OpenGL编程指南（第七版）

蓝月追风

目录

译者序
前言
第 1 章　OpenGL 简介 1
1.1　什么是 OpenGL 1
1.2　一段简单的 OpenGL 代码 3
1.3　OpenGL 函数的语法 4
1.4　OpenGL 是一个状态机 6
1.5　OpenGL 渲染管线 6
1.5.1　显示列表 7
1.5.2　求值器 7
1.5.3　基于顶点的操作 7
1.5.4　图元装配 7
1.5.5　像素操作 8
1.5.6　纹理装配 8
1.5.7　光栅化 8
1.5.8　片断操作 8
1.6　与 OpenGL 相关的函数库 9
1.6.1　包含文件 9
1.6.2　OpenGL 实用工具库（GLUT） 10
1.7　动画 13
1.7.1　暂停刷新 14
1.7.2　动画=重绘+交换 15
1.8　OpenGL 及其废弃机制 17
1.8.1　OpenGL 渲染环境 17
1.8.2　访问 OpenGL 函数 18
第 2 章　状态管理和绘制几何物体 19
2.1　绘图工具箱 20
2.1.1　清除窗口 20
2.1.2　指定颜色 22
2.1.3　强制完成绘图操作 23
2.1.4　坐标系统工具箱 24
2.2　描述点、直线和多边形 25
2.2.1　什么是点、直线和多边形 25

2.2.2　指定顶点 27
2.2.3　OpenGL 几何图元 27
2.3　基本状态管理 31
2.4　显示点、直线和多边形 32
2.4.1　点的细节 32
2.4.2　直线的细节 33
2.4.3　多边形的细节 36
2.5　法线向量 41
2.6　顶点数组 43
2.6.1　步骤 1：启用数组 44
2.6.2　步骤 2：指定数组的数据 44
2.6.3　步骤 3：解引用和渲染 46
2.6.4　重启图元 51
2.6.5　实例化绘制 53
2.6.6　混合数组 54
2.7　缓冲区对象 57
2.7.1　创建缓冲区对象 57
2.7.2　激活缓冲区对象 58
2.7.3　用数据分配和初始化缓冲区对象 58
2.7.4　更新缓冲区对象的数据值 60
2.7.5　在缓冲区对象之间复制数据 62
2.7.6　清除缓冲区对象 63
2.7.7　使用缓冲区对象存储顶点数组数据 63
2.8　顶点数组对象 65
2.9　属性组 69
2.10　创建多边形表面模型的一些提示 71
第 3 章　视图 77
3.1　简介：用照相机打比方 78
3.1.1　一个简单的例子：绘制立方体 80
3.1.2　通用的变换函数 83
3.2　视图和模型变换 84
3.2.1　对变换进行思考 85
3.2.2　模型变换 86
3.2.3　视图变换 89
3.3　投影变换 93

3.3.1　透视投影 94
3.3.2　正投影 95
3.3.3　视景体裁剪 96
3.4　视口变换 96
3.4.1　定义视口 96
3.4.2　变换深度坐标 97
3.5　和变换相关的故障排除 98
3.6　操纵矩阵堆栈 100
3.6.1　模型视图矩阵堆栈 101
3.6.2　投影矩阵堆栈 102
3.7　其他裁剪平面 102
3.8　一些组合变换的例子 104
3.8.1　创建太阳系模型 104
3.8.2　创建机器人手臂 107
3.9　逆变换和模拟变换 109
第 4 章　颜色 113
4.1　颜色感知 113
4.2　计算机颜色 114
4.3　RGBA 和颜色索引模式 115
4.3.1　RGBA 显示模式 116
4.3.2　颜色索引模式 117
4.3.3　在 RGBA 和颜色索引模式中进行选择 118
4.3.4　切换显示模式 118
4.4　指定颜色和着色模型 119
4.4.1　在 RGBA 模式下指定颜色 119
4.4.2　在颜色索引模式下指定颜色 120
4.4.3　指定着色模型 121
第 5 章　光照 123
5.1　隐藏表面消除工具箱 124
5.2　现实世界和 OpenGL 光照 125
5.2.1　环境光、散射光、镜面光和发射光 125
5.2.2　材料颜色 126
5.2.3　光和材料的 RGB 值 126
5.3　一个简单的例子：渲染光照球体 127
5.4　创建光源 129

5.4.1　颜色 130
5.4.2　位置和衰减 131
5.4.3　聚光灯 132
5.4.4　多光源 133
5.4.5　控制光源的位置和方向 133
5.5　选择光照模型 138
5.5.1　全局环境光 138
5.5.2　局部的观察点或无限远的观察点 138
5.5.3　双面光照 139
5.5.4　镜面辅助颜色 139
5.5.5　启用光照 140
5.6　定义材料属性 140
5.6.1　散射和环境反射 141
5.6.2　镜面反射 141
5.6.3　发射光颜色 142
5.6.4　更改材料属性 142
5.6.5　颜色材料模式 143
5.7　和光照有关的数学知识 146
5.7.1　材料的发射光 147
5.7.2　经过缩放的全局环境光 147
5.7.3　光源的贡献 147
5.7.4　完整的光照计算公式 148
5.7.5　镜面辅助颜色 148
5.8　颜色索引模式下的光照 149
第 6 章　混合、抗锯齿、雾和多边形偏移 151
6.1　混合 152
6.1.1　源因子和目标因子 152
6.1.2　启用混合 154
6.1.3　使用混合方程式组合像素 154
6.1.4　混合的样例用法 156
6.1.5　一个混合的例子 157
6.1.6　使用深度缓冲区进行三维混合 159
6.2　抗锯齿 162
6.2.1　对点和直线进行抗锯齿处理 164
6.2.2　使用多重采样对几何图元进行抗锯齿处理 169

6.2.3　对多边形进行抗锯齿处理 172
6.3　雾 172
6.3.1　使用雾 173
6.3.2　雾方程式 175
6.4　点参数 181
6.5　多边形偏移 182
第 7 章　显示列表 185
7.1　为什么使用显示列表 185
7.2　一个使用显示列表的例子 186
7.3　显示列表的设计哲学 188
7.4　创建和执行显示列表 189
7.4.1　命名和创建显示列表 191
7.4.2　存储在显示列表里的是什么 191
7.4.3　执行显示列表 193
7.4.4　层次式显示列表 193
7.4.5　管理显示列表索引 194
7.5　执行多个显示列表 194
7.6　用显示列表管理状态变量 199
第 8 章　绘制像素、位图、字体和图像 202
8.1　位图和字体 203
8.1.1　当前光栅位置 204
8.1.2　绘制位图 205
8.1.3　选择位图的颜色 206
8.1.4　字体和显示列表 206
8.1.5　定义和使用一种完整的字体 207
8.2　图像 209
8.3　图像管线 215
8.3.1　像素包装和解包 216
8.3.2　控制像素存储模式 217
8.3.3　像素传输操作 219
8.3.4　像素映射 221
8.3.5　放大、缩小或翻转图像 222
8.4　读取和绘制像素矩形 224
8.5　使用缓冲区对象存取像素矩形数据 227
8.5.1　使用缓冲区对象传输像素数据 227

8.5.2　使用缓冲区对象提取像素数据 228
8.6　提高像素绘图速度的技巧 229
8.7　图像处理子集 230
8.7.1　颜色表 231
8.7.2　卷积 234
8.7.3　颜色矩阵 240
8.7.4　柱状图 241
8.7.5　最小最大值 243
第 9 章　纹理贴图 245
9.1　概述和示例 248
9.1.1　纹理贴图的步骤 248
9.1.2　一个示例程序 249
9.2　指定纹理 251
9.2.1　纹理代理 255
9.2.2　替换纹理图像的全部或一部分 257
9.2.3　一维纹理 259
9.2.4　三维纹理 261
9.2.5　纹理数组 264
9.2.6　压缩纹理图像 265
9.2.7　使用纹理边框 267
9.2.8　mipmap：多重细节层 267
9.3　过滤 2759.4　纹理对象 277
9.4.1　命名纹理对象 277
9.4.2　创建和使用纹理对象 278
9.4.3　清除纹理对象 280
9.4.4　常驻纹理工作集 280
9.5　纹理函数 282
9.6　分配纹理坐标 284
9.6.1　计算正确的纹理坐标 285
9.6.2　重复和截取纹理 286
9.7　纹理坐标自动生成 289
9.7.1　创建轮廓线 289
9.7.2　球体纹理 293
9.7.3　立方图纹理 294
9.8　多重纹理 296

9.9　纹理组合器函数 299
9.10　在纹理之后应用辅助颜色 303
9.10.1　在禁用光照时使用辅助颜色 303
9.10.2　启用光照后的辅助镜面颜色 303
9.11　点块纹理 303
9.12　纹理矩阵堆栈 304
9.13　深度纹理 305
9.13.1　创建阴影图 306
9.13.2　生成纹理坐标并进行渲染 307
第 10 章　帧缓冲区 309
10.1　缓冲区及其用途 310
10.1.1　颜色缓冲区 311
10.1.2　清除缓冲区 312
10.1.3　选择用于读取和写入的颜色缓冲区 313
10.1.4　缓冲区的屏蔽 315
10.2　片断测试和操作 316
10.2.1　裁剪测试 316
10.2.2　alpha 测试 317
10.2.3　模板测试 318
10.2.4　深度测试 322
10.2.5　遮挡查询 322
10.2.6　条件渲染 324
10.2.7　混合、抖动和逻辑操作 325
10.3　累积缓冲区 327
10.3.1　运动模糊 328
10.3.2　景深 328
10.3.3　柔和阴影 331
10.3.4　微移 331
10.4　帧缓冲区对象 332
10.4.1　渲染缓冲区 333
10.4.2　复制像素矩形 340
第 11 章　分格化和二次方程表面 342
11.1　多边形分格化 342
11.1.1　创建分格化对象 343
11.1.2　分格化回调函数 343

11.1.3　分格化属性 347
11.1.4　多边形定义 350
11.1.5　删除分格化对象 352
11.1.6　提高分格化性能的建议 352
11.1.7　描述 GLU 错误 352
11.1.8　向后兼容性 352
11.2　二次方程表面：渲染球体、圆柱体和圆盘 353
11.2.1　管理二次方程对象 354
11.2.2　控制二次方程对象的属性 354
11.2.3　二次方程图元 355
第 12 章　求值器和 NURBS 360
12.1　前提条件 360
12.2　求值器 361
12.2.1　一维求值器 361
12.2.2　二维求值器 365
12.2.3　使用求值器进行纹理处理 369
12.3　GLU 的 NURBS 接口 371
12.3.1　一个简单的 NURBS 例子 371
12.3.2　管理 NURBS 对象 374
12.3.3　创建 NURBS 曲线或表面 377
12.3.4　修剪 NURBS 表面 380
第 13 章　选择和反馈 383
13.1　选择 383
13.1.1　基本步骤 384
13.1.2　创建名字栈 384
13.1.3　点击记录 385
13.1.4　一个选择例子 386
13.1.5　挑选 389
13.1.6　编写使用选择的程序的一些建议 397
13.2　反馈 398
13.2.1　反馈数组 399
13.2.2　在反馈模式下使用标记 400
13.2.3　一个反馈例子 400
第 14 章　OpenGL 高级技巧 404
14.1　错误处理 405

14.2 OpenGL 版本 406
14.2.1 工具函数库版本 407
14.2.2 窗口系统扩展版本 407
14.3 标准的扩展 407
14.4 实现半透明效果 409
14.5 轻松实现淡出效果 409
14.6 使用后缓冲区进行物体选择 411
14.7 低开销的图像转换 411
14.8 显示层次 412
14.9 抗锯齿字符 413
14.10 绘制圆点 414
14.11 图像插值 414
14.12 制作贴花 415
14.13 使用模板缓冲区绘制填充的凹多边形 416
14.14 寻找冲突区域 416
14.15 阴影 417
14.16 隐藏直线消除 418
14.16.1 使用多边形偏移实现隐藏直线消除 418
14.16.2 使用模板缓冲区实现隐藏直线消除 419
14.17 纹理贴图的应用 419
14.18 绘制深度缓冲的图像 420
14.19 Dirichlet 域 420
14.20 使用模板缓冲区实现生存游戏 421
14.21 glDrawPixels()和 glCopyPixels()的其他应用 422
第 15 章 OpenGL 着色语言 424
15.1 OpenGL 图形管线和可编程着色器 424
15.1.1 顶点处理 425
15.1.2 片断处理 426
15.2 使用 GLSL 着色器 427
15.2.1 着色器示例 427
15.2.2 OpenGL/GLSL 接口 428
15.3 OpenGL 着色语言 432
15.4 使用 GLSL 创建着色器 433
15.4.1 程序起点 433
15.4.2 声明变量 433

15.4.3　聚合类型 434
15.5　uniform 块 439
15.5.1　在着色器中指定 uniform 变量 440
15.5.2　访问在 uniform 块中声明的 uniform 变量 440
15.5.3　计算不变性 446
15.5.4　语句 446
15.5.5　函数 448
15.5.6　在 GLSL 程序中使用 OpenGL 状态值 449
15.6　在着色器中访问纹理图像 449
15.7　着色器预处理器 452
15.7.1　预处理器指令 452
15.7.2　宏定义 452
15.7.3　预处理器条件 453
15.7.4　编译器控制 453
15.8　扩展处理 454
15.9　顶点着色器的细节 454
15.10　变换反馈 458
15.11　片断着色器 462
附录 A　GLUT（OpenGL 实用工具库）基础知识 464
附录 B　状态变量 468
附录 C　齐次坐标和变换矩阵 495
附录 D　OpenGL 和窗口系统 499
术语表 511

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（Graphics Library 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。

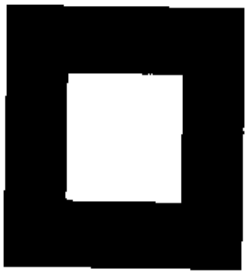


图1-1 黑色背景中的白色矩形

示例程序1-1 一段OpenGL代码

```
#include <whateverYouNeed.h>

main(){
   InitializeAWindowPlease();
   glClearColor(0.0,0.0,0.0,0.0);
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0,1.0,1.0);
   glOrtho(0.0,1.0,0.0,1.0,-1.0,1.0);
   glBegin(GL_POLYGON);
      glVertex3f(0.25,0.25,0.0);
      glVertex3f(0.75,0.25,0.0);
      glVertex3f(0.75,0.75,0.0);
      glVertex3f(0.25,0.75,0.0);
   glEnd();
   glFlush();
   UpdateTheWindowAndCheckForEvents();
}
```

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

后缀	数据类型	典型的对应C语言类型	OpenGL类型定义
b	8位整数	signed char	GLbyte
s	16位整数	short	GLshort
i	32位整数	int或long	GLint, GLsizei
f	32位浮点数	float	GLfloat, GLclampf
d	64位浮点数	double	GLdouble, GLclampd
ub	8位无符号整数	unsigned char	GLubyte, GLboolean
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, GLenum, GLbitfield

因此，下面这两个函数调用

```
glVertex2i(1, 3);
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：

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

GLfloat color_array[] = {1.0, 0.0, 0.0};
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

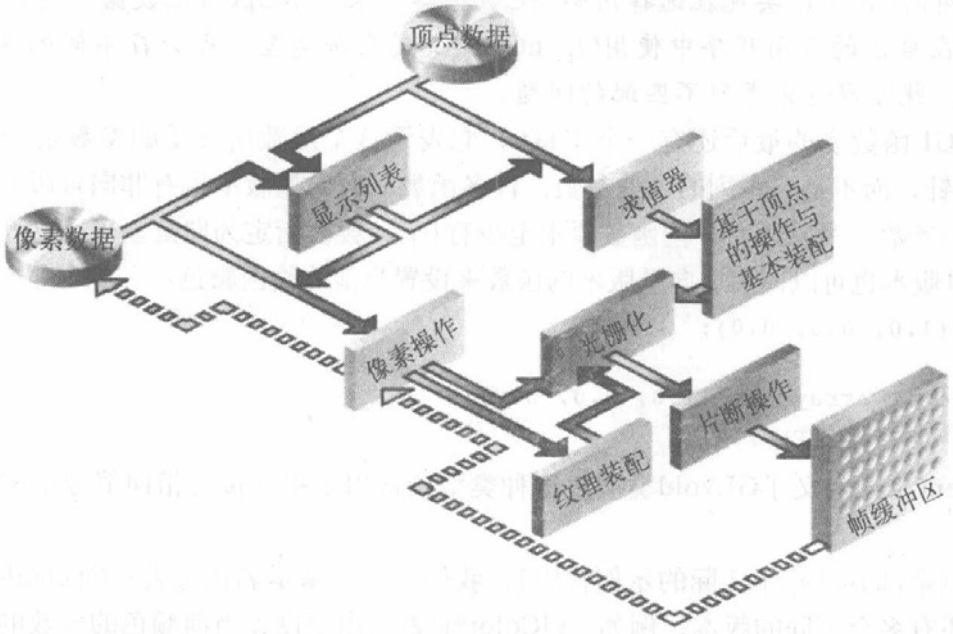


图1-2 操作顺序

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。

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：

```
#include <GL/gl.h>
#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Microsoft Windows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：

```
#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：

```
#include <X11/Xlib.h>
#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：

```
#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：

```
#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：

```
#include <stdlib.h>
#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。

```
#include <GL3/gl3.h>
#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OpenGL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

```
void display(void)
{
/*   clear all pixels */
     glClear(GL_COLOR_BUFFER_BIT);
/*draw white polygon (rectangle)with corners at
 *(0.25,0.25,0.0)and (0.75,0.75,0.0)
 */
 glColor3f(1.0,1.0,1.0);
 glBegin(GL_POLYGON);
     glVertex3f(0.25,0.25,0.0);
     glVertex3f(0.75,0.25,0.0);
     glVertex3f(0.75,0.75,0.0);
     glVertex3f(0.25,0.75,0.0);
 glEnd();

/*  don 't wait!
 *  start processing buffered OpenGL routines
 */
    glFlush();
}

void init(void)
{
/*  select clearing (background)color */
    glClearColor(0.0,0.0,0.0,0.0);

/*  initialize viewing values */
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0,1.0,0.0,1.0,-1.0,1.0);
}

/*
 *  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)(int button, 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：

void **glutWireCube**(GLdouble *size*);

void **glutSolidCube**(GLdouble *size*);

void **glutWireSphere**(GLdouble *radius*, GLint *slices*, GLint *stacks*);

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）60～75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：

```
open_window();
for (i = 0; i < 1000000; i++) {
   clear_the_window();
   draw_frame(i);
   wait_until_a_24th_of_a_second_is_over();
}
```

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：

```
open_window_in_double_buffer_mode();
for (i = 0; i < 1000000; i++) {
   clear_the_window();
   draw_frame(i);
   swap_the_buffers();
}
```

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。

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。

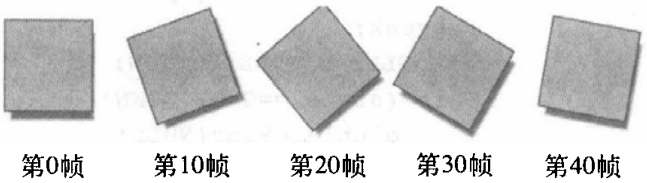


图1-3 双缓冲旋转方块

示例程序1-3 双缓冲程序：double.c

```
static GLfloat spin =0.0;

void init(void)
{
   glClearColor(0.0,0.0,0.0,0.0);
   glShadeModel(GL_FLAT);
}
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glPushMatrix();
   glRotatef(spin,0.0,0.0,1.0);
   glColor3f(1.0,1.0,1.0);
   glRectf(-25.0,-25.0,25.0,25.0);
   glPopMatrix();
   glutSwapBuffers();
}
void spinDisplay(void)
{
   spin =spin +2.0;
   if (spin >360.0)
      spin =spin -360.0;
   glutPostRedisplay();
}

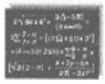
void reshape(int w,int h)
{
   glViewport(0,0,(GLsizei)w,(GLsizei)h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(-50.0,50.0,-50.0,50.0,-1.0,1.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

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，Khronos OpenGL 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Microsoft Windows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

```
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_RGBA |GLUT_DEPTH |GLUT_DOUBLE);
glutInitWindowSize(width,height);
glutInitWindowPosition(xPos,yPos);
glutInitContextVersion(3,0);
glutInitContextFlags(GLUT_FORWARD_COMPATIBLE);
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：

```
glClearColor(0.0, 0.0, 0.0, 0.0);
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：

```
glClearColor(0.0, 0.0, 0.0, 0.0);
glClearDepth(1.0);
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。

---

void **glClearColor**(GLclampf *red*, GLclampf *green*, GLclampf *blue*,
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。

---

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

兼容性扩展
GL_ACCUM_BUFFER_BIT

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：

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

和

```
glClear(GL_COLOR_BUFFER_BIT);
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：

```
set_current_color(red);
draw_object(A);
draw_object(B);
set_current_color(green);
set_current_color(blue);
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：

```
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。

---

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()**。

---

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

```
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);
}
```

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。

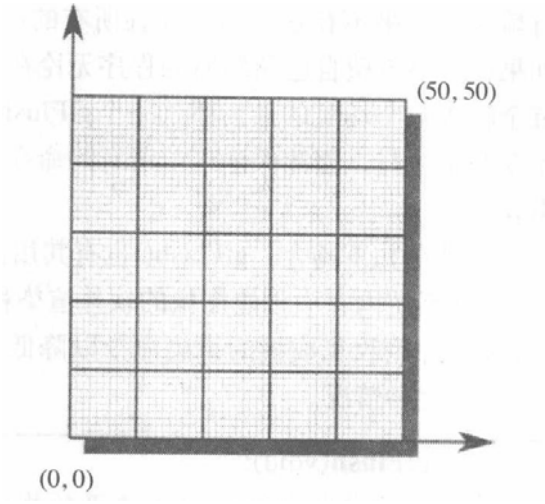


图2-1 由w = 50，h = 50所定义的坐标系统

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

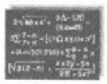
点、直线和多边形的数学概念比较简单。在OpenGL中，它们的概念与数学概念具有相似之处，但并不完全相同。

