

华章程序员书库

OpenGL 编程指南 (原书第 8 版)

OpenGL Programming Guide: The Official Guide to Learning
OpenGL, Version 4.3, Eighth Edition

(美) Dave Shreiner

Graham Sellers

John Kessenich

Bill Licea-Kane 著

王锐 等译

HZ BOOKS

华章图书



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

OpenGL 编程指南 (原书第 8 版) / (美) 施莱尔 (Shreiner, D.) 等著; 王锐等译. —北京: 机械工业出版社, 2014.10

(华章程序员书库)

书名原文: OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3, Eighth Edition

ISBN 978-7-111-48113-3

I.O… II. ① 施… ② 王… III. 图形软件—指南 IV. TP391.41-62

中国版本图书馆 CIP 数据核字 (2014) 第 226183 号

本书版权登记号: 图字: 01-2013-4455

Authorized translation from the English language edition, entitled OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3, 8E, 978-0-321-77303-6 by Dave Shreiner, Graham Sellers, John Kessenich, Bill Licea-Kane, published by Pearson Education, Inc., Copyright © 2013.

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 China Machine Press Copyright © 2014.

本书中文简体字版由 Pearson Education (培生教育出版集团) 授权机械工业出版社在中华人民共和国境内 (不包括中国台湾地区和香港、澳门特别行政区) 独家出版发行。未经出版者书面许可, 不得以任何方式抄袭、复制或节录本书中的任何部分。

本书封底贴有 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 编程指南 (原书第 8 版)

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 秦 健

责任校对: 董纪丽

印 刷:

版 次: 2014 年 10 月第 1 版第 1 次印刷

开 本: 186mm×240mm 1/16

印 张: 41.75

书 号: ISBN 978-7-111-48113-3

定 价: 129.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88378991 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzjsj@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

Praise 推荐语

“这是一本一站式服务的 OpenGL 书籍。它就是我梦寐以求的那种图书。感谢 Dave、Graham、John 和 Bill，感谢你们作出的了不起的贡献。”

——Mike Bailey，俄勒冈州立大学教授

“最近出版的这本红宝书依然遵循了 OpenGL 的伟大传统：不断进化让它拥有了更为强大的力量和效率。第 8 版包含了最前沿的接口标准和新特性的内容，以及对于应用在各行各业的现代 OpenGL 技术的脚踏实地的讲解。红宝书依然是我的公司中所有新员工的必备参考书。还有其他任何一本书可以说的上是必备的指南书吗？它让我喜极而泣，让我觉得无与伦比——我会一遍又一遍地阅读这本书。”

——Bob Kuehne，Blue Newt Software 总裁

“OpenGL 在这 20 年来已经有了巨大的发展。这次的修订版是一本学习使用现代 OpenGL 的实用指南书。现代 OpenGL 侧重于着色器的使用，而这一版的编程指南准确地对应了这一点，它在第 2 章对于着色器进行了深入的叙述。而后继的章节里，它继续深入到方方面面，从纹理到计算着色器。无论你对 OpenGL 了解多少，或者你准备深入到何种程度，只要你准备开始编写 OpenGL 程序，你就一定需要《OpenGL 编程指南》这本手边书。”

——Marc Olano，UMBC 副教授

“如果你正在寻找有关 OpenGL 最新版的编程权威指南，那么你已经找到了。本书的作者深入参与了 OpenGL 4.3 标准的创立，而这本书中恰恰包含了你所需要了解的一切，它将使用一种清晰的、富有逻辑性和见解性的方式，介绍这个行业领先的 API 标准的最新知识。”

——Neil Trevett，Khronos Group 总裁

译者序 *The Translator's Words*

OpenGL 发展至今，已经超过了 20 年的时间。作为一个成熟而久负盛名的跨平台的计算机图形应用程序接口规范，它已经被广泛应用于游戏、影视、军事、航空航天、地理、医学、机械设计，以及各类科学数据可视化的领域。而随着网络和移动平台的飞速发展，异军突起的 OpenGL ES 和 WebGL 标准也吸引了大批开发者的眼球，而这两者与 OpenGL 本身同样有着千丝万缕的联系。

OpenGL 支持几乎所有现有的主流操作系统平台，包括 Windows、Mac OS X 以及各种 UNIX 平台。它同时也可以用于几乎所有主流的编程语言环境当中，例如 C/C++、Java、C#、Visual Basic、Python、Perl 等。因此，无可非议地说，OpenGL 应当是目前全球最为广泛学习和使用的图形开发 API 接口，我们几乎可以在全世界任何一台计算机安装的软件当中找到它的身影（当然，在 Windows 平台下总会有 OpenGL 和 DirectX 两类 API 的地位与优劣之争，这又是本书内容之外的另一番故事了）。

而作为 OpenGL 学习的经典书籍，有着“红宝书”之名的本书也已经更新到了第 8 版。这一版的最大特色就是“变革”。是的，这是一本变革之书，它直接与 OpenGL 4.3 版本的内容相贴合，彻底以核心模式的主要函数与着色器的内容为讲解重点。如果你已经读过以前的一些修订版本，并且已经对 `glBegin()`、`glLoadMatrix()`，或者 `display list` 这些函数和名词耳熟能详，那么不要惊讶：从这一版开始，你将再也见不到有关这些内容的介绍。即使你是一位从业数十年的 OpenGL 开发者，从这一刻开始，你恐怕也需要从头来过。

不过无须惊惶，你手头的已经开发了多年的 OpenGL 程序，依然可以在兼容模式下顺利执行。而基于可编程流水线的全新架构和接口，想必也会给有经验的开发者带来更多的思考与创新力。而对于初涉 OpenGL 开发的新人来说，这恰恰是一个好机会，让你们从新的起点出发，把学习和思考的重心放在以着色语言为基础的体系之上，而不是死记硬背那些程序接口，或者苦恼于繁杂的状态切换。

本书内容翔实，章节划分清晰明确，适合各种层次的读者选择性地阅读。不过，作为一个具有颠覆性质的修订版本，这一版的“红宝书”也暴露出内容讲解上的一些问题，部

分章节略显臃肿冗余，而内容的衔接上也不够连贯，这样都可能让没有基础的新人感到无所适从。应当说，这也是本书未来亟待更新和改进的方面。

译者作为基于 OpenGL 的开源 3 维引擎 OpenSceneGraph 的核心开发者，有幸主持了本版的翻译工作，但是因为时间紧迫，译者才疏学浅，因此错漏之处想必很多，敬请读者谅解。如有任何形式的批评或建议，欢迎随时与译者联系。本书的图文内容与之前的修订版本基本不存在关联，书中第 1~6 章，第 8~11 章均由王锐负责翻译与整理，其他参与本书翻译的人员还有：郭华（第 6 章，附录 G、附录 H 和附录 I）、苏明南（第 7 章和附录 D）、张静（第 12 章）、王凯（附录 A）、陈节（附录 B）、龙海鹰（附录 C）、毕玉玲（附录 E）。感谢他们的辛苦付出，也感谢机械工业出版社的编辑们的信任与帮助！



前言 *Preface*

OpenGL 图形系统是图形硬件的一种软件接口（GL 表示 Graphics Library，即图形库）。它使得用户可以创建交互式的程序以产生运动的 3 维对象的颜色图像。通过 OpenGL，我们可以使用计算机图形学技术来产生逼真的图像，或者通过一些虚构的方式产生虚拟的图像。这本指南将告诉你如何使用 OpenGL 图形系统进行编程，得到你所期望的视觉效果。

本书的主要内容

本书包含以下一些章节：

- ❑ 第 1 章对 OpenGL 可以完成的工作进行了概览。它还提供了一个简单的 OpenGL 程序并解释了一些本质性的编程细节，它们可能会用于后继的章节中。
- ❑ 第 2 章讨论了 OpenGL 中最主要的特性——可编程着色器，并介绍了它们在应用程序中的初始化和使用方法。
- ❑ 第 3 章介绍了使用 OpenGL 进行几何体绘制的各种方法，以及一些可以让渲染更为高效的优化手段。
- ❑ 第 4 章解释了 OpenGL 对于颜色的处理过程，包括像素的处理、缓存的管理，以及像素处理相关的渲染技术。
- ❑ 第 5 章给出了将 3 维场景在一个 2 维计算机屏幕上表现的操作细节，包括各种几何投影类型的数学原理和着色器操作。
- ❑ 第 6 章讨论了将几何模型与图像结合来创建真实的、高质量的 3 维模型的方法。
- ❑ 第 7 章介绍了计算机图形的光照效果模拟方法，主要是这类方法在可编程着色器中的实现。
- ❑ 第 8 章介绍了使用可编程着色器生成纹理和其他表面效果的方法细节，从而增强真实感和其他的渲染特效。
- ❑ 第 9 章解释了 OpenGL 管理和细分几何表面的着色器功能。

- ❑ 第 10 章介绍了一个在 OpenGL 渲染流水线中使用着色器进行几何体图元修改的特别技术。
- ❑ 第 11 章介绍了使用 OpenGL 帧缓存和缓存内存实现高级渲染技术和非图形学应用的相关方法。
- ❑ 第 12 章介绍了最新的着色器阶段，将通用计算的方法融合到 OpenGL 的渲染流水线当中。

此外，我们也提供了一系列作为参考的附录内容。

- ❑ 附录 A 介绍了 OpenGL Utility Toolkit 这个专用于窗口系统操作的库。GLUT 是可移植的，它可以用来实现更简短也更加可读的代码案例。
- ❑ 附录 B 介绍了 OpenGL 体系中的其他 API，包括用于嵌入式和移动平台系统的 OpenGL ES，以及用于 Web 浏览器内的交互式 3D 应用程序的 WebGL。
- ❑ 附录 C 提供了有关 OpenGL 着色语言的详细参考文档。
- ❑ 附录 D 列出了 OpenGL 维护的所有状态变量，并介绍了获取其数值的方法。
- ❑ 附录 E 介绍了与矩阵变换相关的一些数学方法。
- ❑ 附录 F 介绍了不同的窗口系统相关的各种库，它们提供了各种绑定例程，以支持 OpenGL 渲染到本地窗口当中。
- ❑ 附录 G 对于 OpenGL 中所用到的浮点数格式做出了概述。
- ❑ 附录 H 介绍了 OpenGL 中最新的调试特性。
- ❑ 附录 I 给出了有关 uniform 缓存的使用的参考文档，其中使用了 OpenGL 定义的标准内存布局。

本版新增内容

本书的内容是颠覆性的！对于那些已经阅读过本书以前版本的读者来说，这一版完全根据 OpenGL 应用程序开发的最新方法和技术进行了重写。本书将经典红宝书的以函数为中心的方法，与《OpenGL 着色语言》（通常也称作“橘皮书”）一书中的着色技术进行了融合。

在这一版中，作者团队中也纳入了 OpenGL 开发的一些主要贡献者，以及 OpenGL 着色语言标准的编者。因此，这一版将会涵盖 OpenGL 的最新版本，也就是版本 4.3，其中包括了计算着色器的内容。本书还介绍了可编程渲染管线的每个阶段。我们衷心地希望你能够从本书中找到实用的以及有教育价值的内容。

你需要在阅读本书之前掌握的知识

本书假设你已经了解了使用 C 语言进行编程的方法（我们将使用少量的 C++ 程序，不过你应该会比较容易理解它们），并且具有一定的数学背景（几何、三角学、线性代数、微

积分，以及微分几何)。即使你对于计算机图形学技术没有太多的经验或者一无所知，你也可以学习和理解本书中讨论的大部分内容。当然，计算机图形学是一个不断延展的学科，因此你也许还需要阅读以下的补充内容来提升自己的知识。

□《Computer Graphics: Principles and Practice》第3版, John F. Hughes 等著 (Addison-Wesley, 2013): 这本书是有关计算机图形学的一本百科全书。它包含了大量有价值的信息，不过在阅读它之前，你最好已经对这门学科有了一定的了解。

□《3D Computer Graphics》，Andrew S. Glassner 著 (The Lyons Press, 1994): 这本书是有关计算机图形学的非技术性的、适度的介绍。它注重于可以实现的可视化效果本身，而不是实现这些效果所需的技术。

另一个可以有组织地进行系统学习的地方就是 OpenGL 网站。该网站包含了软件、示例程序、文档、FAQ、讨论版，以及新闻页面。如果你想要搜索 OpenGL 相关问题的答案，那么这里是一个好的开始：

<http://www.opengl.org/>

此外，OpenGL 的官方网站中还包含了 OpenGL 的最新版本对应的所有函数和着色语言语法的完整文档。这些网页内容完整地涵盖了《OpenGL Reference Manual》一书的内容，后者由 OpenGL Architecture Review Board 和 Addison-Wesley 出版。

OpenGL 是一个与硬件密切相关的编程接口标准，我们可能会在某一类特定的硬件上使用一个特定的 OpenGL 实现。本书将会介绍如何使用任意的 OpenGL 实现进行编程。但是，因为这些实现之间会存在细微的差异——包括性能上的差异，以及额外的特性支持——你可能需要阅读自己所用的特定设备实现所对应的补充文档。此外，某个特定实现的供应商网站上，也可能也会提供一些 OpenGL 相关的实用工具、工具包、编程和调试支持、窗口组件、示例代码，以及示例程序。

如何获取示例代码

本书包含很多示例程序，它们演示了特定 OpenGL 编程技术的用法。本书的读者群体在计算机图形学和 OpenGL 方面可能有着巨大的经验差异，有的人是新手，而有的人是多年的老手，因此这些章节里给出的案例都会使用最简单的方法去实现一个特定的渲染形式，并且全部使用 OpenGL 4.3 版本的接口。这样的做法主要是为了确保那些刚开始学习 OpenGL 的读者也能够顺利地阅读相关的内容。对于那些已经有了足够的经验，只是希望了解最新的 API 特性实现的读者，我们首先感谢你能够耐心阅读本书前面的内容，之后你可以访问我们的网站：

<http://www.opengl-redbook.com/>

在这里你将会找到本书中所有示例的源代码，它们均使用最新的特性进行实现，而后文的讨论中也会涉及从一个 OpenGL 版本移植到另一个版本所需的修改。

本书中所有的程序都使用了 OpenGL Utility Toolkit (GLUT)，它的原作者为 Mark Kilgard。在这一版中，我们将使用 GLUT 接口的开源版本，它来自 freeglut 工程。这个工程对 Mark 的原始工程（在作者的著作《OpenGL Programming for the X Window System》中进行了详细介绍，Addison-Wesley, 1997）进行了加强。你可以在下面的地址里找到开源的工程页面：

<http://freeglut.sourceforge.net/>

你可以在这个网站中找到相应的代码和二进制程序。

本书还介绍了有关 GLUT 库的信息。可以在 OpenGL 网站的资源页面找到更多帮助你学习和使用 OpenGL 与 GLUT 的资源。

<http://www.opengl.org/resources/>

OpenGL 的很多实现也包含了一些系统相关的代码示例。这些源代码可能是你实现程序时最好的资源，因为它们已经针对系统进行了优化。你可以阅读与自己的系统相关的 OpenGL 文档来了解如何获取这些代码示例。

勘误

遗憾的是，这本书中一定也存在着错误。此外，即使在本书出版的期间，OpenGL 也是不断更新的：有一些错误被修正，并且标准文档中也做出了澄清，同时还有新的标准被发布。我们将在网站 <http://www.opengl-redbook.com/> 上维护一个错误和更新列表，同时我们也会提供一些功能让用户提交自己发现的错误。如果你发现了本书中的错误，我们首先向你郑重道歉，并且非常感谢你的报告。我们将尽快对其进行更正。

目录 Contents

推荐语
译者序
前言

第 1 章 OpenGL 概述..... 1

- 1.1 什么是 OpenGL 1
- 1.2 初识 OpenGL 程序 2
- 1.3 OpenGL 语法 6
- 1.4 OpenGL 渲染管线 7
 - 1.4.1 准备向 OpenGL 传输数据 8
 - 1.4.2 将数据传输到 OpenGL 8
 - 1.4.3 顶点着色 9
 - 1.4.4 细分着色 9
 - 1.4.5 几何着色 9
 - 1.4.6 图元装配 9
 - 1.4.7 剪切 9
 - 1.4.8 光栅化 9
 - 1.4.9 片元着色 10
 - 1.4.10 逐片元的操作 10
- 1.5 第一个程序：深入分析 10
 - 1.5.1 进入 main() 函数 10
 - 1.5.2 OpenGL 的初始化过程 12
 - 1.5.3 第一次使用 OpenGL 进行渲染 21

第 2 章 着色器基础..... 25

- 2.1 着色器与 OpenGL 26
- 2.2 OpenGL 的可编程管线 26
- 2.3 OpenGL 着色语言概述 28
 - 2.3.1 使用 GLSL 构建着色器 28
 - 2.3.2 存储限制符 34
 - 2.3.3 语句 37
 - 2.3.4 计算的不变性 41
 - 2.3.5 着色器的预处理器 43
 - 2.3.6 编译器的控制 45
 - 2.3.7 全局着色器编译选项 45
- 2.4 数据块接口 46
 - 2.4.1 uniform 块 46
 - 2.4.2 指定着色器中的 uniform 块 47
 - 2.4.3 从应用程序中访问 uniform 块 48
 - 2.4.4 buffer 块 53
 - 2.4.5 in/out 块 54
- 2.5 着色器的编译 54
 - 2.5.1 我们的 LoadShaders() 函数 58
- 2.6 着色器子程序 58
 - 2.6.1 GLSL 的子程序设置 59
 - 2.6.2 选择着色器子程序 60

2.7 独立的着色器对象	62	4.4 多重采样	115
第3章 OpenGL 绘制方式	64	4.4.1 采样着色	116
3.1 OpenGL 图元	64	4.5 片元的测试与操作	117
3.1.1 点	65	4.5.1 剪切测试	118
3.1.2 线、条带与循环线	66	4.5.2 多重采样的片元操作	118
3.1.3 三角形、条带与扇面	66	4.5.3 模板测试	119
3.2 OpenGL 缓存数据	69	4.5.4 模板的例子	120
3.2.1 创建与分配缓存	69	4.5.5 深度测试	122
3.2.2 向缓存输入和输出数据	71	4.5.6 融混	124
3.2.3 访问缓存的内容	75	4.5.7 融混参数	125
3.2.4 丢弃缓存数据	80	4.5.8 控制融混的参数	125
3.3 顶点规范	80	4.5.9 融混方程	127
3.3.1 深入讨论 VertexAttrib- Pointer	81	4.5.10 抖动	128
3.3.2 静态顶点属性的规范	84	4.5.11 逻辑操作	128
3.4 OpenGL 的绘制命令	86	4.5.12 遮挡查询	129
3.4.1 图元的重启动	92	4.5.13 条件渲染	132
3.5 多实例渲染	96	4.6 逐图元的反走样	133
3.5.1 多实例的顶点属性	97	4.6.1 线段的反走样	134
3.5.2 在着色器中使用实例 计数器	102	4.6.2 多边形的反走样	135
3.5.3 多实例方法的回顾	104	4.7 帧缓存对象	135
第4章 颜色、像素和帧缓存	105	4.7.1 渲染缓存	137
4.1 基本颜色理论	106	4.7.2 创建渲染缓存的存储 空间	138
4.2 缓存及其用途	107	4.7.3 帧缓存附件	140
4.2.1 缓存的清除	109	4.7.4 帧缓存的完整性	142
4.2.2 缓存的掩码	110	4.7.5 帧缓存的无效化	144
4.3 颜色与 OpenGL	110	4.8 多重渲染缓存的同步写入	145
4.3.1 颜色的表达与 OpenGL	111	4.8.1 选择颜色缓存来进行读写 操作	146
4.3.2 顶点颜色	112	4.8.2 双源融混	148
4.3.3 光栅化	114	4.9 像素数据的读取和拷贝	150
		4.10 拷贝像素矩形	152

第 5 章 视口变换、剪切与反馈	153
5.1 观察视图	154
5.1.1 视图模型	154
5.1.2 相机模型	154
5.1.3 正交视图模型	157
5.2 用户变换	158
5.2.1 矩阵乘法的回顾	159
5.2.2 齐次坐标	161
5.2.3 线性变换与矩阵	163
5.2.4 法线变换	173
5.2.5 OpenGL 矩阵	174
5.3 OpenGL 变换	177
5.3.1 高级技巧：用户剪切	178
5.4 transform feedback	179
5.4.1 transform feedback 对象	180
5.4.2 transform feedback 缓存	181
5.4.3 配置 transform feedback 的变量	183
5.4.4 transform feedback 的启动 和停止	187
5.4.5 transform feedback 的示例： 粒子系统	189
第 6 章 纹理	195
6.1 纹理映射	196
6.2 基本纹理类型	197
6.3 创建和初始化纹理	198
6.3.1 纹理格式	202
6.4 代理纹理	207
6.5 设置纹理数据	208
6.5.1 显式设置纹理数据	208
6.5.2 使用 Pixel Unpack 缓存	210
6.5.3 从帧缓存拷贝数据	211
6.5.4 从文件加载图像	212
6.5.5 查询纹理数据	215
6.5.6 纹理数据布局	215
6.6 采样器对象	219
6.6.1 采样器参数	220
6.7 使用纹理	221
6.7.1 纹理坐标	223
6.7.2 组织纹理数据	226
6.7.3 使用多重纹理	227
6.8 复杂纹理类型	229
6.8.1 3 维纹理	229
6.8.2 数组纹理	231
6.8.3 立方体映射纹理	231
6.8.4 阴影采样器	237
6.8.5 深度模板纹理	238
6.8.6 缓存纹理	238
6.9 纹理视图	240
6.10 压缩纹理	243
6.11 滤波	245
6.11.1 线性滤波	245
6.11.2 使用和生成 mipmap	247
6.11.3 计算 mipmap 级别	251
6.11.4 mipmap 细节层次控制	252
6.12 高级纹理查询函数	252
6.12.1 显式细节层次	252
6.12.2 显式梯度设置	253
6.12.3 偏移后的纹理获取	253
6.12.4 投影纹理	254
6.12.5 着色器中的纹理查询	254
6.12.6 收集纹素	256
6.12.7 合并特殊函数	256
6.13 点精灵	257
6.13.1 带纹理的点精灵	257

6.13.2 控制点的外观	259	8.2.4 法线贴图	326
6.14 渲染到纹理贴图	260	8.3 程序式纹理的反走样	326
6.14.1 丢弃已渲染数据	263	8.3.1 走样的来源	327
6.15 本章总结	264	8.3.2 避免走样问题	328
6.15.1 纹理回顾	264	8.3.3 提高分辨率	329
6.15.2 纹理的最好实践	265	8.3.4 高频率的反走样	330
第 7 章 光照与阴影	266	8.3.5 频率截断	337
7.1 光照介绍	267	8.3.6 程序式反走样的总结	339
7.2 经典光照模型	267	8.4 噪声	339
7.2.1 不同光源类型的片元着色器	268	8.4.1 噪声的定义	341
7.2.2 将计算移到顶点着色器	277	8.4.2 噪声纹理	345
7.2.3 多个光源和材质	279	8.4.3 权衡	348
7.2.4 光照坐标系	285	8.4.4 一个简单的噪声着色器	349
7.2.5 经典光照模型的局限	285	8.4.5 湍流	351
7.3 光照模型进阶	286	8.4.6 大理石	353
7.3.1 半球光照	286	8.4.7 花岗岩	353
7.3.2 基于图像的光照	289	8.4.8 木纹	354
7.3.3 球面光照	293	8.4.9 噪声的总结	357
7.4 阴影映射	296	8.5 更多信息	357
7.4.1 创建一张阴影贴图	297	第 9 章 细分着色器	359
7.4.2 使用阴影贴图	299	9.1 细分着色器	359
第 8 章 程序式纹理	303	9.2 细分面片	360
8.1 程序式纹理	303	9.3 细分控制着色器	361
8.1.1 规则的花纹	305	9.3.1 生成输出面片的顶点	362
8.1.2 玩具球	311	9.3.2 细分控制着色器的变量	362
8.1.3 晶格	318	9.3.3 细分的控制	363
8.1.4 程序式着色方法的总结	319	9.4 细分计算着色器	367
8.2 凹凸贴图映射	319	9.4.1 设置图元生成域	368
8.2.1 应用程序设置	321	9.4.2 设置生成图元的面朝向	368
8.2.2 顶点着色器	323	9.4.3 设置细分坐标的间隔	368
8.2.3 片元着色器	324	9.4.4 更多的细分计算着色器 layout 选项	368

9.4.5 设置顶点的位置	369	第 11 章 内存	420
9.4.6 细分计算着色器的变量	369	11.1 使用纹理存储通用数据	420
9.5 细分实例：茶壶	370	11.1.1 将纹理绑定到图像单元	425
9.5.1 处理面片输入顶点	370	11.1.2 图像数据的读取和写入	427
9.5.2 计算茶壶的细分坐标	371	11.2 着色器存储缓存对象	430
9.6 更多的细分技术	373	11.2.1 写入结构化数据	431
9.6.1 视口相关的细分	373	11.3 原子操作和同步	431
9.6.2 细分的共享边与裂缝	375	11.3.1 图像的原子操作	431
9.6.3 置换贴图映射	376	11.3.2 缓存的原子操作	439
第 10 章 几何着色器	377	11.3.3 同步对象	440
10.1 创建几何着色器	378	11.3.4 图像限定符和屏障	444
10.2 几何着色器的输入和输出	380	11.3.5 高性能的原子计数器	452
10.2.1 几何着色器的输入	380	11.4 示例	455
10.2.2 特殊的几何着色器图元	383	11.4.1 顺序无关的透明	455
10.2.3 几何着色器的输出	387	第 12 章 计算着色器	466
10.3 产生图元	389	12.1 概述	466
10.3.1 几何体的裁减	389	12.2 工作组及其执行	467
10.3.2 几何体的扩充	390	12.2.1 知道工作组的位置	471
10.4 transform feedback 高级篇	394	12.3 通信与同步	472
10.4.1 多重输出流	395	12.3.1 通信	473
10.4.2 图元查询	399	12.3.2 同步	474
10.4.3 使用 transform feedback 的结果	400	12.4 示例	475
10.5 几何着色器的多实例化	408	12.4.1 物理模拟	476
10.6 多视口与分层渲染	409	12.4.2 图像处理	481
10.6.1 视口索引	409	12.5 本章总结	485
10.6.2 分层渲染	414	12.5.1 计算着色器回顾	485
10.7 本章小结	417	12.5.2 计算着色器的最佳实践	485
10.7.1 几何着色器回顾	417	附录 A GLUT 基础知识	487
10.7.2 几何着色器的经验谈	418	附录 B OpenGL ES 与 WebGL	493

附录 C 内置 GLSL 变量与函数	504	的浮点格式	612
附录 D 状态变量	552	附录 H OpenGL 程序的调试与 优化	618
附录 E 齐次坐标与变换矩阵	591	附录 I 缓存对象的布局	632
附录 F OpenGL 与窗口系统	596	术语表	635
附录 G 纹理、帧缓存与渲染缓存			





OpenGL 概述

本章目标

阅读完本章内容之后，你将会具备以下能力：

- ❑ 描述 OpenGL 的目的，它在创建计算机图像时，能够做什么，不能做什么。
- ❑ 了解一个 OpenGL 程序的通用结构。
- ❑ 列举出 OpenGL 渲染管线中的多个着色阶段。

这一章将对 OpenGL 做一个大概的阐述。本章主要包含以下几节：

- ❑ 1.1 节将解释 OpenGL 的含义，它可以做到、不能做到的事情，以及它的工作方式。
- ❑ 1.2 节将展示一个 OpenGL 程序的结构和表现形式。
- ❑ 1.3 节介绍 OpenGL 所使用命令的命名格式。
- ❑ 1.4 节介绍 OpenGL 创建图像的整个处理管线过程。
- ❑ 1.5 节将重新剖析之前的 OpenGL 示例程序，并且对程序的每个部分提供更详尽的解释说明。

1.1 什么是 OpenGL

OpenGL 是一种应用程序编程接口 (Application Programming Interface, API)，它是一种可以对图形硬件设备特性进行访问的软件库。OpenGL 库的 4.3 版本 (即本书所使用的版本) 包含了超过 500 个不同的命令，可以用于设置所需的对象、图像和操作，以便开发交互式的 3 维计算机图形应用程序。

OpenGL 被设计为一个现代化的、硬件无关的接口，因此我们可以在不考虑计算机操作

系统或窗口系统的前提下，在多种不同的图形硬件系统上，或者完全通过软件的方式（如果当前系统没有图形硬件）实现 OpenGL 的接口。OpenGL 自身并不包含任何执行窗口任务或者处理用户输入的函数。事实上，我们需要通过应用程序所运行的窗口系统提供的接口来执行这类操作。与此类似，OpenGL 也没有提供任何用于表达 3 维物体模型，或者读取图像文件（例如 JPEG 文件）的操作。这个时候，我们需要通过一系列的几何图元（geometric primitive）（包括点、线、三角形以及 Patch）来创建 3 维空间的物体。

OpenGL 已经诞生了很长时间，它最早的 1.0 版本是在 1994 年 7 月发布的，通过 Silicon 的图形计算机系统开发出来。而到了今天已经发布了非常多的 OpenGL 版本，以及大量构建于 OpenGL 之上以简化应用程序开发过程的软件库。这些软件库大量用于视频游戏、科学可视化和医学软件的开发，或者只是用来显示图像。不过，如今 OpenGL 的版本与其早期的版本已经有很多显著的不同。本书将介绍如何使用最新的 OpenGL 版本来创建不同的应用程序。

一个用来渲染图像的 OpenGL 程序需要执行的主要操作如下所示。（1.4 节将对这些操作进行详细解释。）

- ❑ 从 OpenGL 的几何图元中设置数据，用于构建形状。
- ❑ 使用不同的着色器（shader）对输入的图元数据执行计算操作，判断它们的位置、颜色，以及其他渲染属性。
- ❑ 将输入图元的数学描述转换为与屏幕位置对应的像素片元（fragment）。这一步也称作光栅化（rasterization）。
- ❑ 最后，针对光栅化过程产生的每个片元，执行片元着色器（fragment shader），从而决定这个片元的最终颜色和位置。
- ❑ 如果有必要，还需要对每个片元执行一些额外的操作，例如判断片元对应的对象是否可见，或者将片元的颜色与当前屏幕位置的颜色进行融合。

OpenGL 是使用客户端－服务端的形式实现的，我们编写的应用程序可以看做客户端，而计算机图形硬件厂商所提供的 OpenGL 实现可以看做服务端。OpenGL 的某些实现（例如 X 窗口系统的实现）允许服务端和客户端在一个网络内的不同计算机上运行。这种情况下，客户端负责提交 OpenGL 命令，这些 OpenGL 命令然后被转换为窗口系统相关的协议，通过共享网络传输到服务端，最终执行并产生图像内容。

1.2 初识 OpenGL 程序

正因为可以用 OpenGL 去做那么多的事情，所以 OpenGL 程序有可能会写得非常庞大和复杂。不过，所有 OpenGL 程序的基本结构通常都是类似的，其步骤如下：

- ❑ 初始化物体渲染所对应的状态。
- ❑ 设置需要渲染的物体。

在阅读代码之前，我们有必要了解一些最常用的图形学名词。**渲染（render）**这个词在前文中已经多次出现，它表示计算机从模型创建最终图像的过程。**OpenGL**只是其中一种渲染系统，除此之外，还有很多其他的渲染系统。**OpenGL**是基于光栅化的系统，但是也有别的方法用于生成图像。例如光线跟踪（ray tracing），而这类技术已经超出了本书的介绍范围。不过，就算是用到了光线跟踪技术的系统，同样有可能需要用到 **OpenGL** 来显示图像，或者计算图像生成所需的信息。

模型（model），或者场景对象（我们会交替地使用这两个名词）是通过几何图元，例如点、线和三角形来构建的，而图元与模型的顶点（vertex）也存在着各种对应的关系。

OpenGL 另一个最本质的概念叫做着色器，它是图形硬件设备所执行的一类特殊函数。理解着色器最好的方法是把它看做专为图形处理单元（通常也叫做 GPU）编译的一种小型程序。**OpenGL** 在其内部包含了所有的编译器工具，可以直接从着色器源代码创建 GPU 所需的编译代码并执行。在 **OpenGL** 中，会用到四种不同的着色阶段（shader stage）。其中最常用的包括的顶点着色器（vertex shader）以及片元着色器，前者用于处理顶点数据，后者用于处理光栅化后的片元数据。所有的 **OpenGL** 程序都需要用到这两类着色器。

最终生成的图像包含了屏幕上绘制的所有像素点。像素（pixel）是显示器上最小的可见单元。计算机系统将所有的像素保存到帧缓存（framebuffer）当中，后者是由图形硬件设备管理的一块独立内存区域，可以直接映射到最终的显示设备上。

图 1-1 所示为一个简单的 **OpenGL** 程序的输出结果，它在一个窗口中渲染了两个蓝色的三角形。这个例子的完整源代码如例 1.1 所示。

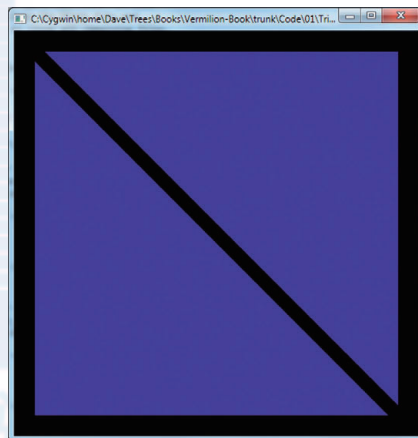


图 1-1 第一个 **OpenGL** 程序 triangles.cpp 的结果图像

例 1.1 第一个 **OpenGL** 程序 triangles.cpp

```

////////////////////////////////////
//
// triangles.cpp
//
////////////////////////////////////

#include <iostream>
using namespace std;

#include "vgl.h"
#include "LoadShaders.h"

enum VAO_IDs { Triangles, NumVAOs };
enum Buffer_IDs { ArrayBuffer, NumBuffers };

```

```

enum Attrib_IDs { vPosition = 0 };

GLuint  VAOs[NumVAOs];
GLuint  Buffers[NumBuffers];

const GLuint  NumVertices = 6;

//-----
//
// init
//

void
init(void)
{
    glGenVertexArrays(NumVAOs, VAOs);
    glBindVertexArray(VAOs[Triangles]);

    GLfloat  vertices[NumVertices][2] = {
        { -0.90, -0.90 }, // Triangle 1
        {  0.85, -0.90 },
        { -0.90,  0.85 },
        {  0.90, -0.85 }, // Triangle 2
        {  0.90,  0.90 },
        { -0.85,  0.90 }
    };

    glGenBuffers(NumBuffers, Buffers);
    glBindBuffer(GL_ARRAY_BUFFER, Buffers[ArrayBuffer]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
                 vertices, GL_STATIC_DRAW);

    ShaderInfo  shaders[] = {
        { GL_VERTEX_SHADER, "triangles.vert" },
        { GL_FRAGMENT_SHADER, "triangles.frag" },
        { GL_NONE, NULL }
    };

    GLuint program = LoadShaders(shaders);
    glUseProgram(program);

    glVertexAttribPointer(vPosition, 2, GL_FLOAT,
                          GL_FALSE, 0, BUFFER_OFFSET(0));
    glEnableVertexAttribArray(vPosition);
}

//-----
//
// display
//

void
display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);

    glBindVertexArray(VAOs[Triangles]);
    glDrawArrays(GL_TRIANGLES, 0, NumVertices);
}

