rcx->ctx = glXCreateContextAttribsARB(rcx->dpy, fbConfigs[0], 0,
                                     True, attribs);
glXMakeCurrent(rcx->dpy, rcx->win, rcx->ctx);
```

一旦绑定了一个环境，我们就可以进行 GL 调用了。首先设置视口：

```
glViewport(0, 0, rcx->nWinWidth, rcx->nWinHeight);
```

接下来，清除颜色缓冲区，准备进行渲染。

```
glClearColor(0.0f, 1.0f, 1.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);
```

图 15.1 所示展示的这个简单的演示应用程序只绘制两个眼球，它们尽力跟随鼠标指针在屏幕中来回移动。为了确定将眼球放在什么位置、鼠标指针在哪里以及这些眼球看向哪里，还要进行一些计算。我们可以看一看 GLXBasics 示例程序的剩余部分，来看看它们是如何一起协作的。因为本章并没有引入新的 OpenGL 功能，所以这里只列出了重要的 GLX 知识点。

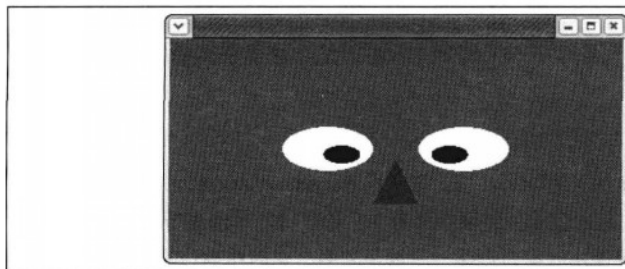


图 15.1

它们在看着你

现在已经完成了 OpenGL 设置，我们可以集中注意力进行渲染了。当接收到窗口改变或指针位置移动等用户输入时，窗口的内容将会被重绘。然后调用 `glXSwapBuffers`。

```
// 清除绘制命令
glXSwapBuffers(rcx->dpy, rcx->win);
```

在应用程序关闭之前，需要进行一些清除工作。在我们开始这个应用程序时，打开了一个到 X Server 的连接，创建了一个 X Window，并且创建和绑定了一个环境。现在，在我们退出之前，必须清除分配的所有资源。请注意，环境必须先解除绑定后才能销毁。

```
glXMakeCurrent(rcx->dpy, None, NULL);

glXDestroyContext(rcx->dpy, rcx->ctx);
rcx->ctx = NULL;

XDestroyWindow(rcx->dpy, rcx->win);
rcx->win = (Window)NULL;

XCLOSEDisplay(rcx->dpy);
rcx->dpy = 0;
```

## 15.4 小结

OpenGL 是 Linux 的一个重要组成部分，因为它是唯一一个得到广泛支持的硬件 3D API。即使我们已经知道 GLUT 如何可以应用在 Linux 中，对于定义缓冲区资源、窗口管理和其他 Linux 特定的 OpenGL 接口来说，直接使用 GLX 也是必要的。

即使 GLUT 能够用来处理 Linux 上的窗口管理，GLX 1.4 和相关扩展能够支持更好的控制，在创建新环境时让应用程序选择一个特定的 OpenGL 版本。与 WGL 和 AGL 接口类似，GLX 提供了与操作系统进行同步渲染的方法。我们学习了如何搜索符合我们要求的配置，还学习了如何创建一个支持特定 OpenGL 版本的环境。最后，我们还了解了如何在应用程序结束之后对 GLX 状态进行清理。



# 第 16 章 OpenGL ES：移动设备上的 OpenGL

作者：Nicholas Haemel、Richard S. Wright, Jr.

## 本章内容

任 务	使用的函数
选择配置	eglGetConfig/eglChooseConfig/eglGetConfigAttrib
创建 EGL 窗口	eglCreateWindowSurface
管理 EGL 环境	eglCreateContext/eglDestroyContext/eglMakeCurrent
将缓冲区传输到窗口并进行同步	eglSwapBuffers/eglSwapInterval/eglWaitGL

本章简单介绍一下 OpenGL ES 渲染。这些 API 用于在传统资源受到限制的嵌入式环境下使用。OpenGL ES 能够做到很多其他渲染 API 做不到的事情。

这个领域需要学习的东西很多，我们在这里将学习很多关于它的基础知识。OpenGL ES 存在几种不同版本，但我们将重点放在最新的也是最常用的 OpenGL ES 2.0 上。我们还介绍设计用于 OpenGL ES 的窗口界面，并接触一些处理嵌入式环境的特定问题。最后，但并不是不重要的，我们学习如何使用 OpenGL ES 2.0 为 iPhone 和 iPad 开发应用程序。

## 16.1 精简的 OpenGL

我们会发现 OpenGL ES 与常规的 OpenGL 非常相似，这并不是偶然的，因为 OpenGL ES 的规范正是基于不同的 OpenGL 版本开发的。正如我们已经看到的，OpenGL 为 3D 渲染提供了强大的渲染接口。它非常灵活，能够用于从游戏、完善的 CAD 工作站到医学成像范围的各种应用程序。

## 16.1.1 ES 指什么

随着时间的流逝, OpenGL API 被不断地扩展和补充着, 以便支持新的特性。这也导致了老版本 OpenGL 应用程序界面的膨胀, 它们提供很多不同的途径做同一件事情。我们以绘制一个点为例。在老版本的 OpenGL 中, 可以通过使用即时模式(immediate mode)进行, 这种方法使用的 `glBegin/glEnd` 函数带有在捕获并重新执行立即模式命令的显示列表之间或之中定义的顶点信息。我们还可以使用 `glDrawArrays` 方法, 这些点是在数组中或通过顶点缓冲区对象预先指定的。

绘制一个点的简单动作可以用 4 种不同的方法来实现, 每种方法都有其自身的优点。虽然能有很多可选的方法来执行应用程序是很不错的, 不过所有这些灵活性也使 API 变得更庞大。这样最终又会导致需要一个非常庞大复杂的驱动程序来支持。另外, 一些特定硬件还常常要求每条通道都是快速高效的。在 OpenGL ES 2.0 规范编写时, 目前的 OpenGL API 确实太大了。OpenGL ES 2.0 只包含 OpenGL 2.1 中最常用和最有用的部分。OpenGL 的新版本大幅度减少了功能上的重复, 但是这些修订包含了大多数 OpenGL ES 硬件无法实现的特性和功能。OpenGL ES 2.0 在灵活性和可用性之间为嵌入式环境提供了良好的平衡。

## 16.1.2 历史概述

随着硬件成本的下降, 以及更多功能能够集成到面积更小的半导体器件中, 嵌入式设备的用户界面也变得越来越复杂。一个常见的例子就是汽车。20 世纪 80 年代, 第一条车载计算机的可视化反馈信息是以单条多行文本的形式出现的。这些界面提供关于安全带使用、当前汽油和里程使用情况等警告信息。在这之后, 二维图像显示开始流行, 通常使用类似位图渲染的方式来显示 2D 图像。最近, 支持 3D 的系统也被整合到汽车上, 以帮助支持 GPS 导航和其他显示密集型特性。航空设备和手机也有着相似的技术发展史。

早期的嵌入式 3D 界面通常是特定硬件所特有, 并且严重依赖特定硬件已有特性的。这种情况非常普遍, 因为支持的特性集很小, 并且在各种设备之间区别很大。但是随着 3D 引擎的复杂度不断增加, 在不同设备和供应商之间对应用程序进行移植就变的更加花费时间, 也变的更加有挑战性了。在这种情况下, 唯一的解决办法就是建立标准接口了。在这种思想下, 一个团体成立了, 它的目的是定义一种接口, 这种接口是灵活的和可移植的, 充分考虑嵌入式环境的种种限制而量身定做的。这个团体就是 Khronos 小组。

### Khronos 小组

Khronos 小组最初是 2000 年由 OpenGL ARB——OpenGL 主管团体的成员建立的。PC 领域有很多优秀的图形 API, 但是 Khronos 的目标是定义更适合个人计算机以外设备的接口。他们开发的第一个嵌入式 API 就是 OpenGL ES。

Khronos 由许多硬件和软件领域的行业领导者组成, 目前的成员包括 AMD、德州仪器、ARM、英特

尔、NVIDIA、诺基亚和高通公司等。完整的名单很长，而且都是赫赫有名的公司。要得到更多信息，可以访问 Khronos 网站（[www.khronos.org](http://www.khronos.org)）。

## 版本发展

OpenGL ES 的第一版发行时命名为 ES 1.0，它尝试对完整的 PC API 进行大幅度删减。这个版本以 OpenGL 1.3 规范为基础。虽然非常优秀，但 OpenGL ES 1.0 删去了完整的 OpenGL 规范中很多不常用或非常复杂的部分。和 OpenGL 1.3 一样，OpenGL ES 1.0 为顶点变换和片段处理定义了一个固定功能的管线。

OpenGL ES SC 1.0 是一个基于 OpenGL ES 1.0 的独立规范，它是为了可靠性要求极高的执行环境而设计的。这些应用程序被认为是“安全关键”（Safety Critical）的，这就是 SC 标识符的由来。典型的应用是在航空电子设备、汽车和军用环境中。在这些领域中 3D 应用软件常常用于仪器使用、绘图和地形再现。

在第一个版本发布之后不久，ES 1.1 就完成了。虽然与第一个规范非常相似，但 1.1 规范是以 OpenGL 1.5 规范为基础的。另外，这个版本还加入了纹理路径、缓冲区对象和绘图纹理接口。总而言之，ES 1.1 与 ES 1.0 非常相似，但加入了一些新的有趣的特性。

ES 2.0 的出现完全是突破性的。它并不向下兼容 ES 1.x 版本。它的最大不同是删去了管线的固定功能部分，取而代之的是使用可编程着色器来完成顶点和片段处理工作。ES 2.0 规范是以 OpenGL 2.0 规范为基础的。

为了全面支持可编程着色器，ES 2.0 采用了 OpenGL ES 着色语言。这是一种高级着色语言，它与 OpenGL 2.0 以上版本相配合的 OpenGL 着色语言非常相似。之所以说 ES 2.0 是一种巨大的进步，是因为在这个版本中所有固定功能都已经不再是 API 的障碍。这意味着应用软件在它们自己的着色器中可以只执行和使用它们需要的特性。

作为总结，目前定义的 OpenGL ES 版本和相关的 OpenGL 版本如表 16.1 所示。

表 16.1 ES 和相关 OpenGL 版本

OpenGL ES	OpenGL
ES 1.0	GL 1.3
ES 1.1	GL 1.5
ES 2.0	GL 2.0
ES SC 1.0	GL 1.3

## 16.2 版本选择

硬件经常会针对特定的 API 开发，这些平台通常只会支持单一版本的 ES。把不同版本的 ES 看作展现下层硬件功能的“肖像”有时会很有帮助。

对于传统的 GL 来说，典型情况下新硬件将会设计为支持可用的最新版本。

这种情况在 ES 中有所不同。新硬件的目标特征类型会基于几个因素来考虑，例如目标产品成本、典型应用和系统支持等。可以说，半导体技术在过去的 5 年中发生了长足的进步，现在制造小型、低成本和高效的芯片已经成为可能。很多普遍使用的智能电话，例如苹果的 iPhone，都在使用 OpenGL ES。本章重点放在 OpenGL ES 2.0 上，而不是向读者介绍那些陈旧过时的老版本 ES。

为了更加有效地学习本章内容，我们应该熟悉大多数 OpenGL 特性集。本章主要展示常规 OpenGL 和 OpenGL ES 的主要区别，而很少对每种特性再次进行详细的描述。

## 16.2.1 ES 2.0

OpenGL ES 2.0 和 OpenGL 3.3 非常相似。它们的界面都很简洁，删除了过去不规范的部分。但是，OpenGL 3.3 加入了很多目前在嵌入式系统中无法支持的新特性。变换反馈、多重采样、几何着色器、浮点缓冲区以及其他许多新添加到 OpenGL 中的功能，在 ES 2.0 发布时甚至还没有出现。实际上，ES 2.0 在刚刚定义时一个很大的改进是在命令中支持了浮点数据类型。而在以前浮点数据则需要使用定点类型进行模拟。纹理可能会占用很大空间！

### 顶点处理和着色

在顶点规范中，必须使用顶点缓冲区对象或者客户端顶点数组。顶点缓冲区对象能够进行映射，就像 OpenGL 3.3 中所允许的一样。我们可以使用 `glVertexAttribPointer` 指定顶点属性。

```
glVertexAttribPointer(GLuint index, GLuint size, GLenum type,
                     GLboolean normalized, GLsizei stride, const void *ptr);
```

要绘制几何图形，我们可以使用 `glDrawArrays` 和 `glDrawElements`。但是，OpenGL 3.3 中更多的专用命令，例如 `glMultiDrawElements`、`glDrawRangeElements` 等，在 OpenGL ES 2.0 中并不支持。

### 着色器

OpenGL ES 2.0 使用可编程着色器，方式与 OpenGL 3.3 大致相同。但是，支持的着色器阶段只有顶点处理和片段处理两种。OpenGL ES 2.0 使用一种与 GLSL 语言规范相类似的着色语言，叫做 OpenGL ES 着色语言（OpenGL ES Shading Language）。这个版本针对具体嵌入式环境以及它们所包含的硬件进行了改进。

虽然内建的编译器对于应用程序来说非常方便，但是在 OpenGL 驱动程序中包含编译器可能会使它变得很大（可以达到几兆字节），而且编译过程也可能非常耗费 CPU 资源。但这些要求在更小的掌上嵌入式系统中很难满足，因为这些系统对内存和处理功率的限定都非常严格。

OpenGL ES 保留了在运行时编译着色器的机制，同时也加入了离线编译着色器，然后在运行时载入编译结果的能力。这两种方法单独来讲都不是必须的，但一个 OpenGL ES 2.0 实现必须至少支持其中一个。

许多原始的 OpenGL 2.0 着色器和程序函数也作为 ES 的一部分保留了下来。程序和着色器管理仍使用同样的语义。使用可编程管道的第一步是创建必要的着色器和程序对象。这些工作由下面的命令来完成。

```
GLuint glCreateShader(GLenum type);
GLuint glCreateProgram(void);
```

在这之后, 着色器对象能够被绑定到程序对象。

```
glAttachShader(GLuint program, GLuint shader);
```

如果我们的实现能够支持 OES\_shader\_source, 那么可以直接传递着色器字符串, 然后在运行时使用我们在 OpenGL 3.3 中已经熟悉的函数对它们进行编译。

```
glShaderSource(GLuint shader, GLsizei count, const char **string,
               const int *length);
glCompileShader(GLuint shader);
```

类似地, 如果实现支持 OES\_shader\_source, 那么着色器源文件就可以使用实现特定的方法进行离线编译了。关于这方面的更多信息, 请参考设备的 SDK。然后, 我们可以只向 OpenGL ES 提供从离线编译中获得的着色器二进制数, 而不是在运行时传递源代码。如果一对片段和顶点是一起进行离线编译的, 那么就可以为它们加载一个二进制数。

```
glShaderBinaryOES(GLint n, GLuint *shaders, GLenum binaryformat,
                  const void *binary, GLint length);
```

我们必须支持这两种方法中的一种, 或者两者都要支持。查看设备文档, 确定哪种选择更适合我们的嵌入式设备。当着色器完成了加载和编译, 就要将属性通道绑定到着色器中使用的属性名上了。

```
glBindAttribLocation(GLuint program, GLuint index, const char *name);
```

然后程序就可以进行连接了。如果支持这个着色器二进制数接口, 那么编译的着色器的着色器二进制数就需要在调用连接方法之前进行加载。

```
glLinkProgram(GLuint program);
```

在程序成功连接之后, 可以通过调用 glUseProgram 将它设为当前执行程序。如果需要的话, 这时也可以设置统一值。大多数标准的 OpenGL 3.3 属性和统一接口都得到了支持。不过, 设置统一矩阵的变换位必须为 GL\_FALSE。这个属性对于可编程管道的功能来说并不是必要的。试图在没有有效的程序绑定的情况下进行绘制, 会产生不确定的结果。统一块也并不是 OpenGL ES 2.0 的一部分, 所以我们需要使用独立的统一值。

```
glUseProgram(GLuint program);
glUniform{1234}{i|f}(GLint location, T values);
glUniform{1234}{i|f}v(GLint location, GLsizei count, T value);
glUniformMatrix{234}fv(GLint location, GLsizei count,
                       GLboolean transpose, T value);
```

与 OpenGL ES 2.0 相匹配的着色器语言与 OpenGL 3.3 中的着色器语言 (GLSL 1.50) 非常相似。实际上我们在创建 ES 着色器时, 经常可以先在 PC 或 Mac 上开发, 然后在达到要求之后将它们移植到 ES 上。

## 光栅化

在 OpenGL ES 2.0 中关于点的处理也略有不同, 只有锯齿点 (aliased point) 得到了支持。顶点着色器负责输出点大小; 在这种 API 中没有其他方式可以用来指定点大小。GL\_COORD\_REPLACE 可以用来为 s 和 t 坐标生成从 0 到 1 的点纹理坐标。点坐标原点也会被设置为 GL\_UPPER\_LEFT, 并且不能改变, 而点参数也是不可用的。

抗锯齿线也没有得到支持。OpenGL ES 2.0 也并不支持多边形平滑、多边形抗锯齿或多重多边形模式。

## 纹理贴图

ES 2.0 支持 2D 纹理和立方体贴图 (cubemap)。深度纹理、矩形纹理和数组纹理并没有得到支持, 而 3D 纹理则是可选项。非 2 的整数次幂的纹理 (nonpower-of-two texture) 在没有使用 Mip 贴图, 并且纹理环绕模式 (texture wrap mode) 为边缘截取 (clamp to edge) 的情况下只对 2D 纹理有效。纹理并不是必须为 2 的整数幂。OpenGL ES 2.0 也没有采样对象。

## 帧缓冲区

和完整版本的 OpenGL 3.3 类似, OpenGL ES 2.0 也支持帧缓冲区和渲染缓冲区对象。应用程序可以创建和绑定自己的帧缓冲区对象。在这里存在一些限制, 同一时间只能绑定一个颜色缓冲区, 但是我们仍然可以使用深度缓冲区的模板缓冲区。我们还可以将纹理绑定到帧缓冲区绑定。点。

## 片段操作

ES 2.0 中的逐片段操作也有了一些变化。必须有至少一个可用配置同时支持深度缓冲区和模板缓冲区, 这样能够保证依靠使用深度信息和模版比较的应用程序在任何支持 OpenGL ES 2.0 的实现中都能够运行。

OpenGL 2.0 规范中还删除了一些与 OpenGL 3.3 规范相关的东西。首先, 透明度测试步骤被删除, 因为应用程序能够在片段着色器中执行这一步骤。glLogicOp 接口也不再支持, 而遮挡查询 (occlusion query) 也不再包括在 OpenGL ES 中。

混合 (blending) 仍像 OpenGL 2.0 中一样可以使用, 但范围更加有限。我们不能为每个渲染目标设置不同的混合, 并且不支持双源混合 (dual source blending)。

## 状态

OpenGL ES 2.0 的状态查询方式与 OpenGL 3.3 相同。我们可以使用 glGetBooleanv、glGetIntegerv 和 glGetFloatv 查询大多数状态。还有其他一些查询也可以使用, 例如 glGetBufferParameteriv 和 glIsTexture。

## 核心增强

OpenGL ES 对扩展的支持与 OpenGL 3.3 类似。虽然这些扩展并不是必需的, 并且可能不会在所有的实现上都支持, 但是在能够支持的平台上, 它们可能会非常有用。

- **半浮点顶点格式 (Half-Float Vertex Format)**——OES\_vertex\_half\_float 这个可选扩展使得使用 16 位浮点值指定顶点数据成为可能。如果实现了这一点, 那么相对于更大的数据类型来说, 顶点数据的存储空间需求将得到大幅度地降低, 并且更小的数据类型将对管道的顶点转换部分的效率起到正面的影响。使用半浮点数据 (在颜色上就经常使用) 没有任何不利影响, 尤其是在显示色彩深度有限的情况下。
- **浮点纹理**——OES\_texture\_half\_float 和 OES\_texture\_float 是两种新的可选扩展, 它们用浮点分量来定义新的纹理格式。OES\_texture\_float 使用 32 位浮点格式, 而 OES\_texture\_half\_float 使用 16 位浮点格式。这两种扩展都支持 GL\_NEAREST 放大滤镜, 也支持 GL\_NEAREST 和 GL\_NEAREST\_MIPMAP\_NEAREST 缩小滤镜。要使用 OpenGL ES 中定义的其他缩小和放大滤镜, 则必须支持 OES\_texture\_half\_float\_linear 和 OES\_texture\_float\_linear 扩展。
- **无符号整数元素索引**——OES\_element\_index\_uint OpenGL ES 中元素索引的使用受到索引数据类型最大长度的固有限制。无符号字节和无符号短整型分别只允许使用 256 或 65 536 个元素。这种可选扩展允许使用无符号整数元素索引, 将最大引用索引数进行扩展并超出了目前硬件的存储能力。
- **映射缓冲区 (Mapping Buffer)**——OES\_mapbuffer 至于以前 OpenGL ES 版本中的顶点缓冲区对象支持, 定义和使用除静态缓冲区以外其他缓冲区的能力被删除了。当这个可选扩展可用时, GL\_STREAM\_DRAW、GL\_STREAM\_COPY、GL\_STREAM\_READ、GL\_STATIC\_READ、GL\_DYNAMIC\_COPY 和 GL\_DYNAMIC\_READ 等标志的使用是合法的, 就像 glMapBuffer 和 glUnmapBuffer 入口点一样。这就允许应用程序映射和编辑已经存在的 VBO 了。
- **3D 纹理**——OES\_texture\_3D 通常, 大多数 ES 应用程序并不要求支持 3D 纹理。这个扩展作为一个可选项保留了下来, 允许实现来决定是否要加速支持, 这在个别情况下非常有用。纹理环绕模式和 Mip 贴图在维度为 2 的整数幂 (power-of-two dimension) 的 3D 纹理中也得到支持。非 2 的整数幂的纹理只在 Mip 贴图和纹理环绕中支持 GL\_CLAMP\_TO\_EDGE。
- **片段着色器中的高精度浮点和整型**——OES\_fragment\_precision\_high 这个可选扩展允许支持片段着色器中定义的整型和浮点数的高精度限定符。
- **爱立信压缩纹理格式**——OES\_compressed\_ETC1\_RGB8\_texture 人们认识到在 OpenGL ES 中需要压缩纹理支持已经有很长时间了, 但格式规范和实现工作被留给了各个独立的实现者 (implementer)。这个可选扩展指定这些格式中的一个来供多种平台使用。

要载入使用 ETC\_RGB8 格式的压缩纹理, 可以调用内部格式为 GL\_ETC1\_RGB8\_OES 的 glCompressedTexImage2D。

这个格式定义了一种方案，将每个  $4 \times 4$  纹理单元 (texel) 块分成一组，从而产生一个基本色，并从一个表中选择每个纹理单元的修改器。然后修改器被加到基本色上，截取到 0~255 范围来决定最后的纹理单元颜色。OES\_compressed\_ETC1\_RGB8\_texture 的完整描述中说明了关于这个处理过程的更多细节。

## 16.3 ES 环境

到目前为止，我们已经了解了这些规范能够允许应用程序做什么，差不多可以来了解一些例子了。图 16.1 所示展示了一个在手机上运行 OpenGL ES 的例子。彩图 22 中也展示了这个图像。但是在这之前，在进行 OpenGL ES 和目标嵌入式环境的相关工作时，我们还要牢记一些关于嵌入式系统的独特问题。

图 16.1

手机上的 OpenGL ES 渲染



### 16.3.1 应用程序设计的注意事项

对于嵌入式领域的新手来说，要注意这里的情况与在 PC 上工作时有些不同。ES 领域涵盖了多种硬件，其中最强大的可能就是广泛用途图形资源 (extensive dedicated graphics resource) 的多核系统了，例如 Sony PlayStation 3。或者，更经常的情况是，我们可能正在为一个拥有 50MHz 处理器和 16MB 存储器的入门级手机进行开发或移植工作。

在性能有限的系统中，必须特别注意指令计数，因为如果我们想要维持一个可接受的性能表现，每一个周期都要精打细算。有些操作可能会非常慢，寻找一个角的正弦值就是一个例子。如果近似值可以接受的话，在一个预先计算好的表中查找将会比调用数学库中的  $\sin()$  函数快得多。一般来说，在 PC 应用程序中可能的计算和运算的类型都要经过更新才能用在嵌入式系统中。

其中一个例子就是物理计算，这经常需要高昂的系统开销。在手机这样的嵌入式系统中，这些运算通常都经过了简化和近似。

在比较老的系统中，了解本地浮点支持也很重要。这些系统中的大多数都没有能力直接进行浮点操作。这就意味着，所有浮点操作都将在软件中模拟。这些操作通常都非常缓慢，无论如何都应该避免。

## 16.3.2 有限环境的处理

在进行嵌入式系统相关工作时, 不仅会受到环境限制, 而且图形处理能力本身也不能与最先进的 PC 图形处理相提并论。这些限制迫使我们在寻求对应用程序的性能进行优化, 或者甚至只是对它进行载入和运行时, 都要对资源情况特别注意。为存储空间建立预算可能会很有帮助。通过这种方式, 我们可以将最大可用图形/系统内存分成小块, 供每个内存密集型的任务使用。

这样将有助于建立应用程序的每个独立部分能够使用多少数据, 以及运行何时会变缓慢的观点。

纹理贴图是最明显的领域之一。拥有大量细节的纹理能够帮助以 PC 为目标的应用程序来创建丰富细致的环境。这对于用户体验来说非常棒, 但在大多数嵌入式系统中这些纹理会“吃掉”大量资源。很多较老的平台可能不能为纹理贴图提供全面的硬件支持。在对许多片段进行纹理贴图时, 特别是如果每片重叠的几何图形以错误的顺序进行纹理贴图和绘制时, 这些问题可能引起大幅度的性能下降。

除了核心硬件纹理贴图性能之外, 贴图尺寸也会成为主要的限制。

3D 和立方体贴图纹理都会迅速使用大量内存, 这也是为什么在 ES 2.0 中 3D 纹理只是可选项的原因。通常在图形和系统内存有限的情况下, 屏幕尺寸也会比较小。这就意味着在相似的视觉效果下能够使用相对小得多的纹理。避免多纹理贴图也是非常必要的, 因为这需要更多的纹理内存, 还需要多重纹理传递。

和纹理类似, 顶点存储也会对内存产生冲击。除了为顶点所能使用的内存总量设置一个上限之外, 确定场景的哪些部分重要并在此基础上分配顶点也会很有帮助。

一个在屏幕上有很多对象时保持渲染平滑的技巧是, 根据相对于观察者的距离来改变对象的顶点数。这是几何图形管理中的一种层次细节方法。例如, 如果我们要生成一个森林场景, 可能会用到 3 种不同的树木模型。其中一种只有非常少的顶点数, 用来渲染最远的树木; 中等距离的树木将使用中等数目的顶点; 而最近的树木将使用数量更多的顶点。相对于所有的树木都以高层次细节呈现的方式来说, 用这种方法能够以快得多的速度来对大量树木进行渲染。由于细节最少的树木距离我们最远并且被部分遮蔽, 所以很难发现这些树木缺少细节, 但结果却在顶点处理过程中极大地节省了资源。

## 16.3.3 定点数学

我们可能会问自己: “什么是定点数学, 我为什么要关心它?” 实际上, 如果硬件支持浮点数, 并且使用的 OpenGL ES 版本也支持的话, 就不需要关心这个问题。但是, 许多平台并不是本地支持浮点数的。在 CPU 中模拟的浮点运算非常缓慢, 我们应该避免这种情况。在这些情况下, 一种浮点数的表示法能够用来传达不完整的数字。我肯定不会把这些内容变成一堂数学课! 但是, 我们会介绍一些关于定点数学的基本知识, 以使读者对此有一定的了解。如果需要更多的了解的话, 也有很多资源能够足够深入地解释定点数学。

首先, 我们来回顾一下定点数字是如何工作的。一个浮点数字基本上可以分为两个组成部分, 即表示小数值的尾数部分和表示进制或幂的指数部分。这样, 庞大的数字就被表示成了很小的有效数字形式。它们以  $m \times 2^e$  的形式表示, 其中  $m$  是尾数部分, 而  $e$  则是指数部分。

定点表示法则不同,它看起来更像是普通的整数。这些位(bit)被分成两个部分,即整数部分和小数部分。整数部分和小数部分中间的位置是“虚点”。这里也可能会有一个符号位(sign bit)。综上所述,一个定点格式的数 s15.16 表示这里有 1 个符号位,15 个位表示整数,而 16 个位表示小数。这是 OpenGL ES 本地使用的用来表示浮点数的格式。

两个定点数的加法很简单。由于一个定点数基本上就是带有一个“点”的整数,这两个数能够通过普通标量的加法操作来实现相加。减法也是同样道理。进行这些操作还有一个要求,即这两个定点数的格式必须相同。如果它们的格式不同,那么必须首先将其中一个数转换成与另一个数相同的格式。所以,要将格式为 s23.8 和 s15.16 的两个数相加或相减,必须选择一个格式,并且两个数都要转换成这个格式。

乘法和除法要稍微复杂一些。当两个定点数字相乘的时候,结果的“虚点”将为这两个操作数中“虚点”相加的和。例如,如果我们要将格式为 s23.8 的两个数相乘,结果的形式将为 s15.16。因此,首先将操作数转换成适合于适当精度结果的格式常常会很有帮助。我们可能不会想要将两个大于 1.0 的 s15.16 格式的数相乘——因为结果格式中将没有整数部分!除法也非常类似,只是第一个数的小数部分大小将减去第二个数的小数部分的大小。在使用定点数字时,我们要特别注意溢出问题。

使用普通的浮点数时,当小数部分要溢出时,指数部分将进行调整,以保持精度并防止溢出。在定点运算中则不是这样。为了在执行操作时避免定点数字溢出(因为这样可能会导致问题),可以对格式进行修改。这些数字可以被转换成有更大整数部分的格式,并在 OpenGL ES 调用它之前转换回去。在乘法运算中,在结果转换回操作数之一的格式时,类似的问题会导致小数部分的精度损失。也有一些数学包可以帮助我们进行转换或转换成定点格式,就像执行数学函数一样。如果需要在整个应用程序中使用定点数学,这可能是它最简单的处理方法了。

好了!现在我们已经了解了如何使用定点格式进行基本数学操作了。如果在进行嵌入式系统相关工作时遇到使用定点值方面的麻烦,那么上面的内容能够帮助我们理出头绪。有很多关于更深入学习定点数学的优秀参考资料。其中一个《Essential Mathematics for Games and Interactive Applications》,作者是 James Van Verth 和 Lars Bishop (Elsevier, Inc. 2004)。

## 16.4 EGL: 新的窗口环境

我们已经听说过 GLX、AGL 和 WGL 了,它们是针对 Linux、Apple 的 Mac OS 和 Microsoft Windows 等操作系统的 OpenGL 相关系统接口。这些接口对设置和管理 OpenGL 将要使用的系统侧资源是非常必要的。EGL 实现也经常由图形硬件提供商提供。

和其他窗口接口不同,EGL 并不是操作系统特定的。这种接口是为了在 Windows、Linux 下,或者在 Brew 和 Symbian 之类的嵌入式操作系统下运行而设计的。图 16.2 所示是 EGL 和 OpenGL ES 如何应用于一个嵌入式系统的方块图。

像 OpenGL 一样,EGL 也有自己的本地类型(native type)。EGLBoolean 有两个值,和它们在 OpenGL 中的对应值命名十分相似:EGL\_TRUE 和 EGL\_FALSE。EGL 也定义了 EGLint 类型。这是一个整型,和本地平台的整形长度相同。在本书编写时,EGL 最新的版本是 EGL 1.4。

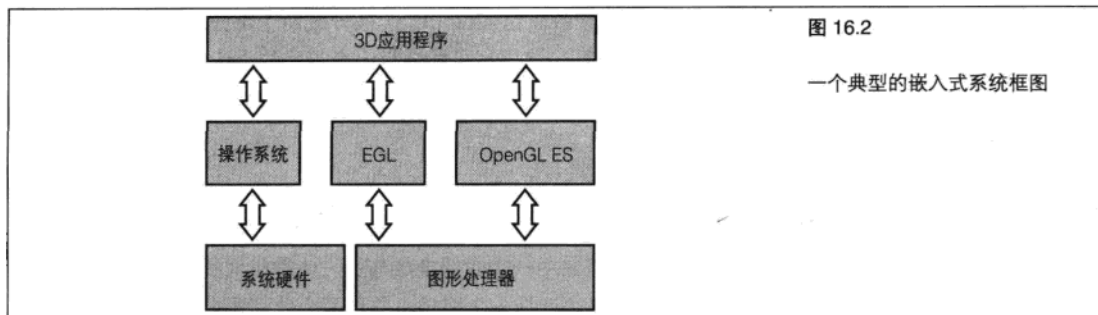


图 16.2

一个典型的嵌入式系统框图

### 16.4.1 EGL 显示

大多数 EGL 入口点都有一个叫做的 `EGLDisplay` 参数，这是对能够进行绘图的渲染目标的一个引用。最简单的理解方式是把它想象成与一个物理监视器进行通讯。设置 EGL 的第一步是获取默认显示器，这项工作由下面的函数来完成。

```
EGLDisplay eglGetDisplay(NativeDisplayType display_id);
```

作为参数的本地显示器 ID 是依赖于系统的。例如，如果我们正在使用 Windows 上的 EGL 实现，那么传递的 `display_id` 参数将会是设备环境。

如果没有显示器 ID 并只希望在默认设备上渲染，也可以传递 `EGL_DEFAULT_DISPLAY`。如果返回 `EGL_NO_DISPLAY`，则说明出现了一个错误。

现在我们有了一个显示句柄，可以用它来初始化 EGL。如果我们在没有事先初始化 EGL 的情况下试图使用其他 EGL 接口，就会出现 `EGL_NOT_INITIALIZED` 错误。

```
EGLBoolean eglInitialize(EGLDisplay dpy, EGLint *major, EGLint *minor);
```

返回的其他两个参数是 EGL 的主版本号和次版本号。通过调用初始化命令，可以通知 EGL 我们已经准备好进行渲染，这将会允许它分配和设置任何必要的资源。

EGL 还引入了一个名为 `eglBindAPI` 的接口。它允许应用程序在不同的渲染 API（例如 OpenGL ES 和 OpenVG）中进行选择。每个线程对每种 API 只能有一个当前环境。使用这个接口通知 EGL 下一次在一个线程中调用 `eglMakeCurrent` 时应该使用哪个接口。我们要传递下面 3 个标志中的一个来表示正确的 API：`EGL_OPENGL_ES_API`、`EGL_OPENVG_API` 或 `EGL_OPENGL_API`。如果我们传递了一个无效的枚举（enum）类型，那么这个调用将会失败。

```
EGLBoolean eglBindAPI(EGLenum api);
```

EGL 还提供了一种查询当前 API 的方法，即 `eglQueryAPI`。这个接口返回前面列出的 3 个 `EGLenum` 类型中的一个：`EGL_OPENGL_ES_API`、`EGL_OPENVG_API` 或 `EGL_OPENGL_API`。

```
EGLenum eglQueryAPI(void);
```

要退出应用程序，或者在渲染完成之后，必须再次调用 EGL 来清除所有分配的资源。在这次调用之

后, 对当前显示相关的 EGL 资源的引用将变为无效的, 除非再次调用 `eglInitialize` 来引用它。

```
EGLBoolean eglTerminate(EGLDisplay dpy);
```

在退出和完成了一个线程的渲染时, 还要调用 `eglReleaseThread`。这样就使 EGL 能够释放它分配给这个线程的任何资源了。如果一个环境仍然是绑定的, `eglReleaseThread` 也会对它进行释放。在调用 `eglReleaseThread` 之后再进行 EGL 调用仍然是合法的, 但是会导致 EGL 重新分配刚刚释放的任何状态。

```
EGLBoolean eglReleaseThread(EGLDisplay dpy);
```

## 16.4.2 创建窗口

在大多数平台上, 创建一个窗口来进行渲染可能是一项复杂的工作。窗口是在本地操作系统中建立的。稍后我们将了解如何通知 EGL 关于本地窗口的信息。幸运的是, 这个过程和 Windows、Linux 中的对应过程非常相似。

### 显示配置

EGL 配置与 Windows 上的像素格式或 Linux 上的视觉 (visual) 类似。每种配置为一组渲染表面呈现一组特性或属性。这样, 渲染表面在显示器上将是一个窗口。这在支持多配置的实现中是非常典型的情况。每种配置有一个唯一的数字来标识。定义不同的常量与一个配置的属性进行关联。表 16.2 列出了这些定义。

表 16.2

EGL 配置属性列表

类 型	描 述
EGL_BUFFER_SIZE	缓冲区位的总深度, 以位为单位
EGL_RED_SIZE	颜色缓冲区的红色通道的位数
EGL_GREEN_SIZE	颜色缓冲区的绿色通道的位数
EGL_BLUE_SIZE	颜色缓冲区的蓝色通道的位数
EGL_ALPHA_SIZE	颜色缓冲区的 alpha 通道的位数
EGL_DEPTH_SIZE	深度缓冲区的位数
EGL_LUMINANCE_SIZE	颜色缓冲区中亮度的位数
EGL_STENCIL_SIZE	模板缓冲区的位数
EGL_BIND_TO_TEXTURE_RGB	如果配置能够绑定到 RGB 纹理上则为真
EGL_BIND_TO_TEXTURE_RGBA	如果配置能够绑定到 RGBA 纹理上则为真
GLX_CONFIG_CAVEAT	设置为下面几个值之一: <code>EGL_NONE</code> 、 <code>EGL_SLOW_CONFIG</code> 或 <code>EGL_NON_CONFORMANT_CONFIG</code> 。它们能够对于这个配置的潜在的问题进行警告。缓慢配置可能是软件模拟的, 因为它超过了硬件限制。非一致性的配置将无法通过一致性测试
EGL_CONFIG_ID	这个配置的唯一标识符
EGL_LEVEL	帧缓冲区层次

续表

类 型	描 述
EGL_NATIVE_RENDERABLE	如果本地 API 可以渲染到这个表面, 那么它设置为 EGL_TRUE
EGL_NATIVE_VISUAL_ID	如果配置支持窗口, 那么它能够显示本地画面的 ID, 否则值为 0
EGL_NATIVE_VISUAL_TYPE	如果配置支持窗口渲染, 那么它表示本地视觉类型
EGL_RENDERABLE_TYPE	画面的本地类型, 可能是 EGL_OPENGL_ES_BIT 或 EGL_OPENVG_BIT
EGL_SURFACE_TYPE	支持的有效表面目标。可以是 EGL_WINDOW_BIT、EGL_PIXMAP_BIT 或 EGL_PBUFFER_BIT
EGL_COLOR_BUFFER_TYPE	颜色缓冲区的类型。可以是 EGL_RGB_BUFFER 或 EGL_LUMINANCE_BUFFER
EGL_MIN_SWAP_INTERVAL	eglSwapInterval 所能接受的最小值。更小的值将由这个最小值代替
EGL_MAX_SWAP_INTERVAL	eglSwapInterval 所能接受的最大值。更大的值将由这个最大值代替
EGL_SAMPLE_BUFFERS	支持的多重采样缓冲区数量, 必须是 0 或者 1
EGL_SAMPLES	多重采样缓冲区中每个像素的样本数量。如果 GLX_SAMPLE_BUFFERS 为 0, 那么它也必须是 0
EGL_ALPHA_MASK_SIZE	透明度遮罩的位数
EGL_TRANSPARENT_TYPE	表示对透明的支持。它的值可以是 EGL_NONE 或 EGL_TRANSPARENT_RGB。如果支持透明, 当一个像素的成分都等于各自的透明 RGB 值时, 就会绘制一个透明像素
EGL_TRANSPARENT_RED_VALUE	一个帧缓冲区像素为透明时必须是这个红色值
EGL_TRANSPARENT_GREEN_VALUE	一个帧缓冲区像素为透明时必须是这个绿色值
EGL_TRANSPARENT_BLUE_VALUE	一个帧缓冲区像素为透明时必须是这个蓝色值

在创建一个渲染表面之前, 选择一个配置是十分必要的。但是属性有各种各样的组合, 处理时看起来会很困难。EGL 提供了几个工具, 来帮助我们确定哪种配置能够最好地支持我们的需要。如果我们有关于窗口需要哪种选项的想法, 可以使用 `eglChooseConfig` 接口让 EGL 选择最适合我们需要的配置。

```
EGLBoolean eglChooseConfig(EGLDisplay dpy, const EGLint *attrib_list,  
                           EGLConfig *configs, EGLint config_size,  
                           EGLint *num_configs);
```

首先确定想要浏览多少匹配配置, 然后为返回的配置句柄分配内存。匹配的配置句柄将通过配置指针返回, 而配置数则通过 `num_config` 指针返回。

接下来是需要技巧的部分了。我们要确定功能配置中哪些参数对我们来说是重要的。然后, 我们创建一个列表, 列出每个属性, 属性后面则是相应的值。对于简单的应用程序来说, 颜色和深度缓冲区的位深度, 以及表面类型可能是重要的属性。这个列表必须以 `EGL_NONE` 来结束。下面是一个属性列表的例子。

```
EGLint attributes[] = {EGL_BUFFER_SIZE, 24,  
                       EGL_RED_SIZE, 6,  
                       EGL_GREEN_SIZE, 6,  
                       EGL_BLUE_SIZE, 6,  
                       EGL_DEPTH_SIZE, 12,  
                       EGL_SURFACE_TYPE, EGL_WINDOW_BIT,  
                       EGL_NONE};
```

对于这个序列中没有列入的属性，则将会使用默认值。在寻找匹配配置的过程中，我们列出属性中的一部分会要求精确的匹配，而其他属性却不需要。表 16.3 列出了每种属性的默认值和比较方法。

表 16.3 EGL 配置属性列表

属 性	比较操作符	默 认 值
EGL_BUFFER_SIZE	Minimum	0
EGL_RED_SIZE	Minimum	0
EGL_GREEN_SIZE	Minimum	0
EGL_BLUE_SIZE	Minimum	0
EGL_ALPHA_SIZE	Minimum	0
EGL_DEPTH_SIZE	Minimum	0
EGL_LUMINANCE_SIZE	Minimum	0
EGL_STENCIL_SIZE	Minimum	0
EGL_BIND_TO_TEXTURE_RGB	Equal	EGL_DONT_CARE
EGL_BIND_TO_TEXTURE_RGBA	Equal	EGL_DONT_CARE
EGL_CONFIG_CAVEAT	Equal	EGL_DONT_CARE
EGL_CONFIG_ID	Equal	EGL_DONT_CARE
EGL_LEVEL	Equal	0
EGL_NATIVE_RENDERABLE	Equal	EGL_DONT_CARE
EGL_NATIVE_VISUAL_TYPE	Equal	EGL_DONT_CARE
EGL_RENDERABLE_TYPE	Mask	EGL_OPENGL_ES_BIT
EGL_SURFACE_TYPE	Equal	EGL_WINDOW_BIT
EGL_COLOR_BUFFER_TYPE	Equal	EGL_RGB_BUFFER
EGL_MIN_SWAP_INTERVAL	Equal	EGL_DONT_CARE
EGL_MAX_SWAP_INTERVAL	Equal	EGL_DONT_CARE
EGL_SAMPLE_BUFFERS	Minimum	0
EGL_SAMPLES	Minimum	0
EGL_ALPHA_MASK_SIZE	Minimum	0
EGL_TRANSPARENT_TYPE	Equal	EGL_NONE
EGL_TRANSPARENT_RED_VALUE	Equal	EGL_DONT_CARE
EGL_TRANSPARENT_GREEN_VALUE	Equal	EGL_DONT_CARE
EGL_TRANSPARENT_BLUE_VALUE	Equal	EGL_DONT_CARE

EGL 使用一系列规则在返回给我们之前为匹配结果排序。基本上说，警告部分会最早进行匹配，然后是颜色通道深度，接着是总缓冲区大小，最后是采样缓冲区信息。所以能够最好匹配的配置应该是最优的。在获得匹配配置之后，我们可以详细查看这些结果来确定最好的选择。通常第一个就足够了。

可以使用 `eglGetConfigAttrib` 来对每个配置的属性进行分析, 这允许我们查询一个配置的属性, 每一次查询一个属性。

```
EGLBoolean eglGetConfigAttrib(EGLDisplay dpy, EGLConfig config,
                             EGLint attribute, EGLint *value);
```

如果要选择更加 “hands-on (亲自动手)” 的方式来选择一个配置, 还有一个访问所支持配置的更直接方法, 即可以使用 `eglGetConfigs` 来获取 EGL 支持的所有配置。

```
EGLBoolean eglGetConfigs(EGLDisplay dpy, EGLConfig *configs,
                        EGLint config_size, EGLint *num_configs);
```

这个函数和 `eglChooseConfig` 非常相似, 只是它会返回一个不依赖搜索条件的列表。返回的配置数将是可用的最大数量, 或者是 `config_size` 传入的数字, 选择两者中较小的一个。还有一个缓冲区需要基于预期的格式数量而进行预分配。在得到列表之后, 我们就该用 `eglGetConfigAttrib` 检查每一个选项, 从而选择最好的一个了。多个不同的平台不太可能有同样的配置或以同样的顺序列出配置, 所以我们要正确地选择配置, 而不是盲目地使用配置句柄。

## 创建渲染表面

现在我们了解了如何选择一个符合我们需要的配置, 可以开始创建一个实际的渲染表面了。虽然创建 `pBuffer` 和像素图等不可显示的表面也是有可能的, 但我们还是重点关注窗口表面。第一步是创建属性和我们选择的配置属性相同的本地窗口。然后就可以使用窗口句柄创建一个窗口表面了。窗口句柄类型和我们使用的平台或操作系统有关。这样同一个接口就可以支持许多不同的操作系统, 而不用为每个操作系统定义新的方法了。

```
EGLSurface eglCreateWindowSurface(EGLDisplay dpy, EGLConfig config,
                                  NativeWindowType win, EGLint *attrib_list);
```

如果调用成功, 将会返回屏幕显示表面的句柄。 `attrib_list` 参数本来是要指定窗口参数的, 但现在却没有进行定义。在完成渲染之后, 我们还要使用 `eglDestroySurface` 函数来销毁表面。

```
EGLBoolean eglDestroySurface(EGLDisplay dpy, EGLSurface surface);
```

在创建了窗口渲染表面并完成了硬件资源的配置之后, 我们的准备工作差不多就做好了。

## 16.4.3 环境管理

最后一步是创建一个渲染环境以供使用。渲染环境是进行渲染时要用到的一组状态, 至少要创建一个能够得到所有硬件支持的环境。

```
EGLContext eglCreateContext(EGLDisplay dpy, EGLConfig config,
                           EGLContext share_context, const EGLint *attrib_list);
```

我们可以调用带有我们一直使用的显示句柄的 `eglCreateContext` 函数创建环境, 还要传入创建渲染表面用的配置。创建环境所用的配置必须和创建窗口所用的配置相兼容。

`share_context` 参数用来在不同环境之间共享纹理和着色器等对象。传入我们想要进行共享的环境。如果没有必要共享,那么通常情况下在这里我们会传递 `EGL_NO_CONTEXT`。如果环境成功创建,环境句柄将会传回;否则,将返回 `EGL_NO_CONTEXT`。

现在已经有有了一个渲染表面和一个渲染环境,我们准备好了!既然我们可以使用多个环境来进行渲染,那么在这里最后要做的就是通知 EGL 我们想用哪个环境。使用 `eglMakeCurrent` 将一个环境设置成当前环境。我们可以使用刚刚创建的表面作为读取和绘制表面。

```
EGLBoolean eglMakeCurrent(EGLDisplay dpy, EGLSurface draw,
                          EGLSurface read, EGLContext ctx);
```

如果绘制和读取表面无效,或者它们与环境不兼容,我们会得到一个错误。为了释放一个绑定的环境,可以调用以 `EGL_NO_CONTEXT` 作为环境的 `eglMakeCurrent`。在释放环境时,必须使用 `EGL_NO_SURFACE` 作为读取和写入表面,调用 `eglDestroyContext` 删除用完的环境。

```
EGLBoolean eglDestroyContext(EGLDisplay dpy, EGLContext ctx);
```

## 16.4.4 呈现缓冲区和渲染同步

在渲染过程中,为了运行顺畅,我们可能会用到一些特定 EGL 函数。其中第一个就是 `eglSwapBuffers`。这个接口允许我们在窗口上呈现一个颜色缓冲区。我们只要传递到所希望的窗口表面即可。

```
EGLBoolean eglSwapBuffers(EGLDisplay dpy, EGLSurface surface);
```

仅仅是调用了 `eglSwapBuffers` 并不意味着已经到了实际将缓冲区传递到监视器的最佳时机。在 `eglSwapBuffers` 被调用时,显示器有可能正在显示帧的过程中。这种情况会导致一种叫做“图像撕裂 (tearing)”的典型后果,看起来就像这一帧在水平线上发生了轻微的歪曲。EGL 提供了一种方式来确定是否要等到当前绘图完成后再将完成交换的缓冲区传递到显示器。

```
EGLBoolean eglSwapInterval(EGLDisplay dpy, EGLint interval);
```

通过将交换间隔设置为 0,我们可以通知 EGL 不要进行同步交换,并且应该立刻传递一个 `eglSwapBuffers` 调用。交换间隔的默认值为 1,这代表每次交换都会与下一次向显示器传递同步。时间间隔将会被截取到 `EGL_MIN_SWAP_INTERVAL` 和 `EGL_MAX_SWAP_INTERVAL` 的值。

如果我们计划使用 OpenGL ES/EGL 以外的其他 API 来渲染到窗口,可以通过一些方法来保证渲染以正确的顺序传递。

```
EGLBoolean eglWaitGL(void);
EGLBoolean eglWaitNative(EGLint engine);
```

使用 `eglWaitGL` 可以防止其他 API 在 OpenGL ES 渲染结束之前通过操作一个窗口表面来进行渲染。而使用 `eglWaitNative` 则可以防止 OpenGL ES 在本地渲染完成之前执行。我们可以在 EGL 扩展中针对一个实现来定义引擎参数,但也可以使用 `EGL_CORE_NATIVE_ENGINE`,并且会引用除 OpenGL ES 之外最常用的本地渲染引擎。这要依具体实现/系统而定。

## 16.4.5 更多关于 EGL 的内容

我们已经了解了最重要和最常用的 EGL 接口。下面还有一些函数要讨论，它们对于通常的执行路径来说是比较次要的。

### EGL 错误

EGL 提供了一种方法来获取 EGL 特有的错误，这些错误可能会在 EGL 执行过程中发生。大多数函数返回 EGL\_TRUE 或 EGL\_FALSE 来指示它们是不是已经成功运行，但在失败的情况下，一个布尔值能够提供的关于问题所在的信息太少了。在这种情况下，可以调用 `eglGetError` 获取更多信息。

```
EGLint eglGetError();
```

最后发生的错误将被返回，它将是下列意义不言而喻的错误中的一个：

EGL\_SUCCESS、EGL\_NOT\_INITIALIZED、EGL\_BAD\_ACCESS、EGL\_BAD\_ALLOC、EGL\_BAD\_ATTRIBUTE、EGL\_BAD\_CONTEXT、EGL\_BAD\_CONFIG、EGL\_BAD\_CURRENT\_SURFACE、EGL\_BAD\_DISPLAY、EGL\_BAD\_SURFACE、EGL\_BAD\_MATCH、EGL\_BAD\_PARAMETER、EGL\_BAD\_NATIVE\_PIXMAP、EGL\_BAD\_NATIVE\_WINDOW 或 EGL\_CONTEXT\_LOST。

### 获取 EGL 字符串

有一些 EGL 状态字符串可能会很有用，这其中就包括 EGL 版本字符串和扩展字符串。要获取这些字符串，可以使用带有 EGL\_VERSION 和 EGL\_EXTENSIONS 枚举值的 `eglQueryString` 接口。

```
const char *eglQueryString(EGLDisplay dpy, EGLint name);
```

### 扩展 EGL

像 OpenGL 一样，EGL 也提供对各种扩展的支持。这些扩展常常是针对当前平台的，并能够提供核心规范所不能提供的扩展功能。我们可以使用前面讨论过的 `eglQueryString` 函数来查看系统支持哪些扩展。可以访问 Khronos 网站，以获取更多关于特定扩展的信息。这些扩展中，有一些可能会需要额外的入口点。将新入口点的名称传递到下面的函数中，可以得到这些扩展的入口点地址。

```
void (*eglGetProcAddress(const char *procname))();
```

这个入口点的使用与 `wglGetProcAddress` 非常相似。如果返回 NULL，说明这个入口点不存在。但是只是返回一个非 NULL 的地址也不能说明实际上能够支持这个函数。相关的扩展还必须包含在 EGL 扩展字符串或 OpenGL ES 扩展字符串中。确保我们调用 `eglGetProcAddress` 后返回了一个有效的函数指针（不是 NULL），这一点非常重要。

## 16.5 处理嵌入式环境

在考查了 OpenGL ES 和 EGL 如何在嵌入式系统中使用之后，现在可以进一步了解嵌入式系统的环境，以及它们如何影响 OpenGL ES 应用程序了。这个环境在我们如何着手创建 ES 应用程序上将扮演非常重要的角色。

### 16.5.1 流行的操作系统

由于 OpenGL ES 不像许多 3D API 那样受限于特定平台，所以使用的操作系统也是多种多样。这一点通常已经为我们决定好了，因为大多数嵌入式系统是为特定操作系统而设计的，而特定操作系统又要在特定硬件上运行。

Brew 和 Symbian 就是常见的手机操作系统。但是 iPhone OS 是速度最快的操作系统之一，它运行在 iPhone、iPod Touch 和 iPad 上。实际上，几乎任何使用 Mac 的人都可以为 iPhone OS 创建一个应用程序，并将它提交到 Apple App Store( 苹果应用程序商店 )。在接下来的内容中，我们将了解如何为 iPhone 创建应用程序。

### 16.5.2 供应商特定扩展

每个 OpenGL ES 供应商通常都有其硬件和平台特定的一系列扩展。它们常常扩展了可用格式的数量和类型。因为这些扩展只能用于有限的硬件，所以在此我们不进行讨论。

### 16.5.3 个人玩家

如果没有条件使用硬件仿真程序或真实硬件，同时又想在 OpenGL ES 上进行尝试，我们还有其他的选项。有一些 OpenGL ES 实现可以在全功能的操作系统上执行。它们也非常适合进行初步开发。

为此 AMD 还提供了一个仿真器，本书后面的参考部分提供了这个仿真器的链接。我们可以使用这个仿真器开始在桌面计算机上编写 OpenGL ES 2.0 应用程序。这个仿真器包含一个简单的程序，我们可以先从它开始。我们可以使用这个简单的示例程序为基础，建立自己的 OpenGL ES 2.0 应用程序。这个仿真器甚至还有一个控制面板，使我们可以复现一个嵌入式环境的约束条件。

如果我们有一台基于英特尔的 Mac，那么还可以免费使用苹果的 iPhone SDK。苹果的 iPhone、iPod Touch 和新 iPad 都是基于 OpenGL ES 的设备。甚至在没有设备的条件下，SDK 和 Xcode 工具也可以在软件仿真器中进行开发，而无需硬件设备。

## 16.6 苹果掌上平台

使用 OpenGL ES 2.0 的苹果掌上平台不只一个, 而是有 3 个: iPhone、the iPod Touch 和最新的 iPad。iPhone 是有史以来最流行的智能手机之一, 它拥有分辨率为 320 x 480 (最新的 iPhone 4 分辨率为 960x640) 的触摸屏界面。第一代和第二代 iPhone 支持 OpenGL ES 1.1, 而第 3 代之后则支持 OpenGL ES 1.1 或 2.0 应用程序。因为我们只关心最新的并且与本书内容联系最紧密的版本, 所以我们将讨论范围限制在 iPhone 上的 OpenGL ES 2.0。那么 iPod Touch 又如何呢? 我可以肯定, 苹果的营销大师不会同意我这个观点, 但是对于 OpenGL 程序设计来说, iPod Touch 只不过是一部没有电话部分的 iPhone 而已。

至于 iPad, 我们可以把它看成是“大字版”的 iPod Touch——实物更大一些而已。在它的外壳下确实配备了一颗不同的图像处理器, 但是幸运的是, OpenGL 为我们隐藏了类似这样的细节。为了让事情看起来简单些, 我们不会在所有的地方都不厌其烦地说明“iPhone/iPod Touch/iPad”。我们只提 iPhone, 读者应该明白对于 OpenGL ES 程序设计来说, 其他两种设备基本上没什么区别。

好了, 现在来看一个例子如何?

### 16.6.1 设置 iPhone 项目

我们需要做的第一件事就是从苹果的开发者相关网站 (<http://developer.apple.com>) 上获取 iPhone SDK。在加载 Xcode 时, 会显示类似于图 16.3 所示的画面。



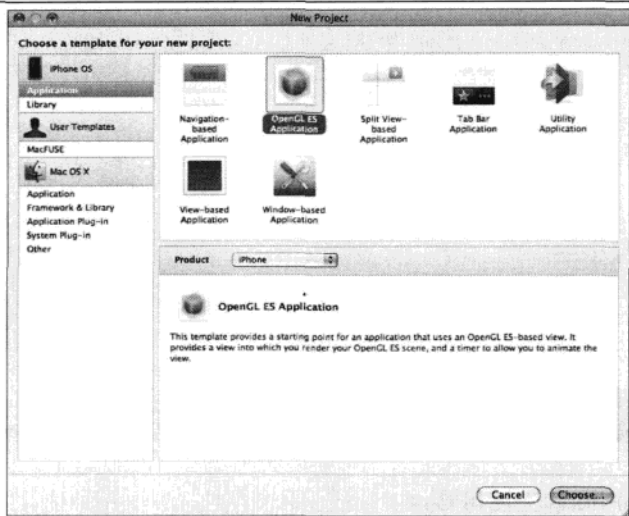
图 16.3

Xcode 欢迎画面

如果我们最近进行过其他项目, 那么就会看到它们就列在了“Recent Projects”(最近项目)下面。单击“Create a New Xcode Project”(创建新的 Xcode 项目)按钮, 打开项目向导窗口。在图 16.4 中列出的“New Project”(新项目)窗口中, 从 iPhone OS 组(看到了吧, 根本没有 iPod Touch OS 组或 iPad OS 组, 它们其实都是一回事)中选择“Application”(应用程序)。在上方的面板中, 我们可以看到各种各样的应用程序模板, 其中一个就是 OpenGL ES 应用程序。单击加亮它, 并单击窗口右下角的“Choose”(选择)按钮。

图 16.4

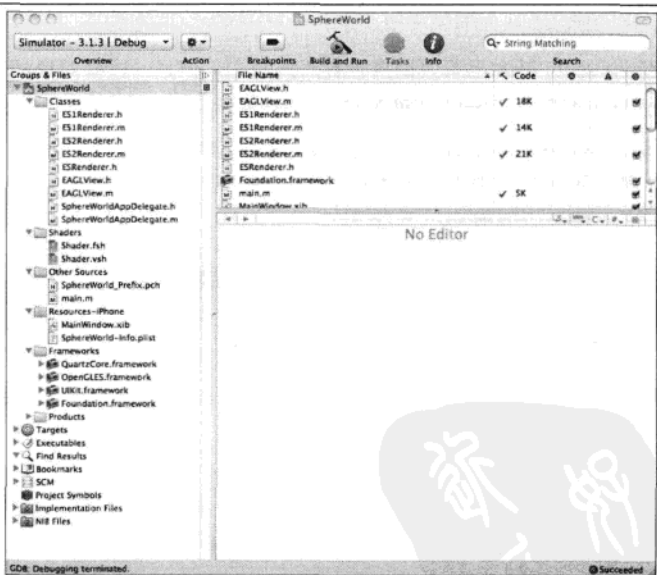
Xcode 新项目画面



下一个界面将要求我们选择一个位置，并对新项目进行命名。选择 SphereWorld——没错，我们还要使用第 5 章中的 SphereWorld，并将它移植到 iPhone 上。我们的项目一创建，就能看到一个与图 16.5 所示类似的界面。

图 16.5

刚刚创建的 SphereWorld 项目



这里我们展开了所有组，以便能够看到所有组成我们项目的文件和框架。我们还要确保在左上角的组合框中选择或改成了一个“Simulator”（仿真）选项，而不是一个“Device”（设备）选项。将应用程序安装到设备上并对硬件证书进行配置，这已经远远超出了本书所涉及的内容。在附录 A 中，列出了两本我们最喜欢的 iPhone 程序设计图书。我们只需按下 Command-R 组合键在仿真器中对应用程序进行编译、连接和加载，这对于 Xcode 来说是典型的操作。默认的 OpenGL ES 应用程序只是一个弹跳的着色三角形带。在桌面系统中运行 Xcode 的仿真器如图 16.6 所示。

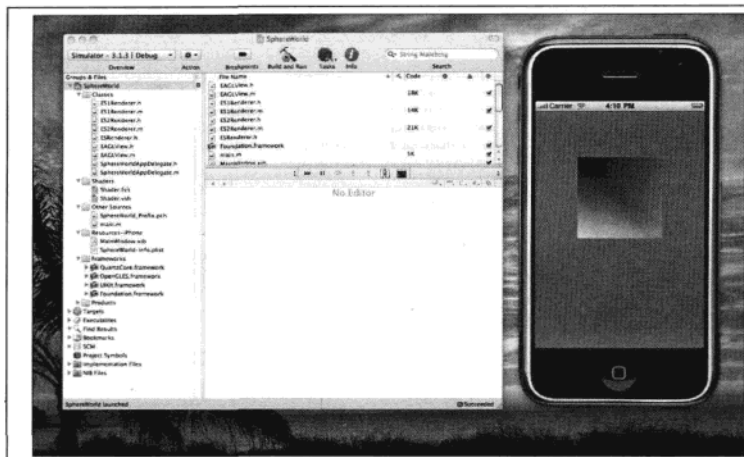


图 16.6

我们的 OpenGL ES 开发环境

### 在 iPhone 上使用 C++

本地 iPhone 编程环境使用的是 Objective C 程序设计语言。这一点引起了广泛的抱怨甚至是批评，因为全世界大部分非 Mac 程序员都愿意使用 C++。实际上，事实证明很多 Mac 程序员也愿意使用 C++。但是，除了利用苹果框架的情况之外，我们代码中的其他部分没有理由不能使用 C++，并且实际上，我们在将 SphereWorld 示例移植到 iPhone 时还要尽可能地使用 Objective C。

Objective C 基本上可以说是带有对象的 C。但是，这些对象的行为和 C++ 对象不同，并且将 C++ 植入 Objective C 并不像将 C 植入 Objective C 那么好用。还有一种简单而且几乎可以说很普通的方法可以做到这一点，就是将所有 Objective C 文件从 \*.m 重命名为 \*.mm。现在我们本质上使用的是“Objective C++”，并且可以轻松地在项目中植入 C++ 代码，在 Objective C 代码中创建并使用 C++ 类，并且从 Objective C 模块中调用 C++ 方法了。

然后，下一步工作是将项目中所有 .m 源文件重名为 .mm。为什么要这样做呢？因为 GLTools 是一个 C++ 库，并且它在 iPhone 上用得很好。右键单击 Groups & Files 窗格中的每个文件，并选择 Rename（重命名）。这样就将文件名变为编辑控制形式，我们可以在文件名上添加一个额外的“m”并按回车键。在项目中的所有 .m 文件上都执行这项操作。我们可以在完成这些工作后重建这个项目，以进行完整性检查。

### 在 iPhone 上使用 GLTools

GLTools 库能够在 iPhone 上使用，并且会有一些 #define 来对平台的差异进行解释。最明显的是，这些着色器针对着色语言中的某些差异进行了一些轻微的调整，但是它们的功能是相同的。从 API 的层面来说，GLTools 在 iPhone 上看起来和用起来都与在桌面系统中相同。

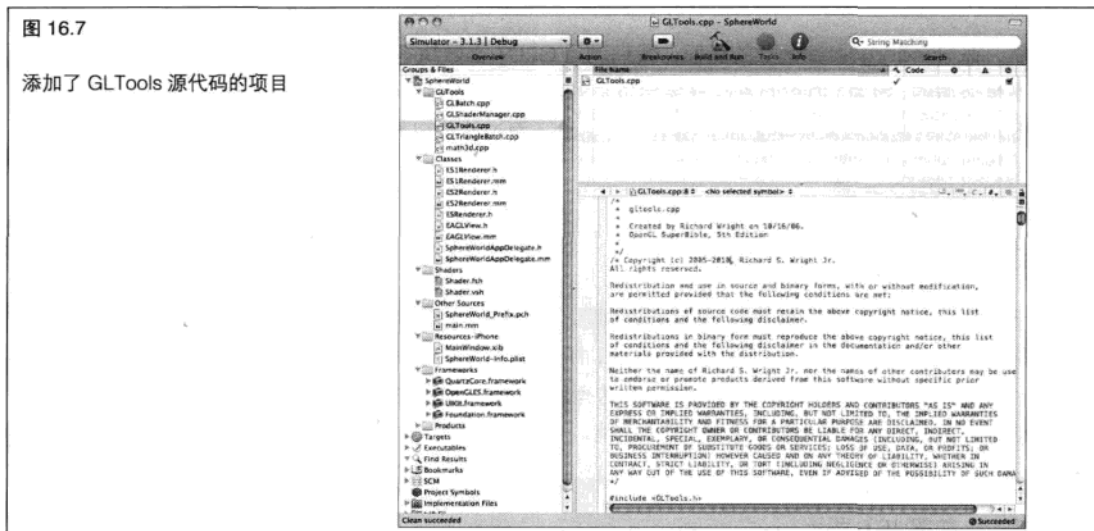
我们可以在从 iPhone 的 Objective C 框架中调用的 C++ 模块中使用它，或者也可以就在一个 Objective 文件中的恰当位置对它进行编码。

我们只是简单地将 GLTools 源代码添加到项目中，而不是为了不同的 iPhone SDK 版本、仿真器、设备等采用多个版本的 GLTools 库。右键单击 Groups & Files 下的项目名称，并在弹出菜单中选择“Add”、

“New Group”。对 GLTools 组进行命名。接下来，右键单击新创建的 GLTools 组，并选择“Add”(添加)、“Existing Files”(已存在文件)。使用出现的文件对话框，导航到 GLTools 源文件目录 (Src/GLTools/src) 并选择除 glew.c 以外的所有源代码。我们不会在 iPhone 上使用 GLEW，因为它只适用于桌面环境开发，而 GLTools 中的 OpenGL ES 2.0 代码则不需要它。一旦添加了 GLTools 文件，我们的 Xcode 项目看起来就应该类似于图 16.7 所示界面。

图 16.7

添加了 GLTools 源代码的项目



但是，我们仍然需要告诉 Xcode，GLTools 头文件在哪里，然后才能进行代码编译。这个处理过程与第 2 章中为桌面 OS X 系统设置 Xcode 项目时介绍的一样，我们在这里就不再重复介绍了。现在，我们应该为了进行健全性测试而进行创建工作。如果一切正常，我们就有了一个能够在 iPhone 仿真器上(或者是在设备上，如果有硬件证书的话)运行的 OpenGL ES 框架，并且已经连接了 GLTools 库。我们马上就可以看到，将桌面系统上的 OpenGL 代码移植到 iPhone 上非常简单。

## 16.6.2 移植到 iPhone

移植第 5 章中基于 GLUT 的示例非常简单。实际上，首先要做的就是将第 5 章中的 SphereWorld.cpp 文件直接复制到项目目录，并将它添加到项目中。另外还要复制 3 个文件，即 Marble.tga、Moonlike.tga 和 Marslike.tga。将这 3 个纹理文件添加到项目中的 Resources-iPhone 文件夹，并将 SphereWorld.cpp 文件添加到“Other Sources”(其他源)组中的项目中。

现在，SphereWorld.cpp 完全是基于 GLUT 的程序了，我们不需要也不希望在 iPhone 程序中有 GLUT。基本上我们需要做的只是对源文件顶部的头文件进行调整，并删除所有 GLUT 特定的代码。在接近顶部，位于大多数头文件后面的地方，我们可以看到 GLUT include 文件。

```
#ifdef __APPLE__
#include <glut/glut.h>
#else
#define FREEGLUT_STATIC
#include <gl/glut.h>
#endif
```

只要将这部分代码整个删除即可。现在没有 GLUT 了。在这里声明的所有模块变量对这个模块来说都是全局的,我们要调用在 GLUT 中注册的那些同样的函数作为 Objective C 框架中的回调函数。下一行有问题的代码是在 SetupRC 函数中,我们就是在这里进行 GLEW 初始化的。现在我們也不需要 GLEW 了,所以同样可以删除这行代码。

```
// 确保 OpenGL 入口点已经设置好  
glewInit();
```

然后,在 RenderScene 函数的底部,我们调用 GLUT 函数执行缓冲区交换,并触发一次更新。这些工作都不是必需的,因为 Objective C 框架会负责对我们的渲染进行显示,并且默认设置一个动画计时器,以最大 60 帧/秒的速度触发屏幕更新。接下来,删除下面这些代码行。

```
// 进行缓冲区交换  
glutSwapBuffers();  
  
// 再来一次  
glutPostRedisplay();
```

下一项删除工作甚至更加简单,只要完全删除 SpecialKeys 和 main 函数即可。因为这个源文件只包含供真正的 main 函数进行访问的函数和数据,所以这个 main 函数就用处不大了。但是,SpecialKeys 在 iPhone 版本的 SphereWorld 中确实仍然非常有用。它能够使用方向键在 SphereWorld 中进行移动,而我们必须使用 Cocoa Touch API 代替这项功能。

现在,我们的项目应该能够进行编译和运行而不出现任何问题。遗憾的是,得到的仍然是那个弹跳的着色矩形,如图 16.6 所示。

## 进行连接

第一堂 C 或 C++ 课或者第一本 C 或 C++ 图书应该已经教会我们如何调用一个已经在另一个模块中进行过声明的函数了。我们只要在想调用这些函数的模块中对这些在另一个其他模块中的函数或类进行声明就可以了。典型情况下,这项工作可以使用一个头文件来完成。在 SphereWorld 中,只需要从 SphereWorld.cpp 中调用 4 个函数,它们的声明如下所示。

```
void ChangeSize(int nWidth, int nHeight);  
void SetupRC(void);  
void ShutdownRC(void);  
void RenderScene(void);
```

我们只要将这些代码放置到 ES2Renderer.mm 文件的顶部就可以了。正如前面说过的那样,Objective C 将会调用这些在.cpp 文件中声明的函数,而不会出现任何问题。现在的问题是,将这 4 个函数放在哪里,以及如何摆脱这个弹跳的矩形。

iPhone SDK 能够支持使用 OpenGL ES 1.1 或 OpenGL ES 2.0。默认情况下,示例程序会使用 OpenGL ES 2.0,这也正是我们所希望的。如果想要面向更广泛的用户,甚至可以在运行时决定哪个可能更有用,但是这样做当然需要有两组完整的渲染代码。对于 OpenGL ES 2.0 的情况来说,所有动作都发生在 ES2Renderer.mm 中,因此我们也会在这里对 OpenGL 代码进行连接。

在 iPhone 上,所有 OpenGL ES 渲染都会在一个帧缓冲区对象上完成。这个模块中的 init 方法将执行这个帧缓冲区对象的初始化,而我们则将对 SetupRC 的调用放在这里,正好在帧缓冲区对象初始化之后。

```

.....
.....
glBindRenderbuffer(GL_RENDERBUFFER, colorRenderbuffer);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                           GL_RENDERBUFFER, colorRenderbuffer);
}

// 向 SphereWorld 中进行调用来完成它的初始化
SetupRC();

return self;
}

```

我们将遇到的下一个方法是 render。render 包含一组 OpenGL 渲染调用和帧缓冲区管理调用，我们不能对这些调用进行干预。

程序清单 16.1 完整地展示了这个函数。

程序清单 16.1 新的精简渲染方法

```

- (void)render
{
    // 替换这个方法的实现，来进行我们自己的自定义绘制
    // 好的，谢谢，我会的

    // 这个应用程序只创建单个环境，这时它已经被设置为当前环境。
    // 这个调用是多余的，但是在处理多个环境时会需要它
    [EAGLContext setCurrentContext:context];

    // 这个应用程序只创建单个默认帧缓冲区，这时它已经进行绑定
    // 这个调用是多余的，但是在处理多个帧缓冲区时会需要它
    glBindFramebuffer(GL_FRAMEBUFFER, defaultFramebuffer);

    // 进行绘制——调用我们的 SphereWorld 渲染例程！
    RenderScene();

    // 这个应用程序只创建颜色渲染缓冲区，这时它已经进行绑定
    // 这个调用是多余的，但是在处理多个渲染缓冲区时会需要它
    glBindRenderbuffer(GL_RENDERBUFFER, colorRenderbuffer);
    [context presentRenderbuffer:GL_RENDERBUFFER];
}

```

差不多完成了！至少要调用一次以帧缓冲区大小为参数的 `resizeFromLayer` 方法。在这里，使用 `backingWidth` 和 `backingHeight` 类变量，并从 `SphereWorld.cpp` 调用 `ChangeSize` 函数。

```
ChangeSize(backingWidth, backingHeight);
```

最后，将下面的回调函数 `ShutdownRC` 放置到 `dealloc` 方法的开始部分。

```

- (void)dealloc
{
    ShutdownRC();

    // 消除 GL
    .....
}

```

最终，应该在 iPhone 上看到 `SphereWorld` 了，是吧？不尽然。如果现在运行这个程序，就会看到一个空白的屏幕。我们还有最后一项工作要做，就是告诉应用程序到哪里去寻找纹理文件。

## 纹理注意事项

要使第 5 章中的纹理代码适用于 iPhone, 还有两件事情要做。第一件事情是, 需要将工作目录设置到与应用程序簇相同的目录。当将纹理文件添加到 Xcode 中的 Resources-iPhone 组时, 告诉 Xcode 将这些文件不加修改地进行打包, 并将它们放到 iPhone 中应用程序簇所在的同一个目录中。当试图加载纹理时, 就会传递这个纹理的文件名, 但是需要确保当前工作文件夹 (因为我们正在使用标准 C 运行时函数 `fopen` 对这个文件进行访问) 与应用程序簇的文件夹相同。在桌面 Mac OS X 上也会遇到同样的问题, 而我们将在 GLUT 程序的 `main` 函数中调用 `glSetWorkingDirectory` 函数。我们只需要做同样的工作, 但这一次是在 `main.mm` 模块中。程序清单 16.2 显示了完整的 `main.mm` 模块。它的内容很少, 只添加了 GLTools 头文件和 `glSetWorkingDirectory` 调用。

程序清单 16.2 修改后的主函数

```
#import <UIKit/UIKit.h>
#include <GLTools.h>

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

    glSetWorkingDirectory(*argv);

    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

第二件事已经完成了, 但是还是值得再提一下的。OpenGL ES 并不支持 TGA 文件使用的 `GL_BGR` 纹理图像格式。GLTools 考虑到了这种情况, 并且在 iPhone 上加载一个 TGA 时进行了颜色交换。否则, 这个示例中微红的纹理看起来就会是蓝色的。OpenGL ES 也不支持通用纹理压缩属性, 所以在 `LoadTGA` 函数中, 需要使用从 `glReadTGABits` 返回的 `nComponents` 值。

```
glTexImage2D(GL_TEXTURE_2D, 0, nComponents, nWidth, nHeight, 0,
             eFormat, GL_UNSIGNED_BYTE, pBits);
```

现在已经接近完成了。图 16.8 所示展示了进展情况, 也显示了最后一个障碍。在花托上看到的不自然痕迹是由于没有深度缓冲区造成的……至少现在还没有。



图 16.8

差不多完成了, 只差一个深度缓冲区

## 添加深度缓冲区

在默认情况下, iPhone SDK 并没有在创建 OpenGL ES 项目时为我们提供深度缓冲区。本来可以开启一个#define, 它能够触发提供深度缓冲区的代码, 但是出于某些原因, 苹果将这项功能从新的 SDK 中删除了。

不过, 创建并绑定自己的深度缓冲区也非常简单(相关细节参见第9章)。程序清单 16.3 完整地展示了 `resizeFromLayer` 程序的代码, 我们就在这里创建完整的帧缓冲区对象。

程序清单 16.3 将一个深度缓冲区绑定到我们的帧缓冲区对象

```
-(BOOL)resizeFromLayer:(CAEAGLLayer *)layer
{
    // 根据当前层大小分配颜色缓冲区背景
    glBindRenderbuffer(GL_RENDERBUFFER, colorRenderbuffer);
    [context renderbufferStorage:GL_RENDERBUFFER fromDrawable:layer];
    glGetRenderbufferParameteriv(GL_RENDERBUFFER, GL_RENDERBUFFER_WIDTH,
                                &backingWidth);
    glGetRenderbufferParameteriv(GL_RENDERBUFFER, GL_RENDERBUFFER_HEIGHT,
                                &backingHeight);

    glGenRenderbuffers(1, &depthRenderbuffer);
    glBindRenderbuffer(GL_RENDERBUFFER, depthRenderbuffer);
    glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT16,
                        backingWidth, backingHeight);
    glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                            GL_RENDERBUFFER, depthRenderbuffer);

    if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
    {
        NSLog(@"Failed to make complete framebuffer object %x",
              glCheckFramebufferStatus(GL_FRAMEBUFFER));
        return NO;
    }

    ChangeSize(backingWidth, backingHeight);

    return YES;
}
```

终于成功了!

## 风景模式

在旋转电话角度的时候, 很多 iPhone 应用程序都会检测到这种变化。我们并不打算讨论这种情况, 因为这需要很多与 OpenGL 无关的垂直检测方面的内容, 而与我们有关的内容很少, 并且在 OpenGL ES 存储模板中添加它们也并不常见。在这里, 我们还是建议读者参考附录 A 来满足自己的好奇心。我们并不想提供一个 iPhone 指南, 要做的只是介绍 OpenGL 如何在这个平台上使用。如此说来, 如何进行风景模式的渲染呢? 实际上, 很多 OpenGL ES 游戏都只能在风景模式下运行, 而只要强制 SphereWorld 在风景模式下运行就可以了, 是吧? 确实, 看一看程序清单 16.4, 其中的 `ChangeSize` 函数进行了一个简单的修改, 就可以实现这一点了。

## 程序清单 16.4 旋转我们的视点

```
void ChangeSize(int nWidth, int nHeight)
{
    glViewport(0, 0, nWidth, nHeight);
    transformPipeline.SetMatrixStacks(modelViewMatrix, projectionMatrix);

    viewFrustum.SetPerspective(60.0f, float(nWidth)/float(nHeight), 1.0f, 100.0f);
    projectionMatrix.LoadMatrix(viewFrustum.GetProjectionMatrix());
    projectionMatrix.Rotate(-90.0f, 0.0f, 0.0f, 1.0f);
    modelViewMatrix.LoadIdentity();
}
```

我们真正需要做的只是将投影矩阵旋转  $90^\circ$  而已。另一项调节就是将视野增加到  $60^\circ$ ，以使我们能够看到更多的场景。这样就能获得 SphereWorld 的一个赏心悦目的视觉效果了，如图 16.9 所示。

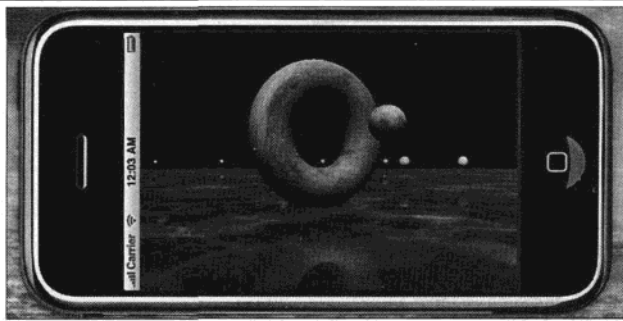


图 16.9

在 iPhone 上以风景模式运行 SphereWorld

## 触摸屏

终于轮到 SphereWorld 中要实现的最后一个特性——移动功能了。有很多方法可以实现这项功能，某些游戏则使用加速度计来通过内部感知进行游戏导航。Touch API 是一个更加简单的接口。Touch API 的使用非常简单，类似于检测屏幕上的鼠标移动。触摸消息被路由到 view（由 UIView 派生而来），所以我们必须跳出 OpenGL ES 特定框架，来到定义类 EAGLView 的 EAGLView.mm 文件。

程序清单 16.5 展示了添加到这个消息，这个消息用于接收触摸移动通知。这个消息同时包含了新的触摸位置和上一轮的触摸位置，以实现  $x$  和  $y$  方向移动探测。

## 程序清单 16.5 使用触摸消息进行照相机移动

```
- (void) touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    CGPoint ptNow = [[touches anyObject] locationInView:self];
    CGPoint ptLast = [[touches anyObject] previousLocationInView:self];

    float deltaX = ptNow.x - ptLast.x;
    float deltaY = ptNow.y - ptLast.y;

    cameraFrame.MoveForward(deltaX * 0.05f);
    cameraFrame.RotateLocalY(m3dDegToRad(deltaY * 0.25f));
}
```

在这里，我们只随着上下移动将照相机前移，而随着左右移动使照相机左右旋转。运动的缩放量完全出于审美角度而选择。当然，照相机帧已经在 SphereWorld.cpp 中进行了声明，所以我们必须在 EAGLView.mm 文件的顶部对它进行声明，从而将它与这个文件进行共享。

```
#include <GLFrame.h>
extern GLFrame cameraFrame; // 照相机帧
```

## 16.7 小结

在本章，我们介绍了大量的背景知识。OpenGL ES 2.0 大体上是基于 OpenGL 2.0 开发的，它是一种更加简单和精简的 OpenGL 版本，专门用于嵌入式环境中。运行 OpenGL ES 2.0 的硬件类型多种多样。我们还介绍了 EGL，以及它是如何用于进行 OpenGL ES 窗口管理的。

另外，我们还了解了在嵌入式环境中进行开发的一些不同之处，以及在哪里能够找到用于进行 OpenGL ES 2.0 开发的仿真器。最后，我们演示了现代桌面 OpenGL 代码能够轻松地移植到类似 iPhone 这样的 OpenGL ES 环境中。OpenGL 确实是一种优秀的跨平台 3D API。



## A

## 附录 A 更多阅读建议

实时 3D 图形和 OpenGL 都是非常流行的话题，我们无法在一本书里表述所有相关的可用信息和应用技术。如果读者希望进一步扩充知识、增加经验，可以借助下面这些有用的资源。

### 其他优秀的 OpenGL 书籍

《Advanced Graphics Programming Using OpenGL》: Tom McReynolds and David Blythe. The Morgan Kaufmann Series in Computer Graphics, 2005。

《Interactive Computer Graphics: A Top-Down Approach with OpenGL, 4th Edition》: Edward Angel. Addison-Wesley, 2005。

《More OpenGL Game Programming》: Dave Astle, Editor. Thomson Course Technology, 2006。

《OpenGL ES 2.0 Programming Guide》: Aaftab Munshi, Dan Ginsburg, and Dave Shreiner. Addison-Wesley, 2008。

《OpenGL Programming Guide, 7th Edition: The Official Guide to Learning OpenGL, Version 3.0 and 3.1》: Dave Shreiner, The Khronos OpenGL ARB Working Group. Addison-Wesley, 2009。

《OpenGL Shading Language, 3rd Edition》: Randi J. Rost and Bill Licea-Kane. Addison-Wesley, 2009。

《OpenGL Programming on Mac OS X: Architecture, Performance, and Integration》: Robert P. Kuehne and J. D. Sullivan. Addison-Wesley, 2007。

《OpenGL Programming for the X Window System》: Mark J. Kilgard. Addison-Wesley, 1996。

### 3D 图形书籍

《3D Computer Graphics, 3rd Edition》: Alan Watt. Addison-Wesley, 1999。

《3D Math Primer for Graphics and Game Development》: Fletcher Dunn and Ian Parbery. Wordware Publishing, 2002。

《Advanced Animation and Rendering Techniques: Theory and Practice》: Alan Watt and Mark Watt (contributor). Addison-Wesley, 1992。

《Essential Mathematics for Games and Interactive Applications》: James Van Verth and Lars Bishop. The Morgan Kaufmann Series in Interactive 3d Technology, 2004。

《Introduction to Computer Graphics》: James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes, and Richard L. Phillips. Addison-Wesley, 1993。

《Mathematics for 3D Game Programming & Computer Graphics, 2nd Edition》: Eric Lengyel. Charles River Media, 2003。

《Open Geometry: OpenGL + Advanced Geometry》: Georg Glaeser and Hellmuth Stachel. Springer-Verlag, 1999。

《Shader X 4: Advanced Rendering Techniques》: Wolfgang Engel, Editor. Charles River Media, 2006。

《Texturing & Modeling: A Procedural Approach, 3rd Edition》: David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. The Morgan Kaufmann Series in Computer Graphics, 2003。

## 网址

OpenGL 官方网址: [www.opengl.org](http://www.opengl.org)

The OpenGL SDK (有大量指南和工具): [www.opengl.org/sdk/](http://www.opengl.org/sdk/)

上述 3 个网址是 OpenGL 网上信息的门户网站。当然, 这些信息包括与 OpenGL 和本书相关的所有官方资源。

下面的网址也是与本书涵盖的内容相关的, 并包含供应商特定 OpenGL 支持、指南、演示和新闻。

Khronos 小组 OpenGL ES 主页: [www.khronos.org/opengles/](http://www.khronos.org/opengles/)

OpenGL Extension Registry: [www.opengl.org/registry/](http://www.opengl.org/registry/)

AMD/ATI 开发者主页: [www.ati.amd.com/developer/](http://www.ati.amd.com/developer/)

NVIDIA 开发者主页: [developer.nvidia.com/](http://developer.nvidia.com/)

Mesa 3D OpenGL "work-a-like": [www.mesa3d.org](http://www.mesa3d.org)

GLView OpenGL Extension Viewer: [www.realtech-vr.com/glview](http://www.realtech-vr.com/glview)



## 附录 B 词汇表

- 锯齿 (Aliasing): 从技术上说, 它是在有限分辨率的情况下产生一幅图像所造成的信号信息的损失。它的特征是沿点、直线和多边形边缘的尖锐锯齿状外观, 这是由于数量有限的固定大小像素的本质决定的。
- Alpha: 第 4 种颜色成分值, 提供一个物体颜色的透明度。Alpha 值为 0.0 表示完全透明, 1.0 表示不透明。
- 环境光 (Ambient Light): 场景中并不是来自任何特定点光源或方向的光。环境光均匀地照射所有物体的所有表面。
- 抗锯齿 (Antialiasing): 一种用于对点、直线和多边形边缘进行平滑的渲染方法。这个技巧对邻近直线的像素颜色取平均值。它在视觉上具有淡化直线和它邻近像素之间的过渡现象的效果, 因此能够提供一种更加平滑的外观。
- ARB (体系结构审查委员会): OpenGL ARB 由 3D 图形硬件厂商组成, 以前负责维护 OpenGL 规范文档, 后来这项职能由 Khronos 小组接替。
- 纵横比 (Aspect Ratio): 一个窗口的宽度与高度之比。更具体地说, 就是一个窗口的宽度 (以像素为单位) 除以它的高度 (以像素为单位)。
- 贝赛尔曲线 (Bézier curve): 这种曲线的形状由靠近它的控制点所定义的, 而不是由组成曲线本身的点集合所定义的。
- 位平面 (Bitplane): 一个直接映射到屏幕像素的位数组。
- 缓冲区 (Buffer): 一种用于存储图像信息的内存区域。它所存储的可以是颜色、深度或混和信息。红、绿、蓝和 Alpha 缓冲区常常合起来称为颜色缓冲区。
- 笛卡尔 (Cartesian): 一种基于 3 条彼此垂直方向轴的坐标系统。这些坐标分别标以 x、y 和 z。
- 裁剪坐标 (Clip coordinates): 根据模型视图和投影转换而产生的 2D 几何坐标。
- 裁剪 (Clipping): 截除一个或一组图元的一部分。位于裁剪区域之外的点将不会被绘制。裁剪区域一般由投影矩阵指定。被裁剪的图元进行了重新构造, 它的边缘不会位于裁剪区域之外。
- 凸 (Convex): 表示多边形的一种形状。凸多边形没有内陷的部分, 不存在一条可以与之相交超过两次 (一次进、一次出) 的直线。
- 剔除 (Culling): 消除那些在渲染之后不被显示的图元。背面剔除消除一个图元的正面或背面, 使该面不可见。平截头体剔除消除位于可视平截头体区域之外的物体。
- 目标颜色 (Destination color): 颜色缓冲区中一个特定位置所存储的颜色。这个术语通常在混和操作中使用, 用于区分已经存在于颜色缓冲区和即将进入颜色缓冲区 (源颜色) 的颜色。
- 抖动 (Dithering): 一种通过把不同颜色的像素混和在一起以模拟更广颜色范围的方法, 以产生

两种颜色之间的着色。

- 双缓冲 (Double buffered): OpenGL 所使用的一种绘图技巧。被显示的图像首先被放在内存中, 然后通过一个更新操作放到屏幕上, 而不是直接就在屏幕上创建。双缓冲可以用来产生动画, 它的速度更快, 效果更平滑。
- 拉伸 (Extruded): 在一个 2D 图像或形状的表面上添加第三维的过程。这个过程可以把 2D 字体转换为 3D 字体。
- 视觉坐标 (Eye coordinates): 基于观察者位置的坐标系。观察者的位置正对 z 轴, 向 z 轴的负方向看过去。
- 平截头体 (Frustum): 一种金字塔形状的可视区域, 可以用来创建一个透视视图 (近的物体较大, 远的物体较小)。
- GLSL: OpenGL 着色语言的首字母缩写, 这是一种类似于 C 语言的高级着色语言。
- GLUT 函数库 (GLUT library): OpenGL 工具函数库。它是一个独立于窗口系统的工具库函数, 可以创建独立于操作系统和窗口系统的示例程序和简单的 3D 渲染程序。在典型情况下, 它用于提供 Windows、X-Window、Linux 等系统之间的可移植性。
- 立即模式 (Immediate mode): 一种图形渲染模式。在这种模式下, 命令和函数对渲染引擎的状态具有立即的效果。
- 实现 (Implementation): 基于软件或硬件的设备, 执行 OpenGL 渲染任务。
- Khronos 小组 (Khronos Group): 一个行业团体, 目前负责除几种工业标准外的 OpenGL 规范的维护和升级工作。
- 字面值 (Literal): 这是个值而不是变量名, 一般为直接嵌入到源代码中的字符串或数值。
- 矩阵 (Matrix): 一个包含数值的 2D 数组。矩阵可以根据数学的方式进行操作, 用于执行坐标转换。
- Mip 贴图 (Mipmapping): 一种使用纹理多层细节的技巧。这个技巧从一幅图像不同大小的一些版本中进行选择, 或者把两个最接近匹配大小的图像进行组合, 以产生用于纹理贴图的最终片断。
- 模型视图矩阵 (Modelview matrix): 把图元从视觉坐标转换为对象坐标的 OpenGL 矩阵。
- 法线 (Normal): 一种方向向量, 垂直于一个平面或表面。在使用时, 必须为一个图元的每个顶点都指定一条法线。
- 规范化 (Normalize): 把一条法线调整为单位长度。单位法线向量的长度正好为 1.0。
- 正交 (Orthographic): 一种绘图模式, 在这种模式下, 不会产生透视或透视缩短现象。又称平行投影。所以图元的长度和大小并不会发生变化, 与它们和观察者之间的方向和距离无关。
- 透视 (Perspective): 一种绘图模式, 远处的物体看上去比近处的物体更小一点。
- 像素 (Pixel): 表示图像元素, 它是计算机屏幕上可以分割的最小可视单位。像素以行和列的形式进行排列, 并单独进行设置, 可以设置为适当的颜色, 以渲染任何特定的图像。
- 像素图 (Pixmap): 组成一幅图像的颜色值的二维数组。像素图的名称由来是每个图像元素对应于屏幕上的一个像素。
- 多边形 (Polygon): 一种具有任何数量的边的 2D 形状 (至少 3 条边以上)。
- 图元 (Primitive): 由 OpenGL 所定义的 2D 多边形形状。所有的物体和场景都是由各种图元的不不同组合所产生的。
- 投影 (Projection): 直线、点和多边形从视觉坐标转换到屏幕上的裁剪坐标。
- 四边形 (Quadrilateral): 一种正好有 4 条边的多边形。
- 光栅化 (Rasterize): 把投影后的图元和位图转换到帧缓冲区中的像素片段的过程。
- 保留模式 (Retained mode) 一种 3D 编程类型, 对象的表示由程序库保持在内存中。
- 渲染 (Render): 把对象坐标中的图元转换到帧缓冲区中的一幅图像。渲染管线就是通过 OpenGL

命令和语句，把图元数据变成屏幕上的像素的过程。

- 闪烁 (Scintillation): 当一个非 Mip 贴图的纹理贴图应用到一个比纹理小得多的多边形时所产生的闪烁效果。
- 着色器 (Shader): 一个由图形处理硬件执行 (经常是并行的) 对独立顶点或像素进行操作。
- 源颜色 (Source Color): 将要进入颜色缓冲区的片段颜色，而已经存在于颜色缓冲区中的颜色则称为目标颜色。这个术语通常用于描述在一个混和操作中源颜色和目标颜色是如何进行组合的。
- 说明 (Specification): 详细说明 OpenGL 操作并详尽描述一个实现如何工作的设计文档。
- 样条 (Spline): 这个术语用于描述任何由邻近的控制点所定义的曲线 (或表面)。这种曲线形状具有拉伸效果。它类似于一块有弹性的材料被挤压时所产生的效果。
- 点画 (Stipple): 一种二进制位模式，用于设置帧缓冲区中像素生成的掩码。它类似黑白位图，但一维的图案用于直线，而二维的图案则用于多边形。
- 镶嵌 (Tessellation): 把一个复杂多边形或解析曲面分解为许多简单凸多边形的过程。这个过程还可以把一条复杂曲线分解为一系列的简单直线。
- 纹理单元 (Texel): 与像素 (图像元素) 类似，纹理单元就是纹理元素。一个纹理单元表示一个纹理中的一种颜色，应用到帧缓冲区中的一个像素片断。
- 纹理 (Texture): 一幅应用到一个图元表面的颜色图像图案。
- 纹理贴图 (Texture Mapping): 把一幅纹理图像应用到一个表面的过程。这个表面并不一定是平面的。纹理贴图常用于回线一个弯曲物体，或者产生如木纹或大理石条纹的表面图案。
- 转换 (Transformation): 对坐标系统的操作，包括旋转、移动、缩放 (一致性和非一致性) 以及透视除法。
- 半透明 (Translucence): 一个物体的透明程度。在 OpenGL 中，它是由 Alpha 值来表示的，其值的变化范围是从 1.0 (不透明) 到 0.0 (透明)。
- 向量 (Vector): 一种有方向的量，经常用 X、Y 和 Z 分量表示。
- 顶点 (Vertex): 空间中的单个点。除了用于点和直线图元之外，还用来定义多边形两条边的交界处。
- 可视区域 (Viewing Volume): 在窗口中可见的 3D 空间。可视区域外的物体和点被裁剪 (不可见)。
- 视口 (Viewport): 窗口中的一个区域，用于显示一幅 OpenGL 图像。通常，它包括了窗口的整个客户区域。经过缩放的视口可以在物理窗口中产生放大或缩小的输出。
- 线框 (Wireframe): 用网格或线表示一个实心物体的方式，而不是用实心着色的多边形。线框模型的渲染速度通常更快，并可以同时看到一个物体的正面和背面。