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。

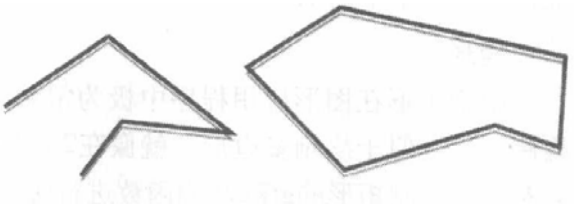


图2-2 两条连接线段系列

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。

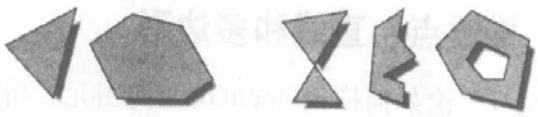


图2-3 合法和非法多边形

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。

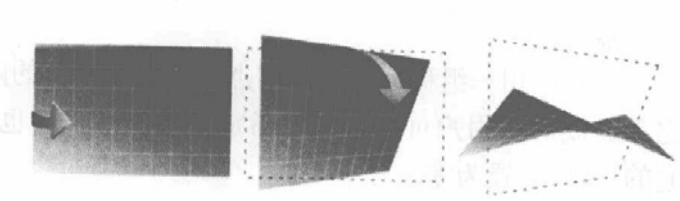


图2-4 非平面多边形转换为非简单多边形

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。

---

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

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。

兼容性扩展
glRect

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）。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。



图2-5 模拟曲线

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。

兼容性扩展
glVertex

void **glVertex**[234]{sifd}(TYPE *coords*);

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

```
glVertex2s(2, 3);
glVertex3d(0.0, 0.0, 3.1415926535898);
glVertex4f(2.3, 1.0, -2.2, 2.0);

GLdouble dvect[3] = {5.0, 9.0, 1992.0};
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

```
glBegin(GL_POLYGON);
   glVertex2f(0.0, 0.0);
   glVertex2f(0.0, 3.0);
   glVertex2f(4.0, 3.0);
   glVertex2f(6.0, 1.5);
   glVertex2f(4.0, 0.0);
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。

兼容性扩展
glBegin
GL_QUADS
GL_QUAD_STRIP
GL_POLYGON

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	含 义
GL_POINTS	单个的点
GL_LINES	一对顶点被解释为一条直线
GL_LINE_STRIP	一系列的连接直线
GL_LINE_LOOP	和上面相同，但第一个顶点和最后一个顶点彼此相连
GL_TRIANGLES	3个顶点被解释为一个三角形
GL_TRIANGLE_STRIP	三角形的连接串
GL_TRIANGLE_FAN	连接成扇形的三角形系列
GL_QUADS	4个顶点被解释为一个四边形
GL_QUAD_STRIP	四边形的连接串
GL_POLYGON	简单的凸多边形的边界

void **glEnd**(void);

标志着一个顶点数据列表的结束。

兼容性扩展
glEnd

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 $(v_0, v_1, v_2, \cdots, v_{n-1})$。

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。这些直线是在v_0和v_1、v_2和v_3（接下来以此类推）之间绘制的。如果n是奇数，最后一条直线是在v_{n-3}和v_{n-2}之间绘制的，v_{n-1}被忽略
GL_LINE_STRIP	从v_0到v_1绘制一条直线，然后从v_1到v_2绘制一条直线，接下来以此类推。最后所绘制

（续）

	的是从v_{n-2}到v_{n-1}的直线。因此，总共有n−1条直线。如果n不大于1，就不会绘制任何直线。描述直线串（或者直线环）的顶点并不存在任何限制，这些直线可以任意相交
GL_LINE_LOOP	与GL_LINE_STRIP相似，只是最后一条直线是从v_{n-1}到v_0，从而构成一个线环
GL_TRIANGLES	绘制一系列的三角形（3条边的多边形），第一个三角形使用顶点v_0、v_1和v_2，第二个三角形使用v_3、v_4和v_5，接下来以此类推。如果n不是3的倍数，则最后的1个或2个顶点被忽略
GL_TRIANGLE_STRIP	绘制一系列的三角形（3条边的多边形），第一个三角形使用顶点v_0、v_1和v_2，第二个三角形使用v_2、v_1和v_3（注意顺序），然后是v_2、v_3和v_4，接下来以此类推。这个顺序是为了保证所有的三角形都是按照相同的方向绘制的，使这个三角形串能够正确地形成表面的一部分。对于有些操作，维持方向是非常重要的，例如剔除（参见第2.4.3小节中的“反转和剔除多边形表面”）。n至少要大于3，否则就不会绘制任何三角形
GL_TRIANGLE_FAN	和GL_TRIANGLE_STRIP相似，只是顶点顺序现在是v_0、v_1、v_2，接着是v_0、v_2、v_3，然后是v_0、v_3、v_4，接下来以此类推（如图2-7所示）
GL_QUADS	绘制一系列的四边形（4条边的多边形），首先使用顶点v_0、v_1、v_2、v_3，然后是v_4、v_5、v_6和v_7，接下来以此类推。如果n不是4的倍数，最后1个、2个或3个顶点被忽略
GL_QUAD_STRIP	绘制一系列的四边形（4条边的多边形），首先使用顶点v_0、v_1、v_3、v_2，接着是v_2、v_3、v_5、v_4，然后是v_4、v_5、v_7和v_6，接下来以此类推（如图2-7所示）。n至少要大于4，否则不会绘制任何四边形。如果n是奇数，最后一个顶点将被忽略
GL_POLYGON	绘制一个多边形，使用点v_0、…、v_{n-1}作为顶点。n必须大于等于3，否则就不会绘制任何多边形。另外，它所指定的多边形本身必须不得相交并且必须是凸多边形。如果顶点不满足这些条件，其结果是不可预测的

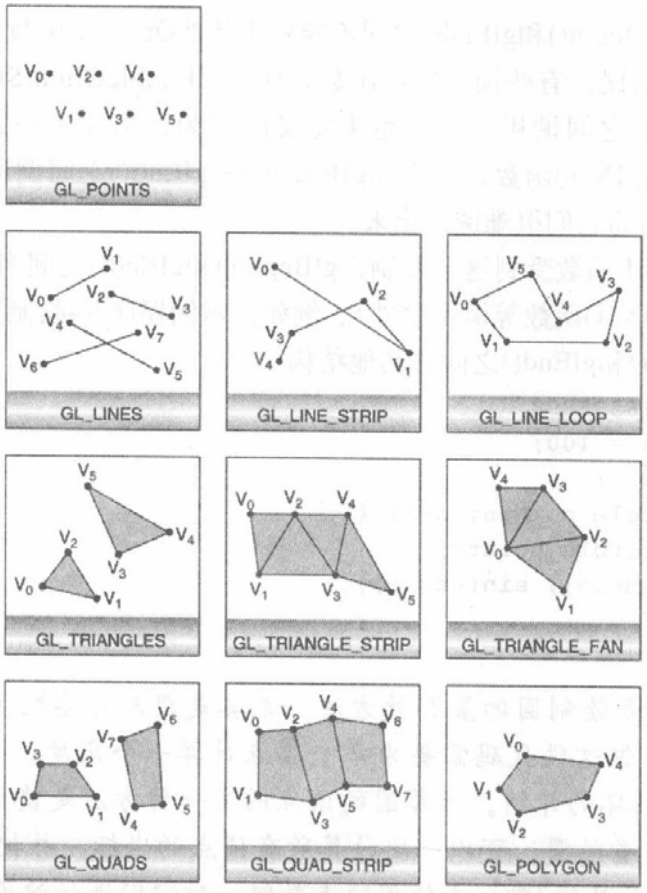


图2-7 几何图元类型

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	参考章节
glVertex*()	设置顶点坐标	第2章
glColor*()	设置RGBA颜色	第4章
glIndex*()	设置颜色索引	第4章
glSecondaryColor*()	设置纹理应用后的辅助颜色	第9章
glNormal*()	设置法线向量坐标	第2章
glMaterial*()	设置材料属性	第5章
glFogCoord*()	设置雾坐标	第6章
glTexCoord*()	设置纹理坐标	第9章
glMultiTexCoord*()	为多重纹理设置纹理坐标	第9章
glVertexAttrib*()	设置通用的顶点属性	第15章
glEdgeFlag*()	控制边界的绘制	第2章
glArrayElement()	提取顶点数组数据	第2章
glEvalCoord*(), glEvalPoint*()	生成坐标	第12章
glCallList(), glCallLists()	执行显示列表	第7章

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

```
#define PI 3.1415926535898
GLint circle_points = 100;
glBegin(GL_LINE_LOOP);
for (i = 0; i < circle_points; i++) {
   angle = 2*PI*i/circle_points;
   glVertex2f(cos(angle), sin(angle));
}
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。

```
glBegin(GL_POINTS);
   glColor3f(0.0, 1.0, 0.0);                    /* green */
   glColor3f(1.0, 0.0, 0.0);                    /* red */
   glVertex(...);
   glColor3f(1.0, 1.0, 0.0);                    /* yellow */
   glColor3f(0.0, 0.0, 1.0);                    /* blue */
   glVertex(...);
   glVertex(...);
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：

---

void **glEnable**(GLenum *capability*);

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。

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：

void **glGetBooleanv**(GLenum *pname*, GLboolean **params*);

void **glGetIntegerv**(GLenum *pname*, GLint **params*);

void **glGetFloatv**(GLenum *pname*, GLfloat **params*);

void **glGetDoublev**(GLenum *pname*, GLdouble **params*);

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）。

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

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。

```
glLineStipple(1, 0x3F07);
glEnable(GL_LINE_STIPPLE);
```

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

兼容性扩展
glLineStipple
GL_LINE_STIPPLE

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。

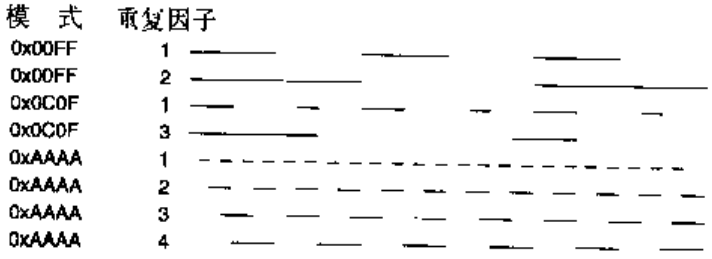


图2-8 点画线

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。

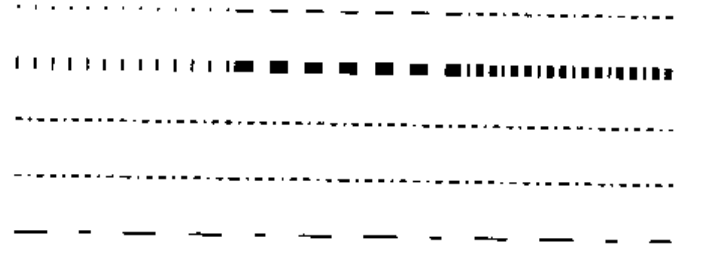


图2-9 宽点画线

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。

示例程序2-5 直线点画模式：lines.c

```
#define drawOneLine(x1,y1,x2,y2) glBegin(GL_LINES); \
   glVertex2f((x1),(y1)); glVertex2f((x2),(y2)); glEnd();

void init(void)
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_FLAT);
}

void display(void)
{
   int i;

   glClear(GL_COLOR_BUFFER_BIT);
/* select white for all lines */
   glColor3f(1.0, 1.0, 1.0);
```

```
/* in 1st row, 3 lines, each with a different stipple */
   glEnable(GL_LINE_STIPPLE);

   glLineStipple(1, 0x0101); /* dotted */
   drawOneLine(50.0, 125.0, 150.0, 125.0);
   glLineStipple(1, 0x00FF); /* dashed */
   drawOneLine(150.0, 125.0, 250.0, 125.0);
   glLineStipple(1, 0x1C47); /* dash/dot/dash */
   drawOneLine(250.0, 125.0, 350.0, 125.0);

/* in 2nd row, 3 wide lines, each with different stipple */
   glLineWidth(5.0);
   glLineStipple(1, 0x0101); /* dotted */
   drawOneLine(50.0, 100.0, 150.0, 100.0);
   glLineStipple(1, 0x00FF); /* dashed */
   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);
    glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}
```

2.4.3 多边形的细节

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。

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。

兼容性扩展
GL_FRONT
GL_BACK

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：

```
glPolygonMode(GL_FRONT, GL_FILL);
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（orientable mainifold）。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（Möbius 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。

void **glFrontFace**(GLenum *mode*);

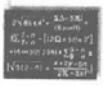
控制多边形的正面是如何决定的。在默认情况下，mode是GL_CCW，它表示窗口坐标上投影多边形的顶点顺序为逆时针方向的表面为正面。如果mode是GL_CW，顶点顺序为顺时针方向的表面被认为是正面。

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。

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，x_i和y_i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。

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）。

兼容性扩展
glPolygonStipple
GL_POLYGON_STIPPLE

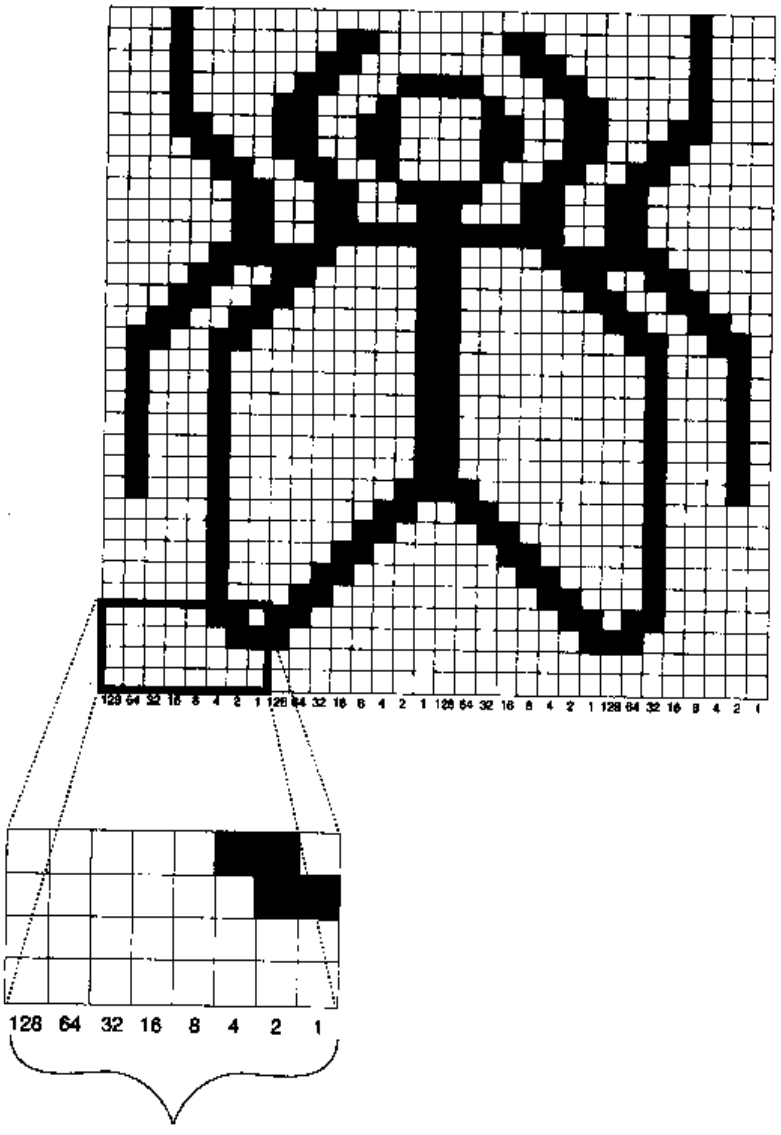


图2-10 创建一个多边形点画模式

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：

```
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。

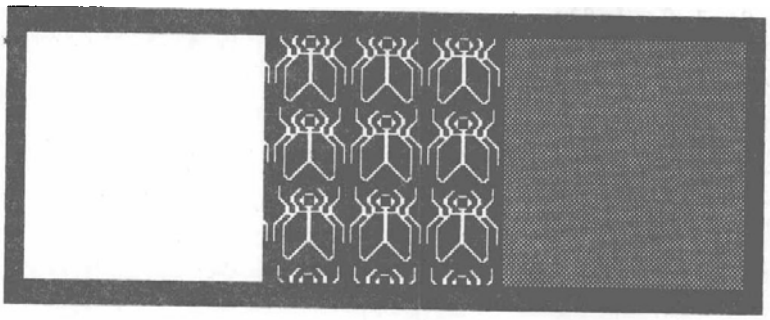


图2-11 点画多边形

示例程序2-6 多边形点画模式：polys.c

```
void display(void)
{
   GLubyte fly[] = {
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x03, 0x80, 0x01, 0xC0, 0x06, 0xC0, 0x03, 0x60,
      0x04, 0x60, 0x06, 0x20, 0x04, 0x30, 0x0C, 0x20,
      0x04, 0x18, 0x18, 0x20, 0x04, 0x0C, 0x30, 0x20,
      0x04, 0x06, 0x60, 0x20, 0x44, 0x03, 0xC0, 0x22,
      0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
      0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
      0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
      0x66, 0x01, 0x80, 0x66, 0x33, 0x01, 0x80, 0xCC,
      0x19, 0x81, 0x81, 0x98, 0x0C, 0xC1, 0x83, 0x30,
      0x07, 0xe1, 0x87, 0xe0, 0x03, 0x3f, 0xfc, 0xc0,
      0x03, 0x31, 0x8c, 0xc0, 0x03, 0x33, 0xcc, 0xc0,
      0x06, 0x64, 0x26, 0x60, 0x0c, 0xcc, 0x33, 0x30,
      0x18, 0xcc, 0x33, 0x18, 0x10, 0xc4, 0x23, 0x08,
      0x10, 0x63, 0xC6, 0x08, 0x10, 0x30, 0x0c, 0x08,
      0x10, 0x18, 0x18, 0x08, 0x10, 0x00, 0x00, 0x08};

   GLubyte halftone[] = {
      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);
    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(350, 150);
    glutCreateWindow(argv[0]);
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}
```

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。

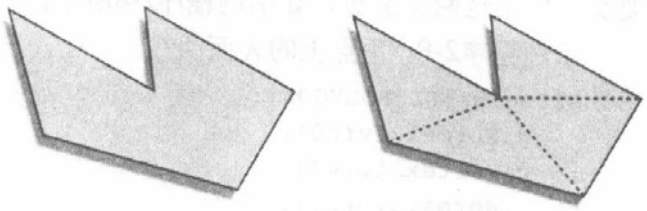


图2-12 细分一个非凸多边形

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。

兼容性扩展
glEdgeFlag

void **glEdgeFlag**(GLboolean *flag*);

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。

示例程序2-7 标记多边形边界边

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glBegin(GL_POLYGON);
    glEdgeFlag(GL_TRUE);
    glVertex3fv(V0);
    glEdgeFlag(GL_FALSE);
    glVertex3fv(V1);
    glEdgeFlag(GL_TRUE);
    glVertex3fv(V2);
glEnd();
```

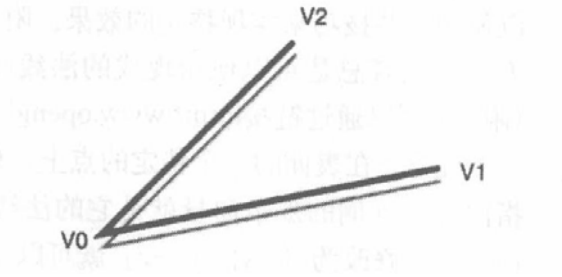


图2-13 使用边界标志绘制轮廓多边形

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();
```

兼容性扩展
glNormal

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：

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

兼容性扩展
GL_NORMALIZE
GL_RESCALE_NORMAL

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。

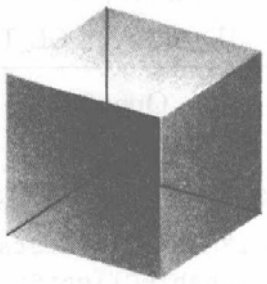


图2-14 6个面，8个共享顶点

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。

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。

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。

兼容性扩展
glEnableClientState

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：

```
glEnableClientState(GL_NORMAL_ARRAY);
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：

```
glDisableClientState(GL_NORMAL_ARRAY);
```

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。

兼容性扩展
glDisableClientState

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。

void **glVertexPointer**(GLint *size*, GLenum *type*, GLsizei *stride*,
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、

兼容性扩展
glVertexPointer

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：

void **glColorPointer**(GLint *size*, GLenum *type*, GLsizei *stride*, const GLvoid **pointer*);
void **glSecondaryColorPointer**(GLint *size*, GLenum *type*, GLsizei *stride*, const GLvoid **pointer*);
void **glIndexPointer**(GLenum *type*, GLsizei *stride*, const GLvoid **pointer*);
void **glNormalPointer**(GLenum *type*, GLsizei *stride*, const GLvoid **pointer*);
void **glFogCoordPointer**(GLenum type, GLsizei *stride*, const GLvoid **pointer*);
void **glTexCoordPointer**(GLint *size*, GLenum *type*, GLsizei *stride*, const GLvoid **pointer*);
void **glEdgeFlagPointer**(GLsizei *stride*, const GLvoid **pointer*);

兼容性扩展
glColorPointer
glSecondaryColorPointer
glIndexPointer
glNormalPointer
glFogCoordPointer
glTexCoordPointer
glEdgeFlagPointer

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	参数的值
glVertexPointer	2、3、4	GL_SHORT、GL_INT、GL_FLOAT、GL_DOUBLE
glColorPointer	3、4	GL_BYTE、GL_UNSIGNED_BYTE、GL_SHORT、GL_UNSIGNED_SHORT、GL_INT、GL_UNSIGNED_INT、GL_FLOAT、GL_DOUBLE
glSecondaryColorPointer	3	GL_BYTE、GL_UNSIGNED_BYTE、GL_SHORT、GL_UNSIGNED_SHORT、L_INT、GL_UNSIGNED_INT、GL_FLOAT、GL_DOUBLE
glIndexPointer	1	GL_UNSIGNED_BYTE、GL_SHORT、GL_INT、GL_FLOAT、GL_DOUBLE
glNormalPointer	3	GL_BYTE、GL_SHORT、GL_INT、GL_FLOAT、GL_DOUBLE
glFogCoordPointer	1	GL_FLOAT、GL_DOUBLE
glTexCoordPointer	1、2、3、4	GL_SHORT、GL_INT、GL_FLOAT、GL_DOUBLE
glEdgeFlagPointer	1	没有type参数（数据的类型必须是GL布尔型）

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

```
static GLint vertices[] = {25, 25,
                           100, 325,
                           175, 25,
                           175, 325,
                           250, 25,
                           325, 325};
static GLfloat colors[] = {1.0, 0.2, 0.2,
                           0.2, 0.2, 1.0,
                           0.8, 1.0, 0.2,
                           0.75, 0.75, 0.75,
                           0.35, 0.35, 0.35,
                           0.5, 0.5, 0.5};
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);

glColorPointer(3, GL_FLOAT, 0, colors);
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：

```
static GLfloat intertwined[] =
      {1.0, 0.2, 1.0, 100.0, 100.0, 0.0,
       1.0, 0.2, 0.2,   0.0, 200.0, 0.0,
       1.0, 1.0, 0.2, 100.0, 300.0, 0.0,
       0.2, 1.0, 0.2, 200.0, 300.0, 0.0,
       0.2, 1.0, 1.0, 300.0, 200.0, 0.0,
       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：