```

```

        glFlush();
    }
    //-----
    //
    // main
    //

    int
    main(int argc, char** argv)
    {
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_RGBA);
        glutInitWindowSize(512, 512);
        glutInitContextVersion(4, 3);
        glutInitContextProfile(GLUT_CORE_PROFILE);
        glutCreateWindow(argv[0]);

        if (glewInit()) {
            cerr << "Unable to initialize GLEW ... exiting" << endl;
            exit(EXIT_FAILURE);
        }

        init();

        glutDisplayFunc(display);

        glutMainLoop();
    }

```

也许你会觉得这里的代码有点多，不过它的确就是几乎每一个 OpenGL 程序所必需的基本内容了。我们用到了不属于 OpenGL 正式部分的一些第三方软件库，以便实现一些简单的工作，例如创建窗口、接收鼠标和键盘输入等——OpenGL 自身并不包含这些功能。我们还创建了一些辅助函数和简单的 C++ 类来简化示例程序的编写。尽管 OpenGL 是一个 C 语言形式的库，但是本书中的所有示例都会使用 C++ 来编写，但只是非常简单的 C++。事实上，我们用到的绝大部分 C++ 代码是用来实现一些数学向量和构建矩阵的。

简单来说，下面列出的就是例 1.1 做的所有事情。不过不用担心，后面的章节会更详细地解释这些概念。

❑ 在程序的起始部分，我们包含了必要的头文件并且声明了一些全局变量[⊖]和其他有用的编程结构。

❑ `init()` 函数负责设置程序中需要用到的数据。它可能是渲染图元时用到的顶点信息，或者用于执行纹理映射（texture mapping）的图像数据，第 6 章会介绍这一技术。

在这个 `init()` 函数中，首先指定了两个被渲染的三角形的位置信息。然后指定了程序中使用的着色器。在这个示例中，我们只需要使用顶点和片元着色器。这里的 `LoadShaders()` 是我们为着色器进入 GPU 的操作专门实现的函数。第 2 章会详细介绍与它相关的内容。

⊖ 没错，对于大型程序而言我们会尽量避开全局变量，不过只是作为演示程序而言，使用它们也没有什么关系。

`init()` 函数的最后一部分叫做着色管线装配 (shader plumbing)，也就是将应用程序的数据与着色器程序的变量关联起来。同样会在第 2 章详细介绍这一部分的内容。

□ `display()` 函数真正执行了渲染的工作。也就是说，它负责调用 OpenGL 函数并渲染需要的内容。几乎所有的 `display()` 函数都要完成类似这个简单示例中的三个步骤。

1) 调用 `glClear()` 来清除窗口内容。

2) 调用 OpenGL 命令来渲染对象。

3) 将最终图像输出到屏幕。

□ 最后，`main()` 函数执行了创建窗口、调用 `init()` 以及最终进入事件循环体系的一系列繁重工作。这里你也会看到一些以 `gl` 开头的函数，但是它们看起来和其他的函数又有一些不同。这些函数就是刚才所说的来自第三方库 GLUT 和 GLEW 的函数，我们会随时使用它们来快速完成一些简单的功能，并且保证 OpenGL 程序可以运行在不同的操作系统和窗口系统上。

在深入了解这些函数之前，我们有必要先解释一下 OpenGL 的函数、常量的命名方式，以及一些有用的编程结构。

1.3 OpenGL 语法

正如你可能已经了解的，OpenGL 库中所有的函数都会以字符 “gl” 作为前缀，然后是一个或者多个大写字母开头的词组，以此来命名一个完整的函数（例如 `glBindVertexArray()`）。OpenGL 的所有函数都是这种格式。在上面的程序中你还看到了以 “glut” 开头的函数，它们来自第三方库 OpenGL Utility Toolkit (GLUT)，作者是 Mark J. Kilgard。这是一个非常流行的跨平台工具库，可以用来显示窗口、管理用户输入，以及执行其他一些操作。我们所用的 GLUT 的版本叫做 Freeglut，它的原作者是 Pawel W. Olszta，其他贡献者还包括 Andreas Umbach 和 Steve Baker（后者目前负责维护这个库）。Freeglut 是原始的 GLUT 库的一个新变种。此外，你可能还会看到一个独立的函数 `glewInit()`，它来自于 Milan Ikits 和 Marcelo Magallon 编写的第三方库 OpenGL Extension Wrangler。附录 A 会进一步讲解这两个库的内容。

与函数命名约定类似，OpenGL 库中定义的常量也是 `GL_COLOR_BUFFER_BIT` 的形式，如 `display()` 函数中所示。所有的常量都以 `GL_` 作为前缀，并且使用下划线来分隔单词。这些常量的定义是通过 `#define` 来完成的，它们基本上都可以在 OpenGL 的头文件 `glcorearb.h` 和 `glxt.h` 中找到。

为了能够方便地在不同的操作系统之间移植 OpenGL 程序，OpenGL 还为函数定义了不同的数据类型，例如 `GLfloat` 是浮点数类型，在例 1.1 中用它来声明 `vertices` 数组。此外，由于 OpenGL 是一个 “C” 语言形式的库，因此它不能使用函数的重载来处理不同类型的数据，此时它使用函数名称的细微变化来管理实现同一类功能的函数集。举例来说，我们

将会在第 2 章遇到一个名为 `glUniform*()` 的函数，它有多种变化形式，例如 `glUniform2f()` 和 `glUniform3fv()`。在函数名称的“核心”部分之后，我们通过后缀的变化来提示函数应当传入的参数。例如，`glUniform2f()` 中的“2”表示这个函数需要传入 2 个参数值（由于还可能传入其他的参数，因此一共定义了 24 种不同的 `glUniform*()` 函数——在本书中，我们使用 `glUniform*()*` 来统一表示所有 `glUniform*()` 函数的集合）。我们还要注意“2”之后的“f”。这个字符表示这两个参数都是 `GLfloat` 类型的。最后，有些类型的函数名称末尾会有一个“v”，它是 `vector` 的缩写，即表示我们需要用一个 1 维的 `GLfloat` 数组来传入 2 个浮点数值（对于 `glUniform2fv()` 而言），而不是两个独立的参数值。

表 1-1 所示为所有可以作为后缀的字母，以及它们所对应的数据类型。

表 1-1 命令后缀与参数数据类型

后缀	数据类型	通常对应的 C 语言数据类型	OpenGL 类型定义
b	8 位整型	signed char	GLbyte
s	16 位整型	signed short	GLshort
i	32 位整型	int	GLint、GLsizei
f	32 位浮点型	float	GLfloat、GLclampf
d	64 位浮点型	double	GLdouble、GLclampd
ub	8 位无符号整型	unsigned char	GLubyte
us	16 位无符号整型	unsigned short	GLushort
ui	32 位无符号整型	unsigned int	GLuint、GLenum、GLbitfield



注意 使用 C 语言的数据类型来直接表示 OpenGL 数据类型时，因为 OpenGL 自身的实现不同，可能会造成类型不匹配。如果直接在应用程序中使用 OpenGL 定义的数据类型，那么当需要在不同的 OpenGL 实现之间移植自己的代码时，就不会产生数据类型不匹配的问题了。

1.4 OpenGL 渲染管线

OpenGL 实现了我们通常所说的渲染管线（`rendering pipeline`），它是一系列数据处理过程，并且将应用程序的数据转换到最终渲染的图像。图 1-2 所示为 OpenGL 4.3 版本的管线。自从 OpenGL 诞生以来，它的渲染管线已经发生了非常大的改变。

OpenGL 首先接收用户提供的几何数据（顶点和几何图元），并且将它输入到一系列着色器阶段中进行处理，包括：顶点着色、细分着色（它本身包含两个着色器），以及最后的几何着色，然后它将被送入光栅化单元（`rasterizer`）。光栅化单元负责对所有剪切区域（`clipping region`）内的图元生成片元数据，然后对每个生成的片元都执行一个片元着色器。

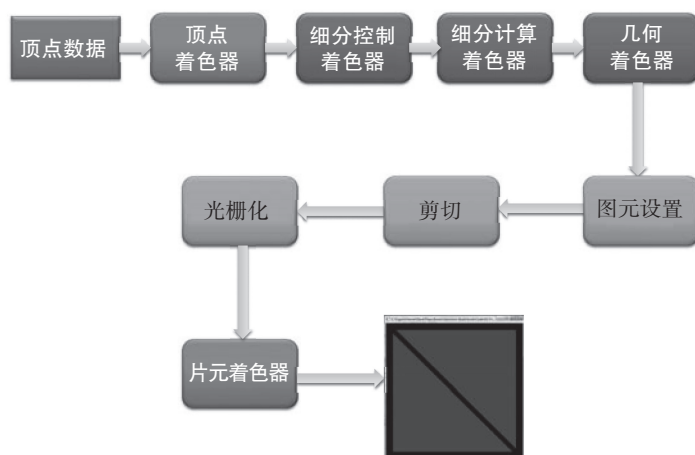


图 1-2 OpenGL 管线

正如你所了解的，对于 OpenGL 应用程序而言着色器扮演了一个最主要的角色。你可以完全控制自己需要用到的着色器来实现自己所需的功能。我们不需要用到所有的着色阶段，事实上，只有顶点着色器和片元着色器是必需的。细分和几何着色器是可选的步骤。

现在，我们将稍微深入到每个着色阶段当中，以了解更多的背景知识。可以理解，现在的阶段多少会让人感到望而却步，但是请不要担心。通过对一些概念的进一步理解，你将会很快习惯 OpenGL 的开发过程。

1.4.1 准备向 OpenGL 传输数据

OpenGL 需要将所有的数据都保存到缓存对象（buffer object）中，它相当于由 OpenGL 服务端维护的一块内存区域。我们可以使用多种方式来创建这样的数据缓存，不过最常用的方法就是使用例 1.1 中的 `glBufferData()` 命令。我们可能还需要对缓存做一些额外的设置，相关的内容请参见第 3 章。

1.4.2 将数据传输到 OpenGL

当将缓存初始化完毕之后，我们可以通过调用 OpenGL 的一个绘制命令来请求渲染几何图元，例 1.1 中的 `glDrawArrays()` 就是一个常用的绘制命令。

OpenGL 的绘制通常就是将顶点数据传输到 OpenGL 服务端。我们可以将一个顶点视为一个需要统一处理的数据包。这个包中的数据可以是我們需要的任何数据（也就是说，我们自己负责定义构成顶点的所有数据），通常其中几乎始终会包含位置数据。其他的数据可能用来决定一个像素的最终颜色。

第 3 章会更详细地介绍绘制命令的内容。

1.4.3 顶点着色

对于绘制命令传输的每个顶点，OpenGL 都会调用一个顶点着色器来处理顶点相关的数据。根据其他光栅化之前的着色器的活跃与否，顶点着色器可能会非常简单，例如，只是将数据复制并传递到下一个着色阶段，这叫做传递着色器（pass-through shader）；它也可能非常复杂，例如，执行大量的计算来得到顶点在屏幕上的位置（一般情况下，我们会用到变换矩阵（transformation matrix）的概念，参见第 5 章），或者通过光照的计算（参见第 7 章）来判断顶点的颜色，或者其他一些技法的实现。

通常来说，一个复杂的应用程序可能包含许多个顶点着色器，但是在同一时刻只能有一个顶点着色器起作用。

1.4.4 细分着色

顶点着色器处理每个顶点的关联数据之后，如果同时激活了细分着色器（tessellation shader），那么它将进一步处理这些数据。正如在第 9 章将会看到的，细分着色器会使用 Patch 来描述一个物体的形状，并且使用相对简单的 Patch 几何体连接来完成细分的工作，其结果是几何图元的数量增加，并且模型的外观会变得更加平顺。细分着色阶段会用到两个着色器来分别管理 Patch 数据并生成最终的形状。

1.4.5 几何着色

下一个着色阶段——几何着色——允许在光栅化之前对每个几何图元做更进一步的处理，例如创建新的图元。这个着色阶段也是可选的，但是我们在第 10 章里会体会到它的强大之处。

1.4.6 图元装配

前面介绍的着色阶段所处理的都是顶点数据，此外这些顶点之间如何构成几何图元的所有信息也会被传递到 OpenGL 当中。图元装配阶段将这些顶点与相关的几何图元之间组织起来，准备下一步的剪切和光栅化工作。

1.4.7 剪切

顶点可能会落在视口（viewport）之外——也就是我们可以进行绘制的窗口区域——此时与顶点相关的图元会做出改动，以保证相关的像素不会在视口外绘制。这一过程叫做剪切（clipping），它是由 OpenGL 自动完成的。

1.4.8 光栅化

剪切之后马上要执行的工作，就是将更新后的图元传递到光栅化单元，生成对应的片

元。我们可以将一个片元视为一个“候选的像素”，也就是可以放置在帧缓存中的像素，但是它也可能被最终剔除，不再更新对应的像素位置。之后的两个阶段将会执行片元的处理，即片元着色和逐片元的操作。

1.4.9 片元着色

最后一个可以通过编程控制屏幕上显示颜色的阶段，叫做片元着色阶段。在这个阶段中，我们使用着色器来计算片元的最终颜色（尽管在下一个阶段（逐片元的操作）时可能还会改变颜色一次）和它的深度值。片元着色器非常强大，在这里我们会使用纹理映射的方式，对顶点处理阶段所计算的颜色值进行补充。如果我们觉得不应该继续绘制某个片元，在片元着色器中还可以终止这个片元的处理，这一步叫做片元的丢弃（discard）。

如果我们需要更好地理解处理顶点的着色器和片元着色器之间的区别，可以用这种方法来记忆：顶点着色（包括细分和几何着色）决定了一个图元应该位于屏幕的什么位置，而片元着色使用这些信息来决定某个片元的颜色应该是什么。

1.4.10 逐片元的操作

除了我们在片元着色器里做的工作之外，片元操作的下一步就是最后的独立片元处理过程。在这个阶段里会使用深度测试（depth test，或者通常也称作 z-buffering）和模板测试（stencil test）的方式来决定一个片元是否是可见的。

如果一个片元成功地通过了所有激活的测试，那么它就可以被直接绘制到帧缓存中了，它对应的像素的颜色值（也可能包括深度值）会被更新，如果开启了融合（blending）模式，那么片元的颜色会与该像素当前的颜色相叠加，形成一个新的颜色值并写入帧缓存中。

从图 1-2 中可以看到，像素数据的传输也有一条路径。通常来说，像素数据来自图像文件，尽管它也可能是 OpenGL 直接渲染的。像素数据通常保存在纹理贴图当中，通过纹理映射的方式调用。在纹理阶段中我们可以从一张或者多张纹理贴图中查找所需的数据值。我们将在第 6 章了解有关纹理映射的内容。

现在我们已经了解 OpenGL 管线的基础知识，接下来回到例 1.1，用渲染管线的方式讲解其中的操作。

1.5 第一个程序：深入分析

1.5.1 进入 main() 函数

为了了解示例程序从一开始是如何运行的，首先了解一下 main() 函数当中都发生了什么。前面的 6 行使用 OpenGL Utility Toolkit 初始化和打开了一个渲染用的窗口。这方面的

详细介绍可以参见附录 A，这里只介绍每一行的执行结果。

```
int
main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA);
    glutInitWindowSize(512, 512);
    glutInitContextVersion(4, 3);
    glutInitContextProfile(GLUT_CORE_PROFILE);
    glutCreateWindow(argv[0]);

    if (glewInit()) {
        cerr << "Unable to initialize GLEW ... exiting" << endl;
        exit(EXIT_FAILURE);
    }

    init();

    glutDisplayFunc(display);

    glutMainLoop();
}
```

第一个函数 `glutInit()` 负责初始化 GLUT 库。它会处理向程序输入的命令行参数，并且移除其中与控制 GLUT 如何操作相关的部分（例如设置窗口的大小）。`glutInit()` 必须是应用程序调用的第一个 GLUT 函数，它会负责设置其他 GLUT 例程所必需的数据结构。

`glutInitDisplayMode()` 设置了程序所使用的窗口的类型。在这个例子中只需要设置窗口使用 RGBA 颜色空间（这会在第 4 章深入进行讨论）。除此之外，还可以给窗口设置更多的 OpenGL 特性，例如使用深度缓存或者动画效果。

`glutInitWindowSize()` 设置所需的窗口大小。如果不想在这里设置一个固定值，也可以先查询显示设备的尺寸，然后根据计算机的屏幕动态设置窗口的大小。

后面的两个调用 `glutInitContextVersion()` 和 `glutInitContextProfile()` 设置了我们所需的 OpenGL 环境（context）的类型——这是 OpenGL 内部用于记录状态设置和操作的数据结构。这个例子中使用 OpenGL 4.3 版本的核心模式（core profile）来创建环境。这个模式可以确保使用的只是 OpenGL 的最新特性，否则也可以选择另外一种兼容模式，这样自 OpenGL 1.0 版本以来的所有特性都可以在程序中使用。

随后的一个调用是 `glutCreateWindow()`，它的功能和它的名字一致。如果当前的系统环境可以满足 `glutInitDisplayMode()` 的显示模式要求，这里就会创建一个窗口（此时会调用计算机窗口系统的接口）。只有 GLUT 创建了一个窗口之后（其中也包含创建 OpenGL 环境的过程），我们才可以使用 OpenGL 相关的函数。

继续这个例子的内容，接下来会调用 `glewInit()` 函数，它属于我们用到的另一个辅助库 GLEW（OpenGL Extension Wrangler）。GLEW 可以简化获取函数地址的过程，并且包含了可以跨平台使用的其他一些 OpenGL 编程方法。如果没有 GLEW，我们可能还需要执行相当多的工作才能够运行程序。

到这里，我们已经完成了使用 OpenGL 之前的全部设置工作。在马上要介绍的 `init()` 例程中，我们将初始化 OpenGL 相关的所有数据，以便完成之后的渲染工作。

下一个例程是 `glutDisplayFunc()`，它设置了显示回调（display callback），即 GLUT 在每次更新窗口内容的时候会自动调用的例程。这里给 GLUT 库传入 `display()` 这个函数的地址，后文会讨论其中的内容。GLUT 可以使用一系列回调函数来处理诸如用户输入、重设窗口尺寸等不同的操作。附录 A 会详细地介绍 GLUT 库的内容。

`main()` 函数中调用的最后一个函数是 `glutMainLoop()`，这是一个无限执行的循环，它会负责一直处理窗口和操作系统的用户输入等操作。举例来说，`glutMainLoop()` 会判断窗口是否需要进行重绘，然后它就会自动调用 `glutDisplayFunc()` 中注册的函数。特别要注意的是，`glutMainLoop()` 是一个无限循环，因此不会执行在它之后的所有命令。

1.5.2 OpenGL 的初始化过程

下面将要讨论例 1.1 中的 `init()` 函数。首先再次列出与之相关的代码。

```
void
init(void)
{
    glGenVertexArrays(NumVAOs, VAOs);
    glBindVertexArray(VAOs[Triangles]);

    GLfloat vertices[NumVertices][2] = {
        { -0.90, -0.90 }, // Triangle 1
        {  0.85, -0.90 },
        { -0.90,  0.85 },
        {  0.90, -0.85 }, // Triangle 2
        {  0.90,  0.90 },
        { -0.85,  0.90 }
    };

    glGenBuffers(NumBuffers, Buffers);
    glBindBuffer(GL_ARRAY_BUFFER, Buffers[ArrayBuffer]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
                 vertices, GL_STATIC_DRAW);

    ShaderInfo shaders[] = {
        { GL_VERTEX_SHADER, "triangles.vert" },
        { GL_FRAGMENT_SHADER, "triangles.frag" },
        { GL_NONE, NULL }
    };

    GLuint program = LoadShaders(shaders);
    glUseProgram(program);

    glVertexAttribPointer(vPosition, 2, GL_FLOAT,
                          GL_FALSE, 0, BUFFER_OFFSET(0));
    glEnableVertexAttribArray(vPosition);
}
```

初始化顶点数组对象

在 `init()` 中使用了不少函数和数据。在函数的起始部分，我们调用 `glGenVertexArrays()` 分配了顶点数组对象（vertex-array object）。OpenGL 会因此分配一部

分顶点数组对象的名称供我们使用，在这里共有 NumVAOs 个对象，即这个全局变量所指代的数值。glGenVertexArrays() 的第二个参数返回的是对象名的数组，也就是这里的 VAOs。

我们对 glGenVertexArrays() 函数的完整解释如下：

```
void glGenVertexArrays(GLsizei n, GLuint *arrays);
```

返回 n 个未使用的对象名到数组 arrays 中，用作顶点数组对象。返回的名字可以用来分配更多的缓存对象，并且它们已经使用未初始化的顶点数组集合的默认状态进行了数值的初始化。

我们会发现很多 OpenGL 命令都是 glGen* 的形式，它们负责分配不同类型的 OpenGL 对象的名称。这里的名称类似 C 语言中的一个指针变量，我们必须分配内存并且用名称引用它之后，名称才有意义。在 OpenGL 中，这个分配的机制叫做绑定对象（bind an object），它是通过一系列 glBind* 形式的 OpenGL 函数集合去实现的。在这个例子中，我们通过 glBindVertexArray() 函数创建并且绑定了一个顶点数组对象。

```
void glBindVertexArray(GLuint array);
```

glBindVertexArray() 完成了三项工作。如果输入的变量 array 非 0，并且是 glGenVertexArrays() 所返回的，那么它将创建一个新的顶点数组对象并且与其名称关联起来。如果绑定到一个已经创建的顶点数组对象中，那么会激活这个顶点数组对象，并且直接影响对象中所保存的顶点数组状态。如果输入的变量 array 为 0，那么 OpenGL 将不再使用程序所分配的任何顶点数组对象，并且将渲染状态重设为顶点数组的默认状态。

如果 array 不是 glGenVertexArrays() 所返回的数值，或者它已经被 glDeleteVertexArrays() 函数释放了，那么这里将产生一个 GL_INVALID_OPERATION 错误。

这个例子中，在生成一个顶点数组对象的名字之后，就会使用 glBindVertexArray() 将它绑定起来。在 OpenGL 中这样的对象绑定操作非常常见，但是我们可能无法立即了解它做了什么。当我们第一次绑定对象时（例如，第一次用指定的对象名作为参数调用 glBind*()），OpenGL 内部会分配这个对象所需的内存并且将它作为当前对象，即所有后续的操作都会作用于这个被绑定的对象，例如，这里的顶点数组对象的状态就会被后面执行的代码所改变。在第一次调用 glBind*() 函数之后，新创建的对象都会初始化为其默认状态，而我们通常需要一些额外的初始化工作来确保这个对象可用。

绑定对象的过程有点类似设置铁路的道岔开关。一旦设置了开关，从这条线路通过的所有列车都会驶向对应的轨道。如果我们将开关设置到另一个状态，那么所有之后经过的列车都会驶向另一条轨道。OpenGL 的对象也是如此。总体上来说，在两种情况下我们需要绑定一个对象：创建对象并初始化它所对应的数据时；以及每次我们准备使用这个对象，而它并不是当前绑定的对象时。我们会在 display() 例程中看到后一种情况，即在程序运行过程中第二次调用 glBindVertexArray() 函数。

由于示例程序需要尽量短小，因此我们不打算做任何多余的操作。举例来说，在较大的程序里当我们完成对顶点数组对象的操作之后，是可以调用 `glDeleteVertexArrays()` 将它释放的。

```
void glDeleteVertexArrays(GLsizei n, GLuint *arrays);
```

删除 `n` 个在 `arrays` 中定义的顶点数组对象，这样所有的名称可以再次用作顶点数组。如果绑定的顶点数组已经被删除，那么当前绑定的顶点数组对象被重设为 0（类似执行了 `glBindBuffer()` 函数，并且输入参数为 0），而默认的顶点数组会变成当前对象。在 `arrays` 当中未使用的名称都会被释放，但是当前顶点数组的状态不会发生任何变化。

最后，为了确保程序的完整性，我们可以调用 `glIsVertexArray()` 检查某个名称是否已经被保留为一个顶点数组对象了。

```
GLboolean glIsVertexArray(GLuint array);
```

如果 `array` 是一个已经用 `glGenVertexArrays()` 创建且没有被删除的顶点数组对象的名称，那么返回 `GL_TRUE`。如果 `array` 为 0 或者不是任何顶点数组对象的名称，那么返回 `GL_FALSE`。

对于 OpenGL 中其他类型的对象，我们都可以看到类似的名为 `glDelete*` 和 `glIs*` 的例程。

分配顶点缓存对象

顶点数组对象负责保存一系列顶点的数据。这些数据保存到缓存对象当中，并且由当前绑定的顶点数组对象管理。我们只有一种顶点数组对象类型，但是却有很多种类型的对象，并且其中一部分对象并不负责处理顶点数据。正如前文中所提到的，缓存对象就是 OpenGL 服务端分配和管理的一块内存区域，并且几乎所有传入 OpenGL 的数据都是存储在缓存对象当中的。

顶点缓存对象的初始化过程与顶点数组对象的创建过程类似，不过需要有向缓存中添加数据的一个过程。

首先，我们需要创建顶点缓存对象的名称。我们调用的还是 `glGen*` 形式的函数，即 `glGenBuffers()`。在这个例子中，我们分配 `NumVBOs` 个对象（VBO 即 Vertex Buffer Objects）到数组 `buffers` 当中。以下是 `glGenBuffers()` 的详细介绍。

```
void glGenBuffers(GLsizei n, GLuint *buffers);
```

返回 `n` 个当前未使用的缓存对象名称，并保存到 `buffers` 数组中。返回到 `buffers` 中的名称不一定是连续的整型数据。

这里返回的名称只用于分配其他缓存对象，它们在绑定之后只会记录一个可用的状态。

0 是一个保留的缓存对象名称，`glGenBuffers()` 永远都不会返回这个值的缓存对象。

当分配缓存的名称之后，就可以调用 `glBindBuffer()` 来绑定它们了。由于 OpenGL 中有很多不同种类的缓存对象，因此绑定一个缓存时，需要指定它所对应的类型。在这个例子中，由于是将顶点数据保存到缓存当中，因此使用 `GL_ARRAY_BUFFER` 类型。缓存对象的类型现在共有 8 种，分别用于不同的 OpenGL 功能实现。本书后面的章节会分别讨论各种类型的对应操作。

`glBindBuffer()` 函数的详细介绍如下。

```
void glBindBuffer(GLenum target, GLuint buffer);
```

指定当前激活的缓存对象。`target` 必须设置为以下类型中的一个：`GL_ARRAY_BUFFER`、`GL_ELEMENT_ARRAY_BUFFER`、`GL_PIXEL_PACK_BUFFER`、`GL_PIXEL_UNPACK_BUFFER`、`GL_COPY_READ_BUFFER`、`GL_COPY_WRITE_BUFFER`、`GL_TRANSFORM_FEEDBACK_BUFFER` 和 `GL_UNIFORM_BUFFER`。`buffer` 设置的是要绑定的缓存对象名称。

`glBindBuffer()` 完成了三项工作：1) 如果是第一次绑定 `buffer`，且它是一个非零的无符号整型，那么将创建一个与该名称相对应的新缓存对象。2) 如果绑定到一个已经创建的缓存对象，那么它将成为当前被激活的缓存对象。3) 如果绑定的 `buffer` 值为 0，那么 OpenGL 将不再对当前 `target` 应用任何缓存对象。

所有的缓存对象都可以使用 `glDeleteBuffers()` 直接释放。

```
void glDeleteBuffers(GLsizei n, const GLuint *buffers);
```

删除 `n` 个保存在 `buffers` 数组中的缓存对象。被释放的缓存对象可以重用（例如，使用 `glGenBuffers()`）。

如果删除的缓存对象已经被绑定，那么该对象的所有绑定将会重置为默认的缓存对象，即相当于用 0 作为参数执行 `glBindBuffer()` 的结果。如果试图删除不存在的缓存对象，或者缓存对象为 0，那么将忽略该操作（不会产生错误）。

我们也可以用 `glIsBuffer()` 来判断一个整数值是否是一个缓存对象的名称。

```
GLboolean glIsBuffer(GLuint buffer);
```

如果 `buffer` 是一个已经分配并且没有释放的缓存对象的名称，则返回 `GL_TRUE`。如果 `buffer` 为 0 或者不是缓存对象的名称，则返回 `GL_FALSE`。

将数据载入缓存对象

初始化顶点缓存对象之后，我们需要把顶点数据从对象传输到缓存对象当中。这一步是通过 `glBufferData()` 例程完成的，它主要有两个任务：分配顶点数据所需的存储空间，然

后将数据从应用程序的数组中拷贝到 OpenGL 服务端的内存中。

有可能在很多不同的场景中多次应用 `glBufferData()`，因此我们有必要在这里深入了解它的过程，尽管我们在这本书中还会多次遇到这个函数。首先，`glBufferData()` 的详细定义介绍如下。

```
void glBufferData(GLenum target, GLsizeiptr size, const GLvoid *data, GLenum usage);
```

在 OpenGL 服务端内存中分配 `size` 个存储单元（通常为 `byte`），用于存储数据或者索引。如果当前绑定的对象已经存在了关联的数据，那么会首先删除这些数据。

对于顶点属性数据，`target` 设置为 `GL_ARRAY_BUFFER`；索引数据为 `GL_ELEMENT_ARRAY_BUFFER`；OpenGL 的像素数据为 `GL_PIXEL_UNPACK_BUFFER`；对于从 OpenGL 中获取的像素数据为 `GL_PIXEL_PACK_BUFFER`；对于缓存之间的复制数据为 `GL_COPY_READ_BUFFER` 和 `GL_COPY_WRITE_BUFFER`；对于纹理缓存中存储的纹理数据为 `GL_TEXTURE_BUFFER`；对于通过 `transform feedback` 着色器获得的结果设置为 `GL_TRANSFORM_FEEDBACK_BUFFER`；而一致变量要设置为 `GL_UNIFORM_BUFFER`。

`size` 表示存储数据的总数量。这个数值等于 `data` 中存储的元素的总数乘以单位元素存储空间的结果。

`data` 要么是一个客户端内存的指针，以便初始化缓存对象，要么是 `NULL`。如果传入的指针合法，那么将会有 `size` 大小的数据从客户端拷贝到服务端。如果传入 `NULL`，那么将保留 `size` 大小的未初始化的数据，以备后用。

`usage` 用于设置分配数据之后的读取和写入方式。可用的方式包括 `GL_STREAM_DRAW`、`GL_STREAM_READ`、`GL_STREAM_COPY`、`GL_STATIC_DRAW`、`GL_STATIC_READ`、`GL_STATIC_COPY`、`GL_DYNAMIC_DRAW`、`GL_DYNAMIC_READ` 和 `GL_DYNAMIC_COPY`。

如果所需的 `size` 大小超过了服务端能够分配的额度，那么 `glBufferData()` 将产生一个 `GL_OUT_OF_MEMORY` 错误。如果 `usage` 设置的不是可用的模式值，那么将产生 `GL_INVALID_VALUE` 错误。

一下子理解这么多的内容可能有点困难，但是这个函数在后面的学习中会多次出现，因此有必要在本书的开始部分就详细地对它做出讲解。

在上面的例子中，直接调用了 `glBufferData()`。因为顶点数据就保存在一个 `vertices` 数组当中。如果需要静态地从程序中加载顶点数据，那么我们可能需要从模型文件中读取这些数值，或者通过某些算法来生成。由于我们的数据是顶点属性数据，因此设置这个缓存为 `GL_ARRAY_BUFFER`，即指定它的第一个参数。我们还需要指定内存分配的大小（单位为 `byte`），因此直接使用 `sizeof(vertices)` 来完成计算。最后，我们需要指定数据在 OpenGL 中使用的方式。因为我们只是用它来绘制几何体，不会在运行时对它做出修改，所

以设置 `glBufferData()` 的 `usage` 参数为 `GL_STATIC_DRAW`。

除此之外, `usage` 还有一系列的选项可用, 第3章会详细地介绍它们。

如果我们仔细观察 `vertices` 数组中的数值, 就会发现它们在 `x` 和 `y` 方向都被限定在 `[-1, 1]` 的范围内。实际上, OpenGL 只能够绘制坐标空间内的几何体图元。而具有该范围限制的坐标系统也称为规格化设备坐标系统 (Normalized Device Coordinate, NDC)。这听起来好像是一个巨大的限制, 但实际上并不是问题。第5章会介绍将3维空间中的复杂物体映射到规格化设备坐标系中的数学方法。在这个例子中直接使用 NDC 坐标, 不过实际上我们通常会使用一些更为复杂的坐标空间。

现在, 我们已经成功地创建了一个顶点数组对象, 并且将它传递到缓存对象中。下一步, 我们要设置程序中用到的着色器了。

初始化顶点与片元着色器

对于每一个 OpenGL 程序, 当它所使用的 OpenGL 版本高于或等于 3.1 时, 都需要指定至少两个着色器: 顶点着色器和片元着色器。在这个例子中, 我们通过一个辅助函数 `LoadShaders()` 来实现这个要求, 它需要输入一个 `ShaderInfo` 结构体数组 (这个结构体的实现过程可以参见示例源代码的头文件 `LoadShaders.h`)。

对于 OpenGL 程序员而言, 着色器就是使用 OpenGL 着色语言 (OpenGL Shading Language, GLSL) 编写的一个小型函数。GLSL 是构成所有 OpenGL 着色器的语言, 它与 C++ 语言非常类似, 尽管 GLSL 中的所有特性并不能用于 OpenGL 的每个着色阶段。我们可以以字符串的形式传输 GLSL 着色器到 OpenGL。不过为了简化这个例子, 并且让读者更容易地使用着色器去进行开发, 我们选择将着色器字符串的内容保存到文件中, 并且使用 `LoadShaders()` 读取文件和创建 OpenGL 着色器程序。使用 OpenGL 着色器进行编程的具体过程可以参见第2章的内容。

为了帮助读者尽快开始了解着色器的内容, 我们并没有将所有相关的细节内容都立即呈现出来。事实上, 本书后面的内容都会与 GLSL 的具体实现相关, 而现在, 我们只需要在例 1.2 中对顶点着色器的代码做一个深入了解。

例 1.2 `triangles.cpp` 对应的顶点着色器: `triangles.vert`

```
#version 430 core

layout(location = 0) in vec4 vPosition;

void
main()
{
    gl_Position = vPosition;
}
```

没错, 它的内容只有这么多。事实上这就是我们之前所说的传递着色器 (pass-through shader) 的例子。它只负责将输入数据拷贝到输出数据中。不过即便如此, 我们也还是要展

开深入讨论。

第一行“`#version 430 core`”指定了我们所用的 OpenGL 着色语言的版本。这里的“430”表示我们准备使用 OpenGL 4.3 对应的 GLSL 语言。这里的命名规范是基于 OpenGL 3.3 版本的。在那之前的 OpenGL 版本中，版本号所用的数字是完全不一样的（详细介绍参见第 2 章）。这里的“core”表示我们将使用 OpenGL 核心模式（core profile），这与之前 GLUT 的函数 `glutInitContextProfile()` 设置的内容应当一致。每个着色器的第一行都应该设置“`#version`”，否则系统会假设使用“110”版本，但是这与 OpenGL 核心模式并不兼容。我们在本书中只针对 330 版本及以上的着色器以及它的特性进行讲解；如果这个版本号不是最新的版本，那么程序的可移植性应该会更好，但是你将无法使用最新的系统特性。