## 附录 C (核心) OpenGL 3.3 参考

### glActiveTexture

选择活动纹理单元。

#### C 规范

```
void glActiveTexture(GLenum texture);
```

#### 参数

texture

texture 指定要设为活动状态的纹理单元。纹理单元的数量随实现而变化,但最少必须为 2 个。texture 必须为 GL\_TEXTURE*i* 中的一个,其中 *i* 取值范围从 0 到(GL\_MAX\_COMBINED\_TEXTURE\_IMAGE\_UNITS - 1)之间。初始值为 GL\_TEXTURE0。

#### 描述

glActiveTexture 选择哪个纹理单元后的纹理状态调用将会起作用。一个实现支持的纹理单元的数量与具体实现有关,但至少必须为 48。

#### 错误

如果 texture 不是 GL\_TEXTURE*i* 中的一个,其中 *i* 的范围是从 0 到(GL\_MAX\_COMBINED\_TEXTURE\_IMAGE\_UNITS - 1),则会产生 GL\_INVALID\_ENUM 错误。

#### 相关 Get 函数

glGet, 其自变量值为 GL\_ACTIVE\_TEXTURE 或 GL\_MAX\_COMBINED\_TEXTURE\_IMAGE\_UNITS。

#### 另外查看

glGenTextures、glBindTexture、glCompressedTexImage1D、glCompressedTexImage2D、glCompressedTexImage3D、glCompressedTexSubImage1D、glCompressedTexSubImage2D、glCompressedTexSubImage3D、glCopyTexImage1D、glCopyTexImage2D、glCopyTexSubImage1D、glCopyTexSubImage2D、glCopyTexSubImage3D、glDeleteTextures、glIsTexture、glTexImage1D、glTexImage2D、glTexImage2DMultisample、glTexImage3D、glTexImage3DMultisample、glTexSubImage1D、glTexSubImage2D、glTexSubImage3D、glTexParameter。

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

### glAttachShader

将一个着色器对象绑定到一个程序对象。

#### C 规范

```
void glAttachShader(GLuint program,  
                    GLuint shader);
```

#### 参数

**program**

指定着色器对象将要绑定到的程序对象。

**shader**

指定要绑定的着色器对象。

**描述**

为了创建一个可执行程序，必须找到一种方法来指定一个列表，指明将要吧什么东西连接到一起。程序对象就提供了这个机制。必须首先将要在一个程序对象中连接在一起的着色器绑定到这个程序对象上。glAttachShader 将由 shader 指定的着色器绑定到由 program 指定的程序对象上。这说明 shader 将要被包含在将在 program 上进行的连接操作中。

所有能够在着色器对象上进行的操作都是合法的，无论着色器对象是否绑定到了程序对象上。在源代码载入到着色器对象之前，或者在着色器对象被编译之前，将着色器对象绑定到程序对象是允许的。绑定同一类型的多个着色器对象是允许的，因为它们中的每一个都可以包含完整着色器的一部分。将一个着色器绑定到多个程序对象也是允许的。如果一个着色器对象在它被绑定到一个程序对象时被删除了，那么它将被标记为删除状态，而在调用 glDetachShader 来将它从所有它绑定到的程序对象分离之前是不会进行删除的。

**错误**

如果 program 或 shader 不是由 OpenGL 产生的值，则产生 GL\_INVALID\_VALUE 错误。

如果 program 不是一个程序对象，则产生 GL\_INVALID\_OPERATION 错误。

如果 shader 不是一个着色器对象，则产生 GL\_INVALID\_OPERATION 错误。

如果 shader 已经绑定到了 program，则产生 GL\_INVALID\_OPERATION 错误。

**相关 Get 函数**

glGetAttachedShaders，带有合法程序对象的句柄。

glGetShaderInfoLog

glGetShaderSource

glIsProgram

glIsShader

**另外查看**

glCompileShader、glCreateShader、glDeleteShader、glDetachShader、glLinkProgram、glShaderSource。

**版权**

Copyright © 2003–2005 3Dlabs Inc. Ltd. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glBeginConditionalRender

开始进行条件渲染。

**C 规范**

```
void glBeginConditionalRender(GLuint id,
                             GLenum mode);
```

**参数**

**id**

指定一个遮挡查询对象的名称，这个对象的结果用来决定渲染命令是否被丢弃。

**mode**

指定 glBeginConditionalRender 如何对遮挡查询的结果进行解释。

**C 规范**

```
void glEndConditionalRender(void);
```

**描述**

通过使用 glBeginConditionalRender 开始条件渲染，而通过使用 glEndConditionalRender 结束条件渲染。在进行条件渲染过程中，如果查询对象 id 的 ( GL\_SAMPLES\_PASSED ) 结果为 0，或者 ( GL\_ANY\_SAMPLES\_PASSED ) 结果为 GL\_FALSE，那么所有顶点数组命令，以及 glClear 和 glClearBuffer 都是无效的。类似 glVertexAttrib 这样的当前顶点状态的设置命令结果将是未定义的。如果 ( GL\_SAMPLES\_PASSED ) 结果为非 0 值，或者如果

(GL\_ANY\_SAMPLES\_PASSED) 结果为 GL\_TRUE, 那么这些命令将会被丢弃。glBeginConditionalRender 的 id 参数必须与前面某次 glGenQueries 调用中返回查询对象的名称相同。mode 指定这个查询对象的结果将如何进行解释。如果 mode 为 GL\_QUERY\_WAIT, 那么 GL 会进行等待直到这个查询的结果可用, 然后使用这个结果来决定后续渲染命令是否被丢弃。如果 mode 为 GL\_QUERY\_NO\_WAIT, 那么 GL 可能会选择无条件执行后续渲染命令, 而不会等待这个查询完成。

如果 mode 为 GL\_QUERY\_BY\_REGION\_WAIT, 那么 GL 会等待遮挡查询结果, 如果这个遮挡查询的结果为 0, 那么渲染命令将被丢弃。如果查询结果是非 0 的, 那么后续渲染命令将被执行, 但是对于帧缓冲区中与特定遮挡查询的样本数无关的区域来说, GL 可能会丢弃命令的结果。这类丢弃会在与实现相关的方法中进行, 但是对于那些与遮挡查询样本数有关的样本, 这些渲染命令结果可能不会被丢弃。如果 mode 为 GL\_QUERY\_BY\_REGION\_NO\_WAIT, 那么 GL 的操作与 GL\_QUERY\_BY\_REGION\_WAIT 的情况相同, 但是可能会选择无条件执行后续渲染命令, 而不会等待这个查询完成。

#### 注意

glBeginConditionalRender 和 glEndConditionalRender 只在 3.0 或更高版本的 GL 中可用。

GL\_ANY\_SAMPLES\_PASSED 查询结果只在 3.3 或更高版本的 GL 中可用。

#### 错误

如果 id 不是一个已存在查询对象的名称, 则产生 GL\_INVALID\_VALUE 错误。

如果 mode 不是一个可接受的标记, 则产生 GL\_INVALID\_ENUM 错误。

如果在条件渲染激活的情况下调用 glBeginConditionalRender, 或者如果在条件渲染未激活的情况下调用 glEndConditionalRender, 则产生 GL\_INVALID\_OPERATION 错误。

如果 id 是一个目标不是 GL\_SAMPLES\_PASSED 或 GL\_ANY\_SAMPLES\_PASSED 的查询对象的名称, 则产生 GL\_INVALID\_OPERATION 错误。

如果 id 是一个当前正在进行中的查询对象的名称, 则产生 GL\_INVALID\_OPERATION 错误。

#### 另外查看

glGenQueries、glDeleteQueries、glBeginQuery。

#### 版权

Copyright © 2009 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glBeginQuery

限定一个查询对象的边界。

### C 规范

```
void glBeginQuery(GLenum target,
                 GLuint id);
```

#### 参数

target

指定在 glBeginQuery 和相应的 glEndQuery 之间建立查询对象的目标类型。符号常量必须是 GL\_SAMPLES\_PASSED、GL\_ANY\_SAMPLES\_PASSED、GL\_PRIMITIVES\_GENERATED、GL\_TRANSFORM\_FEEDBACK\_PRIMITIVES\_WRITTEN 和 GL\_TIME\_ELAPSED 中的一个。

id

指定查询对象的名称。

### C 规范

```
void glEndQuery(GLenum target);
```

#### 参数

target

指定要推断的查询对象的目标类型。符号常量必须是 GL\_SAMPLES\_PASSED、GL\_ANY\_SAMPLES\_PASSED、GL\_PRIMITIVES\_GENERATED、GL\_TRANSFORM\_FEEDBACK\_PRIMITIVES\_WRITTEN 和 GL\_TIME\_ELAPSED 中的一个。

#### 描述

glBeginQuery 和 glEndQuery 限定查询对象的边界。query 必须为以前的一次 glGenQueries 调用所返回的一个名称。如果一个名为 id 的查询对象还不存在，那么它将被创建，其类型由决定 target。target 必须为 GL\_SAMPLES\_PASSED、GL\_ANY\_SAMPLES\_PASSED、GL\_PRIMITIVES\_GENERATED、GL\_TRANSFORM\_FEEDBACK\_PRIMITIVES\_WRITTEN 和 GL\_TIME\_ELAPSED 中的一个。查询对象的行为由它的类型决定，具体如下。

如果 target 为 GL\_SAMPLES\_PASSED，那么 target 必须是一个未使用的名称，或者是一个已存在的遮挡查询结果的名称。在执行 glBeginQuery 时，查询对象的采样通过计数器将被重置为 0。后续渲染将在每一次进行采样并通过深度测试时使这个计数器递增。如果 GL\_SAMPLE\_BUFFERS 的值为 0，那么采样通过计数器将对每个片段递增 1。如果 GL\_SAMPLE\_BUFFERS 的值为 1，那么采样通过计数器增加的数值为覆盖位进行置位样本的数量。但是，如果片段中的任何样本被覆盖，那么实现使采样通过计数器增加的数值可能为 GL\_SAMPLES 的值，这由实现本身决定。当 glEndQuery 执行时，采样通过计数器将被分配为查询对象结果的值。这个值可以通过调用以 GL\_QUERY\_RESULT 为 pname 的 glGetQueryObject 进行查询。

如果 target 为 GL\_ANY\_SAMPLES\_PASSED，那么 target 必须是一个未使用的名称，或者是一个已存在的遮挡查询结果的名称。当 glEndQuery 执行时，查询对象的采样通过标记将被重置为 GL\_FALSE。如果有任何样本通过深度测试，那么后续的渲染将导致这个标记被设置为 GL\_TRUE。当 glEndQuery 执行时，采样通过标记将被分配为查询对象结果的值。这个值可以通过调用以 GL\_QUERY\_RESULT 为 pname 的 glGetQueryObject 进行查询。

如果 target 为 GL\_PRIMITIVES\_GENERATED，那么 target 必须是一个未使用的名称，或者是一个以前被绑定到 GL\_PRIMITIVES\_GENERATED 查询绑定点的已存在的图元查询对象的名称。

当 glEndQuery 执行时，查询对象的图元生成计数器将被重置为 0。

后续渲染将会为几何图形着色器发出的每个顶点，或者在没有几何图形着色器时由顶点着色器发出的每个顶点使计数器递增一次。当 glEndQuery 执行时，图元生成计数器将被分配为查询对象结果的值。这个值可以通过调用以 GL\_QUERY\_RESULT 为 pname 的 glGetQueryObject 来进行查询。

如果 target 为 GL\_TRANSFORM\_FEEDBACK\_PRIMITIVES\_WRITTEN，那么 target 必须是一个未使用的名称，或者是一个以前被绑定到 GL\_TRANSFORM\_FEEDBACK\_PRIMITIVES\_WRITTEN 查询绑定点的已存在的图元查询结果的名称。在执行 glBeginQuery 时，查询对象的图元写入计数器将被重置为 0。后续渲染将为每一个写入到绑定的变换反馈缓冲区的顶点使这个计数器递增一次。如果变换反馈模式没有在 glBeginQuery 调用和 glEndQuery 调用之间激活，那么计数器将不会递增。当 glEndQuery 执行时，图元写入计数器将被分配为查询对象结果的值。这个值可以通过调用以 GL\_QUERY\_RESULT 为 pname 的 glGetQueryObject 来进行查询。

如果 target 为 GL\_TIME\_ELAPSED，那么 target 必须是一个未使用的名称，或者是一个以前被绑定到 GL\_TIME\_ELAPSED 查询绑定点已存在的计时器查询对象的名称。在执行 glBeginQuery 时，查询对象的时间计数器将被重置为 0。在执行 glEndQuery 时，从调用 glBeginQuery 消耗的服务器时间将被写入到查询对象的时间计数器。这个值可以通过调用以 GL\_QUERY\_RESULT 为 pname 的 glGetQueryObject 进行查询。

对 GL\_QUERY\_RESULT 进行查询将在由这个查询对象限定的渲染完成并且结果可用之前，隐式地对 GL 管线进行清理。可以对 GL\_QUERY\_RESULT\_AVAILABLE 进行查询，以确定结果是否立即可用，或者渲染是否还未完成。

#### 注意

如果通过样本计数超出了可用位数所能表示的最大值（就像以 GL\_QUERY\_COUNTER\_BITS 为 pname 的 glGetQueryiv 报告），计数将成为未定义的。

一个实现在它的通过样本计数器中可能会支持 0 位，在这种情况下查询结果总是未定义的，实际上毫无用处。

当 SAMPLE\_BUFFERS 为 0 时，每个通过深度测试的片段将使通过样本计数器递增 1 次；当 SAMPLE\_BUFFERS 为 1 时，一个实现可以在每有一个通过深度测试的片段样本时递增 1 次，也可以选择在一个片段有任何样本通过深度测试时为这个片段的所有样本增加计数。

查询目标 GL\_ANY\_SAMPLES\_PASSED 和 GL\_TIME\_ELAPSED 只在 3.3 或更高版本的 GL 中可用。

#### 错误

如果 tarGet 不是一个可接受的标记，则产生 GL\_INVALID\_ENUM 错误。

如果 glBeginQuery 在同一个 tarGet 的查询对象已经激活的情况下执行，则产生 GL\_INVALID\_OPERATION 错误。

如果 glEndQuery 在同一个 tarGet 的查询对象没有激活的情况下执行，则产生 GL\_INVALID\_OPERATION 错误。

如果 id 为 0，则产生 GL\_INVALID\_OPERATION 错误。

如果 id 是一个已经激活的查询对象名称，则产生 GL\_INVALID\_OPERATION 错误。

如果 id 引用一个类型与 tarGet 不匹配的已存在查询对象，则产生 GL\_INVALID\_OPERATION 错误。

#### 另外查看

glDeleteQueries、glGenQueries、glGetQueryiv、glGetQueryObject、glsQuery。

**版权**

Copyright © 2005 Addison-Wesley. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

**glBeginTransformFeedback**

开始变换反馈操作

**C 规范**

```
void glBeginTransformFeedback(Glenum primitiveMode);
```

**参数**

primitiveMode

指定将要被记录到为了变换反馈进行绑定的缓冲区对象中图元的输出类型。

**C 规范**

```
void glEndTransformFeedback(void);
```

**描述**

变换反馈模式捕获由顶点着色器（或者是几何图形着色器，如果激活的话）写入的 varying 变量值。在调用 glBeginTransformFeedback 之后，并在后续调用 glEndTransformFeedback 之前，我们称变换反馈是激活的。变换反馈命令必须是成对的。

如果不存在几何图形着色器，那么当变换反馈激活时，glDrawArrays 的 mode 参数必须符合下面列表中指定的值。

如果存在几何图形着色器，那么几何图形着色器的输出图元必须符合下面列表中提供的值。

变换反馈 primitiveMode	允许的几何图形着色器输出图元类型
GL_POINTS	points
GL_LINES	line_strip
GL_TRIANGLES	triangle_strip
变换反馈 primitiveMode	允许的渲染图元 modes
GL_POINTS	GL_POINTS
GL_LINES	GL_LINES, GL_LINE_LOOP, GL_LINE_STRIP, GL_LINES_ADJACENCY, GL_LINE_STRIP_ADJACENCY
GL_TRIANGLES	GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES_ADJACENCY, GL_TRIANGLE_STRIP_ADJACENCY

**注意**

几何图形着色器、GL\_TRIANGLES\_ADJACENCY、GL\_TRIANGLE\_STRIP\_ADJACENCY、GL\_LINES\_ADJACENCY 和 GL\_LINE\_STRIP\_ADJACENCY 图元模式只有在 3.2 或更高版本的 GL 中可用。

**错误**

如果 glBeginTransformFeedback 在变换反馈激活时执行，则产生 GL\_INVALID\_OPERATION 错误。

如果 glEndTransformFeedback 在变换反馈未激活时执行，则产生 GL\_INVALID\_OPERATION 错误。

如果不存在几何图形着色器时激活变换反馈，并且 mode 不是允许模式之一，那么 glDrawArrays 将生成 GL\_INVALID\_OPERATION 错误。

如果在存在几何图形着色器时激活变换反馈，并且几何图形着色器的输出图元类型与变换反馈 primitiveMode 不相符，那么 glDrawArrays 将生成 GL\_INVALID\_OPERATION 错误。

如果变换反馈模式下的任何绑定都没有绑定缓冲区对象，那么 glEndTransformFeedback 将生成 GL\_INVALID\_OPERATION 错误。

如果由于没有激活的程序对象，或者由于激活的程序对象没有指定要记录的 varying 变量而不会使用任何绑定，那么 glEndTransformFeedback 将生成 GL\_INVALID\_OPERATION 错误。

**版权**

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

**glBindAttribLocation**

将通用顶点属性索引与指定的属性变量关联起来。

**C 规范**

```
void glBindAttribLocation(GLuint program,
                          GLuint index,
                          const GLchar *name);
```

**参数**

program

指定发生关联的程序对象的句柄。

index

指定将要绑定的一般顶点属性的索引。

name

指定一个包含 index 将要绑定到的顶点着色器属性变量名称的 null 为结尾的字符串。

**描述**

glBindAttribLocation 用来将 program 指定的程序对象中的一个用户定义的属性变量和一个通用顶点属性索引进行绑定。用户定义的属性变量名称将作为一个在 name 中以 null 为结尾的字符串进行传递。将要绑定到这个变量的通用顶点属性索引由 index 指定。当 program 成为当前状态的一部分，通过通用顶点属性 index 提供的值将对 name 指定的用户定义属性变量进行修改。

如果 name 引用一个矩阵属性变量，那么 index 引用矩阵的第一列。然后矩阵的其他列将为 mat2 类型的矩阵自动绑定到 index + 1 的位置，mat3 类型的矩阵自动绑定到 index + 1 和 index + 2，mat4 类型的矩阵自动绑定到 index + 1、index + 2 和 index + 3。

这个命令使顶点着色器可以为属性变量使用描述性的名称，而不是从 0 到 GL\_MAX\_VERTEX\_ATTRIBS - 1 进行编号的通用变量。发送到每个通用属性索引的值是当前状态的一部分。如果通过调用 glUseProgram 将一个不同的程序对象设为当前的，通用顶点属性将以这样一种方式被跟踪，即同样被绑定到 index 的新程序对象中的属性观察同样的值。

一个程序对象的属性变量和指定为通用的 (name-to-generic) 属性索引的绑定可以在任何时刻通过调用 glBindAttribLocation 来显式地进行分配。属性绑定在调用 glLinkProgram 之后才会生效。在一个程序对象成功连接之后，通用属性的索引值在下一个连接命令出现前将保持固定 (并且这些值能够被查询)。

在程序对象连接之后出现的任何属性绑定在下次程序对象连接之前将不会生效。

**注意**

glBindAttribLocation 可以在任何顶点着色器对象被绑定到指定的程序对象之前被调用。将一个通用属性索引绑定到一个在顶点着色器中从来没用使用过的属性变量也是允许的。

如果 name 以前已经被绑定，那么这个信息将会丢失。这样我们就不能将一个用户定义的属性变量绑定到多个索引上了，但是我们可以将多个用户定义的属性变量绑定到同一个索引上。

允许应用程序将多个用户定义属性变量绑定到同一个通用顶点属性索引。这称为混叠 (aliasing)，只有在可执行程序中只有一个混叠属性是活动状态的情况下，或在没有通过着色器中的路径能消耗混叠到同一区域的一组属性中多于 1 个属性的情况下才被允许。

编译器和连接器被允许假定没有混叠发生，并且可以采用只有在没有混叠的情况下使用的优化。OpenGL 实现不需要进行错误检查来检测混叠。

没有明确绑定的活动属性将在调用 glLinkProgram 时被连接器绑定。分配对区域可以通过调用 glGetAttribLocation 来查询。

OpenGL 在 glBindAttribLocation 被调用时复制 name 字符串，所以应用程序可以在函数返回后立即释放它的 name 字符串副本。

通用属性的位置可以在着色器源文本中使用一个 location 布局限定符来指定。在这种情况下，着色器的源中指定的属性的位置优先，并且可以通过调用 glGetAttribLocation 进行查询。

**错误**

如果 index 大于或等于 GL\_MAX\_VERTEX\_ATTRIBS, 则产生 GL\_INVALID\_VALUE 错误。

如果 name 以保留前缀 "gl\_" 开始, 则产生 GL\_INVALID\_OPERATION 错误。

如果 program 不是由 OpenGL 产生的值, 则产生 GL\_INVALID\_VALUE 错误。

如果 program 不是一个程序对象, 则产生 GL\_INVALID\_OPERATION 错误。

**相关 Get 函数**

glGet, 其自变量为 GL\_MAX\_VERTEX\_ATTRIBS。

glGetActiveAttrib, 其自变量为 program。

glGetAttribLocation, 其自变量为 program 和 name。

glIsProgram

**另外查看**

glDisableVertexAttribArray、glEnableVertexAttribArray、glUseProgram、glVertexAttrib、glVertexAttribPointer。

**版权**

Copyright © 2003–2005 3Dlabs Inc. Ltd. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

**glBindBuffer**

绑定一个指定的缓冲区对象。

**C 规范**

```
void glBindBuffer(GLenum target,
                  GLuint buffer);
```

**参数**

target

定义缓冲区对象要绑定到的目标。符号常量必须是 GL\_ARRAY\_BUFFER、GL\_COPY\_READ\_BUFFER、GL\_COPY\_WRITE\_BUFFER、GL\_ELEMENT\_ARRAY\_BUFFER、GL\_PIXEL\_PACK\_BUFFER、GL\_PIXEL\_UNPACK\_BUFFER、GL\_TEXTURE\_BUFFER、GL\_TRANSFORM\_FEEDBACK\_BUFFER 或 GL\_UNIFORM\_BUFFER。

buffer

指定缓冲区对象的名称。

**描述**

glBindBuffer 将一个缓冲区对象绑定到指定的缓冲区绑定目标。调用一个 glBindBuffer, 其中 target 设置为一个可接受符号常量, 而 buffer 则设置为一个缓冲区对象的名称, 将这个缓冲区名称绑定到目标上。如果不存在名称为 buffer 的缓冲区对象, 则会重新创建一个。当一个缓冲区对象绑定到目标时, 这个目标以前的绑定将自动解除。

缓冲区对象是无符号整数。数值 0 被保留, 但不存在针对每个缓冲区对象目标的默认缓冲区对象。相反, 将 buffer 设置为 0 将高效地将以前绑定的缓冲区对象解除绑定, 并恢复这个缓冲区对象目标的用户机内存使用 (如果支持这个目标的话)。缓冲区对象名称和相应的缓冲区对象内容对当前 GL 渲染环境的共享显示列表空间来说是本地的。两个渲染环境只有在它们也共享显示列表的情况下才共享缓冲区对象。

我们必须使用 glGenBuffers 来生成一组未使用的缓冲区对象名称。

在缓冲区对象第一次绑定时, 它的即时状态是一个没有映射的大小为 0 的内存缓冲区, 可以进行 GL\_READ\_WRITE 访问和使用 STATIC\_DRAW。

当一个非 0 缓冲区对象名称被绑定时, GL 将对它绑定到的目标所进行的操作会影响绑定的缓冲区对象, 而对它绑定到的目标进行的查询将返回从绑定的缓冲区对象得到的状态。当缓冲区对象名称 0 被绑定时 (就像初始状态一样), 尝试修改或查询它绑定到的目标的状态会产生一个 GL\_INVALID\_OPERATION 错误。

当一个非 0 缓冲区对象被绑定到 GL\_ARRAY\_BUFFER 目标时, 传统上被认为是到客户端内存的指针的顶点数组索引参数将被解释为缓冲区对象中的一个以基本机器单元来测量的偏移。

当一个非 0 缓冲区对象被绑定到 GL\_ELEMENT\_ARRAY\_BUFFER 目标时, glDrawElements、glDrawElementsInstanced、glDrawElementsBaseVertex、glDrawRangeElements、glDrawRangeElementsBaseVertex、glMultiDrawElements 或 glMultiDrawElementsBaseVertex 索引参数将被解释为缓冲区对象中的一个以基本机器单元来测量的偏移。

当一个非 0 缓冲区对象被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标时,如下命令将受到影响: glGetCompressedTexImage、glGetTexImage 和 glReadPixels。指针参数将被解释为缓冲区对象中的一个以基本机器单元来测量的偏移。

当一个非 0 缓冲区对象被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标时,如下命令将受到影响: glCompressedTexImage1D、glCompressedTexImage2D、glCompressedTexImage3D、glCompressedTexSubImage1D、glCompressedTexSubImage2D、glCompressedTexSubImage3D、glTexImage1D、glTexImage2D、glTexImage3D、glTexSubImage1D、glTexSubImage2D 和 glTexSubImage3D。指针参数将被解释为缓冲区对象中的一个以基本机器单元来测量的偏移。

缓冲区对象 GL\_COPY\_READ\_BUFFER 和 GL\_COPY\_WRITE\_BUFFER 是我们可以使用 glCopyBufferSubData 而不会影响其他绑定的状态。但是, glCopyBufferSubData 可以用于任意缓冲区绑定点对。

GL\_TRANSFORM\_FEEDBACK\_BUFFER 缓冲区绑定点可以传递到 glBindBuffer,但是不会直接影响变换反馈状态。相反,索引 GL\_TRANSFORM\_FEEDBACK\_BUFFER 绑定必须通过调用 glBindBufferBase 或 glBindBufferRange 来使用。这样就会影响通用 GL\_TRANSFORM\_FEEDBACK\_BUFFER 绑定。

类似地,可以使用 GL\_UNIFORM\_BUFFER 缓冲区绑定点,但是不会直接影响统一缓冲区绑定点。

通过 glBindBuffer 创建的缓冲区对象绑定会保持活动状态,直到一个不同的缓冲区对象名称被绑定到同一个目标,或者直到绑定的缓冲区对象通过 glDeleteBuffers 删除。

一旦一个命名的缓冲区对象被创建,它就可以在任何需要的时候被重新绑定到任何目标了。但是, GL 实现可能会根据其初始绑定目标来选择如何对一个缓冲区对象的存储进行优化。

#### 注意

GL\_COPY\_READ\_BUFFER、GL\_UNIFORM\_BUFFER 和 GL\_TEXTURE\_BUFFER 目标只在 3.1 或更高版本的 GL 中可用。

#### 错误

如果 target 不是一个允许值,则产生 GL\_INVALID\_ENUM 错误。

如果 buffer 不是以前的一个 glGenBuffers 调用返回的名称,则产生 GL\_INVALID\_VALUE 错误。

#### 相关 Get 函数

glGet, 其自变量为 GL\_ARRAY\_BUFFER\_BINDING。

glGet, 其自变量为 GL\_COPY\_READ\_BUFFER\_BINDING。

glGet, 其自变量为 GL\_COPY\_WRITE\_BUFFER\_BINDING。

glGet, 其自变量为 GL\_ELEMENT\_ARRAY\_BUFFER\_BINDING。

glGet, 其自变量为 GL\_PIXEL\_PACK\_BUFFER\_BINDING。

glGet, 其自变量为 GL\_PIXEL\_UNPACK\_BUFFER\_BINDING。

glGet, 其自变量为 GL\_TRANSFORM\_FEEDBACK\_BUFFER\_BINDING。

glGet, 其自变量为 GL\_UNIFORM\_BUFFER\_BINDING。

#### 另外查看

glGenBuffers、glBindBufferBase、glBindBufferRange、glMapBuffer、glUnmapBuffer、glDeleteBuffers、glGet、glIsBuffer。

#### 版权

Copyright © 2005 Addison-Wesley。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glBindBufferBase

将一个缓冲区对象绑定到一个索引缓冲区目标。

#### C 规范

```
void glBindBufferBase(GLenum target,
                     GLuint index,
                     GLuint buffer);
```

#### 参数

target

指定绑定操作的目标。target 必须为 GL\_TRANSFORM\_FEEDBACK\_BUFFER 或 GL\_UNIFORM\_BUFFER。

index

指定由 target 指定的数组中绑定点的索引。

buffer

一个绑定到指定的缓冲区绑定点的缓冲区对象的名称。

描述

glBindBufferBase 将缓冲区对象 buffer 绑定到由 target 指定的目标数组中以 ndex 为索引的绑定点。每个 target 表示一个缓冲区绑定点索引数组,也表示能够被其他缓冲区操作函数(例如 glBindBuffer 或 glMapBuffer)使用单个通用绑定点。除了将 buffer 绑定到索引缓冲区绑定目标之外,glBindBufferBase 也会将 buffer 绑定到由 target 指定的通用缓冲区绑定点。

注意

glBindBufferBase 只在 3.0 或更高版本的 GL 中可用。

调用 glBindBufferBase 相当于调用将 offset 设为 0,而将 size 设为缓冲区大小的 glBindBufferRange。

错误

如果 tarGet 不是 GL\_TRANSFORM\_FEEDBACK\_BUFFER 或 GL\_UNIFORM\_BUFFER,则产生 GL\_INVALID\_ENUM 错误。

如果 index 大于或等于 tarGet 指定的索引绑定点数量,则产生 GL\_INVALID\_VALUE 错误。

另外查看

glGenBuffers、glDeleteBuffers、glBindBuffer、glBindBufferRange、glMapBuffer、glUnmapBuffer。

版权

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glBindBufferRange

将一个缓冲区对象中的一个范围绑定到一个索引缓冲区目标。

**C 规范**

```
void glBindBufferRange(GLenum target,
                      GLuint index,
                      GLuint buffer,
                      GLintptr offset,
                      GLsizeiptr size);
```

参数

target

指定绑定操作的目标。target 必须为 GL\_TRANSFORM\_FEEDBACK\_BUFFER 或 GL\_UNIFORM\_BUFFER。

index

指定由 target 指定的数组中绑定点的索引。

buffer

一个绑定到指定的缓冲区绑定点的缓冲区对象的名称。

offset

缓冲区对象 buffer 中以基本机器单元为单位的初始偏置。

size

能够从作为索引目标使用的缓冲区对象中读取的机器单元数量。

描述

glBindBufferBase 将缓冲区对象 buffer 中的一个由 offset 和 size 表示的范围绑定到由 target 指定的目标数组中以 ndex 为索引的绑定点。每个 target 表示一个缓冲区绑定点索引数组,也表示能够被其他缓冲区操作函数(例如 glBindBuffer 或 glMapBuffer)使用单个通用绑定点。除了将 buffer 中的一个范围绑定到索引缓冲区绑定目标之外,glBindBufferBase 也会将这个范围绑定到由 target 指定的通用缓冲区绑定点。

offset 是指缓冲区对象 buffer 中以基本机器单元数量为单位的偏置,而 size 则是指能够从作为索引目标使用的读取的数据量。

错误

如果 `target` 不是 `GL_TRANSFORM_FEEDBACK_BUFFER` 或 `GL_UNIFORM_BUFFER`, 则产生 `GL_INVALID_ENUM` 错误。

如果 `index` 大于或等于 `target` 指定的索引绑定点数量, 则产生 `GL_INVALID_VALUE` 错误。

如果 `index` 大于或等于 0, 或者如果 `offset + size` 大于 `GL_BUFFER_SIZE` 的值, 则产生 `GL_INVALID_VALUE` 错误。

如果 `offset` 违反了任何由 `target` 指定的排列限制, 则可能产生额外错误。

#### 另外查看

`glGenBuffers`、`glDeleteBuffers`、`glBindBuffer`、`glBindBufferBase`、`glMapBuffer`、`glUnmapBuffer`。

#### 版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glBindFragDataLocation

将一个用户定义的 `varying` 输出变量绑定到一个片段着色器颜色号。

### C 规范

```
void glBindFragDataLocation(GLuint program,
                           GLuint colorNumber,
                           const char * name);
```

#### 参数

`program`

包含将要修改绑定的 `varying` 输出变量的程序的名称。

`colorNumber`

要将一个用户定义的 `varying` 输出变量绑定到的颜色号。

`name`

用户定义的将要修改绑定的 `varying` 输出变量的名称。

#### 描述

`glBindFragDataLocation` 显式地为 `program` 程序指定用户定义的 `varying` 输出变量 `name` 到片段着色器颜色号 `colorNumber` 的绑定。如果 `name` 已经进行过绑定, 那么它所分配的绑定将由 `colorNumber` 代替。`name` 必须是一个以 `null` 为结尾的字符串。

`colorNumber` 必须小于 `GL_MAX_DRAW_BUFFERS`。

由 `glBindFragDataLocation` 指定的绑定在 `program` 进行下一次连接之前不会产生任何效果。在创建 `program` 之后, 可以随时指定绑定。特别是, 它们可以在着色器对象被绑定到这个程序之前进行指定。这样, 在 `name` 中可以指定任何名称, 包括那些从没有在任何片段着色器对象中作为 `varying` 输出变量使用的名称。

GL 保留了以 `gl_` 为开头的名称。

除了 `glBindFragDataLocation` 生成的错误之外, 出现下列情况时 `program` 程序将会连接失败:

活动输出的数量大于 `GL_MAX_DRAW_BUFFERS` 的值。

有一个以上的 `varying` 输出变量被绑定到同一个颜色号。

#### 注意

`varying` 输出变量可能会在着色器文本中使用一个 `location` 布局限定符来显式地分配索引位置。如果一个着色器在着色器文本中将一个位置静态地分配给一个 `varying` 输出变量, 那么这个位置将会被使用, 而任何通过 `glBindFragDataLocation` 分配的位置都将被忽略。

#### 错误

如果 `colorNumber` 大于或等于 `GL_MAX_DRAW_BUFFERS`, 则产生 `GL_INVALID_VALUE` 错误。

如果 `name` 以保留前缀 “`gl_`” 开始, 则产生 `GL_INVALID_OPERATION` 错误。

如果 `program` 不是一个程序对象的名称, 则产生 `GL_INVALID_OPERATION` 错误。

#### 相关 Get 函数

`glGetFragDataLocation`, 包含有效的程序对象和一个用户定义的 `varying` 输出变量的名称。

#### 另外查看

`glCreateProgram`、`glGetFragDataLocation`。

**版权**

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

**glBindFragDataLocationIndexed**

将一个用户定义的 varying 输出变量绑定到一个片段着色器颜色号和索引上。

**C 规范**

```
void glBindFragDataLocationIndexed(GLuint program,
                                   GLuint colorNumber,
                                   GLuint index,
                                   const char * name);
```

**参数**

program

包含将要修改绑定的 varying 输出变量的程序的名称。

colorNumber

要将一个用户定义的 varying 输出变量绑定到的颜色号。

index

要将一个用户定义的 varying 输出变量绑定到的颜色输入索引。

name

用户定义的将要修改绑定的 varying 输出变量的名称。

**描述**

glBindFragDataLocationIndexed 指定, program 中的 varying 输出变量 name 应该在这个程序进行下一次连接时被绑定到片段颜色 colorNumber。

由 glBindFragDataLocationIndexed 指定的绑定在 program 进行下一次连接之前不会产生任何效果。在创建 program 之后, 可以随时指定绑定。特别是, 它们可以在着色器对象被绑定到这个程序之前进行指定。这样, 在 name 中可以指定任何名称, 包括那些从没有在任何片段着色器对象中作为 varying 输出变量使用的名称。GL 保留了以 gl\_ 为开头的名称。

如果 name 已经进行过绑定, 那么它所分配的绑定将由 colorNumber 和 index 代替。name 必须是一个以 null 为结尾的字符串。index 必须小于或等于 1, 并且当 index 为 0 时 colorNumber 必须小于 GL\_MAX\_DRAW\_BUFFERS 的值, 而当 index 大于或等于 1 时 colorNumber 则必须小于 GL\_MAX\_DUAL\_SOURCE\_DRAW\_BUFFERS 的值。

除了 glBindFragDataLocationIndexed 生成的错误之外, 出现下列情况时 program 程序将会连接失败:

活动输出的数量大于 GL\_MAX\_DRAW\_BUFFERS 的值。

有一个以上的 varying 输出变量被绑定到同一个颜色号。

**注意**

varying 输出变量可能会在着色器文本中使用一个 location 布局限定符来显式地分配位置。如果一个着色器在着色器文本中将一个位置静态地分配给一个 varying 输出变量, 那么这个位置将会被使用, 而任何通过 glBindFragDataLocation 分配的位置都将被忽略。

**错误**

如果 colorNumber 大于或等于 GL\_MAX\_DRAW\_BUFFERS, 则产生 GL\_INVALID\_VALUE 错误。

如果 colorNumber 大于或等于 GL\_MAX\_DUAL\_SOURCE\_DRAW\_BUFFERS, 并且 index 大于或等于 1, 则产生 GL\_INVALID\_VALUE 错误。

如果 index 大于 1, 则产生 GL\_INVALID\_VALUE 错误。

如果 name 以保留前缀 "gl\_" 开始, 则产生 GL\_INVALID\_OPERATION 错误。

如果 program 不是一个程序对象的名称, 则产生 GL\_INVALID\_OPERATION 错误。

**相关 Get 函数**

glGetFragDataLocation, 包含有效的程序对象和一个用户定义的 varying 输出变量的名称。

glGetFragDataIndex, 包含有效的程序对象和一个用户定义的 varying 输出变量的名称。

**另外查看**

glCreateProgram、glLinkProgram、glGetFragDataLocation、glGetFragDataIndex、glBindFragDataLocation。

## 版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glBindFramebuffer

将一个帧缓冲区绑定到一个帧缓冲区目标上。

### C 规范

```
void glBindFramebuffer(GLenum target,
                       GLuint framebuffer);
```

### 参数

target

指定绑定操作的帧缓冲区目标。

framebuffer

指定将要进行绑定的帧缓冲区对象的名称。

### 描述

glBindFramebuffer 会将名称为 framebuffer 的帧缓冲区对象绑定到 target 指定的帧缓冲区目标。target 必须为 GL\_DRAW\_FRAMEBUFFER、GL\_READ\_FRAMEBUFFER 或 GL\_FRAMEBUFFER。如果一个帧缓冲区被绑定到 GL\_DRAW\_FRAMEBUFFER 或 GL\_READ\_FRAMEBUFFER 上,那么在这两种情况下它将分别成为渲染或回读操作的目标,直到它被删除,或者其他帧缓冲区被绑定到相应的绑定目标。调用 target 被设置为 GL\_FRAMEBUFFER 的 glBindFramebuffer,会将 framebuffer 绑定到读取缓冲区和绘制缓冲区。framebuffer 是之前的一个 glGenFramebuffers 调用返回的帧缓冲区对象的名称,或者也可以设为 0 来解除已经存在的从一个帧缓冲区对象到 target 的绑定。

### 错误

如果 target 不是 GL\_DRAW\_FRAMEBUFFER、GL\_READ\_FRAMEBUFFER 或 GL\_FRAMEBUFFER,则产生 GL\_INVALID\_ENUM 错误。

如果 framebuffer 不是 0 或以前的一次调用所返回的帧缓冲区的名称,则产生 GL\_INVALID\_OPERATION 错误。

### 另外查看

glGenFramebuffers、glDeleteFramebuffers、glFramebufferRenderbuffer、glFramebufferTexture、glFramebufferTexture1D、glFramebufferTexture2D、glFramebufferTexture3D、glFramebufferTextureFace、glFramebufferTextureLayer、glIsFramebuffer。

### 版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glBindRenderbuffer

将一个渲染缓冲区绑定到一个渲染缓冲区目标上。

### C 规范

```
void glBindRenderbuffer(GLenum target,
                       GLuint renderbuffer);
```

### 参数

target

指定绑定操作的渲染缓冲区目标。target 必须为 GL\_RENDERBUFFER。

renderbuffer

指定将要进行绑定的渲染缓冲区对象的名称。

### 描述

glBindRenderbuffer 将名称为 renderbuffer 的帧缓冲区对象绑定到 target 指定的渲染缓冲区目标上。

target 必须为 GL\_RENDERBUFFER。

renderbuffer 是之前的一个 glGenRenderbuffers 调用所返回的渲染缓冲区对象的名称,或者也可以设为 0 来解

除已经存在的从一个渲染缓冲区对象到 target 的绑定。

#### 错误

如果 tarGet 不是 GL\_RENDERBUFFER, 则产生 GL\_INVALID\_ENUM 错误。

如果 renderbuffer 不是 0 或以前的一次 glGenRenderbuffers 调用所返回的渲染缓冲区的名称, 则产生 GL\_INVALID\_OPERATION 错误。

#### 另外查看

glGenRenderbuffers、glDeleteRenderbuffers、glRenderbufferStorage、glRenderbufferStorageMultisample、glIsRenderbuffer。

#### 版权

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glBindSampler

将一个指定的采样器绑定到一个纹理目标上。

### C 规范

```
void glBindSampler(GLuint unit,
                  GLuint texture);
```

#### 参数

unit

指定采样器将要绑定到的纹理单元的索引。

sampler

指定采样器的名称。

#### 描述

glBindSampler 将 sampler 绑定到纹理单元的索引 unit。

sampler 必须是 0 或以前的一次 glGenSamplers 调用所返回的采样器的名称。

unit 必须小于 GL\_MAX\_COMBINED\_TEXTURE\_IMAGE\_UNITS 的值。

当一个采样器对象被绑定到一个纹理单元时, 它的状态会取代绑定到这个纹理单元的纹理对象的状态。如果这个采样器名称 0 被绑定到一个纹理单元, 那么当前绑定纹理的采样器状态将被激活。单个采样器对象可以同时绑定到多个纹理单元。

#### 注意

glBindSampler 只在 3.3 或更高版本的 GL 中可用。

#### 错误

如果 unit 大于或等于 GL\_MAX\_COMBINED\_TEXTURE\_IMAGE\_UNITS 的值, 则产生 GL\_INVALID\_VALUE 错误。

如果 sampler 不是 0 或以前一次调用所返回的名称, 或者这个名称已经通过调用 glDeleteSamplers 删除了, 则产生 GL\_INVALID\_OPERATION 错误。

#### 相关 Get 函数

glGet, 其自变量为 GL\_SAMPLER\_BINDING。

#### 另外查看

glGenSamplers、glDeleteSamplers、glGet、glSamplerParameter、glGetSamplerParameter、glGenTextures、glBindTexture、glDeleteTextures。

#### 版权

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glBindTexture

将一个指定的纹理绑定到一个纹理目标上。

### C 规范

```
void glBindTexture(GLenum target,
                  GLuint texture);
```

**参数**

target

定义纹理要绑定到的目标。必须为 GL\_TEXTURE\_1D、GL\_TEXTURE\_2D、GL\_TEXTURE\_3D、GL\_TEXTURE\_1D\_ARRAY、GL\_TEXTURE\_2D\_ARRAY、GL\_TEXTURE\_RECTANGLE、GL\_TEXTURE\_CUBE\_MAP、GL\_TEXTURE\_2D\_MULTISAMPLE 或 GL\_TEXTURE\_2D\_MULTISAMPLE\_ARRAY。

纹理贴图

指定纹理的名称。

描述

glBindTexture 使我们可以创建或使用一个已命名的纹理。调用 glBindTexture, target 设置为 GL\_TEXTURE\_1D、GL\_TEXTURE\_2D、GL\_TEXTURE\_3D、GL\_TEXTURE\_1D\_ARRAY、GL\_TEXTURE\_2D\_ARRAY、GL\_TEXTURE\_RECTANGLE、GL\_TEXTURE\_CUBE\_MAP、GL\_TEXTURE\_2D\_MULTISAMPLE 或 GL\_TEXTURE\_2D\_MULTISAMPLE\_ARRAY, texture 设置为将纹理名称绑定到目标的新纹理的名称。当一个纹理绑定到目标时, 这个目标以前的绑定将自动解除。

纹理是无符号整数。数值 0 被保留来表示每个纹理目标的默认纹理。纹理名称和相应的纹理内容对当前 GL 渲染环境的共享对象空间来说是本地的。两个渲染环境只有在它们也共享显示列表的情况下才共享纹理名称。

我们必须使用 glGenTextures 来生成一组新的纹理名称。

当一个纹理被第一次绑定时, 它采用指定的目标: 第一次绑定到 GL\_TEXTURE\_1D 的纹理成为一维纹理, 第一次绑定到 GL\_TEXTURE\_2D 的纹理成为二维纹理, 第一次绑定到 GL\_TEXTURE\_3D 的纹理成为三维纹理, 第一次绑定到 GL\_TEXTURE\_1D\_ARRAY 的纹理成为一维数组纹理, 第一次绑定到 GL\_TEXTURE\_2D\_ARRAY 的纹理成为二维数组纹理, 第一次绑定到 GL\_TEXTURE\_RECTANGLE 的纹理成为矩形纹理, 第一次绑定到 GL\_TEXTURE\_CUBE\_MAP 的纹理成为立方体贴图纹理, 第一次绑定到 GL\_TEXTURE\_2D\_MULTISAMPLE 的纹理成为二维多重采样纹理, 而第一次绑定到 GL\_TEXTURE\_2D\_MULTISAMPLE\_ARRAY 的纹理成为二维多重采样数组纹理。一维纹理在它第一次绑定后的即时状态与 GL 初始化时默认 GL\_TEXTURE\_1D 的状态相同, 对于其他纹理类型来说情况也是类似情况。

在一个纹理被绑定时, GL 将对它绑定到的目标所进行的操作会影响绑定的纹理, 而对它绑定到的目标进行的查询将返回从绑定的纹理对象得到的状态。实际上, 纹理目标由于当前绑定到它们的纹理而变得混叠, 并且纹理名称 0 会用在初始化时绑定到它们的默认纹理。

一个由 glBindTexture 创建的纹理绑定 (texture binding) 在一个不同的纹理名称被绑定到同一个目标之前, 或在绑定的纹理被 glDeleteTextures 删除之前会保持活动状态。

一旦创建了一个命名的纹理, 在有需要的时候它就可以被重新绑定到同一个原始目标上。使用 glBindTexture 将一个已经存在的指定名称的纹理绑定到一个纹理目标上, 通常比使用 glTexImage1D、glTexImage2D 或 glTexImage3D 来重新载入纹理图像要快得多。

**注意**

GL\_TEXTURE\_2D\_MULTISAMPLE 和 GL\_TEXTURE\_2D\_MULTISAMPLE\_ARRAY 目标只在 3.2 或更高版本的 GL 中可用。

**错误**

如果 tarGet 不是一个允许值, 则产生 GL\_INVALID\_ENUM 错误。

如果 tarGet 不是以前的一个 glGenTextures 调用返回的名称, 则产生 GL\_INVALID\_VALUE 错误。

如果 texture 是以前创建的, 其目标与 tarGet 的目标不匹配, 则产生 GL\_INVALID\_OPERATION 错误。

**相关 Get 函数**

glGet 其自变量为 GL\_TEXTURE\_BINDING\_1D、GL\_TEXTURE\_BINDING\_2D、GL\_TEXTURE\_BINDING\_3D、GL\_TEXTURE\_BINDING\_1D\_ARRAY、GL\_TEXTURE\_BINDING\_2D\_ARRAY、GL\_TEXTURE\_BINDING\_RECTANGLE、GL\_TEXTURE\_BINDING\_2D\_MULTISAMPLE 或 GL\_TEXTURE\_BINDING\_2D\_MULTISAMPLE\_ARRAY。

**另外查看**

glDeleteTextures、glGenTextures、glGet、glGetTexParameter、glIsTexture、glTexImage1D、glTexImage2D、glTexImage2DMultisample、glTexImage3D、glTexImage3DMultisample、glTexParameter。

**版权**

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见

<http://oss.sgi.com/projects/FreeB/>。

## glBindVertexArray

绑定一个顶点数组对象。

### C 规范

```
void glBindVertexArray(GLuint array);
```

### 参数

array

指定将要进行绑定的顶点数组的名称。

### 描述

glBindVertexArray 绑定名为 array 的顶点数组对象。array 是以前的一个 glGenVertexArrays 调用所返回的顶点数组对象的名称，或者也可以设为 0 来解除已经存在的顶点数组对象绑定。

如果不存在名称为 buffer 的顶点数组对象，则会在数组第一次进行绑定时重新创建一个。如果绑定成功，那么将不会对这个顶点对象的状态进行任何改变，并且将解除以前的任何顶点数组对象绑定。

### 错误

如果 array 不是 0 或以前的一次调用 glGenVertexArrays 所返回的顶点数组对象的名称，则产生 GL\_INVALID\_OPERATION 错误。

### 另外查看

glGenVertexArrays、glDeleteVertexArrays、glVertexAttribPointer、glEnableVertexAttribArray。

### 版权

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glBlendColor

设置混合颜色。

### C 规范

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

### 参数

red

green

blue

alpha

指定 GL\_BLEND\_COLOR 的分量。

### 描述

GL\_BLEND\_COLOR 可以用来计算源和目标混合因子。在被存储之前，颜色分量被截取到[0,1]范围内。关于混合操作的详细描述请参见 glBlendFunc。初始状态下 GL\_BLEND\_COLOR 被设置为(0, 0, 0, 0)。

### 相关 Get 函数

glGet，其中一个自变量为 GL\_BLEND\_COLOR。

### 另外查看

glBlendEquation、glBlendFunc、glGetString。

### 版权

Copyright © 1991–2006 Silicon Graphics, Inc。本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glBlendEquation

定义 RGB 混合方程和 Alpha 混合方程都要使用的方程。

### C 规范

```
void glBlendEquation(GLenum mode);
```

### 参数

mode

指定源和目标颜色如何进行组合。它必须是 GL\_FUNC\_ADD、GL\_FUNC\_SUBTRACT、GL\_FUNC\_REVERSE\_SUBTRACT、GL\_MIN 或 GL\_MAX。

### 描述

混合方程决定一个新的像素（“源”颜色）如何与一个已经存在于帧缓冲区的像素（“目的”颜色）进行结合。这个函数将 RGB 混合方程和 Alpha 混合方程设置为一个单个方程。

这些方程使用由 glBlendFunc 或 glBlendFuncSeparate 指定的源和目的混合因子。关于各种混合因子的描述参见 glBlendFunc 或 glBlendFuncSeparate。

模 式	RGB 分量	Alpha 分量
GL_FUNC_ADD	$R_r = R_s S_R + R_d d_R$ $G_r = G_s S_G + G_d d_G$ $B_r = B_s S_B + B_d d_B$	$A_r = A_s S_A + A_d d_A$
GL_FUNC_SUBTRACT	$R_r = R_s S_R - R_d d_R$ $G_r = G_s S_G - G_d d_G$ $B_r = B_s S_B - B_d d_B$	$A_r = A_s S_A - A_d d_A$
GL_FUNC_REVERSE_SUBTRACT	$R_r = R_d d_R - R_s S_R$ $G_r = G_d d_G - G_s S_G$ $B_r = B_d d_B - B_s S_B$	$A_r = A_d d_A - A_s S_A$
GL_MIN	$R_r = \min(R_s, R_d)$ $G_r = \min(G_s, G_d)$ $B_r = \min(B_s, B_d)$	$A_r = \min(A_s, A_d)$
GL_MAX	$R_r = \max(R_s, R_d)$ $G_r = \max(G_s, G_d)$ $B_r = \max(B_s, B_d)$	$A_r = \max(A_s, A_d)$

在后面的方程中，源和目标颜色分量分别表示为  $(R_s, G_s, B_s, A_s)$  和  $(R_d, G_d, B_d, A_d)$ 。结果得到的颜色用  $(R_r, G_r, B_r, A_r)$  表示。源和结果混合因子分别表示为  $(S_R, S_G, S_B, S_A)$  和  $(d_R, d_G, d_B, d_A)$ 。在这些方程中，规定所有颜色分量的值都在  $[0, 1]$  范围内。

这些方程的结果被截取到  $[0, 1]$  范围内。

GL\_MIN 和 GL\_MAX 方程对于分析图像数据（例如阈值为一个常量颜色的图像）的应用程序来说是非常有用的。GL\_FUNC\_ADD 方程对反锯齿和透明度等都非常有用。

在初始状态下，RGB 混合方程和 Alpha 混合方程都被设置为 GL\_FUNC\_ADD。

### 注意

GL\_MIN 和 GL\_MAX 方程不使用源或目的因子，只使用源和目的颜色。

### 错误

如果 mode 不是 GL\_FUNC\_ADD、GL\_FUNC\_SUBTRACT、GL\_FUNC\_REVERSE\_SUBTRACT、GL\_MAX 或 GL\_MIN 中的一个，则产生 GL\_INVALID\_ENUM 错误。

### 相关 Get 函数

glGet, 其中一个自变量为 GL\_BLEND\_EQUATION\_RGB。

glGet, 其中一个自变量为 GL\_BLEND\_EQUATION\_ALPHA。

另外查看

glBlendColor、glBlendFunc、glBlendFuncSeparate。

版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glBlendEquationSeparate

分别设置 RGB 混合方程和 Alpha 混合方程。

**C 规范**

```
void glBlendEquationSeparate(GLenum modeRGB,
                             GLenum modeAlpha);
```

**参数**

modeRGB

指定 RGB 混合方程, 源和目的颜色的红、绿和蓝分量如何结合。它必须是 GL\_FUNC\_ADD、GL\_FUNC\_SUBTRACT、GL\_FUNC\_REVERSE\_SUBTRACT、GL\_MIN 或 GL\_MAX。

modeAlpha

指定 Alpha 混合方程, 源和目的颜色的 Alpha 分量如何结合。它必须是 GL\_FUNC\_ADD、GL\_FUNC\_SUBTRACT、GL\_FUNC\_REVERSE\_SUBTRACT、GL\_MIN 或 GL\_MAX。

**描述**

混合方程决定一个新的像素 (“源” 颜色) 如何与一个已经存在于帧缓冲区的像素 (“目的” 颜色) 进行结合。这个函数为 RGB 颜色分量指定一个混合方程, 同时为 Alpha 分量指定一个方程。

这些方程使用由 glBlendFunc 或 glBlendFuncSeparate 指定的源和目的混合因子。关于各种混合因子的描述参见 glBlendFunc 或 glBlendFuncSeparate。

模 式	RGB 分量	Alpha 分量
GL_FUNC_ADD	$R_r = R_s S_r + R_d d_r$ $G_r = G_s S_g + G_d d_g$ $B_r = B_s S_b + B_d d_b$	$A_r = A_s S_a + A_d d_a$
GL_FUNC_SUBTRACT	$R_r = R_s S_r - R_d d_r$ $G_r = G_s S_g - G_d d_g$ $B_r = B_s S_b - B_d d_b$	$A_r = A_s S_a - A_d d_a$
GL_FUNC_REVERSE_SUBTRACT	$R_r = R_d d_r - R_s S_r$ $G_r = G_d d_g - G_s S_g$ $B_r = B_d d_b - B_s S_b$	$A_r = A_d d_a - A_s S_a$
GL_MIN	$R_r = \min(R_s, R_d)$ $G_r = \min(G_s, G_d)$ $B_r = \min(B_s, B_d)$	$A_r = \min(A_s, A_d)$
GL_MAX	$R_r = \max(R_s, R_d)$ $G_r = \max(G_s, G_d)$ $B_r = \max(B_s, B_d)$	$A_r = \max(A_s, A_d)$

在后面的方程中, 源和目标颜色分量分别表示为  $(R_s, G_s, B_s, A_s)$  和  $(R_d, G_d, B_d, A_d)$ 。结果得到的颜色用  $(R_r, G_r, B_r, A_r)$  表示。源和结果混合因子分别表示为  $(s_r, s_g, s_b, s_a)$  和  $(d_r, d_g, d_b, d_a)$ 。在这些方程中, 规定所有颜色分量的值都在  $[0, 1]$  范围内。

这些方程的结果被截取到  $[0, 1]$  范围内。

GL\_MIN 和 GL\_MAX 方程对于分析图像数据 (例如阈值为一个常量颜色的图像) 的应用程序来说是非常有用的。GL\_FUNC\_ADD 方程对反锯齿和透明度等都非常有用。

在初始状态下, RGB 混合方程和 Alpha 混合方程都被设置为 GL\_FUNC\_ADD。

#### 注意

GL\_MIN 和 GL\_MAX 方程不使用源或目的因子，只使用源和目的颜色。

#### 错误

如果 modeRGB 或 modeAlpha 不是 GL\_FUNC\_ADD、GL\_FUNC\_SUBTRACT、GL\_FUNC\_REVERSE\_SUBTRACT、GL\_MAX 或 GL\_MIN 中的一个，则产生 GL\_INVALID\_ENUM 错误。

#### 相关 Get 函数

glGet，其中一个自变量为 GL\_BLEND\_EQUATION\_RGB。

glGet，其中一个自变量为 GL\_BLEND\_EQUATION\_ALPHA。

#### 另外查看

glGetString、glBlendColor、glBlendFunc、glBlendFuncSeparate。

#### 版权

Copyright © 2006 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

### glBlendFunc

定义像素算法。

#### C 规范

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

#### 参数

sfactor

指定红、绿、蓝和 Alpha 源混合因子如何计算。初始值为 GL\_ONE。

dfactor

指定红、绿、蓝和 Alpha 目标混合因子如何计算。

下列符号常量都可以接受：GL\_ZERO、GL\_ONE、GL\_SRC\_COLOR、GL\_ONE\_MINUS\_SRC\_COLOR、GL\_DST\_COLOR、GL\_ONE\_MINUS\_DST\_COLOR、GL\_SRC\_ALPHA、GL\_ONE\_MINUS\_SRC\_ALPHA、GL\_DST\_ALPHA、GL\_ONE\_MINUS\_DST\_ALPHA、GL\_CONSTANT\_COLOR、GL\_ONE\_MINUS\_CONSTANT\_COLOR、GL\_CONSTANT\_ALPHA 和 GL\_ONE\_MINUS\_CONSTANT\_ALPHA。初始值为 GL\_ZERO。

#### 描述

像素能够使用一个将输入（源）RGBA 值和已经存在于帧缓冲区（目的）的值进行混合的方程来进行绘制。初始状态下是不支持混合的。使用自变量为 GL\_BLEND 的 glEnable 和 glDisable 来激活和关闭混合。

glBlendFunc 在启用时定义混合操作。sfactor 指定使用哪种方法来缩放源颜色分量。dfactor 指定使用哪种方法来缩放目的颜色分量。这两个参数都必须是下列符号常量中的一个：GL\_ZERO、GL\_ONE、GL\_SRC\_COLOR、GL\_ONE\_MINUS\_SRC\_COLOR、GL\_DST\_COLOR、GL\_ONE\_MINUS\_DST\_COLOR、GL\_SRC\_ALPHA、GL\_ONE\_MINUS\_SRC\_ALPHA、GL\_DST\_ALPHA、GL\_ONE\_MINUS\_DST\_ALPHA、GL\_CONSTANT\_COLOR、GL\_ONE\_MINUS\_CONSTANT\_COLOR、GL\_CONSTANT\_ALPHA、GL\_ONE\_MINUS\_CONSTANT\_ALPHA、GL\_SRC\_ALPHA\_SATURATE、GL\_SRC1\_COLOR、GL\_ONE\_MINUS\_SRC1\_COLOR、GL\_SRC1\_ALPHA 和 GL\_ONE\_MINUS\_SRC1\_ALPHA。下表中描述了可能的的方法。每种方法定义了 4 个比例因子，红、绿、蓝和 Alpha 各对应一个因子。在表中和后面的方程中，第一个源、第二个源和目的颜色分量分别由  $(R_{s0}, G_{s0}, B_{s0}, A_{s0})$ 、 $(R_{s1}, G_{s1}, B_{s1}, A_{s1})$  和  $(R_d, G_d, B_d, A_d)$  表示。由 glBlendColor 指定的颜色由  $(R_c, G_c, B_c, A_c)$  表示。这些值都限制在 0 和  $(K_R, K_G, K_B, K_A)$  之间的范围内，其中

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	$(\frac{R_{s0}}{K_R}, \frac{G_{s0}}{K_G}, \frac{B_{s0}}{K_B}, \frac{A_{s0}}{K_A})$

续表

参 数	$(f_R, f_G, f_B, f_A)$
GL_ONE_MINUS_SRC_COLOR	$(1, 1, 1, 1) - (\frac{R_{s0}}{K_R}, \frac{G_{s0}}{K_G}, \frac{B_{s0}}{K_B}, \frac{A_{s0}}{K_A})$
GL_DST_COLOR	$(\frac{R_d}{K_R}, \frac{G_d}{K_G}, \frac{B_d}{K_B}, \frac{A_d}{K_A})$
GL_ONE_MINUS_DST_COLOR	$(1, 1, 1, 1) - (\frac{R_d}{K_R}, \frac{G_d}{K_G}, \frac{B_d}{K_B}, \frac{A_d}{K_A})$
GL_SRC_ALPHA	$(\frac{A_{s0}}{K_A}, \frac{A_{s0}}{K_A}, \frac{A_{s0}}{K_A}, \frac{A_{s0}}{K_A})$
GL_ONE_MINUS_SRC_ALPHA	$(1, 1, 1, 1) - (\frac{A_{s0}}{K_A}, \frac{A_{s0}}{K_A}, \frac{A_{s0}}{K_A}, \frac{A_{s0}}{K_A})$
GL_DST_ALPHA	$(\frac{A_d}{K_A}, \frac{A_d}{K_A}, \frac{A_d}{K_A}, \frac{A_d}{K_A})$
GL_ONE_MINUS_DST_ALPHA	$(1, 1, 1, 1) - (\frac{A_d}{K_A}, \frac{A_d}{K_A}, \frac{A_d}{K_A}, \frac{A_d}{K_A})$
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	$(1, 1, 1, 1) - (A_c, A_c, A_c, A_c)$
GL_SRC_ALPHA_SATURATE	$(i, i, i, 1)$
GL_SRC1_COLOR	$(\frac{R_{s1}}{K_R}, \frac{G_{s1}}{K_G}, \frac{B_{s1}}{K_B}, \frac{A_{s1}}{K_A})$
GL_ONE_MINUS_SRC1_COLOR	$(1, 1, 1, 1) - (\frac{R_{s1}}{K_R}, \frac{G_{s1}}{K_G}, \frac{B_{s1}}{K_B}, \frac{A_{s1}}{K_A})$
GL_SRC1_ALPHA	$(\frac{A_{s1}}{K_A}, \frac{A_{s1}}{K_A}, \frac{A_{s1}}{K_A}, \frac{A_{s1}}{K_A})$
GL_ONE_MINUS_SRC1_ALPHA	$(1, 1, 1, 1) - (\frac{A_{s1}}{K_A}, \frac{A_{s1}}{K_A}, \frac{A_{s1}}{K_A}, \frac{A_{s1}}{K_A})$

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

而 $(m_R, m_G, m_B, m_A)$  则是红、绿、蓝和 Alpha 位平面 (bitplane) 的数量。

源和目的比例因子由 $(s_R, s_G, s_B, s_A)$ 和 $(d_R, d_G, d_B, d_A)$ 表示。表中描述的比例因子由 $(f_R, f_G, f_B, f_A)$ 表示, 表示源或目的因子。所有比例因子取值范围都是 $[0, 1]$ 。

在表中,

$$i = \frac{\min(A_s, k_A - A_d)}{k_A}$$

为了决定一个像素的混合 RGBA 值, 系统使用下列方程。

$$R_d = \min(k_R, R_s s_R + R_d d_R) \quad G_d = \min(k_G, G_s s_G + G_d d_G) \quad B_d = \min(k_B, B_s s_B + B_d d_B)$$

$$A_d = \min(k_A, A_s s_A + A_d d_A)$$

不管上面的方程看起来精度如何, 由于混合操作中不精确的整数颜色值, 混合算法并没有确切地指定。不过, 等于 1 的一个混合因子确保不会改变它的被乘数, 而等于 0 的混合因子会将它的被乘数减小到 0。举例来说, 当 sfactor 为 GL\_SRC\_ALPHA, dfactor 为 GL\_ONE\_MINUS\_SRC\_ALPHA, 而  $A_s$  等于  $k_A$  时, 方程式简化为替换式  $R_d = R_s G_d = G_s B_d = B_s A_d = A_s$ 。

示例

透明度最好使用从最远到最近存储图元的混合函数(GL\_SRC\_ALPHA、GL\_ONE\_MINUS\_SRC\_ALPHA)来实现。注意这种透明度计算不需要出现帧缓冲区中的 Alpha 位平面。

混合函数 (GL\_SRC\_ALPHA、GL\_ONE\_MINUS\_SRC\_ALPHA) 对于以任意顺序渲染反锯齿点和线也是非常有用的。

多边形反锯齿使用从最远到最近存储多边形的混合函数(GL\_SRC\_ALPHA\_SATURATE, GL\_ONE)来进行优化。(关于多边形反锯齿的信息参见 glEnable 和 glDisable 参考页, 以及自变量 GL\_POLYGON\_SMOOTH。)目的 Alpha 位平面存储累积覆盖, 必须使用这个平面以使这个混合函数正确进行操作。

#### 注意

输入 (源) Alpha 可以被看作一种材料不透明性, 范围从 1.0 (KA) 开始, 这时代表完全不透明, 一直到 0.0 (0), 这时代表完全透明。

当启用一个以上的颜色缓冲区进行绘制时, GL 分别为每个启用的缓冲区执行混合, 使用这个缓冲区的内容作为目的颜色。(参见 glDrawBuffer)。当启用双源混合时 (也就是说, 使用一个需要第二个颜色输入的混合因子), 启用绘制缓冲区的最大数量将由 GL\_MAX\_DUAL\_SOURCE\_DRAW\_BUFFERS 给出, 这个值可能要低于 GL\_MAX\_DRAW\_BUFFERS。

#### 错误

如果 sfactor 或 dfactor 不是可接受的值, 则产生 GL\_INVALID\_ENUM 错误。

#### 相关 Get 函数

glGet, 其自变量为 GL\_BLEND\_SRC。

glGet, 其自变量为 GL\_BLEND\_DST。

glIsEnabled, 其自变量为 GL\_BLEND。

#### 另外查看

glBlendColor、glBlendEquation、glBlendFuncSeparate、glClear、glDrawBuffer、glEnable、glLogicOp、glStencilFunc。

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glBlendFuncSeparate

分别为 RGB 和 Alpha 分量指定像素算法。

### C 规范

```
void glBlendFuncSeparate(GLenum srcRGB,
                        GLenum dstRGB,
                        GLenum srcAlpha,
                        GLenum dstAlpha);
```

#### 参数

srcRGB

指定红、绿和蓝色混合因子如何进行计算。初始值为 GL\_ONE。

dstRGB

指定红、绿和蓝色目标混合因子如何进行计算。初始值为 GL\_ZERO。

srcAlpha

指定 Alpha 源混合因子如何进行计算。初始值为 GL\_ONE。

dstAlpha

指定 Alpha 目标混合因子如何进行计算。初始值为 GL\_ZERO。

#### 描述

像素能够使用一个将输入 (源) RGBA 值和已经存在于帧缓冲区 (目的) 的值进行混合的方程来进行绘制。初始状态下是不支持混合的。使用自变量为 GL\_BLEND 的 glEnable 和 glDisable 来激活和关闭混合。

glBlendFuncSeparate 在激活时定义混合操作。srcRGB 指定使用哪种方法来缩放源 RGB 颜色分量。dstRGB 指定使用哪种方法来缩放目的 RGB 颜色分量。同样, srcAlpha 指定使用哪种方法来缩放源 Alpha 颜色分量, 而 dstAlpha 指定使用哪种方法来缩放目的 Alpha 颜色分量。下表中描述了可能的方法。

每种方法定义了 4 个比例因子, 红、绿、蓝和 Alpha 各对应一个因子。

在表中和后面的方程中, 第一个源、第二个源和目的颜色分量分别由  $(R_{s0}, G_{s0}, B_{s0}, A_{s0})$ 、 $(R_{s1}, G_{s1}, B_{s1}, A_{s1})$  和  $(R_d, G_d, B_d, A_d)$  表示。

参 数	RGB 因子	Alpha 因子
GL_ZERO	(0, 0, 0)	0
GL_ONE	(1, 1, 1)	1
GL_SRC_COLOR	$(\frac{R_{s0}}{k_R}, \frac{G_{s0}}{k_G}, \frac{B_{s0}}{k_B})$	$\frac{A_{s0}}{k_A}$
GL_ONE_MINUS_SRC_COLOR	$(1, 1, 1) - (\frac{R_{s0}}{k_R}, \frac{G_{s0}}{k_G}, \frac{B_{s0}}{k_B})$	$1 - \frac{A_{s0}}{k_A}$
GL_DST_COLOR	$(\frac{R_d}{k_R}, \frac{G_d}{k_G}, \frac{B_d}{k_B})$	$\frac{A_d}{k_A}$
GL_ONE_MINUS_DST_COLOR	$(1, 1, 1) - (\frac{R_d}{k_R}, \frac{G_d}{k_G}, \frac{B_d}{k_B})$	$1 - \frac{A_d}{k_A}$
GL_SRC_ALPHA	$(\frac{A_{s0}}{k_A}, \frac{A_{s0}}{k_A}, \frac{A_{s0}}{k_A})$	$\frac{A_{s0}}{k_A}$
GL_ONE_MINUS_SRC_ALPHA	$(1, 1, 1) - (\frac{A_{s0}}{k_A}, \frac{A_{s0}}{k_A}, \frac{A_{s0}}{k_A})$	$1 - \frac{A_{s0}}{k_A}$
GL_DST_ALPHA	$(\frac{A_d}{k_A}, \frac{A_d}{k_A}, \frac{A_d}{k_A})$	$\frac{A_d}{k_A}$
GL_ONE_MINUS_DST_ALPHA	$(1, 1, 1) - (\frac{A_d}{k_A}, \frac{A_d}{k_A}, \frac{A_d}{k_A})$	$1 - \frac{A_d}{k_A}$
GL_CONSTANT_COLOR	(R <sub>c</sub> , G <sub>c</sub> , B <sub>c</sub> )	A <sub>c</sub>
GL_ONE_MINUS_CONSTANT_COLOR	(1, 1, 1) - (R <sub>c</sub> , G <sub>c</sub> , B <sub>c</sub> )	1 - A <sub>c</sub>
GL_CONSTANT_ALPHA	(A <sub>c</sub> , A <sub>c</sub> , A <sub>c</sub> )	A <sub>c</sub>
GL_ONE_MINUS_CONSTANT_ALPHA	(1, 1, 1) - (A <sub>c</sub> , A <sub>c</sub> , A <sub>c</sub> )	1 - A <sub>c</sub>
GL_SRC_ALPHA_SATURATE	(i, i, i)	1
GL_SRC1_COLOR	$(\frac{R_{s1}}{k_R}, \frac{G_{s1}}{k_G}, \frac{B_{s1}}{k_B})$	$\frac{A_{s1}}{k_A}$
GL_ONE_MINUS_SRC1_COLOR	$(1, 1, 1) - (\frac{R_{s1}}{k_R}, \frac{G_{s1}}{k_G}, \frac{B_{s1}}{k_B})$	$1 - \frac{A_{s1}}{k_A}$
GL_SRC1_ALPHA	$(\frac{A_{s1}}{k_A}, \frac{A_{s1}}{k_A}, \frac{A_{s1}}{k_A})$	$\frac{A_{s1}}{k_A}$
GL_ONE_MINUS_SRC1_ALPHA	$(1, 1, 1) - (\frac{A_{s1}}{k_A}, \frac{A_{s1}}{k_A}, \frac{A_{s1}}{k_A})$	$1 - \frac{A_{s1}}{k_A}$

规定它们为 0 和  $(k_R, k_G, k_B, k_A)$  之间的整数值, 其中  $k_c = 2^{m_c} - 1$ , 而  $(m_R, m_G, m_B, m_A)$  则是红、绿、蓝和 Alpha 位平面 (bitplane) 的数量。

源和目的比例因子由  $(S_R, S_G, S_B, S_A)$  和  $(d_R, d_G, d_B, d_A)$  表示。所有比例因子取值范围都是  $[0, 1]$ 。

在表中,

$$i = \min(A_s, 1 - A_d)$$

为了决定一个像素的混合 RGBA 值, 系统使用下列方程。

$$R_d = \min(k_R, R_s S_R + R_d d_R) \quad G_d = \min(k_G, G_s S_G + G_d d_G) \quad B_d = \min(k_B, B_s S_B + B_d d_B)$$

$$A_d = \min(k_A, A_s S_A + A_d d_A)$$

不管上面的方程看起来精度如何, 由于混合操作中不精确的整数颜色值, 混合算法并没有确切地指定。不过, 等于 1 的一个混合因子确保不会改变它的被乘数, 而等于 0 的混合因子会将它的被乘数减小到 0。举例来说, 当 dstRGB 为 GL\_SRC\_ALPHA, dfactor 为 GL\_ONE\_MINUS\_SRC\_ALPHA, 而  $A_s$  等于  $k_A$  时, 方程式简化为替换式  $R_d = R_s, G_d = G_s, B_d = B_s, A_d = A_s$ 。

注意

输入 (源) Alpha 可以被看作一种材料不透明性, 范围从 1.0 ( $k_A$ ) 开始, 这时代表完全不透明, 一直到 0.0 (0),

这时代表完全透明。

当而启用一个以上的颜色缓冲区进行绘制时，GL 分别为每个启用的缓冲区执行混合，使用这个缓冲区的内容作为目的颜色。(参见 `glDrawBuffer`)。当启用双源混合时(也就是说，使用一个需要第二个颜色输入的混合因子)，启用绘制缓冲区的最大数量将由 `GL_MAX_DUAL_SOURCE_DRAW_BUFFERS` 给出，这个值可能要低于 `GL_MAX_DRAW_BUFFERS`。

#### 错误

如果 `srcRGB` 或 `dstRGB` 不是可接受的值，则产生 `GL_INVALID_ENUM` 错误。

#### 相关 Get 函数

`glGet`，其自变量为 `GL_BLEND_SRC_RGB`。

`glGet`，其自变量为 `GL_BLEND_SRC_ALPHA`。

`glGet`，其自变量为 `GL_BLEND_DST_RGB`。

`glGet`，其自变量为 `GL_BLEND_DST_ALPHA`。

`glIsEnabled`，其自变量为 `GL_BLEND`。

#### 另外查看

`glBlendColor`、`glBlendFunc`、`glBlendEquation`、`glClear`、`glDrawBuffer`、`glEnable`、`glLogicOp`、`glStencilFunc`。

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glBlitFramebuffer

绑定一个顶点数组对象。

### C 规范

```
void glBlitFramebuffer(GLint srcX0,
                       GLint srcY0,
                       GLint srcX1,
                       GLint srcY1,
                       GLint dstX0,
                       GLint dstY0,
                       GLint dstX1,
                       GLint dstY1,
                       GLbitfield mask,
                       GLenum filter);
```

#### 参数

`srcX0`

`srcY0`

`srcX1`

`srcY1`

指定读取帧缓冲区中的读取缓冲区中源矩形的边界。

`dstX0`

`dstY0`

`dstX1`

`dstY1`

指定写入帧缓冲区中的写入缓冲区中目的矩形的边界。

`mask`

对标记进行按位或，指示要对哪些缓冲区进行复制。允许的标志有 `GL_COLOR_BUFFER_BIT`、`GL_DEPTH_BUFFER_BIT` 和 `GL_STENCIL_BUFFER_BIT`。

`filter`

如果图像进行拉伸，那么就要指定进行的插值。必须为 `GL_NEAREST` 或 `GL_LINEAR`。

#### 描述

`glBlitFramebuffer` 将一个像素值矩形从读取帧缓冲区中的一个区域移动到绘制帧缓冲区中的另一个区域。`mask`

就是对一些值进行的按位或，指示要对哪些缓冲区进行复制。这些值包括 GL\_COLOR\_BUFFER\_BIT、GL\_DEPTH\_BUFFER\_BIT 和 GL\_STENCIL\_BUFFER\_BIT。与这些缓冲区对应的像素从由(srcX0; srcY0)和(srcX1; srcY1)位置限定的源矩形被复制到由(dstX0; dstY0)和(dstX1; dstY1)位置限定的目标矩形。其中包含了矩形的底部边界，但没有包含顶部边界。

从读取缓冲区中获取的实际区域被限制在这些传输的源缓冲区的交叉部分，其中可能包括由读取缓冲区选择的颜色缓冲区、深度缓冲区和/或模板缓冲区，根据遮罩而定。写入到绘制缓冲区的实际区域被限制在这些被写入的目的缓冲区的交叉部分，其中可能包括多重绘制缓冲区、深度缓冲区和/或模板缓冲区，根据遮罩而定。源区域或目的区域是否改变要依据这些限制条件而定，应用到被传输的像素上的缩放和偏置会被执行，就像不存在着些限制一样。

如果源矩形和目的矩形的大小不相等，那么 filter 会指定将要在重新设定源图像大小时应用的插值方法，并且必须为 GL\_NEAREST 或 GL\_LINEAR。

GL\_LINEAR 只是一种对于颜色缓冲区有效的插值方法。如果 filter 不是 GL\_NEAREST，并且 mask 包括 GL\_DEPTH\_BUFFER\_BIT 或 GL\_STENCIL\_BUFFER\_BIT，那么不会传输任何数据，并且会产生一个 GL\_INVALID\_OPERATION 错误。

如果 filter 是 GL\_LINEAR，并且源矩形需要在源帧缓冲区的边界之外进行采样，那么这些值将被读取，就像应用了像纹理环绕模式一样。

当颜色缓冲区进行传输时，那么将会从读取帧缓冲区的读取缓冲区中获取值，并写入到绘制帧缓冲区的每个绘制缓冲区中。

如果源矩形和目的矩形重叠，或只是同一个矩形，并且读取缓冲区和绘制缓冲区是同一个缓冲区，那么操作的结果将是未定义的。

#### 注意

glBindVertexArray 在 3.0 或更高版本的 GL 中可用。

#### 错误

如果 mask 包含 GL\_DEPTH\_BUFFER\_BIT 或 GL\_STENCIL\_BUFFER\_BIT 中的任何一个，并且 filter 不是 GL\_NEAREST，则产生 GL\_INVALID\_OPERATION 错误。

如果 mask 包含 GL\_DEPTH\_BUFFER\_BIT，或者下列条件中的任何一个成立，则产生 GL\_INVALID\_OPERATION 错误。

读取缓冲区包含定点或浮点值，而任何绘制缓冲区都不包含定点值或浮点值。

读取缓冲区包含无符号值，而任何绘制缓冲区都不包含无符号值。

读取缓冲区包含有符号值，而任何绘制缓冲区都不包含有符号值。

如果 mask 包含 GL\_DEPTH\_BUFFER\_BIT 或 GL\_STENCIL\_BUFFER\_BIT，并且源深度格式和目的深度格式、源模板格式和目的模板格式不匹配，则产生 GL\_INVALID\_OPERATION 错误。

如果 filter 是 GL\_LINEAR，并且读取缓冲区包含整型数据，则产生 GL\_INVALID\_OPERATION 错误。

如果读取和绘制缓冲区的 GL\_SAMPLES 值不相等，则产生 GL\_INVALID\_OPERATION 错误。

如果读取和绘制缓冲区的 GL\_SAMPLE\_BUFFERS 都大于 0，并且源矩形和目的矩形的维度不相等，则产生 GL\_INVALID\_OPERATION 错误。

如果绑定到 GL\_DRAW\_FRAMEBUFFER\_BINDING 或 GL\_READ\_FRAMEBUFFER\_BINDING 的对象不完全是真缓冲区，则产生 GL\_INVALID\_FRAMEBUFFER\_OPERATION 错误。

#### 另外查看

glReadPixels、glCheckFramebufferStatus、glGenFramebuffers、glBindFramebuffer、glDeleteFramebuffers。

#### 版权

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glBufferData

创建和初始化一个缓冲区对象的数据存储。

#### C 规范

```
void glBufferData(GLenum target,
                  GLsizeiptr size,
                  const GLvoid * data,
                  GLenum usage);
```

## 参数

### target

指定目标缓冲区对象。符号常量必须是 GL\_ARRAY\_BUFFER、GL\_COPY\_READ\_BUFFER、GL\_COPY\_WRITE\_BUFFER、GL\_ELEMENT\_ARRAY\_BUFFER、GL\_PIXEL\_PACK\_BUFFER、GL\_PIXEL\_UNPACK\_BUFFER、GL\_TEXTURE\_BUFFER、GL\_TRANSFORM\_FEEDBACK\_BUFFER 或 GL\_UNIFORM\_BUFFER。

### size

指定缓冲区对象的新数据存储的大小，以 byte 为单位。

### data

指定一个到将要被复制到数据存储以进行初始化的数据的指针，或者在没有数据要复制的情况下为 NULL。

### usage

指定数据存储的预期使用模式。符号常量必须是 GL\_STREAM\_DRAW、GL\_STREAM\_READ、GL\_STREAM\_COPY、GL\_STATIC\_DRAW、GL\_STATIC\_READ、GL\_STATIC\_COPY、GL\_DYNAMIC\_DRAW、GL\_DYNAMIC\_READ 或 GL\_DYNAMIC\_COPY。

## 描述

glBufferData 为当前绑定到 target 的缓冲区对象创建一个新的数据存储。以前存在的所有数据都会被删除。新的数据存储使用指定的 byte 形式的 size (以字节为单位) 和 usage。如果 data 不为 NULL，数据存储会使用来自这个指针的数据进行初始化。在它的初始状态下，新的数据存储没有被映射，它有一个 NULL 映射指针，而它的映射访问是 GL\_READ\_WRITE。

usage 对 GL 实现来说是关于一个缓冲区对象的数据存储将如何被访问的一个提示。

它使得 GL 实现能够做出更智能的决定，这可能会对缓冲区对象性能产生极大的影响。不过，它不会限制数据存储的实际使用。usage 能够分解成两个部分，首先是访问 (包括修改和使用) 频率，其次是访问的种类。访问频率可以是如下几种。

STREAM: 数据存储内容将被改变一次，最多使用几次。

STATIC: 数据存储内容将被改变一次，使用很多次。

DYNAMIC: 数据存储内容将被改变多次，使用很多次。

访问种类可以是如下几种。

DRAW: 数据存储内容由应用程序进行修改，并作为 GL 绘图和图像规范命令使用。

READ: 数据存储内容由 GL 中的读取数据进行修改，并在被应用程序查询时用来返回这个数据。

COPY: 数据存储内容由 GL 中的读取数据进行修改，并作为 GL 绘图和图像规范命令使用。

## 注意

如果 data 为 NULL，仍然会创建一个指定大小的数据存储，但是它的内容保留为未初始化和为定义的。

客户端程序必须按照客户端平台的要求 (还有一个附加的基本要求，即在缓冲区中由 NN 组成的数据的偏移) 排列数据元素。

## 错误

如果 target 不是一个可接受的缓冲区对象，则产生 GL\_INVALID\_ENUM 错误。

如果 target 不是 GL\_STREAM\_DRAW、GL\_STREAM\_READ、GL\_STREAM\_COPY、GL\_STATIC\_DRAW、GL\_STATIC\_READ、GL\_STATIC\_COPY、GL\_DYNAMIC\_DRAW、GL\_DYNAMIC\_READ 或 GL\_DYNAMIC\_COPY，则产生 GL\_INVALID\_ENUM 错误。

如果 size 为负值，则产生 GL\_INVALID\_VALUE 错误。

如果保留的缓冲区对象名称 0 被绑定到 target，则产生 GL\_INVALID\_OPERATION 错误。

如果 GL 不能用指定的 size 创建数据存储，则产生 GL\_OUT\_OF\_MEMORY 错误。

## 相关 Get 函数

glGetBufferSubData

glGetBufferParameter，其自变量为 GL\_BUFFER\_SIZE 或 GL\_BUFFER\_USAGE。

## 另外查看

glBindBuffer、glBufferSubData、glMapBuffer、glUnmapBuffer。

## 版权

Copyright © 2005 Addison-Wesley. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glBufferSubData

更新一个缓冲区对象的数据存储的子集。

### C 规范

```
void glBufferSubData(GLenum target,
                    GLintptr offset,
                    GLsizeiptr size,
                    const GLvoid * data);
```

### 参数

target

指定目标缓冲区对象。符号常量必须是 GL\_ARRAY\_BUFFER、GL\_COPY\_READ\_BUFFER、GL\_COPY\_WRITE\_BUFFER、GL\_ELEMENT\_ARRAY\_BUFFER、GL\_PIXEL\_PACK\_BUFFER、GL\_PIXEL\_UNPACK\_BUFFER、GL\_TEXTURE\_BUFFER、GL\_TRANSFORM\_FEEDBACK\_BUFFER 或 GL\_UNIFORM\_BUFFER。

offset

指定将要开始数据替换的缓冲区对象数据存储中的偏移，以字节为单位。

size

指定被替换的数据存储区域的大小，以字节为单位。

data

指定到将要复制到数据存储的新数据的一个指针。

### 描述

glBufferSubData 重新定义当前捆绑到 target 的缓冲区对象的一部分或全部数据存储。起始于字节偏移 offset 和扩展 size 字节的数据会从由 data 指向的内存复制到数据存储。如果 offset 和 size 一起定义一个超出缓冲区对象数据存储边界的范围，将会抛出一个错误。

### 注意

在替代整个数据存储时，要考虑使用 glBufferSubData，而不是使用 glBufferData 来完全重新创建数据存储。这样可以避免重新定位数据存储的开销。

考虑使用多个缓冲区对象来避免在数据存储更新时对渲染管线的延迟。如果管线中的任何渲染引用缓冲区对象中由 glBufferSubData 来进行更新的数据，尤其是来自被更新的指定区域的数据，那么在数据存储能够更新之前，这个渲染必须从管道中抽出（drain from the pipeline）。

客户端程序必须按照客户端平台的要求（还有一个附加的基本要求，即在缓冲区中由 NN 组成的数据的偏移）排列数据元素。

### 错误

如果 target 不是一个可接受的缓冲区对象，则产生 GL\_INVALID\_ENUM 错误。

如果 offset 或者 size 为负，或者如果 offset 和 size 一起定义一个超出缓冲区对象数据存储范围的内存区域，则产生 GL\_INVALID\_VALUE 错误。

如果保留的缓冲区对象名称 0 被绑定到 target，则产生 GL\_INVALID\_OPERATION 错误。

如果保留的缓冲区对象名称 0 被绑定到 target，则产生 GL\_INVALID\_OPERATION 错误。

### 相关 Get 函数

glGetBufferSubData

### 另外查看

glBindBuffer、glBufferData、glMapBuffer、glUnmapBuffer。

### 版权

Copyright © 2005 Addison-Wesley。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glCheckFramebufferStatus

检查一个帧的完整性状态

### C 规范

```
GGLenum glCheckFramebufferStatus(GGLenum target);
```

#### 参数

target

指定帧缓冲区完整性检查的目标。

#### 描述

glCheckFramebufferStatus 查询当前绑定到 target 的帧缓冲区对象的完整性状态。target 必须为 GL\_DRAW\_FRAMEBUFFER、GL\_READ\_FRAMEBUFFER 或 GL\_FRAMEBUFFER。GL\_FRAMEBUFFER 等价于 GL\_DRAW\_FRAMEBUFFER。

如果被绑定到的帧缓冲区是完整的，那么返回值将为 GL\_FRAMEBUFFER\_COMPLETE。否则返回值将按如下方式确定。

如果 target 为默认帧缓冲区，但默认帧缓冲区不存在，那么将返回 GL\_FRAMEBUFFER\_UNDEFINED。

如果任何绑定点是帧缓冲区不完全的，那么将返回 GL\_FRAMEBUFFER\_INCOMPLETE\_ATTACHMENT。

如果没有图像绑定到帧缓冲区，那么将返回 GL\_FRAMEBUFFER\_INCOMPLETE\_MISSING\_ATTACHMENT。

如果由 GL\_DRAW\_FRAMEBUFFER 指定的任何颜色绑定点的 GL\_FRAMEBUFFER\_ATTACHMENT\_OBJECT\_TYPE 的值为 GL\_NONE，那么将返回 GL\_FRAMEBUFFER\_INCOMPLETE\_DRAW\_BUFFER。

如果 GL\_READ\_BUFFER 不是 GL\_NONE，并且由 GL\_READ\_BUFFER 指定的颜色绑定点的 GL\_FRAMEBUFFER\_ATTACHMENT\_OBJECT\_TYPE 的值为 GL\_NONE，那么将返回 GL\_FRAMEBUFFER\_INCOMPLETE\_READ\_BUFFER。

如果绑定图像的内部的组合违反了与实现相关的限定设置，那么将返回 GL\_FRAMEBUFFER\_UNSUPPORTED。

如果对于所有绑定渲染缓冲区来说 GL\_RENDERBUFFER\_SAMPLES 的值不是都相同的；或者对于所有绑定纹理来说 GL\_TEXTURE\_SAMPLES 的值不是都相同的；或者如果绑定的图像是渲染缓冲区和纹理的混合，而 GL\_RENDERBUFFER\_SAMPLES 的值与 GL\_TEXTURE\_SAMPLES 的值不匹配，那么将返回 GL\_FRAMEBUFFER\_INCOMPLETE\_MULTISAMPLE。

如果对于所有绑定纹理来说 GL\_TEXTURE\_FIXED\_SAMPLE\_LOCATIONS 的值不是都相同的；或者如果绑定的图像是渲染缓冲区和纹理的混合，而对于所有绑定的纹理来说 GL\_TEXTURE\_FIXED\_SAMPLE\_LOCATIONS 的值不都是 GL\_TRUE，那么也将返回 GL\_FRAMEBUFFER\_INCOMPLETE\_MULTISAMPLE。

如果任何帧缓冲区绑定是分层的并且任何填充绑定不是分层的，或者所有填充色绑定不是来自于同一个目标的纹理中，那么将返回 GL\_FRAMEBUFFER\_INCOMPLETE\_LAYER\_TARGETS。

另外，如果出现任何错误，就会返回 0。

#### 错误

如果 target 不是 GL\_DRAW\_FRAMEBUFFER、GL\_READ\_FRAMEBUFFER 或 GL\_FRAMEBUFFER，则产生 GL\_INVALID\_ENUM 错误。

#### 另外查看

glGenFramebuffers、glDeleteFramebuffers、glBindFramebuffer。

#### 版权

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glClampColor

指定是否应该对通过 glReadPixels 读取的数据进行截取。

#### C 规范

```
void glClampColor(GGLenum target,  
                 GGLenum clamp);
```

#### 参数

target

颜色截取的目标。target 必须为 GL\_CLAMP\_READ\_COLOR。

clamp

指定是否应用颜色截取。clamp 必须为 GL\_TRUE 或 GL\_FALSE。

#### 描述

glClampColor 控制在 glReadPixels 过程中执行的颜色截取操作。target 必须为 GL\_CLAMP\_READ\_COLOR。如果 clamp 为 GL\_TRUE, 那么将会开启读取颜色截取; 如果 clamp 为 GL\_FALSE, 那么读取颜色截取将关闭。如果 clamp 为 GL\_FIXED\_ONLY, 那么只有当读取缓冲区有定点部分时才会开启读取颜色截取, 否则将关闭。

#### 错误

如果 target 不是 GL\_RENDERBUFFER, 则产生 GL\_INVALID\_ENUM 错误。

如果 clamp 不是 GL\_TRUE 或 GL\_FALSE, 则产生 GL\_INVALID\_ENUM 错误。

#### 相关 Get 函数

glGet, 其自变量为 GL\_CLAMP\_READ\_COLOR。

#### 版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glClear

清除缓冲区对数值进行预置。

#### C 规范

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

#### 参数

mask

指出将要清除的缓冲区的遮罩的按位或 (Bitwise OR) 运算。这 3 个遮罩为 GL\_COLOR\_BUFFER\_BIT、GL\_DEPTH\_BUFFER\_BIT 和 GL\_STENCIL\_BUFFER\_BIT。

#### 描述

glClear 将窗口的位平面区域设置到之前由 glClearColor、glClearDepth 和 glClearStencil 选择的数值。多重颜色缓冲区能够通过使用 glDrawBuffer 同时选择多个缓冲区来同时进行清除。

像素所有权测试、裁剪测试、抖动和缓冲区写屏蔽会影响 glClear 的执行。裁剪框限制了清除区域。Alpha 函数、混合函数、逻辑操作、模板、纹理贴图和深度缓冲区都会被 glClear 忽略。

glClear 接受单个自变量, 即指示哪个缓冲区要被清除几个值的按位或。

这些值如下所示。

GL\_COLOR\_BUFFER\_BIT 指示当前激活的用来进行颜色写入的缓冲区。

GL\_DEPTH\_BUFFER\_BIT 指示深度缓冲区。

GL\_STENCIL\_BUFFER\_BIT 指示模板缓冲区。

每个缓冲区的清除值根据这个缓冲区的清除值设置而不同。

#### 注意

如果一个缓冲区不存在, 那么指向这个缓冲区的 glClear 不会生效。

#### 错误

GL\_INVALID\_VALUE 产生, 如果 mask 中设置了除上面定义的 3 个位之外的其他位。

#### 相关 Get 函数

glGet, 其自变量为 GL\_DEPTH\_CLEAR\_VALUE。

glGet, 其自变量为 GL\_COLOR\_CLEAR\_VALUE。

glGet, 其自变量为 GL\_STENCIL\_CLEAR\_VALUE。

#### 另外查看

glClearColor、glClearDepth、glClearStencil、glColorMask、glDepthMask、glDrawBuffer、glScissor、glStencilMask。

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 http://oss.sgi.com/projects/FreeB/。

## glClearBuffer

清除当前绑定的绘制帧缓冲区的独立缓冲区。

### C 规范

```
void glClearBufferiv(GLenum buffer,
                    GLint drawBuffer,
                    const GLint * value);

void glClearBufferuiv(GLenum buffer,
                    GLint drawBuffer,
                    const GLuint * value);

void glClearBufferfv(GLenum buffer,
                    GLint drawBuffer,
                    const GLfloat * value);

void glClearBufferfi(GLenum buffer,
                    GLint drawBuffer,
                    GLfloat depth,
                    GLint stencil);
```