```
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）：

```
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

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

兼容性扩展
glArrayElement

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

```
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
glColorPointer(3, GL_FLOAT, 0, colors);
glVertexPointer(2, GL_INT, 0, vertices);

glBegin(GL_TRIANGLES);
glArrayElement(2);
glArrayElement(3);
glArrayElement(5);
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：

```
glBegin(GL_TRIANGLES);
glColor3fv(colors + (2 * 3));
glVertex2iv(vertices + (2 * 2));
glColor3fv(colors + (3 * 3));
glVertex2iv(vertices + (3 * 2));
glColor3fv(colors + (5 * 3));
glVertex2iv(vertices + (5 * 2));
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。

---

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：

```
glBegin(mode);
for (i = 0; i < count; i++)
   glArrayElement(indices[i]);
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。

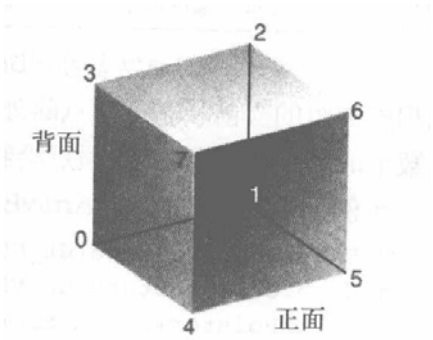


图2-15 对顶点进行了编号的立方体

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

```
static GLubyte frontIndices[] = {4, 5, 6, 7};
static GLubyte rightIndices[] = {1, 2, 6, 5};
static GLubyte bottomIndices[] = {0, 1, 5, 4};
static GLubyte backIndices[] = {0, 3, 2, 1};
static GLubyte leftIndices[] = {0, 4, 7, 3};
static GLubyte topIndices[] = {2, 3, 7, 6};

glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, frontIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, rightIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, bottomIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, backIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, leftIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, topIndices);
```

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

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

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

glDrawElements(GL_LINE_STRIP, 7, GL_UNSIGNED_BYTE, oneIndices);
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。

---

void **glMultiDrawElements**(GLenum *mode*, GLsizei **count*,
GLenum *type*, const GLvoid ***indices*,
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：

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

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

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

glMultiDrawElements(GL_LINE_STRIP, count, GL_UNSIGNED_BYTE,
                    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。

void **glDrawRangeElements**(GLenum *mode*, GLuint *start*,
GLuint *end*, GLsizei *count*,
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**()和同，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glDrawRangeElements()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。

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：

```
glBegin (mode);
for (i = 0; i < count; i++)
   glArrayElement(first + i);
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。

---

void **glMultiDrawArrays**(GLenum *mode*, GLint **first*, GLsizei **count*
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：

```
for (i = 0; i < primcount; i++) {
   if (count[i] > 0)
      glDrawArrays(mode, first[i], count[i]);
}
```

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。

---

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

```
#define BUFFER_OFFSET(offset) ((GLvoid *) NULL + offset)

#define XStart -0.8
#define XEnd 0.8
#define YStart -0.8
#define YEnd 0.8
```

```
#define NumXPoints              11
#define NumYPoints              11
#define NumPoints               (NumXPoints * NumYPoints)
#define NumPointsPerStrip       (2*NumXPoints)
#define NumStrips               (NumYPoints-1)
#define RestartIndex            0xffff

void
init()
{
    GLuint vbo, ebo;
    GLfloat *vertices;
    GLushort *indices;

    /* Set up vertex data */
    glGenBuffers(1, &vbo);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(GL_ARRAY_BUFFER, 2*NumPoints*sizeof(GLfloat),
        NULL, GL_STATIC_DRAW);
vertices = glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);

if (vertices == NULL) {
    fprintf(stderr, "Unable to map vertex buffer\n");
    exit(EXIT_FAILURE);
}
else {
    int i, j;
    GLfloat dx = (XEnd - XStart) / (NumXPoints - 1);
    GLfloat dy = (YEnd - YStart) / (NumYPoints - 1);
    GLfloat *tmp = vertices;
    int n = 0;

    for (j = 0; j < NumYPoints; ++j) {
        GLfloat y = YStart + j*dy;

        for (i = 0; i < NumXPoints; ++i) {
            GLfloat x = XStart + i*dx;
            *tmp++ = x;
            *tmp++ = y;
        }
    }

    glUnmapBuffer(GL_ARRAY_BUFFER);
    glVertexPointer(2, GL_FLOAT, 0, BUFFER_OFFSET(0));
    glEnableClientState(GL_VERTEX_ARRAY);
}

/* Set up index data */
glGenBuffers(1, &ebo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);

/* We allocate an extra restart index because it simplifies
```

```
** the element-array loop logic */
glBufferData( GL_ELEMENT_ARRAY_BUFFER,
    NumStrips*(NumPointsPerStrip+1)*sizeof(GLushort),
    NULL, GL_STATIC_DRAW );
indices = glMapBuffer(GL_ELEMENT_ARRAY_BUFFER,
    GL_WRITE_ONLY);

if (indices == NULL) {
    fprintf(stderr, "Unable to map index buffer\n");
    exit(EXIT_FAILURE);
}
else {
    int i, j;
    GLushort *index = indices;
    for (j = 0; j < NumStrips; ++j) {
        GLushort bottomRow = j*NumYPoints;
        GLushort topRow = bottomRow + NumYPoints;

        for (i = 0; i < NumXPoints; ++i) {
            *index++ = topRow + i;
            *index++ = bottomRow + i;
        }
        *index++ = RestartIndex;
    }

    glUnmapBuffer(GL_ELEMENT_ARRAY_BUFFER);
    }

    glPrimitiveRestartIndex(RestartIndex);
    glEnable(GL_PRIMITIVE_RESTART);
}

void
display()
{
    int i, start;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glColor3f(1, 1, 1);
    glDrawElements(GL_TRIANGLE_STRIP,
        NumStrips*(NumPointsPerStrip + 1),
        GL_UNSIGNED_SHORT, BUFFER_OFFSET(0));

    glutSwapBuffers();
}
```

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）。

void **glDrawArraysInstanced**(GLenum *mode*, GLint *first*, GLsizei *count*
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）：

```
for (i = 0; i < primcount; i++) {
   gl_InstanceID = i;
   glDrawArrays(mode, first, count);
}
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。

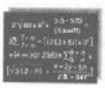
void **glDrawElementsInstanced**(GLenum *mode*, GLsizei *count*,
GLenum *type*, const void **indicies*,
GLsizei *primcount*);

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：

```
for (i = 0; i < primcount; i++) {
   gl_InstanceID = i;
   glDrawElements(mode, count, type, indicies);
}
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：

```
static GLfloat intertwined[] =
      {1.0, 0.2, 1.0, 100.0, 100.0, 0.0,
       1.0, 0.2, 0.2, 0.0, 200.0, 0.0,
       1.0, 1.0, 0.2, 100.0, 300.0, 0.0,
       0.2, 1.0, 0.2, 200.0, 300.0, 0.0,
       0.2, 1.0, 1.0, 300.0, 200.0, 0.0,
       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：

```
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e_t、e_c和e_n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。s_t、s_c和s_v分别表示纹理坐标、颜色和顶点数组的大小（成分的数量）。t_c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。p_c、p_n和p_v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	t_c	p_c	p_n	p_v	s
GL_V2F	F	F	F			2				0	2f
GL_V3F	F	F	F			3				0	3f
GL_C4UB_V2F	F	T	F		4	2	GL_UNSIGNED_BYTE	0		c	c+2f
GL_C4UB_V3F	F	T	F		4	3	GL_UNSIGNED_BYTE	0		c	c+3f
GL_C3F_V3F	F	T	F		3	3	GL_FLOAT	0		3f	6f
GL_N3F_V3F	F	F	T			3			0	3f	6f
GL_C4F_N3F_V3F	F	T	T		4	3	GL_FLOAT	0	4f	7f	10f
GL_T2F_V3F	T	F	F	2		3				2f	5f
GL_T4F_V4F	T	F	F	4		4				4f	8f
GL_T2F_C4UB_V3F	T	T	F	2	4	3	GL_UNSIGNED_BYTE	2f		c+2f	c+5f
GL_T2F_C3F_V3F	T	T	F	2	3	3	GL_FLOAT	2f		5f	8f
GL_T2F_N3F_V3F	T	F	T	2		3			2f	5f	8f
GL_T2F_C4F_N3F_V3F	T	T	T	2	4	3	GL_FLOAT	2f	6f	9f	12f
GL_T4F_C4F_N3F_V4F	T	T	T	4	4	4	GL_FLOAT	4f	8f	11f	15f

void **glInterleavedArrays**(GLenum *format*,
GLsizei *stride*, const GLvoid **pointer*)

兼容性扩展
glInterleavedArrays

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 e_t, e_c, e_n, s_t, s_c, s_v, t_c, p_c, p_n, p_v, and s
 * as a function of Table 2-5 and the value of format
 */

str = stride;
if (str == 0)
   str = s;

glDisableClientState(GL_EDGE_FLAG_ARRAY);
glDisableClientState(GL_INDEX_ARRAY);
glDisableClientState(GL_SECONDARY_COLOR_ARRAY);
glDisableClientState(GL_FOG_COORD_ARRAY);

if (e_t) {
   glEnableClientState(GL_TEXTURE_COORD_ARRAY);
   glTexCoordPointer(s_t, GL_FLOAT, str, pointer);
}
else
   glDisableClientState(GL_TEXTURE_COORD_ARRAY);
if (e_c) {
   glEnableClientState(GL_COLOR_ARRAY);
   glColorPointer(s_c, t_c, str, pointer+p_c);
}
else
   glDisableClientState(GL_COLOR_ARRAY);

if (e_n) {
   glEnableClientState(GL_NORMAL_ARRAY);
   glNormalPointer(GL_FLOAT, str, pointer+pn);
}
else
   glDisableClientState(GL_NORMAL_ARRAY);

glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(s_v, GL_FLOAT, str, pointer+p_v);
```

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。

---

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。

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。

void **glBufferData**(GLenum *target*, GLsizeiptr *size*, const GLvoid **data*,
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	含 义
GL_STREAM_DRAW	数据只指定1次，并且最多只有几次作为绘图和图像指定函数的源数据
GL_STREAM_READ	数据从一个OpenGL缓冲区复制而来，并且最多只有几次由应用程序作为数据值使用
GL_STREAM_COPY	数据从一个OpenGL缓冲区复制而来，并且最多只有几次作为绘图和图像指定函数的源数据
GL_STATIC_DRAW	数据只指定1次，但是可以多次作为绘图和图像指定函数的源数据
GL_STATIC_READ	数据从一个OpenGL缓冲区复制而来，并且可以多次由应用程序作为数据值使用
GL_STATIC_COPY	数据从一个OpenGL缓冲区复制而来，并且可以多次作为绘图和图像指定函数的源数据
GL_DYNAMIC_DRAW	数据可以多次指定，并且可以多次作为绘图和图像指定函数的源数据
GL_DYNAMIC_READ	数据可以多次从一个OpenGL缓冲区复制而来，并且可以多次由应用程序作为数据值使用
GL_DYNAMIC_COPY	数据可以多次从一个OpenGL缓冲区复制而来，并且可以多次作为绘图和图像指定函数的源数据

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。

void **glBufferSubData**(GLenum *target*, GLintptr *offset*, GLsizeiptr *size*,
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。

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。

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。

```
GLfloat* data;

data = (GLfloat*) glMapBuffer(GL_ARRAY_BUFFER, GL_READ_WRITE);

if (data != (GLfloat*) NULL) {
    for( i = 0; i < 8; ++i )
        data[3*i+2] *= 2.0;  /* Modify Z values */
    glUnmapBuffer(GL_ARRAY_BUFFER);
} else {
    /* Handle not being able to update data */
}
```

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。

GLvoid ***glMapBufferRange**(GLenum *target*, GLintptr *offset*,
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 义
GL_MAP_INVALIDATE_RANGE_BIT	说明映射范围内之前的值可以丢弃，但保留缓冲区中其他的值。这个范围中的数据是未定义的，除非明确地写入。如果随后的OpenGL调用访问未定义的数据，不会产生OpenGL错误，但是，这样调用的结果是未定义的（但可能引发应用错误或系统错误）。这个标志不能和GL_READ_BIT一起使用
GL_MAP_INVALIDATE_BUFFER_BIT	说明整个缓冲区中之前的值都可以丢弃，并且，缓冲区中的所有值都是未定义的，除非明确地写入。如果随后的OpenGL调用访问未定义的数据，不会产生OpenGL错误，但是，这样的调用的结果是未定义的（但可能引发应用错误或系统错误）。这个标志不能和GL_READ_BIT一起使用
GL_MAP_FLUSH_EXPLICIT_BIT	表示映射区域可能更新的不连续的范围，当对一个范围的修改被认为是完成了，应用程序应该调用glFlushMappedBufferRange()发出信号。如果映射缓冲区的一个范围更新了，却没有刷新，这些值是未定义的，直到刷新为止
GL_MAP_UNSYNCHRONIZED_BIT	使用这个选项将要求任何修改的范围都显式地刷新到OpenGL服务器，glUnmapBuffer()不会自动刷新缓冲区的数据 说明OpenGL不应该试图同步在缓冲区上的待执行的操作（例如，通过调用glBufferData()更新数据，或者应用程序试图使用缓冲区中的数据来渲染），直到glMapBufferRange()调用完成。访问或修改映射区域的那些待执行操作不会导致OpenGL错误，但是这些操作的结果是未定义的

GLvoid **glFlushMappedBufferRange**(GLenum *target*, GLintptr *offset*,GLsizeiptr *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。

---

void **glCopyBufferSubData**(GLenum *readbuffer*, GLenum *writebuffer*,
GLintptr *readoffset*, GLintptr *writeoffset*,
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。

---

void **glDelcteBuffers**(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：

---

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。

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

```
#define BUFFER_OFFSET(offset) ((GLvoid*) NULL + offset)
#define NumberOf(array)       (sizeof(array)/sizeof(array[0]))

typedef struct {
    GLfloat x, y, z;
} vec3;

typedef struct {
    vec3 xlate; /* Translation */
    GLfloat angle;
    vec3 axis;
} XForm;

enum { Cube, Cone, NumVAOs };
GLuint VAO[NumVAOs];
GLenum PrimType[NumVAOs];
GLsizei NumElements[NumVAOs];
XForm Xform[NumVAOs] = {
    { { -2.0, 0.0, 0.0 }, 0.0, { 0.0, 1.0, 0.0 } },
    { { 0.0, 0.0, 2.0 }, 0.0, { 1.0, 0.0, 0.0 } }
};
GLfloat Angle = 0.0;

void
init()
{
    enum { Vertices, Colors, Elements, NumVBOs };
    GLuint buffers[NumVBOs];

    glGenVertexArrays(NumVAOs, VAO);

    {
        GLfloat cubeVerts[][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 },
        };
        GLfloat cubeColors[][3] = {
```

```
            { 0.0, 0.0, 0.0 },
            { 0.0, 0.0, 1.0 },
            { 0.0, 1.0, 0.0 },
            { 0.0, 1.0, 1.0 },
            { 1.0, 0.0, 0.0 },
            { 1.0, 0.0, 1.0 },
            { 1.0, 1.0, 0.0 },
            { 1.0, 1.0, 1.0 },
        };

        GLubyte cubeIndices[] = {
            0, 1, 3, 2,
            4, 6, 7, 5,
            2, 3, 7, 6,
            0, 4, 5, 1,
            0, 2, 6, 4,
            1, 5, 7, 3
        };
        glBindVertexArray(VAO[Cube]);
        glGenBuffers(NumVBOs, buffers);
        glBindBuffer(GL_ARRAY_BUFFER, buffers[Vertices]);
        glBufferData(GL_ARRAY_BUFFER, sizeof(cubeVerts),
            cubeVerts, GL_STATIC_DRAW);
        glVertexPointer(3, GL_FLOAT, 0, BUFFER_OFFSET(0));
        glEnableClientState(GL_VERTEX_ARRAY);

        glBindBuffer(GL_ARRAY_BUFFER, buffers[Colors]);
        glBufferData(GL_ARRAY_BUFFER, sizeof(cubeColors),
            cubeColors, GL_STATIC_DRAW);
        glColorPointer(3, GL_FLOAT, 0, BUFFER_OFFSET(0));
        glEnableClientState(GL_COLOR_ARRAY);

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,
            buffers[Elements]);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER,
            sizeof(cubeIndices), cubeIndices, GL_STATIC_DRAW);

        PrimType[Cube] = GL_QUADS;
        NumElements[Cube] = NumberOf(cubeIndices);
}
{
        int i, idx;
        float dTheta;
#define NumConePoints 36
        /* We add one more vertex for the cone's apex */
        GLfloat coneVerts[NumConePoints+1][3] = {
            {0.0, 0.0, 1.0}
        };
        GLfloat coneColors[NumConePoints+1][3] = {
            {1.0, 1.0, 1.0}
        };
        GLubyte coneIndices[NumConePoints+1];

        dTheta = 2*M_PI / (NumConePoints - 1);
```

```
        idx = 1;
        for (i = 0; i < NumConePoints; ++i, ++idx) {
            float theta = i*dTheta;
            coneVerts[idx][0] = cos(theta);
            coneVerts[idx][1] = sin(theta);
            coneVerts[idx][2] = 0.0;

            coneColors[idx][0] = cos(theta);
            coneColors[idx][1] = sin(theta);
            coneColors[idx][2] = 0.0;

            coneIndices[idx] = idx;
        }

        glBindVertexArray(VAO[Cone]);
        glGenBuffers(NumVBOs, buffers);
        glBindBuffer(GL_ARRAY_BUFFER, buffers[Vertices]);
        glBufferData(GL_ARRAY_BUFFER, sizeof(coneVerts),
           coneVerts, GL_STATIC_DRAW);
        glVertexPointer(3, GL_FLOAT, 0, BUFFER_OFFSET(0));
        glEnableClientState(GL_VERTEX_ARRAY);

        glBindBuffer(GL_ARRAY_BUFFER, buffers[Colors]);
        glBufferData(GL_ARRAY_BUFFER, sizeof(coneColors),
           coneColors, GL_STATIC_DRAW);
        glColorPointer(3, GL_FLOAT, 0, BUFFER_OFFSET(0));
        glEnableClientState(GL_COLOR_ARRAY);

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,
           buffers[Elements]);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER,
           sizeof(coneIndices), coneIndices, GL_STATIC_DRAW);

        PrimType[Cone] = GL_TRIANGLE_FAN;
        NumElements[Cone] = NumberOf(coneIndices);
    }
    glEnable(GL_DEPTH_TEST);
}
void
display()
{
    int i;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPushMatrix();
    glRotatef(Angle, 0.0, 1.0, 0.0);

    for (i = 0; i < NumVAOs; ++i) {
        glPushMatrix();
        glTranslatef(Xform[i].xlate.x, Xform[i].xlate.y,
            Xform[i].xlate.z);
        glRotatef(Xform[i].angle, Xform[i].axis.x,
            Xform[i].axis.y, Xform[i].axis.z);
        glBindVertexArray(VAO[i]);
        glDrawElements(PrimType[i], NumElements[i],
```

```
            GL_UNSIGNED_BYTE, BUFFER_OFFSET(0));
        glPopMatrix();
    }
    glPopMatrix();
    glutSwapBuffers();
}
```

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。

---

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。

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。）

兼容性扩展

glPushAttrib

glPopAttrib

GL_ACCUM_BUFFER_BIT

GL_ALL_ATTRIB_BITS

GL_COLOR_BUFFER_BIT

GL_CURRENT_BIT

GL_DEPTH_BUFFER_BIT

GL_ENABLE_BIT

GL_EVAL_BIT

GL_FOG_BIT

GL_HINT_BIT

GL_LIGHTING_BIT

GL_LINE_BIT

GL_LIST_BIT

GL_MULTISAMPLE_BIT

GL_PIXEL_MODE_BIT

GL_POINT_BIT

GL_POLYGON_BIT

GL_POLYGON_STIPPLE_BIT

GL_SCISSOR_BIT

GL_STENCIL_BUFFER_BIT

GL_TEXTURE_BIT

GL_TRANSFORM_BIT

GL_VIEWPORT_BIT

void **glPushAttrib**(GLbitfield *mask*);

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

void **glPushClientAttrib**(GLbitfield *mask*);

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。

兼容性扩展
glPushClientAttrib
glPopClientAttrib
GL_CLIENT_PIXEL_STORE_BIT
GL_CLIENT_VERTEX_ARRAY_BIT
GL_CLIENT_ALL_ATTRIB_BITS

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

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。
- 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。

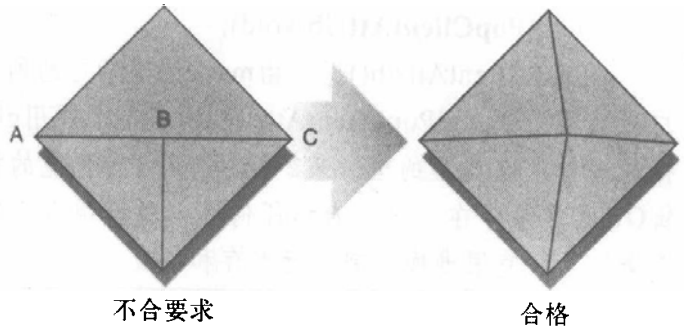


图2-16 修改不合要求的T交叉

```
/* don't use this code */
#define PI 3.14159265
#define EDGES 30

/* draw a circle */
glBegin(GL_LINE_STRIP);
for (i = 0; i <= EDGES; i++)
    glVertex2f(cos((2*PI*i)/EDGES), sin((2*PI*i)/EDGES));
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）。

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

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

```
void normalize(float v [3])
{
   GLfloat d =sqrt(v [0]*v [0]+v [1]*v [1]+v [2]*v [2]);
   if (d ==0.0){
      error("zero length vector ");
```

```
        return;
    }
    v [0] /=d;
    v [1] /=d;
    v [2] /=d;
}
void normcrossprod(float v1 [3],float v2 [3] ,float out [3])
{
    out [0] = v1 [1]*v2 [2] -v1 [2]*v2 [1];
    out [1] = v1 [2]*v2 [0] -v1 [0]*v2 [2];
    out [2] = v1 [0]*v2 [1] -v1 [1]*v2 [0];
    normalize(out);
}
```

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）。

```
glBegin(GL_TRIANGLES);
for (i = 0; i < 20; i++) {
      glNormal3fv(&vdata[tindices[i][0]][0]);
      glVertex3fv(&vdata[tindices[i][0]][0]);
      glNormal3fv(&vdata[tindices[i][1]][0]);
      glVertex3fv(&vdata[tindices[i][1]][0]);
      glNormal3fv(&vdata[tindices[i][2]][0]);
      glVertex3fv(&vdata[tindices[i][2]][0]);
} 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。

