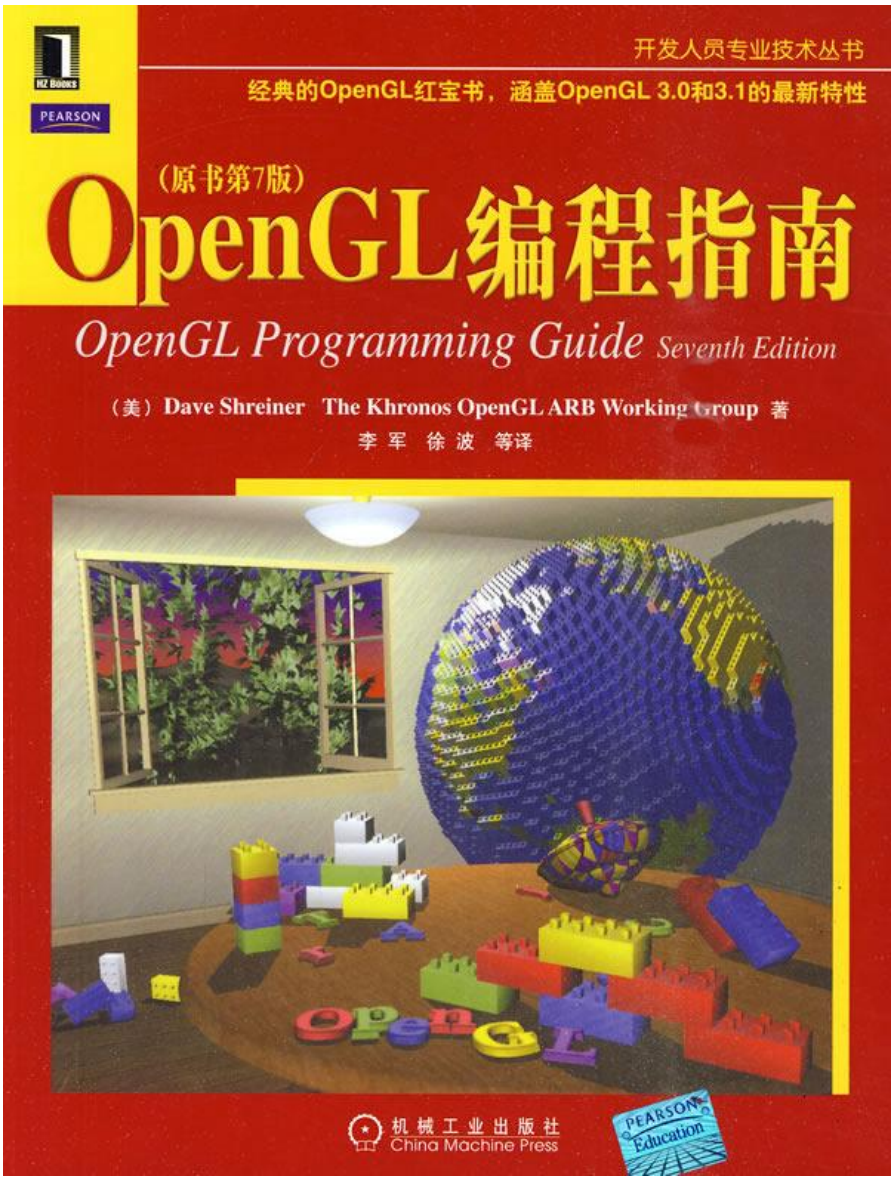


OpenGL 编程指南（原书第 7 版）

（美）Dava Shreiner 著

李军/徐波译

作译者简介：本书由李军和徐波译，原作者为 Dava Shreiner，是 ARM 公司图形技术总监，长期担任 SGI 核心 OpenGL 组的成员。他首次开设了 OpenGL 的商业培训课程，拥有二十多年的计算机图形应用开发经验。

摘要：本书对 OpenGL 以及 OpenGL 实用函数库进行了全面而又权威的介绍，素有“OpenGL 红宝书”之誉。本书的上一个版本覆盖了 OpenGL 2.1 版的所有内容。本版涵盖了 OpenGL 3.0 和 3.1 的最新特性。本书以清晰的语言描述了 OpenGL 的功能以及许多基本的计算机图形技巧，例如，创建和渲染 3D 模型、从不同的透视角度观察物体、使用着色、光照和纹理贴图使场景更加逼真等。另外，本书还深入探讨了许多高级技巧，包括纹理贴图、抗锯齿、雾和

大气效果、NURBS、图像处理等。本书内容详实，讲解生动，图文并茂，是 OpenGL 程序员的绝佳编程指南。

目录

译者序

OpenGL 是图形硬件的一种软件接口。从本质上说，它是一个 3D 图形和模型库，具有高度的可移植性，并且具有非常快的渲染速度。如今，OpenGL 广泛应用于游戏、医学影像、地理信息、气象模拟等领域，是高性能图形和交互性场景处理的行业标准。

OpenGL 的前身是 SGI 公司开发的 IRIS GL 图形函数库。SGI 是一家久负盛名的公司，在计算机图形和动画领域处于业界领先地位。IRIS GL 最初是一个 2D 图形函数库，后来逐渐演化为 SGI 的高端 IRIS 图形工作站所使用的 3D 编程 API。后来，由于图形技术的发展，SGI 对 IRIS GL 的移植性进行了改进和提高，使它逐步发展成如今的 OpenGL。在此期间，OpenGL 得到了各大厂商的支持，从而成为一种广泛流行的三维图形标准。

OpenGL 并不是一种编程语言，而更像是一个 C 运行时函数库。它提供了一些预包装的功能，帮助开发人员编写功能强大的三维图形应用程序。OpenGL 可以在多种操作系统平台上运行，例如各种版本的 Windows、UNIX/Linux、Mac OS 和 OS/2 等。

OpenGL 是一个开放的标准，虽然它由 SGI 首创，但是它的标准并不控制在 SGI 的手中，而是由 OpenGL 体系结构审核委员会（ARB）掌管。ARB 由 SGC、DEC、IBM、Intel 和 Microsoft 等著名公司于 1992 年创立，后来又陆续添加了 nVidia、ATI 等图形芯片领域的巨擘。ARB 每隔 4 年举行一次会议，对 OpenGL 规范进行维护和改善，并出台计划对 OpenGL 标准进行升级，使 OpenGL 一直保持与时代同步。

2006 年，SGI 公司把 OpenGL 标准的控制从 ARB 移交给一个新的工作组—Khronos 小组（www.khronos.org）。Khronos 是一个由成员提供资金的行业协会，专注于开放媒体标准的创建和维护。目前，Khronos 负责 OpenGL 的发展和升级。

《OpenGL 编程指南》就是由 Khronos 小组编写的官方指南，是 OpenGL 领域的权威著作，有“OpenGL 红宝书”之称，曾经帮助许多程序员走上了 OpenGL 专家之路。第 7 版在第 6 版的基础上又有所改进，介绍了 OpenGL 3.0 和 OpenGL 3.1 的新的和更新的内容。

本书历经多次版本升级，其中文版的翻译也是一项延续性的工作，凝结了许多人的辛勤工作。徐波等曾承担《OpenGL 编程指南》第 5 版和第 6 版的主要翻译工作。李军在第 6 版的中文版的基础上，负责了第 7 版新增内容的翻译和更新工作。参与第 7 版翻译工作的还有刘金华、刘伟超、罗庚臣、刘二然、郑芳菲、庄逸川、王世高、郭莹、陈、邓勇、何进伟、贾晓斌、汪蔚和齐国涛。机械工业出版社华章分社的编辑为本书的出版付出了辛勤劳动，感谢他们！

译者

2009 年 10 月

前言

OpenGL（Graphics Library，GL 图形库）图形系统是图形硬件的一个软件接口，它允许我们创建交互性的程序，产生移动三维物体的彩色图像。使用 OpenGL，我们可以对计算机图形技术进行控制，产生逼真的图像或者虚构出现实世界没有的图像。本书解释了如何使用 OpenGL 图形系统进行编程，实现所需要的视觉效果。

本书内容

本书共分 15 章。前 5 章描述了一些基本信息，读者需要理解这些内容，才能在场景中绘制正确着色和光照的三维物体。

第 1 章对 OpenGL 可以实现的功能进行简要的介绍。该章还提供了一个简单的 OpenGL 程序，并介绍需要了解的一些基本编程细节，有助于学习后续章节的内容。

第 2 章解释如何创建一个物体的三维几何图形描述，并最终把它绘制到屏幕上。

第 3 章描述三维模型在绘制到二维屏幕之前如何进行变换。我们可以控制这些变换，显示模型的特定视图。

第 4 章描述如何指定颜色以及用于绘制物体的着色方法。

第 5 章解释如何控制围绕一个物体的光照条件，以及这个物体如何对光照作出反应（也就是说，它是如何反射或吸收光线的）。光照是一个重要的主题，因为物体在没有光照的情况下看上去往往没有立体感。

接下来的几章说明如何对三维场景进行优化以及如何添加一些高级特性。在没有精通OpenGL之前，读者可以选择不使用这些高级特性。有些特别高级的主题在出现时会有特殊的标记。

第6章描述创建逼真场景所需要的一些基本技巧：alpha混合（创建透明物体）、抗锯齿（消除锯齿状边缘）、大气效果（模拟雾和烟雾）以及多边形偏移（在着重显示填充多边形的边框时消除不良视觉效果）。

第7章讨论如何存储一系列的OpenGL命令，用于在以后执行。我们可以使用这个特性来提高OpenGL程序的性能。

第8章讨论如何操作表示位图或图像的二维数据。位图的一种常见用途就是描述字体中的字符。

第9章解释如何把称为纹理的一维、二维和三维图像映射到三维物体表面。纹理贴图可以实现许多非常精彩的效果。

第10章描述OpenGL实现中可能存在的所有缓冲区，并解释如何对它们进行控制。我们可以使用这些缓冲区实现诸如隐藏表面消除、模板、屏蔽、运动模糊和景深聚焦等效果。

第11章显示了如何使用GLU（OpenGL Utility Library，OpenGL工具函数库）中的分格化和二次方程函数。

第12章介绍生成曲线和表面的高级技巧。

第13章说明如何使用OpenGL的选择机制来选择屏幕上的一个物体。此外，该章还解释了反馈机制，它允许我们收集OpenGL所产生的绘图信息，而不是在屏幕上绘制物体。

第14章描述如何用巧妙的或意想不到的方法来使用OpenGL，产生一些有趣的结果。这些技巧是通过对OpenGL及其技术前驱Silicon Graphics IRIS图形函数库的多年应用和实践总结出来的。

第15章讨论OpenGL 2.0所引入的变化，包括对OpenGL着色语言的介绍。OpenGL着色语言通常又称为GLSL，它允许对OpenGL的顶点和片断处理阶段进行控制。这个特性可以极大地提高图像的质量，充分体现OpenGL的计算威力。

另外，本书还包括几个非常实用的附录：

附录A讨论了用于处理窗口系统操作的函数库。GLUT（OpenGL实用工具库）具有可移植性，它可以使代码更短、更紧凑。

附录B列出了OpenGL所维护的状态变量，并描述了如何获取它们的值。

附录C介绍了隐藏在矩阵转换后面的一些数学知识。

附录 D 简单描述了窗口系统特定的函数库所提供的函数，它们进行了扩展，以支持 OpenGL 渲染。本附录讨论了 X 窗口系统、Apple 的 Mac OS 和 Microsoft Windows 的窗口系统接口。

附录 E 对 OpenGL 所执行的操作提供了一个技术性的浏览，简要描述了当应用程序执行时这些操作的出现顺序。

?附录 F 列出了一些基于 OpenGL 设计者思路的编程提示，这些可能对读者有用。

附录 G 描述了 OpenGL 实现在什么时候以及什么地方必须生成 OpenGL 规范所描述的精确像素。

附录 H 描述了如何计算不同类型的几何物体的法线向量。

附录 I 列出了 OpenGL 着色语言所提供的所有内置的变量和函数。

附录 J 介绍了各种浮点数、共享指数像素和纹理单元格式。

附录 K 介绍了存储单成分和双成分压缩纹理的纹理格式。

附录 L 介绍了 GLSL 1.40 的 uniform 变量缓存区的标准内存布局。

最后，本书还提供了一个术语表，对本书所使用的一些关键术语进行了定义。

第 7 版的新增内容

本书包含了 OpenGL 3.0 和 OpenGL 3.1 的新的和更新的内容。通过这些版本（这也是本书值得庆祝的 18 岁生日），OpenGL 经历了与其之前的版本最显著的改变。3.0 版添加了很多新的功能，并且添加了废弃模型，它建立了一种方法把陈旧的功能从库中删除。注意，只有新功能添加到了 3.0 版中，才会使其在源代码和二进制文件上都和之前的版本向后兼容。然而，很多功能标记为废弃的，表示可能在 API 未来的版本中删除。

本书介绍的和 OpenGL 3.0 相关的更新内容包括：

OpenGL 中的新功能：

OpenGL 着色语言更新，创建了 GLSL 1.30 版。

条件渲染。

对映射缓冲区对象的内存的细粒度访问以用于更新和读取。

除了纹理图像格式（在 OpenGL 2.1 中加入），还有用于帧缓冲区的浮点数像素格式。

帧缓冲区和渲染缓冲区对象。

为小的动态范围数据采用紧凑的浮点表示，以减少内存存储占用。

改进了对复制数据时的多采样缓冲区交互的支持。

纹理图像和渲染缓冲区中的非规范化的整数值保留它们最初的表示，相对于 OpenGL 将这些值映射到范围[0,1]的常规操作。

支持一维纹理数组和二维纹理数组。

附加的包装像素格式支持访问新的渲染缓冲区。

针对多渲染目标，分开混合和写屏蔽控制。

纹理压缩格式。

纹理的单成分和双成分的内部格式。

转换反馈。

顶点数组对象。

sRGB 帧缓冲区格式。

废弃模式的深入讨论。

修复错误并更新标记名。

对于 OpenGL 3.1：

标识出 OpenGL 3.0 中因废弃而要删除的功能。

新的功能：

OpenGL 着色语言更新，创建了 GLSL1.40 版。

实例化渲染。

缓冲区之间高效的服务器端数据复制。

在单个调用作用渲染多个类似的图元，用一个特殊标记（由用户指定）来表示何时重新启动一个图元。

纹理缓冲区对象。

纹理矩形。

uniform 缓冲区对象。

带符号的规范化纹理单元格式。

阅读本书所需要的预备知识

本书假定读者知道如何使用 C 语言编写程序，并且了解一些数学背景知识（几何、三角、线性代数、积分和微分几何）。即使读者对计算机图形技术领域了解不多，仍然能够理解本书所讨论的绝大部分内容。当然，计算机图形学是一个巨大的主题，因此读者最好能够扩展自己在这个领域的知识。下面是我们推荐的两本参考书：

《Computer Graphics: Principles and Practice》：作者 James D.Foley、Andries van Dam、Steven K.Feiner 和 John F.Hughes（Addison-Wesley，1990）。该书是计算机图形学领域的百科全书，它包括了丰富的信息。但是，读者在阅读这本书之前最好已经对计算机图形学有一定程度的了解。

《3D Computer Graphics》：作者 Andrew S. Glassner（The Lyons Press，1994）。该书对计算机图形学进行了非技术性的、通俗易懂的介绍。这本书重点介绍计算机图形学可以实现的视觉效果，而不是实现这些效果所需要的技术。

另外，读者还可以访问 OpenGL 的官方网站。这个网站包含了各种类型的通用信息，包括软件、示例程序、文档、FAQ、讨论版和新闻等。如果读者遇到任何 OpenGL 问题，这是首先应该想到的去处：http://www.opengl.org/。

另外，OpenGL 官方网站记录了组成 OpenGL 3.0 版和 OpenGL 3.1 版所有过程的完整文档。这些文档代替了 OpenGL 体系结构审核委员会和 Addison Wesley 出版的《OpenGL Reference Manual》（OpenGL 参考手册）。

OpenGL 实际上是一种独立于硬件的程序接口规范。在一种特定类型的硬件上，所使用的是它的一种特定实现。本书解释了如何使用所有的 OpenGL 实现进行编程。但是，由于各种 OpenGL 实现可能存在微小的差别（例如，在渲染性能以及提供额外的、可选的特性方面），因此读者可能需要寻找与自己所使用的特定 OpenGL 实现相关的补充文档。另外，读者所使用的系统还可能提供了与 OpenGL 相关的实用函数库、工具箱、编程和调试支持工具、各种窗口部件、示例程序和演示程序等。

如何获取本书示例程序的源代码

本书包含了许多示例程序，以说明各种 OpenGL 编程技巧的具体用法。由于本书的读者背景各不相同，从新手到老手，既有计算机图形学的知识又有 OpenGL 的知识，本书中的示例通常都展示了一个特定渲染条件下的最简单的方法，并且使用 OpenGL 3.0 接口来说明。这么做的目的主要是为了让那些刚开始接触 OpenGL 的读者能够更直接和更容易地接受本书所介绍的内容。对于那些有着广泛经验、想要寻找使用最新的 API 功能实现的读者，我们首先感谢你耐心地阅读那些早已掌握的内容，并且建议访问我们的 Web 站点：http://www.opengl-redbook.com/。

在那里，你将会找到本书中所有示例的源代码，使用最新功能的实现，以及介绍从一个版本的 OpenGL 迁移到另一个版本所需的修改的额外讨论。

本书中包含的所有程序都使用了 OpenGL Utility Toolkit (GLUT)，其最初的作者是 Mark Kilgard。对于这个版本，我们使用了开发 freeglut 项目的人们所编写的 GLUT 接口的开源版本。他们扩展了 Mark 最初的工作（那些工作在 Mark Kilgard 的《OpenGL Programming for the X Window System》Addison-Wesley，1996，一书中有详细的介绍）。可以从如下地址找到他们的开源项目页面：http://freeglut.sourceforge.net/。

可以从这个站点获取它们的实现的代码和二进制文件。

本书 1.6 节和附录 A 给出了关于使用 GLUT 的更多信息。其他有助于更快地学习和使用 OpenGL 和 GLUT 编程的资源，可以在 OpenGL Web 的资源页面找到：http://www.opengl.org/resources/。

许多 OpenGL 实现还可能包含一些示例代码，作为系统的一部分。这些源代码可能是这种 OpenGL 实现应该使用的最佳代码，因为它可能针对当前的系统进行了优化。读者可以参阅与自己使用的系统相关的 OpenGL 文档，了解从哪里下载这些示例程序。

Nate Robin 的 OpenGL 教程

Nate Robin 编写了一套教学程序，用于演示 OpenGL 编程的基本概念。它允许用户修改函数的参数，以交互的方式观察它们的效果。这套教程所涵盖的主题包括变换、光照、雾和纹理。我们极力推荐使用这些教程。它们具有可移植性，并且使用了前面所提到的 GLUT。要获取这些教程的源代码，可以访问下面这个网站：

http://www.xmission.com/~nate/tutors.html

勘误表

本书也会存在一些错误。此外，在本书出版过程中，OpenGL 也在更新：随着规范和新规范的发布，错误也会更正并加以澄清。我们在 Web 站点 http://www.opengl-redbook.com/ 维护了一个错误和更新的列表，那里还提供了工具来报告你可能会发现的任何新的错误。如果你发现一个错误，请接受我们的道歉，并且提前感谢你的报告，我们将尽快地更正。

约定字体

本书使用如下的约定字体：

粗体—表示命令和函数的名称和矩阵。

斜体—变量、参数、参数名、空间维度、矩阵成员和第一次出现的关键术语。

常规字体—每句类型和定义的常量。

代码示例以等宽字体显示，命令概览放在专门的框线中。

在命令概览部分，花括号表示可选的数据类型。在下面的例子中，glCommand 具有 4 个可能的后缀：s、i、f 和 d，分别表示数据类型 GLshort、GLint、GLfloat 和 GLdouble。在 glCommand 函数的原型中，TYPE 表示这些后缀所提示的数据类型。

void glCommand{sifd}(TYPE x1, TYPE y1, TYPE x2, TYPE y2);

区分废弃的功能

正如前面所提到的，本书的这一版本主要针对 OpenGL 3.0 和 OpenGL 3.1。OpenGL 3.0 对于目前可用的任何版本完全向后兼容。然而，OpenGL 3.1 采用了废弃模式，删除了很多与现代图形系统不太兼容的旧功能。尽管很多功能从“核心的”OpenGL 中删除了，为了消除版本之间的过渡，OpenGL ARB 发布了 GL_ARB_compatibility 扩展。如果你的实现支持这个扩展，它将能够使用所有删除的功能。为了便于识别那些从 OpenGL 3.1 中删除了的、仍然得到兼容扩展支持的功能，在本书中，介绍命令或函数的边框的旁边会给出一个信息表格，其中列出受到影响的函数或符号。

尽管有些功能从 OpenGL 中废弃并删除了，但一些这样的功能影响着库，例如 OpenGL Utility Library，通常称之为 GLU。这些在 OpenGL 3.1 的变化中受到影响的函数，也会在旁边的表格中列出。

第 1 章 OpenGL 简介

本章目标

大致了解 OpenGL 的功能。

了解不同程度的渲染复杂性。

理解 OpenGL 程序的基本结构。

了解 OpenGL 函数的语法。

了解 OpenGL 渲染管线的操作序列。

大致了解如何在 OpenGL 程序中实现动画。

本章对 OpenGL 进行了简单的介绍，主要包含下面几节：

什么是 OpenGL：介绍 OpenGL 是什么，它能够做什么，不能够做什么，以及它的工作原理。

一段简单的 OpenGL 代码：展示一个小型的 OpenGL 程序，并对它进行了简单的讨论，并定义了一些基本的计算机图形术语。

OpenGL 函数的语法：解释 OpenGL 函数所使用的一些约定和记法。

OpenGL 是一个状态机：描述 OpenGL 状态变量的用法，并介绍一些查询、启用和禁用 OpenGL 状态的函数。

OpenGL 渲染管线：展示一个用于处理几何和图像数据的典型操作序列。

与 OpenGL 相关的函数库：介绍一些与实用 OpenGL 相关的函数，包括对 GLUT（GraphicsLibrary Utility Toolkit，一种可移植的工具库）的详细介绍。

动画：简单介绍如何创建能够在屏幕上移动的图片。

OpenGL 及其废弃机制：介绍在 OpenGL 的最新版本中有哪些废弃修改，这些修改如何影响到应用程序，以及根据这些修改，OpenGL 未来会如何发展。

1.1 什么是 OpenGL

OpenGL 是图形硬件的一种软件接口。这个接口包含的函数超过 700 个（纳入 OpenGL 3.0 的函数大约有 670 个，另外 50 个函数位于 OpenGL 工具库中），这些函数可以用于指定物体和操作，创建交互式的三维应用程序。

OpenGL 的设计目标就是作为一种流线型的、独立于硬件的接口，在许多不同的硬件平台上实现。为了实现这个目标，OpenGL 并未包含用于执行窗口任务或者获取用户输入之类的函数。反之，必须通过具体的窗口系统来控制 OpenGL 应用程序所使用的特定硬件。类似地，OpenGL 并没有提供用于描述三维物体模型的高级函数。这类函数可能允许指定相对较为复杂的形状，例如汽车、身体的某个部位、飞机或分子等。在 OpenGL 中，程序员必须根据一些为数不多的基本几何图元（如点、直线和多边形）来创建所需要的模型。

当然，程序员可以在 OpenGL 的基础之上，创建提供这些特性的高级函数库。OpenGL 工具库（GLU）提供了许多建模功能，例如二次曲面以及 NURBS 曲线和表面。GLU 是所有 OpenGL 实现的一个标准组成部分。

既然读者已经知道了 OpenGL 不能够做什么，现在我们来讨论它可以做什么。可以看一下本书所附的彩图，它们展示了 OpenGL 的典型用法。本书封面上的场景就是在一台计算机上使用 OpenGL 通过一系列复杂的方法渲染（即绘制）产生的。下面，我们简单地说明这些图像是如何产生的。

彩图 1 使用线框模型（wireframe model）显示整个场景。也就是说，场景中的所有物体都是用线型构成的。每条线对应于图元（一般为多边形）的一条边。例如，桌子的表面就是由许多三角形构成的，这些三角形就像切开的蛋糕一样排列在一起。

注意，如果物体是实心的而不是线框的，可能只能看到物体的一部分。例如，在这张彩图中，可以看到窗外小山坡的整个模型。但是，在正常情况下，这个模型的大部分会被房间

的墙壁遮挡。图中的地球仪看上去几乎是实心的，因为它是由几百个彩色块组成的。读者可以看到所有彩色块的所有边的线框，甚至可以看到构成地球仪背面的那些彩色块。这个地球仪的构建方式提供了一种思路，就是通过组装底层的简单物体来创建复杂的物体。

彩图 2 显示了同一个线框场景的深度提示（depth-cued）版本。注意，距离眼睛较远的线型看上去更暗淡一些，显然这更接近于现实。它提供了一种叫做深度的视觉提示。OpenGL 使用各种大气效果（合称为雾）来实现深度提示。

彩图 3 显示了这个线框场景的抗锯齿（antialiased）版本。抗锯齿是一种用于消除锯齿状边缘的技术。锯齿状边缘是在使用像素（pixel，它是图片元素的缩写，指一个矩形网格大小）近似地模拟平滑边缘时产生的。当直线接近水平或垂直时，这种锯齿现象通常最为明显。

彩图 4 显示了这个场景进行单调着色（flat-shading）后的不带光照的版本。现在，场景中的物体以实心的形式显示。它们在场景中看上去像是平面的，这是因为每个多边形只用一种颜色进行渲染。因此，它们之间的过渡显得不够平滑。此外，这个场景也没有应用任何光照效果。

彩图 5 显示了这个场景使用平滑着色（smooth-shading）并且带光照的版本。注意，当物体根据房间内的光源进行着色时，它们看上去更为逼真，而且更具立体效果，它们的表面就像被磨圆
了一样。

彩图 6 在前一个版本场景的基础上添加了阴影（shadow）和纹理（texture）效果。阴影并不是 OpenGL 特性（并不存在“阴影函数”），但是可以使用第 9 章和第 14 章所描述的技巧自行创建这种效果。纹理贴图（texture mapping）允许把一幅二维图像应用到一个三维物体上。在这个场景中，桌面就是一个典型的纹理贴图例子。地板和桌面上的木纹都是使用纹理贴图的结果，墙面和桌上玩具的表面也是如此。

彩图 7 显示了这个场景中一个运动模糊（motion-blurred）的物体。图中的积木好像是在向前运动，它们的背后留下了一道模糊的运动轨迹。

彩图 8 从另一个角度显示了本书封面的那个场景。这张图说明了图像实际上只不过是三维物体模型的一张快照而已。

彩图 9 再次使用了雾，用它来模拟空气中的烟尘。彩图 2 已经显示了雾的效果，但与之相比，彩图 9 的雾效果更为明显。

彩图 10 显示了景深效果（depth-of-field effect），它模拟了照相机无法对场景中的所有物体进行聚焦的现象。当照相机聚焦于场景中一个特定的点时，远离这个点的物体看起来就会比较模糊。

看了这些彩图之后，读者应该对 OpenGL 图形系统的功能有了一个大致的了解。下面，我们简单地描述当 OpenGL 对场景中的图像进行渲染时所执行的主要图形操作。（请参见 1.5 节，了解和这些操作顺序有关的详细信息。）

1) 根据几何图元创建形状，从而建立物体的数学描述。（OpenGL 把点、直线、多边形和位图作为基本的图元。）

2) 在三维空间中排列物体，并选择观察复合场景的有利视角。

3) 计算所有物体的颜色。颜色可以由应用程序明确指定，可以根据特定的光照条件确定，也可以通过把纹理贴到物体的表面而获得，或者是上述三种操作的混合产物。这些操作可能使用着色器来执行，这样可以显式地控制所有的颜色计算，或者可能使用 OpenGL 的预编程算法在其内部执行（我们常用术语固定功能的管线来表示后者）。

4) 把物体的数学描述以及与物体相关的颜色信息转换为屏幕上的像素。这个过程叫做光栅化（rasterization）。

在这些阶段期间，OpenGL 可能还会执行其他操作，例如消除被其他物体所遮挡的物体（或该物体的一部分）。此外，在场景被光栅化之后但在绘制到屏幕之前，仍然可以根据需要对像素数据执行一些操作。

在有些 OpenGL 实现（例如 X 窗口系统的 OpenGL 实现）中，OpenGL 必须实现这样一个目标：显示程序员所创建的图形的计算机和运行图形程序的计算机可以不相同。由多台计算机通过一个数字网络彼此连接在一起所组成的网络计算机环境就属于这种情况。在这种情况下，运行图形程序并发出绘图命令的计算机称为客户机，接收这些命令并执行绘图任务的计算机称为服务器。客户机发送给服务器的命令的传输格式（称为协议）总是相同的，因此 OpenGL 程序可以通过网络运行，即使客户机和服务器并不是同种类型的计算机。如果 OpenGL 程序并不是通过网络运行的，那就只涉及一台计算机，它既是客户机也是服务器。

1.2 一段简单的 OpenGL 代码

由于 OpenGL 图形系统的功能非常强大，因此 OpenGL 程序可能相当复杂。但是，许多实用的 OpenGL 程序的基本结构可能非常简单，它的任务就是初始化一些状态（这些状态用于控制 OpenGL 的渲染方式），并指定需要进行渲染的物体。

在察看具体的 OpenGL 代码之前，首先介绍几个术语。渲染（rendering）是计算机根据模型创建图像的过程，我们已经在前面看到过这个术语。模型（model）是根据几何图元创建的，也称为物体（object）。几何图元包括点、直线和多边形等，它们是通过顶点（vertex）指定的。

最终完成了渲染的图像是由在屏幕上绘制的像素组成的。像素是显示硬件可以在屏幕上显示的最小可视元素。在内存中，和像素有关的信息（例如像素的颜色）组织成位平面（bitplane）的形式。

位平面是一块内存区域，保存了屏幕上每个像素的 1 个位的信息。例如，它指定了一个特定像素的颜色中红色成分的强度。位平面又可以组织成帧缓冲区（framebuffer）的形式，后者保存了图形硬件为了控制屏幕上所有像素的颜色和强度所需要的全部信息。

现在我们观察一个简单的 OpenGL 程序。示例程序 1-1 在黑色背景中渲染了一个白色的矩形，如图 1-1 所示。

图 1-1 黑色背景中的白色矩形

示例程序 1-1 一段 OpenGL 代码

```
1. #include <whateverYouNeed.h>
2. main(){
3. InitializeAWindowPlease();
4. glClearColor(0.0,0.0,0.0,0.0);
5. glClear(GL_COLOR_BUFFER_BIT);
6. glColor3f(1.0,1.0,1.0);
7. glOrtho(0.0,1.0,0.0,1.0,-1.0,1.0);
8. glBegin(GL_POLYGON);
9. glVertex3f(0.25,0.25,0.0);
10. glVertex3f(0.75,0.25,0.0);
11. glVertex3f(0.75,0.75,0.0);
12. glVertex3f(0.25,0.75,0.0);
13. glEnd();
14. glFlush();
15. UpdateTheWindowAndCheckForEvents();
16. }
```

main()函数的第一行代码在屏幕上初始化一个窗口：InitializeAWindowPlease()函数是一个占位符，是一个特定于窗口系统的函数，它通常并不是 OpenGL 函数。接下来的两个函数都是 OpenGL 函数，它们把窗口颜色清除为黑色：glClearColor()函数确定了窗口将清除成什么颜色，而 glClear()函数实际完成清除窗口的任务。在设置了清除颜色之后，以后每次调用

glClear()时，窗口就会清除为这种颜色。当然，可以再次调用 glClearColor()函数，更改当前的清除颜色。类似地，glColor3f()函数确定了绘制物体时所使用的颜色（在此例中为白色）。此后，所有绘制的物体都将使用这种颜色，除非再次调用这个函数更改绘图颜色。

这个程序所使用的下一个 OpenGL 函数是 glOrtho()，它指定了 OpenGL 在绘制最终图像时所使用的坐标系统（cooridinate system），决定了图像将如何映射到屏幕上。接下来的几个函数位于一对 glBegin()和 glEnd()调用之间，它们定义了要绘制的物体，在本例中是一个具有 4 个顶点的多边形。这个多边形的“角”是由 glVertex3f()函数定义的。该函数所使用的参数表示（x, y, z）坐标。根据这些值，可以猜出这个多边形是一个位于 z = 0 平面的矩形。

最后，glFlush()函数保证了绘图命令将实际执行，而不是存储在缓冲区中等待其他的 OpenGL 命令。UpdateTheWindowAndCheckForEvents()也是一个占位符函数，它管理窗口的内容，并开始进行事件处理。

实际上，这段 OpenGL 代码的结构并不是很好。读者可能会问：“如果我试图移动或改变窗口的大小，它会变成什么样子？”或者“当我每次绘制这个矩形时，是不是需要重置坐标系统？”在本章的后面，我们将用实际执行任务的代码来替换 InitializeAWindowPlease()和 UpdateTheWindowAndCheckForEvents()占位符函数。这就要求我们对这段代码的结构进行修改，使它更有效率。

1.3 OpenGL 函数的语法

在前一节的那个简单程序中，读者可能已经发现 OpenGL 使用了前缀“gl”，并把组成函数的每个单词的首字母用大写形式表示（例如，glClearColor()）。类似地，OpenGL 还定义了一些以前缀 GL_开头的常量，所有的单词都使用大写形式，并以下划线分隔（例如 GL_COLOR_BUFFER_BIT）。

除此之外，读者可能还注意到有些 OpenGL 函数中有一些似乎不相关的字母（例如 glColor3f()和 glVertex3f()中的“3f”）。确实，glColor3f()函数名中的“Color”部分就足以定义这个用于设置当前绘图颜色的函数。但是，OpenGL 定义了这个函数的多个不同版本，以便使用不同类型的参数。具体地说，这个后缀中的“3”表示这个函数接受 3 个参数。Color 函数还存在接受 4 个参数的版本。这个后缀中的“f”表示这些参数都是浮点数。OpenGL 之所以为同一个函数定义了不同参数类型的版本，是为了允许用户根据自己的数据格式向 OpenGL 传递参数。

有些 OpenGL 函数可以在它们的参数中接受多达 8 种不同的数据类型。表 1-1 列出了一些后缀字母，它们分别指定了 OpenGL 的 ISO C 实现所提供的数据类型。此外，表 1-1 还列出了对应的 OpenGL 类型定义。读者所使用的 OpenGL 实现可能并不完全与这种方案相对应。例如，OpenGL 的 C++或 Ada 实现就不需要完全遵循这种方案。

表 1-1 函数后缀和参数数据类型

因此，下面这两个函数调用

后缀	数据类型	典型的对应C语言类型	OpenGL
b	8位整数	signed char	GLbyte
s	16位整数	short	GLshort
i	32位整数	int或long	GLint, GLsiz
f	32位浮点数	float	GLfloat, GLc
d	64位浮点数	double	GLdouble, G
ub	8位无符号整数	unsigned char	GLubyte, GL
us	16位无符号整数	unsigned short	GLushort
ui	32位无符号整数	unsigned int或unsigned long	GLuint, GLe

```
1. glVertex2i(1, 3);
2. glVertex2f(1.0, 3.0);
```

是等价的。只不过第一个函数把顶点的坐标指定为 32 位的整数，第二个函数则把它们指定为单精度的浮点数。

注意：不同的 OpenGL 实现在选择用哪些 C 数据类型来表示 OpenGL 数据类型方面存在一些差异。如果坚持在自己的应用程序中使用 OpenGL 定义的数据类型，那么在不同的 OpenGL 实现之间移植代码时，就可以避免类型不匹配的问题。

有些 OpenGL 函数名的最后还有一个字母 v，它表示这个函数所接受的参数是一个指向值向量（或数组）的指针，而不是一系列的单独参数。许多函数既有向量版本也有非向量版本，也有一些函数只接受单独的参数，另外还有一些函数要求至少有 1 个参数被指定为向量。下面这几行代码显示了既可以使用向量版本也可以使用非向量版本的函数来设置当前的绘图颜色：

```
1. glColor3f(1.0, 0.0, 0.0);
2. GLfloat color_array[] = {1.0, 0.0, 0.0};
3. glColor3fv(color_array);
```

最后，OpenGL 还定义了 GLvoid 类型。这种类型最常用于那些接受指向值数组的指针为参数的 OpenGL 函数。

在本书的剩余部分（除了实际的示例代码），我们将只用基本名称来表示 OpenGL 函数，并且加个星号表示它还有多个不同的版本。例如，glColor*()表示用于设置当前颜色的函数

的所有版本。如果想强调所使用的是一个函数的某个特定版本，可以加上必要的后缀来表示这个版本。例如，glVertex*v()表示用于指定顶点的函数的所有向量版本。

1.4 OpenGL 是一个状态机

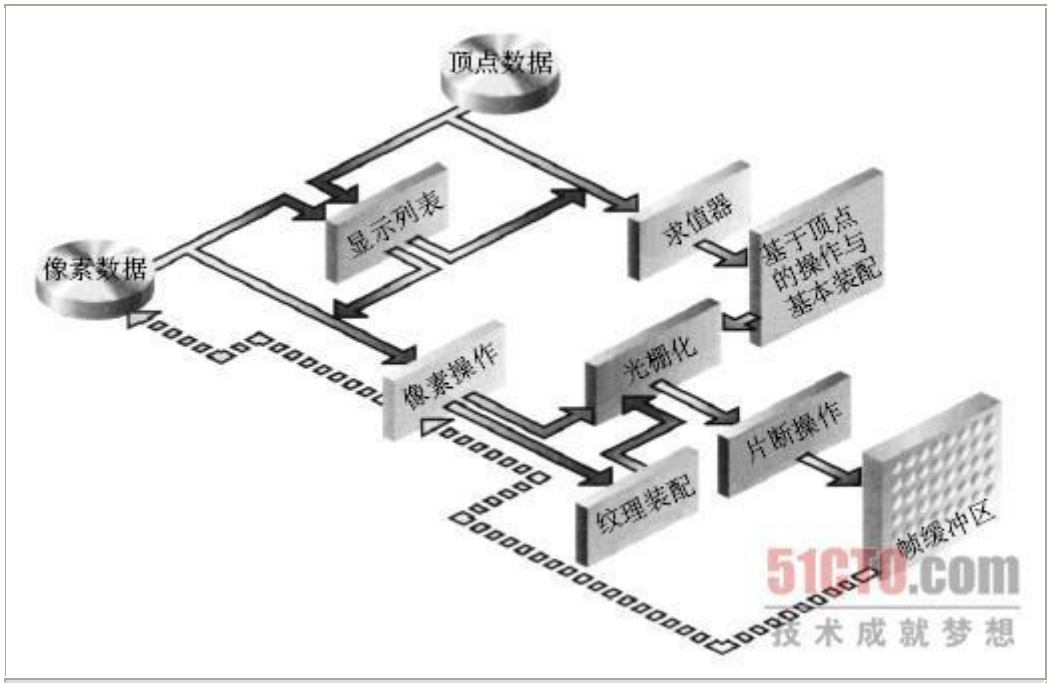
OpenGL 是一个状态机，尤其是如果你使用固定功能的管线。可以设置它的各种状态（或模式），然后让这些状态一直生效，直到再次修改它们。正如所看到的那样，当前颜色就是一个状态变量。可以把当前颜色设置为白色、红色或其他任何颜色，在此之后绘制的所有物体都将使用这种颜色，直到再次把当前颜色设置为其他颜色。当前颜色只是 OpenGL 所维护的许多状态变量之一。其他的状态变量还有很多，并且有着各自的用途，例如控制当前视图和投影变换、直线和多边形点画模式、多边形绘图模式、像素包装约定、光照的位置和特征以及被绘制物体的材料属性等。许多表示模式的状态变量可以用 glEnable()和 glDisable()函数进行启用和禁用。

如果使用可编程的着色器，根据所用的 OpenGL 版本的不同，着色器所能识别的状态的数量也有所不同。每个状态变量（或模式）都有一个默认值。在任何时候都可以向系统查询每个状态变量的当前值。一般情况下，可以使用下面这 6 个函数之一来完成这个任务：glGetBooleanv()、glGetDoublev()、glGetIntegerv()、glGetfloatv()、glGetPointerv()或 glIsEnabled()。具体选择的函数取决于希望返回的结果的数据类型。有些状态变量还有更为特定的查询函数（例如 glGetLight*()、glGetError()或 glGetPolygonStipple()等）。另外，还可以使用 glPushAttrib()、glPushClientAttrib()函数把状态变量的集合保存到一个属性栈中，对它们进行临时的修改，以后再用 glPopAttrirb()或 glPopClientAttrib()恢复这些值。如果需要对状态变量进行临时修改，就应该使用这些函数，而不是使用任何查询函数，因为前者的效率更高。

附录 B 列出了可以查询的状态变量的完整列表。对于每个状态变量，附录 B 还列出了返回这个变量值所建议使用的 glGet*()函数、这个变量所属的属性类以及它的默认值。

1.5 OpenGL 渲染管线

绝大多数 OpenGL 实现都有相似的操作顺序，一系列相关的处理阶段叫做 OpenGL 渲染管线。图 1-2 显示了这些顺序，虽然并没有严格规定 OpenGL 必须采用这样的实现，但是它提供了一个可靠的指南，可以预测 OpenGL 将以什么样的顺序来执行这些操作。



（点击查看大图）图 1-2 操作顺序

如果读者刚开始涉足三维图形编程，可能会对接下来的内容感到吃力。现在可以跳过这一部分内容，但是在读完本书的每一章时，都应该重温一下图 1-2。

图 1-2 显示了 Henry Ford 在福特汽车公司所采用的装配线方法，它也是 OpenGL 处理数据的方法。几何数据（顶点、直线和多边形）所经历的处理阶段包括求值器和基于顶点的操作，而像素数据（像素、图像和位图）的处理过程则有所不同。在最终的像素数据写入到帧缓冲区之前，这两种类型的数据都将经过相同的最终步骤（光栅化和基于片断的操作）。

下面，我们更为详细地介绍 OpenGL 渲染管线的一些关键阶段。

1.5.1 显示列表

任何数据，不论它描述的是几何图元还是像素，都可以保存在显示列表（display list）中，供当
前或以后使用。当然，也可以不把数据保存在显示列表中，而是立即对数据进行处理，这种模式也叫做立即模式（immediate mode）。当一个显示列表执行时，保存的数据就从显示列表中取出，就像在立即模式下直接由应用程序发送的那样。关于显示列表的详细内容，请参见第 7 章。

1.5.2 求值器

所有的几何图元最终都要通过顶点来描述。参数化曲线和表面最初可能是通过控制点以及叫做基函数（basic function）的多项式函数进行描述的。求值器提供了一种方法，根据控制点计算表示表面的顶点。这种方法是一种多项式映射，它可以根据控制点产生表面法线、纹理坐标、颜色以及空间坐标值。关于求值器的详细内容，请参阅第 12 章。

1.5.3 基于顶点的操作

对于顶点数据，接下来的一个步骤是“基于顶点的操作”，就是把顶点变换为图元。有些类型的顶点数据（例如空间坐标）是通过一个 4×4 的浮点矩阵进行变换的。空间坐标从 3D 世界的一个位置投影到屏幕上的一个位置。有关变换矩阵的详细内容，请参阅第 3 章。

如果启用了高级特性，这个阶段将更为忙碌。如果使用了纹理，这个阶段还将生成并变换纹理坐标。如果启用了光照，就需要综合变换后的顶点、表面法线、光源位置、材料属性以及其他光照信息进行光照计算，产生最终的颜色值。

从 OpenGL 2.0 开始，对于使用固定功能的顶点处理，我们有了新的选择，正如前面所提到的，可以使用顶点着色器来完全控制基于顶点的操作。如果使用了着色器，基于顶点的操作阶段中的所有操作都会由着色器取代。在 OpenGL 3.1 中，所有固定功能的顶点操作都删除了（除非具体实现支持 GL_ARB_compatibility 扩展），必须使用顶点着色器。

1.5.4 图元装配

图元装配的一个主要内容就是裁剪，它的任务是消除位于半空间（half-space）之外的那部分几何图元，这个半空间是由一个平面所定义的。点裁剪就是简单地接受或拒绝顶点，直线或多边形裁剪则可能需要添加额外的顶点，具体取决于直线或多边形是如何进行裁剪的。

在有些情况下，接下来需要执行一个叫做透视除法（perspective division）的步骤。它使远处的物体看起来比近处的物体更小一些。接下来所进行的是视口（viewport）和深度（z 坐标）操作。如果启用了剔除功能（culling）并且该图元是一个多边形，那么它就有可能被剔除测试所拒绝。根据多边形模式，多边形可能画成点的形式或者直线的形式。请参见 2.4.3 节，了解这方面的详细信息。

这个阶段产生的结果就是完整的几何图元，也就是根据相关的颜色、深度（有时还有纹理坐标值以及和光栅化处理有关的一些指导信息）进行了变换和裁剪的顶点。

1.5.5 像素操作

在 OpenGL 的渲染管线中，和单路径的几何数据相比，像素数据所经历的流程有所不同。首先，来自系统内存的一个数组中的像素进行解包，从某种格式（像素的原始格式可能有多种）解包为适当数量的数据成分。接着，这些数据被缩放、偏移，并根据一幅像素图进行处理。处理结果先进行截取，
然后或者写入到纹理内存，或者发送到光栅化阶段。详细内容请参阅 8.3 节。

如果像素数据是从帧缓冲区读取的，就对它们执行像素转换操作（缩放、偏移、映射和截取）。然后，这些结果被包装为一种适当的格式，并返回到系统内存的一个数组中。

OpenGL 有几种特殊的像素复制操作，可以把数据从帧缓冲区复制到帧缓冲区的其他位置或纹理内存中。这样，在数据写入到纹理内存或者写回到帧缓冲区之前，只需要进行一道像素转换就可以了。