### 参数

buffer

指定将要清除的缓冲区。

drawBuffer

指定将要清除的特定绘制缓冲区。

value

对于颜色缓冲区来说，这是一个指向代表 R、G、B 和 A 值的四元素向量的指针，这个缓冲区将被清除为这些值。对于深度缓冲区来说，这是一个指向单个深度值的指针，这个缓冲区将被清除为这个值。对于模板缓冲区来说，这是一个指向单个模板值的指针，这个缓冲区将被清除为这个值。

depth

深度渲染缓冲区将要被清除为这个值。

stencil

模板渲染缓冲区将要被清除为这个值。

### 描述

glClearBuffer\* 将指定缓冲区清除为特定值。如果 buffer 为 GL\_COLOR，那么就可以通过传递 i 为 drawBuffer 而指定一个特定的绘制缓冲区 GL\_DRAWBUFFERi。在这种情况下，value 指向一个代表 R、G、B 和 A 颜色的四元素向量，这个绘制缓冲区将被清除为这些值。

如果 buffer 是 GL\_FRONT、GL\_BACK、GL\_LEFT、GL\_RIGHT 或 GL\_FRONT\_AND\_BACK 中的一个，表示多重缓冲区，那么每个选定的缓冲区都会被清除为同样的值。定点颜色缓冲区的截取和变换的执行方式与 glClearColor 相同。

如果 buffer 为 GL\_DEPTH，那么 drawBuffer 必须为 0，而则指向这个缓冲区将被清除为的单个值。在清除深度缓冲区时应该使用的只有 glClearBufferfv。定点深度缓冲区的截取和变换的执行方式与 glClearDepth 相同。

如果 buffer 为 GL\_STENCIL，那么 drawBuffer 必须为 0，而则指向这个模板缓冲区将被清除为的单个值。在清除模板缓冲区时应该使用的只有 glClearBufferiv。遮罩和类型变换的执行方式与 glClearStencil 相同。

glClearBufferfi 可以用来对深度缓冲区和模板缓冲区进行清除。buffer 必须为 GL\_DEPTH\_STENCIL，而 drawBuffer 则必须为 0。depth 和 stencil 分别为深度值和模板值。

如果在 value 的类型和将要进行清除的缓冲区之间没有进行变换，那么 glClearBuffer 的结果将是未定义的。但是，并不是一个错误。

### 错误

如果 buffer 不是 GL\_COLOR、GL\_FRONT、GL\_BACK、GL\_LEFT、GL\_RIGHT、GL\_FRONT\_AND\_BACK、GL\_DEPTH 或 GL\_STENCIL，那么 glClearBufferi、glClearBufferfv 和 glClearBufferuiv 将生成 GL\_INVALID\_ENUM 错误。

如果 buffer 不是 GL\_DEPTH\_STENCIL，则产生 GL\_INVALID\_ENUM 错误。

如果 buffer 是 GL\_COLOR、GL\_FRONT、GL\_BACK、GL\_LEFT、GL\_RIGHT 或 GL\_FRONT\_AND\_BACK，而 drawBuffer 大于或等于 GL\_MAX\_DRAW\_BUFFERS，则产生 GL\_INVALID\_VALUE 错误。

如果 `buffer` 是 `GL_DEPTH`, `GL_STENCIL` 或 `GL_DEPTH_STENCIL`, 并且 `drawBuffer` 不是 0, 则产生 `GL_INVALID_VALUE` 错误。

#### 另外查看

`glClearColor`、`glClearDepth`、`glClearStencil`、`glClear`。

#### 版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。 <http://opencontent.org/openpub/>。

### `glClearColor`

为颜色缓冲区指定清除值。

#### C 规范

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

#### 参数

`red`

`green`

`blue`

`alpha`

指定在颜色缓冲区清除时使用的红、绿、蓝和 Alpha 值。

它们的初始值都为 0。

#### 描述

`glClearColor` 指定 `glClear` 清除颜色缓冲区所使用的红、绿、蓝和 Alpha 值。由 `glClearColor` 指定的值将被截取到 `[0,1]` 范围内。

#### 相关 Get 函数

`glGet`, 其自变量为 `GL_COLOR_CLEAR_VALUE`

#### 另外查看

`glClear`

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

### `glClearDepth`

为深度缓冲区指定清除值。

#### C 规范

```
void glClearDepth(GLclampd depth);
```

#### 参数

`depth`

指定在深度缓冲区清除时使用的深度值。初始值为 1。

#### 描述

`glClearDepth` 指定 `glClear` 清除深度缓冲区所使用的深度值。由 `glClearDepth` 指定的值将被截取到 `[0,1]` 范围内。

#### 相关 Get 函数

`glGet`, 其自变量为 `GL_DEPTH_CLEAR_VALUE`。

#### 另外查看

`glClear`

**版权**

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

**glClearStencil**

为模板缓冲区指定清除值。

**C 规范**

```
void glClearStencil(GLint s);
```

**参数**

s

指定在模板缓冲区清除时使用的索引。初始值为 0。

**描述**

glClearStencil 指定 glClear 清除模板缓冲区所使用的索引。s 由  $2^m - 1$  进行遮罩, 其中 m 是模板缓冲区中的位数。

**相关 Get 函数**

glGet, 其自变量为 GL\_STENCIL\_CLEAR\_VALUE。

glGet, 其自变量为 GL\_STENCIL\_BITS。

**另外查看**

glClear、glStencilFunc、glStencilFuncSeparate、glStencilMask、glStencilMaskSeparate、glStencilOp、glStencilOpSeparate。

**版权**

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

**glClientWaitSync**

进行阻塞并等待一个同步对象变成标记状态。

**C 规范**

```
GLenum glClientWaitSync(GLsync sync,
                        GLbitfield flags,
                        GLuint64 timeout);
```

**参数**

sync

将要等待其状态的同步对象。

flags

一个控制命令清理行为的位段。flags 可能为 GL\_SYNC\_FLUSH\_COMMANDS\_BIT。

timeout

超时值, 以纳秒为单位, 代表实现应该等待 sync 变成标记状态的时间。

**描述**

glClientWaitSync 导致客户端阻塞并等待由 sync 指定的同步对象变成标记状态。如果在调用 glClientWaitSync 时 sync 为标记状态, 那么 glClientWaitSync 将立即返回, 否则它将阻塞并等待 sync 变为标记状态 timeout 纳秒。

返回值是 4 个状态值中的一个。

GL\_ALREADY\_SIGNALED 代表当 glClientWaitSync 调用时 sync 为标记状态。

GL\_TIMEOUT\_EXPIRED 代表至少经过了 timeout 纳秒, 而 sync 并没有变为标记状态。

GL\_CONDITION\_SATISFIED 代表在 timeout 到期之前 sync 变为了标记状态。

GL\_WAIT\_FAILED 代表出现了一个 error。另外, 还会出现一个 OpenGL error。

**注意**

glClientWaitSync 在 3.2 或更高版本的 GL 中可用。

**错误**

如果 sync 不是一个已存在同步对象的名称, 则产生 GL\_INVALID\_VALUE 错误。

如果 flags 包含任意不支持的标记, 则产生 GL\_INVALID\_VALUE 错误。

另外查看

glFenceSync、glIsSync、glWaitSync。

版权

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glColorMask

启用和禁用帧缓冲区颜色分量写入。

C 规范

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

参数

red

green

blue

alpha

指定红、绿、蓝和 Alpha 值能够或者不能被写入帧缓冲区。初始值都是 GL\_TRUE, 表示颜色分量和以被写入。

描述

glColorMask 指定帧缓冲区中单独的颜色分量能够或者不能被写入。例如, 如果 red 是 GL\_FALSE, 不管绘制操作如何尝试, 任何颜色缓冲区中的任何像素的红色分量都不会改变。

不能控制对分量的单独位的改变是。相反, 对于整个颜色分量来说, 改变是同时被激活或禁止的。

相关 Get 函数

glGet, 其自变量为 GL\_COLOR\_WRITEMASK。

另外查看

glClear、glDepthMask、glStencilMask。

版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glCompileShader

编译一个着色器对象。

C 规范

```
void glCompileShader(GLuint shader);
```

参数

shader

指定要编译的着色器对象。

描述

glCompileShader 编译已经被存储到由 shader 指定的着色器对象的源代码字符串。

编译状态将会作为着色器对象状态的一部分被存储。如果着色器被编译, 没有错误且已经准备好供使用, 这个值将被设置为 GL\_TRUE, 否则将被设置为 GL\_FALSE。它可以通过调用自变量为 shader 和 GL\_COMPILE\_STATUS 的 glGetShader 进行查询。

就像着色语言规范中指出的, 着色器的编译可能由于很多原因而失败。无论编译成功与否, 关于编译的信息都能够通过调用 glGetShaderInfoLog 来从着色器对象的信息日志中得到。

## 错误

如果 shader 不是由 OpenGL 产生的值, 则产生 GL\_INVALID\_VALUE 错误。

如果 shader 不是一个着色器对象, 则产生 GL\_INVALID\_OPERATION 错误。

## 相关 Get 函数

glGetShaderInfoLog, 其自变量为 shader。

glGetShader, 其自变量为 shader 和 GL\_COMPILE\_STATUS。

glIsShader

## 另外查看

glCreateShader、glLinkProgram、glShaderSource。

## 版权

Copyright © 2003–2005 3Dlabs Inc. Ltd. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glCompressedTexImage1D

指定一个压缩格式的一维纹理图像。

## C 规范

```
void glCompressedTexImage1D(GLenum target,
                             GLint level,
                             GLenum internalformat,
                             GLsizei width,
                             GLint border,
                             GLsizei imageSize,
                             const GLvoid * data);
```

## 参数

target

指定目标纹理。必须为 GL\_TEXTURE\_1D 或 GL\_PROXY\_TEXTURE\_1D。

level

指定层次细节数量。Level 0 是基本图像层次。Level n 是第 n 级 Mip 贴图缩略图。

internalformat

指定在 data 地址存储的压缩图像数据的格式。

width

指定纹理图像的宽度。所有实现支持纹理图像的宽度至少为 64texel。纹理图像的高度为 1。

border

这个值必须为 0。

imageSize

定起始于由 data 指定地址的图像数据的无符号字节的数量。

data

指定到内存中压缩图像数据的指针。

## 描述

纹理贴图着色器读取一个图像数组的元素。

如果 target 是 GL\_TEXTURE\_1D (参见 glTexImage1D) 的话, 那么 glCompressedTexImage1D 载入一个以前定义并检索的压缩一维纹理图像。

如果 target 是 GL\_PROXY\_TEXTURE\_1D, 那么从 data 中不会读取任何数据, 但是所有的纹理图像状态将会为了一致性而重新进行计算、检查, 并核对实现的性能。如果一个实现不能处理所要求的纹理大小, 那么它会将所有的图像状态设置为 0, 但并不生成错误 (参见 glGetError)。可以使用一个级别等于 1 或大于 1 的图像数组来查询整个 Mip 贴图数组。

internalformat 必须是指定扩展的压缩纹理格式。当使用一个由 GL 从它支持压缩纹理的扩展中选择的一般压缩纹理格式 (例如 GL\_COMPRESSED\_RGB) 载入了 glTexImage1D 的纹理, 为了使用 glCompressedTexImage1D 来载入压缩纹理图像, 要使用 glGetTexLevelParameter 来查询压缩纹理图像的大小和格式。

如果在纹理图像被指定的情况下, 非 0 的指定缓冲区对象被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标 (参见 glBindBuffer), data 将被看作缓冲区对象数据存储的一个 byte 偏移。

#### 错误

如果 internalformat 不是一个支持的特定压缩内部格式, 或者是下列一般压缩内部格式中的一个, 则产生 GL\_INVALID\_ENUM 错误: GL\_COMPRESSED\_RED、GL\_COMPRESSED\_RG、GL\_COMPRESSED\_RGB、GL\_COMPRESSED\_RGBA、GL\_COMPRESSED\_SRGB 或 GL\_COMPRESSED\_SRGB\_ALPHA。

如果 imageSize 与指定压缩图像数据的格式、尺寸和内容不一致, 则产生 GL\_INVALID\_VALUE 错误。

如果 border 为非 0 值, 则产生 GL\_INVALID\_VALUE 错误。

如果指定的压缩内部格式不能像指定纹理压缩扩展中指定的那样支持参数组合, 则生成 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标, 并且缓冲区对象的数据存储当前被映射, 则产生 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标, 并且数据将从缓冲区对象中被解包导致内存读取需求超出数据存储大小, 则产生 GL\_INVALID\_OPERATION 错误。

如果 data 没有被以与定义内部压缩格式的扩展规范一致的方式进行编码, 则会产生未定义的结果, 包括异常程序终止。

#### 相关 Get 函数

glGetCompressedTexImage

glGet, 其自变量为 GL\_TEXTURE\_COMPRESSED。

glGet, 其自变量为 GL\_NUM\_COMPRESSED\_TEXTURE\_FORMATS。

glGet, 其自变量为 GL\_COMPRESSED\_TEXTURE\_FORMATS。

glGet, 其自变量为 GL\_PIXEL\_UNPACK\_BUFFER\_BINDING。

glGetTexLevelParameter, 其自变量为 GL\_TEXTURE\_INTERNAL\_FORMAT 和 GL\_TEXTURE\_COMPRESSED\_IMAGE\_SIZE。

#### 另外查看

glActiveTexture、glCompressedTexImage2D、glCompressedTexImage3D、glCompressedTexSubImage1D、glCompressedTexSubImage2D、glCompressedTexSubImage3D、glCopyTexImage1D、glCopyTexImage2D、glCopyTexSubImage1D、glCopyTexSubImage2D、glCopyTexSubImage3D、glPixelStore、glTexImage2D、glTexImage3D、glTexSubImage1D、glTexSubImage2D、glTexSubImage3D、glTexParameter。

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glCompressedTexImage2D

指定一个压缩格式的二维纹理图像。

#### C 规范

```
void glCompressedTexImage2D(GLenum target,
                             GLint level,
                             GLenum internalformat,
                             GLsizei width,
                             GLsizei height,
                             GLint border,
                             GLsizei imageSize,
                             const GLvoid * data);
```

#### 参数

target

指定目标纹理。Must be GL\_TEXTURE\_2D、GL\_PROXY\_TEXTURE\_2D、GL\_TEXTURE\_1D\_ARRAY、GL\_PROXY\_TEXTURE\_1D\_ARRAY、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Z、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Z 或 GL\_PROXY\_TEXTURE\_CUBE\_MAP。

level

指定层次细节数量。Level 0 是基本图像层次。Level n 是第 n 级 Mip 贴图缩略图。

internalformat

指定在 data 地址存储的压缩图像数据的格式。

width

指定纹理图像的宽度。所有实现支持的 2D 纹理图像的宽度至少为 64texel, 并且立方体贴图纹理图像宽度至少为 16texel。

height

指定纹理图像的高度。所有实现支持的 2D 纹理图像的高度至少为 64texel, 并且立方体贴图纹理图像高度至少为 16texel。

border

这个值必须为 0。

imageSize

定起始于由 data 指定地址的图像数据的无符号字节的数量。

data

指定到内存中压缩图像数据的指针。

描述

纹理贴图着色器读取一个图像数组的元素。

如果 target 是 GL\_TEXTURE\_2D, 或者是诸如 GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X 这样的立方体贴图表面 (参见 glTexImage2D) 的话, 那么 glCompressedTexImage2D 载入一个以前定义并检索的压缩二维纹理图像。

如果 target 是 GL\_TEXTURE\_1D\_ARRAY, 那么 data 将被视为一个压缩 1D 纹理数组。

如果 target 是 GL\_PROXY\_TEXTURE\_2D、GL\_PROXY\_TEXTURE\_1D\_ARRAY 或 GL\_PROXY\_CUBE\_MAP, 那么从 data 中不会读取任何数据, 但是所有的纹理图像状态将会为了一致性而重新进行计算、检查, 并核对实现的性能。如果一个实现不能处理所要求的纹理大小, 那么它会将所有的图像状态设置为 0, 但并不生成错误 (参见 glGetError)。可以使用一个级别等于 1 或大于 1 的图像数组来查询整个 Mip 贴图数组。

internalformat 必须是已知压缩图像格式 (例如 GL\_RGTC), 或者是指定扩展的压缩纹理格式。当使用一个由 GL 从它支持压缩纹理的扩展中选择的一般压缩纹理格式 (例如 GL\_COMPRESSED\_RGB) 载入了 glTexImage2D 的纹理, 为了使用 glCompressedTexImage2D 来载入压缩纹理图像, 要使用 glGetTexLevelParameter 来查询压缩纹理图像的大小和格式。

如果在纹理图像被指定的情况下, 非 0 的指定缓冲区对象被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标 (参见 glBindBuffer), data 将被看作缓冲区对象数据存储的一个 byte 偏移。

错误

如果 internalformat 不是下列一般压缩内部格式之一, 则产生 GL\_INVALID\_ENUM 错误: GL\_COMPRESSED\_RED、GL\_COMPRESSED\_RG、GL\_COMPRESSED\_RGB、GL\_COMPRESSED\_RGBA、GL\_COMPRESSED\_SRGB 或 GL\_COMPRESSED\_SRGB\_ALPHA。

如果 imageSize 与指定压缩图像数据的格式、尺寸和内容不一致, 则产生 GL\_INVALID\_VALUE 错误。

如果 border 为非 0 值, 则产生 GL\_INVALID\_VALUE 错误。

如果指定的压缩内部格式不能像指定纹理压缩扩展中指定的那样支持参数组合, 则生成 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标, 并且缓冲区对象的数据存储当前被映射, 则产生 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标, 并且数据将从缓冲区对象中被解包导致内存读取需求超出数据存储大小, 则产生 GL\_INVALID\_OPERATION 错误。

如果 data 没有被以与定义内部压缩格式的扩展规范一致的方式进行编码, 则会产生未定义的结果, 包括异常程序终止。

相关 Get 函数

glGetCompressedTexImage

glGet, 其自变量为 GL\_TEXTURE\_COMPRESSED。

glGet, 其自变量为 GL\_PIXEL\_UNPACK\_BUFFER\_BINDING。

glGetTexLevelParameter, 其自变量为 GL\_TEXTURE\_INTERNAL\_FORMAT 和 GL\_TEXTURE\_COMPRESSED\_

IMAGE\_SIZE。

#### 另外查看

glActiveTexture, glCompressedTexImage1D, glCompressedTexImage3D, glCompressedTexSubImage1D, glCompressedTexSubImage2D, glCompressedTexSubImage3D, glCopyTexImage1D, glCopyTexSubImage1D, glCopyTexSubImage2D, glCopyTexSubImage3D, glPixelStore, glTexImage2D, glTexImage3D, glTexSubImage1D, glTexSubImage2D, glTexSubImage3D, glTexParameter

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glCompressedTexImage3D

指定一个压缩格式的三维纹理图像。

### C 规范

```
void glCompressedTexImage3D(GLenum target,
                             GLint level,
                             GLenum internalformat,
                             GLsizei width,
                             GLsizei height,
                             GLsizei depth,
                             GLint border,
                             GLsizei imageSize,
                             const GLvoid * data);
```

#### 参数

target

指定目标纹理。必须为 GL\_TEXTURE\_3D、GL\_PROXY\_TEXTURE\_3D、GL\_TEXTURE\_2D\_ARRAY 或 GL\_PROXY\_TEXTURE\_2D\_ARRAY。

level

指定层次细节数量。Level 0 是基本图像层次。Level n 是第 n 级 Mip 贴图缩略图。

internalformat

指定在 data 地址存储的压缩图像数据的格式。

width

指定纹理图像的宽度。所有实现支持纹理图像的宽度至少为 16texel。

height

指定纹理图像的高度。所有实现支持纹理图像的高度至少为 16texel。

depth

指定纹理图像的深度。所有实现支持纹理图像的深度至少为 16texel。

border

这个值必须为 0。

imageSize

定起始于由 data 指定地址的图像数据的无符号字节的数量。

data

指定到内存中压缩图像数据的指针。

#### 描述

纹理贴图着色器读取一个图像数组的元素。

如果 target 是 GL\_TEXTURE\_3D (参见 glTexImage3D) 的话, 那么 glCompressedTexImage3D 载入一个以前定义并检索的压缩三维纹理图像。

如果 target 是 GL\_TEXTURE\_2D\_ARRAY, 那么 data 将被视为一个压缩 2D 纹理数组。

如果 target 是 GL\_PROXY\_TEXTURE\_3D 或 GL\_PROXY\_TEXTURE\_2D\_ARRAY, 那么从 data 中不会读取任何数据, 但是所有的纹理图像状态将会为了一致性而重新进行计算、检查, 并核对实现的性能。如果一个实现不能处理所要求的纹理大小, 那么它会将所有的图像状态设置为 0, 但并不生成错误 (参见 glGetError)。可以使用一个级别

等于 1 或大于 1 的图像数组来查询整个 Mip 贴图数组。

`internalformat` 必须是已知压缩图像格式 (例如 `GL_RGTC`)，或者是指定扩展的压缩纹理格式。当使用一个由 GL 从它支持压缩纹理的扩展中选择的一般压缩纹理格式 (例如 `GL_COMPRESSED_RGB`) 载入了 `glTexImage2D` 的纹理，为了使用 `glCompressedTexImage3D` 来载入压缩纹理图像，要使用 `glGetTexLevelParameter` 来查询压缩纹理图像的大小和格式。

如果在纹理图像被指定的情况下，非 0 的指定缓冲区对象被绑定到 `GL_PIXEL_UNPACK_BUFFER` 目标 (参见 `glBindBuffer`)，`data` 将被看作缓冲区对象数据存储的一个 byte 偏移。

#### 错误

如果 `internalformat` 不是下列一般压缩内部格式之一，则产生 `GL_INVALID_ENUM` 错误：`GL_COMPRESSED_RED`、`GL_COMPRESSED_RG`、`GL_COMPRESSED_RGB`、`GL_COMPRESSED_RGBA`、`GL_COMPRESSED_SRGB` 或 `GL_COMPRESSED_SRGB_ALPHA`。

如果 `imageSize` 与指定压缩图像数据的格式、尺寸和内容不一致，则产生 `GL_INVALID_VALUE` 错误。

如果 `border` 为非 0 值，则产生 `GL_INVALID_VALUE` 错误。

如果指定的压缩内部格式不能像指定纹理压缩扩展中指定的那样支持参数组合，则生成 `GL_INVALID_OPERATION` 错误。

如果非 0 缓冲区对象名称被绑定到 `GL_PIXEL_UNPACK_BUFFER` 目标，并且缓冲区对象的数据存储当前被映射，则产生 `GL_INVALID_OPERATION` 错误。

如果非 0 缓冲区对象名称被绑定到 `GL_PIXEL_UNPACK_BUFFER` 目标，并且数据将从缓冲区对象中被解包导致内存读取需求超出数据存储大小，则产生 `GL_INVALID_OPERATION` 错误。

如果 `data` 没有被以与定义内部压缩格式的扩展规范一致的方式进行编码，则会产生未定义的结果，包括异常程序终止。

#### 相关 Get 函数

`glGetCompressedTexImage`

`glGet`，其自变量为 `GL_TEXTURE_COMPRESSED`。

`glGet`，其自变量为 `GL_PIXEL_UNPACK_BUFFER_BINDING`。

`glGetTexLevelParameter`，其自变量为 `GL_TEXTURE_INTERNAL_FORMAT` 和 `GL_TEXTURE_COMPRESSED_IMAGE_SIZE`。

#### 另外查看

`glActiveTexture`，`glCompressedTexImage1D`，`glCompressedTexImage2D`，`glCompressedTexSubImage1D`，`glCompressedTexSubImage2D`，`glCompressedTexSubImage3D`，`glCopyTexImage1D`，`glCopyTexSubImage1D`，`glCopyTexSubImage2D`，`glCopyTexSubImage3D`，`glPixelStore`，`glTexImage1D`，`glTexImage2D`，`glTexSubImage1D`，`glTexSubImage2D`，`glTexSubImage3D`，`glTexParameter`

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glCompressedTexSubImage1D

定义一个压缩格式的一维纹理子图像。

#### C 规范

```
void glCompressedTexSubImage1D(GLenum target,
                               GLint level,
                               GLint xoffset,
                               GLsizei width,
                               GLenum format,
                               GLsizei imageSize,
                               const GLvoid * data);
```

#### 参数

`target`

指定目标纹理。必须为 `GL_TEXTURE_1D`。

`level`

指定层次细节数量。Level 0 是基本图像层次。Level n 是第 n 级 Mip 贴图缩略图。

xoffset

指定纹理数组中 x 方向上一个 texel 的偏移。

width

指定纹理子图像的宽度。

format

指定在 data 地址存储的压缩图像数据的格式。

imageSize

定起始于由 data 指定地址的图像数据的无符号字节的数量。

data

指定到内存中压缩图像数据的指针。

**描述**

纹理贴图着色器读取一个图像数组的元素。

glCompressedTexSubImage1D 重新定义一个已经存在的一维纹理图像的一个邻近的子区域。由 data 引用的 texel 替代已经存在的纹理数组中 x 索引从 xoffset 到 xoffset + width - 1 (包括边界) 的部分。这个区域可能不包括任何在纹理数组最初指定的范围之外的 texel。指定一个宽度为 0 的子纹理并不是错误, 但是这样做不会产生任何效果。

internalformat 必须是已知压缩图像格式 (例如 GL\_RGTC), 或者是指定扩展的压缩纹理格式。压缩纹理图像的 format 是由压缩它的 GL 实现 (参见 glTexImage1D) 选择的, 并且应该在纹理被带有 glGetTexLevelParameter 压缩的时候被查询。

如果在纹理图像被指定的情况下, 非 0 的指定缓冲区对象被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标 (参见 glBindBuffer), data 将被看作缓冲区对象数据存储的一个 byte 偏移。

**错误**

如果 internalformat 不是下列一般压缩内部格式之一, 则产生 GL\_INVALID\_ENUM 错误: GL\_COMPRESSED\_RED、GL\_COMPRESSED\_RG、GL\_COMPRESSED\_RGB、GL\_COMPRESSED\_RGBA、GL\_COMPRESSED\_SRGB 或 GL\_COMPRESSED\_SRGB\_ALPHA。

如果 imageSize 与指定压缩图像数据的格式、尺寸和内容不一致, 则产生 GL\_INVALID\_VALUE 错误。

如果指定的压缩内部格式不能像指定纹理压缩扩展中指定的那样支持参数组合, 则生成 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标, 并且缓冲区对象的数据存储当前被映射, 则产生 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标, 并且数据将从缓冲区对象中被解包导致内存读取需求超出数据存储大小, 则产生 GL\_INVALID\_OPERATION 错误。

如果 data 没有被以与定义内部压缩格式的扩展规范一致的方式进行编码, 则会产生未定义的结果, 包括异常程序终止。

**相关 Get 函数**

glGetCompressedTexImage

glGet, 其自变量为 GL\_TEXTURE\_COMPRESSED。

glGet, 其自变量为 GL\_PIXEL\_UNPACK\_BUFFER\_BINDING。

glGetTexLevelParameter, 其自变量为 GL\_TEXTURE\_INTERNAL\_FORMAT 和 GL\_TEXTURE\_COMPRESSED\_IMAGE\_SIZE。

**另外查看**

glActiveTexture, glCompressedTexImage1D, glCompressedTexImage2D, glCompressedTexImage3D, glCompressedTexSubImage2D, glCompressedTexSubImage3D, glCopyTexImage1D, glCopyTexImage2D, glCopyTexSubImage1D, glCopyTexSubImage2D, glCopyTexSubImage3D, glPixelStore, glTexImage2D, glTexImage3D, glTexSubImage1D, glTexSubImage2D, glTexSubImage3D, glTexParameter

**版权**

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glCompressedTexSubImage2D

定义一个压缩格式的二维纹理子图像。

### C 规范

```
void glCompressedTexSubImage2D(GLenum target,
                               GLint level,
                               GLint xoffset,
                               GLint yoffset,
                               GLsizei width,
                               GLsizei height,
                               GLenum format,
                               GLsizei imageSize,
                               const GLvoid * data);
```

### 参数

target

指定目标纹理。必须为 GL\_TEXTURE\_2D、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Z 或 GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Z。

level

指定层次细节数量。Level 0 是基本图像层次。Level n 是第 n 级 Mip 贴图缩略图。

xoffset

指定纹理数组中 x 方向上一个 texel 的偏移。

yoffset

指定纹理数组中 y 方向上一个 texel 的偏移。

width

指定纹理子图像的宽度。

height

指定纹理子图像的高度。

format

指定在 data 地址存储的压缩图像数据的格式。

imageSize

定起始于由 data 指定地址的图像数据的无符号字节的数量。

data

指定到内存中压缩图像数据的指针。

### 描述

纹理贴图着色器读取一个图像数组的元素。

glCompressedTexSubImage2D 重新定义一个已经存在的二维纹理图像的一个邻近的子区域。由 data 引用的 texel 替代已经存在的纹理数组中 x 索引从 xoffset 到 xoffset + width - 1, 以及 y 索引从 yoffset 到 yoffset + height - 1 的部分 (包括边界)。

这个区域可能不包括任何在纹理数组最初指定的范围之外的 texel。指定一个宽度为 0 的子纹理并不是错误, 但是这样做不会产生任何效果。

internalformat 必须是已知压缩图像格式 (例如 GL\_RGTC), 或者是指定扩展的压缩纹理格式。压缩纹理图像的 format 是由压缩它的 GL 实现 (参见 glTexImage2D) 选择的, 并且应该在纹理被带有 glGetTexLevelParameter 压缩的时候被查询。

如果在纹理图像被指定的情况下, 非 0 的指定缓冲区对象被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标 (参见 glBindBuffer), data 将被看作缓冲区对象数据存储的一个 byte 偏移。

### 错误

如果 internalformat 是下列一般压缩内部格式, 则产生 GL\_INVALID\_ENUM 错误: GL\_COMPRESSED\_RED、GL\_COMPRESSED\_RG、GL\_COMPRESSED\_RGB、GL\_COMPRESSED\_RGBA、GL\_COMPRESSED\_SRGB 或 GL\_COMPRESSED\_SRGB\_ALPHA。

如果 imageSize 与指定压缩图像数据的格式、尺寸和内容不一致, 则产生 GL\_INVALID\_VALUE 错误。

如果指定的压缩内部格式不能像指定纹理压缩扩展中指定的那样支持参数组合, 则生成 `GL_INVALID_OPERATION` 错误。

如果非 0 缓冲区对象名称被绑定到 `GL_PIXEL_UNPACK_BUFFER` 目标, 并且缓冲区对象的数据存储当前被映射, 则产生 `GL_INVALID_OPERATION` 错误。

如果非 0 缓冲区对象名称被绑定到 `GL_PIXEL_UNPACK_BUFFER` 目标, 并且数据将从缓冲区对象中被解包导致内存读取需求超出数据存储大小, 则产生 `GL_INVALID_OPERATION` 错误。

如果 data 没有被以与定义内部压缩格式的扩展规范一致的方式进行编码, 则会产生未定义的结果, 包括异常程序终止。

#### 相关 Get 函数

`glGetCompressedTexImage`

`glGet`, 其自变量为 `GL_TEXTURE_COMPRESSED`。

`glGet`, 其自变量为 `GL_PIXEL_UNPACK_BUFFER_BINDING`。

`glGetTexLevelParameter`, 其自变量为 `GL_TEXTURE_INTERNAL_FORMAT` 和 `GL_TEXTURE_COMPRESSED_IMAGE_SIZE`。

#### 另外查看

`glActiveTexture`, `glCompressedTexImage1D`, `glCompressedTexImage2D`, `glCompressedTexImage3D`, `glCompressedTexSubImage1D`, `glCompressedTexSubImage3D`, `glCopyTexImage1D`, `glCopyTexImage2D`, `glCopyTexSubImage1D`, `glCopyTexSubImage2D`, `glCopyTexSubImage3D`, `glPixelStore`, `glTexImage2D`, `glTexImage3D`, `glTexSubImage1D`, `glTexSubImage2D`, `glTexSubImage3D`, `glTexParameter`

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

### `glCompressedTexSubImage3D`

定义一个压缩格式的三维纹理子图像。

#### C 规范

```
void glCompressedTexSubImage3D(GLenum target,
                               GLint level,
                               GLint xoffset,
                               GLint yoffset,
                               GLint zoffset,
                               GLsizei width,
                               GLsizei height,
                               GLsizei depth,
                               GLenum format,
                               GLsizei imageSize,
                               const GLvoid * data);
```

#### 参数

target

指定目标纹理。必须为 `GL_TEXTURE_3D`。

level

指定层次细节数量。Level 0 是基本图像层次。Level n 是第 n 级 Mip 贴图缩略图。

xoffset

指定纹理数组中 x 方向上一个 texel 的偏移。

yoffset

指定纹理数组中 y 方向上一个 texel 的偏移。

width

指定纹理子图像的宽度。

height

指定纹理子图像的高度。

depth

指定纹理子图像的深度。

format

指定在 data 地址存储的压缩图像数据的格式。

imageSize

定起始于由 data 指定地址的图像数据的无符号字节的数量。

data

指定到内存中压缩图像数据的指针。

描述

纹理贴图着色器读取一个图像数组的元素。

glCompressedTexSubImage3D 重新定义一个已经存在的三维纹理图像的一个邻近的子区域。由 data 引用的 texel 替代已经存在的纹理数组中 x 索引从 xoffset 到 xoffset + width - 1 的部分, 以及 y 索引从 yoffset 到 yoffset + height - 1 的部分, 以及 z 索引从 zoffset 到 zoffset + depth - 1 的部分 (包括边界)。这个区域可能不包括任何在纹理数组最初指定的范围之外的 texel。指定一个宽度为 0 的子纹理并不是错误, 但是这样做不会产生任何效果。

internalformat 必须是已知压缩图像格式 (例如 GL\_RGTC), 或者是指定扩展的压缩纹理格式。压缩纹理图像的 format 是由压缩它的 GL 实现 (参见 glTexImage3D) 选择的, 并且应该在纹理被带有 glGetTexLevelParameter 压缩的时候被查询。

如果在纹理图像被指定的情况下, 非 0 的指定缓冲区对象被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标 (参见 glBindBuffer), data 将被看作缓冲区对象数据存储的一个 byte 偏移。

错误

如果 internalformat 是下列一般压缩内部格式之一, 则产生 GL\_INVALID\_ENUM 错误: GL\_COMPRESSED\_RED、GL\_COMPRESSED\_RG、GL\_COMPRESSED\_RGB、GL\_COMPRESSED\_RGBA、GL\_COMPRESSED\_SRGB 或 GL\_COMPRESSED\_SRGB\_ALPHA。

如果 imageSize 与指定压缩图像数据的格式、尺寸和内容不一致, 则产生 GL\_INVALID\_VALUE 错误。

如果指定的压缩内部格式不能像指定纹理压缩扩展中指定的那样支持参数组合, 则生成 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标, 并且缓冲区对象的数据存储当前被映射, 则产生 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标, 并且数据将从缓冲区对象中被解包导致内存读取需求超出数据存储大小, 则产生 GL\_INVALID\_OPERATION 错误。

如果 data 没有被以与定义内部压缩格式的扩展规范一致的方式进行编码, 则会产生未定义的结果, 包括异常程序终止。

相关 Get 函数

glGetCompressedTexImage

glGet, 其自变量为 GL\_TEXTURE\_COMPRESSED。

glGet, 其自变量为 GL\_PIXEL\_UNPACK\_BUFFER\_BINDING。

glGetTexLevelParameter, 其自变量为 GL\_TEXTURE\_INTERNAL\_FORMAT 和 GL\_TEXTURE\_COMPRESSED\_IMAGE\_SIZE。

另外查看

glActiveTexture, glCompressedTexImage1D, glCompressedTexImage2D, glCompressedTexImage3D, glCompressedTexSubImage1D, glCompressedTexSubImage2D, glCopyTexImage1D, glCopyTexImage2D, glCopyTexSubImage1D, glCopyTexSubImage2D, glCopyTexSubImage3D, glPixelStore, glTexImage2D, glTexImage3D, glTexSubImage1D, glTexSubImage2D, glTexSubImage3D, glTexParameter

版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

glCopyBufferSubData

将缓冲区对象数据存储的一部分复制到其他缓冲区对象的数据存储中。

C 规范

```
void glCopyBufferSubData(GLenum readtarget,
                        GLenum writetarget,
                        GLintptr readoffset,
                        GLintptr writeoffset,
                        GLsizeiptr size);
```

**参数****readtarget**

指定从哪个目标的数据存储中读取数据。

**writetarget**

指定将数据写入到哪个目标的数据存储中。

**readoffset**

指定 readtarget (数据就应该从中读取的) 的数据存储中以基本机器单元为单位的偏移。

**writeoffset**

指定 writetarget (数据就应该写入其中的) 的数据存储中以基本机器单元为单位的偏移。

**size**

指定从 readtarget 复制到 writetarget 的数据的以基本机器单元为单位的大小。

**描述**

glCopyBufferSubData 将绑定到 readtarget 的数据存储的一部分复制到绑定到 writetarget 的数据存储中。从源位置复制的由 size 指定基本机器单元的数量, 从偏置 readoffset 到目的 writeoffset, 也是以基本机器单元为单位。

readtarget 和 writetarget GL\_ARRAY\_BUFFER、GL\_COPY\_READ\_BUFFER、GL\_COPY\_WRITE\_BUFFER、GL\_ELEMENT\_ARRAY\_BUFFER、GL\_PIXEL\_PACK\_BUFFER、GL\_PIXEL\_UNPACK\_BUFFER、GL\_TEXTURE\_BUFFER、GL\_TRANSFORM\_FEEDBACK\_BUFFER 或 GL\_UNIFORM\_BUFFER。这些目标中任意一个都可能被使用, 虽然特别提供的 GL\_COPY\_READ\_BUFFER 和 GL\_COPY\_WRITE\_BUFFER 目标允许我们在缓冲区之间进行复制而不会影响其他 GL 状态。

Readoffset、writeoffset 和 size 必须都大于或等于 0。此外, readoffset + size 绝不能超出绑定到 readtarget 的缓冲区对象的大小, 而且也绝不能超出绑定到 writetarget 的缓冲区的大小。如果同一个缓冲区对象同时被绑定到 readtarget 和 writetarget, 那么由 readoffset、writeoffset 和 size 指定的范围绝不能重叠。

**注意**

glCopyBufferSubData 只在 3.1 或更高版本的 GL 中可用。

**错误**

如果 readoffset、writeoffset 或 size 中的任何一个为负, 或者如果 readoffset+size 超出绑定到 readtarget 的缓冲区对象的大小, 或者如果 readoffset+size 超出绑定到 writetarget 的缓冲区的大小, 则产生 GL\_INVALID\_VALUE 错误。

如果同一个缓冲区对象同时被绑定到 readtarget 和 writetarget, 并且 [readoffset, readoffset+size) 和 [writeoffset, writeoffset + size) 的范围重叠, 则产生 GL\_INVALID\_VALUE 错误。

如果 0 被绑定到 readtarget 或 writetarget, 则产生 GL\_INVALID\_OPERATION 错误。

如果绑定到 readtarget 或 writetarget 的缓冲区对象被映射, 则产生 GL\_INVALID\_OPERATION 错误。

**另外查看**

glGenBuffers, glBindBuffer, glBufferData, glBufferSubData, glGetBufferSubData

**版权**

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

**glCopyTexImage1D**

将像素复制到一个 1D 纹理图像。

**C 规范**

```
void glCopyTexImage1D(GLenum target,
                    GLint level,
                    GLenum internalformat,
                    GLint x,
```

```
GLint y,
GLsizei width,
GLint border);
```

#### 参数

target

指定目标纹理。必须为 GL\_TEXTURE\_1D。

level

指定层次细节数量。Level 0 是基本图像层次。Level n 是第 n 级 Mip 贴图缩略图。

internalformat

指定纹理的内部格式，必须是下列符号常量中的一个：GL\_COMPRESSED\_RED、GL\_COMPRESSED\_RG、GL\_COMPRESSED\_RGB、GL\_COMPRESSED\_RGBA、GL\_COMPRESSED\_SRGB、GL\_COMPRESSED\_SRGB\_ALPHA、GL\_DEPTH\_COMPONENT、GL\_DEPTH\_COMPONENT16、GL\_DEPTH\_COMPONENT24、GL\_DEPTH\_COMPONENT32、GL\_RED、GL\_RG、GL\_RGB、GL\_R3\_G3\_B2、GL\_RGBA、GL\_RGBA2、GL\_RGBA4、GL\_RGB5\_A1、GL\_RGBA8、GL\_RGB10\_A2、GL\_RGBA12、GL\_RGBA16、GL\_SRGB、GL\_SRGB8、GL\_SRGB\_ALPHA 或 GL\_SRGB8\_ALPHA8。

x

y

指定将要复制的那行像素的左角的窗口坐标。

width

指定纹理图像的宽度。必须是 0 或  $2n + 2$  (border)，其中 n 为整数。

纹理图像的高度为 1。

border

指定边界的宽度。必须为 0 或 1。

#### 描述

glCopyTexImage1D 使用来自当前 GL\_READ\_BUFFER 的像素定义一个一维纹理图像。

左侧角位于(x,y)、宽度为 width + 2 (border)的屏幕对齐像素行在由 level 指定的 Mip 贴图层次定义纹理数组。

internalformat 指定纹理数组的内部格式。

这行中的像素就像调用了 glCopyPixels 一样被处理，但是在最后的变换之前处理过程将停止。这时所有像素分量值都被限定到[0,1]范围，然后被转换到纹理的内部格式，以便存储到 Texel 数组中。

像素的顺序为较低的 x 屏幕坐标对应较低的纹理坐标。

如果当前 GL\_READ\_BUFFER 指定任何像素在当前渲染环境相关联的窗口之外，那么为这些像素所获得的值将是未定义的。

glCopyTexImage1D 使用来自当前 GL\_READ\_BUFFER 的像素定义一个一维纹理图像。

当 internalformat 是一种 sRGB 类型时，GL 不会自动将源像素转换到 sRGB 颜色空间。这样，glPixelMap 函数就能够用来完成这种转换了。

#### 注意

1、2、3 和 4 不是可接受的 internalformat 值。

宽度为 0 的图像表示一个 NULL (无效) 纹理。

#### 错误

如果 tarGet 不是一个允许值，则产生 GL\_INVALID\_ENUM 错误。

如果 level 小于 0，则产生 GL\_INVALID\_VALUE 错误。

如果 level 大于  $\log_2 \max$  值，其中 max 是 GL\_MAX\_TEXTURE\_SIZE 的返回值，则可能产生 GL\_INVALID\_VALUE 错误。

如果 internalformat 不是一个可接受值，则产生 GL\_INVALID\_VALUE 错误。

如果 width 小于 0 或大于  $2 + GL\_MAX\_TEXTURE\_SIZE$ ，则产生 GL\_INVALID\_VALUE 错误。

如果不支持非 2 的幂的纹理且 width 不能被表示成  $2n + 2$  (border) 的形式，其中 n 为整数，则产生 GL\_INVALID\_VALUE 错误。

如果 border 不是 0 或 1，则产生 GL\_INVALID\_VALUE 错误。

如果 internalformat 是 GL\_DEPTH\_COMPONENT、GL\_DEPTH\_COMPONENT16、GL\_DEPTH\_COMPONENT24 或 GL\_DEPTH\_COMPONENT32，并且不存在深度缓冲区，则产生 GL\_INVALID\_OPERATION 错误。

**相关 Get 函数**

glGetTexImage

glIsEnabled, 其自变量为 GL\_TEXTURE\_1D。

**另外查看**

glCopyTexImage2D, glCopyTexSubImage1D, glCopyTexSubImage2D, glPixelStore, glTexImage1D, glTexImage2D, glTexSubImage1D, glTexSubImage2D, glTexParameter

**版权**Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。**glCopyTexImage2D**

将像素复制到一个 2D 纹理图像。

**C 规范**

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

**参数**

target

指定目标纹理。必须为 GL\_TEXTURE\_2D、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Z 或 GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Z。

level

指定层次细节数量。Level 0 是基本图像层次。Level n 是第 n 级 Mip 贴图缩略图。

internalformat

指定纹理的内部格式，必须是下列符号常量中的一个：GL\_COMPRESSED\_RED、GL\_COMPRESSED\_RG、GL\_COMPRESSED\_RGB、GL\_COMPRESSED\_RGBA、GL\_COMPRESSED\_SRGB、GL\_COMPRESSED\_SRGB\_ALPHA、GL\_DEPTH\_COMPONENT、GL\_DEPTH\_COMPONENT16、GL\_DEPTH\_COMPONENT24、GL\_DEPTH\_COMPONENT32、GL\_RED、GL\_RG、GL\_RGB、GL\_R3\_G3\_B2、GL\_RGBA、GL\_RGBA2、GL\_RGBA4、GL\_RGB5\_A1、GL\_RGBA8、GL\_RGB10\_A2、GL\_RGBA12、GL\_RGBA16、GL\_SRGB、GL\_SRGB8、GL\_SRGB\_ALPHA 或 GL\_SRGB8\_ALPHA8。

x

y

指定将要复制的像素矩形区域左下角的窗口坐标。

width

指定纹理图像的宽度。必须是 0 或  $2n + 2$  (border)，其中 n 为整数。

height

指定纹理图像的高度。必须是 0 或  $2n + 2$  (border)，其中 n 为整数。

border

指定边界的宽度。必须为 0 或 1。

**描述**

glCopyTexImage2D 使用来自当前 GL\_READ\_BUFFER 的像素定义一个二维纹理图像或立方体贴图纹理图像。

左下角位于(x,y)、宽度为 width + 2 (border)、高度为 height + 2 (border)的屏幕对齐像素行在由 level 指定的 Mip 贴图层次定义纹理数组。internalformat 指定纹理数组的内部格式。

这个矩形中的像素就像调用了 glCopyPixels 一样被处理，但是在最后的变换之前处理过程将停止。这时所有像素分量值都被限定到[0,1]范围，然后被转换到纹理的内部格式，以便存储到 Texel 数组中。

像素的顺序为较低的  $x$  和  $y$  屏幕坐标对应较低的  $s$  和  $t$  纹理坐标。

如果当前 `GL_READ_BUFFER` 指定矩形中任何像素在当前渲染环境相关联的窗口之外, 那么为这些像素所获得的值将是未定义的。

当 `internalformat` 是一种 sRGB 类型时, GL 不会自动将源像素转换到 sRGB 颜色空间。这样, `glPixelMap` 函数就能够用来完成这种转换了。

**注意**

1、2、3 和 4 不是可接受的 `internalformat` 值。

高度或宽度为 0 的图像表示一个 NULL (无效) 纹理。

**错误**

如果 `target` 不是 `GL_TEXTURE_2D`、`GL_TEXTURE_CUBE_MAP_POSITIVE_X`、`GL_TEXTURE_CUBE_MAP_NEGATIVE_X`、`GL_TEXTURE_CUBE_MAP_POSITIVE_Y`、`GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`、`GL_TEXTURE_CUBE_MAP_POSITIVE_Z` 或 `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`, 则产生 `GL_INVALID_ENUM` 错误。

如果 `level` 小于 0, 则产生 `GL_INVALID_VALUE` 错误。

如果 `level` 大于  $\log_2 \max$  值, 其中  $\max$  是 `GL_MAX_TEXTURE_SIZE` 的返回值, 则可能产生 `GL_INVALID_VALUE` 错误。

如果 `width` 小于 0 或大于  $2 + GL\_MAX\_TEXTURE\_SIZE$ , 则产生 `GL_INVALID_VALUE` 错误。

如果不支持非 2 的幂的纹理且 `width` 或 `depth` 不能被表示成  $2k + 2$  (border) 的形式, 其中  $k$  为整数, 则产生 `GL_INVALID_VALUE` 错误。

如果 `border` 不是 0 或 1, 则产生 `GL_INVALID_VALUE` 错误。

如果 `internalformat` 不是一个可接受的格式, 则产生 `GL_INVALID_VALUE` 错误。

如果 `internalformat` 是 `GL_DEPTH_COMPONENT`、`GL_DEPTH_COMPONENT16`、`GL_DEPTH_COMPONENT24` 或 `GL_DEPTH_COMPONENT32`, 并且不存在深度缓冲区, 则产生 `GL_INVALID_OPERATION` 错误。

**相关 Get 函数**

`glGetTexImage`

`glIsEnabled`, 其自变量为 `GL_TEXTURE_2D` 或 `GL_TEXTURE_CUBE_MAP`。

**另外查看**

`glCopyTexImage1D`, `glCopyTexSubImage1D`, `glCopyTexSubImage2D`, `glPixelStore`, `glTexImage1D`, `glTexImage2D`, `glTexSubImage1D`, `glTexSubImage2D`, `glTexParameter`

**版权**

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glCopyTexSubImage1D

复制一个一维纹理子图像。

**C 规范**

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

**参数**

`target`

指定目标纹理。必须为 `GL_TEXTURE_1D`。

`level`

指定层次细节数量。Level 0 是基本图像层次。Level  $n$  是第  $n$  级 Mip 贴图缩略图。

`xoffset`

指定纹理数组中的 texel 偏移。

`x`

y  
指定将要复制的那行像素的左角的窗口坐标。

width  
指定纹理子图像的宽度。

#### 描述

glCopyTexSubImage1D 用来自当前 GL\_READ\_BUFFER (而不是来自主内存, 就和 glTexSubImage1D 的情况一样) 的像素来代替一维纹理图像的一部分。

左端位于(x,y)且长度为 width 的屏幕对齐像素行代替纹理数组中 x 索引从 xoffset 到 xoffset + width - 1 (包括边界) 的部分。纹理数组中的目的部分不能包括任何最初指定的纹理数组之外的 texel。

这行中的像素就像调用了 glCopyPixels 一样被处理, 但是在最后的变换之前处理过程将停止。这时所有像素分量值都被限定到[0,1]范围, 然后被转换到纹理的内部格式, 以便存储到 Texel 数组中。

指定一个宽度为 0 的子纹理并不是错误, 但是这样做不会产生任何效果。如果当前 GL\_READ\_BUFFER 指定行任何像素在当前渲染环境相关联的读取窗口之外, 那么为这些像素所获得的值将是未定义的。

不能对指定纹理数组的 internalformat、width 或 border 参数, 或者指定子区域外的 texel 值作任何修改。

#### 注意

glPixelStore 会影响纹理图像。

#### 错误

如果 target 不是 GL\_TEXTURE\_1D, 则产生 GL\_INVALID\_ENUM 错误。

如果纹理数组没有被以前的 glTexImage1D 或 glCopyTexImage1D 操作所定义, 则产生 GL\_INVALID\_OPERATION 错误。

如果 level 小于 0, 则产生 GL\_INVALID\_VALUE 错误。

如果 level 大于  $\log_2(\max)$  值, 其中 max 是 GL\_MAX\_TEXTURE\_SIZE 的返回值, 则可能产生 GL\_INVALID\_VALUE 错误。

如果 xoffset <= b 或 (xoffset + width) > (w - b), 其中 w 和 b 分别是被修改的纹理图像的 GL\_TEXTURE\_WIDTH 和 GL\_TEXTURE\_BORDER, 则产生 GL\_INVALID\_VALUE 错误。注意 w 包含边界宽度的两倍。

#### 相关 Get 函数

glGetTexImage

glIsEnabled, 其自变量为 GL\_TEXTURE\_1D。

#### 另外查看

glCopyTexImage1D, glCopyTexImage2D, glCopyTexSubImage2D, glCopyTexSubImage3D, glPixelStore, glReadBuffer, glTexImage1D, glTexImage2D, glTexImage3D, glTexParameter, glTexSubImage1D, glTexSubImage2D, glTexSubImage3D

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glCopyTexSubImage2D

复制一个二维纹理子图像。

#### C 规范

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

#### 参数

target



指定目标纹理。必须为 GL\_TEXTURE\_2D、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Z 或 GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Z。

level

指定层次细节数量。Level 0 是基本图像层次。Level n 是第 n 级 Mip 贴图缩略图。

xoffset

指定纹理数组中 x 方向上一个 texel 的偏移。

yoffset

指定纹理数组中 y 方向上一个 texel 的偏移。

x

y

指定将要复制的像素矩形区域左下角的窗口坐标。

width

指定纹理子图像的宽度。

height

指定纹理子图像的高度。

描述

glCopyTexSubImage2D 用来自当前 GL\_READ\_BUFFER (而不是来自主内存, 就和 glTexSubImage2D 的情况一样) 的像素来代替一个二维纹理图像或一个立方体贴图纹理图像的一个矩形部分。

左下角位于(x, y)且宽度为 width、高度为 height 的屏幕对齐像素矩形代替纹理数组中 x 索引从 xoffset 到 xoffset + width - 1 (包括边界)、y 索引从 yoffset 到 yoffset + height - 1 (包括边界) 的部分, Mip 贴图层次由 level 指定。

这个矩形中的像素就像调用了 glCopyPixels 一样被处理, 但是在最后的变换之前处理过程将停止。这时所有像素分量值都被限定到[0,1]范围, 然后被转换到纹理的内部格式, 以便存储到 Texel 数组中。

纹理数组中的目的矩形不能包含最初指定的纹理数组之外的任何 texel。指定一个宽度或高度为 0 的子纹理并不是错误, 但是这样做不会产生任何效果。

如果当前 GL\_READ\_BUFFER 指定矩形中任何像素在当前渲染环境相关联的读取窗口之外, 那么为这些像素所获得的值将是未定义的。

不能对指定纹理数组的 internalformat、width、height 或 border 参数, 或者指定子区域外的 texel 值作任何修改。

注意

glPixelStore 模式会影响纹理图像。

错误

如果 target 不是 GL\_TEXTURE\_2D、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Z 或 GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Z, 则产生 GL\_INVALID\_ENUM 错误。

如果纹理数组没有被以前的 glTexImage2D 或 glCopyTexImage2D 操作所定义, 则产生 GL\_INVALID\_OPERATION 错误。

如果 level 小于 0, 则产生 GL\_INVALID\_VALUE 错误。

如果 level 大于  $\log_2(\text{max})$  值, 其中 max 是 GL\_MAX\_TEXTURE\_SIZE 的返回值, 则可能产生 GL\_INVALID\_VALUE 错误。

如果 xoffset < -b 或 (xoffset + width) > (w - b), 或者 yoffset < -b 或 (yoffset + height) > (h - b), 其中 w 和 b 分别是被修改的纹理图像的 GL\_TEXTURE\_WIDTH 和 GL\_TEXTURE\_BORDER, 则产生 GL\_INVALID\_VALUE 错误。

注意 w 和 h 包含边界宽度的两倍。

相关 Get 函数

glGetTexImage

glIsEnabled, 其自变量为 GL\_TEXTURE\_2D。

另外查看

glCopyTexImage1D, glCopyTexImage2D, glCopyTexSubImage1D, glCopyTexSubImage3D, glPixelStore, glReadBuffer, glTexImage1D, glTexImage2D, glTexImage3D, glTexParameter, glTexSubImage1D, glTexSubImage2D, glTexSubImage3D

**版权**

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

**glCopyTexSubImage3D**

复制一个三维纹理子图像。

**C 规范**

```
void glCopyTexSubImage3D(GLenum target,
                          GLint level,
                          GLint xoffset,
                          GLint yoffset,
                          GLint zoffset,
                          GLint x,
                          GLint y,
                          GLsizei width,
                          GLsizei height);
```

**参数**

**target**

指定目标纹理。必须为 GL\_TEXTURE\_3D。

**level**

指定层次细节数量。Level 0 是基本图像层次。Level n 是第 n 级 Mip 贴图缩略图。

**xoffset**

指定纹理数组中 x 方向上一个 texel 的偏移。

**yoffset**

指定纹理数组中 y 方向上一个 texel 的偏移。

**zoffset**

指定纹理数组中 z 方向上一个 texel 的偏移。

**x**

**y**

指定将要复制的像素矩形区域左下角的窗口坐标。

**width**

指定纹理子图像的宽度。

**height**

指定纹理子图像的高度。

**描述**

glCopyTexSubImage3D 用来自当前 GL\_READ\_BUFFER (而不是来自主内存, 就和 glTexSubImage3D 的情况一样) 的像素来代替三维纹理图像的一个矩形部分。

左下角位于(x, y)且宽度为 width、高度为 height 的屏幕对齐像素矩形代替纹理数组中 x 索引从 xoffset 到 xoffset + width - 1 (包括边界)、y 索引从 yoffset 到 yoffset + height - 1 (包括边界)、z 索引 zoffset 的部分, Mip 贴图层次由 level 指定。

这个矩形中的像素就像调用了 glCopyPixels 一样被处理, 但是在最后的变换之前处理过程将停止。这时所有像素分量值都被限定到[0,1]范围, 然后被转换到纹理的内部格式, 以便存储到 Texel 数组中。

纹理数组中的目的矩形不能包含最初指定的纹理数组之外的任何 texel。指定一个宽度或高度为 0 的子纹理并不是错误, 但是这样做不会产生任何效果。

如果当前 GL\_READ\_BUFFER 指定矩形中任何像素在当前渲染环境相关联的读取窗口之外, 那么为这些像素所获得的值将是未定义的。

不能对指定纹理数组的 internalformat、width、height、depth 或 border 参数, 或者指定子区域外的 texel 值作任何修改。

**注意**

glPixelStore 模式会影响纹理图像。

## 错误

如果 `target` 不是 `GL_TEXTURE_3D`, 则产生 `GL_INVALID_ENUM` 错误。

如果纹理数组没有被以前的 `glTexImage3D` 操作所定义, 则产生 `GL_INVALID_OPERATION` 错误。

如果 `level` 小于 0, 则产生 `GL_INVALID_VALUE` 错误。

如果 `level` 大于  $\log_2(\text{max})$  值, 其中 `max` 是 `GL_MAX_3D_TEXTURE_SIZE` 的返回值, 则可能产生 `GL_INVALID_VALUE` 错误。

如果  $\text{xoffset} < -b$ ,  $(\text{xoffset} + \text{width}) > (w - b)$ ,  $\text{yoffset} < -b$ ,  $(\text{yoffset} + \text{height}) > (h - b)$ ,  $\text{zoffset} < -b$  或  $\text{zoffset} > (d - b)$ , 其中 `w`, `h`, `d` 和 `b` 分别是被修改的纹理图像的 `GL_TEXTURE_WIDTH`, `GL_TEXTURE_HEIGHT`, `GL_TEXTURE_DEPTH` 和 `GL_TEXTURE_BORDER`, 则产生 `GL_INVALID_VALUE` 错误。注意 `w`, `h` 和 `d` 包含边界宽度的两倍。

## 相关 Get 函数

`glGetTexImage`

`glIsEnabled`, 其自变量为 `GL_TEXTURE_3D`。

## 另外查看

`glCopyTexImage1D`, `glCopyTexImage2D`, `glCopyTexSubImage1D`, `glCopyTexSubImage2D`, `glPixelStore`, `glReadBuffer`, `glTexImage1D`, `glTexImage2D`, `glTexImage3D`, `glTexParameter`, `glTexSubImage1D`, `glTexSubImage2D`, `glTexSubImage3D`

## 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glCreateProgram

创建一个程序对象。

## C 规范

```
GLuint glCreateProgram(void);
```

## 描述

`glCreateProgram` 创建一个空程序对象并返回一个它可以引用的非 0 值。程序对象是着色器对象能够绑定到的对象。这提供了一种机制, 能够指定将要进行连接的着色器对象来创建一个程序。它还提供了一种方法来检查将要用来创建程序的着色器的兼容性 (例如检查一个顶点着色器和一个片段着色器之间的兼容性)。当作为程序对象的一部分而不再需要时, 着色器对象可以被分离。

通过使用 `glAttachShader` 成功地将着色器对象绑定到程序对象, 成功地使用 `glCompileShader` 来编译这些着色器对象, 成功地使用 `glLinkProgram` 来连接这些着色器对象, 能够在这个程序对象中会创建一个或多个可执行文件。这些可执行文件在 `glUseProgram` 被调用时成为当前状态的一部分。可以调用 `glDeleteProgram` 来删除程序对象。

当程序对象相关内存不再是任何环境的当前渲染状态的一部分时, 它将被删除。

## 注意

就像显示列表和纹理对象一样, 程序对象的名字空间可以在一组环境之间进行共享, 只要环境的服务器端共享同一个地址空间就可以。如果名字空间在环境之间共享, 任何绑定的对象和这些绑定对象相关的数据也会被共享。

在对象被不同的执行线程访问时, 应用程序负责提供 API 调用之间的同步。

## 错误

如果在创建程序对象时发生错误, 这个函数返回 0。

## 相关 Get 函数

`glGet`, 自变量为 `GL_CURRENT_PROGRAM`

`glGetActiveAttrib`, 其参数包含一个合法程序对象和一个活动的属性变量的索引。

`glGetActiveUniform`, 其参数包含一个合法程序对象和一个活动的统一变量的索引。

`glGetAttachedShaders`, 其参数包含一个合法程序对象。

`glGetAttribLocation`, 其参数包含一个合法程序对象和一个属性变量的名称。

`glGetProgram`, 其参数包含一个合法程序对象和将要查询的参数。

`glGetProgramInfoLog`, 其参数包含一个合法程序对象。

glGetUniform, 其参数包含一个合法程序对象和一个统一变量的位置。

glGetUniformLocation, 其参数包含一个合法程序对象和一个统一变量的名称。

#### 另外查看

glAttachShader, glBindAttribLocation, glCreateShader, glDeleteProgram, glDetachShader, glLinkProgram, glUniform, glUseProgram, glValidateProgram

#### 版权

Copyright © 2003–2005 3Dlabs Inc. Ltd. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glCreateShader

创建一个着色器对象。

#### C 规范

```
GLuint glCreateShader(GLenum shaderType);
```

#### 参数

shaderType

指定要创建的着色器类型。必须是 GL\_VERTEX\_SHADER、GL\_GEOMETRY\_SHADER 或 GL\_FRAGMENT\_SHADER。

#### 描述

glCreateShader 创建一个空着色器对象并返回一个它可以引用的非 0 值。着色对象被用来保持定义一个着色器的源代码字符串。

shaderType 指出了将要被创建的着色器的类型。一共可以支持 3 种类型的着色器。GL\_VERTEX\_SHADER 类型的着色器是在可编程顶点处理器运行的。

GL\_GEOMETRY\_SHADER 类型的着色器是在可编程几何图形处理器运行的。GL\_FRAGMENT\_SHADER 类型的着色器是在可编程片段处理器运行的。

在创建时, 一个着色器对象的 GL\_SHADER\_TYPE 参数被设置为 GL\_VERTEX\_SHADER、GL\_GEOMETRY\_SHADER 或 GL\_FRAGMENT\_SHADER, 这取决于 shaderType 的值。

#### 注意

就像显示列表和纹理对象一样, 着色器对象的名字空间可以在一组环境之间进行共享, 只要环境的服务器端共享同一个地址空间就可以。如果名字空间在环境之间共享, 任何绑定的对象和这些绑定对象相关的数据也会被共享。

在对象被不同的执行线程访问时, 应用程序负责提供 API 调用之间的同步。

#### 错误

如果在创建着色器对象时发生错误, 这个函数返回 0。

如果 shaderType 不是一个可接受的值, 则产生 GL\_INVALID\_ENUM 错误。

#### 相关 Get 函数

glGetShader, 其参数包含一个合法着色器对象和将要查询的参数。

glGetShaderInfoLog, 其参数包含一个合法着色器对象。

glGetShaderSource, 其参数包含一个合法着色器对象 glIsShader。

#### 另外查看

glAttachShader, glCompileShader, glDeleteShader, glDetachShader, glShaderSource

#### 版权

Copyright © 2003–2005 3Dlabs Inc. Ltd. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glCullFace

指定前向和背向的面是否能够进行剔除。

#### C 规范

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

#### 参数

**mode**

指定前面和背面是否能够进行剔除。符号常量 GL\_FRONT、GL\_BACK 和 GL\_FRONT\_AND\_BACK 可以被接受。初始值为 GL\_BACK。

**描述**

glCullFace 指定在启用表面剔除时前面和背面是否进行剔除 (由 mode 指定)。初始状态下表面剔除是关闭的。可以调用参数为 GL\_CULL\_FACE 的 glEnable 和 glDisable 命令来激活和关闭表面剔除。这些表面包括三角形、四边形、多边形和矩形。

glFrontFace 指定顺时针和逆时针方向的哪一个面为前向和背向的。参见 glFrontFace。

**注意**

如果 mode 为 GL\_FRONT\_AND\_BACK, 将不会绘制任何表面, 但其他图元 (例如点和线) 将会被绘制。

**错误**

如果 mode 不是一个可接受的值, 则产生 GL\_INVALID\_ENUM 错误。

**相关 Get 函数**

glIsEnabled, 其自变量为 GL\_CULL\_FACE。

glGet, 其自变量为 GL\_CULL\_FACE\_MODE。

**另外查看**

glEnable, glFrontFace

**版权**

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glDeleteBuffers

删除指定的缓冲区对象。

**C 规范**

```
void glDeleteBuffers(GLsizei n,
                    const GLuint * buffers);
```

**参数**

**n**

指定将要被删除的缓冲区对象的数量。

**buffers**

指定将要被删除的缓冲区对象的一个序列。

**描述**

glDeleteBuffers 删除由 buffers 数组元素指定的 n 个缓冲区对象。在一个缓冲区对象被删除以后, 它将不再有任何内容, 并且它的名字已经可以进行重用了 (例如被 glGenBuffers 重用)。如果一个当前被绑定的缓冲区对象被删除, 绑定 (binding) 将回复到 0 (任何缓冲区对象的缺失)。

glDeleteBuffers 默认忽略 0 和不符合现有缓冲区对象的名称。

**错误**

如果 n 为负值, 则产生 GL\_INVALID\_VALUE 错误。

**相关 Get 函数**

glIsBuffer

**另外查看**

glBindBuffer, glGenBuffers, glGet

**版权**

Copyright © 2005 Addison–Wesley. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。 <http://opencontent.org/openpub/>。

## glDeleteFramebuffers

删除帧缓冲区对象。

### C 规范

```
void glDeleteFramebuffers(GLsizei n,
                          GLuint *framebuffers);
```

### 参数

n

指定将要被删除的帧缓冲区对象的数量。

framebuffers

指向一个包含 n 个要删除的帧缓冲区对象的指针。

### 描述

glDeleteFramebuffers 删除 n 个帧缓冲区对象，这些帧缓冲区对象的名称被存储在由 framebuffers 进行寻址的数组中。GL 保留了名称 zero (0)，并且将它默认忽略，和其他未使用的名称一样，它也应该出现在 framebuffers 中。一旦一个帧缓冲区被删除，它的名称就会再次变为未使用的，并且没有进行任何绑定。GL\_READ\_FRAMEBUFFER 被删除，就像 glBindFramebuffer 已经以相应的 target 和 framebuffer zero 执行了一样。

### 错误

如果 n 为负值，则产生 GL\_INVALID\_VALUE 错误。

### 另外查看

glGenFramebuffers, glBindFramebuffer, glCheckFramebufferStatus

### 版权

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glDeleteProgram

删除一个程序对象。

### C 规范

```
void glDeleteProgram(GLuint program);
```

### 参数

program

指定将要被删除的程序对象。

### 描述

glDeleteProgram 释放内存，并使由 program 指定的程序对象相关的名字无效。这个命令有效地撤销 glCreateProgram 调用的影响。

如果一个程序对象被当作当前渲染状态的一部分来使用，那么它将为删除做标记，但在它不再是任何渲染环境当前状态的一部分之前都不会被删除。如果有着色器对象绑定到要删除的程序对象上，这些着色器对象将被自动分离，但如果没有被之前调用 glDeleteShader 作删除标记的话，它们将不会被删除。program 的 0 值将被默认忽略。

可以调用自变量为 program 和 GL\_DELETE\_STATUS 的 glGetProgram 来确定一个程序对象是否已经被标记为删除。

### 错误

如果 program 不是由 OpenGL 产生的值，则产生 GL\_INVALID\_VALUE 错误。

### 相关 Get 函数

glGet，其自变量为 GL\_CURRENT\_PROGRAM。

glGetProgram，其自变量为 program 和 GL\_DELETE\_STATUS。

glIsProgram

### 另外查看

glCreateShader, glDetachShader, glUseProgram

## 版权

Copyright © 2003–2005 3Dlabs Inc. Ltd. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glDeleteQueries

删除指定的查询对象。

### C 规范

```
void glDeleteQueries(GLsizei n,
                    const GLuint * ids);
```

### 参数

n

指定将要被删除的查询对象的数量。

ids

指定将要被删除的查询对象的一个序列。

### 描述

glDeleteQueries 删除由 ids 数组元素指定的 n 个缓冲区对象。在一个缓冲区对象被删除以后，它将不再有任何内容，并且它的名字已经可以进行重用了（例如被 glGenQueries 重用）。

glDeleteQueries 默认忽略 0 和不符合现有查询对象的名称。

### 错误

如果 n 为负值，则产生 GL\_INVALID\_VALUE 错误。

如果 glDeleteQueries 在 glBegin 执行和相应的 glEnd 执行之间执行，则产生 GL\_INVALID\_OPERATION 错误。

### 相关 Get 函数

glIsQuery

### 另外查看

glBeginQuery, glEndQuery, glGenQueries, glGetQueryiv, glGetQueryObject

### 版权

Copyright © 2005 Addison–Wesley. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glDeleteRenderbuffers

删除渲染缓冲区对象。

### C 规范

```
void glDeleteRenderbuffers(GLsizei n,
                          GLuint *renderbuffers);
```

### 参数

n

指定将要被删除的渲染缓冲区对象的数量。

renderbuffers

指向一个包含 n 个要删除的渲染缓冲区对象的指针。

### 描述

glDeleteRenderbuffers 删除 n 个渲染缓冲区对象，这些渲染缓冲区对象的名称被存储在由 renderbuffers 进行寻址的数组中。GL 保留了名称 zero(0)，并且将它默认忽略，和其他未使用的名称一样，它也应该出现在 renderbuffers 中。一旦一个渲染缓冲区被删除，它的名称就会再次变为未使用的，并且不包含任何内容。如果一个当前被绑定到 GL\_RENDERBUFFER 目标的渲染缓冲区被删除，就像 glBindRenderbuffer 已经以 GL\_RENDERBUFFER 为 target 和 framebuffer zero 执行了一样。

如果一个渲染缓冲区对象被绑定到当前绑定的帧缓冲区中的一个或多个绑定，那么就会像调用了 glFramebufferRenderbuffer 一样，其中这个图像在当前绑定的帧缓冲区中绑定到的每个绑定点的 renderbuffer 为

zero。也就是说，这个渲染缓冲区对象首先从当前绑定的帧缓冲区 zhogn 的所有绑定点解除绑定。请注意，渲染缓冲区图像尤其不会从任何非绑定的帧缓冲区解除绑定。

#### 错误

如果  $n$  为负值，则产生 `GL_INVALID_VALUE` 错误。

#### 另外查看

`glGenRenderbuffers`, `glFramebufferRenderbuffer`, `glRenderbufferStorage`, `glRenderbufferStorageMultisample`

#### 版权

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glDeleteSamplers

删除指定的采样器对象。

### C 规范

```
void glDeleteSamplers(GLsizei n,
                      const GLuint * ids);
```

### 参数

$n$

指定将要被删除的采样器对象的数量。

$ids$

指定将要被删除的采样器对象的一个序列。

### 描述

`glDeleteSamplers` 删除由  $ids$  数组元素指定的  $n$  个采样器对象。在一个采样器对象被删除之后，它的名称就会再次变为未使用的。如果一个当前被绑定到一个采样器单元的采样器对象被删除，那么就会像调用了 `unit` 被设置为采样器绑定到的单元和采样器 zero 的 `glBindSampler`。采样器中未使用的名称将被默认忽略，就像保留名 zero 一样。

### 注意

`glDeleteSamplers` 只在 3.3 或更高版本的 GL 中可用。

### 错误

如果  $n$  为负值，则产生 `GL_INVALID_VALUE` 错误。

### 相关 Get 函数

`glIsSampler`

### 另外查看

`glGenSamplers`, `glBindSampler`, `glDeleteSamplers`, `glIsSampler`

### 版权

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glDeleteShader

删除一个着色器对象。

### C 规范

```
void glDeleteShader(GLuint shader);
```

### 参数

$shader$

指定将要被删除的着色器对象。

### 描述

`glDeleteShader` 释放内存，并使由  $shader$  指定的着色器对象相关的名字无效。这个命令有效地撤销 `glCreateShader` 调用的影响。

如果一个着色器对象被绑定到一个程序对象，那么它将为删除做标记，但在它不再绑定到任何渲染环境的任何程序对象之前都不会被删除（也就是说，它必须在被删除之前解除到它所绑定到的任何地方的绑定）。shader 的 0 值将被默认忽略。

可以调用自变量为 shader 和 GL\_DELETE\_STATUS 的 glGetShader 来确定一个对象是否已经被标记为删除。

**错误**

如果 shader 不是由 OpenGL 产生的值，则产生 GL\_INVALID\_VALUE 错误。

**相关 Get 函数**

glGetAttachedShaders 以将要被查询的程序对象为参数。

glGetShader，其自变量为 shader 和 GL\_DELETE\_STATUS。

glIsShader

**另外查看**

glCreateProgram, glCreateShader, glDetachShader, glUseProgram

**版权**

Copyright © 2003–2005 3Dlabs Inc. Ltd. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glDeleteSync

就删除一个同步对象。

**C 规范**

```
void glDeleteSync(GLsync sync);
```

**参数**

sync

将要被删除的同步对象。

**描述**

glDeleteSync 删除由 sync 指定的同步对象。如果与指定同步对象对应的围栏命令已经完成，或者没有 glWaitSync 或 glClientWaitSync 命令被 sync 阻塞，那么这个对象将会立即被删除。否则，sync 将会被标记为删除，并且将会在已经不与任何围栏命令相关联，并且不再阻塞任何 glWaitSync 或 glClientWaitSync 命令时被删除。在任何一种情况下，当 glDeleteSync 返回之后，sync 名称将会无效，并且不再能够用于引用同步对象。

glDeleteSync 将会默认忽略一个 sync 值 zero。

**注意**

glSync 在 3.2 或更高版本的 GL 中，或者在支持 ARB\_sync 扩展的情况下可用。

**错误**

如果 sync 不是 zero，也不是一个同步对象的名称，则产生 GL\_INVALID\_VALUE 错误。

**另外查看**

glFenceSync, glWaitSync, glClientWaitSync

**版权**

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glDeleteTextures

删除指定的纹理。

**C 规范**

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

**参数**

n

指定将要被删除的纹理的数量。

textures

指定将要被删除的纹理的一个序列。

描述

glDeleteTextures 删除由 textures 数组元素指定的 n 个纹理。在一个纹理被删除以后, 它将不再有任何内容或维度, 并且它的名字已经可以进行重用了(例如被 glGenTextures 重用)。如果一个当前被绑定的纹理被删除, 绑定将回复到 0 (默认纹理)。

glDeleteTextures 默认忽略 0 和不符合现有纹理的名称。

错误

如果 n 为负值, 则产生 GL\_INVALID\_VALUE 错误。

相关 Get 函数

glIsTexture

另外查看

glBindTexture, glCopyTexImage1D, glCopyTexImage2D, glGenTextures, glGet, glGetTexParameter, glTexImage1D, glTexImage2D, glGetTexParameter

版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glDeleteVertexArrays

删除顶点数组对象。

C 规范

```
void glDeleteVertexArrays(GLsizei n,
                          const GLuint * arrays);
```

参数

n

指定将要被删除的顶点数组对象的数量。

arrays

指定一个包含将要被删除的 n 个对象名称的数组的地址。

描述

glDeleteVertexArrays 删除 n 个顶点数组对象, 这些顶点数组对象的名称被存储在由 arrays 进行寻址的数组中。一旦一个顶点数组对象被删除, 它就不再包含任何内容, 并且它的名称就会再次变为未使用的。如果一个当前被绑定的顶点数组对象被删除, 那么这个对象的绑定将回复到 0, 并且默认顶点数组将会变成当前顶点数组。arrays 中未使用的名称将被默认忽略, 就像 zero 值一样。

错误

如果 n 为负值, 则产生 GL\_INVALID\_VALUE 错误。

另外查看

glGenVertexArrays, glIsVertexArray, glBindVertexArray

版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。 <http://opencontent.org/openpub/>。

## glDepthFunc

指定用于深度缓冲区比较 (depth buffer comparison) 的值。

C 规范

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

**参数**

func

指定深度比较函数。符号常量 GL\_NEVER、GL\_LESS、GL\_EQUAL、GL\_LEQUAL、GL\_GREATER、GL\_NOTEQUAL、GL\_GEQUAL 和 GL\_ALWAYS 可以被接受。初始值为 GL\_LESS。

**描述**

glDepthFunc 指定用来将每个输入的像素深度值与深度缓冲区中出现的深度值进行比较的函数。只有在深度测试被激活的情况下才能执行这种比较。

(参见 GL\_DEPTH\_TEST 的 glEnable 和 glDisable)。func 指定像素将会被绘制的条件。比较函数如下。

GL\_NEVER: 总是不通过。

GL\_LESS: 在输入深度值小于存储的深度值时通过。

GL\_EQUAL: 在输入深度值等于存储的深度值时通过。

GL\_LEQUAL: 在输入深度值小于或等于存储的深度值时通过。

GL\_GREATER: 在输入深度值大于存储的深度值时通过。

GL\_NOEQUAL: 在输入深度值不等于存储的深度值时通过。

GL\_GEQUAL: 在输入深度值大于或等于存储的深度值时通过。

GL\_ALWAYS: 总是通过。

func 的初始值为 GL\_LESS。在初始条件下, 深度测试是被禁用的。如果深度测试被禁用, 或者不存在深度测试, 那么就相当于深度测试总是通过一样。

**注意**

即使在深度缓冲区存在、深度遮罩非 0 的情况下, 如果深度测试被禁用, 深度缓冲区也不会被更新。如果要无条件地写入到深度缓冲区中, 那么深度测试应该被激活, 并且设置为 GL\_ALWAYS。

**错误**

如果 func 不是一个可接受的值, 则产生 GL\_INVALID\_ENUM 错误。

**相关 Get 函数**

glGet, 其自变量为 GL\_DEPTH\_FUNC。

glIsEnabled, 其自变量为 GL\_DEPTH\_TEST。

**另外查看**

glDepthRange, glEnable, glPolygonOffset

**版权**

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

**glDepthMask**

激活或禁用深度缓冲区写入。

**C 规范**

```
void glDepthMask(GLboolean flag);
```

**参数**

flag

指定深度缓冲区是否激活写入。如果 flag 为 GL\_FALSE, 深度缓冲区写入被禁用。否则, 深度缓冲区被激活。在默认情况下, 深度缓冲区写入是激活的。

**描述**

glDepthMask 指定深度缓冲区是否激活写入。如果 flag 为 GL\_FALSE, 深度缓冲区写入被禁用。否则, 深度缓冲区被激活。在默认情况下, 深度缓冲区写入是激活的。

**相关 Get 函数**

glGet, 其自变量为 GL\_DEPTH\_WRITEMASK。

**另外查看**

glColorMask, glDepthFunc, glDepthRange, glStencilMask

**版权**

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

**glDepthRange**

指定从标准化设备坐标到窗口坐标的深度值映射。

**C 规范**

```
void glDepthRange(GLclampd nearVal,
                  GLclampd farVal);
```

**参数**

nearVal

指定近剪切板到窗口坐标的映射。初始值为 0。

farVal

指定远剪切板到窗口坐标的映射。初始值为 1。

**描述**

在进行剪切并除以  $w$  之后，深度坐标范围为 -1 到 1，对应于近剪切板和远剪切板。glDepthRange 指定这个范围内的标准化深度坐标到窗口深度坐标的线性映射。不管实际深度缓冲区实现如何，窗口坐标深度值都会被认为是在从 0 到 1 的范围内（就像颜色分量一样）。这样，glDepthRange 所接受的值在被接受之前都被截取到了这个范围内。

(0,1) 的设置将近剪切板映射到 0，而将远剪切板映射到 1。在这种映射下，深度缓冲区范围被充分利用了。

**注意**

nearVal 并不一定要小于 farVal 的。相反的映射，例如 nearVal = 1、farVal = 0，也是可以接受的。

**相关 Get 函数**

glGet，其自变量为 GL\_DEPTH\_RANGE。

**另外查看**

glDepthFunc, glPolygonOffset, glViewport

**版权**

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

**glDetachShader**

将一个着色器对象从它绑定到的程序对象分离开来。

**C 规范**

```
void glDetachShader(GLuint program,
                    GLuint shader);
```

**参数**

program

指定从哪个程序对象来分离着色器对象。

shader

指定要分离的着色器对象。

**描述**

glDetachShader 将由 shader 指定的着色器对象从由 program 指定的程序对象分离开来。这个命令能够用来撤销 glAttachShader 命令的结果。

如果 shader 已经通过调用 glDeleteShader 来做了删除标记，并且没有被绑定到任何其他程序对象，那么它将在分离之后被删除。

**错误**

如果 program 或 shader 不是由 OpenGL 产生的值，则产生 GL\_INVALID\_VALUE 错误。

如果 program 不是一个程序对象，则产生 GL\_INVALID\_OPERATION 错误。

如果 shader 不是一个着色器对象, 则产生 GL\_INVALID\_OPERATION 错误。

如果 shader 没有绑定到 program, 则产生 GL\_INVALID\_OPERATION 错误。

#### 相关 Get 函数

glGetAttachedShaders 带有合法程序对象的句柄。

glGetShader, 其自变量为 shader 和 GL\_DELETE\_STATUS。

glIsProgram

glIsShader

#### 另外查看

glAttachShader

#### 版权

Copyright © 2003–2005 3Dlabs Inc. Ltd. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glDrawArrays

从数组数据渲染图元。

### C 规范

```
void glDrawArrays(GLenum mode,
                  GLint first,
                  GLsizei count);
```

#### 参数

mode

指定将要渲染哪种类型的图元。符号常量 GL\_POINTS、GL\_LINE\_STRIP、GL\_LINE\_LOOP、GL\_LINES、GL\_LINE\_STRIP\_ADJACENCY、GL\_LINES\_ADJACENCY、GL\_TRIANGLE\_STRIP、GL\_TRIANGLE\_FAN、GL\_TRIANGLES、GL\_TRIANGLE\_STRIP\_ADJACENCY 和 GL\_TRIANGLES\_ADJACENCY 都是可接受的。

first

指定激活的数组中的起始索引。

count

指定将要被渲染的索引数量。

#### 描述

glDrawArrays 通过进行很少的子例程调用来指定多个几何图元。我们可以预先指定单独的定点、法线和颜色数组, 并通过调用一次 glDrawArrays 来使用它们构造一个图元序列, 而不是通过调用一个 GL 程序来传输每个单独的顶点、法线、纹理坐标、边缘标记或颜色。

用 glDrawArrays 时, 他使用 count 时序元素来从每个激活的数组构造一个几何图元序列, 从元素 first 开始。mode 指定要构造哪种图元, 以及如何用这些数组元素来构造这些图元。

由 glDrawArrays 修改的顶点属性在 glDrawArrays 返回后会有一个未指定值。没有被修改过的属性将被保持为良好定义的。

#### 注意

GL\_LINE\_STRIP\_ADJACENCY、GL\_LINES\_ADJACENCY、GL\_TRIANGLE\_STRIP\_ADJACENCY 和 GL\_TRIANGLES\_ADJACENCY 只在 3.2 或更高版本的 GL 中可用。

#### 错误

如果 mode 不是一个可接受的值, 则产生 GL\_INVALID\_ENUM 错误。

如果 count 为负值, 则产生 GL\_INVALID\_VALUE 错误。

如果非 0 缓冲区对象名称被绑定到一个激活的数组, 并且缓冲区对象的数据存储当前被映射, 则产生 GL\_INVALID\_OPERATION 错误。

如果一个几何图形着色器被激活, 并且 mode 与当前安装的程序对象中几何图形着色器的输入图元类型不兼容, 那么将生成 GL\_INVALID\_OPERATION 错误。

#### 另外查看

glDrawArraysInstanced, glDrawElements, glDrawRangeElements.