下一步，我们分配了一个着色器变量。着色器变量是着色器与外部世界的联系所在。换句话说，着色器并不知道自己的数据从哪里来，它只是在每次运行时直接获取数据对应的输入变量。而我们必须自己完成着色管线的装配（在后面内容中你将了解它所表示的意思），然后才可以将应用程序中的数据与不同的 OpenGL 着色阶段互相关联。

在这个简单的例子中，只有一个名为 `vPosition` 的输入变量，它被声明为“in”。事实上，就算是这一行也包含了很多的内容。

```
layout(location = 0) in vec4 vPosition;
```

我们最好从右往左来解读这一行的信息。

□ 显而易见 `vPosition` 就是变量的名称。我们使用一个字符“v”作为这个顶点属性名称的前缀。这个变量所保存的是顶点的位置信息。

□ 下一个字段是 `vec4`，也就是 `vPosition` 类型。在这里它是一个 GLSL 的四维浮点数向量。GLSL 中有非常多的数据类型，这会在第 2 章里详细介绍。

你也许已经注意到，我们在例 1.1 的程序中对每个顶点只设置了两个坐标值，但是在顶点着色器中却使用 `vec4` 来表达它。那么另外两个坐标值来自哪里？事实上 OpenGL 会用默认数值自动填充这些缺失的坐标值。而 `vec4` 的默认值为 (0, 0, 0, 1)，因此当仅指定了 *x* 和 *y* 坐标的时候，其他两个坐标值 (*z* 和 *w*) 将被自动指定为 0 和 1。

□ 在类型之前就是我们刚才提到的 `in` 字段，它指定了数据进入着色器的流向。正如你所见，这里还可以声明变量为 `out`。不过我们在这里暂时还不会用到它。

□ 最后的字段是 `layout(location = 0)`，它也叫做布局限定符（layout qualifier），目的是为变量提供元数据（meta data）。我们可以使用布局限定符来设置很多不同的属性，其中有些是与不同的着色阶段相关的。

在这里，设置 `vPosition` 的位置属性 `location` 为 0。这个设置与 `init()` 函数的最后两行会共同起作用。

最后，在着色器的 `main()` 函数中实现它的主体部分。OpenGL 的所有着色器，无论是处于哪个着色阶段，都会有一个 `main()` 函数。对于这个着色器而言，它所实现的就是将输入的顶点位置复制到顶点着色器的指定输出位置 `gl_Position` 中。后文中我们将会了解

到 OpenGL 所提供的一些着色器变量，它们全部都是以 `gl_` 作为前缀的。

与之类似，我们也需要一个片元着色器来配合顶点着色器的工作。例 1.3 所示就是片元着色器的内容。

例 1.3 `triangles.cpp` 对应的片元着色器：`triangles.frag`

```
#version 430 core

out vec4 fColor;

void
main()
{
    fColor = vec4(0.0, 0.0, 1.0, 1.0);
}
```

令人高兴的是，这里大部分的代码看起来很类似，虽然它们分别属于两个完全不同的着色器类型。我们还是需要声明版本号、变量以及 `main()` 函数。这里存在着一些差异，但是你依然可以看出，几乎所有着色器的基本结构都是这样的。

片元着色器的重点内容如下：

- ❑ 声明的变量名为 `fColor`。没错，它使用了 `out` 限定符！在这里，着色器将会把 `fColor` 对应的数值输出，而这也就是片元所对应的颜色值（因此这里用到了前缀字符“f”）。
- ❑ 设定片元的颜色。在这里，每个片元都会设置一个四维的向量。OpenGL 中的颜色是通过 RGB 颜色空间来表示的，其中每个颜色分量（R 表示红色，G 表示绿色，B 表示蓝色）的范围都是 `[0, 1]`。留心的读者在这里可能会问，“但是这是一个四维的向量”。没错，OpenGL 实际上使用了 RGBA 颜色空间，其中第四个值并不是颜色值。它叫做 `alpha` 值，专用于度量透明度。第 4 章将深入讨论这个话题，但是在现在，我们将它直接设置为 `1.0`，这表示片元的颜色是完全不透明的。

片元着色器具有非常强大的功能，我们可以用它来实现非常多的算法和技巧。

我们已经基本完成了初始化的过程。`init()` 中最后的两个函数指定了顶点着色器的变量与我们存储在缓存对象中数据的关系。这也就是我们所说的着色管线装配的过程，即将应用程序与着色器之间，以及不同着色阶段之间的数据通道连接起来。

为了输入顶点着色器的数据，也就是 OpenGL 将要处理的所有顶点数据，需要在着色器中声明一个 `in` 变量，然后使用 `glVertexAttribPointer()` 将它关联到一个顶点属性数组。

```
void glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean
normalized, GLsizei stride, const GLvoid *pointer);
```

设置 `index`（着色器中的属性位置）位置对应的数据值。`pointer` 表示缓存对象中，从起始位置开始计算的数组数据的偏移值（假设起始地址为 0），使用基本的系统单位（byte）。`size` 表示每个顶点需要更新的分量数目，可以是 1、2、3、4 或者 `GL_BGRA`。`type` 指定了数组中每个元素的数据类型（`GL_BYTE`、`GL_UNSIGNED_BYTE`、`GL_`

SHORT、GL_UNSIGNED_SHORT、GL_INT、GL_UNSIGNED_INT、GL_FIXED、GL_HALF_FLOAT、GL_FLOAT 或 GL_DOUBLE)。normalized 设置顶点数据在存储前是否需要归一化（或者使用 `glVertexAttribFourN*()` 函数）。stride 是数组中每两个元素之间的大小偏移值（byte）。如果 stride 为 0，那么数据应该紧密地封装在一起。

看起来我们有一大堆事情需要考虑，因为 `glVertexAttribPointer()` 其实是一个非常灵活 的命令。只要在内存中数据是规范组织的（保存在一个连续的数组中，不使用其他基于节点的 容器，比如链表），我们就可以使用 `glVertexAttribPointer()` 告诉 OpenGL 直接从内存中 获取数据。在例子中，vertices 里已经包含了我们所需的全部信息。表 1-2 所示为在这个 例子里 `glVertexAttribPointer()` 中各个参数的设置及其意义。

表 1-2 判断 `glVertexAttribPointer()` 中参数的例子

参数名称	数值	解释
index	0	这就是顶点着色器中输入变量的 location 值，也就是之前的 <code>vPosition</code> 。在着色器中这个值用来直接指定布局限位符，不过也可以用于着色器编译后的判断
size	2	这就是数组中每个顶点的元素数目。vertices 中共有 <code>NumVertices</code> 个顶点，每个顶点有两个元素值
type	GL_FLOAT	这个枚举量表示 <code>GLfloat</code> 类型
normalized	GL_FALSE	这里设置为 <code>GL_FALSE</code> 的原因有两个：最重要的第一点是因为它表示位置坐标值，因此可以是任何数值，不应当限制在 <code>[-1, 1]</code> 的归一化范围内，第二点是因为它不是整型（ <code>GLint</code> 或者 <code>GLshort</code> ）
stride	0	数据在这里是“紧密封装”的，即每组数据值在内存中都是立即与下一组数据值相衔接的，因此可以直接设置为 0
pointer	<code>BUFFER_OFFSET(0)</code>	这里设置为 0，因为数据是从缓存对象的第一个字节（地址为 0）开始的

希望上面的参数解释能够帮助你判断自己的数据结构所对应的数值。在后文中我们还会多次用到 `glVertexAttribPointer()` 来实现示例程序。

这里我们还用到了一个技巧，就是用 `glVertexAttribPointer()` 中的 `BUFFER_OFFSET` 宏来指定偏移量。这个宏的定义没有什么特别的，如下所示。

```
#define BUFFER_OFFSET(offset) ((void *) (offset))
```

在以往版本的 OpenGL 当中并不需要用到这个宏[⊖]，不过现在我们希望使用它来设置数据在缓存对象中的偏移量，而不是像 `glVertexAttribPointer()` 的原型那样直接设置一个指向内存块的指针。

在 `init()` 中，我们还有一项任务没有完成，那就是启用顶点属性数组。我们通过调用 `glEnableVertexAttribArray()` 来完成这项工作，同时将 `glVertexAttribPointer()` 初始化的属性数组指针索引传入这个函数。有关 `glEnableVertexAttribArray()` 的详细解释如下所示。

⊖ 在 OpenGL 的早期版本当中（3.1 版本之前），顶点属性数据可以直接保存在应用程序内存中，而不一定是 GPU 的缓存对象，因此这个时候使用指针的形式也是合理的。

```
void glEnableVertexAttribArray(GLuint index);
void glDisableVertexAttribArray(GLuint index);
```

设置是否启用与 index 索引相关联的顶点数组。index 必须是一个介于 0 到 GL_MAX_VERTEX_ATTRIBS-1 之间的值。

现在，我们只需要完成绘制的工作即可。

1.5.3 第一次使用 OpenGL 进行渲染

在设置和初始化所有数据之后，渲染的工作（在这个例子中）就非常简单了。display() 函数只有 4 行代码，不过它所包含的内容在所有 OpenGL 程序中都会用到。下面我们先阅读其中的代码。

```
void
display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);

    glBindVertexArray(VAOs[Triangles]);
    glDrawArrays(GL_TRIANGLES, 0, NumVertices);

    glFlush();
}
```

首先，我们要清除帧缓存的数据再进行渲染。清除的工作由 glClear() 完成。

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

清除指定的缓存数据并重设为当前的清除值。mask 是一个可以通过逻辑“或”操作来指定多个数值的参数，可用的数值如表 1-3 所示。

我们会在第 4 章中学习深度缓存（depth buffer）与模板缓存（stencil buffer）的内容，当然还有对颜色缓存（color buffer）的深入探讨。

现在你可能想知道，glClear() 会使用一个什么样的清除数值。在这个例子中，我们直接使用 OpenGL 默认的清除颜色，即黑色。如果要改变清除颜色的数值，可以使用 glClearColor()。

表 1-3 清除缓存

缓存	名称
颜色缓存	GL_COLOR_BUFFER_BIT
深度缓存	GL_DEPTH_BUFFER_BIT
模板缓存	GL_STENCIL_BUFFER_BIT

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

设置当前使用的清除颜色值，用于 RGBA 模式下对颜色缓存的清除工作。（参见第 4 章有关 RGBA 模式的内容）这里的 red、green、blue、alpha 都会被截断到 [0, 1] 的范围内。默认的清除颜色是 (0, 0, 0, 0)，在 RGBA 模式下它表示黑色。

清除颜色本身也是 OpenGL 状态机制的一个例子，它的数值会一直保留在当前 OpenGL 环境当中。OpenGL 有一个庞大的状态量列表（详细的介绍参见附录 D），当创建一个新的 OpenGL 环境时，所有的状态量都会被初始化为默认数值。因为 OpenGL 会保留所有更改的状态值，所以我们可以减少设置状态数值的次数。

举例说明清除颜色的用法，比如我们希望将当前视口的背景颜色设置为白色，那么需要调用 `glClearColor(1, 1, 1, 1)`。但是我们应该在什么时候调用这个函数呢？当然，我们可以直接在 `display()` 函数中调用 `glClear()` 之前调用它。但是这样的话，除了第一次进入循环之时，其他所有对 `glClearColor()` 的调用都是多余的——因为 OpenGL 在每次渲染时都会重复设置清除颜色的状态值为白色。另一个效率更高的方法是在 `init()` 函数中设置清除颜色。事实上，这样我们就可以避免冗余的状态切换；所有在程序运行时不会发生变化的数值都应该在 `init()` 中设置。当然，冗余的函数调用本身并没有危害，但是它会造成程序运行速度稍微变慢。

试一试 在 `triangles.cpp` 中添加对于 `glClearColor()` 的调用。

使用 OpenGL 进行绘制

例子中后面两行的工作是选择我们准备绘制的顶点数据，然后请求进行绘制。首先调用 `glBindVertexArray()` 来选择作为顶点数据使用的顶点数组。正如前文中提到的，我们可以用这个函数来切换程序中保存的多个顶点数据对象集合。

其次调用 `glDrawArrays()` 来实现顶点数据向 OpenGL 管线的传输。

```
void glDrawArrays(GLenum mode, GLint first, GLsizei count);
```

使用当前绑定的顶点数组元素来建立一系列的几何图元，起始位置为 `first`，而结束位置为 `first + count - 1`。`mode` 设置了构建图元的类型，它可以是 `GL_POINTS`、`GL_LINES`、`GL_LINE_STRIP`、`GL_LINE_LOOP`、`GL_TRIANGLES`、`GL_TRIANGLE_STRIP`、`GL_TRIANGLE_FAN` 和 `GL_PATCHES` 中的任意一种。

在这个例子中，我们使用 `glVertexAttribPointer()` 设置渲染模式为 `GL_TRIANGLES`，起始位置位于缓存的 0 偏移位置，共渲染 `NumVertices` 个元素（这个例子中为 6 个），这样就可以渲染出独立的三角形图元了。我们会在第 3 章详细介绍所有的图元形状。

试一试 修改 `triangles.cpp` 让它渲染一个不同类型的几何图元，例如 `GL_POINTS` 或者 `GL_LINES`。你可以使用上文中列出的任何一种图元，但是有些的结果可能会比较奇怪，此外 `GL_PATCHES` 类型是不会输出任何结果的，因为它是用于细分着色器的，参见第 9 章的内容。

最后在 `display()` 函数中调用 `glFlush()`，即强制所有进行中的 OpenGL 命令立即完成并传输到 OpenGL 服务端处理。在后文中我们很快就会把 `glFlush()` 替换为另一个更为平滑的命令，但是这样的话还需要对当前这个例子进行更多的设置。

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

强制之前的 OpenGL 命令立即执行，这样就可以保证它们在一定时间内全部完成。

深入理解

在你的 OpenGL 编程生涯的某个时刻，你可能会被问及（或者自问）“这需要多少时间”？“这”可能是渲染一个物体、绘制一整个场景，或者 OpenGL 能够实现的其他操作。为了能够精确地度量和执行自己的任务，我们有必要了解 OpenGL 是在什么时候完成这些操作的。

上文所述的 `glFlush()` 命令看起来像是一个正确的答案，但是它不是。事实上 `glFlush()` 只是强制所有运行中的命令送入 OpenGL 服务端而已，并且它会立即返回——它并不会等待所有的命令完成，而等待却是我们所需要的。为此，我们需要使用 `glFinish()` 命令，它会一直等待所有当前的 OpenGL 操作完成后，再返回。

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

强制所有当前的 OpenGL 命令立即执行，并且等待它们全部完成。



注意 你最好只是在开发阶段使用 `glFinish()`——如果你已经完成了开发的工作，那么最好去掉对这个命令的调用。虽然它对于判断 OpenGL 命令运行效率很有帮助，但是对于程序的整体性能却有着相当的拖累。

启用和禁用 OpenGL 的操作

在第一个例子当中有一个重要的特性并没有用到，但是在后文中我们会反复用到它，那就是对于 OpenGL 操作模式的启用和禁用。绝大多数的操作模式都可以通过 `glEnable()` 和 `glDisable()` 命令开启或者关闭。

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

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

`glEnable()` 会开启一个模式，`glDisable()` 会关闭它。有很多枚举量可以作为模式参数传入 `glEnable()` 和 `glDisable()`。例如 `GL_DEPTH_TEST` 可以用来开启或者关闭深度测试；`GL_BLEND` 可以用来控制融合的操作，而 `GL_RASTERIZER_DISCARD` 用于 transform feedback 过程中的高级渲染控制。

很多时候，尤其是我们用 OpenGL 编写的库需要提供给其他程序员使用的时候，可以根据自己的需要来判断是否开启某个特性，这时候可以使用 `glIsEnabled()` 来返回是否启用指定模式的信息。

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

根据是否启用当前指定的模式，返回 `GL_TRUE` 或者 `GL_FALSE`。





着色器基础

本章目标

阅读完本章内容之后，你将会具备以下能力：

- ❑ 区分 OpenGL 创建图像所用的不同类型的着色器。
- ❑ 使用 OpenGL 着色语言构建和编译着色器。
- ❑ 使用 OpenGL 中提供的多种机制将数据传入着色器。
- ❑ 使用高级 GLSL 着色技巧来创建可复用性更强的着色器。

本章将介绍如何在 OpenGL 中使用可编程着色器（shader）。首先介绍 OpenGL 着色语言（OpenGL Shading Language，通常也称作 GLSL），然后详细解释着色器将如何与 OpenGL 应用程序交互。

这一章将会包含以下几节：

- ❑ 2.1 节将会介绍 OpenGL 应用程序中经常用到的可编程图形着色器。
- ❑ 2.2 节将会详细解释 OpenGL 可编程管线的每个阶段。
- ❑ 2.3 节将会介绍 OpenGL 着色语言。
- ❑ 2.4 节将会介绍如何构建着色器变量，以及它们是如何与应用程序或者在阶段之间共享的。
- ❑ 2.5 节将会介绍将 GLSL 着色器转换为可编程着色器程序的过程，然后你就可以在 OpenGL 应用程序中使用它了。
- ❑ 2.6 节将会介绍一种增加着色器可用性的方法，它可以在不用重新编译着色器的前提下选择执行某个子程序。
- ❑ 2.7 节将介绍如何使用多个着色器的元素组合为单一的、可配置的图形管线。

2.1 着色器与 OpenGL

现代 OpenGL 渲染管线严重依赖着色器来处理传入的数据。如果不使用着色器，那么用 OpenGL 可以做的事情可能只有清除窗口内容了，可见着色器对于 OpenGL 的重要性。在 OpenGL 3.0 版本以前（含该版本），或者如果你用到了兼容模式（compatibility profile）环境，OpenGL 还包含一个固定功能管线（fixed-function pipeline），它可以在不使用着色器的情况下处理几何与像素数据。从 3.1 版本开始，固定功能管线从核心模式中去除，因此我们必须使用着色器来完成工作。

无论是 OpenGL 还是其他图形 API 的着色器，通常都是通过一种特殊的编程语言去编写的。对于 OpenGL 来说，我们会使用 GLSL，也就是 OpenGL Shading Language，它是在 OpenGL 2.0 版本左右发布的（在之前它属于扩展功能）。它与 OpenGL 的发展是同时进行的，并通常会与每个新版本的 OpenGL 一起更新。虽然 GLSL 是一种专门为图形开发设计的编程语言，但是你会发现它与“C”语言非常类似，当然还有一点 C++ 的影子。

本章将介绍编写着色器的方法，以循序渐进的方式讲解 GLSL，讨论如何编译着色器并且与应用程序相结合，以及如何将应用程序中的数据传递到不同的着色器中。

2.2 OpenGL 的可编程管线

在第 1 章已经对 OpenGL 的渲染管线进行了一个概要的介绍，现在将更加详细地介绍它的每个阶段以及其中所承载的工作。4.3 版本的图形管线有 4 个处理阶段，还有 1 个通用计算阶段，每个阶段都需要由一个专门的着色器进行控制。

1) 顶点着色阶段（vertex shading stage）将接收你在顶点缓存对象中给出的顶点数据，独立处理每个顶点。这个阶段对于所有的 OpenGL 程序都是必需的，并且必须绑定一个着色器。第 3 章将对顶点着色的操作进行介绍。

2) 细分着色阶段（tessellation shading stage）是一个可选的阶段，与应用程序中显式地指定几何图元的方法不同，它会在 OpenGL 管线内部生成新的几何体。这个阶段启用之后，会收到来自顶点着色阶段的输出数据，并且对收到的顶点进行进一步的处理。第 9 章会介绍细分着色阶段的内容。

3) 几何着色阶段（geometry shading stage）也是一个可选的阶段，它会在 OpenGL 管线内部对所有几何图元进行修改。这个阶段会作用于每个独立的几何图元。此时你可以选择从输入图元生成更多的几何体，改变几何图元的类型（例如将三角形转化为线段），或者放弃所有的几何体。如果这个阶段被启用，那么几何着色阶段的输入可能会来自顶点着色阶段完成几何图元的顶点处理之后，也可能来自细分着色阶段生成的图元数据（如果它也被启用）。第 10 章会介绍几何着色阶段的内容。

4) OpenGL 着色管线的最后一个部分是片元着色阶段（Fragment shading stage）。这个阶

段会处理 OpenGL 光栅化之后生成的独立片元（如果启用了采样着色的模式，就是采样数据），并且这个阶段也必须绑定一个着色器。在这个阶段中，计算一个片元的颜色和深度值，然后传递到管线的片元测试和混合的模块。片元着色阶段的介绍将会贯穿本书的很多章节。

5) 计算着色阶段 (Compute shading stage) 和上述阶段不同，它并不是图形管线的一部分，而是在程序中相对独立的一个阶段。计算着色阶段处理的并不是顶点和片元这类图形数据，而是应用程序给定范围的内容。计算着色器在应用程序中可以处理其他着色器程序所创建和使用的缓存数据。这其中也包括帧缓存的后处理效果，或者我们所期望的任何事物。计算着色器的介绍参见第 12 章。

现在我们需要大概了解一个重要的概念，就是着色阶段之间数据传输的方式。正如在第 1 章中看到的，着色器类似一个函数调用的方式——数据传输进来，经过处理，然后再传输出去。例如，在 C 语言中，这一过程可以通过全局变量，或者函数参数来完成。GLSL 与之稍有差异。每个着色器看起来都像是个完整的 C 程序，它的输入点就是一个名为 `main()` 的函数。但与 C 不同的是，GLSL 的 `main()` 函数没有任何参数，在某个着色阶段中输入和输出的所有数据都是通过着色器中的特殊全局变量来传递的（请不要将它们与应用程序中的全局变量相混淆——着色器变量与你在应用程序代码中声明的变量是完全不相干的）。例如，下面的例 2.1 中的内容。

例 2.1 一个简单的顶点着色器

```
#version 330 core

in vec4  vPosition;
in vec4  vColor;

out vec4  color;

uniform mat4  ModelViewProjectionMatrix;

void
main()
{
    color = vColor;
    gl_Position = ModelViewProjectionMatrix * vPosition;
}
```

虽然这是一个非常短的着色器，但是还是有许多需要注意的地方。我们先不考虑自己需要对哪个着色阶段进行编程，所有常见的着色器代码都应该与这个例子有着相同的结构。在程序起始的位置总是要使用 `#version` 来声明所使用的版本。

首先，应注意这些全局变量。OpenGL 会使用输入和输出变量来传输着色器所需的数据。除了每个变量都有一个类型之外（例如 `vec4`，后文将深入地进行介绍），OpenGL 还定义了 `in` 变量将数据拷贝到着色器中，以及 `out` 变量将着色器的内容拷贝出去。这些变量的值会在 OpenGL 每次执行着色器的时候更新（如果 OpenGL 处理的是顶点，那么这里会为每个顶点传递新的值；如果是处理片元，那么将为每个片元传递新值）。另一类变量是直接来自 OpenGL 应用程序中接收数据的，称作 `uniform` 变量。`uniform` 变量不会随着顶点或者片元

的变化而变化，它对于所有的几何体图元的值都是一样的，除非应用程序对它进行了更新。

2.3 OpenGL 着色语言概述

本节将会对 OpenGL 中着色语言的使用进行一个概述。GLSL 具备了 C++ 和 Java 的很多特性，它也被 OpenGL 所有阶段中使用的着色器所支持，尽管不同类型的着色器也会有一些专属特性。我们首先介绍 GLSL 的需求、类型，以及其他所有着色阶段所共有的语言特性，然后对每种类型的着色器中的专属特性进行讨论。

2.3.1 使用 GLSL 构建着色器

从这里出发

一个着色器程序和一个 C 程序类似，都是从 `main()` 函数开始执行的。每个 GLSL 着色器程序一开始都如下所示：

```
#version 330 core

void
main()
{
    // 在这里编写代码
}
```

这里的 `//` 是注释符号，它到当前行的末尾结束，这一点与 C 语言一致。此外，着色器程序也支持 C 语言形式的多行注释符号——`/*` 和 `*/`。但是，与 ANSI C 语言不同，这里的 `main()` 函数不需要返回一个整数值，它被声明为 `void`。此外，着色器程序与 C 语言以及衍生的各种语言相同，每一行的结尾都必须有一个分号。这里给出的 GLSL 程序绝对合法，可以直接编译甚至运行，但是它的功能目前还是空白。为了能够进一步丰富着色器代码中的内容，下面将进一步介绍变量的概念以及相关的操作。

变量的声明

GLSL 是一种强类型语言，所有变量都必须事先声明，并且要给出变量的类型。变量名称的命名规范与 C 语言相同：可以使用字母、数字，以及下划线字符（`_`）来组成变量的名字。但是数字或者下划线不能作为变量名称的第一个字符。此外，变量名称也不能包含连续的下划线（这些名称是 GLSL 保留使用的）。

表 2-1 中给出了 GLSL 支持的基本数据类型。

这些类型（以及后文中它们的聚合类型）都是透明的。也就是说，它们的内部形式都是

表 2-1 GLSL 中的基本数据类型

类型	描述
float	IEEE 32 位浮点值
double	IEEE 64 位浮点值
int	有符号二进制补码的 32 位整数
uint	无符号的 32 位整数
bool	布尔值

暴露出来的，因此着色器代码中可以假设其内部的构成方式。

与之对应的一部分类型，称作不透明类型，它们的内部形式没有暴露出来。这些类型包括采样器（sampler）、图像（image），以及原子计数器（atomic counter）。它们所声明的变量相当于一个不透明的句柄，可以用来读取纹理贴图、图像，以及原子计数器数据，参见第4章。

不同的采样器类型以及它们的应用可以参见第6章。

变量的作用域

虽然所有的变量都需要声明，但是我们可以使用它们之前的任何时候声明这些变量（这一点与 C++ 一致，即变量声明需要放在一个代码块中靠前的语句里）。我们可以对照 C++ 的语法来了解 GLSL 变量的作用域规则，如下所示：

- ❑ 在任何函数定义之外声明的变量拥有全局作用域，因此对着色器程序中的所有函数都是可见的。
- ❑ 在一组大括号之内（例如函数定义、循环或者“if”引领的代码块等）声明的变量，只能在大括号的范围内存在。
- ❑ 循环的迭代自变量，例如，下面的循环中的 i：

```
for (int i = 0; i < 10; ++i) {
    // 循环体
}
```

只能在循环体内起作用。

变量的初始化

所有变量都必须在声明的同时进行初始化。例如：

```
int    i, numParticles = 1500;
float  force, g = -9.8;
bool   falling = true;
double pi = 3.1415926535897932384626LF;
```

整型字面量常数可以表示为八进制、十进制或者十六进制的值。我们也可以在数字之前加上一个符号来表示负数，或者在末尾添加“u”或者“U”来表示一个无符号的整数。

浮点字面量必须包含一个小数点，除非我们用科学计数法来表示它，例如 3E-7（不过，很多时候我们也可以将一个整数隐式地转换为一个浮点数）。此外，浮点数也可以选择末尾添加一个“f”或者“F”后缀，这一点与 C 语言中浮点数的表示法相同。如果要表达一个 double 精度的浮点数，必须在末尾添加后缀“lf”或者“LF”。

布尔变量可以是 true 或者 false，对它进行初始化的时候，可以直接指定这两个值之一，也可以对一个布尔表达式进行解析并且将结果赋予变量。

构造函数

正如前面提到的，GLSL 比 C++ 更注重类型安全，因此它支持的数值隐式转换更少一

些。例如，

```
int f = false;
```

这样的写法会返回一个编译错误，因为布尔值不能赋予整型变量。可以进行隐式转换的类型如表 2-2 所示。

上面的类型转换适用于这些类型的标量、向量以及矩阵。转换不会改变向量或者矩阵本身的形式，也不会改变它们的组成元素数量。类型转换不能应用于数组或者结构体之上。

表 2-2 GLSL 中的隐式类型转换

所需的类型	可以从这些类型隐式转换
uint	int
float	int、uint
double	int、uint、float

所有其他的数值转换都需要提供显式的转换构造函数。这里构造函数的意义与 C++ 等语言类似，它是一个名字与类型名称相同的函数，返回值就是对应类型的值。例如，

```
float f = 10.0;
int ten = int(f);
```

这里用到了一个 int 转换构造函数来完成转换。此外，其他一些类型也有转换构造函数，包括 float、double、uint、bool，以及这些类型的向量和矩阵。每种构造函数都可以传入多个其他类型的值并且进行显式转换。这些函数也体现了 GLSL 的另一个特性：函数重载，即每个函数都可以接受不同类型的输入，但是它们都使用了同一个基函数名称。我们稍后将对函数进行更多的讲解。

聚合类型

GLSL 的基本类型可以进行合并，从而与核心 OpenGL 的数据类型相匹配，以及简化计算过程的操作。

首先，GLSL 支持 2 个、3 个以及 4 个分量的向量，每个分量都可以使用 bool、int、uint、float 和 double 这些基本类型。此外，GLSL 也支持 float 和 double 类型的矩阵。表 2-3 给出了所有可用的向量和矩阵类型。

表 2-3 GLSL 的向量与矩阵类型

基本类型	2D 向量	3D 向量	4D 向量	矩阵类型		
float	vec2	vec3	vec4	mat2	mat3	mat4
				mat2 × 2	mat2 × 3	mat2 × 4
				mat3 × 2	mat3 × 3	mat3 × 4
				mat4 × 2	mat4 × 3	mat4 × 4
double	dvec2	dvec3	dvec4	dmat2	dmat3	dmat4
				dmat2 × 2	dmat2 × 3	dmat2 × 4
				dmat3 × 2	dmat3 × 3	dmat3 × 4
				dmat4 × 2	dmat4 × 3	dmat4 × 4
int	ivec2	ivec3	ivec4	—		
uint	uvec2	uvec3	uvec4	—		
bool	bvec2	bvec3	bvec4	—		

矩阵类型需要给出两个维度的信息，例如 `mat4x3`，其中第一个值表示列数，第二个值表示行数。

使用这些类型声明的变量的初始化过程与它们的标量部分是类似的：

```
vec3 velocity = vec3(0.0, 2.0, 3.0);
```

类型之间也可以进行等价转换：

```
ivec3 steps = ivec3(velocity);
```

向量的构造函数还可以用来截短或者加长一个向量。如果将一个较长的向量传递给一个较短向量的构造函数，那么向量将被自动取短到对应的长度。

```
vec4 color;
vec3 RGB = vec3(color); // 现在 RGB 只有前三个分量了
```

类似地，也可以使用同样的方式来加长一个向量。并且可以直接将标量值传递给向量，例如：

```
vec3 white = vec3(1.0); // white = (1.0, 1.0, 1.0)
vec4 translucent = vec4(white, 0.5);
```

矩阵的构建方式与此相同，并且可以将它初始化为一个对角矩阵或者完全填充的矩阵。对于对角矩阵，只需要向构造函数传递一个值，矩阵的对角线元素就设置为这个值，其他元素全部设置为 0，例如：

$$m = \text{mat3}(4.0) = \begin{pmatrix} 4.0 & 0.0 & 0.0 \\ 0.0 & 4.0 & 0.0 \\ 0.0 & 0.0 & 4.0 \end{pmatrix}$$

矩阵也可以通过在构造函数中指定每一个元素的值来构建。传入元素可以是标量和向量的集合，只要给定足够数量的数据即可，每一列的设置方式也遵循这样的原则。此外，矩阵的指定需要遵循列主序的原则，也就是说，传入的数据将首先填充列，然后填充行（这一点与 C 语言中 2 维数组的初始化是相反的）。

例如，可以通过下面几种形式之一来初始化一个 3×3 的矩阵：

```
mat3 M = mat3(1.0, 2.0, 3.0,
               4.0, 5.0, 6.0,
               7.0, 8.0, 9.0);

vec3 column1 = vec3(1.0, 2.0, 3.0);
vec3 column2 = vec3(4.0, 5.0, 6.0);
vec3 column3 = vec3(7.0, 8.0, 9.0);

mat3 M = mat3(column1, column2, column3);
```

甚至是

```
vec2 column1 = vec2(1.0, 2.0);
vec2 column2 = vec2(4.0, 5.0);
vec2 column3 = vec2(7.0, 8.0);

mat3 M = mat3(column1, 3.0,
               column2, 6.0,
               column3, 9.0);
```

得到的结果都是一样的，

$$\begin{pmatrix} 1.0 & 4.0 & 7.0 \\ 2.0 & 5.0 & 8.0 \\ 3.0 & 6.0 & 9.0 \end{pmatrix}$$

访问向量和矩阵中的元素

向量与矩阵中的元素是可以单独访问和设置的。向量支持两种类型的元素访问方式：使用分量的名称，或者数组访问的形式。矩阵可以以 2 维数组的形式进行访问。

向量中的各个分量是可以通过名称进行访问的，例如：

```
float red = color.r;
float v_y = velocity.y;
```

或者通过一个从 0 开始的索引。下面的代码与上面的结果是完全等价的：

```
float red = color[0];
float v_y = velocity[1];
```

事实上，正如表 2-4 所示，分量的名称总共有三种形式的集合，它们实现的工作是一样的。不同的名称集合只是为了在使用时便于区分不同的操作。

表 2-4 向量分量的访问符

分量访问符	符号描述
(x, y, z, w)	与位置相关的分量
(r, g, b, a)	与颜色相关的分量
(s, t, p, q)	与纹理坐标相关的分量

这种分量访问符的一个常见应用叫做 swizzle，对于颜色的处理，比如颜色空间的转换时可能会用到它。例如，可以通过下面的代码，基于输入颜色的红色分量来设置一个亮度值：

```
vec3 luminance = color.rrr;
```

类似地，如果需要改变向量中分量各自的位置，可以这样做：

```
color = color.abgr; // 反转 color 的每个分量
```

唯一的限制是，在一条语句的一个变量中，只能使用一种类型的访问符。也就是说，下面的代码是错误的：

```
vec4 color = otherColor.rgz; // 错误：“z”来自不同的访问符集合
```

此外，如果我们访问的元素超出了变量类型的范围，也会引发编译时错误。例如：

```
vec2 pos;
float zPos = pos.z; // 错误：2D 向量不存在“z”分量
```

矩阵元素的访问可以使用数组标记方式。或者从矩阵中直接得到一个标量值，或者一组元素：

```
mat4 m = mat4(2.0);
vec4 zVec = m[2]; // 获取矩阵的第 2 列
```

```
float yScale = m[1][1]; // 也可以使用 m[1].y
```

结构体

你也可以从逻辑上将不同类型的数据组合到一个结构体当中。结构体可以简化多组数据传入函数的过程。如果定义了一个结构体，那么它会自动创建一个新类型，并且隐式定义一个构造函数，将各种类型的结构体元素作为输入参数。

```
struct Particle {
    float lifetime;
    vec3 position;
    vec3 velocity;
};
```

```
Particle p = Particle(10.0, pos, vel); // pos, vel 均为 vec3s 类型
```

与 C 语言中的用法类似，如果我们需要引用结构体的某个元素，可以直接使用“点”(.) 符号。

数组

GLSL 还支持任意类型的数组，包括结构体数组。和 C 语言相同，数组的索引可以通过方括号来完成 ([])。一个大小为 n 的数组的元素范围是 0 到 $n-1$ 。但是与 C 语言中不同的是，负数形式的数组索引，或者超出范围的索引值都是不允许的。GLSL 4.3 中，数组的组成元素也可以是另一个数组，因此可以处理多维度的数据。不过，GLSL 4.2 和更早的版本不允许建立数组类型的数组（因此无法创建多维度的数组）。

数组可以定义为有大小的，或者没有大小的。我们可以使用没有大小的数组作为一个数组变量的前置声明，然后重新用一个合适的大小来声明它。数组的声明需要用到方括号的形式，例如：

```
float coeff[3]; // 有 3 个 float 元素的数组
float[3] coeff; // 与上面相同
int indices[]; // 未定义维数，稍后可以重新声明它的维数
```