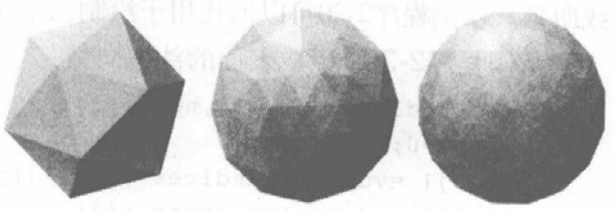


图2-17 通过细分改善多边形近似模拟法对表面的模拟精度

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

```
void drawtriangle(float *v1,float *v2,float *v3)
{
   glBegin(GL_TRIANGLES);
      glNormal3fv(v1);
      glVertex3fv(v1);
      glNormal3fv(v2);
```

```
        glVertex3fv(v2);
        glNormal3fv(v3);
        glVertex3fv(v3);
    glEnd();
    }

void subdivide(float *v1,float *v2,float *v3)
{
    GLfloat v12 [3],v23 [3],v31 [3];
    GLint i;

    for (i =0;i <3;i++){
        v12 [i] =(v1 [i]+v2 [i])/2.0;
        v23 [i] =(v2 [i]+v3 [i])/2.0;
        v31 [i] =(v3 [i]+v1 [i])/2.0;
    }
    normalize(v12);
    normalize(v23);
    normalize(v31);
    drawtriangle(v1,v12,v31);
    drawtriangle(v2,v23,v12);
    drawtriangle(v3,v31,v23);
    drawtriangle(v12,v23,v31);
}

for (i =0;i <20;i++){
    subdivide(&vdata [tindices [i][0]][0],
              &vdata [tindices [i][1]][0],
              &vdata [tindices [i][2]][0]);
}
```

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

```
void subdivide(float *v1,float *v2,float *v3,long depth)
{
    GLfloat v12 [3],v23 [3],v31 [3];
    GLint i;

    if (depth ==0){
        drawtriangle(v1,v2,v3);
        return;
    }
    for (i =0;i <3;i++){
        v12 [i] =(v1 [i]+v2 [i])/2.0;
        v23 [i] =(v2 [i]+v3 [i])/2.0;
        v31 [i] =(v3 [i]+v1 [i])/2.0;
    }
    normalize(v12);
    normalize(v23);
    normalize(v31);
    subdivide(v1,v12,v31,depth-1);
```

```
    subdivide(v2,v23,v12,depth-1);
    subdivide(v3,v31,v23,depth-1);
    subdivide(v12,v23,v31,depth-1);
}
```

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：

```
void surf(GLfloat u[2], GLfloat vertex[3], GLfloat normal[3]);
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

```
void subdivide(float u1 [2] ,float u2 [2],float u3 [2],
               float cutoff,long depth)
{
   GLfloat v1 [3],v2 [3],v3 [3],n1 [3],n2 [3],n3 [3];
   GLfloat u12 [2],u23 [2],u32 [2];
   GLint i;

   if (depth == maxdepth ||(curv(u1)<cutoff &&
       curv(u2)<cutoff &&curv(u3)<cutoff)){
      surf(u1,v1,n1);
      surf(u2,v2,n2);
      surf(u3,v3,n3);
      glBegin(GL_POLYGON);
         glNormal3fv(n1);glVertex3fv(v1);
         glNormal3fv(n2);glVertex3fv(v2);
         glNormal3fv(n3);glVertex3fv(v3);
      glEnd();
      return;
   }

   for (i =0;i <2;i++){
      u12 [i] =(u1 [i] +u2 [i])/2.0;
      u23 [i] =(u2 [i] +u3 [i])/2.0;
      u31 [i] =(u3 [i] +u1 [i])/2.0;
   }
   subdivide(u1,u12,u31,cutoff,depth+1);
   subdivide(u2,u23,u12,cutoff,depth+1);
   subdivide(u3,u31,u23,cutoff,depth+1);
   subdivide(u12,u23,u31,cutoff,depth+1);
}
```

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$\mathbf{M}$，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。

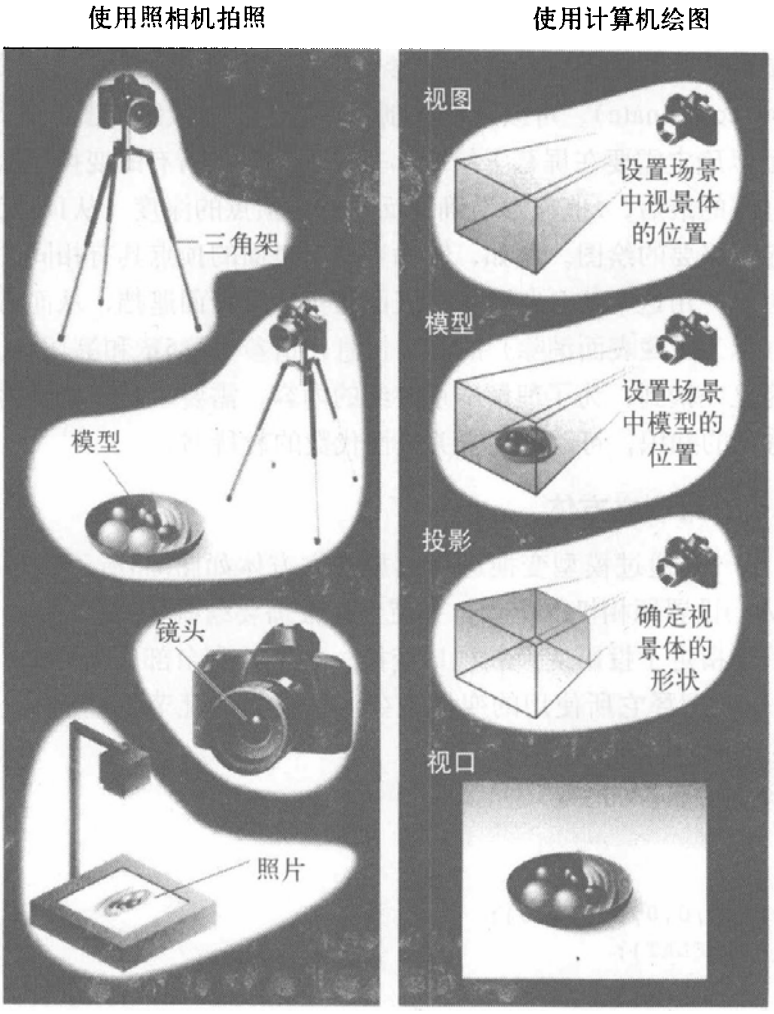


图3-1　照相机比喻

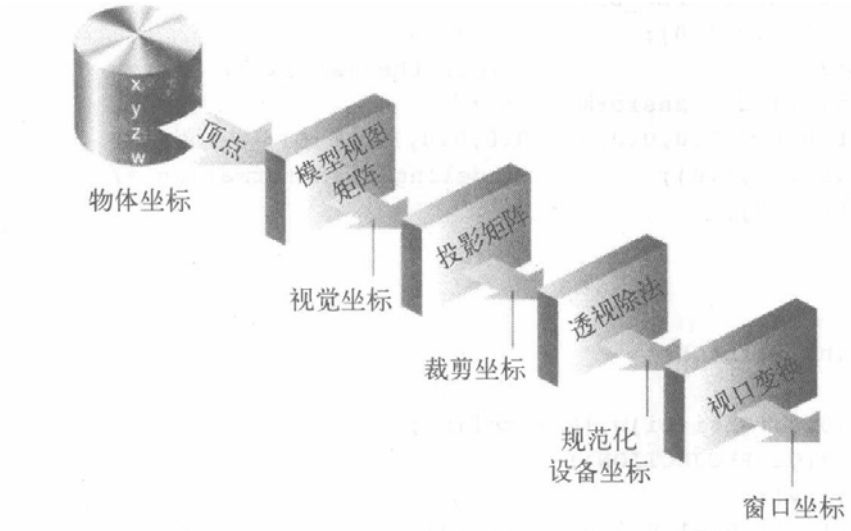


图3-2　顶点变换的步骤

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（perspective

division)，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。

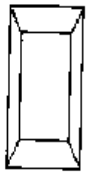


图3-3 变换立方体

示例程序3-1 变换立方体：cube.c

```
void init(void)
{
   glClearColor(0.0,0.0,0.0,0.0);
   glShadeModel(GL_FLAT);
}
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0,1.0,1.0);
   glLoadIdentity();              /*clear the matrix */
          /*viewing transformation */
   gluLookAt(0.0,0.0,5.0,0.0,0.0,0.0,0.0,1.0,0.0);
   glScalef(1.0,2.0,1.0);         /*modeling transformation */
   glutWireCube(1.0);
   glFlush();
}

void reshape(int w,int h)
{
   glViewport(0,0,(GLsizei)w,(GLsizei)h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glFrustum(-1.0,1.0,-1.0,1.0,1.5,20.0);
   glMatrixMode(GL_MODELVIEW);
}
int main(int argc,char**argv)
{
   glutInit(&argc,argv);
```

```
    glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB);
    glutInitWindowSize(500,500);
    glutInitWindowPosition(100,100);
    glutCreateWindow(argv [0]);
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}
```

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）。

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。

兼容性扩展
glMatrixMode
GL_MODELVIEW
GL_PROJECTION
GL_TEXTURE

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。

void **glLoadIdentity**(void);

把当前的可修改矩阵设置为4×4的单位矩阵。

兼容性扩展
glLoadIdentity

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。

void **glLoadMatrix**{fd}(const *TYPE* *m);
把当前矩阵的16个值设置为m指定的值。

兼容性扩展
glLoadMatrix
glMultMatrix

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（$m_1, m_2, \cdots, m_{16}$），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glMultTransposeMatrix*()，它们使用行主序（标准C约定）的矩阵作为参数。

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*(m^T)具有相同的效果。

兼容性扩展
glLoadTranspose Matrix
glMultTranspose Matrix

void **glMultTransposeMatrix**{fd}(const *TYPE* *m);
把参数m指定的16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*(m^T)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。

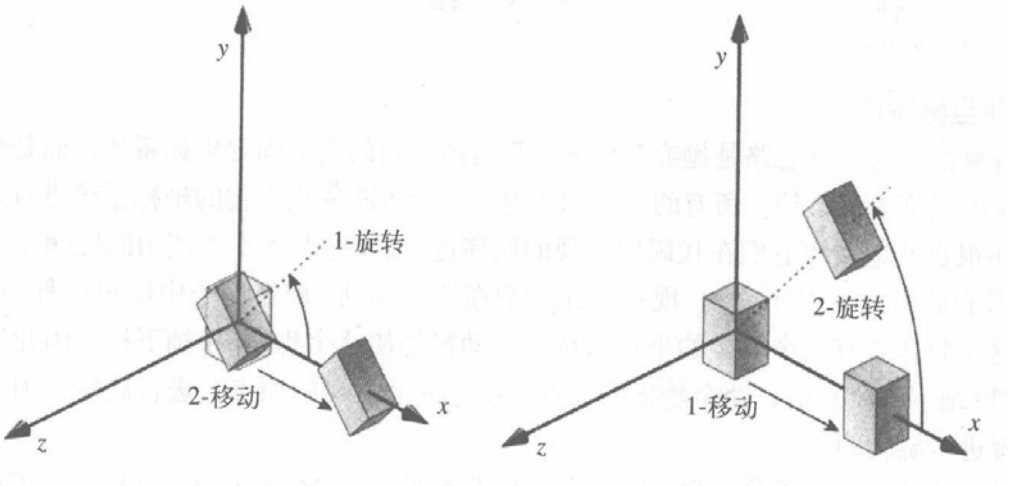


图3-4　首先旋转或首先移动

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：

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrixf(N);                    /* apply transformation N */
glMultMatrixf(M);                    /* apply transformation M */
glMultMatrixf(L);                    /* apply transformation L */
glBegin(GL_POINTS);
glVertex3f(v);                       /* draw transformed vertex v */
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）：

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrixf(T);                    /* translation */
glMultMatrixf(R);                    /* rotation */
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

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）。

兼容性扩展
glTranslate

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。

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。

兼容性扩展
glRotate

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。

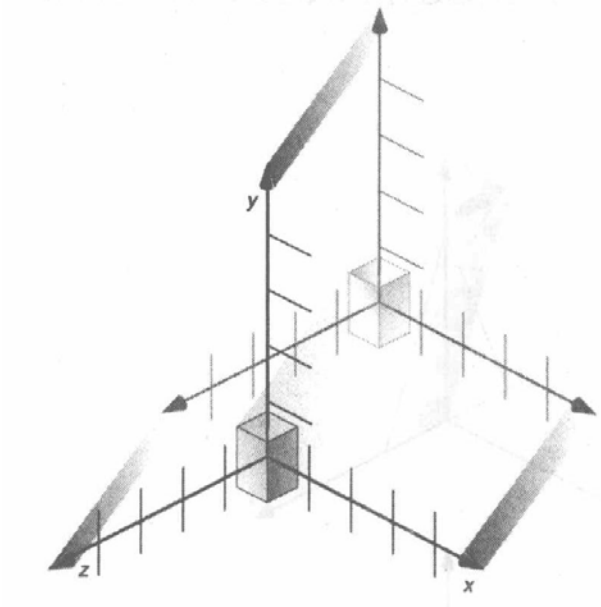


图3-5　移动物体

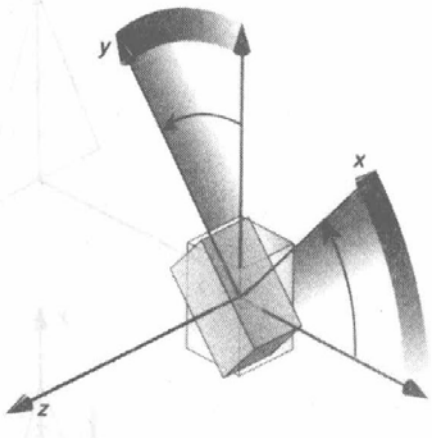


图3-6　旋转物体

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。

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。

兼容性扩展
glScale

图3-7显示了glScalef(2.0, −0.5, 1.0)的效果。

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。

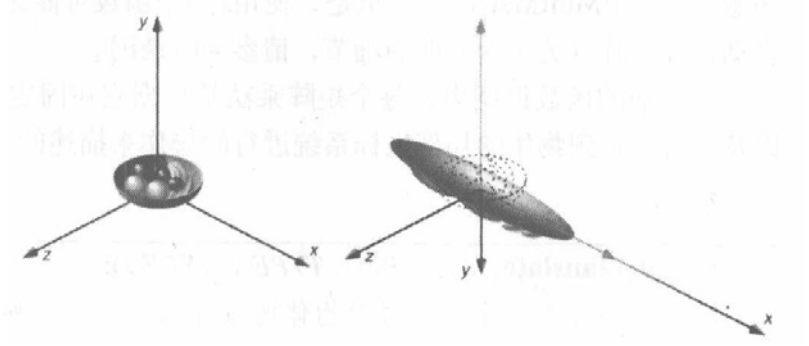


图3-7 缩放和反射物体

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：

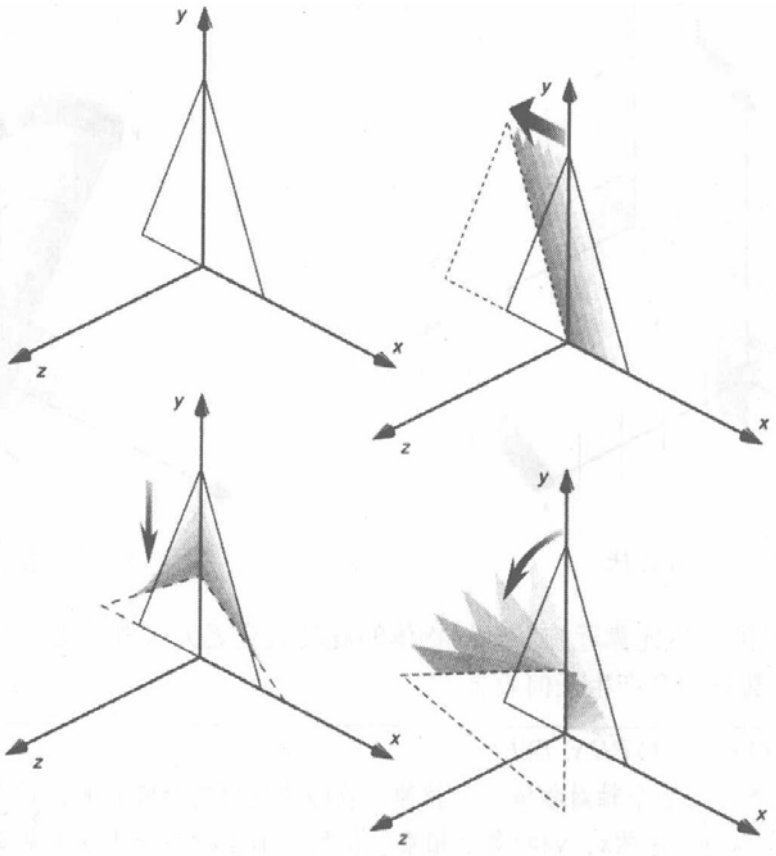


图3-8 模型变换例子

- 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

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*()

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）。

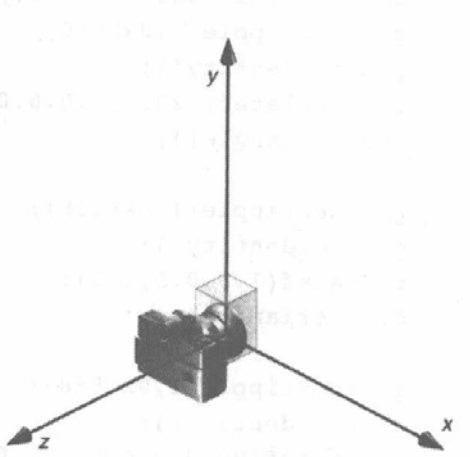


图3-9 原点的物体和观察点

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：

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

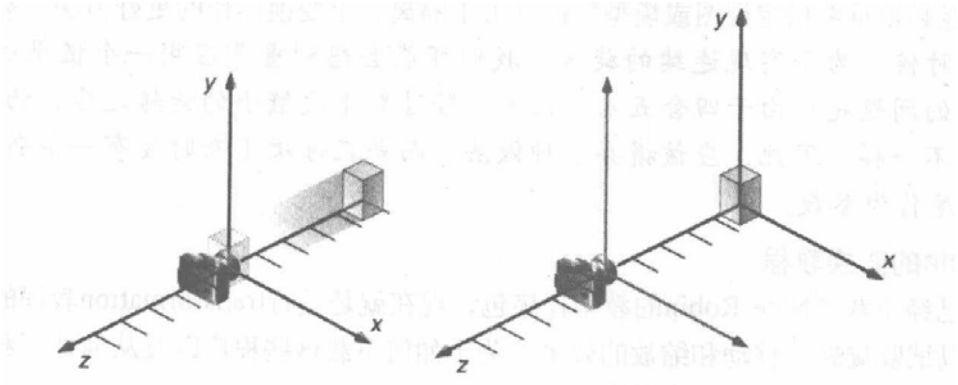


图3-10 分离观察点和物体

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：

```
glTranslatef(0.0, 0.0, -5.0);
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。

---

void **gluLookAt**(GLdouble *eyex*, GLdouble *eyey*, GLdouble *eyez*,
GLdouble *centerx*, GLdouble *centery*, GLdouble *centerz*,
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：

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

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：

```
gluLookAt(4.0, 2.0, 1.0, 2.0, 4.0, -3.0, 2.0, 2.0, -1.0);
```

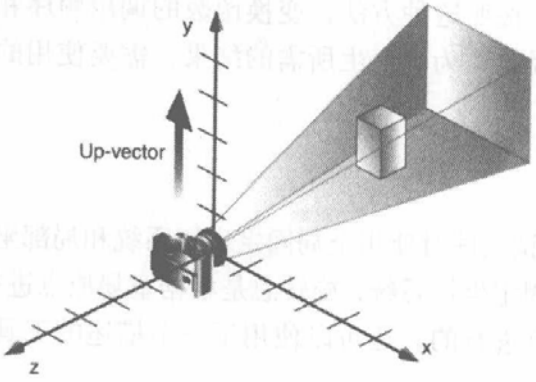


图3-11 默认的照相机位置

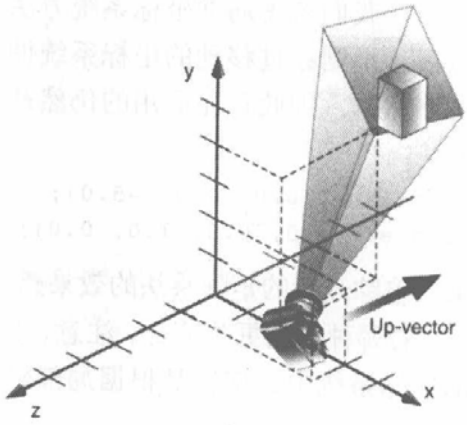


图3-12 使用gluLookAt()

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。

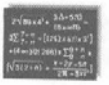


高级话题

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。

```
void pilotView{GLdouble planex, GLdouble planey,
               GLdouble planez, GLdouble roll,
               GLdouble pitch, GLdouble heading)
{
      glRotated(roll, 0.0, 0.0, 1.0);
      glRotated(pitch, 0.0, 1.0, 0.0);
      glRotated(heading, 1.0, 0.0, 0.0);
      glTranslated(-planex, -planey, -planez);
}
```

- 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。

```
void polarView{GLdouble distance, GLdouble twist,
               GLdouble elevation, GLdouble azimuth)
{
      glTranslated(0.0, 0.0, -distance);
      glRotated(-twist, 0.0, 0.0, 1.0);
      glRotated(-elevation, 1.0, 0.0, 0.0);
      glRotated(azimuth, 0.0, 0.0, 1.0);
}
```