**版权**

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

**glDrawArraysInstanced**

绘制一个元素范围的多个实例。

**C 规范**

```
void glDrawArraysInstanced(GLenum mode,
                           GLint first,
                           GLsizei count,
                           GLsizei primcount);
```

**参数**

mode

指定将要渲染哪种类型的图元。符号常量 GL\_POINTS、GL\_LINE\_STRIP、GL\_LINE\_LOOP、GL\_LINES、GL\_TRIANGLE\_STRIP、GL\_TRIANGLE\_FAN、GL\_TRIANGLES、GL\_LINES\_ADJACENCY、GL\_LINE\_STRIP\_ADJACENCY、GL\_TRIANGLES\_ADJACENCY 和 GL\_TRIANGLE\_STRIP\_ADJACENCY 都是可接受的。

first

指定激活的数组中的起始索引。

count

指定将要被渲染的索引数量。

primcount

指定将要被渲染的指定范围索引的实例的数量。

**描述**

glDrawArraysInstanced 的行为与 glDrawArrays 相同，除了执行元素范围的 primcount 实例，以及内部计数器 instanceID 的值在每次迭代时都会递增之外。instanceID 是一个内部 32 位整型计数器，可以由 gl\_InstanceID 这样的顶点着色器进行读取。

glDrawArraysInstanced 的效果与下列代码相同。

```
<![CDATA[ if ( mode or count is invalid )
generate appropriate error
else {
for (int i = 0; i < primcount ; i++) {
instanceID = i;
glDrawArrays(mode, first, count);
}
instanceID = 0;
}]]>
```

**注意**

glDrawArraysInstanced 只在 3.1 或更高版本的 GL 中可用。

GL\_LINE\_STRIP\_ADJACENCY、GL\_LINES\_ADJACENCY、GL\_TRIANGLE\_STRIP\_ADJACENCY 和 GL\_TRIANGLES\_ADJACENCY 只在 3.2 或更高版本的 GL 中可用。

**错误**

如果 mode 不是一个可接受的值，则产生 GL\_INVALID\_ENUM 错误。

如果一个几何图形着色器被激活，并且 mode 与当前安装的程序对象中几何图形着色器的输入图元类型不兼容，那么将生成 GL\_INVALID\_OPERATION 错误。

如果 count 或者 primcount 为负值，则产生 GL\_INVALID\_VALUE 错误。

如果非 0 缓冲区对象名称被绑定到一个激活的数组，并且缓冲区对象的数据存储当前被映射，则产生 GL\_INVALID\_OPERATION 错误。

**另外查看**

glDrawArrays, glDrawElementsInstanced

**版权**

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glDrawBuffer

指定绘制到哪个颜色缓冲区。

### C 规范

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

#### 参数

mode

指定最多 4 个将要被绘制到的颜色缓冲区。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 和 GL\_FRONT\_AND\_BACK 都是可以接受的。单缓冲区环境的初始值为 GL\_FRONT，而双缓冲区环境的初始值为 GL\_BACK。

#### 描述

在将颜色写入到帧缓冲区时，这些颜色被写入到由 glDrawBuffer 指定的颜色或缓冲区。这些规范如下所示。

GL\_NONE：不写入任何颜色缓冲区。

GL\_FRONT\_LEFT：只写入左前颜色缓冲区。

GL\_FRONT\_RIGHT：只写入右前颜色缓冲区。

GL\_BACK\_LEFT：只写入左后颜色缓冲区。

GL\_BACK\_RIGHT：只写入右后颜色缓冲区。

GL\_FRONT：只写入左前和右前颜色缓冲区。如果没有右前颜色缓冲区，则只写入左前颜色缓冲区。

GL\_BACK：只写入左后和右后颜色缓冲区。如果没有右后颜色缓冲区，则只写入左后颜色缓冲区。

GL\_LEFT：只写入左前和左后颜色缓冲区。如果没有左后颜色缓冲区，则只写入左前颜色缓冲区。

GL\_RIGHT：只写入右前和右后颜色缓冲区。如果没有右后颜色缓冲区，则只写入右前颜色缓冲区。

GL\_FRONT\_AND\_BACK：写入所有前后颜色缓冲区（左前、右前、左后、右后）。如果没有后颜色缓冲区，则只写入左前和右前颜色缓冲区。如果没有右颜色缓冲区，则只写入左前和左后颜色缓冲区。如果没有右颜色缓冲区和后颜色缓冲区，则只写入左前颜色缓冲区。

如果选择了多个颜色缓冲区来进行绘制，那么每个颜色缓冲区将单独计算和请求混合和逻辑操作，并且在每个缓冲区内将产生不同的结果。

单视场环境（monoscopic context）只包含 left 缓冲区，而立体视场环境（stereoscopic context）则包含 left 和 right 缓冲区。同样，单缓冲区环境只包含 front 缓冲区，而双缓冲区环境则包含 front 和 back 缓冲区。环境是在 GL 初始化时进行选择的。

#### 错误

如果 mode 不是一个可接受的值，则产生 GL\_INVALID\_ENUM 错误。

如果不存在由 mode 表示的缓冲，则产生 GL\_INVALID\_OPERATION 错误。

#### 相关 Get 函数

glGet，其自变量为 GL\_DRAW\_BUFFER。

#### 另外查看

glBlendFunc, glColorMask, glDrawBuffers, glLogicOp, glReadBuffer

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 http://oss.sgi.com/projects/FreeB/。

## glDrawBuffers

指定一个要绘制到的颜色缓冲区的列表。

### C 规范

```
void glDrawBuffers(GLsizei n,
                   const GLenum * bufs);
```

**参数**

n

指定 bufs 中的缓冲区数量。

bufs

指向一个符号常量数组, 指定将要向哪个缓冲区中写入片段颜色或数据值。

**描述**

glDrawBuffers 定义一个要将片段着色器数据写入到其中的颜色缓冲区的列表。如果一个片段着色器将一个值写入一个或多个用户定义的输出变量, 那么每个变量的值将被写入指定到 bufs 中一个与分配给这个用户定义输出的位置相对应的位置的缓冲区。分配到大于或等于 n 的位置的用户定义输出所使用的绘制缓冲区将被隐式地设置为 GL\_NONE, 并且任何写入到这样一个输出的数据都将被丢弃。

bufs 中包含的符号常量可以是下列任何一个。

GL\_NONE: 片段着色器输出值不会写入任何颜色缓冲区。

GL\_FRONT\_LEFT: 片段着色器输出值写入左前颜色缓冲区。

GL\_FRONT\_RIGHT: 片段着色器输出值写入右前颜色缓冲区。

GL\_BACK\_LEFT: 片段着色器输出值写入左后颜色缓冲区。

GL\_BACK\_RIGHT: 片段着色器输出值写入右后颜色缓冲区。

GL\_COLOR\_ATTACHMENTn: 片段着色器输出值被写入当前帧缓冲区的第 n 个颜色绑定。n 的取值范围可以从 0 到 GL\_MAX\_COLOR\_ATTACHMENTS 的值。

除 GL\_NONE 以外, 上述符号常量可能在 bufs 中不会出现多于一次。支持的最大绘图缓冲区数与实现有关, 可以通过调用自变量为 GL\_MAX\_DRAW\_BUFFERS 的 glGet 来进行查询。

**注意**

由于符号常量 GL\_FRONT、GL\_BACK、GL\_LEFT、GL\_RIGHT 和 GL\_FRONT\_AND\_BACK 可能引用多个缓冲区, 所以它们在 bufs 数组中是不允许的。

如果一个片段着色器不写入一个用户定义输出变量, 那么着色器执行后的片段颜色值将是未定义的。对于在这种情况下生成的每个片段来说, 一个不同的值可能会被写入由 bufs 指定的每一个缓冲区。

**错误**

如果 bufs 中的某个值不是可接受的值, 则产生 GL\_INVALID\_ENUM 错误。

如果 GL 被绑定到默认帧缓冲区, 并且 bufs 中的一个或多个值是 GL\_COLOR\_ATTACHMENTn 标记中的一个, 则产生 GL\_INVALID\_ENUM 错误。

如果 GL 被绑定到一个帧缓冲区对象, 并且 bufs 中的一个或多个值是 GL\_NONE 或 GL\_COLOR\_ATTACHMENTSn 标记之外的任意值, 则产生 GL\_INVALID\_ENUM 错误。

如果 n 小于 0, 则产生 GL\_INVALID\_ENUM 错误。

如果 GL\_NONE 以外的一个符号常量在 bufs 中出现了不止一次, 则产生 GL\_INVALID\_OPERATION 错误。

如果 bufs 中的任何入口 (除 GL\_NONE 以外) 表示一个在当前 GL 环境中不存在的颜色缓冲区, 则产生 GL\_INVALID\_OPERATION 错误。

如果 n 大于 GL\_MAX\_DRAW\_BUFFERS, 则产生 GL\_INVALID\_VALUE 错误。

**相关 Get 函数**

glGet, 其自变量为 GL\_MAX\_DRAW\_BUFFERS。

glGet, 其自变量为 GL\_DRAW\_BUFFERSi (其中 i 表示其值被查询的绘图缓冲区的数量)。

**另外查看**

glBlendFunc, glColorMask, glDrawBuffers, glLogicOp, glReadBuffer

**版权**

Copyright © 2003–2005 3Dlabs Inc. Ltd. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

**glDrawElements**

从数组数据渲染图元。

**C 规范**

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

#### 参数

mode

指定将要渲染哪种类型的图元。符号常量 GL\_POINTS、GL\_LINE\_STRIP、GL\_LINE\_LOOP、GL\_LINES、GL\_LINE\_STRIP\_ADJACENCY、GL\_LINES\_ADJACENCY、GL\_TRIANGLE\_STRIP、GL\_TRIANGLE\_FAN、GL\_TRIANGLES、GL\_TRIANGLE\_STRIP\_ADJACENCY 和 GL\_TRIANGLES\_ADJACENCY 都是可接受的。

count

指定将要被渲染的元素数量。

type

指定 indices 中的值的类型，必须是 GL\_UNSIGNED\_BYTE、GL\_UNSIGNED\_SHORT 和 GL\_UNSIGNED\_INT 中的一个。

indices

指定一个指向索引存储位置的指针。

#### 描述

glDrawElements 通过进行很少的子例程调用来指定多个几何图元。我们可以预先指定单独的顶点、法线等数组，并通过调用一次 glDrawElements 来使用它们构造一个图元序列，而不是通过调用一个 GL 函数来传输每个单独的顶点、法线、纹理坐标、边缘标记或颜色。

在调用 glDrawElements 时，它使用 count 时序元素来从一个从 indices 开始的激活的数组构造一个几何图元序列。mode 指定要构造哪种图元，以及如何用这些数组元素来构造这些图元。如果激活了一个以上的数组，那么每一个都会用到。

由 glDrawElements 修改的顶点属性在 glDrawElements 返回后会有一个未指定值。没有被修改过的属性将保持它们以前的值。

#### 注意

GL\_LINE\_STRIP\_ADJACENCY、GL\_LINES\_ADJACENCY、GL\_TRIANGLE\_STRIP\_ADJACENCY 和 GL\_TRIANGLES\_ADJACENCY 只在 3.2 或更高版本的 GL 中可用。

#### 错误

如果 mode 不是一个可接受的值，则产生 GL\_INVALID\_ENUM 错误。

如果 count 为负值，则产生 GL\_INVALID\_VALUE 错误。

如果一个几何图形着色器被激活，并且 mode 与当前安装的程序对象中几何图形着色器的输入图元类型不兼容，那么将生成 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到一个激活的数组或这个元素数组，并且缓冲区对象的数据存储当前被映射，则产生 GL\_INVALID\_OPERATION 错误。

#### 另外查看

glDrawArrays, glDrawElementsInstanced, glDrawElementsBaseVertex, glDrawRangeElements

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

### glDrawElementsBaseVertex

从数组数据渲染图元，并进行逐元素偏置。

#### C 规范

```
void glDrawElementsBaseVertex(GLenum mode,
                              GLsizei count,
                              GLenum type,
                              GLvoid * indices,
                              GLint basevertex);
```

**参数****mode**

指定将要渲染哪种类型的图元。符号常量 `GL_POINTS`、`GL_LINE_STRIP`、`GL_LINE_LOOP`、`GL_LINES`、`GL_TRIANGLE_STRIP`、`GL_TRIANGLE_FAN` 和 `GL_TRIANGLES` 都是可接受的。

**count**

指定将要被渲染的元素数量。

**type**

指定 `indices` 中的值的类型，必须是 `GL_UNSIGNED_BYTE`、`GL_UNSIGNED_SHORT` 和 `GL_UNSIGNED_INT` 中的一个。

**indices**

指定一个指向索引存储位置的指针。

**basevertex**

指定一个应该在从激活的顶点数组中选择元素时被添加到 `indices` 的每个元素的常量

**描述**

`glDrawElementsBaseVertex` 的行为与 `glDrawElements` 相同，除了由相应绘制调用进行变换的第 *i* 个元素将会从每个激活的数组的 `indices[i] + basevertex` 元素中获取之外。如果结果得到的值大于 `type` 能够表示的最大值，那么计算就像上转换到 32 位无符号整数（在溢出条件下进行环绕）一样。如果得到的和为负，那么操作将为位定义的。

**注意**

`glDrawElementsBaseVertex` 在 3.2 或更高版本的 GL 中，或者在支持 `ARB_draw_elements_base_vertex` 扩展的情况下可用。

**错误**

如果 `mode` 不是一个可接受的值，则产生 `GL_INVALID_ENUM` 错误。

如果 `count` 为负值，则产生 `GL_INVALID_VALUE` 错误。

如果一个几何图形着色器被激活，并且 `mode` 与当前安装的程序对象中几何图形着色器的输入图元类型不兼容，那么将生成 `GL_INVALID_OPERATION` 错误。

如果非 0 缓冲区对象名称被绑定到一个激活的数组或这个元素数组，并且缓冲区对象的数据存储当前被映射，则产生 `GL_INVALID_OPERATION` 错误。

**另外查看**

`glDrawElements`, `glDrawRangeElements`, `glDrawRangeElementsBaseVertex`, `glDrawElementsInstanced`, `glDrawElementsInstancedBaseVertex`

**版权**

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。 <http://opencontent.org/openpub/>。

**glDrawElementsInstanced**

绘制一组元素的多个实例。

**C 规范**

```
void glDrawElementsInstanced(GLenum mode,
                             GLsizei count,
                             GLenum type,
                             const void * indices,
                             GLsizei primcount);
```

**参数****mode**

指定将要渲染哪种类型的图元。符号常量 `GL_POINTS`、`GL_LINE_STRIP`、`GL_LINE_LOOP`、`GL_LINES`、`GL_LINE_STRIP_ADJACENCY`、`GL_LINES_ADJACENCY`、`GL_TRIANGLE_STRIP`、`GL_TRIANGLE_FAN`、`GL_TRIANGLES`、`GL_TRIANGLE_STRIP_ADJACENCY` 和 `GL_TRIANGLES_ADJACENCY` 都是可接受的。

**count**

指定将要被渲染的元素数量。

type

指定 indices 中的值的类型, 必须是 GL\_UNSIGNED\_BYTE、GL\_UNSIGNED\_SHORT 和 GL\_UNSIGNED\_INT 中的一个。

indices

指定一个指向索引存储位置的指针。

primcount

指定将要被渲染的指定范围索引的实例的数量。

描述

glDrawElementsInstanced 的行为与 glDrawElements 相同, 除了执行一组元素的 primcount 实例, 以及内部计数器 instanceID 的值在每次迭代时都会递增之外。instanceID 是一个内部 32 位整型计数器, 可以由 gl\_InstanceID 这样的顶点着色器进行读取。

glDrawElementsInstanced 的效果与下列代码相同。

```
<![CDATA[ if (mode, count, or type is invalid )
generate appropriate error
else {
for (int i = 0; i < primcount ; i++) {
instanceID = i;
glDrawElements(mode, count, type, indices);
}
instanceID = 0;
}]]>
```

注意

glDrawElementsInstanced 只在 3.1 或更高版本的 GL 中可用。

GL\_LINE\_STRIP\_ADJACENCY 、 GL\_LINES\_ADJACENCY 、 GL\_TRIANGLE\_STRIP\_ADJACENCY 和 GL\_TRIANGLES\_ADJACENCY 只在 3.2 或更高版本的 GL 中可用。

错误

如果 mode 不是 GL\_POINTS、GL\_LINE\_STRIP、GL\_LINE\_LOOP、GL\_LINES、GL\_TRIANGLE\_STRIP、GL\_TRIANGLE\_FAN 或 GL\_TRIANGLES, 则产生 GL\_INVALID\_ENUM 错误。

如果 count 或者 primcount 为负值, 则产生 GL\_INVALID\_VALUE 错误。

如果一个几何图形着色器被激活, 并且 mode 与当前安装的程序对象中几何图形着色器的输入图元类型不兼容, 那么将生成 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到一个激活的数组, 并且缓冲区对象的数据存储当前被映射, 则产生 GL\_INVALID\_OPERATION 错误。

另外查看

glDrawElements, glDrawArraysInstanced

版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glDrawElementsInstancedBaseVertex

从数组数据渲染一组图元的多个实例, 并进行逐元素偏置。

C 规范

```
void glDrawElementsInstancedBaseVertex(GLenum mode,
                                         GLsizei count,
                                         GLenum type,
                                         GLvoid * indices,
                                         GLsizei primcount,
                                         GLint basevertex);
```

参数

mode

指定将要渲染哪种类型的图元。符号常量 `GL_POINTS`、`GL_LINE_STRIP`、`GL_LINE_LOOP`、`GL_LINES`、`GL_TRIANGLE_STRIP`、`GL_TRIANGLE_FAN` 和 `GL_TRIANGLES` 都是可接受的。

`count`

指定将要被渲染的元素数量。

`type`

指定 `indices` 中的值的类型，必须是 `GL_UNSIGNED_BYTE`、`GL_UNSIGNED_SHORT` 和 `GL_UNSIGNED_INT` 中的一个。

`indices`

指定一个指向索引存储位置的指针。

`primcount`

指定应该进行绘制的索引几何图形的实例的数量。

`basevertex`

指定一个应该在从激活的顶点数组中选择元素时被添加到 `indices` 的每个元素的常量

描述

`glDrawElementsInstancedBaseVertex` 的行为与 `glDrawElementsInstanced` 相同，除了由相应绘制调用进行变换的第 *i* 个元素将会从每个激活的数组的 `indices[i] + basevertex` 元素中获取之外。如果结果得到的值大于 `type` 能够表示的最大值，那么计算就像上转换到 32 位无符号整数（在溢出条件下进行环绕）一样。如果得到的和为负，那么操作将为位定义的。

注意

`glDrawElementsInstancedBaseVertex` 只在 3.2 或更高版本的 GL 中能够支持。

错误

如果 `mode` 不是一个可接受的值，则产生 `GL_INVALID_ENUM` 错误。

如果 `count` 或者 `primcount` 为负值，则产生 `GL_INVALID_VALUE` 错误。

如果一个几何图形着色器被激活，并且 `mode` 与当前安装的程序对象中几何图形着色器的输入图元类型不兼容，那么将生成 `GL_INVALID_OPERATION` 错误。

如果非 0 缓冲区对象名称被绑定到一个激活的数组或这个元素数组，并且缓冲区对象的数据存储当前被映射，则产生 `GL_INVALID_OPERATION` 错误。

另外查看

`glDrawElements`, `glDrawRangeElements`, `glDrawRangeElementsBaseVertex`, `glDrawElementsInstanced`, `glDrawElementsInstancedBaseVertex`

版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。 <http://opencontent.org/openpub/>。

## glDrawRangeElements

从数组数据渲染图元。

**C 规范**

```
void glDrawRangeElements(GLenum mode,
                        GLuint start,
                        GLuint end,
                        GLsizei count,
                        GLenum type,
                        const GLvoid * indices);
```

参数

`mode`

指定将要渲染哪种类型的图元。符号常量 `GL_POINTS`、`GL_LINE_STRIP`、`GL_LINE_LOOP`、`GL_LINES`、`GL_LINE_STRIP_ADJACENCY`、`GL_LINES_ADJACENCY`、`GL_TRIANGLE_STRIP`、`GL_TRIANGLE_FAN`、`GL_TRIANGLES`、`GL_TRIANGLE_STRIP_ADJACENCY` 和 `GL_TRIANGLES_ADJACENCY` 都是可接受的。

`start`

指定中包含的最小数组索引。

end

指定中包含的最大数组索引。

count

指定将要被渲染的元素数量。

type

指定 indices 中的值的类型，必须是 GL\_UNSIGNED\_BYTE、GL\_UNSIGNED\_SHORT 和 GL\_UNSIGNED\_INT 中的一个。

indices

指定一个指向索引存储位置的指针。

描述

glDrawRangeElements 是 glDrawElements 的一种限制形式。Mode、start、end 和 count 分别对应 glDrawElements 的相应变量，并带有一个附加的约束条件，即数组 count 中的所有值必须在 start 和 end 之间（包含边界）。

实现指出了推荐的顶点和索引数据的最大数量值，它们可以通过调用自变量为 GL\_MAX\_ELEMENTS\_VERTICES 和 GL\_MAX\_ELEMENTS\_INDICES 的 glGet 进行查询。如果  $end - start + 1$  大于 GL\_MAX\_ELEMENTS\_VERTICES 的值，或者如果 count 大于 GL\_MAX\_ELEMENTS\_INDICES 的值，那么这个调用可能在性能降低的状态下运行。并没有提到对于所有顶点都要在 [start,end] 范围中的要求。但是，实现可能部分地对未使用的顶点进行处理，在最佳索引集所能达到的基础上降低性能。

在调用 glDrawRangeElements 时，它使用 count 时序元素来从一个从 start 开始的激活的数组构造一个几何图元序列。mode 指定要构造哪种图元，以及如何用这些数组元素来构造这些图元。如果激活了一个以上的数组，那么每一个都会用到。

由 glDrawRangeElements 修改的顶点属性在 glDrawRangeElements 返回后会有一个未指定值。没有被修改过的属性将保持它们以前的值。

注意

GL\_LINE\_STRIP\_ADJACENCY、GL\_LINES\_ADJACENCY、GL\_TRIANGLE\_STRIP\_ADJACENCY 和 GL\_TRIANGLES\_ADJACENCY 只在 3.2 或更高版本的 GL 中可用。

错误

索引位于 [start,end] 范围之外是一个错误，但实现可能不会对这种情况进行检查。这样的索引会导致和实现相关的行为。

如果 mode 不是一个可接受的值，则产生 GL\_INVALID\_ENUM 错误。

如果 count 为负值，则产生 GL\_INVALID\_VALUE 错误。

如果  $end < start$ ，则产生 GL\_INVALID\_VALUE 错误。

如果一个几何图形着色器被激活，并且 mode 与当前安装的程序对象中几何图形着色器的输入图元类型不兼容，那么将生成 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到一个激活的数组或这个元素数组，并且缓冲区对象的数据存储当前被映射，则产生 GL\_INVALID\_OPERATION 错误。

相关 Get 函数

glGet，其自变量为 GL\_MAX\_ELEMENTS\_VERTICES。

glGet，其自变量为 GL\_MAX\_ELEMENTS\_INDICES。

另外查看

glDrawArrays, glDrawElements, glDrawElementsBaseVertex

版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glDrawRangeElementsBaseVertex

从数组数据渲染图元，并进行逐元素偏置。

C 规范

```
void glDrawRangeElementsBaseVertex(GLenum mode,
```

```
GLuint start,
GLuint end,
GLsizei count,
GLenum type,
GLvoid * indices,
GLint basevertex);
```

**参数****mode**

指定将要渲染哪种类型的图元。符号常量 `GL_POINTS`、`GL_LINE_STRIP`、`GL_LINE_LOOP`、`GL_LINES`、`GL_TRIANGLE_STRIP`、`GL_TRIANGLE_FAN` 和 `GL_TRIANGLES` 都是可接受的。

**start**

指定中包含的最小数组索引。

**end**

指定中包含的最大数组索引。

**count**

指定将要被渲染的元素数量。

**type**

指定 `indices` 中的值的类型，必须是 `GL_UNSIGNED_BYTE`、`GL_UNSIGNED_SHORT` 和 `GL_UNSIGNED_INT` 中的一个。

**indices**

指定一个指向索引存储位置的指针。

**basevertex**

指定一个应该在从激活的顶点数组中选择元素时被添加到 `indices` 的每个元素的常量

**描述**

`glDrawRangeElementsBaseVertex` 是 `glDrawElementsBaseVertex` 的一种限制形式。`Mode`、`start`、`end` 和 `basevertex` 分别对应 `glDrawElementsBaseVertex` 的相应变量，并带有一个附加的约束条件，即数组 `indices` 中的所有值必须在 `start` 和 `end` 之间（包含边界），添加 `basevertex` 之前。`[start, end]` 范围之外的索引值的处理方式与 `glDrawElementsBaseVertex` 相同。第 *i* 个元素将会从每个激活的数组的 `indices[i] + basevertex` 元素中获取。如果结果得到的值大于 `type` 能够表示的最大值，那么计算就像上转换到 32 位无符号整数（在溢出条件下进行环绕）一样。

如果得到的和为负，那么操作将为位定义的。

**注意**

`glDrawRangeElementsBaseVertex` 在 3.2 或更高版本的 GL 中，或者在支持 `ARB_draw_elements_base_vertex` 扩展的情况下可用。

**错误**

如果 `mode` 不是一个可接受的值，则产生 `GL_INVALID_ENUM` 错误。

如果 `count` 为负值，则产生 `GL_INVALID_VALUE` 错误。

如果 `end < start`，则产生 `GL_INVALID_VALUE` 错误。

如果一个几何图形着色器被激活，并且 `mode` 与当前安装的程序对象中几何图形着色器的输入图元类型不兼容，那么将生成 `GL_INVALID_OPERATION` 错误。

如果非 0 缓冲区对象名称被绑定到一个激活的数组或这个元素数组，并且缓冲区对象的数据存储当前被映射，则产生 `GL_INVALID_OPERATION` 错误。

**另外查看**

`glDrawElements`, `glDrawElementsBaseVertex`, `glDrawRangeElements`, `glDrawElementsInstanced`, `glDrawElementsInstancedBaseVertex`

**版权**

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

**glEnable**

激活或禁止服务器端 GL 性能。

### C 规范

```
void glEnable(GLenum cap);
```

#### 参数

cap

指定一个符号常量来指示一种 GL 性能。

### C 规范

```
void glDisable(GLenum cap);
```

#### 参数

cap

指定一个符号常量来指示一种 GL 性能。

### C 规范

```
void glEnablei(GLenum cap,
               GLuint index);
```

#### 参数

cap

指定一个符号常量来指示一种 GL 性能。

index

指定将要激活的转换的索引。

### C 规范

```
void glDisablei(GLenum cap,
                GLuint index);
```

#### 参数

cap

指定一个符号常量来指示一种 GL 性能。

index

指定将要关闭的转换的索引。

#### 描述

glEnable 和 glDisable 用来激活和关闭各种性能。使用 glIsEnabled 或 glGet 来决定任何性能的当前设置。除 GL\_DITHER 和 GL\_MULTISAMPLE 以外，其他每一种性能的初始值都是 GL\_FALSE。GL\_DITHER 和 GL\_MULTISAMPLE 的初始值是 GL\_TRUE。

glEnable 和 glDisable 都使用同一个自变量 cap，这个自变量可以采用下列值中的一个。

GL 的某些性能是被指定的。glEnablei 和 glDisablei 激活和关闭索引性能。

GL\_BLEND

如果被激活，则将计算的片段颜色值和颜色缓冲区中的颜色值进行混合。参见 glBlendFunc。

GL\_CLIP\_DISTANCEi

如果被激活，则用用户定义的半平面 i 来裁剪几何图形。

GL\_COLOR\_LOGIC\_OP

如果被激活，则将当前选择的逻辑操作应用于输入的计算的片段颜色值和颜色缓冲区值上。参见 glLogicOp。

GL\_CULL\_FACE

如果被激活，则根据多边形围绕窗口坐标轴的情况来对其进行剔除。参见 glCullFace。

GL\_DEPTH\_TEST

如果被激活，则进行深度比较并更新深度缓冲区。请注意即使在深度缓冲区存在、深度遮罩非 0 的情况下，如果深度测试被禁用，深度缓冲区也不会被更新。参见 glDepthFunc 和 glDepthRange。

GL\_DITHER

如果被激活，则在颜色分量或索引写入颜色缓冲区之前对其进行抖动。

GL\_LINE\_SMOOTH

如果被激活，则使用正确的滤波来绘制线。否则绘制锯齿线。参见 glLineWidth。

GL\_MULTISAMPLE

如果被激活,则在最终像素颜色的计算中使用多片段样本。参见 `glSampleCoverage`。

`GL_POLYGON_OFFSET_FILL`

如果被激活,并且多边形在 `GL_FILL` 模式下被渲染,那么将在深度比较进行前向一个多边形的片段深度值中加入一个偏置。参见 `glPolygonOffset`。

`GL_POLYGON_OFFSET_LINE`

如果被激活,并且多边形在 `GL_LINE` 模式下被渲染,那么将在深度比较进行前向一个多边形的片段深度值中加入一个偏置。参见 `glPolygonOffset`。

`GL_POLYGON_OFFSET_POINT`

如果被激活,那么将在深度比较进行前向一个多边形的片段深度值中加入一个偏置,如果这个多边形是在 `GL_POINT` 模式下进行渲染的话。参见 `glPolygonOffset`。

`GL_POLYGON_SMOOTH`

如果被激活,则使用正确的滤波来绘制多边形。否则绘制锯齿多边形。对于正确抗锯齿的多边形来说,需要一个 `alpha` 缓冲区,并且这个多边形必须从前向后存储。

`GL_PRIMITIVE_RESTART`

激活图元重启。如果被激活,那么任意一个将一组通用属性数组元素变换到 `GL` 的绘制命令都将在顶点的索引与图元重启索引相等时对图元进行重启。参见 `glPrimitiveRestartIndex`。

`GL_SAMPLE_ALPHA_TO_COVERAGE`

如果被激活,则计算一个临时覆盖值 (temporary coverage value), 其中每一个位都由相应采样位置的 `Alpha` 值来决定。然后,这个临时覆盖值与片段覆盖值 (fragment coverage value) 进行逻辑乘 (AND, 即“与”运算)。

`GL_SAMPLE_ALPHA_TO_ONE`

如果被激活,则每一个采样 `Alpha` 值将由能够表示的最大 `Alpha` 值代替。

`GL_SAMPLE_COVERAGE`

如果被激活,则片段的覆盖值将与临时覆盖值进行逻辑乘。如果 `GL_SAMPLE_COVERAGE_INVERT` 被设置为 `GL_TRUE`,则将覆盖值反相。参见 `glSampleCoverage`。

`GL_SCISSOR_TEST`

如果被激活,则抛弃裁剪矩形以外的片段。参见 `glScissor`。

`GL_STENCIL_TEST`

如果被激活,则进行模板测试并更新模板缓冲区。参见 `glStencilFunc` 和 `glStencilOp`。

`GL_TEXTURE_CUBE_MAP_SEAMLESS`

如果被激活,则修改对立方体贴图纹理执行采样的方式。更多信息请参考规范。

`GL_PROGRAM_POINT_SIZE`

如果被激活,并且激活了一个顶点着色器或几何图形着色器,那么点大小则是从在 `glPointSize` 中创建的 (可能是被裁剪的) 着色器中获取的,并被截取到实现相关的点大小范围。

**错误**

如果 `cap` 不是前面列出的一个允许值,则产生 `GL_INVALID_ENUM` 错误。

如果 `index` 大于或等于 `cap` 的索引功能数量,则 `glEnablei` 和 `glDisablei` 会产生 `GL_INVALID_VALUE` 错误。

**注意**

`GL_PRIMITIVE_RESTART` 只在 3.1 或更高版本的 `GL` 中可用。

`GL_TEXTURE_CUBE_MAP_SEAMLESS` 在 3.2 或更高版本的 `GL` 中可用。

`glEnable` 或 `glDisable` 能够接受的任何标记也可以被 `glEnablei` 和 `glDisablei` 接受,但是如果这项功能没有进行索引,那么 `index` 能够获取的最大值则为 0。

一般来说,向 `glEnable` 或 `glDisable` 传递一个索引功能将会分别为所有索引激活或关闭这项功能。

**相关 Get 函数**

`glIsEnabled`

`glGet`

**另外查看**

`glActiveTexture`, `glBlendFunc`, `glCullFace`, `glDepthFunc`, `glDepthRange`, `glGet`, `glIsEnabled`, `glLineWidth`, `glLogicOp`, `glPointSize`, `glPolygonMode`, `glPolygonOffset`, `glSampleCoverage`, `glScissor`, `glStencilFunc`, `glStencilOp`, `glTexImage1D`, `glTexImage2D`, `glTexImage3D`

**版权**

Copyright © 1991–2006 Silicon Graphics, Inc. Copyright © 2010 Khronos Group. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glEnableVertexAttribArray

激活或关闭一个通用顶点属性数组。

### C 规范

```
void glEnableVertexAttribArray(GLuint index);
void glDisableVertexAttribArray(GLuint index);
```

### 参数

index

指定将要激活或绑定的一般顶点属性的索引。

### 描述

glEnableVertexAttribArray 激活由 index 指定的通用顶点属性数组。

glDisableVertexAttribArray 禁止由 index 指定的通用顶点属性数组。在默认情况下, 所有客户端性能都是关闭的, 包括所有通用顶点属性数组。如果被激活, 通用顶点属性数组中的值将在对顶点数组进行 glDrawArrays、glDrawElements、glDrawRangeElements、glMultiDrawElements 或 glMultiDrawArrays 调用时为了进行渲染而被访问和使用。

### 错误

如果 index 大于或等于 GL\_MAX\_VERTEX\_ATTRIBS, 则产生 GL\_INVALID\_VALUE 错误。

### 相关 Get 函数

glGet, 其自变量为 GL\_MAX\_VERTEX\_ATTRIBS。

glGetVertexAttrib, 其自变量为 index 和 GL\_VERTEX\_ATTRIB\_ARRAY\_ENABLED。

glGetVertexAttribPointerv, 其自变量为 index 和 GL\_VERTEX\_ATTRIB\_ARRAY\_POINTER。

### 另外查看

glBindAttribLocation, glDrawArrays, glDrawElements, glDrawRangeElements, glMultiDrawElements, glVertexAttrib, glVertexAttribPointer

### 版权

Copyright © 2003–2005 3Dlabs Inc. Ltd. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。 <http://opencontent.org/openpub/>。

## glFenceSync

创建一个新的同步对象并将它插入到 GL 命令流中。

### C 规范

```
GLsync glFenceSync(GLenum condition,
                  GLbitfield flags);
```

### 参数

condition

指定为了将同步对象设置为标记状态必须满足的条件。

condition 必须为 GL\_SYNC\_GPU\_COMMANDS\_COMPLETE。

flags

指定标志的一个按位组合, 控制同步对象的行为。目前没有为这个操作定义任何标志, 而 flags 必须为 0。<sup>\*</sup>

### 描述

glFenceSync 创建一个新的围栏同步对象, 将一个围栏命令插入 GL 命令流并将它与这个同步对象关联起来, 并返回一个与这个同步对象对应的非 0 名称。

当围栏命令满足了同步对象的指定 condition, 同步对象由 GL 设为标记状态, 导致 sync 中的任何 glWaitSync 和

<sup>\*</sup> flags 是围栏同步对象功能的一个预期的未来执行的占位符。

glClientWaitSync 命令被阻塞为 unblock。其他任何状态都不会受 glFenceSync 或者关联围栏命令执行的影响。

condition 必须为 GL\_SYNC\_GPU\_COMMANDS\_COMPLETE。这个条件通过完成与这个同步对象对应的围栏命令和同一个命令流中前面所有命令来满足。在这些命令对 GL 客户端、服务器状态和帧缓冲区的影响完全实现之前，所有这个同步对象不会被设置为标记状态。请注意当对应同步对象的状态改变时，围栏命令会完成一次，但是在围栏命令完成之前，等待这个同步对象的命令不会解除阻塞。

#### 注意

glFenceSync 在 3.2 或更高版本的 GL 中，或者在支持 ARB\_sync 扩展的情况下可用。

#### 错误

如果 condition 不是 GL\_SYNC\_GPU\_COMMANDS\_COMPLETE，则产生 GL\_INVALID\_ENUM 错误。

如果 flags 为非 0 值，则产生 GL\_INVALID\_VALUE 错误。

另外，如果 glFenceSync 失败，就会返回 0。

#### 另外查看

glDeleteSync, glGetSync, glWaitSync, glClientWaitSync

#### 版权

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glFinish

在所有 GL 执行完成之前保持阻塞。

#### C 规范

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

#### 描述

载前面调用的所有 GL 命令的影响完成之前，glFinish 不会返回。

这些影响包括 GL 状态的所有改变，连接状态的所有改变，以及帧缓冲区内内容的所有改变。

#### 注意

glFinish 要求对服务器进行一次往返。

#### 另外查看

glFlush

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glFlush

在有限的时间内强制执行 GL 命令。

#### C 规范

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

#### 描述

不同的 GL 实现缓冲区命令在几个不同的区域中，这些区域包括各种缓冲区和图形加速器本身。glFlush 能够清空所有这些缓冲区，以使所有发出的命令在它们一旦被实际渲染引擎接受就可以马上执行。虽然这个执行可能不在任何特定的时间段中完成，但它确实能在限定的时间内完成。

由于任何 GL 程序都可能通过网络来执行，或者在一个缓冲命令的加速器上执行，所以所有程序都要调用 glFlush，如果它们想要它们以前发出的命令都能完成的话。例如，在开始等待基于生成图象的用户输入之前就要调用 glFlush。

#### 注意

glFlush 可以在任何时间返回。它不会等待所有以前发出的 GL 命令执行完成。

#### 另外查看

glFinish

**版权**

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

**glFlushMappedBufferRange**

表示修改一个映射缓冲区的一个范围。

**C 规范**

```
GLsync glFlushMappedBufferRange(GLenum target,
                                GLintptr offset,
                                GLsizeiptr length);
```

**参数****target**

指定清理操作的目标。target 必须是 GL\_ARRAY\_BUFFER、GL\_COPY\_READ\_BUFFER、GL\_COPY\_WRITE\_BUFFER、GL\_ELEMENT\_ARRAY\_BUFFER、GL\_PIXEL\_PACK\_BUFFER、GL\_PIXEL\_UNPACK\_BUFFER、GL\_TEXTURE\_BUFFER、GL\_TRANSFORM\_FEEDBACK\_BUFFER 或 GL\_UNIFORM\_BUFFER。

**offset**

指定缓冲区子范围的开始位置，以及本机器单元为单位。

**length**

指定缓冲区子范围的长度，以及本机器单元为单位。

**描述**

glFlushMappedBufferRange 表示已经对一个映射缓冲区的一个范围作了修改。这个缓冲区以前必须与 GL\_MAP\_FLUSH\_EXPLICIT 标记进行映射。offset 和 length 表示映射修改的子范围，以及本单元为单位。指定要被清理的子范围被关联到缓冲区中当前映射的范围的开始位置。

glFlushMappedBufferRange 可能被多次调用，来指示需要进行清理的映射的不同子范围。

**错误**

如果 offset 或 length 为负，或者如果 offset + length 超出映射的大小，则产生 GL\_INVALID\_VALUE 错误。

如果 0 被绑定到 target，则产生 GL\_INVALID\_OPERATION 错误。

如果绑定到 target 的缓冲区对象没有被映射，或者在没有 GL\_MAP\_FLUSH\_EXPLICIT 标记的情况下被绑定，则产生 GL\_INVALID\_OPERATION 错误。

**另外查看**

glMapBufferRange, glMapBuffer, glUnmapBuffer

**版权**

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。 <http://opencontent.org/openpub/>。

**glFramebufferRenderbuffer**

将一个渲染缓冲区作为逻辑缓冲区绑定到当前绑定的帧缓冲区对象。

**C 规范**

```
GLsync glFramebufferRenderbuffer(GLenum target,
                                  GLenum attachment,
                                  GLenum renderbuffertarget,
                                  GLuint renderbuffer);
```

**参数****target**

指定帧缓冲区目标。target 必须为 GL\_DRAW\_FRAMEBUFFER、GL\_READ\_FRAMEBUFFER 或 GL\_FRAMEBUFFER。GL\_FRAMEBUFFER 等价于 GL\_DRAW\_FRAMEBUFFER。

**attachment**

指定帧缓冲区的绑定点。

**renderbuffertarget**

指定渲染缓冲区目标, 必须为 GL\_RENDERBUFFER。

**renderbuffer**

指定将要进行绑定的一个已经存在的 renderbuffertarget 类型渲染缓冲区对象的名称。

**描述**

glFramebufferRenderbuffer 将一个渲染缓冲区绑定为当前绑定的帧缓冲区对象的一个逻辑缓冲区。renderbuffer 是要绑定的渲染缓冲区对象的名称, 必须为 0 或一个已经存在的 renderbuffertarget 类型渲染缓冲区对象的名称。如果 renderbuffer 不为 0, 并且如果 glFramebufferRenderbuffer 成功, 那么渲染缓冲区名称 renderbuffer 将被作为由当前绑定到 target 的帧缓冲区的 attachment 来标识的逻辑缓冲区使用。

指定绑定点的 GL\_FRAMEBUFFER\_ATTACHMENT\_OBJECT\_TYPE 的值被设置为 GL\_RENDERBUFFER, 并且 GL\_FRAMEBUFFER\_ATTACHMENT\_OBJECT\_NAME 的值被设置为 renderbuffer。由 attachment 指定的绑定点的所有其他状态都被设置为它们的默认值。渲染缓冲区对象的状态不会发生任何改变, 并且以前到这个帧缓冲区 target 的 attachment 逻辑缓冲区 attachment 的任何绑定都会被解除。

调用帧缓冲区名为 zero 的 glFramebufferRenderbuffer 将会对当前绑定到 target 的帧缓冲区中任何由 attachment 标识的图像 (如果有的话) 解除绑定。绑定到 target 的对象中由 attachment 指定的绑定点的所有状态都被设置为它们的默认值。

将 attachment 设置为 GL\_DEPTH\_STENCIL\_ATTACHMENT 是一种特别的情况, 会导致帧缓冲区对象的深度和模板绑定都被设置成应该为基本内部格式 GL\_DEPTH\_STENCIL 的 renderbuffer。

**错误**

如果 tarGet 不是一个可接受的标记, 则产生 GL\_INVALID\_ENUM 错误。

如果 renderbuffertarget 不是 GL\_RENDERBUFFER, 则产生 GL\_INVALID\_ENUM 错误。

如果 0 被绑定到 tarGet, 则产生 GL\_INVALID\_OPERATION 错误。

**另外查看**

glGenFramebuffers, glBindFramebuffer, glGenRenderbuffers, glFramebufferTexture, glFramebufferTexture1D, glFramebufferTexture2D, glFramebufferTexture3D

**版权**

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glFramebufferTexture

将一个纹理对象的层次作为逻辑缓冲区绑定到当前绑定的帧缓冲区对象。

**C 规范**

```
void glFramebufferTexture(GLenum target,
                          GLenum attachment,
                          GLuint texture,
                          GLint level);

void glFramebufferTexture1D(GLenum target,
                             GLenum attachment,
                             GLuint texture,
                             GLint level);

void glFramebufferTexture2D(GLenum target,
                             GLenum attachment,
                             GLuint texture,
                             GLint level);

void glFramebufferTexture3D(GLenum target,
                             GLenum attachment,
                             GLuint texture,
                             GLint level,
                             GLint layer);
```

**参数**

target

指定帧缓冲区目标。target 必须为 GL\_DRAW\_FRAMEBUFFER、GL\_READ\_FRAMEBUFFER 或 GL\_FRAMEBUFFER。GL\_FRAMEBUFFER 等价于 GL\_DRAW\_FRAMEBUFFER。

attachment

指定帧缓冲区的绑定。attachment 必须为 GL\_COLOR\_ATTACHMENTi、GL\_DEPTH\_ATTACHMENT、GL\_STENCIL\_ATTACHMENT 或 GL\_DEPTH\_STENCIL\_ATTACHMENT。

texture

指定将要绑定到由 attachment 指定的帧缓冲区绑定点的纹理对象。

level

指定将要绑定的 Mip 贴图层次。

描述

glFramebufferTexture、glFramebufferTexture1D、glFramebufferTexture2D 和 glFramebufferTexture3D 将一个纹理对象的选定 Mip 贴图层次或图像作为当前绑定的帧缓冲区对象的一个逻辑缓冲区绑定 target。target 必须为 GL\_DRAW\_FRAMEBUFFER、GL\_READ\_FRAMEBUFFER 或 GL\_FRAMEBUFFER。GL\_FRAMEBUFFER 等价于 GL\_DRAW\_FRAMEBUFFER。

attachment 指定帧缓冲区的逻辑绑定，并且必须为 GL\_COLOR\_ATTACHMENTi、GL\_DEPTH\_ATTACHMENT、GL\_STENCIL\_ATTACHMENT 或 GL\_DEPTH\_STENCIL\_ATTACHMENT。GL\_COLOR\_ATTACHMENTi 中的 i 的取值范围为 0 到 GL\_MAX\_COLOR\_ATTACHMENTS - 1 的值。将一个纹理的一个层次绑定到 GL\_DEPTH\_STENCIL\_ATTACHMENT 就相当于同时将这个层次绑定到 GL\_DEPTH\_ATTACHMENT 和 GL\_STENCIL\_ATTACHMENT 绑定。

如果 texture 不为 0，那么名为 texture 的纹理对象的指定 level 就会被绑定到由 attachment 指定的帧缓冲区绑定。对于 glFramebufferTexture1D、glFramebufferTexture2D 和 glFramebufferTexture3D，texture 必须为 0 或者一个已经存在的以 textarget 为目标的纹理的名称，或者 texture 必须为一个已经存在的立方体贴图名称并且 textarget 必须为 GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Z、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Y 或 GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Z 中的一个。

如果 textarget 为 GL\_TEXTURE\_RECTANGLE、GL\_TEXTURE\_2D\_MULTISAMPLE 或 GL\_TEXTURE\_2D\_MULTISAMPLE\_ARRAY，那么 level 必须为 0。如果 textarget 为 GL\_TEXTURE\_3D，那么 level 必须大于或等于 0，并且小于或等于 GL\_MAX\_3D\_TEXTURE\_SIZE 以 2 为底的对数。如果 textarget 为 GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Z、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Y 或 GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Z，那么 level 必须大于或等于 0，并且小于或等于 GL\_MAX\_CUBE\_MAP\_TEXTURE\_SIZE 以 2 为底的对数。对于 textarget 的所有其他值，level 必须大于或等于 0，并且小于或等于 GL\_MAX\_TEXTURE\_SIZE 以 2 为底的对数。

layer 指定一个三维纹理中的一个二维图像的层次。

对于 glFramebufferTexture1D，如果 textarget 为非 0 值，那么 textarget 必须为 GL\_TEXTURE\_1D。对于 glFramebufferTexture2D，如果 texture 不为 0，那么 textarget 必须为 GL\_TEXTURE\_2D、GL\_TEXTURE\_RECTANGLE、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Z、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Z 或 GL\_TEXTURE\_2D\_MULTISAMPLE 中的一个。对于 glFramebufferTexture3D，如果 texture 为非 0 值，那么 textarget 必须为 GL\_TEXTURE\_3D。

注意

glFramebufferTexture 只在 3.2 或更高版本的 GL 中可用。

错误

如果 tarGet 不是一个可接受的标记，则产生 GL\_INVALID\_ENUM 错误。

如果 renderbuffertarGet 不是 GL\_RENDERBUFFER，则产生 GL\_INVALID\_ENUM 错误。

如果 0 被绑定到 tarGet，则产生 GL\_INVALID\_OPERATION 错误。

另外查看

glGenFramebuffers、glBindFramebuffer、glGenRenderbuffers、glFramebufferTexture、glFramebufferTexture1D、glFramebufferTexture2D、glFramebufferTexture3D

版权

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glFramebufferTextureFace

将一个立方体贴图纹理的表面作为逻辑缓冲区绑定到当前绑定的帧缓冲区。

### C 规范

```
void glFramebufferTextureFace(GLenum target,
                              GLenum attachment,
                              GLuint texture,
                              GLint level,
                              GLenum face);
```

### 参数

target

指定帧缓冲区目标。target 必须为 GL\_DRAW\_FRAMEBUFFER、GL\_READ\_FRAMEBUFFER 或 GL\_FRAMEBUFFER。GL\_FRAMEBUFFER 等价于 GL\_DRAW\_FRAMEBUFFER。

attachment

指定帧缓冲区的绑定点。attachment 必须为 GL\_COLOR\_ATTACHMENTi、GL\_DEPTH\_ATTACHMENT、GL\_STENCIL\_ATTACHMENT 或 GL\_DEPTH\_STENCIL\_ATTACHMENT。

纹理贴图

指定将要绑定到由 attachment 指定的帧缓冲区绑定点的纹理对象。texture 必须为一个已经存在的立方体贴图纹理的名称。

level

指定将要绑定的 Mip 贴图层次。

face

指定将要绑定的纹理的表面。

### 描述

glFramebufferTextureFace 的作用与 glFramebufferTexture 相似，除了只有由 face 给出的立方体贴图纹理的一个表面被绑定到这个绑定点之外。face 必须为 GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Z、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Y 或 GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Z。texture 必须为 0 或者一个已经存在的立方体贴图纹理的名称。

### 错误

如果 target 不是一个可接受的标记，则产生 GL\_INVALID\_ENUM 错误。

如果 attachment 不是一个可接受的标记，则产生 GL\_INVALID\_ENUM 错误。

如果 face 不是一个可接受的标记，则产生 GL\_INVALID\_ENUM 错误。

如果 0 被绑定到 target，则产生 GL\_INVALID\_OPERATION 错误。

如果 texture 不是 0 或一个已经存在的立方体贴图纹理的名称，则产生 GL\_INVALID\_OPERATION 错误。

### 另外查看

glGenFramebuffers, glBindFramebuffer, glGenRenderbuffers, glFramebufferTexture, glFramebufferTextureLayer

### 版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glFramebufferTextureLayer

将一个立方体贴图纹理的表面作为逻辑缓冲区绑定到当前绑定的帧缓冲区。

### C 规范

```
void glFramebufferTextureLayer(GLenum target,
                              GLenum attachment,
                              GLuint texture,
                              GLint level,
                              GLint layer);
```

## 参数

### target

指定帧缓冲区目标。target 必须为 GL\_DRAW\_FRAMEBUFFER、GL\_READ\_FRAMEBUFFER 或 GL\_FRAMEBUFFER。GL\_FRAMEBUFFER 等价于 GL\_DRAW\_FRAMEBUFFER。

### attachment

指定帧缓冲区的绑定点。attachment 必须为 GL\_COLOR\_ATTACHMENTi、GL\_DEPTH\_ATTACHMENT、GL\_STENCIL\_ATTACHMENT 或 GL\_DEPTH\_STENCIL\_ATTACHMENT。

### texture

指定将要绑定到由 attachment 指定的帧缓冲区绑定点的纹理对象。

### level

指定将要绑定的 Mip 贴图层次。

### layer

指定将要绑定的纹理的层次。

## 描述

glFramebufferTextureLevel 的作用与 glFramebufferTexture 相似,除了只有由 layer 给出的纹理层次的单个层被绑定到这个绑定点之外。如果 texture 不为 0,那么 layer 就必须大于或等于 0。texture 必须为 0,或者一个已经存在的三维或二维数组纹理的名称。

## 注意

glFramebufferTextureLayer 只在 3.2 或更高版本的 GL 中可用。

## 错误

如果 target 不是一个可接受的标记,则产生 GL\_INVALID\_ENUM 错误。

如果 attachment 不是一个可接受的标记,则产生 GL\_INVALID\_ENUM 错误。

如果 texture 不是 0 或者一个已存在的纹理对象的名称,则产生 GL\_INVALID\_VALUE 错误。

如果 texture 为非 0 值,并且 layer 为负,则产生 GL\_INVALID\_VALUE 错误。

如果 0 被绑定到 target,则产生 GL\_INVALID\_OPERATION 错误。

如果 texture 不是 0 或一个已经存在的立方体贴图纹理的名称,则产生 GL\_INVALID\_OPERATION 错误。

## 另外查看

glGenFramebuffers, glBindFramebuffer, glGenRenderbuffers, glFramebufferTexture, glFramebufferTextureFace

## 版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glFrontFace

定义前向和背向多边形。

## C 规范

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

## 参数

### mode

指定前向多边形的方向。GL\_CW 和 GL\_CCW 都可以被接受。

初始值为 GL\_CCW。

## 描述

在一个完全由不透明的封闭曲面组成的场景中,背向多边形总是不可见的。

除去这些不可见的多边形对提升图像渲染速度的好处是显而易见的。可以调用自变量为 GL\_CULL\_FACE 的 glEnable 和 glDisable 来激活和禁止背向多边形。

如果一个假象的目标沿着从它的第一个顶点、第二个顶点等等,直到它的最后一个顶点,并且最后返回它的第一条顶点的路径按照顺时针方向围绕多变形的内部移动,则称多边形到窗口坐标的投影顺时针旋转 (clockwise winding)。如果目标按照逆时针方向沿着同样的路径围绕多变形的内部移动,那么称多边形的旋转被称为是逆时针的。glFrontFace 指定在窗口坐标中按照顺时针方向旋转,或者按照逆时针方向旋转的多边形是否为前向 (front-facing)。

的。将 GL\_CCW 传入 mode 会选择逆时针为前向；传入 GL\_CW 则为顺时针为前向。在默认情况下，逆时针多边形为前向的。

#### 错误

如果 mode 不是一个可接受的值，则产生 GL\_INVALID\_ENUM 错误。

#### 相关 Get 函数

glGet，其自变量为 GL\_FRONT\_FACE。

#### 另外查看

glCullFace

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glGenBuffers

创建缓冲区对象名称。

#### C 规范

```
void glGenBuffers(GLsizei n,
                  GLuint * buffers);
```

#### 参数

n

指定将要生成的缓冲区对象名称的数量。

buffers

指定一个生成的缓冲区对象名称将要存储到其中的数组。

#### 描述

glGenBuffers 在 buffers 中返回 n 个缓冲区对象名称。并不能保证这些名称会组成一个连续的一组整数，不过可以保证这些返回的名称不会在调用 glGenBuffers 前就立即被使用。

调用 glGenBuffers 而返回的缓冲区对象名称不会在后续的调用中返回，除非他们首先由 glDeleteBuffers 来删除。没有缓冲区对象会被关联到返回的缓冲区对象名称，除非它们首先通过调用 glBindBuffer 来进行绑定。

#### 错误

如果 n 为负值，则产生 GL\_INVALID\_VALUE 错误。

#### 相关 Get 函数

glIsBuffer

#### 另外查看

glBindBuffer, glDeleteBuffers, glGet

#### 版权

Copyright © 2005 Addison–Wesley. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。 <http://opencontent.org/openpub/>。

## glGenerateMipmap

为指定的纹理目标生成 Mip 贴图。

#### C 规范

```
void glGenerateMipmap(GLenum target);
```

#### 参数

target

定义要生成 Mip 贴图的纹理要绑定到的目标。target 必须为 GL\_TEXTURE\_1D、GL\_TEXTURE\_2D、GL\_TEXTURE\_3D、GL\_TEXTURE\_1D\_ARRAY、GL\_TEXTURE\_2D\_ARRAY 或 GL\_TEXTURE\_CUBE\_MAP。

#### 描述

glGenerateMipmap 为绑定到活动纹理单元的 tarGet 的纹理生成 Mip 贴图。对于立方体贴图纹理，如果绑定到 tarGet 的纹理不完全是立方体，那么将产生 GL\_INVALID\_OPERATION 错误。

Mip 贴图的生成会替代从这个数组衍生出的数组的 texel 数组层次，不管它们以前的内容如何。所有其他 Mip 贴图数组，也包含这个数组本身，在这种计算中都不会改变。

衍生 Mip 贴图数组的内部各式都与这个数组的相应格式相匹配。衍生数组的内容由这个数组经过重复和过滤进行计算。对于一维和二维纹理数组来说，每个层都会单独进行过滤。

#### 错误

如果 tarGet 不是一个可接受的纹理目标，则产生 GL\_INVALID\_ENUM 错误。

如果 tarGet 不是 GL\_TEXTURE\_CUBE\_MAP，并且绑定到活动纹理单元的 GL\_TEXTURE\_CUBE\_MAP 目标的纹理不完全是立方体，则会产生 GL\_INVALID\_OPERATION 错误。

#### 另外查看

glTexImage2D, glBindTexture, glGenTextures

#### 版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glGenFramebuffers

创建帧缓冲区对象名称。

### C 规范

```
void glGenFramebuffers(GLsizei n,
                       GLuint *ids);
```

#### 参数

n

指定将要生成的帧缓冲区对象名称的数量。

ids

指定一个生成的帧缓冲区对象名称将要存储到其中的数组。

#### 描述

glGenFramebuffers 在 ids 中返回 n 个帧缓冲区对象名称。并不能保证这些名称会组成一个连续的一组整数，不过可以保证这些返回的名称不会在调用 glGenFramebuffers 前就立即被使用。

调用 glGenFramebuffers 而返回的帧缓冲区对象名称不会在后续的调用中返回，除非他们首先由 glDeleteFramebuffers 来删除。

ids 中返回的名称将标记为已使用的，只是为了 glGenFramebuffers 的目的，但是它们只在第一次绑定时才需要。

#### 错误

如果 n 为负值，则产生 GL\_INVALID\_VALUE 错误。

#### 另外查看

glBindFramebuffer, glDeleteFramebuffers

#### 版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glGenQueries

创建查询对象名称。

### C 规范

```
void glGenQueries(GLsizei n,
                  GLuint * ids);
```

#### 参数

n

指定将要生成的查询对象名称的数量。

ids

指定一个生成的查询对象名称将要存储到其中的数组。

**描述**

glGenQueries 在 ids 中返回 n 个查询对象名称。并不能保证这些名称会组成一个连续的一组整数,不过可以保证这些返回的名称不会在调用 glGenQueries 前就立即被使用。

调用 glGenQueries 而返回的查询对象名称不会在后续的调用中返回,除非他们首先由 glDeleteQueries 来删除。

没有查询对象会被关联到返回的查询对象名称,除非它们首先通过调用 glBeginQuery 来使用。

**错误**

如果 n 为负值,则产生 GL\_INVALID\_VALUE 错误。

如果 glGenQueries 在 glBeginQuery 执行和相应的 glEndQuery 执行之间执行,则产生 GL\_INVALID\_OPERATION 错误。

**相关 Get 函数**

glIsQuery

**另外查看**

glBeginQuery, glDeleteQueries, glEndQuery

**版权**

Copyright © 2005 Addison-Wesley. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glGenRenderbuffers

创建渲染缓冲区对象名称。

**C 规范**

```
void glGenRenderbuffers(GLsizei n,
                        GLuint *renderbuffers);
```

**参数**

n

指定将要生成的渲染缓冲区对象名称的数量。

renderbuffers

指定一个生成的渲染缓冲区对象名称将要存储到其中的数组。

**描述**

glGenRenderbuffers 在 renderbuffers 中返回 n 个渲染缓冲区对象名称。并不能保证这些名称会组成一个连续的一组整数,不过可以保证这些返回的名称不会在调用 glGenRenderbuffers 前就立即被使用。

调用 glGenRenderbuffers 而返回的渲染缓冲区对象名称不会在后续的调用中返回,除非他们首先由 glDeleteRenderbuffers 来删除。

renderbuffers 中返回的名称将标记为已使用的,只是为了 glGenRenderbuffers 的目的,但是它们只在第一次绑定时才需要。

**错误**

如果 n 为负值,则产生 GL\_INVALID\_VALUE 错误。

**另外查看**

glFramebufferRenderbuffer, glDeleteRenderbuffers

**版权**

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glGenSamplers

生成采样器对象名称。

## C 规范

```
void glGenSamplers(GLsizei n,
                  GLuint *samplers);
```

### 参数

n

指定将要生成的采样器对象名称的数量。

samplers

指定一个生成的采样器对象名称将要存储到其中的数组。

### 描述

glGenSamplers 在 samplers 中返回 n 个采样器对象名称。并不能保证这些名称会组成一个连续的一组整数，不过可以保证这些返回的名称不会在调用 glGenSamplers 前就立即被使用。

调用 glGenSamplers 而返回的采样器对象名称不会在后续的调用中返回，除非他们首先由 glDeleteSamplers 来删除。

samplers 中返回的名称将标记为已使用的，只是为了 glGenSamplers 的目的，但是它们只在第一次绑定时才需要。

### 注意

glGenSamplers 只在 3.3 或更高版本的 GL 中可用。

### 错误

如果 n 为负值，则产生 GL\_INVALID\_VALUE 错误。

### 另外查看

glBindSampler, glIsSampler, glDeleteSamplers

### 版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/.

## glGenTextures

生成纹理名称。

## C 规范

```
void glGenTextures(GLsizei n,
                  GLuint * textures);
```

### 参数

n

指定将要生成的纹理名称的数量。

textures

指定一个生成的纹理名称将要存储到其中的数组。

### 描述

glGenTextures 在 textures 中返回 n 个纹理名称。并不能保证这些名称会组成一个连续的一组整数，不过可以保证这些返回的名称不会在调用 glGenTextures 前就立即被使用。

生成的纹理没有维度，它们会采用第一次绑定到的纹理目标的维度（参见 glBindTexture）。

调用 glGenTextures 而返回的纹理名称不会在后续的调用中返回，除非他们首先由 glDeleteTextures 来删除。

### 错误

如果 n 为负值，则产生 GL\_INVALID\_VALUE 错误。

### 相关 Get 函数

glIsTexture

### 另外查看

glBindTexture, glCopyTexImage1D, glCopyTexImage2D, glDeleteTextures, glGet, glGetTexParameter, glTexImage1D, glTexImage2D, glTexImage3D, glTexParameter

### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glGenVertexArrays

生成顶点数组对象名称。

### C 规范

```
void glGenVertexArrays(GLsizei n,
                       GLuint *arrays);
```

### 参数

n

指定将要生成的顶点数组对象名称的数量。

arrays

指定一个生成的顶点数组对象名称将要存储到其中的数组。

### 描述

glGenVertexArrays 在 arrays 中返回 n 个顶点数组对象名称。并不能保证这些名称会组成一个连续的一组整数，不过可以保证这些返回的名称不会在调用 glGenVertexArrays 前就立即被使用。

调用 glGenVertexArrays 而返回的顶点数组对象名称不会在后续的调用中返回，除非他们首先由 glDeleteVertexArrays 来删除。

arrays 中返回的名称将标记为已使用的，只是为了 glGenVertexArrays 的目的，但是它们只在第一次绑定时才需要。

### 错误

如果 n 为负值，则产生 GL\_INVALID\_VALUE 错误。

### 另外查看

glBindVertexArray, glDeleteVertexArrays

### 版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。 <http://opencontent.org/openpub/>。

## glGet

返回选定参数的一个或多个值。

### C 规范

```
void glGetBooleanv(GLenum pname,
                   GLboolean * params);
```

### C 规范

```
void glGetDoublev(GLenum pname,
                  GLdouble * params);
```

### C 规范

```
void glGetFloatv(GLenum pname,
                 GLfloat * params);
```

### C 规范

```
void glGetIntegerv(GLenum pname,
                  GLint * params);
```

### C 规范

```
void glGetInteger64v(GLenum pname,
                    GLint64 * params);
```

### 参数

pname



指定要返回的参数值。下面列出的符号常量可以被接受。

params

返回指定参数的一个或多个值。

#### C 规范

```
void glGetBooleani_v(GLenum pname,
                    GLuint index,
                    GLboolean * data);
```

#### C 规范

```
void glGetIntegeri_v(GLenum pname,
                    GLuint index,
                    GLint * data);
```

#### C 规范

```
void glGetInteger64i_v(GLenum pname,
                    GLuint index,
                    GLint64 * data);
```

#### 参数

pname

指定要返回的参数值。下面列出的符号常量可以被接受。

index

指定将要查询的特定元素的索引。

data

返回指定参数的一个或多个值。

#### 描述

这 4 个命令返回 GL 中简单状态变量的值。pname 是一个指示将要返回的状态变量的符号常量，而 params 则是指向一个防止返回数据的指定类型数组的指针。

如果 params 的类型与被请求的状态变量值的类型不同，那么将会执行类型变换。如果调用了 glGetBooleanv，那么当且仅当一个浮点值（或整数值）为 0.0（或 0）时，它将被转换为 GL\_FALSE。否则，它将被转换为 GL\_TRUE。如果调用了 glGetIntegerv，那么将会返回布尔值 GL\_TRUE 或 GL\_FALSE，并且大部分浮点值将被四舍五入到最近的整数值。不过，浮点颜色和法线将返回一个线性映射，将 1.0 映射到能够表示的最大的正整数值，并将 -1.0 映射到能够表示的最小的负整数值。如果调用了 glGetFloatv 或 glGetDoublev，则会返回布尔值 GL\_TRUE 或 GL\_FALSE，并且整数值会被转换到浮点值。

下列符号常量都可以被 pname 所接受。

GL\_ACTIVE\_TEXTURE

params 返回单个值，代表活动的多纹理单元。初始值为 GL\_TEXTURE0。参见 glActiveTexture。

GL\_ALIASED\_LINE\_WIDTH\_RANGE

params 返回一对值，指示支持锯齿线的宽度范围。参见 glLineWidth。

GL\_SMOOTH\_LINE\_WIDTH\_RANGE

params 返回一对值，指示支持平滑（抗锯齿）线的宽度范围。参见 glLineWidth。

GL\_SMOOTH\_LINE\_WIDTH\_GRANULARITY

params 返回单个值，指示应用到平滑线宽度参数的分层方式的层次。

GL\_ARRAY\_BUFFER\_BINDING

params 返回单个值，即当前绑定到 GL\_ARRAY\_BUFFER 目标的缓冲区对象名。如果没有缓冲区对象被绑定到这个目标，则返回 0。初始值为 0。参见 glBindBuffer。

GL\_BLEND

params 返回单个布尔值，指示是否激活混合。初始值为 GL\_FALSE。参见 glBlendFunc。

GL\_BLEND\_COLOR

params 返回 4 个值，分别为混合颜色的红色、绿色、蓝色和 Alpha 分量值。参见 glBlendColor。

GL\_BLEND\_DST\_ALPHA

params 返回一个值，即识别 Alpha 目的混合函数的符号常量。初始值为 GL\_ZERO。参见和 glBlendFunc 和

glBlendFuncSeparate。

GL\_BLEND\_DST\_RGB

params 返回一个值，即识别 RGB 目的混合函数的符号常量。初始值为 GL\_ZERO。参见和 glBlendFunc 和 glBlendFuncSeparate。

GL\_BLEND\_EQUATION\_RGB

params 返回一个值，即指示 RGB 混合方程是否为 GL\_FUNC\_ADD、GL\_FUNC\_SUBTRACT、GL\_FUNC\_REVERSE\_SUBTRACT、GL\_MIN 或 GL\_MAX 的符号常量。参见 glBlendEquationSeparate。

GL\_BLEND\_EQUATION\_ALPHA

params 返回一个值，即指示 Alpha 混合方程是否为 GL\_FUNC\_ADD、GL\_FUNC\_SUBTRACT、GL\_FUNC\_REVERSE\_SUBTRACT、GL\_MIN 或 GL\_MAX 的符号常量。参见 glBlendEquationSeparate。

GL\_BLEND\_SRC\_ALPHA

params 返回一个值，即识别 Alpha 源混合函数的符号常量。初始值为 GL\_ONE。参见和 glBlendFunc 和 glBlendFuncSeparate。

GL\_BLEND\_SRC\_RGB

params 返回一个值，即识别 RGB 源混合函数的符号常量。

初始值为 GL\_ONE。参见和 glBlendFunc 和 glBlendFuncSeparate。

GL\_COLOR\_CLEAR\_VALUE

params 返回 4 个值，即用于清除颜色缓冲区的红色、绿色、蓝色和 Alpha 值。如果被请求的话，内部浮点表示法将被线性映射到整数上，这样 1.0 将会返回能够表示的最大正整数，而 -1.0 则会返回能够表示的最小负整数。初始值为 (0, 0, 0, 0)。参见 glClearColor。

GL\_COLOR\_LOGIC\_OP

返回一个单个布尔值，指示一个片段的 RGBA 颜色值是否使用一个逻辑操作来合并到帧缓冲区中。初始值为 GL\_FALSE。参见 glLogicOp。

GL\_COLOR\_WRITEMASK

params 返回 4 个布尔值，即颜色缓冲区的红色、绿色、蓝色和 Alpha 写入激活。初始值为 (GL\_TRUE, GL\_TRUE, GL\_TRUE, GL\_TRUE)。参见 glColorMask。

GL\_COMPRESSED\_TEXTURE\_FORMATS

params 返回一个符号常量的 GL\_NUM\_COMPRESSED\_TEXTURE\_FORMATS，指示哪个压缩纹理格式是可用的。参见 glCompressedTexImage2D。

GL\_CULL\_FACE

params 返回一个布尔值，指示是否激活多边形剔除。初始值为 GL\_FALSE。参见 glCullFace。

GL\_CURRENT\_PROGRAM

params 返回一个值，即当前活动的程序对象的名称，或者在没有活动程序对象时为 0。参见 glUseProgram。

GL\_DEPTH\_CLEAR\_VALUE

params 返回一个值，即用来清除深度缓冲区的值。如果被请求的话，内部浮点表示法将被线性映射到整数上，这样 1.0 将会返回能够表示的最大正整数，而 -1.0 则会返回能够表示的最小负整数。初始值为 1。参见 glClearDepth。

GL\_DEPTH\_FUNC

params 返回一个值，即指示深度比较函数的符号常量。初始值为 GL\_LESS。参见 glDepthFunc。

GL\_DEPTH\_RANGE

params 返回两个值，即深度缓冲区的近映射界限和远映射界限。如果被请求的话，内部浮点表示法将被线性映射到整数上，这样 1.0 将会返回能够表示的最大正整数，而 -1.0 则会返回能够表示的最小负整数。初始值为 (0, 1)。参见 glDepthRange。

GL\_DEPTH\_TEST

params 返回一个单个布尔值，指示是否激活片段深度测试。初始值为 GL\_FALSE。参见 glDepthFunc 和 glDepthRange。

GL\_DEPTH\_WRITEMASK

params 返回一个单个布尔值，指示是否激活深度缓冲区写入。初始值为 GL\_TRUE。参见 glDepthMask。

GL\_DITHER

params 返回一个单个布尔值，指示是否激活片断颜色抖动和索引抖动。初始值为 GL\_TRUE。

GL\_DOUBLEBUFFER

params 返回一个单个布尔值, 指示是否支持双缓冲区。

GL\_DRAW\_BUFFER

params 返回一个值, 即一个指示要写入到哪个缓冲区的符号常量。参见 glDrawBuffer。如果有后置缓冲区, 则初始值为 GL\_BACK, 否则初始值为 GL\_FRONT。

GL\_DRAW\_BUFFER0

params 返回一个值, 即一个指示要被相应输出颜色写入到哪个缓冲区的符号常量。参见 glDrawBuffers。如果有后置缓冲区, 则 GL\_DRAW\_BUFFER0 的初始值为 GL\_BACK, 否则为 GL\_FRONT。对于所有其他输出颜色, 绘制缓冲区的初始值为 GL\_NONE。

GL\_DRAW\_FRAMEBUFFER\_BINDING

params 返回一个值, 即当前绑定到 GL\_DRAW\_FRAMEBUFFER 目标的帧缓冲区对象的名称。如果默认帧缓冲区被绑定, 那么这个值将为 0。初始值为 0。参见 glBindFramebuffer。

GL\_READ\_FRAMEBUFFER\_BINDING

params 返回一个值, 即当前绑定到 GL\_READ\_FRAMEBUFFER 目标的帧缓冲区对象的名称。如果默认帧缓冲区被绑定, 那么这个值将为 0。初始值为 0。参见 glBindFramebuffer。

GL\_ELEMENT\_ARRAY\_BUFFER\_BINDING

params 返回单个值, 即当前绑定到 GL\_ELEMENT\_ARRAY\_BUFFER 目标的缓冲区对象名。如果没有缓冲区对象被绑定到这个目标, 则返回 0。初始值为 0。参见 glBindBuffer。

GL\_RENDERBUFFER\_BINDING

params 返回单个值, 即当前绑定到 GL\_RENDERBUFFER 目标的帧缓冲区对象名。如果没有帧缓冲区对象被绑定到这个目标, 则返回 0。初始值为 0。参见 glBindRenderbuffer。

GL\_FRAGMENT\_SHADER\_DERIVATIVE\_HINT

params 返回一个值, 即指示片段着色器派生精度提示模式的符号常量。初始值为 GL\_DONT\_CARE。参见 glHint。

GL\_LINE\_SMOOTH

params 返回一个布尔值, 指示是否激活线抗锯齿。初始值为 GL\_FALSE。参见 glLineWidth。

GL\_LINE\_SMOOTH\_HINT

params 返回一个值, 即一个指示先反锯齿提示模式的符号常量。初始值为 GL\_DONT\_CARE。参见 glHint。

GL\_LINE\_WIDTH

params 返回一个值, 即由指定的线宽度 glLineWidth。初始值为 1。

GL\_LINE\_WIDTH\_GRANULARITY

params 返回一个值, 即抗锯齿线的相邻的支持宽度之间的差别。参见 glLineWidth。

GL\_LINE\_WIDTH\_RANGE

params 返回两个值, 即抗锯齿线支持的最小和最大宽度。参见 glLineWidth。参见 glLineWidth。

GL\_LOGIC\_OP\_MODE

params 返回一个值, 即一个指示选定的逻辑操作模式的符号常量。初始值为 GL\_COPY。参见 glLogicOp。

GL\_MAX\_3D\_TEXTURE\_SIZE

params 返回一个值, 即 GL 能够处理的最大 3D 纹理的大体估计。这个值必须至少为 64。使用 GL\_PROXY\_TEXTURE\_3D 来确定一个纹理是否太大。参见 glTexImage3D。

GL\_MAX\_CLIP\_DISTANCES

params 返回一个值, 即应用程序定义的裁剪距离的最大数量。这个值必须至少为 8。

GL\_MAX\_COMBINED\_FRAGMENT\_UNIFORM\_COMPONENTS

params 返回一个值, 即所有统一块中片段着色器统一变量字节的数量(包括默认值)。这个值必须至少必须为 1。参见 glUniform。

GL\_MAX\_COMBINED\_TEXTURE\_IMAGE\_UNITS

params 返回一个值, 即能够被用于从顶点着色器和片段处理器组合来访问纹理贴图的最大支持纹理图像单元。如果顶点着色器和片段处理器都访问同一个纹理图像单元, 那么这将被看作是在这个限制内使用两个纹理图像单元。这个值必须至少必须为 48。参见 glActiveTexture。

GL\_MAX\_COMBINED\_VERTEX\_UNIFORM\_COMPONENTS

params 返回一个值, 即所有统一块中顶点着色器统一变量字节的数量(包括默认值)。这个值必须至少必须为 1。参见 glUniform。

GL\_MAX\_COMBINED\_GEOMETRY\_UNIFORM\_COMPONENTS

params 返回一个值, 即所有统一块中几何图形着色器统一变量字节的数量(包括默认值)。这个值必须至少必须

须为 1。参见 glUniform。

GL\_MAX\_VARYING\_COMPONENTS

params 返回一个值, varying 变量的分量数, 必须至少为 60。

GL\_MAX\_COMBINED\_UNIFORM\_BLOCKS

params 返回一个值, 即每个程序的统一块的最大数量。这个值必须至少必须为 36。参见 glUniformBlockBinding。

GL\_MAX\_CUBE\_MAP\_TEXTURE\_SIZE

params 返回一个值, 这个值给出了 GL 能够处理的最大立方体贴图纹理的大体估计。这个值必须至少为 1024。

使用 GL\_PROXY\_TEXTURE\_CUBE\_MAP 来确定一个纹理是否太大。参见 glTexImage2D。

GL\_MAX\_DRAW\_BUFFERS

params 返回一个值, 即能够写入到一个片段着色器的同步输出的最大数量。这个值必须至少必须为 8。参见 glDrawBuffers。

GL\_MAX\_DUALSOURCE\_DRAW\_BUFFERS

params 返回一个值, 即应使用双源混合时活动绘制缓冲区的最大数量。这个值必须至少必须为 1。参见 glBlendFunc 和 glBlendFuncSeparate。

GL\_MAX\_ELEMENTS\_INDICES

params 返回一个值, 即顶点数组索引的推荐最大数量。

参见 glDrawRangeElements。

GL\_MAX\_ELEMENTS\_VERTICES

params 返回一个值, 即顶点数组顶点的推荐最大数量。

参见 glDrawRangeElements。

GL\_MAX\_FRAGMENT\_UNIFORM\_COMPONENTS

params 返回一个值, 即能够保存在片段着色器的统一变量存储中的独立浮点值、整数或布尔值的最大数量。

这个值必须至少必须为 1024。参见 glUniform。

GL\_MAX\_FRAGMENT\_UNIFORM\_BLOCKS

params 返回一个值, 即每个片段着色器的统一块的最大数量。

这个值必须至少必须为 12。参见 glUniformBlockBinding。

GL\_MAX\_FRAGMENT\_INPUT\_COMPONENTS

params 返回一个值, 即由片段着色器读取的输入的分量最大数量, 必须至少为 128。

GL\_MIN\_PROGRAM\_TEXEL\_OFFSET

params 返回一个值, 即一个纹理查询中允许的最小 texel 偏置, 必须至多为 -8。

GL\_MAX\_PROGRAM\_TEXEL\_OFFSET

params 返回一个值, 即一个纹理查询中允许的最大 texel 偏置, 必须至少为 7。

GL\_MAX\_RECTANGLE\_TEXTURE\_SIZE

params 返回一个值, 这个值给出了 GL 能够处理的最大矩形纹理的大体估计。这个值必须至少为 1024。使用 GL\_PROXY\_RECTANGLE\_TEXTURE 来确定一个纹理是否太大。参见 glTexImage2D。

GL\_MAX\_TEXTURE\_IMAGE\_UNITS

params 返回一个值, 即能够被用于从片段着色器来访问纹理贴图的最大支持纹理图像单元。这个值必须至少必须为 16。参见 glActiveTexture。

GL\_MAX\_TEXTURE\_LOD\_BIAS

params 返回一个值, 即纹理层次细节偏移的最大绝对值。这个值必须至少为 2.0。

GL\_MAX\_TEXTURE\_SIZE

params 返回一个值, 这个值给出了 GL 能够处理的最大纹理的大体估计。这个值必须至少为 1024。使用类似 GL\_PROXY\_TEXTURE\_1D 或 GL\_PROXY\_TEXTURE\_2D 这样的纹理代理来确定一个纹理是否太大。参见 glTexImage1D 和 glTexImage2D。

GL\_MAX\_RENDERBUFFER\_SIZE

params 返回一个值, 这个值表示渲染缓冲区的最大支持大小。

参见 glFramebufferRenderbuffer。

GL\_MAX\_ARRAY\_TEXTURE\_LAYERS

params 返回一个值, 这个值表示一个数组纹理中允许的最大层数, 必须至少为 256。参见 glTexImage2D。

GL\_MAX\_TEXTURE\_BUFFER\_SIZE

params 返回一个值, 这个值给出了一个纹理缓冲区对象的 texel 数组中允许的最大 texel 数量。这个值必须至少

为 65536。

`GL_MAX_UNIFORM_BLOCK_SIZE`

params 返回一个值, 即一个统一块大小的最大值, 以基本机器单元为单位。

这个值必须至少必须为 16384。参见 `glUniformBlockBinding`。

`GL_MAX_VARYING_FLOATS`

params 返回一个值, 即处理顶点和片段着色器使用的 varying 变量时可用的插值器 (interpolator) 的最大数量。这个值代表能够被插值的独立浮点值的最大数量。作为向量、矩阵和数组而声明的变化变量都将消耗多个插值器。这个值必须至少为 32。

`GL_MAX_VERTEX_ATTRIBS`

params 返回一个值, 即一个顶点着色器能够访问的 4 分量通用顶点属性的最大数量。这个值必须至少必须为 16。参见 `glVertexAttrib`。

`GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS`

params 返回一个值, 即能够被用于从片顶点着色器来访问纹理贴图的最大支持纹理图像单元。这个值至少可以为 16。参见 `glActiveTexture`。

`GL_MAX_GEOMETRY_TEXTURE_IMAGE_UNITS`

params 返回一个值, 即能够被用于从几何图形着色器来访问纹理贴图的最大支持纹理图像单元。这个值必须至少为 16。参见 `glActiveTexture`。

`GL_MAX_VERTEX_UNIFORM_COMPONENTS`

params 返回一个值, 即能够保存在顶点着色器的统一变量存储中的独立浮点值、整数值或布尔值的最大数量。这个值必须至少为 1024。参见 `glUniform`。

`GL_MAX_VERTEX_OUTPUT_COMPONENTS`

params 返回一个值, 即由顶点着色器写出分量的最大数量, 必须至少为 64。

`GL_MAX_GEOMETRY_UNIFORM_COMPONENTS`

params 返回一个值, 即能够保存在几何图形着色器的统一变量存储中的独立浮点值、整数值或布尔值的最大数量。这个值必须至少为 1024。参见 `glUniform`。

`GL_MAX_SAMPLE_MASK_WORDS`

params 返回一个值, 即应采样遮罩字节的最大数量。

`GL_MAX_COLOR_TEXTURE_SAMPLES`

params 返回一个值, 即一个彩色多纹理中样本的最大数量。

`GL_MAX_DEPTH_TEXTURE_SAMPLES`

params 返回一个值, 即一个多重采样深度或深度-模板纹理中样本的最大数量。

`GL_MAX_DEPTH_TEXTURE_SAMPLES`

params 返回一个值, 即一个多重采样深度或深度-模板纹理中样本的最大数量。

`GL_MAX_INTEGER_SAMPLES`

params 返回一个值, 即一个整数格式多重采样缓冲区中支持样本的最大数量。

`GL_MAX_SERVER_WAIT_TIMEOUT`

params 返回一个值, 即最大 `glWaitSync` 超时间隔。

`GL_MAX_UNIFORM_BUFFER_BINDINGS`

params 返回一个值, 即这个环境中统一缓冲区绑定点的最大数量, 必须至少为 36。

`GL_MAX_UNIFORM_BLOCK_SIZE`

params 返回一个值, 即一个统一块大小的最大值, 以基本机器单元为单位, 最少必须为 16384。

`GL_UNIFORM_BUFFER_OFFSET_ALIGNMENT`

params 返回一个值, 即统一缓冲区大小和偏置的最小所需排列。

`GL_MAX_VERTEX_UNIFORM_BLOCKS`

params 返回一个值, 即每个顶点着色器的统一块的最大数量。

这个值必须至少为 12。参见 `glUniformBlockBinding`。

`GL_MAX_GEOMETRY_UNIFORM_BLOCKS`

params 返回一个值, 即每个几何图形着色器的统一块的最大数量。这个值必须至少为 12。参见 `glUniformBlockBinding`。

`GL_MAX_GEOMETRY_INPUT_COMPONENTS`

params 返回一个值, 即由几何图形着色器读取的输入分量的最大数量, 必须至少为 64。

**GL\_MAX\_GEOMETRY\_OUTPUT\_COMPONENTS**

params 返回一个值, 即由几何图形着色器写出分量的最大数量, 必须至少为 128。

**GL\_MAX\_VIEWPORT\_DIMS**

params 返回两个值, 视口支持的最大宽度和高度。他们必须至少与渲染到的显示器的可视大小一样大。

参见 glViewport。

**GL\_NUM\_COMPRESSED\_TEXTURE\_FORMATS**

params 返回单个整数值, 指示可用压缩纹理格式的数量。最小值为 4。参见 glCompressedTexImage2D。

**GL\_PACK\_ALIGNMENT**

params 返回一个值, 即用于向内存写入像素数据的字节排列。

初始值为 4。参见 glPixelStore。

**GL\_PACK\_IMAGE\_HEIGHT**

params 返回一个值, 即用于向内存写入像素数据的图像高度。初始值为 0。参见 glPixelStore。

**GL\_PACK\_LSB\_FIRST**

params 返回单个布尔值, 指示下入到内存的单个位像素 (single-bit pixel) 是否先写入每个无符号字节的最小位。初始值为 GL\_FALSE。参见 glPixelStore。

**GL\_PACK\_ROW\_LENGTH**

params 返回一个值, 即用于向内存写入像素数据的行长度。初始值为 0。参见 glPixelStore。

**GL\_PACK\_SKIP\_IMAGES**

params 返回一个值, 即第一个像素被写入到内存中之前所跳过的像素图像的数量。初始值为 0。参见 glPixelStore。

**GL\_PACK\_SKIP\_PIXELS**

params 返回一个值, 即第一个像素被写入到内存中之前所跳过的像素位置的数量。初始值为 0。参见 glPixelStore。

**GL\_PACK\_SKIP\_ROWS**

params 返回一个值, 即第一个像素被写入到内存中之前所跳过的像素位置的行数。初始值为 0。参见 glPixelStore。

**GL\_PACK\_SWAP\_BYTES**

params 返回单个布尔值, 指示二字节和 4 字节像素索引和分量的字节是否在写入到内存之前进行交换。初始值为 GL\_FALSE。参见 glPixelStore。

**GL\_PIXEL\_PACK\_BUFFER\_BINDING**

params 返回单个值, 即当前绑定到 GL\_PIXEL\_PACK\_BUFFER 目标的缓冲区对象名。如果没有缓冲区对象被绑定到这个目标, 则返回 0。

初始值为 0。参见 glBindBuffer。

**GL\_PIXEL\_UNPACK\_BUFFER\_BINDING**

params 返回单个值, 即当前绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标的缓冲区对象名。如果没有缓冲区对象被绑定到这个目标, 则返回 0。初始值为 0。参见 glBindBuffer。

**GL\_POINT\_FADE\_THRESHOLD\_SIZE**

params 返回一个值, 即确定点大小的点大小阈值。参见 glPointParameter。

**GL\_PRIMITIVE\_RESTART\_INDEX**

params 返回一个值, 即当前图元重启索引。初始值为 0。参见 glPrimitiveRestartIndex。

**GL\_PROVOKING\_VERTEX**

params 返回一个值, 即当前选择的 “provoking vertex” 约定。初始值为 GL\_LAST\_VERTEX\_CONVENTION。参见 glProvokingVertex。

**GL\_POINT\_SIZE**

params 返回一个值, 即由 glPointSize 指定的点大小。初始值为 1。

**GL\_POINT\_SIZE\_GRANULARITY**

params 返回一个值, 即抗锯齿点的相邻的支持大小之间的差别。参见 glPointSize。

**GL\_POINT\_SIZE\_RANGE**

params 返回两个值, 即抗锯齿点支持大小的最小和最大值。大小的最小值必须至多为 1, 而最大值则至少为 1。参见 glPointSize。

**GL\_POLYGON\_OFFSET\_FACTOR**

params 返回一个值, 用于确定添加到在多边形进行光栅化时生成的每个片段深度值的变量偏置缩放因子。初始值为 0。参见 glPolygonOffset。

GL\_POLYGON\_OFFSET\_UNITS

params 返回一个值, 这个值将乘以一个平台特定的值, 然后加到在多边形进行光栅化时生成的每个片段的深度值上。

初始值为 0。参见 glPolygonOffset。

GL\_POLYGON\_OFFSET\_FILL

params 返回单个布尔值, 指示是否为填充模式下的多边形激活多边形偏置。初始值为 GL\_FALSE。参见 glPolygonOffset。

GL\_POLYGON\_OFFSET\_LINE

params 返回单个布尔值, 指示是否为线模式下的多边形激活多边形偏置。初始值为 GL\_FALSE。参见 glPolygonOffset。

GL\_POLYGON\_OFFSET\_POINT

params 返回单个布尔值, 指示是否为点模式下的多边形激活多边形偏置。初始值为 GL\_FALSE。参见 glPolygonOffset。

GL\_POLYGON\_SMOOTH

params 返回单个布尔值, 指示是否激活多边形抗锯齿。初始值为 GL\_FALSE。参见 glPolygonMode。

GL\_POLYGON\_SMOOTH\_HINT

params 返回一个值, 即一个指示多边形反锯齿提示模式的符号常量。初始值为 GL\_DONT\_CARE。参见 glHint。

GL\_READ\_BUFFER

params 返回一个值, 即一个指示要选择写入到哪个颜色缓冲区的符号常量。如果有后置缓冲区, 则初始值为 GL\_BACK, 否则初始值为 GL\_FRONT。参见 glReadPixels。

GL\_SAMPLE\_BUFFERS

params 返回单个整数值, 指示与帧缓冲区相关联的采样缓冲区的数量。参见 glSampleCoverage。

GL\_SAMPLE\_COVERAGE\_VALUE

params 返回单个正负点值, 代表当前的采样覆盖值。参见 glSampleCoverage。

GL\_SAMPLE\_COVERAGE\_INVERT

params 返回一个单个布尔值, 指示是否应该对临时覆盖值取反。参见 glSampleCoverage。

GL\_SAMPLER\_BINDING

params 返回单个值, 即当前绑定到活动纹理单元的采样器对象名。初始值为 0。参见 glBindSampler。

GL\_SAMPLES

params 返回单个整数值, 代表覆盖遮罩大小。参见 glSampleCoverage。

GL\_SCISSOR\_BOX

params 返回 4 个值, 依次为裁剪框的 x 和 y 窗口坐标, 然后是它的宽度和高度。初始状态下 x 和 y 窗口坐标都为 0, 而宽度和高度则按照窗口的大小进行设置。参见 glScissor。

GL\_SCISSOR\_TEST

params 返回单个布尔值, 指示是否激活裁剪。初始值为 GL\_FALSE。参见 glScissor。

GL\_STENCIL\_BACK\_FAIL

params 返回一个值, 这是一个符号常量, 指示在模板测试失败的情况下将对背向多边形采取什么样的处理方式。初始值为 GL\_KEEP。参见 glStencilOpSeparate。

GL\_STENCIL\_BACK\_FUNC

params 返回一个值, 这是一个符号常量, 指示采用哪个函数来为背向多边形来对模板参考值和模板缓冲区值进行比较。初始值为 GL\_ALWAYS。参见 glStencilFuncSeparate。

GL\_STENCIL\_BACK\_PASS\_DEPTH\_FAIL

params 返回一个值, 这是一个符号常量, 指示在模板测试通过但深度测试失败的情况下将对背向多边形采取什么样的处理方式。初始值为 GL\_KEEP。参见 glStencilOpSeparate。

GL\_STENCIL\_BACK\_PASS\_DEPTH\_PASS

params 返回一个值, 这是一个符号常量, 指示在模板测试通过并且深度测试也通过的情况下将对背向多边形采取什么样的处理方式。初始值为 GL\_KEEP。参见 glStencilOpSeparate。

GL\_STENCIL\_BACK\_REF

params 返回一个值, 即用来为背向多边形而与模板缓冲区的内容进行比较的参考值。初始值为 0。参见

glStencilFuncSeparate。

GL\_STENCIL\_BACK\_VALUE\_MASK

params 返回一个值，即一个遮罩，它应用于背向多边形，在对模板参考值和模板缓冲区值进行比较之前对它们都进行遮罩。初始值都为 1。参见 glStencilFuncSeparate。

GL\_STENCIL\_BACK\_WRITEMASK

params 返回一个值，即控制背向多边形模板位平面写入的遮罩。初始值都为 1。参见 glStencilMaskSeparate。

GL\_STENCIL\_CLEAR\_VALUE

params 返回一个值，即模板位平面被清除到的索引。初始值为 0。参见 glClearStencil。

GL\_STENCIL\_FAIL

params 返回一个值，这是一个符号常量，指示在模板测试失败的情况下将采取什么样的处理方式。初始值为 GL\_KEE。参见 glStencilOp。这个模板状态只影响非多边形和前向多边形。背向多边形使用单独的模板状态。参见 glStencilOpSeparate。

GL\_STENCIL\_FUNC

params 返回一个值，这是一个符号常量，指示采用哪个函数来对模板参考值和模板缓冲区值进行比较。初始值为 GL\_ALWAYS。参见 glStencilFunc。这个模板状态只影响非多边形和前向多边形。背向多边形使用单独的模板状态。参见 glStencilFuncSeparate。

GL\_STENCIL\_PASS\_DEPTH\_FAIL

params 返回一个值，这是一个符号常量，指示在模板测试通过但深度测试失败的情况下将采取什么样的处理方式。初始值为 GL\_KEE。参见 glStencilOp。

这个模板状态只影响非多边形和前向多边形。背向多边形使用单独的模板状态。参见 glStencilOpSeparate。

GL\_STENCIL\_PASS\_DEPTH\_PASS

params 返回一个值，这是一个符号常量，指示在模板测试通过并且深度测试也通过的情况下将采取什么样的处理方式。初始值为 GL\_KEE。参见 glStencilOp。

这个模板状态只影响非多边形和前向多边形。背向多边形使用单独的模板状态。参见 glStencilOpSeparate。

GL\_STENCIL\_REF

params 返回一个值，即用来与模板缓冲区的内容进行比较的参考值。初始值为 0。参见 glStencilFunc。这个模板状态只影响非多边形和前向多边形。背向多边形使用单独的模板状态。参见 glStencilFuncSeparate。

GL\_STENCIL\_TEST

params 返回一个单个布尔值，指示是否激活片段模板测试。初始值为 GL\_FALSE。参见 glStencilFunc 和 glStencilOp。

GL\_STENCIL\_VALUE\_MASK

params 返回一个值，即一个遮罩，它在对模板参考值和模板缓冲区值进行比较之前对它们都进行遮罩。初始值都为 1。参见 glStencilFunc。这个模板状态只影响非多边形和前向多边形。

背向多边形使用单独的模板状态。参见 glStencilFuncSeparate。

GL\_STENCIL\_WRITEMASK

params 返回一个值，即控制模板位平面写入的遮罩。初始值都为 1。参见 glStencilMask。这个模板状态只影响非多边形和前向多边形。背向多边形使用单独的模板状态。参见 glStencilMaskSeparate。

GL\_STEREO

params 返回一个单个布尔值，指示是否支持立体缓冲区（左缓冲区和右缓冲区）。

GL\_SUBPIXEL\_BITS

params 返回一个值，即用来在窗口坐标中定位光栅化几何图形的亚像素（subpixel）分辨率的位数估计值。这个值必须至少为 4。

GL\_TEXTURE\_BINDING\_1D

params 返回单个值，即当前绑定到 GL\_TEXTURE\_1D 目标的纹理。初始值为 0。参见 glBindTexture。

GL\_TEXTURE\_BINDING\_1D\_ARRAY

params 返回单个值，即当前绑定到 GL\_TEXTURE\_1D\_ARRAY 目标的纹理。初始值为 0。参见 glBindTexture。

GL\_TEXTURE\_BINDING\_2D

params 返回单个值，即当前绑定到 GL\_TEXTURE\_2D 目标的纹理。初始值为 0。参见 glBindTexture。

GL\_TEXTURE\_BINDING\_2D\_ARRAY

params 返回单个值，即当前绑定到 GL\_TEXTURE\_2D\_ARRAY 目标的纹理。初始值为 0。参见 glBindTexture。

GL\_TEXTURE\_BINDING\_2D\_MULTISAMPLE

params 返回单个值, 即当前绑定到 GL\_TEXTURE\_2D\_MULTISAMPLE 目标的纹理。初始值为 0。参见 glBindTexture。

GL\_TEXTURE\_BINDING\_2D\_MULTISAMPLE\_ARRAY

params 返回单个值, 即当前绑定到 GL\_TEXTURE\_2D\_MULTISAMPLE\_ARRAY 目标的纹理。初始值为 0。参见 glBindTexture。

GL\_TEXTURE\_BINDING\_3D

params 返回单个值, 即当前绑定到 GL\_TEXTURE\_3D 目标的纹理。初始值为 0。参见 glBindTexture。

GL\_TEXTURE\_BINDING\_BUFFER

params 返回单个值, 即当前绑定到 GL\_TEXTURE\_BUFFER 目标的纹理。初始值为 0。参见 glBindTexture。

GL\_TEXTURE\_BINDING\_CUBE\_MAP

params 返回单个值, 即当前绑定到 GL\_TEXTURE\_CUBE\_MAP 目标的纹理。初始值为 0。参见 glBindTexture。

GL\_TEXTURE\_BINDING\_RECTANGLE

params 返回单个值, 即当前绑定到 GL\_TEXTURE\_RECTANGLE 目标的纹理。初始值为 0。参见 glBindTexture。

GL\_TEXTURE\_COMPRESSION\_HINT

params 返回单个整数, 指示纹理压缩提示的模式。初始值为 GL\_DONT\_CARE。

GL\_TEXTURE\_BUFFER\_BINDING

params 返回单个值, 即当前绑定的纹理缓冲区对象名。

初始值为 0。参见 glBindBuffer。

GL\_TIMESTAMP

params 返回单个值, 即当前 GL 时间的 64 位值。参见 glQueryCounter。

GL\_TRANSFORM\_FEEDBACK\_BUFFER\_BINDING

当使用 glGet (例如 glGetIntegerv) 的非索引变量时, params 返回单个值, 即当前绑定到 GL\_TRANSFORM\_FEEDBACK\_BUFFER 目标的缓冲区对象名。如果没有缓冲区对象被绑定到这个目标, 则返回 0。

当使用 glGet (例如 glGetIntegeri\_v) 的索引变量时, params 返回单个值, 即绑定到索引变换反馈属性流的缓冲区对象。所有目标的初始值都为 0。参见 glBindBuffer、glBindBufferBase 和 glBindBufferRange。

GL\_TRANSFORM\_FEEDBACK\_BUFFER\_START

当使用 glGet (例如 glGetInteger64i\_v) 的索引变量时, params 返回单个值, 即每个变换反馈属性流的绑定范围的初始偏置。所有流的初始值都为 0。参见 glBindBufferRange。

GL\_TRANSFORM\_FEEDBACK\_BUFFER\_SIZE

当使用 glGet (例如 glGetInteger64i\_v) 的索引变量时, params 返回单个值, 即每个变换反馈属性流的绑定范围的大小。所有流的初始值都为 0。参见 glBindBufferRange。

GL\_UNIFORM\_BUFFER\_BINDING

当使用 glGet (例如 glGetIntegerv) 的非索引变量时, params 返回单个值, 即当前绑定到 GL\_UNIFORM\_BUFFER 目标的缓冲区对象名。如果没有缓冲区对象被绑定到这个目标, 则返回 0。当使用 glGet (例如 glGetIntegeri\_v) 的索引变量时, params 返回单个值, 即绑定到索引统一缓冲区绑定点的缓冲区对象。所有目标的初始值都为 0。参见 glBindBuffer、glBindBufferBase 和 glBindBufferRange。

GL\_UNIFORM\_BUFFER\_START

当使用 glGet (例如 glGetInteger64i\_v) 的索引变量时, params 返回单个值, 即每个索引统一缓冲区绑定的绑定范围的初始偏置。所有绑定的初始值都为 0。参见 glBindBufferRange。

GL\_UNIFORM\_BUFFER\_SIZE

当使用 glGet (例如 glGetInteger64i\_v) 的索引变量时, params 返回单个值, 即每个索引统一缓冲区绑定的绑定范围的大小。所有绑定的初始值都为 0。参见 glBindBufferRange。

GL\_UNIFORM\_BUFFER\_OFFSET\_ALIGNMENT

params 返回单个值, 即统一缓冲区大小和偏置的最小所需排列。初始值为 1。参见 glUniformBlockBinding。

GL\_UNPACK\_ALIGNMENT

params 返回一个值, 即用于从内存中读取像素数据的字节排列。初始值为 4。参见 glPixelStore。

GL\_UNPACK\_IMAGE\_HEIGHT

params 返回一个值, 即用于从内存中读取像素数据的图像高度。

初始值为 0。参见 glPixelStore。

GL\_UNPACK\_LSB\_FIRST

params 返回单个布尔值, 指示首先从内存中读取的单个位像素 (single-bit pixel) 是否先写入每个无符号字节

的最小位。初始值为 GL\_FALSE。参见 glPixelStore。

GL\_UNPACK\_ROW\_LENGTH

params 返回一个值，即用于从内存中读取像素数据的行长度。初始值为 0。参见 glPixelStore。

GL\_UNPACK\_SKIP\_IMAGES

params 返回一个值，即第一个像素从内存中读取之前所跳过的像素图像的数量。初始值为 0。参见 glPixelStore。

GL\_UNPACK\_SKIP\_PIXELS

params 返回一个值，即第一个像素从内存中读取之前所跳过的像素位置的数量。初始值为 0。参见 glPixelStore。

GL\_UNPACK\_SKIP\_ROWS

params 返回一个值，即第一个像素从内存中读取之前所跳过的像素位置的行数。初始值为 0。参见 glPixelStore。

GL\_UNPACK\_SWAP\_BYTES

params 返回单个布尔值，指示二字节和 4 字节像素索引和分量的字节是否在从内存中读取之后进行交换。初始值为 GL\_FALSE。参见 glPixelStore。

GL\_NUM\_EXTENSIONS

params 返回一个值，即当前环境的 GL 实现所支持扩展的数量。参见 glGetString。

GL\_MAJOR\_VERSION

params 返回一个值，即当前环境所支持 OpenGL API 的主版本号。

GL\_MINOR\_VERSION

params 返回一个值，即当前环境所支持 OpenGL API 的辅版本号。

GL\_CONTEXT\_FLAGS

params 返回一个值，即这个环境创建所使用的标记（例如调试功能）。

GL\_VERTEX\_PROGRAM\_POINT\_SIZE

params 返回单个布尔值，指示是否激活顶点程序点大小模式。如果被启用，并且一个顶点着色器被激活，那么就会从着色器内建的 gl\_PointSize 中获取点大小。如果被禁用，并且一个顶点着色器被激活，那么就会从由 glPointSize 指定的点状态中获取点大小。初始值为 GL\_FALSE。

GL\_VIEWPORT

params 返回 4 个值，依次为视口的 x 和 y 窗口坐标，然后是它的宽度和高度。在初始情况下，x 和 y 窗口坐标都被设置为 0，而宽度和高度则被设置为 GL 将在其中进行渲染的窗口的宽度和高度。参见 glViewport。

许多布尔参数也可以使用 glIsEnabled 来更方便地进行查询。

**注意**

下列参数返回活动纹理单元的关联值：

GL\_TEXTURE\_1D、GL\_TEXTURE\_BINDING\_1D、GL\_TEXTURE\_2D、GL\_TEXTURE\_BINDING\_2D、GL\_TEXTURE\_3D 和 GL\_TEXTURE\_BINDING\_3D。

GL\_MAX\_RECTANGLE\_TEXTURE\_SIZE、GL\_MAX\_TEXTURE\_BUFFER\_SIZE、GL\_UNIFORM\_BUFFER\_BINDING、GL\_TEXTURE\_BUFFER\_BINDING、GL\_MAX\_VERTEX\_UNIFORM\_BLOCKS、GL\_MAX\_FRAGMENT\_UNIFORM\_BLOCKS、GL\_MAX\_COMBINED\_FRAGMENT\_UNIFORM\_COMPONENTS、GL\_MAX\_COMBINED\_VERTEX\_UNIFORM\_COMPONENTS、GL\_MAX\_COMBINED\_UNIFORM\_BLOCKS、GL\_MAX\_UNIFORM\_BLOCK\_SIZE 和 GL\_UNIFORM\_BUFFER\_OFFSET\_ALIGNMENT 只在 3.1 或更高版本的 GL 中可用。

GL\_MAX\_COMBINED\_GEOMETRY\_UNIFORM\_COMPONENTS、GL\_MAX\_GEOMETRY\_UNIFORM\_BLOCKS、GL\_MAX\_GEOMETRY\_INPUT\_COMPONENTS、GL\_MAX\_GEOMETRY\_OUTPUT\_COMPONENTS、GL\_MAX\_GEOMETRY\_OUTPUT\_VERTICES、GL\_MAX\_GEOMETRY\_TOTAL\_OUTPUT\_COMPONENTS 和 GL\_MAX\_GEOMETRY\_TEXTURE\_IMAGE\_UNITS 只在 3.2 或更高版本的 GL 中可用。

glGetInteger64v 和 glGetInteger64i 只在 3.2 或更高版本的 GL 中可用。

GL\_MAX\_DUALSOURCE\_DRAW\_BUFFERS、GL\_SAMPLER\_BINDING 和 GL\_TIMESTAMP 只在 3.3 或更高版本的 GL 中可用。

**错误**

如果 pname 不是一个可接受的值，则产生 GL\_INVALID\_ENUM 错误。

如果在索引状态 tarGet 的有效范围之外，那么 glGetBooleani\_v、glGetIntegeri\_v 或 glGetInteger64i\_v 将会产生 GL\_INVALID\_VALUE 错误。

**另外查看**

glGetActiveUniform, glGetAttachedShaders, glGetAttribLocation, glGetBufferParameter, glGetBufferPointerv,

glGetBufferSubData, glGetCompressedTexImage, glGetError, glGetProgram, glGetProgramInfoLog, glGetQueryiv, glGetQueryObject, glGetShader, glGetShaderInfoLog, glGetShaderSource, getString, glGetTexImage, glGetTexLevelParameter, glGetTexParameter, glGetUniformLocation, glGetVertexAttrib, glVertexAttribPointerv, glIsEnabled