所介绍的很多像素操作都是固定功能的像素管线的一部分，并且常常会在系统中来回移动大量数据。现代的图形实现倾向于通过尝试把图形操作集中到位于图形硬件中的内存，从而优化性能（当然，这一描述是一般性的，但是，当前的大多数系统正是这样实现的）。OpenGL 3.0 支持所有这些操作，并且引入了帧缓冲对象来帮助优化这些数据移动，特别是，这些对象可以完全避免某些数据移动。帧缓冲对象和可编程的片段着色器组合到一起，替代了很多这样的操作（尤其显著的是那些划分为像素转移的操作），并且大大增强了灵活性。

1.5.6 纹理装配

OpenGL 应用程序可以在几何物体上应用纹理图像，使它们看上去更为逼真。如果需要使用多幅纹理图像，把它们放在纹理对象中是一种明智的做法。这样，就可以很方便地在它们之间进行切换。几乎所有的 OpenGL 实现都拥有一些特殊的资源，可以加速纹理的处理（这些资源可能是图形实现中从一个共享资源池中分配而来的）。为了帮助 OpenGL 实现高效地管理这些内存资源，优先使用纹理对象来帮助控制纹理贴图潜在的缓存和定位问题，详见第 9 章。

1.5.7 光栅化

光栅化就是把几何数据和像素数据转换为片断（fragment）的过程。每个片断方块对应于帧缓冲区中的一个像素。把顶点连接起来形成直线或者计算填充多边形的内部像素时，需要考虑直线和多边形的点画模式、直线的宽度、点的大小、着色模型以及用于支持抗锯齿处理的覆盖计算。每个片断方块都将具有各自的颜色和深度值。

1.5.8 片断操作

在数据实际存储到帧缓冲区之前，要执行一系列的操作。这些操作可能会修改甚至丢弃这些片断。所有这些操作都可以启用或禁用。

第一个可能执行的操作是纹理处理。在纹理内存中为每个片断生成一个纹理单元（texel，也就是纹理元素），并应用到这个片断上。接下来，组合主颜色和辅助颜色，可能还会应用一次雾计算。如果应用程序使用了片段着色器，前面这三个操作可能都在着色器中完成。

前面的操作生成了最终的颜色和深度之后，如果有效，执行可用的剪裁测试、alpha 测试、模板测试和深度缓冲区测试（深度缓冲区实际是隐藏面消除）。某种可用的测试的失败将会终止片段方块的继续处理。随后，将要执行的可能是混合、抖动、逻辑操作以及根据一个位掩码的屏蔽操作（参阅第 6 章和第 10 章）。最后，经过完整处理的片断就被绘制到适当的缓冲区，最终成为一个像素并到达它的最终栖息地。

1.6 与 OpenGL 相关的函数库

OpenGL 提供了一组功能强大但又非常基本的渲染函数，所有的高级绘图操作都是在这些函数的基础上完成的。另外，OpenGL 程序还必须使用窗口系统的底层机制。有一些函数库可以帮助程序员简化编程任务，它们是：

OpenGL 工具函数库（GLU）包含了一些函数，它们利用底层的 OpenGL 函数来执行一些特定的任务，例如设置特定的矩阵（例如用于视图方向和投影的矩阵）、多边形分格化以及表面渲染等。所有的 OpenGL 实现都把 GLU 作为它们的一部分。《OpenGL Reference Manual》描述了 GLU 的部分函数。本书介绍一些非常实用的 GLU 函数，分布于相关的章节中，例如第 11 章和 12.3 节。GLU 函数都使用前缀 glu。

所有的窗口系统都提供了一个函数库，对该窗口系统的功能进行扩展，以支持 OpenGL 渲染。对于使用 X 窗口系统的计算机而言，它所使用的 OpenGL 扩展（GLX）是作为 OpenGL 的一个附件提供的。所有的 GLX 函数都使用前缀 glX。对于 Microsoft Windows 而言，WGL 函数库提供了 Windows 和 OpenGL 之间的接口。所有的 WGL 函数都使用前缀 wgl。对于 Mac OS 而言，这 3 种接口都可以使用：AGL（前缀为 agl）、CGL（cgl)）和 Cocoa（NSOpenGL 类）。所有这些窗口系统的扩展库都在附录 D 中有进一步的介绍。

OpenGL 实用工具库（OpenGL Utility Toolkit，GLUT）是 Mark Kilgard 编写的一个独立于窗口系统的工具包，它的目的是隐藏不同窗口系统 API 所带来的复杂性。在本版中，我们使用名为 Freeglut 的 GLUT 开源实现，它扩展了 GLUT 最初的功能。下面的小节介绍了编写使用 GLUT 的程序所必需的基本过程，而所有这些程序都使用 glut 为前缀。本书的大部分内容继续使用术语 GLUT，读者只要知道我们在使用 Freeglut 实现就可以了。

1.6.1 包含文件

对于所有的 OpenGL 应用程序，都需要在每个文件中包含 OpenGL 头文件。几乎所有的 OpenGL 应用程序都使用 GLU（前面提到的 OpenGL 工具函数库）。要使用这个函数库，必须包含 glu.h 头文件。因此，几乎所有的 OpenGL 源代码文件都是以下面这两行开始的：

```
1. #include <GL/gl.h>
2. #include <GL/glu.h>
```

注意：Microsoft Windows 要求在 gl.h 或 glu.h 之前包含 windows.h 头文件，因为 MicrosoftWindows 版本的 gl.h 和 glu.h 文件内部使用的一些宏是在 windows.h 中定义的。

OpenGL 库总是不断地发生变化。制造图形硬件的各个厂商都可能会增加一些新特性。由于这些新特性太新，可能还没有添加到 gl.h 中。为了使程序员能够使用这些新的 OpenGL 扩展，OpenGL 提供了另一个头文件，叫做 glext.h。这个头文件包含了所有最新版本和扩展函数以及标记，可以在 OpenGL 网站的 OpenGL Registry 页面（http://www.opengl.org/registry）

上找到它。这个页面还有每个 OpenGL 扩展发布的规范。和所有其他头文件一样，可以通过下面这行代码包含这个头文件：

```
1. #include "glext.h"
```

读者可能注意到这个文件名两边使用的是双引号，而不是普通的尖括号。由于 glext.h 的目的是允许程序访问图形卡生产厂商提供的新扩展，因此可能需要经常从 Internet 下载各种版本。因此，在编译程序时，应尽可能在应用程序的本地目录上保留这个文件的一份拷贝。另外，程序员可能没有足够的权限把 glext.h 文件放在系统头文件包含目录中（例如 UNIX 类型系统的/usr/include）。

如果想直接访问一个支持 OpenGL 的窗口接口库（例如 GLX、AGL、PGL 或 WGL），必须包含额外的头文件。例如，如果想调用 GLX 所提供的函数，可能需要增加如下代码：

```
1. #include <X11/Xlib.h>
2. #include <GL/glx.h>
```

在 Microsoft Windows 中，可以通过如下代码获得对 WGL 函数的访问：

```
1. #include <windows.h>
```

如果想使用 GLUT 来实现窗口管理任务，应该包含如下代码：

```
1. #include <freeglut.h>
```

注意：GLUT 头文件最初名为 glut.h。glut.h 和 freeglut.h 都保证已经正确地包含了 gl.h 和 glu.h，因此包含所有这 3 个文件是没有必要的。此外，这些头文件还保证在包含 gl.h 和 glu.h 之前，已经正确地定义了所有依赖操作系统的内部的宏。为了使 GLUT 程序具有可移植性，在包含了 glut.h 或 freeglut.h 之后，就不要再包含 gl.h 或 glu.h 了。

绝大多数 OpenGL 应用程序还使用了标准 C 函数库的系统调用，因此包含与图形无关的头文件也是很常见的，例如：

```
1. #include <stdlib.h>
2. #include <stdio.h>
```

在本书中，我们并没有在示例程序中包含头文件，这样代码看上去显得比较简洁。

针对 OpenGL 3.1 的头文件

OpenGL 3.0 只是向 OpenGL 的功能集中添加了新的函数和特性，相比较而言，OpenGL 3.1 删除了标记为废弃的函数。为了让软件开发者更容易适应这种变化，OpenGL 3.1 提供了一个全新的头文件集合，并且为厂商推荐了一个位置以便将它们整合到各自的操作系统中。

仍然可以使用 gl.h 和 glext.h 文件，它们将继续记录所有的 OpenGL 入口点，而不管 OpenGL 是什么版本。

然而，如果要把代码移植到仅使用 OpenGL 3.1 的系统上，可能要考虑使用新的 OpenGL 3.1 头文件。

```
1. #include <GL3/gl3.h>
2. #include <GL3/gl3ext.h>
```

它们包含了针对 OpenGL 3.1 的函数和标记（考虑到未来的版本，这个特性集将会限制于该特定版本）。读者应该会发现，这些头文件简化了把已有的 OpenGL 代码移植到新版本上的过程。和任何 OpenGL 头文件一样，这些文件也可以从 OpenGL Registry 页面 (http://www.opengl.org/registry)下载。

1.6.2 OpenGL 实用工具库（GLUT）

如前所述，OpenGL 包含了许多渲染函数，但是它们的设计目的是独立于任何窗口系统或操作系统。因此，它并没有包含打开窗口或者从键盘或鼠标读取事件的函数。遗憾的是，如果连最基本的打开窗口的功能都没有，编写一个完整的图形程序几乎是不可能的。并且，绝大多数有趣的程序都需要一些用户输入，或者需要操作系统和窗口系统的其他服务。大多数情况下，只有完整的程序才能形成有趣的示例程序。因此，本书使用 GLUT 来简化打开窗口、检测输入等任务。如果读者所使用的系统中提供了 penGL 和 GLUT 实现，本书的示例程序就可以不经修改地链接到读者使用的 OpenGL 和
GLUT 函数库。

另外，由于 OpenGL 绘图函数仅限于生成简单的几何图元（点、直线和多边形），GLUT 还包含了一些函数，用于创建一些更为复杂的三维物体，例如球体、圆环面和茶壶。这样，程序输出的快照看上去就会比较有趣。（注意，OpenGL 工具函数库，即 GLU，也包含了一些二次方程函数，其功能与 GLUT 相似，可以创建一些三维物体，例如球体、圆柱体、圆锥体等。）

如果想编写功能完整的 OpenGL 应用程序，GLUT 可能无法满足要求。但是，GLUT 可以作为学习 OpenGL 的一个非常好的起点。本节的剩余部分将简单地描述 GLUT 的一个较小的子集，以便读者可以理解本书其他部分的示例程序（参阅附录 A 了解 GLUT 的更多信息）。

窗口管理

GLUT 通过几个函数执行初始化窗口所需要的任务：

glutInit (int *argc, char **argv) 对 GLUT 进行初始化，并处理所有的命令行参数（对于 X 系统，这将是类似-display 和-geometry 的选项）。glutInit()应该在调用其他任何 GLUT 函数之前调用。

glutInitDisplayMode (unsigned int mode) 指定了是使用 RGBA 模式还是颜色索引模式。另外还可以指定是使用单缓冲还是使用双缓冲窗口。如果想使用颜色索引模式，就需要把一些颜色加载到颜色映射表中，这个任务可以用 glutSetColor()完成。最后，还可以使用这个函数表示希望窗口拥有相关联的深度、模板、多重采样和/或累积缓冲区。例如，如果需要一个双缓冲、RGBA 颜色模式以及带有一个深度缓冲区的窗口， 可以调用 glutInitDisplay Mode(GLUT_ DOUBLE |GLUT_RGBA | GLUT_DEPTH)。

glutInitWindowPosition (int x, int y) 指定了窗口左上角的屏幕位置。

glutInitWindowSize (int width, int size) 指定了窗口的大小（以像素为单位）。

glutInitContextVersion(int majorVersion, int minorVersion)声明了要使用 OpenGL 的哪个版本。（这是新增加的函数，只有在使用 Freeglut 的时候才能使用，并且引入到了 OpenGL 3.0 中。参见 1.8.1 节，了解关于 OpenGL 渲染环境和版本的更多细节。）

glutInitContextFlags(int flags)声明了想要使用的 OpenGL 渲染环境的类型。对于常规的 OpenGL 操作，可以在自己的程序中省略这一调用，然而，如果想要使用向前兼容的 OpenGL 渲染环境，需要调用这一函数。（这也是新增加的函数，只有在使用 Freeglut 的时候才能使用，并且引入到了 OpenGL 3.0 中。参见 1.8.1 节，了解关于 OpenGL 渲染环境和版本的更多细节。）

int glutCreateWindow (char *string)创建了一个支持 OpenGL 渲染环境的窗口。这个函数返回一个唯一的标识符，标识了这个窗口。注意：在调用 glutMainLoop()函数之前，这个窗口并没有显示。

显示回调函数

glutDisplayFunc (void (*func)(void)) 是读者所看到的第一个也是最为重要的事件回调函数。每当 GLUT 确定一个窗口的内容需要重新显示时，通过 glutDisplayFunc()注册的那个回调函数就会被执行。因此，应该把重绘场景所需要的所有代码都放在这个显示回调函数里。

如果程序修改了窗口的内容，有时候可能需要调用 glutPostRedisplay()，这个函数将会指示 glutMainLoop()调用已注册的显示回调函数。

运行程序

最后，必须调用 glutMainLoop()来启动程序。所有已经创建的窗口将会在这时显示，对这些窗口的渲染也开始生效。事件处理循环开始启动，已注册的显示回调函数被触发。一旦进入循环，它就永远不会退出。

示例程序 1-2 展示了如何使用 GLUT 创建示例程序 1-1。注意代码结构的变化。为了最大限度地提高效率，只需要调用一次的操作（设置背景颜色和坐标系统）都位于一个叫做 init()的函数中。用于渲染场景（并可能需要重新渲染）的操作放在 display()函数中，后者就是被注册的 GLUT 显示回调函数。

示例程序 1-2 简单的 OpenGL 示例程序，使用 GLUT：hello.c

```
1. void display(void)
2. {
3. /* clear all pixels */
4. glClear(GL_COLOR_BUFFER_BIT);
5. /*draw white polygon (rectangle)with corners at
6. *(0.25,0.25,0.0)and (0.75,0.75,0.0)
7. */
8. glColor3f(1.0,1.0,1.0);
9. glBegin(GL_POLYGON);
10. glVertex3f(0.25,0.25,0.0);
11. glVertex3f(0.75,0.25,0.0);
12. glVertex3f(0.75,0.75,0.0);
13. glVertex3f(0.25,0.75,0.0);
14. glEnd();
15. /* don ’t wait!
16. * start processing buffered OpenGL routines
17. */
18. glFlush();
19. }
20. void init(void)
21. {
22. /* select clearing (background)color */
23. glClearColor(0.0,0.0,0.0,0.0);
24. /* initialize viewing values */
25. glMatrixMode(GL_PROJECTION);
26. glLoadIdentity();
27. glOrtho(0.0,1.0,0.0,1.0,-1.0,1.0);
28. }
```

```
/*
* Declare initial window size,position,and display mode
* (single buffer and RGBA).Open window with "hello "
* in its title bar.Call initialization routines.
* Register callback function to display graphics.
* Enter main loop and process events.
*/
int main(int argc,char**argv)
{
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB);
glutInitWindowSize(250,250);
glutInitWindowPosition(100,100);
glutCreateWindow("hello ");
init();
glutDisplayFunc(display);
glutMainLoop();
return 0;/*ISO C requires main to return int.*/
}
```

处理输入事件

可以使用下面这些函数注册一些回调函数，当指定的事件发生时，这些函数便会被调用：

glutReshapeFunc(void(*func)int w, int h))表示当窗口的大小发生改变时应该采取什么行动。

glutKeyboardFunc (void(*func)(unsigned char key, int x, int y))和glutMouseFunc(void(*func)(intbutton, int state, int x, int y))允许把键盘上的一个键或鼠标上的一个按钮与一个函数相关联，当这个键或按钮被按下或释放时，这个函数就会调用。

glutMotionFunc(void(*func)(int x, int y))注册了一个函数，当按下一个鼠标按钮移动鼠标时，这个函数就会调用。

空闲处理

可以在 glutIdleFunc(void(*func)(void))回调函数中指定一个函数，如果不存在其他尚未完成的事件（例如，当事件循环处于空闲的时候），就执行这个函数。这个回调函数接受一个函数指针作为它的唯一参数。如果向它传递 NULL（0），就相当于禁用这个函数。

绘制三维物体

GLUT 包含了几个函数，用于绘制下面这些三维物体：

圆锥体	二十面体	茶壶
立方体	八面体	四面体
十二面体	球体	圆环面

可以根据已定义的法线把这些物体画成线框模型或实心模型。例如，用于绘制立方体和球体的函数如下所示：

```
1. void glutWireCube(GLdouble size);
2. void glutSolidCube(GLdouble size);
3. void glutWireSphere(GLdouble radius, GLint slices, GLint stacks);
4. void glutSolidSphere(GLdouble radius, GLint slices, GLint stacks);
```

在绘制这些模型时，它们的中心都位于全局坐标系统的原点。关于这些绘图函数的原型，请参阅附录 A。

1.7 动画

在图形计算机上可以实现的最激动人心的事情之一就是绘制能够运动的图片。无论是试图看到自己设计的机械零件所有侧面的工程师，还是使用模拟飞行器学习飞机驾驶的飞行员，或者仅仅是计算机游戏的狂热爱好者，都会觉得动画是计算机图形的一个重要组成部分。

在电影院里，屏幕上的运动画面是通过拍摄大量的图片，然后以每秒 24 帧的频率把它们投影到屏幕上来实现的。每一帧移动到镜头后的一个位置，接着快门打开，然后这一帧便显示。在影片切换到下一帧的一瞬间，快门关闭，然后又打开以显示下一帧，然后以此类推。尽管观众所看到的是每秒 24 帧切换的不同的画面，但大脑会把它们混合成一段平滑的动画。（老式的卓别林电影每秒播放 16 帧，因此有比较明显的抖动现象。）在一般情况下，计算机屏幕每秒大约刷新（重绘画面）60～76 次，有些甚至每秒刷新 120 次。显然，每秒 60 帧比每秒 30 帧的效果更为平滑，而每秒 120 帧的效果又强于每秒 60 帧。但是，如果刷新率超过每秒 120 帧，就有可能到达衰减点，具体取决于人眼感知的极限。

运动图片投影方法之所以可行的关键原因是每个帧在显示的时候已经完成绘制。如果试图用如下程序来实现计算机动画，播放数以百万帧计的影片：

```
1. open_window();
2. for (i = 0; i < 1000000; i++) {
3. clear_the_window();
4. draw_frame(i);
5. wait_until_a_24th_of_a_second_is_over();
6. }
```

如果加上系统清除屏幕以及绘制一个典型的帧所需要的时间，这个程序所产生的效果将会越来越差，具体取决于清除屏幕和绘图所占用的时间与 1/24 秒的接近程度。假如绘图时间差不多需要 1/24 秒。那么第一个绘制的物体在这 1/24 秒的时间内是可见的，并在屏幕上显示一幅实体图像。但是，越到后面，正在绘制的物体将会以越来越快的速度清除，因为程序紧接着要显示下一帧。这就导致了一个非常可怕的场景：在大多数的 1/24 秒中，观众所看到的并不是最终绘制完成的物体，而是清除的背景。

绝大多数 OpenGL 实现提供了双缓冲（硬件或软件），即提供了两个完整的颜色缓冲区。当一个缓冲区显示时，另一个缓冲区正在进行绘图。当一个帧绘制完成之后，两个缓冲区就进行交换。这样，刚才用来显示的那个缓冲区现在就用于绘图，刚才用于绘图的那个缓冲区现在就用于显示。这有点类似于只循环播放两个帧的电影放映机。当其中一帧投影到屏幕上时，画家迅速擦掉并重绘当前未显示的那个帧。只要画家的动作足够快，观众并不会注意到这种方式和事先画好所有的帧然后再投影的区别。电影放映机只是简单地一幅又一幅地显示这些帧而已。使用双缓冲，每一帧只有在绘制完成后才会显示，观众永远不会看到不完整的帧。

下面的代码对前面的程序进行了修改，用双缓冲平滑地显示动画：

```
1. open_window_in_double_buffer_mode();
2. for (i = 0; i < 1000000; i++) {
3. clear_the_window();
4. draw_frame(i);
5. swap_the_buffers();
6. }
```

1.7.1 暂停刷新

在有些 OpenGL 实现中，除了简单地交换显示和绘图缓冲区之外，swap_the_buffers() 函数将会等待，直到当前的屏幕刷新周期结束，这样前一个缓冲区的内容就能够完整地显示。这个函数还允许从头开始完整地显示新缓冲区。假定系统每秒刷新显示画面 60 次，意味着可以实现的最快帧率是每秒 60 帧（fps）。如果所有的帧都可以在不到 1/60 秒的时间内完成清除和绘制，那么在这个帧率下，动画的显示将会非常平滑。

如果帧的内容过于复杂，无法在 1/60 秒的时间内完成绘制，那会发生什么情况呢？此时每帧显示的次数将不止 1 次。例如，如果每帧需要 1/45 秒的时间才能完成绘制，而帧率是 30 fps，这样每帧就有 1/30 秒－1/45 秒=1/90 秒（或者说三分之一）的空闲时间。

另外，视频刷新频率是固定的，这就有可能导致一些意想不到的性能问题。例如，在一台刷新速度最快为 1/60 秒并采用固定帧率的显示器上，可以在 60 fps、30 fps、20 fps、15 fps、12 fps（60/1、60/2、60/3、60/4、60/5、..）等帧率下运行。这意味着如果程序员正在编写一

个应用程序，并且逐渐增加功能（假如这个应用程序是一个飞行模拟器，并且正在添加地面场景），最初添加的每个特性对总体性能不会有影响，因此仍然可以获得 60 fps 的帧率。突然在程序中又增加了一个新特性之后，系统无法在 1/60 秒的时间画完一帧中的所有物体，于是动画的帧率就从 60 fps 下降到 30 fps，因为系统错过了第一次缓冲区交换的时间。当每帧的绘图时间超过 1/30 秒时，也会发生类似的事情，动画的帧率将从 30 fps 下降到 20 fps。

当场景的复杂度接近于任一魔幻时间（指本例中的 1/60 秒、2/60 秒、3/60 秒等临界时间），由于随机偏差，有些帧会稍微多于这个时间，有些帧则稍微小于这个时间。这样，帧率便会变得没有规律，可能会导致视觉上的混乱。在这种情况下，如果无法以场景进行简化，使所有的帧都足够快，最好有意增加一小段延迟，使它们都错过这个魔幻时间，统一到下一个更慢的固定帧率。如果各个帧的复杂性具有极大的差异，就可能需要采取一种更为复杂的方法。

1.7.2 动画=重绘+交换

真实的动画程序的结构与上面描述的相差并不大。通常，对于每个帧而言，与判断缓冲区的哪些部分需要重绘相比，重新绘制整个缓冲区要更容易一些。对于诸如三维飞行模拟器这样的应用程序，情况更是如此。在这种应用程序中，飞机方向的略微改变就可能导致窗外所有物体的位置都发生变化。在绝大多数动画中，场景中的物体简单地根据不同的变换进行重绘，例如根据移动的观察者、路上一辆前行的汽车或一个略微旋转的物体为视点（viewpoint）。如果非绘图操作所需要的重新计算量非常大，动画的帧率常常会降低。但是，记住 swap_the_buffers()函数之后的空闲时间总是可以用来进行这类计算。

OpenGL 并没有提供 swap_the_buffers()函数，因为并不是所有的硬件都支持这个特性。并且，在任何情况下，这个特征总是高度依赖于窗口系统。例如，如果使用的是 X 窗口系统，并且想直接使用这个特性，可以使用下面这个 GLX 函数：

```
void glXSwapBuffers(Display *dpy, Window window);
```

（关于其他窗口系统的相应函数，可以参阅附录 D）。

如果读者所使用的是 GLUT 函数库，只要调用下面这个函数就可以了：

```
void glutSwapBuffers(void);
```

示例程序 1-3 绘制了一个旋转的方块，说明了 glutSwapBuffers()函数的用法。其结果如图 1-3 所示。

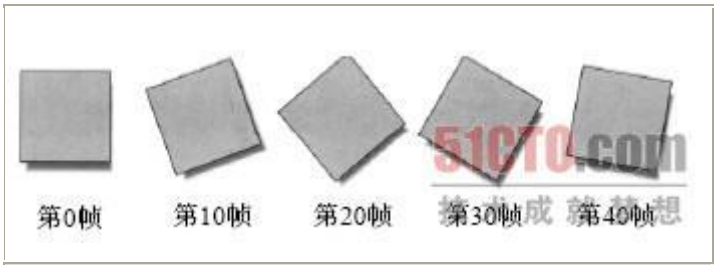


图 1-3 双缓冲旋转方块

这个例子还显示了如何使用 GLUT 控制输入设备，并打开或关闭空闲处理函数。在这个示例程序中，鼠标按钮用于切换方块是否进行旋转。

示例程序 1-3 双缓冲程序：double.c

```
1.  static GLfloat spin =0.0;
2.  void init(void)
3.  {
4.  glClearColor(0.0,0.0,0.0,0.0);
5.  glShadeModel(GL_FLAT);
6.  }
7.  void display(void)
8.  {
9.  glClear(GL_COLOR_BUFFER_BIT);
10. glPushMatrix();
11. glRotatef(spin,0.0,0.0,1.0);
12. glColor3f(1.0,1.0,1.0);
13. glRectf(-25.0,-25.0,25.0,25.0);
14. glPopMatrix();
15. glutSwapBuffers();
16. }
17. void spinDisplay(void)
18. {
19. spinspin =spin +2.0;
20. if (spin >360.0)
21. spinspin =spin -360.0;
22. glutPostRedisplay();
23. }
24. void reshape(int w,int h)
25. {
26. glViewport(0,0,(GLsizei)w,(GLsizei)h);
27. glMatrixMode(GL_PROJECTION);
```

```
glLoadIdentity();
glOrtho(-50.0,50.0,-50.0,50.0,-1.0,1.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}
void mouse(int button,int state,int x,int y)
{
switch (button){
case GLUT_LEFT_BUTTON:
if (state ==GLUT_DOWN)
glutIdleFunc(spinDisplay);
break;
case GLUT_MIDDLE_BUTTON:
if (state ==GLUT_DOWN)
glutIdleFunc(NULL);
break;
default:
break;
}
}
/*
*Request double buffer display mode.
*Register mouse input callback functions
*/
int main(int argc,char**argv)
{
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_DOUBLE |GLUT_RGB);
glutInitWindowSize(250,250);
glutInitWindowPosition(100,100);
glutCreateWindow(argv [0]);
init();
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutMouseFunc(mouse);
glutMainLoop();
return 0;
}
```

1.8 OpenGL 及其废弃机制

高级话题

正如前面提到的，OpenGL 不断地进行着改进和优化。执行图形操作的新方法开发出来，并且像通用图形处理器（general-purpose computing on graphics processing units，GPGPU）这样的全新领域诞生，这些都引领了图形硬件能力的提升。厂商建议对 OpenGL 进行新的扩展，并且最终某些扩展作为新的 OpenGL 核心修订融入了进去。多年以来，这一发展过程使得 API 中出现了大量的、完成同一操作的冗余方法。在很多时候，尽管功能是相似的，方法的应用程序性能却通常不同，这给人们这样一种印象：OpenGL API 很慢，并且不能在现代硬件上很好地工作。在 OpenGL 3.0 中，KhronosOpenGL ARB 工作组制定了一种废弃模式，该模式说明了如何从 API 中删除功能。然而，这一修改不只是需要修改核心 OpenGL API，它也影响到如何创建 OpenGL 渲染环境以及可用的 OpenGL 渲染环境的类型。

1.8.1 OpenGL 渲染环境

OpenGL 渲染环境是 OpenGL 在其中存储状态信息的数据结构，渲染图像的时候要用到这些信息。它们包括纹理、服务器端的缓存对象、函数入口点、混合状态以及编译过的渲染器对象，简而言之，包括此后各章讨论的所有内容。在 OpenGL 3.0 之前，只有一种 OpenGL 渲染环境，即完整 OpenGL 渲染环境，它包含了 OpenGL 实现中所有可用的内容，并且只有一种创建 OpenGL 渲染环境的方法（这和窗口系统有关）。在 OpenGL 3.0 中，创建了一种新的渲染环境，即向前兼容的渲染环境，它隐藏了那些标记为将来要从 OpenGL API 中删除的特性，以帮助应用程序开发者修改自己的应用，从而适应 OpenGL 未来的版本。

模式

OpenGL 3.0 除了增加了不同的渲染环境类型，还引入了模式（profile）的概念。模式是特定于应用程序领域的 OpenGL 功能集的子集，例如，游戏、计算机辅助设计（CAD）或针对嵌入式平台编写的程序。

当前，只定义了一种模式，它包含了创建的 OpenGL 渲染环境中支持的整个的功能集。OpenGL 未来版本中可能会引入新的模式类型。

每个窗口系统都有自己的函数集合，用来把 OpenGL 整合到自己的操作中（例如，针对 MicrosoftWindows 的 WGL），这些操作才真正地创建了我们所需的 OpenGL 渲染环境（这也是根据模式完成的）。同样，尽管过程基本相同，但是使用的函数调用是特定于窗口系统的。好在 GLUT 隐藏了这些操作的细节。有时候，我们可能需要知道细节。这些在附录 D 中介绍，在那里，可以找到特定于自己的窗口系统函数的相关信息。

使用 GLUT 指定 OpenGL 渲染环境

当调用 glutCreateWindow()的时候，GLUT 库自动负责创建 OpenGL 渲染环境。默认情况下，所需的 OpenGL 渲染环境将会与 OpenGL 2.1 兼容。要针对 OpenGL 3.0 及其以后的版本来分配渲染环境，需要调用 glutInitContextVersion()。同样，如果想要使用向前兼容渲染环境以便于移植，还需要通过调用 glutInitContextFlags()指定环境属性。示例程序 1-4 展示了这些概念。附录 A 更为详细地介绍了这些函数。

示例程序 1-4 使用 GLUT 创建一个 OpenGL 3.0 渲染环境

```
1. glutInit(&argc,argv);
2. glutInitDisplayMode(GLUT_RGBA |GLUT_DEPTH |GLUT_DOUBLE);
3. glutInitWindowSize(width,height);
4. glutInitWindowPosition(xPos,yPos);
5. glutInitContextVersion(3,0);
6. glutInitContextFlags(GLUT_FORWARD_COMPATIBLE);
7. glutCreateWindow(argv [0]);
```

1.8.2 访问 OpenGL 函数

根据用来开发应用程序的操作系统的不同，可能需要做一些额外的工作来访问某些 OpenGL 函数。我们会知道何时有这样的必要，因为编译器会报告各种各样的函数没有定义（当然，每种编译器报告的这些错误是不同的，但是问题的关键之处是一致的）。在这种情况下，需要找到该函数的地址（位于一个函数指针中）。有几种不同的方法可以做到这一点：

如果应用程序使用本地窗口系统来打开窗口并进行事件处理，那么，针对应用程序将要使用的操作系统，使用相应的*GetProcAddress()函数。这些函数的例子包括 wglGetProcAddress()和 glXGetProcAddress()。

如果使用 GLUT，那么使用 GLUT 的函数指针获取函数 glutGetProcAddress()。

使用开源的项目 GLEW（OpenGL Extension Wrangler）。GLEW 定义了每个 OpenGL 函数，自动获取函数指针并验证扩展。访问 http://glew.sourceforge.net/获取详细信息，并获取代码或二进制文件。

尽管我们不会在本书正文所包含的程序中明确地展示这些选项，我们还是使用 GLEW 来简化过程。

第 2 章 状态管理和绘制几何物体

本章目标

用任意一种颜色清除窗口。

强制完成所有尚未执行的绘图操作。

在二维或三维空间绘制几何图元，如点、直线和多边形。

打开或关闭状态，以及查询状态变量的值。

控制几何图元的显示。例如，绘制虚线或轮廓多边形。

在实心物体表面的适当位置指定法线向量。

用顶点数组和缓冲区对象存储和访问几何数据，可以减少函数调用的数量。

同时保存和恢复几个状态变量。

尽管可以使用 OpenGL 绘制复杂和有趣的图形，但这些图形都是由几个为数不多的基本图形元素构建而成的。这应该不会令人吃惊，因为达 芬奇的那些伟大艺术品也不过是由铅笔和画刷完成的。

在最高抽象层次上，有 3 种绘图操作是最基本的：清除窗口、绘制几何图形，以及绘制光栅对象。光栅对象包括了像二维图像、位图和字体之类的东西。第 8 章将详细介绍光栅对象。本章将介绍如何清除屏幕以及如何绘制几何物体，包括点、直线和平面多边形。

此时，读者可能会产生疑问：我在电影和电视上看到过许多计算机图形，它们有大量优美着色的曲线和表面，如果 OpenGL 只能画直线和平面多边形，那么这些图形又是怎么产生的呢？本书封面上的图像包括了一张圆桌，桌上的物体都有弯曲的表面。事实上，读者看到的所有曲线和表面都是由大量的小型多边形或直线近似模拟出来的，就像本书封面的那个地球仪是由大量的小矩形块模拟出来的一样。这个地球仪的表面看上去并不是很平滑，这是因为这些矩形块相对于地球仪而言还不够小。在本章后面，读者将会看到如何用大量的微小几何图形构建曲线和表面。

本章主要由下面各节组成：

绘图工具箱：说明如何清除窗口，以及如何强制完成所有未执行的绘图操作。本节还介绍了如何控制几何物体的颜色，以及如何描述坐标系统。

描述点、直线和多边形：介绍这些基本的几何图形，并说明如何绘制它们。

基本状态管理：描述如何打开或关闭一些状态（模式），以及如何查询这些状态。

显示点、直线和多边形：说明可以对基本图元的显示施加什么样的控制，例如点的直径为多大、直线为虚线还是实线，以及多边形为轮廓多边形还是填充多边形。

法线向量：讨论如何指定几何物体的法线向量，并简单介绍它们的用途。

顶点数组：解释如何把大量的几何数据放在几个数组中，并且只需要少量几个函数调用，就可以渲染它们所描述的几何物体。减少函数调用的数量不仅可以改善编程效率，而且可以提高渲染性能。

缓冲区对象：详细解释如何使用服务器端的内存缓冲区存储顶点数组的数据，以实现更高效的几何渲染。

顶点数组对象：通过说明如何高效地在顶点数组集合中修改，展开对顶点数组和缓冲区对象的讨论。

属性组：揭示如何查询状态变量的当前值，并介绍如何同时保存和恢复几个相关的状态值。

创建多边形表面模型的一些提示：提供一些建议和技巧，帮助读者更好地理解如何用多边形近似地模拟物体的表面。

在阅读本章内容时，需要记住一件事：在 OpenGL 中，除非另有指定，否则每次调用一个绘图函数时，指定的物体就会被绘制。这似乎是显而易见的，但是在有些系统中，首先要列出需要绘制的物体清单。在完成这个清单后，再告诉图形硬件绘制这个清单中的所有物体。第一种风格称为立即模式的图形编程，它也是默认的 OpenGL 风格。除了立即模式之外，还可以选择把一些绘图命令保存在一个列表（称为显示列表）中，以后再一起执行。一般而言，立即模式更易于编程，但显示列表常常具有更高的效率。第 7 章将介绍如何使用显示列表以及在什么情况下可能需要使用它们。

OpenGL 1.1 版还引入了顶点数组的概念。

在 1.2 版本中，OpenGL 还增加了表面法线的缩放（GL_RESCALE_NORMAL）。另外，它还增加了 glDrawRangeElements() 函数，提供了对顶点数组的进一步支持。

在 OpenGL 1.3 版本中，对多重纹理单元的纹理坐标的支持已经成为 OpenGL 的一个核心特性。在此之前，多重纹理只是一个可选的 OpenGL 扩展。

在 1.4 版本中， 雾坐标和辅助颜色也可以存储在顶点数组中， glMultiDrawArrays()和 glMultiDrawElements()可以根据顶点数组来渲染图元。

在 1.5 版本中，顶点数组可以存储在缓冲区对象中，后者可以使用服务器内存来存储数组，这可以提高它们的渲染速度。

OpenGL 3.0 添加了对顶点数组对象的支持，允许通过一个单独的调用来绑定和激活与顶点数组相关的所有状态。这反过来使顶点数组集合之间的切换更加简单和快速。

OpenGL 3.1 删除了大多数立即模式程序并且添加了图元重启索引（primitive restart index），这允许我们用一个单独的绘图调用来渲染（具有相同类型的）多个图元。

2.1 绘图工具箱

本节首先介绍如何清除窗口，为绘图做好准备。然后，介绍如何设置绘制的物体的颜色以及如何强制完成绘图操作。上面这些主题和几何物体并没有直接的关系，但任何绘制几何物体的程序都需要处理这些问题。

2.1.1 清除窗口

在计算机屏幕上绘图和在纸上绘图是不一样的，因为纸本来就是白色的，只要直接在上面画图就

可以了。在计算机中，保存图片的内存通常被计算机所绘制的前一幅图像所填充，因此在绘制新场景之前，一般需要把它清除为某种背景颜色。至于应该使用哪种背景颜色，取决于应用程序本身。如果是字处理程序，在绘制新文本之前，一般把背景清除为白色（就像纸的颜色一样）。如果应用程序绘制的是在航天飞机上看到的太空景象，那么在开始绘制恒星、行星以及遥远的飞船之前，需要把背景清除为黑色。有时候，可能并不需要清除背景。例如，如果图像的内容是一个房间的内部，在绘制房间的墙面时，整个图形窗口都会被覆盖，原先的背景颜色并不会对新场景产生影响。

此时，读者可能会疑惑为什么要在绘图之前清除窗口？如果画一个适当颜色的矩形，让它足够大，能够覆盖整个窗口不就行了吗？这种方法当然也不是不行，但是清除窗口这种方式具有几点优势。首先，特殊的清除窗口函数的效率可能远远高于普通的绘图函数。其次，就像读者将在第 3 章所看到的那样，OpenGL 允许程序员任意设置坐标系统、观察位置和观察方向。这样一来，判断窗口清除矩形的大小和位置可能非常困难。最后，在许多机器上，图形硬件除了包括屏幕上显示的像素颜色的缓冲区之外，还包括了许多别的缓冲区。这些缓冲区随时可能清除，如果有一条命令能够清除按照任意形式组合的缓冲区，无疑是非常方便的（关于这些缓冲区的内容，请参阅第 10 章）。

我们还必须知道像素颜色是如何存储在名为位平面的图形硬件中的。可以采用的存储方式有两种：可以把像素颜色的红、绿、蓝和 alpha 值（RGBA）直接存储在位平面中，也可以存储一个颜色索引值，用它来引用颜色查找表中的一个颜色项。RGBA 颜色显示模式更为常用，所以本书绝大多数示例程序都将使用这种模式。关于这两种颜色模式的详细信息，请参阅第 4 章。另外，在第 6 章之前，读者完全可以忽略 alpha 值。例如，下面这两行代码把一个 RGBA 模式的窗口清除为黑色：

```
1. glClearColor(0.0, 0.0, 0.0, 0.0);
2. glClear(GL_COLOR_BUFFER_BIT);
```

第一行代码把清除颜色设置为黑色，第二行代码把整个窗口清除为当前清除颜色。glClear()的唯一参数表示需要清除的缓冲区。在这个例子中，程序只清除颜色缓冲区，在屏幕上显示的图像仍然保持原样。一般情况下，只要在程序的早期设置 1 次清除颜色就可以了，以后可以根据需要随时清除缓冲区。OpenGL 把当前的清除颜色作为一个状态变量，这样就

不必在每次清除缓冲区时重新指定清除颜色。第4章和第10章讨论了如何使用其他缓冲区。现在，读者只需要知道清除这些缓冲区是一件非常简单的事情。例如，为了同时清除颜色缓冲区和深度缓冲区，只需要使用下面这个函数序列就可以了：

```
1. glClearColor(0.0, 0.0, 0.0, 0.0);
2. glClearDepth(1.0);
3. glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

在上面的代码中，glClearColor()函数的作用和前一段代码相同。glClearDepth()函数指定了深度缓冲区中的每个像素需要设置的值。现在，glClear()函数的参数使用了位逻辑操作符OR，把所有需要清除的缓冲区组合起来。下面是 glClear()函数的总结，其中包括一张表，表中列出了可以清除的缓冲区、这些缓冲区的名称，以及将在哪些章节对它们进行详细讨论。

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

设置当前清除颜色，用于清除 RGBA 模式下的颜色缓冲区（关于 RGBA 颜色模式的详细信息，请参阅第 4 章）。red、green、blue 和 alpha 值会根据需要进行截取，其范围限定在[0, 1]之内。默认的清除颜色是（0, 0, 0, 0），也就是黑色。

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

用当前的缓冲区清除值清除指定的缓冲区。mask 参数的值是表 2-1 列出的值的位逻辑 OR 组合。

表 2-1 清除缓冲区

缓冲区	名称	参考章节
颜色缓冲区	GL_COLOR_BUFFER_BIT	第4章
深度缓冲区	GL_DEPTH_BUFFER_BIT	第10章
累积缓冲区	GL_ACCUM_BUFFER_BIT	第10章
模板缓冲区	GL_STENCIL_BUFFER_BIT	第10章

在发出命令清除多个缓冲区之前，如果想使用的并不是默认的 RGBA 值、深度值、累积值和模板索引值，就必须为每个缓冲区设置需要清除的值。除了用于设置颜色缓冲区和深度缓冲区当前清除值的 glClearColor()和 glClearDepth()函数之外，还可以使用 glClearIndex()、glClearAccum()和 glClearStencil()函数设置用于清除相应缓冲区的颜色索引、累积颜色和模板索引值（关于这些缓冲区以及它们的用途，请参阅第 4 章和第 10 章）。

OpenGL 允许同时清除多个缓冲区，这是因为清除通常是一种相对较慢的操作，涉及窗口中的每个像素（可能数以百万计）。有些图形硬件允许同时清除一组缓冲区。如果硬件不支持同时清除多个缓冲区，它就会线性地执行这些清除操作。下面这两段代码：

```
1. glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

和

```
1. glClear(GL_COLOR_BUFFER_BIT);
2. glClear(GL_DEPTH_BUFFER_BIT);
```

它们在功能上是等价的，但是在许多机器上，前者的执行速度要快得多。在任何机器上，它肯定不会比后者慢。

2.1.2 指定颜色

在 OpenGL 中，物体的形状和它的颜色无关。当一个特定的几何物体被绘制时，它是根据当前指定的颜色方案进行绘制的。颜色方案有可能非常简单，例如“用红色绘制所有的物体”。颜色方案也可能非常复杂，例如“物体由蓝色塑料制成，有一盏黄色的聚光灯从某个方向对准物体表面的某个点，物体的其他地方由较淡的红褐色普通光照射”。一般而言，OpenGL 程序员首先设置颜色或颜色方案，然后再绘制物体。在这种颜色或颜色方案被修改之前，所有的物体都用这种颜色或这种颜色方案进行绘制。这种追踪当前颜色的方法可以使 OpenGL 具有更高的绘图性能。

例如，下面的伪码：

```
1. set_current_color(red);
2. draw_object(A);
3. draw_object(B);
4. set_current_color(green);
5. set_current_color(blue);
6. draw_object(C);
```

用红色绘制物体 A 和 B，用蓝色绘制 C。把当前颜色设置为绿色的第 4 行代码完全是浪费。颜色、光照和着色都是非常庞大的主题，分别可以用一整章的篇幅来描述。但是，为了绘制可见的几何图形，必须了解这些方面的一些基本知识，必须知道如何设置当前颜色。接下来的几段文字提供了这方面的信息（关于这些主题的详细内容，请参阅第 4 章和第 5 章）。

为了设置颜色，可以使用 glColor3f()函数。这个函数接受 3 个参数，它们都是 0.0～1.0 之间的浮点数，分别表示颜色的红、绿和蓝色成分。读者可以认为这 3 个值指定了一种混合颜色：0.0 表示不使用这种成分，1.0 表示最大限度地使用这种成分。因此，下面这行代码：

```
1. glColor3f(1.0, 0.0, 0.0);
```

表示使用系统所具有的最亮红色，但不使用绿色和蓝色成分。如果这 3 个参数的值均为 0.0，最终的颜色就是黑色。如果它们均为 1.0，最终的颜色就是白色。如果这 3 种成分的值

都设置为 0.5，则最终颜色为灰色（黑色和白色的中间色）。下面是 8 条用于设置颜色的命令，并注明了它们所设置的具体颜色：

```
glColor3f(0.0, 0.0, 0.0); /* black */
glColor3f(1.0, 0.0, 0.0); /* red */
glColor3f(0.0, 1.0, 0.0); /* green */
glColor3f(1.0, 1.0, 0.0); /* yellow */
glColor3f(0.0, 0.0, 1.0); /* blue */
glColor3f(1.0, 0.0, 1.0); /* magenta */
glColor3f(0.0, 1.0, 1.0); /* cyan */
glColor3f(1.0, 1.0, 1.0); /* white */
```

读者可能已经注意到，前面用于设置清除颜色的函数 glClearColor()接受 4 个参数，前 3 个参数与 glColor3f()的参数相同，第 4 个参数表示 alpha 值，我们将在第 6.1 节对这个参数的含义进行详细的解释。现在，只须把这个参数值设置为 0.0，这也是它的默认值。

2.1.3 强制完成绘图操作

正如我们在 1.5 节中看到的那样，绝大多数的现代图形系统都可以看成是一条装配线。中央处理器（CPU）发出一条绘图命令，但执行几何变换、裁剪、着色、纹理操作的也许是其他硬件。最终，经过处理的数据写入用于显示的位平面中。在高端架构的计算机中，每一种操作都是由不同的硬件执行的，这些硬件的设计目标就是快速执行各自的特定任务。在这种架构的计算机中，CPU 在发出下一条绘图命令之前无需等待前一条绘图命令的完成。例如，当 CPU 沿着管线发送一个顶点时，用于实现几何变换的硬件可能正在对前一个发送的顶点执行变换，而再之前所发送的一个顶点可能正在进行裁剪处理。在这种系统中，如果 CPU 在发出下一条绘图命令之前还要等待前一条命令的完成，无疑会导致严重的性能问题。