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：

```
glMatrixMode(GL_PROJECTION);
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。

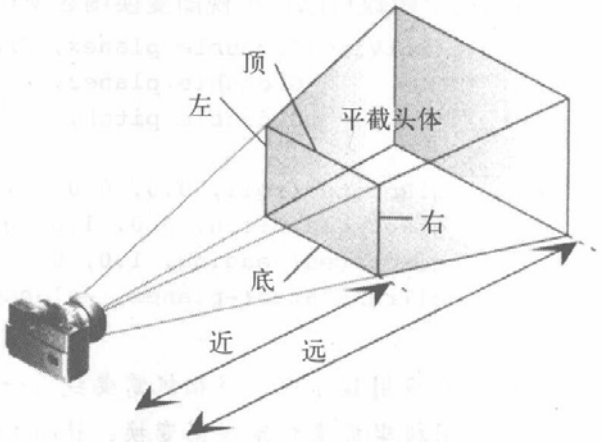


图3-13 glFrustum()函数所指定的透视视景体

兼容性扩展
glFrustum

void **glFrustum**(GLdouble *left*, GLdouble *right*,
GLdouble *bottom*, GLdouble *top*,
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。

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）。

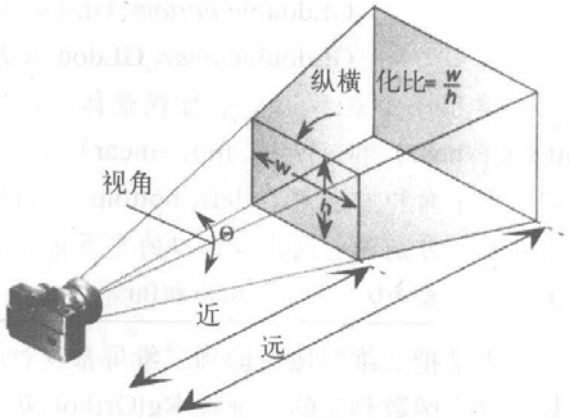


图3-14　由gluPerspective()所指定的透视视景体

void **gluPerspective**(GLdouble *fovy*, GLdouble *aspect*,
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。

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。

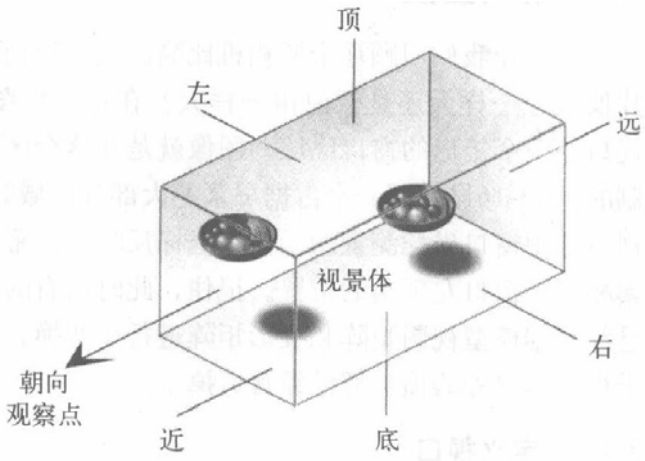


图3-15　正投影下的视景体

兼容性扩展
glOrtho

void **glOrtho**(GLdouble *left*, GLdouble *right*,
GLdouble *bottom*, GLdouble *top*,
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。

void **gluOrtho2D**(GLdouble *left*, GLdouble *right*,
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。

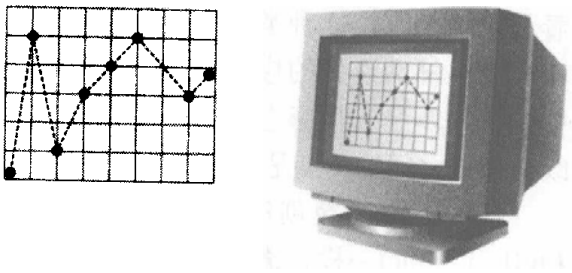


图3-16 视口矩形

3.4.1 定义视口

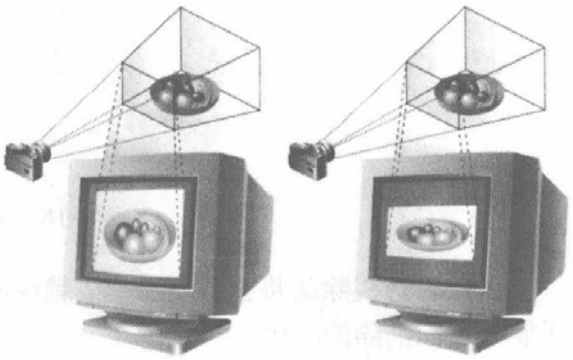
在屏幕上打开窗口的任务是由窗口系统而不是OpenGL负责的。但是，在默认情况下，视口被设

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。

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。



未扭曲的　　扭曲的

图3-17　把视景体映射到视口

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：

```
gluPerspective(fovy, 1.0, near, far);
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：

```
gluPerspective(fovy, 1.0, near, far);
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：

```
gluPerspective(fovy, 2.0, near, far);
glViewport(0, 0, 400, 200);
```



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：

```
glViewport(0, 0, sizex/2, sizey);
                    .
                    .
                    .
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。

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）。

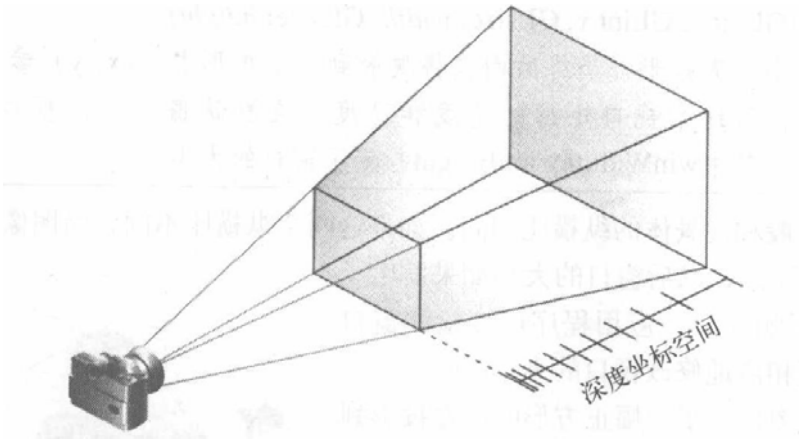


图3-18 透视投影和变换后的深度坐标

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。

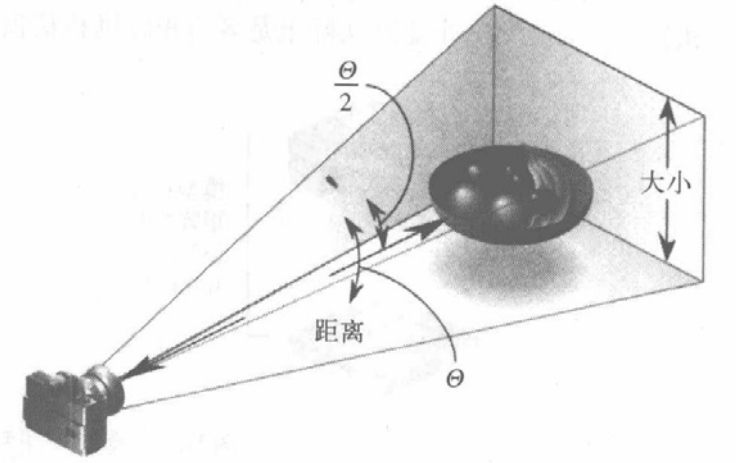


图3-19　使用三角函数计算视野

示例程序3-3　计算视野

```
#define PI 3.1415926535

double calculateAngle(double size,double distance)
{
    double radtheta,degtheta;
    radtheta =2.0 *atan2 (size/2.0,distance);
    degtheta =(180.0 *radtheta)/PI;
    return degtheta;
}
```

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$-1 \leqslant x \leqslant 3$、$5 \leqslant y \leqslant 7$和$-5 \leqslant z \leqslant 5$，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）。

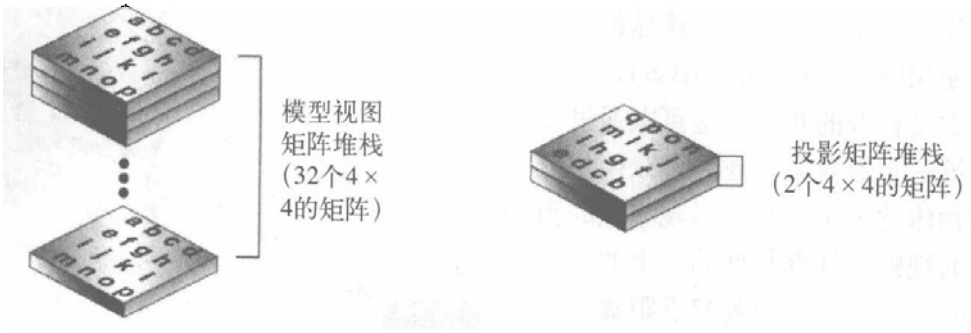


图3-20 模型视图和投影矩阵堆栈

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，glPush Matrix()表示“记住自己的位置”，glPopMatrix()表示“回到原来的位置”。

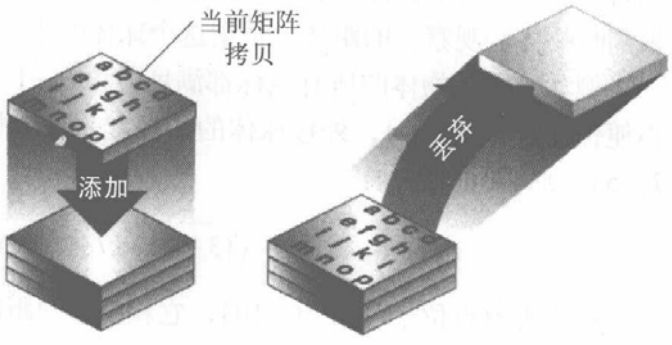


图3-21 在矩阵堆栈中进行压入和弹出

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。

兼容性扩展
glPushMatrix
glPopMatrix

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

```
draw_wheel_and_bolts()
{
   int i;
   draw_wheel();
   for(i=0;i<5;i++){
      glPushMatrix();
      glRotatef(72.0*i,0.0,0.0,1.0);
      glTranslatef(3.0,0.0,0.0);
        draw_bolt();
      glPopMatrix();
   }
}

draw_body_and_wheel_and_bolts()
{
   draw_car_body();
   glPushMatrix();
      glTranslatef(40,0,30);    /*move to first wheel position*/
      draw_wheel_and_bolts();
   glPopMatrix();
   glPushMatrix();
      glTranslatef(40,0,-30);   /*move to 2nd wheel position*/
      draw_wheel_and_bolts();
   glPopMatrix();
   ...                          /*draw last two wheels similarly*/
}
```

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。

```
glMatrixMode(GL_PROJECTION);
glPushMatrix();                    /*save the current projection*/
     glLoadIdentity();
     glOrtho(...);                  /*set up for displaying help*/
     display_the_help();
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。

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。

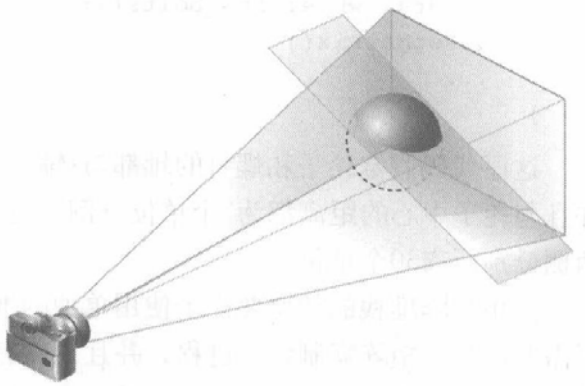


图3-22 其他裁剪平面和视景体

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）

$M^{-1}(x_e\ y_e\ z_e\ w_e)^T \geq 0$的所有视觉坐标（$x_e\ y_e\ z_e\ w_e$）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。

兼容性扩展
glClipPlane
GL_CLIP_DISTANCE*i*
GL_CLIP_PLANE*i*
GL_MAX_CLIP_PLANES

我们需要启用每个被定义的裁剪平面：

```
glEnable(GL_CLIP_PLANEi);
```

也可以用下面这个函数禁用一个裁剪平面：

```
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。

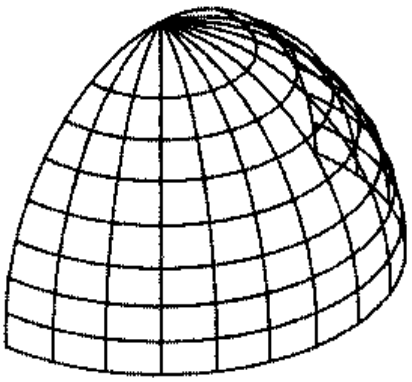


图3-23　裁剪后的线框球体

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。

示例程序3-5　经过两个裁剪平面裁剪的线框球体：clip.c

```
void init(void)
{
   glClearColor(0.0,0.0,0.0,0.0);
   glShadeModel(GL_FLAT);
}
void display(void)
{
   GLdouble eqn [4]={0.0,1.0,0.0,0.0};
   GLdouble eqn2 [4] ={1.0,0.0,0.0,0.0};

   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0,1.0,1.0);
   glPushMatrix();
   glTranslatef(0.0,0.0,-5.0);

/*   clip lower half --y <0 */
   glClipPlane(GL_CLIP_PLANE0,eqn);
   glEnable(GL_CLIP_PLANE0);
/*   clip left half --x <0 */
   glClipPlane(GL_CLIP_PLANE1,eqn2);
   glEnable(GL_CLIP_PLANE1);

   glRotatef(90.0,1.0,0.0,0.0);
```

```
    glutWireSphere(1.0,20,16);
    glPopMatrix();
    glFlush();
}

void reshape(int w,int h)
{
    glViewport(0,0,(GLsizei)w,(GLsizei)h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0,(GLfloat)w/(GLfloat)h,1.0,20.0);
    glMatrixMode(GL_MODELVIEW);
}

int main(int argc,char**argv)
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB);
    glutInitWindowSize(500,500);
    glutInitWindowPosition(100,100);
    glutCreateWindow(argv [0]);
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}
```



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glTranslate*()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。

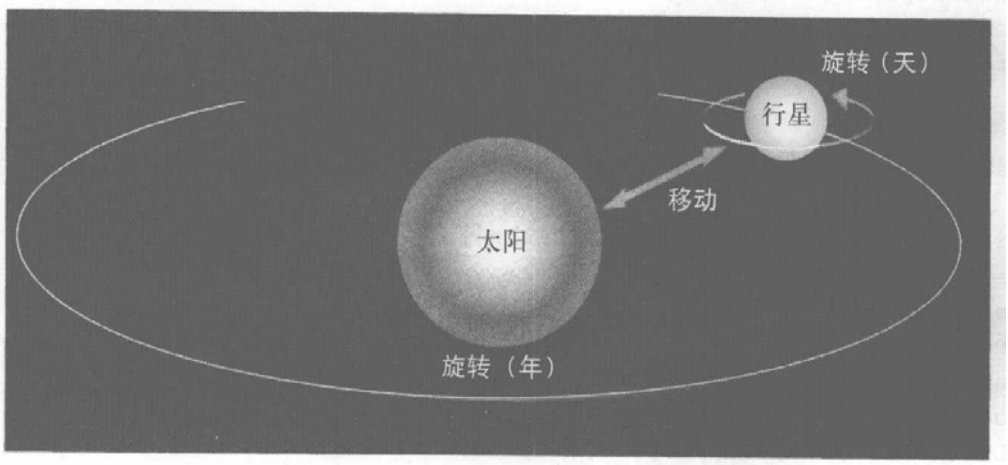


图3-24　行星和太阳

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。

```
glPushMatrix();
glutWireSphere(1.0, 20, 16);      /* draw sun */
glRotatef((GLfloat) year, 0.0, 1.0, 0.0);
glTranslatef(2.0, 0.0, 0.0);
glRotatef((GLfloat) day, 0.0, 1.0, 0.0);
glutWireSphere(0.2, 10, 8);       /* draw smaller planet */
glPopMatrix();
```

示例程序3-6　行星系统：planet.c

```
static int year =0,day =0;

void init(void)
{
   glClearColor(0.0,0.0,0.0,0.0);
   glShadeModel(GL_FLAT);
}
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0,1.0,1.0);
   glPushMatrix();
   glutWireSphere(1.0,20,16);          /*draw sun */
   glRotatef((GLfloat)year,0.0,1.0,0.0);
```

```
    glTranslatef(2.0,0.0,0.0);
    glRotatef((GLfloat)day,0.0,1.0,0.0);
    glutWireSphere(0.2,10,8);            /*draw smaller planet */
    glPopMatrix();
    glutSwapBuffers();
}

void reshape(int w,int h)
{
    glViewport(0,0,(GLsizei)w,(GLsizei)h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0,(GLfloat)w/(GLfloat)h,1.0,20.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0.0,0.0,5.0,0.0,0.0,0.0,0.0,1.0,0.0);
}

void keyboard(unsigned char key,int x,int y)
{
    switch (key){
       case 'd ':
          day =(day +10)%360;
          glutPostRedisplay();
          break;
       case 'D ':
          day =(day -10)%360;
          glutPostRedisplay();
          break;
       case 'y ':
          year =(year +5)%360;
          glutPostRedisplay();
          break;
       case 'Y ':
          year =(year -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;
}
```



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。

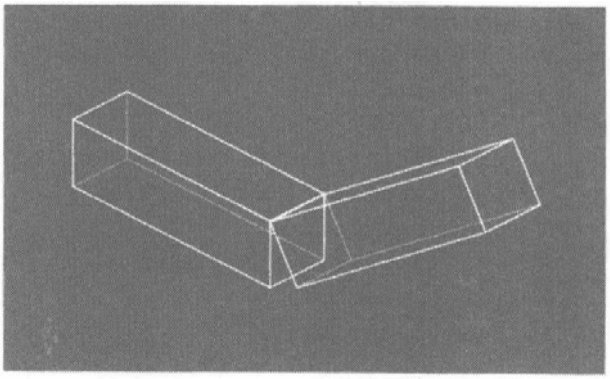


图3-25　机器人手臂

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glTranslate*()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。

```
glTranslatef(-1.0, 0.0, 0.0);
glRotatef((GLfloat) shoulder, 0.0, 0.0, 1.0);
glTranslatef(1.0, 0.0, 0.0);
glPushMatrix();
glScalef(2.0, 0.4, 1.0);
glutWireCube(1.0);
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）。

```
glTranslatef(1.0, 0.0, 0.0);
glRotatef((GLfloat) elbow, 0.0, 0.0, 1.0);
glTranslatef(1.0, 0.0, 0.0);
glPushMatrix();
glScalef(2.0, 0.4, 1.0);
glutWireCube(1.0);
glPopMatrix();
```

示例程序3-7 机器人手臂：robot.c

```
static int shoulder =0,elbow =0;

void init(void)
{
   glClearColor(0.0,0.0,0.0,0.0);
   glShadeModel(GL_FLAT);
}
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glPushMatrix();
   glTranslatef(-1.0,0.0,0.0);
   glRotatef((GLfloat)shoulder,0.0,0.0,1.0);
   glTranslatef(1.0,0.0,0.0);
   glPushMatrix();
   glScalef(2.0,0.4,1.0);
   glutWireCube(1.0);
   glPopMatrix();
   glTranslatef(1.0,0.0,0.0);
   glRotatef((GLfloat)elbow,0.0,0.0,1.0);
   glTranslatef(1.0,0.0,0.0);
   glPushMatrix();
   glScalef(2.0,0.4,1.0);
   glutWireCube(1.0);
   glPopMatrix();

   glPopMatrix();
   glutSwapBuffers();
}

void reshape(int w,int h)
{
   glViewport(0,0,(GLsizei)w,(GLsizei)h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(65.0,(GLfloat)w/(GLfloat)h,1.0,20.0);
   glMatrixMode(GL_MODELVIEW);
   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;
}
```



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。

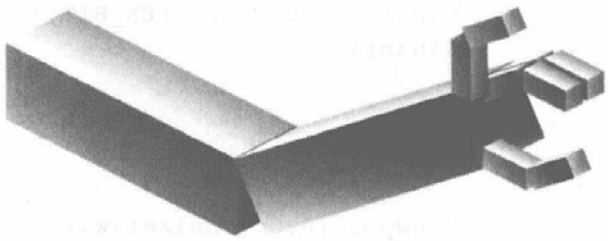


图3-26 带手指的机器人手臂

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。

int **gluUnProject**(GLdouble *winx*, GLdouble *winy*, GLdouble *winz*,
const GLdouble *modelMatrix[16]*,
const GLdouble *projMatrix[16]*,
const GLint *viewport[4]*,
GLdouble **objx*, GLdouble **objy*, GLdouble **objz*);

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

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

void reshape(int w,int h)
{
   glViewport(0,0,(GLsizei)w,(GLsizei)h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(45.0,(GLfloat)w/(GLfloat)h,1.0,100.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

void mouse(int button,int state,int x,int y)
{
   GLint viewport [4];
   GLdouble mvmatrix [16],projmatrix [16];
   GLint realy;/*OpenGL y coordinate position */
   GLdouble wx,wy,wz;/*returned world x,y,z coords */
```

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

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。

---

int **gluUnProject4**(GLdouble *winx*, GLdouble *winy*, GLdouble *winz*,
GLdouble *clipw*, const GLdouble *modelMatrix[16]*,
const GLdouble *projMatrix*[*16*],
const GLint *viewport[4]*,
GLclampd *zNear*, GLclampd *zFar*,

GLdouble *objx, GLdouble *objy,
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。

int **gluProject**(GLdouble *objx*, GLdouble objy, GLdouble *objz*,
const GLdouble *modelMatrix[16]*,
const GLdouble *projMatrix[16]*,
const GLint *viewport[4]*,
GLdouble *winx, GLdouble *winy, GLdouble *winz);

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()、gluUnProject4()和gluProject()中使用。

第4章 颜 色

本章目标

- 学会在应用程序中选择RGBA模式或颜色索引模式。
- 指定物体使用的绘图颜色。
- 使用平滑着色模式，用多种颜色绘制多边形。

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。特别是，已经不再支持颜色索引渲染。RGBA相关的概念仍然是有效的，但是，需要在一个顶点着色器或片段着色器中实现，相关内容在第15章介绍。

几乎所有的OpenGL应用程序的目的都是在屏幕窗口中绘制具有各种颜色的图像。窗口是一个矩形的像素数组，每个像素包含并显示它自己的颜色。因此，从某种意义上说，OpenGL执行的所有计算（需要考虑OpenGL函数、状态信息和参数值的各种计算）就是确定将要在窗口中绘制的每个像素的最终颜色。本章将讨论用于指定颜色的函数，并介绍OpenGL是如何解释它们的。本章的主要内容分布于下面几节中：

- 颜色感知：介绍眼睛是如何感知颜色的。
- 计算机颜色：描述计算机屏幕上的像素和它们的颜色之间的关系，并介绍了两种显示模式——RGBA和颜色索引。
- RGBA模式和颜色索引模式：说明这两种颜色模式是如何使用图形硬件的，并讨论了应该怎样选择颜色模式。
- 指定颜色和着色模型：描述一些用于指定目标颜色或着色模型的OpenGL函数。