数组属于 GLSL 中的第一等（first-class）类型，也就是说它有构造函数，并且可以用作函数的参数和返回类型。如果我们要静态初始化一个数组的值，那么可以按照下面的形式来使用构造函数：

```
float coeff[3] = float[3](2.38, 3.14, 42.0);
```

这里构造函数的维数值可以不填。

此外，GLSL 的数组与 Java 类似，它有一个隐式的方法可以返回元素的个数：即取长度的方法 `length()`。如果我们需要操作一个数组中所有的值，可以根据下面的例子来使用 `length()` 方法：

```
for (int i = 0; i < coeff.length(); ++i) {
    coeff[i] *= 2.0;
}
```


向量和矩阵类型也可以使用 `length()` 方法。向量的长度也就是它包含的分量的个数，矩阵的长度是它包含的列的个数。事实上，当我们使用数组的形式来索引向量和矩阵的值的时候（例如，`m[2]` 是矩阵 `m` 的第三列），这个方法返回的就是我们需要的数据。

```
mat3x4 m;
int c = m.length(); // m 包含的列数为 3
int r = m[0].length(); // 第 0 个列向量中分量的个数为 4
```

因为长度值在编译时就是已知的，所以 `length()` 方法会返回一个编译时常量，我们可以在需要使用常量的场合直接使用它，例如：

```
mat4 m;
float diagonal[m.length()]; // 设置数组的大小与矩阵大小相等
float x[gl_in.length()]; // 设置数组的大小与几何着色器的输入顶点数相等
```

对于所有向量和矩阵，以及大部分的数组来说，`length()` 都是一个编译时就已知的常量。但是对于某些数组来说，`length()` 的值在链接之前可能都是未知的。如果使用链接器来减少同一阶段中多个着色器的大小，那么可能发生这种情况。对于着色器中保存的缓存对象（使用 `buffer` 来进行声明，后文将会介绍），`length()` 的值直到渲染时才可能得到。如果我们需要 `length()` 返回一个编译时常量，那么我们需要保证着色器中的数组大小在使用它的 `length()` 方法之前就已经确定了。

多维数组相当于从数组中再创建数组，它的语法与 C 语言当中类似：

```
float coeff[3][5]; // 一个大小为 3 的数组，其中包含了大小为 5 的多个数组
coeff[2][1] *= 2.0; // 内层索引设置为 1，外层设置为 2
coeff.length(); // 这个方法会返回常量 3
coeff[2]; // 这是一个大小为 5 的 1 维数组
coeff[2].length(); // 这个方法会返回常量 5
```

多维数组可以使用任何类型或者形式来构成。如果需要与应用程序共享，那么最内层（最右侧）维度的数据在内存布局中的变化是最快的。

2.3.2 存储限制符

数据类型也可以通过一些修饰符来改变自己的行为。GLSL 中一共定义了 4 种全局范围内的修饰符，如表 2-5 所示。

表 2-5 GLSL 的类型修饰符

类型修饰符	描述
const	将一个变量定义为只读形式。如果它初始化时用的是一个编译时常量，那么它本身也会成为编译时常量
in	设置这个变量为着色器阶段的输入变量
out	设置这个变量为着色器阶段的输出变量
uniform	设置这个变量为用户应用程序传递给着色器的数据，它对于给定的图元而言是一个常量

(续)

类型修饰符	描述
buffer	设置应用程序共享的一块可读写的内存。这块内存也作为着色器中的存储缓存 (storage buffer) 使用
shared	设置变量是本地工作组 (local work group) 中共享的。它只能用于计算着色器中

const 存储限制符

与 C 语言中相同, const 类型的修饰符设置变量为只读类型。例如, 下面的语句

```
const float Pi = 3.141529;
```

会设置一个变量 Pi 为圆周率 π 的近似值。对变量的声明添加了 const 修饰符之后, 如果再向这个变量写入, 那么将会产生一个错误, 因此这种变量必须在声明的时候就进行初始化。

in 存储限制符

in 修饰符用于定义着色器阶段的输入变量。这类输入变量可以是顶点属性 (对于顶点着色器), 或者前一个着色器阶段的输出变量。

片元着色器也可以使用一些其他的关键词来限定自己的输入变量, 这会在第 4 章中进行讲解。

out 存储限制符

out 修饰符用于定义一个着色器阶段的输出变量——例如, 顶点着色器中输出变换后的齐次坐标, 或者片元着色器中输出的最终片元颜色。

uniform 存储限制符

在着色器运行之前, uniform 修饰符可以指定一个在应用程序中设置好的变量, 它不会在图元处理的过程中发生变化。uniform 变量在所有可用的着色阶段之间都是共享的, 它必须定义为全局变量。任何类型的变量 (包括结构体和数组) 都可以设置为 uniform 变量。着色器无法写入到 uniform 变量, 也无法改变它的值。

举例来说, 我们可能需要设置一个给图元着色的颜色值。此时可以声明一个 uniform 变量, 将颜色值信息传递到着色器当中。而着色器中会进行如下声明:

```
uniform vec4 BaseColor;
```

在着色器中, 可以根据名字 BaseColor 来引用这个变量, 但是如果需要在用户应用程序中设置它的值, 还需要多做一些工作。GLSL 编译器会在链接着色器程序时创建一个 uniform 变量列表。如果需要设置应用程序中 BaseColor 的值, 我们需要首先获得 BaseColor 在列表中的索引, 这一步可以通过 glGetUniformLocation() 函数来完成。

```
GLint glGetUniformLocation(GLuint program, const char* name);
```

返回着色器程序中 uniform 变量 name 对应的索引值。name 是一个以 NULL 结尾的字符串，不存在空格。如果 name 与启用的着色器程序中的所有 uniform 变量都不相符，或者 name 是一个内部保留的着色器变量名称（例如，以 gl_ 开头的变量），那么返回值为 -1。

name 可以是单一的变量名称、数组中的一个元素（此时 name 主要包含方括号以及对应的索引数字），或者结构体的域变量（设置 name 时，需要在结构体变量名称之后添加 “.” 符号，再添加域变量名称，并与着色器程序中的写法一致）。对于 uniform 变量数组，也可以只通过指定数组的名称来获取数组中的第一个元素（例如，直接用 “arrayName”），或者也可以通过指定索引值来获取数组的第一个元素（例如，写作 “arrayName[0]”）。

除非我们重新链接着色器程序（参见 glLinkProgram()），否则这里的返回值不会发生变化。

当得到 uniform 变量的对应索引值之后，我们就可以通过 glUniform*() 或者 glUniformMatrix*() 系列函数来设置 uniform 变量的值了。

例 2.2 是一个获取 uniform 变量的索引并且设置具体值的示例。

例 2.2 获取 uniform 变量的索引并且设置具体值

```
GLint    timeLoc; /* 着色器中的 uniform 变量 time 的索引 */
GLfloat  timeValue; /* 程序运行时间 */

timeLoc = glGetUniformLocation(program, "time");
glUniform1f(timeLoc, timeValue);
```

```
void glUniform{1234}{fdi ui}(GLint location, TYPE value);
void glUniform{1234}{fdi ui}v(GLint location, GLsizei count, const TYPE* values);
void glUniformMatrix{234}{fd}v(GLint location, GLsizei count, GLboolean transpose,
const GLfloat* values);
void glUniformMatrix{2x3,2x4,3x2,3x4,4x2,4x3}{fd}v(GLint location, GLsizei count,
GLboolean transpose, const GLfloat* values);
```

设置与 location 索引位置对应的 uniform 变量的值。其中向量形式的函数会载入 count 个数据的集合（根据 glUniform*() 的调用方式，读入 1 ~ 4 个值），并写入 location 位置的 uniform 变量。如果 location 是数组的起始索引值，那么数组之后的连续 count 个元素都会被载入。

GLfloat 形式的函数（后缀中有 f）可以用来载入单精度类型的浮点数、float 类型的向量、float 类型的数组、或者 float 类型的向量数组。与之类似，GLdouble 形式的函数（后缀中有 d）可以用来载入双精度类型的标量、向量和数组。GLfloat 形式的函数也可以载入布尔数据。

GLint 形式的函数（后缀中有 i）可以用来更新单个有符号整型、有符号整型向量、有符号整型数组，或者有符号整型向量数组。此外，可以用这种形式载入独立纹理采样器或者纹理数组、布尔类型的标量、向量和数组。与之类似，GLuint 形式的函数（后缀中有 ui）也可以用来载入无符号整型标量、向量和数组。

对于 glUniformMatrix{234}*() 系列函数来说，可以从 values 中读入 2×2 、 3×3 或者 4×4 个值来构成矩阵。

对于 glUniformMatrix{2x3,2x4,3x2,3x4,4x2,4x3}*() 系列函数来说，可以从 values 中读入对应矩阵维度的数值并构成矩阵。如果 transpose 设置为 GL_TRUE，那么 values 中的数据是以行主序的顺序读入的（与 C 语言中的数组类似），如果是 GL_FALSE，那么 values 中的数据是以列主序的顺序读入的。

buffer 存储限制符

如果需要在应用程序中共享一大块缓存给着色器，那么最好的方法是使用 buffer 变量。它与 uniform 变量非常类似，不过也可以用着色器对它的内容进行修改。通常来说，需要在一个 buffer 块中使用 buffer 变量，本章后面将对“块”的概念进行介绍。

buffer 修饰符指定随后的块作为着色器与应用程序共享的一块内存缓存。这块缓存对于着色器来说是可读的也是可写的。缓存的大小可以在着色器编译和程序链接完成后设置。

shared 存储限制符

shared 修饰符只能用于计算着色器当中，它可以建立本地工作组内共享的内存。第 12 章会详细介绍它。

2.3.3 语句

着色器的真正工作是计算数值以及完成一些决策工作。与 C++ 中的形式类似，GLSL 也提供了大量的操作符，来实现各种数值计算所需的算术操作，以及一系列控制着色器运行的逻辑操作。

算术操作符

任何一种语言的教程如果缺少有关操作符以及优先级的介绍（参见表 2-6），那么这个教程是不完整的。表 2-6 中操作符的优先级采取降序排列。总体上来说，操作符对应的类型必须是相同的，并且对于向量和矩阵而言，操作符的操作对象也必须是同 1 维度的。在表 2-6 中注明的整型包括 int 和 uint，以及对应的向量；浮点数类型包括 float 和 double，以及对应的向量与矩阵；算术类型包括所有的整型和浮点数类型，以及所有相关的结构体和数组。

表 2-6 GLSL 操作符与优先级

优先级	操作符	可用类型	描述
1	()	—	成组操作
2	[] f() (句点) ++ --	数组、矩阵、向量 函数 结构体 算术类型	数组的下标 函数调用与构造函数 访问结构体的域变量或者方法 后置递增 / 递减
3	++ -- + - ~ !	算术类型 算术类型 整型 布尔型	前置递增 / 递减 一元正 / 负数 一元按位 “非”(not) 一元逻辑 “非”(not)
4	* / %	算术类型	乘法运算
5	+ -	算术类型	相加运算
6	<< >>	整型	按位操作
7	< > <= >=	算术类型	关系比较操作
8	== !=	任意	相等操作
9	&	整型	按位 “与”(and)
10	^	整型	按位 “异或”(xor)
11		整型	按位 “或”(or)
12	&&	布尔型	逻辑 “与”(and)
13	^^	布尔型	逻辑 “异或”(xor)
14		布尔型	逻辑 “或”(or)
15	a ? b : c	布尔型 ? 任意 : 任意	三元选择操作符 (相当于内部进行了条件判断, 如果 a 成立则执行 b, 否则执行 c)
16	= += -= *= /= %= <<= >>= &= ^= =	任意 算术类型 算术类型 整型 整型	赋值 算术赋值
17	, (逗号)	任意	操作符序列

操作符重载

GLSL 中的大部分操作符都是经过重载的, 也就是说它们可以用于多种类型的数据操作。特别是, 矩阵和向量的算术操作符 (包括前置和后置的递增 / 递减符号 “++” 和 “--”) 在 GLSL 中都是经过严格定义的。例如, 如果我们需要进行向量和矩阵之间的乘法 (注意, 操作数的顺序非常重要, 从数学上来说, 矩阵乘法是不遵循交换律的), 可以使用下面的操作:

```
vec3 v;  
mat3 m;  
vec3 result = v * m;
```

基本的限制条件是要求矩阵和向量的维度必须是匹配的。此外, 也可以对向量或者矩

阵执行标量乘法，以得到希望的结果。一个必须要提及的例外是，两个向量相乘得到的是一个逐分量相乘的新向量，但是两个矩阵相乘得到的是通常矩阵相乘的结果。

```
vec2 a, b, c;
mat2 m, u, v;
c = a * b; // c = (a.x*b.x, a.y*b.y)
m = u * v; // m = (u00*v00+u01*v10    u00*v01+u01*v11
                //      u01*v00+u11*v10    u10*v01+u11*v11)
```

我们还可以通过函数调用的方式实现常见的一些向量操作（例如，点乘和叉乘），以及各种逐分量执行的向量和矩阵操作。

流控制

GLSL 的逻辑控制方式用的也是流行的 if-else 和 switch 语句。与 C 语言中的方式相同，else 的分支是可选的，并且有多行语句时必须用到语句块：

```
if (truth) {
    // 条件为 true 的分支
}
else {
    // 条件为 false 的分支
}
```

switch 语句的使用（从 GLSL 1.30 开始）与 C 语言中也是类似的，可以采用下面的方式：

```
switch (int_value) {
    case n:
        // 语句
        break;
    case m:
        // 语句
        break;
    default:
        // 语句
        break;
}
```

GLSL 的 switch 语句也支持“fall-through”形式：一个 case 语句如果没有使用 break 结尾，那么会继续执行下一个 case 的内容。每个 case 都需要执行一些语句，直到整个 switch 块结束（在右花括号之前）。此外，与 C++ 当中不同的是，GLSL 不允许在第一个 case 之前添加语句。如果所有的 case 条件都不符合，那么将会找到并执行 default 分支中的内容。

循环语句

GLSL 支持 C 语言形式的 for、while 和 do ... while 循环。其中 for 循环可以在循环初始条件中声明循环迭代变量。此时迭代变量的作用域只限于循环体内。

```
for (int i = 0; i < 10; ++i) {
    ...
}

while (n < 10) {
```



```
    ...
}
do {
    ...
} while (n < 10);
```

流控制语句

除了条件和循环之外，GLSL 还支持一些别的控制语句。表 2-7 所示为其他可用的流控制语句。

表 2-7 GLSL 的流控制语句

语句	描述
break	终止循环体的运行，并且继续执行循环体外的内容
continue	终止循环体内当前迭代过程的执行，跳转到代码块开始部分并继续执行下一次迭代的内容
return [结果]	从当前子例程返回，可以带有一个函数返回值（返回值必须与函数声明的返回类型相符）
discard	丢弃当前的片元，终止着色器的执行。discard 语句只能用于片元着色器中

discard 语句只适用于片元着色器中。片元着色器的运行会在 discard 语句的位置上立即终止，不过这也取决于具体的硬件实现。

函数

我们可以使用函数调用来取代可能反复执行的通用代码。这样当然可以减少代码的总量，并且减少发生错误的机会。GLSL 支持用户自定义函数，同时它也定义了一些内置函数，具体列表可以参见附录 C。用户自定义函数可以在单个着色器对象中定义，然后在多个着色器程序中复用。

声明

函数声明语法与 C 语言非常类似，只是变量名需要添加访问修饰符：

```
returnType functionName([accessModifier] type1 variable1,
                        [accessModifier] type2 variable2,
                        ...)
{
    // 函数体
    return returnValue; // 如果 returnType 为 void，则不需要 return 语句
}
```

函数名称可以是任何字符、数字和下划线字符的组合，但是不能使用数字、连续下划线或者 gl_ 作为函数的开始。

返回值可以是任何内置的 GLSL 类型，或者用户定义的结构体和数组类型。返回值为数组时，必须显式地指定其大小。如果一个函数的返回值类型是 void，那么它可以没有返回值。

函数的参数也可以是任何类型，包括数组（但是也必须设置数组的大小）。

在使用一个函数之前，必须声明它的原型或者直接给出函数体。GLSL 的编译器与 C++ 一致，必须在使用函数之前找到函数的声明，否则会产生错误。如果函数的定义和使用不在同一个着色器对象当中，那么必须声明一个函数原型。函数原型只是给出了函数的形式，但是并没有给出具体的实现内容。下面是一个简单的例子：

```
float HornerEvalPolynomial(float coeff[10], float x);
```

参数限制符

尽管 GLSL 中的函数可以在运行后修改和返回数据，但是它与“C”或者 C++ 不同，并没有指针或者引用的概念。不过与之对应，此时函数的参数可以指定一个参数限制符，来表明它是否需要在函数运行时将数据拷贝到函数中，或者从函数中返回修改的数据。表 2-8 给出了 GLSL 中可用的参数限制符。

表 2-8 GLSL 函数参数的访问修饰符

访问修饰符	描述
in	将数据拷贝到函数中（如果没有指定修饰符，默认这种形式）
const in	将只读数据拷贝到函数中
out	从函数中获取数值（因此输入函数的值是未定义的）
inout	将数据拷贝到函数中，并且返回函数中修改的数据

关键字 in 是可选的。如果一个变量没有包含任何访问修饰器，那么参数的声明会默认设置为使用 in 修饰符。但是，如果变量的值需要从函数中拷贝出来，那么我们就必须设置它为 out（只能写出的变量）或者 inout（可以读入也可以写出的变量）修饰符。如果我们写出到一个没有设置上述修饰符的变量上，那么会产生编译时错误。

此外，如果需要在编译时验证函数是否修改了某个输入变量，可以添加一个 const in 修饰符来阻止函数对变量进行写操作。如果不这么做，那么在函数中写入一个 in 类型的变量，相当于对变量的局部拷贝进行了修改，因此只在函数自身范围内产生作用。

2.3.4 计算的不变性

GLSL 无法保证在不同的着色器中，两个完全相同的计算式会得到完全一样的结果。这一情形与 CPU 端应用程序进行计算时的的问题相同，即不同的优化方式可能会导致结果非常细微的差异。这些细微的差异对于多通道的算法会产生问题，因为各个着色器阶段可能需要计算得到完全一致的结果。GLSL 有两种方法来确保着色器之间的计算不变性，即 invariant 或者 precise 关键字。

这两种方法都需要在图形设备上完成计算过程，来确保同一表达式的结果可以保证重复性（不变性）。但是，对于宿主计算机和图形硬件各自的计算，这两种方法都无法保证结果是完全一致的。着色器编译时的常量表达式是由编译器的宿主计算机计算的，因此我们无法保证宿主计算机计算的结果与图形硬件计算的结果完全相同。例如：

```

uniform float ten;           // 假设应用程序设置这个值为 10.0
const float f = sin(10.0);   // 宿主机的编译器负责计算
float g = sin(ten);          // 图形硬件负责计算
void main()
{
    if (f == g)              // f 和 g 不一定相等
        ;
}

```

在这个例子当中，无论对任何一个变量设置 `invariant` 或者 `precise` 限制符，结果都不会有任何改变，因为它们都只能影响到图形设备中的计算结果。

invariant 限制符

`invariant` 限制符可以设置任何着色器的输出变量。它可以确保如果两个着色器的输出变量使用了同样的表达式，并且表达式中的变量也是相同值，那么计算产生的结果也是相同的。

可以将一个内置的输出变量声明为 `invariant`，也可以声明一个用户自定义的变量为 `invariant`。例如：

```

invariant gl_Position;
invariant centroid out vec3 Color;

```

你可能还记得，输出变量的作用是将一个着色器的数据从一个阶段传递到下一个。可以在着色器中用到某个变量或者内置变量之前的任何位置，对该变量设置关键字 `invariant`。标准的做法是只使用 `invariant` 来声明这个变量，如上文中对 `gl_Position` 的设置。

在调试过程中，可能需要将着色器中的所有可变量都设置为 `invariance`。可以通过顶点着色器的预编译命令 `pragma` 来完成这项工作。

```

#pragma STDGL invariant(all)

```

全局都设置为 `invariance` 可以帮助我们解决调试问题；但是，这样对于着色器的性能也会有所影响。而为了保证不变性，通常也会导致 GLSL 编译器所执行的一些优化工作被迫停止。

precise 限制符

`precise` 限制符可以设置任何计算中的变量或者函数的返回值。它的名字有点望文生义，它的用途并不是增加数据精度，而是增加计算的可复用性。我们通常在细分着色器中用它来避免造成几何体形状的裂缝。第 9 章将大致介绍细分着色的内容，并且在其中对 `precise` 限制符进行更进一步的讲解和示例分析。

总体上说，如果必须保证某个表达式产生的结果是一致的，即使表达式中的数据发生了变化（但是在数学上并不影响结果）也是如此，那么此时我们应该使用 `precise` 而非 `invariant`。举例来说，下面的表达式中，即使 `a` 和 `b` 的值发生了交换，得到的结果也是不变的。此外即使 `c` 和 `d` 的值发生了交换，或者 `a` 和 `c` 同时与 `b` 和 `d` 发生了交换等，都应该得到同样的计算结果。

```
Location = a * b + c * d;
```

`precise` 限制符可以设置内置变量、用户变量，或者函数的返回值。

```
precise gl_Position;
precise out vec3 Location;
precise vec3 subdivide(vec3 P1, vec3 P2) { ... }
```

在着色器中，关键字 `precise` 可以在使用某个变量之前的任何位置上设置这个变量，并且可以修改之前已经声明过的变量。

编译器使用 `precise` 的一个实际影响是，类似上面的表达式不能再使用两种不同的乘法命令来同时参与计算。例如，第一次相乘使用普通的乘法，而第二次相乘使用混合乘加运算（fused multiply-and-add, `fma`）。这是因为这两个命令对于同一组值的计算结果可能会存在微小的差异。而这种差异是 `precise` 所不允许的，因此编译器会直接阻止你在代码中这样做。由于混合乘加运算对于性能的提升非常重要，因此不可能完全禁止用户使用它们。所以 GLSL 提供了一个内置的函数 `fma()`，让用户可以直接使用这个函数代替原先的操作。

```
precise out float result;
...
float f = c * d;
float result = fma(a, b, f);
```

当然，如果不需要考虑交换 `a` 和 `c` 的值，那么没有必要使用这种写法，因为那个时候也没有必要使用 `precise` 了。

2.3.5 着色器的预处理器

编译一个 GLSL 着色器的第一步是解析预处理器。这一点与 C 语言中的预处理器类似，并且 GLSL 同样提供了一系列命令来有条件地生成编译代码块，或者定义数值。不过，与 C 语言的预处理器不同的是，GLSL 中没有文件包含的命令（`#include`）。

预处理器命令

表 2-9 给出了 GLSL 预处理器所支持的预处理器命令以及对应的函数。

表 2-9 GLSL 的预处理器命令

预处理器命令	描述
<code>#define</code> <code>#undef</code>	控制常量与宏的定义，与 C 语言的预处理器命令类似
<code>#if</code> <code>#ifdef</code> <code>#ifndef</code> <code>#else</code> <code>#elif</code> <code>#endif</code>	代码的条件编译，与 C 语言的预处理器命令和 <code>defined</code> 操作符均类似。 条件表达式中只可以使用整数表达式或者 <code>#define</code> 定义的值
<code>#error text</code>	强制编译器将 <code>text</code> 文字内容（直到第一个换行符为止）插入到着色器的信息日志当中

(续)

预处理器命令	描述
#pragma options	控制编译器的特定选项
#extension options	设置编译器支持特定 GLSL 扩展功能
#version number	设置当前使用的 GLSL 版本名称
#line options	设置诊断行号

宏定义

GLSL 预处理器可以采取与 C 语言预处理器类似的宏定义方式，不过它不支持字符串替换以及预编译连接符。宏可以定义为单一的值，例如：

```
#define NUM_ELEMENTS 10
```

或者带有参数，例如：

```
#define LPos(n) gl_LightSource[(n)].position
```

此外，GLSL 还提供了一些预先定义好的宏，用于记录一些诊断信息（可以通过 #error 命令输出），如表 2-10 所示。

表 2-10 GLSL 预处理器中的预定义宏

__LINE__	行号，默认为已经处理的所有换行符的个数加一，也可以通过 #line 命令修改
__FILE__	当前处理的源字符串编号
__VERSION__	OpenGL 着色语言版本的整数表示形式

此外，也可以通过 #undef 命令来取消之前定义过的宏（GLSL 内置的宏除外）。例如

```
#undef LPos
```

预处理器中的条件分支

GLSL 的预处理器与 C 语言的预处理器相同，都可以根据宏定义以及整型常数的条件来判断进入不同的分支，包含不同的代码段。

宏定义可以通过两种方式来参与条件表达式：第一种方式是使用 #ifdef 命令：

```
#ifdef NUM_ELEMENTS
...
#endif
```

或者在 #if 和 #elif 命令中使用操作符来进行判断：

```
#if defined(NUM_ELEMENTS) && NUM_ELEMENTS > 3
...
#elif NUM_ELEMENTS < 7
...
#endif
```

2.3.6 编译器的控制

`#pragma` 命令可以向编译器传递附加信息，并在着色器代码编译时设置一些额外属性。

编译器优化选项

优化选项用于启用或者禁用着色器的优化，它会直接影响该命令所在的着色器源代码。可以通过下面的命令分别启用或者禁用优化选项：

```
#pragma optimize(on)
```

或者

```
#pragma optimize(off)
```

这类选项必须在函数定义的代码块之外设置。一般默认所有着色器都开启了优化选项。

编译器调试选项

调试选项可以启用或者禁用着色器的额外诊断信息输出。可以通过下面的命令分别启用或者禁用调试选项：

```
#pragma debug(on)
```

或者

```
#pragma debug(off)
```

与优化选项类似，这些选项只在函数定义的代码块之外设置，而默认情况下，所有着色器都会禁用调试选项。

2.3.7 全局着色器编译选项

另一个可用的 `#pragma` 命令选项就是 `STDGL`。这个选项目前用于启用所有输出变量值的不变性检查。

着色器的扩展功能处理

GLSL 与 OpenGL 类似，都可以通过扩展的方式来增加功能。设备生产商也可以在自己的 OpenGL 实现中加入特殊的扩展，因此很有必要对着色器中可能用到的扩展功能进行编译级别的控制。

GLSL 预处理器提供了 `#extension` 命令，用于提示着色器的编译器在编译时如何处理可用的扩展内容。对于任何扩展，或者全部扩展，我们都可以在编译器编译过程中设置它们的处理方式。

```
#extension extension_name : <directive>
```

这里的 `extension_name` 与调用 `glGetString(GL_EXTENSIONS)` 时获取的扩展功能名称是一致的，或者也可以使用

`#extension all : <directive>`

从而直接影响所有扩展的行为。

`<directive>` 可用的选项如表 2-11 所示。

表 2-11 GLSL 扩展命令修饰符

命令	描述
require	如果无法支持给定的扩展功能，或者被设置为 all，则提示错误
enable	如果无法支持给定的扩展功能，则给出警告；如果设置为 all，则提示错误
warn	如果无法支持给定的扩展功能，或者在编译过程中使用了任何扩展，则给出警告
disable	禁止支持给定的扩展（即强制编译器不提供对扩展功能的支持），或者如果设置为 all 则禁止所有的扩展支持，之后当代码中涉及这个扩展使用时，提示警告或者错误

2.4 数据块接口

着色器与应用程序之间，或者着色器各阶段之间共享的变量可以组织为变量块的形式，并且有的时候必须采用这种形式。uniform 变量可以使用 uniform 块，输入和输出变量可以使用 in 和 out 块，着色器的存储缓存可以使用 buffer 块。

它们的形式都是类似的。首先了解一下 uniform 块的写法。

```
uniform b {           // 限定符可以为 uniform、in、out 或者 buffer
    vec4 v1;           // 块中的变量列表
    bool v2;           // ...
};                     // 访问块成员时使用 v1、v2
```

或者

```
uniform b {           // 限定符可以为 uniform、in、out 或者 buffer
    vec4 v1;           // 块中的变量列表
    bool v2;           // ...
} name;               // 访问块成员时使用 name.v1、name.v2
```

各种类型的块接口的详细介绍如下文所示。综合来说，块（block）开始部分的名称（上面的代码中为 b）对应于外部访问时的接口名称，而结尾部分的名称（上面的代码中为 name）用于在着色器代码中访问具体成员变量。

2.4.1 uniform 块

如果着色器程序变得比较复杂，那么其中用到的 uniform 变量的数量也会上升。通常会在多个着色器程序中用到同一个 uniform 变量。由于 uniform 变量的位置是着色器链接的时候产生的（也就是调用 `glLinkProgram()` 的时候），因此它在应用程序中获得的索引可能会有变化，即使我们给 uniform 变量设置的值可能是完全相同的。而 uniform 缓存对象（Uniform buffer object）就是一种优化 uniform 变量访问，以及在不同的着色器程序之间共

享 uniform 数据的方法。

正如你所知道的，uniform 变量是同时存在于用户应用程序和着色器当中的，因此需要同时修改着色器的内容并调用 OpenGL 函数来设置 uniform 缓存对象。

2.4.2 指定着色器中的 uniform 块

访问一组 uniform 变量的方法是使用诸如 `glMapBuffer()` 的 OpenGL 函数（参见第3章），但是我们需要在着色器中对它们的声明方式略作修改。不再分别声明每个 uniform 变量，而是直接将它们成组，形成一个类似结构体的形式，也就是 uniform 块。一个 uniform 块需要使用关键字 `uniform` 指定。然后将块中所有需要用到的变量包含在一对花括号当中，如例 2.3 所示。

例 2.3 声明一个 uniform 块

```
uniform Matrices {
    mat4 ModelView;
    mat4 Projection;
    mat4 Color;
};
```

注意，着色器中的数据类型有两种：不透明的和透明的；其中不透明类型包括采样器、图像和原子计数器。一个 uniform 块中只可以包含透明类型的变量。此外，uniform 块必须在全局作用域内声明。

uniform 块的布局控制

在 uniform 块中可以使用不同的限制符来设置变量的布局方式。这些限制符可以用来设置单个的 uniform 块，也可以用来设置所有后继 uniform 块的排列方式（需要使用布局声明）。可用的限制符及其介绍如表 2-12 所示。

表 2-12 uniform 的布局限制符

布局限制符	描述
shared	设置 uniform 块是多个程序间共享的（这是默认的布局方式，与 shared 存储限制符不存在混淆）
packed	设置 uniform 块占用最小的内存空间，但是这样会禁止程序间共享这个块
std140	使用标准布局方式来设置 uniform 块或者着色器存储的 buffer 块，参见附录 I
std430	使用标准布局方式来设置 buffer 块，参见附录 I
row_major	使用行主序的方式来存储 uniform 块中的矩阵
column_major	使用列主序的方式来存储 uniform 块中的矩阵（这也是默认的顺序）

例如，如果需要共享一个 uniform 块，并且使用行主序的方式来存储数据，那么可以使用下面的代码来声明它：

```
layout (shared, row_major) uniform { ... };
```

多个限制符可以通过圆括号中的逗号来分隔。如果需要对所有后继的 uniform 块设置同

一种布局，那么可以使用下面的语句：

```
layout (packed, column_major) uniform;
```

这样一来，当前行之后的所有 uniform 块都会使用这种布局方式，除非再次改变全局的布局，或者对某个块的声明单独设置专属的布局方式。

访问 uniform 块中声明的 uniform 变量

虽然 uniform 块已经命名了，但是块中声明的 uniform 变量并不会受到这个命名的限制。也就是说，uniform 块的名称并不能作为 uniform 变量的父名称，因此在两个不同名的 uniform 块中声明同名变量会在编译时造成错误。然而，在访问一个 uniform 变量的时候，也不一定非要使用块的名称。

2.4.3 从应用程序中访问 uniform 块

uniform 变量是着色器与应用程序之间共享数据的桥梁，因此如果着色器中的 uniform 变量是定义在命名的 uniform 块中，那么就有必要找到不同变量的偏移值。如果获取了这些变量的具体位置，那么就可以使用数据对它们进行初始化，这一过程与处理缓存对象（使用 `glBufferData()` 等函数）是一致的。

首先假设已知应用程序的着色器中 uniform 块的名字。如果要对 uniform 块中的 uniform 变量进行初始化，那么第一步就是找到块在着色器程序中的索引位置。可以调用 `glGetUniformBlockIndex()` 函数返回对应的信息，然后在应用程序的地址空间里完成 uniform 变量的映射。

```
GLuint glGetUniformBlockIndex(GLuint program, const char * uniformBlockName);
```

返回 program 中名称为 uniformBlockName 的 uniform 块的索引值。如果 uniformBlockName 不是一个合法的 uniform 程序块，那么返回 `GL_INVALID_INDEX`。

如果要初始化 uniform 块对应的缓存对象，那么我们需要使用 `glBindBuffer()` 将缓存对象绑定到目标 `GL_UNIFORM_BUFFER` 之上，如后文中的示例所示（第 3 章将会给出更详细的解释）。

当对缓存对象进行初始化之后，我们需要判断命名的 uniform 块中的变量总共占据了多大的空间。我们可以使用函数 `glGetActiveUniformBlockiv()` 并且设置参数为 `GL_UNIFORM_BLOCK_DATA_SIZE`，这样就可以返回编译器分配的块的大小（根据 uniform 块的布局设置，编译器可能会自动排除着色器中没有用到的 uniform 变量）。`glGetActiveUniformBlockiv()` 函数还可以用来获取一个命名的 uniform 块的其他一些相关参数。

在获取 uniform 块的索引之后，我们需要将一个缓存对象与这个块相关联。最常见的方法是调用 `glBindBufferRange()`，或者如果 uniform 块是全部使用缓存来存储的，那么可以

使用 `glBindBufferBase()`。

```
void glBindBufferRange(GLenum target, GLuint index, GLuint buffer, GLintptr offset,
GLsizeiptr size);
void glBindBufferBase(GLenum target, GLuint index, GLuint buffer);
```

将缓存对象 `buffer` 与索引为 `index` 的命名 `uniform` 块关联起来。`target` 可以是 `GL_UNIFORM_BUFFER` (对于 `uniform` 块) 或者 `GL_TRANSFORM_FEEDBACK_BUFFER` (用于 `transform feedback`, 参见第5章)。`index` 是 `uniform` 块的索引。`offset` 和 `size` 分别指定了 `uniform` 缓存映射的起始索引和大小。

调用 `glBindBufferBase()` 等价于调用 `glBindBufferRange()` 并设置 `offset` 为 0, `size` 为缓存对象的大小。

在下列情况下调用这两个函数可能会产生 OpenGL 错误 `GL_INVALID_VALUE`: `size` 小于 0; `offset + size` 大于缓存大小; `offset` 或 `size` 不是 4 的倍数; `index` 小于 0 或者大于等于 `GL_MAX_UNIFORM_BUFFER_BINDINGS` 的返回值。

当建立了命名 `uniform` 块和缓存对象之间的关联之后, 只要使用缓存相关的命令即可对块内的数据进行初始化或者修改。

我们也可以直接设置某个命名 `uniform` 块和缓存对象之间的绑定关系, 也就是说, 不使用链接器内部自动绑定块对象并且查询关联结果的方式。如果多个着色器程序需要共享同一个 `uniform` 块, 那么你可能需要用到这种方法。这样可以避免对于不同的着色器程序同一个块有不同的索引号。如果需要显式地控制一个 `uniform` 块的绑定方式, 可以在调用 `glLinkProgram()` 之前调用 `glUniformBlockBinding()` 函数。

```
GLint glUniformBlockBinding(GLuint program, GLuint uniformBlockIndex, GLuint
uniformBlockBinding);
```