另外，应用程序也可能在多台计算机上运行。例如，主程序可能在其他地方（在一台称为客户机的计算机上）运行，而用户在自己的工作站或终端（即服务器，它通过网络与客户机相连）上查看绘图结果。在这种情况下，如果每条绘图命令都单独通过网络发送，其效率将极为低下，因为网络传输所造成的开销相当巨大。通常，客户机把一组命令收集到一个网络包中，然后再将它们一起发送。遗憾的是，客户机上的网络代码一般无法知道图形程序是否完成了一个帧或一个场景的绘制。在最坏的情况下，它会一直等待下去，等待其他的绘图命令来填满一个包，其结果是用户永远无法看到完成之后的图形。

由于这个原因，OpenGL 提供了 glFlush()函数，它强制客户机发送网络数据包，即使这个包并没有填满。如果不存在网络，并且所有的命令都在服务器上立即执行，glFlush()可能并无作用。但是，如果程序员希望自己所编写的程序无论在有网络还是没有网络的情况下都能够正确地运行，就应该在每个帧或每个场景的最后添加一个 glFlush()调用。注意 glFlush()

并不等待绘图完成，它只是强制绘图命令开始执行，因此保证以前所有的命令都在有限的时间内执行，即使在此之后并没有任何渲染命令需要执行。

另外在如下场合，glFlush()也有其用武之地：

在系统内存中创建图像的软件渲染程序，并且并不想经常更新屏幕。

在那些收集成批的渲染命令以降低启动开销的 OpenGL 实现中。前面提到的网络传输例子就属于这种情况。

```
1. void glFlush(void);
```

强制以前发出的 OpenGL 命令开始执行，因此保证它们能够在有限的时间内完成。

有些命令（例如在双缓冲模式下交换缓冲区的命令）在执行之前会自动把尚未执行的命令发送到网络上。

如果觉得 glFlush()还不够用，可以试试 glFinish()。这个命令像 glFlush()一样对网络进行刷新，然后等待图形硬件或网络提示帧缓冲区的绘图已经完成。如果需要执行一些同步性的任务，就可能要用到 glFinish()。例如，如果使用 Display PostScript 在渲染结果上绘制标签，就希望在绘制标签之前已经确保完成了三维图像的渲染。另一种情况是希望确保绘图程序在接收用户输入之前已经完成了绘图。在发出 glFinish()命令之后，图形处理进程就会阻塞，直到图形硬件通知它绘图已经完成。记住，过多地使用 glFinish()命令会降低应用程序的性能，尤其是当图形程序是通过网络运行的情况下，因为它需要来回的通信。如果 glFlush()已经够用，就不要再使用 glFinish()。

```
1. void glFinish(void);
```

强制以前发出的 OpenGL 命令完成执行。在以前的命令完成执行之前，这个函数并不会返回。

2.1.4 坐标系统工具箱

无论是在刚打开窗口的时候，还是在以后移动窗口或改变窗口大小的时候，窗口系统都会发送一个事件作为通知。如果使用的是 GLUT，它会自动产生通知，并且在 glutReshapeFunc()中注册的那个函数会被调用。另外，必须注册一个回调函数，完成下列这些任务：

重新建立一个矩形区域，把它作为新的渲染画布。

定义一个用于绘制物体的坐标系统。

在第 3 章中，读者会看到如何定义三维坐标系统。但是，现在我们只要创建一个简单的、基本的二维坐标系统，并且在它上面绘制一些物体。然后，调用 glutReshapeFunc(reshape)函数，其中 reshape()是示例程序 2-1 所定义的函数。

示例程序 2-1 Reshape 回调函数

```
1. void reshape(int w, int h)
2. {
3. glViewport(0, 0, (GLsizei) w, (GLsizei) h);
4. glMatrixMode(GL_PROJECTION);
5. glLoadIdentity();
6. gluOrtho2D(0.0, (GLdouble) w, 0.0, (GLdouble) h);
7. }
```

GLUT 的内核将向这个函数传递 2 个参数：width 和 height，它们表示这个新的（或经过移动的、或改变了大小的） 窗口的宽度和高度（ 以像素为单位）。glViewport()函数调整用于绘图的像素矩形，使它占据整个新窗口。接下来的 3 行代码调整用于绘图的坐标系统，使左下角的坐标是（0，0），右上角的坐标是（w，h），如图 2-1 所示。

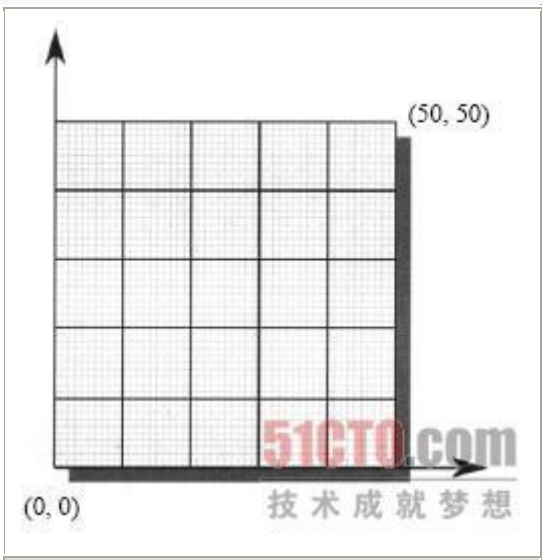


图 2-1 由 w = 50，h = 50 所定义的坐标系统

也可以采用另一种解释方式。读者可以想象自己的面前有一张图纸，reshape()函数中的 w 和 h 值表示图纸中方块的行数和列数。然后，必须在图纸上建立坐标轴。gluOrtho2D()函数把原点（0，0）放在最左下角的那个方块，然后让每个方块表示 1 个单位。现在，在接着渲染点、直线和多边形时，可以很方便地看清它们在图纸上的位置（读者可以把所有的物体都看成是二维的）。

2.2 描述点、直线和多边形

本节讨论如何描述 OpenGL 几何图元。所有的几何图元最终都是根据它们的顶点（vertex）来描述的。顶点就是坐标位置，用于定义点、线段的终点以及多边形的角。在下一节中，我们将讨论如何显示这些图元，以及如何控制它们的显示方式。

2.2.1 什么是点、直线和多边形

点、直线和多边形的数学概念比较简单。在 OpenGL 中，它们的概念与数学概念具有相似之处，但并不完全相同。

其中一个区别来自于计算机本身的限制。在任何 OpenGL 实现中，浮点计算的精度都是有限的，存在四舍五入的误差。因此，OpenGL 中点、直线和多边形的坐标也存在相同的问题。

另一个更为重要的差别来自于光栅图形显示的限制。在这种显示模式下，最小的可显示单位是像素。尽管像素的宽度可能小于百分之一英寸，但它仍然远远大于数学概念上的无穷小（针对点）和无穷细（针对直线）。当 OpenGL 执行计算时，它假定构成图元的点是用浮点数向量表示的。但是，在一般情况下，点被画作单个像素（但并非总是如此）。对于许多坐标略微不同的点，OpenGL 可能把它们画在同一个像素上。

点

点可以用一组称为顶点的浮点数表示。所有的内部计算都是建立在把顶点看成是三维数据的基础之上完成的。用户可以把顶点指定为二维形式（也就是说，只指定 x 坐标和 y 坐标），并由 OpenGL 把它的 z 坐标设置为零。

高级话题

OpenGL 是根据三维投影几何的齐次坐标进行操作的。因此，在内部的计算中，所有的顶点都是用 4 个浮点坐标值表示的（x, y, z 和 w）。如果 w 不等于 0，那么这些坐标值就对应于欧几里德三维点（x/w, y/w, z/w）。虽然可以在 OpenGL 函数中指定 w 坐标，但是这种做法极为罕见。如果未指定 w 坐标，它就默认为 1.0。关于齐次坐标系统的更多信息，请参阅附录 C。

直线

在 OpenGL 中，直线这个术语表示一段线段，而不是数学意义上在两端无限延伸的直线。在 OpenGL 中，指定一系列彼此相连接、甚至闭合的线段都是非常容易的（如图 2-2 所示）。但是，不管在哪种情况下，构成连线系列的直线都是根据它们的端点（顶点）指定的。

图 2-2 两条连接线段系列

多边形

多边形是由线段构成的单闭合环，其中线段是由它们的端点位置的顶点指定的。一般情况下，在绘制多边形时，它内部的像素将被填充。但是，也可以仅仅绘制多边形的外框，甚至把它画成一系列的点的形式（参见第 2.4.3 节）。

一般而言，多边形可能非常复杂。因此，OpenGL 对基本多边形的构成施加了很强的限制。首先，OpenGL 多边形的各条边不能相交（按照数学术语，满足这种条件的多边形称为简单多边形）。其次，OpenGL 多边形必须是凸多边形，也就是不存在内陷的部分。准确地说，在多边形的内部任意取两个点，如果连接这两个点的线段都在这个多边形的内部，那么这个多边形就是凸多边形。图 2-3 列出了一些合法的和非法的多边形例子。但是，OpenGL 并没有限制构成凸多边形边界线段的数量。注意，OpenGL 无法描述中间有洞的多边形，因为它们是非凸多边形，并且它的边界无法用一个单闭合线段环来表示。注意，如果在 OpenGL 中描述一个非凸的填充多边形，其结果可能会出乎预料。例如，在大多数系统中，实际填充的是不大于多边形凸包的部分。但在有些系统中，实际填充的是小于凸包的部分。

图 2-3 合法和非法多边形

OpenGL 在合法多边形的构成方面施加这些限制的原因是，这些限制有利于生产商提供快速的多边形渲染硬件来渲染符合条件的多边形。简单多边形的渲染速度非常快，而那些困难的情况就难以快速检测。因此，为了最大限度地提高性能，OpenGL 只能做出取舍，要求所有的多边形都是简单多边形。

现实世界的许多表面是由非简单多边形、非凸多边形或有洞的多边形组成。由于所有这些多边形都可以由简单多边形组合而成，因此 GLU 函数库提供了一些函数，可以创建这些更为复杂的形状。这些函数根据复杂的几何图形描述对多边形进行分格化，把它们分解为许多可以被渲染的简单多边形（关于分格化函数的更多信息，参见第 11.1 节）。

由于 OpenGL 的顶点总是三维的，因此构成一个特定多边形边界的点不必位于空间中的同一个平面上（当然，在许多情况下，它们确实位于同一个平面上。例如，当多边形的所有顶点的 z 坐标都是 0 的时候，或者当多边形是一个三角形的时候）。如果一个多边形的所有顶点并不位于同一个平面上，当它们在空间中经过各种不同的旋转，并改变观察点，然后再投影显示到屏幕上之后，这些点可能不再构成一个简单的凸多边形。例如，想象一个由 4 个点组成的四边形，它的 4 个点都稍稍偏离原平面。如果从侧面看过去，将会看到一个像蝴蝶结一样的非简单多边形，如图 2-4 所示。这种多边形无法保证能够进行正确的渲染。当利用真实表面上的点组成的四边形来模拟表面时，常常会发生这种情况。为了避免这个问题，可以使用三角形来模拟表面，因为任何三角形都保
证位于同一个平面上。

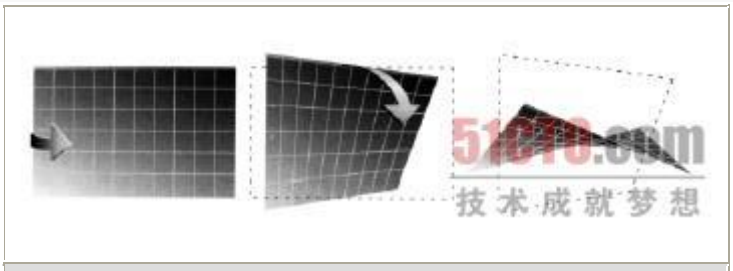


图 2-4 非平面多边形转换为非简单多边形

矩形

由于矩形在图形应用程序中极为常见，OpenGL 特别提供了填充矩形图元函数 glRect*()。矩形的绘制方法类似于绘制多边形，就像在 2.2.3 节描述的那样。但是，读者所使用的特定 OpenGL 实现可能会对用于绘制矩形的 glRect*()函数进行优化。

```
1. void glRect{sifd}(TYPE x1, TYPE y1, TYPE x2, TYPE y2);
2. void glRect{sifd}v(const TYPE *v1, const TYPE *v2);
```

绘制由角顶点（x1, y1）和（x2, y2）定义的矩形。这个矩形位于 z = 0 的平面上，并且它的边与 x 和 y 轴平行。如果使用了这个函数的向量形式，角顶点是由两个数组指针指定的，它们分别包含了一对（x, y）坐标值。

注意，尽管矩形在三维空间中有一个初始的特定方向（在 xy 平面，并且与 z 轴平行），但是可以通过旋转或其他变换更改矩形的方向（关于如何完成这个任务的详细内容，请参阅第 3 章）。

曲线和弯曲表面

所有的曲线或弯曲表面都可以通过模拟实现，并且可以达到任意高的精度，采用的方法是组合大量的短直线或小多边形。因此，只要对曲线和弯曲表面进行足够的细分，并用直线段和平面多边形近似地模拟它们，它们看上去就像是真的弯曲一样（如图 2-5 所示）。

图 2-5 模拟曲线

如果读者怀疑这种做法是否可行，可以想象一下把曲线或弯曲表面不断进行细分，直到每个线段或多边形非常小，甚至比屏幕上的一个像素还小。

尽管曲线并不是几何图元，但 OpenGL 还是提供了一些直接的支持，对它们进行细分以及绘制它们（关于如何绘制曲线和弯曲表面的详细信息，请参阅第 12 章）。

2.2.2 指定顶点

在 OpenGL 中，所有的几何物体最终都描述成一组有序的顶点。glVertex*()函数用于指定顶点。

```
1. void glVertex[234]{sifd}(TYPE coords);
2. void glVertex[234]{sifd}v(const TYPE* coords);
```

指定了一个用于描述几何物体的顶点。可以选择这个函数的适当版本，既可以为一个顶点提供多达 4 个的坐标（x, y, z, w），也可以只提供 2 个坐标（x, y）。如果选择的函数版本并没有显式地指定 z 或 w，z 就会当作 0，w 则默认为 1。glVertext*()函数只有当它位于 glBegin()和 glEnd()之间时才有效。

示例程序 2-2 提供了使用 glVertex*()函数的一些例子。

示例程序 2-2 glVertex*()的合法用法

```
1. glVertex2s(2, 3);
2. glVertex3d(0.0, 0.0, 3.1415926535898);
3. glVertex4f(2.3, 1.0, -2.2, 2.0);
4. GLdouble dvect[3] = {5.0, 9.0, 1992.0};
5. glVertex3dv(dvect);
```

第一个例子表示一个具有三维坐标（2，3，0）的顶点（记住，如果未指定 z 坐标，它便默认为 0）。第二个例子的坐标是（0.0，0.0，3.1415926535898），类型为双精度的浮点值。第三个例子用齐次坐标表示一个具有三维坐标（1.15，0.5，-1.1）的顶点（记住，x、y 和 z 坐标最终将除以 w）。在最后一个例子里，dvect 是一个指向数组的指针，这个数组包含了 3 个双精度浮点值。

在有些计算机上，glVertex*()的向量形式具有更高的效率，这是因为它只需要向图形系统传递 1 个参数。特殊的硬件可以一次发送整个系列的坐标。如果读者使用的机器正好提供了这种硬件，应该对数据进行排列，使顶点坐标在内存中线性地排列在一起。在这种情况下，使用 OpenGL 的顶点数组操作可能会带来性能上的提升（参见第 2.6 节）。

2.2.3 OpenGL 几何图元

我们已经知道了如何指定顶点，现在还需要知道如何告诉 OpenGL 根据这些顶点创建一组点、一条直线或一个多边形。为了实现这个目的，需要把一组顶点放在一对 glBegin()和 glEnd()之间。传递给 glBegin()的参数决定了这些顶点所构建的几何图元的类型。例如，示例程序 2-3 指定了图 2-6 所示的多边形的顶点。

图 2-6 绘制一个多边形或一组点

示例程序 2-3 填充多边形

```
1. glBegin(GL_POLYGON);
2. glVertex2f(0.0, 0.0);
3. glVertex2f(0.0, 3.0);
4. glVertex2f(4.0, 3.0);
5. glVertex2f(6.0, 1.5);
6. glVertex2f(4.0, 0.0);
7. glEnd();
```

如果用 GL_POINTS 代替 GL_POLYGON，这个图元就是如图 2-6 所示的简单的 5 个点。表 2-2 总结了 glBegin()可以使用的 10 种参数以及对应的图元类型。

```
1. void glBegin(GLenum mode);
```

标志着一个顶点数据列表的开始，它描述了一个几何图元。mode 参数指定了图元的类型，它可以是表 2-2 列出的任何一个值。

表 2-2 几何图元的名称和含义

值	
GL_POINTS	单个的点
GL_LINES	一对顶点被解释为一条
GL_LINE_STRIP	一系列的连接直线
GL_LINE_LOOP	和上面相同，但第一个
GL_TRIANGLES	3个顶点被解释为一个
GL_TRIANGLE_STRIP	三角形的连接串
GL_TRIANGLE_FAN	连接成扇形的三角形系
GL_QUADS	4个顶点被解释为一个
GL_QUAD_STRIP	四边形的连接串
GL_POLYGON	简单的凸多边形的边界

```
1. void glEnd(void);
```

标志着一个顶点数据列表的结束。

图 2-7 显示了表 2-2 列出的所有几何图元的例子，并标出了组成每个物体的顶点。注意，除了点之外，它还定义了几种类型的直线和多边形。显然，可以找到许多方法绘制相同的图元，实际所选择的方法取决于具体的顶点数据。

在阅读下面的描述时，可以假设已经在一对 glBegin()和 glEnd()之间描述了 n 个顶点（v0, v1,v2, ., vn-1）。

GL_POINTS	为n个顶点的每一个都绘制一个点
GL_LINES	绘制一系列的非连接直线段。这
	间绘制的。如果n是奇数，最后一条
GL_LINE_STRIP	从v_0到v_1绘制一条直线，然后从v_1

（续）

	的是从v_{n-2}到v_{n-1}的直线。因此， 直线。描述直线串（或者直线环
GL_LINE_LOOP	与GL_LINE_STRIP相似，只
GL_TRIANGLES	绘制一系列的三角形（3条边 二个三角形使用v_3、v_4和v_5，接 2个顶点被忽略
GL_TRIANGLE_STRIP	绘制一系列的三角形（3条边 个三角形使用v_2、v_1和v_3（注意 序是为了保证所有的三角形都是 形成表面的一部分。对于有些操 小节中的“反转和剔除多边形表
GL_TRIANGLE_FAN	和GL_TRIANGLE_STRIP相 v_3，然后是v_0、v_3、v_4，接下来
GL_QUADS	绘制一系列的四边形（4条边 v_5、v_6和v_7，接下来以此类推。
GL_QUAD_STRIP	绘制一系列的四边形（4条边 v_3、v_5、v_4，然后是v_4、v_5、v_7和 4，否则不会绘制任何四边形。
GL_POLYGON	绘制一个多边形，使用点v_0、 绘制任何多边形。另外，它所指 如果顶点不满足这些条件，其结

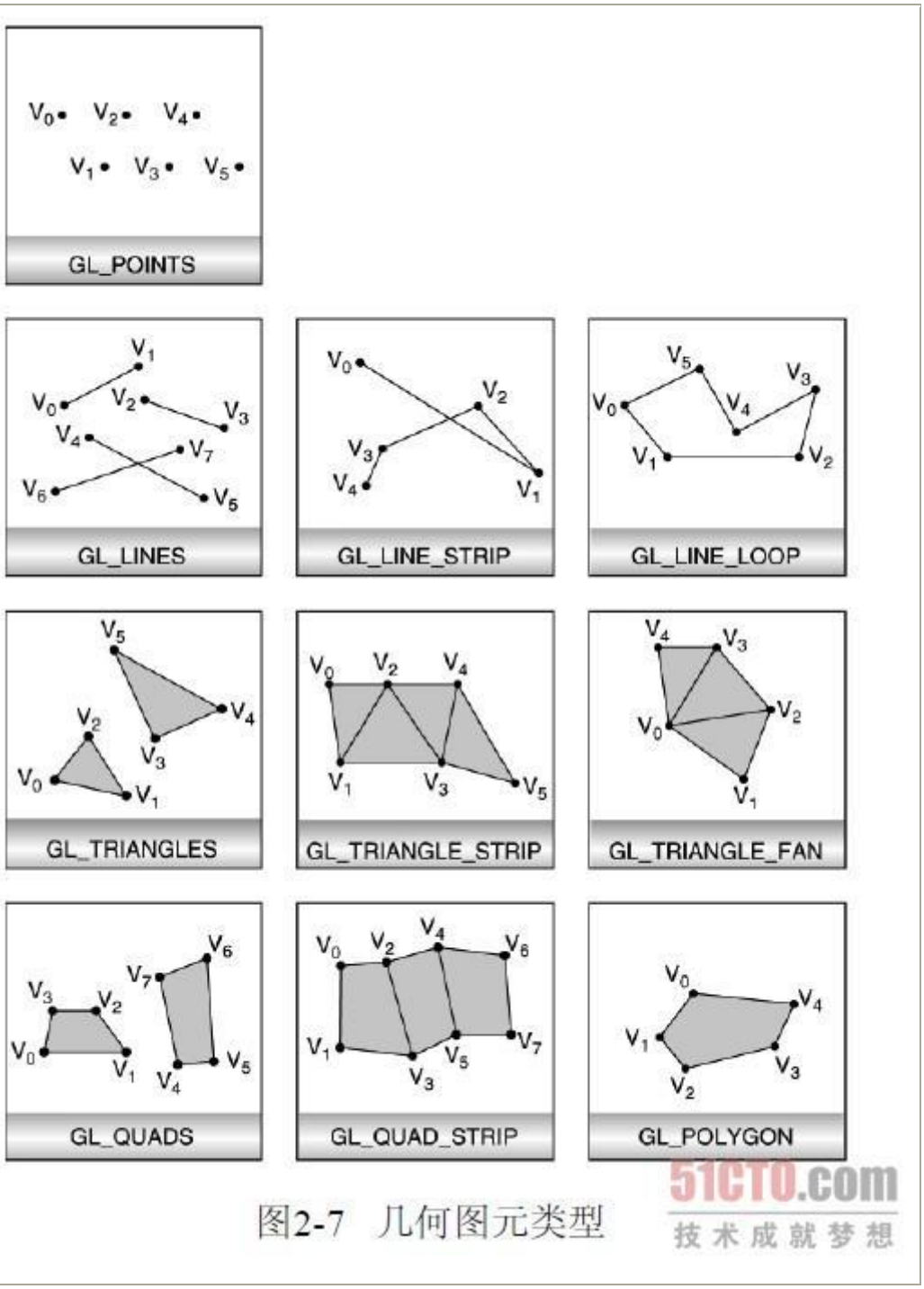


图2-7 几何图元类型

使用 glBegin()和 glEnd()的限制

顶点最重要的信息是它们的坐标，它们是由 glVertex*()函数指定的。另外，还可以使用特殊的函数，为每个顶点指定额外的特定数据，例如颜色、法线向量、纹理坐标，或上述的任意组合。另外，还有一些函数可以在 glBegin()和 glEnd()之间使用。表 2-3 提供了这些函数的完整列表。

表 2-3 glBegin()和 glEnd()之间的合法函数

函数	函数的作用
glVertex*()	设置顶点坐标
glColor*()	设置RGBA颜色
glIndex*()	设置颜色索引
glSecondaryColor*()	设置纹理应用后的辅助颜色
glNormal*()	设置法线向量坐标
glMaterial*()	设置材料属性
glFogCoord*()	设置雾坐标
glTexCoord*()	设置纹理坐标
glMultiTexCoord*()	为多重纹理设置纹理坐标
glVertexAttrib*()	设置通用的顶点属性
glEdgeFlag*()	控制边界的绘制
glArrayElement()	提取顶点数组数据
glEvalCoord*(), glEvalPoint*()	生成坐标
glCallList(), glCallLists()	执行显示列表

除了这些函数之外，glBegin()和 glEnd()之间不能使用其他 OpenGL 函数。如果采用了这样的做法，绝大多数情况下将会出现错误。有些顶点数组函数，例如 glEnableClientState()和 glVertexPointer()，如果在 glBegin()和 glEnd()之间调用，将产生未定义的行为，但并不一定会产生错误。同样，与 OpenGL 相关的函数，例如 glX*()函数，如果在 glBegin()和 glEnd()之间调用，也具有未定义的行为。这种情况应该尽量避免，因为它们很难调试出来。

但是，注意只有 OpenGL 函数受到这个限制，glBegin()和 glEnd()之间当然可以包含其他编程语言结构（除了前面提到的如 glX*()函数等调用之外）。例如，示例程序 2-4 绘制了一个轮廓圆。

示例程序 2-4 glBegin()和 glEnd()之间的其他结构

```
1. #define PI 3.1415926535898
2. GLint circle_points = 100;
3. glBegin(GL_LINE_LOOP);
4. for (i = 0; i < circle_points; i++) {
5. angle = 2*PI*i/circle_points;
6. glVertex2f(cos(angle), sin(angle));
7. }
```

```
8. glEnd();
```

注意：这个例子并不是绘制圆的最有效方法，尤其是想反复绘制多个圆的时候。图形函数的运行速度一般非常快，但这段代码需要为每个顶点计算一个角度，并调用 sin()和 cos()函数。（另外，这段代码还存在循环的开销。计算圆的顶点的另一种方法是使用一个 GLU 函数，参见第 11.2 节。）如果需要绘制大量的圆，可以一次计算所有顶点的坐标，并把它们保存在一个数组里，然后创建一个显示列表（参见第 7 章）或使用顶点数组，对它们进行渲染。

除非被编译到一个显示列表中，否则，所有的 glVertex*()函数都应该出现在 glBegin()和 glEnd()之间。如果它们出现在其他地方，就不会有任何效果。如果它们出现在一个显示列表中，只有当这个显示列表出现在 glBegin()和 glEnd()之间时，它们才会执行（关于显示列表的更多信息，请参阅第 7 章）。

尽管有许多函数可以在 glBegin()和 glEnd()之间调用，但是只有调用 glVertex*()函数才会生成顶点。当 glVertex*()函数被调用时，OpenGL 为最终生成的顶点分配当前的颜色、纹理坐标、法线向量等值。为此，读者可以观察下面的代码序列。第一个点是用红点绘制的，第二和第三个点使用蓝色绘制。尽管这段代码还存在其他的颜色函数调用，但是它们并不会产生效果。

```
1. glBegin(GL_POINTS);
2. glColor3f(0.0, 1.0, 0.0); /* green */
3. glColor3f(1.0, 0.0, 0.0); /* red */
4. glVertex(...);
5. glColor3f(1.0, 1.0, 0.0); /* yellow */
6. glColor3f(0.0, 0.0, 1.0); /* blue */
7. glVertex(...);
8. glVertex(...);
9. glEnd();
```

在 glBegin()和 glEnd()之间，可以使用 glVertex*()函数的 24 个版本的任意组合。不过在真正的应用程序中，对于某个特定的物体，所有的 glVertex*()一般都使用相同的形式。如果应用程序使用的顶点数据规范是一致的和可重复的（例如 glColor*、glVertex*、glColor*、glVertex*、..），可以使用顶点数组来提高应用程序的性能（参见第 2.6 节）。

2.3 基本状态管理

在前一节中，我们已经看到了一个状态变量的例子，即当前 RGBA 颜色。我们还看到了如何把它与图元相关联。OpenGL 维护了许多状态和状态变量。物体在进行渲染时可能会使用光照、纹理、隐藏表面消除、雾以及其他影响物体外观的状态。

在默认情况下，这些状态的大部分一开始是处于不活动状态的。激活这些状态可能需要较大的开销。例如，启用纹理贴图几乎肯定会减慢图元渲染的速度。但是，图像的质量可以得到明显的提高，看上去更逼真，这归功于增强的图形功能。

为了打开或关闭这些状态，可以使用下面这两个简单的函数：

```
1. void glEnable(GLenum capability);
2. void glDisable(GLenum capability);
```

glEnable()启用一个功能，glDisable()用于关闭一个功能。程序员可以向 glEnable()或 glDisable()传递超过 60 个的枚举值作为参数， 例如 GL_BLEND （ 用于控制 RGBA 颜色的混合）、GL_DEPTH_TEST（用于控制深度比较，并对深度缓冲区进行更新）、GL_FOG（控制雾）、GL_LINE_STIPPLE（直线的点画模式）和 GL_LIGHTING（光照）等。

另外，还可以查询一个状态当前是处于打开还是关闭状态。

```
1. GLboolean glIsEnabled(GLenum capability)
```

根据被查询的状态当前处于启用还是禁用状态，它返回 GL_TRUE 或 GL_FALSE。

目前读者看到的状态都具有两个值：打开或关闭。但是，大部分 OpenGL 函数可以用来设置更为复杂的状态变量。例如，glColor3f()函数可以设置 3 个值，它们都是 GL_CURRENT_COLOR 状态的一部分。可以使用的查询函数共有 5 个，可以查询许多状态的当前值：

```
1. void glGetBooleanv(GLenum pname, GLboolean *params);
2. void glGetIntegerv(GLenum pname, GLint *params);
3. void glGetFloatv(GLenum pname, GLfloat *params);
4. void glGetDoublev(GLenum pname, GLdouble *params);
5. void glGetPointerv(GLenum pname, GLvoid **params);
```

这些函数分别用于获取布尔型、整型、单精度浮点型、双精度浮点型以及指针类型的状态变量。pname 参数是一个符号常量，表示需要返回的状态变量。params 是一个数组指针，指向包含返回数据的位置。可以参阅附录 B 的表格，了解 pname 的可用值。例如，为了获取当前的 RGBA 颜色，附录 B 的一张表格建议使用 glGetIntergerv (GL_CURRENT_COLOR, params)或 glGetFloatv(GL_ CURRENT_COLOR, params)。在必要的时候，这些函数会执行类型转换，以返回与被查询类型相匹配的变量。

这些函数负责绝大多数（但不是全部）的状态信息查询任务。可以参阅第 B.1 节，了解所有可用的 OpenGL 状态查询的函数。

2.4 显示点、直线和多边形

在默认情况下，点被画成屏幕上的 1 个像素，直线被画成宽度为 1 个像素的实线，而多边形则被画成实心填充的形式。下面几段内容讨论如何更改这些默认的显示模式。

2.4.1 点的细节

为了控制被渲染的点的大小，可以使用 glPointSize()函数，并在参数中提供一个值，表示所需要的点的大小（以像素为单位）。

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

设置被渲染点的宽度，以像素为单位。size 必须大于 0.0，在默认情况下为 1.0。

屏幕上所绘制的各种宽度的点所包含的像素集合取决于是否启用了抗锯齿功能(抗锯齿是一种在渲染点和直线时对它们进行平滑处理的技巧，详见第 6.2 节和第 6.4 节）。如果抗锯齿功能被禁用（默认情况），带小数的宽度值将四舍五入为整型值，在屏幕上所绘制的是对齐的正方形像素区域。因此，如果宽度值是 1.0，这个方块的大小就是 1 个像素乘以 1 个像素；如果宽度为 2.0，这个方块就是 2 个像素乘以 2 个像素，以此类推。

如果启用了抗锯齿或多重采样，屏幕上所绘制的将是一个圆形的像素区域。一般情况下，位于边界的像素所使用的颜色强度较小，使边缘具有更平滑的外观。在这种模式下，非整型的宽度值并不会四舍五入。

大多数 OpenGL 实现都支持渲染非常大的点。可以使用 glGetFloatv() 函数（ 以 GL_ALIASED_POINT_RANGE 为参数）查询在未进行抗锯齿处理的情况下最小和最大的点。类似地，可以在这个函数中使用 GL_SMOOTH_POINT_SIZE_RANGE 为参数查询在进行了抗锯齿处理的情况下最小和最大的点。OpenGL 支持的非抗锯齿点的大小均匀地分布在最小和最大范围之间。以 GL_SMOOTH_POINT_SIZE_GRANULARITY 为参数调用 glGetFloatv()函数将返回 OpenGL 支持的特定抗锯齿点大小的精度。例如，如果调用 glPointSize(2.37)，并且返回的粒度值为 0.1，那么这个点的大小将四舍五入为 2.4。

2.4.2 直线的细节

在 OpenGL 中， 可以指定不同宽度的直线， 也可以指定不同点画模式的直线， 如点线（doottedline）、用点和短直线交替绘制而成的段线（dash line）等。

直线的宽度

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

以像素为单位设置宽度，用于直线的渲染。width 参数必须大于 0.0，在默认情况下为 1.0。OpenGL 3.1 不支持大于 1.0 的值， 并且如果指定了一个大于 1.0 的值， 将会产生一个 GL_INVALID_VALUE 错误。

直线的实际渲染还受到是否启用了抗锯齿处理和多重采样功能的影响（参见第 6.2.1 节和第 6.2.2 节）。如果未使用抗锯齿功能，那么宽度为 1、2 和 3 的直线将分别画成 1、2 和 3 个像素的宽度。如果启用了抗据齿功能，它就允许使用非整数的宽度，位于边界处的像素一般会画得淡一些。特定的 OpenGL 实现可以限制非抗锯齿直线的宽度，把它限制在最大抗锯齿直线宽度之内，并四舍五入为最邻近的整数值。为了查询系统所支持的带锯齿直线宽度的范围， 可以使用 GL_ALIASED_LINE_WIDTH_RANGE 为参数调用 glGetFloatv()函数。为了当前 OpenGL 实现所支持的抗锯齿直线的最小和最大宽度， 以及它所支持的直线宽度的粒度， 分别可以使用 GL_SMOOTH_LINE_WIDTH_RANGE 和 GL_SMOOTH_LINE_WIDTH_GRANULARITY 为参数调用 glGetFloatv()函数。

注意：记住，在默认情况下直线的宽度为 1 个像素，因此它们在低分辨率的屏幕上看起来会显得更粗一点。对于计算机画面而言，这一般不会造成什么问题。但是，如果使用 OpenGL 渲染到一台高分辨率的绘图仪，1 个像素宽的直线可能接近于不可见。为了获取与分辨率无关的直线宽度，需要考虑像素的物理大小。

高级话题

在未使用抗锯齿功能的情况下，直线的宽度并不是根据与直线垂直的方向进行测量的。实际上，如果直线斜率的绝对值小于 1.0，它是根据 y 轴的方向进行测量的。否则，它就根据 x 轴的方向进行测量。抗锯齿直线的渲染方式就相当于按照这个特定的宽度渲染一个填充多边形，这个多边形的位置恰好与这条直线准确对应。

点画线

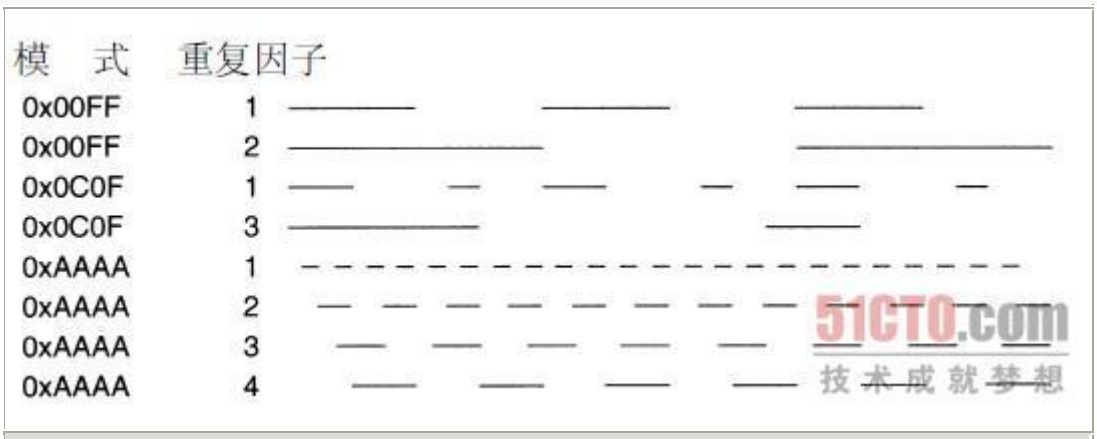
为了创建点画线（点线或段线），可以使用 glLineStipple()函数定义点画模式，然后用 glEnable()函数启用直线点画功能。

```
1. glLineStipple(1, 0x3F07);
2. glEnable(GL_LINE_STIPPLE);
3. void glLineStipple(GLint factor, GLushort pattern);
```

设置直线的当前点画模式。pattern 参数是一个由 1 或 0 组成的 16 位序列，它们根据需要进行重复，对一条特定的直线进行点画处理。从这个模式的低位开始，一个像素一个像素地进行处理。如果模型中对应的位是 1，就绘制这个像素，否则就不绘制。模式可以使用 factor 参数（表示重复因子）进行扩展，它与 1 和 0 的连续子序列相乘。因此，如果模式中出现了连续 3 个 1，并且 factor 是 2，那么它们就扩展为 6 个连续的 1。必须以 GL_LINE_STIPPLE

为参数调用 glEnable()才能启用直线点画功能。为了禁用直线点画功能，可以向 glDisable()函数传递同一个参数。

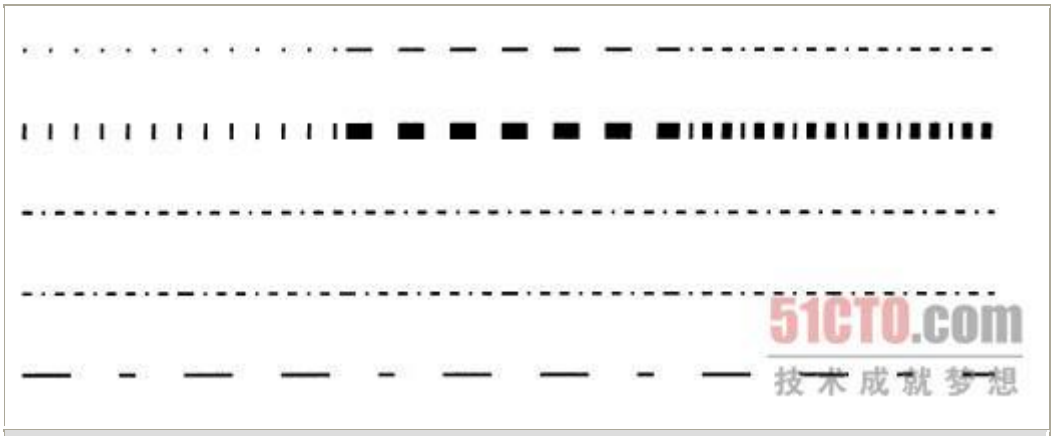
在前面那个例子中，如果模式为 0x3F07（二进制形式为 0011111100000111），它所画出来的直线是这样的：先是连续绘制 3 个像素，然后连续 5 个像素留空，然后再连续绘制 6 个像素，最后 2 个像素留空（注意，首先开始的是低位）。如果 factor 是 2，那么这个模式便被扩展为：绘制 6 个像素、留空 10 个像素、绘制 12 个像素，最后留空 4 个像素。图 2-8 显示了用不同的模式和重复因子所绘制的点画线。如果没有启用点画线功能，OpenGL 会自动把 pattern 当成是 0xFFFF，把 factor 当成 1（以 GL_LINE_STIPPLE 为参数调用 glDisable()函数可以禁用点画线功能）。注意，点画线可以与宽直线一起使用，以产生宽点画线。



（点击查看大图）图 2-8 点画线

也可以按照下面这种方式考虑点画线：在开始绘制直线时，每绘制 1 个像素（如果 factor 参数不为 1，则为 factor 个像素）时，模式就移动 1 位。在一对 glBegin()和 glEnd()之间绘制一系列的连接线段时，每画完一个线段转向下一线段时，模式就会移动。这样，点画线模式就会沿着一系列的线段移动。当 glEnd()函数执行时，模式就被重置。如果在禁用点画线功能之前还有其他直线需要绘制，便从头开始应用这个模式。如果使用 GL_LINES 绘制直线，每画完一条直线之后，模式也会被重置。

示例程序 2-5 演示了用一些不同的点画模式和直线宽度进行绘图的结果。它还演示了如果这些直线被画成一系列的独立直线而不是连接直线串时会发生什么情况。图 2-9 显示了这个程序的运行结果。



（点击查看大图）图 2-9 宽点画线

示例程序 2-5 直线点画模式：lines.c

```
1. #define drawOneLine(x1,y1,x2,y2) glBegin(GL_LINES); \
2. glVertex2f((x1),(y1)); glVertex2f((x2),(y2)); glEnd();
3. void init(void)
4. {
5. glClearColor(0.0, 0.0, 0.0, 0.0);
6. glShadeModel(GL_FLAT);
7. }
8. void display(void)
9. {
10. int i;
11. glClear(GL_COLOR_BUFFER_BIT);
12. /* select white for all lines */
13. glColor3f(1.0, 1.0, 1.0);
14. /* in 1st row, 3 lines, each with a different stipple */
15. glEnable(GL_LINE_STIPPLE);
16. glLineStipple(1, 0x0101); /* dotted */
17. drawOneLine(50.0, 125.0, 150.0, 125.0);
18. glLineStipple(1, 0x00FF); /* dashed */
19. drawOneLine(150.0, 125.0, 250.0, 125.0);
20. glLineStipple(1, 0x1C47); /* dash/dot/dash */
21. drawOneLine(250.0, 125.0, 350.0, 125.0);
22. /* in 2nd row, 3 wide lines, each with different stipple */
23. glLineWidth(5.0);
24. glLineStipple(1, 0x0101); /* dotted */
25. drawOneLine(50.0, 100.0, 150.0, 100.0);
26. glLineStipple(1, 0x00FF); /* dashed */
27. drawOneLine(150.0, 100.0, 250.0, 100.0);
```

```
glLineStipple(1, 0x1C47); /* dash/dot/dash */
drawOneLine(250.0, 100.0, 350.0, 100.0);
glLineWidth(1.0);
/* in 3rd row, 6 lines, with dash/dot/dash stipple */
/* as part of a single connected line strip */
glLineStipple(1, 0x1C47); /* dash/dot/dash */
glBegin(GL_LINE_STRIP);
for (i = 0; i < 7; i++)
glVertex2f(50.0 + ((GLfloat) i * 50.0), 75.0);
glEnd();
/* in 4th row, 6 independent lines with same stipple */
for (i = 0; i < 6; i++) {
drawOneLine(50.0 + ((GLfloat) i * 50.0), 50.0,
50.0 + ((GLfloat)(i+1) * 50.0), 50.0);
}
/* in 5th row, 1 line, with dash/dot/dash stipple */
/* and a stipple repeat factor of 5 */
glLineStipple(5, 0x1C47); /* dash/dot/dash */
drawOneLine(50.0, 25.0, 350.0, 25.0);
glDisable(GL_LINE_STIPPLE);
glFlush();
}
void reshape(int w, int h)
{
glViewport(0, 0, (GLsizei) w, (GLsizei) h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, (GLdouble) w, 0.0, (GLdouble) h);
}
int main(int argc, char** argv)
{
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize(400, 150);
glutInitWindowPosition(100, 100);
glutCreateWindow(argv[0]);
init();
glutDisplayFunc(display);
```

```
66. glutReshapeFunc(reshape);
67. glutMainLoop();
68. return 0;
69. }
```

2.4.3 多边形的细节（1）

一般情况下，多边形是按填充模式绘制的，边界之内的像素均被绘制。但是，也可以把它们画成轮廓形式，甚至只绘制它们的顶点。填充多边形可以是实心填充，也可以用某种模式进行点画填充。尽管这里省略了具体的细节，但我们还是要说明一点，相邻的填充多边形如果共享一条边或一个顶点，组成这条边或这个顶点的像素只绘制一次，它们只包含在其中一个多边形中。这样，部分透明的多边形的边不会绘制两次，它们的边缘看上去不会更暗（或更亮，取决于具体使用的绘图颜色）。注意，这可能导致狭窄多边形有一行或多行（或列）像素未绘制。

为了对填充多边形进行抗锯齿处理，强烈推荐使用多重采样。关于这方面的细节，请参阅第 6.2.2 节。

点、轮廓或实心形式的多边形多边形具有正面和背面两个面。取决于哪一面朝向观察者，多边形可能会被渲染成不同的样子。这样，就可以获得实心物体正反两面截然不同的剖面视图。在默认情况下，多边形的正面和背面是按照相同的方式绘制的。为了更改这个行为，或者只绘制它的轮廓或顶点，可以使用 glPolygonMode()函数。

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

控制一个多边形的正面和背面的绘图模式。face 参数可以是 GL_FRONT_AND_BACK、GL_FRONT 或 GL_BACK。mode 参数可以是 GL_POINT、GL_LINE 或 GL_FILL，表示多边形应该被画成点、轮廓还是填充形式。在默认情况下，多边形的正面和背面都画成填充形式。

OpenGL 3.1 只接受 GL_FRONT_AND_BACK 作为 face 的值，并且不管是多边形的正面还是背面都以相同的方式渲染。

例如，可以通过下面这两个调用，把多边形的正面画成填充形式，把背面画成轮廓形式：

```
1. glPolygonMode(GL_FRONT, GL_FILL);
2. glPolygonMode(GL_BACK, GL_LINE);
```

反转和剔除多边形表面

按照约定，如果多边形的顶点以逆时针顺序出现在屏幕上，它便称为“正面”。可以根据方向一致的多边形构建任何“合理的” 实心表面。按照数学的术语， 这种表面称为可定向簇

（orientablemainifold）。例如，球体、圆环体和茶壶都是可定向的，克莱因瓶（Klein bottles）和麦比乌斯带（M.bius strip）都是不可定向的。换句话说，为了创建可定向的表面，可以使用全部是逆时针方向的多边形，也可以使用全部是顺时针方向的多边形，这正是可定向的数学定义。