## 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glGetActiveAttrib

返回关于指定程序对象的活动属性变量的信息。

### C 规范

```
void glGetActiveAttrib(GLuint program,
                      GLuint index,
                      GLsizei bufSize,
                      GLsizei * length,
                      GLint * size,
                      GLenum * type,
                      GLchar * name);
```

### 参数

program

指定将要被查询的程序对象。

index

指定将要被查询的属性变量的索引。

bufSize

指定允许 OpenGL 写入由 name 指示的字符缓冲区中的字符的最大数量。

length

如果除 NULL 以外的值被传递的话, 则返回实际上由 OpenGL 写入由 name 指示的字符串的字符数 (不包括空终止符)。

size

返回属性变量的大小。

type

返回属性变量的数据类型。

name

返回一个包括属性变量名的空终止字符串。

### 描述

glGetActiveAttrib 返回关于由 program 指定的程序对象的活动属性变量的信息。可以通过调用以 GL\_ACTIVE\_ATTRIBUTES 为参数的 glGetProgram 来获取活动属性数量。index 的值取 0 时选择第一个活动属性变量。index 的允许取值范围为 0 到活动属性变量的数量减去 1 的值。

一个顶点着色器可以使用内建属性变量、用户定义属性变量, 或两者同时使用。内建属性变量有一个 “gl\_” 前缀, 并引用常规的 OpenGL 顶点属性 (例如 gl\_Vertex、gl\_Normal 等, 完整列表参见 OpenGL 着色语言规范)。用户定义的属性变量有任意的名称, 并通过有限的通用顶点属性来获取它们的值。一个属性变量 (无论是内建的还是用户定义的) 如果在程序执行过程中可以进行访问的链接操作过程中决定, 那么它会被看作是活动的。

因此, program 应该在这之前就成为一次 glLinkProgram 调用的目标, 但是不需要已经成功地进行链接。

被请求用来存储 program 中最长的属性变量名称字符缓冲器的大小可以通过调用值为 GL\_ACTIVE\_ATTRIBUTE\_MAX\_LENGTH 的 glGetProgram 来获得。这个值能够被用来分配大小足够的缓冲区来存储返回的属性名。字符缓冲器的大小在 bufSize 中传递, 而指向这个字符缓冲区的指针会传递到 name。

glGetActiveAttrib 返回 index 指示的属性变量的名称, 存储在由 name 指定的字符缓冲区中。返回的字符串以空字符 null 为结尾。写入这个缓冲区的字符的实际数量在 length 中返回, 这个数字不包括空终止字符。如果返回字符串的长度没有被请求, 那么可以在 length 自变量中传递一个 NULL 值。

type 自变量将返回一个指向属性变量数据类型的指针。可能返回符号常量 GL\_FLOAT、GL\_FLOAT\_VEC2、

GL\_FLOAT\_VEC3、GL\_FLOAT\_VEC4、GL\_FLOAT\_MAT2、GL\_FLOAT\_MAT3、GL\_FLOAT\_MAT4、GL\_FLOAT\_MAT2x3、GL\_FLOAT\_MAT2x4、GL\_FLOAT\_MAT3x2、GL\_FLOAT\_MAT3x4、GL\_FLOAT\_MAT4x2、GL\_FLOAT\_MAT4x3、GL\_INT、GL\_INT\_VEC2、GL\_INT\_VEC3、GL\_INT\_VEC4、GL\_UNSIGNED\_INT\_VEC、GL\_UNSIGNED\_INT\_VEC2、GL\_UNSIGNED\_INT\_VEC3 或 GL\_UNSIGNED\_INT\_VEC4。自变量 `size` 将在 `type` 中返回的类型单元中返回属性的大小。

活动属性变量列表可以包含在内建属性变量（这些属性变量以“gl\_”前缀开始），同样也可以包含在用户定义属性变量名称中。

这个函数将返回它所能返回的尽可能多的关于指定活动属性变量的信息。

如果没有可用的信息，那么 `length` 将为 0，而 `name` 将为空字符串。在一个链接操作失败后调用这个函数就可能会出现这种情况。如果出现错误，那么返回值 `length`、`size`、`type` 和 `name` 将为未修改的。

#### 错误

如果 `program` 不是由 OpenGL 产生的值，则产生 GL\_INVALID\_VALUE 错误。

如果 `program` 不是一个程序对象，则产生 GL\_INVALID\_OPERATION 错误。

如果 `index` 大于或等于 `program` 中活动属性变量的数量，则产生 GL\_INVALID\_VALUE 错误。

如果 `bufSize` 小于 0，则产生 GL\_INVALID\_VALUE 错误。

#### 相关 Get 函数

`glGet`，其自变量为 GL\_MAX\_VERTEX\_ATTRIBS。

`glGetProgram`，自变量为 GL\_ACTIVE\_ATTRIBUTES 或 GL\_ACTIVE\_ATTRIBUTE\_MAX\_LENGTH。

`glIsProgram`

#### 另外查看

`glBindAttribLocation`、`glLinkProgram`、`glVertexAttrib`、`glVertexAttribPointer`

#### 版权

Copyright © 2003–2005 3Dlabs Inc. Ltd. Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glGetActiveUniform

返回关于一个指定程序对象的活动统一变量的信息。

### C 规范

```
void glGetActiveUniform(GLuint program,
                        GLuint index,
                        GLsizei bufSize,
                        GLsizei *length,
                        GLint *size,
                        GLenum *type,
                        GLchar *name);
```

#### 参数

`program`

指定将要被查询的程序对象。

`index`

指定将要被查询的统一变量的索引。

`bufSize`

指定允许 OpenGL 写入由 `name` 指示的字符缓冲区中的字符的最大数量。

`length`

如果除 NULL 以外的值被传递的话，则返回实际上由 OpenGL 写入由 `name` 指示的字符串的字符数（不包括空终止符）。

`size`

返回统一变量的大小。

`type`

返回统一变量的数据类型。

`name`

返回一个包括统一变量名的空终止字符串。

### 描述

glGetActiveUniform

返回关于由 program 指定的程序对象的活动统一变量的信息。可以通过调用以 GL\_ACTIVE\_UNIFORMS 为参数的 glGetProgram 来获取活动统一变量数量。index 的值取 0 时选择第一个活动统一变量。index 的允许取值范围为 0 到活动统一变量的数量减去 1 的值。

着色器可以使用内建统一变量、用户定义统一变量，或两者同时使用。内建统一变量有一个“gl\_”前缀，并引用已存在的 OpenGL 状态或得自这种状态的值（例如 gl\_DepthRangeParameters，完整列表参见 OpenGL 着色语言规范）。用户定义的统一变量有任意的名称，并通过调用 glUniform 来从应用程序中获取它们的值。一个统一变量（无论是内建的还是用户定义的）如果在程序执行过程中可以进行访问的链接操作过程中决定，那么它会被看作是活动的。

因此，program 应该在这之前就成为一次 glLinkProgram 调用的目标，但是不需要已经成功地进行链接。

被请求用来存储 program 中最长的统一变量名称字符缓冲器的大小可以通过调用值为 GL\_ACTIVE\_UNIFORM\_MAX\_LENGTH 的 glGetProgram 来获得。这个值能够被用来分配大小足够的缓冲区来存储返回的统一变量名。字符缓冲器的大小在 bufSize 中传递，而指向这个字符缓冲区的指针会传递到 name。

glGetActiveUniform

返回 index 指示的统一变量的名称，存储在由 name 指定的字符缓冲区中。返回的字符串以空字符 null 为结尾。写入这个缓冲区的字符的实际数量在 length 中返回，这个数字不包括空终止字符。如果返回字符串的长度没有被请求，那么可以在 length 自变量中传递一个 NULL 值。

### type

自变量将返回一个指向统一变量数据类型的指针。下表列出了为统一变量返回的符号常量。

如果一个数组的一个或多个元素是活动的，那么数组的名称将会返回到 name，而类型则返回到 type，并且 size 参数将返回使用的最高数组元素索引，并加上 1，这是由编译器和/或连接器决定的。只有一个活动统一变量将为统一数组而报告。

作为结构或结构数组而声明的统一变量不会被这个函数直接返回。取而代之的是，这些统一变量中的每一个都会被简化到它的包括“.”和“[]”运算符的基本组成部分，这样每一个名称作为 glGetUniformLocation 的自变量都是合法的。这些简化统一变量中的每一个都算作一个活动统一变量，并分配给它一个索引。合法的名称不能是结构、结构数组，或者向量或数组的子分量。

返回的符号常量	着色器统一变量类型
GL_FLOAT	float
GL_FLOAT_VEC2	vec2
GL_FLOAT_VEC3	vec3
GL_FLOAT_VEC4	vec4
GL_INT	int
GL_INT_VEC2	ivec2
GL_INT_VEC3	ivec3
GL_INT_VEC4	ivec4
GL_UNSIGNED_INT	unsigned int
GL_UNSIGNED_INT_VEC2	uvec2
GL_UNSIGNED_INT_VEC3	uvec3
GL_UNSIGNED_INT_VEC4	uvec4
GL_BOOL	bool
GL_BOOL_VEC2	bvec2
GL_BOOL_VEC3	bvec3
GL_BOOL_VEC4	bvec4

续表

返回的符号常量	着色器统一变量类型
GL_FLOAT_MAT2	mat2
GL_FLOAT_MAT3	mat3
GL_FLOAT_MAT4	mat4
GL_FLOAT_MAT2x3	mat2x3
GL_FLOAT_MAT2x4	mat2x4
GL_FLOAT_MAT3x2	mat3x2
GL_FLOAT_MAT3x4	mat3x4
GL_FLOAT_MAT4x2	mat4x2
GL_FLOAT_MAT4x3	mat4x3
GL_SAMPLER_1D	sampler1D
GL_SAMPLER_2D	sampler2D
GL_SAMPLER_3D	sampler3D
GL_SAMPLER_CUBE	samplerCube
GL_SAMPLER_1D_SHADOW	sampler1DShadow
GL_SAMPLER_2D_SHADOW	sampler2DShadow
GL_SAMPLER_1D_ARRAY	sampler1DArray
GL_SAMPLER_2D_ARRAY	sampler2DArray
GL_SAMPLER_1D_ARRAY_SHADOW	sampler1DArrayShadow
GL_SAMPLER_2D_ARRAY_SHADOW	sampler2DArrayShadow
GL_SAMPLER_2D_MULTISAMPLE	sampler2DMS
GL_SAMPLER_2D_MULTISAMPLE_ARRAY	sampler2DMSArray
GL_SAMPLER_CUBE_SHADOW	samplerCubeShadow
GL_SAMPLER_BUFFER	samplerBuffer
GL_SAMPLER_2D_RECT	sampler2DRect
GL_SAMPLER_2D_RECT_SHADOW	sampler2DRectShadow
GL_INT_SAMPLER_1D	isampler1D
GL_INT_SAMPLER_2D	isampler2D
GL_INT_SAMPLER_3D	isampler3D
GL_INT_SAMPLER_CUBE	isamplerCube
GL_INT_SAMPLER_1D_ARRAY	isampler1DArray
GL_INT_SAMPLER_2D_ARRAY	isampler2DArray
GL_INT_SAMPLER_2D_MULTISAMPLE	isampler2DMS
GL_INT_SAMPLER_2D_MULTISAMPLE_ARRAY	isampler2DMSArray
GL_INT_SAMPLER_BUFFER	isamplerBuffer
GL_INT_SAMPLER_2D_RECT	isampler2DRect
GL_UNSIGNED_INT_SAMPLER_1D	usampler1D

续表

返回的符号常量	着色器统一变量类型
GL_UNSIGNED_INT_SAMPLER_2D	usampler2D
GL_UNSIGNED_INT_SAMPLER_3D	usampler3D
GL_UNSIGNED_INT_SAMPLER_CUBE	usamplerCube
GL_UNSIGNED_INT_SAMPLER_1D_ARRAY	usampler2DArray
GL_UNSIGNED_INT_SAMPLER_2D_ARRAY	usampler2DArray
GL_UNSIGNED_INT_SAMPLER_2D_MULTISAMPLE	usampler2DMS
GL_UNSIGNED_INT_SAMPLER_2D_MULTISAMPLE_ARRAY	usampler2DMSArray
GL_UNSIGNED_INT_SAMPLER_BUFFER	usamplerBuffer
GL_UNSIGNED_INT_SAMPLER_2D_RECT	usampler2DRect

统一变量的大小将会返回到 size。除数组以外的统一变量的大小将为 1。就像前面描述的一样，结构和结构数组将被进行简化，这样返回的每一个名称都将是原来列表中的数据类型。如果这种简化的结果是一个数组，那么返回的大小将如统一数组描述的一样。否则，返回的大小将为 1。

活动统一变量列表可以包含在内建统一变量（这些统一变量以“gl\_”前缀开始），同样也可以包含在用户定义统一变量名称中。

这个函数将返回它所能返回的尽可能多的关于指定活动统一变量的信息。

如果没有可用的信息，那么 length 将为 0，而 name 将为空字符串。在一个链接操作失败后调用这个函数就可能会出现这种情况。如果出现错误，那么返回值 length、size、type 和 name 将为未修改的。

**错误**

如果 program 不是由 OpenGL 产生的值，则产生 GL\_INVALID\_VALUE 错误。

如果 program 不是一个程序对象，则产生 GL\_INVALID\_OPERATION 错误。

如果 index 大于或等于 program 中活动统一变量的数量，则产生 GL\_INVALID\_VALUE 错误。

如果 bufSize 小于 0，则产生 GL\_INVALID\_VALUE 错误。

**相关 Get 函数**

glGet，自变量为 GL\_MAX\_VERTEX\_UNIFORM\_COMPONENTS、GL\_MAX\_GEOMETRY\_UNIFORM\_COMPONENTS、GL\_MAX\_FRAGMENT\_UNIFORM\_COMPONENTS 或 GL\_MAX\_COMBINED\_UNIFORM\_COMPONENTS。

glGetProgram，自变量为 GL\_ACTIVE\_UNIFORMS 或 GL\_ACTIVE\_UNIFORM\_MAX\_LENGTH。

glIsProgram

**另外查看**

glGetUniform, glGetUniformLocation, glLinkProgram, glUniform, glUseProgram

**版权**

Copyright © 2003–2005 3Dlabs Inc. Ltd. Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glGetActiveUniformBlock

查询关于一个活动统一块的信息。

**C 规范**

```
void glGetActiveUniformBlockiv(GLuint program,
                               GLuint uniformBlockIndex,
                               GLenum pname,
                               GLint params);
```

**参数**

program

指定一个包含统一块的程序的名称。

`uniformBlockIndex`

指定 `program` 中统一块的索引。

`pname`

指定将要进行查询的参数的名称。

`params`

指定接收查询结果的变量的地址。

**描述**

`glGetActiveUniformBlockiv` 检索关于 `program` 中一个活动统一块的信息。

`program` 必须是一个程序对象的名称, 过去必须已经为这个程序对象调用了 `glLinkProgram` 命令, 虽然并不要求 `glLinkProgram` 必须成功。这个连接可能会失败, 因为活动统一变量的数量超出了限制。

`uniformBlockIndex` 是 `program` 的一个活动统一块索引, 并且必须小于 `GL_ACTIVE_UNIFORM_BLOCKS` 的值。

一旦成功, 这个 (这些) 由 `pname` 指定的统一块参数将会返回到 `params`。如果出现错误, 那么就不会向 `params` 写入任何东西。

如果 `pname` 为 `GL_UNIFORM_BLOCK_BINDING`, 那么将会返回最后由 `uniformBlockIndex` 为 `program` 指定的统一块选择的统一缓冲区绑定点的索引。如果以前没有指定任何统一块, 那么将会返回 0。

如果 `pname` 为 `GL_UNIFORM_BLOCK_DATA_SIZE`, 那么这个与实现相关的最小总缓冲区对象大小 (以基本机器单元为单位) 则要求返回由 `uniformBlockIndex` 指定的统一块中所有活动的统一变量。我们既不保证也不希望一个给定的实现会像在一个缓冲区对象中一样对统一值进行紧密的包装。这里的例外情况是 `std140` 统一块布局, 它确保了指定的包装行为, 并且不要求应用程序查询偏置和步长。在这种情况下, 仍然会需要大小的最小值, 即使它已经只根据统一块声明提前确定了。

如果 `pname` 是 `GL_UNIFORM_BLOCK_NAME_LENGTH`, 那么将会返回由 `uniformBlockIndex` 指定的统一块名称的总长度 (包含空终止符)。

如果 `pname` 是 `GL_UNIFORM_BLOCK_ACTIVE_UNIFORMS`, 那么将会返回由 `uniformBlockIndex` 指定的统一块中的活动统一变量数量。

如果 `pname` 是 `GL_UNIFORM_BLOCK_ACTIVE_UNIFORM_INDICES`, 那么将会返回由 `uniformBlockIndex` 指定的统一块的活动统一变量索引的一个列表。将被写入到 `params` 的元素的数量是 `uniformBlockIndex` 的 `GL_UNIFORM_BLOCK_ACTIVE_UNIFORMS` 值。

如果 `pname` 是 `GL_UNIFORM_BLOCK_REFERENCED_BY_VERTEX_SHADER`、`GL_UNIFORM_BLOCK_REFERENCED_BY_GEOMETRY_SHADER` 或 `GL_UNIFORM_BLOCK_REFERENCED_BY_FRAGMENT_SHADER`, 那么将返回一个布尔值, 分别指示由 `uniformBlockIndex` 指定的统一块是否被程序的顶点、几何图形或片段编程阶段引用。

**错误**

如果 `uniformBlockIndex` 大于或等于 `GL_ACTIVE_UNIFORM_BLOCKS` 的值, 或者不是 `program` 中一个活动统一块的索引, 则产生 `GL_INVALID_VALUE` 错误。

如果 `pname` 不是一个可接受的标记, 则产生 `GL_INVALID_ENUM` 错误。

如果 `program` 不是过去为其调用过 `glLinkProgram` 的一个程序对象的名称, 则产生 `GL_INVALID_OPERATION` 错误。

**注意**

`glGetActiveUniformBlockiv` 只在 3.1 或更高版本的 GL 中可用。

**另外查看**

`glGetActiveUniformBlockName`, `glGetUniformBlockIndex`, `glLinkProgram`

**版权**

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。 <http://opencontent.org/openpub/>。

## `glGetActiveUniformBlockName`

检索一个活动统一块的名称。

**C 规范**

```
void glGetActiveUniformBlockName(GLuint program,
```

```
GLuint uniformBlockIndex,
GLsizei bufSize,
GLsizei *length,
GLchar *uniformBlockName);
```

#### 参数

program

指定一个包含统一块的程序的名称。

uniformBlockIndex

指定 program 中统一块的索引。

bufSize

指定由 uniformBlockName 进行寻址的缓冲区的大小。

length

指定接收写入到 uniformBlockName 的字符数的变量的地址。

uniformBlockName

指定一个位于 uniformBlockIndex 的接收统一块名称的字符数组的地址。

#### 描述

glGetActiveUniformBlockName 检索 program 中位于 uniformBlockIndex 的一个活动统一块的名称。

program 必须是一个程序对象的名称, 过去必须已经为这个程序对象调用了 glLinkProgram 命令, 虽然并不要求 glLinkProgram 必须成功。这个连接可能会失败, 因为活动统一变量的数量超出了限制。

uniformBlockIndex 是 program 的一个活动统一块索引, 并且必须小于 GL\_ACTIVE\_UNIFORM\_BLOCKS 的值。

一旦成功, 由 uniformBlockIndex 指定的统一块名称就会返回到 uniformBlockName。它的名称以 null 为结尾。字符的实际数量被写入到 uniformBlockName, 但不包括空终结符 null, 它被返回到 length。如果 length 为 NULL, 则不会返回长度。

bufSize 包含字符的最大数量 (包括空终结符 null), 它将被写入到 uniformBlockName。

如果出现错误, 那么就不会向 uniformBlockName 或 length 写入任何东西。

#### 错误

如果 program 不是过去为其调用过 glLinkProgram 的一个程序对象的名称, 则产生 GL\_INVALID\_OPERATION 错误。

如果 uniformBlockIndex 大于或等于 GL\_ACTIVE\_UNIFORM\_BLOCKS 的值, 或者不是 program 中一个活动统一块的索引, 则产生 GL\_INVALID\_VALUE 错误。

#### 注意

glGetActiveUniformBlockName 只在 3.1 或更高版本的 GL 中可用。

#### 另外查看

glGetActiveUniformBlock, glGetUniformBlockIndex

#### 版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

### glGetActiveUniformName

查询一个活动统一变量的名称。

#### C 规范

```
void glGetActiveUniformName(GLuint program,
GLuint uniformIndex,
GLsizei bufSize,
GLsizei *length,
GLchar *uniformName);
```

#### 参数

program

指定包含活动统一索引 uniformIndex 的程序。

uniformIndex

指定其名称将要被查询的活动统一变量的索引。

bufSize

指定在 uniformName 中指定地址的缓冲区的大小 (以 GLchar 为单位)。

length

指定变量的地址, 这个变量将要接收已经或将要写入到由 uniformName 进行寻址的缓冲区中的字符数。

uniformName

指定一个缓冲区的地址, GL 会将 program 中 uniformIndex 位置的活动统一变量的名称写入到这个缓冲区。

描述

glGetActiveUniformName 返回 program 中位于 uniformIndex 的一个活动统一变量的名称。如果 uniformName 不为 NULL, 那么最多会有 bufSize 个字符(包含空终结符)被写入到由 uniformName 指定地址的数组中。如果 length 不为 NULL, 那么已经被写入到(或应该已经被写入到)uniformName 的字符数(不包含空终结符)将被放置到由 length 指定地址的变量中。如果 length 为 NULL, 则不会返回长度。program 中最长的统一变量名称的长度是由 GL\_ACTIVE\_UNIFORM\_MAX\_LENGTH 的值给出的, 它可以使用 glGetProgram 查询。

如果 glGetActiveUniformName 不成功, 那么就不会向 length 或 uniformName 写入任何东西。

Program 必须是前面为其发出过 glLinkProgram 命令的一个程序对象的名称。program 不必已经成功进行了连接。这个连接可能会失败, 因为活动统一变量的数量超出了限制。

uniformIndex 必须为 program 程序的一个活动统一索引, 范围在 0 到 GL\_ACTIVE\_UNIFORMS-1 之间, 可以通过 glGetProgram 查询 GL\_ACTIVE\_UNIFORMS 的值。

错误

如果 uniformIndex 大于或等于 GL\_ACTIVE\_UNIFORMS 的值, 则产生 GL\_INVALID\_VALUE 错误。

如果 bufSize 为负值, 则产生 GL\_INVALID\_VALUE 错误。

如果 program 不是一个已经为其发出 glLinkProgram 命令的程序对象的名称, 则产生 GL\_INVALID\_VALUE 错误。

另外查看

glGetActiveUniform, glGetUniformLocation, glGetProgram, glLinkProgram

版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glGetAttachedShaders

返回绑定到一个程序对象的着色器对象的句柄。

**C 规范**

```
void glGetAttachedShaders(GLuint program,
                           GLsizei maxCount,
                           GLsizei * count,
                           GLuint * shaders);
```

参数

program

指定将要被查询的程序对象。

maxCount

指定用于存储返回的对象名称的数组的大小。

count

返回在 objects 中实际返回的名称数量。

shaders

指定一个用于返回绑定着色器对象的名称的数组。

描述

glGetAttachedShaders 返回绑定到 program 的着色器对象的名称。绑定到 program 的着色器对象的名称将被返回到 shaders 中。写入 shaders 的着色器名称的实际数量返回到 count 中。如果没有着色器被绑定到 program, 那么 count 将被设为 0。可以返回到 shaders 的着色器名称的最大数量由 maxCount 指定。

如果实际返回的名称数量不是必需的(例如,它刚刚通过调用 `glGetProgram` 而获得),可能会传递一个 `NULL` 值来充数。如果没有着色器对象被绑定到 `program`, `count` 中将返回一个 0 值。绑定的实际数量可以通过调用值为 `GL_ATTACHED_SHADERS` 的 `glGetProgram` 来获取。

#### 错误

如果 `program` 不是由 OpenGL 产生的值,则产生 `GL_INVALID_VALUE` 错误。

如果 `program` 不是一个程序对象,则产生 `GL_INVALID_OPERATION` 错误。

如果 `maxCount` 小于 0,则产生 `GL_INVALID_VALUE` 错误。

#### 相关 Get 函数

`glGetProgram`, 其自变量为 `GL_ATTACHED_SHADERS`。

`glIsProgram`

#### 另外查看

`glAttachShader`, `glDetachShader`

#### 版权

Copyright © 2003–2005 3Dlabs Inc. Ltd. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glGetAttribLocation

返回一个属性变量的位置。

#### C 规范

```
GLint glGetAttribLocation(GLuint program,
                          const GLchar * name);
```

#### 参数

`program`

指定将要被查询的程序对象。

`name`

指向一个包含其位置要被查询的属性变量名称的空终止字符串。

#### 描述

`glGetAttribLocation` 查询以前连接的由 `program` 指定的程序对象,以确定由 `name` 指定的属性变量,并返回绑定到这个属性变量的通用顶点属性的索引。如果 `name` 是一个矩阵属性变量,则返回矩阵第一列的索引。如果指定的属性变量不是指定程序对象中的一个活动属性,或者如果 `name` 以保留的“gl”前缀开始,那么将返回值-1。

可以在任何时间通过调用 `glBindAttribLocation` 来指定一个属性变量名称和一个通用属性索引之间的关联。属性绑定在调用 `glLinkProgram` 之后才会生效。在一个程序对象被成功连接之后,属性变量索引值将保持不变,直到下次连接命令发生。如果一次连接成功,那么属性值只能在这次连接之后查询。`glGetAttribLocation` 返回上次为指定程序对象调用 `glLinkProgram` 时实际生效的绑定。从上一次连接操作时被指定的参数绑定不会被 `glGetAttribLocation` 返回。

#### 错误

如果 `program` 不是由 OpenGL 产生的值,则产生 `GL_INVALID_OPERATION` 错误。

如果 `program` 不是一个程序对象,则产生 `GL_INVALID_OPERATION` 错误。

如果 `program` 没有成功进行连接,则产生 `GL_INVALID_OPERATION` 错误。

#### 相关 Get 函数

`glGetActiveAttrib`, 其自变量为 `program` 和一个活动属性的索引。

`glIsProgram`

#### 另外查看

`glBindAttribLocation`, `glLinkProgram`, `glVertexAttrib`, `glVertexAttribPointer`

#### 版权

Copyright © 2003–2005 3Dlabs Inc. Ltd. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glGetBufferParameteriv

返回一个缓冲区对象的参数。

### C 规范

```
void glGetBufferParameteriv(GLenum target,
                             GLenum value,
                             GLint * data);
```

### 参数

target

指定目标缓冲区对象。符号常量必须是 GL\_ARRAY\_BUFFER、GL\_COPY\_READ\_BUFFER、GL\_COPY\_WRITE\_BUFFER、GL\_ELEMENT\_ARRAY\_BUFFER、GL\_PIXEL\_PACK\_BUFFER、GL\_PIXEL\_UNPACK\_BUFFER、GL\_TEXTURE\_BUFFER、GL\_TRANSFORM\_FEEDBACK\_BUFFER 或 GL\_UNIFORM\_BUFFER。

value

指定缓冲区对象参数的符号明。可接受的值为 GL\_BUFFER\_ACCESS、GL\_BUFFER\_MAPPED、GL\_BUFFER\_SIZE 或 GL\_BUFFER\_USAGE。

data

返回所需参数。

### 描述

glGetBufferParameteriv 将一个由 tarGet 指定的缓冲区对象的一个选定参数返回到 data。

value 指定一个具体的缓冲区对象参数，如下所示。

GL\_BUFFER\_ACCESS

params 返回在进行缓冲区对象映射时设置的访问策略。初始值为 GL\_READ\_WRITE。

GL\_BUFFER\_MAPPED

params 返回一个指示缓冲区对象当前是否被映射的标记。初始值为 GL\_FALSE。

GL\_BUFFER\_SIZE

params 返回缓冲区对象的大小，以字节为单位。初始值为 0。

GL\_BUFFER\_USAGE

params 返回缓冲区对象的使用模式。初始值为 GL\_STATIC\_DRAW。

### 注意

如果有错误产生，那么 data 的内容不会进行任何修改。

### 错误

如果 tarGet 或 value 不是一个可接受的值，则产生 GL\_INVALID\_ENUM 错误。

如果保留的缓冲区对象名称 0 被绑定到 tarGet，则产生 GL\_INVALID\_OPERATION 错误。

### 另外查看

glBindBuffer, glBufferData, glMapBuffer, glUnmapBuffer

### 版权

Copyright © 2005 Addison-Wesley。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glGetBufferPointerv

返回一个指向被映射的缓冲区对象数据存储的指针。

### C 规范

```
void glGetBufferPointerv(GLenum target,
                         GLenum pname,
                         GLvoid ** params);
```

### 参数

target

指定目标缓冲区对象。符号常量必须是 GL\_ARRAY\_BUFFER、GL\_COPY\_READ\_BUFFER、GL\_COPY\_WRITE\_

BUFFER、GL\_ELEMENT\_ARRAY\_BUFFER、GL\_PIXEL\_PACK\_BUFFER、GL\_PIXEL\_UNPACK\_BUFFER、GL\_TEXTURE\_BUFFER、GL\_TRANSFORM\_FEEDBACK\_BUFFER 或 GL\_UNIFORM\_BUFFER。

pname

指定要返回的指针。符号常量必须为 GL\_BUFFER\_MAP\_POINTER。

params

返回由 pname 指定的指针值。

描述

glGetBufferPointerv 返回指针信息。pname 是一个指示将要返回的指针的符号常量必须为 GL\_BUFFER\_MAP\_POINTER, 即缓冲区对象的数据存储映射到的指针。如果数据存储当前没有被映射, 则返回 NULL。params 是一个指向放置返回指针值的位置的指针。

注意

如果有错误产生, 那么 params 的内容不会进行任何修改。

指针的初始值为 NULL。

错误

如果 tarGet 或 pname 不是一个可接受的值, 则产生 GL\_INVALID\_ENUM 错误。

如果保留的缓冲区对象名称 0 被绑定到 tarGet, 则产生 GL\_INVALID\_OPERATION 错误。

另外查看

glBindBuffer, glMapBuffer

版权

Copyright © 2005 Addison-Wesley。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glGetBufferSubData

返回一个缓冲区对象的数据存储的子集。

C 规范

```
void glGetBufferSubData(GLenum target,
                        GLintptr offset,
                        GLsizeiptr size,
                        GLvoid * data);
```

参数

target

指定目标缓冲区对象。符号常量必须是 GL\_ARRAY\_BUFFER、GL\_COPY\_READ\_BUFFER、GL\_COPY\_WRITE\_BUFFER、GL\_ELEMENT\_ARRAY\_BUFFER、GL\_PIXEL\_PACK\_BUFFER、GL\_PIXEL\_UNPACK\_BUFFER、GL\_TEXTURE\_BUFFER、GL\_TRANSFORM\_FEEDBACK\_BUFFER 或 GL\_UNIFORM\_BUFFER。

offset

指定缓冲区对象的数据存储 (数据将从中返回) 的偏移, 以字节为单位。

size

指定返回的数据存储区域的大小, 以字节为单位。

data

指定一个指向缓冲区对象数据返回到的区域的指针。

描述

glGetBufferSubData 从当前捆绑到 tarGet 的缓冲区对象中返回一部分或全部数据存储。起始于字节偏移 offset 和扩展 size 字节的数据会从数据存储复制到由 data 指向的内存。如果缓冲区对象当前被映射, 或者 offset 和 size 一起定义一个超出缓冲区对象数据存储边界的范围, 则会抛出一个错误。

注意

如果有错误产生, 那么 data 的内容不会进行任何修改。

错误

如果 tarGet 不是 GL\_ARRAY\_BUFFER、GL\_ELEMENT\_ARRAY\_BUFFER、GL\_PIXEL\_PACK\_BUFFER 或

GL\_PIXEL\_UNPACK\_BUFFER, 则产生 GL\_INVALID\_ENUM 错误。

如果 offset 或者 size 为负, 或者如果 offset 和 size 一起定义一个超出缓冲区对象数据存储范围的内存区域, 则产生 GL\_INVALID\_VALUE 错误。

如果保留的缓冲区对象名称 0 被绑定到 tarGet, 则产生 GL\_INVALID\_OPERATION 错误。

如果被查询的缓冲区对象被映射, 则产生 GL\_INVALID\_OPERATION 错误。

另外查看

glBindBuffer, glBufferData, glBufferSubData, glMapBuffer, glUnmapBuffer

版权

Copyright © 2005 Addison-Wesley. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glGetCompressedTexImage

返回一个压缩纹理图像

**C 规范**

```
void glGetCompressedTexImage(GLenum target,
                             GLint lod,
                             GLvoid * img);
```

**参数**

target

target 指定将要获得哪个纹理。GL\_TEXTURE\_1D、GL\_TEXTURE\_2D、GL\_TEXTURE\_3D、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Z 和 GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Z 都可以被接受。

lod

指定期望图像的层次细节数量。Level 0 是基本图像层次。

Level n 是第 n 级 Mip 贴图缩略图。

img

返回这个压缩纹理图像

**描述**

glGetCompressedTexImage 将与 target 和 lod 关联的压缩纹理图像返回到 img。img 应该是一个 GL\_TEXTURE\_COMPRESSED\_IMAGE\_SIZE 字节的数组。target 指定所期望的纹理图像是否是由 glTexImage1D (GL\_TEXTURE\_1D)、glTexImage2D (GL\_TEXTURE\_2D 或任何 GL\_TEXTURE\_CUBE\_MAP\_\*) 或 glTexImage3D (GL\_TEXTURE\_3D) 指定的。lod 指定期望图像的层次细节数量。

如果在纹理图像被指定的情况下, 所需的非 0 缓冲区对象被绑定到 GL\_PIXEL\_PACK\_BUFFER 目标 (参见 glBindBuffer), img 将被看作缓冲区对象数据存储的一个字节偏移。

为了将错误减到最少, 首先要核实纹理是通过调用自变量为 GL\_TEXTURE\_COMPRESSED 的 glGetTexLevelParameter 来进行压缩的。如果纹理被压缩, 那么调用自变量为 GL\_TEXTURE\_COMPRESSED\_IMAGE\_SIZE 的 glGetTexLevelParameter 来确定存储压缩纹理所需的内存总量。最后, 通过调用自变量为 GL\_TEXTURE\_INTERNAL\_FORMAT 的 glGetTexLevelParameter 来检索纹理的内部格式。要存储纹理以供后面使用, 将内部格式和大小与检索的纹理图像相关联。这些数据能够被各自用于载入 tarGet 纹理的的纹理或子纹理输入程序所使用。

**错误**

如果 lod i 小于 0 或大于实现所能允许的 LOD 最大值, 则产生 GL\_INVALID\_VALUE 错误。

如果 glGetCompressedTexImage 被用来检索一个未压缩内部格式的纹理, 则产生 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_PACK\_BUFFER 目标, 并且缓冲区对象的数据存储当前被映射, 则产生 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_PACK\_BUFFER 目标, 并且数据将要被打包到缓冲区对象, 以致内存写入请求将超出数据存储大小, 则产生 GL\_INVALID\_OPERATION 错误。

**相关 Get 函数**

glGetTexLevelParameter, 其自变量为 GL\_TEXTURE\_COMPRESSED。

glGetTexLevelParameter, 其自变量为 GL\_TEXTURE\_COMPRESSED\_IMAGE\_SIZE。

glGetTexLevelParameter, 其自变量为 GL\_TEXTURE\_INTERNAL\_FORMAT。

glGet, 其自变量为 GL\_PIXEL\_PACK\_BUFFER\_BINDING。

#### 另外查看

glActiveTexture, glCompressedTexImage1D, glCompressedTexImage2D, glCompressedTexImage3D, glCompressedTexSubImage1D, glCompressedTexSubImage2D, glCompressedTexSubImage3D, glReadPixels, glTexImage1D, glTexImage2D, glTexImage3D, glTexParameter, glTexSubImage1D, glTexSubImage2D, glTexSubImage3D

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glGetError

返回错误信息。

#### C 规范

```
GLenum glGetError(void);
```

#### 描述

glGetError 返回错误标记的值。每个可检出的错误都会被分配一个数字代码和符号名。在出现错误时, 错误标记将被设置为适当的错误代码值。在 glGetError 被调用、错误代码被返回并且标记被重设为 GL\_NO\_ERROR 之前, 不会有其他错误被记录。如果一次 glGetError 调用返回 GL\_NO\_ERROR, 那就说明从上一次调用 glGetError 或 GL 初始化开始没有出现可检出的错误。

考虑到分布式实现, 还需要几个错误标记。如果任何单个错误标记记录了一个错误, 那么在 glGetError 被调用时这个标记的值会被返回, 而这个标记则被重设为 GL\_NO\_ERROR。如果一个以上的标记都记录了一个错误, 那么 glGetError 返回, 并清除任意一个错误标记值。这样, 如果所有的错误标记都要被重置, 那么 glGetError 在一个循环中将总是被调用, 直到它返回 GL\_NO\_ERROR。

在初始情况下, 所有错误标记都被设置为 GL\_NO\_ERROR。

下面的错误是当前被定义的。

GL\_NO\_ERROR

没有记录任何错误。这个符号常量的值保证为 0。

GL\_INVALID\_ENUM

一个列举自变量被指定了一个不可接受的值。错误的命令将被忽略, 并且除了设置错误标记之外, 不产生任何其它副作用。

GL\_INVALID\_VALUE

一个数字自变量超出了允许范围。错误的命令将被忽略, 并且除了设置错误标记之外, 不产生任何其它副作用。

GL\_INVALID\_OPERATION

指定的操作在当前状态下不允许。错误的命令将被忽略, 并且除了设置错误标记之外, 不产生任何其它副作用。

GL\_INVALID\_FRAMEBUFFER\_OPERATION

帧缓冲区对象不完整。错误的命令将被忽略, 并且除了设置错误标记之外, 不产生任何其它副作用。

GL\_OUT\_OF\_MEMORY

没有足够的剩余内存用来执行此命令。在这个错误被记录之后, 除了这个错误标记的状态之外, GL 的状态为未定义的。

当一个错误标记被设置时, 一个 GL 操作的结果只有在 GL\_OUT\_OF\_MEMORY 出现时才是未定义的。在其他所有情况下, 产生错误的命令都将被忽略, 并且不会对 GL 状态或帧缓冲区内容产生任何影响。如果产生的命令返回一个值, 那么它将返回 0。如果 glGetError 本身生成了一个错误, 那么它返回 0。

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glGetFragDataIndex

查询颜色索引到用户定义的 varying 输出变量的绑定。

### C 规范

```
GLint glGetFragDataIndex(GLuint program,
                        const char * name);
```

### 参数

program

包含将要查询绑定的 varying 输出变量的程序的名称。

name

用户定义的将要查询索引的 varying 输出变量的名称。

### 描述

glGetFragDataIndex 返回当程序对象 program 上一次连接时变量 name 被绑定到的片段颜色的索引。如果 name 不是一个 program 的 varying 输出变量, 或者如果出现错误, 则将会返回-1。

### 注意

glGetFragDataIndex 只在 3.3 或更高版本的 GL 中可用。

### 错误

如果 program 不是一个程序对象的名称, 则产生 GL\_INVALID\_OPERATION 错误。

### 另外查看

glCreateProgram, glBindFragDataLocation, glBindFragDataLocationIndexed, glGetFragDataLocation

### 版权

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glGetFragDataLocation

查询从颜色号到用户定义的 varying 输出变量的数量。

### C 规范

```
GLint glGetFragDataLocation(GLuint program,
                        const char * name);
```

### 参数

program

包含将要查询绑定的 varying 输出变量的程序的名称。

name

用户定义的将要查询绑定的 varying 输出变量的名称。

### 描述

glGetFragDataLocation 为 program 程序检索从已分配颜色号到用户定义 varying 输出变量名的绑定。Program 必须是已经进行过连接的。name 必须是一个以 null 为结尾的字符串。如果 name 不是 program 中的一个活动的用户定义 varying 输出片段着色器变量, 则将会返回-1。

### 错误

如果 program 不是一个程序对象的名称, 则产生 GL\_INVALID\_OPERATION 错误。

### 另外查看

glCreateProgram, glBindFragDataLocation

### 版权

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glGetFramebufferAttachmentParameteriv

获取关于一个绑定的帧缓冲区对象的信息。

### C 规范

```
void glGetFramebufferAttachmentParameteriv(GLenum target,
                                             GLenum attachment,
                                             GLenum pname,
                                             GLint *params);
```

### 参数

target

指定查询操作的目标。

attachment

指定 target 中的绑定。

pname

指定将要进行查询的绑定的参数。

params

指定接收 attachment 的 pname 值的变量的地址。

### 描述

glGetFramebufferAttachmentParameteriv 返回关于一个绑定的帧缓冲区对象的绑定信息。Target 制定帧缓冲区绑定点, 必须为 GL\_DRAW\_FRAMEBUFFER、GL\_READ\_FRAMEBUFFER 或 GL\_FRAMEBUFFER。GL\_FRAMEBUFFER 等价于 GL\_DRAW\_FRAMEBUFFER。

如果默认帧缓冲区被绑定到 target, 那么 attachment 必须为 GL\_FRONT\_LEFT、GL\_FRONT\_RIGHT、GL\_BACK\_LEFT 或 GL\_BACK\_RIGHT 中的一个来指定一个颜色缓冲区, 或者 GL\_DEPTH 用来指定深度缓冲区, 或者是 GL\_STENCIL 来指定模板缓冲区。

如果一个帧缓冲区被绑定, 那么 attachment 必须为 GL\_COLOR\_ATTACHMENTi、GL\_DEPTH\_ATTACHMENT、GL\_STENCIL\_ATTACHMENT、GL\_DEPTH\_STENCIL\_ATTACHMENT 或 GL\_DEPTH\_STENCIL\_ATTACHMENT。

GL\_COLOR\_ATTACHMENTi 中的 i 必须是在从 0 到 GL\_MAX\_COLOR\_ATTACHMENTS-1 的范围之内。

如果 attachment 为 GL\_DEPTH\_STENCIL\_ATTACHMENT, 并且不同的对象被绑定到 target 的深度和模板绑定点, 那么查询将会失败。如果这两个绑定点都绑定了同一个对象, 那么将会返回关于这个对象的信息。

在成功从 glGetFramebufferAttachmentParameteriv 返回的情况下, 如果 pname 为 GL\_FRAMEBUFFER\_ATTACHMENT\_OBJECT\_TYPE, 那么 params 将包含 GL\_NONE、GL\_FRAMEBUFFER\_DEFAULT、GL\_TEXTURE 或 GL\_RENDERBUFFER 中的一个, 指定包含这些绑定图像的对象类型。pname 能够接受的其他值取决于对象的类型, 如下所示。

如果 GL\_FRAMEBUFFER\_ATTACHMENT\_OBJECT\_TYPE 的值为 GL\_NONE, 那么不会有任何帧缓冲区被绑定到 target。在这种情况下查询 pname, GL\_FRAMEBUFFER\_ATTACHMENT\_OBJECT\_NAME 将会返回 0, 并且所有其他查询都回生成一个错误。

如果 GL\_FRAMEBUFFER\_ATTACHMENT\_OBJECT\_TYPE 的值不为 GL\_NONE, 那么这些查询将会应用到所有其他帧缓冲区类型。

如果 pname 为 GL\_FRAMEBUFFER\_ATTACHMENT\_RED\_SIZE、GL\_FRAMEBUFFER\_ATTACHMENT\_GREEN\_SIZE、GL\_FRAMEBUFFER\_ATTACHMENT\_BLUE\_SIZE、GL\_FRAMEBUFFER\_ATTACHMENT\_ALPHA\_SIZE、GL\_FRAMEBUFFER\_ATTACHMENT\_DEPTH\_SIZE 或 GL\_FRAMEBUFFER\_ATTACHMENT\_STENCIL\_SIZE, 那么 params 将会包含指定绑定的相应红色、绿色、蓝色、alpha、深度或模板分量的位数。如果请求的分量在 attachment 中不存在, 那么将返回 0。

如果 pname 为 GL\_FRAMEBUFFER\_ATTACHMENT\_COMPONENT\_TYPE, 那么 pname 将会包含指定绑定的分量的格式, 即 GL\_FLOAT、GL\_INT、GL\_UNSIGNED\_INT、GL\_SIGNED\_NORMALIZED 或 GL\_UNSIGNED\_NORMALIZED, 分别对应浮点、有符号整数、无符号整数、有符号标准化定点和无符号标准化定点分量。只有颜色缓冲区才会有整数分量。

如果 pname 为 GL\_FRAMEBUFFER\_ATTACHMENT\_COLOR\_ENCODING, 那么 param 将会包含指定绑定的分量的编码, 为 GL\_LINEAR 或 GL\_SRGB, 分别对应线性或 sRGB 编码分量。只有颜色缓冲区分量可以是 sRGB 编码

的; 这些分量将会按照 4.1.7 和 4.1.8 部分中所描述的方式进行处理。

对于默认帧缓冲区来说, 颜色编码由实现决定。对于帧缓冲区对象来说, 如果一个颜色绑定的内部格式是某种颜色可渲染得 SRGB 格式, 那么分量将是 sRGB 编码的。

如果 GL\_FRAMEBUFFER\_ATTACHMENT\_OBJECT\_TYPE 的值为 GL\_RENDERBUFFER, 那么:

如果 pname 为 GL\_FRAMEBUFFER\_ATTACHMENT\_OBJECT\_NAME, 那么 pname 将会包含那个包含绑定图像的渲染缓冲区的名称。

如果 GL\_FRAMEBUFFER\_ATTACHMENT\_OBJECT\_TYPE 的值为 GL\_TEXTURE, 那么:

如果 pname 为 GL\_FRAMEBUFFER\_ATTACHMENT\_OBJECT\_NAME, 那么 pname 将会包含那个包含绑定图像的纹理对象的名称。

如果 pname 为 GL\_FRAMEBUFFER\_ATTACHMENT\_TEXTURE\_LEVEL, 那么 pname 将会包含那个包含绑定图像的纹理对象的 Mip 贴图层次。

如果 pname 为 GL\_FRAMEBUFFER\_ATTACHMENT\_TEXTURE\_CUBE\_MAP\_FACE, 并且名为 GL\_FRAMEBUFFER\_ATTACHMENT\_OBJECT\_NAME 的纹理对象是一个立方体贴图纹理, 那么 params 将会包含那个包含绑定图像的立方体贴图纹理对象的立方体贴图表面。否则 params 将会包含 0 值。

如果 pname 为 GL\_FRAMEBUFFER\_ATTACHMENT\_TEXTURE\_LAYER, 并且名为 GL\_FRAMEBUFFER\_ATTACHMENT\_OBJECT\_NAME 的纹理对象是一个三维纹理或二维数组纹理, 那么 params 将会包含那个包含绑定图像的纹理层的编号。否则 params 将会包含 0 值。

如果 pname 为 GL\_FRAMEBUFFER\_ATTACHMENT\_LAYERED, 那么当一个三维纹理、立方体贴图纹理或者一维或二维数组纹理的整个一个层次被绑定时, pname 将包含 GL\_TRUE。否则 params 将会包含 GL\_FALSE。

任何帧缓冲区类型与上面没有提到的任何 pname 的组合, 都会产生一个错误。

**错误**

如果 tarGet 不是一个可接受的标记, 则产生 GL\_INVALID\_ENUM 错误。

如果 pname 对于 GL\_FRAMEBUFFER\_ATTACHMENT\_OBJECT\_TYPE 的值来说不是有效值, 则产生 GL\_INVALID\_ENUM 错误。

如果 attachment 对于 tarGet 来说不是一个可接受的值, 则产生 GL\_INVALID\_OPERATION 错误。

如果 attachment 为 GL\_DEPTH\_STENCIL\_ATTACHMENT, 并且不同的对象被绑定到 tarGet 的深度和模板绑定点, 则产生 GL\_INVALID\_OPERATION 错误。

如果 GL\_FRAMEBUFFER\_ATTACHMENT\_OBJECT\_TYPE 的值为 GL\_NONE, 并且 pname 不是 GL\_FRAMEBUFFER\_ATTACHMENT\_OBJECT\_NAME, 则产生 GL\_INVALID\_OPERATION 错误。

**另外查看**

glGenFramebuffers, glBindFramebuffer

**版权**

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glGetMultisamplefv

检索采样器的位置。

**C 规范**

```
void glGetMultisamplefv(GLenum pname,
                        GLuint index,
                        GLfloat *val);
```

**参数**

pname

指定采样参数名称。pname 必须为 GL\_SAMPLE\_POSITION。

index

指定其位置将要被查询的样本。

val

指定接收样本位置的数组的地址。

**描述**

`glGetMultisamplefv` 查询一个给定样本的位置。`pname` 代表要检索的样本参数，并且必须为 `GL_SAMPLE_POSITION`。`index` 与应该返回其位置的样本相对应。样本位置将作为两个浮点值返回到 `val[0]` 和 `val[1]` 中，它们的值都在 0 和 1 之间，分别对应这个样本的 GL 像素空间的 `x` 和 `y` 位置。(0.5, 0.5) 对应于像素中心。`index` 必须在 0 和 `GL_SAMPLES - 1` 的值之间。

如果多重采样模式没有固定的样本位置，那么返回的值可能只会反映某些像素中样本的位置。

#### 错误

如果 `pname` 不是一个 `GL_SAMPLE_POSITION`，则产生 `GL_INVALID_ENUM` 错误。

如果 `index` 大于或等于 `GL_SAMPLES` 的值，则产生 `GL_INVALID_VALUE` 错误。

#### 另外查看

`glGenFramebuffers`, `glBindFramebuffer`

#### 版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glGetProgramiv

返回一个来自程序对象的参数。

### C 规范

```
void glGetProgramiv(GLuint program,
                    GLenum pname,
                    GLint *params);
```

#### 参数

`program`

指定将要被查询的程序对象。

`pname`

指定对象参数。可接受的符号名称为 `GL_DELETE_STATUS`、`GL_LINK_STATUS`、`GL_VALIDATE_STATUS`、`GL_INFO_LOG_LENGTH`、`GL_ATTACHED_SHADERS`、`GL_ACTIVE_ATTRIBUTES`、`GL_ACTIVE_ATTRIBUTE_MAX_LENGTH`、`GL_ACTIVE_UNIFORMS`、`GL_ACTIVE_UNIFORM_BLOCKS`、`GL_ACTIVE_UNIFORM_BLOCK_MAX_NAME_LENGTH`、`GL_ACTIVE_UNIFORM_MAX_LENGTH`、`GL_TRANSFORM_FEEDBACK_BUFFER_MODE`、`GL_TRANSFORM_FEEDBACK_VARYINGS`、`GL_TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH`、`GL_GEOMETRY_VERTICES_OUT`、`GL_GEOMETRY_INPUT_TYPE` 和 `GL_GEOMETRY_OUTPUT_TYPE`。

`params`

返回所需对象参数。

#### 描述

`glGetProgram` 将指定程序对象的参数值返回到 `params`。定义的参数如下。

`GL_DELETE_STATUS`

`params` 在 `program` 当前被标记为删除的情况下返回 `GL_TRUE`，在其他情况下返回 `GL_FALSE`。

`GL_LINK_STATUS`

`params` 在最后一次 `program` 上的连接操作成功的情况下返回 `GL_TRUE`，否则返回 `GL_FALSE`。

`GL_VALIDATE_STATUS`

`params` 在最后一次 `program` 上的验证操作成功的情况下返回 `GL_TRUE`，否则返回 `GL_FALSE`。

`GL_INFO_LOG_LENGTH`

`params` 返回 `program` 的信息日志中字符的数量，包括空终止符（也就是说，被请求存储信息日志的字符缓冲器的大小）。如果 `program` 没有任何信息日志，则返回 0。

`GL_ATTACHED_SHADERS`

`params` 返回绑定到 `program` 的着色器对象的数量。

`GL_ACTIVE_ATTRIBUTES`

`params` 返回 `program` 的活动属性变量的数量。

`GL_ACTIVE_ATTRIBUTE_MAX_LENGTH`

`params` 返回 `program` 最长的活动属性名的长度，包括空终止符（也就是说，被请求存储最长的活动属性名的字

符缓冲器的大小)。如果不存在活动的属性,则返回 0。

`GL_ACTIVE_UNIFORMS`

params 返回 program 的活动统一变量的数量。

`GL_ACTIVE_UNIFORM_MAX_LENGTH`

params 返回 program 最长的统一变量名的长度,包括空终止符(也就是说,被请求存储最长的统一变量名的字符缓冲器的大小)。如果不存在活动的统一变量,则返回 0。

`GL_TRANSFORM_FEEDBACK_BUFFER_MODE`

params 返回一个符号常量,指示在激活变换返回时使用的缓冲区模式。这个参数可以为 `GL_SEPARATE_ATTRIBS` 或 `GL_INTERLEAVED_ATTRIBS`。

`GL_TRANSFORM_FEEDBACK_VARYINGS`

params 返回变换反馈模式下要为程序捕捉的 varying 变量的数量。

`GL_TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH`

params 返回要用于变换反馈的最长可用名称的长度,包含空终止符。

`GL_GEOMETRY_VERTICES_OUT`

params 返回 program 中的几何图形着色器将会输出的顶点数的最大值。

`GL_GEOMETRY_INPUT_TYPE`

params 返回一个符号常量,指示可以作为输入而被 program 中包含的几何图形着色器接受的图元类型。

`GL_GEOMETRY_OUTPUT_TYPE`

params 返回一个符号常量,指示可以被 program 中包含的几何图形着色器输出的图元类型。

**注意**

`GL_ACTIVE_UNIFORM_BLOCKS` 和 `GL_ACTIVE_UNIFORM_BLOCK_MAX_NAME_LENGTH` 只在 3.1 或更高版本的 GL 中可用。

`GL_GEOMETRY_VERTICES_OUT`、`GL_GEOMETRY_INPUT_TYPE` 和 `GL_GEOMETRY_OUTPUT_TYPE` 只在 3.2 或更高版本的 GL 中可用。

如果有错误产生,那么 params 的内容不会进行任何修改。

**错误**

如果 program 不是由 OpenGL 产生的值,则产生 `GL_INVALID_VALUE` 错误。

如果 program 没有引用一个程序对象,则产生 `GL_INVALID_OPERATION` 错误。

如果 pname 为 `GL_GEOMETRY_VERTICES_OUT`、`GL_GEOMETRY_INPUT_TYPE` 或 `GL_GEOMETRY_OUTPUT_TYPE`,并且 program 不包含几何图形着色,则产生 `GL_INVALID_OPERATION` 错误。

如果 pname 不是一个可接受的值,则产生 `GL_INVALID_ENUM` 错误。

**相关 Get 函数**

`glGetActiveAttrib`, 自变量为 program。

`glGetActiveUniform`, 自变量为 program。

`glGetAttachedShaders`, 自变量为 program。

`glGetProgramInfoLog`, 自变量为 program。

`glIsProgram`

**另外查看**

`glAttachShader`, `glCreateProgram`, `glDeleteProgram`, `glGetShader`, `glLinkProgram`, `glValidateProgram`

**版权**

Copyright © 2003–2005 3Dlabs Inc. Ltd. Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## `glGetProgramInfoLog`

返回一个程序对象的信息日志。

**C 规范**

```
void glGetProgramInfoLog(GLuint program,
                        GLsizei maxLength,
                        GLsizei *length,
```

GLchar \*infoLog)

#### 参数

program

指定其信息日志将要被查询的程序对象。

maxLength

指定用来存储返回的信息日志的字符缓冲器的大小。

length

返回在 infoLog 中返回的字符串的长度 (不包括空终止符)。

infoLog

指定一个用来返回信息日志的字符数组。

#### 描述

glGetProgramInfoLog 返回指定程序对象的信息日志。一个程序对象的信息日志在这个程序对象被连接或验证时将会被修改。返回的字符串以空字符 null 为结尾。

glGetProgramInfoLog 将尽可能多的信息日志返回到 infoLog, 最多可以有 maxLength 个字符。实际返回的字符的数量 (不包括空终止符) 由 length 指定。如果没有要求返回字符串的长度, 那么可以在 length 自变量中传递一个 NULL 值。可以通过调用值为 GL\_INFO\_LOG\_LENGTH 的 glGetProgram 来获取被请求用来存储返回信息日志的缓冲区的大小。

一个程序对象的信息日志既不是一个空字符串, 也不是一个包含关于最后一次连接操作的信息的字符串, 也不是一个包含关于最后一次验证操作的信息的字符串。它可以包含诊断信息、警告信息和其它信息。当一个程序对象被创建时, 它的信息日志将为一个长度为 0 的字符串。

#### 注意

一个程序对象的信息日志是 OpenGL 实现者 (implementer) 用来输送连接和验证信息的首要机制。因此, 信息日志在开发过程中可能对应用程序开发者有所帮助, 甚至在这些操作已经成功时也是如此。

应用程序开发者不能指望不同的 OpenGL 实现会产生相同的信息日志。

#### 错误

如果 program 不是由 OpenGL 产生的值, 则产生 GL\_INVALID\_VALUE 错误。

如果 program 不是一个程序对象, 则产生 GL\_INVALID\_OPERATION 错误。

如果 maxLength 小于 0, 则产生 GL\_INVALID\_VALUE 错误。

#### 相关 Get 函数

glGetProgram, 其自变量为 GL\_INFO\_LOG\_LENGTH。

glIsProgram

#### 另外查看

glCompileShader, glGetShaderInfoLog, glLinkProgram, glValidateProgram

Copyright

Copyright © 2003–2005 3Dlabs Inc. Ltd. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

### glGetQueryiv

返回一个查询对象目标的参数。

#### C 规范

```
void glGetQueryiv(GLenum target,
                  GLenum pname,
                  GLint * params);
```

#### 参数

target

指定一个查询对象目标。必须是 GL\_SAMPLES\_PASSED、GL\_ANY\_SAMPLES\_PASSED、GL\_PRIMITIVES\_GENERATED、GL\_TRANSFORM\_FEEDBACK\_PRIMITIVES\_WRITTEN、GL\_TIME\_ELAPSED 或 GL\_TIMESTAMP。

pname

指定一个查询对象目标参数的符号名。可接受的值为 GL\_CURRENT\_QUERY 或 GL\_QUERY\_COUNTER\_BITS。

params

返回所需数据。

**描述**

glGetQueryiv 将一个由 tarGet 指定的查询对象目标的选定参数返回到 params。

pname 指定一个特定查询对象目标参数。当 pname 为 GL\_CURRENT\_QUERY 时, tarGet 的当前活动查询的名称 (或者在没有活动查询时为 0) 将会被放入 params。

如果 pname 为 GL\_QUERY\_COUNTER\_BITS, 那么用于保存 target 的查询结果的实现相关位数将会被返回到 params。

**注意**

如果有错误产生, 那么 params 的内容不会进行任何修改。

**错误**

如果 tarGet 或 pname 不是一个可接受的值, 则产生 GL\_INVALID\_ENUM 错误。

**另外查看**

glGetQueryObject, glIsQuery

**版权**

Copyright © 2005 Addison-Wesley. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glGetQueryObject

返回一个查询对象的参数。

**C 规范**

```
void glGetQueryObjectiv(GLuint id,
                        GLenum pname,
                        GLint * params);
void glGetQueryObjectuiv(GLuint id,
                        GLenum pname,
                        GLuint * params);
void glGetQueryObjecti64v(GLuint id,
                        GLenum pname,
                        GLint64 * params);
void glGetQueryObjectui64v(GLuint id,
                        GLenum pname,
                        GLuint64 * params);
```

**参数**

id

指定查询对象的名称。

pname

指定查询对象参数的符号名。可接受的值为 GL\_QUERY\_RESULT 或 GL\_QUERY\_RESULT\_AVAILABLE。

params

返回所需数据。

**描述**

glGetQueryObject 将一个由 id 指定的查询对象目标的选定参数返回到 params。

pname 指定一个特定查询对象参数。pname 可以是如下情况:

GL\_QUERY\_RESULT

params 返回查询对象通过采样计数器的值。初始值为 0。

GL\_QUERY\_RESULT\_AVAILABLE

params 返回通过采样计数器是否立即可用的信息。如果出现等待查询结果的延迟, 则返回 GL\_FALSE; 否则将返回 GL\_TRUE, 这也表明以前所有的查询的结果也都是可用的。

**注意**

如果有错误产生, 那么 params 的内容不会进行任何修改。

glGetQueryObject 隐式地刷新了 GL 管线, 所以由封闭查询所界定的任何未完成的渲染将在有限时间内完成。

如果在调用 glGetQueryObject 之前使用同一个查询对象 id 发布了多个查询, 那么将返回最近查询的结果。这样, 在发布一个新查询时, 以前查询的结果将被丢弃。

glGetQueryObjecti64v 和 glGetQueryObjectui64v 只在 3.3 或更高版本的 GL 中可用。

#### 错误

如果 pname 不是一个可接受的值, 则产生 GL\_INVALID\_ENUM 错误。

如果 id 不是一个查询对象的名称, 则产生 GL\_INVALID\_OPERATION 错误。

如果 id 是一个当前激活的查询对象的名称, 则产生 GL\_INVALID\_OPERATION 错误。

#### 另外查看

glBeginQuery, glEndQuery, glGetQueryiv, glIsQuery, glQueryCounter

#### 版权

Copyright © 2005 Addison-Wesley. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glGetRenderbufferParameteriv

获取关于一个绑定的渲染缓冲区对象的信息。

#### C 规范

```
void glGetRenderbufferParameteriv(GLenum target,
                                   GLenum pname,
                                   GLint *params);
```

#### 参数

target

指定查询操作的目标。target 必须为 GL\_RENDERBUFFER。

pname

指定要从绑定到 target 的渲染缓冲区中获取其值的参数。

params

指定接收查询参数值的数组的地址。

#### 描述

glGetRenderbufferParameteriv 获取关于一个绑定的渲染缓冲区对象的信息。

target 指定查询操作的目标, 必须为 GL\_RENDERBUFFER。pname 指定要查询其值的参数, 必须为 GL\_RENDERBUFFER\_WIDTH、GL\_RENDERBUFFER\_HEIGHT、GL\_RENDERBUFFER\_INTERNAL\_FORMAT、GL\_RENDERBUFFER\_RED\_SIZE、GL\_RENDERBUFFER\_GREEN\_SIZE、GL\_RENDERBUFFER\_BLUE\_SIZE、GL\_RENDERBUFFER\_ALPHA\_SIZE、GL\_RENDERBUFFER\_DEPTH\_SIZE、GL\_RENDERBUFFER\_DEPTH\_SIZE、GL\_RENDERBUFFER\_STENCIL\_SIZE 或 GL\_RENDERBUFFER\_SAMPLES 中的一个。

在成功地从 glGetRenderbufferParameteriv 返回的情况下, 如果 pname 为 GL\_RENDERBUFFER\_WIDTH、GL\_RENDERBUFFER\_HEIGHT、GL\_RENDERBUFFER\_INTERNAL\_FORMAT 或 GL\_RENDERBUFFER\_SAMPLES, 那么 params 将分别包含当前绑定到 target 的渲染缓冲区中图像的宽度像素值、高度像素值、内部格式或样本数量。

如果 pname 为 GL\_RENDERBUFFER\_RED\_SIZE、GL\_RENDERBUFFER\_GREEN\_SIZE、GL\_RENDERBUFFER\_BLUE\_SIZE、GL\_RENDERBUFFER\_ALPHA\_SIZE、GL\_RENDERBUFFER\_DEPTH\_SIZE 或 GL\_RENDERBUFFER\_STENCIL\_SIZE, 那么 params 将会包含当前绑定到 target 的渲染缓冲区中图像的红色、绿色、蓝色、alpha、深度或模板分量的实际分辨率 (并非在图像数组定义时指定的分辨率)。

#### 错误

如果 pname 不是一个可接受的标记, 则产生 GL\_INVALID\_ENUM 错误。

#### 另外查看

glGenRenderbuffers, glFramebufferRenderbuffer, glBindRenderbuffer, glRenderbufferStorage, glRenderbufferStorageMultisample

#### 版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glGetSamplerParameter

返回采样器参数值。

### C 规范

```
void glGetSamplerParameterfv(GLuint sampler,
                             GLenum pname,
                             GLfloat * params);
void glGetSamplerParameteriv(GLuint sampler,
                             GLenum pname,
                             GLint * params);
```

### 参数

sampler

指定从中获取参数的采样器对象的名称。

pname

指定采样器参数的符号名。GL\_TEXTURE\_MAG\_FILTER、GL\_TEXTURE\_MIN\_FILTER、GL\_TEXTURE\_MIN\_LOD、GL\_TEXTURE\_MAX\_LOD、GL\_TEXTURE\_LOD\_BIAS、GL\_TEXTURE\_WRAP\_S、GL\_TEXTURE\_WRAP\_T、GL\_TEXTURE\_WRAP\_R、GL\_TEXTURE\_BORDER\_COLOR、GL\_TEXTURE\_COMPARE\_MODE 和 GL\_TEXTURE\_COMPARE\_FUNC 都可以被接受。

params

返回采样器参数。

### 描述

glGetSamplerParameter 将指定为 pname 的采样器参数的一个或多个值返回到 params。

sampler 定义目标采样器, 并且必须为一个已存在的采样器对象的名称, 从以前的一个 glGenSamplers 调用返回。

pname 接受与 glSamplerParameter 相同的符号, 解释也是相同的;

GL\_TEXTURE\_MAG\_FILTER 返回单个值的纹理放大过滤器, 这是一个符号常量。初始值为 GL\_LINEAR。

GL\_TEXTURE\_MIN\_FILTER 返回单个值的纹理缩小过滤器, 这是一个符号常量。初始值为 GL\_NEAREST\_MIPMAP\_LINEAR。

GL\_TEXTURE\_MIN\_LOD 返回单个值的纹理最小层次细节值。初始值为 -1000。

GL\_TEXTURE\_MAX\_LOD 返回单个值的纹理最大层次细节值。初始值为 1000。

GL\_TEXTURE\_WRAP\_S 返回单个值的纹理坐标环绕函数, 这是一个符号常量。初始值为 GL\_REPEAT。

GL\_TEXTURE\_WRAP\_T 返回单个值的纹理坐标环绕函数, 这是一个符号常量。初始值为 GL\_REPEAT。

GL\_TEXTURE\_WRAP\_R 返回单个值的纹理坐标环绕函数, 这是一个符号常量。初始值为 GL\_REPEAT。

GL\_TEXTURE\_BORDER\_COLOR 返回 4 个整数或浮点数, 包含纹理边缘的 RGBA 颜色。返回的浮点值在 [0,1] 范围内。整数值作为内部浮点表示法一个线性映射返回, 这样 1.0 将会映射到能够表示的最大正整数值, 而 -1.0 则会映射到能够表示的最小负整数值。初始值为 (0, 0, 0, 0)。

GL\_TEXTURE\_COMPARE\_MODE 返回单个值的纹理比较模式, 这是一个符号常量。初始值为 GL\_NONE。参见 glSamplerParameter。

GL\_TEXTURE\_COMPARE\_FUNC 返回单个值的纹理比较函数, 这是一个符号常量。初始值为 GL\_LEQUAL。参见 glSamplerParameter。

### 注意

如果有错误产生, 那么 params 的内容不会进行任何修改。

glGetSamplerParameter 只在 3.3 或更高版本的 GL 中可用。

### 错误

如果 sampler 不是以前的一个 glGenSamplers 调用返回的采样器对象的名称, 则产生 GL\_INVALID\_VALUE 错误。

如果 pname 不是一个可接受的值, 则产生 GL\_INVALID\_ENUM 错误。

### 另外查看

glSamplerParameter, glGenSamplers, glDeleteSamplers, glSamplerParameter

## 版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glGetShaderiv

返回一个来自着色器对象的参数。

### C 规范

```
void glGetShaderiv(GLuint shader,
                  GLenum pname,
                  GLint *params);
```

### 参数

shader

指定要查询的着色器对象。

pname

指定对象参数。可接受的符号名为 GL\_SHADER\_TYPE、GL\_DELETE\_STATUS、GL\_COMPILE\_STATUS、GL\_INFO\_LOG\_LENGTH、GL\_SHADER\_SOURCE\_LENGTH。

params

返回所需对象参数。

### 描述

glGetShader 将指定着色器对象的参数值返回到 params。定义下面的参数。

如果 shader 是一个顶点着色器对象,那么 GL\_SHADER\_TYPE params 返回 GL\_VERTEX\_SHADER,如果 shader 是一个几何图形着色器对象则返回 GL\_GEOMETRY\_SHADER,而如果 shader 是一个片段着色器对象则返回 GL\_FRAGMENT\_SHADER。

GL\_DELETE\_STATUS params 在 shader 当前被标记为删除的情况下返回 GL\_TRUE,在其他情况下返回 GL\_FALSE。

GL\_COMPILE\_STATUS params 在最后一次着色器上的编译操作成功的情况下返回 GL\_TRUE,否则返回 GL\_FALSE。

GL\_INFO\_LOG\_LENGTH params 返回 shader 的信息日志中字符的数量,包括空终止符(也就是说,被请求存储信息日志的字符缓冲器的大小)。如果 shader 没有任何信息日志,则返回 0。

GL\_SHADER\_SOURCE\_LENGTH params 返回组成 shader 的着色器源的一连串的源字符串的长度,包括空终止字符。也就是用来存储着色器源的字符缓冲器的大小。如果不存在源代码,则返回 0。

### 注意

如果有错误产生,那么 params 的内容不会进行任何修改。

### 错误

如果 shader 不是由 OpenGL 产生的值,则产生 GL\_INVALID\_VALUE 错误。

如果 shader 没有引用一个着色器对象,则产生 GL\_INVALID\_OPERATION 错误。

如果 pname 不是一个可接受的值,则产生 GL\_INVALID\_ENUM 错误。

### 相关 Get 函数

glGetShaderInfoLog, 其自变量为 shader。

glGetShaderSource, 其自变量为 shader。

glIsShader

### 另外查看

glCompileShader, glCreateShader, glDeleteShader, glGetProgram, glShaderSource

### 版权

Copyright © 2003–2005 3Dlabs Inc. Ltd. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glGetShaderInfoLog

返回一个着色器对象的信息日志。

### C 规范

```
void glGetShaderInfoLog(GLuint shader,
                        GLsizei maxLength,
                        GLsizei *length,
                        GLchar *infoLog);
```

### 参数

shader

指定其信息日志将要被查询的着色器对象。

maxLength

指定用来存储返回的信息日志的字符缓冲器的大小。

length

返回在 infoLog 中返回的字符串的长度（不包括空终止符）。

infoLog

指定一个用来返回信息日志的字符数组。

### 描述

glGetShaderInfoLog 返回指定着色器对象的信息日志。一个着色器对象的信息日志在这个着色器对象进行编译时将会被修改。返回的字符串以空字符 null 为结尾。

glGetShaderInfoLog 将尽可能多的信息日志返回到 infoLog，最多可以有 maxLength 个字符。实际返回的字符的数量（不包括空终止符）由 length 指定。如果没有要求返回字符串的长度，那么可以在 length 自变量中传递一个 NULL 值。可以通过调用值为 GL\_INFO\_LOG\_LENGTH 的 glGetShader 来获取被请求用来存储返回信息日志的缓冲区的大小。

一个着色器对象的信息日志是一个可能包含诊断消息、警告消息和关于最后一个编译操作的其他信息的字符串。当一个着色器对象被创建时，它的信息日志将为一个长度为 0 的字符串。

### 注意

一个着色器对象的信息日志是 OpenGL 实现者（implementer）用来输送编译处理信息的首要机制。因此，信息日志在开发过程中可能对应用程序开发者有所帮助，甚至在编译已经成功时也是如此。

应用程序开发者不能指望不同的 OpenGL 实现会产生相同的信息日志。

### 错误

如果 shader 不是由 OpenGL 产生的值，则产生 GL\_INVALID\_VALUE 错误。

如果 shader 不是一个着色器对象，则产生 GL\_INVALID\_OPERATION 错误。

如果 maxLength 小于 0，则产生 GL\_INVALID\_VALUE 错误。

### 相关 Get 函数

glGetShader，其自变量为 GL\_INFO\_LOG\_LENGTH。

glIsShader

另外查看

glCompileShader, glGetProgramInfoLog, glLinkProgram, glValidateProgram

### 版权

Copyright © 2003–2005 3Dlabs Inc. Ltd. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glGetShaderSource

返回来自一个着色器对象的源代码字符串。

### C 规范

```
void glGetShaderSource(GLuint shader,
                      GLsizei bufSize,
```

```
GLsizei * length,
GLchar * source);
```

#### 参数

shader

指定要查询的着色器对象。

bufSize

指定用来存储返回的源代码字符串的字符缓冲器的大小。

length

返回在 source 中返回的字符串的长度 (不包括空终止符)。

source

指定一个用来返回源代码字符串的字符数组。

#### 描述

glGetShaderSource 返回来由 shader 指定的着色器对象的源代码字符串的串联。着色器对象的源代码字符串是以前调用 glShaderSource 的结果。由函数返回的字符串将以空字符 null 为结尾。

glGetShaderSource 将尽可能多的源代码字符串 (最多可达到 bufSize 个字符) 返回到 isource 中。实际返回的字符的数量 (不包括空终止符) 由 length 指定。如果没有要求返回字符串的长度, 那么可以在 length 自变量中传递一个 NULL 值。可以通过调用值为 GL\_SHADER\_SOURCE\_LENGTH 的 glGetShader 来获取被请求用来存储返回源代码字符串的缓冲区的大小。

#### 错误

如果 shader 不是由 OpenGL 产生的值, 则产生 GL\_INVALID\_VALUE 错误。

如果 shader 不是一个着色器对象, 则产生 GL\_INVALID\_OPERATION 错误。

如果 bufSize 小于 0, 则产生 GL\_INVALID\_VALUE 错误。

#### 相关 Get 函数

glGetShader, 其自变量为 GL\_SHADER\_SOURCE\_LENGTH。

glIsShader

#### 另外查看

glCreateShader, glShaderSource

#### 版权

Copyright © 2003–2005 3Dlabs Inc. Ltd. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glGetString

返回一个描述当前 GL 连接的字符串。

#### C 规范

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

#### C 规范

```
const GLubyte* glGetStringi (GLenum name,
                             GLuint index);
```

#### 参数

name

name 指定一个符号常量, 为 GL\_VENDOR、GL\_RENDERER、GL\_VERSION 或 GL\_SHADING\_LANGUAGE\_VERSION 中的一个。此外, glGetStringi 可以接受 GL\_EXTENSIONS 标记。

index

对于 glGetStringi, 指定将要返回的字符串的索引。

#### 描述

glGetString 返回一个指向描述当前 GL 连接的某些方面的静态字符串的指针。

name 可以是下列值中的一个。

GL\_VENDOR 返回为这个 GL 实现负责的公司名。这个名称在各发布版本之间不会改变。

GL\_RENDERER

返回渲染的名称。典型情况下, 这个名称是一个硬件平台的特殊配置所特有的。这个名称在各发布版本之间不会改变。

GL\_VERSION

返回一个版本或发布号。

GL\_SHADING\_LANGUAGE\_VERSION

返回一个着色语言的版本或发布号。

glGetStringi 返回一个指向由 index 进行索引的一个静态字符串的指针。name 可以是下列值中的一个。

GL\_EXTENSIONS 只针对 glGetStringi, 返回实现在 index 支持的扩展字符串。

GL\_VENDOR 和 GL\_RENDERER 字符串一起唯一地指定一个平台。它们在发布版本之间不会改变, 并且应该用于平台识别算法中。

GL\_VERSION 和 GL\_SHADING\_LANGUAGE\_VERSION 字符串以一个版本数字开始。

这个版本数字使用下列格式中的一个。

major\_number.minor\_number major\_number.minor\_number.release\_number

供应商指定的信息可能会在版本数字之后出现。它的格式取决于实现, 但总是用空格来分隔版本数和供应商指定的信息。

所有字符串都以 null 结束。

注意

如果有错误产生, glGetString 返回 0。

客户机和服务器可能支持不同版本或扩展。glGetString 总是返回一个兼容的版本数字或扩展列表。发行号总是描述服务器的。

错误

如果 name 不是一个可接受的值, 则产生 GL\_INVALID\_ENUM 错误。

如果 index 位于索引状态 name 的有效范围之外, 则 glGetStringi 产生 GL\_INVALID\_VALUE 错误。

版权

Copyright © 1991–2006 Silicon Graphics, Inc. Copyright © 2010 Khronos Group. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glGetSynciv

查询同步对象的属性。

### C 规范

```
void glGetSynciv(GLsync sync,
                 GLenum pname,
                 GLsizei bufSize,
                 GLsizei *length,
                 GLint *values);
```

### 参数

sync

指定其属性将要被查询的同步对象。

pname

指定要从 sync 中指定的同步对象中获取其值的参数。

bufSize

指定其地址在 values 中给出的缓冲区的大小。

length

指定接收位于 values 中的整数值数量的变量的地址。

values

指定接收查询参数值的数组的地址。

描述

glGetSynciv 检索一个同步对象的属性。sync 指定要检索其属性的同步对象的名称。

如果成功, glGetSynciv 将会使用被查询对象的相应正确值来替换 values 中的最多 bufSize 个整数。替换的整数值的实际数量返回到在 length 中指定地址的变量中。如果 length 为 NULL, 则不会返回长度。

如果 pname 为 GL\_OBJECT\_TYPE, 那么一个代表同步对象指定类型的单个值就会被放入 values 中。唯一支持的类型就是 GL\_SYNC\_FENCE。

如果 pname 为 GL\_SYNC\_STATUS, 那么一个代表同步对象状态 (GL\_SIGNALED 或 GL\_UNSIGNALED) 的单个值就会被放入 values 中。

如果 pname 为 GL\_SYNC\_CONDITION, 那么一个代表同步对象条件的单个值就会被放入 values 中。唯一支持的条件就是 GL\_SYNC\_GPU\_COMMANDS\_COMPLETE。

如果 pname 为 GL\_SYNC\_FLAGS, 那么一个代表创建同步对象的标记的单个值就会被放入 values 中。目前不支持任何标记。

如果出现错误, 那么就不会向 values 或 length 写入任何东西。

**错误**

如果 sync 不是一个同步对象的名称, 则产生 GL\_INVALID\_VALUE 错误。

如果 pname 不是一个可接受的标记, 则产生 GL\_INVALID\_ENUM 错误。

**另外查看**

glFenceSync, glWaitSync, glClientWaitSync

**版权**

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glGetTexImage

返回一个纹理图像

**C 规范**

```
void glGetTexImage(GLenum target,
                  GLint level,
                  GLenum format,
                  GLenum type,
                  GLvoid * img);
```

**参数**

target

target 指定将要获得哪个纹理。GL\_TEXTURE\_1D、GL\_TEXTURE\_2D、GL\_TEXTURE\_3D、GL\_TEXTURE\_1D\_ARRAY、GL\_TEXTURE\_2D\_ARRAY、GL\_TEXTURE\_RECTANGLE、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Z 和 GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Z 都可以被接受。

level

指定期望图像的层次细节数量。Level 0 是基本图像层次。

Level n 是第 n 级 Mip 贴图缩略图。

format

为返回数据指定一个像素格式。支持的格式有 GL\_STENCIL\_INDEX、GL\_DEPTH\_COMPONENT、GL\_DEPTH\_STENCIL、GL\_RED、GL\_GREEN、GL\_BLUE、GL\_RG、GL\_RGB、GL\_RGBA、GL\_BGR、GL\_BGRA、GL\_RED\_INTEGER、GL\_GREEN\_INTEGER、GL\_BLUE\_INTEGER、GL\_RG\_INTEGER、GL\_RGB\_INTEGER、GL\_RGBA\_INTEGER、GL\_BGR\_INTEGER、GL\_BGRA\_INTEGER。

type

为返回数据指定一个像素类型。支持的类型有 GL\_UNSIGNED\_BYTE、GL\_BYTE、GL\_UNSIGNED\_SHORT、GL\_SHORT、GL\_UNSIGNED\_INT、GL\_INT、GL\_HALF\_FLOAT、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、GL\_UNSIGNED\_INT\_2\_10\_10\_10\_REV、GL\_UNSIGNED\_INT\_24\_8、

GL\_UNSIGNED\_INT\_10F\_11F\_11F\_REV、GL\_UNSIGNED\_INT\_5\_9\_9\_9\_REV 和 GL\_FLOAT\_32\_UNSIGNED\_INT\_24\_8\_REV。

img

返回这个纹理图像 应该为指向一个由 type 指定类型的数组。

描述

glGetTexImage 将一个纹理图像返回到 img。

tarGet 指定需要的纹理图像是否是由 glTexImage1D (GL\_TEXTURE\_1D)、glTexImage2D (GL\_TEXTURE\_1D\_ARRAY、GL\_TEXTURE\_RECTANGLE、GL\_TEXTURE\_2D 或任意 GL\_TEXTURE\_CUBE\_MAP\_\*), 或者 glTexImage3D (GL\_TEXTURE\_2D\_ARRAY、GL\_TEXTURE\_3D)指定的。level 指定期望图像的层次细节数量。format 和 type 指定期望图像数组的格式和类型。要了解关于 format 和 type 参数可接受值的描述, 可以参考 glTexImage1D 的参考页。

如果在纹理图像被指定的情况下, 所需的非 0 缓冲区对象被绑定到 GL\_PIXEL\_PACK\_BUFFER 目标 (参见 glBindBuffer), img 将被看作缓冲区对象数据存储的一个字节偏移。

要理解 glGetTexImage 的操作, 可以将选定的内部四分量纹理图像看作一个大小为图像大小的 RGBA 颜色缓冲区。这样 glGetTexImage 的语义就和 glReadPixels 的语义相同了, 除了在以同样的 format 和 type、x 和 y 设为 0、width 设为纹理图像的宽度、height 设为 1 (1D 图像) 或设为纹理图像的高度 (2D 图像) 进行调用时不执行任何像素变换操作之外。

如果选定的纹理图像不包括 4 个分量, 那么将会应用下列映射。单分量纹理都被视为红色设为单分量值、绿色设为 0、蓝色设为 0、Alpha 设为 1 的 RGBA 缓冲区。二分量纹理都被视为红色设为分量 0 的值、Alpha 设为分量 1 的值、绿色设为 0、蓝色设为 0 的 RGBA 缓冲区。最后, 三分量纹理都被视为红色设为分量 0、绿色设为分量 1、蓝色设为分量 2、Alpha 设为 1 的 RGBA 缓冲区。

要确定 img 所需要的大小, 使用 glGetTexLevelParameter 来确定内部纹理图像的尺寸, 然后基于 format 和 type 将所需的像素数放到每个像素需要的存储空间。确保把像素存储参数计算进去, 特别是 GL\_PACK\_ALIGNMENT。

注意

如果有错误产生, 那么 img 的内容不会进行任何修改。

glGetTexImage 为活动的纹理单元返回纹理图像。

错误

如果 tarGet、format 或 type 不是一个可接受的值, 则产生 GL\_INVALID\_ENUM 错误。

如果 level 小于 0, 则产生 GL\_INVALID\_VALUE 错误。

如果 level 大于  $\log_2 \max$  值, 其中 max 是 GL\_MAX\_TEXTURE\_SIZE 的返回值, 则可能产生 GL\_INVALID\_VALUE 错误。

如果 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 或 GL\_UNSIGNED\_INT\_10F\_11F\_11F\_REV 中的一个, 并且 format 不是 GL\_RGB, 则返回 GL\_INVALID\_OPERATION。

如果 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、GL\_UNSIGNED\_INT\_2\_10\_10\_10\_REV 或 GL\_UNSIGNED\_INT\_5\_9\_9\_9\_REV 中的一个, 并且 format 既不是 GL\_RGBA 也不是 GL\_BGRA, 则返回 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_PACK\_BUFFER 目标, 并且缓冲区对象的数据存储当前被映射, 则产生 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_PACK\_BUFFER 目标, 并且数据将要被打包到缓冲区对象, 以致内存写入请求将超出数据存储大小, 则产生 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_PACK\_BUFFER 目标, 并且 img 没有平均分成将一个由 type 指定的数据存储到内存中所需的字节数, 则产生 GL\_INVALID\_OPERATION 错误。

相关 Get 函数

glGetTexLevelParameter, 其自变量为 GL\_TEXTURE\_WIDTH。

glGetTexLevelParameter, 其自变量为 GL\_TEXTURE\_HEIGHT。

glGetTexLevelParameter, 其自变量为 GL\_TEXTURE\_INTERNAL\_FORMAT。

glGet, 其自变量为 GL\_PACK\_ALIGNMENT 和其他一些值。

glGet, 其自变量为 GL\_PIXEL\_PACK\_BUFFER\_BINDING。

另外查看

glActiveTexture, glReadPixels, glTexImage1D, glTexImage2D, glTexImage3D, glTexSubImage1D, glTexSubImage2D, glTexSubImage3D, glGetTexParameter

版权

Copyright © 1991–2006 Silicon Graphics, Inc. Copyright © 2010 Khronos Group. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glGetTexLevelParameter

返回一个指定层次细节的纹理参数值。

### C 规范

```
void glGetTexLevelParameterfv(GLenum target,
                              GLint level,
                              GLenum pname,
                              GLfloat * params);

void glGetTexLevelParameteriv(GLenum target,
                              GLint level,
                              GLenum pname,
                              GLint * params);
```

### 参数

target

指定目标纹理的符号名, 为 GL\_TEXTURE\_1D、GL\_TEXTURE\_2D、GL\_TEXTURE\_3D、GL\_TEXTURE\_1D\_ARRAY、GL\_TEXTURE\_2D\_ARRAY、GL\_TEXTURE\_RECTANGLE、GL\_TEXTURE\_2D\_MULTISAMPLE、GL\_TEXTURE\_2D\_MULTISAMPLE\_ARRAY、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Z、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Z、GL\_PROXY\_TEXTURE\_1D、GL\_PROXY\_TEXTURE\_2D、GL\_PROXY\_TEXTURE\_3D、GL\_PROXY\_TEXTURE\_1D\_ARRAY、GL\_PROXY\_TEXTURE\_2D\_ARRAY、GL\_PROXY\_TEXTURE\_RECTANGLE、GL\_PROXY\_TEXTURE\_2D\_MULTISAMPLE、GL\_PROXY\_TEXTURE\_2D\_MULTISAMPLE\_ARRAY、GL\_PROXY\_TEXTURE\_CUBE\_MAP 或 GL\_TEXTURE\_BUFFER 中的一个。

level

指定期望图像的层次细节数量。Level 0 是基本图像层次。

Level n 是第 n 级 Mip 贴图缩略图。

pname

指定纹理参数的符号名。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\_DEPTH\_SIZE、GL\_TEXTURE\_COMPRESSED 和 GL\_TEXTURE\_COMPRESSED\_IMAGE\_SIZE 都可以被接受。

params

返回所需数据。

### 描述

glGetTexLevelParameter 在 params 中返回一个指定层次细节的纹理参数值, 指定为 level。

target 定义目标纹理, 可以是 GL\_TEXTURE\_1D、GL\_TEXTURE\_2D、GL\_TEXTURE\_3D、GL\_PROXY\_TEXTURE\_1D、GL\_PROXY\_TEXTURE\_2D、GL\_PROXY\_TEXTURE\_3D、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Z、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Z 或 GL\_PROXY\_TEXTURE\_CUBE\_MAP。

GL\_MAX\_TEXTURE\_SIZE 和 GL\_MAX\_3D\_TEXTURE\_SIZE 都没有进行充分的描述。它需要报告能够以 Mip 贴图和边界的最大正方形纹理图像, 但是细长的纹理或没有 Mip 贴图和边界的纹理能够轻松适应纹理内存。代理目标允许用户进行更加精确地查询 GL 是否能够适应一个给定配置的纹理。如果纹理不能适应, 那么能够通过 glGetTexLevelParameter 进行查询的纹理状态变量将回被设为 0。如果纹理能够适应, 那么纹理状态值将被按照非代

理目标一样设置。

pname 指定其一个或多个值将被返回的纹理参数。

可以接受的参数名如下。

GL\_TEXTURE\_WIDTH

params 返回单个整数值, 即纹理图像的宽度。这个值包括纹理图像的边界。初始值为 0。

GL\_TEXTURE\_HEIGHT

params 返回单个整数值, 即纹理图像的高度。这个值包括纹理图像的边界。初始值为 0。

GL\_TEXTURE\_DEPTH

params 返回单个整数值, 即纹理图像的深度。这个值包括纹理图像的边界。初始值为 0。

GL\_TEXTURE\_INTERNAL\_FORMAT

params 返回单个整数值, 即纹理图像的内部格式。

GL\_TEXTURE\_RED\_TYPE,

GL\_TEXTURE\_GREEN\_TYPE,

GL\_TEXTURE\_BLUE\_TYPE,

GL\_TEXTURE\_ALPHA\_TYPE,

GL\_TEXTURE\_DEPTH\_TYPE

用于存储分量的数据类型。可以返回 GL\_NONE, GL\_SIGNED\_NORMALIZED、GL\_UNSIGNED\_NORMALIZED、GL\_FLOAT, GL\_INT 和 GL\_UNSIGNED\_INT 来分别指示有符号标准化定点数、无符号标准化定点数、浮点数、整数非标准化值和无符号整数非标准化分量。

GL\_TEXTURE\_RED\_SIZE,

GL\_TEXTURE\_GREEN\_SIZE,

GL\_TEXTURE\_BLUE\_SIZE,

GL\_TEXTURE\_ALPHA\_SIZE,

GL\_TEXTURE\_DEPTH\_SIZE

一个独立分量的内部存储格式精度。GL 选择的精度将与用户要求的 glTexImage1D、glTexImage2D、glTexImage3D、glCopyTexImage1D 和 glCopyTexImage2D 的 component 变量精度非常接近。初始值为 0。

GL\_TEXTURE\_COMPRESSED

params 返回单个布尔值, 指示纹理图像要以一种压缩内部格式进行存储。初始值为 GL\_FALSE。

GL\_TEXTURE\_COMPRESSED\_IMAGE\_SIZE

params 返回单个整数值, 即将要从 glGetCompressedTexImage 中返回的压缩纹理图像的无符号字节数。

**注意**

如果有错误产生, 那么 params 的内容不会进行任何修改。

glGetTexLevelParameter 为活动的纹理单元返回纹理层次参数。

**错误**

如果 tarGet 或 pname 不是一个可接受的值, 则产生 GL\_INVALID\_ENUM 错误。

如果 level 小于 0, 则产生 GL\_INVALID\_VALUE 错误。

如果 level 大于  $\log_2 \max$  值, 其中 max 是 GL\_MAX\_TEXTURE\_SIZE 的返回值, 则可能产生 GL\_INVALID\_VALUE 错误。

如果 tarGet 为 GL\_TEXTURE\_BUFFER, 并且 level 为非 0 值, 则产生 GL\_INVALID\_VALUE 错误。

如果 GL\_TEXTURE\_COMPRESSED\_IMAGE\_SIZE 在非压缩内部格式的纹理图像或代理目标上进行查询, 则产生 GL\_INVALID\_OPERATION 错误。

**另外查看**

glActiveTexture, glGetTexParameter, glCopyTexImage1D, glCopyTexImage2D, glCopyTexSubImage1D, glCopyTexSubImage2D, glCopyTexSubImage3D, glTexImage1D, glTexImage2D, glTexImage3D, glTexSubImage1D, glTexSubImage2D, glTexSubImage3D, glGetTexParameter

**版权**

Copyright © 1991–2006 Silicon Graphics, Inc. Copyright © 2010 Khronos Group. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glGetTexParameter

返回纹理参数值。

### C 规范

```
void glGetTexParameterfv(GLenum target,
                        GLenum pname,
                        GLfloat * params);

void glGetTexParameteriv(GLenum target,
                        GLenum pname,
                        GLint * params);
```

### 参数

#### target

指定目标纹理的符号名。GL\_TEXTURE\_1D、GL\_TEXTURE\_2D、GL\_TEXTURE\_1D\_ARRAY、GL\_TEXTURE\_2D\_ARRAY、GL\_TEXTURE\_3D、GL\_TEXTURE\_RECTANGLE 和 GL\_TEXTURE\_CUBE\_MAP 都可以被接受。

#### pname

指定纹理参数的符号名。GL\_TEXTURE\_BASE\_LEVEL、GL\_TEXTURE\_BORDER\_COLOR、GL\_TEXTURE\_COMPARE\_MODE、GL\_TEXTURE\_COMPARE\_FUNC、GL\_TEXTURE\_LOD\_BIAS、GL\_TEXTURE\_MAG\_FILTER、GL\_TEXTURE\_MAX\_LEVEL、GL\_TEXTURE\_MAX\_LOD、GL\_TEXTURE\_MIN\_FILTER、GL\_TEXTURE\_MIN\_LOD、GL\_TEXTURE\_SWIZZLE\_R、GL\_TEXTURE\_SWIZZLE\_G、GL\_TEXTURE\_SWIZZLE\_B、GL\_TEXTURE\_SWIZZLE\_A、GL\_TEXTURE\_SWIZZLE\_RGBA、GL\_TEXTURE\_WRAP\_S、GL\_TEXTURE\_WRAP\_T 和 GL\_TEXTURE\_WRAP\_R 都可以被接受。

#### params

返回这个纹理参数。

### 描述

glGetTexParameter 将指定为 pname 的纹理参数的一个或多个值返回到 params。target 定义目标纹理。GL\_TEXTURE\_1D、GL\_TEXTURE\_2D、GL\_TEXTURE\_3D、GL\_TEXTURE\_1D\_ARRAY、GL\_TEXTURE\_2D\_ARRAY、GL\_TEXTURE\_RECTANGLE 和 GL\_TEXTURE\_CUBE\_MAP 分别指定一维纹理、二维纹理、三维纹理、一维数组纹理、二维数组纹理、矩形纹理和立方体贴图纹理。pname 接受与 glGetTexParameter 相同的符号，解释也是相同的。

GL\_TEXTURE\_MAG\_FILTER 返回单个值的纹理放大过滤器，这是一个符号常量。初始值为 GL\_LINEAR。

GL\_TEXTURE\_MIN\_FILTER 返回单个值的纹理缩小过滤器，这是一个符号常量。初始值为 GL\_NEAREST\_MIPMAP\_LINEAR。