显式地将块 `uniformBlockIndex` 绑定到 `uniformBlockBinding`。

在一个命名的 `uniform` 块中, `uniform` 变量的布局是通过各种布局限制符在编译和链接时控制的。如果使用了默认的布局方式, 那么需要判断每个变量在 `uniform` 块中的偏移量和数据存储大小。为此, 需要调用两个命令: `glGetUniformIndices()` 负责获取指定名称 `uniform` 变量的索引位置, 而 `glGetActiveUniformsiv()` 可以获得指定索引位置的偏移量和大小, 如例 2.4 所示。

```
void glGetUniformIndices(GLuint program, GLsizei uniformCount, const char**
uniformNames, GLuint* uniformIndices);
```

返回所有 `uniformCount` 个 `uniform` 变量的索引位置, 变量的名称通过字符串

数组 `uniformNames` 来指定，程序返回值保存在数组 `uniformIndices` 当中。在 `uniformNames` 中的每个名称都是以 `NULL` 来结尾的，并且 `uniformNames` 和 `uniformIndices` 的数组元素数都应该是 `uniformCount` 个。如果在 `uniformNames` 中给出的某个名称不是当前启用的 `uniform` 变量名称，那么 `uniformIndices` 中对应的位置将会记录为 `GL_INVALID_INDEX`。

例 2.4 初始化一个命名 `uniform` 块中的 `uniform` 变量

/* 顶点着色器和片元着色器共享同一个名称为 “Uniforms” 的 `uniform` 块 */

```

** named "Uniforms" */
const char* vShader = {
    "#version 330 core\n"
    "uniform Uniforms {"
    "    vec3  translation;"
    "    float scale;"
    "    vec4  rotation;"
    "    bool  enabled;"
    "};"
    "in vec2  vPos;"
    "in vec3  vColor;"
    "out vec4 fColor;"
    "void main()"
    "{"
    "    vec3  pos = vec3(vPos, 0.0);"
    "    float  angle = radians(rotation[0]);"
    "    vec3  axis = normalize(rotation.yzw);"
    "    mat3  I = mat3(1.0);"
    "    mat3  S = mat3(
        0, -axis.z,  axis.y,
        axis.z, 0, -axis.x,
        -axis.y, axis.x, 0);"
    "    mat3  uuT = outerProduct(axis, axis);"
    "    mat3  rot = uuT + cos(angle)*(I - uuT)
        + sin(angle)*S;"
    "    pos *= scale;"
    "    pos *= rot;"
    "    pos += translation;"
    "    fColor = vec4(scale, scale, scale, 1);"
    "    gl_Position = vec4(pos, 1);"
    "}"
};

const char* fShader = {
    "#version 330 core\n"
    "uniform Uniforms {"
    "    vec3  translation;"
    "    float scale;"
    "    vec4  rotation;"
    "    bool  enabled;"
    "};"
    "in vec4 fColor;"
    "out vec4 color;"
    "void main()"
    "{"

```

```

        "    color = fColor;"
        "}"
    };

/* 用于将 GLSL 类型转换为存储大小的辅助函数 */
size_t
TypeSize(GLenum type)
{
    size_t    size;

#define CASE(Enum, Count, Type) \
    case Enum:    size = Count * sizeof(Type); break

    switch (type) {
        CASE(GL_FLOAT,          1, GLfloat);
        CASE(GL_FLOAT_VEC2,     2, GLfloat);
        CASE(GL_FLOAT_VEC3,     3, GLfloat);
        CASE(GL_FLOAT_VEC4,     4, GLfloat);
        CASE(GL_INT,            1, GLint);
        CASE(GL_INT_VEC2,       2, GLint);
        CASE(GL_INT_VEC3,       3, GLint);
        CASE(GL_INT_VEC4,       4, GLint);
        CASE(GL_UNSIGNED_INT,    1, GLuint);
        CASE(GL_UNSIGNED_INT_VEC2, 2, GLuint);
        CASE(GL_UNSIGNED_INT_VEC3, 3, GLuint);
        CASE(GL_UNSIGNED_INT_VEC4, 4, GLuint);
        CASE(GL_BOOL,           1, GLboolean);
        CASE(GL_BOOL_VEC2,      2, GLboolean);
        CASE(GL_BOOL_VEC3,      3, GLboolean);
        CASE(GL_BOOL_VEC4,      4, GLboolean);
        CASE(GL_FLOAT_MAT2,      4, GLfloat);
        CASE(GL_FLOAT_MAT2x3,    6, GLfloat);
        CASE(GL_FLOAT_MAT2x4,    8, GLfloat);
        CASE(GL_FLOAT_MAT3,      9, GLfloat);
        CASE(GL_FLOAT_MAT3x2,    6, GLfloat);
        CASE(GL_FLOAT_MAT3x4,   12, GLfloat);
        CASE(GL_FLOAT_MAT4,     16, GLfloat);
        CASE(GL_FLOAT_MAT4x2,    8, GLfloat);
        CASE(GL_FLOAT_MAT4x3,   12, GLfloat);
        #undef CASE

        default:
            fprintf(stderr, "Unknown type: 0x%x\n", type);
            exit(EXIT_FAILURE);
            break;
    }

    return size;
}

void
init()
{
    GLuint program;

    glClearColor(1, 0, 0, 1);

    ShaderInfo shaders[] = {

```



```

        { GL_VERTEX_SHADER, vShader },
        { GL_FRAGMENT_SHADER, fShader },
        { GL_NONE, NULL }
    };

    program = LoadShaders(shaders);
    glUseProgram(program);

    /* 初始化 uniform 块 "Uniforms" 中的变量 */
    GLuint uboIndex;
    GLint uboSize;
    GLuint ubo;
    GLvoid *buffer;

    /* 查找 "Uniforms" 的 uniform 缓存索引, 并判断整个块的大小 */
    uboIndex = glGetUniformBlockIndex(program, "Uniforms");

    glGetActiveUniformBlockiv(program, uboIndex,
        GL_UNIFORM_BLOCK_DATA_SIZE, &uboSize);

    buffer = malloc(uboSize);

    if (buffer == NULL) {
        fprintf(stderr, "Unable to allocate buffer\n");
        exit(EXIT_FAILURE);
    }

    else {
        enum { Translation, Scale, Rotation, Enabled, NumUniforms };

        /* 准备存储在缓存对象中的值 */
        GLfloat scale = 0.5;
        GLfloat translation[] = { 0.1, 0.1, 0.0 };
        GLfloat rotation[] = { 90, 0.0, 0.0, 1.0 };
        GLboolean enabled = GL_TRUE;

        /* 我们可以建立一个变量名称数组, 对应块中已知的 uniform 变量 */
        const char* names[NumUniforms] = {
            "translation",
            "scale",
            "rotation",
            "enabled"
        };

        /* 查询对应的属性, 以判断向数据缓存中写入数值的位置 */
        GLuint indices[NumUniforms];
        GLint size[NumUniforms];
        GLint offset[NumUniforms];
        GLint type[NumUniforms];

        glGetUniformIndices(program, NumUniforms, names, indices);
        glGetActiveUniformsiv(program, NumUniforms, indices,
            GL_UNIFORM_OFFSET, offset);
        glGetActiveUniformsiv(program, NumUniforms, indices,
            GL_UNIFORM_SIZE, size);
        glGetActiveUniformsiv(program, NumUniforms, indices,

```

```

        GL_UNIFORM_TYPE, type);

    /* 将 uniform 变量值拷贝到缓存中 */
    memcpy(buffer + offset[Scale], &scale,
           size[Scale] * TypeSize(type[Scale]));
    memcpy(buffer + offset[Translation], &translation,
           size[Translation] * TypeSize(type[Translation]));
    memcpy(buffer + offset[Rotation], &rotation,
           size[Rotation] * TypeSize(type[Rotation]));
    memcpy(buffer + offset[Enabled], &enabled,
           size[Enabled] * TypeSize(type[Enabled]));
    /* 建立 uniform 缓存对象, 初始化存储内容, 并且与着色器程序建立关联 */
    glGenBuffers(1, &ubo);
    glBindBuffer(GL_UNIFORM_BUFFER, ubo);
    glBufferData(GL_UNIFORM_BUFFER, uboSize,
                 buffer, GL_STATIC_RAW);
    glBindBufferBase(GL_UNIFORM_BUFFER, uboIndex, ubo);
}
...
}

```

2.4.4 buffer 块

GLSL 中的 buffer 块, 或者对于应用程序而言, 就是着色器的存储缓存对象 (shader storage buffer object), 它的行为类似 uniform 块。不过两者之间有两个决定性的差别, 使得 buffer 块的功能更为强大。首先, 着色器可以写入 buffer 块, 修改其中的内容并呈现给其他的着色器调用或者应用程序本身。其次, 可以在渲染之前再决定它的大小, 而不是编译和链接的时候。例如:

```

buffer BufferObject { // 创建一个可读写的 buffer 块
    int mode;          // 序言 (preamble) 成员
    vec4 points[];     // 最后一个成员可以是未定义大小的数组
};

```

如果在着色器中没有给出上面的数组的大小, 那么可以在应用程序中编译和连接之后, 渲染之前设置它的大小。着色器中可以通过 `length()` 方法获取渲染时的数组大小。

着色器可以对 buffer 块中的成员执行读或写操作。写入操作对着色器存储缓存对象的修改对于其他着色器调用都是可见的。这种特性对于计算着色器非常有意义, 尤其是对非图像的内存区域进行处理的时候。

有关 buffer 块的内存限制符 (例如 `coherent`) 以及原子操作的相关深入讨论请参见第 11 章。

设置着色器存储缓存对象的方式与设置 uniform 缓存的方式类似, 不过 `glBindBuffer()` 和 `glBufferData()` 需要使用 `GL_SHADER_STORAGE_BUFFER` 作为目标参数。我们可以在 11.2 节中看到一个更完整的例子。

如果你不需要写入缓存中, 那么可以直接使用 uniform 块, 并且硬件设备本身可能也没有足够的资源空间来支持 buffer 块, 但是 uniform 块通常是足够的。

2.4.5 in/out 块

着色器变量从一个阶段输出，然后再输入到下一个阶段中，这一过程可以使用块接口来表示。使用逻辑上成组的方式来进行组织也更有利于判断两个阶段的数据接口是否一致，同样对单独程序的链接也会变得更为简单。

例如，一个顶点着色器的输出可能为：

```
out Lighting {
    vec3 normal;
    vec2 bumpCoord;
};
```

它必须与片元着色器的输入是匹配的：

```
in Lighting {
    vec3 normal;
    vec2 bumpCoord;
};
```

顶点着色器可以输出材质和光照的信息，并且都分成独立的数据块。OpenGL 着色语言中内置的接口同样也是以块的方式存在的，例如 `gl_PerVertex`，其中包含了内置变量 `gl_Position` 等信息。我们可以在附录 C 中找到一个完整的内置变量列表。

2.5 着色器的编译

OpenGL 着色器程序的编写与 C 语言等基于编译器的语言非常类似。我们使用编译器来解析程序，检查是否存在错误，然后将它翻译为目标代码。然后，在链接过程中将一系列目标文件合并，并产生最终的可执行程序。在程序中使用 GLSL 着色器的过程与之类似，只不过编译器和链接器都是 OpenGL API 的一部分而已。

图 2-1 给出了创建 GLSL 着色器对象并且通过链接来生成可执行着色器程序的过程。

对于每个着色器程序，我们都需要在应用程序中通过下面的步骤进行设置。

对于每个着色器对象：

- 1) 创建一个着色器对象。
- 2) 将着色器源代码编译为对象。
- 3) 验证着色器的编译是否成功。

然后需要将多个着色器对象链接为一个着

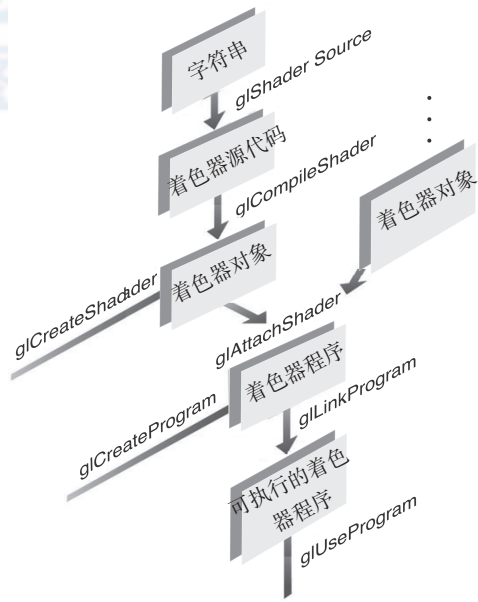


图 2-1 着色器编译命令序列

色器程序，包括：

- 1) 创建一个着色器程序。
- 2) 将着色器对象关联到着色器程序。
- 3) 链接着色器程序。
- 4) 判断着色器的链接过程是否成功完成。
- 5) 使用着色器来处理顶点和片元。

为什么要创建多个着色器对象？这是因为我们有可能在不同的程序中复用同一个函数，而 GLSL 程序也是同一个道理。我们创建的通用函数可以在多个着色器中得到复用。因此不需要使用大量的通用代码来编译大量的着色器资源，只需要将合适的着色器对象链接为一个着色器程序即可。

调用 `glCreateShader()` 来创建着色器对象。

```
GLuint glCreateShader(GLenum type);
```

分配一个着色器对象。`type` 必须是 `GL_VERTEX_SHADER`、`GL_FRAGMENT_SHADER`、`GL_TESS_CONTROL_SHADER`、`GL_TESS_EVALUATION_SHADER` 或者 `GL_GEOMETRY_SHADER` 中的一个。返回值可能是一个非零的整数值，如果为 0 则说明发生了错误。

当我们使用 `glCreateShader()` 创建了着色器对象之后，就可以将着色器的源代码关联到这个对象上。这一步需要调用 `glShaderSource()` 函数。

```
void glShaderSource(GLuint shader, GLsizei count, const GLchar** string, const GLint* length);
```

将着色器源代码关联到一个着色器对象 `shader` 上。`string` 是一个由 `count` 行 `GLchar` 类型的字符串组成的数组，用来表示着色器的源代码数据。`string` 中的字符串可以是 NULL 结尾的，也可以不是。而 `length` 可以是以下三种值的一种。如果 `length` 是 NULL，那么我们假设 `string` 给出的每行字符串都是 NULL 结尾的。否则，`length` 中必须有 `count` 个元素，它们分别表示 `string` 中对应行的长度。如果 `length` 数组中的某个值是一个整数，那么它表示对应的字符串中的字符数。如果某个值是负数，那么 `string` 中的对应行假设为 NULL 结尾。

如果要编译着色器对象的源代码，需要使用 `glCompileShader()` 函数。

```
void glCompileShader(GLuint shader);
```

编译着色器的源代码。结果查询可以调用 `glGetShaderiv()`，并且参数为 `GL_COMPILE_STATUS`。

这里与 C 语言程序的编译类似，需要自己判断编译过程是否正确地完成。调用 `glGetShaderiv()` 并且参数为 `GL_COMPILE_STATUS`，返回的就是编译过程的状态。如果返回为 `GL_TRUE`，那么编译成功，下一步可以将对象链接到一个着色器程序中。如果编译失败，那么可以通过调取编译日志来判断错误的原因。`glGetShaderInfoLog()` 函数会返回一个与具体实现相关的信息，用于描述编译时的错误。这个错误日志的大小可以通过调用 `glGetShaderiv()`（带参数 `GL_INFO_LOG_LENGTH`）来查询。

```
void glGetShaderInfoLog(GLuint shader, GLsizei bufSize, GLsizei* length, char* infoLog);
```

返回 `shader` 的最后编译结果。返回的日志信息是一个以 `NULL` 结尾的字符串，它保存在 `infoLog` 缓存中，长度为 `length` 个字符串。日志可以返回的最大值是通过 `bufSize` 来定义的。

如果 `length` 设置为 `NULL`，那么将不会返回 `infoLog` 的大小。

当创建并编译了所有必要的着色器对象之后，下一步就是链接它们以创建一个可执行的着色器程序。这个过程与创建着色器对象的过程类似。首先，我们创建一个着色器程序，以便将着色器对象关联到其上。这里用到了 `glCreateProgram()` 函数。

```
GLuint glCreateProgram(void);
```

创建一个空的着色器程序。返回值是一个非零的整数，如果为 0 则说明发生了错误。

当得到着色器程序之后，下一步可以将它关联到必要的着色器对象上，以创建可执行的程序。关联着色器对象的步骤可以通过调用 `glAttachShader()` 函数来完成。

```
void glAttachShader(GLuint program, GLuint shader);
```

将着色器对象 `shader` 关联到着色器程序 `program` 上。着色器对象可以在任何时候关联到着色器程序，但是它的功能只有经过程序的成功链接之后才是可用的。着色器对象可以同时关联到多个不同的着色器程序上。

与之对应的是，如果我们需要从程序中移除一个着色器对象，从而改变着色器的操作，那么可以调用 `glDetachShader()` 函数，设置对应的着色器对象标识符来解除对象的关联。

```
void glDetachShader(GLuint program, GLuint shader);
```

移除着色器对象 `shader` 与着色器程序 `program` 的关联。如果着色器已经被标记为要删除的对象（调用 `glDeleteShader()`），然后又被解除了关联，那么它将会被即时删除。

当我们将所有必要的着色器对象关联到着色器程序之后，就可以链接对象来生成可执

行程序了。这一步需要调用函数 `glLinkProgram()`。

```
void glLinkProgram(GLuint program);
```

处理所有与 `program` 关联的着色器对象来生成一个完整的着色器程序。链接操作的结果查询可以调用 `glGetProgramiv()`，且参数为 `GL_LINK_STATUS`。如果返回 `GL_TRUE`，那么链接成功；否则，返回 `GL_FALSE`。

由于着色器对象中可能存在问题，因此链接过程依然可能会失败。我们可以调用 `glGetProgramiv()`（带参数 `GL_LINK_STATUS`）来查询链接操作的结果。如果返回 `GL_TRUE`，那么链接操作成功，然后我们可以指定着色器程序来处理顶点和片元数据了。如果链接失败，即返回结果为 `GL_FALSE`，那么我们可以通过调用 `glGetProgramInfoLog()` 函数来获取程序链接的日志信息并判断错误原因。

```
void glGetProgramInfoLog(GLuint program, GLsizei bufSize, GLsizei* length, char* infoLog);
```

返回最后一次 `program` 链接的日志信息。日志返回的字符串以 `NULL` 结尾，长度为 `length` 个字符，保存在 `infoLog` 缓存中。`log` 可返回的最大值通过 `bufSize` 指定。如果 `length` 为 `NULL`，那么不会再返回 `infoLog` 的长度。

如果我们成功地完成了程序的链接，那么就可以调用函数 `glUseProgram()`，并且参数设置为程序对象的句柄来启用顶点或者片元程序。

```
void glUseProgram(GLuint program);
```

使用链接过的着色器程序 `program`。如果 `program` 为零，那么所有当前使用的着色器都会被清除。如果没有绑定任何着色器，那么 OpenGL 的操作结果是未定义的，但是不会产生错误。

如果已经启用了程序，而它需要关联新的着色器对象，或者解除之前关联的对象，那么我们需要重新对它进行链接。如果链接过程成功，那么新的程序会直接替代之前启用的程序。如果链接失败，那么当前绑定的着色器程序依然是可用的，不会被替代，直到我们成功地重新链接或者使用 `glUseProgram()` 指定了新的程序为止。

当着色器对象的任务完成之后，我们可以通过 `glDeleteShader()` 将它删除，并且不需要关心它是否关联到某个活动程序上。这一点与 C 语言程序的链接是相同的，当我们得到可执行程序之后，就不再需要对象文件了，直到我们再次进行编译为止。


```
void glDeleteShader(GLuint shader);
```

删除着色器对象 `shader`。如果 `shader` 当前已经链接到一个或者多个激活的着色器程序上，那么它将被标识为“可删除”，当对应的着色器程序不再使用的时候，就会自动删除这个对象。

与此类似，如果我们不再使用某个着色器程序，也可以直接调用 `glDeleteProgram()` 删除它。

```
void glDeleteProgram(GLuint program);
```

立即删除一个当前没有在任何环境中使用的着色器程序 `program`，如果程序正在被某个环境使用，那么等到它空闲时再删除。

最后，为了确保接口的完整性，还可以调用 `glIsShader()` 来判断某个着色器对象是否存在，或者通过 `glIsProgram()` 判断着色器程序是否存在。

```
GLboolean glIsShader(GLuint shader);
```

如果 `shader` 是一个通过 `glCreateShader()` 生成的着色器对象的名称，并且没有被删除，那么返回 `GL_TRUE`。如果 `shader` 是零或者不是着色器对象名称的非零值，则返回 `GL_FALSE`。

```
GLboolean glIsProgram(GLuint program);
```

如果 `program` 是一个通过 `glCreateProgram()` 生成的程序对象的名称，并且没有被删除，那么返回 `GL_TRUE`。如果 `program` 是 0 或者不是着色器程序名称的非零值，则返回 `GL_FALSE`。

2.5.1 我们的 LoadShaders() 函数

为了简化应用程序中使用着色器的过程，我们在示例中使用一个 `LoadShaders()` 函数来辅助载入和创建着色器程序。我们已经在第 1 章的第一个程序中用到了这个函数来加载简单的着色器代码。

2.6 着色器子程序

高级技巧

GLSL 允许我们在着色器中定义函数，而这些函数的调用过程总是静态的。如果需要动

态地选择调用不同的函数，那么可以创建两个不同的着色器，或者使用 if 语句来进行运行时的选择，如例 2.5 所示。

例 2.5 静态着色器的控制流程

```
#version 330 core

void func_1() { ... }
void func_2() { ... }

uniform int func;

void
main()
{
    if (func == 1)
        func_1();
    else
        func_2();
}
```

着色器子程序在概念上类似于 C 语言中的函数指针，它可以实现动态子程序选择过程。在着色器当中，可以预先声明一个可用子程序的集合，然后动态地指定子程序的类型。然后，通过设置一个子程序的 uniform 变量，从预设的子程序选择一个并加以执行。

2.6.1 GLSL 的子程序设置

当我们需要在着色器中进行子程序的选择时，通常需要三个步骤来设置一个子程序池。

1) 通过关键字 `subroutine` 来定义子程序的类型：

```
subroutine returnType subroutineType(type param, ...);
```

其中 `returnType` 可以是任何类型的函数返回值，而 `subroutineType` 是一个合法的子程序名称。由于它相当于函数的原型，因此我们只需要给出参数的类型，不一定给出参数的名称（我们可以将它设想为 C 语言中的 `typedef`，而 `subroutineType` 就是新定义的类型）。

2) 使用刚才定义的 `subroutineType`，通过 `subroutine` 关键字来定义这个子程序集合的内容，以便稍后进行动态的选择。某个子程序函数的原型定义类似于下面的形式：

```
subroutine (subroutineType) returnType functionName(...);
```

3) 最后，指定一个子程序 uniform 变量，其中保存了相当于“函数指针”的子程序选择信息，这可以在应用程序中更改：

```
subroutine uniform subroutineType variableName;
```

将上面的三个步骤整合在一起，我们可以通过例 2.6 来实现环境光照和漫反射光照方式的动态选择。

例 2.6 声明一个子程序集合

```
subroutine vec4 LightFunc(vec3); // 第 1 步

subroutine (LightFunc) vec4 ambient(vec3 n) // 第 2 步
{
    return Materials.ambient;
```

```

}

subroutine (LightFunc) vec4 diffuse(vec3 n) // 第 2 步 (重复)
{
    return Materials.diffuse *
        max(dot(normalize(n), LightVec.xyz), 0.0);
}

subroutine uniform LightFunc materialShader; // 第 3 步

```

子程序并不一定只属于一个子程序类型（例如，例 2.6 中的 `LightFunc`）。如果定义了多种类型的子程序，那么我们可以设置一个子程序属于多个类型，方法是在定义子函数时把类型添加到列表中，如下所示：

```

subroutine void Type_1();
subroutine void Type_2();
subroutine void Type_3();

subroutine (Type_1, Type_2) Func_1();
subroutine (Type_1, Type_3) Func_2();

subroutine uniform Type_1 func_1;
subroutine uniform Type_2 func_2;
subroutine uniform Type_3 func_3;

```

在上面的例子中，`func_1` 可以使用 `Func_1` 和 `Func_2`，这是因为两个子程序都指定了 `Type_1`。但是，`func_2` 就只能使用 `Func_1`，而 `func_3` 只能使用 `Func_2`。

2.6.2 选择着色器子程序

如果我们已经在着色器中定义了所有子程序类型和函数，那么只需要在链接后的着色器程序中查询一些数值，然后使用这些数值来选择合适的函数即可。

在之前所示的步骤 3 当中，我们声明了一个子程序的 `uniform` 变量，之后就可以获取它的位置并设置它的值。与其他的 `uniform` 变量不同的是，子程序的 `uniform` 需要使用 `glGetSubroutineUniformLocation()` 来获取自身的位置。

```
GLint glGetUniformLocation(GLuint program, GLenum shadertype, const char* name);
```

返回名为 `name` 的子程序 `uniform` 的位置，相应的着色阶段通过 `shadertype` 来指定。`name` 是一个以 `NULL` 结尾的字符串，而 `shadertype` 的值必须是 `GL_VERTEX_SHADER`、`GL_TESS_CONTROL_SHADER`、`GL_TESS_EVALUATION_SHADER`、`GL_GEOMETRY_SHADER` 或者 `GL_FRAGMENT_SHADER` 中的一个。

如果 `name` 不是一个激活的子程序 `uniform`，则返回 `-1`。如果 `program` 不是一个可用的着色器程序，那么会生成一个 `GL_INVALID_OPERATION` 错误。

当取得了子程序 `uniform` 数值之后，我们需要判断某个子程序在着色器中的索引号。这

一步可以通过调用函数 `glGetSubroutineIndex()` 来完成。

```
GLuint glGetSubroutineIndex(GLuint program, GLenum shadertype, const char* name);
```

从程序 `program` 中返回 `name` 所对应的着色器函数的索引，相应的着色阶段通过 `shadertype` 来指定。`name` 是一个以 NULL 结尾的字符串，而 `shadertype` 的值必须是 `GL_VERTEX_SHADER`、`GL_TESS_CONTROL_SHADER`、`GL_TESS_EVALUATION_SHADER`、`GL_GEOMETRY_SHADER` 或者 `GL_FRAGMENT_SHADER` 中的一个。

如果 `name` 不是 `shadertype` 着色器的一个活动子程序，那么会返回 `GL_INVALID_INDEX`。

当我们得到了子程序的索引以及 `uniform` 的位置之后，可以使用 `glUniformSubroutinesuiv()` 来指定在着色器中执行哪一个子程序函数。某个着色阶段中，所有的子程序 `uniform` 都必须先经过初始化的过程。

```
GLuint glUniformSubroutinesuiv(GLenum shadertype, GLsizei count, const GLuint* indices);
```

设置所有 `count` 个着色器子程序 `uniform` 使用 `indices` 数组中的值，相应的着色阶段通过 `shadertype` 来指定。`shadertype` 的值必须是 `GL_VERTEX_SHADER`、`GL_TESS_CONTROL_SHADER`、`GL_TESS_EVALUATION_SHADER`、`GL_GEOMETRY_SHADER` 或者 `GL_FRAGMENT_SHADER` 中的一个。第 `i` 个子程序 `uniform` 对应于 `indices[i]` 的值。

如果 `count` 不等于当前绑定程序的着色阶段 `shadertype` 的 `GL_ACTIVE_SUBROUTINE_UNIFORM_LOCATIONS` 值，那么会产生一个 `GL_INVALID_VALUE` 错误。`indices` 中的所有值都必须小于 `GL_ACTIVE_SUBROUTINES`，否则会产生一个 `GL_INVALID_VALUE` 错误。

将上面的步骤组合在一起，可以得到下面的代码段，它演示了例 2.6 中的顶点着色器的调用过程。

```
GLint  materialShaderLoc;
GLuint ambientIndex;
GLuint diffuseIndex;

glUseProgram(program);

materialShaderLoc = glGetSubroutineUniformLocation(
    program, GL_VERTEX_SHADER, "materialShader");

if (materialShaderLoc < 0) {
    // 错误: materialShader 不是着色器中启用的子程序 uniform
}

ambientIndex = glGetSubroutineIndex(program,
```

```

        GL_VERTEX_SHADER,
        "ambient");
diffuseIndex = glGetSubroutineIndex(program,
        GL_VERTEX_SHADER,
        "diffuse");
if (ambientIndex == GL_INVALID_INDEX ||
    diffuseIndex == GL_INVALID_INDEX) {
    // 错误：指定的子程序在 GL_VERTEX_SHADER 阶段当前绑定的程序中没有启用
}
else {
    GLsizei n;
    glGetIntegerv(GL_MAX_SUBROUTINE_UNIFORM_LOCATIONS, &n);

    GLuint *indices = new GLuint[n];
    indices[materialShaderLoc] = ambientIndex;

    glUniformSubroutinesuiv(GL_VERTEX_SHADER, n, indices);

    delete [] indices;
}

```



注意 调用 `glUseProgram()` 时会重新设置所有子程序 `uniform` 的值，具体的顺序与硬件实现相关。

2.7 独立的着色器对象

高级技巧

在 OpenGL 4.1 版本之前（不考虑扩展功能），在应用程序中，同一时间只能绑定一个着色器程序。如果你的程序需要使用多个片元着色器来处理来自同一个顶点着色器的几何体变换数据，那么这样会变得很不方便。此时只能将同一个顶点着色器复制多份，并且多次绑定到不同的着色器程序，从而造成了资源的浪费和代码的重复。

独立的着色器对象可以将不同程序的着色阶段（例如顶点着色）合并到同一个程序管线中。

第一步，我们需要创建用于着色器管线的着色器程序。我们可以调用 `glProgramParameteri()` 函数并且设置参数为 `GL_PROGRAM_SEPARABLE`，然后再链接着色器程序。这样该程序就被标识为在程序管线中使用。如果想要简化这个过程，还可以直接使用新增的 `glCreateShaderProgramv()` 来封装着色器编译过程，并且将程序标识为可共享（如上文所述），然后链接到最终的对象。

将着色器程序集合合并之后，就需要用这个新的着色器管线结构来合并多个程序中的着色阶段。对于 OpenGL 中的大部分对象来说，都有一个生成 - 绑定 - 删除的过程，以及对应可用的函数。着色器管线的创建可以调用 `glGenProgramPipelines()`，即创建一个未使用的程序管线标识符，然后将它传入 `glBindProgramPipeline()`，使得该程序可以自由编辑（例如，添加或者替换着色阶段）和使用。与其他生成的对象相似，程序管线可以通过

`glDeleteProgramPipelines()` 来删除。

当绑定了一个程序管线之后，可以调用 `glUseProgramStages()` 将之前标记为独立的程序对象关联到管线上，它通过位域的方式来描述该管线处理几何体和着色片元时，给定程序所处的着色阶段。而之前的 `glUseProgram()` 只能直接调用一个程序并且替换当前绑定的程序管线。

为了确保管线可以使用，着色器阶段之间的接口——in 和 out 变量——也必须是匹配的。非独立的着色器对象在程序链接时就可以检查这些接口的匹配情况，与之相比，使用独立程序对象的着色器管线只能在绘制 - 调用过程中进行检查。如果接口没有正确匹配，那么所有的可变变量（out 变量）都未定义。

内置的 `gl_PerVertex` 块必须重新声明，以便显式地指定固定管线接口中的哪些部分可以使用。如果管线用到了多个程序，那么这一步是必需的。

例如：

```
out gl_PerVertex {
    vec4 gl_Position;    // 设置 gl_Position 在接口中可用
    float gl_PointSize;  // 设置 gl_PointSize 在接口中可用
};                      // 不再使用 gl_PerVertex 的其他成员
```

这样我们就建立了着色器的输出接口，它将用于后继的管线阶段当中。这里必须使用 `gl_PerVertex` 自己的内置成员。如果不同的着色器程序都用到了同一个内置的块接口，那么所有的着色器都必须使用相同的方式重新声明这个内置的块。

因为独立的着色器对象可以有各自独立的程序 uniform 集合，所以我们可以使用两种方法来设置 uniform 变量的值。第一种方法是通过 `glActiveShaderProgram()` 来选择一个活动的着色器程序，然后调用 `glUniform*()` 和 `glUniformMatrix*()` 来设置某个着色器程序的 uniform 变量的值。另一种方法，也是我们推荐的方法，是调用 `glProgramUniform*()` 和 `glProgramUniformMatrix*()` 函数，它们有一个显式的 program 对象参数，这样可以独立地设置某个程序的 uniform 变量的值。

```
void glProgramUniform{1234}{fdi ui}(GLuint program, GLint location, TYPE value);
void glProgramUniform{1234}{fdi ui}v(GLuint program, GLint location, GLsizei
count, const TYPE* values);
void glProgramUniformMatrix{234}{fd}v(GLuint program, GLint location, GLsizei
count, GLboolean transpose, const GLfloat* values);
void glProgramUniformMatrix{2x3,2x4,3x2,3x4,4x2,4x3}{fd}v( GLuint program,
GLint location, GLsizei count, GLboolean transpose, const GLfloat* values);
```

`glProgramUniform*()` 和 `glProgramUniformMatrix*()` 函数的使用与 `glUniform*()` 和 `glUniformMatrix*()` 的使用是一样的，唯一的区别是使用一个 program 参数来设置准备更新 uniform 变量的着色器程序。这些函数的主要优点是，program 可以不是当前绑定的程序（即最后一个使用 `glUseProgram()` 指定的着色器程序）。

OpenGL 绘制方式

本章目标

阅读完本章内容之后，你将会具备以下的能力：

- ❑ 辨别所有 OpenGL 中可用的渲染图元。
- ❑ 初始化和设置数据缓存，用于几何体的渲染。
- ❑ 使用多实例渲染（instanced rendering）等高级技法对渲染进行优化。

OpenGL 的主要作用就是将图形渲染到帧缓存当中。为了实现这一要求，需要将复杂的物体分解成图元的形式（包括点、线，以及三角形），当它们的分布密度足够高时，就可以表达为 2D 以及 3D 物体的形态。OpenGL 中包含了很多渲染这类图元的函数。这些函数允许我们描述图元在内存中的布局、渲染的数量和渲染所采取的形式，甚至是同一组图元在一个函数调用中所复制的数量。这些函数几乎是 OpenGL 最为重要的函数组成，如果没有它们的话，那么我们可能除了清除屏幕之外无法再完成任何事情。

这一章将会包含以下几节：

- ❑ 3.1 节介绍 OpenGL 中可以用于渲染的图元类型。
- ❑ 3.2 节解释 OpenGL 中数据处理的机制。
- ❑ 3.3 节给出顶点数据的渲染以及顶点着色器中的处理过程。
- ❑ 3.4 节介绍 OpenGL 中用于绘制的函数集。
- ❑ 3.5 节讲解高效地使用同一顶点数据来实现多个物体的渲染方法。