假设我们根据一种一致性的方式描述了一个可定向表面的模型，但是它的外侧恰好为顺时针方向。在这种情况下，可以使用 glFrontFace()函数，向它传递一个参数，表示希望把多边形的哪一面作为正面，从而交换了 OpenGL 的正面和背面的概念。

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

控制多边形的正面是如何决定的。在默认情况下，mode 是 GL_CCW，它表示窗口坐标上投影多边形的顶点顺序为逆时针方向的表面为正面。如果 mode 是 GL_CW，顶点顺序为顺时针方向的表面被认为是正面。

注意：顶点的方向（顺时针或逆时针）又称为环绕（winding）。

在一个完全闭合的表面（由方向一致的不透明多边形所组成）上，所有的背面多边形都是不可见的，因为它们总是被多边形的正面所遮挡。如果观察者位于这个表面的外侧，可以启用剔除（culling）功能，丢弃那些被 OpenGL 认为是背面的多边形。类似地，如果观察者位于物体的内侧，只有背面的多边形才是可见的。为了告诉 OpenGL 丢弃哪些不可见的正面或背面多边形，可以使用 glCullFace()函数。当然，在此之前必须调用 glEnable()函数启用剔除功能。

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

表示哪些多边形在转换到屏幕坐标之前应该丢弃（剔除）。mode 参数可以是 GL_FRONT、GL_BACK 或 GL_FRONT_AND_BACK，分别表示正面多边形、背面多边形和所有多边形。为了使剔除生效，必须以 GL_CULL_FACE 为参数调用 glEnable()函数来启用剔除功能。另外，可以用同一个参数调用 glDisable()函数禁用剔除功能。

高级话题

根据更规范的术语，多边形的一个面是正面还是背面取决于窗口坐标计算产生的多边形区域的符号。计算多边形区域的其中一种方法是：

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

其中，xi 和 yi 是 n 顶点的多边形的第 i 个顶点的 x 和 y 窗口坐标，而

$i \oplus 1$是 (i+1) mod n。

假设指定了 GL_CCW，如果 a > 0，那么与顶点对应的多边形便被认为是正面的。反之，它便被认为是背面的。如果指定了 GL_CW，并且 a < 0，那么与顶点对应的多边形便被认为是正面的，否则就是背面的。

尝试一下

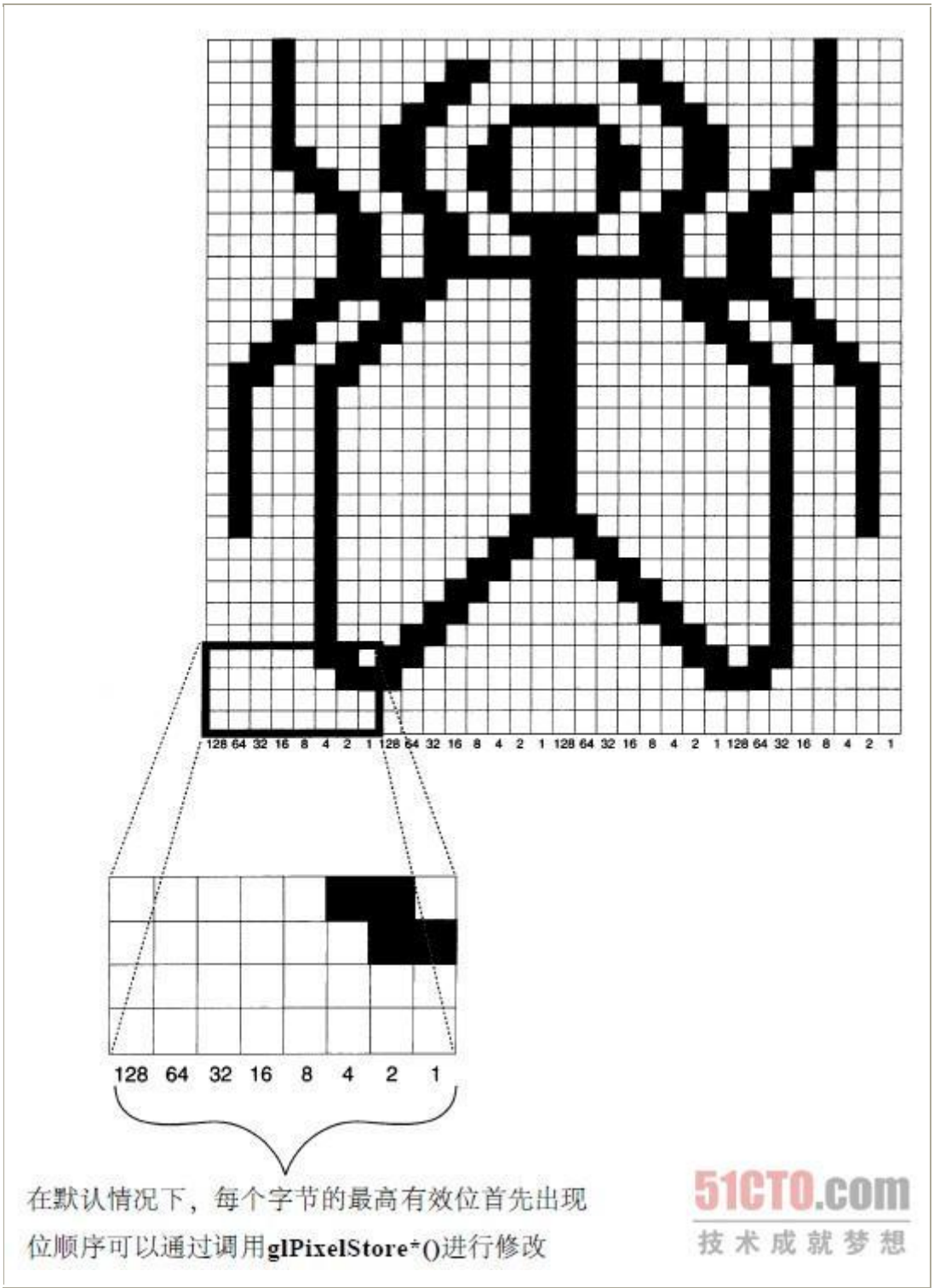
对示例程序 2-5 进行修改，添加一些填充多边形。可以尝试使用不同的颜色，并尝试不同的多边形模式。另外，可以启用剔除功能来观察它的效果。

点画多边形

在默认情况下，填充多边形是用实心模式绘制的。此外，它们还可以使用一种 32 位×32 位的窗口对齐的点画模式。glPolygonStipple()函数用于指定多边形的点画模式。

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

定义填充多边形的当前点画模式。mask 参数是一个指向 32×32 位图的指针，后者被解释为 0 和 1 的掩码。如果模式中出现的是 1，那么多边形中对应的像素就被绘制；如果出现的是 0，多边形中对应的像素就不被绘制。图 2-10 显示了如何根据 mask 中的字符构建点画模式。可以使用 GL_POLYGON_STIPPLE 为参数调用 glEnable()和 glDisable()函数，分别启用和禁用多边形点画功能。mask 数据的解释受到 glPixelStore*()GL_UNPACK*模式的影响（参见第 8.3.2 节）。



（点击查看大图）图 2-10 创建一个多边形点画模式

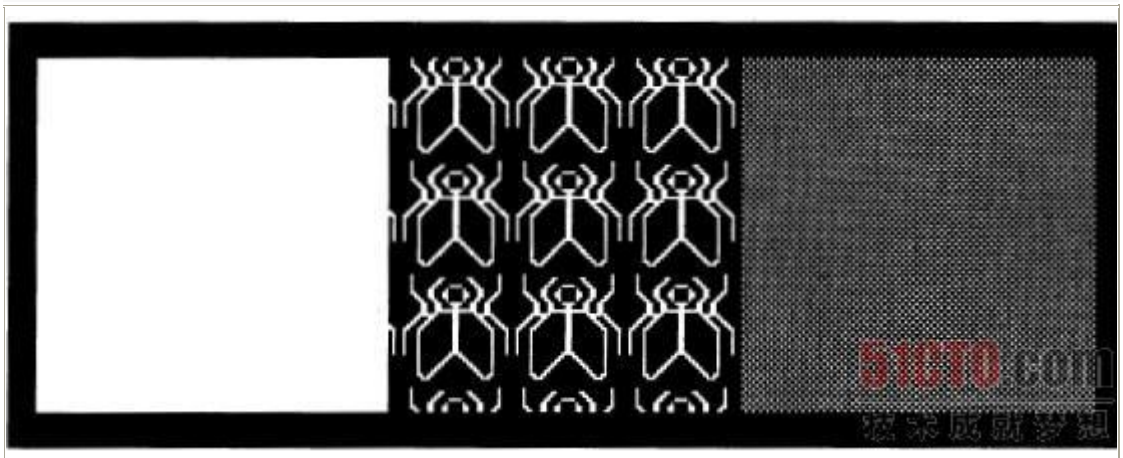
2.4.3 多边形的细节（2）

除了定义当前多边形的点画模式之外，还必须启用多边形点画功能：

```
1. glEnable(GL_POLYGON_STIPPLE);
```

用同一个参数调用 glDisable()函数可以禁用多边形点画功能。

图 2-11 显示了一个不使用点画模式的多边形，然后是两个使用不同点画模式的多边形。示例程序 2-6 显示了这个程序的源代码。从图 2-10 至图 2-11 所出现的从白色到黑色的反转是因为这个程序是在黑色背景上用白色进行绘图的，并把图 2-10 的模式作为模板。



（点击查看大图）图 2-11 点画多边形

示例程序 2-6 多边形点画模式：polys.c

```
1. void display(void)
2. {
3. GLubyte fly[] = {
4. 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
5. 0x03, 0x80, 0x01, 0xC0, 0x06, 0xC0, 0x03, 0x60,
6. 0x04, 0x60, 0x06, 0x20, 0x04, 0x30, 0x0C, 0x20,
7. 0x04, 0x18, 0x18, 0x20, 0x04, 0x0C, 0x30, 0x20,
8. 0x04, 0x06, 0x60, 0x20, 0x44, 0x03, 0xC0, 0x22,
9. 0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
10. 0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
11. 0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
12. 0x66, 0x01, 0x80, 0x66, 0x33, 0x01, 0x80, 0xCC,
13. 0x19, 0x81, 0x81, 0x98, 0x0C, 0xC1, 0x83, 0x30,
14. 0x07, 0xe1, 0x87, 0xe0, 0x03, 0x3f, 0xfc, 0xc0,
15. 0x03, 0x31, 0x8c, 0xc0, 0x03, 0x33, 0xcc, 0xc0,
16. 0x06, 0x64, 0x26, 0x60, 0x0c, 0xcc, 0x33, 0x30,
17. 0x18, 0xcc, 0x33, 0x18, 0x10, 0xc4, 0x23, 0x08,
18. 0x10, 0x63, 0xC6, 0x08, 0x10, 0x30, 0x0c, 0x08,
19. 0x10, 0x18, 0x18, 0x08, 0x10, 0x00, 0x00, 0x08};
20.
21. GLubyte halftone[] = {
22. 0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
```

```
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,

0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55};
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0, 1.0, 1.0);
/* draw one solid, unstippled rectangle, */
/* then two stippled rectangles */
glRectf(25.0, 25.0, 125.0, 125.0);
glEnable(GL_POLYGON_STIPPLE);
glPolygonStipple(fly);
glRectf(125.0, 25.0, 225.0, 125.0);
glPolygonStipple(halftone);
glRectf(225.0, 25.0, 325.0, 125.0);
glDisable(GL_POLYGON_STIPPLE);
glFlush();
}
void init(void)
{
glClearColor(0.0, 0.0, 0.0, 0.0);
glShadeModel(GL_FLAT);
}

void reshape(int w, int h)
{
glViewport(0, 0, (GLsizei) w, (GLsizei) h);
```

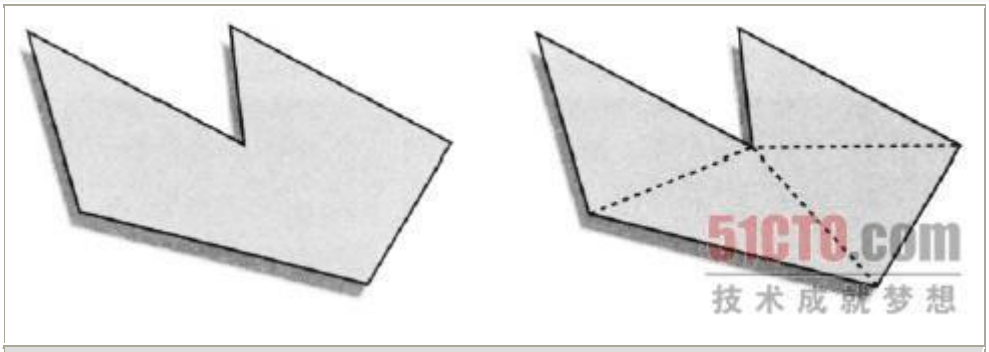
```
61. glMatrixMode(GL_PROJECTION);
62. glLoadIdentity();
63. gluOrtho2D(0.0, (GLdouble) w, 0.0, (GLdouble) h);
64. }
65. int main(int argc, char** argv)
66. {
67. glutInit(&argc, argv);
68. glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
69. glutInitWindowSize(350, 150);
70. glutCreateWindow(argv[0]);
71. init();
72. glutDisplayFunc(display);
73. glutReshapeFunc(reshape);
74. glutMainLoop();
75. return 0;
76. }
```

读者可能想要使用显示列表来存储多边形点画模式来使效率最大化（参见第 7.3 节）。

标记多边形的边界边

高级话题

OpenGL 只能渲染凸多边形，但是在实际应用中可以看到很多非凸多边形。为了绘制这些非凸多边形，一般把它们分解为几个凸多边形（通常是三角形，如图 2-12 所示），然后再分别绘制这些三角形。遗憾的是，如果把一个普通的多边形分解为几个三角形，然后再分别绘制这些三角形，就无法使用 glPolygonMode()函数绘制多边形的真正轮廓，我们所看到的是它内部的这些三角形的轮廓。为了解决这个问题，可以告诉 OpenGL 一个特定的顶点是否是一条边界边的起点。OpenGL 将追踪这个信息，用 1 个位来记录每个顶点，表示这个顶点是否为一条边界边的起点。然后， 当 OpenGL 使用 GL_LINE 模式绘制这个多边形时，那些非边界边就不会绘制。在图 2-12 中，虚线表示多边形内部的所有非边界边。



（点击查看大图）图 2-11 点画多边形

在默认情况下，所有的顶点都标记为边界边的起点，但是可以使用 glEdgeFlag*()函数手工控制边界标志（edge flag）的设置。这个函数在 glBegin()和 glEnd()之间调用，它将影响在它之后所指定的所有顶点，直到再次调用 glEdgeFlag()函数。它只作用于那些为多边形、三角形和四边形所指定的顶点，对那些为三角形带或四边形带所指定的顶点无效。

```
1. void glEdgeFlag(GLboolean flag);
2. void glEdgeFlagv(const GLboolean *flag);
```

表示一个顶点是否应该被认为是多边形的一条边界边的起点。如果 flag 是 GL_TRUE，边界标志就设置为 TRUE（默认），在此之后创建的所有顶点都认为是边界边的起点，直到用 GL_FALSE 为 flag 参数的值再次调用了这个函数。

例如，示例程序 2-7 绘制了图 2-13 所示的轮廓。

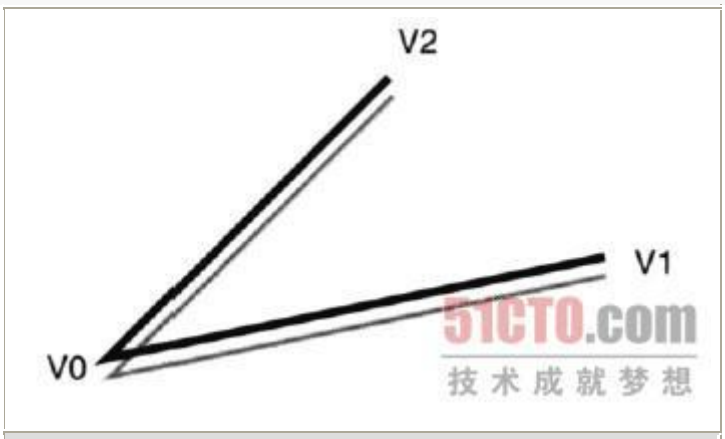


图 2-13 使用边界标志绘制轮廓多边形

示例程序 2-7 标记多边形边界边

```
1. glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
2. glBegin(GL_POLYGON);
3. glEdgeFlag(GL_TRUE);
4. glVertex3fv(V0);
5. glEdgeFlag(GL_FALSE);
6. glVertex3fv(V1);
7. glEdgeFlag(GL_TRUE);
8. glVertex3fv(V2);
9. glEnd();
```

2.5 法线向量

法线向量（或简称为法线）是一条垂直于某个表面的方向向量。对于平表面而言，它上面每个点的垂直方向都是相同的。但是，对于普通的曲面而言，表面上每个点的法线方向可能各不相同。在 OpenGL 中，既可以为每个多边形指定一条法线，也可以为多边形的每个顶

点分别指定一条法线。同一个多边形的顶点可能共享同一条法线（平表面），也可能具有不同的法线（曲面）。除了顶点之外，不能为多边形的其他地方分配法线。

物体的法线向量定义了它的表面在空间中的方向。具体地说，定义了它相对于光源的方向。OpenGL 使用法线向量确定这个物体的各个顶点所接收的光照。光照本身是一个非常庞大的主题，我们将在第 5 章对它进行详细介绍。在读完第 5 章之后，读者可能需要回顾这一节的内容。在这里，我们只是简单地讨论法线向量，因为在定义物体的几何形状时，同时也定义了它的法线向量。

可以使用 glNormal*()函数，把当前的法线向量设置为这个函数的参数所表示的值。以后调用 glVertex*()时，就会把当前法线向量分配给它所指定的顶点。每个顶点常常具有不同的法线，因此需要交替调用这两个函数，如示例程序 2-8 所示。

示例程序 2-8 顶点上的表面法线

```
glBegin (GL_POLYGON);
glNormal3fv(n0);
glVertex3fv(v0);
glNormal3fv(n1);
glVertex3fv(v1);
glNormal3fv(n2);
glVertex3fv(v2);
glNormal3fv(n3);
glVertex3fv(v3);
glEnd();

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

根据参数设置当前的法线向量。这个函数的非向量版本（没有 v）接受 3 个参数，把一个（nx, ny,nz）向量指定为法线向量。此外，还可以使用这个函数的向量版本（带 v），并提供一个包含 3 个元素的数组来指定所需的法线向量。b、s 和 i 版本的函数会对它们的参数值进行线性缩放，使它们位于范围[-1.0, 1.0]之间。

寻找物体的法线向量并没有神奇之处。在很多情况下，需要执行一些涉及求导的计算。但是，可以使用一些技巧来实现特定的效果。附录 H 解释了如何寻找表面的法线向量。如果读者已经知道这些方法，或者总是可以使用现成的法线向量，或者不想使用 OpenGL 的光照功能，就可以忽略附录 H（附录 H 可以通过链接 http://www.opengl-redbook.com/appendices/ 访问）。

注意，在表面的一个特定的点上，有两条向量垂直于这个表面，它们指向相反的方向。按照约定，指向表面外侧的那条向量就是它的法线。如果想反转模型的内侧和外侧，只要把所有的法线向量从（x, y, z）修改为（-x, -y, -z）就可以了。

另外需要记住的是，由于法线向量只表示方向，因此它的长度是无关紧要的。法线可以指定为任意长度，但是在执行光照计算之前，它的长度会转换为 1（长度为 1 的向量称为单位向量或规范化的向量）。一般而言，我们应该提供规范化的法线向量。为了使一条法线向量具有单位长度，只要把它的每个 x、y 和 z 成分除以法线的长度就可以了：

$$\text{法线的长度} = \sqrt{x^2 + y^2 + z^2}$$

如果模型变换只涉及旋转和移动，法线向量就能够保持它的规范化（参见第 3 章）。如果进行了不规则变换（例如进行了缩放或者乘以了剪切矩阵），或者指定了非单位长度的法线，那么在经过变换之后，OpenGL 会自动对法线向量进行规范化。为了启用这个功能，可以调用 glEnable(GL_ NORMALIZE)。

如果提供了单位长度的法线，并且只执行均匀的缩放（也就是说，在 x、y 和 z 方向上使用相同的缩放因子），就可以使用 glEnable(GL_RESCALE_NORMAL)，用一个常量因子（从模型视图变换矩阵得出）对法线进行缩放，使它们在变换之后恢复为单位长度。

注意，自动规范化或重新缩放一般需要额外的计算，因此可能会降低应用程序的性能。用 GL_RESCALE_NORMAL 对法线进行均匀缩放通常要比使用 GL_NORMALIZE 进行完整的规范化开销更少。在默认情况下，法线的自动规范化和重新缩放操作都是禁用的。

2.6 顶点数组

读者可能已经注意到，OpenGL 需要进行大量的函数调用才能完成对几何图元的渲染。绘制一个 20 条边的多边形至少需要 22 个函数调用。首先调用 1 次 glBegin()，然后为每个顶点调用 1 次函数，最后调用 1 次 glEnd()。在前面的两个代码示例中，由于还需要额外的信息（多边形边界标志或表面法线），所以在每个顶点上还要增加函数调用。这可能会成倍地增加渲染几何物体所需要的函数调用数量。在有些系统中，函数调用具有相当大的开销，可能会影响应用程序的性能。

另外一个问题是相邻多边形的共享顶点的冗余处理。例如，图 2-14 的立方体具有 6 个面和 8 个共享顶点。遗憾的是，如果按照标准方法描述这个物体，每个顶点必须指定 3 次，分别用于每个需要使用这个顶点的面。这样，一共指定了 24 个顶点，尽管实际上只要处理 8 个顶点就够了。

OpenGL 提供了一些顶点数组函数，允许只用少数几个数组指定大量的与顶点相关的数据，并用少量函数调用（与顶点数组的数量相仿）访问这些数据。使用顶点数组函数，一个拥有 20 条边的多边形的 20 个顶点可以放在 1 个数组中，并且只通过 1 个函数进行调用。如果每个顶点还有一条法线向量，所有 20 条法线向量可以放在另一个数组中，也可以只通过 1 个函数进行调用。

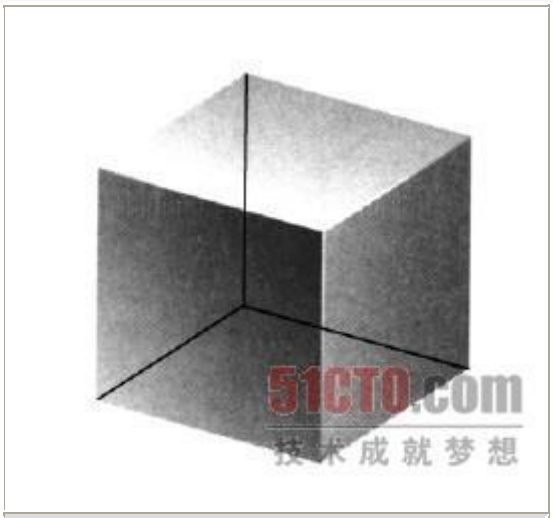


图 2-14 6 个面，8 个共享顶点

把数据放在顶点数组中可以提高应用程序的性能。使用顶点数组可以减少函数调用的次数，从而提高性能。另外，使用顶点数组还可以避免共享顶点的冗余处理。

注意：顶点数组是在 OpenGL 1.1 版中成为标准的。1.4 版本增加了对在顶点数组中存储雾坐标和辅助颜色的支持。

使用顶点数组对几何图形进行渲染需要 3 个步骤：

1) 激活（启用）最多可达 8 个数组，每个数组用于存储不同类型的数据：顶点坐标、表面法线、RGBA 颜色、辅助颜色、颜色索引、雾坐标、纹理坐标以及多边形的边界标志。

2) 把数据放入数组中。这些数组是通过它们的内存位置的地址（即指针）进行访问的。在客户机-服务器模型中，这些数组存储在客户机的地址空间中，除非选择使用缓冲区对象（参见 2.7 节），这时候，数组存储在服务器内存中。

3) 用这些数据绘制几何图形。OpenGL 通过指针从所有的被激活数组中获取数据。在客户机-服务器模型中，数据被传输到服务器的地址空间中。有 3 种方式可以完成这个任务：

访问单独的数组元素（随机存取）。

创建一个单独数组元素的列表（系统存取）。

线性地处理数组元素。

具体选择的数据访问方式取决于需要处理的问题类型。OpenGL 1.4 版本增加了在 1 个函数调用中访问多个数组的功能。

另一种常用的数据组织方式是混合顶点数组（interleaved vertex array）数据。和普通的使用多个不同的数组、让每个数组保存一种不同类型的数据（颜色、表面法线、坐标等）不同，我们也可以把不同类型的数据混合放在同一个数组中（参见第 2.6.6 节）。

2.6.1 步骤 1：启用数组

第一个步骤是调用 glEnableClientState()函数（使用一个枚举值参数），激活选择的数组。从理论上说，最多可能调用这个函数 8 次，激活 8 个可用的数组。但是在实践中，可以激活的数组最多只有 6 个， 这是因为有些数组不能同时激活。例如， 不可能同时激活 GL_COLOR_ARRAY 和 GL_INDEX_ARRAY。应用程序的显示模式可以支持 RGBA 模式，也可以支持颜色索引模式，但是不能同时支持这两种模式。

```
1. void glEnableClientState(GLenum array)
```

指定了需要启用的数组。array 参数可以使用下面这些符号常量：

GL_VERTEX_ARRAY、GL_COLOR_ARRAY、GL_SECONDARY_COLOR_ARRAY、GL_INDEX_ARRAY、GL_NORMAL_ARRAY、GL_FOG_COORDINATE_ARRAY、GL_TEXTURE_COORD_ ARRAY 和 GL_EDGE_FLAG_ARRAY。

注意：OpenGL 3.1 只支持顶点数组数据存储在缓冲区对象中（参见第 2.7 节了解详细内容）。

如果需要使用光照，可能需要为每个顶点定义一条法线向量（参见第 2.5 节）。在这种情况下使用顶点数组时，需要同时激活表面法线数组和顶点坐标数组：

```
1. glEnableClientState(GL_NORMAL_ARRAY);
2. glEnableClientState(GL_VERTEX_ARRAY);
```

现在，假设想在某个时刻关闭光照，并且只用一种颜色来绘制几何图元。需要调用 glDisable()函数关闭光照状态（参见第 5 章）。在取消了光照的激活状态之后，还需要停止更改表面法线状态的值，因为这种做法是完全浪费的。为此，可以调用：

```
1. glDisableClientState(GL_NORMAL_ARRAY);
2. void glDisableClientState(GLenum array);
```

指定了需要禁用的数组。它接受的参数与 glEnableClientState()函数相同。

读者可能会问，OpenGL 的设计者为什么要创建这些新（ 并且很长） 的函数名（ 例如 gl*ClientState()），为什么不能像前面一样调用 glEnable()和 glDisable()呢？其中一个原因

是 glEnable()和 glDisable()可以存储在显示列表中，但是顶点数组却不可以放在显示列表中，因为这些数据被保存在客户端。

如果启用了多重纹理功能，启用和禁用顶点数组只会影响活动纹理单元。关于多重纹理的更多信息，请参阅第 9.8 节。

2.6.2 步骤 2：指定数组的数据

可以通过一种简单的方法，用一条命令指定客户空间中的一个数组。共有 8 个不同的函数可以用来指定数组，每个函数用于指定一个不同类型的数组。另外，还有一个函数可以一次指定客户空间中的几个数组，它们均来源于一个混合数组。

```
1. void glVertexPointer(GLint size, GLenum type, GLsizei stride,
2. const GLvoid *pointer);
```

指定了需要访问的空间坐标数据。pointer 是数组包含的第一个顶点的第一个坐标的内存地址。type 指定了数组中每个坐标的数据类型（GL_SHORT、GL_INT、GL_FLOAT 或 GL_DOUBLE）。size 是每个顶点的坐标数量，它必须是 2、3 或 4。stride 是连续顶点之间的字节偏移量。如果 stride 是 0，数组中的顶点便是紧密相邻的。

为了访问其他几个数组，可以使用下面这些类似的函数：

```
1. void glColorPointer(GLint size, GLenum type, GLsizei stride,
2. const GLvoid *pointer);
3. void glSecondaryColorPointer(GLint size, GLenum type, GLsizei stride,

4. const GLvoid *pointer);
5. void glIndexPointer(GLenum type, GLsizei stride, const GLvoid *pointer
   );
6. void glNormalPointer(GLenum type, GLsizei stride,
7. const GLvoid *pointer);
8. void glFogCoordPointer(GLenum type, GLsizei stride,
9. const GLvoid *pointer);
10. void glTexCoordPointer(GLint size, GLenum type, GLsizei stride,
11. const GLvoid *pointer);
12. void glEdgeFlagPointer(GLsizei stride, const GLvoid *pointer);
```

注意：可编程着色器使用的其他顶点属性可以存储在顶点数组中。由于它们与着色器相关联，所以将在第 15 章中讨论。OpenGL 3.1 只支持通用顶点数组来存储顶点数据。

这些函数的主要区别在于 size 和 type 参数是唯一的还是必须予以指定。例如，表面法线总是具有 3 个成分，因此指定它的 size 参数是冗余的。边界标志总是一个布尔值，它的长度或类型根本无关紧要。size 和 type 参数的合法值见表 2-4。

表 2-4 顶点数组的大小（每个顶点的值）和数据类型

函数大小	大小	
glVertexPointer	2、3、4	GL_SHORT、GL_
glColorPointer	3、4	GL_BYTE、G GL_UNSIGNED_ GL_FLOAT、GL_D(
glSecondaryColorPointer	3	GL_BYTE、G GL_UNSIGNED_ GL_FLOAT、GL_D(
glIndexPointer	1	GL_UNSIGNED_ GL_DOUBLE
glNormalPointer	3	GL_BYTE、GL_S
glFogCoordPointer	1	GL_FLOAT、GL_
glTexCoordPointer	1、2、3、4	GL_SHORT、GL_
glEdgeFlagPointer	1	没有type参数（数

对于那些支持多重纹理的 OpenGL 实现，用 glTexCoordPointer()函数指定一个纹理坐标数组只影响当前的活动纹理单元。详细信息参见第 9.8 节。

示例程序 2-9 使用顶点数组表示 RGBA 颜色和顶点坐标。RGB 浮点值以及与它们对应的整型坐标值（x, y）被加载到 GL_COLOR_ARRAY 和 GL_VERTEX_ARRAY 数组中。

示例程序 2-9 启用和加载顶点数组：varray.c

```
1. static GLint vertices[] = {25, 25,
2. 100, 325,
3. 175, 25,
4. 175, 325,
5. 250, 25,
6. 325, 325};
7. static GLfloat colors[] = {1.0, 0.2, 0.2,
```

```
8.  0.2, 0.2, 1.0,
9.  0.8, 1.0, 0.2,
10. 0.75, 0.75, 0.75,
11. 0.35, 0.35, 0.35,
12. 0.5, 0.5, 0.5};
13. glEnableClientState(GL_COLOR_ARRAY);
14. glEnableClientState(GL_VERTEX_ARRAY);
15. glColorPointer(3, GL_FLOAT, 0, colors);
16. glVertexPointer(2, GL_INT, 0, vertices);
```

跨距（Stride）

gl*Pointer()函数的 stride 参数告诉 OpenGL 如何访问指针数组中的数据。它的值应该是两个连续的指针元素之间的字节数量（或者是 0，这是一种特殊情况）。例如，假设顶点的 RGB 值和（x, y, z）坐标存储在同一个数组中，如下所示：

```
1.  static GLfloat intertwined[] =
2.  {1.0, 0.2, 1.0, 100.0, 100.0, 0.0,
3.  1.0, 0.2, 0.2, 0.0, 200.0, 0.0,
4.  1.0, 1.0, 0.2, 100.0, 300.0, 0.0,
5.  0.2, 1.0, 0.2, 200.0, 300.0, 0.0,
6.  0.2, 1.0, 1.0, 300.0, 200.0, 0.0,
7.  0.2, 0.2, 1.0, 200.0, 100.0, 0.0};
```

如果只想引用这个 intertwined 数组中的颜色值，下面这个调用从数组的起始地址（可以用&intertwined[0]表示）开始，然后跳跃 6*sizeof(GLfloat)个字节（也就是颜色和顶点坐标值的字节总数）。这次跳跃确保到达了下一个顶点数据的开始位置：

```
1.  glColorPointer(3, GL_FLOAT, 6*sizeof(GLfloat), &intertwined[0]);
```

对于顶点坐标指针，需要从这个数组的其他位置开始引用。我们从 intertwined 的第 4 个元素开始（记住，C 的数组是从 0 开始计数的）：

```
1.  glVertexPointer(3, GL_FLOAT, 6*sizeof(GLfloat), &intertwined[3]);
```

如果读者使用的数据存储方式类似上面的 intertwined 数组，可以参阅第 2.6.6 节，了解如何更方便地访问这类数据。

如果 stride 参数的值为 0，每种类型的顶点数组（RGB 颜色、颜色索引、顶点坐标等）必须紧密相邻。数组中的数据必须是一致的。也就是说，数据必须全部是 RGB 颜色值，或者全部是顶点坐标，也可以是其他类似的数据。

2.6.3 步骤 3：解引用和渲染（1）

在顶点数组的内容被解引用(即提取指针所指向的数据)之前,数组一直保存在客户端,它们的内容很容易进行修改。在步骤 3 中,数组中的数据被提取,接着发送到服务器,然后发送到图形处理管线进行渲染。

可以从单个数组元素(索引位置)提取数据,也可以从一个有序的数组元素列表(可能被限制为整个顶点数组数据的一个子集)中提取数据,或者从一个数组元素序列中提取数据。

解引用单个数组元素

```
1. void glArrayElement(GLint ith)
```

获取当前所有已启用数组的一个顶点(第 ith 个)的数据。对于顶点坐标数组,对应的函数是 glVertex[size][type]v(),其中 size 是[2, 3, 4]之一。type 是[s, i, f, d]之一,分别表示 GLshort、GLint、GLfloat 和 GLdouble。size 和 type 都是由 glVertexPointer()函数定义的。对于其他启用的数组,glArrayElement()分别调用 glEdgeFlagv()、glTexCoord[size][type]v()、glColor[size][type]v()、glSecondaryColor3[type]v()、glInde[type]v()、glNormal3[type]v()和 glFogCoord[type]v()。如果启用了顶点坐标数组,在其他几个数组(如果启用)相对应的函数(与数组值相对应,最多可达 7 个)被执行之后,glVertex*v()函数在最后执行。

glArrayElement()通常是在 glBegin()和 glEnd()之间调用。否则,glArrayElement()函数就会设置所有启用的数组的当前状态(顶点除外,因为它不存在当前状态)。在示例程序 2-10 中,使用取自启用的顶点数组的第 3、第 4 和第 6 个顶点绘制了一个三角形(同样,我们需要记住 C 的数组是从 0 开始计数的)。

示例程序 2-10 使用 glArrayElement()定义颜色和顶点

```
1. glEnableClientState(GL_COLOR_ARRAY);
2. glEnableClientState(GL_VERTEX_ARRAY);
3. glColorPointer(3, GL_FLOAT, 0, colors);
4. glVertexPointer(2, GL_INT, 0, vertices);
5. glBegin(GL_TRIANGLES);
6. glArrayElement(2);
7. glArrayElement(3);
8. glArrayElement(5);
9. glEnd();
```

当这段代码执行时,最后 5 行代码和下面的代码段具有相同的效果:

```
1. glBegin(GL_TRIANGLES);
2. glColor3fv(colors + (2 * 3));
3. glVertex2iv(vertices + (2 * 2));
4. glColor3fv(colors + (3 * 3));
```

```
5. glVertex2iv(vertices + (3 * 2));
6. glColor3fv(colors + (5 * 3));
7. glVertex2iv(vertices + (5 * 2));
8. glEnd();
```

由于 glArrayElement()对于每个顶点只调用 1 次，因此它可能会减少函数调用的数量，从而提高程序的总体性能。

注意，如果数组的内容在 glBegin()和 glEnd()之间进行了修改，就无法保证所获得的是最初的数据还是经过修改的数据。为了安全起见，在图元绘制完成之前，不要修改任何可能被访问的数组元素的内容。

解引用数组元素的一个列表

glArrayElement()对于随机存取数据的数组是非常有效的。与它类似的函数，例如 glDrawElements()、glMultiDrawElements()和 glDrawRangeElements()则采用一种更为有序的方式对数据数组进行随机存取。

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

使用 count 个元素定义一个几何图元序列，这些元素的索引值保存在 indices 数组中。type 必须是 GL_UNSIGNED_BYTE、GL_UNSIGNED_SHORT 或 GL_UNSIGNED_INT，表示 indices 数组的数据类型。mode 参数指定了被创建的是哪种类型的图元，它的值和 glBegin()函数所接受的参数值相同，例如 GL_POLYGON、GL_LINE_LOOP、GL_LINES 和 GL_POINTS 等。

glDrawElements()的效果差不多相当于下面这段代码：

```
1. glBegin(mode);
2. for (i = 0; i < count; i++)
3. glArrayElement(indices[i]);
4. glEnd();
```

glDrawElements()还会执行检查，确保 mode、count 和 type 参数的值都是合法的。另外，和前一个代码序列不同，执行 glDrawElements()会使几个状态处于不确定。在执行 glDrawElements()之后，如果相应的数组被启用，当前的 RGB 颜色、辅助颜色、颜色索引、法线坐标、雾坐标、纹理坐标和边界标志将处于不确定状态。

使用 glDrawElements()，立方体每个侧面的顶点可以放在一个索引数组中。示例程序 2-11 显示了用 glDrawElements()对这个立方体进行渲染的两种方法。图 2-15 显示了示例程序 2-11 使用的顶点的编号。

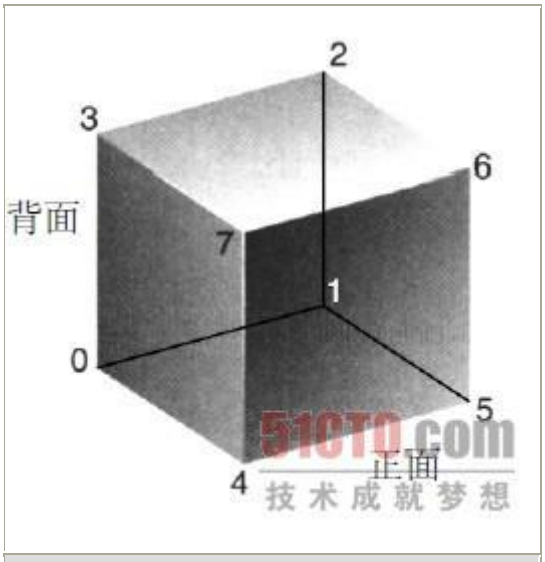


图 2-15 对顶点进行了编号的立方体

示例程序 2-11 使用 glDrawElements()对几个数组元素进行解引用

```
1. static GLubyte frontIndices[] = {4, 5, 6, 7};
2. static GLubyte rightIndices[] = {1, 2, 6, 5};
3. static GLubyte bottomIndices[] = {0, 1, 5, 4};
4. static GLubyte backIndices[] = {0, 3, 2, 1};
5. static GLubyte leftIndices[] = {0, 4, 7, 3};
6. static GLubyte topIndices[] = {2, 3, 7, 6};
7. glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, frontIndices);
8. glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, rightIndices);
9. glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, bottomIndices);
10. glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, backIndices);
11. glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, leftIndices);
12. glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, topIndices);
```

2.6.3 步骤 3：解引用和渲染（2）

注意：把 glDrawElements()放在 glBegin()和 glEnd()之间是错误的做法。

对于其中几种图元类型（例如 GL_QUADS、GL_TRIANGLES 和 GL_LINES），可以把几个索引列表压缩在一起，放在同一个数组中。由于 GL_QUADS 图元把每 4 个顶点解释为一个多边形，因此可以把示例程序 2-11 使用的所有索引值压缩到一个数组中，如示例程序 2-12 所示。

示例程序 2-12 把几个 glDrawElements()调用压缩为一个

```
1. static GLubyte allIndices[] = {4, 5, 6, 7, 1, 2, 6, 5,
2. 0, 1, 5, 4, 0, 3, 2, 1,
3. 0, 4, 7, 3, 2, 3, 7, 6};
```

```
4. glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, allIndices);
```

至于其他图元类型，把几个数组的索引值压缩在一个数组中可能会导致不同的渲染结果。在示例程序 2-13 中，以 GL_LINE_STRIP 为参数两次调用 glDrawElements()将会渲染两条直线串。如果简单地组合这两个数组并只调用 glDrawElements()函数 1 次，这样做的结果将只有一条直线带，其中顶点#6 和顶点#7 会连接在一起（注意，顶点#1 在两条直线串中均被使用，这只是为了说明这样做是合法的）。

示例程序 2-13 渲染两条直线串的两个 glDrawElements()调用

```
1. static GLubyte oneIndices[] = {0, 1, 2, 3, 4, 5, 6};
2. static GLubyte twoIndices[] = {7, 1, 8, 9, 10, 11};
3. glDrawElements(GL_LINE_STRIP, 7, GL_UNSIGNED_BYTE, oneIndices);
4. glDrawElements(GL_LINE_STRIP, 6, GL_UNSIGNED_BYTE, twoIndices);
```

glMultiDrawElements()函数是在 OpenGL 1.4 版本中引入的，它的作用是把几个 glDraw-Elements()调用合并到 1 个函数调用中。

```
1. void glMultiDrawElements(GLenum mode, GLsizei *count,
2. GLenum type, const GLvoid **indices,
3. GLsizei primcount);
```

调用一系列的 glDrawElements()函数（数量为 primcount 个）。indices 是一个指针数组，包含了数组元素的列表。count 是一个数组，包含了每个相应的数组元素列表中能够找到的顶点数量。mode（图元类型）和 type（数据类型）与它们在 glDrawElements()函数中的含义相同。

调用 glMultiDrawElements()函数的效果相当于：

```
1. for (i = 0; i < primcount; i++) {
2. if (count[i] > 0)
3. glDrawElements(mode, count[i], type, indices[i]);
4. }
```

示例程序 2-13 中的 2 个 glDrawElements()调用可以合并为 1 个 glMultiDrawElements()调用，如示例程序 2-14 所示：

示例程序 2-14 glMultiDrawElements()函数的用法：mvarray.c

```
1. static GLubyte oneIndices[] = {0, 1, 2, 3, 4, 5, 6};
2. static GLubyte twoIndices[] = {7, 1, 8, 9, 10, 11};
3. static GLsizei count[] = {7, 6};
4. static GLvoid * indices[2] = {oneIndices, twoIndices};
```

```
5. glMultiDrawElements(GL_LINE_STRIP, count, GL_UNSIGNED_BYTE,
6. indices, 2);
```

与 glDrawElements()和 glMultiDrawElements()相似，glDrawRangeElements()函数也适用于随机存取的数据数组，用于对它们的内容进行渲染。glDrawRangeElements()还对它所接受的合法索引值引入了范围限制，这可以提高程序的性能。为了实现优化的性能，有些 OpenGL 实现能够预先提取（在渲染之前获取）有限数量的顶点数组数据。glDrawRangeElements()允许指定预先提取的顶点的范围。

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

创建了一个几何图元序列，类似于 glDrawElements()创建的序列，但是具有更强的限制。glDrawRangeElements()的有些参数与 glDrawElements()相同，包括 mode（图元的类型）、count（元素的数量）、type（数据类型）和 indices（顶点数据的数组位置）。glDrawRangeElements()引入了两个新参数：start 和 end，它们指定了 indices 可以接受的值的范围。indices 数组中的值必须位于 start 和 end 之间才是合法的（包含 start 和 end）。

引用 indices 数组中位于范围[start, end]之外的顶点是错误的做法。但是，OpenGL 实现并不一定会发现或报告这个错误。因此，非法的索引值可能会产生一个 OpenGL 错误，也可能不产生错误，这完全取决于具体的 OpenGL 实现。

为了获取推荐的可以预先提取的最大顶点数量以及可以被引用的索引值的最大数量(表示可以进行渲染的顶点数量），可以分别以 GL_MAX_ELEMENTS_VERTICES 和 GL_MAX_ELEMENTS_INDICES 为参数调用 glGetIntegerv()函数。如果 end-start + 1 大于可以预先提取的最大顶点数量，或者 count 大于推荐的最大索引值数，glDrawRangeElements()函数仍然能够进行正确的渲染，但性能会受到影响。

并不是位于范围[start, end]之间的所有顶点都必须被引用。但是，在有些 OpenGL 实现中，如果指定范围内的顶点只有极少部分被引用，系统可能会不必要地处理许多未使用的顶点。

在调用 glArrayElement()、glDrawElements()、glMultiDrawElements()和 glDrawRangeElements()时，OpenGL 实现很可能会对最近处理（也就是变换、光照等）的顶点进行缓存处理，允许应用程序对它们进行“复用”，而不必另外再把它们经过变换管线进行传输。以前面提到的立方体为例，它具有 6 个面（ 多边形），却只有 8 个顶点。每个顶点正好由 3 个面使用。如果不使用 gl*Elements()，渲染所有 6 个面将要求处理 24 个顶点，虽然其中的 16 个顶点是冗余的。OpenGL 实现可能会最大限度地降低冗余度，

最少可能只处理 8 个顶点（顶点的复用可能限于一个 glDrawElements()或 glDrawRange-Elements()调用内部的所有顶点，或者是一个 glMultiDrawElements()调用内部的一个索引数组。对于 glArrayElements()，则限制在一对 glBegin()和 glEnd()之间）。

解引用一个数组元素序列

glArrayElements()、glDrawElements()和 glDrawRangeElements()能够对数据数组进行随机存取，但是 glDrawArrays()只能按顺序访问它们。

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

创建一个几何图元序列，使用每个被启用的数组中从 first 开始，到 first + count-1 结束的数组元素。mode 指定了创建的图元类型，它的值和 glBegin()函数所接受的参数值相同。例如：GL_POLYGON、GL_LINE_LOOP、GL_LINES 和 GL_POINTS 等。

调用 glDrawArrays()函数的效果差不多相当于下面这段代码：