4.1 颜色感知

从物理学讲，光是由光子组成的。光子是一种极为微小的光微粒，每个光子沿自己的路径前进，而且具有自己的频率（或波长、能量——频率、波长和能量只要确定其一，另外两者也随之确定）。光子的特征完全是由它的位置、方向和频率（或波长、能量）决定的。波长范围在390纳米（紫色）至720纳米（红色）的光子覆盖了可见光谱的颜色，形成了彩虹的彩色（紫、青、蓝、绿、黄、橙、红）。但是，人的眼睛还可以感受到很多彩虹所不包括的颜色，如白、黑、褐和粉红等。这是怎么回事呢？

人眼所看到的实际上是不同频率的光子的混合体。真正的光源是由它们所发射光子的频率分布决定的。理想的白光是由相同数量的所有频率的光组成的。激光是一种非常纯的光，所有的光子几乎具有相同的频率（方向和相位也相同）。钠蒸气灯发出的光主要是黄色光。太空中绝大多数星球发出的光的频率在很大程度上取决于它们的温度（黑体辐射）。我们周围绝大多数光源所发出的光的频率分布是极为复杂的。

当视网膜上的一些细胞（称为锥细胞）受到光子撞击而兴奋时，人眼就感知到了颜色。3种不同类型的锥细胞分别对3种不同波长的光最为敏感。有一种锥细胞对红光最为敏感，另一种对绿光最为敏感，还有一种则对蓝光最为敏感（患有色盲的人常常缺少其中一种或两种锥细胞）。当一束特定的

混合光线进入人眼时，视网膜上的不同类型的锥细胞就会根据光线中各种光子的不同强度产生不同程度的兴奋度。如果两束不同的混合光对3种类型的锥细胞产生的刺激程度恰好相同，人眼就无法区分它们的颜色了。

由于每种颜色是由锥细胞对入射光所产生的兴奋程度决定的，因此人眼能够感受到一些并不位于透镜或彩虹所产生的光谱范围内的颜色。例如，如果发射一束由红光和蓝光组成的光线，视网膜上的视红锥细胞和视蓝锥细胞都会激活，人眼所看到的就是洋红色，它并不位于光谱中。其他的组合颜色还包括褐色、青绿色和紫红色等，它们也都不包括在颜色光谱中。

计算机显示器通过点亮像素模拟可见光。点亮像素使用的颜色成分比例和相同颜色的光刺激视网膜上不同类型的锥细胞所使用的红、绿、蓝光的比例相同。如果人眼还具有其他类型的锥细胞（例如视黄锥细胞），彩色显示器很可能还会使用黄色电子枪，而且我们可能会用RGBY（红、绿、蓝、黄）组合来指定颜色。如果人眼的锥细胞类型更少一些，我们在讨论本章时也会轻松很多。

为了显示一种特定的颜色，显示器发射适当数量的红、绿和蓝（RGB）光来模拟人眼中不同类型的锥细胞。彩色显示器可以让每个像素发射不同比例的红、绿、蓝光，人眼可以看到数以百万计的具有各自颜色的光微粒。

注意：表示颜色（或颜色模型）的其他方法还有很多，这些方法的缩写有HLS、HSV和CMYK等。如果想用这些模式来表示颜色数据，就需要把数据在这些格式和RGB格式之间进行转换。关于执行这些转换的公式，请参阅Foley、van Dam等著的《Computer Graphics: Principle and Practice》（Addison-Wesley，1990）。

本节只考虑人眼感知的进入眼睛的光子的组合。光线从材料上反弹再进入眼睛的情况更为复杂。例如，从红球弹回的白光将呈现红色，从蓝色玻璃反射回来的黄光看上去几乎是黑色的（关于这些效果的讨论，请参阅第5.2节）。

4.2 计算机颜色

在彩色计算机屏幕上，硬件装置使屏幕上的每个像素发射不同数量的红、绿和蓝光。这几种颜色的数量分别称为R、G和B值。它们常常包装在一起（有时候还有第4个值，称为alpha，用A表示）。经过包装的值称为RGB值（或RGBA值）。关于alpha值的解释，请参阅第6.1节。每个像素的颜色信息可以按照RGBA模式或颜色索引模式进行存储。在RGBA模式下，每个像素分别保存一个R、G、B值，并可能保存一个A值。在颜色索引模式下，每个像素存储一个表示颜色序号的数值（称为颜色索引）。每个颜色索引表示颜色表中一个特定的项。颜色表定义了一组特定的R、G、B值，称为颜色映射表。

在颜色索引模式下，我们可能想修改颜色映射表中的值。由于颜色映射是由窗口系统控制的，因此并不存在专门用于完成这种任务的OpenGL函数。在本书的所有例子中，颜色显示模式的初始化任务是在窗口打开时由GLUT实用工具库的函数完成的（详见附录A）。

不同的图形硬件在像素数组的大小以及每个像素可以显示的颜色数量方面存在很大的区别。在所有的图形系统中，每个像素都使用相同数量的内存来存储它的颜色。用于存储所有像素颜色的内存称为颜色缓冲区（color buffer）。缓冲区的大小通常是用位来表示的，因此8位的缓冲区使用8位数据存储每个像素的颜色（一共可以表示256种不同的颜色）。颜色缓冲区的大小因机器而异（关于这方面的详细信息，请参阅第10章）。

R、G和B值的范围从0.0（无）到1.0（完全强度）。例如，R = 0.0、G = 0.0、B = 1.0表示最亮的蓝色。如果一种颜色的R、G、B值都为0.0，那么它就是黑色。如果R、G、B值都是1.0，那么计算机

屏幕将用最亮的白色来绘制。混合蓝色和绿色的结果是青色，混合红色和蓝色的结果是洋红色，混合红色和绿色的结果是黄色。为了帮助读者根据R、G、B值创建所需的颜色，可以参考彩图12的颜色立方体。这个立方体的轴分别表示红、蓝和绿色的强度。图4-1显示了这个颜色立方体的黑白版本。

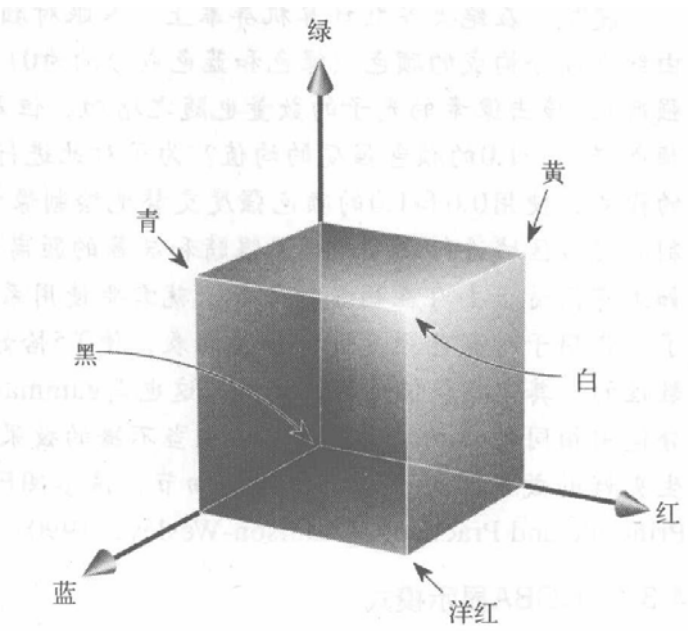


图4-1　黑白版本的颜色立方体

指定物体（在此例中为一个点）颜色的命令就像下面这样简单：

```
glColor3f(1.0, 0.0, 0.0);
/* the current RGB color is red: */
/* full red, no green, no blue. */
glBegin(GL_POINTS);
    glVertex3fv(point_array);
glEnd();
```

在有些模式下（例如执行了光照或纹理计算），被分配的颜色值（表示像素的颜色）在进入帧缓冲区之前会经历一些其他操作。事实上，像素的颜色是由一连串长长的操作决定的。

在程序执行的早期，颜色显示模式设置为RGBA模式或颜色索引模式。当颜色显示模式初始化之后，它就不能再进行更改了。程序在执行时，根据顶点确定每个几何图元的颜色（无论是RGBA值还是颜色索引值）。这种颜色可能是为顶点显式指定的值。如果启用了光照，它的颜色取决于变换矩阵与表面法线以及其他材料属性的交互效果。换句话说，用蓝光照射的红球和没有光线照射的红球看上去是不一样的（详见第5章）。在执行了相关的光照计算之后，程序所选择的着色模式随之生效。如4.4节所述，可以选择单调着色（flat shading）或平滑着色（smooth shading），这两种着色模式会对像素的最终颜色产生不同的影响。

接着，图元被光栅化（转换为二维图像）。光栅化过程决定了图元将占据窗口坐标中的哪些整型栅格方块，并为每个方块分配颜色和其他值。一个具有相关颜色值、z值（深度）和纹理坐标值的栅格方块称为片断（fragment）。像素是帧缓冲区的基本元素。片断来自图元，它与对应的像素组合，形成一个新的像素。在创建了片断之后，OpenGL随后会对片断应用纹理、雾和抗锯齿处理（如果已经启用了这些功能）。在此之后，OpenGL使用片断以及已经存储在帧缓冲区中的像素，执行所有指定的alpha混合、抖动和位逻辑操作。最后，片断的颜色（RGBA值或颜色索引值）写入到像素，根据窗口的颜色显示模式在窗口中显示。

4.3　RGBA和颜色索引模式

无论是颜色索引模式还是RGBA模式，每个像素都存储了一定数量的颜色数据。这个数量是由帧缓冲区的位平面（bitplane）数量决定的。在每个像素中，1个位平面表示1位数据。如果有8个颜色位平面，每个像素便用8位来表示颜色，因此它可以存储2^8（即256）种不同的颜色。

位平面常常被均匀地划分，分别存储R、G和B颜色成分（例如，在24个位平面的系统中，红、绿、蓝分别使用8位），但这并非金科玉律。为了找出系统中用于表示红、绿、蓝、alpha或颜色索引值的有效位平面的数量，可以用GL_RED_BITS、GL_GREEN_BITS、GL_BLUE_BITS、

GL_ALPHA_BITS和GL_INDEX_BITS为参数调用glGetIntegerv()函数。

注意：在绝大多数计算机屏幕上，人眼对颜色强度的感知并不是线性的。现在我们只考虑由红色成分构成的颜色（绿色和蓝色成分均为0）。随着颜色的强度从0.0（无）增加到1.0（完全强度），撞击像素的光子的数量也随之增加。但是，问题在于0.5的颜色强度是不是正好为0.0的颜色强度和1.0的颜色强度的均值？为了对此进行测试，可以编写一个程序，使用国际象棋棋盘的模式，使用0.0和1.0的颜色强度交替地绘制像素，然后把绘图结果与一个用0.5的颜色强度绘制的实心区域进行比较。如果眼睛和屏幕的距离适中，这两个区域看上去应该具有相同的强度。如果它们看上去存在明显的差异，就需要使用系统提供的某种校正机制。例如，许多系统提供了一张用于对颜色强度进行调整的表，使0.5恰好是0.0和1.0的均值。一般使用的映射方式是指数性的，其中指数值称为gamma（这也是gamma校正这个术语的由来）。对红色、绿色和蓝色成分使用相同的gamma值能够产生相当不错的效果。但是，如果使用不同的gamma值，往往会产生更好的效果。关于这个话题的细节，请参阅Foley、van Dam等著的《Computer Graphics: Principle and Practice》(Addison-Wesley，1990)。

4.3.1 RGBA显示模式

在RGBA模式下，硬件为R、G、B和A成分保留一定数量的位平面（但每种成分的位平面数量并不一定相同），如图4-2所示。R、G和B值一般以整数而不是浮点数的形式存储，并且根据可用的位数进行缩放，以便于存储和提取。例如，如果系统用8位表示R成分，它就可以存储0～255之间的整数。因此，位平面中的0、1、2、…、255将对应于0/255=0.0、1/255、2/255、…、255/255 = 1的R值。不管位平面的数量是多少，0.0总是表示最低的强度，1.0总是表示最高的强度。

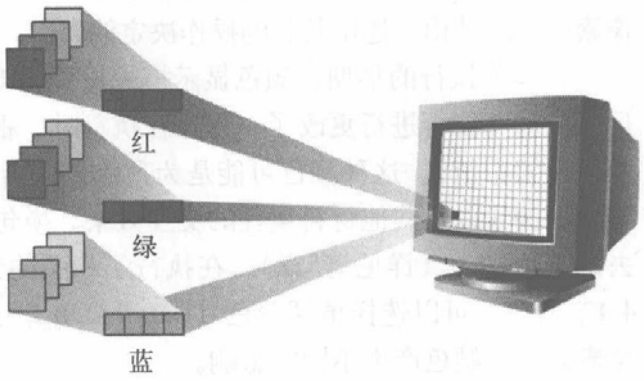


图4-2 来自位平面的RGB值

注意：alpha值（RGBA中的A）对屏幕显示的颜色并没有直接的效果。它有许多用途，包括混合和透明。它还会影响写入到帧缓冲区的R、G和B值。关于alpha值的更多信息，请参阅第6.1节。

像素可以显示的不同颜色的数量取决于位平面的数量以及硬件是如何解释这些位平面的。不同颜色的数量不能超过2^n，其中n是位平面的数量。因此，一种用24个位平面表示RGB成分的机器可以显示多达1 677万种不同的颜色。

抖动



高级话题

有些图形硬件使用抖动来增加可以显示的颜色数量。为了说明抖动的工作原理，假定系统分别只用1个位来表示R、G和B。这样，它一共可以显示8种颜色：黑、白、红、蓝、绿、黄、青和洋红。为了显示一块粉红色的区域，图形硬件仍然采用前面那种棋盘模式的方法，用红色和白色交替对像素进行着色。如果眼睛距离屏幕足够远，不能看到单独的像素，这块区域看上去就是粉红色的，也就是红色和白色的均值。深一点的粉红色可以通过提高红色像素的比例来实现，淡一点的粉红色可以通过提高白色像素的比例来实现。

使用这种技巧，便不存在粉红色的像素，实现“粉红”效果的唯一方法就是覆盖一块由多个像素组成的区域（单个的像素无法进行抖动）。如果为一种不可用的颜色指定了一个RGB值，并用它填充一个多边形，图形硬件就会混合邻近颜色（混合后的颜色均值接近于所需要的颜色）来填充这个多边形的内部区域。但是，需要记住的一点是，如果是从帧缓冲区读取像素信息，所得到的是实际的红色和白色像素值，因为帧缓冲区中不存在“粉红色”。关于读取像素值的更多信息，请参阅第8章。

图4-3显示了一种简单的抖动，用黑色和白色来产生灰色。顶部从左到右那几个4×4的模式分别表示50%、19%和69%的灰度。在每种模式的下面，可以看到重复使用这种模式所形成的图案，但是黑色和白色方块依然清晰可辨。如果隔着相当远的距离观察它们，就会发现它们融合在一起，形成3幅不同程度的灰色图案。

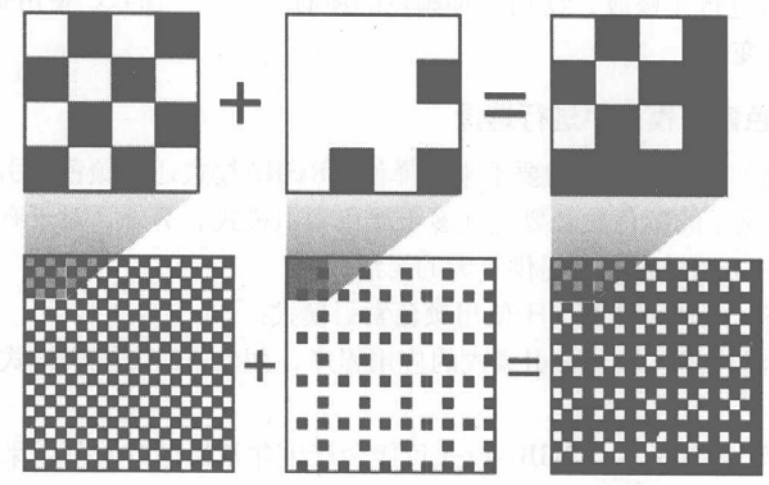


图4-3　抖动黑色和白色，创建灰色

如果分别用8位来表示R、G和B值，即使不借助抖动也可以创建高质量的图像。但是，并不是说，如果计算机具有24位的位平面，就可以不需要使用抖动了。例如，如果是在双缓冲模式下运行，位平面可能被分成两组，每组12位，这样每种颜色成分实际上只有4位。在许多情况下，如果不使用抖动，每种成分4位的颜色是无法产生令人满意的效果的。

可以用GL_DITHER为参数调用glEnable()或glDisable()函数，分别启用或禁用抖动功能。注意，和其他许多特性不同，抖动在默认情况下是启用的。

4.3.2　颜色索引模式

在颜色索引模式下，OpenGL使用一个颜色映射表（或颜色查找表），类似于使用调色板来混合颜料，准备根据颜色编号来绘制场景。画家的调色板提供了空间，可用于混合颜料。类似地，计算机的颜色映射表也提供了索引，可以混合基本的红、绿和蓝色值，如图4-4所示。

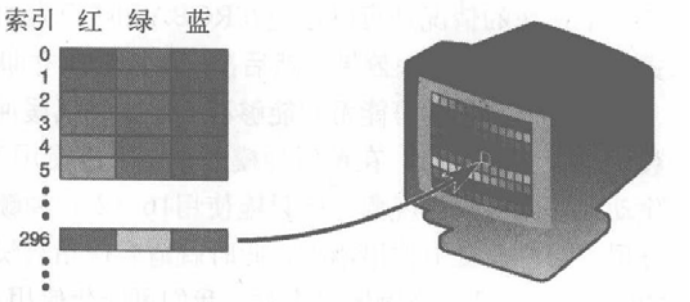


图4-4　颜色映射表

当画家根据标号绘制场景时，他从颜料调色板中选择一种颜色，并用它填充对应标号的区域。计算机在每个像素的位平面中存储了颜色索引值。然后，用这些平面值引用颜色映射表，根据颜色映射表中对应的红、绿和蓝值来绘制屏幕，如图4-5所示。

在颜色索引模式下，同时可用的颜色数量受限于颜色映射表的大小以及可用的位平面的数量。颜色映射表的大小是由专用的硬件决定的。颜色映射表的大小总是2的整数次方，一般在256（2^8）到4 096（2^{12}）之间，其中指数就是它所使用的位平面的数量。如果颜色映射表共有2^n个索引项以及m个可用的位平面，可用的颜色值的数量就是2^n和2^m中较小的那个。

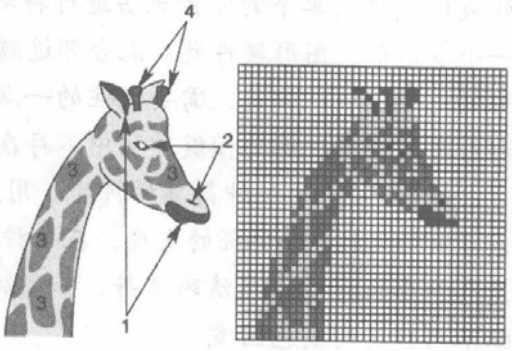


图4-5 使用颜色映射表来绘图

在RGBA模式下，每个像素的颜色与其他像素的颜色独立。但是，在颜色索引模式下，具有相同索引值的每个像素在它们的位平面中共享相同的颜色映射位置。如果颜色映射表中有一个项目进行了修改，使用这种颜色的所有像素的颜色也随之发生变化。

4.3.3 在RGBA和颜色索引模式中进行选择

应该根据可用的硬件和应用程序的要求来选择使用RGBA模式还是颜色索引模式。在绝大多数系统中，RGBA模式可以显示的颜色数量要远远多于颜色索引模式。另外，对于有些效果（如着色、光照、纹理贴图和雾），RGBA模式能够提供更大的灵活性。

在下面这些情况下，我们可能倾向于使用颜色索引模式：

- 移植的是一个广泛使用了颜色索引模式的应用程序，如果不改变显示模式，我们的任务也许可以轻松很多。
- 如果可用的位平面数量很少，RGBA模式可能会产生非常粗糙的着色效果。例如，如果一共只有8个位平面。在RGBA模式下，可能用3位表示红色，用3位表示绿色，用2色表示蓝色。因此，只能使用8种（2^3）红色和绿色色调、4种（2^2）蓝色色调。色调之间的过渡会非常明显。

在这种情况下，如果我们在着色方面的要求并不高，就可以使用颜色查找表载入更多的色调。例如，如果只需要蓝色色调，就可以使用颜色索引模式，在颜色映射表中存储多达256（2^8）种的蓝色色调。这比RGBA模式下的4种色调要好得多。当然，这种用法将耗尽整个颜色查找表，所以无法使用红色、绿色或其他混合颜色。

- 颜色索引模式可以实现一些特殊的技巧，例如颜色映射动画和层次绘图，详见第14章。

一般而言，应该尽可能使用RGBA模式。RGBA模式可以在纹理贴图中使用，并且在使用光照、着色、雾和抗锯齿功能时更为灵活。

4.3.4 切换显示模式

最理想的情况是可以避免在RGBA和颜色索引模式之间二选其一。例如，我们可能想使用索引模式实现颜色映射动画效果，然后在必要的时候立即把场景修改回到RGBA模式，以使用纹理贴图功能。

类似地，我们可能希望能够在单缓冲和双缓冲模式之间进行切换。例如，假设可以使用的位平面数量很少（如8位）。在单缓冲模式下，可以使用256（2^8）种颜色。但是，如果使用双缓冲模式来消除动画程序的闪烁现象，就只能使用16（2^4）种颜色。当绘制不闪烁的移动物体时，我们可能愿意在可用的颜色数量上作出牺牲，此时就适合使用双缓冲模式（也许物体移动太快，观察者无法注意物体的细节）。但是，当物体静止时，我们可能想使用单缓冲模式，因为这时可以使用更多的颜色。

遗憾的是，大多数窗口系统并不允许在这些模式之间进行简单的切换。例如，在X窗口系统中，颜色显示模式是X Visual的一个属性，而X Visual必须在创建窗口之前指定。当它被指定之后，它在

窗口的生存期间就不能修改。当使用双缓冲模式和RGBA颜色模式创建了一个窗口之后，在这个窗口中就只能一直使用这种模式。

我们可以用一种较为复杂的技巧解决这个问题。可以创建多个窗口，每个窗口使用不同的模式。必须控制每个窗口的可见性（例如，映射X窗口或取消它的映射、管理Motif或Athena部件或取消它的管理），并在适当的可见窗口中绘制物体。

4.4 指定颜色和着色模型

OpenGL维护一种当前颜色（在RGBA模式下）或一个当前颜色索引（在颜色索引模式下）。一般每个物体都是用当前颜色（或当前颜色索引）绘制的，除非使用了一些更为复杂的着色模型，例如光照和纹理贴图。注意下面的伪码：

```
set_color(RED);
draw_item(A);
draw_item(B);
set_color(GREEN);
set_color(BLUE);
draw_item(C);
```