3.1 OpenGL 图元

OpenGL 可以支持很多种不同的图元类型。不过它们最后都可以归结为三种类型中的

一种，即点、线，或者三角形。线和三角形图元类型可以再组合为条带、循环体（线），或者扇面（三角形）。点、线和三角形也是大部分图形硬件设备所支持的基础图元类型^⑨。OpenGL 还支持其他一些图元类型，包括作为细分器输入的 Patch 类型，以及作为几何着色器输入的邻接图元（adjacency primitive）。细分（以及细分着色器）的介绍可以参见第 9 章，而几何着色器的介绍可以参见第 10 章。在这两章中也会对 Patch 和邻接图元类型进行更深入的讲解。在这一节当中，我们只介绍点、线和三角形这三种图元类型。

3.1.1 点

点可以通过单一的顶点来表示。一个点也就是一个四维的齐次坐标值。因此，点实际上不存在面积，因此在 OpenGL 中它是通过显示屏幕（或者绘制缓存）上的一个四边形区域来模拟的。当渲染点图元的时候，OpenGL 会通过一系列光栅化规则来判断点所覆盖的像素位置。在 OpenGL 中对点进行光栅化是非常直接的，如果某个采样值落入点在窗口坐标系中的四边形当中，那么就认为这个采样值被点所覆盖。四边形区域的边长等于点的大小，它是一个固定的状态（通过 `glPointSize()` 设置），也可以在顶点、细分和几何着色器中向内置变量 `gl_PointSize` 写入值来进行改变。只有开启了 `GL_PROGRAM_POINT_SIZE` 状态之后，我们才能在着色器中写入 `gl_PointSize`，否则这个值将被忽略，系统依然会使用 `glPointSize()` 所设置的数值。

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

设置固定的像素大小，如果没有开启 `GL_PROGRAM_POINT_SIZE`，那么它将被用于设置点的大小。

默认的点大小为 1.0。因此当我们渲染点的时候，每个顶点实际上都是屏幕上的一个像素（当然，被剪切的点除外）。如果点的大小增加（无论是通过 `glPointSize()` 还是向 `gl_PointSize` 写入一个大于 1.0 的值），那么每个点的顶点都会占据超过 1 个像素的值。例如，如果点的尺寸为 1.2，并且顶点正好处于一个像素的中心，那么只有这个像素会受到光照的影响。但是如果顶点正好处于两个水平或者垂直的相邻像素中心之间，那么这两个像素都会受到光照的影响（即这两个像素都会被照亮）。如果顶点正好位于 4 个相邻像素的中点上，那么这 4 个像素都会被照亮，也就是说一个点会同时影响 4 个像素的值！

点精灵

如果使用 OpenGL 来渲染点，那么点的每个片元都会执行片元着色器。在本质上每个点都是屏幕上的方形区域，而每个像素都可以使用不同的颜色来着色。我们可以在片元着色器中通过解析纹理图来实现点的着色。OpenGL 的片元着色器中提供了一个特殊的内置

⑨ 所谓的硬件支持，也就是图形处理器当中直接提供了这些图元类型的光栅化操作。其他图元类型，例如 Patch 和邻接图元，是无法直接进行光栅化的。

变量来辅助这一需求，它叫做 `gl_PointCoord`，其中包含了当前片元在点区域内的坐标信息。`gl_PointCoord` 只能在片元着色器中工作（将它包含在其他着色器中也没有什么意义），它的值只对于点的渲染有效。如果将 `gl_PointCoord` 作为输入纹理坐标使用，那么就可以使用位图和纹理替代简单的方块颜色。将结果进行 `alpha` 融混或者直接抛弃某些片元（使用 `discard` 关键字）之后，我们还可以创建不同形状的点精灵（point sprite）对象。

我们会在后面的内容即 6.13 节中给出有关点精灵的简短例子。

3.1.2 线、条带与循环线

OpenGL 当中的线表示一条线段，而不是数学上的无限延伸的方向向量。独立的线可以通过一对顶点来表达，每个顶点表示线的一个端点。多段线也可以进行链接来表示一系列的线段，它们还可以是首尾闭合的。闭合的多段线叫做循环线（line loop），而开放的多段线（没有首尾闭合）叫做条带线（line strip）。与点类似的是，线从原理上来说也不存在面积，因此也需要使用特殊规则来判断线段的光栅化会影响哪些像素值。线段光栅化的规则也称作 diamond exit 规则。在 OpenGL 的标准说明书中给出了其详细的解释。但是，在这里还是需要对其进行重新讲解。假设每个像素在屏幕上的方形区域中都存在一个菱形，当对一条从点 A 到点 B 的线段进行光栅化，并且线段穿过了菱形的假想边时，这个像素应该受到其影响——除非菱形中包含的正好是点 B（即线段的末端点位于菱形内）。不过，如果还绘制了另一条从点 B 到点 C 的线段，那么此时 B 点所在的像素只会更新一次。

diamond exit 规则对于细线段是有效的，但是 OpenGL 也可以通过 `glLineWidth()` 函数来设置线段的宽度大小（相当于之前的 `glPointSize()`）。

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

设置线段的固定宽度。默认值为 1.0。这里的 `width` 表示线段的宽度，它必须是一个大于 0.0 的值，否则会产生一个错误信息。

线段并不能通过类似 `gl_PointSize` 的着色器变量来设置，OpenGL 中的线段绘制必须通过固定宽度的渲染状态来切换。如果线段宽度大于 1，那么线段将被水平和垂直复制宽度大小的次数。如果线段为 Y 主序（即它主要是向垂直方向延伸的，而不是水平方向），那么复制过程是水平方向的。如果它是 X 主序，那么复制过程就是垂直方向进行的。

如果没有开启反走样的话，OpenGL 标准对于线段端点的表示方法以及线宽的光栅化方法是相对自由的。如果开启了反走样的话，那么线段将被视为沿着线方向对齐的矩形块，其宽度等于当前设置的线宽。

3.1.3 三角形、条带与扇面

三角形是三个顶点的集合组成的。当我们分别渲染多个三角形的时候，每个三角形都

与其他三角形完全独立。三角形的渲染是通过三个顶点到屏幕的投影以及三条边的链接来完成的。如果屏幕像素的采样值位于三条边的正侧半空间内的话，那么它受到了三角形的影响。如果两个三角形共享了一条边（即共享一对顶点），那么不可能有任何采样值同时位于这两个三角形之内。这一点非常重要的原因是，虽然 OpenGL 可以支持多种不同的光栅化算法，但是对于共享边上的像素值设置却有着严格的规定：

❑ 两个三角形的共享边上的像素值因为同时被两者所覆盖，因此不可能不受到光照计算的影响。

❑ 两个三角形的共享边上的像素值，不可能受到多于一个三角形的光照计算的影响。

这也就是说，OpenGL 对于模型三角形共享边的光栅化过程不会产生任何裂缝，也不会产生重复的绘制^①。这一点对于三角形条带（triangle strip）或者扇面（triangle fan）的光栅化过程非常重要，此时前三个顶点将会构成第一个三角形，后继的顶点将与之前三角形的后两个顶点一起构成新的三角形。这一过程的图示如图 3-1 所示。

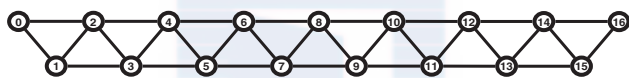


图 3-1 三角形条带的顶点布局

当渲染三角形扇面的时候，第一个顶点会作为一个共享点存在，它将作为每一个后继三角形的组成部分。而之后的每两个顶点都会与这个共享点一起组成新的三角形。三角形条带可以用于表达任何复杂程度的凸多边形形状。图 3-2 所示为三角形扇面的顶点布局。

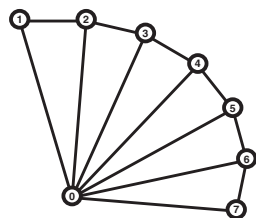


图 3-2 三角形扇面的顶点布局

这些图元类型都可以用于下一节介绍的绘制函数当中。它们通过 OpenGL 的枚举量来进行表达，并且作为渲染函数的输入参数。表 3-1 所示为图元类型与 OpenGL 枚举量之间的对应关系。

表 3-1 OpenGL 图元的模式标识

图元类型	OpenGL 枚举量
点	GL_POINTS
线	GL_LINES
条带线	GL_LINE_STRIP
循环线	GL_LINE_LOOP
独立三角形	GL_TRIANGLES
三角形条带	GL_TRIANGLE_STRIP
三角形扇面	GL_TRIANGLE_FAN

① 重复绘制也就是对同一个像素执行多于一次的光照计算，这样可能会造成一些问题，例如在融混（blending）时产生错误的结果。

将多边形渲染为点集、轮廓线或者实体

一个多边形有两个面：正面和背面，当不同的面朝向观察者时，它们的渲染结果可能是不一样的。因此在观察一个实体物体的剖面时，可以很明显地区分出它的内部和外部表面。默认情况下，正面和背面的绘制方法是一致的。如果要改变这一属性，或者仅仅使用轮廓线或者顶点来进行绘制的话，可以调用 `glPolygonMode()` 命令。

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

控制多边形的正面与背面绘制模式。参数 `face` 必须是 `GL_FRONT_AND_BACK`，而 `mode` 可以是 `GL_POINT`、`GL_LINE` 或者 `GL_FILL`，它们分别设置多边形的绘制模式是点集、轮廓线还是填充模式。默认情况下，正面和背面的绘制都使用填充模式来完成。

多边形面的反转和裁减

从惯例上来说，多边形正面的顶点在屏幕上应该是逆时针方向排列的。因此我们可以构建任何“可能”的实体表面——从数学上来说，这类表面称作有向流形（orientable manifold），即方向一致的多边形——例如球体、环形体和茶壶等都是有向的；而克林瓶（Klein bottle）和莫比乌斯带（Möbius strip）不是。换句话说，这样的多边形可以是完全顺时针的，或者完全逆时针的。

假设我们一致地描述了一个有向的模型表面，但是它的外侧需要使用顺时针方向来进行描述。此时可以通过 OpenGL 的函数 `glFrontFace()` 来反转（reversing）背面，并重新设置多边形正面所对应的方向。

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

控制多边形正面的判断方式。默认模式为 `GL_CCW`，即多边形投影到窗口坐标系之后，顶点按照逆时针排序的面作为正面。如果模式为 `GL_CW`，那么采用顺时针方向的面将被认为是物体的正面。



注意 顶点的方向（顺时针或者逆时针）也可以被称为顶点的趋势（winding）。

对于一个由不透明的且方向一致的多边形组成的、完全封闭的模型表面来说，它的所有背面多边形都是不可见的——它们永远会被正面多边形所遮挡。如果位于模型的外侧，那么可以开启裁减（culling）来直接抛弃 OpenGL 中的背面多边形。与之类似，如果位于模型之内，那么只有背面多边形是可见的。如果需要指示 OpenGL 自动抛弃正面或者背面的多边形，可以使用 `glCullFace()` 命令，同时通过 `glEnable()` 开启裁减。


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

在转换到屏幕空间渲染之前，设置需要抛弃（裁减）哪一类多边形。`mode` 可以是 `GL_FRONT`、`GL_BACK` 或者 `GL_FRONT_AND_BACK`，分别表示正面、背面或者所有多边形。要使命令生效，我们还需要使用 `glEnable()` 和 `GL_CULL_FACE` 参数来开启裁减；之后也可以使用 `glDisable()` 和同样的参数来关闭它。

高级技巧

从更专业的角度来说，判断多边形的面是正面还是背面，需要依赖于这个多边形在窗口坐标系下的面积计算。而面积计算的一种方法是

$$a = \frac{1}{2} \sum_{i=0}^{n-1} x_i y_{i \oplus 1} - x_{i \oplus 1} y_i$$

其中 x_i 和 y_i 分别为多边形的 n 个顶点中第 i 个顶点的窗口坐标 x 和 y ，而 $i \oplus 1$ 是公式 $(i + 1) \bmod n$ 的缩写形式，其中 `mod` 表示取余数的操作。

假设我们设置为 `GL_CCW`，那么 $a > 0$ 的时候，顶点所对应的多边形就是位于正面的；否则它位于背面。如果设置为 `GL_CW` 且 $a < 0$ ，那么对应的多边形位于正面；否则它位于背面。

3.2 OpenGL 缓存数据

几乎所有使用 OpenGL 完成的事情都用到了缓存 buffers 中的数据。OpenGL 的缓存表示为缓存对象（buffer object）。在第 1 章里我们已经简要地介绍了缓存对象的意义。不过，这一节将稍微深入到缓存对象的方方面面当中，包括它的种类、创建方式、管理和销毁，以及与缓存对象有关的一些最优解决方案。

3.2.1 创建与分配缓存

与 OpenGL 中的很多其他实现类似，缓存对象也是使用 `GLuint` 的值来进行命名的。这个值可以使用 `glGenBuffers()` 命令来创建。我们已经在第 1 章介绍过这个函数了，但是在这里会再次给出它的原型，以方便读者参考。

```
void glGenBuffers(GLsizei n, GLuint* buffers);
```

返回 n 个当前未使用的缓存对象名称，并保存到 `buffers` 数组中。

调用 `glGenBuffers()` 完成之后，我们将在 `buffers` 中得到一个缓存对象名称的数组，但是此时这些名称只是徒有其表。它们还不是真正的缓存对象。只有某个名称首次绑定到系统环境中的一个结合点之后，它所对应的缓存对象才会真正创建出来。这一点非常重要，

因为 OpenGL 会采取一种最优内存管理策略，根据缓存对象完成绑定的情况来分配它对应的内存。可用的缓存结合点（称作目标，target）如表 3-2 中所示。

表 3-2 缓存绑定的目标

目标	用途
GL_ARRAY_BUFFER	这个结合点可以用来保存 <code>glVertexAttribPointer()</code> 设置的顶点数组数据。在实际工程中这一目标可能是最为常用的
GL_COPY_READ_BUFFER 和 GL_COPY_WRITE_BUFFER	这两个目标是一对互相匹配的结合点，用于拷贝缓存之间的数据，并且不会引起 OpenGL 的状态变化，也不会产生任何特殊形式的 OpenGL 调用
GL_DRAW_INDIRECT_BUFFER	如果采取间接绘制（indirect drawing）的方法，那么这个缓存目标用于存储绘制命令的参数，详细的解释请参见下一节
GL_ELEMENT_ARRAY_BUFFER	绑定到这个目标的缓存中可以包含顶点索引数据，以便用于 <code>glDrawElements()</code> 等索引形式的绘制命令
GL_PIXEL_PACK_BUFFER	这一缓存目标用于从图像对象中读取数据，例如纹理和帧缓存数据。相关的 OpenGL 命令包括 <code>glGetTexImage()</code> 和 <code>glReadPixels()</code> 等
GL_PIXEL_UNPACK_BUFFER	这一缓存目标与之前的 GL_PIXEL_PACK_BUFFER 相反，它可以作为 <code>glTexImage2D()</code> 等命令的数据源使用
GL_TEXTURE_BUFFER	纹理缓存也就是直接绑定到纹理对象的缓存，这样就可以直接在着色器中读取它们的数据信息。GL_TEXTURE_BUFFER 可以提供一 个操控此类缓存的目标，但是我们还需要将缓存关联到纹理，才能确保它们在着色器中可用
GL_TRANSFORM_FEEDBACK_BUFFER	transform feedback 是 OpenGL 提供的一种便捷方案，它可以在管线的顶点处理部分结束时（即经过了顶点着色，可能还有几何着色阶段），将经过变换的顶点重新捕获，并且将部分属性写入到缓存对象中。这一目标就提供了这样的结合点，可以建立专门的缓存来记录这些属性数据。transform feedback 的详细介绍请参见 5.4 节的内容
GL_UNIFORM_BUFFER	这个目标可以用于创建 uniform 缓存对象（uniform buffer object）的缓存数据。uniform 缓存的相关介绍请参见 2.4.1 节的内容

缓存对象的建立，实际上就是通过调用 `glGenBuffers()` 函数生成一系列名称，然后通过 `glBindBuffer()` 将一个名称绑定到表 3-2 中的一个目标来完成的。第 1 章当中已经介绍过 `glBindBuffer()` 和 `glBindBuffer()` 函数，不过这里将再次给出函数的原型，以保证文字的完整性。

```
void glBindBuffer(GLenum target, GLuint buffer);
```

将名称为 buffer 的缓存对象绑定到 target 所指定的缓存结合点。target 必须是 OpenGL 支持的缓存绑定目标之一，buffer 必须是通过 `glGenBuffers()` 分配的名称。如果 buffer 是第一次被绑定，那么它所对应的缓存对象也将同时被创建。

好了，现在我们已经将缓存对象绑定到表 3-2 中的某一个目标上了，然后呢？新创建的缓存对象的默认状态，相当于是不存在任何数据的一处缓存区域。如果想要将它实际使用起来，就必须向其中输入一些数据才行。

3.2.2 向缓存输入和输出数据

将数据输入和输出 OpenGL 缓存的方法有很多种。比如直接显式地传递数据，又比如用新的数据替换缓存对象中已有的部分数据，或者由 OpenGL 负责生成数据然后将它记录到缓存对象中。向缓存对象中传递数据最简单的方法就是在分配内存的时候读入数据。这一步可以通过 `glBufferData()` 函数来完成。下面再次给出 `glBufferData()` 的原型。

```
void glBufferData(GLenum target, GLsizeiptr size, const GLvoid* data, GLenum usage);
```

为绑定到 `target` 的缓存对象分配 `size` 大小（单位为字节）的存储空间。如果参数 `data` 不是 `NULL`，那么将使用 `data` 所在的内存区域的内容来初始化整个空间。`usage` 允许应用程序向 OpenGL 端发出一个提示，指示缓存中的数据可能具备一些特定的用途。

要特别注意的是，`glBufferData()` 是真正为缓存对象分配（或者重新分配）存储空间的。也就是说，如果新的数据大小比缓存对象当前所分配的存储空间要大，那么缓存对象的大小将被重设以获取更多空间。与之类似，如果新的数据大小比当前所分配的缓存要小，那么缓存对象将会收缩以适应新的大小。因此，虽然我们可以直接在初始化的时候指定缓存对象中的数据，但是这只是一种方便的用法而已，并不一定就是最好的方法（有的时候也不一定是最方便的用法）。

OpenGL 对于缓存对象存储数据中的最优分配方案的管理，并不仅仅依赖于初始化绑定时的 `target` 参数。另一个重要的参数就是 `glBufferData()` 中的 `usage`。`usage` 必须是内置标准标识符中的一个，例如 `GL_STATIC_DRAW` 或者 `GL_DYNAMIC_COPY`。注意这里的标识符名称要分解为两个部分去理解：第一部分可以是 `STATIC`、`DYNAMIC` 或者 `STREAM` 中的一个，而第二部分可以是 `DRAW`、`READ` 或者 `COPY` 中的一个。

这些“分解”的标识符的具体含义如表 3-3 所示。

表 3-3 缓存用途标识符

“分解”的标识符	意义
<code>_STATIC_</code>	数据存储内容只写入一次，然后多次使用
<code>_DYNAMIC_</code>	数据存储内容会被反复写入和反复使用
<code>_STREAM_</code>	数据存储内容只写入一次，然后也不会被频繁使用
<code>_DRAW</code>	数据存储内容由应用程序负责写入，并且作为 OpenGL 绘制和图像命令的数据源
<code>_READ</code>	数据存储内容通过 OpenGL 反馈的数据写入，然后在应用程序进行查询时返回这些数据
<code>_COPY</code>	数据存储内容通过 OpenGL 反馈的数据写入，并且作为 OpenGL 绘制和图像命令的数据源

如何为 `usage` 参数提供一个准确的定义，这关系到能否达到最优的性能。这个参数向 OpenGL 提供了重要的缓存使用策略信息。首先考虑相关标识符的第一部分。如果标识符

使用 `_STATIC_` 开头，那么就是说数据的变动是非常有限的，或者根本就没有——因为它在本质上是静态数据。这类标识符显然需要用于那些只修改过一次就不再变动的数据类型。如果 `usage` 包含了 `_STATIC_`，那么 OpenGL 会在内部对数据重新进行处理，以保证它在内存中的布置更为合理，或者使用更为优化的数据格式。这一步操作的代价可能较大，但是由于数据已经是静态的，因此这一操作只需要执行一次，整体上还是非常理想的。

如果在 `usage` 中包含了 `_DYNAMIC_`，那么说明数据的变动是频繁的，而变动过程中对数据的使用也是频繁的。例如，如果有一个建模程序，它所使用的数据可能被用户所编辑，此时有必要用到这个标识符。这种时候，一个可能的情况是数据在多帧内被持续使用，然后被修改，然后再次被更多帧使用，如此反复。这种情况的相反面就是 `GL_STREAM_` 标识符。它的含义是，缓存数据的修改是有规律的，并且每次修改数据后只会少量地加以使用（可能只使用一次）。这种时候，OpenGL 甚至可能不会将数据拷贝到快速的图像内存中，而是直接在原地进行访问。这种情形通常发生在 CPU 端执行应用程序诸如物理仿真的操作时，此时每帧都会给出一些新的数据集，供程序调取。

现在我们要了解 `usage` 标识符的第二部分。这一部分指示更新和使用数据的责任者。如果这个标识符包含 `_DRAW`，那么就是说这处缓存将作为标准 OpenGL 绘制操作的数据源使用。它会被频繁地读取；而与之相反的就是在标识符中包含 `_READ`，这类标识符会被频繁地写入。如果应用程序需要从缓存中回读数据（参见“访问缓存内容”一节），那么应当使用 `_READ` 标识符，这样 OpenGL 会认为这处数据是需要多次写入的。如果缓存中保存的是顶点数据，那么 `usage` 参数中必须包含 `_DRAW`；而像素缓存对象（pixel buffer object）和其他从 OpenGL 端获取数据的缓存则必须使用含有 `_READ` 的标识符。最后，如果 `usage` 中包含 `_COPY`，那么说明应用程序会通过 OpenGL 端来生成数据并且保存到缓存中，然后将它作为后继的绘制操作的输入源。使用 `_COPY` 标识符的一个相应例子就是 transform feedback 缓存，这个缓存需要由 OpenGL 写入数据，然后在之后的绘制命令中再作为顶点缓存使用。

缓存的部分初始化

假设有一个包含部分顶点数据的数组，另一个数组则包含一部分颜色信息，还有一个数组包含纹理坐标或者别的什么数据。你需要将这些数据进行紧凑的打包，并且存入一个足够大的缓存对象让 OpenGL 使用。在内存中数组之间可能是连续的，也可能不连续，因此无法使用 `glBufferData()` 一次性地更新所有的数据。此外，如果使用 `glBufferData()` 进行更新的话，那么首先是顶点数据，然后缓存的大小与顶点数据的大小完全一致，并且也就不再有空间去存储颜色或者纹理坐标信息了。因此我们需要引入新的 `glBufferSubData()` 函数。

```
void glBufferSubData(GLenum target, GLintptr offset, GLsizeiptr size, const GLvoid* data);
```

使用新的数据替换缓存对象中的部分数据。绑定到 `target` 的缓存对象要从 `offset` 字节处开始需要使用地址为 `data`、大小为 `size` 的数据块来进行更新。如果 `offset` 和 `size` 的总和超出了缓存对象绑定数据的范围，那么将产生一个错误。

如果将 `glBufferData()` 和 `glBufferSubData()` 结合起来使用,那么我们就可以对一个缓存对象进行分配和初始化,然后将数据更新到它的不同区块当中。一个相应的示例可以参见例 3.1。

例 3.1 使用 `glBufferSubData()` 来初始化缓存对象

```
// 顶点位置
static const GLfloat positions[] =
{
    -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
};

// 顶点颜色
static const GLfloat colors[] =
{
    1.0f, 0.0f, 0.0f,
    0.0f, 1.0f, 0.0f,
    0.0f, 0.0f, 1.0f,
    1.0f, 1.0f, 1.0f,
};

// 缓存对象
GLuint buffer;

// 为缓存对象生成一个名称
glGenBuffers(1, &buffer);
// 将它绑定到 GL_ARRAY_BUFFER 目标
glBindBuffer(GL_ARRAY_BUFFER, buffer);
// 分配足够的空间 (sizeof(positions) + sizeof(colors))
glBufferData(GL_ARRAY_BUFFER,                // 目标
             sizeof(positions) + sizeof(colors), // 总计大小
             NULL,                             // 无数据
             GL_STATIC_DRAW);                  // 用途
// 将位置信息放置在缓存的偏移地址为 0 的位置
glBufferSubData(GL_ARRAY_BUFFER,             // 目标
                0,                           // 偏移地址
                sizeof(positions),           // 大小
                positions);                   // 数据
// 放置在缓存中的颜色信息的偏移地址为当前填充大小值的位置,也就是 sizeof(positions)
glBufferSubData(GL_ARRAY_BUFFER,             // 目标
                sizeof(positions),           // 偏移地址
                sizeof(colors),             // 大小
                colors);                     // 数据
// 现在位置信息位于偏移 0,而颜色信息保存在同一缓存中,紧随其后
```

如果只是希望将缓存对象的数据清除为一个已知的值,那么也可以使用 `glClearBufferData()` 或者 `glClearBufferSubData()` 函数。它们的原型如下所示:

```
void glClearBufferData(GLenum target, GLenum internalformat, GLenum format,
GLenum type, const void* data);
```

```
void glClearBufferSubData(GLenum target, GLenum internalformat, GLintptr offset,
GLintptr size, GLenum format, GLenum type, const void* data);
```

清除缓存对象中所有或者部分数据。绑定到 `target` 的缓存存储空间将使用 `data` 中存储的数据进行填充。`format` 和 `type` 分别指定了 `data` 对应数据的格式和类型。

首先将数据被转换到 `internalformat` 所指定的格式，然后填充缓存数据的指定区域范围。

对于 `glClearBufferData()` 来说，整个区域都会被指定的数据所填充。而对于 `glClearBufferSubData()` 来说，填充区域是通过 `offset` 和 `size` 来指定的，它们分别给出了以字节为单位的起始偏移地址和大小。

`glClearBufferData()` 和 `glClearBufferSubData()` 函数允许我们初始化缓存对象中存储的数据，并且不需要保留或者清除任何一处系统内存。

缓存对象中的数据也可以使用 `glCopyBufferSubData()` 函数互相进行拷贝。与 `glBufferSubData()` 函数对较大缓存中的数据依次进行组装的做法不同，此时我们可以使用 `glBufferData()` 将数据更新到独立的缓存当中，然后将这些缓存直接用 `glCopyBufferSubData()` 拷贝到一个较大的缓存中。根据 OpenGL 的具体实现，这些拷贝之间还可以存在互相重叠的部分，因为每次调用 `glBufferData()` 时，缓存对象都会将当前区域的内容标记为需要更新的状态。因此，有的时候我们可以让 OpenGL 直接分配一整块数据缓存区域，并且不用关心之前的数据拷贝操作是否已经完成了。旧的数据会在之后的某个时刻直接释放。

`glCopyBufferSubData()` 的原型如下所示：

```
void glCopyBufferSubData(GLenum readtarget, GLenum writetarget, GLintptr
readoffset, GLintptr writeoffset, GLsizeiptr size);
```

将绑定到 `readtarget` 的缓存对象的一部分存储数据拷贝到与 `writetarget` 相绑定的缓存对象的数据区域上。`readtarget` 对应的数据从 `readoffset` 位置开始复制 `size` 个字节，然后拷贝到 `writetarget` 对应数据的 `writeoffset` 位置。如果 `readoffset` 或者 `writeoffset` 与 `size` 的和超出了绑定的缓存对象的范围，那么 OpenGL 会产生一个 `GL_INVALID_VALUE` 错误。

`glCopyBufferSubData()` 可以在两个目标对应的缓存之间拷贝数据，而 `GL_COPY_READ_BUFFER` 和 `GL_COPY_WRITE_BUFFER` 这两个目标正是为了这个目的而生。它们不能用于其他 OpenGL 的操作当中，并且如果将缓存与它们进行绑定，并且只用于数据的拷贝和存储目的，不影响 OpenGL 的状态也不需要记录拷贝之前的目标区域信息的话，那么整个操作过程都是可以保证安全的。

读取缓存的内容

我们可以通过多种方式从缓存对象中回读数据。第一种方式就是使用 `glGetBufferSubData()` 函数。这个函数可以从绑定到某个目标的缓存中回读数据，然后将它放置到应用程序保有的一处内存当中。`glGetBufferSubData()` 的原型如下所示：

```
void glGetBufferSubData(GLenum target, GLintptr offset, GLsizeiptr size, GLvoid* data);
```

返回当前绑定到 `target` 的缓存对象中的部分或者全部数据。起始数据的偏移字节位置为 `offset`，回读的数据大小为 `size` 个字节，它们将从缓存的数据区域拷贝到 `data` 所指向的内存区域中。如果缓存对象当前已经被映射，或者 `offset` 和 `size` 的和超出了缓存对象数据区域的范围，那么将提示一个错误。

如果我们使用 OpenGL 生成了一些数据，然后希望重新获取到它们的内容，那么此时应该使用 `glGetBufferSubData()`。这样的例子包括在 GPU 级别使用 transform feedback 处理顶点数据，以及将帧缓存或者纹理数据读取到像素缓存对象（Pixel Buffer Object）中。后文将依次给出这些内容的具体介绍。当然，我们也可以使用 `glGetBufferSubData()` 简单地将之前存入到缓存对象中的数据读回到内存中。

3.2.3 访问缓存的内容

目前为止，在这一节当中给出的所有函数（`glBufferData()`、`glBufferSubData()`、`glCopyBufferSubData()` 和 `glGetBufferSubData()`）都存在同一个问题，就是它们都会导致 OpenGL 进行一次数据的拷贝操作。`glBufferData()` 和 `glBufferSubData()` 会将应用程序内存中的数据拷贝到 OpenGL 管理的内存当中。显而易见 `glCopyBufferSubData()` 会将源缓存中的内容进行一次拷贝。`glGetBufferSubData()` 则是将 OpenGL 管理的内存中的数据拷贝到应用程序内存中。根据硬件的配置，其实也可以通过获取一个指针的形式，直接在应用程序中对 OpenGL 管理的内存进行访问。当然，获取这个指针的对应函数就是 `glMapBuffer()`。

```
void* glMapBuffer(GLenum target, GLenum access);
```


将当前绑定到 `target` 的缓存对象的整个数据区域映射到客户端的地址空间中。之后可以根据给定的 `access` 策略，通过返回的指针对数据进行直接读或者写的操作。如果 OpenGL 无法将缓存对象的数据映射出来，那么 `glMapBuffer()` 将产生一个错误并且返回 `NULL`。发生这种情况的原因可能是与系统相关的，比如可用的虚拟内存过低等。

当我们调用 `glMapBuffer()` 时，这个函数会返回一个指针，它指向绑定到 `target` 的缓存对象的数据区域所对应的内存。注意这块内存只是对应于这个缓存对象本身——它不一定是图形处理器用到的内存区域。`access` 参数指定了应用程序对于映射后的内存区域的使用方式。它必须是表 3-4 中列出的标识符之一。

表 3-4 glMapBuffer() 的访问模式

标识符	意义
GL_READ_ONLY	应用程序仅对 OpenGL 映射的内存区域执行读操作
GL_WRITE_ONLY	应用程序仅对 OpenGL 映射的内存区域执行写操作
GL_READ_WRITE	应用程序对 OpenGL 映射的内存区域可能执行读或者写的操作

如果 glMapBuffer() 无法映射缓存对象的数据，那么它将返回 NULL。access 参数相当于用户程序与 OpenGL 对内存访问的一个约定。如果用户违反了这个约定，那么将产生很不好的结果，例如写缓存的操作将被忽略，数据将被破坏，甚至用户程序会直接崩溃[Ⓐ]。

 **注意** 当你要求映射到应用程序层面的数据正处于无法访问的内存当中，OpenGL 可能会被迫将数据进行移动，以保证能够获取到数据的指针，也就是你期望的结果。与之类似，当你完成了对数据的操作，以及对它进行了修改，那么 OpenGL 将再次把数据移回到图形处理器所需的位置上。这样的操作对于性能上的损耗是比较高的，因此必须特别加以对待。

如果缓存已经通过 GL_READ_ONLY 或者 GL_READ_WRITE 访问模式进行了映射，那么缓存对象中的数据对于应用程序就是可见的。我们可以回读它的内容，将它写入磁盘文件，甚至直接对它进行修改（如果使用了 GL_READ_WRITE 作为访问模式的话）。如果访问模式为 GL_READ_WRITE 或者 GL_WRITE_ONLY，那么可以通过 OpenGL 返回的指针向映射内存中写入数据。当结束数据的读取或者写入到缓存对象的操作之后，必须使用 glUnmapBuffer() 执行解除映射操作，它的原型如下所示：

GLboolean glUnmapBuffer(GLenum target);

解除 glMapBuffer() 创建的映射。如果对象数据的内容在映射过程中没有发生损坏，那么 glUnmapBuffer() 将返回 GL_TRUE。发生损坏的原因通常与系统相关，例如屏幕模式发生了改变，这会影响图形内存的可用性。这种情况下，函数的返回值为 GL_FALSE，并且对应的数据内容是不可预测的。应用程序必须考虑到这种几率较低的情形，并且及时对数据进行重新初始化。

如果解除了缓存的映射，那么之前写入到 OpenGL 映射内存中的数据将会重新对缓存对象可见。这句话的意义是，我们可以先使用 glBufferData() 分配数据空间，并且在 data 参数中直接传递 NULL，之后进行映射并且直接将数据写入，最后解除映射，从而完成了数据向缓存对象传递的操作。例 3.2 所示就是一个将文件内容读取并写入到缓存对象的例子。

Ⓐ 遗憾的是，很多应用程序都会破坏这样的约定，而大部分 OpenGL 的实现都会假设用户其实并不知道如何正确调用 glMapBuffer()，因此直接将访问模式参数重新设定为 GL_READ_WRITE，因此这些程序还是可以正常工作的。

例 3.2 使用 glMapBuffer() 初始化缓存对象

```

GLuint buffer;
FILE * f;
size_t filesize;

// 打开文件并确定它的大小
f = fopen("data.dat", "rb");
fseek(f, 0, SEEK_END);
filesize = ftell(f);
fseek(f, 0, SEEK_SET);

// 生成缓存名字并将它绑定到缓存绑定点上——这里是
// GL_COPY_WRITE_BUFFER (在这里这个绑定并没有实际意义),
// 这样就可以创建缓存了
glGenBuffers(1, &buffer);
glBindBuffer(GL_COPY_WRITE_BUFFER, buffer);

// 分配缓存中存储的数据空间, 向 data 参数传入 NULL 即可

glBufferData(GL_COPY_WRITE_BUFFER, (GLsizei)filesize, NULL,
             GL_STATIC_DRAW);

// 映射缓存……
void * data = glMapBuffer(GL_COPY_WRITE_BUFFER, GL_WRITE_ONLY);

// 将文件读入缓存
fread(data, 1, filesize, f);

// 好了, 现在我们已经完成了实验, 可以解除缓存映射并关闭文件了
glUnmapBuffer(GL_COPY_WRITE_BUFFER);
fclose(f);

```

在例 3.2 中, 文件的所有内容都在单一操作中被读入到缓存对象当中。缓存对象创建时的大小与文件是相同的。当缓存映射之后, 我们就可以直接将文件内容读入到缓存对象的数据区域当中。应用程序端并没有拷贝的操作, 并且如果数据对于应用程序和图形处理器都是可见的, 那么 OpenGL 端也没有进行任何拷贝的操作。