```
1. glBegin (mode);
2. for (i = 0; i < count; i++)
3. glArrayElement(first + i);
4. glEnd();
```

和 glDrawElements()相似，glDrawArrays()也会对它的参数值执行错误检查，如果对应的数组被启用，它会导致当前的 RGB 颜色、辅助颜色、颜色索引、法线坐标、雾坐标、纹理坐标和边界标志处于不确定状态。

尝试一下

修改示例程序 2-19 绘制二十面体的绘图代码，使用顶点数组。

与 glMultiDrawElements()类似，glMultiDrawArrays()函数也是在 OpenGL 1.4 版本中引入的，它的用途是把几个 glDrawArrays()调用组合到一个调用中。

```
1. void glMultiDrawArrays(GLenum mode, GLint *first, GLsizei *count
2. GLsizei primcount);
```

调用一系列的 glDrawArrays()函数（共 primcount 个）。mode 指定了被创建的图元的类型，它的值和 glBegin()函数所接受的参数值相同。first 和 count 包含了数组位置的列表，表示从什么地方开始处理每个数组元素列表。因此，对于数组元素的第 i 个列表，OpenGL 将创建一个从 first[i]开始，到 first[i]+count[i]-1 结束的几何图元。

调用 glMultiDrawArrays()函数的效果相当于下面这段代码：

```
1. for (i = 0; i < primcount; i++) {
```

```
2. if (count[i] > 0)
3. glDrawArrays(mode, first[i], count[i]);
4. }
```

2.6.4 重启图元

当开始操作大量成组的顶点数据的时候，可能会发现需要很多次地调用 OpenGL 绘图函数，通常是渲染之前的绘图调用中所用过的相同类型的图元（例如，GL_TRIANGLE_STRIP）。当然，可以使用 glMultiDraw*()函数，但是，它们需要额外的开销来维护数组，以保存每个图元的起始索引和长度。OpenGL 3.1 通过指定一个专门由 OpenGL 处理的特定的值（图元重启索引），增加了在同一绘图调用中重启图元的能力。当在绘图调用中遇到图元重启索引的时候，从紧接着索引的顶点开始对相同类型的图元进行一次新的渲染。图元重启索引由 glPrimitiveRestartIndex()函数指定。

```
1. void glPrimitiveRestartIndex(GLuint index);
```

指定一个顶点数组元素索引，用来表示一个新的图元在渲染时的开始位置。当顶点数组元素索引的处理中遇到和 index 匹配的一个值的时候，就没有顶点数据需要处理了，当前的图形图元就终止了，相同类型的一个新图元开始。

通过调用 glEnable()或 glDisable()来控制图元重启并指定 GL_PRIMITIVE_RESTART，如示例程序 2-15 所示。

示例程序 2-15 使用 glPrimitiveRestartIndex()来渲染多个三角形串：primrestart.c

```
1. #define BUFFER_OFFSET(offset) ((GLvoid *) NULL + offset)
2. #define XStart -0.8
3. #define XEnd 0.8
4. #define YStart -0.8
5. #define YEnd 0.8
6. #define NumXPoints 11
7. #define NumYPoints 11
8. #define NumPoints (NumXPoints * NumYPoints)
9. #define NumPointsPerStrip (2*NumXPoints)
10.#define NumStrips (NumYPoints-1)
11.#define RestartIndex 0xffff
12.void
13.init()
14.{
15.GLuint vbo, ebo;
16.GLfloat *vertices;
```

```
17. GLushort *indices;
18. /* Set up vertex data */
19. glGenBuffers(1, &vbo);
20. glBindBuffer(GL_ARRAY_BUFFER, vbo);
21. glBufferData(GL_ARRAY_BUFFER, 2*NumPoints*sizeof(GLfloat),
22. NULL, GL_STATIC_DRAW);
23. vertices = glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);
24. if (vertices == NULL) {
25. fprintf(stderr, "Unable to map vertex buffer\n");
26. exit(EXIT_FAILURE);
27. }
28. else {
29. int i, j;
30. GLfloat dx = (XEnd - XStart) / (NumXPoints - 1);
31. GLfloat dy = (YEnd - YStart) / (NumYPoints - 1);
32. GLfloat *tmp = vertices;
33. int n = 0;
34. for (j = 0; j < NumYPoints; ++j) {
35. GLfloat y = YStart + j*dy;
36. for (i = 0; i < NumXPoints; ++i) {
37. GLfloat x = XStart + i*dx;
38. *tmp++ = x;
39. *tmp++ = y;
40. }
41. }
42. glUnmapBuffer(GL_ARRAY_BUFFER);
43. glVertexPointer(2, GL_FLOAT, 0, BUFFER_OFFSET(0));
44. glEnableClientState(GL_VERTEX_ARRAY);
45. }
46. /* Set up index data */
47. glGenBuffers(1, &ebo);
48. glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
49. /* We allocate an extra restart index because it simplifie
50. ** the element-array loop logic */
51. glBufferData( GL_ELEMENT_ARRAY_BUFFER,
52. NumStrips*(NumPointsPerStrip+1)*sizeof(GLushort),
53. NULL, GL_STATIC_DRAW );
54. indices = glMapBuffer(GL_ELEMENT_ARRAY_BUFFER,
```

```
55. GL_WRITE_ONLY);
56. if (indices == NULL) {
57. fprintf(stderr, "Unable to map index buffer\n");
58. exit(EXIT_FAILURE);
59. }
60. else {
61. int i, j;
62. GLushort *index = indices;
63. for (j = 0; j < NumStrips; ++j) {
64. GLushort bottomRow = j*NumYPoints;
65. GLushort topRow = bottomRow + NumYPoints;
66. for (i = 0; i < NumXPoints; ++i) {
67. *index++ = topRow + i;
68. *index++ = bottomRow + i;
69. }
70. *index++ = RestartIndex;
71. }
72. glUnmapBuffer(GL_ELEMENT_ARRAY_BUFFER);
73. }
74. glPrimitiveRestartIndex(RestartIndex);
75. glEnable(GL_PRIMITIVE_RESTART);
76. }
77. void
78. display()
79. {
80. int i, start;
81. glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
82. glColor3f(1, 1, 1);
83. glDrawElements(GL_TRIANGLE_STRIP,
84. NumStrips*(NumPointsPerStrip + 1),
85. GL_UNSIGNED_SHORT, BUFFER_OFFSET(0));
86. glutSwapBuffers();
87. }
```

2.6.5 实例化绘制

高级话题

OpenGL 3.1（尤其是 GLSL 1.40）增加了对实例化绘制的支持，它提供了另一个额外的值 gl_InstanceID（叫做实例 ID，并且它只在顶点着色器中可用），对于指定的每一组图元，该 ID 相应递增。

glDrawArraysInstanced()的运行和 glMultiDrawArrays()类似，只不过对于 glDrawArrays()的每次调用，开始索引和顶点计数是相同的（分别由 first 和 count 指定）。

```
1. void glDrawArraysInstanced(GLenum mode, GLint first, GLsizei count
2. GLsizei primcount);
```

primcount 次有效地调用 glDrawArrays()，在每次调用前设置 GLSL 顶点着色器值 gl_InstanceID。mode 指定了图元类型。first 和 count 指定了传递给 glDrawArrays()的数组元素的范围。

glDrawArraysInstanced()和如下的连续调用具有相同的效果（只不过我们的应用程序不必手动更新 gl_InstanceID）：

```
1. for (i = 0; i < primcount; i++) {
2. gl_InstanceID = i;
3. glDrawArrays(mode, first, count);
4. }
5. gl_InstanceID = 0;
```

同样，glDrawElementsInstanced()执行同样的操作，但是允许随机访问顶点数组中的数据。

```
1. void glDrawElementsInstanced(GLenum mode, GLsizei count,
2. GLenum type, const void *indicies,
3. GLsizei primcount);
```

primcount 次有效地调用 glDrawElements()，在每次调用前设置 GLSL 顶点着色器值 gl_InstanceID。mode 指定了图元类型。type 指定了数组索引的数据类型，并且必须是如下之一：GL_UNSIGNED_BYTE、GL_UNSIGNED_SHORT 或 GL_UNSIGNED_INT。indicies 和 count 指定了传递给 glDrawElements()的数组元素的范围。

glDrawElementsInstanced()的实现如下所示：

```
1. for (i = 0; i < primcount; i++) {
2. gl_InstanceID = i;
3. glDrawElements(mode, count, type, indicies);
4. }
5. gl_InstanceID = 0;
```

2.6.6 混合数组

高级话题

在本章的前面（参见 2.6.2 节的“跨距”）， 我们提到了混合数组这种特殊情况。在那一节中，我们介绍了混合存储了 RGB 颜色和 3D 顶点坐标的 intertwined 数组是通过调用 glColorPointer()和 glVertexPointer()进行访问的。精心使用 stride 参数可以帮助我们正确地指定数组元素：

```
1. static GLfloat intertwined[] =
2. {1.0, 0.2, 1.0, 100.0, 100.0, 0.0,
3. 1.0, 0.2, 0.2, 0.0, 200.0, 0.0,
4. 1.0, 1.0, 0.2, 100.0, 300.0, 0.0,
5. 0.2, 1.0, 0.2, 200.0, 300.0, 0.0,
6. 0.2, 1.0, 1.0, 300.0, 200.0, 0.0,
7. 0.2, 0.2, 1.0, 200.0, 100.0, 0.0};
```

另外，glInterleavedArrays()函数也可以同时指定几个顶点数组。glInterleavedArrays()函数还能启用和禁用适当的数组（因此它组合了“步骤 1：启用数组”和“步骤 2：指定数组的数据”）。intertwined 正好满足 glInterleavedArrays()支持的 14 种数据混合配置之一。因此，为了把 intertwined 数组的内容指定到 RGB 颜色数组和顶点数组，并启用这两个数组，只需要调用：

```
1. glInterleavedArrays(GL_C3F_V3F, 0, intertwined);
```

这个 glInterleavedArrays()调用启用了 GL_COLOR_ARRAY 和 GL_VERTEX_ARRAY。它禁用了 GL_SECONDARY_COLOR_ARRAY、GL_INDEX_ARRAY、GL_NORMAL_ARRAY、GL_FOG_COORDINATE_ARRAY、GL_TEXTRUE_COORD_ARRAY 和 GL_EDGE_FLAG_ARRAY。

这个调用还具有与使用 glColorPointer()和 glVertexPointer()在每个数组中指定 6 个顶点值相同的效果。现在，可以准备进入步骤 3 ：调用 glArrayElement()、glDrawElements()、glDrawRangeElements()或 glDrawArrays()，对数组元素进行解引用。

注意，glInterleavedArrays()并不支持边界标志。

glInterleavedArrays()的机制比较复杂，需要参考示例程序 2-16 和表 2-5。在这个示例程序和这张表中，我们看到的 et、ec 和 en 都是布尔值，分别用于启用或禁用纹理坐标、颜色和法线数组。st、sc 和 sv 分别表示纹理坐标、颜色和顶点数组的大小（成分的数量）。tc 表示 RGBA 颜色的数据类型，它是唯一可以具有非浮点混合值的数组。pc、pn 和 pv 是经过

计算产生的跨距值，用于跳转到单独的颜色、法线和顶点值。s（如果用户没有指定）表示从一个数组元素跳转到下一个数组元素的跨距值。

表 2-5 glInterleavedArrays()所使用的符号常量

Format	e_t	e_c	e_n	s_t	s_c	s_v
GL_V2F	F	F	F			2
GL_V3F	F	F	F			3
GL_C4UB_V2F	F	T	F		4	2
GL_C4UB_V3F	F	T	F		4	3
GL_C3F_V3F	F	T	F		3	3
GL_N3F_V3F	F	F	T			3
GL_C4F_N3F_V3F	F	T	T		4	3
GL_T2F_V3F	T	F	F	2		3
GL_T4F_V4F	T	F	F	4		4
GL_T2F_C4UB_V3F	T	T	F	2	4	3
GL_T2F_C3F_V3F	T	T	F	2	3	3
GL_T2F_N3F_V3F	T	F	T	2		3
GL_T2F_C4F_N3F_V3F	T	T	T	2	4	3
GL_T4F_C4F_N3F_V4F	T	T	T	4	4	4

```
1. void glInterleavedArrays(GLenum format,
2. GLsizei stride, const GLvoid *pointer)
```

初始化全部 8 个数组，禁用 format 参数并没有指定的数组，并启用 format 参数所指定的数组。format 是 14 个符号常量之一，这些符号常量表示 14 种数据配置。表 2-5 显示了 format 参数可以使用的值。stride 指定了连续顶点之间的字节偏移量。如果 stride 为 0，那么数组中的顶点便认为是紧密相邻的。pointer 是数组第一个顶点的第一个坐标的内存地址。如果启用了多重纹理，

glInterleavedArrays()只影响当前的活动纹理单元。详见第 9.8 节。

调用 glInterleavedArrays()函数的效果相当于用表 2-5 所定义的许多值调用示例程序 2-16 中的函数序列。所有的指针运算都是以 sizeof(GLubyte)为单位的。

示例程序 2-16 glInterleavedArrays()（format、stride、pointer）的效果

```
int str;
/* set et, ec, en, st, sc, sv, tc, pc, pn, pv, and s
* as a function of Table 2-5 and the value of format
*/
str = stride;
if (str == 0)
sstr = s;
glDisableClientState(GL_EDGE_FLAG_ARRAY);
glDisableClientState(GL_INDEX_ARRAY);
glDisableClientState(GL_SECONDARY_COLOR_ARRAY);
glDisableClientState(GL_FOG_COORD_ARRAY);
if (et) {
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
glTexCoordPointer(st, GL_FLOAT, str, pointer);
}
else
glDisableClientState(GL_TEXTURE_COORD_ARRAY);
if (ec) {
glEnableClientState(GL_COLOR_ARRAY);
glColorPointer(sc, tc, str, pointer+pc);
}
else
glDisableClientState(GL_COLOR_ARRAY);
if (en) {
glEnableClientState(GL_NORMAL_ARRAY);
glNormalPointer(GL_FLOAT, str, pointer+pn);
}
else
glDisableClientState(GL_NORMAL_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(sv, GL_FLOAT, str, pointer+pv);
```

在表 2-5 中，T 和 F 表示 True 和 False。f 是 sizeof(GLfloat)。c 是 sizeof(GLubype)的 4 倍，并四舍五入到最邻近的 f 的倍数。

首先我们讨论几种较为简单的格式：GL_V2F、GL_V3F 和 GL_C3F_V3F。如果使用了任何带 C4UB 的格式，可能必须使用一种结构数据类型，并进行一些专门的类型转换和指针匹配，把 4 个无符号字节包装为一个 32 位的字。

在有些 OpenGL 实现中，使用混合数组可能会提高应用程序的性能。在混合数组中，数据的准确布局是已知的。我们知道自己的数据是紧密相邻的，可以成块地读取。如果未使用混合数组，就必须检查跨距和大小信息，检测数据是否紧密相邻。

注意：glInterleavedArrays()只启用和禁用顶点数组，并指定了顶点数组的数据值。它并不会渲染任何东西。我们仍然必须完成“步骤 3：解引用和渲染”，调用 glArrayElement()、glDrawElements()、glDrawRangeElements()或 glDrawArrays()对指针进行解引用，并渲染几何图形。

2.7 缓冲区对象

高级话题

在许多 OpenGL 操作中，我们都向 OpenGL 发送一大块数据，例如向它传递需要处理的顶点数组数据。传输这种数据可能非常简单，例如把数据从系统的内存中复制到图形卡。但是，由于 OpenGL 是按照客户机-服务器模式设计的，在 OpenGL 需要数据的任何时候，都必须把数据从客户机内存传输到服务器。如果数据并没有修改，或者客户机和服务器位于不同的计算机（分布式渲染），数据的传输可能会比较缓慢，或者是冗余的。

OpenGL 1.5 版本增加了缓冲区对象（buffer object），允许应用程序显式地指定把哪些数据存储在图形服务器中。

当前版本的 OpenGL 中使用了很多不同类型的缓冲区对象：

从 OpenGL 1.5 开始，数组中的顶点数据可以存储在服务器端缓冲区对象中。第 2.7.7 节有详细介绍。

在 OpenGL 2.1 中，加入了在缓冲区对象中存储像素数据（例如，纹理贴图或像素块）的支持。第 8.5 节有详细介绍。

OpenGL 3.1 增加了统一缓冲对象（uniform buffer object）以存储成块的、用于着色器的统一变量数据。

读者还会发现 OpenGL 中有很多其他的功能用到了术语“对象”，但是这些功能并不都适用于存储块数据。例如，（OpenGL 1.1 引入的）纹理对象只是封装了和纹理贴图相关联的各种状态设置（参见第 9.4 节）。同样，OpenGL 3.0 中增加的顶点数组对象，封装了和使用顶点数组相关的状态参数。这些类型的对象允许我们使用较少的函数调用就能够修改大量的状态设置。为了使性能最大化，只要习惯它们的操作，就应该尽可能地尝试使用它们。

注意：通过对象的名字来引用它，其名字是一个无符号的整型标识符。从 OpenGL 3.1 开始，所有的名字必须由 OpenGL 使用 glGen*()函数之一来生成，不再接受用户定义的名字。

2.7.1 创建缓冲区对象

任何非零的无符号整数都可以作为缓冲区对象的标识符使用。可以任意选择一个有代表性的值，也可以让 OpenGL 负责分配和管理这些标识符。这两种做法有什么区别呢？让 OpenGL 分配标识符可以保证避免重复使用已被使用的缓冲区对象标识符，从而消除无意修改数据的风险。

为了让 OpenGL 分配缓冲区对象标识符，可以调用 glGenBuffers()函数。

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

在 buffers 数组中返回 n 个当前未使用的名称，表示缓冲区对象。在 buffers 数组中返回的名称并不需要是连续的整数。

返回的名称被标记为已使用，以便分配给缓冲区对象。但是，当它们被绑定之后，它们只获得一个合法的状态。

零是一个被保留的缓冲区对象名称，从来不会被 glGenBuffers()作为缓冲区对象返回。

还可以调用 glIsBuffer()函数，判断一个标识符是否是一个当前被使用的缓冲区对象的标识符。

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

如果 buffer 是一个已经绑定的缓冲区对象的名称，而且还没有删除，这个函数返回 GL_TRUE。

如果 buffer 为 0 或者它不是一个缓冲区对象的名称，这个函数返回 GL_FALSE。

2.7.2 激活缓冲区对象

为了激活缓冲区对象，首先需要将它绑定。绑定缓冲区对象表示选择未来的操作（对数据进行初始化或者使用缓冲区对象进行渲染）将影响哪个缓冲区对象。也就是说，如果应用程序有多个缓冲区对象，就需要多次调用 glBindBuffer()函数：一次用于初始化缓冲区对象以及它的数据，以后的调用要么选择用于渲染的缓冲区对象，要么对缓冲区对象的数据进行更新。

为了禁用缓冲区对象，可以用 0 作为缓冲区对象的标识符来调用 glBindBuffer()函数。这将把 OpenGL 切换为默认的不使用缓冲区对象的模式。

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

指定了当前的活动缓冲区对象。target 必须设置为 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 或者 GL_UNIFORM_BUFFER。buffer 指定了将要绑定的缓冲区对象。

glBindBuffer()完成 3 个任务之一：①当 buffer 是一个首次使用的非零无符号整数时，它就创建一个新的缓冲区对象，并把 buffer 分配给这个缓冲区对象，作为它的名称。②当绑定到一个以前创建的缓冲区对象时，这个缓冲区对象便成为活动的缓冲区对象。③当绑定到一个值为零的 buffer 时，OpenGL 就会停止使用缓冲区对象。

2.7.3 用数据分配和初始化缓冲区对象

一旦绑定了一个缓冲区对象，就需要保留空间以存储数据，这是通过调用 glBufferData()函数实现的。

```
1. void glBufferData(GLenum target, GLsizeiptr size, const GLvoid *data,

2. GLenum usage);
```

分配 size 个存储单位（通常是字节）的 OpenGL 服务器内存，用于存储顶点数据或索引。以前所有与当前绑定对象相关联的数据都将删除。

target 可以是 GL_ARRAY_BUFFER（表示顶点数据）、GL_ELEMENT_ARRAY_BUFFER（表示索引数据）、GL_PIXEL_UNPACK_BUFEER（ 表示传递给 OpenGL 的像素数据） 或 GL_PIXEL_PACK_BUFFER（表示从 OpenGL 获取的像素数据）、GL_COPY_READ_BUFFER 和 GL_COPY_WRITE_BUFFER（表示在缓冲区之间复制数据）、GL_TEXTURE_BUFFER（表示作为纹理缓冲区存储的纹理数据）、GL_TRANSFORM_FEEDBACK_BUFFER（表示执行一个变换反馈着色器的结果），或者 GL_UNIFORM_BUFFER（表示统一变量值）。

size 是存储相关数据所需要的内存数量。这个值通常是数据元素的个数乘以它们各自的存储长度。data 可以是一个指向客户机内存的指针（用于初始化缓冲区对象），也可以是 NULL。如果它传递的是一个有效的指针，size 个单位的存储空间就从客户机复制到服务器。如果它传递的是 NULL，这个函数将会保留 size 个单位的存储空间供以后使用，但不会对它进行初始化。

usage 提供了一个提示， 就是数据在分配之后将如何进行读取和写入。它的有效值包括 GL_STREAM_DRAW、GL_STREAM_READ、GL_STREAM_COPY、GL_STATIC_DRAW、GL_STATIC_READ、GL_STATIC_COPY、GL_DYNAMIC_DRAW、GL_DYNAMIC_READ、GL_DYNAMIC_COPY。

如果请求分配的内存数量超过了服务器能够分配的内存， glBufferData() 将返回 GL_OUT_OF_MEMORY。如果 usage 并不是允许使用的值之一，这个函数就返回

GL_INVALID_VALUE。glBufferData()首先在 OpenGL 服务器中分配内存以存储数据。如果请求的内存太多，它会设置 GL_OUT_OF_MEMORY 错误。如果成功分配了存储空间，并且 data 参数的值不是 NULL，size 个存储单位（通常是字节）就从客户机的内存复制到这个缓冲区对象。但是，如果需要在创建了缓冲区对象之后的某个时刻动态地加载数据，可以把 data 参数设置为 NULL，为数据保留适当的存储空间，但不对它进行初始化。

glBufferData()的最后一个参数 usage 是向 OpenGL 提供的一个性能提示。根据 usage 参数指定的值，
OpenGL 可能会对数据进行优化，进一步提高性能。它也可以选择忽略这个提示。在缓冲区对象数据
上，可以进行 3 种类型的操作：

1) 绘图：客户机指定了用于渲染的数据。

2) 读取：从 OpenGL 缓冲区读取（例如帧缓冲区）数据值，并且在应用程序中用于各种与渲染并不直接相关的计算过程。

3) 复制：从 OpenGL 缓冲区读取数据值，作为用于渲染的数据。

另外，根据数据更新的频率，有几种不同的操作提示描述了数据的读取频率或在渲染中使用的频率：

流模式：缓冲区对象中的数据常常需要更新，但是在绘图或其他操作中使用这些数据的次数较少。

静态模式：缓冲区对象中的数据只指定 1 次，但是这些数据被使用的频率很高。

动态模式：缓冲区对象中的数据不仅常常需要进行更新，而且使用频率也非常高。

usage 参数可能使用的值见表 2-6。

表 2-6 glBufferData()的 usage 参数的值

参　数	
GL_STREAM_DRAW	数据只指定1次，并且最多只
GL_STREAM_READ	数据从一个OpenGL缓冲区复
	值使用
GL_STREAM_COPY	数据从一个OpenGL缓冲区复
	函数的源数据
GL_STATIC_DRAW	数据只指定1次，但是可以多
GL_STATIC_READ	数据从一个OpenGL缓冲区复
GL_STATIC_COPY	数据从一个OpenGL缓冲区复
	的源数据
GL_DYNAMIC_DRAW	数据可以多次指定，并且可
GL_DYNAMIC_READ	数据可以多次从一个OpenGI
	数据值使用
GL_DYNAMIC_COPY	数据可以多次从一个OpenGI
	指定函数的源数据

（点击查看大图）

2.7.4 更新缓冲区对象的数据值

有两种方法可以更新存储在缓冲区对象中的数据。第一种方法假设我们已经在应用程序的一个缓冲区中准备了相同类型的数据。glBufferSubData()将用我们提供的数据替换被绑定的缓冲区对象的一些数据子集。

```
1. void glBufferSubData(GLenum target, GLintptr offset, GLsizeiptr size,

2. const GLvoid *data);
```

用 data 指向的数据更新与 target 相关联的当前绑定缓冲区对象中从 offset（以字节为单位）开始的 size 个字节数据。target 必须是 GL_ARRAY_BUFFER、GL_ELEMENT_ARRAY_BUFFER、GL_PIXEL_UNPACK_BUFFER、GL_PIXEL_PACK_BUFFER、GL_COPY_READ_BUFFER、GL_COPY_WRITE_BUFFER、GL_TRANSFORM_FEEDBACK_BUFFER 或 GL_UNIFORM_BUFFER。

如果 size 小于 0 或者 size+offset 大于缓冲区对象创建时所指定的大小，glBufferSubData()将产生一个 GL_INVALID_VALUE 错误。

第二种方法允许我们更灵活地选择需要更新的数据。glMapBuffer()返回一个指向缓冲区对象的指针，可以在这个缓冲区对象中写入新值（或简单地读取数据，这取决于内存访问权限），就像对数组进行赋值一样。在完成了对缓冲区对象的数据更新之后，可以调用glUnmapBuffer()，表示已经完成了对数据的更新。

glMapBuffer()提供了对缓冲区对象中包含的整个数据集合的访问。如果需要修改缓冲区中的大多数数据，这种方法很有用，但是，如果有一个很大的缓冲区并且只需要更新很小的一部分值，这种方法效率很低。

```
1. GLvoid *glMapBuffer(GLenum target, GLenum access);
```

返回一个指针，指向与 target 相关联的当前绑定缓冲区对象的数据存储。target 可以是 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 或 GL_UNIFORM_BUFFER。access 必须是 GL_READ_ONLY、GL_WRITE_ONLY 或 GL_READ_WRITE 之一，表示客户可以对数据进行的操作。

如果这个缓冲区无法被映射（把 OpenGL 错误状态设置为 GL_OUT_OF_MEMORY）或者它以前已经被映射（把 OpenGL 错误状态设置为 GL_INVALID_OPERATION），glMapBuffer()将返回 NULL。

在完成了对数据存储的访问之后，可以调用 glUnmapBuffer()取消对这个缓冲区的映射。

```
1. GLboolean glUnmapBuffer(GLenum target);
```

表示对当前绑定缓冲区对象的更新已经完成， 并且这个缓冲区可以释放。target 必须是 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 或 GL_UNIFORM_BUFFER。

下面是一个简单的例子，说明了如何选择性地更新数据元素。我们将使用 glMapBuffer()获取一个指向缓冲区对象中的数据（这些数据包含了三维的位置坐标）的指针，然后，只更新 z 坐标。

```
1. GLfloat* data;
2. data = (GLfloat*) glMapBuffer(GL_ARRAY_BUFFER, GL_READ_WRITE);
3. if (data != (GLfloat*) NULL) {
4. for( i = 0; i < 8; ++i )
5. data[3*i+2] *= 2.0; /* Modify Z values */
```

```
6. glUnmapBuffer(GL_ARRAY_BUFFER);
7. } else {
8. /* Handle not being able to update data */
9. }
```

如果只需要更新缓冲区中相对较少的值（与值的总体数目相比），或者更新一个很大的缓冲区对象中的很小的连续范围的值，使用 glMapBufferRange()效率更高。它允许只修改所需的范围内的数据值。

```
1. GLvoid *glMapBufferRange(GLenum target, GLintptr offset,
2. GLsizeiptr length, GLbitfield access);
```

返回一个指针，指向与 target 相关联的当前绑定缓冲区对象的数据存储。target 可以是 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 或 GL_UNIFORM_BUFFER。offset 和 length 指定了映射的范围。access 是 GL_MAP_READ_BIT 和 GL_MAP_WRITE_BIT 的一个位掩码组合，表示客户可以对数据进行的操作；也可以是 GL_MAP_INVALIDATE_RANGE_BIT、GL_MAP_INVALIDATE_BUFFER_BIT、GL_MAP_FLUSH_EXPLICIT_BIT 或 GL_MAP_UNSYNCHRONIZED_BIT，它们针对 OpenGL 应该如何管理缓冲区中的数据给出提示。

如果发生错误，glMapBufferRange()将返回 NULL。如果 offset 或 length 为负值，或者 offset+length 比缓冲区的大小还要大，将会产生 GL_INVALID_VALUE。如果不能获取足够的内存来映射缓冲区，将会产生 GL_OUT_OF_MEMORY 错误。如果发生如下的任何一种情况， 将会产生 GL_INVALID_OPERATION： 缓冲区已经映射； access 没有 GL_MAP_READ_BIT 或 GL_MAP_WRITE_BIT 设置；access 拥有 GL_MAP_READ_BIT 设置， 并且 GL_MAP_INVALIDATE_RANGE_BIT、GL_MAP_INVALIDATE_BUFFER_BIT 或 GL_MAP_UNSYNCHRONIZED_BIT 中的任何一个也设置了；access 中的 GL_MAP_WRITE_BIT 和 GL_MAP_FLUSH_EXPLICIT_BIT 都设置了。

使用 glMapBufferRange()，可以通过在 access 中设置额外的位来指定可选的提示。这些标志描述了在映射之前 OpenGL 服务器需要如何保护缓冲区中原有的数据。这个提示用来帮助 OpenGL 实现确定需要保留哪些数据值，以及保持这些数据的任何内部拷贝正确和一致需要达到多长时间。

正如表 2-7 中所述，当使用 glMapBufferRange()映射一个缓冲区的时候，若在 access 标志中指定了 GL_MAP_FLUSH_EXPLICIT_BIT，应该通过调用 glFlushMappedBufferRange()向 OpenGL 表明映射缓冲区中的范围需要修改。

表 2-7 glMapBufferRange()的 access 参数值

参　数	含
GL_MAP_INVALIDATE_RANGE_BIT	说明映射范围内之前的值可 这个范围中的数据是未定义 OpenGL调用访问未定义的数 这样调用的结果是未定义的 这个标志不能和GL_READ_B
GL_MAP_INVALIDATE_BUFFER_BIT	说明整个缓冲区中之前的值 有值都是未定义的，除非明确 问未定义的数据，不会产生O 果是未定义的（但可能引发应 和GL_READ_BIT一起使用
GL_MAP_FLUSH_EXPLICIT_BIT	表示映射区域可能更新的不 被认为是完成了，应用程序应 发出信号。如果映射缓冲区的 些值是未定义的，直到刷新为 使用这个选项将要求任何修 务器，glUnmapBuffer()不会自
GL_MAP_UNSYNCHRONIZED_BIT	说明OpenGL不应该试图同步 通过调用glBufferData()更新数 中的数据来渲染），直到glMa 改映射区域的那些待执行操作 作的结果是未定义的

```
1. GLvoid glFlushMappedBufferRange(GLenum target, GLintptr offset,GLsizei
   ptr length);
```

表示一个缓冲区范围中的值已经修改，这可能引发 OpenGL 服务器更新缓冲区对象的缓存版本。target 必须是如下值之一：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 或 GL_UNIFORM_BUFFER。offset 和 length 指定了映射缓冲区区域的范围，它们相对于缓冲区映射范围的开始处。

如果 offset 或 length 为负值， 或者 offset+length 比映射区域的大小还要大，将会产生 GL_INVALID_VALUE。如果没有缓冲区绑定到 target（例如，在 glBindBuffer()调用中，0 指定为绑定到 target 的缓冲区），或者如果绑定到 target 的缓冲区没有映射，或者如果它映射了却没有设置 GL_MAP_FLUSH_EXPLICIT_BIT，将会产生一个 GL_INVALID_OPERATION 错误。

2.7.5 在缓冲区对象之间复制数据

有时候，我们可能需要把数据从一个缓冲区对象复制到另一个缓冲区对象。在 OpenGL 3.1 以前的版本中，这个过程分两步：

1) 把数据从缓冲区对象复制到应用程序的内存中。可以通过以下两种方法之一来做到：映射缓冲区并将其复制到本地内存缓冲区中，或者调用 glGetBufferSubData()从服务器复制数据。

2) 通过绑定到新的对象，然后使用 glBufferData()发送新的数据（或者如果只是替换一个子集，使用 glBufferSubData()），来更新另一个缓冲区对象中的数据。也可以映射缓冲区，然后把数据从一个本地内存缓冲区复制到映射的缓冲区。

在 OpenGL 3.1 中，glCopyBufferSubData()命令复制数据，而不需要迫使数据在应用程序的内存中做短暂停留。

```
1. void glCopyBufferSubData(GLenum readbuffer, GLenum writebuffer,
2. GLintptr readoffset, GLintptr writeoffset,
3. GLsizeiptr size);
```

把数据从与 readbuffer 相关联的缓冲区对象复制到绑定到 writebuffer 的缓冲区对象。readbuffer 和 writebuffer 必须是如下的值之一： GL_ARRAY_BUFFER, GL_COPY_READ_BUFFER、GL_COPY_WRITE_BUFFER、GL_ELEMENT_ARRAY_BUFFER, GL_PIXEL_PACK_BUFFER、GL_PIXEL_UNPACK_BUFFER、GL_TEXTURE_BUFFER、GL_TRANSFORM_ FEEDBACK_BUFFER 或 GL_UNIFORM_BUFFER。

readoffset 和 size 指定了复制到目标缓冲区对象中的数据的数量，会从 writeoffset 开始替换同样大小的数据。

下面的情形会导致 GL_INVALID_VALUE 错误：readoffset、writeoffset 或 size 为负值；readoffset +size 超过了绑定到 readbuffer 的缓冲区对象的范围；writeoffset + size 超过了绑定到 writebuffer 的缓冲区对象的范围；如果 readbuffer 和 writebuffer 绑定到同一个对象，并且 readoffset 和 size 所指定的区域与 writeoffset 和 size 所确定的区域有交叉。

如果 readbuffer 或 writebuffer 中的任意一个绑定为 0，或者任意一个缓冲区是当前映射的，将会产生 L_INVALID_OPERATION 错误。

2.7.6 清除缓冲区对象

完成了对缓冲区对象的操作之后，可以释放它的资源，并使它的标识符可以由其他缓冲区对象使用。为此，可以调用 glDeleteBuffers()。被删除的当前绑定缓冲区对象的所有绑定都将重置为零。

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

删除 n 个缓冲区对象，它们的名称就是 buffers 数组的元素。释放的缓冲区对象可以被复用（例如，通过调用 glGenBuffers()）。

如果一个缓冲区对象是在绑定时删除的，这个对象的所有绑定都重置为默认的缓冲区对象，就像以 0 作为指定的缓冲区对象参数调用了 glBindBuffer()一样。如果试图删除不存在的缓冲区对象或名称为 0 的缓冲区对象，这个操作将被忽略，并不会产生错误。

2.7.7 使用缓冲区对象存储顶点数组数据

要在缓冲区对象中存储顶点数组数据，需要给应用程序添加如下步骤：

1) 生成缓冲区对象标识符（这个步骤是可选的）。

2) 绑定一个缓冲区对象，确定它是用于存储顶点数据还是索引。

3) 请求数据的存储空间，并且对这些数据元素进行初始化（后一个步骤可选）。

4) 指定相对于缓冲区起始位置的偏移量，对诸如 glVertexPointer()这样的顶点数组函数进行初始化。

5) 绑定适当的缓冲区对象，用于渲染。

6) 使用适当的顶点数组渲染函数进行渲染，例如 glDrawArrays()或 glDrawElements()。

如果想初始化多个缓冲区对象，就需要为每个缓冲区对象重复步骤 2)～4)。

顶点数组数据的所有“格式”都适用于缓冲区对象。如第 2.6.2 节所述，顶点、颜色、光照法线或其他任何类型的相关联顶点数据都可以存储在缓冲区对象中。另外，第 2.6.6 节所描述的混合顶点数组数据也可以存储在缓冲区对象中。不论是哪种情况，我们都将创建一个缓冲区对象，保存所有作为顶点数组使用的数据。

就像在客户机的内存中指定内存地址一样（OpenGL 应该在客户机的内存中访问顶点数组数据），需要根据机器单位（通常是字节）指定缓冲区对象中数据的偏移量。为了帮助读者理解偏移量的计算，我们将使用下面这个宏来简化偏移量的表达形式：

```
#define BUFFER_OFFSET(bytes) ((GLubyte*) NULL + (bytes))
```

例如，如果每个顶点的颜色和位置数据是浮点类型，也许它们可以用下面这个数组来表示：

```
GLfloat vertexData[][6] = {
{ R0, G0, B0, X0, Y0, Z0 },
{ R1, G1, B1, X1, Y1, Z1 },
...
{ Rn, Gn, Bn, Xn, Yn, Zn }
};
```

这个数组用于初始化缓冲区对象，可以用两个独立的顶点数组调用来指定数据，其中一个表示颜色，另一个表示顶点：

```
glColorPointer(3, GL_FLOAT, 6*sizeof(GLfloat),BUFFER_OFFSET(0));
glVertexPointer(3, GL_FLOAT, 6*sizeof(GLfloat),
BUFFER_OFFSET(3*sizeof(GLfloat));
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
```

相反， 由于 vertexData 中的数据与一个混合顶点数组的格式相匹配， 因此可以使用 glInterleavedArrays()来指定顶点数组数据：

```
glInterleavedArrays(GL_C3F_V3F, 0, BUFFER_OFFSET(0));
```

示例程序 2-17 综合了所有这些内容，演示了如何使用包含顶点数据的缓冲区对象。这个例子创建了两个缓冲区对象，一个包含顶点数据，另一个包含索引数据。

示例程序 2-17 在缓冲区对象中使用顶点数据

```
#define VERTICES 0
#define INDICES 1
#define NUM_BUFFERS 2
GLuint buffers[NUM_BUFFERS];
GLfloat vertices[][3] = {
{ -1.0, -1.0, -1.0 },
{ 1.0, -1.0, -1.0 },
{ 1.0, 1.0, -1.0 },
{ -1.0, 1.0, -1.0 },
{ -1.0, -1.0, 1.0 },
{ 1.0, -1.0, 1.0 },
{ 1.0, 1.0, 1.0 },
```

```
{ -1.0, 1.0, 1.0 }
};
GLubyte indices[][4] = {
{ 0, 1, 2, 3 },
{ 4, 7, 6, 5 },
{ 0, 4, 5, 1 },
{ 3, 2, 6, 7 },
{ 0, 3, 7, 4 },
{ 1, 5, 6, 2 }
};
glGenBuffers(NUM_BUFFERS, buffers);
glBindBuffer(GL_ARRAY_BUFFER, buffers[VERTICES]);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
GL_STATIC_DRAW);
glVertexPointer(3, GL_FLOAT, 0, BUFFER_OFFSET(0));
glEnableClientState(GL_VERTEX_ARRAY);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[INDICES]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices
GL_STATIC_DRAW);
glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE,
BUFFER_OFFSET(0));
```

2.8 顶点数组对象

随着程序逐渐增大并且使用更多的模型，读者可能发现要在每个帧的多组顶点数组之间切换。根据你为每个顶点使用多少个顶点属性，像对 glVertexPointer()这样的函数的调用次数可能变得很大。顶点数组对象捆绑了调用的集合，以设置顶点数组的状态。在初始化之后，可以通过单次调用在不同的顶点数组集合之间快速修改。

要创建一个顶点数组对象，首先调用 glGenVertexArrays()，这将会创建所要求数目的未初始化的对象：

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

返回 n 个当前未使用的名字，用作数组 arrays 中的顶点数组对象。返回的名字标记为是用来分配额外的缓冲区对象，并且用表示未初始化的顶点数组集合的默认状态值来进行初始化。

创建了自己的顶点数组对象之后，需要初始化新的对象，并且把要使用的顶点数组数据的集合与单个已分配的对象关联起来。使用 glBindVertexArray()函数来做到这一点。一旦初

始化了所有的顶点数组对象，就可以使用 glBindVertexArray()在建立的不同顶点数组集合之间切换。

```
1. GLvoid gBindVertexArray(GLuint array);
```

glBindVertexArray()做 3 件事情。当使用的值 array 不是零并且是从 glGenVertexArrays()返回的值时，创建一个新的顶点数组对象并且分配该名字。当绑定到之前创建的一个顶点数组对象的时候，该顶点数组对象变成活动的，这还会影响到存储在该对象中的顶点数组状态。当绑定到一个为零的 array 值时，OpenGL 停止使用顶点数组对象并且返回顶点数组的默认状态。

如果 array 不是之前从 glGenVertexArrays()返回的值；如果它是 glDeleteVertexArrays()已经释放的值；如果调用了任何一个 gl*Pointer()函数来指定一个顶点数组，而在绑定一个非零顶点数组对象的时候，它没有和一个缓冲区对象关联起来（例如，使用一个客户端顶点数据存储），将会产生 GL_INVALID_OPERATION 错误。

示例程序 2-18 展示了使用顶点数组对象在两组顶点数组之间切换。

示例程序 2-18 使用顶点数组对象: vao.c

```
1. #define BUFFER_OFFSET(offset) ((GLvoid*) NULL + offset)
2. #define NumberOf(array) (sizeof(array)/sizeof(array[0]))
3. typedef struct {
4. GLfloat x, y, z;
5. } vec3;
6. typedef struct {
7. vec3 xlate; /* Translation */
8. GLfloat angle;
9. vec3 axis;
10.} XForm;
11.enum { Cube, Cone, NumVAOs };
12.GLuint VAO[NumVAOs];
13.GLenum PrimType[NumVAOs];
14.GLsizei NumElements[NumVAOs];
15.XForm Xform[NumVAOs] = {
16.{ { -2.0, 0.0, 0.0 }, 0.0, { 0.0, 1.0, 0.0 } },
17.{ { 0.0, 0.0, 2.0 }, 0.0, { 1.0, 0.0, 0.0 } }
18.};
19.GLfloat Angle = 0.0;
20.void
21.init()
```

```
22. {
23. enum { Vertices, Colors, Elements, NumVBOs };
24. GLuint buffers[NumVBOs];
25. glGenVertexArrays(NumVAOs, VAO);
26. {
27. GLfloat cubeVerts[][3] = {
28. { -1.0, -1.0, -1.0 },
29. { -1.0, -1.0, 1.0 },
30. { -1.0, 1.0, -1.0 },
31. { -1.0, 1.0, 1.0 },
32. { 1.0, -1.0, -1.0 },
33. { 1.0, -1.0, 1.0 },
34. { 1.0, 1.0, -1.0 },
35. { 1.0, 1.0, 1.0 },
36. };
37. GLfloat cubeColors[][3] = {
38. { 0.0, 0.0, 0.0 },
39. { 0.0, 0.0, 1.0 },
40. { 0.0, 1.0, 0.0 },
41. { 0.0, 1.0, 1.0 },
42. { 1.0, 0.0, 0.0 },
43. { 1.0, 0.0, 1.0 },
44. { 1.0, 1.0, 0.0 },
45. { 1.0, 1.0, 1.0 },
46. };
47. GLubyte cubeIndices[] = {
48. 0, 1, 3, 2,
49. 4, 6, 7, 5,
50. 2, 3, 7, 6,
51. 0, 4, 5, 1,
52. 0, 2, 6, 4,
53. 1, 5, 7, 3
54. };
55. glBindVertexArray(VAO[Cube]);
56. glGenBuffers(NumVBOs, buffers);
57. glBindBuffer(GL_ARRAY_BUFFER, buffers[Vertices]);
58. glBufferData(GL_ARRAY_BUFFER, sizeof(cubeVerts),
59. cubeVerts, GL_STATIC_DRAW);
```

```
60. glVertexPointer(3, GL_FLOAT, 0, BUFFER_OFFSET(0));
61. glEnableClientState(GL_VERTEX_ARRAY);
62. glBindBuffer(GL_ARRAY_BUFFER, buffers[Colors]);
63. glBufferData(GL_ARRAY_BUFFER, sizeof(cubeColors),
64. cubeColors, GL_STATIC_DRAW);
65. glColorPointer(3, GL_FLOAT, 0, BUFFER_OFFSET(0));
66. glEnableClientState(GL_COLOR_ARRAY);
67. glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,
68. buffers[Elements]);
69. glBufferData(GL_ELEMENT_ARRAY_BUFFER,
70. sizeof(cubeIndices), cubeIndices, GL_STATIC_DRAW);
71. PrimType[Cube] = GL_QUADS;
72. NumElements[Cube] = NumberOf(cubeIndices);
73. }
74. {
75. int i, idx;
76. float dTheta;
77. #define NumConePoints 36
78. /* We add one more vertex for the cone's apex */
79. GLfloat coneVerts[NumConePoints+1][3] = {
80. {0.0, 0.0, 1.0}
81. };
82. GLfloat coneColors[NumConePoints+1][3] = {
83. {1.0, 1.0, 1.0}
84. };
85. GLubyte coneIndices[NumConePoints+1];
86. dTheta = 2*M_PI / (NumConePoints - 1);
87. idx = 1;
88. for (i = 0; i < NumConePoints; ++i, ++idx) {
89. float theta = i*dTheta;
90. coneVerts[idx][0] = cos(theta);
91. coneVerts[idx][1] = sin(theta);
92. coneVerts[idx][2] = 0.0;
93. coneColors[idx][0] = cos(theta);
94. coneColors[idx][1] = sin(theta);
95. coneColors[idx][2] = 0.0;
96. coneIndices[idx] = idx;
97. }
```