GL\_TEXTURE\_MIN\_LOD 返回单个值的纹理最小层次细节值。初始值为 -1000。

GL\_TEXTURE\_MAX\_LOD 返回单个值的纹理最大层次细节值。初始值为 1000。

GL\_TEXTURE\_BASE\_LEVEL 返回单个值的基本纹理 Mip 贴图层次。初始值为 0。

GL\_TEXTURE\_MAX\_LEVEL 返回单个值的最大纹理 Mip 贴图数组层次。初始值为 1000。

GL\_TEXTURE\_SWIZZLE\_R 返回红色分量 swizzle 操作。初始值为 GL\_RED。

GL\_TEXTURE\_SWIZZLE\_G 返回绿色分量 swizzle 操作。初始值为 GL\_GREEN。

GL\_TEXTURE\_SWIZZLE\_B 返回蓝色分量 swizzle 操作。初始值为 GL\_BLUE。

GL\_TEXTURE\_SWIZZLE\_A 返回 alpha 分量 swizzle 操作。初始值为 GL\_ALPHA。

GL\_TEXTURE\_SWIZZLE\_RGBA 返回单个查询中所有通道的分量 swizzle 操作。

GL\_TEXTURE\_WRAP\_S 返回单个值的纹理坐标环绕函数，这是一个符号常量。初始值为 GL\_REPEAT。

GL\_TEXTURE\_WRAP\_T 返回单个值的纹理坐标环绕函数，这是一个符号常量。初始值为 GL\_REPEAT。

GL\_TEXTURE\_WRAP\_R 返回单个值的纹理坐标环绕函数，这是一个符号常量。初始值为 GL\_REPEAT。

GL\_TEXTURE\_BORDER\_COLOR 返回 4 个整数或浮点数，包含纹理边缘的 RGBA 颜色。返回的浮点值在 [0,1] 范围内。整数值作为内部浮点表示法一个线性映射返回，这样 1.0 将会映射到能够表示的最大正整数值，而 -1.0 则会映射到能够表示的最小负整数值。初始值为 (0, 0, 0, 0)。

GL\_TEXTURE\_COMPARE\_MODE 返回单个值的纹理比较模式，这是一个符号常量。初始值为 GL\_NONE。参见 glGetTexParameter。

GL\_TEXTURE\_COMPARE\_FUNC 返回单个值的纹理比较函数，这是一个符号常量。初始值为 GL\_LEQUAL。参

见 `glTexParameter`。

#### 注意

如果有错误产生, 那么 `params` 的内容不会进行任何修改。

#### 错误

如果 `target` 或 `pname` 不是一个可接受的值, 则产生 `GL_INVALID_ENUM` 错误。

#### 另外查看

`glTexParameter`

#### 版权

Copyright©1991–2006 Silicon Graphics, Inc. Copyright©2010 Khronos Group. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glGetTransformFeedbackVarying

检索关于为了进行变换反馈而选择的 `varying` 变量的信息。

### C 规范

```
void glGetTransformFeedbackVarying(GLuint program,
                                   GLuint index,
                                   GLsizei bufSize,
                                   GLsizei * length,
                                   GLsizei size,
                                   GLenum * type, char * name);
```

#### 参数

`program`

目标程序对象的名称。

`index`

指定相关信息将要被检索的 `varying` 变量的索引。

`bufSize`

字符的最大数量 (包括空终结符 `null`) , 它将被写入到 `name`。

`length`

将要接收写入到 `name` 的字符数的变量的地址, 不包含空终结符。如果 `length` 为 `NULL`, 则不会返回长度。

`size`

接收 `varying` 变量大小的变量的地址。

`type`

接收 `varying` 变量类型的变量的地址。

`name`

将要写入 `varying` 变量名称的缓冲区的地址。

#### 描述

可以通过调用 `glGetTransformFeedbackVarying` 来获取将在变换反馈过程中捕获的连接程序中的 `varying` 变量设置的相关信息。

`glGetTransformFeedbackVarying` 提供关于由 `index` 选择的 `varying` 变量的信息。`index` 为 0 会选择由传递到 `glTransformFeedbackVaryings` 的 `varyings` 数组中指定的第一个 `varying` 变量, 而 `index` 为 `GL_TRANSFORM_FEEDBACK_VARYINGS-1` 则会选择最后一个。

选定 `varying` 变量的名称将作为一个以 `null` 为结尾的字符串返回到 `name` 中。字符的实际数量被写入到 `name`, 但不包括空终结符 `null`, 它被返回到 `length`。如果 `length` 为 `NULL`, 则不会返回长度。字符的最大数量 (包括空终结符 `null`) 由 `bufSize` 指定。

`program` 中最长的 `varying` 变量的名称的长度是由 `GL_TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH` 给出的, 它可以使用 `glGetProgram` 查询。

对于选定的 `varying` 变量, 这个类型被返回到 `type`。`varying` 变量的大小将会返回到 `size`。`size` 中的值以 `type` 中返回的类型为单位。返回的类型可以是由 `glGetActiveAttrib` 返回的任何标量、向量或矩阵属性类型。如果出现错误, 那么返回参数 `length`、`size`、`type` 和 `name` 将为未修改的。这个命令将会返回关于这个 `varying` 变量尽可能多的信息。

如果没有可用的信息, 那么 `length` 将被设为 0, 而 `name` 将为空字符串。在一次失败的链接后调用 `glGetTransformFeedbackVarying` 就可能会出现这种情况。

#### 错误

如果 `program` 不是一个程序对象的名称, 则产生 `GL_INVALID_VALUE` 错误。

如果 `index` 大于或等于 `GL_TRANSFORM_FEEDBACK_VARYINGS` 的值, 则产生 `GL_INVALID_VALUE` 错误。

如果 `program` 没有进行连接, 则产生 `GL_INVALID_OPERATION` 错误。

#### 相关 Get 函数

`glGetProgram`, 其自变量为 `GL_TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH`。

#### 另外查看

`glBeginTransformFeedback`, `glEndTransformFeedback`, `glTransformFeedbackVaryings`, `glGetProgram`

#### 版权

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glGetUniform

返回一个统一变量的值。

### C 规范

```
void glGetUniformfv(GLuint program,
                    GLint location,
                    GLfloat * params);

void glGetUniformiv(GLuint program,
                    GLint location,
                    GLint * params);
```

### 参数

`program`

指定将要被查询的程序对象。

`location`

指定将要被查询的统一变量的位置。

`params`

返回指定统一变量的值。

### 描述

`glGetUniform` 将指定统一变量的一个或多个值返回到 `params`。由 `location` 指定的统一变量的类型决定了返回值的数量。如果统一变量是在着色器中作为布尔值、整数或浮点值定义的, 那么将返回单个值。如果它是作为 `vec2`、`ivec2` 或 `bvec2` 定义的, 那么将返回两个值。如果它是作为 `vec3`、`ivec3` 或 `bvec3` 定义的, 那么将返回 3 个值, 依此类推。要查询存储在作为数组声明的统一变量中的值, 可以为数组中的每一个元素调用 `glGetUniform`。要查询存储在作为结构声明的统一变量中的值, 可以为结构中的每一个域调用 `glGetUniform`。为矩阵声明的统一变量的值将以列为主的顺序被返回。

分配给统一变量的位置在程序对象被连接之前是未知的。

在连接出现之后, `glGetUniformLocation` 命令可以被用来获得一个统一变量的位置。随后这个位置值可以被传输到 `glGetUniform`, 以便查询这个统一变量的当前值。在一个程序对象被成功连接之后, 统一变量索引值将保持不变, 直到下次连接命令发生。如果一个连接成功, 那么统一变量值只能在这个连接之后进行查询。

### 注意

如果有错误产生, 那么 `params` 的内容不会进行任何修改。

### 错误

如果 `program` 不是由 OpenGL 产生的值, 则产生 `GL_INVALID_VALUE` 错误。

如果 `program` 不是一个程序对象, 则产生 `GL_INVALID_OPERATION` 错误。

如果 `program` 没有成功进行连接, 则产生 `GL_INVALID_OPERATION` 错误。

如果 `location` 没有与指定程序对象的一个有效统一变量位置一致, 则产生 `GL_INVALID_OPERATION` 错误。

### 相关 Get 函数

glGetActiveUniform, 其自变量为 program 和一个活动统一变量的索引。

glGetProgram, 其自变量为 program, 以及 GL\_ACTIVE\_UNIFORMS 或 GL\_ACTIVE\_UNIFORM\_MAX\_LENGTH。

glGetUniformLocation, 其自变量为 program 和一个统一变量的名称。

glIsProgram

另外查看

glCreateProgram, glLinkProgram, glUniform

版权

Copyright © 2003–2005 3Dlabs Inc. Ltd. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glGetUniformBlockIndex

检索一个指定统一块的索引。

### C 规范

```
GLuint glGetUniformBlockIndex(GLuint program,
                              const GLchar *uniformBlockName);
```

### 参数

program

指定一个包含统一块的程序的名称。

uniformBlockName

指定一个要检索其索引的包含统一块名称的字符数组的地址。

### 描述

glGetUniformBlockIndex 检索 program 中一个统一块的索引。

program 必须是一个程序对象的名称, 过去必须已经为这个程序对象调用了 glLinkProgram 命令, 虽然并不要求 glLinkProgram 必须成功。这个连接可能会失败, 因为活动统一变量的数量超出了限制。

统一变量 BlockName 必须包含指定统一块名称的空终止字符串。

glGetUniformBlockIndex 返回 program 中名为 uniformBlockName 的统一块的同一块索引。如果 uniformBlockName 没有定义 program 的一个活动统一块, 那么 glGetUniformBlockIndex 将返回特殊标识符 GL\_INVALID\_INDEX。一个程序的活动统一块的索引会从 0 开始按顺序分配。

### 错误

如果 program 不是过去为其调用过 glLinkProgram 的一个程序对象的名称, 则产生 GL\_INVALID\_OPERATION 错误。

### 注意

glGetUniformBlockIndex 只在 3.1 或更高版本的 GL 中可用。

### 另外查看

glGetActiveUniformBlockName, glGetActiveUniformBlock, glLinkProgram

### 版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glGetUniformIndices

检索一个指定统一块的索引。

### C 规范

```
GLuint glGetUniformIndices(GLuint program,
                           GLsizei uniformCount,
                           const GLchar **uniformNames,
                           GLuint *uniformIndices);
```

### 参数

program

指定一个包含将要查询索引的统一变量的程序的名称。

uniformCount

指定将要查询索引的统一变量数量。

uniformNames

指定一个指向包含查询的统一变量名称的缓冲区的指针数组的地址。

uniformIndices

指定将要接收统一变量索引的数组的地址。

描述

glGetUniformIndices 检索 program 中一些统一变量的索引。

program 必须是一个程序对象的名称, 过去必须已经为这个程序对象调用了 glLinkProgram 命令, 虽然并不要求 glLinkProgram 必须成功。这个连接可能会失败, 因为活动统一变量的数量超出了限制。

uniformCount 指示 uniformNames 名称数组中元素数量, 以及可能被写入 uniformIndices 的索引数量。

uniformNames 包含一个 uniformCount 名称列表字符串, 这些名称定义了将要查询索引的统一变量名称。对于 uniformNames 中的每一个名称字符串, 分配给以此命名的活动统一变量的索引将被写入到 uniformIndices 中的相应元素中。如果 uniformNames 中的一个名称字符串不是活动统一变量的名称, 那么特殊值 GL\_INVALID\_INDEX 将被写入到 uniformIndices 中的相应元素中。

如果出现错误, 那么就不会向 uniformIndices 写入任何东西。

错误

如果 program 不是过去为其调用过 glLinkProgram 的一个程序对象的名称, 则产生 GL\_INVALID\_OPERATION 错误。

注意

glGetUniformIndices 只在 3.1 或更高版本的 GL 中可用。

另外查看

glGetActiveUniform, glGetActiveUniformName, glLinkProgram

版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glGetUniformLocation

返回一个统一变量的位置。

C 规范

```
GLint glGetUniformLocation(GLuint program,  
                           const GLchar * name);
```

参数

program

指定将要被查询的程序对象。

name

指向一个包含其位置要被查询的统一变量名称的空终止字符串。

描述

glGetUniformLocation 返回一个整数, 代表一个程序对象中的指定统一变量的位置。name 必须是一个不包含任何空白的空终止字符串。

name 必须是非结构、结构数组或者向量或矩阵的子分量的 program 中的一个活动统一变量名称。如果 name 与 program 中的一个活动统一变量不一致, 或者 name 以保留前缀 "gl\_" 开始, 那么这个函数返回 -1。

对于作为结构或结构数组的统一变量来说, 可以通过调用 glGetUniformLocation 来查询结构中的每个域。数组元素操作符 "[" 和结构域操作符 "." 可以用在 name 中, 以选择结构中的一个数组或域中的元素。使用这些操作符的结果不允许是其他结构、结构数组或者向量或矩阵的子分量。除非 name 的最后部分表示一个统一变量数组, 一个数组的第一个元素的位置可以通过使用数组名或使用附加 "[0]" 的名称来检索。

分配给统一变量的实际位置在程序对象成功连接之前是未知的。在连接出现之后, `glGetUniformLocation` 命令可以被用来获得一个统一变量的位置。随后这个位置值可以被传递到 `glUniform` 来设置统一变量的值, 或者传递到 `glGetUniform` 来查询统一变量的当前值。在一个程序对象被成功连接之后, 统一变量索引值将保持不变, 直到下次连接命令发生。如果一个连接成功, 那么统一变量的位置和价值就只能在这个连接之后进行查询了。

#### 错误

如果 `program` 不是由 OpenGL 产生的值, 则产生 `GL_INVALID_VALUE` 错误。

如果 `program` 不是一个程序对象, 则产生 `GL_INVALID_OPERATION` 错误。

如果 `program` 没有成功进行连接, 则产生 `GL_INVALID_OPERATION` 错误。

#### 相关 Get 函数

`glGetActiveUniform`, 自变量为 `program` 和一个活动统一变量的索引。

`glGetProgram`, 自变量为 `program`, 以及 `GL_ACTIVE_UNIFORMS` 或 `GL_ACTIVE_UNIFORM_MAX_LENGTH`。

`glGetUniform`, 自变量为 `program` 和一个统一变量的名称。

`glIsProgram`

#### 另外查看

`glLinkProgram`, `glUniform`

#### 版权

Copyright © 2003–2005 3Dlabs Inc. Ltd. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。 <http://opencontent.org/openpub/>。

## glGetVertexAttrib

返回一个通用顶点属性参数。

### C 规范

```
void glGetVertexAttribdv(GLuint index,
                        GLenum pname,
                        GLdouble * params);
void glGetVertexAttribfv(GLuint index,
                        GLenum pname,
                        GLfloat * params);
void glGetVertexAttribiv(GLuint index,
                        GLenum pname,
                        GLint * params);
void glGetVertexAttribIiv(GLuint index,
                        GLenum pname,
                        GLint * params);
void glGetVertexAttribIuiv(GLuint index,
                        GLenum pname,
                        GLuint * params);
```

#### 参数

`index`

指定将要被查询的通用顶点属性参数。

`pname`

指定将要被查询的通用顶点属性参数的符号名。可接受的值为 `GL_VERTEX_ATTRIB_ARRAY_BUFFER_BINDING`、`GL_VERTEX_ATTRIB_ARRAY_ENABLED`、`GL_VERTEX_ATTRIB_ARRAY_SIZE`、`GL_VERTEX_ATTRIB_ARRAY_STRIDE`、`GL_VERTEX_ATTRIB_ARRAY_TYPE`、`GL_VERTEX_ATTRIB_ARRAY_NORMALIZED`、`GL_VERTEX_ATTRIB_ARRAY_INTEGER`、`GL_VERTEX_ATTRIB_ARRAY_DIVISOR` 或 `GL_CURRENT_VERTEX_ATTRIB`。

`params`

返回所需数据。

#### 描述

`glGetVertexAttrib` 将通用顶点属性参数的值返回到 `params`。将要被查询的通用顶点属性由 `index` 指定, 而将要被查询的参数则由 `pname` 指定。

可以接受的参数名如下。

`params` 返回一个单个值, 即当前被绑定到与通用顶点属性数组 `index` 一致的绑定点的缓冲区对象的名称。如果没有缓冲区对象被绑定, 则返回 0。初始值为 0。

`GL_VERTEX_ATTRIB_ARRAY_ENABLED`

`params` 返回一个单个值, 如果 `index` 的顶点属性数组被激活, 那么这个值为非 0 (true); 如果 `index` 的顶点属性数组被禁止, 那么这个值为 0 (false)。初始值为 `GL_FALSE`。

`GL_VERTEX_ATTRIB_ARRAY_SIZE`

`params` 返回一个单个值, 即 `index` 的顶点属性数组的大小。这个大小是顶点属性数组的每个元素的值的数量, 它应该为 1、2、3 或 4。初始值为 4。

`GL_VERTEX_ATTRIB_ARRAY_STRIDE`

`params` 返回一个单个值, 即 `index` 的顶点属性数组的数组跨度 (连续元素之间的字节数)。值 0 表示数组元素按照时序存储在存储器中。初始值为 0。

`GL_VERTEX_ATTRIB_ARRAY_TYPE`

`pparams` 返回一个单个值, 即一个符号常量, 指示 `index` 的顶点属性数组的数组类型。可能的值有 `GL_BYTE`、`GL_UNSIGNED_BYTE`、`GL_SHORT`、`GL_UNSIGNED_SHORT`、`GL_INT`、`GL_UNSIGNED_INT`、`GL_FLOAT` 和 `GL_DOUBLE`。初始值为 `GL_FLOAT`。

`GL_VERTEX_ATTRIB_ARRAY_NORMALIZED`

`params` 返回一个单个值, 如果由 `index` 指示的顶点属性数组定点数据类型在转换到浮点数时被标准化, 那么这个值为非 0 (true); 在其他情况下, 这个值为 0 (false)。初始值为 `GL_FALSE`。

`GL_VERTEX_ATTRIB_ARRAY_INTEGER`

`params` 返回一个单个值, 如果由 `index` 指示的顶点属性数组定点数据类型包含整数数据类型, 那么这个值为非 0 (true); 在其他情况下, 这个值为 0 (false)。

初始值为 0 (`GL_FALSE`)。

`GL_VERTEX_ATTRIB_ARRAY_DIVISOR`

`params` 返回单个值, 即实例渲染使用的频率因数。

参见 `glVertexAttribDivisor`。初始值为 0。

`GL_CURRENT_VERTEX_ATTRIB`

`params` 返回 4 个值, 表示由 `index` 指定的通用顶点属性的当前值。其中通用顶点属性 0 是特殊的, 它没有当前状态, 所以如果 `index` 为 0 将会产生错误。所有其他通用顶点属性的初始值为 (0,0,0,1)。

除 `GL_CURRENT_VERTEX_ATTRIB` 以外, 所有参数都表示当前绑定的顶点数组对象中存储的端状态。

**注意**

如果有错误产生, 那么 `params` 的内容不会进行任何修改。

**错误**

如果 `pname` 不是 `GL_CURRENT_VERTEX_ATTRIB`, 并且当前没有绑定任何顶点数组对象, 则产生 `GL_INVALID_OPERATION` 错误。

如果 `index` 大于或等于 `GL_MAX_VERTEX_ATTRIBS`, 则产生 `GL_INVALID_VALUE` 错误。

如果 `pname` 不是一个可接受的值, 则产生 `GL_INVALID_ENUM` 错误。

如果 `index` 为 0, 并且 `pname` 为 `GL_CURRENT_VERTEX_ATTRIB`, 则产生 `GL_INVALID_OPERATION` 错误。

**相关 Get 函数**

`glGet`, 其自变量为 `GL_MAX_VERTEX_ATTRIBS`。

`glGetVertexAttribPointerv`, 其自变量为 `index` 和 `GL_VERTEX_ATTRIB_ARRAY_POINTER`。

**另外查看**

`glBindAttribLocation`, `glBindBuffer`, `glDisableVertexAttribArray`, `glEnableVertexAttribArray`, `glVertexAttrib`, `glVertexAttribDivisor`, `glVertexAttribPointer`

**版权**

Copyright © 2003–2005 3Dlabs Inc. Ltd. Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## `glGetVertexAttribPointerv`

返回指定通用顶点属性指针的地址。

**C 规范**

```
void glGetVertexAttribPointerv(GLuint index,
                               GLenum pname,
                               GLvoid ** pointer);
```

**参数**

index

指定将要返回的通用顶点属性参数。

pname

指定将要返回的通用顶点属性参数的符号名。必须为 GL\_VERTEX\_ATTRIB\_ARRAY\_POINTER。

pointer

返回指针值。

**描述**

glGetVertexAttribPointerv 返回指针信息。index 是将被查询的通用顶点属性，pname 是一个代表将要返回的指针的符号常量，而 params 则是一个指向用于放置返回数据的位置的指针。

返回的 pointer 是缓冲区对象数据存储的字节偏移，这个缓冲区对象是在期望指针被指定时绑定到 GL\_ARRAY\_BUFFER 目标（参见 glBindBuffer）的。

**注意**

返回的状态是从当前绑定的顶点数组对象获取的。

每个指针的初始值都为 0。

**错误**

如果当前未绑定任何顶点数组对象，则产生 GL\_INVALID\_OPERATION 错误。

如果 index 大于或等于 GL\_MAX\_VERTEX\_ATTRIBS，则产生 GL\_INVALID\_VALUE 错误。

如果 pname 不是一个可接受的值，则产生 GL\_INVALID\_ENUM 错误。

**相关 Get 函数**

glGet，其自变量为 GL\_MAX\_VERTEX\_ATTRIBS。

**另外查看**

glGetVertexAttrib，glVertexAttribPointer

**版权**

Copyright © 2003–2005 3Dlabs Inc. Ltd. Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

**glHint**

指定实现特有的提示（hint）。

**C 规范**

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

**参数**

target

指定一个符号常量，代表将要被控制的行为。

GL\_LINE\_SMOOTH\_HINT、GL\_POLYGON\_SMOOTH\_HINT、GL\_TEXTURE\_COMPRESSION\_HINT 和 GL\_FRAGMENT\_SHADER\_DERIVATIVE\_HINT 都可以被接受。

mode

指定一个符号常量，代表期望的行为。GL\_FASTEST、GL\_NICEST 和 GL\_DONT\_CARE 都可以被接受。

**描述**

如果存在解释的空间，那么 GL 行为的某些方面可以通过提示（hint）来控制。一个提示由两个自变量来指定。target 是一个符号常量，代表将要被控制的行为，而 mode 是另一个符号常量，代表期望的行为。每个 target 的初始值都为 GL\_DONT\_CARE。mode 可以为下列值其中之一。

GL\_FASTEST

应当选择效率最高的选项。

GL\_NICEST

应当选择最正确或质量最高的选项。

GL\_DONT\_CARE

没有优先选择。

虽然实现中可以被提示 (hint) 的特性是良好定义的, 对提示的解释还是要以实现为基础。和推荐的语义一样, 可以由 target 指定的提示特性如下。

GL\_FRAGMENT\_SHADER\_DERIVATIVE\_HINT

代表 GL 着色语言片段处理内建函数派生计算的精度: dFdx、dFdy 和 fwidth。

GL\_LINE\_SMOOTH\_HINT

代表抗锯齿线的采样质量。如果应用一个更大的滤波函数, 提示 GL\_NICEST 能够导致在光栅化时生成更多的像素片段。

GL\_POLYGON\_SMOOTH\_HINT

代表抗锯齿多边形的采样质量。如果应用一个更大的滤波函数, 提示 GL\_NICEST 能够导致在光栅化时生成更多的像素片段。

GL\_TEXTURE\_COMPRESSION\_HINT

代表压缩纹理图像的质量和性能。提示 GL\_FASTEST 表示纹理图像应该被尽可能快地进行压缩, 而 GL\_NICEST 则表示纹理图像应该以尽可能少的图像质量损失被压缩。如果纹理为了重用而将要被 glGetCompressedTexImage 检索, 那么应该选择 GL\_NICEST。

注意

对提示的解释要以实现为基础。某些实现会忽略 glHint 设置。

错误

如果 target 或 mode 不是可接受的值, 则产生 GL\_INVALID\_ENUM 错误。

版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glIsBuffer

确定一个名称是否与一个缓冲区对象一致。

### C 规范

```
GLboolean glIsBuffer(GLuint buffer);
```

参数

buffer

指定一个可能为缓冲区对象名称的值。

描述

如果 buffer 当前是一个缓冲区对象的名称, 那么 glIsBuffer 将返回 GL\_TRUE。如果 buffer 为 0, 或者为一个当前不是一个缓冲区对象名称的非 0 值, 又或者出现错误, glIsBuffer 将返回 GL\_FALSE。

一个由 glGenBuffers 返回, 但还没有通过调用 glBindBuffer 与缓冲区对象建立关联的名称, 不是一个缓冲区对象名称。

另外查看

glBindBuffer, glDeleteBuffers, glGenBuffers, glGet

版权

Copyright © 2005 Addison–Wesley. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。 <http://opencontent.org/openpub/>。

## glIsEnabled

测试一个功能是否被激活。

### C 规范

```
GLboolean glIsEnabled(GLenum cap);
```

**参数**

cap

指定一个符号常量来指示一种 GL 性能。

**描述**

如果 cap 是一个激活的功能, 那么 glIsEnabled 将返回 GL\_TRUE, 否则返回 GL\_FALSE。  
在初始情况下除了 GL\_DITHER 之外的所有功能将被禁止; GL\_DITHER 在初始状态下是激活的。  
下列功能都能被 cap 所接受。

**注意**

如果有错误产生, glIsEnabled 返回 GL\_FALSE。

**错误**

如果 cap 不是一个可接受的值, 则产生 GL\_INVALID\_ENUM 错误。

**另外查看**

glEnable, glGet

**版权**

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

常 量	查 看
GL_BLEND	glBlendFunc, glLogicOp
GL_CLIP_DISTANCEi	glEnable
GL_COLOR_LOGIC_OP	glLogicOp
GL_CULL_FACE	glCullFace
GL_DEPTH_CLAMP	glEnable
GL_DEPTH_TEST	glDepthFunc, glDepthRange
GL_DITHER	glEnable
GL_FRAMEBUFFER_SRGB	glEnable
GL_LINE_SMOOTH	glLineWidth
GL_MULTISAMPLE	glSampleCoverage
GL_POLYGON_SMOOTH	glPolygonMode
GL_POLYGON_OFFSET_FILL	glPolygonOffset
GL_POLYGON_OFFSET_LINE	glPolygonOffset
GL_POLYGON_OFFSET_POINT	glPolygonOffset
GL_PROGRAM_POINT_SIZE	glEnable
GL_PRIMITIVE_RESTART	glEnable, glPrimitiveRestartIndex
GL_SAMPLE_ALPHA_TO_COVERAGE	glSampleCoverage
GL_SAMPLE_ALPHA_TO_ONE	glSampleCoverage
GL_SAMPLE_COVERAGE	glSampleCoverage
GL_SAMPLE_MASK	glEnable
GL_SCISSOR_TEST	glScissor
GL_STENCIL_TEST	glStencilFunc, glStencilOp
GL_TEXTURE_CUBEMAP_SEAMLESS	glEnable

## glIsFramebuffer

确定一个名称是否与一个帧缓冲区对象一致。

### C 规范

```
GLboolean glIsFramebuffer(GLuint framebuffer);
```

### 参数

framebuffer

指定一个可能为帧缓冲区对象名称的值。

### 描述

如果 framebuffer 当前是一个帧缓冲区对象的名称, 那么 glIsFramebuffer 将返回 GL\_TRUE。如果 framebuffer 为 0, 或者为一个当前不是一个帧缓冲区对象名称的值, 又或者出现错误, glIsFramebuffer 将返回 GL\_FALSE。如果 framebuffer 是 glGenFramebuffers 返回的一个名称, 并且这时还没有通过一个 glBindFramebuffer 调用来进行绑定, 那么这个名称就不是一个帧缓冲区对象, 并且 glIsFramebuffer 将返回 GL\_FALSE。

### 另外查看

glGenFramebuffers, glBindFramebuffer, glDeleteFramebuffers

### 版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glIsProgram

确定一个名称是否与一个程序对象一致。

### C 规范

```
GLboolean glIsProgram(GLuint program);
```

### 参数

program

指定一个潜在的程序对象。

### 描述

如果 program 是以前由 glCreateProgram 创建的一个程序对象的名称, 并且还没有由 glDeleteProgram 删除, 那么 glIsProgram 将返回 GL\_TRUE。如果 program 为 0 或一个不是程序对象名称的非 0 值, 或者出现错误, glIsProgram 将返回 GL\_FALSE。

### 注意

如果 program 不是一个有效的程序对象名, 则不产生任何错误。

一个由 glDeleteProgram 标记为删除但仍然作为当前渲染状态的一部分的程序对象, 仍然会被当作一个程序对象, 并且 glIsProgram 将返回 GL\_TRUE。

### 相关 Get 函数

glGet, 其自变量为 GL\_CURRENT\_PROGRAM。

glGetActiveAttrib, 其自变量为 program 和一个活动属性变量的索引。

glGetActiveUniform, 其自变量为 program 和一个活动统一变量的索引。

glGetAttachedShaders, 其自变量为 program。

glGetAttribLocation, 其自变量为 program 和一个属性变量的名称。

glGetProgram, 其自变量为 program 和将要查询的参数。

glGetProgramInfoLog, 其自变量为 program。

glGetUniform, 其自变量为 program 和一个统一变量的位置。

glGetUniformLocation, 其自变量为 program 和一个统一变量的名称。

### 另外查看

glAttachShader, glBindAttribLocation, glCreateProgram, glDeleteProgram, glDetachShader, glLinkProgram, glUniform, glUseProgram, glValidateProgram

## 版权

Copyright © 2003–2005 3Dlabs Inc. Ltd. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glIsQuery

确定一个名称是否与一个查询对象一致。

### C 规范

```
GLboolean glIsQuery(GLuint id);
```

### 参数

id

指定一个可能为查询对象名称的值。

### 描述

如果 id 当前是一个查询对象的名称, 那么 glIsQuery 将返回 GL\_TRUE。如果 id 为 0, 或者为一个当前不是一个查询对象名称的非 0 值, 又或者出现错误, glIsQuery 将返回 GL\_FALSE。

一个由 glGenQueries 返回, 但还没有通过调用 glBeginQuery 与查询对象建立关联的名称, 不是一个查询对象名称。

### 另外查看

glBeginQuery, glDeleteQueries, glEndQuery, glGenQueries

### 版权

Copyright © 2005 Addison–Wesley. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glIsRenderbuffer

确定一个名称是否与一个渲染缓冲区对象一致。

### C 规范

```
GLboolean glIsRenderbuffer(GLuint renderbuffer);
```

### 参数

renderbuffer

指定一个可能为渲染缓冲区对象名称的值。

### 描述

如果 renderbuffer 当前是一个渲染缓冲区对象的名称, 那么 glIsRenderbuffer 将返回 GL\_TRUE。如果 renderbuffer 为 0, 或者为一个当前不是一个渲染缓冲区对象名称的值, 又或者出现错误, glIsRenderbuffer 将返回 GL\_FALSE。如果 renderbuffer 是 glGenRenderbuffers 返回的一个名称, 但是这时还没有通过一个 glBindRenderbuffer 或 glFramebufferRenderbuffer 调用来进行绑定, 那么这个名称就不是一个渲染缓冲区对象, 并且 glIsRenderbuffer 将返回 GL\_FALSE。

### 另外查看

glGenRenderbuffers, glBindRenderbuffer, glFramebufferRenderbuffer, glDeleteRenderbuffers

### 版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glIsSampler

确定一个名称是否与一个采样对象一致。

### C 规范

```
GLboolean glIsSampler(GLuint id);
```

### 参数

id

指定一个可能为采样器对象名称的值。

#### 描述

如果 id 当前是一个采样器对象的名称, 那么 glIsSampler 将返回 GL\_TRUE。如果 id 为 0, 或者为一个当前不是一个查询对象名称的非 0 值, 又或者出现错误, glIsQuery 将返回 GL\_FALSE。

由 glGenSamplers 返回的名称, 就是采样器对象的名称。

#### 注意

glIsSampler 只在 3.3 或更高版本的 GL 中可用。

#### 另外查看

glGenSamplers, glBindSampler, glDeleteSamplers

#### 版权

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glIsShader

确定一个名称是否与一个着色器对象一致。

#### C 规范

```
GLboolean glIsShader(GLuint shader);
```

#### 参数

shader

指定一个潜在的着色器对象。

#### 描述

如果 shader 是以前由 glCreateShader 创建的一个程序对象的名称, 并且还没有由 glDeleteShader 删除, 那么 glIsShader 将返回 GL\_TRUE。如果 shader 为 0 或一个不是着色器对象名称的非 0 值, 或者出现错误, glIsShader 将返回 GL\_FALSE。

#### 注意

如果 shader 不是一个有效的着色器对象名, 则不产生任何错误。

一个由 glDeleteShader 标记为删除但仍然被绑定在一个程序对象上的着色器对象, 仍然会被当作一个着色器对象, 并且 glIsShader 将返回 GL\_TRUE。

#### 相关 Get 函数

glGetAttachedShaders, 其参数包含一个合法程序对象。

glGetShader, 其自变量为 shader 和将要查询的参数。

glGetShaderInfoLog, 其自变量为 object。

glGetShaderSource, 其自变量为 object。

#### 另外查看

glAttachShader, glCompileShader, glCreateShader, glDeleteShader, glDetachShader, glLinkProgram, glShaderSource

#### 版权

Copyright © 2003–2005 3Dlabs Inc. Ltd。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glIsSync

确定一个名称是否与一个同步对象一致。

#### C 规范

```
GLboolean glIsSync(GLsync sync);
```

#### 参数

sync

指定一个可能为同步对象名称的值。

#### 描述

如果 sync 当前是一个渲染同步对象的名称, 那么 glIsSync 将返回 GL\_TRUE。如果 sync 不是一个同步对象名称的值, 又或者出现错误, glIsSync 将返回 GL\_FALSE。请注意 0 不是一个同步对象的名称。

#### 注意

glIsSync 在 3.2 或更高版本的 GL 中可用。

#### 另外查看

glFenceSync, glWaitSync, glClientWaitSync, glDeleteSync

#### 版权

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glIsTexture

确定一个名称是否与一个纹理一致。

#### C 规范

```
GLboolean glIsTexture(GLuint texture);
```

#### 参数

texture

指定一个可能为纹理名称的值。

#### 描述

如果 texture 当前是一个纹理的名称, 那么 glIsTexture 将返回 GL\_TRUE。如果 texture 为 0, 或者为一个当前不是一个纹理名称的非 0 值, 又或者出现错误, glIsTexture 将返回 GL\_FALSE。

一个由 glGenTextures 返回, 但还没有通过调用 glBindTexture 与纹理建立关联的名称, 不是一个纹理的名称。

#### 另外查看

glBindTexture, glCopyTexImage1D, glCopyTexImage2D, glDeleteTextures, glGenTextures, glGet, glGetTexParameter, glTexImage1D, glTexImage2D, glTexImage3D, glGetTexParameter

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 http://oss.sgi.com/projects/FreeB/。

## glIsVertexArray

确定一个名称是否与一个顶点数组对象一致。

#### C 规范

```
GLboolean glIsVertexArray(GLuint array);
```

#### 参数

array

指定一个可能为顶点数组对象名称的值。

#### 描述

如果 array 当前是一个渲染缓冲区对象的名称, 那么 glIsVertexArray 将返回 GL\_TRUE。如果 renderbuffer 为 0, 或者 array 为一个当前不是一个渲染缓冲区对象名称的值, 又或者出现错误, glIsVertexArray 将返回 GL\_FALSE。如果 array 是 glGenVertexArrays 返回的一个名称, 并且这时还没有通过一个 glBindVertexArray 调用来进行绑定, 那么这个名称就不是一个顶点数组对象, 并且 glIsVertexArray 将返回 GL\_FALSE。

#### 另外查看

glGenVertexArrays, glBindVertexArray, glDeleteVertexArrays

#### 版权

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的

条款和条件进行发布。<http://opencontent.org/openpub/>。

## glLineWidth

指定光栅化线的宽度。

### C 规范

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

#### 参数

width

指定光栅化线的宽度。初始值为 1。

#### 描述

glLineWidth 指定锯齿线和反锯齿线的宽度。使用一个不为 1 的线宽度会有不同的影响，这取决于线反锯齿是否被激活。可以通过调用自变量为 GL\_LINE\_SMOOTH 的 glEnable 和 glDisable 来激活和禁止线反锯齿。初始状态下抗锯齿是关闭的。

如果线抗锯齿被禁止，那么实际宽度将通过把提供的宽度舍入到最接近的整数来决定。（如果舍入到 0 的话，那么线宽度将视为 1）。如果  $\lfloor px \rfloor \neq \lfloor py \rfloor$ ， $i$  个像素将被填入每个被光栅化的列中，其中  $i$  为 width 被舍入后的值。否则， $i$  个像素将被填入每个被光栅化的行中。

如果线抗锯齿被激活，光栅化将为每个与宽等于当前线段宽度、长等于线段实际长度并以数学线段为中心的矩形相交的像素正方形生成一个片段。每个片段的覆盖值是矩形区域和相应像素正方形的交集的窗口坐标区域。这个值将在最后的光栅化步骤中保存和使用。

在线反锯齿被激活时，并不是所有的宽度都能得到支持。如果一个不支持的宽度被请求，那么将使用与它最近的支持宽度。只有宽度 1 是保证支持的，其他宽度值则要视具体实现而定。同样，锯齿线宽度也有一个范围。要查询这个范围内支持宽度之间的范围和大小区别可以调用自变量为 GL\_ALIASED\_LINE\_WIDTH\_RANGE、GL\_SMOOTH\_LINE\_WIDTH\_RANGE 和 GL\_SMOOTH\_LINE\_WIDTH\_GRANULARITY 的 glGet。

#### 注意

在 GL\_LINE\_WID 被查询时，由 glLineWidth 指定的线宽度总是被返回。

锯齿线和反锯齿线的截取和舍入对指定值不产生影响。

非抗锯齿线宽度可以被截取到一个平台相关的最大值。调用参数为 GL\_ALIASED\_LINE\_WIDTH\_RANGE 的 glGet 来确定最大宽度。

在 OpenGL 1.2 中，标记 GL\_LINE\_WIDTH\_RANGE 和 GL\_LINE\_WIDTH\_GRANULARITY 将由 GL\_ALIASED\_LINE\_WIDTH\_RANGE、GL\_SMOOTH\_LINE\_WIDTH\_RANGE 和 GL\_SMOOTH\_LINE\_WIDTH\_GRANULARITY 来代替。旧的名称将为了向下的兼容性而被保留，但在新的代码中不会使用。

#### 错误

如果 width 小于或等于 0，则产生 GL\_INVALID\_VALUE 错误。

#### 相关 Get 函数

glGet，其自变量为 GL\_LINE\_WIDTH。

glGet，其自变量为 GL\_ALIASED\_LINE\_WIDTH\_RANGE。

glGet，其自变量为 GL\_SMOOTH\_LINE\_WIDTH\_RANGE。

glGet，其自变量为 GL\_SMOOTH\_LINE\_WIDTH\_GRANULARITY。

glIsEnabled，其自变量为 GL\_LINE\_SMOOTH。

#### 另外查看

glEnable

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glLinkProgram

连接一个程序对象。

### C 规范

```
void glLinkProgram(GLuint program);
```

#### 参数

program

指定将要被连接的程序对象的句柄。

#### 描述

glLinkProgram 连接由 program 指定的程序对象。如果 GL\_VERTEX\_SHADER 类型的任何着色器对象被绑定到 program, 那么它们将被用来创建一个可执行文件, 这个可执行文件将在可编程顶点处理器中运行。如果 GL\_GEOMETRY\_SHADER 类型的任何着色器对象被绑定到 program, 那么它们将被用来创建一个可执行文件, 这个可执行文件将在可编程几何图形处理器中运行。如果 GL\_FRAGMENT\_SHADER 类型的任何着色器对象被绑定到 program, 那么它们将被用来创建一个可执行文件, 这个可执行文件将在可编程片段处理器中运行。

连接操作的状态将作为程序对象状态的一部分被存储。如果程序对象被无错误地连接并准备好使用, 这个值将被设为 GL\_TRUE, 否则将被设为 GL\_FALSE。它可以通过调用自变量为 program 和 GL\_LINK\_STATUS 的 glGetProgram 来进行查询。

成功的连接操作的一个结果是, 所有属于 program 的活动用户定义统一变量都将被初始化为 0, 并且程序对象的每一个活动统一变量都将分配一个位置, 这个位置可以通过调用 glGetUniformLocation 来进行查询。同时, 任何没有被绑定到通用顶点属性索引的活动的用户定义属性变量在这时都会被绑定到一个通用顶点属性索引上。

就像着色语言规范中指出的, 程序对象的连接可能由于很多原因而失败。下面列出了可以导致连接错误的一些情况。

- 超出了实现所支持的活动属性变量的数量。
- 超出了统一变量的存储限制。
- 超出了实现所支持的活动统一变量的数量。
- 顶点着色器、几何图形着色器或片段着色器的 main 函数丢失。
- 在片段着色器中实际使用的一个可变量没有在顶点着色器或几何图形着色器中以同样的方式进行声名 (或者根本就没有声名)。
- 一个到函数或变量名的引用没有被决定。
- 一个共享的全局符以两种不同的类型或两种不同的初始值被声明。
- 一个或多个附加着色器对象没有被成功进行编译。
- 绑定一个通用属性矩阵导致矩阵中的一些行超出了 GL\_MAX\_VERTEX\_ATTRIBS 的最大值。
- 找不到足够的连续顶点属性槽来绑定属性矩阵。
- 程序对象包含形成一个片段着色器的对象, 但是不包含形成一个顶点着色器的对象。
- 程序对象包含形成一个几何图形着色器的对象, 但是不包含形成一个顶点着色器的对象。
- 程序对象包含形成一个几何图形着色器的对象和输入图元类型、输出图元类型或者在任何编译的几何图形着色器对象中都没有指定的最大输出顶点计数。
- 程序对象包含形成一个几何图形着色器的对象和输入图元类型、输出图元类型或者在多个几何图形着色器对象中进行不同指定的最大输出顶点计数。
- 片段着色器中活动输出的数量大于 GL\_MAX\_DRAW\_BUFFERS 的值。
- 程序中有一个分配到一个大于或等于 GL\_MAX\_DUAL\_SOURCE\_DRAW\_BUFFERS 的值的活动输出, 并且有一个分配了大于或等于 1 的索引的活动输出。
- 有一个以上的 varying 输出变量被绑定到同一个数字或索引。
- 明确的绑定分配没留下足够的空间供连接器为一个 varying 输出数组自动分配一个位置, 这个数组需要多个连续的位置。
- 由 glTransformFeedbackVaryings 指定的 count 是非 0 的, 但是这个程序对象没有任何顶点着色器或几何图形着色器。
- 任何在 varyings 数组中指定到 glTransformFeedbackVaryings 的变量名都不会作为顶点着色器 (或几何图形着色器, 如果激活的话) 中的一个输出进行声明。
- varyings 数组中分配给 glTransformFeedbackVaryings 的任意两个入口都指定同一个 varying 变量。
- 任何变换反馈 varying 变量中捕获的分量总数都大于常量 GL\_MAX\_TRANSFORM\_FEEDBACK\_SEPARATE\_COMPONENTS, 缓冲区模式为 SEPARATE\_ATTRIBS。

当一个程序成功连接时, 可以通过调用 glUseProgram 将程序对象设为当前状态的一部分。无论连接操作是否成功, 程序对象的信息日志都会被覆盖。信息日志可以通过调用 glGetProgramInfoLog 获取。

如果连接操作成功, 并且指定的程序对象目前已经作为以前调用 glUseProgram 的结果而使用, glLinkProgram 也

将安装生成可执行文件来作为当前渲染状态的一部分。如果当前使用的程序对象重新连接不成功,那么它的连接状态将被设定为 `GL_FALSE`,但是可执行文件和相关状态将被保留为当前状态的一部分,直到以后调用 `glUseProgram` 来使它不再使用。在它移除不用之后,它将不能再作为当前状态的一部分,直到它被成功重连接为止。

如果 `program` 包含 `GL_VERTEX_SHADER` 类型的着色器对象,但不包含 `GL_FRAGMENT_SHADER` 类型的着色器对象,那么顶点着色器将被安装到可编程顶点处理器,可执行几何图形着色器(如果存在的话)将被安装到可编程几何图形处理器,但是不会再片段处理器上安装任何可执行程序。这样一个程序的光栅化图元的结果将为未定义的。

在进行连接操作时,程序对象的信息日志将被刷新,程序将生成。在连接操作之后,应用程序将可以修改绑定的着色器对象、编译绑定的着色器对象、分离着色器对象、删除着色器对象和绑定附加着色器对象。这些操作都不会影响作为程序对象一部分的日志或程序。

#### 注意

如果连接操作成功,那么任何关于以前在 `program` 上的连接操作的信息都将丢失(也就是说,一个失败的连接不会恢复 `program` 的原有数据)。即使是在不成功的连接操作之后,某些信息仍然可以从 `program` 中重新获得。作为例子,参见 `glGetActiveAttrib` 和 `glGetActiveUniform`。

#### 错误

如果 `program` 不是由 OpenGL 产生的值,则产生 `GL_INVALID_VALUE` 错误。

如果 `program` 不是一个程序对象,则产生 `GL_INVALID_OPERATION` 错误。

如果 `program` 是当前活动的程序对象,并且变换反馈模式激活,则产生 `GL_INVALID_OPERATION` 错误。

#### 相关 Get 函数

`glGet`, 其自变量为 `GL_CURRENT_PROGRAM`。

`glGetActiveAttrib`, 其自变量为 `program` 和一个活动属性变量的索引。

`glGetActiveUniform`, 其自变量为 `program` 和一个活动统一变量的索引。

`glGetAttachedShaders`, 其自变量为 `program`。

`glGetAttribLocation`, 其自变量为 `program` 和一个属性变量的名称。

`glGetProgram`, 其自变量为 `program` 和 `GL_LINK_STATUS`。

`glGetProgramInfoLog`, 其自变量为 `program`。

`glGetUniform`, 其自变量为 `program` 和一个统一变量的位置。

`glGetUniformLocation`, 其自变量为 `program` 和一个统一变量名称。

`glIsProgram`

#### 另外查看

`glAttachShader`, `glBindAttribLocation`, `glCompileShader`, `glCreateProgram`, `glDeleteProgram`, `glDetachShader`, `glUniform`, `glUseProgram`, `glValidateProgram`

#### 版权

Copyright © 2003–2005 3Dlabs Inc. Ltd. Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glLogicOp

为渲染指定一个逻辑像素操作。

### C 规范

```
void glLogicOp(GLenum opcode);
```

#### 参数

`opcode`

指定一个符号常量,这个符号常量用于选择一个逻辑操作。下列符号都可以接受: `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` 初始值为 `GL_COPY`。

#### 描述

`glLogicOp` 指定一个逻辑操作。如果被激活,这个逻辑操作将应用在在输入颜色索引或 RGBA 颜色与帧缓冲区中相应位置的颜色索引或 RGBA 颜色之间。可以调用使用符号常量 `GL_COLOR_LOGIC_OP` 的 `glEnable` 和 `glDisable`

来激活或禁止逻辑操作。初始值是禁止的。

opcode 是一个从列表中选择的符号常量。在逻辑操作的解释中，s 代表输入颜色，而 d 代表帧缓冲区中的索引。这里使用标准 C 语言操作。就像这些位操作符建议的，逻辑操作将单独应用到每个源和目的索引或颜色比特对（bit pair）。

注意

当而启用一个以上的 RGBA 颜色缓冲区进行绘制时，逻辑操作将单独为每个激活的缓冲区而执行，为目的值使用这个缓冲区内容的目的值（参见 glDrawBuffer）。

逻辑操作不会影响浮点绘制缓冲区。但是，如果 GL\_COLOR\_LOGIC\_OP 被激活，在这种情况下混合仍然是禁止的。

opcode	对应操作
GL_CLEAR	0
GL_SET	1
GL_COPY	s
GL_COPY_INVERTED	~s
GL_NOOP	d
GL_INVERT	~d
GL_AND	s & d
GL_NAND	~(s & d)
GL_OR	s   d
GL_NOR	~(s   d)
GL_XOR	s ^ d
GL_EQUIV	~(s ^ d)
GL_AND_REVERSE	s & ~d
GL_AND_INVERTED	~s & d
GL_OR_REVERSE	s   ~d
GL_OR_INVERTED	~s   d

错误

如果 opcode 不是一个可接受的值，则产生 GL\_INVALID\_ENUM 错误。

相关 Get 函数

glGet，其自变量为 GL\_LOGIC\_OP\_MODE。

GLsEnabled，其自变量为 GL\_COLOR\_LOGIC\_OP。

另外查看

glBlendFunc, glDrawBuffer, glEnable, glStencilOp

版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

glMapBuffer

映射一个缓冲区对象的数据存储。

C 规范

```
void * glMapBuffer(GLenum target,
                  GLenum access);
```

## 参数

### target

指定被映射的目标缓冲区对象。符号常量必须是 GL\_ARRAY\_BUFFER、GL\_COPY\_READ\_BUFFER、GL\_COPY\_WRITE\_BUFFER、GL\_ELEMENT\_ARRAY\_BUFFER、GL\_PIXEL\_PACK\_BUFFER、GL\_PIXEL\_UNPACK\_BUFFER、GL\_TEXTURE\_BUFFER、GL\_TRANSFORM\_FEEDBACK\_BUFFER 或 GL\_UNIFORM\_BUFFER。

### access

指定访问策略, 指示从缓冲区对象的数据存储中读取、向缓冲区对象的数据存储写入或同时读取或写入是否可能。符号常量必须为 GL\_READ\_ONLY、GL\_WRITE\_ONLY 或 GL\_READ\_WRITE。

## C 规范

```
GLboolean glUnmapBuffer(GLenum target);
```

## 参数

### target

指定被取消映射的目标缓冲区对象。符号常量必须是 GL\_ARRAY\_BUFFER、GL\_COPY\_READ\_BUFFER、GL\_COPY\_WRITE\_BUFFER、GL\_ELEMENT\_ARRAY\_BUFFER、GL\_PIXEL\_PACK\_BUFFER、GL\_PIXEL\_UNPACK\_BUFFER、GL\_TEXTURE\_BUFFER、GL\_TRANSFORM\_FEEDBACK\_BUFFER 或 GL\_UNIFORM\_BUFFER。

## 描述

glMapBuffer 将当前被绑定到 target 的缓冲区对象的整个数据存储映射到客户端的地址空间。然后, 根据指定的 access 策略, 这些数据就可以相对于返回的指针直接进行读取和/或写入了。如果 GL 不能被映射到缓冲区对象的数据存储, glMapBuffer 将产生一个错误, 并返回 NULL。这可能是由于可用虚拟内存不足等系统特定原因而发生的。

如果映射的数据存储以与指定 access 策略不一致的方式进行访问, 将不会产生错误, 但性能会受到负面影响, 并可能出现程序中止等系统错误。和 glBufferData 的 usage 参数不同, access 不是一个提示, 并且实际上会限制映射数据存储在一些 GL 实现上的使用。为了获得可用的最高性能, 一个缓冲区对象的数据存储应该以与它指定的 usage 和 access 参数一致的方式使用。

一个映射的数据存储必须在它的缓冲区对象被使用之前通过 glUnmapBuffer 来解除映射。

否则, 任何试图对缓冲区对象数据存储解除引用的 GL 命令都会产生错误。当一个数据存储解除映射时, 指向它的数据存储的指针将变成无效的。

glUnmapBuffer 返回 GL\_TRUE, 除非数据存储的内容在数据存储进行映射时被破坏。这可能与影响图形内存可用性的系统特定原因而引起, 例如屏幕模式变更。在这种情况下, GL\_FALSE 将返回, 并且数据存储内容会成为未定义的。一个应用程序必须检测这种罕见的情况并重新初始化数据存储。

一个缓冲区对象的映射数据存储当缓冲区对象被删除或它的数据存储由 glBufferData 重新创建时将自动解除映射。

## 注意

如果有错误产生, glMapBuffer 将返回 NULL, 并且 glUnmapBuffer 将返回 GL\_FALSE。

传递到 GL 命令的参数值可能并非来自于返回的指针。这不会产生任何错误, 但结果将为未定义的, 并且可能在不同的 GL 实现之间区别很大。

## 错误

如果 target 不是 GL\_ARRAY\_BUFFER、GL\_COPY\_READ\_BUFFER、GL\_COPY\_WRITE\_BUFFER、GL\_ELEMENT\_ARRAY\_BUFFER、GL\_PIXEL\_PACK\_BUFFER、GL\_PIXEL\_UNPACK\_BUFFER、GL\_TEXTURE\_BUFFER、GL\_TRANSFORM\_FEEDBACK\_BUFFER 或 GL\_UNIFORM\_BUFFER, 则会产生 GL\_INVALID\_ENUM。

如果 access 不是 GL\_READ\_ONLY、GL\_WRITE\_ONLY 或 GL\_READ\_WRITE, 则产生 GL\_INVALID\_ENUM 错误。

如果 glMapBuffer 在 GL 不能对缓冲区对象的数据存储进行映射时执行, 则会产生 GL\_OUT\_OF\_MEMORY 错误。这可能由多种系统特定原因产生, 例如保留虚拟内存不足。

如果保留的缓冲区对象名称 0 被绑定到 target, 则产生 GL\_INVALID\_OPERATION 错误。

如果 glMapBuffer 为了一个数据存储已经被映射的缓冲区对象而执行, 则产生 GL\_INVALID\_OPERATION 错误。

如果 glUnmapBuffer 为了一个数据存储当前没有被映射的缓冲区对象而执行, 则产生 GL\_INVALID\_OPERATION 错误。

## 相关 Get 函数

glGetBufferPointerv, 其自变量为 GL\_BUFFER\_MAP\_POINTER。

glGetBufferParameter, 其自变量为 GL\_BUFFER\_MAPPED、GL\_BUFFER\_ACCESS 或 GL\_BUFFER\_USAGE。

## 另外查看

glBindBuffer, glBindBufferBase, glBindBufferRange, glBufferData, glBufferSubData, glDeleteBuffers

**版权**

Copyright © 2005 Addison-Wesley. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

**glMapBufferRange**

映射一个缓冲区对象的数据存储的一部分。

**C 规范**

```
void *glMapBufferRange(GLenum target,
                       GLintptr offset,
                       GLsizeiptr length,
                       GLbitfield access);
```

**参数**

target

指定目标缓冲区要绑定到的绑定点。

offset

指定要被映射的范围的缓冲区中的初始偏置。

length

指定被映射的范围的长度。

access

指定访问标志的一个组合，指示到这个范围的期望访问。

**描述**

glMapBufferRange 将缓冲区对象数据存储的全部或一部分映射到客户端地址空间中。target 指定缓冲区绑定的目标，必须是 GL\_ARRAY\_BUFFER、GL\_COPY\_READ\_BUFFER、GL\_COPY\_WRITE\_BUFFER、GL\_ELEMENT\_ARRAY\_BUFFER、GL\_PIXEL\_PACK\_BUFFER、GL\_PIXEL\_UNPACK\_BUFFER、GL\_TEXTURE\_BUFFER、GL\_TRANSFORM\_FEEDBACK\_BUFFER 或 GL\_UNIFORM\_BUFFER 中的一个。offset 和 length 表示被映射的缓冲区对象中的数据范围，以基本机器单元为单位。access 是一个位段包含标记，描述所需的映射。这些标记描述如下。

如果不出现任何错误，那么一旦这个缓冲区中所有挂起的操作完成，将会返回一个指向这个映射范围起始位置的指针，可以根据在 access 中设置的如下标记位，用来修改和/或查询缓冲区相应范围。

GL\_MAP\_READ\_BIT 表示返回的指针可以用来读取缓冲区对象数据。如果这个指针用来查询不包含这个标记的一个映射，则不会产生任何 GL 错误，但是结果将是未定义的，并且可能会出现系统错误（可能包括程序终止）。

GL\_MAP\_WRITE\_BIT 表示返回的指针可以用来修改缓冲区对象数据。如果这个指针用来修改不包含这个标记的一个映射，则不会产生任何 GL 错误，但是结果将是未定义的，并且可能会出现系统错误（可能包括程序终止）。

此外，后面 access 中的 optional 标记位可能用于修改映射。

GL\_MAP\_INVALIDATE\_RANGE\_BIT 表示指定范围中前面的内容可能被丢弃。这个范围中的数据是未定义的，除了后续的写入数据之外。如果后续 GL 操作访问未写入数据，则不会产生任何 GL 错误，但是结果将是未定义的，并且可能会出现系统错误（可能包括程序终止）。这个标记可能不会用在与 GL\_MAP\_READ\_BIT 的组合中。

GL\_MAP\_INVALIDATE\_BUFFER\_BIT 表示整个缓冲区中前面的内容可能被丢弃。整个缓冲区中的数据是未定义的，除了后续的写入数据之外。如果后续 GL 操作访问未写入数据，则不会产生任何 GL 错误，但是结果将是未定义的，并且可能会出现系统错误（可能包括程序终止）。这个标记可能不会用在与 GL\_MAP\_READ\_BIT 的组合中。

GL\_MAP\_FLUSH\_EXPLICIT\_BIT 表示映射的一个或多个不连续的子范围可以被修改。当这个标记被置位时，每个子范围的修改必须通过调用 glFlushMappedBufferRange 来显式地进行清理。如果映射的一个子范围被修改并且没有被清理，则不会产生 GL 错误，但是缓冲区中相应子范围中的数据将是未定义的。这个标记可能只会用在与 GL\_MAP\_WRITE\_BIT 的组合中。如果选择了这个选项，清理会被严格地限制在通过解除映射之前调用 glFlushMappedBufferRange 显式地指明的区域之内；如果没有选择这个选项，那么 glUnmapBuffer 在调用时将会自动清理整个映射范围。

GL\_MAP\_UNSYNCHRONIZED\_BIT 表示 GL 不应该在从 glMapBufferRange 返回之前试图对缓冲区中挂起的操作进行同步。如果挂起的源于或修改缓冲区的数据覆盖了映射区域，则不会产生 GL 错误，但是前面这些操作和后面的任何操作的结果都会为未定义的。

如果出现错误，glMapBufferRange 就会返回一个 NULL 指针。

**错误**

如果 `offset` 或 `length` 为负, 或者如果 `offset + length` 大于或等于 `GL_BUFFER_SIZE` 的值, 则产生 `GL_INVALID_VALUE` 错误。

如果 `access` 有任何一位设置为前面定义值之外的值, 则产生 `GL_INVALID_VALUE` 错误。

如果下列条件中的任何一个成立, 则产生 `GL_INVALID_OPERATION` 错误。

缓冲区已经为映射状态。

`GL_MAP_READ_BIT` 或 `GL_MAP_WRITE_BIT` 被置位。

`GL_MAP_READ_BIT` 被置位, 并且 `GL_MAP_INVALIDATE_RANGE_BIT`、`GL_MAP_INVALIDATE_BUFFER_BIT` 或 `GL_MAP_UNSYNCHRONIZED_BIT` 中的任何一个被置位。

`GL_MAP_FLUSH_EXPLICIT_BIT` 被置位, 并且 `GL_MAP_WRITE_BIT` 没有被置位。

如果 `glMapBufferRange` 失败则产生 `GL_OUT_OF_MEMORY` 错误, 因为无法获取映射的内存。

**另外查看**

`glMapBuffer`, `glFlushMappedBufferRange`, `glBindBuffer`

**版权**

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。 <http://opencontent.org/openpub/>。

**glMultiDrawArrays**

从数组数据渲染多组图元。

**C 规范**

```
void glMultiDrawArrays(GLenum mode,
                       GLint * first,
                       GLsizei * count,
                       GLsizei primcount);
```

**参数**

`mode`

指定将要渲染哪种类型的图元。符号常量 `GL_POINTS`、`GL_LINE_STRIP`、`GL_LINE_LOOP`、`GL_LINES`、`GL_LINE_STRIP_ADJACENCY`、`GL_LINES_ADJACENCY`、`GL_TRIANGLE_STRIP`、`GL_TRIANGLE_FAN`、`GL_TRIANGLES`、`GL_TRIANGLE_STRIP_ADJACENCY` 和 `GL_TRIANGLES_ADJACENCY` 都是可接受的。

`first`

指向一个由活动数组中的起始索引组成的数组。

`count`

指向一个由将要被渲染的索引数组组成的数组。

`primcount`

指定第一个和计数的大小。

**描述**

`glMultiDrawArrays` 通过进行很少的子例程调用来指定多组几何图元。我们可以预先指定单独的定点、法线和颜色数组, 并通过调用一次 `glMultiDrawArrays` 来使用它们构造一个图元序列, 而不是通过调用一个 GL 程序来传输每个单独的顶点、法线、纹理坐标、边缘标记或颜色。

`glMultiDrawArrays` 的行为与 `glDrawArrays` 相同, 除了 `primcount` 分离元素范围被指定之外。

用 `glMultiDrawArrays` 时, 他使用 `count` 时序元素来从每个激活的数组构造一个几何图元序列, 从元素 `first` 开始。`mode` 指定要构造哪种图元, 以及如何用这些数组元素来构造这些图元。

由 `glMultiDrawArrays` 修改的顶点属性在 `glMultiDrawArrays` 返回后会有一未指定值。没有被修改过的属性将被保持为良好定义的。

**注意**

`GL_LINE_STRIP_ADJACENCY`、`GL_LINES_ADJACENCY`、`GL_TRIANGLE_STRIP_ADJACENCY` 和 `GL_TRIANGLES_ADJACENCY` 只在 3.2 或更高版本的 GL 中可用。

**错误**

如果 `mode` 不是一个可接受的值, 则产生 `GL_INVALID_ENUM` 错误。

如果 `primcount` 为负值, 则产生 `GL_INVALID_VALUE` 错误。

如果非 0 缓冲区对象名称被绑定到一个激活的数组, 并且缓冲区对象的数据存储当前被映射, 则产生 `GL_INVALID_OPERATION` 错误。

另外查看

`glDrawElements`, `glDrawRangeElements`

版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glMultiDrawElements

通过指定数组数据元素来渲染多组图元。

### C 规范

```
void glMultiDrawElements(GLenum mode,
                        const GLsizei * count,
                        GLenum type, const
                        GLvoid ** indices,
                        GLsizei primcount);
```

#### 参数

`mode`

指定将要渲染哪种类型的图元。符号常量 `GL_POINTS`、`GL_LINE_STRIP`、`GL_LINE_LOOP`、`GL_LINES`、`GL_LINE_STRIP_ADJACENCY`、`GL_LINES_ADJACENCY`、`GL_TRIANGLE_STRIP`、`GL_TRIANGLE_FAN`、`GL_TRIANGLES`、`GL_TRIANGLE_STRIP_ADJACENCY` 和 `GL_TRIANGLES_ADJACENCY` 都是可接受的。

`count`

指向一个元素计数的数组。

`type`

指定 `indices` 中的值的类型, 必须是 `GL_UNSIGNED_BYTE`、`GL_UNSIGNED_SHORT` 和 `GL_UNSIGNED_INT` 中的一个。

`indices`

指定一个指向索引存储位置的指针。

`primcount`

指定 `count` 数组的大小。

#### 描述

`glMultiDrawElements` 通过进行很少的子例程调用来指定多组几何图元。我们可以预先指定单独的顶点、法线等数组, 并通过调用一次 `glMultiDrawElements` 来使用它们构造一个图元序列, 而不是通过调用一个 GL 函数来传输每个单独的顶点、法线、纹理坐标、边缘标记或颜色。

`glMultiDrawElements` 在操作上与 `glDrawElements` 相同, 除了 `primcount` 单独的元素列表被指定之外。

由 `glMultiDrawElements` 修改的顶点属性在 `glMultiDrawElements` 返回后会有一未指定值。没有被修改过的属性将保持它们以前的值。

#### 注意

`GL_LINE_STRIP_ADJACENCY`、`GL_LINES_ADJACENCY`、`GL_TRIANGLE_STRIP_ADJACENCY` 和 `GL_TRIANGLES_ADJACENCY` 只在 3.2 或更高版本的 GL 中可用。

#### 错误

如果 `mode` 不是一个可接受的值, 则产生 `GL_INVALID_ENUM` 错误。

如果 `primcount` 为负值, 则产生 `GL_INVALID_VALUE` 错误。

如果非 0 缓冲区对象名称被绑定到一个激活的数组或这个元素数组, 并且缓冲区对象的数据存储当前被映射, 则产生 `GL_INVALID_OPERATION` 错误。

另外查看

`glDrawArrays`, `glDrawRangeElements`

版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glMultiDrawElementsBaseVertex

通过指定数组数据元素和一个应用到每个索引的索引来渲染多组图元。

### C 规范

```
void glMultiDrawElementsBaseVertex(GLenum mode,
                                   const GLsizei *count,
                                   GLenum type,
                                   const GLvoid **indices,
                                   GLsizei primcount,
                                   GLint *basevertex);
```

#### 参数

mode

指定将要渲染哪种类型的图元。符号常量 GL\_POINTS、GL\_LINE\_STRIP、GL\_LINE\_LOOP、GL\_LINES、GL\_LINE\_STRIP\_ADJACENCY、GL\_LINES\_ADJACENCY、GL\_TRIANGLE\_STRIP、GL\_TRIANGLE\_FAN、GL\_TRIANGLES、GL\_TRIANGLE\_STRIP\_ADJACENCY 和 GL\_TRIANGLES\_ADJACENCY 都是可接受的。

count

指向一个元素计数的数组。

type

指定 indices 中的值的类型，必须是 GL\_UNSIGNED\_BYTE、GL\_UNSIGNED\_SHORT 和 GL\_UNSIGNED\_INT 中的一个。

indices

指定一个指向索引存储位置的指针。

primcount

指定 count 数组的大小。

basevertex

指定一个指向基本顶点存储位置的指针。

#### 描述

glMultiDrawElementsBaseVertex 的行为与 glDrawElementsBaseVertex 相同，除了 primcount 分离元素列表被指定之外。

它的效果与下列代码相同。

```
for (int i = 0; i < primcount; i++)
    if (count[i] > 0)
        glDrawElementsBaseVertex(mode,
                                   count[i],
                                   type,
                                   indices[i],
                                   basevertex[i]);
```

#### 注意

glMultiDrawElementsBaseVertex 只在 3.1 或更高版本的 GL 中可用。

GL\_LINE\_STRIP\_ADJACENCY、GL\_LINES\_ADJACENCY、GL\_TRIANGLE\_STRIP\_ADJACENCY 和 GL\_TRIANGLES\_ADJACENCY 只在 3.2 或更高版本的 GL 中可用。

#### 错误

如果 mode 不是一个可接受的值，则产生 GL\_INVALID\_ENUM 错误。

如果 primcount 为负值，则产生 GL\_INVALID\_VALUE 错误。

如果非 0 缓冲区对象名称被绑定到一个激活的数组或这个元素数组，并且缓冲区对象的数据存储当前被映射，则产生 GL\_INVALID\_OPERATION 错误。

#### 另外查看

glMultiDrawElements, glDrawElementsBaseVertex, glDrawArrays, glVertexAttribPointer

**版权**

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

**glMultiTexCoord**

设置当前纹理坐标。

**C 规范**

```
void glMultiTexCoord1s GLenum target GLshort s
void glMultiTexCoord1i GLenum target GLint s
void glMultiTexCoord1f GLenum target GLfloat s
void glMultiTexCoord1d GLenum target GLdouble s
void glMultiTexCoord2s GLenum target GLshort s GLshort t
void glMultiTexCoord2i GLenum target GLint s GLint t
void glMultiTexCoord2f GLenum target GLfloat s GLfloat t
void glMultiTexCoord2d GLenum target GLdouble s GLdouble t
void glMultiTexCoord3s GLenum target GLshort s GLshort t GLshort r
void glMultiTexCoord3i GLenum target GLint s GLint t GLint r
void glMultiTexCoord3f GLenum target GLfloat s GLfloat t GLfloat r
void glMultiTexCoord3d GLenum target GLdouble s GLdouble t GLdouble r
void glMultiTexCoord4s GLenum target GLshort s GLshort t GLshort r GLshort q
void glMultiTexCoord4i GLenum target GLint s GLint t GLint r GLint q
void glMultiTexCoord4f GLenum target GLfloat s GLfloat t GLfloat r GLfloat q
void glMultiTexCoord4d GLenum target GLdouble s GLdouble t GLdouble r GLdouble q
```

**参数**

target

指定坐标要被修改的纹理单元。纹理单元的数量是实现相关的，但必须至少为 2。符号常量必须为 GL\_TEXTUREi 中的一个，其中 i 的取值范围为从 0 到 GL\_MAX\_TEXTURE\_COORDS - 1，这是一个与实现相关的值。

s、t、r、q 为 target 纹理单元指定 s、t、r 和 q 纹理坐标。并不是所有参数都会在所有形式的命令中出现。

**C 规范**

```
void glMultiTexCoord1sv GLenum target const GLshort * v
void glMultiTexCoord1iv GLenum target const GLint * v
void glMultiTexCoord1fv GLenum target const GLfloat * v
void glMultiTexCoord1dv GLenum target const GLdouble * v
void glMultiTexCoord2sv GLenum target const GLshort * v
void glMultiTexCoord2iv GLenum target const GLint * v
void glMultiTexCoord2fv GLenum target const GLfloat * v
void glMultiTexCoord2dv GLenum target const GLdouble * v
void glMultiTexCoord3sv GLenum target const GLshort * v
void glMultiTexCoord3iv GLenum target const GLint * v
void glMultiTexCoord3fv GLenum target const GLfloat * v
void glMultiTexCoord3dv GLenum target const GLdouble * v
void glMultiTexCoord4sv GLenum target const GLshort * v
void glMultiTexCoord4iv GLenum target const GLint * v
void glMultiTexCoord4fv GLenum target const GLfloat * v
void glMultiTexCoord4dv GLenum target const GLdouble * v
```

**参数**

target

指定坐标要被修改的纹理单元。纹理单元的数量是实现相关的，但必须至少为 2。符号常量必须为 GL\_TEXTUREi 中的一个，其中 i 的取值范围为从 0 到 GL\_MAX\_TEXTURE\_COORDS - 1，这是一个与实现相关的值。

v 指定一个指向一个包含 1 个、2 个、3 个或 4 个元素的数组，这些元素一次指定 s、t、r 和 q 纹理坐标。

**描述**

glMultiTexCoord 指定 1 维、2 维、3 维或 4 维纹理坐标。

glMultiTexCoord1 将当前纹理坐标设置为 (s, 0, 0, 1)；对 glMultiTexCoord2 的一次调用则将它们设置为 (s, t, 0, 1)。

同样, `glMultiTexCoord3` 将指定为  $(s, t, r, 1)$ , 而 `glMultiTexCoord4` 则将所有 4 个分量定义为  $(s, t, r, q)$ 。

当前纹理坐标是与每个顶点和每个当前光栅位置相关的数据的一部分。在初始情况下,  $(s, t, r, q)$  的值为  $(0, 0, 0, 1)$ 。

#### 注意

当前纹理坐标可以在任何时间更新。

`GL_TEXTUREi = GL_TEXTURE0 + i` 总是成立的。

#### 相关 Get 函数

`glGet`, 其自变量为 `GL_CURRENT_TEXTURE_COORDS`, 选择适当的纹理单元。

`glGet`, 其自变量为 `GL_MAX_TEXTURE_COORDS`。

#### 另外查看

`glActiveTexture`, `glTexCoord`, `glVertex`

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glPixelStore

设置像素存储模式。

### C 规范

```
void glPixelStoref(GLenum pname,
                  GLfloat param);
void glPixelStorei(GLenum pname,
                  GLint param);
```

### 参数

`pname`

指定将要进行设置的参数的符号名。共有 6 个值对将像素数据打包到内存有影响: `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`。另外 6 个数据影响像素数据从内存中解压缩: `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 `pname` 将被设置的值。

### 描述

`glPixelStore` 设置像素存储模式, 除了纹理模式解包 (参见 `glTexImage1D`、`glTexImage2D`、`glTexImage3D`、`glTexSubImage1D`、`glTexSubImage2D`、`glTexSubImage3D`) 之外, 这个存储模式还会影响后续的 `glReadPixels`。

`pname` 是一个符号常量, 指示将要被设置的参数, `param` 为新的值。12 个存储参数中的 6 个会影响像素数据如何被返回到客户端内存。这些参数如下所示。

#### GL\_PACK\_SWAP\_BYTES

如果为真, 多 byte 颜色分量、深度分量、颜色索引或模板索引的字节排序将被倒转。也就是说, 如果一个 4 字节分量包含  $b_0$ 、 $b_1$ 、 $b_2$ 、 $b_3$  四个 byte, 那么在 `GL_UNPACK_SWAP_BYTES` 为真时, 它将以  $b_3$ 、 $b_2$ 、 $b_1$ 、 $b_0$  的顺序从内存中获得。

`GL_UNPACK_SWAP_BYTES` 对一个像素中分量的内存顺序没有影响, 而只对分量中的字节或索引顺序产生影响。例如, 一个 `GL_RGB` 格式像素的 3 个分量总是以红色第一、绿色第二、蓝色第三的顺序存储, 无论 `GL_UNPACK_SWAP_BYTES` 为何值。

#### GL\_PACK\_LSB\_FIRST

如果为真, 字节中的 bit 将以从最小 (least significant) 到最大 (most significant) 的顺序排列。否则, 每个字节中的第一个位将为最大的一个。

#### GL\_PACK\_ROW\_LENGTH

如果大于 0 的话, `GL_PACK_ROW_LENGTH` 将定义一行像素数字。如果在这一行中的第一个像素被放置在内存中的位置  $p$ , 那么下一行的第一个像素的位置将通过跳过

$$k = \begin{cases} nl & s \geq a \\ \left\lfloor \frac{a}{s} \left\lceil \frac{snl}{a} \right\rceil \right\rfloor & s < a \end{cases}$$

个分量或索引来获得, 其中  $n$  为一个像素中分量或索引的数量,  $l$  为一行中像素的个数 (如果它大于 0 则为 `GL_PACK_ROW_LENGTH`, 否则为像素例程序的 `width` 自变量),  $a$  为 `GL_PACK_ALIGNMENT` 的值,  $s$  为单个分量的大小, 以字节为单位 (如果  $a < s$ , 则视为  $a = s$ )。对于 1bit 值的情况, 下一行的位置通过跳过

$$k = 8a \left\lceil \frac{nl}{8a} \right\rceil$$

个分量或索引来获得。

在这个描述中的语句 `component` 引用非索引 (nonindex) 值 `red`、`green`、`blue`、`alpha` 和 `depth`。举例来说, `GL_RGB` 存储格式的每个像素有 3 个分量,

第一个为红色, 第二个为绿色, 最后是蓝色。

`GL_PACK_IMAGE_HEIGHT`

如果大于 0 的话, `GL_PACK_IMAGE_HEIGHT` 将定义一个图像三维纹理体中的像素数量, 其中的 “image (图像)” 由所有使用同一个三维索引的像素定义。如果在这一行中的第一个像素被放置在内存中的位置  $p$ , 那么下一行的第一个像素的位置将通过跳过

$$k = \begin{cases} nlh & s \geq a \\ \left\lfloor \frac{a}{s} \left\lceil \frac{snlh}{a} \right\rceil \right\rfloor & s < a \end{cases}$$

个分量或索引来获得, 其中  $n$  为一个像素中分量或索引的数量,  $l$  为一行中像素的个数 (如果它大于 0 则为 `GL_PACK_ROW_LENGTH`, 否则为 `glTexImage3D` 例程序的 `width` 自变量),  $h$  为一个像素图像中的行数 (如果它大于 0 则为 `GL_PACK_IMAGE_HEIGHT`, 否则为 `glTexImage3D` 例程序的 `height` 自变量),  $a$  为 `GL_PACK_ALIGNMENT` 的值,  $s$  为单个分量的大小, 以字节为单位 (如果  $a < s$ , 则视为  $a = s$ )。

在这个描述中的语句 `component` 引用非索引 (nonindex) 值 `red`、`green`、`blue`、`alpha` 和 `depth`。举例来说, `GL_RGB` 存储格式的每个像素有 3 个分量,

第一个为红色, 第二个为绿色, 最后是蓝色。

`GL_PACK_SKIP_PIXELS`, `GL_PACK_SKIP_ROWS`, and `GL_PACK_SKIP_IMAGES`

这些值是为了方便而提供给程序员的。它们并没有提供通过简单地增加传递到 `glReadPixels` 的指针的方式所不能实现的功能。将 `GL_PACK_SKIP_PIXELS` 设为  $i$  就相当于将指针增加  $in$  个分量或索引, 其中  $n$  是每个像素中的分量或索引数量。将 `GL_PACK_SKIP_ROWS` 设为  $j$  就相当于将指针增加  $jm$  个分量或索引, 其中  $m$  是每行中的分量或索引数量, 就像刚刚在 `GL_PACK_ROW_LENGTH` 部分计算的一样。将 `GL_PACK_SKIP_IMAGES` 设为  $k$  就相当于将指针增加  $kp$  个分量或索引, 其中  $p$  是个图像中的分量或索引数量, 就像刚刚在 `GL_PACK_IMAGE_HEIGHT` 部分计算的一样。

`GL_PACK_ALIGNMENT`

指定内存中每个像素行起点的排列请求。允许值为 1 (byte 排列)、2 (排列为偶数 byte 的行)、4 (字 (word) 排列) 和 8 (行从双字边界开始)。

这 12 个存储参数中的其他 6 个影响像素数据如何从客户端内存中读取。

这些值对 `glTexImage1D`、`glTexImage2D`、`glTexImage3D`、`glTexSubImage1D`、`glTexSubImage2D` 和 `glTexSubImage3D` 都很重要。

这些值如下所示。

`GL_UNPACK_SWAP_BYTES`

如果为真, 多字节颜色分量、深度分量、颜色索引或模板索引的字节排序将被倒转。也就是说, 如果一个 4byte 分量包含  $b_0$ 、 $b_1$ 、 $b_2$ 、 $b_3$  四个 byte, 那么在 `GL_UNPACK_SWAP_BYTES` 为真时, 它将以  $b_3$ 、 $b_2$ 、 $b_1$ 、 $b_0$  的顺序从内存中获得。

`GL_UNPACK_SWAP_BYTES` 对一个像素中分量的内存顺序没有影响, 而只对分量中的字节或索引顺序产生影响。例如, 一个 `GL_RGB` 格式像素的 3 个分量总是以红色第一、绿色第二、蓝色第 3 的顺序存储, 无论 `GL_UNPACK_SWAP_BYTES` 为何值。

`GL_UNPACK_LSB_FIRST`

如果为真, 字节中的 bit 将以从最小 (least significant) 到最大 (most significant) 的顺序排列。否则, 每个字

节中的第一个位将为最大的一个。

GL\_UNPACK\_ROW\_LENGTH

如果大于 0 的话, GL\_UNPACK\_ROW\_LENGTH 将定义一行像素数字。如果在这一行中的第一个像素被放置在内存中的位置  $p$ , 那么下一行的第一个像素的位置将通过跳过

$$k = \begin{cases} nl & s \geq a \\ \left\lfloor \frac{a}{s} \left\lceil \frac{snl}{a} \right\rceil \right\rfloor & s < a \end{cases}$$

个分量或索引来获得, 其中  $n$  为一个像素中分量或索引的数量,  $l$  为一行中像素的个数 (如果它大于 0 则为 GL\_UNPACK\_ROW\_LENGTH, 否则为像素例行程序的 width 自变量),  $a$  为 GL\_UNPACK\_ALIGNMENT 的值,  $s$  为单个分量的大小, 以字节为单位 (如果  $a < s$ , 则视为  $a = s$ )。对于 1bit 值的情况, 下一行的位置通过跳过

$$k = 8a \left\lceil \frac{nl}{8a} \right\rceil$$

个分量或索引来获得。

在这个描述中的语句 component 引用非索引 (nonindex) 值 red、green、blue、alpha 和 depth。举例来说, GL\_RGB 存储格式的每个像素有 3 个分量,

第一个为红色, 第二个为绿色, 最后是蓝色。

GL\_UNPACK\_IMAGE\_HEIGHT

如果大于 0 的话, GL\_UNPACK\_IMAGE\_HEIGHT 将定义一个图像空间纹理体中的像素数量, 其中的 "image (图像)" 由所有使用同一个空间索引的像素定义。如果在这一行中的第一个像素被放置在内存中的位置  $p$ , 那么下一行的第一个像素的位置将通过跳过

$$k = \begin{cases} nlh & s \geq a \\ \left\lfloor \frac{a}{s} \left\lceil \frac{snlh}{a} \right\rceil \right\rfloor & s < a \end{cases}$$

个分量或索引来获得, 其中  $n$  为一个像素中分量或索引的数量,  $l$  为一行中像素的个数 (如果它大于 0 则为 GL\_UNPACK\_ROW\_LENGTH, 否则为像素例行程序的 width 自变量),  $h$  为一个像素图像中的行数 (如果它大于 0 则为 GL\_UNPACK\_IMAGE\_HEIGHT, 否则为像素例行程序的 height 自变量),  $a$  为 GL\_UNPACK\_ALIGNMENT 的值,  $s$  为单个分量的大小, 以字节为单位 (如果  $a < s$ , 则视为  $a = s$ )。

在这个描述中的语句 component 引用非索引 (nonindex) 值 red、green、blue、alpha 和 depth。举例来说, GL\_RGB 存储格式的每个像素有 3 个分量, 第一个为红色, 第二个为绿色, 最后是蓝色。

GL\_UNPACK\_SKIP\_PIXELS 和 GL\_UNPACK\_SKIP\_ROWS

这些值是为了方便而提供给程序员的。它们并没有提供通过增加传递到 glTexImage1D、glTexImage2D、glTexSubImage1D 或 glTexSubImage2D 的指针的方式所不能实现的功能。将 GL\_UNPACK\_SKIP\_PIXELS 设为  $i$  就相当于将指针增加  $in$  个分量或索引, 其中  $n$  是每个像素中的分量或索引数量。将 GL\_UNPACK\_SKIP\_ROWS 设为  $j$  就相当于将指针增加  $jm$  个分量或索引, 其中  $m$  是每行中的分量或索引数量, 就像刚刚在 GL\_UNPACK\_ROW\_LENGTH 部分计算的一样。

GL\_UNPACK\_ALIGNMENT

指定内存中每个像素行起点的排列请求。允许值为 1 (byte 排列)、2 (排列为偶数 byte 的行)、4 (字 (word) 排列) 和 8 (行从双字边界开始)。

下面的列表给出了每个能够通过 glPixelStore 进行设置的存储参数类型、初始值和有效值范围。

pname	类 型	初 始 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	$[0, \infty)$
GL_PACK_IMAGE_HEIGHT	integer	0	$[0, \infty)$
GL_PACK_SKIP_ROWS	integer	0	$[0, \infty)$
GL_PACK_SKIP_PIXELS	integer	0	$[0, \infty)$

续表

pname	类 型	初 始 值	有效范围
GL_PACK_SKIP_IMAGES	integer	0	[0, $\infty$ )
GL_PACK_ALIGNMENT	integer	4	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, $\infty$ )
GL_UNPACK_IMAGE_HEIGHT	integer	0	[0, $\infty$ )
GL_UNPACK_SKIP_ROWS	integer	0	[0, $\infty$ )
GL_UNPACK_SKIP_PIXELS	integer	0	[0, $\infty$ )
GL_UNPACK_SKIP_IMAGES	integer	0	[0, $\infty$ )
GL_UNPACK_ALIGNMENT	integer	4	1, 2, 4 或 8

glPixelStoref 可以用来设置任何像素存储参数。如果参数类型为布尔型, 那么在 param 为 0 时参数值为 false; 在其他情况下则为 true。如果 pname 是一个整型参数, 那么 param 将被舍入到最接近的整数。

同样, glPixelStorei 也可以被用来设置任何像素存储参数。布尔型在 param 为 0 时参数值为 false; 在其他情况下则为 true。

#### 错误

如果 pname 不是一个可接受的值, 则产生 GL\_INVALID\_ENUM 错误。

如果一个负的行长度、像素跳跃 (pixel skip) 或行跳跃 (row skip) 值被指定, 或者一个除 1、2、4 或 8 以外的其他排列被指定, 则产生 GL\_INVALID\_VALUE 错误。

#### 相关 Get 函数

glGet, 其自变量为 GL\_PACK\_SWAP\_BYTES。

glGet, 其自变量为 GL\_PACK\_LSB\_FIRST。

glGet, 其自变量为 GL\_PACK\_ROW\_LENGTH。

glGet, 其自变量为 GL\_PACK\_IMAGE\_HEIGHT。

glGet, 其自变量为 GL\_PACK\_SKIP\_ROWS。

glGet, 其自变量为 GL\_PACK\_SKIP\_PIXELS。

glGet, 其自变量为 GL\_PACK\_SKIP\_IMAGES。

glGet, 其自变量为 GL\_PACK\_ALIGNMENT。

glGet, 其自变量为 GL\_UNPACK\_SWAP\_BYTES。

glGet, 其自变量为 GL\_UNPACK\_LSB\_FIRST。

glGet, 其自变量为 GL\_UNPACK\_ROW\_LENGTH。

glGet, 其自变量为 GL\_UNPACK\_IMAGE\_HEIGHT。

glGet, 其自变量为 GL\_UNPACK\_SKIP\_ROWS。

glGet, 其自变量为 GL\_UNPACK\_SKIP\_PIXELS。

glGet, 其自变量为 GL\_UNPACK\_SKIP\_IMAGES。

glGet, 其自变量为 GL\_UNPACK\_ALIGNMENT。

#### 另外查看

glReadPixels, glTexImage1D, glTexImage2D, glTexImage3D, glTexSubImage1D, glTexSubImage2D, glTexSubImage3D

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glPointParameter

指定点参数。

### C 规范

```
void glPointParameterf(GLenum pname,
                       GLfloat param);
void glPointParameteri(GLenum pname,
                       GLint param);
```

### 参数

**pname**

指定一个单值点参数。GL\_POINT\_FADE\_THRESHOLD\_SIZE 和 GL\_POINT\_SPRITE\_COORD\_ORIGIN 都可以被接受。

**param**

指定 pname 将被设置的值。

### C 规范

```
void glPointParameterfv(GLenum pname,
                        const GLfloat * params);
void glPointParameteriv(GLenum pname,
                        const GLint * params);
```

### 参数

**pname**

指定一个点参数。GL\_POINT\_FADE\_THRESHOLD\_SIZE 和 GL\_POINT\_SPRITE\_COORD\_ORIGIN 都可以被接受。

**params**

指定要分配给 pname 的值。

### 描述

下列值都能被 pname 所接受。

**GL\_POINT\_FADE\_THRESHOLD\_SIZE**

params 是一个单个浮点值，它指定点大小在超出指定值时将被截取到的阈值。初始值为 1.0。

**GL\_POINT\_SPRITE\_COORD\_ORIGIN**

params 是一个单个枚举类型 (enum) 值，它指定点精灵纹理坐标原点，可以是 GL\_LOWER\_LEFT 或 GL\_UPPER\_LEFT。默认值为 GL\_UPPER\_LEFT。

### 错误

如果为 GL\_POINT\_FADE\_THRESHOLD\_SIZE 指定的值小于 0，则产生 GL\_INVALID\_VALUE 错误。

如果为 GL\_POINT\_SPRITE\_COORD\_ORIGIN 指定的值不是 GL\_LOWER\_LEFT 或 GL\_UPPER\_LEFT，则产生 GL\_INVALID\_ENUM 错误。

### 相关 Get 函数

glGet，其自变量为 GL\_POINT\_FADE\_THRESHOLD\_SIZE。

glGet，其自变量为 GL\_POINT\_SPRITE\_COORD\_ORIGIN。

### 另外查看

glPointSize

### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. Copyright © 2010 Khronos Group. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glPointSize

指定光栅化点的直径。

### C 规范

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

**参数**

size

指定光栅化点的直径。初始值为 1。

**描述**

glPointSize 指定点的光栅化直径。如果点大小模式被禁用 (参见以 GL\_PROGRAM\_POINT\_SIZE 为参数的 glEnable)，这个值将用于对点进行光栅化。

否则，将会使用写入到着色语言内建 glPointSize 变量的值。

**注意**

在 GL\_POINT\_SIZE 被查询时总是返回由 glPointSize 指定的点大小。

点的截取和舍入对指定值不产生影响。

**错误**

如果 size 小于或等于 0，则产生 GL\_INVALID\_VALUE 错误。

**相关 Get 函数**

glGet，其自变量为 GL\_POINT\_SIZE\_RANGE。

glGet，其自变量为 GL\_POINT\_SIZE\_GRANULARITY。

glGet，其自变量为 GL\_POINT\_SIZE。

glGet，其自变量为 GL\_POINT\_SIZE\_MIN。

glGet，其自变量为 GL\_POINT\_SIZE\_MAX。

glGet，其自变量为 GL\_POINT\_FADE\_THRESHOLD\_SIZE。

glIsEnabled，其自变量为 GL\_PROGRAM\_POINT\_SIZE。

**另外查看**

glEnable, glPointParameter

**版权**

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

**glPolygonMode**

选择一个多边形光栅化模式。

**C 规范**

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

**参数**

face

指定 mode 应用到的多边形。对于前向多边形和背向多边形，必须为 GL\_FRONT\_AND\_BACK。

mode

指定多边形将如何被光栅化。可接受的值为 GL\_POINT、GL\_LINE 和 GL\_FILL。对前向多边形和背向多边形，初始值均为 GL\_FILL。

**描述**

glPolygonMode 控制多边形光栅化的解释。face 描述应用哪个多边形 mode：前向多边形和背向多边形 (GL\_FRONT\_AND\_BACK)。多边形模式只影响多边形的最终光栅化。特别是，一个多边形的顶点被光照，并且多边形被裁减，而且可能在这些模式应用之前被筛选。

在 mode 中有 3 个模式被定义并能够被指定。

GL\_POINT

被标记为一个边界边缘起点的多边形顶点将作为点来绘制。

诸如 GL\_POINT\_SIZE 和 GL\_POINT\_SMOOTH 这样的点属性控制点的光栅化。除 GL\_POLYGON\_MODE 之外的多边形光栅化属性不会产生影响。

GL\_LINE

多边形的边界边缘将作为线段来绘制。诸如 GL\_LINE\_WIDTH 和 GL\_LINE\_SMOOTH 这样的线属性控制线的光

栅化。除 GL\_POLYGON\_MODE 之外的多边形光栅化属性不会产生影响。

GL\_FILL

多边形的内部被填充。诸如 GL\_POLYGON\_SMOOTH 这样的多边形属性控制多边形的光栅化。

示例

调用 glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_LINE);来绘制多边形轮廓的表面。

注意

顶点通过边缘标记来标示是否为有边界或无边界的。边缘标记最初在它分解三角形带和三角形扇时由 GL 内部生成。

错误

如果 face 或 mode 不是可接受的值, 则产生 GL\_INVALID\_ENUM 错误。

相关 Get 函数

glGet, 其自变量为 GL\_POLYGON\_MODE。

另外查看

glLineWidth, glPointSize

版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glPolygonOffset

设置计算深度值的比例和单位。

C 规范

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

参数

factor

指定一个用于为每个多边形创建一个深度偏移的缩放因子。初始值为 0。

units

将乘以一个实现指定的值来创建一个常数深度偏移。初始值为 0。

描述

当 GL\_POLYGON\_OFFSET\_FILL、GL\_POLYGON\_OFFSET\_LINE 或 GL\_POLYGON\_OFFSET\_POINT 被激活时, 每个片段的 depth 值都将在它被从适当顶点的 depth 值进行插值替换后进行偏移。偏移的值为  $\text{factor} * \text{DZ} + r * \text{units}$  其中 DZ 是关于多边形屏幕区域的深度变化的度量单位, 而 r 则是为一个给定实现生成一个可解析偏移所保证的最小值。偏移会在深度测试执行前和这个值被写入到深度缓冲区前添加。

glPolygonOffset 对于隐藏线图象渲染、在表面上应用贴花 (decals) 和加亮边缘固体渲染都非常有用。

注意

glPolygonOffset 对于放置在反馈缓冲区中的深度坐标不会产生影响。

glPolygonOffset 对选择不会产生影响。

相关 Get 函数

glIsEnabled, 其自变量为 GL\_POLYGON\_OFFSET\_FILL、GL\_POLYGON\_OFFSET\_LINE 或 GL\_POLYGON\_OFFSET\_POINT。

glGet, 其自变量为 GL\_POLYGON\_OFFSET\_FACTOR 或 GL\_POLYGON\_OFFSET\_UNITS。

另外查看

glDepthFunc, glEnable, glGet, glIsEnabled

版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glPrimitiveRestartIndex

指定图元重启索引。

### C 规范

```
void glPrimitiveRestartIndex(GLuint index);
```

### 参数

index

指定要解释为图元重启索引的值。

### 描述

glPrimitiveRestartIndex 指定一个在开启图元重启时特别对待的一个顶点数组。这就是我们所说的图元重启索引。

当一个 Draw\* 命令将一组通用属性数组元素传输到 GL 时, 如果与这一组相对应的顶点数组与图元重启索引相同, 那么 GL 就不会将这些元素作为一个顶点进行处理。取而代之的是, 就像绘制命令以前一个传输结束, 而另一个绘制命令立即以同样的参数开始一样, 但是只对紧接下来的元素直到最初指定元素的末尾进行传输。

在使用 glDrawElementsBaseVertex、glDrawElementsInstancedBaseVertex 或 glMultiDrawElementsBaseVertex 时, 图元重启比较会在基本顶点偏置被添加到数组索引之前发生。

### 注意

glPrimitiveRestartIndex 只在 3.1 或更高版本的 GL 中可用。

### 另外查看

glDrawArrays, glDrawElements, glDrawElementsBaseVertex, glDrawElementsInstancedBaseVertex

### 版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glProvokingVertex

指定要作为平面着色 varying 变量的数据源使用的顶点。

### C 规范

```
void glProvokingVertex(GLenum provokeMode);
```

### 参数

provokeMode

指定要作为平面着色 varying 变量的数据源使用的顶点。

### 描述

对一个顶点着色器 varying 输出变量进行平面着色 (Flatshading) 就是这个图元的所有顶点的这个输出分配同一个值。从中获取这些值的顶点就叫做 provoking vertex, 而 glProvokingVertex 则指定将哪个顶点作为平面着色 varying 变量源来使用。

provokeMode 必须为 GL\_FIRST\_VERTEX\_CONVENTION 或 GL\_LAST\_VERTEX\_CONVENTION, 并且控制将其值分配给平面着色 varying 变量输出的顶点。支持图元类型的这些值的插值是:

如果一个顶点或几何图形着色器是活动的, 那么用户定义的 varying 变量输出可能会在声明输出时使用 flat 限定符进行平面着色。

### 注意

glProvokingVertex 在 3.2 或更高版本的 GL 中可用。

### 错误

如果 provokeMode 不是一个可接受的值, 则产生 GL\_INVALID\_ENUM 错误。

### 版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

多边形 i 的图元类型	第一个顶点约定	最后一个顶点约定
point	i	i
independentline	$2i - 1$	$2i$
line loop	i	$i+1$ , if $i < n$ $1$ , if $i = n$
line strip	i	$i+1$
independent triangle	$3i - 2$	$3i$
triangle strip	i	$i+2$
triangle fan	$i+1$	$i+2$
line adjacency	$4i - 2$	$4i - 1$
line strip adjacency	$i+1$	$i+2$
triangle adjacency	$6i - 5$	$6i - 1$
triangle strip adjacency	$2i - 1$	$2i + 3$