使用这种方式来初始化缓存对象可能会带来显著的性能优势。其理由如下: 如果调用 `glBufferData()` 或者 `glBufferSubData()`, 当返回这些函数后, 我们可以对返回的内存区域中的数据进行任何操作——释放它, 使用它做别的事情——都是可以的。这也就是说, 这些函数在完成后不能与内存区域再有任何瓜葛, 因此必须采取数据拷贝的方式。但是, 如果调用 `glMapBuffer()`, 它所返回的指针是 OpenGL 端管理的。当调用 `glUnmapBuffer()` 时, OpenGL 依然负责管理这处内存, 而用户程序与这处内存已经不再有瓜葛了。这样的话即使数据需要移动或者拷贝, OpenGL 都可以在调用 `glUnmapBuffer()` 之后才开始这些操作并且立即返回, 而内容操作是在系统的空闲时间之内完成, 不再受到应用程序的影响。因此, OpenGL 的数据拷贝操作与应用程序之后的操作 (例如建立更多的缓存, 读取别的文件, 等等) 实际上是同步进行的。如果不需要进行拷贝的话, 那么结果就再好不过了! 此时在本质上解除映射的操作相当于是对空间的释放。

异步和显式的映射

为了避免 `glMapBuffer()` 可能造成的缓存映射问题（例如应用程序错误地指定了 `access` 参数，或者总是使用 `GL_READ_WRITE`），`glMapBufferRange()` 函数使用额外的标识符来更精确地设置访问模式，`glMapBufferRange()` 函数的原型如下所示：

```
void* glMapBufferRange(GLenum target, GLintptr offset, GLsizeiptr length, GLbitfield access);
```

将缓存对象数据的全部或者一部分映射到应用程序的地址空间中。`target` 设置了缓存对象当前绑定的目标。`offset` 和 `length` 一起设置了准备映射的数据范围（单位为字节）。`access` 是一个位域标识符，用于描述映射的模式。

对于 `glMapBufferRange()` 来说，`access` 位域中必须包含 `GL_MAP_READ_BIT` 和 `GL_MAP_WRITE_BIT` 中的一个或者两个，以确认应用程序是否要对映射数据进行读操作、写操作，或者两者皆有。此外，`access` 中还可以包含一个或多个其他的标识符，如表 3-5 所示。

表 3-5 `glMapBufferRange()` 中使用的标识符

标识符	意义
<code>GL_MAP_INVALIDATE_RANGE_BIT</code>	如果设置的话，给定的缓存区域内任何数据都可以被抛弃以及无效化。如果给定区域范围内任何数据没有被随后重新写入的话，那么它将变成未定义的数据。这个标识符无法与 <code>GL_MAP_READ_BIT</code> 同时使用
<code>GL_MAP_INVALIDATE_BUFFER_BIT</code>	如果设置的话，缓存的整个内容都可以被抛弃和无效化，不再受到区域范围的设置影响。所有映射范围之外的数据都会变成未定义的状态，而如果范围内的数据没有被随后重新写入的话，那么它也会变成未定义。这个标识符无法与 <code>GL_MAP_READ_BIT</code> 同时使用
<code>GL_MAP_FLUSH_EXPLICIT_BIT</code>	应用程序将负责通知 OpenGL 映射范围内的哪个部分包含了可用数据，方法是在调用 <code>glUnmapBuffer()</code> 之前调用 <code>glFlushMappedBufferRange()</code> 。如果缓存中较大范围内的数据都会被映射，而并不是全部被应用程序写入的话，应当使用这个标识符。这个位标识符必须与 <code>GL_MAP_WRITE_BIT</code> 结合使用。如果 <code>GL_MAP_FLUSH_EXPLICIT_BIT</code> 没有定义的话，那么 <code>glUnmapBuffer()</code> 会自动刷新整个映射区域的内容
<code>GL_MAP_UNSYNCHRONIZED_BIT</code>	如果这个位标识符没有设置的话，那么 OpenGL 会等待所有正在处理的缓存访问操作结束，然后再返回映射范围的内存。如果设置了这个标识符，那么 OpenGL 将不会尝试进行这样的缓存同步操作

正如你在表 3-5 中看到的这些标识符所提示的，对于 OpenGL 数据的使用以及数据访问时的同步操作，这个命令可以实现一个更精确的控制过程。

如果打算通过 `GL_MAP_INVALIDATE_RANGE_BIT` 或者 `GL_MAP_INVALIDATE_BUFFER_BIT` 标识符来实现缓存数据的无效化，那么也就意味着 OpenGL 可以对缓存对象中任何已有的数据进行清理。除非你确信自己要同时使用 `GL_MAP_WRITE_BIT` 标识符对

缓存进行写入操作，否则不要设置这两个标识符中的任意一个。如果你设置了 `GL_MAP_INVALIDATE_RANGE_BIT` 的话，你的目的应该是对某个区域的整体进行更新（或者至少是其中对你的程序有意义的部分）。如果设置了 `GL_MAP_INVALIDATE_BUFFER_BIT`，那么就意味着你不打算再关心那些没有被映射的缓存区域的内容了。无论是哪种方法，你都必须通过标识符的设置来声明你准备在后继的映射当中对缓存中剩下的部分进行更新[⊖]。由于此时 OpenGL 是可以抛弃缓存数据中剩余的部分，因此即使你将修改过的数据重新合并到原始缓存中也没有什么意义了。因此，如果打算对映射缓存的第一个部分使用 `GL_MAP_INVALIDATE_BUFFER_BIT`，然后对缓存其他的部分使用 `GL_MAP_INVALIDATE_RANGE_BIT`，那么应该是一个不错的想法。

`GL_MAP_UNSYNCHRONIZED_BIT` 标识符用于禁止 OpenGL 数据传输和使用时的自动同步机制。没有这个标志符的话，OpenGL 会在使用缓存对象之前完成任何正在执行的命令。这一步与 OpenGL 的管线有关，因此可能会造成性能上的损失。如果可以确保之后的操作可以在真正修改缓存内容之前完成（不过在调用 `glMapBufferRange()` 之前这并不是必须的），例如调用 `glFinish()` 或者使用一个同步对象（参见 11.3 节），那么 OpenGL 也就不需要专门为此维护一个同步功能了。

最后，`GL_MAP_FLUSH_EXPLICIT_BIT` 标识符表明了应用程序将通知 OpenGL 它修改了缓存的哪些部分，然后再调用 `glUnmapBuffer()`。通知的操作可以通过 `glFlushMappedBufferRange()` 函数的调用来完成，其原型如下：

```
void glFlushMappedBufferRange(GLenum target, GLintptr offset, GLsizeiptr length);
```

通知 OpenGL，绑定到 `target` 的映射缓存中由 `offset` 和 `length` 所划分的区域已经发生了修改，需要立即更新到缓存对象的数据区域中。

我们可以对缓存对象中独立的或者互相重叠的映射范围多次调用 `glFlushMappedBufferRange()`。缓存对象的范围是通过 `offset` 和 `length` 划分的，这两个值必须位于缓存对象的映射范围之内，并且映射范围必须通过 `glMapBufferRange()` 以及 `GL_MAP_FLUSH_EXPLICIT_BIT` 标识符来映射。当执行这个操作之后，会假设 OpenGL 对于映射缓存对象中指定区域的修改已经完成，并且开始执行一些相关的操作，例如重新激活数据的可用性，将它拷贝到图形处理器的显示内存中，或者进行刷新，数据缓存的重新更新等。就算缓存的一部分或者全部还处于映射状态下，这些操作也可以顺利完成。这一操作对于 OpenGL 与其他应用程序操作的并行化处理是非常有意义的。举例来说，如果需要从文件加载一个非常庞大的数据块并将他们送入缓存，那么需要在缓存中分配足够囊括整个文件大小的区域，然后读取文件的各个子块，并且对每个子块都调用一次

⊖ 不要对每个区域都设置 `GL_MAP_INVALIDATE_BUFFER_BIT`，否则只有最后一个映射区域中的数据才是有效的！

`glFlushMappedBufferRange()`。然后 OpenGL 就可以与应用程序并行地执行一些工作，从文件读取更多的数据并且存入下一个子块当中。

通过这些标识符的不同混合方式，我们可以对应用程序和 OpenGL 之间的数据传输过程进行优化，或者实现一些高级的技巧，例如多线程或者异步的文件操作。

3.2.4 丢弃缓存数据

高级技巧

如果已经完成了对缓存数据的处理，那么可以直接通知 OpenGL 我们不再需要使用这些数据。例如，如果我们正在向 transform feedback 的缓存中写入数据，然后使用这些数据进行绘制。如果最后访问数据的是绘制命令，那么我们就可以及时通知 OpenGL，让它适时地抛弃数据并且将内存用作其他用途。这样 OpenGL 的实现就可以完成一些优化工作，诸如紧密的内存分配策略，或者避免系统与多个 GPU 之间产生代价高昂的拷贝操作。

如果要抛弃缓存对象中的部分或者全部数据，那么我们可以调用 `glInvalidateBufferData()` 或者 `glInvalidateBufferSubData()` 函数。这两个函数的原型如下所示：

```
void glInvalidateBufferData(GLuint buffer);  
void glInvalidateBufferSubData(GLuint buffer, GLintptr offset, GLsizeiptr length);
```

通知 OpenGL，应用程序已经完成对缓存对象中给定范围内容的操作，因此可以随时根据实际情况抛弃数据。`glInvalidateBufferSubData()` 会抛弃名称为 `buffer` 的缓存对象中，从 `offset` 字节处开始共 `length` 字节的数据。`glInvalidateBufferData()` 会直接抛弃整个缓存的数据内容。

注意，从理论上来说，如果调用 `glBufferData()` 并且传入一个 NULL 指针的话，那么所实现的功能与直接调用 `glInvalidateBufferData()` 是非常相似的。这两个方法都会通知 OpenGL 实现可以安全地抛弃缓存中的数据。但是，从逻辑上 `glBufferData()` 会重新分配内存区域，而 `glInvalidateBufferData()` 不会。根据 OpenGL 的具体实现，通常调用 `glInvalidateBufferData()` 的方法会更为优化一些。此外，`glInvalidateBufferSubData()` 也是唯一一个可以抛弃缓存对象中的区域数据的方法。

3.3 顶点规范

现在我们已经在缓存中存储了数据，并且知道如何编写一个基本的顶点着色器，因此我们有必要将数据传递到着色器当中。我们已经了解顶点数组对象（vertex array object）的概念，它包含数据的位置和布局信息，以及类似 `glVertexAttribPointer()` 的一系列函数。现

在，我们将更深入地了解顶点规范的相关内容、`glVertexAttribPointer()` 的其他变种函数，以及如何设置一些非浮点数或者还没有启用的顶点属性数据。

3.3.1 深入讨论 VertexAttribPointer

我们已经在第1章里简要地介绍过 `glVertexAttribPointer()` 命令。它的原型如下所示：

```
void glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean
normalized, GLsizei stride, const GLvoid* pointer);
```

设置顶点属性在 `index` 位置可访问的数据值。`pointer` 的起始位置也就是数组中的第一组数据值，它是以基本计算机单位（例如字节）度量的，由绑定到 `GL_ARRAY_BUFFER` 目标的缓存对象中的地址偏移量确定的。`size` 表示每个顶点中需要更新的元素个数。`type` 表示数组中每个元素的数据类型。`normalized` 表示顶点数据是否需要在传递到顶点数组之前进行归一化处理。`stride` 表示数组中两个连续元素之间的偏移字节数。如果 `stride` 为 0，那么在内存当中各个数据就是紧密贴合的。

`glVertexAttribPointer()` 所设置的状态会保存到当前绑定的顶点数组对象（VAO）中。`size` 表示属性向量的元素个数（1、2、3、4），或者是一个特殊的标识符 `GL_BGRA`，它专用于压缩顶点数据的格式设置。`type` 参数设置了缓存对象中存储的数据类型。表 3-6 所示就是 `type` 中可以指定的标识符名称，以及对应的 OpenGL 数据类型。

表 3-6 `glVertexAttribPointer()` 的数据类型标识符

标识符	OpenGL 类型
<code>GL_BYTE</code>	<code>GLbyte</code> （有符号 8 位整型）
<code>GL_UNSIGNED_BYTE</code>	<code>GLubyte</code> （无符号 8 位整型）
<code>GL_SHORT</code>	<code>GLshort</code> （有符号 16 位整型）
<code>GL_UNSIGNED_SHORT</code>	<code>GLushort</code> （无符号 16 位整型）
<code>GL_INT</code>	<code>GLint</code> （有符号 32 位整型）
<code>GL_UNSIGNED_INT</code>	<code>GLuint</code> （无符号 32 位整型）
<code>GL_FIXED</code>	<code>GLfixed</code> （有符号 16 位定点型）
<code>GL_FLOAT</code>	<code>GLfloat</code> （32 位 IEEE 单精度浮点型）
<code>GL_HALF_FLOAT</code>	<code>GLhalf</code> （16 位 S1E5M10 半精度浮点型）
<code>GL_DOUBLE</code>	<code>GLdouble</code> （64 位 IEEE 双精度浮点型）
<code>GL_INT_2_10_10_10_REV</code>	<code>GLuint</code> （压缩数据类型）
<code>GL_UNSIGNED_INT_2_10_10_10_REV</code>	<code>GLuint</code> （压缩数据类型）

注意，如果在 `type` 中传入了 `GL_SHORT` 或者 `GL_UNSIGNED_INT` 这样的整数类型，那么 OpenGL 只能将这些数据类型存储到缓存对象的内存中。OpenGL 必须将这些数据转换为浮点数才可以将它们读取到浮点数的顶点属性中。执行这一转换过程可以通过 `normalize`

参数来控制。如果 `normalize` 为 `GL_FALSE`，那么整数将直接被强制转换为浮点数的形式，然后再传入到顶点着色器中。换句话说，如果将一个整数 4 置入缓存，设置 `type` 为 `GL_INT`，而 `normalize` 为 `GL_FALSE`，那么着色器中传入的值就是 4.0。如果 `normalize` 为 `GL_TRUE`，那么数据在传入到顶点着色器之前需要首先进行归一化。为此，OpenGL 会使用一个固定的依赖于输入数据类型的常数去除每个元素。如果数据类型是有符号的，那么相应的计算公式如下：

$$f = \frac{c}{2^b - 1}$$

如果数据类型是无符号的，那么相应的计算公式如下：

$$f = \frac{2c + 1}{2^b - 1}$$

这两个公式当中， f 的结果就是浮点数值， c 表示输入的整数分量， b 表示数据类型的位数（例如 `GL_UNSIGNED_BYTE` 就是 8，`GL_SHORT` 就是 16，以此类推）。注意，无符号数据类型在除以类型相关的常数之前，还需要进行缩放和偏移操作。之前我们向整数顶点属性中传入 4 作为示例，那么这里我们将得到：

$$f = \frac{4}{2^{32} - 1}$$

它的结果相当于 0.000000009313——这是一个非常小的数字！

整型顶点属性

如果你对浮点数值的工作方式比较熟悉的话，那么你应该也知道如果它的数值很大的时候，会造成精度的丢失，因此大范围的整数值不能直接使用浮点型属性传入顶点着色器中。因此，我们需要引入整型顶点属性。它们在顶点着色器中的表示方法为 `int`、`ivec2`、`ivec3` 以及 `ivec4`，当然也有无符号的表现形式，即 `uint`、`uvec2`、`uvec3` 以及 `uvec4`。

我们需要用到另一个顶点属性的函数将整数传递到顶点属性中，它不会执行自动转换到浮点数的操作。这个函数叫做 `glVertexAttribPointer()`，其中 `I` 表示整型的意思。

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

与 `glVertexAttribPointer()` 类似，不过它专用于向顶点着色器中传递整型的顶点属性。`type` 必须是整型数据类型的一种，包括 `GL_BYTE`、`GL_UNSIGNED_BYTE`、`GL_SHORT`、`GL_UNSIGNED_SHORT`、`GL_INT`，以及 `GL_UNSIGNED_INT`。

注意，`glVertexAttribPointer()` 的参数与 `glVertexAttribPointer()` 是完全等价的，只是不再需要 `normalize` 参数。这是因为 `normalize` 对于整型顶点属性来说是没有意义的。这里的 `type` 参数只能使用 `GL_BYTE`、`GL_UNSIGNED_BYTE`、`GL_SHORT`、`GL_UNSIGNED_`

SHORT、GL_INT，以及 GL_UNSIGNED_INT 这些标识符。

双精度顶点属性

glVertexAttribPointer() 的第三个变化就是 glVertexAttribLPointer()——这里的 L 表示“long”。这个函数专门用于将属性数据加载到 64 位的双精度浮点型顶点属性中。

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

与 glVertexAttribPointer() 类似，不过对于传入顶点着色器的 64 位的双精度浮点型顶点属性来说，type 必须设置为 GL_DOUBLE。

这里再次说明，normalize 参数依然是不需要的。glVertexAttribPointer() 中的 normalize 只是用来处理那些不适宜直接使用的整型类型，因此在这里它并不是必须的。如果 glVertexAttribPointer() 函数也使用了 GL_DOUBLE 类型，那么实际上数据在传递到顶点着色器之前会被自动转换到 32 位单精度浮点型方式——即使我们的目标顶点属性已经声明为双精度类型，例如 double、dvec2、dvec3、dvec4，或者双精度的矩阵类型，例如 dmat4。但是，glVertexAttribLPointer() 可以保证输入数据的完整精度，并且将它们直接传递到顶点着色器阶段。

顶点属性的压缩数据格式

回到 glVertexAttribPointer() 命令，之前已经提及，size 参数的可选值包括 1、2、3、4，以及一个特殊的标识符 GL_BGRA。此外，type 参数也可以使用某些特殊的数值，即 GL_INT_2_10_10_10_REV 或者 GL_UNSIGNED_INT_2_10_10_10_REV，它们都对应于 GLuint 数据类型。这些特殊的标识符可以用来表达 OpenGL 支持的压缩数据格式。GL_INT_2_10_10_10_REV 和 GL_UNSIGNED_INT_2_10_10_10_REV 标识符表示了一种有四个分量的数据格式，前三个分量均占据 10 个字节，第四个分量占据 2 个字节，这样压缩后的大小是一个 32 位单精度数据（GLuint）。GL_BGRA 可以被简单地视为 GL_ZYXW 的格式[⊖]。根据 32 位字符类型的数据布局方式，我们可以得到如图 3-3 的数据划分方式。

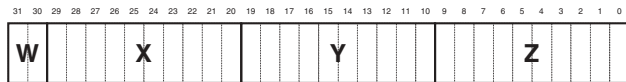


图 3-3 BGRA 格式的压缩顶点属性数据元素分布

图 3-3 中，顶点元素分布在一个 32 位单精度整数中，顺序为 w、x、y、z——反转之后就是 z、y、x、w，或者用颜色分量来表示就是 b、g、r、a。图 3-4 中，各个分量的压缩顺

⊖ 这不是一个真的 OpenGL 标识符，只是为了更好地解释这个问题。

序为 w、z、y、x，反转并写作颜色分量的形式就是 r、g、b、a。

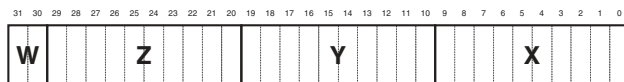


图 3-4 RGBA 格式的压缩顶点属性数据元素分布

顶点数据可以使用 GL_INT_2_10_10_10_REV 或者 GL_UNSIGNED_INT_2_10_10_10_REV 这两种格式中的一种来设置。如果 glVertexAttribPointer() 的 type 参数设置为其中一种标识符，那么顶点数组中的每个顶点都会占据 32 位。这个数据会被分解为各个分量然后根据需要进行归一化（根据 normalize 参数的设置），最后被传递到对应的顶点属性当中。这种数据的排布方式对于法线等类型的属性设置特别有益处，三个主要分量的大小均为 10 位，因此精度可以得到额外的提高，并且此时通常不需要达到半浮点数的精度级别（每个分量占据 16 位）。这样节约了内存空间和系统带款，因此有助于提升程序性能。

3.3.2 静态顶点属性的规范

在第 1 章里，我们已经了解了 glEnableVertexAttribArray() 和 glDisableVertexAttribArray() 函数。

这些函数用来通知 OpenGL，顶点缓存中记录了哪些顶点属性数据。在 OpenGL 从顶点缓存中读取数据之前，我们必须使用 glEnableVertexAttribArray() 启用对应的顶点属性数组。如果某个顶点属性对应的属性数组没有启用的话，会发生什么事情呢？此时，OpenGL 会使用静态顶点属性。每个顶点的静态顶点属性都是一个默认值，如果某个属性没有启用任何属性数组的话，就会用到这个默认值。举例来说，我们的顶点着色器中可能需要从某个顶点属性中读取顶点的颜色值。如果某个模型中所有的顶点或者一部分顶点的颜色值是相同的，那么我们使用一个常数值来填充模型中所有顶点的数据缓存，这无疑是一种内存浪费和性能损失。因此，这里可以禁止顶点属性数组，并且使用静态的顶点属性值来设置所有顶点的颜色。

每个属性的静态顶点属性可以通过 glVertexAttrib*() 系列函数来设置。如果顶点属性在顶点着色器中是一个浮点型的变量（例如 float、vec2、vec3、vec4 或者浮点型矩阵类型，例如 mat4），那么我们就可以使用下面的 glVertexAttrib*() 来设置它的数值。

```
void glVertexAttrib{1234}{fdfs}(GLuint index, TYPE values);
void glVertexAttrib{1234}{fdfs}v(GLuint index, const TYPE* values);
void glVertexAttrib4{bsifd ub us ui}v(GLuint index, const TYPE* values);
```

设置索引为 index 的顶点属性的静态值。如果函数名称末尾没有 v，那么最多可以指定 4 个参数值，即 x、y、z、w 参数。如果函数末尾有 v，那么最多有 4 个参数值是保存在一个数组中传入的，它的地址通过 values 来指定，存储顺序依次为 x、y、z 和 w 分量。

所有这些函数都会自动将输入参数转换为浮点数（除非它们本来就是浮点数形式），然后再传递到顶点着色器中。这里的转换就是简单的强制类型转换。也就是说，输入的数值被转换为浮点数的过程，与缓存中的数据通过 `glVertexAttribPointer()` 并设置 `normalize` 参数为 `GL_FALSE` 的转换过程是一样的。对于函数中需要传入整型数值的情况，我们也可以使用另外的函数，将数据归一化到 `[0, 1]` 或者 `[-1, 1]` 的范围内，其依据是输入参数是否有符号（或者无符号）类型。这些函数的声明为：

```
void glVertexAttrib4Nub(GLuint index, GLubyte x, GLubyte y, GLubyte z, GLubyte w);
void glVertexAttrib4N{bsi ub us ui}v(GLuint index, const TYPE* v);
```

设置属性 `index` 所对应的一个或者多个顶点属性值，并且在转换过程中将无符号参数归一化到 `[0, 1]` 的范围，将有符号参数归一化到 `[-1, 1]` 的范围。

即使使用了这些函数，输入参数依然会转换为浮点数的形式，然后再传递给顶点着色器。因此他们只能用来设置单精度浮点数类型的静态属性数据。如果顶点属性变量必须声明为整数或者双精度浮点数的话，那么应该使用下面的函数形式：

```
void glVertexAttribI{1234}{i ui}(GLuint index, TYPE values);
void glVertexAttribI{123}{i ui}v(GLuint index, const TYPE* values);
void glVertexAttribI4{bsi ub us ui}v(GLuint index, const TYPE* values);
```

设置一个或者多个静态整型顶点属性值，以用于 `index` 位置的整型顶点属性。

此外，如果顶点属性声明为双精度浮点数类型，那么应该使用带有 `L` 字符的 `glVertexAttrib*`() 函数，也就是：

```
void glVertexAttribL{1234}(GLuint index, TYPE values);
void glVertexAttribL{1234}v(GLuint index, const TYPE* values);
```

设置一个或者多个静态顶点属性值，以用于 `index` 位置的双精度顶点属性。

`glVertexAttribI*()` 和 `glVertexAttribL*()` 系列函数都是 `glVertexAttrib*()` 的变种，它们将参数到传递顶点属性的过程与 `glVertexAttribPointer()` 等函数的实现过程是一样的。

如果你使用了某个 `glVertexAttrib*()` 函数，但是传递给顶点属性的分量个数不足的话（例如使用 `glVertexAttrib*()` 的 `2f` 形式，所设置的顶点属性实际上声明为 `vec4`），那么缺少的分量中将自动填充为默认的值。对于 `w` 分量，默认值为 `1.0`，而 `y` 和 `z` 分量的默认值为 `0.0`[⊖]。如果函数中包含的分量个数多于着色器中顶点属性的声明个数，那么多余的分量会被简单地抛弃处理。

⊖ 我们故意没有设置 `x` 分量的默认值——因为当我们设置 `y`、`z`、`w` 的属性值的时候，不可能漏过 `x` 的值。



注意

静态顶点属性值是保存在当前 VAO 当中的，而不是程序对象。这也就意味着，如果当前的顶点着色器中存在一个 `vec3` 的输入属性，而我们使用 `glVertexAttrib*()` 的 `4fv` 形式设置了一个四分量的向量给它，那么第四个分量值虽然会被忽略，但是依然被保存了。如果改变顶点着色器的内容，重新设置当前属性为 `vec4` 的输入形式，那么之前设置的第四个分量值就会出现在属性 `w` 分量当中了。

3.4 OpenGL 的绘制命令

大部分 OpenGL 绘制命令都是以 `Draw` 这个单词开始的[⊖]。绘制命令大致可以分为两个部分：索引形式和非索引形式的绘制。索引形式的绘制需要用到绑定 `GL_ELEMENT_ARRAY_BUFFER` 的缓存对象中存储的索引数组，它可以用来间接地对已经启用的顶点数组进行索引。另一方面，非索引的绘制不需要使用 `GL_ELEMENT_ARRAY_BUFFER`，只需要简单地按顺序读取顶点数据即可。OpenGL 当中，最基本的非索引形式的绘制命令就是 `glDrawArrays()`。

```
void glDrawArrays(GLenum mode, GLint first, GLsizei count);
```

使用数组元素建立连续的几何图元序列，每个启用的数组中起始位置为 `first`，结束位置为 `first + count - 1`。`mode` 表示构建图元的类型，它必须是 `GL_TRIANGLES`、`GL_LINE_LOOP`、`GL_LINES`、`GL_POINTS` 等类型标识符之一。

与之类似，最基本的索引形式的绘制命令是 `glDrawElements()`。

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

使用 `count` 个元素来定义一系列几何图元，而元素的索引值保存在一个绑定到 `GL_ELEMENT_ARRAY_BUFFER` 的缓存中（元素数组缓存，`element array buffer`）。`indices` 定义了元素数组缓存中的偏移地址，也就是索引数据开始的位置，单位为字节。`type` 必须是 `GL_UNSIGNED_BYTE`、`GL_UNSIGNED_SHORT` 或者 `GL_UNSIGNED_INT` 中的一个，它给出了元素数组缓存中索引数据的类型。`mode` 定义了图元构建的方式，它必须是图元类型标识符中的一个，例如 `GL_TRIANGLES`、`GL_LINE_LOOP`、`GL_LINES` 或者 `GL_POINTS`。

这些函数都会从当前启用的顶点属性数组中读取顶点的信息，然后使用它们来构建 `mode` 指定的图元类型。顶点属性数组的启用可以通过 `glEnableVertexAttribArray()` 来完成，如第 1 章所介绍的。而 `glDrawArrays()` 只是直接将缓存对象中的顶点属性按照自身的排列顺序，直接取出并使用。`glDrawElements()` 使用了元素数组缓存中的索引数据来索引

⊖ 实际上，OpenGL 中还出现了两个 `Draw` 字样的函数，但是不会执行任何绘制操作，它们是 `glDrawBuffer()` 和 `glDrawBuffers()`。

各个顶点属性数组。所有看起来更为复杂的 OpenGL 绘制函数，在本质上都是基于这两个函数来完成功能实现的。例如，`glDrawElementsBaseVertex()` 可以将元素数组缓存中的索引数据进行一个固定数量的偏移。

```
void glDrawElementsBaseVertex(GLenum mode, GLsizei count, GLenum type, const GLvoid* indices, GLint basevertex);
```

本质上与 `glDrawElements()` 并无区别，但是它的第 i 个元素在传入绘制命令时，实际上读取的是各个顶点属性数组中的第 `indices[i] + basevertex` 个元素。

`glDrawElementsBaseVertex()` 可以根据某个索引基数来解析元素数组缓存中的索引数据。例如，如果一个模型存在多个版本（例如模型动画的多帧数据），并且保存在一个独立的顶点缓存集合中，只通过缓存中不同的偏移量来区分。那么 `glDrawElementsBaseVertex()` 就可以通过设置某一帧对应的索引基数，直接绘制这一帧所对应的动画数据。而每一帧用到的索引数据集总是一致的。

另一个与 `glDrawElements()` 行为很类似的函数是 `glDrawRangeElements()`。

```
void glDrawRangeElements(GLenum mode, GLuint start, GLuint end, GLsizei count, GLenum type, const GLvoid* indices);
```

这是 `glDrawElements()` 的一种更严格的形式，它实际上相当于应用程序（也就是开发者）与 OpenGL 之间形成的一种约定，即元素数组缓存中所包含的任何一个索引值（来自 `indices`）都会落入到 `start` 和 `end` 所定义的范围当中。

我们还可以通过这些功能的组合来实现一些更为高级的命令，例如，`glDrawRangeElementsBaseVertex()` 就相当于 `glDrawElementsBaseVertex()` 与 `glDrawRangeElements()` 功能的一种组合形式。

```
void glDrawRangeElementsBaseVertex(GLenum mode, GLuint start, GLuint end, GLsizei count, GLenum type, const GLvoid* indices, GLint basevertex);
```

同应用程序之间建立一种约束，其形式与 `glDrawRangeElements()` 类似，不过它同时也支持使用 `basevertex` 来设置顶点索引的基数。在这里，这个函数将首先检查元素数组缓存中保存的数据是否落入 `start` 和 `end` 之间，然后再对其添加 `basevertex` 基数。

这些函数同时还存在一些多实例的版本。多实例的介绍请参见下一节“多实例渲染”。多实例形式的命令包括 `glDrawArraysInstanced()`、`glDrawElementsInstanced()`，甚至还有 `glDrawElementsInstancedBaseVertex()`。最后，我们还要介绍两个特殊的命令，它们的参数不是直接从程序中得到的，而是从缓存对象当中获取。它们被称作间接绘制函数，

如果要使用的话，必须先将一个缓存对象绑定到 `GL_DRAW_INDIRECT_BUFFER` 目标上。`glDrawArrays()` 的间接版本叫做 `glDrawArraysIndirect()`。

```
void glDrawArraysIndirect(GLenum mode, const GLvoid* indirect);
```

特性与 `glDrawArraysInstanced()` 完全一致，不过绘制命令的参数是从绑定到 `GL_DRAW_INDIRECT_BUFFER` 的缓存（间接绘制缓存，draw indirect buffer）中获取的结构体数据。`indirect` 记录间接绘制缓存中的偏移地址。`mode` 必须是 `glDrawArrays()` 所支持的某个图元类型。

`glDrawArraysIndirect()` 中的实际绘制命令参数，是从间接绘制缓存中 `indirect` 地址的结构体中获取的。这个结构体的 C 语言形式的声明如例 3.3 所示。

例 3.3 DrawArraysIndirectCommand 结构体的声明

```
typedef struct DrawArraysIndirectCommand_t
{
    GLuint count;
    GLuint primCount;
    GLuint first;
    GLuint baseInstance;
} DrawArraysIndirectCommand;
```

`DrawArraysIndirectCommand` 结构体的所有域成员都会作为 `glDrawArraysInstanced()` 的参数进行解析。其中 `first` 和 `count` 会被直接传递到内部函数中。`primCount` 表示多实例的个数，而 `baseInstance` 就相当于多实例顶点属性的 `baseInstance` 偏移（不用担心，我们马上就会介绍多实例渲染的相关命令）。

`glDrawElements()` 的间接版本叫做 `glDrawElementsIndirect()`，它的原型定义如下：

```
void glDrawElementsIndirect(GLenum mode, GLenum type, const GLvoid* indirect);
```

本质上与 `glDrawElements()` 是一致的，但是绘制命令的参数是从绑定到 `GL_DRAW_INDIRECT_BUFFER` 的缓存中获取的。`indirect` 记录了间接绘制缓存中的偏移地址。`mode` 必须是 `glDrawElements()` 所支持的某个图元类型，而 `type` 指定了绘制命令调用时元素数组缓存中索引值的类型。

如果要使用 `glDrawArraysIndirect()`，那么 `glDrawArraysIndirect()` 中需要的参数也来自于元素数组缓存中 `indirect` 偏移地址所存储的结构体。这个结构体的 C 语言形式的声明如例 3.4 所示：

例 3.4 DrawElementsIndirectCommand 结构体的声明

```
typedef struct DrawElementsIndirectCommand_t
{
    GLuint count;
    GLuint primCount;
    GLuint firstIndex;
    GLuint baseVertex;
    GLuint baseInstance;
} DrawElementsIndirectCommand;
```



```

        GLsizei primcount);
{
    GLsizei i;

    for (i = 0; i < primcount; i++)
    {
        glDrawElements(mode, count[i], type, indices[i]);
    }
}

```

`glMultiDrawElements()` 的扩展版本包含了额外的 `baseVertex` 参数，也就是 `glMultiDrawElementsBaseVertex()` 函数。它的原型如下所示：

```

void glMultiDrawElementsBaseVertex(GLenum mode, const GLint* count, GLenum
type, const GLvoid* const* indices, GLsizei primcount, const GLint* baseVertex);

```

在一个 OpenGL 函数调用过程中绘制多组几何图元集。`first`、`indices` 和 `baseVertex` 都是数组的形式，数组的每个元素都相当于一次 `glDrawElementsBaseVertex()` 调用，元素的总数由 `primcount` 决定。

与之前所述的其他 OpenGL 多变量绘制命令类似，`glMultiDrawElementsBaseVertex()` 也可以等价于下面的 OpenGL 代码段：

```

void glMultiDrawElementsBaseVertex(GLenum mode,
                                   const GLsizei * count,
                                   GLenum type,
                                   const GLvoid * const * indices,
                                   GLsizei primcount,
                                   const GLint * baseVertex);
{
    GLsizei i;

    for (i = 0; i < primcount; i++)
    {
        glDrawElements(mode, count[i], type,
                       indices[i], baseVertex[i]);
    }
}

```

最后，如果有大量的绘制内容需要处理，并且相关参数已经保存到一个缓存对象中，可以直接使用 `glDrawArraysIndirect()` 或者 `glDrawElementsIndirect()` 处理的话，那么也可以使用这两个函数的多变量版本，即 `glMultiDrawArraysIndirect()` 和 `glMultiDrawElementsIndirect()`。

```

void glMultiDrawArraysIndirect(GLenum mode, const void* indirect, GLsizei
drawcount, GLsizei stride);

```

绘制多组图元集，相关参数全部保存到缓存对象中。在 `glMultiDrawArraysIndirect()` 的一次调用当中，可以分发总共 `drawcount` 个独立的绘制命令，命令中的参数与

`glDrawArraysIndirect()` 所用的参数是一致的。每个 `DrawArraysIndirectCommand` 结构体之间的间隔都是 `stride` 个字节。如果 `stride` 是 0 的话, 那么所有的数据结构体将构成一个紧密排列的数组。

```
void glMultiDrawElementsIndirect(GLenum mode, GLenum type, const void* indirect,
GLsizei drawcount, GLsizei stride);
```