```
98. glBindVertexArray(VAO[Cone]);
99. glGenBuffers(NumVBOs, buffers);
100. glBindBuffer(GL_ARRAY_BUFFER, buffers[Vertices]);
101. glBufferData(GL_ARRAY_BUFFER, sizeof(coneVerts),
102. coneVerts, GL_STATIC_DRAW);
103. glVertexPointer(3, GL_FLOAT, 0, BUFFER_OFFSET(0));
104. glEnableClientState(GL_VERTEX_ARRAY);
105. glBindBuffer(GL_ARRAY_BUFFER, buffers[Colors]);
106. glBufferData(GL_ARRAY_BUFFER, sizeof(coneColors),
107. coneColors, GL_STATIC_DRAW);
108. glColorPointer(3, GL_FLOAT, 0, BUFFER_OFFSET(0));
109. glEnableClientState(GL_COLOR_ARRAY);
110. glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,
111. buffers[Elements]);
112. glBufferData(GL_ELEMENT_ARRAY_BUFFER,
113. sizeof(coneIndices), coneIndices, GL_STATIC_DRAW);
114. PrimType[Cone] = GL_TRIANGLE_FAN;
115. NumElements[Cone] = NumberOf(coneIndices);
116. }
117. glEnable(GL_DEPTH_TEST);
118. }
119. void
120. display()
121. {
122. int i;
123. glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
124. glPushMatrix();
125. glRotatef(Angle, 0.0, 1.0, 0.0);
126. for (i = 0; i < NumVAOs; ++i) {
127. glPushMatrix();
128. glTranslatef(Xform[i].xlate.x, Xform[i].xlate.y,
129. Xform[i].xlate.z);
130. glRotatef(Xform[i].angle, Xform[i].axis.x,
131. Xform[i].axis.y, Xform[i].axis.z);
132. glBindVertexArray(VAO[i]);
133. glDrawElements(PrimType[i], NumElements[i],
134. GL_UNSIGNED_BYTE, BUFFER_OFFSET(0));
135. glPopMatrix();
```

```
136. }
137. glPopMatrix();
138. glutSwapBuffers();
139. }
```

要删除顶点数组对象并释放它们的名字以便重用，调用 glDeleteVertexArrays()。如果使用缓冲区对象存储数据，当引用它们的顶点数组对象删除的时候，这些缓冲区对象没有删除。它们继续存在（直到删除它们）。唯一的变化是，如果缓冲区对象当前是绑定的，当删除顶点数组对象时，它们变成了未绑定的。

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

删除 arrays 中指定的 n 个顶点数组对象，使这些名字随后可以作为顶点数组重用。如果删除了一个绑定的顶点数组，该顶点数组的绑定变成 0（就好像使用了 0 值调用了 glBindBuffer()），默认的顶点数组变成当前的顶点数组。arrays 中未使用的名字会释放掉，但是，当前顶点数组的状态没有改变。

最后，如果需要确定一个特定的值是否可以表示一个已分配的（但不必是已初始化的）顶点数组对象，可以调用 glIsVertexArray()来检查。

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

如果 array 是之前 glGenVertexArrays()产生的一个顶点数组对象的名字，但是随后没有删除，返回 GL_TRUE。如果 array 是 0 或者一个并非顶点数组对象的名字的非零值，返回 GL_FALSE。

2.9 属性组

在第 2.3 节中，我们看到了如何设置或查询一个单独的状态或状态变量。也可以用一个命令保存或恢复一组相关的状态变量的值。

OpenGL 将相关的状态变量进行归组，称为属性组（attribute group）。例如，GL_LINE_BIT 属性包括了 5 个状态变量：直线的宽度、GL_LINE_STIPPLE 启用状态、直线点画模式、直线点画重复计数器和 GL_LINE_SMOOTH 启用状态（参见第 6.2 节）。使用 glPushAttrib()和 glPopAttrib()函数，可以同时保存和恢复全部这 5 个状态变量。

有些状态变量可能存在于几个属性组中。例如，状态变量 GL_CULL_FACE 既属于多边形属性组，
又属于启用属性组。

在 OpenGL 1.1 版本中，有两个不同的属性堆栈。除了原来的属性堆栈外（用于保存服务器状态变量的值），还有一个客户属性堆栈，是通过 glPushClientAttrib()和 glPopClientAttrib()函数访问的。一般而言，使用这些函数，获取、保存和恢复状态值的速度会更快一点。有些

状态值可能是由硬件维护的，访问它们的开销可能较大。另外，如果是在远程客户机上进行操作，在获取、保存和恢复属性数据时，它们都要通过网络传输。但是，OpenGL 实现可以把属性堆栈保存在服务器上，从而避免了不必要的网络延迟。

OpenGL 大约有 20 个不同的属性组，它们都可以用 glPushAttrib()和 glPopAttrib()进行保存和恢复。客户属性组共有 2 个，它们可以用 glPushClientAttrib()和 glPopClientAttrib()进行保存和恢复。无论是在服务器还是客户机上，属性都存储在堆栈上。属性堆栈的深度至少可以保存 16 个属性组（可以调用 glGetIntegerv()函数，以 GL_MAX_ATTRIB_STACK_ DEPTH 和 GL_MAX_CLIENT_ ATTRIB_STACK_DEPTH 为参数，查询 OpenGL 实现所支持的堆栈深度）。如果试图压
入到一个已满的堆栈或者试图从一个空堆栈弹出元素，将会产生错误。（可以参阅附录 B，了解各个属性是用什么掩码值进行保存的。也就是说，各个属性分别属于哪个特定的属性组。）

```
1. void glPushAttrib(GLbitfield mask);
2. void glPopAttrib(void);
```

glPushAttrib()保存由 mask 指定的所有属性，把它们压入到属性堆栈中。glPopAttrib()恢复上一次调用 glPushAttrib()时所保存的状态变量的值。表 2-8 列出的所有可能的掩码位，它们可以用逻辑 OR 操作组合在一起，表示任何属性组合。例如，GL_LIGHTING_BIT 表示所有与光照有关的状态变量，包括当前的材料颜色、环境光、散射光、镜面光和发射光、被启用的光源列表以及聚光灯的方向。当 glPopAttrib()函数被调用时，所有这些变量的值都被恢复。

特殊掩码 GL_ALL_ATTRIB_BITS 用于保存和恢复所有属性组中的所有状态变量。

表 2-8 属性组

掩　码	位属性组
GL_ACCUM_BUFFER_BIT	累积缓冲区
GL_ALL_ATTRIB_BITS	—
GL_COLOR_BUFFER_BIT	颜色缓冲区
GL_CURRENT_BIT	当前
GL_DEPTH_BUFFER_BIT	深度缓冲区
GL_ENABLE_BIT	启用
GL_EVAL_BIT	求值
GL_FOG_BIT	雾
GL_HINT_BIT	提示
GL_LIGHTING_BIT	光照
GL_LINE_BIT	直线
GL_LIST_BIT	列表
GL_MULTISAMPLE_BIT	多重采样
GL_PIXEL_MODE_BIT	像素
GL_POINT_BIT	点
GL_POLYGON_BIT	多边形
GL_POLYGON_STIPPLE_BIT	多边形点画
GL_SCISSOR_BIT	裁剪
GL_STENCIL_BUFFER_BIT	模板缓冲区
GL_TEXTURE_BIT	纹理
GL_TRANSFORM_BIT	转换
GL_VIEWPORT_BIT	视口


```
1. void glPushClientAttrib(GLbitfield mask);
2. void glPopClientAttrib(void);
```

glPushClientAttrib()保存由 mask 掩码指定的所有属性，把它们压到客户属性堆栈中。glPopClientAttrib()恢复上次调用 glPushClientAttrib()时保存的那些状态变量的值。表 2-9 列出了所有可能的掩码位，它们可以用逻辑 OR 操作组合在一起，表示任何客户属性组合。有两个属性组（反馈和选择）无法使用堆栈机制进行保存和恢复。

表 2-9 客户属性组

掩码位	属性组
GL_CLIENT_PIXEL_STORE_BIT	像素存储
GL_CLIENT_VERTEX_ARRAY_BIT	顶点数组
GL_ALL_CLIENT_ATTRIB_BITS	—
无法压入或弹出	反馈
无法压入或弹出	选择

2.10 创建多边形表面模型的一些提示（1）

本节提供了一些技巧。当读者使用多边形近似模拟的方式创建表面模型时，可以使用这些技巧。在读完了第 5 章和第 7 章之后，可能需要回顾本节的内容。光照条件会影响模型被绘制之后的外观。如果在创建表面模型时使用了显示列表，本节所介绍的技巧就更加有用。当阅读完这些技巧之后，需要记住一点，如果启用了光照计算，就必须指定法线向量，才能获得正确的结果。

用多边形近似模拟法创建表面模型是一项艺术，经验是无可替代的。但是，本节还是列出了一些要点，可以更好地帮助读者上手。

使多边形的方向（环绕）保持一致。从外侧观察表面时，确保组成这个表面的所有多边形都具有相同的方向（都为顺时针方向或都为逆时针方向）。一致的方向对于多边形剔除和双面光照是极为重要的。一开始就要掌握正确的做法，如果到了后期再来修正这个问题，将会痛苦万分（例如，如果使用 glScale*()沿一些对称轴反射几何图形，可以使用 glFrontFace()函数更改多边形的方向，使所有多边形的方向保持一致）。

对表面进行细分时，要密切注意那些非三角形的多边形。具有 3 个顶点的三角形能够保证位于同一个平面上，而具有 4 个或更多顶点的多边形却无法保证。可以通过某些方向观察到非平面多边形，例如多边形相互之间的边缘。OpenGL 可能无法正确地渲染非平面多边形。

在显示速度和图像质量之间总存在一种权衡关系。如果把一个表面细分为数量较少的多边形，它的渲染速度可能非常快，但是很可能具有锯齿状外观。如果把它细分为数以百万计的微小多边形，它的显示质量可能非常出色，但是可能需要相当长的渲染时间。理想的做法是向多边形细分函数提供一个参数，表示希望细分所达到的精度。如果物体距离较远，可以使用较为粗糙的细分。另外，在进行细分时，在表面相对较平的区域，可以使用较大的多边形。反之，在曲率很大的表面部分，应该使用很小的多边形。

为了实现高质量的图像，在轮廓边缘进行更精细的划分显然要比在表面内部进行精细划分的效果更好。如果表面需要根据观察点进行旋转，这样做的难度更大一些，因为它的轮廓边缘会一直处于运动状态。当法线向量垂直于从表面指向观察点的向量时（也就是说，它们

的向量积为零），就会出现轮廓边缘。如果这个向量积接近于零，可以选择进行更精细的划分。

避免在模型中出现 T 交叉，如图 2-16 所示，线段 AB 和 BC 不能保证与 AC 重合。有时候确实如此，但有时候情况却不是这样，具体取决于变换和方向。这可能导致表面出现间歇性的开裂现象。

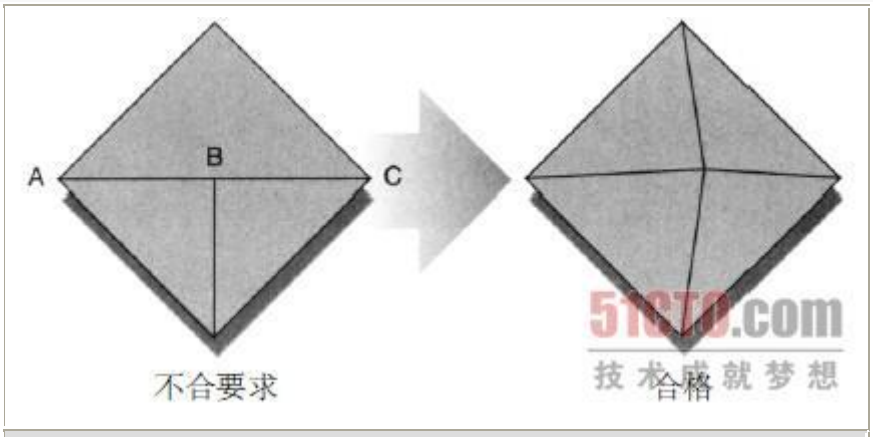


图 2-16 修改不合要求的 T 交叉

如果想创建一个闭合的表面，确保闭合环的起点和终点使用完全相同的坐标，不然可能因为数值的四舍五入而产生有缺口的环。下面是一个绘制二维圆形的例子，它的代码存在问题。

```
1. /* don't use this code */
2. #define PI 3.14159265
3. #define EDGES 30
4. /* draw a circle */
5. glBegin(GL_LINE_STRIP);
6. for (i = 0; i <= EDGES; i++)
7. glVertex2f(cos((2*PI*i)/EDGES), sin((2*PI*i)/EDGES));
8. glEnd();
```

如果读者的机器计算产生的 0 和（2*PI*EDGES/EDGES）的正弦值和余弦值完全相同，这个圆的首尾能够完全吻合。但是，如果读者坚信计算机的浮点运算单元总是能够执行正确的计算，它肯定会令人失望的。为了修正这段代码， 确信当 i == EDGES 时， 计算 0 的正弦和余弦值， 而不是 2*PI*EDGES/EDGES 的正弦和余弦值。或者使用更简单的做法，用 GL_LINE_LOOP 来代替 GL_LINE_STRIP，然后把循环的终止条件修改为 i < EDGES。

例子：创建一个二十面体

为了更好地说明在使用多边形近似模拟法构建表面时所需要考虑的因素，我们来看几个例子。这些代码涉及一个规则二十面体（它是一种柏拉图式的实心形状，由 20 个面组成，跨越 12 个顶点，每个面都是一个等边三角形）的诸多顶点。二十面体可以认为是球体的一

种粗略近似。示例程序 2-19 定义了组成一个二十面体的顶点和三角形，然后绘制这个二十面体。

示例程序 2-19 绘制一个二十面体

```
#define X .525731112119133606
#define Z .850650808352039932
static GLfloat vdata [12][3] ={
{-X,0.0,Z},{X,0.0,Z},{-X,0.0,-Z},{X,0.0,-Z},
{0.0,Z,X},{0.0,Z,-X},{0.0,-Z,X},{0.0,-Z,-X},
{Z,X,0.0},{-Z,X,0.0},{Z,-X,0.0},{-Z,-X,0.0}
};
static GLuint tindices [20][3] ={
{1,4,0},{4,9,0},{4,5,9},{8,5,4},{1,8,4},
{1,10,8},{10,3,8},{8,3,5},{3,2,5},{3,7,2},
{3,10,7},{10,6,7},{6,11,7},{6,0,11},{6,1,0},
{10,1,6},{11,0,9},{2,11,9},{5,2,9},{11,2,7}
};
int i;
glBegin(GL_TRIANGLES);
for (i =0;i <20;i++){
/*color information here */
glVertex3fv(&vdata [tindices [i][0]][0]);
glVertex3fv(&vdata [tindices [i][1]][0]);
glVertex3fv(&vdata [tindices [i][2]][0]);
}
glEnd();
```

我们为 X 和 Z 选择了两个似乎很奇怪的数，其用意在于使原点到这个二十面体的每个顶点的距离均为 1.0。数组 vdata[][]保存了 12 个顶点的坐标，其中第 0 个顶点的坐标是{-X, 0.0, Z}，第 1 个顶点的坐标是{X, 0.0, Z}，以此类推。tindices[][]数组告诉 OpenGL 如何连接顶点来构成三角形。例如，第 1 个三角形是由第 0、第 4 和第 1 个顶点构成的。如果按照上面代码给定的顺序用顶点构建三角形，可以确保所有的三角形具有相同的方向。

涉及颜色信息的那行注释应该由一条用于设置第 i 个面颜色的命令来代替。如果此处未出现代码，那么所有的面都用相同的颜色绘制。如果是这样，显然无法分辨这个物体的三维质量。另一种可供使用的方法是显式地指定颜色来定义表面法线，并使用光照，如下一小节所述。

注意：在本节描述的所有例子中，除非表面只绘制一次，否则很可能需要保存经过计算的顶点和法线坐标。这样，每次在绘制这个表面时，这些计算就不必重复进行。这个任务可以通过使用自己创建的数据结构来完成，也可以通过创建显示列表来完成（参见第 7 章）。

2.10 创建多边形表面模型的一些提示（2）

计算表面的法线向量

如果一个表面需要光照，必须提供这个表面的法线向量。我们可以计算这个表面上任意两条向量的向量积，然后对它进行规范化，这样就可以得到它的单位法线向量。在平表面或二十面体中，定义了每个表面的 3 个顶点具有相同的法线向量。在这种情况下，这组顶点（共包括 3 个顶点）只需要指定一条法线向量。示例程序 2-20 可以取代用于绘制二十面体的示例程序 2-19 的“color information here”行。

示例程序 2-20 生成表面的法线向量

```
1. GLfloat d1 [3],d2 [3],norm [3];
2. for (j =0;j <3;j++){
3. d1 [j] =vdata [tindices [i][0]][j] -vdata [tindices [i][1]][j];
4. d2 [j] =vdata [tindices [i][1]][j] -vdata [tindices [i][2]][j];
5. }
6. normcrossprod(d1,d2,norm);
7. glNormal3fv(norm);
```

normcrossprod()函数计算两个向量的向量积，然后对它进行规范化，如示例程序 2-21 所示。

示例程序 2-21 计算两个向量的规范化向量积

```
1. void normalize(float v [3])
2. {
3. GLfloat d =sqrt(v [0]*v [0]+v [1]*v [1]+v [2]*v [2]);
4. if (d ==0.0){
5. error("zero length vector ");
6. return;
7. }
8. v [0] /=d;
9. v [1] /=d;
10. v [2] /=d;
11. }
12. void normcrossprod(float v1 [3],float v2 [3] ,float out [3])
13. {
```

```
14. out [0] = v1 [1]*v2 [2] -v1 [2]*v2 [1];
15. out [1] = v1 [2]*v2 [0] -v1 [0]*v2 [2];
16. out [2] = v1 [0]*v2 [1] -v1 [1]*v2 [0];
17. normalize(out);
18. }
```

如果使用一个二十面体来模拟一个着色球体，需要使用垂直于球体真正表面的法线向量，而不是垂直于各个面的法线向量。对于球体而言，法线向量是非常简单的，每个点上法线向量的方向就是从原点到对应顶点的那条向量的方向。由于二十面体顶点数据表示一个半径为 1 的二十面体，因此法线数据和顶点数据是相同的。下面这段代码绘制了一个二十面体，用它来模拟一个平滑着色的球体（假定已经启用了光照，详见第 5 章）。

```
1.  glBegin(GL_TRIANGLES);
2.  for (i = 0; i < 20; i++) {
3.  glNormal3fv(&vdata[tindices[i][0]][0]);
4.  glVertex3fv(&vdata[tindices[i][0]][0]);
5.  glNormal3fv(&vdata[tindices[i][1]][0]);
6.  glVertex3fv(&vdata[tindices[i][1]][0]);
7.  glNormal3fv(&vdata[tindices[i][2]][0]);
8.  glVertex3fv(&vdata[tindices[i][2]][0]);
9.  } glEnd();
```

改进模型

用二十面体来模拟球体效果并不好，除非屏幕上球体的图像非常小。但是，可以使用一种非常容易的方法来改善模拟的精度。我们可以想象一下在一个在球体上所雕刻的二十面体，然后像图 2-17 一样对三角形进行细分。新增加的顶点稍稍嵌入球体的内部，因此通过对它们进行规范化，可以把它们推到表面上（把它们除以一个因子，使它们的长度为 1）。这个细分过程可以重复多次，以获得所需要的任意精度。图 2-17 所示的 3 个物体分别使用 20、80 和 320 个三角形来模拟球体。

图 2-17 通过细分改善多边形近似模拟法对表面的模拟精度

示例程序 2-22 执行一次细分，创建一个 80 个面的模拟球体。

示例程序 2-22 单次细分

```
1. void drawtriangle(float *v1,float *v2,float *v3)
2. {
3. glBegin(GL_TRIANGLES);
4. glNormal3fv(v1);
5. glVertex3fv(v1);
6. glNormal3fv(v2);
7. glVertex3fv(v2);
8. glNormal3fv(v3);
9. glVertex3fv(v3);
10. glEnd();
11. }
12. void subdivide(float *v1,float *v2,float *v3)
13. {
14. GLfloat v12 [3],v23 [3],v31 [3];
15. GLint i;
16. for (i =0;i <3;i++){
17. v12 [i] =(v1 [i]+v2 [i])/2.0;
18. v23 [i] =(v2 [i]+v3 [i])/2.0;
19. v31 [i] =(v3 [i]+v1 [i])/2.0;
20. }
21. normalize(v12);
22. normalize(v23);
23. normalize(v31);
24. drawtriangle(v1,v12,v31);
25. drawtriangle(v2,v23,v12);
26. drawtriangle(v3,v31,v23);
27. drawtriangle(v12,v23,v31);
28. }
29. for (i =0;i <20;i++){
30. subdivide(&vdata [tindices [i][0]][0],
31. &vdata [tindices [i][1]][0],
32. &vdata [tindices [i][2]][0]);
33. }
```

2.10 创建多边形表面模型的一些提示（3）

示例程序 2-23 对示例程序 2-22 稍稍作了修改，用递归的方法对三角形进行细分，以获得适当的深度。如果深度值是 0，就不会进行细分，三角形按照原样绘制。如果深度值是 1，就执行 1 次细分，以此类推。

示例程序 2-23 递归细分

```
1. void subdivide(float *v1,float *v2,float *v3,long depth)
2. {
3. GLfloat v12 [3],v23 [3],v31 [3];
4. GLint i;
5. if (depth ==0){
6. drawtriangle(v1,v2,v3);
7. return;
8. }
9. for (i =0;i <3;i++){
10.v12 [i] =(v1 [i]+v2 [i])/2.0;
11.v23 [i] =(v2 [i]+v3 [i])/2.0;
12.v31 [i] =(v3 [i]+v1 [i])/2.0;
13.}
14.normalize(v12);
15.normalize(v23);
16.normalize(v31);
17.subdivide(v1,v12,v31,depth-1);
18.
19.subdivide(v2,v23,v12,depth-1);
20.subdivide(v3,v31,v23,depth-1);
21.subdivide(v12,v23,v31,depth-1);
22.}
```

通用的细分法

示例程序 2-23 描述的递归细分法也可以用于其他类型的表面。一般情况下，在到达一定的深度或者满足某个曲率条件时，递归就会终止（表面上高度弯曲的部分细分越多效果越好）。

为了寻找一个更为通用的细分问题的解决方案，考虑由两个变量 u[0]和 u[1]为参数所表示的任意表面。假如提供了下面这两个函数：

```
1. void surf(GLfloat u[2], GLfloat vertex[3], GLfloat normal[3]);
2. float curv(GLfloat u[2]);
```

如果向 surf()函数传递 u[]，它就返回对应的三维顶点和法线向量（长度为 1）。如果向 curv()函数传递 u[]，它就计算并返回在这个点上表面的曲率（关于如何计算不同几何表面的曲率的更多信息，可以参阅相应的教科书）。

示例程序 2-24 显示了一个对三角形进行细分的递归函数。它将一直进行细分，直到到达最大深度或者 3 个顶点上的最大曲率小于某个门槛值。

示例程序 2-24 通用的细分法

```
1. void subdivide(float u1 [2] ,float u2 [2],float u3 [2],
2. float cutoff,long depth)
3. {
4. GLfloat v1 [3],v2 [3],v3 [3],n1 [3],n2 [3],n3 [3];
5. GLfloat u12 [2],u23 [2],u32 [2];
6. GLint i;
7. if (depth == maxdepth ||(curv(u1)<cutoff &&
8. curv(u2)<cutoff &&curv(u3)<cutoff)){
9. surf(u1,v1,n1);
10. surf(u2,v2,n2);
11. surf(u3,v3,n3);
12. glBegin(GL_POLYGON);
13. glNormal3fv(n1);glVertex3fv(v1);
14. glNormal3fv(n2);glVertex3fv(v2);
15. glNormal3fv(n3);glVertex3fv(v3);
16. glEnd();
17. return;
18. }
19. for (i =0;i <2;i++){
20. u12 [i] =(u1 [i] +u2 [i])/2.0;
21. u23 [i] =(u2 [i] +u3 [i])/2.0;
22. u31 [i] =(u3 [i] +u1 [i])/2.0;
23. }
24. subdivide(u1,u12,u31,cutoff,depth+1);
25. subdivide(u2,u23,u12,cutoff,depth+1);
26. subdivide(u3,u31,u23,cutoff,depth+1);
27. subdivide(u12,u23,u31,cutoff,depth+1);
28. }
```

第 3 章 视图

本章目标

在三维空间中对几何模型进行变换，以便从任何方向对它进行观察。

控制模型在三维空间中的位置。

裁剪位于场景之外的不需要的模型部分。

对用于控制模型变换的适当矩阵栈进行操作，以及查看模型，并把它投影到屏幕上。

组合多种变换，模拟运动中的复杂系统，例如太阳系或者带关节的机器人手臂。

逆变换或模拟几何图形处理管线的操作。

注意：在 OpenGL 3.1 中，本章介绍的很多技术和函数已经废弃删除了。正如第 15.1.1 节所述，概念仍然是相关的，但是，所提及的变换需要在一个顶点着色器中实现。

第 2 章介绍了如何指示 OpenGL 绘制希望在场景中显示的几何模型。现在，我们必须决定模型在场景中的位置，并且选择一个观察点来观察场景。虽然可以使用默认的物体位置和观察点，但是我们很可能想自己进行指定。

请观察本书封面的图像。产生这幅图像的程序包含了对建筑块的几何描述。每个建筑块在场景中的位置各异：有些建筑块散落在地面上，有些堆叠在桌子上，还有一些则构成了地球仪。另外，必须选择一个特定的观察点。显然，我们希望看到的是房间内部包含地球仪的那个角落。但是，观察者应该距离场景多远呢？或者准确地说，他应该站在什么位置？我们希望从这个场景的最终图像能够清楚地看到窗外的景色，另外还能够看到一部分地板。场景中的物体不仅都能够被看到，而且它们的排列也要比较和谐。本章将介绍如何使用 OpenGL 来完成这些任务，包括如何在三维空间中设置模型的位置和方向，以及如何确定观察者的位置（也是在三维空间中）。综合了所有这些因素之后，就能够准确地判断屏幕上所显示的图像。

记住，计算机图形的要点就是创建三维物体的二维图像（图像必须是二维的，因为它是在平面的屏幕上显示的）。但是，当我们决定怎样在屏幕上绘图时，必须使用三维坐标的方式来考虑。人们在创建三维图像时经常犯的一个错误就是太早考虑在平面的二维屏幕上所显示的最终图像。我们要避免考虑屏幕上的像素是如何绘制的，而是要尽量在三维空间中想象物体的形状。我们需要在深入计算机内部的三维空间中创建模型，让计算机去计算哪些像素需要绘制。

为了把一个物体的三维坐标变换为屏幕上的像素坐标，需要完成如下 3 个步骤：

变换包括模型、视图和投影操作，它们是由矩阵乘法表示的。这些操作包括旋转、移动、缩放、反射、正投影和透视投影等。一般情况下，在绘制场景时需要组合使用几种变换。

由于场景是在一个矩形窗口中渲染的，因此位于窗口之外的物体（或物体的一部分）必须裁剪掉。在三维计算机图形中，裁剪就是丢弃位于裁剪平面之外的物体。

最后，经过了变换的坐标和屏幕像素之间必须建立对应关系。这个过程称为视口（viewport）变换。

本章将描述所有这些操作，并介绍如何对它们进行控制，具体内容分布如下：

“简介：照相机比喻”：对变换过程进行简单的介绍。本节以照相机拍照为比喻，提供了一个对物体进行变换的简单示例程序。另外，还简单地介绍了一些基本的 OpenGL 变换函数。

视图和模型变换：详细解释如何指定并想象视图和模型变换的效果。这些变换对模型和照相机的相对位置进行定位，以获得所需的最终图像。

投影变换：描述如何指定视景体（viewing volume）的形状和方向。视景体决定了场景是如何投影到屏幕上的（使用正投影或透视投影），并决定了哪些物体（或物体的一部分）将裁剪掉。

视口变换：解释如何把三维的模型坐标转换为屏幕坐标。

和变换相关的故障排除：提供一些指导意见，如果模型、视图、投影和视口变换无法获得预期的效果，它可以帮助读者诊断其中的原因。

操纵矩阵堆栈：讨论如何保存和恢复一些变换。当根据一些简单的物体绘制复杂的物体时，这方面的内容就显得格外重要。

其他裁剪平面：描述如何在视景体所定义的裁剪平面之外指定其他裁剪平面。

一些组合变换的例子：带领读者领略一些更为复杂的变换用途。

逆变换或模拟变换：显示如何取窗口坐标中的一点，并对它进行逆变换，以获得原来的物体坐标。另外，还可以对变换本身进行模拟（不使用逆变换）。

OpenGL 1.3 版本增加了一些新函数，对行主序的矩阵（OpenGL 的术语称为变换矩阵）提供了直接的支持。

3.1 简介：用照相机打比方

产生目标场景视图的变换过程类似于用照相机进行拍照。如图 3-1 所示，用照相机（或计算机）进行拍照的步骤大致如下：

1) 把照相机固定在三角架上，并让它对准场景（视图变换）。

2) 对场景进行安排，使各个物体在照片中的位置是我们所希望的（模型变换）。

3) 选择照相机镜头，并调整放大倍数（投影变换）。

4) 确定最终照片的大小。例如，我们很可能需要把它放大（视口变换）。

在完成这些步骤之后，就可以进行拍照（或者绘制场景）了。

注意，这些步骤对应于程序指定的变换顺序，但是并不一定就是 OpenGL 在物体的顶点上所执行的相关数学操作的顺序。在代码中，视图变换必须出现在模型变换之前，但是可以在绘图之前的任何时候执行投影变换和视口变换。图 3-2 显示了这些操作在计算机上的出现顺序。

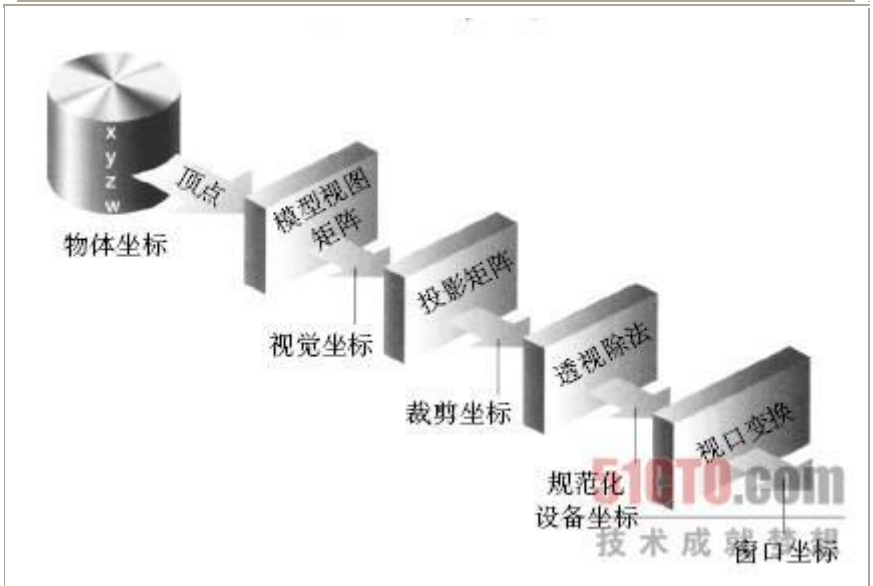
为了指定视图、模型和投影变换，可以创建一个 4×4 的矩阵 M，然后把它与场景中每个顶点 v 的坐标相乘，以实现变换：

$$v' = \mathbf{M}v$$

记住，顶点总是有 4 个坐标（x, y, z, w），尽管在绝大多数情况下 w 都固定为 1。对于二维数据，z 总是为 0。注意，除了顶点之外，视图和模型变换还会自动作用于表面法线向量（法线向量只用于视觉坐标，eye coordinate）。这就保证了法线向量和顶点数据之间具有正确的对应关系。

视图和模型变换一起形成了模型视图矩阵，这个矩阵作用于物体坐标（object coordinate），产生视觉坐标。接着，如果还指定了其他裁剪平面，用于从场景中删除某些物体或者提供物体的裁剪视图，这些裁剪平面就会在此时生效。

（点击查看大图）图 3-1 照相机比喻



（点击查看大图）图 3-2 顶点变换的步骤

此后，OpenGL 使用投影矩阵产生裁剪坐标（clip coordinate）。这个变换定义了一个视景体，位于这个空间之外的物体将裁剪掉，不会在最终的场景中出现。随后发生的是透视除法（perspectivedivision），它把坐标值除以 w，产生规范化的设备坐标（normalized device coordinates）（关于 w 坐标的含义以及它如何影响矩阵变换的更多信息，请参阅附录 F）。最后，经过变换的坐标经过视口变换，成为窗口坐标（window coordinate）。可以通过控制视口的大小对最终的图像进行放大、缩小或拉伸。尽管 x 和 y 坐标就足以确定需要在屏幕上绘制哪些像素，但是所有的变换也会对 z 坐标也进行操作。

如此一来，到了变换过程的最后，z 值能够准确地反映特定顶点的深度（从顶点到屏幕的距离）。深度值的作用之一就是消除不必要的绘图。例如，假如有两个表面的顶点具有相同的 x 值和 y 值，但是它们的 z 值不同。OpenGL 可以使用这个信息判断哪个表面被另一个表面遮挡，从而避免绘制那个被隐藏的表面。关于这个技巧（称为隐藏表面消除）的更多信息，请参见第 5 章和第 10 章。

现在，读者很可能已经猜到，为了理解本章介绍的内容，需要了解一些有关矩阵的数学知识。如果读者想温习一下这方面的知识，可以参阅有关线性代数的教科书。

3.1.1 一个简单的例子：绘制立方体

示例程序 3-1 绘制了一个通过模型变换进行缩放的立方体如图 3-3 所示。视图变换函数 gluLookAt()设置照相机的位置，并使它对准需要绘制立方体的位置。另外，这个例子还指定了投影变换和视口变换。本节的剩余部分将讨论示例程序 3-1，并简单地解释它所使用的变换命令。接下来的几节完整而又详细地讨论了所有的 OpenGL 变换函数。

图 3-3 变换立方体

示例程序 3-1 变换立方体：cube.c

```
1. void init(void)
2. {
3. glClearColor(0.0,0.0,0.0,0.0);
4. glShadeModel(GL_FLAT);
5. }
6. void display(void)
7. {
8. glClear(GL_COLOR_BUFFER_BIT);
```

```
9.  glColor3f(1.0,1.0,1.0);
10. glLoadIdentity(); /*clear the matrix */
11. /*viewing transformation */
12. gluLookAt(0.0,0.0,5.0,0.0,0.0,0.0,0.0,1.0,0.0);
13. glScalef(1.0,2.0,1.0); /*modeling transformation */
14. glutWireCube(1.0);
15. glFlush();
16. }
17. void reshape(int w,int h)
18. {
19. glViewport(0,0,(GLsizei)w,(GLsizei)h);
20. glMatrixMode(GL_PROJECTION);
21. glLoadIdentity();
22. glFrustum(-1.0,1.0,-1.0,1.0,1.5,20.0);
23. glMatrixMode(GL_MODELVIEW);
24. }
25. int main(int argc,char**argv)
26. {
27. glutInit(&argc,argv);
28. glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB);
29. glutInitWindowSize(500,500);
30. glutInitWindowPosition(100,100);
31. glutCreateWindow(argv [0]);
32. init();
33. glutDisplayFunc(display);
34. glutReshapeFunc(reshape);
35. glutMainLoop();
36. return 0;
37. }
```

视图变换

视图变换相当于把照相机固定在三角架上并使它对准场景。在这个示例程序中，在指定视图变换之前，需要使用 glLoadIdentity()函数把当前矩阵（current matrix）设置为单位矩阵。这个步骤是非常必要的，因为绝大多数变换是把当前矩阵与指定的矩阵相乘，然后把结果指定为当前矩阵。如果没有通过加载单位矩阵来清除当前矩阵，它所进行的变换实际上是把当前的变换与上一次变换进行了组合。在有些情况下，确实需要执行这种组合式的变换。但是在更多的情况下，还是需要清除当前矩阵。

在示例程序 3-1 中，对矩阵进行初始化之后，使用 gluLookAt()函数指定了视图变换。这个函数的参数表示照相机（或眼睛）的位置、瞄向以及哪个方向是朝上的。这个程序所使用的参数把照相机放在（0，0，5），把镜头瞄准（0，0，0），并把朝上向量指定为（0，1，0）。朝上向量为照相机指定了唯一的方向。

如果并没有调用 gluLookAt()，那么照相机就被设置为默认的位置和方向。在默认情况下，照相机位于原点，指向 z 轴的负方向，朝上向量为（0，1，0）。因此，在示例程序 3-1 中，调用这个 gluLookAt()函数的总体效果就是把照相机沿 z 轴移动 5 个单位（关于视图变换的更多信息，请参阅第 3.2 节）。

模型变换

使用模型变换的目的是设置模型的位置和方向。例如，可以对模型进行旋转、移动和缩放，或者联合应用这几种操作。示例程序 3-1 使用的模型变换函数是 glScalef()。这个函数的参数指定了物体在 3 个轴上是如何进行缩放的。如果所有的参数均设置为 1.0，这个函数就没有任何效果。在示例程序 3-1 中，这个立方体在 y 轴方向上的大小是原来的两倍。因此，如果这个立方体的其中一个角原先位于（3.0，3.0，3.0），那么经过缩放之后，它将位于（3.0，6.0，3.0）。这个模型变换的效果就是使这个立方体不再是原先的立方体，而成为了长方体。

尝试一下

修改示例程序 3-1 的 gluLookAt()函数，把它改为模型变换函数 glTranslatef()，并使用参数（0.0,0.0, -5.0）。这个函数的效果应该和使用 gluLookAt()函数的效果完全相同。为什么这两个命令的效果会完全相同呢？

注意，可以不必通过移动照相机（使用视图变换）来观察这个立方体，而是移动这个立方体（使用模型变换）。视图变换和模型变换的这种双重性本质就是需要同时考虑这两种类型的变换的原因。把这两种变换割裂开来是没有意义的，但有时候使用其中一种变换比使用另外一种变换要方便得多。这也是在进行变换之前把模型和视图变换组合为模型视图矩阵（modelview matrix）的原因（关于如何考虑模型和视图变换以及如何指定它们以获得所需的结果，可以参阅第 3.2 节）。

另外，注意 display()函数包括了模型和视图变换，另外还包括了用于绘制立方体的函数 glutWireCube()。这样，display()函数可以重复调用，用于绘制窗口的内容。例如，当窗口被移动或者原先遮住这个窗口的物体被移开时，就需要重复调用 display()方法。这样，可以保证这个立方体是按照预想的方式绘制的，并且经过了适当的变换。display()的这种潜在的重复使用性进一步强调了在执行视图和模型变换之前加载单位矩阵的重要性，特别是在两次 display()调用之间还插入了其他变换的情况下。

投影变换

指定投影变换就好像为照相机选择镜头。可以认为这种变换的目的是确定视野（或视景体），并因此确定哪些物体位于视野之内以及它们能够被看到的程度。以照相机作比喻，这个操作相当于选择广角镜头、标准镜头还是长焦镜头。使用广角镜头，最终照片的场景范围要远大于使用长焦镜头。但是，长焦镜头可以使远处的物体看上去比实际上更靠近相机。使用计算机图形，就不必花费 1 万美元买一个 2 000 毫米的长焦镜头。如果拥有自己的图形工作站，只要使用更小的视野就可以实现这个效果。

除了考虑视野之外，投影变换还决定了物体是如何投影到屏幕上的（就像它的名称所提示的那样）。OpenGL 提供了两种基本类型的投影，并且提供了一些对应的函数以不同的方法描述相关的参数。其中一种类型是透视投影（perspective projection），它类似于我们在日常生活看到的景象。透视投影使远处的物体看上去更小一些。例如，它使铁轨看上去似乎在远处的某个点会聚。如果想创建现实感很强的图像，就需要选择透视投影。在示例程序 3-1 中，通过 glFrustum()函数指定了使用透视投影。

另一种类型的投影称为正投影（orthographic projection），它把物体直接映射到屏幕上，而不影响它们的相对大小。正投影一般用于建筑和计算机辅助设计（CAD）应用程序中。在这些应用程序中，最终的图像需要反映物体的实际大小，而不是它们看上去的样子。建筑师需要使用透视图，从不同的角度观察特定建筑的外部或内部空间是什么样子的。当建筑师创建规划或评估蓝图显示建筑物的结构时，需要使用正投影（关于这两种投影类型的详细信息，请参见第 3.3 节）。

在调用 glFrustum()设置投影变换之前，需要做一些准备工作。如示例程序 3-1 的 reshape()函数所示，首先需要以 GL_PROJECTION 为参数调用 glMatrixMode()函数，表示把当前矩阵指定为用于投影变换，并且后续的变换调用所影响的是投影矩阵（projection matrix）。可以看到，在几行代码之后，又以 GL_MODELVIEW 为参数调用了 glMatrixMode()函数。这次调用表示以后的变换所影响的是模型视图矩阵而不再是投影矩阵（关于如何控制投影矩阵和模型视图矩阵的更多信息，请参阅第 3.6 节）。注意，glLoadIdentity()函数用于对当前的投影矩阵进行初始化。这样，只有指定的投影变换才会产生效果。接下来，调用了 glFrustum()函数，它的参数定义了投影变换的参数。在这个例子中，reshape()函数同时包括了投影变换和视口变换。当窗口初次创建时，或者当窗口移动或改变形状时，reshape()函数就会调用。这是合理的，因为投影（投影视景体的宽度-高度纵横比）和视口都是与屏幕直接相关的，特别是与屏幕或窗口的大小或纵横比直接相关。

尝试一下

把示例程序的 glFrustum()调用修改为更常用的工具函数库 gluPerspective()函数，并以（60.0，1.0，1.5，20.0）为参数。然后试验不同的参数值，尤其是 fovy 和 aspect 参数。

视口变换

投影变换和视口变换共同决定了场景是如何映射到计算机屏幕的。投影变换指定了映射的发生机制，而视口变换则决定了场景所映射的有效屏幕区域的形状。由于视口指定了场景在屏幕上所占据的区域，因此可以把视口变换看成是定义了最终经过处理的照片的大小和位置。例如，照片是否应该放大或缩小。

glViewport()的参数描述了窗口内部有效屏幕空间的原点，在此例中为（0，0）。另外，它还描述了有效屏幕区域的宽度和高度（均以屏幕像素为单位）。这个函数需要在 reshape()函数内部调用的原因是，如果窗口的大小发生了变化，视口也需要相应地改变。注意，宽度和高度是用窗口的实际宽度和高度指定的。我们常常需要以这种方式来指定视口，而不是给出绝对大小（关于如何定义视口的更多信息，请参阅第 3.4 节）。

绘制场景

在指定了所有必要的变换之后，就可以绘制场景了（也就是说，可以进行拍照了）。在绘制场景时，OpenGL 通过模型和视图变换对场景中每个物体的每个顶点进行变换。然后，根据指定的投影变换对每个顶点进行变换。如果顶点位于视景体（由投影变换描述）之外，它就被裁剪掉。最后，经过变换的剩余顶点除以 w，然后映射到视口中。

3.1.2 通用的变换函数

本节讨论一些常用的 OpenGL 函数，可以使用这些函数指定需要的变换。读者已经看到了其中两个函数：glMatrixMode() 和 glLoadIdentity()。本节描述的 4 个函数（glLoadMatrix*()、glLoadTransposeMatrix*()、glMultMatrix*()和 glMultTransposeMatrix*()）允许直接指定任何变换，或者把当前矩阵与指定的矩阵相乘。至于其他几个更为特殊的变换函数（如 gluLookAt()和 glScale*()），将在以后的章节中讨论。

正如前一节所述，在调用变换函数之前，需要确定自己想修改的是模型视图矩阵还是投影矩阵。可以使用 glMatrixMode()选择矩阵。在使用可能重复调用的 OpenGL 函数嵌套子集时，记住要正确地对矩阵模式进行重置。glMatrixMode()函数还可以用于指定纹理矩阵（有关纹理矩阵的细节，请参阅第 9.12 节）。

```
1. void glMatrixMode(GLenum mode);
```

指定了需要修改的是模型视图矩阵、投影矩阵还是纹理矩阵。mode 的值可以是 GL_MODELVIEW、GL_PROJECTION 或 GL_TEXTURE。接下来调用的变换函数将影响它指定的矩阵。注意，一次只能修改一个矩阵。在默认情况下，变换函数修改的矩阵是模型视图矩阵。另外，在默认情况下这 3 个矩阵均为单位矩阵。

可以使用 glLoadIdentity()函数，把当前的可修改矩阵清除为单位矩阵，供未来的变换函数使用（这些函数将对当前矩阵进行修改）。一般而言，我们总是在指定投影或视图变换之前调用这个函数。但是，也可以在指定模型变换之前调用这个函数。

```
1. void glLoadIdentity(void);
```

把当前的可修改矩阵设置为 4×4 的单位矩阵。

如果需要显式地指定一个特定的矩阵，并把它加载到当前矩阵，可以使用 glLoadMatrix*()或 glLoadTransposeMatrix*()函数。类似地，可以使用 glMultMatrix*()或 glMultTransposeMatrix*()函数把当前矩阵与参数所指定的矩阵相乘。

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

把当前矩阵的 16 个值设置为 m 指定的值。