## glQueryCounter

在前面所有的命令到达 GL 服务器之后, 但是未必已经执行时, 将 GL 时间记录到一个查询对象中。

### C 规范

```
void glQueryCounter(GLuint id,
                   GLenum target);
```

#### 参数

id

指定要记录 GL 时间的查询对象的名称。

target

指定将要查询的计数器。target 必须为 GL\_TIMESTAMP。

#### 描述

glQueryCounter 会使 GL 将当前时间记录到查询对象指定的 id。

target 必须为 GL\_TIMESTAMP。在 GL 客户端、服务器状态和帧缓冲区中前面所有命令完全实现之后, 这个时间将进行记录。当时间被记录时, 那个对象的查询结果将被标记为可用的。可以在一个 glBeginQuery / glEndQuery 块中使用 glQueryCounter 计时器查询, 这个块的目标为 GL\_TIME\_ELAPSED 并且不影响这个查询的结果。

#### 注意

glQueryCounter 只在 3.3 或更高版本的 GL 中可用。

#### 错误

如果 id 是一个已经在 glBeginQuery / glEndQuery 块中使用的查询对象的名称, 则产生 GL\_INVALID\_OPERATION 错误。

如果 id 不是以前的一个 glGenQueries 调用返回的查询对象的名称, 则产生 GL\_INVALID\_VALUE 错误。

如果 target 不是 GL\_TIMESTAMP, 则产生 GL\_INVALID\_ENUM 错误。

#### 另外查看

glGenQueries, glBeginQuery, glEndQuery, glDeleteQueries, glGetQueryObject, glGetQueryiv, glGet

#### 版权

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glReadBuffer

为像素选择一个颜色缓冲区源。

**C 规范**

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

**参数**

mode

指定一个颜色缓冲区。可以接受的值为 GL\_NONE、GL\_FRONT\_LEFT、GL\_FRONT\_RIGHT、GL\_BACK\_LEFT、GL\_BACK\_RIGHT、GL\_FRONT、GL\_BACK、GL\_LEFT 和 GL\_RIGHT。

**描述**

glReadBuffer 指定一个颜色缓冲区作为后续 glReadPixels、glCopyTexImage1D、glCopyTexImage2D、glCopyTexSubImage1D、glCopyTexSubImage2D 和 glCopyTexSubImage3D 命令的源。mode 接受 12 个或更多预定义值中的一个。在一个完全配置的系统中，GL\_FRONT、GL\_LEFT 和 GL\_FRONT\_LEFT 都指定左前缓冲区 (front left buffer)，GL\_FRONT\_RIGHT 和 GL\_RIGHT 指定右前缓冲区 (front right buffer)，而 GL\_BACK\_LEFT 和 GL\_BACK 则指定左后缓冲区 (back left buffer)。

非立体双缓冲区 (Nonstereo double-buffered) 配置只有一个左前缓冲区和一个左后缓冲区。单缓冲区配置有一个左前缓冲区和一个右前缓冲区。如果为非立体 (nonstereo)，则只有一个左前缓冲区。将一个不存在的缓冲区指定到 glReadBuffer 是一个错误。

在初始状态下，mode 在单缓冲区配置中为 GL\_FRONT，而在双缓冲区配置中则为 GL\_BACK。

**错误**

如果 mode 不是 12 个 (或者更多) 可接受的值中的一个，则产生 GL\_INVALID\_ENUM 错误。

如果 mode 指定一个不存在的缓冲区，则产生 GL\_INVALID\_OPERATION 错误。

**相关 Get 函数**

glGet，其自变量为 GL\_READ\_BUFFER。

**另外查看**

glCopyTexImage1D, glCopyTexImage2D, glCopyTexSubImage1D, glCopyTexSubImage2D, glCopyTexSubImage3D, glDrawBuffer, glReadPixels

**版权**

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

**glReadPixels**

从帧缓冲区中读取一个像素块 (block of pixels)。

**C 规范**

```
void glReadPixels(GLint x,
                  GLint y,
                  GLsizei width,
                  GLsizei height,
                  GLenum format,
                  GLenum type,
                  GLvoid * data);
```

**参数**

x

y

指定从帧缓冲区读取的第一个像素的窗口坐标。这个位置为像素矩形块的左下角。

width

height

指定像素矩形的尺寸。width 和 height 都为 1 则代表单个像素。

format

指定像素数据的格式。下列符号值都可以接受：

GL\_STENCIL\_INDEX、GL\_DEPTH\_COMPONENT、GL\_DEPTH\_STENCIL、GL\_RED、GL\_GREEN、GL\_BLUE、GL\_RGB、GL\_BGR、GL\_RGBA 和 GL\_BGRA。

type

指定像素数据的数据类型。必须为 GL\_UNSIGNED\_BYTE、GL\_BYTE、GL\_UNSIGNED\_SHORT、GL\_SHORT、GL\_UNSIGNED\_INT、GL\_INT、GL\_HALF\_FLOAT、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、GL\_UNSIGNED\_INT\_2\_10\_10\_10\_REV、GL\_UNSIGNED\_INT\_24\_8、GL\_UNSIGNED\_INT\_10F\_11F\_11F\_REV、GL\_UNSIGNED\_INT\_5\_9\_9\_9\_REV 和 GL\_FLOAT\_32\_UNSIGNED\_INT\_24\_8\_REV 中之一。

data

返回像素数据。

描述

glReadPixels 将像素数据从帧缓冲区中返回到从 data 位置开始的客户端内存中，从左下角位于 (x, y) 的像素开始。几个参数在像素数据被放置到客户端内存之前控制像素数据的处理。这些参数由 glPixelStore 命令进行设置。这个参考页描述由这些命令指定的大多数，但并非全部参数对 glReadPixels 的影响。

如果一个非 0 指定缓冲区对象在各像素块被请求时被绑定到 GL\_PIXEL\_PACK\_BUFFER 目标 (参见 glBindBuffer)，data 将被视作缓冲区对象数据存储的一个字节偏移，而不是一个指向客户端内存的指针。

glReadPixels 从每个左下角位于 (x + i, y + j)，其中  $0 \leq i < \text{width}$ 、 $0 \leq j < \text{height}$  的像素返回值。这个像素被称为第 j 行的第 i 个像素。像素将以从最低行到最高行的行顺序和每行中从左到右的顺序来返回。

format 指定返回像素值的格式，可接受的值如下。

GL\_STENCIL\_INDEX

模板索引将从模板缓冲区读取。每个索引都被转换为定点值，根据 GL\_INDEX\_SHIFT 的值和符号来进行左移或右移，并被加到 GL\_INDEX\_OFFSET 上。如果 GL\_MAP\_STENCIL 为 GL\_TRUE，索引将由它们在 GL\_PIXEL\_MAP\_S\_TO\_S 表中的映射来替换。

GL\_DEPTH\_COMPONENT

深度值将从深度缓冲区读取。个索引都被转换为浮点值，最小深度值映射到 0 而最大深度值映射到 1。

每个分量随后都会乘以 GL\_DEPTH\_SCALE，并被加到 GL\_DEPTH\_BIAS 上，并且最后被缩放到 [0,1] 范围。

GL\_DEPTH\_STENCIL

深度值将从深度缓冲区和模板缓冲区中读取。type 参数必须为 GL\_UNSIGNED\_INT\_24\_8 或 GL\_FLOAT\_32\_UNSIGNED\_INT\_24\_8\_REV。

GL\_RED

GL\_GREEN

GL\_BLUE

GL\_RGB

GL\_BGR

GL\_RGBA

GL\_BGRA

最后，这些索引或分量都会转换为正确的格式，由 type 指定。如果 format 为 GL\_STENCIL\_INDEX，并且 type 不为 GL\_FLOAT，那么每个索引都会由后面列表中给出的遮罩值进行遮罩。如果 type 为 GL\_FLOAT，那么每个整数索引都会被转换到单精度浮点格式。

如果 format 为 GL\_RED、GL\_GREEN、GL\_BLUE、GL\_RGB、GL\_BGR、GL\_RGBA 或 GL\_BGRA，并且 type 不为 GL\_FLOAT，那么每个索引都会乘以后面列表中给出的乘数。如果 type 为 GL\_FLOAT，那么每个分量都会原封不动地 (或者在它与其所使用的 GL 不同时，转换成客户端的单精度浮点格式) 进行传递。

这些值将按如下方式方式到内存中。如果 format 为 GL\_STENCIL\_INDEX、GL\_DEPTH\_COMPONENT、GL\_RED、GL\_GREEN 或 GL\_BLUE，那么将返回一个单个值，并且第 j 行第 i 个像素的数据将被放置到 (j)width + i 位置。

类 型	索引遮罩	分量变换
GL_UNSIGNED_BYTE	$2^8 - 1$	$(2^8 - 1) c$
GL_BYTE	$2^7 - 1$	$\frac{(2^8 - 1) c - 1}{2}$
GL_UNSIGNED_SHORT	$2^{16} - 1$	$(2^{16} - 1) c$

续表

类 型	索引遮罩	分量变换
GL_SHORT	$2^{15} - 1$	$\frac{(2^{15}-1)c-1}{2}$
GL_UNSIGNED_INT	$2^{32} - 1$	$(2^{32} - 1)c$
GL_INT	$2^{31} - 1$	$\frac{(2^{32}-1)c-1}{2}$
GL_HALF_FLOAT	none	c
GL_FLOAT	none	c
GL_UNSIGNED_BYTE_3_3_2	$2^8 - 1$	$(2^8 - 1)c$
GL_UNSIGNED_BYTE_2_3_3_REV	$2^8 - 1$	$(2^8 - 1)c$
GL_UNSIGNED_SHORT_5_6_5	$2^8 - 1$	$(2^8 - 1)c$
GL_UNSIGNED_SHORT_5_6_5_REV	$2^8 - 1$	$(2^8 - 1)c$
GL_UNSIGNED_SHORT_4_4_4_4	$2^8 - 1$	$(2^8 - 1)c$
GL_UNSIGNED_SHORT_4_4_4_4_REV	$2^8 - 1$	$(2^8 - 1)c$
GL_UNSIGNED_SHORT_5_5_5_1	$2^8 - 1$	$(2^8 - 1)c$
GL_UNSIGNED_SHORT_1_5_5_5_REV	$2^8 - 1$	$(2^8 - 1)c$
GL_UNSIGNED_INT_8_8_8_8	$2^8 - 1$	$(2^8 - 1)c$
GL_UNSIGNED_INT_8_8_8_8_REV	$2^8 - 1$	$(2^8 - 1)c$
GL_UNSIGNED_INT_10_10_10_2	$2^8 - 1$	$(2^8 - 1)c$
GL_UNSIGNED_INT_2_10_10_10_REV	$2^8 - 1$	$(2^8 - 1)c$
GL_UNSIGNED_INT_24_8	$2^8 - 1$	$(2^8 - 1)c$
GL_UNSIGNED_INT_10F_11F_11F_REV	-	Special
GL_UNSIGNED_INT_5_9_9_9_REV	-	Special
GL_FLOAT_T_32_UNSIGNED_INT_24_8_REV	none	c(Depth Only)

GL\_RGB 和 GL\_BGR 为每个像素返回 3 个值, GL\_RGBA 和 GL\_BGRA 为每个像素返回 4 个值, 所有对应一个单个像素的值都要在 data 中占据连续的空间。由 glPixelStore 设置的存储参数, 例如 GL\_PACK\_LSB\_FIRST 和 GL\_PACK\_SWAP\_BYTES, 将影响数据被写入内存的方式。

详细描述参见 glPixelStore。

#### 注意

位于与当前 GL 环境相联接的窗口之外的像素的值为未定义的。

如果有错误产生, 那么 data 的内容不会进行任何修改。

#### 错误

如果 format 或 type 不是一个可接受的值, 则产生 GL\_INVALID\_ENUM 错误。

如果 width 或者 height 为负值, 则产生 GL\_INVALID\_VALUE 错误。

如果 format 是 GL\_STENCIL\_INDEX, 并且不存在模板缓冲区, 则产生 GL\_INVALID\_OPERATION 错误。

如果 format 是 GL\_DEPTH\_INDEX, 并且不存在模板缓冲区, 则产生 GL\_INVALID\_OPERATION 错误。

如果 format 是 GL\_DEPTH\_STENCIL, 并且不存在深度缓冲区或者不存在模板缓冲区, 则产生 GL\_INVALID\_OPERATION 错误。

如果 format 是 GL\_DEPTH\_STENCIL, 并且 type 不是 GL\_UNSIGNED\_INT\_24\_8 或 GL\_FLOAT\_32\_UNSIGNED\_INT\_24\_8\_REV, 则产生 GL\_INVALID\_ENUM 错误。

如果 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 中的一个, 并且 format 不是 GL\_RGB, 则产生 GL\_INVALID\_

OPERATION 错误。

如果 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\_REVGL\_UNSIGNED\_INT\_5\_9\_9\_9\_REV 中的一个, 并且 format 既不是 GL\_RGBA 也不是 GL\_BGRA, 则产生 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_PACK\_BUFFER 目标, 并且缓冲区对象的数据存储当前被映射, 则产生 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_PACK\_BUFFER 目标, 并且数据将要被打包到缓冲区对象, 以致内存写入请求将超出数据存储大小, 则产生 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_PACK\_BUFFER 目标, 并且 data 没有平均分成将一个由 type 指定的数据存储到内存中所需的字节数, 则产生 GL\_INVALID\_OPERATION 错误。

如果 GL\_READ\_FRAMEBUFFER\_BINDING 是非 0 的, 读取帧缓冲区完成, 并且读取帧缓冲区的 GL\_SAMPLE\_BUFFERS 的值大于 0, 则产生 GL\_INVALID\_OPERATION 错误。

#### 相关 Get 函数

glGet, 其自变量为 GL\_INDEX\_SIZE。

glGet, 其自变量为 GL\_PIXEL\_PACK\_BUFFER\_BINDING。

#### 另外查看

glPixelStore, glReadBuffer

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glRenderbufferStorage

建立渲染缓冲区对象的图像的数据存储、格式和维度。

### C 规范

```
void glRenderbufferStorage(GLenum target,
                           GLenum internalformat,
                           GLsizei width,
                           GLsizei height);
```

#### 参数

target

指定位置目标要绑定到的绑定, 必须为 GL\_RENDERBUFFER。

internalformat

指定渲染缓冲区对象的图像所使用的内部格式。

width

指定渲染缓冲区的宽度, 以像素为单位。

height

指定渲染缓冲区的高度, 以像素为单位。

#### 描述

glRenderbufferStorage 等价于调用 samples 设为 0 的 glRenderbufferStorageMultisample。

target 指定的操作目标, 必须为 GL\_RENDERBUFFER。Internalformat 指定用于渲染缓冲区对象的存储的内部格式, 必须为颜色可渲染、深度可渲染或模板可渲染的格式。width 和 height 是渲染缓冲区的维度, 以像素为单位。width 和 height 必须都小于或等于 GL\_MAX\_RENDERBUFFER\_SIZE 的值。

如果成功, glRenderbufferStorage 会在调用 glRenderbufferStorage 变为未定义之后删除存在的任何渲染缓冲区图像数据存储和数据存储的内容。

#### 错误

如果 tarGet 不是 GL\_RENDERBUFFER, 则产生 GL\_INVALID\_ENUM 错误。

如果 width 或者 height 为负值, 或者大于 GL\_MAX\_RENDERBUFFER\_SIZE 的值, 则产生 GL\_INVALID\_VALUE

错误。

如果 internalformat 不是一个颜色可渲染、深度可渲染或模板可渲染的格式，则产生 GL\_INVALID\_ENUM 错误。

如果 GL 不能创建要求的 size 的数据存储，则产生 GL\_OUT\_OF\_MEMORY 错误。

另外查看

glGenRenderbuffers, glBindRenderbuffer, glRenderbufferStorageMultisample, glFramebufferRenderbuffer, glDeleteRenderbuffers

版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glRenderbufferStorageMultisample

建立渲染缓冲区对象的图像的数据存储、格式、维度和样本计数。

**C 规范**

```
void glRenderbufferStorageMultisample(GLenum target,
                                       GLsizei samples,
                                       GLenum internalformat,
                                       GLsizei width,
                                       GLsizei height);
```

**参数**

target

指定位置目标要绑定到的绑定，必须为 GL\_RENDERBUFFER。

samples

指定渲染缓冲区对象的存储所使用的样本数量。

internalformat

指定渲染缓冲区对象的图像所使用的内部格式。

width

指定渲染缓冲区的宽度，以像素为单位。

height

指定渲染缓冲区的高度，以像素为单位。

**描述**

glRenderbufferStorageMultisample 建立渲染缓冲区对象的图像的数据存储、格式、维度和样本数量。

target 指定的操作目标，必须为 GL\_RENDERBUFFER。Internalformat 指定用于渲染缓冲区对象的存储的内部格式，必须为颜色可渲染、深度可渲染或模板可渲染的格式。width 和 height 是渲染缓冲区的维度，以像素为单位。width 和 height 必须都小于或等于 GL\_MAX\_RENDERBUFFER\_SIZE 的值。samples 指定用于渲染缓冲区对象的存储的样本数量，必须小于或等于 GL\_MAX\_SAMPLES 的值。如果 internalformat 是一个有符号或无符号的整数格式，那么必须小于或等于 GL\_MAX\_INTEGER\_SAMPLES 的值。

如果成功，glRenderbufferStorageMultisample 会在调用 glRenderbufferStorageMultisample 变为未定义之后删除存在的任何渲染缓冲区图像数据存储和数据存储的内容。

**错误**

如果 target 不是 GL\_RENDERBUFFER，则产生 GL\_INVALID\_ENUM 错误。

如果 samples 大于 GL\_MAX\_SAMPLES，则产生 GL\_INVALID\_VALUE 错误。

如果 internalformat 不是一个颜色可渲染、深度可渲染或模板可渲染的格式，则产生 GL\_INVALID\_ENUM 错误。

如果 internalformat 是一个有符号或无符号的整数格式，并且 samples 大于或等于 GL\_MAX\_INTEGER\_SAMPLES 的值，则产生 GL\_INVALID\_OPERATION 错误。

如果 width 或者 height 为负值，或者大于 GL\_MAX\_RENDERBUFFER\_SIZE 的值，则产生 GL\_INVALID\_VALUE 错误。

如果 GL 不能创建要求的 size 的数据存储，则产生 GL\_OUT\_OF\_MEMORY 错误。

另外查看

glGenRenderbuffers, glBindRenderbuffer, glRenderbufferStorage, glFramebufferRenderbuffer,

glDeleteRenderbuffers

版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glSampleCoverage

指定多重采样覆盖参数。

C 规范

```
void glSampleCoverage(GLclampf value,
                      GLboolean invert);
```

参数

value

指定单个浮点采样覆盖值。这个值将被截取到[0,1]范围内。初始值为 1.0。

invert

指定单个布尔值，指示是否应该对覆盖遮罩取反。

GL\_TRUE 和 GL\_FALSE 都可以被接受。初始值为 GL\_FALSE。

描述

多采样对一个像素以不同的与具体实现相关的亚像素进行多次采样，以产生抗锯齿效果。如果被激活，则对抗锯齿点、线、多边形和图像进行透明度多重采样。

value 用于构建一个临时遮罩，这个遮罩用于确定在解析最终片段颜色时将使用哪些采样。这个遮罩将与从多采样计算中生成的覆盖遮罩进行按位与（bitwise-and）。如果 invert 标记被设置，那么临时遮罩将被转置（所有 bit 都被翻转），然后将进行按位与。

如果一个实现没有任何可用的多采样缓冲区，或者多采样被禁止，那么光栅化将在只有一个采样的情况下进行，计算一个像素的最终 RGB 颜色。

如果提供了一个得到实现支持的多采样缓冲区，并且多重采样被激活，那么一个像素的最终颜色将通过对每个像素综合几个采样而生成。每个采样包含颜色、深度和模板信息，允许这些操作在每个采样上执行。

相关 Get 函数

glGet，其自变量为 GL\_SAMPLE\_COVERAGE\_VALUE。

glGet，其自变量为 GL\_SAMPLE\_COVERAGE\_INVERT。

glIsEnabled，其自变量为 GL\_MULTISAMPLE。

glIsEnabled，其自变量为 GL\_SAMPLE\_ALPHA\_TO\_COVERAGE。

glIsEnabled，其自变量为 GL\_SAMPLE\_ALPHA\_TO\_ONE。

glIsEnabled，其自变量为 GL\_SAMPLE\_COVERAGE。

另外查看

glEnable

版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 http://oss.sgi.com/projects/FreeB/。

## glSampleMaski

设置采样遮罩的子字节值。

C 规范

```
void glSampleMaski(GLuint maskNumber,
                   GLbitfield mask);
```

参数

maskNumber

指定要更新采样遮罩子字节的哪个 32 位子字节。

mask

指定遮罩子字节的新值。

**描述**

glSampleMaski 设置多字节采样遮罩子字节的一个 32 位子字节 GL\_SAMPLE\_MASK\_VALUE。

maskIndex 指定要更新采样遮罩子字节的哪个 32 位子字节, 而 mask 则指定这个子字节要使用的新值。maskIndex 必须小于 GL\_MAX\_SAMPLE\_MASK\_WORDS 的值。遮罩第 M 字节的第 B 位与样本  $32 \times M + B$  对应。

**注意**

glSampleMaski 只在 3.2 或更高版本的 GL 中, 或者在支持 ARB\_texture\_multisample 扩展的情况下可用。

**错误**

如果 maskIndex 大于或等于 GL\_MAX\_SAMPLE\_MASK\_WORDS 的值, 则产生 GL\_INVALID\_VALUE 错误。

**另外查看**

glGenRenderbuffers, glBindRenderbuffer, glRenderbufferStorageMultisample, glFramebufferRenderbuffer, glDeleteRenderbuffers

**版权**

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glSamplerParameter

设置采样器参数。

**C 规范**

```
void glSamplerParameterf(GLuint sampler,
                        GLenum pname,
                        GLfloat param);
void glSamplerParameteri(GLuint sampler,
                        GLenum pname,
                        GLint param);
```

**参数**

sampler

指定其参数将要被修改的采样器对象。

pname

指定一个单值纹理参数的符号名。pname 可以为下列值之一: GL\_TEXTURE\_WRAP\_S、GL\_TEXTURE\_WRAP\_T、GL\_TEXTURE\_WRAP\_R、GL\_TEXTURE\_MIN\_FILTER、GL\_TEXTURE\_MAG\_FILTER、GL\_TEXTURE\_MIN\_LOD、GL\_TEXTURE\_MAX\_LOD、GL\_TEXTURE\_LOD\_BIAS、GL\_TEXTURE\_COMPARE\_MODE 或 GL\_TEXTURE\_COMPARE\_FUNC。

param

指定 pname 的值。

**C 规范**

```
void glSamplerParameterfv(GLuint sampler, GLenum pname, const GLfloat * params)
void glSamplerParameteriv(GLuint sampler, GLenum pname, const GLint * params)
```

**参数**

sampler

指定其参数将要被修改的采样器对象。

pname

指定一个纹理参数的符号名。pname 可以为下列值之一:

GL\_TEXTURE\_WRAP\_S、GL\_TEXTURE\_WRAP\_T、GL\_TEXTURE\_WRAP\_R、GL\_TEXTURE\_MIN\_FILTER、GL\_TEXTURE\_MAG\_FILTER、GL\_TEXTURE\_BORDER\_COLOR、GL\_TEXTURE\_MIN\_LOD、GL\_TEXTURE\_MAX\_LOD、GL\_TEXTURE\_LOD\_BIAS、GL\_TEXTURE\_COMPARE\_MODE 或 GL\_TEXTURE\_COMPARE\_FUNC。

params

指定一个指向的一个或多个 pname 值所存储的数组的指针。

## 描述

glSamplerParameter 将 pname 的一个或多个值分配给指定为 pname 的采样器参数。Sampler 指定将要修改的采样器对象，并且必须为一个采样器对象的名称，从以前的一个 glGenSamplers 调用返回。在 pname 中，下列符号都可以接受：

GL\_TEXTURE\_MIN\_FILTER

纹理缩小函数在进行纹理贴图的像素被映射到大于一个纹理元素的区域时使用。一共有 6 个定义的缩小函数。其中两个使用最近的一个或最近的 4 个纹理元素来计算纹理值。

其他 4 个使用 Mip 贴图。

Mip 贴图是一组排序的数组，代表同一个图像的一组分辨率逐渐降低的版本。如果图像的维度为  $2^n \times 2^m$ ，那么就有  $\max(n,m)+1$  个 Mip 贴图。

第一个 Mip 贴图是原始纹理，维度为  $2^n \times 2^m$ 。每个后续 Mip 贴图的纹理为  $2^{k-1} \times 2^{l-1}$ ，其中  $2^k \times 2^l$  是前一个 Mip 贴图的维度，直到  $k=0$  或  $l=0$  为止。这时，后续 Mip 贴图的维度为  $1 \times 2^{l-1}$  或  $2^{k-1} \times 1$ ，直到最后一个 Mip 贴图为止，其维度为  $1 \times 1$ 。可以调用以 level 参数指示 Mip 贴图顺序的 glTexImage1D、glTexImage2D、glTexImage3D、glCopyTexImage1D 或 glCopyTexImage2D 来定义 Mip 贴图。层次 0 是原始纹理，层次  $\max(n,m)$  是最后的  $1 \times 1$  Mip 贴图。

params 支持下列函数来缩小纹理。

GL\_NEAREST

返回距离进行纹理贴图的像素的中心最近（曼哈顿距离）的纹理元素的值。

GL\_LINEAR

返回距离进行纹理贴图的像素的中心最近的 4 个纹理元素的加权平均值。其中可以包含边缘纹理元素，根据 GL\_TEXTURE\_WRAP\_S 和 GL\_TEXTURE\_WRAP\_T 的值和确切的映射而定。

GL\_NEAREST\_MIPMAP\_NEAREST

选择最接近地匹配进行纹理贴图的像素大小的 Mip 贴图，并使用 GL\_NEAREST 标准（与像素中心最近的纹理元素）来生成一个纹理值。

GL\_LINEAR\_MIPMAP\_NEAREST

选择最接近地匹配进行纹理贴图的像素大小的 Mip 贴图，并使用 GL\_LINEAR 标准（4 个与像素中心最近的纹理元素的加权平均值）来生成一个纹理值。

GL\_NEAREST\_MIPMAP\_LINEAR

选择最接近地匹配进行纹理贴图的像素大小的两个 Mip 贴图，并使用 GL\_NEAREST 标准（与像素中心最近的纹理元素）来从每个 Mip 贴图生成一个纹理值。最终的纹理值是这两个值的一个加权平均值。

GL\_LINEAR\_MIPMAP\_LINEAR

选择最接近地匹配进行纹理贴图的像素大小两个的 Mip 贴图，并使用 GL\_LINEAR 标准（4 个与像素中心最近的纹理元素的加权平均值）来从每个 Mip 贴图生成一个纹理值。最终的纹理值是这两个值的一个加权平均值。

在缩小处理过程中采样的纹理元素越多，出现的锯齿假影就越少。当 GL\_NEAREST 和 GL\_LINEAR 缩小函数能够比其他 4 个更快时，它们只对 4 个纹理元素中的一个进行采样，以决定进行渲染的像素的纹理值，并且可以生成莫尔条纹（moire pattern）或不规则过渡（ragged transitions）。GL\_TEXTURE\_MIN\_FILTER 的初始值为 GL\_NEAREST\_MIPMAP\_LINEAR。

GL\_TEXTURE\_MAG\_FILTER

纹理放大函数在进行纹理贴图的像素被映射到小于或等于一个纹理元素的区域时使用。它会将纹理放大函数设置为 GL\_NEAREST 或 GL\_LINEAR（如下所示）。GL\_NEAREST 通常要比 GL\_LINEAR 快，但是它能够产生边缘更加锐利的纹理图像，因为纹理元素之间的过渡不那么平滑。GL\_TEXTURE\_MAG\_FILTER 的初始值为 GL\_LINEAR。

GL\_NEAREST

返回距离进行纹理贴图的像素的中心最近（曼哈顿距离）的纹理元素的值。

GL\_LINEAR

返回距离进行纹理贴图的像素的中心最近的 4 个纹理元素的加权平均值。其中可以包含边缘纹理元素，根据 GL\_TEXTURE\_WRAP\_S 和 GL\_TEXTURE\_WRAP\_T 的值和确切的映射而定。

GL\_TEXTURE\_MIN\_LOD

设置最小层次细节参数。这个浮点值限制了最高分辨率 Mip 贴图（最低层次的 Mip 贴图）的选择。初始值为 -1000。

GL\_TEXTURE\_MAX\_LOD

设置最大层次细节参数。这个浮点值限制了最低分辨率 Mip 贴图（最高层次的 Mip 贴图）的选择。初始值为 1000。

GL\_TEXTURE\_WRAP\_S

将纹理坐标环绕参数设置为 GL\_CLAMP\_TO\_EDGE、GL\_MIRRORED\_REPEAT 或 GL\_REPEAT。

GL\_CLAMP\_TO\_BORDER 会使坐标截取到 $[-\frac{1}{2}, +\frac{1}{2}]$ 范围内, 其中 N 是截取方向上纹理的大小。

GL\_CLAMP\_TO\_EDGE 会使坐标截取到 $[\frac{1}{2}, 1-\frac{1}{2}]$ 范围内, 其中 N 是截取方向上纹理的大小。

GL\_REPEAT 导致坐标的整数部分被忽略; GL 只使用小数部分, 从而创建一个重复模式。

GL\_MIRRORED\_REPEAT 导致在 s 的整数部分为偶数时坐标被设置为纹理坐标的小数部分, 而在这个整数部分为奇数时纹理坐标则被设置为  $1-\text{frac}(s)$ , 其中  $\text{frac}(s)$  代表 s 的小数部分。初始状态下, GL\_TEXTURE\_WRAP\_S 被设置为 GL\_REPEAT。

GL\_TEXTURE\_WRAP\_T

将纹理坐标环绕参数设置为 GL\_CLAMP\_TO\_EDGE、GL\_MIRRORED\_REPEAT 或 GL\_REPEAT。参见 GL\_TEXTURE\_WRAP\_S 相关讨论。

初始状态下, GL\_TEXTURE\_WRAP\_T 被设置为 GL\_REPEAT。

GL\_TEXTURE\_WRAP\_R

将纹理坐标环绕参数设置为 GL\_CLAMP\_TO\_EDGE、GL\_MIRRORED\_REPEAT 或 GL\_REPEAT。参见 GL\_TEXTURE\_WRAP\_S 相关讨论。

初始状态下, GL\_TEXTURE\_WRAP\_R 被设置为 GL\_REPEAT。

GL\_TEXTURE\_BORDER\_COLOR

Sets a border color. params 包含 4 个值, 即纹理边缘的 RGBA 颜色。整数颜色分量进行线性插值, 这样最大正整数被映射到 1.0, 而最小的负整数则被映射到 -1.0。这些值在指定时将被截取到 $[0, 1]$ 范围内。初始状态下, 边界颜色为(0, 0, 0, 0)。

GL\_TEXTURE\_COMPARE\_MODE

为当前绑定的纹理指定纹理比较模式。也就是一个内部格式为 GL\_DEPTH\_COMPONENT\_\*; (参见 glTexImage2D) 的纹理。允许的值为:

GL\_COMPARE\_REF\_TO\_TEXTURE

指定进行插值和截取的纹理坐标应该与当前绑定纹理的值进行比较。更多关于如何进行比较的细节参见关于 GL\_TEXTURE\_COMPARE\_FUNC 的讨论。比较的结果被分配到红色通道。

GL\_NONE

指定红色通道应该从当前绑定的纹理分配正确的值。

GL\_TEXTURE\_COMPARE\_FUNC

指定当 GL\_TEXTURE\_COMPARE\_MODE 设置为 GL\_COMPARE\_REF\_TO\_TEXTURE 时使用的比较操作符。

允许值为:

其中 r 为当前插值纹理坐标, 而 Dt 则为从当前绑定纹理中采样的纹理值。result 被分配给 Rt。

注意

glSamplerParameter 只在 3.3 或更高版本的 GL 中可用。

错误

如果 sampler 不是以前的一个 glGenSamplers 调用返回的采样器对象的名称, 则产生 GL\_INVALID\_VALUE 错误。

如果 params 应该有一个定义的常数值(基于 pname 的值)但实际却不是这样, 则产生 GL\_INVALID\_ENUM 错误。

纹理比较函数	计算结果
GL_EQUAL	$\text{Result} = \begin{cases} 1.0 & r \leq D_i \\ 0.0 & r > D_i \end{cases}$
GL_GEQUAL	$\text{Result} = \begin{cases} 1.0 & r \geq D_i \\ 0.0 & r < D_i \end{cases}$
GL_LESS	$\text{Result} = \begin{cases} 1.0 & r < D_i \\ 0.0 & r \geq D_i \end{cases}$
GL_GREATER	$\text{Result} = \begin{cases} 1.0 & r > D_i \\ 0.0 & r \leq D_i \end{cases}$
GL_EQUAL	$\text{Result} = \begin{cases} 1.0 & r = D_i \\ 0.0 & r \neq D_i \end{cases}$

续表

纹理比较函数	计算结果
GL_NOTEQUAL	$\text{Result} = \begin{cases} 1.0 & r \neq D_t \\ 0.0 & r = D_t \end{cases}$
GL_ALWAYS	result= 0.0
GL_NEVER	result= 0.0

**相关 Get 函数**

glGetSamplerParameter

**另外查看**

glGenSamplers, glBindSampler, glDeleteSamplers, glIsSampler, glBindTexture

**版权**

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

**glScissor**

定义裁剪框。

**C 规范**

```
void glScissor(GLint x,
               GLint y,
               GLsizei width,
               GLsizei height);
```

**参数**

x

y

指定裁剪框的左下角。初始值为 (0, 0)。

width

height

指定裁剪框的宽度和高度。当一个 GL 环境第一次被绑定到一个窗口时, width 和 height 将被设置为这个窗口的尺寸。

**描述**

glScissor 在窗口坐标中定义一个矩形, 这个矩形就叫做裁剪框。最前面的两个参数 x 和 y 指定裁剪框的左下角。width 和 height 指定裁剪框的宽度和高度。

可以通过调用自变量为 GL\_SCISSOR\_TEST 的 glEnable 和 glDisable 来激活和禁止裁剪测试。初始状态下, 模板测试是禁止的。如果这个测试被激活, 那么只有位于这个裁剪框中的像素能够通过绘制命令来进行修改。窗口坐标在帧缓冲区像素的共享角 (shared corner) 上有整数值。glScissor(0,0,1,1) 允许只在窗口左下角进行的修改, 而 glScissor(0,0,0,0) 则不允许对窗口中的任何像素进行修改。

当裁剪测试被禁止时, 就相当于裁剪框包含了整个窗口。

**错误**

如果 width 或者 height 为负值, 则产生 GL\_INVALID\_VALUE 错误。

**相关 Get 函数**

glGet, 其自变量为 GL\_SCISSOR\_BOX。

glIsEnabled, 其自变量为 GL\_SCISSOR\_TEST。

**另外查看**

glEnable, glViewport

**版权**

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见

<http://oss.sgi.com/projects/FreeB/>.

## glShaderSource

替换一个着色器对象中的源代码。

### C 规范

```
void glShaderSource(GLuint shader,
                   GLsizei count,
                   const GLchar **string,
                   const GLint *length);
```

### 参数

shader

指定源代码将被替换的着色器对象的句柄。

count

指定 string 和 length 数组中的元素数量。

string

指定一个指向包含将要被载入着色器的源代码的字符串的指针数组。

length

指定字符串长度数组。

### 描述

glShaderSource 将 shader 中的源代码设置设置为由 string 指定的字符串数组中的源代码。着色器对象中原来存储的任何源代码都会被完全替代。数组中的字符串数量由 count 指定。如果 length 为 NULL，那么每个字符串将被假定为以空结束符结束。如果 length 为除 NULL 以外的其他值，那么它将指向一个为每一个 string 的相应元素包含一个字符串长度的数组。length 数组中的每一个元素可以包含相应字符串的长度（空字符并不作为字符串长度的一部分来进行计数），或者一个小于 0 的值来指明字符串为以空结束符结束的。源代码字符串在这时不会被扫描或解析；它们将简单地被复制到指定的着色器对象。

### 注意

OpenGL 在 glShaderSource 被调用时复制着色器源代码字符串，所以一个应用程序可以在函数返回后立即释放它的副本。

### 错误

如果 shader 不是由 OpenGL 产生的值，则产生 GL\_INVALID\_VALUE 错误。

如果 shader 不是一个着色器对象，则产生 GL\_INVALID\_OPERATION 错误。

如果 count 小于 0，则产生 GL\_INVALID\_VALUE 错误。

### 相关 Get 函数

glGetShader，其自变量为 shader 和 GL\_SHADER\_SOURCE\_LENGTH。

glGetShaderSource，其自变量为 shader。

glIsShader

### 另外查看

glCompileShader, glCreateShader, glDeleteShader

### 版权

Copyright © 2003–2005 3Dlabs Inc. Ltd. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glStencilFunc

为模板测试设置前向和背向函数及引用值。

### C 规范

```
void glStencilFunc(GLenum func,
                  GLint ref,
                  GLuint mask);
```

**参数**

func

指定测试函数。共有 8 个标记可用, 分别是 GL\_NEVER、GL\_LESS、GL\_LEQUAL、GL\_GREATER、GL\_GEQUAL、GL\_EQUAL、GL\_NOTEQUAL 和 GL\_ALWAYS。初始值为 GL\_ALWAYS。

ref

为模板测试指定引用值。ref 将被限制到  $[0, 2^n - 1]$  范围, 其中  $n$  是模板缓冲区中位平面的数量。初始值为 0。

mask

指定一个遮罩, 这个遮罩在测试完成后与引用值和存储模板值都进行与运算。初始值都为 1。

**描述**

和深度缓冲一样, 模板化 (Stenciling) 在每个像素偏移上激活和禁止绘制。我们使用 GL 会址图元来在模板平面进行绘制, 然后对几何图形和图象进行渲染, 使用模板平面来遮蔽屏幕的一部分。典型情况下, 模板化在多道渲染 (multipass rendering) 运算中用来获得特殊效果, 这些效果有贴花 (decals)、轮廓 (outlining) 和结构立体几何渲染 (constructive solid geometry rendering) 等。

模板测试根据对引用值和模板缓冲区中的值进行比较的结果消除一个像素。可以通过调用自变量为 GL\_STENCIL\_TEST 的 glEnable 和 glDisable 来激活和禁止模板测试。要根据模板测试的结果来指定动作, 可以调用 glStencilOp 或 glStencilOpSeparate。

可以有两组单独的 func、ref 和 mask 参数。其中一组影响背向多边形, 而另外一组则影响前向多边形和其他非多边形图元。

glStencilFunc 将前向和背向模板状态设置为同一个值。使用 glStencilFuncSeparate 来将前向和背向模板状态设置为不同的值。

func 是一个符号常量, 它决定模板比较函数。它接受后面列出的 8 个值中的一个。ref 是一个整数引用值, 它将在模板比较中使用。它会被截取到  $[0, 2^n - 1]$  范围, 其中  $n$  为模板缓冲区中位平面的数量。mask 指定一个与引用值和存储模板值进行按位与的遮罩, 进行按位与的值将参与比较。

如果 stencil 表示存储在相应模板缓冲区位置的值, 下面的列表展示了每个能够由 func 指定的比较函数的影响。只有在比较成功时, 像素才会传递到下光栅化处理的一个阶段 (参见 glStencilOp)。所有测试都将 stencil 视为  $[0, 2^n - 1]$  范围内的无符号整数, 其中  $n$  为模板缓冲区中位平面的数量。

下列值都能被 func 所接受。

GL_NEVER	总是失败。
GL_LESS	当 $(ref \& mask) < (stencil \& mask)$ 时通过。
GL_LEQUAL	当 $(ref \& mask) \leq (stencil \& mask)$ 时通过。
GL_GREATER	当 $(ref \& mask) > (stencil \& mask)$ 时通过。
GL_GEQUAL	当 $(ref \& mask) \geq (stencil \& mask)$ 时通过。
GL_EQUAL	当 $(ref \& mask) = (stencil \& mask)$ 时通过。
GL_NOTEQUAL	当 $(ref \& mask) \neq (stencil \& mask)$ 时通过。
GL_ALWAYS	总是通过。

**注意**

在初始条件下, 模板测试是被禁用的。如果不存在模板缓冲区, 那么将不会出现模板修改, 这就好像模板测试总是通过一样。

glStencilFunc 与调用 face 被设置为 GL\_FRONT\_AND\_BACK 的 glStencilFuncSeparate 一样。

**错误**

如果 func 不是 8 个可接受值中的一个, 则产生 GL\_INVALID\_ENUM 错误。

**相关 Get 函数**

glGet 参数为 GL\_STENCIL\_FUNC、GL\_STENCIL\_VALUE\_MASK、GL\_STENCIL\_REF、GL\_STENCIL\_BACK\_FUNC、GL\_STENCIL\_BACK\_VALUE\_MASK、GL\_STENCIL\_BACK\_REF 或 GL\_STENCIL\_BITS。

glIsEnabled, 其自变量为 GL\_STENCIL\_TEST。

**另外查看**

glBlendFunc, glDepthFunc, glEnable, glLogicOp, glStencilFuncSeparate, glStencilMask, glStencilMaskSeparate, glStencilOp, glStencilOpSeparate

**版权**

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glStencilFuncSeparate

为模板测试设置前向和/或背向函数及引用值。

### C 规范

```
void glStencilFuncSeparate(GLenum face,
                           GLenum func,
                           GLint ref,
                           GLuint mask);
```

#### 参数

face

指定前向和/或背向模板状态是否被更新。共有 3 个标记可用，分别是 GL\_FRONT、GL\_BACK 和 GL\_FRONT\_AND\_BACK。

func

指定测试函数。共有 8 个标记可用，分别是 GL\_NEVER、GL\_LESS、GL\_EQUAL、GL\_GREATER、GL\_GEQUAL、GL\_NOTEQUAL 和 GL\_ALWAYS。初始值为 GL\_ALWAYS。

ref

为模板测试指定引用值。ref 将被限制到  $[0, 2^n - 1]$  范围，其中  $n$  是模板缓冲区中位平面的数量。初始值为 0。

mask

指定一个遮罩，这个遮罩在测试完成后与引用值和存储模板值都进行与运算。初始值都为 1。

#### 描述

和深度缓冲一样，模板化 (Stenciling) 在每个像素偏移上激活和禁止绘制。我们使用 GL 会址图元来在模板平面进行绘制，然后对几何图形和图象进行渲染，使用模板平面来遮蔽屏幕的一部分。典型情况下，模板化在多道渲染 (multipass rendering) 运算中用来获得特殊效果，这些效果有贴花 (decals)、轮廓 (outlining) 和结构立体几何渲染 (constructive solid geometry rendering) 等。

模板测试根据对引用值和模板缓冲区中的值进行比较的结果消除一个像素。可以通过调用自变量为 GL\_STENCIL\_TEST 的 glEnable 和 glDisable 来激活和禁止模板测试。要根据模板测试的结果来指定动作，可以调用 glStencilOp 或 glStencilOpSeparate。

可以有两组单独的 func、ref 和 mask 参数。其中一组影响背向多边形，而另外一组则影响前向多边形和其他非多边形图元。

glStencilFunc 将前向和背向模板状态设置为同一个值，就像调用了 face 被设置为 GL\_FRONT\_AND\_BACK 的 glStencilFuncSeparate 一样。

func 是一个符号常量，它决定模板比较函数。它接受后面列出的 8 个值中的一个。ref 是一个整数引用值，它将在模板比较中使用。它会被截取到  $[0, 2^n - 1]$  范围，其中  $n$  为模板缓冲区中位平面的数量。mask 指定一个与引用值和存储模板值进行按位与的遮罩，进行按位与的值将参与比较。

如果 stencil 表示存储在相应模板缓冲区位置的值，下面的列表展示了每个能够由 func 指定的比较函数的影响。只有在比较成功时，像素才会传递到下光栅化处理的一个阶段 (参见 glStencilOp)。所有测试都将 stencil 视为  $[0, 2^n - 1]$  范围内的无符号整数，其中  $n$  为模板缓冲区中位平面的数量。

下列值都能被 func 所接受。

GL_NEVER	总是失败。
GL_LESS	当 $(ref \& mask) < (stencil \& mask)$ 时通过。
GL_LEQUAL	当 $(ref \& mask) \leq (stencil \& mask)$ 时通过。
GL_GREATER	当 $(ref \& mask) > (stencil \& mask)$ 时通过。
GL_GEQUAL	当 $(ref \& mask) \geq (stencil \& mask)$ 时通过。
GL_EQUAL	当 $(ref \& mask) = (stencil \& mask)$ 时通过。
GL_NOTEQUAL	当 $(ref \& mask) \neq (stencil \& mask)$ 时通过。
GL_ALWAYS	总是通过。

#### 注意

在初始条件下, 模板测试是被禁用的。如果不存在模板缓冲区, 那么将不会出现模板修改, 这就好像模板测试总是通过一样。

#### 错误

如果 func 不是 8 个可接受值中的一个, 则产生 GL\_INVALID\_ENUM 错误。

#### 相关 Get 函数

glGet 参数为 GL\_STENCIL\_FUNC、GL\_STENCIL\_VALUE\_MASK、GL\_STENCIL\_REF、GL\_STENCIL\_BACK\_FUNC、GL\_STENCIL\_BACK\_VALUE\_MASK、GL\_STENCIL\_BACK\_REF 或 GL\_STENCIL\_BITS。

glIsEnabled, 其自变量为 GL\_STENCIL\_TEST。

#### 另外查看

glBlendFunc, glDepthFunc, glEnable, glLogicOp, glStencilFunc, glStencilMask, glStencilMaskSeparate, glStencilOp, glStencilOpSeparate

#### 版权

Copyright © 2006 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glStencilMask

控制模板平面中的独立位的前向和背向写入。

#### C 规范

```
void glStencilMask(GLuint mask);
```

#### 参数

mask

指定一个位遮罩来激活和禁止模板平面中的独立位的写入。

在初始情况下, 遮罩为全 1。

#### 描述

glStencilMask 模板平面中的独立位的前向和背向写入。的最低有效位 n 个位指定了一个遮罩, 其中 n 为模板缓冲区中的位的数量。如果遮罩中某个位置出现了一个 1, 那么将可以写入模板缓冲区中相应位。如果遮罩中某个位置出现了一个 0, 那么模板缓冲区中相应位将为写保护的。在初始情况下所有位都被激活写入。

可以有二个独立的 mask 写入遮罩 (writemasks); 其中一个影响背向多边形, 而另一个则影响前向多边形以及其他非多边形图元。glStencilMask 将前向和背向写入遮罩设置为同一个值。使用 glStencilMaskSeparate 来将前向和背向模板写入遮罩设置为不同的值。

#### 注意

glStencilMask 与调用 face 被设置为 GL\_FRONT\_AND\_BACK 的 glStencilMaskSeparate 一样。

#### 相关 Get 函数

glGet, 其自变量为 GL\_STENCIL\_WRITEMASK、GL\_STENCIL\_BACK\_WRITEMASK 或 GL\_STENCIL\_BITS。

#### 另外查看

glColorMask, glDepthMask, glStencilFunc, glStencilFuncSeparate, glStencilMaskSeparate, glStencilOp, glStencilOpSeparate

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 http://oss.sgi.com/projects/FreeB/。

## glStencilMaskSeparate

控制模板平面中的独立位的前向和/或背向写入。

#### C 规范

```
void glStencilMaskSeparate(GLenum face,
                           GLuint mask);
```

**参数****face**

指定前向和/或背向模板写入遮罩是否被更新。共有 3 个标记可用, 分别是 GL\_FRONT、GL\_BACK 和 GL\_FRONT\_AND\_BACK。

**mask**

指定一个位遮罩来激活和禁止模板平面中的独立位的写入。

在初始情况下, 遮罩为全 1。

**描述**

glStencilMaskSeparate 控制模板平面中的独立位的前向和背向写入。的最低有效位  $n$  个位指定了一个遮罩, 其中  $n$  为模板缓冲区中的位的数量。如果遮罩中某个位置出现了一个 1, 那么将可以写入模板缓冲区中相应位。如果遮罩中某个位置出现了一个 0, 那么模板缓冲区中相应位将为写保护的。在初始情况下所有位都被激活写入。

可以有二个独立的 mask 写入遮罩 (writemasks); 其中一个影响背向多边形, 而另一个则影响前向多边形以及其他非多边形图元。glStencilMask 将前向和背向写入遮罩设置为同一个值, 就像调用 face 被设置为 GL\_FRONT\_AND\_BACK 的 glStencilMaskSeparate 一样。

**错误**

如果 face 不是一个可接受的标记, 则产生 GL\_INVALID\_ENUM 错误。

**相关 Get 函数**

glGet, 其自变量为 GL\_STENCIL\_WRITEMASK、GL\_STENCIL\_BACK\_WRITEMASK 或 GL\_STENCIL\_BITS。

**另外查看**

glColorMask, glDepthMask, glStencilFunc, glStencilFuncSeparate, glStencilMask, glStencilOp, glStencilOpSeparate

**版权**

Copyright © 2006 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

**glStencilOp**

设置前向和背向模板测试动作。

**C 规范**

```
void glStencilOp(GLenum sfail,
                 GLenum dpfail,
                 GLenum dppass);
```

**参数****sfail**

指定在模板测试失败后采取的动作。共有 8 个标记可以接受: GL\_KEEP、GL\_ZERO、GL\_REPLACE、GL\_INCR、GL\_INCR\_WRAP、GL\_DECR、GL\_DECR\_WRAP 和 GL\_INVERT。初始值为 GL\_KEEP。

**dpfail**

指定在模板测试通过但深度测试失败后采取的模板动作。dpfail 接受和 sfail 相同的符号常量。初始值为 GL\_KEEP。

**dppass**

指定在模板测试都通过, 或在模板测试通过并且没有被激活的深度缓冲区或深度测试时采取的模板动作。

dppass 接受和 sfail 相同的符号常量。初始值为 GL\_KEEP。

**描述**

和深度缓冲一样, 模板化 (Stenciling) 在每个像素偏移上激活和禁止绘制。我们使用 GL 会址图元来在模板平面进行绘制, 然后对几何图形和图像进行渲染, 使用模板平面来遮蔽屏幕的一部分。典型情况下, 模板化在多道渲染 (multipass rendering) 运算中用来获得特殊效果, 这些效果有贴花 (decal)、轮廓 (outlining) 和结构立体几何渲染 (constructive solid geometry rendering) 等。

模板测试根据对引用值和模板缓冲区中的值进行比较的结果消除一个像素。可以通过调用自变量为 GL\_STENCIL\_TEST 的 glEnable 和 glDisable 来激活和禁止模板测试; 如果要控制它, 则可以调用 glStencilFunc 或 glStencilFuncSeparate。

可以有两组单独的 `sfail`、`dpfail` 和 `dppass` 参数。其中一组影响背向多边形，而另外一组则影响前向多边形和其他非多边形图元。

`glStencilOp` 将前向和背向模板状态设置为同一个值。使用 `glStencilOpSeparate` 来将前向和背向模板状态设置为不同的值。

`glStencilOp` 接受 3 个自变量，这 3 个自变量指示在模板化被激活时存储的模板值上将会发生什么。如果模板测试失败，那么像素的颜色或深度缓冲区将不会有任何改变，并且 `sfail` 指定模板缓冲区内容上将会发生什么。下列 8 个动作都可能发生。

<code>GL_KEEP</code>	保持当前值。
<code>GL_ZERO</code>	将模板缓冲区的值设为 0
<code>GL_REPLACE</code>	将模板缓冲区值设为 <code>ref</code> ，正如 <code>glStencilFunc</code> 所指定的。
<code>GL_INCR</code>	增加当前模板缓冲区值。截取到可表示的最大无符号值。
<code>GL_INCR_WRAP</code>	增加当前模板缓冲区值。在增加可表示的最大无符号值时将模板缓冲区值环绕 (wrap) 到 0。
<code>GL_DECR</code>	增加当前模板缓冲区值。截取到 0。
<code>GL_DECR_WRAP</code>	增加当前模板缓冲区值。在增加模板缓冲区值 0 时将模板缓冲区值环绕 (wrap) 到可表示的最大无符号值。
<code>GL_INVERT</code>	对模板缓冲区值按位取反 (bitwise invert)。

模板缓冲区值将被视为无符号整数。在增加和减少时，这些值被限制 (wrap) 到 0 和  $2^n-1$ ，其中  $2^n-1$  为查询 `GL_STENCIL_BITS` 返回的值。

`glStencilOp` 的其他两个自变量根据后续深度缓冲区测试成功 (`dppass`) 或失败 (`dpfail`) (参见 `glDepthFunc`) 来指定模板缓冲区动作。这些动作都是用与 `sfail` 相同的 8 个符号常量来指定的。注意，当不存在深度缓冲区，或者当深度缓冲区没有被激活时，`dpfail` 将被忽略。在这些情况下，`sfail` 和 `dppass` 分别指定在模板测试失败和通过时的模板动作。

#### 注意

在初始条件下，模板测试是被禁用的。如果不存在模板缓冲区，那么将不会发生任何模板修改，并且就像模板测试总是通过一样，不考虑 `glStencilOp` 的任何调用。

`glStencilOp` 与调用 `face` 被设置为 `GL_FRONT_AND_BACK` 的 `glStencilOpSeparate` 一样。

#### 错误

如果 `sfail`、`dpfail` 或 `dppass` 为已定义常数值以外的任何值，则产生 `GL_INVALID_ENUM` 错误。

#### 相关 Get 函数

`glGet` 参数为 `GL_STENCIL_FAIL`、`GL_STENCIL_PASS_DEPTH_PASS`、`GL_STENCIL_PASS_DEPTH_FAIL`、`GL_STENCIL_BACK_FAIL`、`GL_STENCIL_BACK_PASS_DEPTH_PASS`、`GL_STENCIL_BACK_PASS_DEPTH_FAIL` 或 `GL_STENCIL_BITS`。

`glIsEnabled`，其自变量为 `GL_STENCIL_TEST`。

#### 另外查看

`glBlendFunc`，`glDepthFunc`，`glEnable`，`glLogicOp`，`glStencilFunc`，`glStencilFuncSeparate`，`glStencilMask`，`glStencilMaskSeparate`，`glStencilOpSeparate`

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glStencilOpSeparate

设置前向和/或背向模板测试动作。

### C 规范

```
void glStencilOpSeparate(GLenum face,
                        GLenum sfail,
                        GLenum dpfail,
                        GLenum dppass);
```

### 参数

face

指定前向和/或背向模板状态是否被更新。共有 3 个标记可用, 分别是 GL\_FRONT、GL\_BACK 和 GL\_FRONT\_AND\_BACK。

sfail

指定在模板测试失败后采取的动作。共有 8 个标记可以接受: GL\_KEEP、GL\_ZERO、GL\_REPLACE、GL\_INCR、GL\_INCR\_WRAP、GL\_DECR、GL\_DECR\_WRAP 和 GL\_INVERT。初始值为 GL\_KEEP。

dpfail

指定在模板测试通过但深度测试失败后采取的模板动作。dpfail 接受与 sfail 相同的符号常量。初始值为 GL\_KEEP。

dppass

指定在模板测试都通过, 或在模板测试通过并且没有被激活的深度缓冲区或深度测试时采取的模板动作。

dppass 接受和 sfail 相同的符号常量。初始值为 GL\_KEEP。

描述

和深度缓冲一样, 模板化 (Stenciling) 在每个像素偏移上激活和禁止绘制。我们使用 GL 会址图元来在模板平面进行绘制, 然后对几何图形和图象进行渲染, 使用模板平面来遮蔽屏幕的一部分。典型情况下, 模板化在多道渲染 (multipass rendering) 运算中用来获得特殊效果, 这些效果有贴花 (decals)、轮廓 (outlining) 和结构立体几何渲染 (constructive solid geometry rendering) 等。

模板测试根据对引用值和模板缓冲区中的值进行比较的结果消除一个像素。可以通过调用自变量为 GL\_STENCIL\_TEST 的 glEnable 和 glDisable 来激活和禁止模板测试; 如果要控制它, 则可以调用 glStencilFunc 或 glStencilFuncSeparate。

可以有两组单独的 sfail、dpfail 和 dppass 参数。其中一组影响背向多边形, 而另外一组则影响前向多边形和其他非多边形图元。

glStencilOp 将前向和背向模板状态设置为同一个值, 就像调用了 face 被设置为 GL\_FRONT\_AND\_BACK 的 glStencilOpSeparate 一样。

glStencilOpSeparate 接受 3 个自变量, 这 3 个自变量指示在模板化被激活时存储的模板值上将会发生什么。如果模板测试失败, 那么像素的颜色或深度缓冲区将不会有任何改变, 并且 sfail 指定模板缓冲区内内容上将会发生什么。下列 8 个动作都可能发生。

GL\_KEEP           保持当前值。

GL\_ZERO           将模板缓冲区的值设为 0。

GL\_REPLACE        将模板缓冲区值设为 ref, 正如 glStencilFunc 所指定的。

GL\_INCR           增加当前模板缓冲区值。截取到可表示的最大无符号值。

GL\_INCR\_WRAP      增加当前模板缓冲区值。在增加可表示的最大无符号值时将模板缓冲区值环绕 (wrap) 到 0。

GL\_DECR           增加当前模板缓冲区值。截取到 0。

GL\_DECR\_WRAP      增加当前模板缓冲区值。在增加模板缓冲区值 0 时将模板缓冲区值环绕 (wrap) 到可表示的最大无符号值。

GL\_INVERT         对模板缓冲区值按位取反 (bitwise invert)。

模板缓冲区值将被视为无符号整数。在增加和减少时, 这些值被限制 (wrap) 到 0 和  $2^n-1$ , 其中  $n$  为查询 GL\_STENCIL\_BITS 返回的值。

glStencilOpSeparate 的其他两个自变量根据后续深度缓冲区测试成功 (dppass) 或失败 (dpfail) (参见 glDepthFunc) 来指定模板缓冲区动作。这些动作都是用与 sfail 相同的 8 个符号常量来指定的。注意, 当不存在深度缓冲区, 或者当深度缓冲区没有被激活时, dpfail 将被忽略。在这些情况下, sfail 和 dppass 分别指定在模板测试失败和通过时的模板动作。

注意

在初始条件下, 模板测试是被禁用的。如果不存在模板缓冲区, 那么将不会出现模板修改, 这就好像模板测试总是通过一样。

错误

如果 face 为除 GL\_FRONT、GL\_BACK 或 GL\_FRONT\_AND\_BACK 以外的任何值, 则产生 GL\_INVALID\_ENUM 错误。

如果 sfail、dpfail 或 dppass 为 8 个已定义常数值以外的任何值, 则产生 GL\_INVALID\_ENUM 错误。

相关 Get 函数

glGet 参数为 GL\_STENCIL\_FAIL、GL\_STENCIL\_PASS\_DEPTH\_PASS、GL\_STENCIL\_PASS\_DEPTH\_FAIL、

GL\_STENCIL\_BACK\_FAIL、GL\_STENCIL\_BACK\_PASS\_DEPTH\_PASS、GL\_STENCIL\_BACK\_PASS\_DEPTH\_FAIL 或 GL\_STENCIL\_BITS。

glIsEnabled, 其自变量为 GL\_STENCIL\_TEST。

#### 另外查看

glBlendFunc, glDepthFunc, glEnable, glLogicOp, glStencilFunc, glStencilFuncSeparate, glStencilMask, glStencilMaskSeparate, glStencilOp

#### 版权

Copyright © 2006 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glTexBuffer

将一个缓冲区对象的存储绑定到活动缓冲区纹理。

### C 规范

```
void glTexBuffer(GLenum target,
                 GLenum internalFormat,
                 GLuint buffer);
```

#### 参数

target

指定操作的目标, 必须为 GL\_TEXTURE\_BUFFER。

internalFormat

指定 buffer 的存储中数据的内部格式。

buffer

指定将其存储绑定到活动缓冲区纹理的缓冲区对象的名称。

#### 描述

glTexBuffer 将名为 buffer 的缓冲区对象的存储绑定到活动缓冲区纹理, 并为在绑定的缓冲区对象中找到的 texel 数组指定内部格式。如果 buffer 为 0, 任何绑定到这个缓冲区纹理的缓冲区对象都会被解除绑定, 并且不会绑定任何新的缓冲区对象。如果 buffer 是非 0 的, 那么它就必须为一个已经存在的缓冲区对象的名称。target 必须为 GL\_TEXTURE\_BUFFER。internalFormat 指定存储格式, 并且必须为下列指定大小的内部格式之一。

当一个缓冲区对象被绑定到一个缓冲区纹理时, 这个缓冲区对象的数据存储就会被视为这个纹理的纹理单元数组。在缓冲区纹理的纹理单元数组中, 纹理单元的数量由下式给出:

$$\frac{\text{buffer size}}{\text{componentsA-size of (base\_type)}}$$

其中 buffer\_size 是缓冲区对象的大小, 以基本机器单元为单位, 而其分量和基本类型则为元素数量和元素的基本数据类型, 正如下表所示。纹理单元数组中的纹理单元数量随后被截取到实现相关的限制值 GL\_MAX\_TEXTURE\_BUFFER\_SIZE。当一个缓冲区纹理在一个着色器中进行访问时, 如果指定纹理单元坐标为负值, 或者大于或等于纹理数组中纹理的截取数, 那么一次纹理单元获取所得到的结果将为未定义的。

指定大小内部格式	基本类型	Components	Norm	分量			
				0	1	2	3
GL_R8	ubyte	1	YES	R	0	0	1
GL_R16	ushort	1	YES	R	0	0	1
GL_R16F	half	1	NO	R	0	0	1
GL_R32F	float	1	NO	R	0	0	1
GL_R8I	byte	1	NO	R	0	0	1
GL_R16I	short	1	NO	R	0	0	1
GL_R32I	int	1	NO	R	0	0	1

续表

指定大小内部格式	基本类型	Components	Norm	0	1	2	3
GL_R8UI	ubyte	1	NO	R	0	0	1
GL_R16UI	ushort	1	NO	R	0	0	1
GL_R32UI	uint	1	NO	R	0	0	1
GL_RG8	ubyte	2	YES	R	G	0	1
GL_RG16	ushort	2	YES	R	G	0	1
GL_RG16F	half	2	NO	R	G	0	1
GL_RG32F	float	2	YES	R	G	0	1
GL_RG8I	byte	2	NO	R	G	0	1
GL_RG16I	short	2	NO	R	G	0	1
GL_RG32I	int	2	NO	R	G	0	1
GL_RG8UI	ubyte	2	NO	R	G	0	1
GL_RG16UI	ushort	2	NO	R	G	0	1
GL_RG32UI	uint	2	NO	R	G	0	1
GL_RGBA8	uint	4	YES	R	G	B	A
GL_RGBA16	short	4	YES	R	G	B	A
GL_RGBA16F	half	4	NO	R	G	B	A
GL_RGBA32F	float	4	NO	R	G	B	A
GL_RGBA8I	byte	4	NO	R	G	B	A
GL_RGBA16I	short	4	NO	R	G	B	A
GL_RGBA32I	int	4	NO	R	G	B	A
GL_RGBA8UI	ubyte	4	NO	R	G	B	A
GL_RGBA16UI	ushort	4	NO	R	G	B	A
GL_RGBA32UI	uint	4	NO	R	G	B	A

**错误**

如果 `target` 不是 `GL_TEXTURE_BUFFER`, 则产生 `GL_INVALID_ENUM` 错误。

如果 `internalFormat` 不是一个可接受的标记, 则产生 `GL_INVALID_ENUM` 错误。

如果 `buffer` 不是 0 或一个已经存在的缓冲区对象的名称, 则产生 `GL_INVALID_OPERATION` 错误。

**注意**

`glTexBuffer` 只在 3.1 或更高版本的 GL 中可用。

**相关 Get 函数**

`glGet`, 其自变量为 `GL_MAX_TEXTURE_BUFFER_SIZE`。

`glGet`, 其自变量为 `GL_TEXTURE_BINDING_BUFFER`。

`glGetTexLevelParameter`, 其自变量为 `GL_TEXTURE_BUFFER_DATA_STORE_BINDING`。

**另外查看**

`glGenBuffers`, `glBindBuffer`, `glBufferData`, `glDeleteBuffers`, `glGenTextures`, `glBindTexture`, `glDeleteTextures`

**版权**

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glTexImage1D

指定一个一维纹理图像。

### C 规范

```
void glTexImage1D(GLenum target,
                  GLint level,
                  GLint internalFormat,
                  GLsizei width,
                  GLint border,
                  GLenum format,
                  GLenum type,
                  const GLvoid * data);
```

### 参数

target

指定目标纹理。必须为 GL\_TEXTURE\_1D 或 GL\_PROXY\_TEXTURE\_1D。

level

指定层次细节数量。Level 0 是基本图像层次。Level n 是第 n 级 Mip 贴图缩略图。

internalFormat

指定这个纹理中颜色分量的数量。必须是下列符号常量中的一个：GL\_COMPRESSED\_RED、GL\_COMPRESSED\_RG、GL\_COMPRESSED\_RGB、GL\_COMPRESSED\_RGBA、GL\_COMPRESSED\_SRGB、GL\_COMPRESSED\_SRGB\_ALPHA、GL\_DEPTH\_COMPONENT、GL\_DEPTH\_COMPONENT16、GL\_DEPTH\_COMPONENT24、GL\_DEPTH\_COMPONENT32、GL\_R3\_G3\_B2、GL\_RED、GL\_RG、GL\_RGB、GL\_RGBA、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、GL\_RGBA16、GL\_SRGB、GL\_SRGB8、GL\_SRGB\_ALPHA 或 GL\_SRGB8\_ALPHA8。

width

指定纹理图像的宽度。所有实现支持纹理图像的宽度至少为 1024 个纹理单元。纹理图像的高度为 1。

border

这个值必须为 0。

format

指定像素数据的格式。下列符号值都可以接受：

GL\_RED、GL\_RG、GL\_RGB、GL\_BGR、GL\_RGBA 和 GL\_BGRA。

type

指定像素数据的数据类型。下列符号值都可以接受：

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。

data

指定到内存中图像数据的指针。

### 描述

纹理渲染将一个指定纹理图像的一部分映射到每个纹理渲染被激活的基本图元 ( graphical primitive ) 上。可以通过调用自变量为 GL\_TEXTURE\_1D 的 glEnable 和 glDisable 来激活和禁止一维纹理渲染。

纹理图像由 glTexImage1D 定义。自变量描述纹理图像的参数，诸如宽度、边界宽度、层次细节数量 ( 参见 glTexParameter ) 和用于存储图像的内部分辨率与格式。最后 3 个自变量描述图像如何在内存中表示。

如果 target 是 GL\_PROXY\_TEXTURE\_1D，那么从 data 中不会读取任何数据，但是所有的纹理图像状态将会为了一致性而重新进行计算、检查，并核对实现的性能。如果一个实现不能处理所要求的纹理大小，那么它会将所有的图像状态设置为 0，但并不生成错误 ( 参见 glGetError )。可以使用一个级别等于 1 或大于 1 的图像数组来查询整个 Mip 贴图数组。

如果 target 为 GL\_TEXTURE\_1D，那么数据将根据 type 从 data 中作为一个有符号或无符号字节、短整形、长

整形或单精度浮点值序列读取。这些值将根据 format 被分组为 1 个、2 个、3 个或 4 个值的组来形成元素。每个数据字节都会被视作 8 个 1 位元素, 位顺序由 GL\_UNPACK\_LSB\_FIRST 决定 (参见 glPixelStore)。

如果在纹理图像被指定的情况下, 非 0 的指定缓冲区对象被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标 (参见 glBindBuffer), data 将被看作缓冲区对象数据存储的一个 byte 偏移。

第一个元素与纹理数组的左端对应。后续元素按照从左到右的顺序对应纹理数组中剩下的纹理单元。最后一个元素与纹理数组的右端对应。

format 决定 data 中每个元素的分量。它可以采用下列符号值中的一个。

GL\_RED

每个元素都是单个红色分量。GL 将它转换成浮点值, 并将它组合成一个 RGBA 元素, 绿色和蓝色绑定到 0, 而 Alpha 绑定到 1。然后, 每个分量都乘以有符号缩放因子 GL\_c\_SCALE, 加到有符号偏移 GL\_c\_BIAS 上, 并截取到 [0,1] 范围内。

GL\_RG

每个元素都是单个红色/绿色双重分量。GL 将它转换成浮点值, 并将它组合成一个 RGBA 元素, 蓝色绑定到 0, 而 Alpha 绑定到 1。然后, 每个分量都乘以有符号缩放因子 GL\_c\_SCALE, 加到有符号偏移 GL\_c\_BIAS 上, 并截取到 [0,1] 范围内。

GL\_RGB GL\_BGR

每个元素都是一个 RGB 三元组。GL 将它转换成浮点值, 并通过将 alpha 绑定到 1 而将它组合成一个 RGBA 元素。然后, 每个分量都乘以有符号缩放因子 GL\_c\_SCALE, 加到有符号偏移 GL\_c\_BIAS 上, 并截取到 [0,1] 范围内。

GL\_RGBA GL\_BGRA

每个元素都包含所有 4 个分量。每个分量都乘以有符号缩放因子 GL\_c\_SCALE, 加到有符号偏移 GL\_c\_BIAS 上, 并截取到 [0,1] 范围内。

GL\_DEPTH\_COMPONENT

每个元素都是单个深度值。GL 将它转换成浮点值, 乘以有符号缩放因子 GL\_DEPTH\_SCALE, 加到有符号偏移 GL\_DEPTH\_BIAS 上, 并截取到 [0,1] 范围内。

如果一个应用程序想要以一个特定分辨率或特定格式来存储纹理, 那么它可以通过 internalFormat 来请求相应的分辨率和格式。GL 将选择一个与 internalFormat 请求的非常接近的内部表示法, 但可能不会严格地匹配。

由 GL\_RED、GL\_RG、GL\_RGB 和 GL\_RGBA 指定的表示法必须严格匹配)。如果 internalFormat 参数是一种一般压缩格式 GL\_COMPRESSED\_RED、GL\_COMPRESSED\_RG、GL\_COMPRESSED\_RGB 或 GL\_COMPRESSED\_RGBA, 那么 GL 将使用一个指定内部格式的符号常量来代替这个内部格式, 并且在存储之前对纹理进行压缩。如果没有相应的内部格式可用, 或者 GL 由于任何原因而不能对图像进行压缩, 那么内部格式将由一个相应的基础内部格式来代替。

如果 internalFormat 参数为 GL\_SRGB、GL\_SRGB8、GL\_SRGB\_ALPHA 或 GL\_SRGB8\_ALPHA8, 那么纹理将按照红色、绿色或蓝色分量被编码到 sRGB 颜色空间的情况来对待。任何 Alpha 分量都会保持不变。从 sRGB 编码分量  $c_s$  到线性分量  $c_l$  的转换如下。

$$c_l = \begin{cases} \frac{c_s}{12.92} & \text{if } c_s \leq 0.04045 \\ \left( \frac{c_s + 0.055}{1.055} \right)^{2.4} & \text{if } c_s > 0.04045 \end{cases}$$

假设  $c_s$  为 [0,1] 范围内的 sRGB 分量。

使用 GL\_PROXY\_TEXTURE\_1D 目标来对分辨率和格式进行试验。实现将更新和重新计算它对要求的存储分辨率和格式的最佳匹配。要查询这个状态, 可以调用 glGetTexLevelParameter。如果纹理不能适应, 那么纹理状态将被设为 0。

一个单分量纹理图像只使用 data 中 RGBA 颜色的红色分量。一个 2 分量图像使用 R 和 A 值。一个 3 分量图像使用 R、G 和 B 值。

一个 4 分量图像则使用所有的 RGBA 分量。

基于图像的着色 (Image-based shadowing) 可以通过对 r 坐标和深度纹理值进行比较来生成一个布尔结果而激活。关于纹理比较的细节请参见 glTexParameter。

注意

glPixelStore 模式会影响纹理图像。

data 可以是一个空指针。在这种情况下, 将会分配纹理内存来容纳一个宽度为 width 的纹理。随后我们就可以下

载子纹理来对纹理内存进行初始化了。如果程序试图将纹理图像的一个未初始化部分应用到一个图元上, 那么这个图像将为未定义的。

`glTexImage1D` 为当前纹理单元 (由 `glActiveTexture` 指定) 指定一维纹理。

#### 错误

如果 `target` 不是 `GL_TEXTURE_1D` 或 `GL_PROXY_TEXTURE_1D`, 则产生 `GL_INVALID_ENUM` 错误。

如果 `format` 不是一个可接受的格式常量, 则产生 `GL_INVALID_ENUM` 错误。除 `GL_STENCIL_INDEX` 之外的格式常量都可以接受。

如果 `type` 不是一个可接受的类型常量, 则产生 `GL_INVALID_ENUM` 错误。

如果 `level` 小于 0, 则产生 `GL_INVALID_VALUE` 错误。

如果 `level` 大于  $\log_2 \max$  值, 其中 `max` 是 `GL_MAX_TEXTURE_SIZE` 的返回值, 则可能产生 `GL_INVALID_VALUE` 错误。

如果 `internalformat` 不是一个可接受的分辨率和格式符号常量, 则产生 `GL_INVALID_VALUE` 错误。

如果 `width` 小于 0 或大于 `GL_MAX_TEXTURE_SIZE`, 则产生 `GL_INVALID_VALUE` 错误。

如果不支持非 2 的幂的纹理且 `width` 不能被表示成  $2^n+2$  (border) 的形式, 其中 `n` 为整数, 则产生 `GL_INVALID_VALUE` 错误。

如果 `border` 不是 0 或 1, 则产生 `GL_INVALID_VALUE` 错误。

如果 `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` 中的一个, 并且 `format` 不是 `GL_RGB`, 则产生 `GL_INVALID_OPERATION` 错误。

如果 `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`、`GL_UNSIGNED_INT_5_9_9_9_REV` 中的一个, 并且 `format` 既不是 `GL_RGBA` 也不是 `GL_BGRA`, 则产生 `GL_INVALID_OPERATION` 错误。

如果 `format` 是 `GL_DEPTH_INDEX`, 并且 `internalFormat` 不是 `GL_DEPTH_COMPONENT`、`GL_DEPTH_COMPONENT16`、`GL_DEPTH_COMPONENT24` 或 `GL_DEPTH_COMPONENT32`, 则产生 `GL_INVALID_OPERATION` 错误。

如果 `internalFormat` 是 `GL_DEPTH_COMPONENT`、`GL_DEPTH_COMPONENT16`、`GL_DEPTH_COMPONENT24` 或 `GL_DEPTH_COMPONENT32`, 并且 `format` 不是 `GL_DEPTH_INDEX`, 则产生 `GL_INVALID_OPERATION` 错误。

如果非 0 缓冲区对象名称被绑定到 `GL_PIXEL_UNPACK_BUFFER` 目标, 并且缓冲区对象的数据存储当前被映射, 则产生 `GL_INVALID_OPERATION` 错误。

如果非 0 缓冲区对象名称被绑定到 `GL_PIXEL_UNPACK_BUFFER` 目标, 并且数据将从缓冲区对象中被解包导致内存读取需求超出数据存储大小, 则产生 `GL_INVALID_OPERATION` 错误。

如果非 0 缓冲区对象名称被绑定到 `GL_PIXEL_UNPACK_BUFFER` 目标, 并且 `data` 没有平均分成一个由 `type` 指定的数据存储到内存中所需的字节数, 则产生 `GL_INVALID_OPERATION` 错误。

#### 相关 Get 函数

`glGetTexImage`

`glGet`, 其自变量为 `GL_PIXEL_UNPACK_BUFFER_BINDING`。

#### 另外查看

`glActiveTexture`, `glCompressedTexImage1D`, `glCompressedTexSubImage1D`, `glCopyTexImage1D`, `glCopyTexSubImage1D`, `glGetCompressedTexImage`, `glPixelStore`, `glTexImage2D`, `glTexImage3D`, `glTexSubImage1D`, `glTexSubImage2D`, `glTexSubImage3D`, `glTexParameter`

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glTexImage2D

指定一个二维纹理图像。

### C 规范

```
void glTexImage2D(GLenum target,
                  GLint level,
                  GLint internalFormat,
                  GLsizei width,
                  GLsizei height,
                  GLint border,
                  GLenum format,
                  GLenum type,
                  const GLvoid * data);
```

### 参数

#### target

指定目标纹理。必须为 GL\_TEXTURE\_2D、GL\_PROXY\_TEXTURE\_2D、GL\_TEXTURE\_1D\_ARRAY、GL\_PROXY\_TEXTURE\_1D\_ARRAY、GL\_TEXTURE\_RECTANGLE、GL\_PROXY\_TEXTURE\_RECTANGLE、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Z、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Z 或 GL\_PROXY\_TEXTURE\_CUBE\_MAP。

#### level

指定层次细节数量。Level 0 是基本图像层次。Level n 是第 n 级 Mip 贴图缩略图。如果 target 为 GL\_TEXTURE\_RECTANGLE 或 GL\_PROXY\_TEXTURE\_RECTANGLE, 那么 level 必须为 0。

#### internalFormat

指定这个纹理中颜色分量的数量。必须是下列符号常量中的一个: GL\_COMPRESSED\_RED、GL\_COMPRESSED\_RG、GL\_COMPRESSED\_RGB、GL\_COMPRESSED\_RGBA、GL\_COMPRESSED\_SRGB、GL\_COMPRESSED\_SRGB\_ALPHA、GL\_DEPTH\_COMPONENT、GL\_DEPTH\_COMPONENT16、GL\_DEPTH\_COMPONENT24、GL\_DEPTH\_COMPONENT32、GL\_R3\_G3\_B2、GL\_RED、GL\_RG、GL\_RGB、GL\_RGBA、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、GL\_RGBA16、GL\_SRGB、GL\_SRGB8、GL\_SRGB\_ALPHA 或 GL\_SRGB8\_ALPHA8。

#### width

指定纹理图像的宽度。所有实现支持纹理图像的宽度至少为 1024 个纹理单元。

#### height

在 GL\_TEXTURE\_1D\_ARRAY 和 GL\_PROXY\_TEXTURE\_1D\_ARRAY 目标的情况下, 指定纹理图像的高度, 或者一个纹理数组中的层数。

所有实现支持的 2D 纹理图像的高度至少为 1024 个纹理单元, 并且纹理单元的深度至少为 256 层。

#### border

这个值必须为 0。

#### format

指定像素数据的格式。下列符号值都可以接受:

GL\_RED、GL\_RG、GL\_RGB、GL\_BGR、GL\_RGBA 和 GL\_BGRA。

#### type

指定像素数据的数据类型。下列符号值都可以接受:

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。

#### data

指定到内存中图像数据的指针。

#### 描述

纹理贴图着色器读取一个图像数组的元素。

调用 glTexImage2D 来定义纹理图像。自变量描述纹理图像的参数, 诸如高度、宽度、边界宽度、层次细节数量 (参见 glTexParameter) 和提供的颜色分量数量。最后三个自变量描述图像如何在内存中表示。

如果 `target` 是 `GL_PROXY_TEXTURE_2D`、`GL_PROXY_TEXTURE_1D_ARRAY`、`GL_PROXY_TEXTURE_CUBE_MAP` 或 `GL_PROXY_TEXTURE_RECTANGLE`，那么从 `data` 中不会读取任何数据，但是所有的纹理图像状态将会为了一致性而重新进行计算、检查，并核对实现的性能。如果一个实现不能处理所要求的纹理大小，那么它会将所有的图像状态设置为 0，但并不生成错误（参见 `glGetError`）。可以使用一个级别等于 1 或大于 1 的图像数组来查询整个 Mip 贴图数组。

如果 `target` 为 `GL_TEXTURE_2D`、`GL_TEXTURE_RECTANGLE` 或者一个 `GL_TEXTURE_CUBE_MAP` 目标，那么数据将根据 `type` 从 `data` 中作为一个有符号或无符号字节、短整形、长整形或单精度浮点值序列读取。这些值将根据 `format` 被分组为 1 个、2 个、3 个或 4 个值的组来形成元素。每个数据字节都会被视作 8 个 1 位元素，位顺序由 `GL_UNPACK_LSB_FIRST` 决定（参见 `glPixelStore`）。

如果 `target` 是 `GL_TEXTURE_1D_ARRAY`，那么 `data` 将被解释为一个一维图像数组。

如果在纹理图像被指定的情况下，非 0 的指定缓冲区对象被绑定到 `GL_PIXEL_UNPACK_BUFFER` 目标（参见 `glBindBuffer`），`data` 将被看作缓冲区对象数据存储的一个字节偏移。

第一个元素与纹理图像的左下角对应。后续元素按照从左到右的顺序对应纹理图像最低行中剩下的纹理单元，然后是纹理图像中紧邻着的较高一行。最后一个元素与纹理图像的右上角对应。

`format` 决定 `data` 中每个元素的分量。它可以采用下列符号值中的一个。

`GL_RED`

每个元素都是单个红色分量。GL 将它转换成浮点值，并将它组合成一个 RGBA 元素，绿色和蓝色绑定到 0，而 Alpha 绑定到 1。然后，每个分量都乘以有符号缩放因子 `GL_c_SCALE`，加到有符号偏移 `GL_c_BIAS` 上，并截取到 `[0,1]` 范围内。

`GL_RG`

每个元素都是一个红色/绿色二元组。GL 将它转换成浮点值，并将它组合成一个 RGBA 元素，蓝色绑定到 0，而 Alpha 绑定到 1。然后，每个分量都乘以有符号缩放因子 `GL_c_SCALE`，加到有符号偏移 `GL_c_BIAS` 上，并截取到 `[0,1]` 范围内。

`GL_RGB GL_BGR`

每个元素都是一个 RGB 三元组。GL 将它转换成浮点值，并通过将 `alpha` 绑定到 1 而将它组合成一个 RGBA 元素。然后，每个分量都乘以有符号缩放因子 `GL_c_SCALE`，加到有符号偏移 `GL_c_BIAS` 上，并截取到 `[0,1]` 范围内。

`GL_RGBA GL_BGRA`

每个元素都包含所有 4 个分量。每个分量都乘以有符号缩放因子 `GL_c_SCALE`，加到有符号偏移 `GL_c_BIAS` 上，并截取到 `[0,1]` 范围内。

`GL_DEPTH_COMPONENT`

每个元素都是单个深度值。GL 将它转换成浮点值，乘以有符号缩放因子 `GL_DEPTH_SCALE`，加到有符号偏移 `GL_DEPTH_BIAS` 上，并截取到 `[0,1]` 范围内。

`GL_DEPTH_STENCIL`

每个元素都是一个深度值和模板值对。这个数值对中的深度分量被解释为在 `GL_DEPTH_COMPONENT` 中。模板分量的解释要根据指定的 `depth + stencil` 内部格式。

如果一个应用程序想要以一个特定分辨率或特定格式来存储纹理，那么它可以通过 `internalFormat` 来请求相应的分辨率和格式。GL 将选择一个与 `internalFormat` 请求的非常接近的内部表示法，但可能不会严格地匹配。

由 `GL_RED`、`GL_RG`、`GL_RGB` 和 `GL_RGBA` 指定的表示法必须严格匹配）。

如果 `internalFormat` 参数是一种一般压缩格式 `GL_COMPRESSED_RED`、`GL_COMPRESSED_RG`、`GL_COMPRESSED_RGB` 或 `GL_COMPRESSED_RGBA`，那么 GL 将使用一个指定内部格式的符号常量来代替这个内部格式，并且在存储之前对纹理进行压缩。如果没有相应的内部格式可用，或者 GL 由于任何原因而不能对图像进行压缩，那么内部格式将由一个相应的基础内部格式来代替。

如果 `internalFormat` 参数为 `GL_SRGB`、`GL_SRGB8`、`GL_SRGB_ALPHA` 或 `GL_SRGB8_ALPHA8`，那么纹理将按照红色、绿色或蓝色分量被编码到 sRGB 颜色空间的情况来对待。任何 Alpha 分量都会保持不变。从 sRGB 编码分量  $c_s$  到线性分量  $c_l$  的转换如下。

$$c_l = \begin{cases} \frac{c_s}{12.92} & \text{if } c_s \leq 0.04045 \\ \left( \frac{c_s + 0.055}{1.055} \right)^{2.4} & \text{if } c_s > 0.04045 \end{cases}$$

假设  $c_s$  为 `[0,1]` 范围内的 sRGB 分量。

使用 GL\_PROXY\_TEXTURE\_2D、GL\_PROXY\_TEXTURE\_1D\_ARRAY、GL\_PROXY\_TEXTURE\_RECTANGLE 或 GL\_PROXY\_TEXTURE\_CUBE\_MAP 目标来对分辨率和格式进行试验。实现将更新和重新计算它对要求的存储分辨率和格式的最佳匹配。要查询这个状态,可以调用 glGetTexLevelParameter。如果纹理不能适应,那么纹理状态将被设为 0。

一个单分量纹理图像只使用从 data 中提取的 RGBA 颜色的红色分量。一个 2 分量图像使用 R 和 G 值。一个 3 分量图像使用 R、G 和 B 值。一个 4 分量图像则使用所有的 RGBA 分量。

基于图像的着色 (Image-based shadowing) 可以通过对 r 坐标和深度纹理值进行比较来生成一个布尔结果而激活。关于纹理比较的细节请参见 glGetTexParameter。

#### 注意

glPixelStore 会影响纹理图像。

data 可以是一个空指针。在这种情况下,将会分配纹理内存来容纳一个宽度为 width、高度为 height 的纹理。随后我们就可以下载子纹理来对纹理内存进行初始化了。如果用户试图将纹理图像的一个未初始化部分应用到一个图元上,那么这个图像将为未定义的。

glTexImage2D 为当前纹理单元 (由 glActiveTexture 指定) 指定二维纹理。

#### 错误

如果 tarGet 不是 GL\_TEXTURE\_2D、GL\_TEXTURE\_1D\_ARRAY、GL\_TEXTURE\_RECTANGLE、GL\_PROXY\_TEXTURE\_2D、GL\_PROXY\_TEXTURE\_1D\_ARRAY、GL\_PROXY\_TEXTURE\_RECTANGLE、GL\_PROXY\_TEXTURE\_CUBE\_MAP、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Z 或 GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Z,则产生 GL\_INVALID\_ENUM 错误。

如果 tarGet 是 6 个立方体贴图 2D 图像目标中的一个,并且宽度和高度参数不相等,则产生 GL\_INVALID\_ENUM 错误。

如果 type 不是一个可接受的类型常量,则产生 GL\_INVALID\_ENUM 错误。

如果 width 小于 0 或大于 GL\_MAX\_TEXTURE\_SIZE,则产生 GL\_INVALID\_VALUE 错误。

如果 tarGet 不是 GL\_TEXTURE\_1D\_ARRAY 或 GL\_PROXY\_TEXTURE\_1D\_ARRAY,并且 height 小于 0 或大于 GL\_MAX\_TEXTURE\_SIZE,则产生 GL\_INVALID\_VALUE 错误。

如果 tarGet 是 GL\_TEXTURE\_1D\_ARRAY 或 GL\_PROXY\_TEXTURE\_1D\_ARRAY,并且 height 小于 0 或大于 GL\_MAX\_ARRAY\_TEXTURE\_LAYERS,则产生 GL\_INVALID\_VALUE 错误。

如果 level 小于 0,则产生 GL\_INVALID\_VALUE 错误。

如果 level 大于 log<sub>2</sub>max,其中 max 是 GL\_MAX\_TEXTURE\_SIZE 的返回值,则可能产生 GL\_INVALID\_VALUE 错误。

如果 internalformat 不是一个可接受的分辨率和格式符号常量,则产生 GL\_INVALID\_VALUE 错误。

如果 width 或 height 小于 0 或大于 GL\_MAX\_TEXTURE\_SIZE,则产生 GL\_INVALID\_VALUE 错误。

如果不支持非 2 的幂的纹理且 width 或 height 不能被表示成  $2^k + 2$  (border) 的形式,其中 k 为整数,则产生 GL\_INVALID\_VALUE 错误。

如果 border 为非 0 值,则产生 GL\_INVALID\_VALUE 错误。

如果 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 或 GL\_UNSIGNED\_INT\_10F\_11F\_11F\_REV 中的一个,并且 format 不是 GL\_RGB,则生成 GL\_INVALID\_OPERATION 错误。

如果 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、GL\_UNSIGNED\_INT\_2\_10\_10\_10\_REV 或 GL\_UNSIGNED\_INT\_5\_9\_9\_9\_REV 中的一个,并且 format 既不是 GL\_RGBA 也不是 GL\_BGRA,则生成 GL\_INVALID\_OPERATION 错误。

如果 tarGet 不是 GL\_TEXTURE\_2D、GL\_PROXY\_TEXTURE\_2D、GL\_TEXTURE\_RECTANGLE 或 GL\_PROXY\_TEXTURE\_RECTANGLE,并且 internalFormat 是 GL\_DEPTH\_COMPONENT、GL\_DEPTH\_COMPONENT16、GL\_DEPTH\_COMPONENT24 或 GL\_DEPTH\_COMPONENT32F,则产生 GL\_INVALID\_OPERATION 错误。

如果 format 是 GL\_DEPTH\_INDEX,并且 internalFormat 不是 GL\_DEPTH\_COMPONENT、GL\_DEPTH\_COMPONENT16、GL\_DEPTH\_COMPONENT24 或 GL\_DEPTH\_COMPONENT32F,则产生 GL\_INVALID\_OPERATION 错误。

如果 internalFormat 是 GL\_DEPTH\_COMPONENT、GL\_DEPTH\_COMPONENT16、GL\_DEPTH\_COMPONENT24 或 GL\_DEPTH\_COMPONENT32F, 并且 format 不是 GL\_DEPTH\_INDEX, 则产生 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标, 并且缓冲区对象的数据存储当前被映射, 则产生 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标, 并且数据将从缓冲区对象中被解包导致内存读取需求超出数据存储大小, 则产生 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标, 并且 data 没有平均分成将一个由 type 指定的数据存储到内存中所需的字节数, 则产生 GL\_INVALID\_OPERATION 错误。

如果 target 为 GL\_TEXTURE\_RECTANGLE 或 GL\_PROXY\_TEXTURE\_RECTANGLE, 并且 level 为非 0 值, 则产生 GL\_INVALID\_VALUE 错误。

#### 相关 Get 函数

glGetTexImage

glGet, 其自变量为 GL\_PIXEL\_UNPACK\_BUFFER\_BINDING。

#### 另外查看

glActiveTexture, glCopyTexImage1D, glCopyTexImage2D, glCopyTexSubImage1D, glCopyTexSubImage2D, glCopyTexSubImage3D, glPixelStore, glTexImage1D, glTexImage3D, glTexSubImage1D, glTexSubImage2D, glTexSubImage3D, glTexParameter

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glTexImage2DMultisample

建立多重采样纹理的图像的数据存储、格式、维度和样本数量。

### C 规范

```
void glTexImage2DMultisample(GLenum target,
                             GLsizei samples,
                             GLint internalformat,
                             GLsizei width,
                             GLsizei height,
                             GLboolean fixedsamplelocations);
```

#### 参数

target

指定操作的目标。target 必须为 GL\_TEXTURE\_2D\_MULTISAMPLE\_ARRAY 或 GL\_PROXY\_TEXTURE\_2D\_MULTISAMPLE\_ARRAY。

samples

多重采样纹理图像中的样本数量。

internalformat

存储多重采样纹理图像所使用的内部格式。

internalformat 必须指定一个颜色可渲染、深度可渲染或模板可渲染的格式。

width

多重采样纹理图像的宽度, 以纹理单元为单位。

height

多重采样纹理图像的高度, 以纹理单元为单位。

fixedsamplelocations

指定图像是否将要对图像中的所有纹理单元使用同样的样本位置和同样的采样数, 并且样本位置不取决于内部格式或图像大小。

#### 描述

glTexImage2DMultisample 建立多重采样纹理的图像的数据存储、格式、维度和样本数量。

target 必须为 GL\_TEXTURE\_2D\_MULTISAMPLE 或 GL\_PROXY\_TEXTURE\_2D\_MULTISAMPLE。

width 和 height 为纹理中的纹理单元的维度, 必须在从 0 到 GL\_MAX\_TEXTURE\_SIZE - 1 的范围内。samples 指

定图像中采样数量, 必须在从 0 到 `GL_MAX_SAMPLES - 1` 的范围之内。

`Internalformat` 必须是一个颜色可渲染、深度可渲染或模板可渲染的格式。

如果 `fixedsamplelocations` 为 `GL_TRUE`, 图像将要对图像中的所有纹理单元使用同样的样本位置和同样的采样数, 并且样本位置不取决于内部格式或图像大小。

当在一个着色器中访问一个多重采样纹理时, 这个访问将采用一个整数向量来描述要获取哪个纹理单元, 以及一个与采样数相对应的整数来描述获取纹理单元中的哪个样本。在多重采样纹理目标上不允许有标准采样指令。

**注意**

`glTexImage2DMultisample` 在 3.2 或更高版本的 GL 中可用。

**错误**

如果 `internalformat` 是一个深度可渲染或模板可渲染格式, 并且 `samples` 大于或等于 `GL_MAX_DEPTH_TEXTURE_SAMPLES` 的值, 则产生 `GL_INVALID_OPERATION` 错误。

如果 `internalformat` 是一个颜色可渲染格式, 并且 `samples` 大于或等于 `GL_MAX_COLOR_TEXTURE_SAMPLES` 的值, 则产生 `GL_INVALID_OPERATION` 错误。

如果 `internalformat` 是一个有符号或无符号的整数格式, 并且 `samples` 大于或等于 `GL_MAX_INTEGER_SAMPLES` 的值, 则产生 `GL_INVALID_OPERATION` 错误。

如果 `width` 或者 `height` 为负值或者大于 `GL_MAX_TEXTURE_SIZE`, 则产生 `GL_INVALID_VALUE` 错误。

如果 `samples` 大于 `GL_MAX_SAMPLES`, 则产生 `GL_INVALID_VALUE` 错误。

**另外查看**

`glTexImage3D`, `glTexImage2DMultisample`

**版权**

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。 <http://opencontent.org/openpub/>。

## glTexImage3D

指定一个三维纹理图像。

**C 规范**

```
void glTexImage3D(GLenum target,
                  GLint level,
                  GLint internalFormat,
                  GLsizei width,
                  GLsizei height,
                  GLsizei depth,
                  GLint border,
                  GLenum format,
                  GLenum type,
                  const GLvoid * data);
```