绘制多组图元集, 相关参数全部保存到缓存对象中。在 `glMultiDrawElementsIndirect()` 的一次调用当中, 可以分发总共 `drawcount` 个独立的绘制命令, 命令中的参数与 `glDrawElementsIndirect()` 所用的参数是一致的。每个 `DrawElementsIndirectCommand` 结构体之间的间隔都是 `stride` 个字节。如果 `stride` 是 0 的话, 那么所有的数据结构体将构成一个紧密排列的数组。

OpenGL 绘制练习

这里给出一个相对比较简单例子, 它使用了本章中介绍的一部分 OpenGL 绘制命令。例 3.5 中所示为数据载入到缓存中, 并准备用于绘制的过程。例 3.6 中所示为绘制命令调用的过程。

例 3.5 绘制命令的准备过程示例

```
// 4 个顶点
static const GLfloat vertex_positions[] =
{
    -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,
};

// 每个顶点的颜色
static const GLfloat vertex_colors[] =
{
    1.0f, 1.0f, 1.0f, 1.0f,
    1.0f, 1.0f, 0.0f, 1.0f,
    1.0f, 0.0f, 1.0f, 1.0f,
    0.0f, 1.0f, 1.0f, 1.0f
};

// 三个索引值 (我们这次只绘制一个三角形)
static const GLushort vertex_indices[] =
{
    0, 1, 2
};

// 设置元素数组缓存
glGenBuffers(1, ebo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo[0]);
```

```

glBufferData(GL_ELEMENT_ARRAY_BUFFER,
             sizeof(vertex_indices), vertex_indices, GL_STATIC_DRAW);

// 设置顶点属性
glGenVertexArrays(1, vao);
glBindVertexArray(vao[0]);

glGenBuffers(1, vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
glBufferData(GL_ARRAY_BUFFER,
             sizeof(vertex_positions) + sizeof(vertex_colors),
             NULL, GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0,
                sizeof(vertex_positions), vertex_positions);
glBufferSubData(GL_ARRAY_BUFFER,
                sizeof(vertex_positions), sizeof(vertex_colors),
                vertex_colors);

```

例 3.6 绘制命令示例

```

// DrawArrays
model_matrix = vmath::translation(-3.0f, 0.0f, -5.0f);
glUniformMatrix4fv(render_model_matrix_loc, 4, GL_FALSE, model_matrix);
glDrawArrays(GL_TRIANGLES, 0, 3);

// DrawElements
model_matrix = vmath::translation(-1.0f, 0.0f, -5.0f);
glUniformMatrix4fv(render_model_matrix_loc, 4, GL_FALSE, model_matrix);
glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_SHORT, NULL);

// DrawElementsBaseVertex
model_matrix = vmath::translation(1.0f, 0.0f, -5.0f);
glUniformMatrix4fv(render_model_matrix_loc, 4, GL_FALSE, model_matrix);
glDrawElementsBaseVertex(GL_TRIANGLES, 3, GL_UNSIGNED_SHORT, NULL, 1);

// DrawArraysInstanced
model_matrix = vmath::translation(3.0f, 0.0f, -5.0f);
glUniformMatrix4fv(render_model_matrix_loc, 4, GL_FALSE, model_matrix);
glDrawArraysInstanced(GL_TRIANGLES, 0, 3, 1);

```

例 3.5 和例 3.6 的程序运行结果如图 3-5 所示。它看起来并不是特别引人入胜，不过这里你可以看到四个相似的三角形，并且每个三角形的渲染都用到了一个不同的绘制命令。

3.4.1 图元的重启动

当需要处理较大的顶点数据集的时候，我们可能会被迫执行大量的 OpenGL 绘制操作，并且每次绘制的内容总是与前一次图元的类型相同（例如 GL_TRIANGLE_STRIP）。当然，我们可以使用 `glMultiDraw*()` 形式的函数，但是这样需要额外去管理图元的起始索引位置和长度的数组。

OpenGL 支持在同一个渲染命令中进行图元重启动的功能，此时需要指定一个特殊的值，叫做图元重启动索引（primitive restart index），OpenGL 内部会对它做特殊的处理。如

果绘制调用过程中遇到了这个重启动索引，那么就会从这个索引之后的顶点开始，重新开始进行相同图元类型的渲染。图元重启动索引的定义是通过 `glPrimitiveRestartIndex()` 函数来完成的。

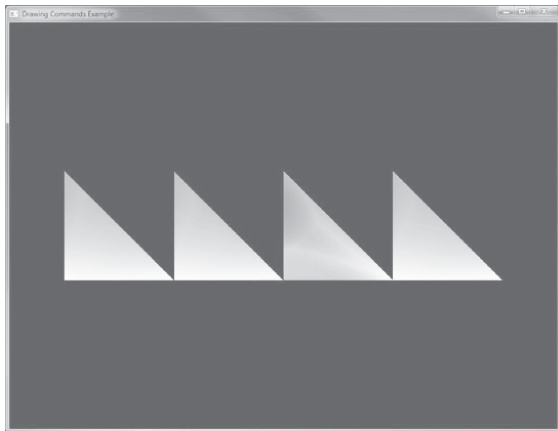


图 3-5 绘制命令的简单示例

```
void glPrimitiveRestartIndex(GLuint index);
```

设置一个顶点数组元素的索引值，用来指定渲染过程中，从什么地方启动新的图元绘制。如果在处理定点数组元素索引的过程中遇到了一个符合该索引的数值，那么系统不会处理它对应的顶点数据，而是终止当前的图元绘制，并且从下一个顶点重新开始渲染同一类型的图元集合。

如果顶点的渲染需要在某一个 `glDrawElements()` 系列的函数调用中完成，那么它可以用到 `glPrimitiveRestartIndex()` 所指定的索引，并且检查这个索引值是否会出现元素数组缓存中。不过，我们必须启用图元重启动特性之后才可以进行这种检查。图元重启动的控制可以通过 `glEnable()` 和 `glDisable()` 函数来完成，调用的参数为 `GL_PRIMITIVE_RESTART`。

考虑图 3-6 中的顶点布局，它给出了一个三角形条带，并且通过图元重启动的方式打断为两个部分。在图中，图元重启动索引设置为 8。在三角形渲染过程中，OpenGL 会一直监控元素数组缓存中是否出现索引 8，当这个值出现的时候，OpenGL 不会创建一个顶点，而是结束当前的三角形条带绘制。下一个顶点（索引 9）将成为一个新的三角形条带的第一个顶点，因此我们最终构建了两个三角形条带。



图 3-6 使用图元重启动的特性打断三角形条带

下面的例子演示了图元重启动的一个简单应用——这里使用图元重启动索引将一个立方体分割为两个三角形条带。例 3.7 和例 3.8 所示为立方体的数据设置过程，以及绘制过程。

例 3.7 初始化立方体数据，它是由两个三角形条带组成的

```
// 设置立方体的 8 个角点，边长为 2，中心为原点
static const GLfloat cube_positions[] =
{
    -1.0f, -1.0f, -1.0f, 1.0f,
    -1.0f, -1.0f, 1.0f, 1.0f,
    -1.0f, 1.0f, -1.0f, 1.0f,
    -1.0f, 1.0f, 1.0f, 1.0f,
    1.0f, -1.0f, -1.0f, 1.0f,
    1.0f, -1.0f, 1.0f, 1.0f,
    1.0f, 1.0f, -1.0f, 1.0f,
    1.0f, 1.0f, 1.0f, 1.0f
};

// 每个顶点的颜色
static const GLfloat cube_colors[] =
{
    1.0f, 1.0f, 1.0f, 1.0f,
    1.0f, 1.0f, 0.0f, 1.0f,
    1.0f, 0.0f, 1.0f, 1.0f,
    1.0f, 0.0f, 0.0f, 1.0f,
    0.0f, 1.0f, 1.0f, 1.0f,
    0.0f, 1.0f, 0.0f, 1.0f,
    0.0f, 0.0f, 1.0f, 1.0f,
    0.5f, 0.5f, 0.5f, 1.0f
};

// 三角形条带的索引
static const GLushort cube_indices[] =
{
    0, 1, 2, 3, 6, 7, 4, 5, // 第一组条带
    0xFFFF, // <<- 这是重启动的索引
    2, 6, 0, 4, 1, 5, 3, 7 // 第二组条带
};

// 设置元素数组缓存
glGenBuffers(1, ebo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo[0]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER,
             sizeof(cube_indices),
             cube_indices, GL_STATIC_DRAW);

// 设置顶点属性
glGenVertexArrays(1, vao);
glBindVertexArray(vao[0]);

glGenBuffers(1, vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
glBufferData(GL_ARRAY_BUFFER,
             sizeof(cube_positions) + sizeof(cube_colors),
             NULL, GL_STATIC_DRAW);
```

```

glBufferSubData(GL_ARRAY_BUFFER, 0,
                sizeof(cube_positions), cube_positions);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(cube_positions),
                sizeof(cube_colors), cube_colors);

glVertexAttribPointer(0, 4, GL_FLOAT,
                     GL_FALSE, 0, NULL);
glVertexAttribPointer(1, 4, GL_FLOAT,
                     GL_FALSE, 0,
                     (const GLvoid *)sizeof(cube_positions));
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);

```

图 3-7 所示就是例 3.7 给出的三角形数据，它使用两个独立的三角形条带来表达一个立方体的形状。

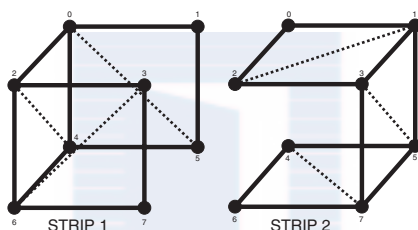


图 3-7 使用两个三角形条带组成立方体

例 3.8 使用图元重启的方式绘制由两个三角形条带组成的立方体

```

// 设置使用 glDrawElements
glBindVertexArray(vao[0]);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo[0]);

#ifdef USE_PRIMITIVE_RESTART
// 如果开启了图元重启，那么只需要调用一次绘制命令
glEnable(GL_PRIMITIVE_RESTART);
glPrimitiveRestartIndex(0xFFFF);
glDrawElements(GL_TRIANGLE_STRIP, 17, GL_UNSIGNED_SHORT, NULL);
#else
// 如果没有开启图元重启，那么需要调用两次绘制命令
glDrawElements(GL_TRIANGLE_STRIP, 8, GL_UNSIGNED_SHORT, NULL);
glDrawElements(GL_TRIANGLE_STRIP, 8, GL_UNSIGNED_SHORT,
               (const GLvoid *) (9 * sizeof(GLushort)));
#endif

```



注意 每当 OpenGL 在元素数组缓存中遇到当前设置的重启动索引时，都会执行图元重启的操作。因此，不妨将重启动索引设置为一个代码中绝对不会用到的数值。默认的重启动索引为 0，但是这个值非常容易出现在元素数组缓存当中。一个不错的选择是 $2^n - 1$ ，这里的 n 表示索引值的位数（例如 `GL_UNSIGNED_SHORT` 的索引就是 16，而 `GL_UNSIGNED_INT` 的索引就是 32）。这个数不太可能是一个真实的索引值。如果将它作为重启动索引标准值的话，那么我们就不需要为程序中的每一个模型都单独设置一个索引了。

3.5 多实例渲染

实例化 (instancing) 或者多实例渲染 (instanced rendering) 是一种连续执行多条相同的渲染命令的方法, 并且每个渲染命令所产生的结果都会有轻微的差异。这是一种非常有效的, 使用少量 API 调用来渲染大量几何体的方法。OpenGL 中已经提供了一些常用绘制函数的多变量形式来优化命令的多次执行。此外, OpenGL 中也提供了多种机制, 允许着色器使用绘制的不同实例作为输入, 并且对每个实例 (而不是每个顶点) 都赋予不同的顶点属性值。最简单的多实例渲染的命令是:

```
void glDrawArraysInstanced(GLenum mode, GLint first, GLsizei count, GLsizei primCount);
```

通过 mode、first 和 count 所构成的几何体图元集 (相当于 glDrawArrays() 函数所需的独立参数), 绘制它的 primCount 个实例。对于每个实例, 内置变量 gl_InstanceID 都会依次递增, 新的数值会被传递到顶点着色器, 以区分不同实例的顶点属性。

这个函数是 glDrawArrays() 的多实例版本, 我们可以注意到这两个函数之间的相似之处。glDrawArraysInstanced() 的参数与 glDrawArrays() 是完全等价的, 只是多了一个 primCount 参数。这个参数用于设置准备渲染的实例个数。当 OpenGL 执行这个函数的时候, 它实际上会执行 glDrawArrays() 的 primCount 次拷贝, 每次的 mode、first 和 count 参数都是直接传入的。其他 OpenGL 的绘制命令也有对应的 *Instanced 版本, 例如 glDrawElementsInstanced() (对应 glDrawElements()) 和 glDrawElementsInstancedBaseVertex() (对应 glDrawElementsBaseVertex())。glDrawElementsInstanced() 函数的定义如下:

```
void glDrawElementsInstanced(GLenum mode, GLsizei count, GLenum type, const void* indices, GLsizei primCount);
```

通过 mode、count 和 indices 所构成的几何体图元集 (相当于 glDrawElements() 函数所需的独立参数), 绘制它的 primCount 个实例。与 glDrawArraysInstanced() 类似, 对于每个实例, 内置变量 gl_InstanceID 都会依次递增, 新的数值会被传递到顶点着色器, 以区分不同实例的顶点属性。

再次注意到, glDrawElementsInstanced() 的参数与 glDrawElements() 的是等价的, 只是新增了 primCount 参数。每次调用多实例函数时, 在本质上 OpenGL 都会根据 primCount 参数来设置多次运行整个命令。这看起来并不是很有用的功能。不过, OpenGL 提供了两种机制来设置对应不同实例的顶点属性, 并且在顶点着色器中可以获取当前实例所对应的索引号。

```
void glDrawElementsInstancedBaseVertex(GLenum mode, GLsizei count, GLenum
type, const void* indices, GLsizei instanceCount, GLuint baseVertex);
```

通过 mode、count、indices 和 baseVertex 所构成的几何体图元集（相当于 glDrawElementsBaseVertex() 函数所需的独立参数），绘制它的 instanceCount 个实例。与 glDrawArraysInstanced() 类似，对于每个实例，内置变量 gl_InstanceID 都会依次递增，新的数值会被传递到顶点着色器，以区分不同实例的顶点属性。

3.5.1 多实例的顶点属性

多实例的顶点属性与正规的顶点属性是类似的。它们在顶点着色器中的声明和使用方式都是完全一致的。对于应用程序端来说，它们的配置方法与正规的顶点属性也是相同的。也就是说，它们需要保存到缓存对象中，可以通过 glGetAttribLocation() 查询，通过 glVertexAttribPointer() 来设置，以及通过 glEnableVertexAttribArray() 和 glDisableVertexAttribArray() 进行启用与禁用。下面的重要的函数就是用来启用多实例的顶点属性的：

```
void glVertexAttribDivisor(GLuint index, GLuint divisor);
```

设置多实例渲染时，位于 index 位置的顶点着色器中顶点属性是如何分配值到每个实例的。divisor 的值如果是 0 的话，那么该属性的多实例特性将被禁用，而其他的值则表示顶点着色器，每 divisor 个实例都会分配一个新的属性值。

glVertexAttribDivisor() 函数用于控制顶点属性更新的频率。index 表示设置多实例特性的顶点属性的索引位置，它与传递给 glVertexAttribPointer() 和 glEnableVertexAttribArray() 的索引值是一致的。默认情况下，每个顶点都会分配到一个独立的属性值。如果 divisor 设置为 0 的话，那么顶点属性将遵循这一默认，非实例化的规则。如果 divisor 设置为一个非零的值，那么顶点属性将启用多实例的特性，此时 OpenGL 从属性数组中每隔 divisor 个实例都会读取一个新的数值（而不是之前的每个顶点）。此时在这个属性所对应的顶点属性数组中，数据索引值的计算将变成 instance/divisor 的形式，其中 instance 表示当前的实例数目，而 divisor 就是当前属性的更新频率值。对于每个多实例的顶点属性来说，在顶点着色器中，每个实例中的所有顶点都会共享同一个属性值。如果 divisor 设置为 2 的话，那么每两个实例会共享同一个属性值；如果值为 3，那么就是每三个实例，以此类推。我们可以参考例 3.9 中的顶点属性声明，这其中已经包含了一些多实例的属性。

例 3.9 多实例的顶点着色器属性示例

```
#version 410 core

// 位置和法线都是规则的顶点属性
layout (location = 0) in vec4 position;
layout (location = 1) in vec3 normal;
```

```
// 颜色是一个逐实例的属性
layout (location = 2) in vec4 color;

// model_matrix 是一个逐实例的变换矩阵。注意一个 mat4 会占据 4 个连续的位置，
// 因此它实际上占据了 3、4、5、6 四个索引位
layout (location = 3) in mat4 model_matrix;
```

注意在例 3.9 中，多实例顶点属性 `color` 和 `model_matrix` 的声明并没有什么特别的地方。现在再阅读例 3.10 中的代码，其中已经将例 3.9 中的一部分顶点属性设置为多实例的形式。

例 3.10 多实例顶点属性的设置示例

```
// 获取顶点属性在 prog 当中的位置，prog 就是我们准备用于渲染的着色器程序对象。
// 注意，这一步并不是必需的，因为我们已经在顶点着色器中设置了所有属性的位置。
// 这里的代码可以编写的更简单一些，只需要直接给出程序中已经设置的属性位置即可

int position_loc    = glGetAttribLocation(prog, "position");
int normal_loc      = glGetAttribLocation(prog, "normal");
int color_loc       = glGetAttribLocation(prog, "color");
int matrix_loc      = glGetAttribLocation(prog, "model_matrix");

// 配置正规的顶点属性数组——顶点和法线
glBindBuffer(GL_ARRAY_BUFFER, position_buffer);
glVertexAttribPointer(position_loc, 4, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(position_loc);
glBindBuffer(GL_ARRAY_BUFFER, normal_buffer);
glVertexAttribPointer(normal_loc, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(normal_loc);

// 设置颜色的数组。我们希望几何体的每个实例都有一个不同的颜色，因此
// 直接将颜色值置入缓存对象中，然后设置一个实例化的顶点属性
glBindBuffer(GL_ARRAY_BUFFER, color_buffer);
glVertexAttribPointer(color_loc, 4, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(color_loc);
// 这里的设置很重要……设置颜色数组的更新频率为 1，那么 OpenGL 会给每个实例
// 设置一个新的颜色值，而不是每个顶点的设置了
glVertexAttribDivisor(color_loc, 1);

// 与之类似，我们给模型矩阵也做同样的设置。注意输入到顶点着色器的矩阵
// 会占据 N 个连续的输入位置，其中 N 表示矩阵的列数。因此……我们等于设置了
// 4 个顶点属性
glBindBuffer(GL_ARRAY_BUFFER, model_matrix_buffer);
// 循环遍历矩阵的每一列……
for (int i = 0; i < 4; i++)
{
    // 设置顶点属性
    glVertexAttribPointer(matrix_loc + i,                // 位置
                           4, GL_FLOAT, GL_FALSE,        // vec4
                           sizeof(mat4),                 // 数据步幅
    }
```

```

        (void *) (sizeof(vec4) * i)); // 起始偏移值
// 启用顶点属性
glEnableVertexAttribArray(matrix_loc + i);
// 实现多实例化
glVertexAttribDivisor(matrix_loc + i, 1);
}

```

例 3.10 当中，`position` 和 `normal` 是规则的，非实例化的顶点属性。而 `color` 是一个 `divisor` 被设置为 1 的多实例顶点属性。也就是说，每个实例的 `color` 属性都会有一个独立的值（而实例当中的所有顶点都会使用这一个值）。此外，`model_matrix` 属性也被设置为多实例的属性，它可以为每个实例都提供一个新的模型变换矩阵。`mat4` 类型的属性会占用多个连续的位置。因此我们需要遍历矩阵的每一列并且分别进行设置。顶点着色器中剩余的代码部分可以参见例 3.11。

例 3.11 多实例属性的顶点着色器示例

```

// 观察矩阵和投影矩阵在绘制过程中都是常数
uniform mat4 view_matrix;
uniform mat4 projection_matrix;

// 顶点着色器的输出（对应于片元着色器的输入）
out VERTEX
{
    vec3    normal;
    vec4    color;
} vertex;

// 现在开始
void main(void)
{
    // 根据 uniform 的观察矩阵和逐实例的模型矩阵构建完整的模型 - 视点矩阵
    mat4 model_view_matrix = view_matrix * model_matrix;

    // 首先用模型 - 视点矩阵变换位置，然后是投影矩阵
    gl_Position = projection_matrix * (model_view_matrix * position);

    // 使用模型 - 视点矩阵的左上 3×3 子矩阵变换法线
    vertex.normal = mat3(model_view_matrix) * normal;

    // 将逐实例的颜色值直接传入片元着色器
    vertex.color = color;
}

```

上面的代码设置了各个实例的模型矩阵，然后使用例 3.12 中的着色器代码来绘制几何体实例。每个实例都有自己的模型矩阵，而观察矩阵（包括一个绕 Y 轴的旋转，以及一个 Z 方向的平移操作）对于所有的实例都是相同的。模型矩阵是通过 `glMapBuffer()` 映射的方式直接写入到缓存中的。每个模型矩阵都会将物体移动到远离原点的位置，然后绕着原点对平移过的物体进行旋转。观察和投影矩阵都是简单地通过 `uniform` 变量来传递的。然后，我们直接调用一次 `glDrawArraysInstanced()`，绘制模型的所有实例。

例 3.12 多实例绘制的代码示例

```
// 映射缓存
mat4 * matrices = (mat4 *)glMapBuffer(GL_ARRAY_BUFFER,
                                       GL_WRITE_ONLY);

// 设置每个实例的模型矩阵
for (n = 0; n < INSTANCE_COUNT; n++)
{
    float a = 50.0f * float(n) / 4.0f;
    float b = 50.0f * float(n) / 5.0f;
    float c = 50.0f * float(n) / 6.0f;

    matrices[n] = rotation(a + t * 360.0f, 1.0f, 0.0f, 0.0f) *
                  rotation(b + t * 360.0f, 0.0f, 1.0f, 0.0f) *
                  rotation(c + t * 360.0f, 0.0f, 0.0f, 1.0f) *
                  translation(10.0f + a, 40.0f + b, 50.0f + c);
}

// 完成后解除映射
glUnmapBuffer(GL_ARRAY_BUFFER);

// 启用多实例的程序
glUseProgram(render_prog);

// 设置观察矩阵和投影矩阵
mat4 view_matrix(translation(0.0f, 0.0f, -1500.0f) *
                 rotation(t * 360.0f * 2.0f, 0.0f, 1.0f, 0.0f));
mat4 projection_matrix(frustum(-1.0f, 1.0f,
                              -aspect, aspect, 1.0f, 5000.0f));

glUniformMatrix4fv(view_matrix_loc, 1,
                   GL_FALSE, view_matrix);
glUniformMatrix4fv(projection_matrix_loc, 1,
                   GL_FALSE, projection_matrix);

// 渲染 INSTANCE_COUNT 个模型
glDrawArraysInstanced(GL_TRIANGLES, 0, object_size, INSTANCE_COUNT);
```

程序运行的结果如图 3-8 所示。在这个例子中，常量 `INSTANCE_COUNT`（在例 3.10 和例 3.12 的代码中被使用）的值为 100。一共绘制了 100 份模型的拷贝，每个拷贝都有一个不同的位置和颜色。这些模型也可以很简单地改成森林中的数目、太空舰队中的飞船，或者城市中的一栋建筑。

例 3.9 到例 3.12 中存在一些效率问题。每个实例中的所有顶点都会产生一些相同的结果值，但是它们依然会被逐顶点地进行计算。有的时候应当考虑解决这类问题。例如，`model_view_matrix` 的计算结果矩阵对于单个实例中的所有顶点都是相同的。这里，我们可以通过第二个实例化的 `mat4` 属性，输入逐实例的模型视点矩阵数据来避免重复的计算工作，其他时候可能无法避免这种计算，但是还是可以把它移动到几何着色器中完成，这样每次计算都是逐图元，而非逐顶点完成的，或者也可以用到几何着色器的多实例方法。我们会在第 10 章介绍这些技术的内容。

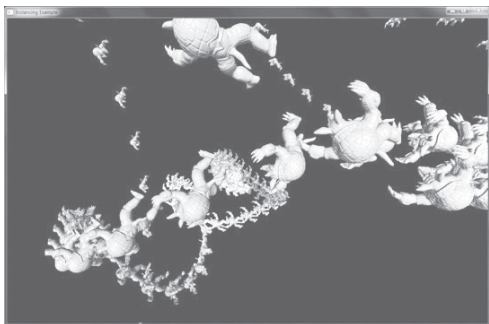


图 3-8 多实例顶点属性的渲染结果



注意

调用一个多实例的绘制命令，与多次调用它的非实例化的版本然后再执行其他的 OpenGL 命令，几乎是等价的操作。因此，如果将循环当中已有的一系列 OpenGL 函数直接转换成一系列的实例化绘制命令，那么得到的结果不会是一致的。

另一个使用多实例顶点属性的例子就是将一系列纹理打包到一个 2D 纹理数组中，然后将数组的序号通过实例化的顶点属性传递给每个实例。顶点着色器可以将实例对应的序号传递到片元着色器中，然后使用不同的纹理来渲染不同的几何体实例。

我们也可以在系统内部设置一个偏移值，以改变顶点缓存中得到实例化的顶点属性时的索引位置。与 `glDrawElementsBaseVertex()` 中提供的 `baseVertex` 参数类似，在多实例绘制函数当中，实例的索引偏移值可以通过一个额外的 `baseInstance` 参数来设置。带有这个 `baseInstance` 参数的函数包括 `glDrawArraysInstancedBaseInstance()`、`glDrawElementsInstancedBaseInstance()` 和 `glDrawElementsInstancedBaseVertexBaseInstance()`。它们的原型如下：

```
void glDrawArraysInstancedBaseInstance(GLenum mode, GLint first, GLsizei count,
    GLsizei instanceCount, GLuint baseInstance);
```

对于通过 `mode`、`first` 和 `count` 所构成的几何体图元集（相当于 `glDrawArrays()` 函数所需的独立参数），绘制它的 `primCount` 个实例。对于每个实例，内置变量 `gl_InstanceID` 都会依次递增，新的数值会被传递到顶点着色器，以区分不同实例的顶点属性。此外，`baseInstance` 的值用来对实例化的顶点属性设置一个索引的偏移值，从而改变 OpenGL 取出的索引位置。

```
void glDrawElementsInstancedBaseInstance(GLenum mode, GLsizei count,
    GLenum type, const GLvoid* indices, GLsizei instanceCount, GLuint baseInstance);
```

对于通过 `mode`、`count` 和 `indices` 所构成的几何体图元集（相当于 `glDrawElements()` 函数所需的独立参数），绘制它的 `primCount` 个实例。与 `glDrawArraysInstanced()` 类似，

对于每个实例，内置变量 `gl_InstanceID` 都会依次递增，新的数值会被传递到顶点着色器，以区分不同实例的顶点属性。此外，`baseInstance` 的值用来对实例化的顶点属性设置一个索引的偏移值，从而改变 OpenGL 取出的索引位置。

```
void glDrawElementsInstancedBaseVertexBaseInstance(GLenum mode, GLsizei count,
GLenum type, const GLvoid* indices, GLsizei instanceCount, GLuint baseVertex, GLuint baseInstance);
```

对于通过 `mode`、`count`、`indices` 和 `baseVertex` 所构成的几何体图元集（相当于 `glDrawElementsBaseVertex()` 函数所需的独立参数），绘制它的 `primCount` 个实例。与 `glDrawArraysInstanced()` 类似，对于每个实例，内置变量 `gl_InstanceID` 都会依次递增，新的数值会被传递到顶点着色器，以区分不同实例的顶点属性。此外，`baseInstance` 的值用来对实例化的顶点属性设置一个索引的偏移值，从而改变 OpenGL 取出的索引位置。

3.5.2 在着色器中使用实例计数器

除了多实例的顶点属性之外，当前实例的索引值可以在顶点着色器中通过内置 `gl_InstanceID` 变量获得。这个变量被声明为一个整数。它从 0 开始计数，每当一个实例被渲染之后，这个值都会加 1。`gl_InstanceID` 总是存在于顶点着色器中，即使当前的绘制命令并没有用到多实例的特性也是如此。这种时候，它的值保持为 0。`gl_InstanceID` 的值可以作为 `uniform` 数组的索引使用，也可以作为纹理查找的参数使用，或者作为某个分析函数的输入，以及其他的目的。

在下面的例子中，我们使用 `gl_InstanceID` 重现了例 3.9 到例 3.12 的功能，不过这一次使用的是纹理缓存对象（Texture Buffer Objects, TBO）而非实例化的顶点属性。这里我们将例 3.9 中的顶点属性替换为 TBO 的查找，因此移除了相应的顶点属性设置代码。使用一个 TBO 来记录每个实例的颜色值，而第二个 TBO 用来记录模型矩阵的值。其他顶点属性的声明和设置代码与例 3.9 和例 3.10 的内容相同（当然，忽略了 `color` 和 `model_matrix` 属性的设置）。因为现在采用显式的方法在顶点着色器中获得了每个实例的颜色和模型矩阵，所以在顶点着色器的主体中也要添加更多额外的代码，如例 3.13 所示。

例 3.13 顶点着色器的 `gl_VertexID` 示例

```
// 矩阵和投影矩阵在绘制过程中都是常数
uniform mat4 view_matrix;
uniform mat4 projection_matrix;
// 设置 TBO 来保存逐实例的颜色数据和模型矩阵数据
uniform samplerBuffer color_tbo;
uniform samplerBuffer model_matrix_tbo;

// 顶点着色器的输出（对应于片元着色器的输入）
out VERTEX
{
    vec3    normal;
```

```

    vec4    color;
} vertex;

// 现在开始
void main(void)
{
    // 使用 gl_InstanceID 从颜色值的 TBO 当中获取数据
    vec4 color = texelFetch(color_tbo, gl_InstanceID);

    // 模型矩阵的生成更为复杂一些，因为我们不能直接在 TBO 中存储 mat4 数据
    // 我们需要将每个矩阵都保存为四个 vec4 的变量，然后在着色器中重新装配
    // 为矩阵的形式。首先，获取矩阵的四列数据（注意，矩阵在内存中的存储
    // 采用了列主序的方式）
    vec4 col1 = texelFetch(model_matrix_tbo, gl_InstanceID * 4);
    vec4 col2 = texelFetch(model_matrix_tbo, gl_InstanceID * 4 + 1);
    vec4 col3 = texelFetch(model_matrix_tbo, gl_InstanceID * 4 + 2);
    vec4 col4 = texelFetch(model_matrix_tbo, gl_InstanceID * 4 + 3);

    // 现在将四列装配为一个矩阵
    mat4 model_matrix = mat4(col1, col2, col3, col4);

    // 根据 uniform 观察矩阵和逐实例的模型矩阵构建完整的模型 - 视点矩阵

    mat4 model_view_matrix = view_matrix * model_matrix;

    // 首先用模型 - 视点矩阵变换位置，然后是投影矩阵

    gl_Position = projection_matrix * (model_view_matrix *
                                      position);

    // 使用模型 - 视点矩阵的左上 3×3 子矩阵变换法线
    vertex.normal = mat3(model_view_matrix) * normal;
    // 将逐实例的颜色值直接传入片元着色器
    vertex.color = color;
}

```

为了使用例 3.13 中的着色器，我们还需要创建和初始化 TBO 对象，以存储 color_tbo 和 model_matrix_tbo 的采样信息，只是不需要再初始化多实例的顶点属性了。不过，除了这些代码设置之间存在差异之外，程序的本质是没有发生变化的。

例 3.14 多实例顶点属性的设置示例

```

// 获取顶点属性在 prog 当中的位置，prog 就是准备用于渲染的着色器程序
// 对象。注意，这一步并不是必需的，因为我们已经在顶点着色器中设置了所
// 有属性的位置。这里的代码可以编写的更简单一些，只需要直接给出程序中
// 已经设置的属性位置即可
int position_loc    = glGetAttribLocation(prog, "position");
int normal_loc      = glGetAttribLocation(prog, "normal");

// 配置正规的顶点属性数组——顶点和法线
glBindBuffer(GL_ARRAY_BUFFER, position_buffer);
glVertexAttribPointer(position_loc, 4, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(position_loc);
glBindBuffer(GL_ARRAY_BUFFER, normal_buffer);
glVertexAttribPointer(normal_loc, 3, GL_FLOAT, GL_FALSE, 0, NULL);

```

```

glEnableVertexAttribArray(normal_loc);

// 现在设置多实例颜色和模型矩阵的 TBO.....
// 首先创建 TBO 来存储颜色值, 绑定一个缓存然后初始化数据格式。缓存必须
// 在之前已经创建, 并且大小可以包含一个 vec4 的逐实例数据
glGenTextures(1, &color_tbo);
glBindTexture(GL_TEXTURE_BUFFER, color_tbo);
glTexBuffer(GL_TEXTURE_BUFFER, GL_RGBA32F, color_buffer);
// 再次使用 TBO 来存储模型矩阵值。这个缓存对象 (model_matrix_buffer) 必须
// 在之前已经创建, 并且大小可以包含一个 mat4 的逐实例数据
glGenTextures(1, &model_matrix_tbo);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_BUFFER, model_matrix_tbo);
glTexBuffer(GL_TEXTURE_BUFFER, GL_RGBA32F, model_matrix_buffer);

```

注意, 例 3.14 中的代码实际上比例 3.10 更为短小和简单。这是因为不再使用内置的 OpenGL 功能来获取逐实例的数据, 而是直接使用着色器写出。这一点从例 3.13 比例 3.11 增加的复杂性就可以看出。而这样的变化也带来了更多的强大功能和灵活性。举例来说, 如果实例的数量较少, 那么使用 uniform 数组可能比使用 TBO 来存储数据更为合适, 但是后者对性能的改善更为理想。除此之外, 使用 `gl_InstanceID` 来驱动的方法与原始的例子相比并没有更多的改动。实际上, 例 3.12 中的渲染代码是被完整迁移过来的, 它所产生的渲染结果与原来的程序完全相同。我们可以参看下面的截图 (见图 3-9)。

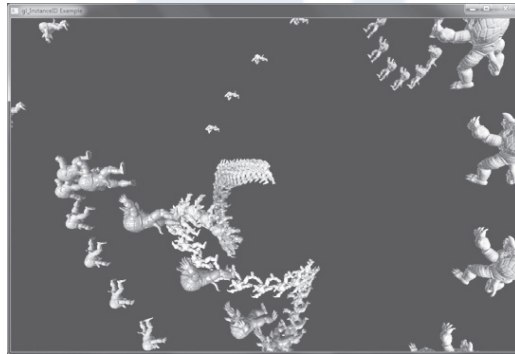


图 3-9 使用 `gl_InstanceID` 进行多实例渲染的结果

3.5.3 多实例方法的回顾

如果要在程序中使用多实例的方法, 那么我们应当:

- ☐ 为准备实例化的内容创建顶点着色器输入。
- ☐ 使用 `glVertexAttribDivisor()` 设置顶点属性的分隔频率。
- ☐ 在顶点着色器中使用内置的 `gl_InstanceID` 变量。
- ☐ 使用渲染函数的多实例版本, 例如 `glDrawArraysInstanced()`、`glDrawElementsInstanced()` 和 `glDrawElementsInstancedBaseVertex()`。