```
1. void glMultMatrix{fd}(const TYPE *m);
```

把 m 指定的 16 个值作为一个矩阵，与当前矩阵相乘，并把结果存储在当前矩阵中。

在 OpenGL 中， 所有的矩阵乘法都是按照下面这种方式进行的： 假设当前矩阵是 C ，glMultMatrix*()或其他任何变换函数指定的矩阵是 M，在相乘之后，最终的矩阵总是 CM。由于矩阵乘法一般不满足交换律，所以这两个矩阵的顺序是不可交换的。

glLoadMatrix*()和 glMultMatrix*()函数的参数是一个包含 16 个值的向量（m1, m2,., m16），它按照列主序指定了一个矩阵 M，如下所示：

$$M = \begin{bmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \\ m_4 & m_8 & m_{12} & m_{16} \end{bmatrix}$$

如果用 C 语言编程，并且声明了一个像 m[4][4]这样的矩阵，那么元素[i][j]相当于普通 OpenGL 变换矩阵的第 i 列和第 j 行。这正好与[i][j]是第 i 行第 j 列的标准 C 约定相反。为了避免矩阵行列上的混淆，可以把矩阵声明为 m[16]。

避免出现这种混淆的另一种办法是调用 OpenGL 函数 glLoadTransposeMatrix*()和 glMult-TransposeMatrix*()，它们使用行主序（标准 C 约定）的矩阵作为参数。

```
1. void glLoadTransposeMatrix{fd}(const TYPE *m);
```

把当前矩阵的 16 个值设置为参数 m 指定的矩阵。这个矩阵是按照行主序存储的。glLoad TransposeMatrix*(m)和 glLoadMatrix*(mT)具有相同的效果。

```
1. void glMultTransposeMatrix{fd}(const TYPE *m);
```

把参数 m 指定的 1 6 个值所组成的矩阵与当前矩阵相乘， 并把结果存储在当前矩阵中。glMultTransposeMatrix*(m)和 glMultMatrix*(mT)具有相同的效果。

为了最大限度地提高效率，可以使用显示列表来存储经常使用的矩阵（以及它们的逆矩阵），而不是每次重新计算它们（参见第 7.3 节）。OpenGL 实现通常必须计算模型视图矩阵的逆矩阵，使法线和裁剪平面能够正确地变换到视觉坐标中。

3.2 视图和模型变换

在 OpenGL 中，视图和模型变换是密切相关的，事实上它们可以组合为单个模型视图矩阵。在第 3.1.1 节中，我们曾经提到过这一点。在计算图形表面时，初学者所面临的最严峻的问题之一就是理解三维变换的组合效果。如前所述，可以换一种思路来考虑变换：从一个方向移动照相机相当于从相反的方向移动物体。每种思路都有各自的优势和劣势。有些情况下，其中一种思路能够更自然地符合目标变换的结果。如果在应用程序中能够找到一种更为自然的方法，显然更容易想象它所需要的必要变换，并编写相应的代码来指定矩阵操作。本节的第一部分讨论如何思考变换。然后，再介绍一些特定的函数。就目前而言，我们只使用前面介绍的矩阵操纵函数。最后再强调一次，在执行模型或视图变换之前，必须以 GL_MODELVIEW 为参数调用 glMatrixMode()函数。

3.2.1 对变换进行思考

首先，考虑两个简单的变换：一个是沿原点绕 z 轴逆时针旋转 45 度，另一个是沿 x 轴向下平移。假设我们绘制的物体相对于移动的距离而言较小（这样就容易看到移动的效果），并且它原先并不是位于原点上。如果首先旋转这个物体，然后移动它，经过旋转的物体将会出现在 x 轴上。但是，如果首先将它沿 x 轴下移，然后再让它沿原点旋转 45 度，那么这个物体将位于 y = x 的直线上，如图 3-4 所示。一般而言，变换的顺序是至关重要的。如果先执行变换 A，然后再执行变换 B，其结果几乎肯定和以相反的顺序进行这两个变换的结果不同。

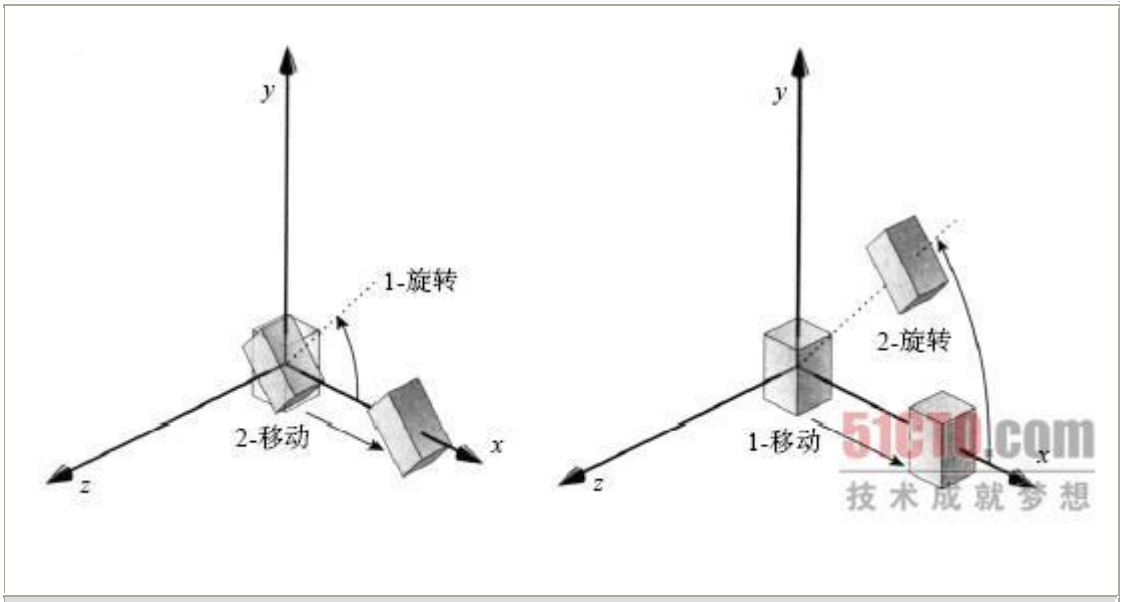


图 3-4 首先旋转或首先移动

现在我们讨论这些变换的发生顺序。所有的视图和模型变换都是用一个 4×4 的矩阵表示的。每个后续的 glMultMatrix*()函数或变换函数把一个新的 4×4 矩阵 M 与当前的模型视图矩阵 C 相乘，产生结果矩阵 CM。最后，每个顶点 v 与当前的模型视图矩阵相乘。这个过程意味着程序所调用的最后一个变换函数实际上是首先应用于顶点的：CMv。因此，另一种说法就是我们以相反的顺序指定了这些矩阵。但是，和许多其他事情一样，一旦我们已经习以为常（并认为它是正确的），后退和前进其实是一样的。

考虑下面的代码序列，它使用 3 个变换绘制了 1 个点：

```
1. glMatrixMode(GL_MODELVIEW);
2. glLoadIdentity();
3. glMultMatrixf(N); /* apply transformation N */
4. glMultMatrixf(M); /* apply transformation M */
5. glMultMatrixf(L); /* apply transformation L */
6. glBegin(GL_POINTS);
7. glVertex3f(v); /* draw transformed vertex v */
8. glEnd();
```

在这段代码中，模型视图矩阵按顺序分别包含了 I、N、NM，最后是 NML，其中 I 表示单位矩阵。经过变换的顶点是 NMLv。因此，顶点变换就是 N(M(Lv))，也就是说，v 首先与 L 相乘，Lv 再与 M 相乘，MLv 再与 N 相乘。注意，顶点 v 的变换是按照相反的顺序发生，而不是按照它们的指定顺序出现的（实际上，一个顶点与模型视图矩阵的乘法只出现一次。在这个例子中，N、M 和 L 在应用于 v 之前已经与一个矩阵相乘）。

全局固定坐标系统

因此，如果想根据一个全局固定的坐标系统来考虑问题（在这种坐标系统中，矩阵乘法将影响模型的位置、方向和缩放），就必须注意乘法的出现顺序与它们在代码中出现的顺序相反。我们简单地以图 3-4 左侧的那个变换（沿原点旋转，然后再沿 x 轴下移）为例，如果想让物体在经过变换之后出现在轴上，必须首先进行旋转，然后再进行移动。为此，需要反转操作的顺序，因此它的代码大概像下面这样（其中 R 是旋转矩阵，T 是移动矩阵）：

```
1. glMatrixMode(GL_MODELVIEW);
2. glLoadIdentity();
3. glMultMatrixf(T); /* translation */
4. glMultMatrixf(R); /* rotation */
5. draw_the_object();
```

局部移动坐标系统

考虑矩阵乘法的另一种思路是抛弃在变换模型时所使用的全局固定坐标系统，而是想象一个固定到所绘制物体的局部坐标系统。所有的操作都相对于这个不断发生变化的坐标系统进行。按照这种方法，矩阵乘法很自然地按照它们在代码中出现的顺序进行。（这和我们所使用的比喻无关，代码是一样的，但思考它的方式可以不同。）现在，让我们在这个移动-旋转例子中使用这种局部坐标系统。首先，想象这个物体具有一个固定的坐标系统。移动操作使这个物体沿 x 轴下移，因此这个局部坐标系统也随之沿 x 轴下移。然后，这个物体沿原点（经过移动之后的原点）进行旋转，因此这个物体是在轴上的位置进行旋转的。

在诸如带关节的机器人手臂（肩、肘、腕以及手指等部位均存在关节）这样的应用程序中，应该使用这种方法。为了判断指尖相对于身体的位置，首先要从肩开始，然后是肘、腕，接着才是手指，在每个关节应用适当的旋转和移动。如果以相反的顺序考虑这些变换，显然会复杂无比。但是，如果变换包含了缩放，尤其是当缩放为非规则缩放的时候（在不同的轴上以不同的数量进行缩放），第二种方法就会出现问题。在规则缩放之后，如果再对一个顶点进行移动，它的实际移动量就是原先移动量乘以缩放倍数，因为坐标系统也随之进行了缩放。如果混合了非规则缩放和旋转，在变换之后，这个局部坐标系统的各个轴可能不再互相垂直。

如前所述，我们一般是在进行任何模型变换之前调用视图变换命令。这样，模型中的顶点首先变换到所需要的方向，然后根据视图操作进行变换。由于矩阵乘法必须以相反的顺序出现，因此视图命令需要首先出现。但是，如果满足于默认的条件，就不需要指定视图或模型变换。如果不进行视图变换，照相机就位于默认的原点位置，指向 z 轴的负方向。如果不进行模型变换，模型就不会移动，它仍然保持原先指定的位置、方向和大小。

由于执行模型变换的函数也可以用于执行视图变换，所以我们首先讨论模型变换，尽管实际上首先进行的是视图变换。这种讨论顺序和许多程序员在设计代码时所考虑的方式相同。他们常常编写所有必要的代码来组成场景，这就涉及对物体的位置和方向进行正确的变换。接着，决定在什么地方设置观察点（相对于这些物体所构成的场景），然后编写相应的视图变换代码。

3.2.2 模型变换

在 OpenGL 中，有 3 个函数用于执行模型变换，它们是 glTranslate*()、glRotate*()和 glScale*()。正如我们预想的那样，这些函数通过移动、旋转、拉伸、收缩或反射，对物体（或坐标系统，取决于思考方式）进行变换。这 3 个函数都相当于产生一个适当的移动、旋转或缩放矩阵，然后以这个矩阵作为参数调用 glMultMatrix*()。但是，使用这 3 个函数可能比使用 glMultMatrix*()速度更快。OpenGL 会自动计算矩阵（关于这方面的细节，请参阅附录 F）。

在下面的函数说明中，每个矩阵乘法是根据它在固定坐标系统对几何物体的顶点所进行的操作，以及它对固定到物体的局部坐标系统进行的操作来描述的。

变换

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

把当前矩阵与一个表示移动物体的矩阵相乘。这个移动矩阵由 x、y 和 z 值指定（或者在局部坐标系统中移动相同的数量）。

图 3-5 显示了 glTranslate*()的效果。

注意，使用（0.0, 0.0, 0.0）为参数调用 glTranslate*()是单位操作。也就是说，它对物体或物体的局部坐标系统不会产生任何效果。

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

把当前矩阵与一个表示移动物体（或物体的局部坐标系统）的矩阵相乘，以逆时针方向绕着从原点到点（x, y, z）的直线进行旋转。angle 参数指定了旋转的度数。

glRotatef (45.0, 0.0, 0.0, 1.0)的效果相当于沿 z 轴旋转 45 度，如图 3-6 所示。

图 3-5 移动物体

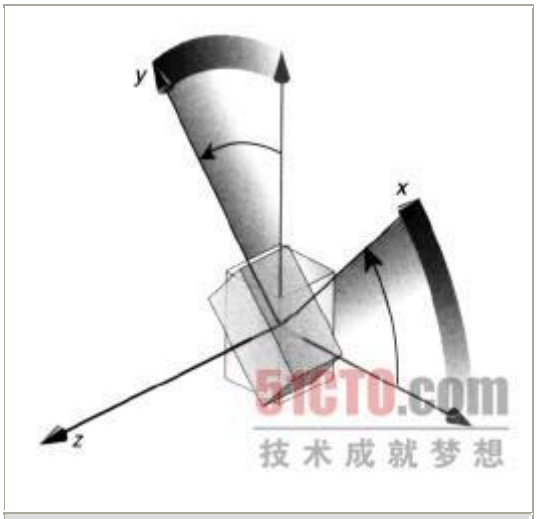


图 3-6 旋转物体

注意，远离旋转轴的物体比靠近旋转轴的物体的旋转辐度更大（轨道更长）。另外，如果 angle 参数为零，glRotate*()函数就不会产生任何效果。

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

把当前矩阵与一个表示沿各个轴对物体进行拉伸、收缩或反射的矩阵相乘。这个物体中每个点的 x、y 和 z 坐标与对应的 x、y 和 z 参数相乘。使用局部坐标系统方法，局部坐标系统的轴将由 x、y 和 z 因子拉伸、收缩或反射，相关联的物体也根据它们进行变换。

图 3-7 显示了 glScalef(2.0, -0.5, 1.0)的效果。

图 3-7 缩放和反射物体

glScale*()是 3 种模型变换中唯一能够改变物体大小的：如果缩放值大于 1.0，它就拉伸物体；如果缩放值小于 1.0，它就收缩物体；如果缩放值为-1.0，它就沿相应的轴反射这个物体。单位缩放值是（1.0, 1.0, 1.0）。在一般情况下，应该限制 glScale*()的使用，仅把它

用于那些必须进行缩放的场合。使用 glScale*()会降低光照计算的效率，因为在变换之后，法线向量必须重新进行规范化。

注意：如果缩放值是 0，它会把所有沿这个轴的物体坐标收缩为 0。通常，这并不是一种好的做法，因为这种操作是无法还原的。从数学角度而言，这种矩阵是没有逆矩阵的。但是对于有些光照计算，逆矩阵又是必需的（参见第 5 章）。有时候，将坐标值收缩为 0 确实有意义，例如在计算平表面的阴影的时候（参见第 14.15 节）。一般而言，如果一个坐标系统需要收缩为 0，应该使用投影矩阵，而不是模型视图矩阵。

一个模型变换的代码例子

示例程序 3-2 是一个程序的一部分，这个程序对一个三角形进行了 4 次渲染，如图 3-8 所示。下面就是这 4 个经过变换的三角形：

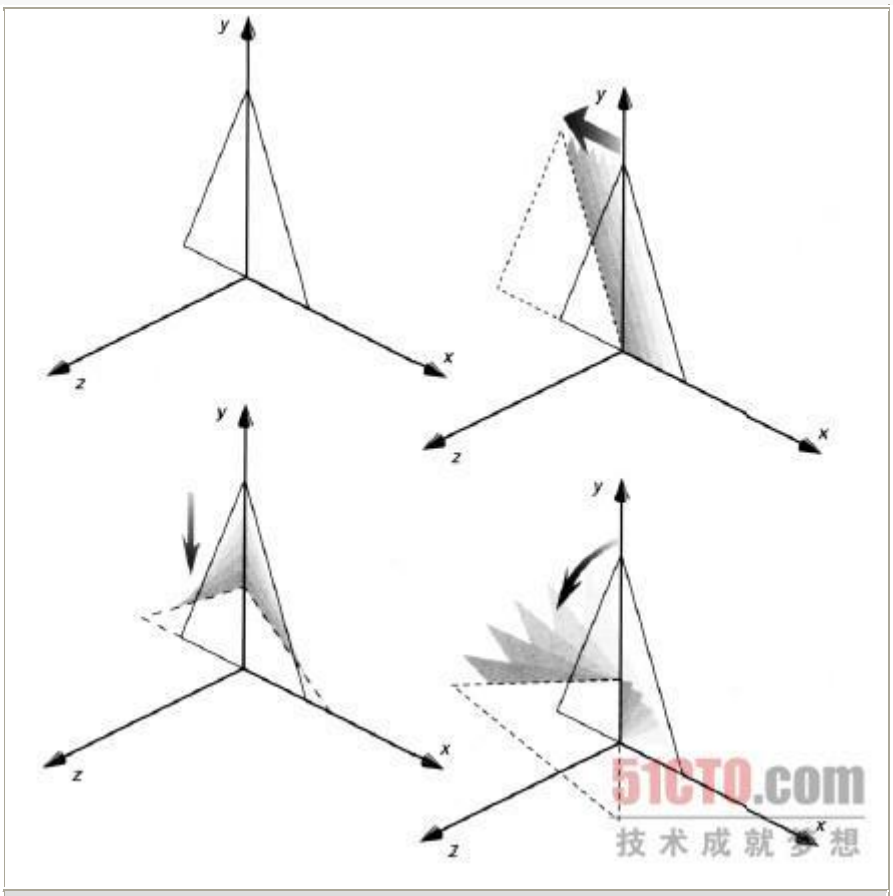


图 3-8 模型变换例子

一个用实线绘制的线框三角形，不使用模型变换。

再次绘制同一个三角形，但这次使用短虚线点画模型，并且移动它（向左，即 x 轴负方向）。

用长虚线点画模式绘制一个三角形，它的高度（y 轴）是原来的一半，宽度（x 轴）增加 50%。

一个旋转三角形，由点画线绘制而成。

示例程序 3-2 使用模型变换：model.c

```
glLoadIdentity();
glColor3f(1.0,1.0,1.0);
draw_triangle(); /*solid lines */
glEnable(GL_LINE_STIPPLE); /*dashed lines */
glLineStipple(1,0xF0F0);
glLoadIdentity();
glTranslatef(-20.0,0.0,0.0);
draw_triangle();
glLineStipple(1,0xF00F); /*long dashed lines */
glLoadIdentity();
glScalef(1.5,0.5,1.0);
draw_triangle();
glLineStipple(1,0x8888); /*dotted lines */
glLoadIdentity();
glRotatef(90.0,0.0,0.0,1.0);
draw_triangle();
glDisable(GL_LINE_STIPPLE);
```

注意，使用 glLoadIdentity()函数的目的是隔离各个模型变换的效果。对矩阵值进行初始化能够防止连续变换产生的累积效果。尽管反复使用 glLoadIdentity()能够实现预想的效果，但是它的效率可能较低，因为可能必须重新指定视图或模型变换（关于隔离各个变换操作的更好方法，参见第 3.6 节）。注意：有时候，为了实现连续的旋转，我们可能会想到重复应用一个值很小的旋转矩阵。

这个技巧存在的问题是：由于四舍五入的误差，经过数千次微小的旋转之后，物体的最终位置可能和预想的不一样。因此，应该摒弃这种做法，而是在每次更新时发布一条新的命令，并使用一个新的角度作为参数。

Nate Robin 的变换教程

如果读者已经下载了 Nate Robin 的教学程序包，现在就是运行 transformation 教程的好时机。在这个教程中，可以试验旋转、移动和缩放的效果（关于如何下载这些程序以及从何处下载它们，请参阅前言部分的“Nate Robin 的 OpenGL 教程”一节）。

3.2.3 视图变换（1）

视图变换用于修改观察点的位置和方向。如果回忆那个照相机比喻，视图变换就相当于把照相机固定到三角架上，并让它对准模型。这就像把照相机移动到某个位置，并对它进行

旋转，直到它指向我们所需要的方向一样。视图变换一般也是由移动和旋转组成的。为了在最终的图像或照片上实现某种场景组合，可以移动照相机，也可以从相反的方向移动所有的物体。因此，一个按照逆时针方向旋转物体的模型变换相当于一个按照顺时针方向旋转照相机的视图变换。最后，要记住视图变换函数必须在调用任何模型变换函数之前调用，以确保首先作用于物体的是模型变换。

可以使用好几种方法生成视图变换（稍后描述）。也可以选择使用默认的观察点位置（原点）和方向（z 轴的负方向）。

使用一个或多个模型变换函数（也就是 glTranslate*()和 glRotate*()）。可以把这些变换的效果看成是移动照相机位置，也可以看成是在全局范围内移动所有的物体（相对于静止的照相机）。

使用工具函数库的 gluLookAt()函数定义一条视线。这个函数封装了一系列的旋转和移动函数。

创建自己的工具函数，对旋转和移动进行封装。有些应用程序可能要求自定义的函数，允许采用一种方便的方式指定视图变换。例如，我们可能想为一架飞行中的飞机指定倾测角、螺旋角和航向改变角，或者可能想根据极坐标为一架绕物体旋转的照相机指定一个变换。

使用 glTranslate*()和 glRotate*()

图 3-9 原点的物体和观察点

使用模型变换函数模拟视图变换时，它的做法相当于在场景中的物体保持静止的前提下按照预想的方式移动观察点。由于观察点最初位于原点，并且由于在原点创建物体非常方便（如图 3-9 所示），因此必须执行一些变换，使这些物体能够被看到。注意，在图 3-9 中，照相机最初指向 z 的负方向（我们所看到的是照相机的背面）。

在最简单的情况下，可以向后移动观察点，使它远离物体。这种做法的效果相当于向前移动物体（也就是远离观察点）。记住，在默认情况下，向前就是沿 z 轴的负方向。如果旋

转了观察点，向前就具有不同的含义。因此，为了像图 3-10 一样，通过移动观察点让它和物体之间距离 5 个单位，可以使用下面的方法：

```
1. glTranslatef(0.0, 0.0, -5.0);
```

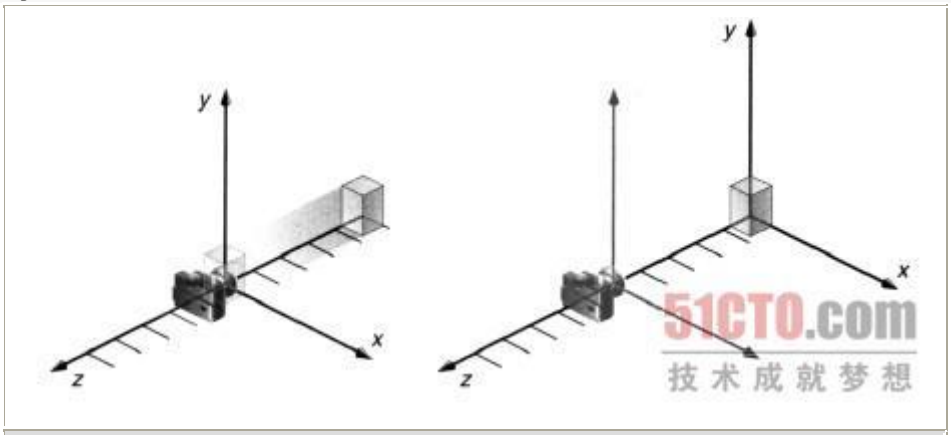


图 3-10 分离观察点和物体

这个函数在场景中把物体沿 z 轴移动-5 个单位，相当于把照相机沿 z 轴移动+5 个单位。

现在，假设我们想从侧面观察物体。在执行移动函数之前（或之后），是否应该调用一个旋转函数呢？如果根据全局固定坐标系统来考虑，首先想象物体和照相机都位于原点。首先应该旋转物体，然后把它移离照相机，以便看到我们想要看到的侧面。使用全局固定坐标系统方法，必须以相反的顺序调用变换函数，使它们按照预想的顺序产生效果。因此，需要在代码中首先调用移动函数，然后再调用旋转函数。

现在，让我们考虑局部坐标系统方法。在这种情况下，考虑把物体以及它的局部坐标系统移离原点。然后，根据经过移动的坐标系统调用旋转函数。按照这种方法，变换函数的调用顺序和它们的应用顺序相同，因此首先调用的仍然是移动函数。因此，为了产生所需的结果，需要使用的变换函数序列是：

```
1. glTranslatef(0.0, 0.0, -5.0);
2. glRotatef(90.0, 0.0, 1.0, 0.0);
```

如果对追踪连续的矩阵乘法的效果感到麻烦，可以试试同时使用全局固定坐标系统和局部坐标系统方法，看看哪种方法更为适合。注意，如果使用全局固定坐标系统，旋转总是根据全局原点进行的。但在局部坐标系统中，旋转是根据局部坐标系统的原点进行的。还可以使用下一节描述的工具函数 gluLookAt()。

使用工具函数 gluLookAt()

程序员常常在原点附近或者在其他方便的位置上构建场景，然后从合适的位置观察场景，以获得较好的观察效果。正如 gluLookAt()函数的名字所提示的那样，这个工具函数就用于完成上面这个任务。它接受 3 组参数，分别指定了观察点的位置，定义了照相机瞄准的参考

点，并且提示哪个方向是朝上的。选择观察点的目的是产生预期的场景视图。参考点一般位于场景的中间（如果是在原点创建场景，参考点很可能就是原点）。指定正确的朝上向量稍微有点难度。如果是在原点或它的附近创建一些现实世界的场景，而且把 y 轴的正方向表示为向上，那么它就是 gluLookAt()的朝上向量。但是，如果我们设计的是一个飞行模拟器，向上方向是垂直于机翼的方向。当飞机停在地面上时，就是从飞机指向天空。

如果想要进行全景扫描，gluLookAt()就显得特别有用。在一个沿 x 轴和 y 轴都对称的视景体中，（eyex, eyey, eyez）点总是位于场景中图像的中心。因此，可以使用一系列的命令稍微移动这个点，以实现全景扫描的效果。

```
1. void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez,
2. GLdouble centerx, GLdouble centery, GLdouble centerz,
3. GLdouble upx, GLdouble upy, GLdouble upz);
```

定义了一个视图矩阵，并把它与当前矩阵进行右乘。目标观察点是由 eyex、eyey 和 eyez 指定的。centerx、centery 和 centerz 参数指定了视线上的任意一点。upx、upy 和 upz 参数表示哪个方向是朝上的（也就是说，在视景体中自底向上的方向）。

在默认情况下，照相机位于原点，指向 z 轴的负方向，以 y 轴的正方向为朝上方向。这相当于调用：

```
1. gluLookAt(0.0, 0.0, 0.0, 0.0, 0.0, -100.0, 0.0, 1.0, 0.0);
```

3.2.3 视图变换（2）

参考点的 z 值是-100.0，但它也可以是任意的负值，因为它不会影响视线的方向。此时，不需要调用 gluLookAt()，因为它是默认的（参见图 3-11），我们已经实现了这个效果（从照相机延伸的直线表示视景体，也就是照相机的视野）。

图 3-12 显示了一个典型的 gluLookAt()调用的结果。照相机的位置（eyex, eyey, eyez）位于（4,2,1）。在这种情况下，照相机正对着模型，因此参考点是（2, 4, -3）。另外，这张图选择了一个方向向量（2, 2, -1），把观察点旋转 45 度。

因此，为了实现这个效果，可以调用：

```
1. gluLookAt(4.0, 2.0, 1.0, 2.0, 4.0, -3.0, 2.0, 2.0, -1.0);
```

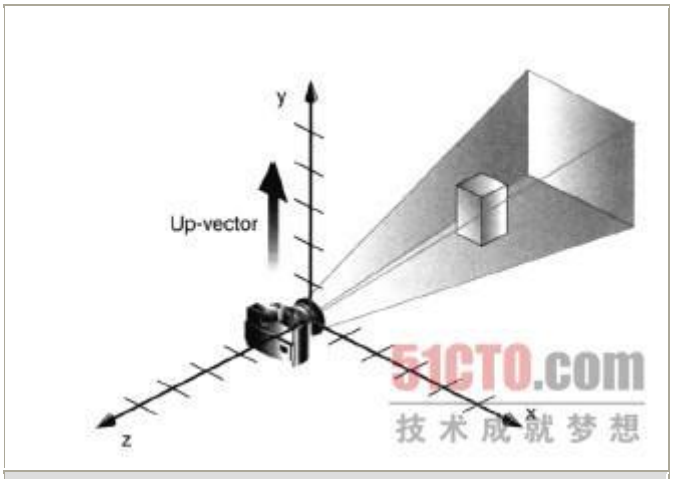


图 3-11 默认的照相机位置

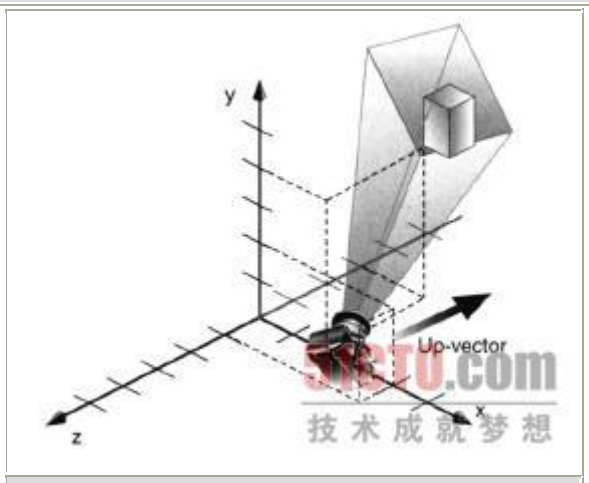


图 3-12 使用 gluLookAt()

注意，gluLookAt()是 OpenGL 工具函数库的一部分，而不是基本的 OpenGL 实用函数库的一部分。这并不是因为它用处不大， 而是因为它封装了一些基本的 OpenGL 函数（ 具体地说， 就是 glTranslate*()和 glRotate*()）。为了说明这一点，想象有一架照相机位于一个任意的观察点，并指向视线的方向（均在 gluLookAt()函数中设置），并且有一个场景位于原点。为了“还原”gluLookAt()所执行的操作，需要对照相机进行变换，使它位于原点，并指向 z 轴的负方向，也就是默认位置。为此，可以通过简单的移动操作把照相机移到原点，然后根据全局固定坐标系统的 3 个轴进行一系列的旋转，使照相机指向 z 轴的负方向。由于 OpenGL 允许沿任意轴旋转，因此使用一个 glRotate*()调用便可以实现所需要的任何旋转。

注意：在任何时候，只能有一个视图变换处于活动状态。我们不能组合两个视图变换的效果，就像不能把一架照相机固定在两个三角架上一样。如果想改变照相机的位置，要确保调用 glLoadIdentity()函数，消除任何当前视图变换的效果。

Nate Robin 的投影教程

如果读者已经下载了 Nate Robin 的教学程序包，现在就可以运行“projection”教程。在这个教程中，可以看到修改 glLookAt()函数的参数所产生的各种效果。

高级话题

为了变换一个任意的向量，使它与另一个任意的向量（例如 z 轴负方向）重合，需要进行一些数学计算。可以求出两个规范化向量的向量积，并把它作为旋转轴。为了计算旋转角度，需要对两个初始向量进行规范化。两个向量之间角度的余弦值相当于这两个向量经过规范化后的数量积。由向量积所给出的围绕旋转轴的旋转角度总是位于 0～180 度之间。关于向量的数量积和向量积的定义，请参阅附录 I（该附录可以通过 http://www.opengl-redbook.com/appendices/在线访问）。

注意，通过取两个经过规范化的向量的数量积的反余弦，来计算它们之间的角度所得到的结果并不十分精确，尤其是当这个角度非常小的时候。但是，对于刚刚起步的初学者而言，这种计算方法已经足够了。

创建自定义的工具函数

高级话题

在一些专业的应用程序中，我们可能想定义自己的变换函数。由于这种做法非常少见，而且属于相对高级的话题，还是把它作为练习留给读者。下面的练习建议了两种可能有用的自定义视图变换。

尝试一下

假定我们正在编写一个飞机模拟器，并且以飞机的驾驶员作为观察点显示飞机外面的景象。我们可以用一个原点位于跑道上的坐标系统来描述整个场景，飞机位于坐标（x, y, z）。然后，假设飞机还具有倾测角、螺旋角和航向改变角（这些都是飞机相对于它的重心的旋转角度）。

下面这个函数可以作用视图变换函数使用。

```
1. void pilotView{GLdouble planex, GLdouble planey,
2. GLdouble planez, GLdouble roll,
3. GLdouble pitch, GLdouble heading)
4. {
5. glRotated(roll, 0.0, 0.0, 1.0);
6. glRotated(pitch, 0.0, 1.0, 0.0);
7. glRotated(heading, 1.0, 0.0, 0.0);
8. glTranslated(-planex, -planey, -planez);
9. }
```

假设在应用程序中，照相机需要绕着一个位于原点的物体作轨道运动。在这种情况下，我们可能想用极坐标来指定视图变换，让 distance 变量定义轨道的半径（或照相机与原点的距离）。一开始，照相机沿 z 轴的正方向移动 distance 个单位。azimuth 描述了照相机在 xy 平面上围绕物体的旋转角度，这是从 y 轴的正方向开始测量的。类似地，elevation 是照相机在 yz 平面上绕物体旋转的角度，从 z 轴的正方向开始测量。最后，twist 表示视景体围绕它的视线的旋转角度。

下面这个函数可以作为视图变换函数使用。

```
1. void polarView{GLdouble distance, GLdouble twist,
2. GLdouble elevation, GLdouble azimuth)
3. {
4. glTranslated(0.0, 0.0, -distance);
5. glRotated(-twist, 0.0, 0.0, 1.0);
6. glRotated(-elevation, 1.0, 0.0, 0.0);
7. glRotated(azimuth, 0.0, 0.0, 1.0);
8. }
```

3.3 投影变换

前一节描述了如何合成预期的模型视图矩阵，以便应用正确的模型和视图变换。本节描述如何定义预期的投影矩阵。投影矩阵也用于对场景中的顶点进行变换。记住，在调用本节描述的任何变换函数之前，首先要调用下面的函数：

```
1. glMatrixMode(GL_PROJECTION);
2. glLoadIdentity();
```

这样，接下来的变换函数将影响投影矩阵，而不是模型变换矩阵，并避免产生复合的投影变换。由于每个投影变换函数都完整地描述了一个特定的变换，因此一般并不需要把投影变换与其他变换进行组合。

投影变换的目的是定义一个视景体。视景体有两种用途。首先，视景体决定了一个物体是如何映射到屏幕上的（即通过透视投影还是正投影）。其次，视影体定义了哪些物体（或物体的一部分）被裁剪到最终的图像之外。可以把原先一直讨论的观察点看成是视景体的一端。现在，可以回顾第 3.3.1 节，了解所有变换的简单介绍，其中包括投影变换。

3.3.1 透视投影

透视投影最显著的特征就是透视缩短，物体距离照相机越远，它在最终图像中看上去就越小。这是因为透视投影的视景体可以看成是一个金字塔的平截头体（顶部被一个平行于底面的平面截除）。位于视景体之内的物体被投影到金字塔的顶点，也就是照相机或观察点的位置。靠近观察点的物体看上去更大一些，因为和远处的物体相比，它们占据了视景体中相

对较大的区域。这种投影方法常用于动画、视觉模拟以及其他要求某种程度的现实感的应用领域，因为它和我们在日常生活中观察事物的方式相同。

glFrustum()函数定义了一个平截头体，它计算一个用于实现透视投影的矩阵，并把它与当前的投影矩阵（一般为单位矩阵）相乘。记住，视景体用于裁剪那些位于它之外的物体。平截头体的 4 个侧面、顶面和底面对应于视景体的 6 个裁剪平面，如图 3-13 所示。位于这些平面之外的物体(或物体的一部分)将裁剪掉，不会出现在最终的图像中。注意 glFrustum()函数并不需要定义一个对称的视景体。

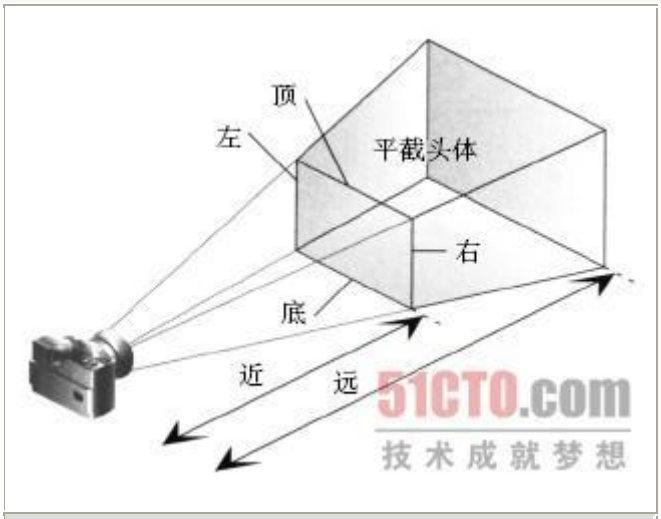


图 3-13 glFrustum()函数所指定的透视视景体

```
1. void glFrustum(GLdouble left, GLdouble right,
2. GLdouble bottom, GLdouble top,
3. GLdouble near, GLdouble far);
```

创建一个表示透视视图平截头体的矩阵，并把它与当前矩阵相乘。平截头体的视景体是由这个函数的参数定义的：（left, bottom, -near）和（right, top, -near）分别指定了近侧裁剪平面左上角和右下角的（x, y, z）坐标。near 和 far 分别表示从观察点到近侧和远侧裁剪平面的距离，它们的值都应该是正的。

平截头体在三维空间中有一个默认的方向。可以在投影矩阵上执行旋转或移动，对这个方向进行修改。但是，这种做法难度较大，因此最好还是避免。

高级话题

平截头体并不一定要求是对称的， 它的轴也并不需要与 z 轴对齐。例如， 可以使用 glFrustum()函数绘制一幅图片，就像透过房子右上角的一个矩阵窗口向外观察一样。摄像师使用这种视景体创建人工透视效果。通过这种方法，可以让硬件按照两倍于常规的分辨率计算图像，供打印机使用。例如，如果想让一幅图像的分辨率两倍于屏幕的分辨率，可以分 4 次绘制同一幅图像，每次使用平截头体用四分之一的图像覆盖整个屏幕。在图像的每个四分

之一均被渲染之后，就可以读取像素，以收集高分辨率图像的数据（关于读取像素数据的更多信息，请参阅第 8 章）。

尽管在概念上理解起来非常轻松，但是 glFrustum()函数用起来并不是非常直观。因此，可以试用 OpenGL 工具函数库的 gluPerspective()函数。这个函数创建一个视景体，它与调用 glFrustum()产生的视景体相同，可以用一种不同的方式来指定它。这个函数并不是指定近侧裁剪平面的两个角，而是指定 y 方向上视野的角度（见图 3-14）和纵横比（x/y）。对于正方形的屏幕，纵横比为 1.0。这两个参数足以确定沿视线方向的平截头体金字塔，如图 3-14 所示。还需要指定观察点和近侧以及远侧裁剪平面的距离，也就是对这个金字塔进行截除。注意，gluPerspective()仅限于创建沿视线方向同时对称于 x 轴和 y 轴的平截头体，但是我们通常所需要的就是这种平截头体。

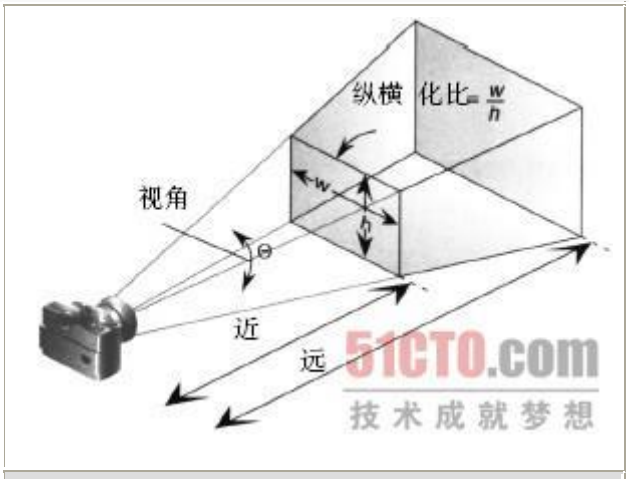


图 3-14 由 gluPerspective()所指定的透视视景体

和 glFrustum()函数一样，可以执行旋转或移动，改变 gluPerspective()创建的视景体的默认方向。如果不执行这样的变换，观察点就位于原点，视线的方向是沿 z 轴的负方向。

使用 gluPerspective()时，需要挑选正确的视野值，否则图像看上去就会变形。为了获得完美的视野，可以推测自己的眼睛在正常情况下距离屏幕有多远以及窗口有多大，并根据距离和大小计算视野的角度。计算结果可能比自己想象的要小。我们也可以换一种方法考虑这个问题。一个 35mm 的照相机如果要达到 94 度的视野，它的镜头就要求达到 20mm，这已经是非常宽的镜头了（关于如何计算视野的详细信息，请参阅第 3.5 节）。

```
1. void gluPerspective(GLdouble fovy, GLdouble aspect,
2. GLdouble near, GLdouble far);
```

创建一个表示对称透视视图平截头体的矩阵，并把它与当前矩阵相乘。`fovy` 是 `yz` 平面上视野的角度，它的值必须在`[0.0, 180.0]`的范围之内。`aspect` 是这个平截头体的纵横比，也就是它的宽度除以高度。`near` 和 `far` 值分别是观察点与近侧裁剪平面以及远侧裁剪平面的距离（沿 z 轴负方向），这两个值都应该是正的。

前面一段内容提到了英寸和毫米这些单位，它们真的和 OpenGL 有关的吗？答案是否定的。投影和其他变换在本质上是没有单位的。如果我们想把近侧和远侧裁剪平面看成是位于 1.0 和 20.0m（或英寸、km 等其他长度单位），那也没有关系。唯一的规则是必须使用一致的测量单位。最终绘制好的图像将会进行缩放。

3.3.2 正投影

在正投影下，视景体是一个平行的长方体。用更通俗的话说，是一个箱子（如图 3-15 所示）。和透视投影不同，视景体两端的大小并没有不同。也就是说，物体和照相机之间的距离并不影响它看上去的大小。这种类型的投影常用于建筑蓝图和计算机辅助设计的应用程序。在这类应用程序中，当物体经过投影之后，保持它们的实际大小以及它们之间的角度是至关重要的。

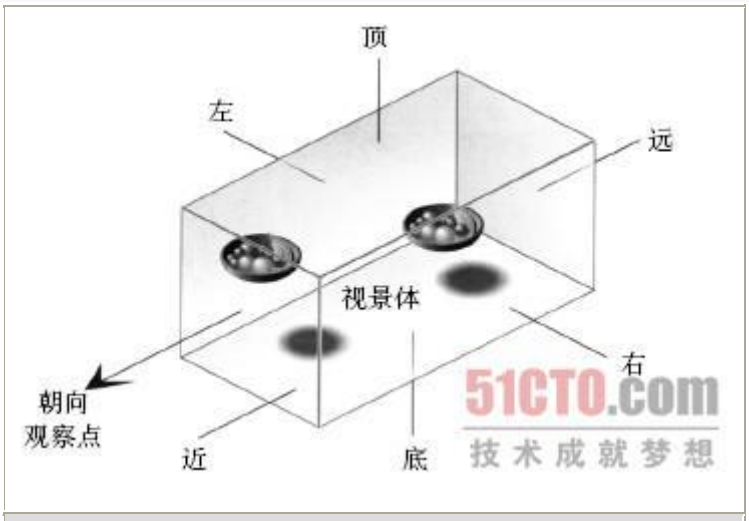


图 3-15 正投影下的视景体

glOrtho()函数创建一个正交平行的视景体。和 glFrustum()一样，需要在参数中指定近侧裁剪平面的各个角以及它与远侧裁剪平面的距离。

如果没有其他变换，投影的方向就与 z 轴平行，观察点的方向直接朝向 z 轴的负方向。

```
1. void glOrtho(GLdouble left, GLdouble right,
2. GLdouble bottom, GLdouble top,
3. GLdouble near, GLdouble far);
```

创建了一个表示正交平行视景体的矩阵，并把它与当前矩阵相乘。（left,bottom, -near）和（right, top, -near）是近侧裁剪平面上的点，分别映射到视口窗口的左下角和右上角。（left, bottom, -far）和（right, top, -far）是远侧裁剪平面上的点，分别映射到视口窗口的左下角和右上角。near 和 far 可以是正值或负值，甚至可以设置为 0。但是，near 和 far 不应该取相同的值。

如果是把二维图像投影到二维屏幕这种特殊情况，可以使用 OpenGL 工具函数库中 gluOrtho2D()函数。这个函数和它的三维版本 glOrtho()基本相同，只是场景中物体的所有 z 坐标都假定位于-1.0～1.0 之间。如果使用二维版本的顶点函数绘制二维物体，所有的 z 坐标都是 0。因此，不会有物体因为它的 z 值而裁剪掉。