项目A和B是用红色绘制的，项目C是用蓝色绘制的。第4行代码（也就是把当前颜色设置为绿色的代码）并没有任何效果（除了浪费一些时间之外）。如果没有光照和纹理，在设置了当前颜色之后，此后所有的物体都是用这种颜色绘制的，直到当前颜色发生了变化。

4.4.1 在RGBA模式下指定颜色

在RGBA模式下，可以使用glColor*()函数选择一种当前颜色。

兼容性扩展
glColor

void **glColor3**{b s i f d ub us ui}(*TYPE r, TYPE g, TYPE b*);

void **glColor4**{b s i f d ub us ui}(*TYPE r, TYPE g, TYPE b, TYPE a*);

void **glColor3**{b s i f d ub us ui}v(const *TYPE* *v);

void **glColor4**{b s i f d ub us ui}v(const *TYPE* *v);

设置当前的红、绿、蓝和alpha值。这个函数最多可达3个后缀，以区分它所接受的不同参数。第一个后缀是3或4，表示是否应该在红、绿、蓝值之外提供一个alpha值。如果没有提供alpha值，它会自动设置为1.0。第二个后缀表示参数的数据类型：byte、short、int、float、double、unsigned byte、unsigned short或unsigned int。第三个后缀是可选的v，表示参数是否为一个特定数据类型的数组指针。

在使用接受浮点数据类型的glColor*()版本时，参数值一般位于0.0～1.0之内，也就是帧缓冲区可以存储的最小和最大值。如果用无符号整数值表示颜色成分值，它们将会线性地映射到浮点值范围。最大的可表示值映射到1.0（完全强度），0映射到0.0（零强度）。如果用有符号整型值表示颜色成分值，它们也会线性地映射到浮点值范围，其中最大的正值映射到1.0，最大的负值映射到−1.0，见表4-1。

在更新当前颜色或当前光照材料参数之前，无论是浮点值还是有符号整型值都不必截取在[0, 1]范围之内。经过光照计算之后，位于[0, 1]范围之外的最终颜色值在插值或写入到颜色缓冲区之前会截取在[0, 1]范围之间。即使禁用了光照，颜色成分值在光栅化之前也会进行截取。

表4-1 把颜色值转换为浮点数

后缀	数据类型	最小值	最小值映射到	最大值	最大值映射到
b	1字节整数	−128	−1.0	127	1.0
s	2字节整数	−32 768	−1.0	32 767	1.0
i	4字节整数	−2 147 483 648	−1.0	2 147 483 647	1.0
ub	1字节无符号整数	0	0.0	255	1.0
us	2字节无符号整数	0	0.0	65 535	1.0
ui	4字节无符号整数	0	0.0	4 294 967 295	1.0

另外还有一个类似的glSecondaryColor*()函数。这个函数也指定了一种颜色，一般是在纹理贴图之后使用（在光照被禁用的前提下）。关于这方面的更多细节，请参阅第9.10节。

颜色截取

OpenGL 3.0引入了浮点帧缓冲区，其中，RGBA颜色成分可以存储为真正的浮点数值。如果有了更大的浮点值动态范围，你可能想要允许那些落在OpenGL最初指定的范围（0.0～1.0）之外的颜色值。

void **glClampColor**(GLenum *target*, GLenum *clamp*);

指定主颜色值和辅助颜色值是否截取。target必须设置为GL_CLAMP_VERTEX_COLOR、GL_CLAMP_FRAGMENT_COLOR或GL_CLAMP_READ_COLOR，clamp必须是GL_TRUE、GL_FALSE或GL_FIXED_ONLY之一。

兼容性扩展
GL_CLAMP_VERTEX_COLOR
GL_CLAMP_FRAGMENT_COLOR

OpenGL 3.0中可用的颜色截取选项见表4-2。

表4-2 用于glClampColor()的值

参 数	说 明
GL_TRUE	颜色应该截取到范围[0, 1]
GL_FALSE	颜色不能截取
GL_FIXED_ONLY	只有帧缓冲区格式是定点格式的时候，颜色才能截取

4.4.2 在颜色索引模式下指定颜色

在颜色索引模式下，可以使用glIndex*()函数选择一个单值颜色索引，把它作为当前的颜色索引。

void **glIndex**{sifd ub}(TYPE *c*);

void **glIndex**{sifd ub}**v**(const TYPE *c);

把当前颜色索引值设置为c。这个函数的第一个后缀表示参数的数据类型：short、int、float、double或unsigned byte。第二个后缀是可选的v，表示参数是一个特定数据类型的值数组（这个数组只包含了1个值）。

兼容性扩展
glIndex

在第2.1.1节中，我们介绍了glClearColor()函数的细节。在颜色索引模式下，存在一个对应的glClearIndex()函数。

注意：OpenGL并没有提供任何函数在颜色查找表中加载颜色索引值。窗口系统一般已经提供了这样的操作。GLUT提供了一个glutSetColor()函数，可以调用这个因窗口系统而异的函数。

void **glClearIndex**(GLfloat *cindex*);

在颜色索引模式下设置当前的清除颜色。在一个使用颜色索引模式的窗口中，调用glClear(GL_ COLOR_BUFFER_BIT)将用cindex清除当前缓冲区。默认的清除索引值是0.0。

兼容性扩展
glClearIndex



高级话题

当前索引值是以浮点值的形式存储的。整型值被直接转换为浮点值，而不需要特殊的映射。位于颜色索引缓冲区可表示的范围之外的索引值并不会被截取。但是，当一个索引值进行了抖动（如果已经启用这个功能）并写入到帧缓冲区，它会转换为定点形式。这个定点值的整数部分的任何位如果并不对应于帧缓冲区的任何位，它们就会被屏蔽。

4.4.3 指定着色模型

直线或填充多边形可以用一种颜色进行绘制（单调着色），也可以用多种颜色进行绘制（平滑着色，也称Gouraud着色）。可以用glShadeModel()函数指定所需的着色模型。

void **glShadeModel**(GLenum *mode*);

设置着色模型。mode参数可以是GL_SMOOTH（默认值）或GL_FLAT。

兼容性扩展
glShadeModel

在单调着色模型下，整个图元的颜色就是它的任何一个顶点的颜色。在平滑着色模型下，每个顶点都是单独进行处理的。如果图元是直线，线段的颜色将根据两个顶点的颜色进行均匀插值。如果图元是多边形，多边形的内部颜色是所有顶点颜色的均匀插值。示例程序4-1绘制了一个平滑着色的三角形，如彩图11所示。

示例程序4-1 绘制一个平滑着色的三角形：smooth.c

```
void init(void)
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_SMOOTH);
}

void triangle(void)
{
   glBegin(GL_TRIANGLES);
   glColor3f(1.0, 0.0, 0.0);
   glVertex2f(5.0, 5.0);
   glColor3f(0.0, 1.0, 0.0);
   glVertex2f(25.0, 5.0);
   glColor3f(0.0, 0.0, 1.0);
   glVertex2f(5.0, 25.0);
   glEnd();
}
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   triangle();
   glFlush();
}
void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
```

```
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
       gluOrtho2D(0.0, 30.0, 0.0, 30.0*(GLfloat) h/(GLfloat) w);
    else
       gluOrtho2D(0.0, 30.0*(GLfloat) w/(GLfloat) h, 0.0, 30.0);
    glMatrixMode(GL_MODELVIEW);
    }

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(100, 100);
    glutCreateWindow(argv[0]);
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}
```

在平滑着色模型下，相邻的像素使用稍微不同的颜色。在RGBA模式下，颜色稍微不同的相邻像素看上去极为相似，因此整个多边形的颜色看上去就是平滑的。在颜色索引模式下，相邻的像素可以引用颜色索引中的不同位置，因此它们的颜色可能并不相似。相邻的颜色索引项可能包含完全不同的颜色。因此，在颜色索引模式下，使用平滑着色的多边形看上去会显得光怪陆离。

为了避免这个问题，必须创建一个颜色索引表，其中相邻的索引值所包含的颜色应该是平滑的。记住，把颜色加载到颜色映射表的操作是由窗口系统执行的，而不是由OpenGL执行的。如果使用GLUT，可以调用glutSetColor()函数，用指定的红、绿和蓝值在颜色映射表中加载颜色。glutSetColor()的第一个参数是一个索引值，其他几个参数是相应的红、绿、蓝值。为了在颜色映射表中加载32个逐渐变化的黄色色调的颜色索引值（颜色索引16～47），可以调用：

```
for (i = 0; i < 32; i++) {
   glutSetColor(16+i, 1.0*(i/32.0), 1.0*(i/32.0), 0.0);
}
```

现在，如果只使用索引值16～47之间的颜色来渲染平滑着色的多边形，这些多边形将具有平滑的抖动黄色色调。

在单调着色模型下，单个顶点的颜色就定义了整个图元的颜色。如果图元是直线，直线的颜色就是在指定第二个顶点（终点）时的当前颜色。如果图元是多边形，这个多边形的颜色就是它的任一顶点的颜色，见表4-3。这张表从1开始对顶点和多边形进行计数。OpenGL会严格遵循这些规则，但是避免使用单调着色的图元出现不确定性的最好办法是只为这种图元指定一种颜色。

表4-3 OpenGL如何为第i个单调着色的多边形选择颜色

多边形类型	为第i个多边形选择哪个顶点的颜色
单个多边形	1
三角形带	$i+2$
三角形扇	$i+2$
独立的三角形	$3i$
四边形带	$2i+2$
独立的四边形	$4i$

第5章 光 照

本章目标

- 理解OpenGL是如何模拟现实世界的光照条件的。
- 通过定义光源、材料和光照模型属性，渲染光照物体。
- 定义光照物体的材料属性。
- 操纵矩阵栈，控制光源的位置。

注意：在OpenGL 3.1中，本章所介绍的很多技术和函数已经废弃删除了。虽然概念仍然是相关的，但是通过光照计算颜色必须在一个顶点着色器或片段着色器中进行。第15章简要地介绍了这些话题，而本书的姊妹篇《The OpenGL Shading Language Guide》则详尽地讨论了这些主题。

正如我们在第4章看到的那样，OpenGL计算最终场景（保存在帧缓冲区中）中每个像素的颜色。这种计算部分取决于场景所使用的光照以及场景中的物体是如何反射或吸收光线的。想象一下，在一个晴朗、充满阳光的日子里，我们所看到的大海颜色和我们在一个多云、灰蒙蒙的日子里看到的大海颜色截然不同。阳光和云的存在，决定了我们看到的大海是亮蓝色的还是灰绿色的。事实上，如果没有光照，绝大多数物体看上去甚至不像是三维物体。图5-1显示了同一个场景（一个球体）的两个版本，其中一个有光照，另一个没有光照。

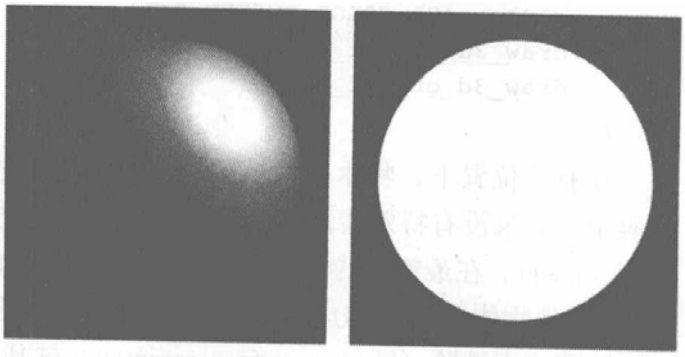


图5-1 有光照和无光照的球体

正如读者所看到的那样，那个没有光照的球体看上去和二维的圆盘没有什么两样。这就说明了物体和光线之间的交互在创建三维场景时的重要性。

在OpenGL中，可以对场景中的光照和物体进行操纵，创建许多不同类型的效果。本章首先简单介绍隐藏表面消除，然后解释如何控制场景中的光照，讨论OpenGL的光照概念模型，并详细描述如何设置数量众多的光照参数，以实现特定的效果。在本章的最后，我们将介绍相关的数学知识，解释光照是如何影响颜色的。

本章的内容主要分布于下面这几节中：

- 隐藏表面消除工具箱：描述消除隐藏表面的基本知识。
- 现实世界和OpenGL光照：用通俗的语言解释光照在现实世界和OpenGL中的行为。
- 一个简单的例子：渲染光照球体：通过一个渲染光照球体的简单程序，介绍OpenGL的光照功能。
- 创建光源：解释如何定义光源以及如何设置光源的位置。
- 选择光照模型：讨论光照模型的元素，并介绍如何指定这些元素。
- 定义材料属性：解释如何描述物体的属性，使它们按照预想的方式与光照进行交互。
- 和光照有关的数学知识：介绍OpenGL使用的一些数学计算，它们决定了场景的光照效果。

• 颜色索引模式下的光照：讨论在RGBA和颜色索引模式下光照的不同之处。

在一般情况下，光照计算是在纹理处理之前进行的。OpenGL 1.2版引入了一种光照模式(GL_SEPARATE_SPECULAR_COLOR)，把镜面光的计算从发射、环境和散射光成分的计算中分离出来，在纹理处理之后再应用镜面颜色。使用独立的镜面颜色常常可以增强亮点，使它较少受到纹理图像的颜色的影响。

5.1 隐藏表面消除工具箱

在本节中，我们首先绘制一些填充的三维物体。在绘制填充多边形时，应该着重绘制离观察位置较近的物体，并消除那些被更靠近观察者的其他物体所遮挡的物体。

当绘制一个由三维物体组成的场景时，有些物体可能会遮挡其他物体（或其他物体的一部分）。如果改变了观察点，物体之间的遮挡关系也随之发生变化。例如，如果从相反的方向观察场景，原来位于其他物体前面的物体现在就到了后面。为了绘制逼真的场景，必须维护这种遮挡关系。假如有一段代码如下：

```
while (1) {
   get_viewing_point_from_mouse_position();
   glClear(GL_COLOR_BUFFER_BIT);
   draw_3d_object_A();
   draw_3d_object_B();
}
```

在有些位置上，物体A可能会遮挡物体B。但在其他的位置上，情况可能正好相反。在前面那段代码中，如果没有特殊情况，不管选择什么观察位置，物体B总是较晚绘制的（因此，它总是位于物体A的上面）。在最糟糕的情况下，物体A和物体B彼此相交，这样物体A会遮挡物体B的一部分，物体B也会遮挡物体A的一部分。即使改变这两个物体的绘图顺序，也无法解决这个问题。

隐藏表面消除（hidden-surface removal）就是消除实心物体被其他物体所遮挡的那部分（隐藏直线消除与此类似，用于线框模型，但它所使用的技巧更为复杂，此处不讨论。详见第14.16节）。实现隐藏表面消除的最简单方法就是使用深度缓冲区（又称z缓冲区），参见第10章。

深度缓冲区的原理是把一个距离观察平面（通常是近侧裁剪平面）的深度值（或距离）与窗口中的每个像素相关联。一开始，使用glClear()函数（以GL_DEPTH_BUFFER_BIT为参数），把所有像素的深度值设置为最大可能的距离（通常是远侧裁剪平面）。随后，在场景中以任意顺序绘制所有的物体。

硬件或软件所执行的图形计算把每个被绘制表面转换为窗口上一些像素的集合，此时并不考虑它们是否会被其他物体遮挡。此外，OpenGL还计算这些表面和观察平面的距离。如果启用了深度缓冲区，在绘制每个新像素之前，OpenGL就会把它的深度值与已经存储在深度缓冲区中的这个像素的深度值进行比较。如果新像素比原先的像素更靠近观察平面，这个新像素的颜色和深度值就会取代原先那个像素。如果新像素的深度值大于原先像素的深度值，新像素就会被遮挡，它的颜色和深度值将丢弃。

为了使用深度缓冲区，首先需要启用它（只要启用一次就可以了）。每次绘制场景时，需要在绘图之前清除深度缓冲区，然后以任意顺序绘制场景中的物体。

为了对前面那段代码进行转换，使它能够执行隐藏表面消除，可以把它修改成如下代码：

```
glutInitDisplayMode(GLUT_DEPTH | .... );
glEnable(GL_DEPTH_TEST);
...
while (1) {
```

```
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    get_viewing_point_from_mouse_position();
    draw_3d_object_A();
    draw_3d_object_B();
}
```

在这段代码中，glClear()函数的参数同时清除了深度缓冲区和颜色缓冲区。

深度缓冲区测试可能会影响应用程序的性能。隐藏表面消除丢弃了一些信息，而不是将它们用来绘图，因此能够稍稍提高性能。但是，根据不同的OpenGL实现，深度缓冲区也可能会对性能造成很大的影响。用“软件”实现的深度缓冲区（用处理器内存实现）可能比那些专用的硬件缓冲区要慢得多。

5.2 现实世界和OpenGL光照

当观察一个物理表面时，眼睛对颜色的感知取决于到达和刺激锥细胞的光子的能量分布（参见第4.1节）。这些光子可能来自单个光源，也可能来自多个光源。有些光子被表面吸收，有些光子则被表面反射。另外，不同表面的属性可能存在非常大的区别。有些表面非常光亮，会把光线反射到某个方向。有些表面则把入射光均匀地发散到所有的方向。绝大多数表面都位于两者之间。

OpenGL在模拟光照时，它假定光可以分解为红、绿和蓝成分。因此，光源的特征是由它所发射的红、绿和蓝光的数量决定的。表面材料的特征是由它向各个不同方向反射的红、绿、蓝入射光的百分比决定的。OpenGL的光照方程式仅仅是一种模拟，但它在大部分情况下还是不错的，并且计算速度也相对较快。如果需要一种更为精确的（或者仅仅是另一种不同的）光照模型，必须用软件实现自己的计算过程。这种软件的复杂性可能极为惊人，读者只要阅读几本光学方面的参考书就能明白此言非虚。

在OpenGL的光照模型中，场景中来自几个不同光源的光可以单独打开或关闭。有些光来自一个特定的方向或位置，有些光则均匀地散布在场景中。例如，在房间内打开一盏电灯时，绝大多数光直接来自于这个灯泡，但也有一些光是从墙壁上反弹回来的。这种反射回来的光（称为环境光）被认为是均匀散布的，并没有一个特定的源方向。但是，当电灯熄灭时，它也随之消失了。

最后，场景中可能存在一些基本的环境光，它们并不来自某个特定的光源。它们被多次散射，已经无法分辨最初的来源。

在OpenGL的光照模型中，只有当存在能够吸引和反射光线的表面时，光源才会产生效果。OpenGL假定每个表面是由一种具有某些属性的材料组成的。材料可能会自己发光（例如汽车的前灯），可能从所有的方向散射一些入射光，可能把部分入射光反射到某个特定的方向（例如镜子或其他光亮的表面）。

OpenGL的光照模型把光分成4种独立的成分：环境光、散射光、镜面光和发射光。这4种成分都可以单独进行计算，并叠加在一起。

5.2.1 环境光、散射光、镜面光和发射光

环境光（ambient light）是那些在环境中进行了充分的散射，无法分辨其方向的光，它似乎是来自于所有的方向。房间里的逆光包含了非常多的环境光成分，由于这些光在到达眼睛前已经经过了许多表面的反弹。户外的探照灯所包含的环境光成分非常少，绝大多数光线都朝同一个方向前进，并且由于是在室外，探照灯所发出的光很少会通过其他物体的反弹进入眼睛。当环境光撞击表面时，它会向所有方向均匀发散。

散射光（diffuse light）来自某个方向。因此，如果它从正面照射表面，它看起来显得更亮一些。反之，如果它斜着掠过表面，它看起来就显得暗一些。但是，当它撞击表面时，它会均匀地向所有的方向发散。因此，不管眼睛在哪个位置，散射光看上去总是一样亮。来自某个特定位置或方向的任何光很可能具有散射成分。

镜面光（specular light）来自一个特定的方向，并且倾向于从表面向某个特定的方向反射。当一束经过充分校准的激光从一面高质量的镜子上反弹回来时，它所产生的几乎是百分之百的镜面反射光。具有光泽的金属或塑料具有非常高的镜面成分，而粉笔和地毯则几乎不存在镜面成分。可以把镜面理解成有光泽。

除了环境、散射和镜面颜色之外，材料还可能具有一种发射颜色（emissive color），它模拟那些源自某个物体的光。在OpenGL光照模型中，表面的发射颜色可以增加物体的强度，但是它不受任何光源的影响。另外，在整体场景中，发射颜色并没有作为一种额外的光照。

尽管光源发出的光具有相同分布的频率，但是它的环境、散射和镜面成分可能不同。例如，如果房间内有一盏白灯，墙壁的颜色为红色，那些发散开来的光趋于红色，而直接撞击物体的光则是白色的。OpenGL允许我们单独设置光的红、绿和蓝色成分。

5.2.2 材料颜色

OpenGL的光照模型根据材料反射的红、绿和蓝光的比例来模拟它的颜色。例如，一个纯红的球体反射所有的红色光线，吸收所有的绿色和蓝色光线。如果在白光（由相同数量的红、绿、蓝光组成）下观察这个球体，所有的红光都被反射，因此我们看到的是一个红球。如果照射这个球体的是纯粹的红光，那么它看上去仍然是红色的。但是，如果照射这个球体的是纯粹的绿光，它看上去就是黑色的。这是因为所有的绿光都被吸收，由于不存在红光，所以它不会反射任何光线。

和光一样，材料也具有不同的环境、散射和镜面颜色，它们决定了材料对红、绿和蓝光的反射率。材料的环境反射属性与每种入射光的环境光成分组合，散射反射属性与入射光的散射成分组合，镜面反射属性和入射光的镜面成分组合。环境和散射属性定义了材料的颜色，它们一般很相似，却不相同。镜面反射属性通常是白色或灰色的，因此镜面亮点的颜色也就是光源的镜面光的颜色。如果一束白光照射一个有光泽的红色塑料球体，这个球体的整体看上去是红色的，它上面的那些亮点则是白色的。

5.2.3 光和材料的RGB值

光线的颜色成分与材料的颜色成分的含义并不相同。对于光线而言，颜色成分的数量对应于每种颜色的完全强度的百分比。如果一种光线颜色的R、G和B值都为1.0，这种光线就是可以实现的最亮白色。如果这些值都是0.5，这种光线应该是白色的，但强度只有一半，因此看上去像是灰色的。如果R=G=1并且B=0（完全的红色和绿色，没有蓝色成分），这种光线看上去就是黄色的。

对于材料而言，它的RGB值和它对这些颜色的反射比例相对应。因此，如果一种材料的R = 1、G = 0.5、B = 0，它将反射所有的入射红光，一半的入射绿光，但是不反射任何入射蓝光。换句话说，如果一束OpenGL光线的成分是（LR，LG，LB），一种材料具有相应的成分（MR，MG，MB），那么在忽略了其他所有的反射效果之后，进入眼睛的光就是（LR · MR，LG · MG，LB · MB）。