**参数**

`target`

指定目标纹理。必须为 `GL_TEXTURE_3D`、`GL_PROXY_TEXTURE_3D`、`GL_TEXTURE_2D_ARRAY` 或 `GL_PROXY_TEXTURE_2D_ARRAY`。

`level`

指定层次细节数量。Level 0 是基本图像层次。Level n 是第 n 级 Mip 贴图缩略图。

`internalFormat`

指定这个纹理中颜色分量的数量。必须是下列符号常量中的一个: `GL_RGBA32F`、`GL_RGBA32I`、`GL_RGBA32UI`、`GL_RGBA16`、`GL_RGBA16F`、`GL_RGBA16I`、`GL_RGBA16UI`、`GL_RGBA8`、`GL_RGBA8UI`、`GL_SRGB8_ALPHA8`、`GL_RGB10_A2`、`GL_RGBA10_A2UI`、`GL_R11_G11_B10F`、`GL_RG32F`、`GL_RG32I`、`GL_RG32UI`、`GL_RG16`、`GL_RG16F`、`GL_RGB16I`、`GL_RGB16UI`、`GL_RG8`、`GL_RG8I`、`GL_RG8UI`、`GL_R23F`、`GL_R32I`、`GL_R32UI`、`GL_R16F`、`GL_R16I`、`GL_R16UI`、`GL_R8`、`GL_R8I`、`GL_R8UI`、`GL_RGBA16_UNORM`、`GL_RGBA8_SNORM`、`GL_RGB32F`、`GL_RGB32I`、`GL_RGB32UI`、`GL_RGB16_SNORM`、`GL_RGB16F`、`GL_RGB16I`、`GL_RGB16UI`、`GL_RGB16`、`GL_RGB8_SNORM`、`GL_RGB8`、`GL_RGB8I`、`GL_RGB8UI`、`GL_SRGB8`、`GL_RGB9_E5`、

GL\_RG16\_SNORM、GL\_RG8\_SNORM、GL\_COMPRESSED\_RG\_RGTC2、GL\_COMPRESSED\_SIGNED\_RG\_RGTC2、GL\_R16\_SNORM、GL\_R8\_SNORM、GL\_COMPRESSED\_RED\_RGTC1、GL\_COMPRESSED\_SIGNED\_RED\_RGTC1、GL\_DEPTH\_COMPONENT32F、GL\_DEPTH\_COMPONENT24、GL\_DEPTH\_COMPONENT16、GL\_DEPTH32F\_STENCIL8、GL\_DEPTH24\_STENCIL8。

width

指定纹理图像的宽度。所有实现都支持宽度至少为 16 个纹理单元的 3D 纹理图像。

height

指定纹理图像的高度。所有实现都支持高度至少为 256 个纹理单元的 3D 纹理图像。

depth

指定纹理图像的深度，或者纹理数组的层数。所有实现都支持深度至少为 256 个纹理单元的 3D 纹理图像，以及深度至少为 256 层的纹理数组。

border

这个值必须为 0。

format

指定像素数据的格式。下列符号值都可以接受：

GL\_RED、GL\_RG、GL\_RGB、GL\_BGR、GL\_RGBA 和 GL\_BGRA。

type

指定像素数据的数据类型。下列符号值都可以接受：

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。

data

指定到内存中图像数据的指针。

描述

纹理渲染将一个指定纹理图像的一部分映射到每个纹理渲染被激活的基本图元 (graphical primitive) 上。可以通过调用自变量为 GL\_TEXTURE\_3D 的 glEnable 和 glDisable 来激活和禁止三维纹理渲染。

调用 glTexImage3D 来定义纹理图像。自变量描述纹理图像的参数，诸如高度、宽度、深度、边界宽度、层次细节数量 (参见 glTexParameter) 和提供的颜色分量数量。最后 3 个自变量描述图像如何在内存中表示。

如果 target 是 GL\_PROXY\_TEXTURE\_3D，那么从 data 中不会读取任何数据，但是所有的纹理图像状态将会为了一致性而重新进行计算、检查，并核对实现的性能。如果一个实现不能处理所要求的纹理大小，那么它会将所有的图像状态设置为 0，但并不生成错误 (参见 glGetError)。可以使用一个级别等于 1 或大于 1 的图像数组来查询整个 Mip 贴图数组。

如果 target 为 GL\_TEXTURE\_3D，那么数据将根据 type 从 data 中作为一个有符号或无符号字节、短整形、长整形或单精度浮点值序列读取。这些值将根据 format 被分组为 1 个、2 个、3 个或 4 个值的组来形成元素。每个数据字节都会被视作 8 个 1 位元素，位顺序由 GL\_UNPACK\_LSB\_FIRST 决定 (参见 glPixelStore)。

如果在纹理图像被指定的情况下，非 0 的指定缓冲区对象被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标 (参见 glBindBuffer)，data 将被看作缓冲区对象数据存储的一个字节偏移。

第一个元素与纹理图像的左下角对应。后续元素按照从左到右的顺序对应纹理图像最低行中剩下的纹理单元，然后是纹理图像中紧邻着的较高一行。最后一个元素与纹理图像的右上角对应。

format 决定 data 中每个元素的分量。它可以采用下列符号值中的一个。

GL\_RED

每个元素都是单个红色分量。GL 将它转换成浮点值，并将它组合成一个 RGBA 元素，绿色和蓝色绑定到 0，而 Alpha 绑定到 1。然后，每个分量都乘以有符号缩放因子 GL\_c\_SCALE，加到有符号偏移 GL\_c\_BIAS 上，并截取到 [0,1] 范围内。

GL\_RG

每个元素都是一个红色/绿色二元组。GL 将每一个都转换成浮点值，并将它组合成一个 RGBA 元素，蓝色绑定到 0，而 Alpha 绑定到 1。然后，每个分量都乘以有符号缩放因子 GL\_c\_SCALE，加到有符号偏移 GL\_c\_BIAS 上，并截取到 [0,1] 范围内。

GL\_RGB

GL\_BGR

每个元素都是一个 RGB 三元组。GL 将它转换成浮点值, 并通过将 alpha 绑定到 1 而将它组合成一个 RGBA 元素。然后, 每个分量都乘以有符号缩放因子 GL\_c\_SCALE, 加到有符号偏移 GL\_c\_BIAS 上, 并截取到[0,1]范围内。

GL\_RGBA GL\_BGRA

每个元素都包含所有 4 个分量。每个分量都乘以有符号缩放因子 GL\_c\_SCALE, 加到有符号偏移 GL\_c\_BIAS 上, 并截取到[0,1]范围内。

如果一个应用程序想要以一个特定分辨率或特定格式来存储纹理, 那么它可以通过 internalFormat 来请求相应的分辨率和格式。GL 将选择一个与 internalFormat 请求的非常接近的内部表示法, 但可能不会严格地匹配。

由 GL\_RED、GL\_RG、GL\_RGB 和 GL\_RGBA 指定的表示法必须严格匹配)。如果 internalFormat 参数是一种一般压缩格式 GL\_COMPRESSED\_RED、GL\_COMPRESSED\_RG、GL\_COMPRESSED\_RGB 或 GL\_COMPRESSED\_RGBA, 那么 GL 将使用一个指定内部格式的符号常量来代替这个内部格式, 并且在存储之前对纹理进行压缩。如果没有相应的内部格式可用, 或者 GL 由于任何原因而不能对图像进行压缩, 那么内部格式将由一个相应的基础内部格式来代替。

如果 internalFormat 参数为 GL\_SRGB、GL\_SRGB8、GL\_SRGB\_ALPHA 或 GL\_SRGB8\_ALPHA8, 那么纹理将按照红色、绿色、蓝色或透明度分量被编码到 sRGB 颜色空间的情况来对待。任何 Alpha 分量都会保持不变。从 sRGB 编码分量  $c_s$  到线性分量  $c_l$  的转换如下。

$$c_l = \begin{cases} \frac{c_s}{12.92} & \text{if } c_s \leq 0.04045 \\ \left( \frac{c_s + 0.055}{1.055} \right)^{2.4} & \text{if } c_s > 0.04045 \end{cases}$$

假设  $c_s$  为[0,1]范围内的 sRGB 分量。

使用 GL\_PROXY\_TEXTURE\_3D 目标来对分辨率和格式进行试验。实现将更新和重新计算它对要求的存储分辨率和格式的最佳匹配。要查询这个状态, 可以调用 glGetTexLevelParameter。如果纹理不能适应, 那么纹理状态将被设为 0。

一个单分量纹理图像只使用从 data 中提取的 RGBA 颜色的红色分量。一个 2 分量图像使用 R 和 A 值。一个 3 分量图像使用 R、G 和 B 值。一个 4 分量图像则使用所有的 RGBA 分量。

注意

glPixelStore 会影响纹理图像。

data 可以是一个空指针。在这种情况下, 将会分配纹理内存来容纳一个宽度为 width、高度为 height、深度为 depth 的纹理。随后我们就可以下载子纹理来对纹理内存进行初始化了。如果用户试图将纹理图像的一个未初始化部分应用到一个图元上, 那么这个图像将为未定义的。

glTexImage3D 为当前纹理单元 (由 glActiveTexture 指定) 指定三维纹理。

错误

如果 target 不是 GL\_TEXTURE\_3D 或 GL\_PROXY\_TEXTURE\_3D, 则产生 GL\_INVALID\_ENUM 错误。

如果 format 不是一个可接受的格式常量, 则产生 GL\_INVALID\_ENUM 错误。除 GL\_STENCIL\_INDEX 和 GL\_DEPTH\_COMPONENT 之外的格式常量都可以接受。

如果 type 不是一个可接受的类型常量, 则产生 GL\_INVALID\_ENUM 错误。

如果 level 小于 0, 则产生 GL\_INVALID\_VALUE 错误。

如果 level 大于  $\log_2 \text{max}$  值, 其中 max 是 GL\_MAX\_TEXTURE\_SIZE 的返回值, 则可能产生 GL\_INVALID\_VALUE 错误。

如果 internalFormat 不是一个可接受的分辨率和格式符号常量, 则产生 GL\_INVALID\_VALUE 错误。

如果 width、height 或 depth 小于 0 或大于 GL\_MAX\_TEXTURE\_SIZE, 则产生 GL\_INVALID\_VALUE 错误。

如果不支持非 2 的幂的纹理且 width、height 或 depth 不能被表示成  $2^k + 2$  (border) 的形式, 其中 k 为整数, 则产生 GL\_INVALID\_VALUE 错误。

如果 border 不是 0 或 1, 则产生 GL\_INVALID\_VALUE 错误。

如果 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 中的一个, 并且 format 不是 GL\_RGB, 则产生 GL\_INVALID\_OPERATION 错误。

如果 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\_REVGL\_UNSIGNED\_INT\_5\_9\_9\_9\_REV 中的一个, 并且 format 既不是 GL\_RGBA 也不是 GL\_BGRA, 则产生 GL\_INVALID\_OPERATION 错误。

如果 format 或 internalFormat 是 GL\_DEPTH\_COMPONENT、GL\_DEPTH\_COMPONENT16、GL\_DEPTH\_COMPONENT24 或 GL\_DEPTH\_COMPONENT32, 则产生 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标, 并且缓冲区对象的数据存储当前被映射, 则产生 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标, 并且数据将从缓冲区对象中被解包导致内存读取需求超出数据存储大小, 则产生 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标, 并且 data 没有平均分成将一个由 type 指定的数据存储到内存中所需的字节数, 则产生 GL\_INVALID\_OPERATION 错误。

#### 相关 Get 函数

glGetTexImage

glGet, 其自变量为 GL\_PIXEL\_UNPACK\_BUFFER\_BINDING。

#### 另外查看

glActiveTexture, glCompressedTexImage1D, glCompressedTexImage2D, glCompressedTexImage3D, glCompressedTexSubImage1D, glCompressedTexSubImage2D, glCompressedTexSubImage3D, glCopyTexImage1D, glCopyTexImage2D, glCopyTexSubImage1D, glCopyTexSubImage2D, glCopyTexSubImage3D, glGetCompressedTexImage, glPixelStore, glTexImage1D, glTexImage2D, glTexSubImage1D, glTexSubImage2D, glTexSubImage3D, glTexParameter

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glTexImage3DMultisample

建立多重采样纹理的图像的数据存储、格式、维度和样本数量。

### C 规范

```
void glTexImage3DMultisample(GLenum target,
                             GLsizei samples,
                             GLint internalformat,
                             GLsizei width,
                             GLsizei height,
                             GLsizei depth,
                             GLboolean fixedsamplelocations);
```

#### 参数

target

指定操作的目标。target 必须为 GL\_TEXTURE\_2D\_MULTISAMPLE\_ARRAY 或 GL\_PROXY\_TEXTURE\_2D\_MULTISAMPLE\_ARRAY。

samples

多重采样纹理图像中的样本数量。

internalformat

存储多重采样纹理图像所使用的内部格式。

Internalformat 必须指定一个颜色可渲染、深度可渲染或模板可渲染的格式。

width

多重采样纹理图像的宽度, 以纹理单元为单位。

height

多重采样纹理图像的高度, 以纹理单元为单位。

fixedsamplelocations

指定图像是否将要图像中的所有纹理单元使用同样的样本位置和同样的采样数, 并且样本位置不取决于内部格式或图像大小。

**描述**

`glTexImage3DMultisample` 建立多重采样纹理的图像的数据存储、格式、维度和样本数量。

`Target` 必须为 `GL_TEXTURE_2D_MULTISAMPLE_ARRAY` 或 `GL_PROXY_TEXTURE_2D_MULTISAMPLE_ARRAY`。`width` 和 `height` 为纹理中的纹理单元的维度，必须在从 0 到 `GL_MAX_TEXTURE_SIZE - 1` 的范围内。`depth` 是数组纹理的图像中的数组“切片”数量。`samples` 指定图像中采样数量，必须在从 0 到 `GL_MAX_SAMPLES - 1` 的范围之内。

`Internalformat` 必须是一个颜色可渲染、深度可渲染或模板可渲染的格式。

如果 `fixedsamplelocations` 为 `GL_TRUE`，图像将要对图像中的所有纹理单元使用同样的样本位置和同样的采样数，并且样本位置不取决于内部格式或图像大小。

当在一个着色器中访问一个多重采样纹理时，这个访问将采用一个整数向量来描述要获取哪个纹理单元，以及一个与采样数相对应的整数来描述获取纹理单元中的哪个样本。在多重采样纹理目标上不允许有标准采样指令。

**注意**

`glTexImage2DMultisample` 在 3.2 或更高版本的 GL 中可用。

**错误**

如果 `internalformat` 是一个深度可渲染或模板可渲染格式，并且 `samples` 大于或等于 `GL_MAX_DEPTH_TEXTURE_SAMPLES` 的值，则产生 `GL_INVALID_OPERATION` 错误。

如果 `internalformat` 是一个颜色可渲染格式，并且 `samples` 大于或等于 `GL_MAX_COLOR_TEXTURE_SAMPLES` 的值，则产生 `GL_INVALID_OPERATION` 错误。

如果 `internalformat` 是一个有符号或无符号的整数格式，并且 `samples` 大于或等于 `GL_MAX_INTEGER_SAMPLES` 的值，则产生 `GL_INVALID_OPERATION` 错误。

如果 `width` 或者 `height` 为负值或者大于 `GL_MAX_TEXTURE_SIZE`，则产生 `GL_INVALID_VALUE` 错误。

如果 `depth` 为负值或者大于 `GL_MAX_ARRAY_TEXTURE_LAYERS`，则产生 `GL_INVALID_VALUE` 错误。

如果 `samples` 大于 `GL_MAX_SAMPLES`，则产生 `GL_INVALID_VALUE` 错误。

**另外查看**

`glTexImage3D`, `glTexImage2DMultisample`

**版权**

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

**glTexParameter**

设置纹理参数。

**C 规范**

```
void glTexParameterf(GLenum target,
                    GLenum pname,
                    GLfloat param);

void glTexParameteri(GLenum target,
                    GLenum pname,
                    GLint param);
```

**参数**

`target`

指定目标纹理，必须为 `GL_TEXTURE_1D`、`GL_TEXTURE_2D`、`GL_TEXTURE_3D`、`GL_TEXTURE_1D_ARRAY`、`GL_TEXTURE_2D_ARRAY`、`GL_TEXTURE_RECTANGLE` 或 `GL_TEXTURE_CUBE_MAP`。

`pname`

指定一个单值纹理参数的符号名。`pname` 可以为下列值之一：`GL_TEXTURE_BASE_LEVEL`、`GL_TEXTURE_COMPARE_FUNC`、`GL_TEXTURE_COMPARE_MODE`、`GL_TEXTURE_LOD_BIAS`、`GL_TEXTURE_MIN_FILTER`、`GL_TEXTURE_MAG_FILTER`、`GL_TEXTURE_MIN_LOD`、`GL_TEXTURE_MAX_LOD`、`GL_TEXTURE_MAX_LEVEL`、`GL_TEXTURE_SWIZZLE_R`、`GL_TEXTURE_SWIZZLE_G`、`GL_TEXTURE_SWIZZLE_B`、`GL_TEXTURE_SWIZZLE_A`、`GL_TEXTURE_WRAP_S`、`GL_TEXTURE_WRAP_T` 或 `GL_TEXTURE_WRAP_R`。

`param`

指定 pname 的值。

### C 规范

```
void glTexParameterfv GLenum target GLenum pname const GLfloat * params
void glTexParameteriv GLenum target GLenum pname const GLint * params
void glTexParameterIiv GLenum target GLenum pname const GLint * params
void glTexParameterIuiv GLenum target GLenum pname const GLuint * params
```

#### 参数

##### target

指定目标纹理，必须为 GL\_TEXTURE\_1D、GL\_TEXTURE\_2D、GL\_TEXTURE\_3D、GL\_TEXTURE\_1D\_ARRAY、GL\_TEXTURE\_2D\_ARRAY、GL\_TEXTURE\_RECTANGLE 或 GL\_TEXTURE\_CUBE\_MAP。

##### pname

指定一个纹理参数的符号名。pname 可以为下列值之一：

GL\_TEXTURE\_BASE\_LEVEL、GL\_TEXTURE\_BORDER\_COLOR、GL\_TEXTURE\_COMPARE\_FUNC、GL\_TEXTURE\_COMPARE\_MODE、GL\_TEXTURE\_LOD\_BIAS、GL\_TEXTURE\_MIN\_FILTER、GL\_TEXTURE\_MAG\_FILTER、GL\_TEXTURE\_MIN\_LOD、GL\_TEXTURE\_MAX\_LOD、GL\_TEXTURE\_MAX\_LEVEL、GL\_TEXTURE\_SWIZZLE\_R、GL\_TEXTURE\_SWIZZLE\_G、GL\_TEXTURE\_SWIZZLE\_B、GL\_TEXTURE\_SWIZZLE\_A、GL\_TEXTURE\_SWIZZLE\_RGBA、GL\_TEXTURE\_WRAP\_S、GL\_TEXTURE\_WRAP\_T 或 GL\_TEXTURE\_WRAP\_R。

##### params

指定一个指向的一个或多个 pname 值所存储的数组的指针。

#### 描述

glTexParameter assigns the value or values in params to the texture parameter specified as pname. target 定义目标纹理，为 GL\_TEXTURE\_1D、GL\_TEXTURE\_2D、GL\_TEXTURE\_3D、GL\_TEXTURE\_1D\_ARRAY、GL\_TEXTURE\_2D\_ARRAY、GL\_TEXTURE\_RECTANGLE 或 GL\_TEXTURE\_3D。在 pname 中，下列符号都可以接受：

GL\_TEXTURE\_BASE\_LEVEL

指定最低定义 Mip 贴图层次的索引。这是一个整数值。初始值为 0。

GL\_TEXTURE\_BORDER\_COLOR

数据指定 4 个值，这 4 个值指定边界值指定应该用于边界纹理单元的边界值。如果一个纹理单元是从纹理的边缘采样的，那么 GL\_TEXTURE\_BORDER\_COLOR 的值将被解释为一个与纹理的内部格式匹配并代替并不存在的纹理单元数据的 RGBA 颜色。如果纹理包含深度分量，那么 GL\_TEXTURE\_BORDER\_COLOR 的第一个分量将被解释为一个深度值。初始值为

如果 GL\_TEXTURE\_BORDER\_COLOR 的值由 glTexParameteriv 或 glTexParameterIuiv 指定，那么这些值将不加修改地以一种整数内部数据类型进行存储。如果由 glTexParameteriv 指定，那么它们都会通过下面的方程转换为浮点值。

$$f = \frac{2_o + 1}{2_o - 1}$$

如果由 glTexParameterfv 指定，那么它们都会不加修改地作为浮点值进行存储。

GL\_TEXTURE\_COMPARE\_FUNC

指定当 GL\_TEXTURE\_COMPARE\_MODE 设置为 GL\_COMPARE\_REF\_TO\_TEXTURE 时使用的比较操作符。允许值为：

其中 r 为当前插值纹理坐标，而 Dt 则为从当前绑定的深度纹理中采样的深度纹理值。result 被分配给红色通道。

GL\_TEXTURE\_COMPARE\_MODE

为当前绑定的深度纹理指定纹理比较模式。也就是一个内部格式为 GL\_DEPTH\_COMPONENT\_\*；(参见 glTexImage2D) 的纹理。允许的值为：

GL\_COMPARE\_REF\_TO\_TEXTURE

指定进行插值和截取的纹理坐标应该与当前绑定深度纹理中的值进行比较。更多关于如何进行比较的细节参见关于 GL\_TEXTURE\_COMPARE\_FUNC 的讨论。比较的结果被分配到红色通道。

GL\_NONE

指定红色通道应该从当前绑定的深度纹理分配正确的值。

纹理比较函数	计算结果
GL_LEQUAL	$\text{Result} = \begin{cases} 1.0 & r \leq D_i \\ 0.0 & r > D_i \end{cases}$
GL_GEQUAL	$\text{Result} = \begin{cases} 1.0 & r \geq D_i \\ 0.0 & r < D_i \end{cases}$
GL_LESS	$\text{Result} = \begin{cases} 1.0 & r < D_i \\ 0.0 & r \geq D_i \end{cases}$
GL_GREATER	$\text{Result} = \begin{cases} 1.0 & r > D_i \\ 0.0 & r \leq D_i \end{cases}$
GL_EQUAL	$\text{Result} = \begin{cases} 1.0 & r = D_i \\ 0.0 & r \neq D_i \end{cases}$
GL_NOTEQUAL	$\text{Result} = \begin{cases} 1.0 & r \neq D_i \\ 0.0 & r = D_i \end{cases}$
GL_ALWAYS	Result = 1.0
GL_NEVER	Result = 0.0

GL\_TEXTURE\_LOD\_BIAS

params 指定一个固定偏置值，这个值在进行纹理采样之前将被加到纹理的层次细节参数上。这个指定值被加到着色器支持的偏置值（如果有的话），并且随后截取到实现定义的范围[bias<sub>max</sub>, bias<sub>max</sub>], 其中 bias<sub>max</sub>是实现定义的常量 GL\_MAX\_TEXTURE\_LOD\_BIAS 的值。初始值为 0.0。

GL\_TEXTURE\_MIN\_FILTER

当层次细节函数在从纹理进行采样时如果确定应该对纹理进行缩小，就会使用纹理缩小函数。一共有 6 个定义的缩小函数。其中两个使用最近的一个纹理元素或多个纹理元素的加权平均值来计算纹理值。其他 4 个使用 Mip 贴图。

Mip 贴图是一组排序的数组，代表同一个图像的一组分辨率逐渐降低的版本。如果图像的维度为 2<sup>n</sup>\*2<sup>m</sup>，那么就有 max(n,m)+1 个 Mip 贴图。

第一个 Mip 贴图是原始纹理，维度为 2<sup>n</sup>\*2<sup>m</sup>。每个后续 Mip 贴图的纹理为 2<sup>k-1</sup>\*2<sup>l-1</sup>，其中 2<sup>k-1</sup>\*2<sup>l-1</sup>是前一个 Mip 贴图的维度，直到 k=0 或 l=0 为止。这时，后续 Mip 贴图的维度为 1\*2<sup>l-1</sup>或 2<sup>k-1</sup>\*1，直到最后一个 Mip 贴图为止，其维度为 1\*1。可以调用以 level 参数指示 Mip 贴图顺序的 glTexImage1D、glTexImage2D、glTexImage3D、glCopyTexImage1D 或 glCopyTexImage2D 来定义 Mip 贴图。层次 0 是原始纹理，层次 max(n,m)是最后的 1\*1Mip 贴图。

params 支持下列函数来缩小纹理。

GL\_NEAREST

返回距离指定纹理坐标最近（曼哈顿距离）的纹理元素的值。

GL\_LINEAR

返回距离指定纹理坐标最近的四个纹理元素的加权平均值。其中可以包含另一部分纹理环绕或重复的项目，根据 GL\_TEXTURE\_WRAP\_S 和 GL\_TEXTURE\_WRAP\_T 的值和确切的映射而定。

GL\_NEAREST\_MIPMAP\_NEAREST

选择最接近地匹配进行纹理贴图的像素大小的 Mip 贴图，并使用 GL\_NEAREST 标准（与指定纹理坐标最近的纹理元素）来生成一个纹理值。

GL\_LINEAR\_MIPMAP\_NEAREST

选择最接近地匹配进行纹理贴图的像素大小的 Mip 贴图，并使用 GL\_LINEAR 标准（4 个与指定纹理坐标最近的纹理元素的加权平均值）来生成一个纹理值。

GL\_NEAREST\_MIPMAP\_LINEAR

选择最接近地匹配进行纹理贴图的像素大小的两个 Mip 贴图，并使用 GL\_NEAREST 标准（与指定纹理坐标最近的纹理元素）来从每个 Mip 贴图生成一个纹理值。最终的纹理值是这两个值的一个加权平均值。

GL\_LINEAR\_MIPMAP\_LINEAR

选择最接近地匹配进行纹理贴图的像素大小的两个 Mip 贴图，并使用 GL\_LINEAR 标准（这些与指定纹理坐标最

接近的纹理元素的加权平均值)来生成一个纹理值。最终的纹理值是这两个值的一个加权平均值。

在缩小处理过程中采样的纹理元素越多,出现的锯齿假影就越少。当 GL\_NEAREST 和 GL\_LINEAR 缩小函数能够比其他 4 个更快时,它们只对一个或者对多个纹理元素进行采样,以决定进行渲染的像素的纹理值,并且可以生成莫尔条纹 (moire pattern) 或不规则过渡 (ragged transitions)。GL\_TEXTURE\_MIN\_FILTER 的初始值为 GL\_NEAREST\_MIPMAP\_LINEAR。

GL\_TEXTURE\_MAG\_FILTER

当层次细节函数在从纹理进行采样时如果确定应该对纹理进行放大,就会使用纹理放大函数。它会将纹理放大函数设置为 GL\_NEAREST 或 GL\_LINEAR (如下所示)。

GL\_NEAREST 通常要比 GL\_LINEAR 快,但是它能够产生边缘更加锐利的纹理图像,因为纹理元素之间的过渡不那么平滑。

GL\_TEXTURE\_MAG\_FILTER 的初始值为 GL\_LINEAR。

GL\_NEAREST

返回距离指定纹理坐标最近 (曼哈顿距离) 的纹理元素的值。

GL\_LINEAR

返回距离指定纹理坐标最近的那些纹理元素的加权平均值。其中可以包含另一部分纹理环绕或重复的项目,根据 GL\_TEXTURE\_WRAP\_S 和 GL\_TEXTURE\_WRAP\_T 的值和确切的映射而定。

GL\_TEXTURE\_MIN\_LOD

设置最小层次细节参数。这个浮点值限制了最高分辨率 Mip 贴图 (最低层次的 Mip 贴图) 的选择。初始值为 -1000。

GL\_TEXTURE\_MAX\_LOD

设置最大层次细节参数。这个浮点值限制了最低分辨率 Mip 贴图 (最高层次的 Mip 贴图) 的选择。初始值为 1000。

GL\_TEXTURE\_MAX\_LEVEL

Sets the index of the highest defined mipmap level. This is an integer value. The initial value is 1000.

GL\_TEXTURE\_SWIZZLE\_R

设置将要在一个纹理单元的分量返回着色器之前在它上面进行应用的 swizzle 操作。param 的可用值为 GL\_RED、GL\_GREEN、GL\_BLUE、GL\_ALPHA、GL\_ZERO 和 GL\_ONE。如果 GL\_TEXTURE\_SWIZZLE\_R 为 GL\_RED,那么 r 的值将会从所获取的纹理单元的第二个通道获得。如果 GL\_TEXTURE\_SWIZZLE\_R 为 GL\_GREEN,那么 r 的值将会从所获取的纹理单元的第三个通道获得。如果 GL\_TEXTURE\_SWIZZLE\_R 为 GL\_BLUE,那么 r 的值将会从所获取的纹理单元的第四个通道获得。如果 GL\_TEXTURE\_SWIZZLE\_R 为 GL\_ALPHA,那么 r 的值将会从所获取的纹理单元的第五个通道获得。如果 GL\_TEXTURE\_SWIZZLE\_R 为 GL\_ZERO,那么 r 的值将由 0 代替。如果 GL\_TEXTURE\_SWIZZLE\_R 为 GL\_ONE,那么 r 的值将由 1 代替。初始值为 GL\_RED。

GL\_TEXTURE\_SWIZZLE\_G

设置将要在一个纹理单元的分量返回着色器之前在它上面进行应用的 swizzle 操作。param 的有效值和它们的效果与 GL\_TEXTURE\_SWIZZLE\_R 的情况类似。初始值为 GL\_GREEN。

GL\_TEXTURE\_SWIZZLE\_B

设置将要在一个纹理单元的分量返回着色器之前在它上面进行应用的 swizzle 操作。param 的有效值和它们的效果与 GL\_TEXTURE\_SWIZZLE\_R 的情况类似。初始值为 GL\_BLUE。

GL\_TEXTURE\_SWIZZLE\_A

设置将要在一个纹理单元的分量返回着色器之前在它上面进行应用的 swizzle 操作。param 的有效值和它们的效果与 GL\_TEXTURE\_SWIZZLE\_R 的情况类似。初始值为 GL\_ALPHA。

GL\_TEXTURE\_SWIZZLE\_RGBA

设置将要在一个纹理单元的分量返回着色器之前在它们上面进行应用的 swizzle 操作。params 的有效值和它们的效果与 GL\_TEXTURE\_SWIZZLE\_R 的情况类似,除了所有通道是同时指定的之外。

设置 GL\_TEXTURE\_SWIZZLE\_RGBA 的值相当于 (假设不生成任何错误) 连续地对 GL\_TEXTURE\_SWIZZLE\_R、GL\_TEXTURE\_SWIZZLE\_G、GL\_TEXTURE\_SWIZZLE\_B 和 GL\_TEXTURE\_SWIZZLE\_A 每一个的参数进行设置。

GL\_TEXTURE\_WRAP\_S

将纹理坐标环绕参数设置为 GL\_CLAMP\_TO\_EDGE、GL\_CLAMP\_TO\_BORDER、GL\_MIRRORED\_REPEAT 或 GL\_REPEAT。

GL\_CLAMP\_TO\_EDGE 会使坐标截取到  $[\frac{1}{2}, 1 - \frac{1}{2}]$  范围内,其中 N 是截取方向上纹理的大小。GL\_CLAMP\_TO\_BORDER 用一种与 GL\_CLAMP\_TO\_EDGE 类似的方式来对坐标进行估算。但是,在截取将要发生在 GL\_CLAMP\_TO\_EDGE 模式下的情况下,获取的纹理单元数据将由 GL\_TEXTURE\_BORDER\_COLOR 指定的值代替。GL\_REPEAT 导致坐标的整数部分被忽略;GL 只使用小数部分,从而创建一个重复模式。GL\_MIRRORED\_REPEAT

导致在  $s$  的整数部分为偶数时坐标被设置为纹理坐标的小数部分, 而在这个整数部分为奇数时纹理坐标则被设置为  $1 - \text{frac}(s)$ , 其中  $\text{frac}(s)$  代表  $s$  的小数部分。初始状态下, `GL_TEXTURE_WRAP_S` 被设置为 `GL_REPEAT`。

`GL_TEXTURE_WRAP_T` 将纹理坐标环绕参数设置为 `GL_CLAMP_TO_EDGE`、`GL_CLAMP_TO_BORDER`、`GL_MIRRORED_REPEAT` 或 `GL_REPEAT`。参见 `GL_TEXTURE_WRAP_S` 相关讨论。初始状态下, `GL_TEXTURE_WRAP_T` 被设置为 `GL_REPEAT`。

`GL_TEXTURE_WRAP_R`

将纹理坐标环绕参数设置为 `GL_CLAMP_TO_EDGE`、`GL_CLAMP_TO_BORDER`、`GL_MIRRORED_REPEAT` 或 `GL_REPEAT`。参见 `GL_TEXTURE_WRAP_S` 相关讨论。初始状态下, `GL_TEXTURE_WRAP_R` 被设置为 `GL_REPEAT`。

**注意**

假设一个程序试图从一个纹理上进行采样, 并已经将 `GL_TEXTURE_MIN_FILTER` 设置为一个需要 Mip 贴图的函数。如果当前定义 (通过前面的 `glTexImage1D`、`glTexImage2D`、`glTexImage3D`、`glCopyTexImage1D` 或 `glCopyTexImage2D` 调用) 的纹理图像的维度没有按照正确的序列 (如前所述), 或者定义的纹理图像比所需的数量少, 或者纹理图像集合的纹理分量数量不同, 那么就认为这个纹理是 *incomplete* (不完整) 的。

现行过滤只在 2D 纹理中访问这 4 个最接近的纹理元素。在 1D 纹理中, 线性过滤访问两个最接近的纹理元素。在 3D 纹理中, 线性过滤访问 8 个最接近的纹理元素。

`glTexParameter` 为活动的纹理单元 (通过调用 `glActiveTexture` 指定) 指定纹理参数。

**错误**

如果 `target` 或 `pname` 不是一个可接受的已定义值, 则产生 `GL_INVALID_ENUM` 错误。

如果 `params` 应该有一个定义的常数值 (基于 `pname` 的值) 但实际却不是这样, 则产生 `GL_INVALID_ENUM` 错误。

**相关 Get 函数**

`glGetTexParameter`

`glGetTexParameter`

**另外查看**

`glActiveTexture`, `glBindTexture`, `glCopyTexImage1D`, `glCopyTexImage2D`, `glCopyTexSubImage1D`, `glCopyTexSubImage2D`, `glCopyTexSubImage3D`, `glPixelStore`, `glSamplerParameter`, `glTexImage1D`, `glTexImage2D`, `glTexImage3D`, `glTexSubImage1D`, `glTexSubImage2D`, `glTexSubImage3D`

**版权**

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glTexSubImage1D

指定一个一维纹理子图像。

**C 规范**

```
void glTexSubImage1D(GLenum target,
                    GLint level,
                    GLint xoffset,
                    GLsizei width,
                    GLenum format,
                    GLenum type,
                    const GLvoid * data);
```

**参数**

`target`

指定目标纹理。必须为 `GL_TEXTURE_1D`。

`level`

指定层次细节数量。Level 0 是基本图像层次。Level  $n$  是第  $n$  级 Mip 贴图缩略图。

`xoffset`

指定纹理数组中  $x$  方向上一个 texel 的偏移。

`width`

指定纹理子图像的宽度。

format

指定像素数据的格式。下列符号值都可以接受:

GL\_RED、GL\_RG、GL\_RGB、GL\_BGR、GL\_RGBA 和 GL\_BGRA。

type

指定像素数据的数据类型。下列符号值都可以接受:

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。

data

指定到内存中图像数据的指针。

描述

纹理渲染将一个指定纹理图像的一部分映射到每个纹理渲染被激活的基本图元 ( graphical primitive ) 上。可以通过调用自变量为 GL\_TEXTURE\_1D 的 glEnable 或 glDisable 来激活和禁止一维纹理渲染。

glTexSubImage1D 重新定义一个已经存在的一维纹理图像的一个邻近的子区域。由 data 引用的 texel 替代已经存在的纹理数组中 x 索引从 xoffset 到 xoffset + width - 1 ( 包括边界 ) 的部分。这个区域可能不包括任何在纹理数组最初指定的范围之外的 texel。指定一个宽度为 0 的子纹理并不是错误,但是这样做不会产生任何效果。

如果在纹理图像被指定的情况下,非 0 的指定缓冲区对象被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标 (参见 glBindBuffer), data 将被看作缓冲区对象数据存储的一个字节偏移。

注意

glPixelStore 模式会影响纹理图像。

glTexSubImage1D 为当前纹理单元 ( 由 glActiveTexture 指定 ) 指定一维子纹理。

错误

如果 tarGet 不是一个允许值,则产生 GL\_INVALID\_ENUM 错误。

如果 format 不是一个可接受的格式常量,则产生 GL\_INVALID\_ENUM 错误。

如果 type 不是一个可接受的类型常量,则产生 GL\_INVALID\_ENUM 错误。

如果 level 小于 0,则产生 GL\_INVALID\_VALUE 错误。

如果 level 大于  $\log_2 \text{max\_val}$ , 其中 max 是 GL\_MAX\_TEXTURE\_SIZE 的返回值,则可能产生 GL\_INVALID\_VALUE 错误。

如果 xoffset - b 或 (xoffset + width) - b, 其中 w 是被修改的纹理图像的 GL\_TEXTURE\_WIDTH, 而 b 则是被修改的纹理图像的 GL\_TEXTURE\_BORDER 的宽度,则产生 GL\_INVALID\_VALUE 错误。注意 w 包含边界宽度的两倍。

如果 width 小于 0,则产生 GL\_INVALID\_VALUE 错误。

如果纹理数组没有被以前的 glTexImage1D 操作所定义,则产生 GL\_INVALID\_OPERATION 错误。

如果 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 中的一个,并且 format 不是 GL\_RGB,则产生 GL\_INVALID\_OPERATION 错误。

如果 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、GL\_UNSIGNED\_INT\_5\_9\_9\_9\_REV 中的一个,并且 format 既不是 GL\_RGBA 也不是 GL\_BGRA,则产生 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标,并且缓冲区对象的数据存储当前被映射,则产生 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标,并且数据将从缓冲区对象中被解包导致内存读取需求超出数据存储大小,则产生 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标,并且 data 没有平均分成将一个由 type 指定的数据存储到内存中所需的字节数,则产生 GL\_INVALID\_OPERATION 错误。

相关 Get 函数

glGetTexImage

glGet, 其自变量为 GL\_PIXEL\_UNPACK\_BUFFER\_BINDING。

#### 另外查看

glActiveTexture, glCopyTexImage1D, glCopyTexImage2D, glCopyTexSubImage1D, glCopyTexSubImage2D, glCopyTexSubImage3D, glPixelStore, glTexImage1D, glTexImage2D, glTexImage3D, glTexParameter, glTexSubImage2D, glTexSubImage3D

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glTexSubImage2D

指定一个二维纹理子图像。

### C 规范

```
void glTexSubImage2D(GLenum target,
                    GLint level,
                    GLint xoffset,
                    GLint yoffset,
                    GLsizei width,
                    GLsizei height,
                    GLenum format,
                    GLenum type,
                    const GLvoid * data);
```

#### 参数

target

指定目标纹理。必须为 GL\_TEXTURE\_2D、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_X、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Y、GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Z 或 GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Z。

level

指定层次细节数量。Level 0 是基本图像层次。Level n 是第 n 级 Mip 贴图缩略图。

xoffset

指定纹理数组中 x 方向上一个 texel 的偏移。

yoffset

指定纹理数组中 y 方向上一个 texel 的偏移。

width

指定纹理子图像的宽度。

height

指定纹理子图像的高度。

format

指定像素数据的格式。下列符号值都可以接受：

GL\_RED、GL\_RG、GL\_RGB、GL\_BGR、GL\_RGBA 和 GL\_BGRA。

type

指定像素数据的数据类型。下列符号值都可以接受：

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。

data

指定到内存中图像数据的指针。

#### 描述

纹理渲染将一个指定纹理图像的一部分映射到每个纹理渲染被激活的基本图元 (graphical primitive) 上。可以通

过调用自变量为 `GL_TEXTURE_2D` 的 `glEnable` 和 `glDisable` 来激活和禁止二维纹理渲染。

`glTexSubImage2D` 重新定义一个已经存在的二维纹理图像的一个邻近的子区域。由 `data` 引用的 `texel` 替代已经存在的纹理数组中 `x` 索引从 `xoffset` 到 `xoffset + width - 1` (包括边界), 以及 `y` 索引从 `yoffset` 到 `yoffset + height - 1` 的部分 (包括边界)。

这个区域可能不包括任何在纹理数组最初指定的范围之外的 `texel`。

指定一个宽度或高度为 0 的子纹理并不是错误, 但是这样做不会产生任何效果。

如果在纹理图像被指定的情况下, 非 0 的指定缓冲区对象被绑定到 `GL_PIXEL_UNPACK_BUFFER` 目标 (参见 `glBindBuffer`), `data` 将被看作缓冲区对象数据存储的一个字节偏移。

#### 注意

`glPixelStore` 模式会影响纹理图像。

`glTexSubImage2D` 为当前纹理单元 (由 `glActiveTexture` 指定) 指定二维子纹理。

#### 错误

如果 `target` 不是 `GL_TEXTURE_2D`、`GL_TEXTURE_CUBE_MAP_POSITIVE_X`、`GL_TEXTURE_CUBE_MAP_NEGATIVE_X`、`GL_TEXTURE_CUBE_MAP_POSITIVE_Y`、`GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`、`GL_TEXTURE_CUBE_MAP_POSITIVE_Z` 或 `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`, 则产生 `GL_INVALID_ENUM` 错误。

如果 `format` 不是一个可接受的格式常量, 则产生 `GL_INVALID_ENUM` 错误。

如果 `type` 不是一个可接受的类型常量, 则产生 `GL_INVALID_ENUM` 错误。

如果 `level` 小于 0, 则产生 `GL_INVALID_VALUE` 错误。

如果 `level` 大于 `log2 max`, 其中 `max` 是 `GL_MAX_TEXTURE_SIZE` 的返回值, 则可能产生 `GL_INVALID_VALUE` 错误。

如果 `xoffset < -b` 或 `(xoffset + width) > (w - b)`, 或者 `yoffset < -b` 或 `(yoffset + height) > (h - b)`, 其中 `w`、`b` 和 `h` 分别是被修改的纹理图像的 `GL_TEXTURE_WIDTH`、`GL_TEXTURE_BORDER` 和边界宽度, 则产生 `GL_INVALID_VALUE` 错误。注意 `w` 和 `h` 包含边界宽度的两倍。

如果 `width` 或 `height` 小于 0, 则产生 `GL_INVALID_VALUE` 错误。

如果纹理数组没有被以前的 `glTexImage2D` 操作所定义, 则产生 `GL_INVALID_OPERATION` 错误。

如果 `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` 中的一个, 并且 `format` 不是 `GL_RGB`, 则产生 `GL_INVALID_OPERATION` 错误。

如果 `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` 中的一个, 并且 `format` 既不是 `GL_RGBA` 也不是 `GL_BGRA`, 则产生 `GL_INVALID_OPERATION` 错误。

如果非 0 缓冲区对象名称被绑定到 `GL_PIXEL_UNPACK_BUFFER` 目标, 并且缓冲区对象的数据存储当前被映射, 则产生 `GL_INVALID_OPERATION` 错误。

如果非 0 缓冲区对象名称被绑定到 `GL_PIXEL_UNPACK_BUFFER` 目标, 并且数据将从缓冲区对象中被解包导致内存读取需求超出数据存储大小, 则产生 `GL_INVALID_OPERATION` 错误。

如果非 0 缓冲区对象名称被绑定到 `GL_PIXEL_UNPACK_BUFFER` 目标, 并且 `data` 没有平均分成将一个由 `type` 指定的数据存储到内存中所需的字节数, 则产生 `GL_INVALID_OPERATION` 错误。

#### 相关 Get 函数

`glGetTexImage`

`glGet`, 其自变量为 `GL_PIXEL_UNPACK_BUFFER_BINDING`。

#### 另外查看

`glActiveTexture`, `glCopyTexImage1D`, `glCopyTexImage2D`, `glCopyTexSubImage1D`, `glCopyTexSubImage2D`, `glCopyTexSubImage3D`, `glPixelStore`, `glTexImage1D`, `glTexImage2D`, `glTexImage3D`, `glTexSubImage1D`, `glTexSubImage3D`, `glTexParameter`

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glTexSubImage3D

指定一个三维纹理子图像。

### C 规范

```
void glTexSubImage3D(GLenum target,
                     GLint level,
                     GLint xoffset,
                     GLint yoffset,
                     GLint zoffset,
                     GLsizei width,
                     GLsizei height,
                     GLsizei depth,
                     GLenum format,
                     GLenum type,
                     const GLvoid * data);
```

### 参数

**target**

指定目标纹理。必须为 `GL_TEXTURE_3D`。

**level**

指定层次细节数量。Level 0 是基本图像层次。Level n 是第 n 级 Mip 贴图缩略图。

**xoffset**

指定纹理数组中 x 方向上一个 texel 的偏移。

**yoffset**

指定纹理数组中 y 方向上一个 texel 的偏移。

**zoffset**

指定纹理数组中 z 方向上一个 texel 的偏移。

**width**

指定纹理子图像的宽度。

**height**

指定纹理子图像的高度。

**depth**

指定纹理子图像的深度。

**format**

指定像素数据的格式。下列符号值都可以接受：

`GL_RED`、`GL_RG`、`GL_RGB`、`GL_BGR`、`GL_RGBA` 和 `GL_BGRA`。

**type**

指定像素数据的数据类型。下列符号值都可以接受：

`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`。

**data**

指定到内存中图像数据的指针。

### 描述

纹理渲染将一个指定纹理图像的一部分映射到每个纹理渲染被激活的基本图元 (graphical primitive) 上。可以通过调用自变量为 `GL_TEXTURE_3D` 的 `glEnable` 和 `glDisable` 来激活和禁止三维纹理渲染。

`glTexSubImage3D` 重新定义一个已经存在的三维纹理图像的一个邻近的子区域。由 data 引用的 texel 替代已经存在的纹理数组中 x 索引从 xoffset 到 xoffset + width - 1 (包括边界)，y 索引从 yoffset 到 yoffset + height - 1 的部分 (包括边界)，以及 z 索引从 zoffset 到 zoffset + depth - 1 的部分 (包括边界)。这个区域可能不包括任何在纹理数组最初指定的范围之外的 texel。指定一个宽度、高度或深度为 0 的子纹理并不是错误，但是这样做不会产生任何效果。

如果在纹理图像被指定的情况下, 非 0 的指定缓冲区对象被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标 (参见 glBindBuffer), data 将被看作缓冲区对象数据存储的一个字节偏移。

#### 注意

glPixelStore 模式会影响纹理图像。

glTexSubImage3D 为当前纹理单元 (由 glActiveTexture 指定) 指定三维子纹理。

#### 错误

如果 tarGet 不是 GL\_TEXTURE\_3D, 则产生 GL\_INVALID\_ENUM 错误。

如果 format 不是一个可接受的格式常量, 则产生 GL\_INVALID\_ENUM 错误。

如果 type 不是一个可接受的类型常量, 则产生 GL\_INVALID\_ENUM 错误。

如果 level 小于 0, 则产生 GL\_INVALID\_VALUE 错误。

如果 level 大于 log<sub>2</sub> max, 其中 max 是 GL\_MAX\_TEXTURE\_SIZE 的返回值, 则可能产生 GL\_INVALID\_VALUE 错误。

如果 xoffset < -b, (xoffset + width) > (w-b), yoffset < -b, (yoffset + height) > (h-b), zoffset < -b 或 (zoffset + depth) > (d-b), 其中 w, h, d 和 b 分别是被修改的纹理图像的 GL\_TEXTURE\_WIDTH、GL\_TEXTURE\_HEIGHT、GL\_TEXTURE\_DEPTH 和边界宽度, 则产生 GL\_INVALID\_VALUE 错误。注意 w, h 和 d 包含边界宽度的两倍。

如果 width、height 或 depth 小于 0, 则产生 GL\_INVALID\_VALUE 错误。

如果纹理数组没有被以前的 glTexImage3D 操作所定义, 则产生 GL\_INVALID\_OPERATION 错误。

如果 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 中的一个, 并且 format 不是 GL\_RGB, 则产生 GL\_INVALID\_OPERATION 错误。

如果 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、GL\_UNSIGNED\_INT\_5\_9\_9\_9\_REV 中的一个, 并且 format 既不是 GL\_RGBA 也不是 GL\_BGRA, 则产生 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标, 并且缓冲区对象的数据存储当前被映射, 则产生 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标, 并且数据将从缓冲区对象中被解包导致内存读取需求超出数据存储大小, 则产生 GL\_INVALID\_OPERATION 错误。

如果非 0 缓冲区对象名称被绑定到 GL\_PIXEL\_UNPACK\_BUFFER 目标, 并且 data 没有平均分成将由 type 指定的数据存储到内存中所需的字节数, 则产生 GL\_INVALID\_OPERATION 错误。

#### 相关 Get 函数

glGetTexImage

glGet, 其自变量为 GL\_PIXEL\_UNPACK\_BUFFER\_BINDING。

#### 另外查看

glActiveTexture, glCopyTexImage1D, glCopyTexImage2D, glCopyTexSubImage1D, glCopyTexSubImage2D, glCopyTexSubImage3D, glPixelStore, glGetTexImage1D, glGetTexImage2D, glGetTexImage3D, glGetTexSubImage1D, glGetTexSubImage2D, glGetTexParameter

#### 版权

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

## glTransformFeedbackVaryings

指定要记录到变换反馈缓冲区中的值。

### C 规范

```
void glTransformFeedbackVaryings(GLuint program,
                                GLsizei count,
                                const char **varyings,
                                GLenum bufferMode);
```

#### 参数

program

目标程序对象的名称。

count

用于进行变换反馈的 varying 变量的数量。

varyings

一个以 0 为结尾的 count 字符串数组指定用于变换反馈的 varying 变量的名称。

bufferMode

表示在变换反馈激活时用于捕获 varying 变量的模式。bufferMode 必须为 GL\_INTERLEAVED\_ATTRIBS 和 GL\_SEPARATE\_ATTRIBS。

描述

在变换反馈模式下进行记录的顶点着色器或几何图形着色器输出的名称由 glTransformFeedbackVaryings 指定。如果几何图形着色器是活动的,那么变换反馈将记录选定的几何图形着色器从输出顶点输出的变量的值。

否则,选定顶点着色器输出的值将被记录。

由 glTransformFeedbackVaryings 设置的状态将被存储,并且在下一次在 program 上调用 glLinkProgram 时生效。当调用 glLinkProgram 时,program 将进行连接,所以如果 bufferMode 为 L\_INTERLEAVED\_ATTRIBS,那么为 GL 生成的每个图元的顶点指定的 varying 变量值将被写入单个缓冲区;而如果 bufferMode 为 GL\_SEPARATE\_ATTRIBS,则会写入多重缓冲区。

除了 glTransformFeedbackVaryings 生成的错误之外,出现下列情况时 program 程序将会连接失败:

由 glTransformFeedbackVaryings 指定的 count 是非 0 的,但是这个程序对象没有任何顶点着色器或几何图形着色器。

任何在 varyings 数组中指定的变量名都不会作为顶点着色器(或几何图形着色器,如果激活的话)中的一个输出进行声明。

varyings 数组中的任意两个入口都指定同一个 varying 变量。

varyings 中任意 varying 变量中捕获的分量总数都大于常量 GL\_MAX\_TRANSFORM\_FEEDBACK\_SEPARATE\_COMPONENTS,缓冲区模式为 GL\_SEPARATE\_ATTRIBS。

捕获的分量总数都大于常量 GL\_MAX\_TRANSFORM\_FEEDBACK\_INTERLEAVED\_COMPONENTS,缓冲区模式为 GL\_INTERLEAVED\_ATTRIBS。

注意

glGetTransformFeedbackVarying 在 3.0 或更高版本的 GL 中可用。

错误

如果 program 不是一个程序对象的名称,则产生 GL\_INVALID\_VALUE 错误。

如果 bufferMode 为负值或者大于 GL\_SEPARATE\_ATTRIBS,并且 count 大于平台相关的限制 GL\_MAX\_TRANSFORM\_FEEDBACK\_SEPARATE\_ATTRIBS,则产生 GL\_INVALID\_VALUE 错误。

相关 Get 函数

glGetTransformFeedbackVarying

另外查看

glBeginTransformFeedback, glEndTransformFeedback, glGetTransformFeedbackVarying

版权

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glUniform

为当前程序对象指定一个统一变量的值。

**C 规范**

```
void glUniform1f(GLint location,
                 GLfloat v0);
void glUniform2f(GLint location,
                 GLfloat v0,
                 GLfloat v1);
void glUniform3f(GLint location,
                 GLfloat v0,
```

```

        GLfloat v1,
        GLfloat v2);
void glUniform4f(GLint location,
        GLfloat v0,
        GLfloat v1,
        GLfloat v2,
        GLfloat v3);
void glUniform1i(GLint location,
        GLint v0);
void glUniform2i(GLint location,
        GLint v0,
        GLint v1);
void glUniform3i(GLint location,
        GLint v0,
        GLint v1,
        GLint v2);
void glUniform4i(GLint location,
        GLint v0,
        GLint v1,
        GLint v2,
        GLint v3);
void glUniform1ui(GLint location,
        GLuint v0);
void glUniform2ui(GLint location,
        GLint v0,
        GLuint v1);
void glUniform3ui(GLint location,
        GLint v0,
        GLint v1,
        GLuint v2);
void glUniform4ui(GLint location,
        GLint v0,
        GLint v1,
        GLint v2,
        GLuint v3);

```

#### 参数

location

指定将要被修改的统一变量的位置。

v0, v1, v2, v3

指定特定统一变量将使用的新值。

#### C 规范

```

void glUniform1fv(GLint location,
        GLsizei count,
        const GLfloat * value);
void glUniform2fv(GLint location,
        GLsizei count,
        const GLfloat * value);
void glUniform3fv(GLint location,
        GLsizei count,
        const GLfloat * value);
void glUniform4fv(GLint location,
        GLsizei count,
        const GLfloat * value);
void glUniform1iv(GLint location,
        GLsizei count,
        const GLint * value);
void glUniform2iv(GLint location,
        GLsizei count,
        const GLint * value);

```



```

void glUniform3iv(GLint location,
                  GLsizei count,
                  const GLint * value);
void glUniform4iv(GLint location,
                  GLsizei count,
                  const GLint * value);
void glUniform1uiv(GLint location,
                  GLsizei count,
                  const GLuint * value);
void glUniform2uiv(GLint location,
                  GLsizei count,
                  const GLuint * value);
void glUniform3uiv(GLint location,
                  GLsizei count,
                  const GLuint * value);
void glUniform4uiv(GLint location,
                  GLsizei count,
                  const GLuint * value);

```

**参数**

location

指定将要被修改的统一变量的位置。

count

指定将要被修改的元素的数量。如果目标统一变量不是一个数组，那么这个值应该为 1。如果目标统一变量是一个数组，那么这个值应该为 1 或更多。

value

指定一个指向将用于更新指定统一变量的 count 值数组的指针。

**C 规范**

```

void glUniformMatrix2fv(GLint location,
                        GLsizei count,
                        GLboolean transpose,
                        const GLfloat * value);
void glUniformMatrix3fv(GLint location,
                        GLsizei count,
                        GLboolean transpose,
                        const GLfloat * value);
void glUniformMatrix4fv(GLint location,
                        GLsizei count,
                        GLboolean transpose,
                        const GLfloat * value);
void glUniformMatrix2x3fv(GLint location,
                          GLsizei count,
                          GLboolean transpose,
                          const GLfloat * value);
void glUniformMatrix3x2fv(GLint location,
                          GLsizei count,
                          GLboolean transpose,
                          const GLfloat * value);
void glUniformMatrix2x4fv(GLint location,
                          GLsizei count,
                          GLboolean transpose,
                          const GLfloat * value);
void glUniformMatrix4x2fv(GLint location,
                          GLsizei count,
                          GLboolean transpose,
                          const GLfloat * value);
void glUniformMatrix3x4fv(GLint location,
                          GLsizei count,
                          GLboolean transpose,

```

```

        const GLfloat * value);
void glUniformMatrix4x3fv(GLint location,
                           GLsizei count,
                           GLboolean transpose,
                           const GLfloat * value);

```

### 参数

location

指定将要被修改的统一变量的位置。

count

指定将要被修改的矩阵的数量。如果目标统一变量不是一个矩阵数组，那么这个值应该为 1。如果目标统一变量是一个矩阵数组，那么这个值应该为 1 或更多。

transpose

指定在矩阵的值被载入变量时是否要对矩阵进行变换。

value

指定一个指向将用于更新指定统一变量的 count 值数组的指针。

### 描述

glUniform 修改一个统一变量或统一变量数组的值。将要被修改的统一变量的位置由 location 指定，这是一个由 glGetUniformLocation 返回的值。glUniform 在通过调用 glUseProgram 而成为当前状态的一部分的程序对象上进行操作。

glUniform{1|2|3|4}{f|l|i|ui} 命令用于改变由 location 使用作为自变量传递的值而指定的统一变量的值。在这个命令中指定的数量应与指定统一变量（例如，对 float、int、unsigned int、bool 为 1，对 vec2、ivec2、uvec2、bvec2 为 2，等等）的数据类型中的分量数相匹配。后缀 f 指示将要被传递的浮点值；后缀 i 指示将要被传递的整数值；后缀 ui 指示将要被传递的无符号整数值，并且这个类型也应该与指定统一变量的数据类型相匹配。这个函数中的变量 i 应该用于为定义为 int、ivec2、ivec3、ivec4 或它们的数组的统一变量提供值。这个函数中的变量 ui 应该用于为定义为 unsigned int、uvec2、uvec3、uvec4 或它们的数组的统一变量提供值。变量 f 应该用于为定义为 float、vec2、vec3、vec4 或它们的数组的统一变量提供值。

无论是变量 i、变量 ui 还是变量 f 都可以用于为 bool、bvec2、bvec3、bvec4 类型或它们的数组的统一变量提供值。如果输入值为 0 或 0.0f，那么统一变量将被设为 false，而在其它情况下则将被设为 true。

所有在一个程序对象中定义的活动统一变量在程序对象被成功连接时都被初始化为 0。它们通过在下次成功连接操作发生在程序对象上（这时它们将再一次被初始化为 0）之前调用一次 glUniform 保留分配给它们的值。

glUniform{1|2|3|4}{f|l|i|ui}v 命令能够用于修改单个统一变量或一个统一变量数组。这些命令将一个计数和一个指向要被载入的值的指针传递到一个统一变量或一个统一变量数组。如果要改变一个单个的值，那么应该使用计数 1；而要修改一个整个或部分数组，则应使用计数 1 或更大的计数。在载入开始于一个统一变量数组的任意位置 m 的 n 个元素时，数组中的第 m + n - 1 个元素将被新的值所代替。如果 m + n - 1 大于统一变量数组的大小，那么超出数组末端的所有数组元素都将被忽略。在命令的名称中指定的数字代表 value 中每个元素的分量数量，并且它应该与指定统一变量（例如，对 float、int、bool 来说为 1，对 vec2、ivec2、bvec2 来说为 2，等等）的数据类型中的分量数相匹配。命令的名称中指定的数据类型必须与以前在 glUniform{1|2|3|4}{f|l|i|ui} 部分描述的指定统一变量相匹配。

对于统一变量数组，数组中的每个元素都被视为命令名称中指定的类型（例如，glUniform3f 或 glUniform3fv 能够被用于载入一个 vec3 类型的统一变量数组）。统一变量数组中将要被修改的元素的数量由 count 指定。

glUniformMatrix{2|3|4}{2x3|3x2|2x4|4x2|3x4|4x3}fv 命令用于修改一个矩阵或一个矩阵数组。命令名称中的数字被解释为矩阵的维度。数字 2 代表 2×2 矩阵（也就是说 4 个元素），数字 3 代表 3×3 矩阵（也就是说 9 个元素），而数字 4 则代表 4×4 矩阵（也就是说 16 个元素）。非正方形矩阵的维度将被明确表示，第一个数字代表类的数量，第二个数字则代表行的数量。举例来说，2×4 代表一个 2×4 矩阵，其列数为 2，行数为 4（也就是说 8 个元素）。如果 transpose 为 GL\_FALSE，那么每个矩阵将被假定为列主序。如果 transpose 为 GL\_TRUE，那么每个矩阵将被假定为行主序。自变量 count 指示将要被传递的矩阵数量。如果要修改一个单个矩阵的值，应该使用计数 1；大于 1 的计数可以用于修改一个矩阵数组。

### 注意

glUniform1i 和 glUniform1iv 是仅有的两个可以用于载入定义为采样器类型统一变量的函数。其它任何函数载入采样器都将导致一个 GL\_INVALID\_OPERATION 错误。

如果 count 大于 1，并且代表的统一变量不是一个数组，那么将产生一个 GL\_INVALID\_OPERATION 错误，并且指定统一变量将保持不变。

不同于前述例外,如果在着色器中定义的统一变量的类型和大小与在用于载入它的值的命令的名称中指定的类型和大小不匹配,那么将会产生一个 `GL_INVALID_OPERATION` 错误,并且指定统一变量将保持不变。

如果 `location` 为一个不为 -1 的值,并且它不代表当前程序对象中的一个有效统一变量位置,那么将会产生一个错误,并且当前程序对象的统一变量存储将不会发生任何改变。如果 `location` 等于 -1,那么传入的数据将被默认忽略,并且指定统一变量将不会被改变。

#### 错误

如果不存在当前程序对象,则产生 `GL_INVALID_OPERATION` 错误。

如果着色器中声明的统一变量的大小与 `glUniform` 命令指示的大小不匹配,则产生 `GL_INVALID_OPERATION` 错误。

如果这个函数的一个有符号或无符号整数变量用于载入一个 `float`、`vec2`、`vec3`、`vec4` 类型或它们的数组的统一变量,或者这个函数的一个浮点值变量用于载入一个 `int`、`ivec2`、`ivec3`、`ivec4`、`unsigned int`、`uvec2`、`uvec3`、`uvec4` 类型或它们的数组的统一变量,则产生 `GL_INVALID_OPERATION` 错误。

如果这个程序的一个有符号整数变量用于载入一个 `unsigned int`、`uvec2`、`uvec3`、`uvec4` 类型或它们的数组的统一变量,则产生 `GL_INVALID_OPERATION` 错误。

如果这个程序的一个无符号整数变量用于载入一个 `int`、`ivec2`、`ivec3`、`ivec4` 类型或它们的数组的统一变量,则产生 `GL_INVALID_OPERATION` 错误。

如果 `location` 是当前程序对象的一个无效的统一位置, `location` 不等于 -1,则产生 `GL_INVALID_OPERATION` 错误。

如果 `count` 小于 0,则产生 `GL_INVALID_VALUE` 错误。

如果 `count` 大于 1,并且代表的统一变量不是一个数组变量,那么将产生一个 `GL_INVALID_OPERATION` 错误。

如果一个采样器使用 `glUniform1i` 和 `glUniform1iv` 之外的一个命令进行加载,则产生 `GL_INVALID_OPERATION` 错误。

#### 相关 Get 函数

`glGet`, 其自变量为 `GL_CURRENT_PROGRAM`。

`glGetActiveUniform`, 其参数包含一个程序对象的句柄和一个活动统一变量的索引。

`glGetUniform`, 其参数包含一个程序对象的句柄和一个统一变量的位置。

`glGetUniformLocation`, 其参数包含一个程序对象的句柄和一个统一变量的名称。

#### 另外查看

`glLinkProgram`, `glUseProgram`

#### 版权

Copyright © 2003–2005 3Dlabs Inc. Ltd. Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。 <http://opencontent.org/openpub/>。

## glUniformBlockBinding

为一个活动统一块分配一个绑定点。

#### C 规范

```
void glUniformBlockBinding(GLuint program,
                           GLuint uniformBlockIndex,
                           GLuint uniformBlockBinding);
```

#### 参数

`program`

包含将要分配绑定的活动统一块的程序对象的名称。

`uniformBlockIndex`

包含 `program` 中将要分配绑定的活动统一块的索引。

`uniformBlockBinding`

指定将统一块绑定到 `program` 中 `uniformBlockIndex` 的绑定点。

#### 描述

使用 `glUniformBlockBinding` 分配的活动统一块的绑定点。一个程序的每个活动统一块都有一个对应的统一缓冲区绑定点。`Program` 是前面为其发出过 `glLinkProgram` 命令的一个程序对象的名称。

如果成功, `glUniformBlockBinding` 就会指定 `program` 将使用绑定到 `uniformBlockBinding` 绑定点的缓冲区对象

的数据存储来提取由 `uniformBlockIndex` 标识的统一块中的统一变量值。

当一个程序对象进行连接或重连接时, 分配给它的每一个活动统一块的统一缓冲区对象绑定点都会被重设为 0。

#### 错误

如果 `uniformBlockIndex` 不是 `program` 的一个活动统一块索引, 则产生 `GL_INVALID_VALUE` 错误。

如果 `uniformBlockBinding` 大于或等于 `GL_MAX_UNIFORM_BUFFER_BINDINGS` 的值, 则产生 `GL_INVALID_VALUE` 错误。

如果 `program` 不是一个由 GL 生成的程序对象的名称, 则产生 `GL_INVALID_VALUE` 错误。

#### 注意

`glUniformBlockBinding` 只在 3.1 或更高版本的 GL 中可用。

#### 相关 Get 函数

`glGetActiveUniformBlock`, 其自变量为 `GL_UNIFORM_BLOCK_BINDING`。

#### 另外查看

`glLinkProgram`, `glBindBufferBase`, `glBindBufferRange`, `glGetActiveUniformBlock`

#### 版权

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glUseProgram

将一个程序对象作为当前渲染状态的一部分进行安装。

#### C 规范

```
void glUseProgram(GLuint program);
```

#### 参数

`program`

指定程序对象的句柄, 这个程序对象的可执行文件将作为当前渲染状态的一部分来使用。

#### 描述

`glUseProgram` 将由 `program` 指定的程序对象作为当前渲染状态的一部分来进行安装。通过使用 `glAttachShader` 成功地将着色器对象绑定到程序对象、成功使用 `glCompileShader` 来编译这些着色器对象、成功使用 `glLinkProgram` 来连接这些着色器对象, 能够在这个程序对象中会创建一个或多个可执行文件。

如果一个程序对象包含一个或多个已经成功编译和连接的 `GL_VERTEX_SHADER` 类型的着色器对象, 那么这个程序对象将包含一个可执行文件, 这个可执行文件将在顶点处理器上运行。如果一个程序对象包含一个或多个已经成功编译和连接的 `GL_GEOMETRY_SHADER` 类型的着色器对象, 那么这个程序对象将包含一个可执行文件, 这个可执行文件将在几何图形处理器上运行。类似地, 如果一个程序对象包含一个或多个已经成功编译和连接的 `GL_FRAGMENT_SHADER` 类型的着色器对象, 那么这个程序对象将包含一个可执行文件, 这个可执行文件将在片段处理器上运行。

当一个程序对象正在使用时, 应用程序将可以修改绑定的着色器对象、编译绑定的着色器对象、绑定附加着色器对象, 以及分离或删除着色器对象。这些操作都不会影响作为当前状态一部分的可执行文件。但是, 对当前正在使用的程序对象进行重连接, 如果连接操作成功 (参见 `glLinkProgram`), 则会将这个程序对象安装为当前渲染状态的一部分。如果当前使用的程序对象重新连接不成功, 那么它的连接状态将被设定为 `GL_FALSE`, 但是可执行文件和相关状态将被保留为当前状态的一部分, 直到以后调用 `glUseProgram` 来使它不再使用。

在它移除不用之后, 它将不能再作为当前状态的一部分, 直到它被成功重连接为止。

如果 `program` 为 0, 那么当前渲染状态将会引用一个无效程序对象, 而着色器执行结果将为未定义的。但是, 并不是一个错误。

如果 `program` 不包含 `GL_FRAGMENT_SHADER` 类型的着色器对象, 那么一个可执行文件将会安装到顶点处理器或者也可能是几何图形着色器上, 但是片段着色器的执行结果将为未定义的。

#### 注意

就像显示列表和纹理对象一样, 程序对象的名字空间可以在一组环境之间进行共享, 只要环境的服务器端共享同一个地址空间就可以。如果名字空间在环境之间共享, 任何绑定的对象和这些绑定对象相关的数据也会被共享。

在对象被不同的执行线程访问时, 应用程序负责提供 API 调用之间的同步。

**错误**

如果 program 不是 0, 也不是由 OpenGL 产生的值, 则产生 GL\_INVALID\_VALUE 错误。

如果 program 不是一个程序对象, 则产生 GL\_INVALID\_OPERATION 错误。

如果 program 不能成为当前状态的一部分, 则产生 GL\_INVALID\_OPERATION 错误。

如果变换反馈模式被激活, 则产生 GL\_INVALID\_OPERATION 错误。

**相关 Get 函数**

glGet, 其自变量为 GL\_CURRENT\_PROGRAM。

glGetActiveAttrib, 其参数包含一个合法程序对象和一个活动属性变量的索引。

glGetActiveUniform, 其参数包含一个合法程序对象和一个活动统一变量的索引。

glGetAttachedShaders, 其参数包含一个合法程序对象。

glGetAttribLocation, 其参数包含一个合法程序对象和一个属性变量的名称。

glGetProgram, 其参数包含一个合法程序对象和将要查询的参数。

glGetProgramInfoLog, 其参数包含一个合法程序对象。

glGetUniform, 其参数包含一个合法程序对象和一个统一变量的位置。

glGetUniformLocation, 其参数包含一个合法程序对象和一个统一变量的名称。

glIsProgram

**另外查看**

glAttachShader, glBindAttribLocation, glCompileShader, glCreateProgram, glDeleteProgram, glDetachShader, glLinkProgram, glUniform, glValidateProgram, glVertexAttrib

**版权**

Copyright © 2003–2005 3Dlabs Inc. Ltd. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

**glValidateProgram**

验证一个程序对象。

**C 规范**

```
void glValidateProgram(GLuint program);
```

**参数**

program

指定将要被验证的程序对象的句柄。

**描述**

glValidateProgram 针对 program 中包含的可执行文件在给定当前 OpenGL 状态时是否能够执行而进行检查。验证过程生成的信息将被存储在 program 的信息日志中。验证信息可以包含一个空字符串, 或者可以是一个包含关于当前程序对象如何与其他 OpenGL 状态相互作用的信息的字符串。这就提供了一种方法, 使 OpenGL 实现能够传递更多关于当前程序为什么会发生无效、不理想或执行失败等情况的信息。

验证操作的状态将作为程序对象状态的一部分被存储。如果验证成功, 这个值将被设置为 GL\_TRUE, 否则将被设置为 GL\_FALSE。它可以通过调用自变量为 program 和 GL\_VALIDATE\_STATUS 的 glGetProgram 来进行查询。如果验证成功, 那么在给定当前状态的情况下 program 将被确保执行。否则, program 将被确保不执行。

在典型情况下, 这个函数只在应用程序开发过程中有用。存储在信息日志中的信息字符串是与实现完全相关的。因此, 一个应用程序不能期望不同的 OpenGL 实现能够生成同样的信息字符串。

**注意**

这个函数模拟了在可编程着色器作为当前状态时发出渲染命令的情况下 OpenGL 实现必须执行的验证操作。如果发生下述情况, GL\_INVALID\_OPERATION 错误将由任何执行一个触发几何图形渲染的命令生成。

当前程序对象中任意两个活动采样器为不同的类型, 但却引用同一个纹理图像单元;

在程序中活动采样器的数量超出了允许的纹理图像单元最大数量。

在渲染命令发出时, 应用程序捕获这些错误可能非常困难, 或者会导致性能下降。因此, 在应用程序开发过程中, 建议应用程序调用 glValidateProgram 来检测这些问题。

**错误**

如果 program 不是由 OpenGL 产生的值, 则产生 GL\_INVALID\_VALUE 错误。

如果 program 不是一个程序对象, 则产生 GL\_INVALID\_OPERATION 错误。

#### 相关 Get 函数

glGetProgram, 其自变量为 program 和 GL\_VALIDATE\_STATUS。

glGetProgramInfoLog, 其自变量为 program。

glIsProgram

#### 另外查看

glLinkProgram, glUseProgram

#### 版权

Copyright © 2003–2005 3Dlabs Inc. Ltd. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

### glVertexAttrib

指定一个通用顶点属性的值。

#### C 规范

```
void glVertexAttrib1f(GLuint index,
                     GLfloat v0);
void glVertexAttrib1s(GLuint index,
                     GLshort v0);
void glVertexAttrib1d(GLuint index,
                     GLdouble v0);
void glVertexAttrib1i(GLuint index,
                     GLint v0);
void glVertexAttrib1ui(GLuint index,
                     GLuint v0);
void glVertexAttrib2f(GLuint index,
                     GLfloat v0,
                     GLfloat v1);
void glVertexAttrib2s(GLuint index,
                     GLshort v0,
                     GLshort v1);
void glVertexAttrib2d(GLuint index,
                     GLdouble v0,
                     GLdouble v1);
void glVertexAttrib2i(GLuint index,
                     GLint v0,
                     GLint v1);
void glVertexAttrib2ui(GLuint index,
                     GLuint v0,
                     GLuint v1);
void glVertexAttrib3f(GLuint index,
                     GLfloat v0,
                     GLfloat v1,
                     GLfloat v2);
void glVertexAttrib3s(GLuint index,
                     GLshort v0,
                     GLshort v1,
                     GLshort v2);
void glVertexAttrib3d(GLuint index,
                     GLdouble v0,
                     GLdouble v1,
                     GLdouble v2);
void glVertexAttrib3i(GLuint index,
                     GLint v0,
                     GLint v1,
                     GLint v2);
```



```

void glVertexAttribI3ui(GLuint index,
                       GLint v0,
                       GLint v1,
                       GLint v2);
void glVertexAttrib4f(GLuint index,
                     GLfloat v0,
                     GLfloat v1,
                     GLfloat v2,
                     GLfloat v3);
void glVertexAttrib4s(GLuint index,
                     GLshort v0,
                     GLshort v1,
                     GLshort v2,
                     GLshort v3);
void glVertexAttrib4d(GLuint index,
                     GLdouble v0,
                     GLdouble v1,
                     GLdouble v2,
                     GLdouble v3);
void glVertexAttrib4Nub(GLuint index,
                       GLubyte v0,
                       GLubyte v1,
                       GLubyte v2,
                       GLubyte v3);
void glVertexAttribI4i(GLuint index,
                      GLint v0,
                      GLint v1,
                      GLint v2,
                      GLint v3);
void glVertexAttribI4ui(GLuint index,
                       GLuint v0,
                       GLuint v1,
                       GLuint v2,
                       GLuint v3);

```

**参数**

index

指定将要修改的一般顶点属性的索引。

v0, v1, v2, v3

指定特定顶点属性将使用的新值。

**C 规范**

```

void glVertexAttrib1fv(GLuint index,
                      const GLfloat * v);
void glVertexAttrib1sv(GLuint index,
                      const GLshort * v);
void glVertexAttrib1dv(GLuint index,
                      const GLdouble * v);
void glVertexAttribI1iv(GLuint index,
                      const GLint * v);
void glVertexAttribI1uiv(GLuint index,
                      const GLuint * v);
void glVertexAttrib2fv(GLuint index,
                      const GLfloat * v);
void glVertexAttrib2sv(GLuint index,
                      const GLshort * v);
void glVertexAttrib2dv(GLuint index,
                      const GLdouble * v);
void glVertexAttribI2iv(GLuint index,
                      const GLint * v);
void glVertexAttribI2uiv(GLuint index,

```

```

        const GLuint * v);
void glVertexAttrib3fv(GLuint index,
        const GLfloat * v);
void glVertexAttrib3sv(GLuint index,
        const GLshort * v);
void glVertexAttrib3dv(GLuint index,
        const GLdouble * v);
void glVertexAttribI3iv(GLuint index,
        const GLint * v);
void glVertexAttribI3uiv(GLuint index,
        const GLuint * v);
void glVertexAttrib4fv(GLuint index,
        const GLfloat * v);
void glVertexAttrib4sv(GLuint index,
        const GLshort * v);
void glVertexAttrib4dv(GLuint index,
        const GLdouble * v);
void glVertexAttrib4iv(GLuint index,
        const GLint * v);
void glVertexAttrib4bv(GLuint index,
        const GLbyte * v);
void glVertexAttrib4ubv(GLuint index,
        const GLubyte * v);
void glVertexAttrib4usv(GLuint index,
        const GLushort * v);
void glVertexAttrib4uiv(GLuint index,
        const GLuint * v);
void glVertexAttrib4Nbv(GLuint index,
        const GLbyte * v);
void glVertexAttrib4Nsv(GLuint index,
        const GLshort * v);
void glVertexAttrib4Niv(GLuint index,
        const GLint * v);
void glVertexAttrib4Nubv(GLuint index,
        const GLubyte * v);
void glVertexAttrib4Nusv(GLuint index,
        const GLushort * v);
void glVertexAttrib4Nuiv(GLuint index,
        const GLuint * v);
void glVertexAttribI4bv(GLuint index,
        const GLbyte * v);
void glVertexAttribI4ubv(GLuint index,
        const GLubyte * v);
void glVertexAttribI4sv(GLuint index,
        const GLshort * v);
void glVertexAttribI4usv(GLuint index,
        const GLushort * v);
void glVertexAttribI4iv(GLuint index,
        const GLint * v);
void glVertexAttribI4uiv(GLuint index,
        const GLuint * v);

```

**参数**

index

指定将要修改的一般顶点属性的索引。

v

指定一个指向通用顶点属性将要使用的值的数组的指针。

**C 规范**

```

void glVertexAttribPui(GLuint index,
        GLenum type,

```

PDF

```

        GLboolean normalized,
        GLuint value);
void glVertexAttribP2ui(GLuint index,
        GLenum type,
        GLboolean normalized,
        GLuint value);
void glVertexAttribP3ui(GLuint index,
        GLenum type,
        GLboolean normalized,
        GLuint value);
void glVertexAttribP4ui(GLuint index,
        GLenum type,
        GLboolean normalized,
        GLuint value);

```

**参数****index**

指定将要修改的一般顶点属性的索引。

**type**

数据使用的包装类型要指定有符号或无符号数据，这个参数必须分别为 GL\_INT\_10\_10\_10\_2 或 GL\_UNSIGNED\_INT\_10\_10\_10\_2。

**normalized**

如果为 GL\_TRUE，那么这些值将通过标准化被转换为浮点值。

否则，它们将被直接转换为浮点值。

**value**

指定特定顶点属性将使用的新包装值。

**描述**

OpenGL 定义很多标准顶点属性，应用程序可以通过标准 API 入口点（颜色、法线、纹理坐标等）修改它们。入口点的 glVertexAttrib 族允许一个应用程序将通用顶点属性传入已编号的位置。

通用属性定义为并入一个数组的 4 分量值。这个数组的第一个入口计数为 0，数组的大小则由实现相关的常量 GL\_MAX\_VERTEX\_ATTRIBS 来指定。这个数组的单个元素可以通过调用一次指定了这个将要被修改的元素的索引和这个元素的值的 glVertexAttrib 来进行修改。

这些命令可以被用来指定由 index 指定的通用顶点属性的 1 个、2 个、3 个或所有 4 个分量。命令名称中的 1 指示只有一个值被传递，并且这个值将被用来修改通用顶点属性的第 1 个分量。第 2 个和第 3 个分量将被设置为 0，而第 4 个分量将被设置为 1。同样地，命令名称中的 2 指示提供了前两个分量的值，第 3 个分量将被设置为 0，而第 4 个分量将被设置为 1。命令名称中的 3 指示提供了前 3 个分量的值，第 4 个分量将被设置为 1。命令名称中的 3 指示为所有 4 个分量都提供了值。

字母 s、f、i、d、ub、us 和 ui 指示自变量是否为 short、float、int、double、unsigned byte、unsigned short 或 unsigned int 类型。当名称中添加了 v 时，命令可以接受一个指向这些值的数组的指针。

附加的大写字母能够指示对 glVertexAttrib 函数默认行为的进一步修改。

包含 N 的命令指示自变量将作为根据 OpenGL 规范定义的分量转换规则缩放到标准化范围定点值被传递。有符号值将被理解为在 [-1,1] 范围内表示定点值，而无符号值将被理解为在 [0,1] 范围内表示定点值。

包含 i 的命令指示自变量将扩展为完整的有符号或无符号整数。

包含 P 的命令指示自变量将存储为一个更大的自然类型中的包装分量。

OpenGL 着色语言属性变量允许为 mat2、mat3 或 mat4 类型。

这些类型的属性可以使用 glVertexAttrib 入口点载入。矩阵必须以列主序被载入到连续的通用属性段槽内，每个通用属性槽内有矩阵的一个列。

一个在顶点着色器中声明的用户定义属性变量能够通过调用 glBindAttribLocation 来绑定到一个通用属性索引上。这样就允许一个应用程序来使用更多在顶点着色器中指定的描述性的变量。对指定通用顶点属性的后续修改将立即映射为一个对顶点着色器中相应属性变量的修改。

一个通用顶点属性索引和顶点着色器中的一个用户定义属性变量之间的绑定是一个程序对象状态的一部分，但是通用顶点属性的当前值则不是。就像标准顶点属性一样，每个通用顶点属性的值都是当前状态的一部分，并且即使是在使用一个不同的程序对象时它也将被保留。

应用程序可以自由地修改没有被绑定到指定顶点着色器属性变量的通用顶点属性。这些值只是简单地作为当前状态的一部分保留，并且不能被顶点着色器访问。如果一个绑定到顶点着色器中的一个属性变量的通用顶点属性在顶点着色器执行时没有被更新，那么顶点着色器将为通用顶点属性重复使用当前值。

#### 注意

通用顶点属性可以在任何时刻进行更新。

应用程序将一个以上的属性名绑定到同一个通用顶点属性索引是可能的。这称为混叠 (aliasing)，只有在顶点着色器中只有一个混叠属性变量是活动的情况下，或在没有通过顶点着色器中的路径能消耗混叠到同一区域的多于 1 个属性的情况下才被允许。OpenGL 实现不是必需进行错误检查来检测混叠，它们被允许假定混叠不会发生，并且被允许采用只在不存在混叠时有效的优化。

这里没有准备绑定标准顶点属性，因此，将通用属性化名为标准属性是不可能的。

glVertexAttrib4bv、glVertexAttrib4sv、glVertexAttrib4iv、glVertexAttrib4ubv、glVertexAttrib4usv、glVertexAttrib4uiv 和 glVertexAttrib4N 版本只在 3.1 或更高版本的 GL 中可用。

glVertexAttribP 版本只在 3.3 或更高版本的 GL 中可用。

#### 错误

如果 index 大于或等于 GL\_MAX\_VERTEX\_ATTRIBS，则产生 GL\_INVALID\_VALUE 错误。

如果 glVertexAttribP 用于除 GGL\_INT\_10\_10\_10\_2 或 GL\_UNSIGNED\_INT\_10\_10\_10\_2 以外的任何值，则产生 GL\_INVALID\_ENUM 错误。

#### 相关 Get 函数

glGet，其自变量为 GL\_CURRENT\_PROGRAM。

glGetActiveAttrib，其自变量为 program 和一个活动属性变量的索引。

glGetAttribLocation，其自变量为 program 和一个属性变量的名称。

glGetVertexAttrib，其自变量为 GL\_CURRENT\_VERTEX\_ATTRIB 和 index。

#### 另外查看

glBindAttribLocation, glVertexAttribPointer

#### 版权

Copyright © 2003–2005 3Dlabs Inc. Ltd. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

## glVertexAttribDivisor

指定通用顶点属性在实例渲染中更新的速度。

### C 规范

```
void glVertexAttribDivisor(GLuint index,
                          GLuint divisor);
```

#### 参数

index

指定通用顶点属性的索引。

divisor

指定 index 槽中通用属性相邻两次更新之间通过的实例数量。

#### 描述

glVertexAttribDivisor 修改通用顶点属性在一次绘制调用中的图元实例进行渲染时实例渲染中更新的速度。如果 divisor 为 0，那么 index 槽中的属性对每个顶点更新一次。果如 divisor 非 0，那么 index 槽中的属性对进行渲染的一组或几组顶点的 divisor 个实例更新一次。如果一个属性的 GL\_VERTEX\_ATTRIB\_ARRAY\_DIVISOR 值是非 0 的，那么它将作为一个实例进行引用。

index 必须小于 GL\_MAX\_VERTEX\_ATTRIBUTES 的值。

#### 注意

glVertexAttribDivisor 只在 3.3 或更高版本的 GL 中可用。

#### 错误

如果 index 大于或等于 GL\_MAX\_VERTEX\_ATTRIBUTES 的值，则产生 GL\_INVALID\_VALUE 错误。

**另外查看**

glVertexAttribPointer, glEnableVertexAttribArray, glDisableVertexAttribArray

**版权**

Copyright © 2010 Khronos Group. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。http://opencontent.org/openpub/。

**glVertexAttribPointer**

定义一个通用顶点属性数据的数组。

**C 规范**

```
void glVertexAttribPointer(GLuint index,
                           GLint size,
                           GLenum type,
                           GLboolean normalized,
                           GLsizei stride,
                           const GLvoid * pointer);

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

**参数**

index

指定将要修改的一般顶点属性的索引。

size

返回每个通用顶点属性指针的分量数。必须为 1、2、3 或 4。

另外, glVertexAttribPointer 接受符号常量 GL\_BGRA。

初始值为 4。

type

指定数组中每个分量的数据类型。符号常量 GL\_BYTE、GL\_UNSIGNED\_BYTE、GL\_SHORT、GL\_UNSIGNED\_SHORT、GL\_INT、GL\_UNSIGNED\_INT、GL\_FLOAT 或 GL\_DOUBLE 都可以被这些函数接受。此外 GL\_HALF\_FLOAT、GL\_FLOAT、GL\_DOUBLE、GL\_INT\_2\_10\_10\_10\_REV 和 GL\_UNSIGNED\_INT\_2\_10\_10\_10\_REV 都可以被 glVertexAttribPointer 接受。初始值为 GL\_FLOAT。

normalized

对于 glVertexAttribPointer, 指定定点数据值在被访问时是否应该被标准化 (GL\_TRUE), 还是直接作为定点值进行转换 (GL\_FALSE)。

stride

指定连续通用顶点属性之间的字节偏移。如果 stride 为 0, 那么通用顶点属性将被理解是要被紧密打包到数组中。初始值为 0。

pointer

指定一个指向当前绑定到 GL\_ARRAY\_BUFFER 目标的缓冲区的数据存储中的数组中第一个通用顶点属性的第一个分量的指针。初始值为 0。

**描述**

glVertexAttribPointer 和 glVertexAttribIPointer 指定在渲染时将要使用的索引为 index 的通用顶点属性数组的位置和数据格式。

size 指定每个属性的分量数, 必须为 1、2、3、4 或 GL\_BGRA。

type 指定每个分量的数据类型, 而 stride 指定从一个属性到下一个属性的字节步长, 允许顶点和属性被打包到一个单个数组或存储到各自的数组。

对于 glVertexAttribPointer, 如果 normalized 被设置为 GL\_TRUE, 那么它就指示以一种整数格式存储的值在它们被访问和转换到浮点格式时将被映射到 [-1,1] 范围 (对于有符号值) 或 [0,1] 范围 (对于无符号值)。否则这些值将被直接转换成浮点数而不进行标准化。

对于 glVertexAttribIPointer, 只有整数类型 GL\_BYTE、GL\_UNSIGNED\_BYTE、GL\_SHORT、GL\_UNSIGNED\_

SHORT、GL\_INT、GL\_UNSIGNED\_INT、GL\_FLOAT 或 GL\_DOUBLE 可以被接受。这些值总是保持为整数值。

如果 pointer 不为 NULL，一个非 0 的指定缓冲区对象必须被绑定到 GL\_ARRAY\_BUFFER 目标 (参见 glBindBuffer)，否则就会产生一个错误。pointer 将被看作缓冲区对象数据存储的一个字节偏移。缓冲区对象绑定 (GL\_ARRAY\_BUFFER\_BINDING) 也被作为索引为 index 的通用顶点属性数组状态 (GL\_VERTEX\_ATTRIB\_ARRAY\_BUFFER\_BINDING) 而进行保存。

当一个通用顶点属性数组被指定时，除了当前顶点数组缓冲区对象绑定之外，size、type、normalized、stride 和 pointer 也都将保存为顶点数组状态。

要激活和禁止一个通用顶点属性数组，可以调用以 index 为参数的 glEnableVertexAttribArray 和 glDisableVertexAttribArray。如果被激活，则通用顶点属性数组将在 glDrawArrays、glMultiDrawArrays、glDrawElements、glMultiDrawElements 或 glDrawRangeElements 被调用时使用。

#### 注意

每个通用顶点属性数组在初始条件下都是被禁止的，并且在 glDrawElements、glDrawRangeElements、glDrawArrays、glMultiDrawArrays 或 glMultiDrawElements 被调用时不能被访问。

glVertexAttribPointer 只在 3.0 或更高版本的 GL 中可用。

#### 错误

如果 index 大于或等于 GL\_MAX\_VERTEX\_ATTRIBS，则产生 GL\_INVALID\_VALUE 错误。

如果 size 不是 1、2、3、4 或 (对于 glVertexAttribPointer) GL\_BGRA，则产生 GL\_INVALID\_VALUE 错误。

如果 type 不是一个可接受的值，则产生 GL\_INVALID\_ENUM 错误。

如果 stride 为负值，则产生 GL\_INVALID\_VALUE 错误。

如果 size 为 GL\_BGRA，并且 type 不为 GL\_INT\_2\_10\_10\_10\_REV 或 GL\_UNSIGNED\_INT\_2\_10\_10\_10\_REV，则产生 GL\_INVALID\_OPERATION 错误。

如果 type 为 GL\_INT\_2\_10\_10\_10\_REV 或 GL\_UNSIGNED\_INT\_2\_10\_10\_10\_REV，并且 size 不为 4 或 GL\_BGRA，则产生 GL\_INVALID\_OPERATION 错误。

如果 size 为 GL\_BGRA，并且 normalized 为 GL\_FALSE，则 glVertexAttribPointer 将产生 GL\_INVALID\_OPERATION 错误。

如果 0 被绑定到 GL\_ARRAY\_BUFFER 缓冲区对象绑定点，并且自变量 pointer 不为 NULL，则产生 GL\_INVALID\_OPERATION 错误。

#### 相关 Get 函数

glGet，其自变量为 GL\_MAX\_VERTEX\_ATTRIBS。

glGetVertexAttrib，其自变量为 index 和 GL\_VERTEX\_ATTRIB\_ARRAY\_ENABLED。

glGetVertexAttrib，其自变量为 index 和 GL\_VERTEX\_ATTRIB\_ARRAY\_SIZE。

glGetVertexAttrib，其自变量为 index 和 GL\_VERTEX\_ATTRIB\_ARRAY\_TYPE。

glGetVertexAttrib，其自变量为 index 和 GL\_VERTEX\_ATTRIB\_ARRAY\_NORMALIZED。

glGetVertexAttrib，其自变量为 index 和 GL\_VERTEX\_ATTRIB\_ARRAY\_STRIDE。

glGetVertexAttrib，其自变量为 index 和 GL\_VERTEX\_ATTRIB\_ARRAY\_BUFFER\_BINDING。

glGet，其自变量为 GL\_ARRAY\_BUFFER\_BINDING。

glGetVertexAttribPointerv，其自变量为 index 和 GL\_VERTEX\_ATTRIB\_ARRAY\_POINTER。

#### 另外查看

glBindAttribLocation, glBindBuffer, glDisableVertexAttribArray, glDrawArrays, glDrawElements, glDrawRangeElements, glEnableVertexAttribArray, glMultiDrawArrays, glMultiDrawElements, glVertexAttrib

#### 版权

Copyright © 2003–2005 3Dlabs Inc. Ltd. 这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

## glViewport

设置视口。

#### C 规范

```
void glViewport(GLint x,
               GLint y,
```

```
GLsizei width,
GLsizei height);
```

**参数**

x

y

指定视口矩形的左下角，以像素为单位。初始值为(0,0)。

width

height

指定视口的宽度和高度。当一个 GL 环境第一次被绑定到一个窗口时，width 和 height 将被设置为这个窗口的尺寸。

**描述**

glViewport 指定 x 和 y 从标准化设备坐标到窗口坐标的仿射变换 (affine transformation)。令  $(x_{nd}, y_{nd})$  为标准化的设备坐标，那么窗口坐标  $(x_w, y_w)$  将如下所示进行计算。

$$x_w = (x_{nd} + 1) \left( \frac{\text{width}}{2} \right) + x$$

$$y_w = (y_{nd} + 1) \left( \frac{\text{height}}{2} \right) + y$$

视口宽度和高度将被默认限制到一个范围，这个范围根据具体实现而确定。

要查询这个范围，可以调用自变量为 GL\_MAX\_VIEWPORT\_DIMS 的 glGet。

**错误**

如果 width 或者 height 为负值，则产生 GL\_INVALID\_VALUE 错误。

**相关 Get 函数**

glGet，其自变量为 GL\_VIEWPORT。

glGet，其自变量为 GL\_MAX\_VIEWPORT\_DIMS。

**另外查看**

glDepthRange

**版权**

Copyright © 1991–2006 Silicon Graphics, Inc. 本文档由 SGI Free Software B License 授权。相关细节参见 <http://oss.sgi.com/projects/FreeB/>。

**glWaitSync**

命令 GL 服务器进行阻塞，直到指定同步对象成为标记状态。

**C 规范**

```
void glWaitSync(GLsync sync,
               GLbitfield flags,
               GLuint64 timeout);
```

**参数**

sync

指定将要等待其状态的同步对象。

flags

一个控制命令清理行为的位段。flags 可能为 0。

timeout

指定服务器在继续运行之前应该等待的超时时间。timeout 必须为 GL\_TIMEOUT\_IGNORED。

**描述**

glWaitSync 会使服务器阻塞并等待，直到 sync 变为标记状态。sync 是所等待的一个已存在的同步对象的名称。flags 和 timeout 当前没有使用，并且必须分别被设为 0 和特殊值 GL\_TIMEOUT\_IGNORED。glWaitSync 的等待时间总是不大于一个平台相关的超时时间。这个超时以纳秒为单位的时间长度可以通过调用以 GL\_MAX\_SERVER\_WAIT\_TIMEOUT 为参数的 glGet 进行查询。

目前还没有任何方法可以确定 glWaitSync 是否是因为达到超时时间还是因为等待的同步对象变为标记状态而解

除阻塞的。

如果出现一个错误，那么 `glWaitSync` 就不会使 GL 服务器进行阻塞。

**注意**

`glWaitSync` 只在 3.2 或更高版本的 GL 中可用。

**错误**

如果 `sync` 不是一个同步对象的名称，则产生 `GL_INVALID_OPERATION` 错误。

如果 `flags` 为非 0 值，则产生 `GL_INVALID_VALUE` 错误。

如果 `timeout` 不是 `GL_TIMEOUT_IGNORED`，则产生 `GL_INVALID_VALUE` 错误。

**另外查看**

`glFenceSync`, `glClientWaitSync`

**版权**

Copyright © 2010 Khronos Group。这些材料可以按照 Open Publication License, v 1.0, 8 June 1999 中规定的条款和条件进行发布。<http://opencontent.org/openpub/>。