```
1. void gluOrtho2D(GLdouble left, GLdouble right,
2. GLdouble bottom, GLdouble top);
```

创建一个表示把二维坐标投影到屏幕上的矩阵，并把当前矩阵与它相乘。裁剪区域为矩形，它的左下角坐标为（left, bottom），右上角坐标为（right, top）。

Nate Robin 的投影教程

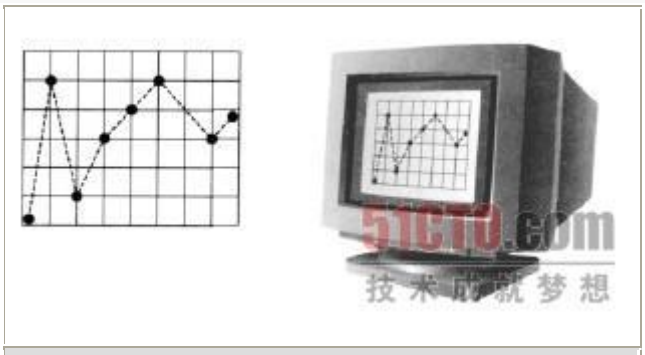
如果已经下载了 Nate Robin 的教学程序包，现在可以再次运行“projection”教程。这次，可以对 gluPerspective()、glOrtho()和 glFurstum()函数的参数进行试验。

3.3.3 视景体裁剪

当场景中物体的顶点通过模型视图矩阵和投影矩阵进行变换之后，位于视景体之外的所有图元都将裁剪掉。6 个裁剪平面就是定义视景体 6 个侧面的平面。还可以指定其他裁剪平面，使它们位于我们所需要的地方（关于这个相对较为高级的语题，可以参阅第 3.7 节）。记住，OpenGL 将会重新构建被裁剪的多边形的边。

3.4 视口变换

现在让我们回顾那个照相机比喻。视口变换对应于选择被冲洗相片的大小这个阶段。我们希望照片像钱包一样大还是像海报一样大？在计算机图形中，视口是一个矩形的窗口区域，图像就是在这个区域中绘制的。图 3-16 显示了一个占据屏幕绝大部分区域的视口。视口是用窗口坐标测量的。窗口坐标反映了屏幕上的像素相对于窗口左下角的位置。记住，此时所有的顶点都已经根据模型视图矩阵和投影矩阵进行了变换，那些位于视景体之外的顶点都已经裁剪掉了。



（点击查看大图）

3.4.1 定义视口

在屏幕上打开窗口的任务是由窗口系统而不是 OpenGL 负责的。但是，在默认情况下，视口被设置为占据打开窗口的整个像素矩形。可以使用 glViewPort()函数选择一个更小的绘图区域。例如，可以对窗口进行划分，在同一个窗口中显示分割屏幕的效果，以显示多个视图。

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

在窗口中定义一个像素矩形，最终的图像将映射到这个矩形中。（x, y）参数指定了视口的左下角，width 和 height 表示这个视口矩形的宽度和高度。在默认情况下，视口的初始值是（0, 0,winWidth,winHeight），其中 winWidth 和 winHeight 指定了窗口的大小。

视口的纵横比一般和视景体的纵横比相同。如果这两个纵横比不同，当图像投影到视口时就会变形，如图 3-17 所示。注意，以后窗口的大小如果发生了变化，并不会自动影响视口，应用程序应该检测窗口大小改变事件，以便相应地修改视口的大小。

在图 3-17 中，左图显示了一幅正方形的图像投影到一个正方形的视口中，它所使用的函数如下：

```
1. gluPerspective(fovy, 1.0, near, far);
2. glViewport(0, 0, 400, 400);
```

但是，在图 3-17 的右图中，窗口的大小发生了变化，成了一个非正方形的矩形视口，而投影并没有发生变化。于是，图像沿 x 轴被压缩：

```
1. gluPerspective(fovy, 1.0, near, far);
2. glViewport(0, 0, 400, 200);
```

为了避免这种变形，可以修改投影的纵横比，使它与视口相匹配：

```
1. gluPerspective(fovy, 2.0, near, far);
2. glViewport(0, 0, 400, 200);
```

图 3-17 把视景体映射到视口

尝试一下

对原先的程序进行修改，用不同的视口两次绘制同一个物体。在每个视口中，可以使用不同的投影（或模型视图）变换来绘制这个物体。为了创建两个并排的视口，可以使用下面这些函数，并辅以适当的模型、视图和投影变换：

```
1. glViewport(0, 0, sizex/2, sizey);
2. .
3. .
4. .
5. glViewport(sizex/2, 0, sizex/2, sizey);
```

3.4.2 变换深度坐标

深度坐标是在视口变换期间进行编码的（ 以后存储在深度缓冲区中）。可以使用glDepthRange()函数，对 z 值进行缩放，使它位于我们所需要的范围之内（有关深度缓冲区以及对应的深度坐标的用途，请参阅第 10 章）。与 x 和 y 窗口坐标不同，在 OpenGL 中，z 坐标总是被认为位于 0.0～1.0 的范围之间。

```
1. void glDepthRange(GLclampd near, GLclampd far);
```

为 z 坐标定义了一种编码形式，它是在视口变换期间执行的。near 和 far 值表示经过调整后可以存储在深度缓冲区中的最小值和最大值。在默认情况下，它们分别是 0.0 和 1.0，适用于绝大多数应用程序。这两个参数的范围被限定在[0, 1]之间。

在透视投影中，变换后的深度坐标（和 x 坐标及 y 坐标一样）也进行了透视除法（除以 w 坐标）。当变换后的深度坐标远离近侧平面时，它的位置就逐渐变得不太精确（如图 3-18 所示）。

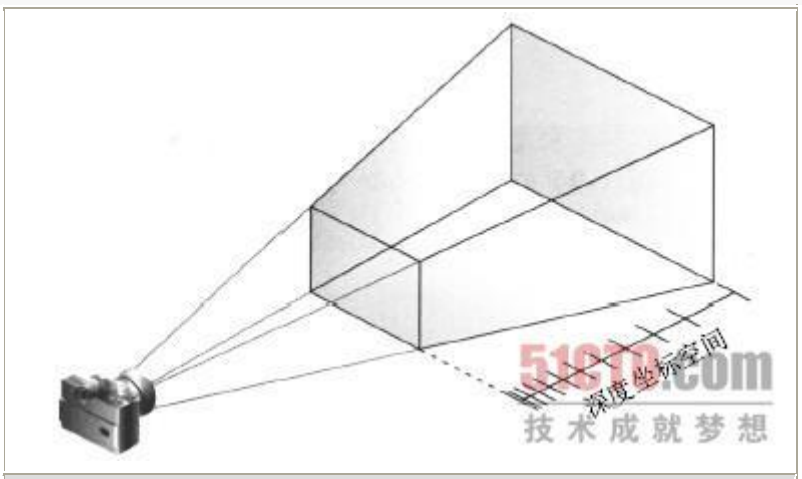


图 3-18 透视投影和变换后的深度坐标

因此，透视除法将会影响那些依赖经过变换的深度坐标的操作的精度，尤其是在双缓冲模式下用于隐藏表面消除的时候。

3.5 和变换相关的故障排除

让照相机指向正确的方向是非常容易的。但是，在计算机图形中，必须使用坐标和角度指定位置和方向。我们可以证明，实现众所周知的黑屏效果简直太容易了。导致出错的原因可能有很多，我们常常可以发现这样的效果，在屏幕上打开的窗口中没有绘制任何东西，就像是因为照相机没有对准位置或者照相机对准的方向根本没有东西。如果选择的视野不够宽，无法看到整个物体，而是被物体的某个部分完全占据了视野，也会出现这种问题。

如果觉得已经尽了很大的努力，结果得到的仍然只是一个黑色的窗口，可以试试下面这几个诊断步骤：

1) 检查一些显而易见的原因。系统是否已经插上电？绘图颜色是否和用来清除背景的颜色相同？程序所使用的各种状态（例如光照、纹理、Alpha 混合、逻辑操作或抗锯齿）是否已经正确地打开或关闭？

2) 使用投影函数时，记住近侧和远侧裁剪平面是根据它们在 z 轴的负方向（默认情况下）上和观察点的距离进行测量的。因此，如果近侧平面的距离是 1.0，远侧平面的距离是 3.0，在场景中可见的所有物体的 z 坐标必须在-1.0～-3.0 之间。为了确保不会裁剪掉任何物体，可以暂时把近侧平面和远侧平面设置为非常大的范围，例如 0.001 和 1 000 000.0。虽然，这种做法会对深度缓冲和雾效果等操作造成负面影响，但它可以发现那些被不小心裁剪掉的物体。

3) 确定观察点的位置和方向，以及物体的位置。这些参数有助于创建一个真正的三维空间（例如，借助双手想象所有物体在三维空间中的位置）。

4) 确定物体是绕什么旋转的。首先要把物体移回到原点，不然它很可能绕任意点旋转。当然，绕任意点进行旋转也是可以的，除非明确指定了必须绕原点旋转。

5) 检查瞄向。使用 gluLookAt()函数使视景体对准物体，或者在靠近原点的地方进行绘图，并使用 glTranslate*()函数进行一次视图变换，把照相机移动到 z 轴上适当的位置，使物体位于视景体中。一旦已经在视景体中看到了物体，可以慢慢地改变视景体，以获取所需要的效果。我们将在稍后描述这个过程。

6) 进行透视投影变换时，确信近侧裁剪平面不要太靠近观察者（照相机），不然会对深度缓冲区
的准确性产生不利影响。

即使已经让照相机对准了正确的方向，并且可以看到物体，但是它们仍然可能显得太大或太小。如果使用的是 gluPerspective()，可能需要修改用于定义视野的角度。这可以通过修改这个函数的第一个参数来实现。可以使用三角函数，根据物体的大小以及它与观察点的距离来计算所需的视野：视野角度的一半的正切就是这个物体的大小除以它与观察点距离的一

半（如图 3-19 所示）。因此，我们可以使用一个反正切函数来计算所需角度的一半。示例程序 3-3 使用了一个三角函数 atan2()，它根据一个直角三角形的对边和邻边的长度计算反正切值。随后，这个计算结果需要从弧度变换为角度。

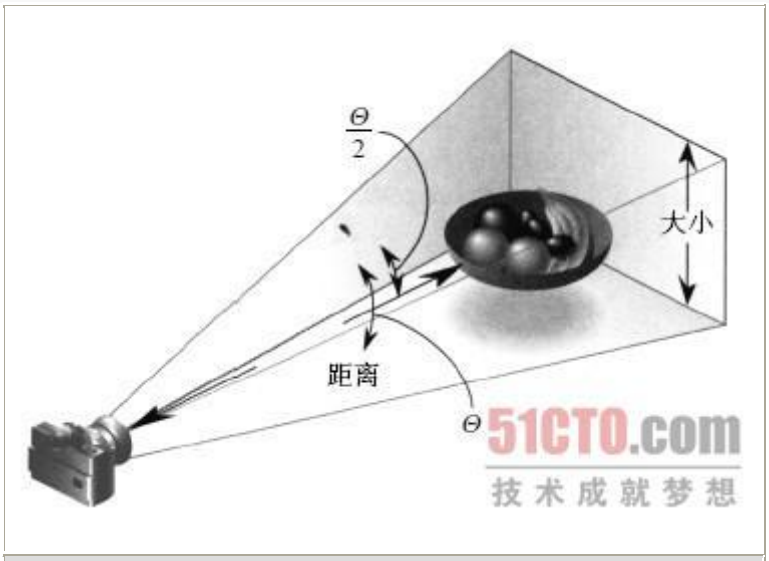


图 3-19 使用三角函数计算视野

示例程序 3-3 计算视野

```
1. #define PI 3.1415926535
2. double calculateAngle(double size,double distance)
3. {
4. double radtheta,degtheta;
5. radtheta =2.0 *atan2 (size/2.0,distance);
6. degtheta =(180.0 *radtheta)/PI;
7. return degtheta;
8. }
```

当然，在一般情况下，我们并不知道物体的准确大小，而只知道场景中的一个点和观察点之间的距离。为了获得一个相对合理的近似值，可以通过确定场景中所有物体的最大和最小 x、y 和 z 坐标值，判断包含整个场景的边界长方体。然后，计算这个长方体的外接球体的半径，并根据这个球体的中心来判断物体和观察点的距离，根据这个球体的半径来确定物体的大小。

例如，假设物体的所有坐标都满足方程式-1≤x≤3、5≤y≤7 和-5≤z≤5，这个边界长方体的中心便位于（1，6，0），外接球体的半径就是长方体的中心到任意一个角的距离。假设这个角为（3，7，5），那么距离就是：

$$\sqrt{(3-1)^2+(7-6)^2+(5-0)^2}=\sqrt{30}=5.477$$

如果观察点位于（8，9，10），它和中心的距离就是：

$$\sqrt{(8-1)^2+(9-6)^2+(10-0)^2}=\sqrt{158}=12.570$$

半角的正切就是 5.477 除以 12.570，也就是 0.4357。因此，半角的度数为 23.54（即视角大约为 47 度）。

为了实现逼真的图像，需要记住视野的角度将会影响观察点的最佳位置。例如，经过计算得到的视野角度为 179 度，为了获得逼真感，观察点和屏幕的距离必须小于 1 英寸（2.54cm）。如果计算出来的视野角度太大，可能需要移动观察点，使之远离物体。

3.6 操纵矩阵堆栈

模型视图矩阵和投影矩阵的创建、加载和乘法只是“冰山露出水面的一角”。当我们对这些矩阵执行操作时，每一个矩阵实际上是各自矩阵堆栈最顶部的那个元素（如图 3-20 所示）。

图 3-20 模型视图和投影矩阵堆栈

矩阵堆栈适用于创建层次式的模型，也就是通过简单的模型构建复杂的模型。例如，假如绘制的是一辆具有 4 个轮子的汽车，每个轮子用 5 颗螺钉固定到汽车上。由于所有的轮子都是相同的，所有的螺钉看上去也没什么区别，因此可以用一个函数绘制轮子，用另一个函数绘制螺钉。这两个函数在适当的位置和方向绘制一个轮子或一颗螺钉，例如，它们的中心在原点，并且它们的轴与 z 轴对齐。在绘制这辆包括了轮子和螺钉的汽车时，需要 4 次调用画轮子的函数，每次都使用不同的变换，使每个轮子处于正确的位置。在绘制每个轮子时，需要 5 次调用画螺钉的函数，每次都要相对于轮子的位置进行适当的变换。

假设只需要绘制车身和轮子，下面这段话描述了需要做的事情：

绘制车身。记住自己的位置，并移动到右前轮的位置。绘制轮子，并丢弃上一次所进行的变换（即移动到右前轮的位置），使自己回到车身的原点位置。记住自己的位置，然后移动到左前轮..

类似地，对于每个轮子，我们需要绘制轮子，记住自己的位置，然后移动到绘制螺钉的每个位置，在画完每个螺钉之后丢弃上一次进行的变换。

由于变换是以矩阵的形式存储的，因此矩阵堆栈就是一种理想的机制，可以用来完成这种类型的记忆、移动和丢弃操作。到目前为止所描述的所有矩阵操作（glLoadMatrix()、glLoadTransposeMatrix()、glMultMatrix()、glMultTransposeMatrix()、glLoadIndentity()以及用于创建特定的变换矩阵的函数）都是对当前矩阵（堆栈顶部的那个矩阵）进行处理。可以用执行堆栈操作的函数 glPushMatrix()和 glPopMatrix()来控制堆栈顶部的矩阵。glPushMatrix()复制一份当前矩阵，并把这份复制添加到堆栈的顶部。glPopMatrix()丢弃堆栈顶部的那个矩阵。图 3-21 显示了这两种操作。记住，当前矩阵就是位于堆栈顶部的矩阵。事实上，glPushMatrix()表示“记住自己的位置”，glPopMatrix()表示“回到原来的位置”。

图 3-21 在矩阵堆栈中进行压入和弹出

```
1. void glPushMatrix(void);
```

把当前堆栈中的所有矩阵都下压一级。当前矩阵堆栈是由 glMatrixMode()函数指定的。这个函数复制当前的顶部矩阵，并把它压到堆栈中。因此，堆栈最顶部的两个矩阵的内容相同。如果压入的矩阵太多，这个函数会导致错误。

```
1. void glPopMatrix(void);
```

把堆栈顶部的那个矩阵弹出堆栈，销毁被弹出矩阵的内容。堆栈原先的第二个矩阵成为顶部矩阵。当前堆栈是由 glMatrixMode()函数指定的。如果堆栈只包含了一个矩阵，调用 glPopMatrix()将会导致错误。示例程序 3-4 绘制了一辆汽车，假设已经存在用于绘制车身、轮子和螺钉的相应函数。

示例程序 3-4 压入和弹出矩阵

```
1. draw_wheel_and_bolts()
2. {
3. int i;
4. draw_wheel();
5. for(i=0;i<5;i++){
```

```
6.  glPushMatrix();
7.  glRotatef(72.0*i,0.0,0.0,1.0);
8.  glTranslatef(3.0,0.0,0.0);
9.  draw_bolt();
10. glPopMatrix();
11. }
12. }
13. draw_body_and_wheel_and_bolts()
14. {
15. draw_car_body();
16. glPushMatrix();
17. glTranslatef(40,0,30); /*move to first wheel position*/
18. draw_wheel_and_bolts();
19. glPopMatrix();
20. glPushMatrix();
21. glTranslatef(40,0,-30); /*move to 2nd wheel position*/
22. draw_wheel_and_bolts();
23. glPopMatrix();
24. ..
25. /*draw last two wheels similarly*/
26. }
```

这段代码假定轮子和螺钉的轴都与 z 轴对齐，固定每个轮子的 5 颗螺钉均匀地分布，每隔 72 度一颗，并且与轮子中心的距离都为 3 个单位（例如英寸）。两个前轮的位置是车身原点向前 40 个单位，向左右两侧分别距离 30 个单位。

使用矩阵堆栈的效率要高于使用单独的堆栈，尤其是堆栈是用硬件实现时。压入一个矩阵时，并不需要把当前矩阵复制到主进程，并且硬件有可能一次能够复制多个矩阵元素。有时候，我们可能想在矩阵底部保存一个单位矩阵，以避免重复调用 glLoadIdentity()。

3.6.1 模型视图矩阵堆栈

在前面第 3.2 节中，我们解释了模型视图矩阵是视图变换矩阵与模型变换矩阵相乘的结果。每个视图或模型变换都创建了一个新的矩阵，并与当前的模型视图矩阵相乘，其结果成为新的当前矩阵，表示组合后的变换。模型视图矩阵堆栈至少可以包含 32 个 4×4 的矩阵。模型视图矩阵一开始的顶部矩阵是单位矩阵。有些 OpenGL 实现支持在矩阵堆栈上保存超过 32 个的矩阵。为了确定矩阵堆栈的最大允许矩阵数，可以使用查询函数 glGetIntegerv(GL_MAX_MODELVIEW_STACK_DEPTH, GLint *params)。

3.6.2 投影矩阵堆栈

投影矩阵包含了一个表示投影变换的矩阵，它描述了视景体。一般情况下，并不需要对投影矩阵进行组合，因此在执行投影变换之前需要调用 glLoadIdentity()函数。另外，由于这个原因，投影堆栈的深度一般只需要两层。有些 OpenGL 实现可能允许在投影矩阵堆栈中存储超过 2 个的 4×4 矩阵。为了获取投影堆栈的深度，可以调用 glGetIntegerv(GL_MAX_PROJECTION_STACK_DEPTH, GLint *params)。在投影矩阵堆栈中保存第二个矩阵有什么用途呢？有些应用程序除了显示三维场景的主窗口之外，还需要显示一个包含文本的帮助窗口。由于文本在正投影模式下比较容易定位，因此当我们需要显示帮助窗口时，可以暂时把投影模型设置为正投影，显示这个帮助窗口，然后再返回到原来的透视投影模式。

```
1. glMatrixMode(GL_PROJECTION);
2. glPushMatrix(); /*save the current projection*/
3. glLoadIdentity();
4. glOrtho(...); /*set up for displaying help*/
5. display_the_help();
6. glPopMatrix();
```

注意，当修改投影模式时，很可能需要对模型视图矩阵进行相应的修改。

高级话题

如果读者的数学知识足够丰富，可以创建自定义的投影矩阵，执行任意的投影变换。例如，OpenGL 和它的工具函数库并没有提供内置的机制，对两点透视（two-point perspective）提供支持。如果想模拟手写文本的绘图方式，就可能需要一个像这样的投影矩阵。

3.7 其他裁剪平面

除了视景体的 6 个裁剪平面（左、右、底、顶、近和远）之外，还可以另外再指定最多可达 6 个的其他裁剪平面，对视景体施加进一步的限制，如图 3-22 所示。这些裁剪平面可以用于删除场景中的无关物体。例如，我们可能只想显示一个物体的剖面视图。

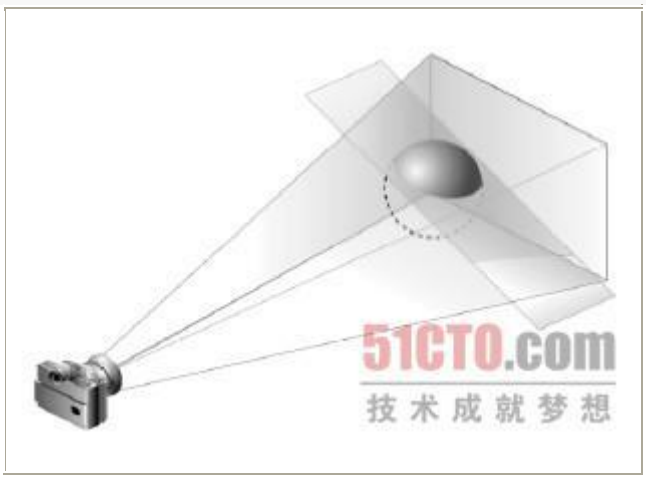


图 3-22 其他裁剪平面和视景体

每个平面都是由它的方程式 Ax + By + Cz + D= 0 的系数所指定的。裁剪平面会根据模型和视图矩阵自动执行适当的变换。最终的裁剪区域将是视景体与其他裁剪平面定义的所有半空间的交集。记住，OpenGL 会自动对部分被裁剪的多边形的边进行正确的重构。

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

定义一个裁剪平面。equation 参数指向平面方程 Ax + By + Cz + D = 0 的 4 个系数。满足（A B C D）M-1 (xe ye ze we) T≥0 的所有视觉坐标（xe ye ze we）点都位于这个平面定义的半空间中，其中 M 是在调用 glClipPlane()时的当前模型视图矩阵。所有不是位于这个半空间内的点都将裁剪掉。plane 参数是 GL_CLIP_PLANEi，其中 i 是一个整数，表示需要定义哪个有效裁剪平面。i 的值位于 0 和最大其他裁剪平面数减 1 之间。

我们需要启用每个被定义的裁剪平面：

```
1. glEnable(GL_CLIP_PLANEi);
```

也可以用下面这个函数禁用一个裁剪平面：

```
1. glDisable(GL_CLIP_PLANEi);
```

所有的 OpenGL 实现都必须支持至少 6 个其他裁剪平面，有些实现可能允许超过 6 个的其他裁剪平面。可以用 GL_MAX_CLIP_PLANES 为参数调用 glGetIntegerv()函数，查询自己使用的 OpenGL 实现所支持的其他裁剪平面的最大数量。

注意：调用 glClipPlane()函数所执行的裁剪是在视觉坐标中完成的，而不是在裁剪坐标中进行的。如果投影矩阵为奇异矩阵（也就是把三维坐标压平到二维坐标的真正投影矩阵），这个区别就非常大。在视觉坐标中进行裁剪时，即使投影矩阵是奇异矩阵，裁剪仍然是在三维空间中进行的。

裁剪平面的代码例子

示例 3-5 是经过两个裁剪平面裁剪的线框球体，裁去了 3/4 体积，如图 3-23 所示。

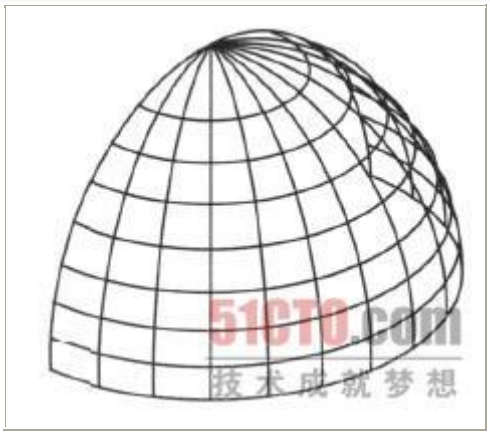


图 3-23 裁剪后的线框球体

示例程序 3-5 经过两个裁剪平面裁剪的线框球体：clip.c

```
1. void init(void)
2. {
3. glClearColor(0.0,0.0,0.0,0.0);
4. glShadeModel(GL_FLAT);
5. }
6. void display(void)
7. {
8. GLdouble eqn [4]={0.0,1.0,0.0,0.0};
9. GLdouble eqn2 [4] ={1.0,0.0,0.0,0.0};
10.glClear(GL_COLOR_BUFFER_BIT);
11.glColor3f(1.0,1.0,1.0);
12.glPushMatrix();
13.glTranslatef(0.0,0.0,-5.0);
14./* clip lower half --y <0 */
15.glClipPlane(GL_CLIP_PLANE0,eqn);
16.glEnable(GL_CLIP_PLANE0);
17./* clip left half --x <0 */
18.glClipPlane(GL_CLIP_PLANE1,eqn2);
19.glEnable(GL_CLIP_PLANE1);
20.glRotatef(90.0,1.0,0.0,0.0);
21.
22.glutWireSphere(1.0,20,16);
23.glPopMatrix();
24.glFlush();
25.}
26.void reshape(int w,int h)
27.{
28.glViewport(0,0,(GLsizei)w,(GLsizei)h);
29.glMatrixMode(GL_PROJECTION);
30.glLoadIdentity();
31.gluPerspective(60.0,(GLfloat)w/(GLfloat)h,1.0,20.0);
32.glMatrixMode(GL_MODELVIEW);
33.}
34.int main(int argc,char**argv)
35.{
```

```
36. glutInit(&argc,argv);
37. glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB);
38. glutInitWindowSize(500,500);
39. glutInitWindowPosition(100,100);
40. glutCreateWindow(argv [0]);
41. init();
42. glutDisplayFunc(display);
43. glutReshapeFunc(reshape);
44. glutMainLoop();
45. return 0;
46. }
```

尝试一下

修改示例程序 3-5 中描述裁剪平面的系数。

调用一个模型变换函数（如 glRotate*()）对 glClipPlane()产生影响，使裁剪平面在场景中的移动与物体无关。

3.8 一些组合变换的例子

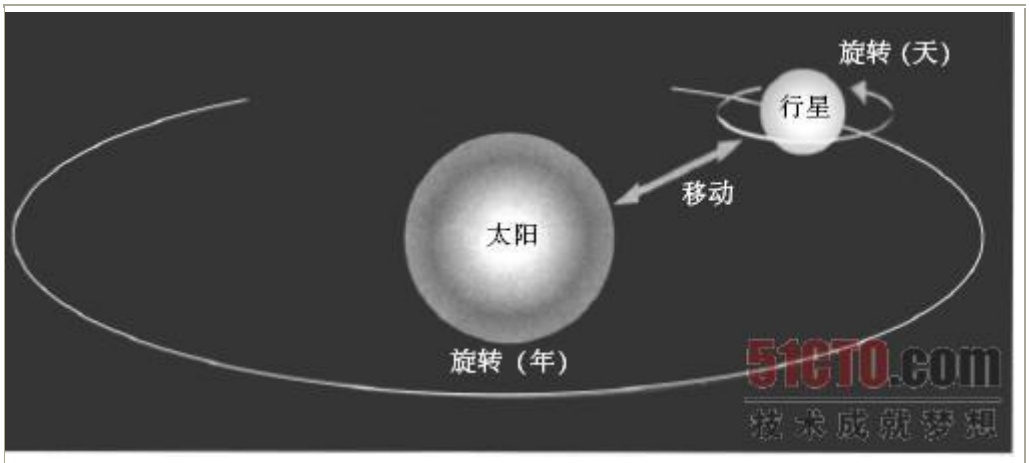
本节演示如何组合几个变换实现特定的效果。本节讨论的两个例子分别是太阳系和机器人手臂。在太阳系中，物体不仅需要绕自身的轴旋转，而且相互之间需要作轨道运动。机器人手臂具有几个关节，当它们相对其他关节移动时，坐标系统也要进行变换。

3.8.1 创建太阳系模型

本节描述的这个程序绘制了一个简单的太阳系，其中有一颗行星和一颗太阳，它们是用同一个函数绘制的。为了编写这个程序，需要使用 glRotate*()函数让这颗行星绕太阳旋转，并且绕自身的轴旋转。还需要使用 glTr a n s l a t e * ( )函数让这颗行星离开原点，移动到它自己的轨道上。可以在 glutWireSphere()函数中使用适当的参数，在绘制球体时指定球体的大小。

为了绘制这个太阳系，首先需要设置一个投影变换和一个视图变换。在这个例子中，可以使用 gluPerspective()和 gluLookAt()。

绘制太阳比较简单，因为它应该位于全局固定坐标系统的原点，也就是球体函数绘图的默认位置。因此，绘制太阳时并不需要移动，可以使用 glRotate*()函数使太阳绕一个任意的轴旋转。绘制一颗绕太阳旋转的行星要求进行几次模型变换，如图 3-24 所示。这颗行星每天需要绕自己的轴旋转一周，每年沿着自己的轨道绕太阳旋转一周。



（点击查看大图）图 3-24 行星和太阳

为了确定模型变换的顺序，可以从局部坐标系统的角度进行考虑。首先，调用 glRotate*()函数对局部坐标系统进行旋转，这个局部坐标系统一开始与全局固定坐标系统是一致的。接着，调用 glTranslate*()把局部坐标系统移动到行星轨道上的一个位置。移动的距离应该等于轨道的半径。因此，第一个 glRotate*()函数实际上确定了这颗行星从什么地方开始绕太阳旋转（或者说，确定了刚开始时是一年的什么时候）。

第二次调用 glRotate*()使局部坐标系统沿局部坐标系统的轴进行旋转，因此确定了这颗行星在一天中的时间。在调用了这些变换函数之后，就可以绘制这颗行星了。

下面简要总结了绘制太阳和行星需要使用的函数，完整的程序见示例程序 3-6。

```
1. glPushMatrix();
2. glutWireSphere(1.0, 20, 16); /* draw sun */
3. glRotatef((GLfloat) year, 0.0, 1.0, 0.0);
4. glTranslatef(2.0, 0.0, 0.0);
5. glRotatef((GLfloat) day, 0.0, 1.0, 0.0);
6. glutWireSphere(0.2, 10, 8); /* draw smaller planet */
7. glPopMatrix();
```

示例程序 3-6 行星系统：planet.c

```
1. static int year =0,day =0;
2. void init(void)
3. {
4. glClearColor(0.0,0.0,0.0,0.0);
5. glShadeModel(GL_FLAT);
6. }
7. void display(void)
```

```
8. {
9. glClear(GL_COLOR_BUFFER_BIT);
10. glColor3f(1.0,1.0,1.0);
11. glPushMatrix();
12. glutWireSphere(1.0,20,16); /*draw sun */
13. glRotatef((GLfloat)year,0.0,1.0,0.0);
14.
15. glTranslatef(2.0,0.0,0.0);
16. glRotatef((GLfloat)day,0.0,1.0,0.0);
17. glutWireSphere(0.2,10,8); /*draw smaller planet */
18. glPopMatrix();
19. glutSwapBuffers();
20. }
21. void reshape(int w,int h)
22. {
23. glViewport(0,0,(GLsizei)w,(GLsizei)h);
24. glMatrixMode(GL_PROJECTION);
25. glLoadIdentity();
26. gluPerspective(60.0,(GLfloat)w/(GLfloat)h,1.0,20.0);
27. glMatrixMode(GL_MODELVIEW);
28. glLoadIdentity();
29. gluLookAt(0.0,0.0,5.0,0.0,0.0,0.0,0.0,1.0,0.0);
30. }
31. void keyboard(unsigned char key,int x,int y)
32. {
33. switch (key){
34. case 'd ':
35. day =(day +10)%360;
36. glutPostRedisplay();
37. break;
38. case 'D ':
39. day =(day -10)%360;
40. glutPostRedisplay();
41. break;
42. case 'y ':
43. year =(year +5)%360;
44. glutPostRedisplay();
45. break;
```

```
46. case 'Y ':
47. year =(year -5)%360;
48. glutPostRedisplay();
49. break;
50. default:
51. break;
52. }
53. }
54. int main(int argc,char**argv)
55. {
56. glutInit(&argc,argv);
57. glutInitDisplayMode(GLUT_DOUBLE |GLUT_RGB);
58. glutInitWindowSize(500,500);
59. glutInitWindowPosition(100,100);
60. glutCreateWindow(argv [0]);
61. init();
62. glutDisplayFunc(display);
63. glutReshapeFunc(reshape);
64. glutKeyboardFunc(keyboard);
65. glutMainLoop();
66. return 0;
67. }
```

尝试一下

试着在行星系统中增加一颗卫星，或者增加几颗卫星以及另外一颗行星。提示：使用glPushMatrix()和 glPopMatrix()在适当的时候保存和恢复坐标系统的位置。如果打算绘制几颗卫星绕同一颗行星旋转，需要在移动每颗卫星的位置之前保存坐标系统，并且在绘制每颗卫星之后恢复坐标系统。

尝试把行星的轴倾斜。

3.8.2 创建机器人手臂

本节讨论一个创建带关节的机器人手臂的程序。这个手臂在肩、肘或其他关节处应该用节点进行连接。图 3-25 显示了一个带关节的手臂。

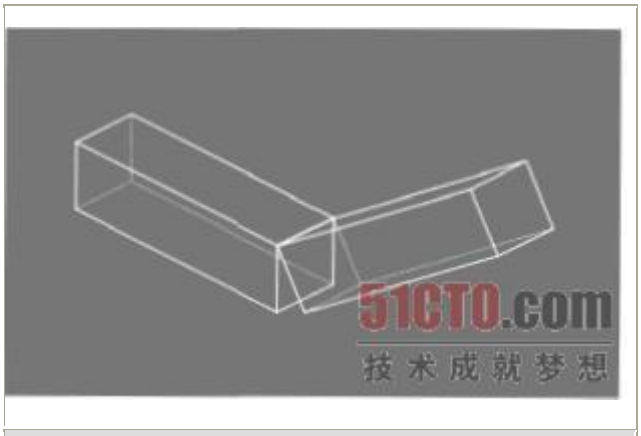


图 3-25 机器人手臂

可以使用一个经过缩放的立方体作为机器人手臂的一段。首先必须调用适当的模型变换函数，把手臂的每一段放在适当的位置。由于局部坐标系统的原点最初位于立方体的中心，因此需要把局部坐标系统移动到立方体的其中一条边上。否则，立方体将沿着它的中心旋转，而不是沿节点旋转。

调用 glTr a n s l a t e * ( )函数建立了节点并调用 glRotate*()函数使立方体绕节点旋转之后，需要把局部坐标系统移回到立方体的中心。然后，在绘制立方体之前对它进行缩放（压扁并拉宽）。可以使用 glPushMatrix()和 glPopMatrix()函数限制 glScale*()函数的效果。下面是创建第一段手臂时可以使用的代码，完整的程序见示例程序 3-7。

```
1. glTranslatef(-1.0, 0.0, 0.0);
2. glRotatef((GLfloat) shoulder, 0.0, 0.0, 1.0);
3. glTranslatef(1.0, 0.0, 0.0);
4. glPushMatrix();
5. glScalef(2.0, 0.4, 1.0);
6. glutWireCube(1.0);
7. glPopMatrix();
```

为了创建第二段手臂，需要把局部坐标系统移动到下一个节点。由于坐标系统此前已经进行了旋转，x 轴已经与经过旋转的手臂的长边方向对齐。因此，可以沿 x 轴把局部坐标系统移动到下一个节点。到达这个节点之后，使用和绘制第一段手臂相同的代码绘制手臂的第二段。按照这种方法，可以继续绘制更多的手臂段（肩、肘、腕、指等）。

```
1. glTranslatef(1.0, 0.0, 0.0);
2. glRotatef((GLfloat) elbow, 0.0, 0.0, 1.0);
3. glTranslatef(1.0, 0.0, 0.0);
4. glPushMatrix();
5. glScalef(2.0, 0.4, 1.0);
6. glutWireCube(1.0);
```

```
7. glPopMatrix();
```

示例程序 3-7 机器人手臂：robot.c

```
1. static int shoulder =0,elbow =0;
2. void init(void)
3. {
4. glClearColor(0.0,0.0,0.0,0.0);
5. glShadeModel(GL_FLAT);
6. }
7. void display(void)
8. {
9. glClear(GL_COLOR_BUFFER_BIT);
10. glPushMatrix();
11. glTranslatef(-1.0,0.0,0.0);
12. glRotatef((GLfloat)shoulder,0.0,0.0,1.0);
13. glTranslatef(1.0,0.0,0.0);
14. glPushMatrix();
15. glScalef(2.0,0.4,1.0);
16. glutWireCube(1.0);
17. glPopMatrix();
18. glTranslatef(1.0,0.0,0.0);
19. glRotatef((GLfloat)elbow,0.0,0.0,1.0);
20. glTranslatef(1.0,0.0,0.0);
21. glPushMatrix();
22. glScalef(2.0,0.4,1.0);
23. glutWireCube(1.0);
24. glPopMatrix();
25. glPopMatrix();
26. glutSwapBuffers();
27. }
28. void reshape(int w,int h)
29. {
30. glViewport(0,0,(GLsizei)w,(GLsizei)h);
31. glMatrixMode(GL_PROJECTION);
32. glLoadIdentity();
33. gluPerspective(65.0,(GLfloat)w/(GLfloat)h,1.0,20.0);
34. glMatrixMode(GL_MODELVIEW);
35. glLoadIdentity();
```

```
glTranslatef(0.0,0.0,-5.0);
}
void keyboard(unsigned char key,int x,int y)
{
switch (key){
case 's ' :/*s key rotates at shoulder */
shoulder =(shoulder +5)%360;
glutPostRedisplay();
break;
case 'S ':
shoulder =(shoulder -5)%360;
glutPostRedisplay();
break;
case 'e ' :/*e key rotates at elbow */
elbow =(elbow +5)%360;
glutPostRedisplay();
break;
case 'E ':
elbow =(elbow -5)%360;
glutPostRedisplay();
break;
default:
break;
}
}
int main(int argc,char**argv)
{
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_DOUBLE |GLUT_RGB);
glutInitWindowSize(500,500);
glutInitWindowPosition(100,100);
glutCreateWindow(argv [0]);
init();
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutKeyboardFunc(keyboard);
glutMainLoop();
return 0;
```

```
74. }
```

尝试一下

修改示例程序 3-7，在机器人手臂上再增加几段。

修改示例程序 3-7，在相同的位置上增加几段。例如，在机器人的腕部增加几个“手指”，如图 3-26 所示。提示：使用 glPushMatrix()和 glPopMatrix()保存手腕处坐标系统的位置和方向。如果打算绘制手指，需要在设置每个手指的位置之前保存当前矩阵，并在画完每个手指之后恢复当前矩阵。

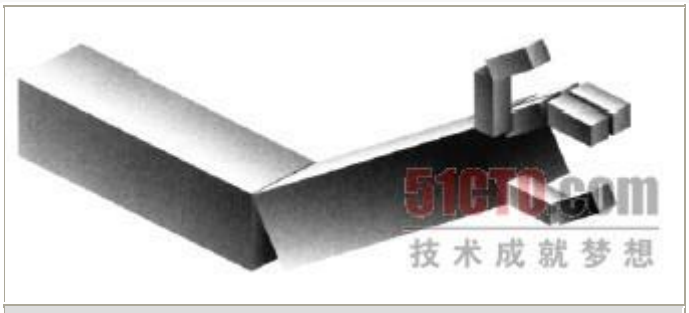


图 3-26 带手指的机器人手臂

Nate Robin 的变换教程

如果已经下载了 Nate Robin 的教学程序包，可以回到“transformation”教程，并再次运行它。可以使用弹出菜单修改 glRotate*()和 glTranslate()的顺序，并注意交换这些函数调用的效果。

3.9 逆变换和模拟变换

几何处理管线擅长使用视图和投影矩阵以及用于裁剪的视口把顶点的世界（物体）坐标变换为窗口（屏幕）坐标。但是，在有些情况下，需要反转这个过程。一种常见的情形就是应用程序的用户利用鼠标选择了三维空间中的一个位置。鼠标只返回一个二维值，也就是鼠标光标的屏幕位置。因此，应用程序必须反转变换过程，确定这个屏幕位置源于三维空间的什么地方。

OpenGL 工具函数库中 gluUnProject()和 gluUnProject4()函数用于执行这种逆变换操作。只要提供一个经过变换的顶点的三维窗口坐标以及所有对它产生影响的变换，gluUnProject()就可以返回这个顶点的源物体坐标。如果深度范围不是默认的[0, 1]，应该使用 gluUnProject4()函数。

```
1. int gluUnProject(GLdouble winx, GLdouble winy, GLdouble winz,
2. const GLdouble modelMatrix[16],
3. const GLdouble projMatrix[16],
4. const GLint viewport[4],
5. GLdouble *objx, GLdouble *objy, GLdouble *objz);
```

这个函数使用由模型视图矩阵（modelMatrix）、投影矩阵（projMatrix）和视口（viewport）定义的变换，把指定的窗口坐标（winx, winy, winz）映射到物体坐标。它产生的物体坐标是在 objx、objy 和 objz 中返回的。这个函数返回 GL_TRUE（表示成功）或 GL_FALSE（表示失败，例如矩阵不可逆）。这个函数并不会使用视口对坐标进行裁剪，也不会消除那些位于 glDepthRange()范围之外的深度值。

对变换过程进行逆操作存在一些固有的难度。二维屏幕上的一个位置可以来自于三维空间中一条直线上的任意一点。为了消除这种歧义，gluUnProject()要求提供一个窗口深度坐标（winz），它是根据 glDepthRange()函数指定的。关于深度范围的更多信息，参见第 3.4.2 节。如果使用的是 glDepthRange()的默认值，winz 为 0.0 表示这个经过变换的点的物体坐标位于近侧裁剪平面上，如果 winz 是 1.0 表示它的坐标位于远侧裁剪平面上。

示例程序 3-8 说明了 gluUnProject()函数的用法。这个程序读取鼠标位置，并确定鼠标光标位置经过变换之后在三维空间的近侧裁剪平面和远侧裁剪平面上的点。经过计算得的物体坐标被打印到标准输出，但渲染的窗口本身仍然保持为黑色。

示例程序 3-8 逆变换几何处理管线：unproject.c

```
1. void display(void)
2. {
3. glClear(GL_COLOR_BUFFER_BIT);
4. glFlush();
5. }
6. void reshape(int w,int h)
7. {
8. glViewport(0,0,(GLsizei)w,(GLsizei)h);
9. glMatrixMode(GL_PROJECTION);
10.glLoadIdentity();
11.gluPerspective(45.0,(GLfloat)w/(GLfloat)h,1.0,100.0);
12.glMatrixMode(GL_MODELVIEW);
13.glLoadIdentity();
14.}
15.void mouse(int button,int state,int x,int y)
16.{
17.GLint viewport [4];
18.GLdouble mvmatrix [16],projmatrix [16];
19.GLint realy;/*OpenGL y coordinate position */
20.GLdouble wx,wy,wz;/*returned world x,y,z coords */
21.switch (button){
```

```
case GLUT_LEFT_BUTTON:
if (state ==GLUT_DOWN){
glGetIntegerv(GL_VIEWPORT,viewport);
glGetDoublev(GL_MODELVIEW_MATRIX,mvmatrix);
glGetDoublev(GL_PROJECTION_MATRIX,projmatrix);
/*note viewport [3] is height of window in pixels */
realy =viewport [3] -(GLint)y -1;
printf("Coordinates at cursor are (%4d,%4d)\n " ,
x,realy);
gluUnProject((GLdouble)x,(GLdouble)realy,0.0,
mvmatrix,projmatrix,viewport,&wx,&wy,&wz);
printf("World coords at z=0.0 are (%f,%f,%f)\n ",
wx,wy,wz);
gluUnProject((GLdouble)x,(GLdouble)realy,1.0,
mvmatrix,projmatrix,viewport,&wx,&wy,&wz);
printf("World coords at z=1.0 are (%f,%f,%f)\n ",
wx,wy,wz);
}
break;
case GLUT_RIGHT_BUTTON:
if (state == GLUT_DOWN)
exit(0);
break;
default:
break;
}
}
int main(int argc,char**argv)
{
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB);
glutInitWindowSize(500,500);
glutInitWindowPosition(100,100);
glutCreateWindow(argv [0]);
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutMouseFunc(mouse);
glutMainLoop();
```

```
60. return 0;
61. }
```

GLU 1.3 引入了一个经过修改的 gluUnProject()版本。gluUnProject4()可以处理非标准的 glDepthRange()值，也可以处理 w 坐标值不等于 1 的情况。

```
1. int gluUnProject4(GLdouble winx, GLdouble winy, GLdouble winz,
2. GLdouble clipw, const GLdouble modelMatrix[16],
3. const GLdouble projMatrix[16],
4. const GLint viewport[4],
5. GLclampd zNear, GLclampd zFar,
6. GLdouble *objx, GLdouble *objy,
7. GLdouble *objz, GLdouble *objw);
```

总体上说，这个函数的功能类似于 gluUnProject()。它使用由模型视图矩阵、投影矩阵、视口和深度范围值 zNear 和 zFar 定义的变换，把指定的窗口坐标（winx, winy, winz）映射到物体坐标。它产生的物体坐标是在 objx、objy、ojbz 和 objw 中返回的。

gluProject()是另一个工具函数库中的函数，和 gluUnProject()相对应。gluProject()模拟变换管线的操作。只要提供三维物体坐标以及对它们产生影响的所有变换，gluProject()就会返回经过变换的窗口坐标。

```
1. int gluProject(GLdouble objx, GLdouble objy, GLdouble objz,
2. const GLdouble modelMatrix[16],
3. const GLdouble projMatrix[16],
4. const GLint viewport[4],
5. GLdouble *winx, GLdouble *winy, GLdouble *winz);
```

这个函数使用由模型视图矩阵、投影矩阵和视口定义的变换，把指定的物体坐标（objx, objy,objz）变换为窗口坐标。它产生的窗口坐标是在 winx、winy 和 winz 中返回的。这个函数返回 GL_TRUE（表示成功）或 GL_FALSE（表示失败）。

注意：传递给 gluUnProject()、gluUnProject4()和 gluProject()的是 OpenGL 标准格式的列主序矩阵。可以使用 glGetDoublev()和 glGetIntegerv()，获取 GL_MODELVIEW_MATRIX、GL_PROJECTION_MATRIX 和 GL_VEIWPORT 的当前值，以便在 gluUnProject()、gluUn-Project4()和 gluProject()中使用。