类似地，如果有两束光进入眼睛，它们的成分分别是（R1，G1，B1）和（R2，G2，B2），OpenGL就会把它们的成分叠加在一起，也就是（R1 + R2，G1 + G2，B1 + B2）。如果相加之后的任何一个值大于1（相当于超出了设备能够显示的这种颜色的亮度），它就会被截取为1。

5.3 一个简单的例子：渲染光照球体

为了在场景中增加光照，需要执行如下步骤：

1) 定义每个物体的每个顶点的法线向量。这些法线决定了物体相对于光源的方向。

2) 创建和选择一个或多个光源，并设置它们的位置。

3) 创建和选择一种光照模型（lighting model），它定义了全局环境光的层次以及观察点的有效位置（便于进行光照计算）。

4) 定义场景中物体的材料属性。

示例程序5-1执行了这些任务。它显示了一个被照亮的球体。照射这个球体的光来自于单个光源，如图5-1所示。

示例程序5-1 绘制一个光照球体：ligth.c

```
void init(void)
{
   GLfloat mat_specular [] ={1.0,1.0,1.0,1.0 };
   GLfloat mat_shininess [] ={50.0 };
   GLfloat light_position [] ={1.0,1.0,1.0,0.0 };
   GLfloat white_light [] ={1.0,1.0,1.0,1.0 };
   GLfloat lmodel_ambient [] ={0.1,0.1,0.1,1.0 };
   glClearColor(0.0,0.0,0.0,0.0);
   glShadeModel(GL_SMOOTH);
   glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
   glMaterialfv(GL_FRONT,GL_SHININESS,mat_shininess);
   glLightfv(GL_LIGHT0,GL_POSITION,light_position);
   glLightfv(GL_LIGHT0,GL_DIFFUSE,white_light);
   glLightfv(GL_LIGHT0,GL_SPECULAR,white_light);
   glLightModelfv(GL_LIGHT_MODEL_AMBIENT,lmodel_ambient);

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);
   glutSolidSphere(1.0,20,16);
   glFlush();
}
void reshape(int w,int h)
{
   glViewport(0,0,(GLsizei)w,(GLsizei)h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <=h)
   glOrtho(-1.5,1.5,-1.5*(GLfloat)h/(GLfloat)w,
      1.5*(GLfloat)h/(GLfloat)w,-10.0,10.0);
   else
      glOrtho(-1.5*(GLfloat)w/(GLfloat)h,
```

```
        1.5*(GLfloat)w/(GLfloat)h,-1.5,1.5,-10.0,10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

int main(int argc,char**argv)
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB |GLUT_DEPTH);
    glutInitWindowSize(500,500);
    glutInitWindowPosition(100,100);
    glutCreateWindow(argv [0]);
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}
```

在这个例子中，与光照相关的函数调用都位于init()函数内部。接下来的几段文字将简单地介绍这些函数。在本章的后面，我们还将详细介绍这些函数。注意，示例程序5-1使用了RGBA颜色模型，而不是颜色索引模型。OpenGL的光照计算在这两种模式下是不同的。事实上，光照功能在颜色索引模式下受到很大的限制。因此，如果需要使用光照，RGBA是优选的模式，本章的所有例子都使用了这种模式。关于这个问题的更多细节，请参阅第5.8节。

为每个物体的每个顶点定义法线向量

物体的法线向量决定了它相对于光源的方向。对于物体的每个顶点，OpenGL使用法线判断这个顶点从每个光源接收的光线数量。在这个程序中，定义球体的法线正是gluSolidSphere()函数所完成的任务之一。

为了进行正确的光照计算，表面法线必须为单位长度。还必须保证对物体所进行的模型视图变换并没有对表面法线进行缩放，最终的法线仍然保持为单位长度。为了保证法线仍然为单位长度，可能需要以GL_NORMALIZE或GL_RESCALE_NORMAL为参数调用glEnable()函数。

GL_RESCALE_NORMAL导致表面法线的每个成分与同一个值相乘，而这个值是由模型视图变换矩阵决定的。因此，只有当法线向量所经历的缩放为一致性收缩（即各个方向的缩放数量相同），并且当法线最初为单位向量时，这种方法才能得到正确的结果。相比之下，GL_NORMALIZE是一种更彻底的操作。当使用GL_NORMALIZE时，它计算法线向量的长度，然后把法线的各个成分除以这个长度。这个操作能够保证最终的法线为单位长度。但是，与前一种方法相比，它的开销可能更大一些。关于如何定义法线向量的更多细节，请参阅第2.5节以及附录H（可通过http://www.opengl-redbook.com/appendices/在线访问该附录）。

注意：有些OpenGL实现在实现GL_RESCALE_NORMAL时，实际上对法线向量进行了规范化，而不是仅仅对它们进行缩放。但是，我们无法知道自己使用的OpenGL实现是不是这样做的，并且也不应该指望它肯定会这样做。

创建、定位和启用光源

示例程序5-1只使用了一个白色的光源，它的位置是由glLightfv()函数指定的。这个程序把光源0（GL_LIGHT0）的颜色指定为白色，用于散射和镜面反射的计算。如果想使用一种不同颜色的光，可

以对glLigh*()函数进行修改。

至少可以在场景中包含8个光源，它们可以是不同的颜色（特定OpenGL实现可能允许指定超过8个的光源）。GL_LIGHT0之外的其他光源的默认颜色是黑色。也可以把光源放在自己希望的地方。例如，我们可以让它们靠近场景，就像壁灯一样。也可以把它们放在无限远处，用来模拟太阳光。除此之外，还可以对光源进行控制，让它产生狭窄的聚焦光束或角度较广的光束。记住，每个光源都需要大量的计算，因此场景的渲染性能将受到场景中光源数量的影响。

关于如何创建符合要求的光源的更多信息，请参阅第5.4节。

定义了所需光源的特征之后，必须调用glEnable ( ) 函数来启用它。还需要以GL_LIGHTING为参数调用glEnable()函数，让OpenGL准备执行光照计算。关于这方面的更多信息，请参阅第5.5.5节。

选择光照模型

正如读者所预料的那样，glLightModel*()函数描述了光照模型的参数。在示例程序5-1中，唯一显式定义的光照模型元素就是全局环境光。光照模型还需要定义场景的观察者应该位于无限远处还是位于场景的本地，并确定场景中物体的正面和背面是否应该执行不同的光照计算。对于这两个设置，示例程序5-1采用了默认设置，即观察者位于无限远处（“无限远观察者”模式），并且只有一面接受光照。如果使用本地的观察者，需要执行的计算会复杂很多，因为OpenGL必须计算观察点和每个物体之间的角度。但是，在无限远观察者模式下，这个角度可以忽略，渲染结果的逼真性只会稍稍受到影响。并且，在这个例子中，由于球体的背面始终是不可见的（它位于球体的内部），单面光照已经绰绰有余。关于OpenGL光照模型元素的更详细描述，请参阅第5.5节。

为场景中的物体定义材料属性

物体的材料属性决定了它是如何反射光线的，因此也决定了它的颜色。由于物体的材料表面和入射光之间的交互十分复杂，因此指定物体的材料属性，使它具有预想的外观可以说是一项艺术。可以指定材料的环境、散射和镜面颜色以及它的光泽度。这个例子只指定了最后两种材料属性——镜面材料颜色和光泽度（使用glMaterialfv()函数）。关于所有材料属性参数的描述和例子，参见第5.6节。

一些重要注意事项

当编写光照程序时，记住有些光照参数可以使用默认值，另外一些光照参数则需要进行修改。另外，不要忘了启用自己定义的光照，使光照计算生效。最后，记住当光照条件发生变化时，可以使用显示列表实现最大限度的效率（参见第7.3节）。

兼容性扩展
glLight
GL_LIGHTi
GL_AMBIENT
GL_DIFFUSE
GL_SPECULAR
GL_POSITION
GL_SPOT_DIRECTION
GL_SPOT_EXPONENT
GL_SPOT_CUTOFF
GL_CONSTANT_ATTENUATION
GL_LINEAR_ATTENUATION
GL_QUADRATIC_ATTENUATION

5.4 创建光源

光源具有几种属性，例如颜色、位置和方向。接下来的几节我们阐述如何控制这些属性以及它们对光照效果的影响。用于指定所有光源属性的函数是glLight*()。它的3个参数决定了它所指定的是哪个光源的属性、具体的属性以及该属性的预想值。

void **glLight**{if}(GLenum *light*, GLenum *pname*, TYPE *param*);

void **glLight**{if}v(GLenum *light*, GLenum *pname*, const TYPE *param);

创建light指定的光源，它可以是GL_LIGHT0、GL_LIGHT1，

……、或GL_LIGHT7。被设置的光源属性是由pname定义的，它指定了一个命名参数（见表5-1）。param表示pname属性将要设置的值。如果是向量版本，它是指向一组值的数组。如果是非向量版本，它就是需要设置的值本身。非向量版本只能用于设置单值的光源属性。

表5-1 glLight*()函数的pname参数的默认值

参数名称	默认值	含 义
GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)	光的环境强度
GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0)或(0.0, 0.0, 0.0, 1.0)	光的散射强度（light0的默认值为白色，其他为黑色）
GL_SPECULAR	(1.0, 1.0, 1.0, 1.0)或(0.0, 0.0, 0.0, 1.0)	光的镜面强度（light0的默认值为白色，其他为黑色）
GL_POSITION	(0.0, 0.0, 1.0, 0.0)	表示光源位置的坐标（x, y, z, w）
GL_SPOT_DIRECTION	(0.0, 0.0, −1.0)	表示聚光灯方向的向量（x, y, z）
GL_SPOT_EXPONENT	0.0	聚光指数
GL_SPOT_CUTOFF	180.0	聚光灯的切角
GL_CONSTANT_ATTENUATION	1.0	常量衰减因子
GL_LINEAR_ATTENUATION	0.0	线性衰减因子
GL_QUADRATIC_ATTENUATION	0.0	二次衰减因子

注意：表5-1所列的GL_LIGHT0的GL_SPECULAR和GL_DIFFUSE的默认值和其他光源（GL_LIGHT1、GL_LIGHT2等）不同。对于GL_LIGHT0、GL_DIFFUSE和GL_SPECULAR的默认值都是(1.0, 1.0, 1.0, 1.0)。对于其他光源，这两种光源属性的默认值是(0.0, 0.0, 0.0, 1.0)。

示例程序5-2显示了glLight*()函数的用法。

示例程序5-2 定义光源的颜色和位置

```
GLfloat light_ambient [] ={0.0,0.0,0.0,1.0 };
GLfloat light_diffuse [] ={1.0,1.0,1.0,1.0 };
GLfloat light_specular [] ={1.0,1.0,1.0,1.0 };
GLfloat light_position [] ={1.0,1.0,1.0,0.0 };

glLightfv(GL_LIGHT0,GL_AMBIENT,light_ambient);
glLightfv(GL_LIGHT0,GL_DIFFUSE,light_diffuse);
glLightfv(GL_LIGHT0,GL_SPECULAR,light_specular);
glLightfv(GL_LIGHT0,GL_POSITION,light_position);
```

正如读者所看到的那样，这个例子用数组来定义参数值，并且通过反复调用glLightfv()函数设置不同的参数。在这个例子中，前3个glLightfv()调用是多余的，因为它们指定的是GL_AMBIENT、GL_DIFFUSE和GL_SPECULAR参数的默认值。

注意：记得用glEnable()函数启用每个光源。关于这方面的更多信息，请参阅第5.5.5节。

接下来的几节介绍glLight*()函数的所有参数以及它们可以使用的值。这些参数与场景中用于定义整体光照模型的参数进行交互。关于这两个话题的详细信息，请参阅第5.5节和第5.6节。另外，第5.7节从数学的角度解释了这些参数是如何进行交互的。

5.4.1 颜色

OpenGL允许把与颜色相关的3个不同参数（GL_AMBIENT、GL_DIFFUSE和GL_SPECULAR）

与任何特定的光源相关联。GL_AMBIENT参数表示一个特定的光源在场景中所添加的环境光的RGBA强度。在表5-1中，我们可以看到，在默认情况下是不存在环境光的，因为GL_AMBIENT的默认值是（0.0, 0.0, 0.0, 1.0）。示例程序5-1就使用了这个默认值。如果这个程序像如下代码这样指定了蓝色的环境光，其结果就如彩图13的左侧所示。

```
GLfloat light_ambient[] = { 0.0, 0.0, 1.0, 1.0};
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
```

GL_DIFFUSE参数可能最接近于我们想像的“光的颜色”。它定义了特定的光源在场景中所添加的散射光的RGBA强度。在默认情况下，GL_LIGHT0的GL_DIFFUSE值是（1.0, 1.0, 1.0, 1.0），它产生一种明亮的白光，如彩图13的左侧所示。其他所有光源（GL_LIGHT1，……，GL_LIGHT7）的默认值是（0.0, 0.0, 0.0, 0.0）。

GL_SPECULAR参数影响物体上镜面亮点的颜色。一般情况下，在现实世界中，像玻璃瓶这样的物体具有和照射它表面的光线颜色相同的镜面颜色（通常是白色）。因此，如果我们想创建逼真的效果，可以把GL_SPECULAR参数的值设置为和GL_DIFFUSE参数的值相同。在默认情况下，GL_LIGHT0的GL_SPECULAR值是（1.0, 1.0, 1.0, 1.0），其他所有光源的GL_SPECULAR值是（0.0,0.0, 0.0, 0.0）。

注意：*在启用混合（参见第6章）之前，这些颜色中的alpha成分是不会使用的。在此之前，可以安全地忽略alpha值。*

5.4.2 位置和衰减

如前所述，可以选择让光源位于无限远处，也可以让它靠近场景。第一种类型称为方向性光源。如果光源位于无限远处，当光线到达物体表面时，可以认为所有的光线都是平行的。方向性光源的一个现实世界的例子就是太阳。第二种类型称为位置性光源，因为它在场景中的准确位置决定了它对场景所产生的效果。更具体地说，它决定了光线的方向。台灯就是一个位置性光源的例子。可以在彩图13中看到方向性光源和位置性光源的区别。示例程序5-1所用的光源是方向性光源。

```
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

这段代码把一个包含了4个值（x, y, z, w）的向量提供给GL_POSITION参数。如果最后一个值（w）为0.0，表示对应的光源是方向性光源，（x, y, z）值描述了它的方向。这个方向通过模型视图矩阵进行变换。在默认情况下，GL_POSITION是（0, 0, 1, 0），它定义了一个指向z轴负方向的方向性光源。当然，可以创建一种方向为（0, 0, 0）的方向性光源，但这种光源显然没有任何意义。

如果w是非零值，对应的光源就是位置性光源，（x, y, z）值指定了在相同的物体坐标下这个光源的位置（参见附录F）。这个位置通过模型视图矩阵进行变换，并以视觉坐标的形式存储。关于如何控制光源位置变换的更多信息，请参阅第5.4.5节。另外，在默认情况下，位置性光源向所有的方向发射光线，但可以通过把光源定义为聚光灯，把它的照明范围限制在一个锥体里。关于如何把光源定义为聚光灯的更多信息，请参阅第5.4.3节。

注意：*平滑着色的多边形表面的颜色是通过计算各个顶点的颜色决定的。因此，最好能够避免用本地光源照射大型的多边形。如果把光源靠近多边形的中间，顶点可能由于太远而无法接收到光线，整个多边形看上去会比预想的要暗一些。为了避免这个问题，可以把大型的多边形分解为多个小多边形。*

对于现实世界的光照，随着光源的距离增加，光的强度也随之衰减。由于方向性光源位于无限远处，因此这个原则不适用于方向性光源。但是，我们可能想对位置性光源所发出的光进行衰减。

OpenGL把光源的强度乘以衰减因子，对它实行衰减：

$$\text{衰减因子} = \frac{1}{k_c + k_l d + k_q d^2}$$

其中：

d = 光源和顶点之间的距离

k_c = GL_CONSTANT_ATTENUATION

k_l = GL_LINEAR_ATTENUATION

k_q = GL_QUADRATIC_ATTENUATION

在默认情况下，k_c为1.0，k_l和k_q都是0，但是可以向这些参数提供不同的值：

```
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 2.0);
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.5);
```

注意，环境光、散射光和镜面光的强度都进行了衰减。只有发射光和全局环境光的强度没有衰减。另外，由于衰减需要在每种经过计算产生的颜色上再进行一次除法（可能需要更多的数学计算），因此使用衰减光可能会影响应用程序的性能。

5.4.3 聚光灯

如前所述，可以对位置性光源的形状加以限制，使它的发射范围限于一个锥体之内，就像聚光灯一样。为了创建聚光灯，需要确定光锥的发射范围（记住，由于聚光灯是位置性光源，因此需要设置它们的位置。也可以创建方向性聚光灯，但它无法实现我们所需要的效果）。为了指定光锥轴和光锥边缘之间的角度，可以使用GL_SPOT_CUTOFF参数。光锥的最大角度是这个值的两倍，如图5-2所示。

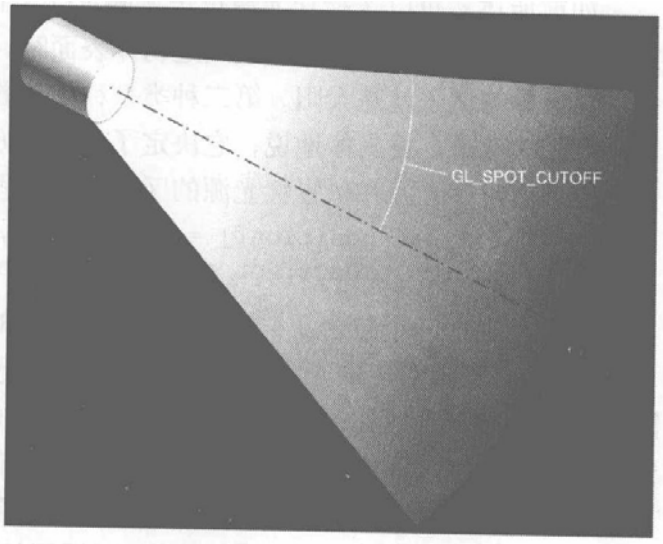


图5-2 GL_SPOT_CUTOFF参数

注意，聚光灯所发出的光不会到达光锥边缘的外侧。在默认情况下，聚光灯这个特性是被禁用的，因为GL_SPOT_CUTOFF的值是180.0。这意味着光是向所有方向发射的（光锥的最大角度是360度，此时已经不再是锥形了）。GL_SPOT_CUTOFF的值被限制在[0.0, 90.0]之间（另外还有180.0这个特殊值）。下面这行代码把光锥切角参数设置为45度：

```
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 45.0);
```

我们还需要指定聚光灯的方向，确定光锥的轴：

```
GLfloat spot_direction[] = { -1.0, -1.0, 0.0 };
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spot_direction);
```

方向是根据物体坐标指定的。默认的方向是（0.0, 0.0, −1.0），因此如果没有显式地设置GL_SPOT_DIRECTION的值，光线将指向z轴的负方向。另外，记住聚光灯的方向就像法线向量一样，也

需要经过模型视图矩阵的变换，并以视觉坐标的形式存储。关于这些变换的更多细节，请参阅第5.4.5节。

除了聚光灯的切角和方向之外，还可以使用两种方法控制光锥内部光的分布强度。首先，可以像前面那样设置衰减因子，这个值将和光源的强度相乘。还可以设置GL_SPOT_EXPONENT参数（默认值为0），控制光的集中度。光的强度在光锥的中心达到最高。越靠近光锥的边缘，光的强度就越弱，衰减的幅度就是光线的方向与光线和它所照射顶点的方向之间夹角的余弦值的聚光指数次方。因此，聚光指数越高，光源的强度就越集中。关于计算光源强度的方程式的更多细节，请参阅第5.7节。

5.4.4 多光源

如前所述，至少可以在场景中指定8个光源（很可能更多，取决于所使用的OpenGL实现）。由于OpenGL需要通过计算来确定每个顶点从每个光源接收多少光，因此光源的数量越多，对应用程序性能的影响就越大。表示8个光源的常量是GL_LIGHT0、GL_LIGHT1、GL_LIGHT2、GL_LIGHT3等。在前面的讨论中，我们设置的是和GL_LIGHT0相关的参数。如果想要另外一个光源，还需要指定它的参数。另外，如表5-1所示，其他光源的默认参数值和GL_LIGHT0并不相同。示例程序5-3定义了一个白色的衰减聚光灯。

示例程序5-3　第二个光源

```
GLfloat light1_ambient [] ={0.2,0.2,0.2,1.0 };
GLfloat light1_diffuse [] ={1.0,1.0,1.0,1.0 };
GLfloat light1_specular [] ={1.0,1.0,1.0,1.0 };
GLfloat light1_position [] ={-2.0,2.0,1.0,1.0 };
GLfloat spot_direction [] ={-1.0,-1.0,0.0 };

glLightfv(GL_LIGHT1,GL_AMBIENT,light1_ambient);
glLightfv(GL_LIGHT1,GL_DIFFUSE,light1_diffuse);
glLightfv(GL_LIGHT1,GL_SPECULAR,light1_specular);
glLightfv(GL_LIGHT1,GL_POSITION,light1_position);
glLightf(GL_LIGHT1,GL_CONSTANT_ATTENUATION,1.5);
glLightf(GL_LIGHT1,GL_LINEAR_ATTENUATION,0.5);
glLightf(GL_LIGHT1,GL_QUADRATIC_ATTENUATION,0.2);

glLightf(GL_LIGHT1,GL_SPOT_CUTOFF,45.0);
glLightfv(GL_LIGHT1,GL_SPOT_DIRECTION,spot_direction);
glLightf(GL_LIGHT1,GL_SPOT_EXPONENT,2.0);

glEnable(GL_LIGHT1);
```

如果把上面这些代码添加到示例程序5-1中，球体将被两个光源所照射，其中一个是方向性光源，另一个是聚光灯。



尝试一下

按照下面的方式修改示例程序5-1：

- 把第一个光源修改为某种颜色的位置性光源，而不是原来的白色方向性光源。
- 另外增加一个某种颜色的聚光灯。提示：使用前一节所示的一些代码。
- 测量这两处修改对应用程序性能的影响。

5.4.5 控制光源的位置和方向

OpenGL处理光源位置和方向的方式类似于处理几何图元的位置。换句话说，控制光源的矩阵变

换和控制图元的矩阵变换相同。更具体地说，当调用glLight*()函数指定一个光源的位置或方向时，这个光源的位置或方向将根据当前的模型视图矩阵进行变换，并以视觉坐标的形式存储。这意味着我们可以通过修改模型视图矩阵的内容来操纵光源的位置和方向（投影矩阵对光源的位置和方向没有影响）。本节介绍如何在程序中根据模型变换或视图变换对设置光源位置的点进行修改，从而实现下面3种效果：

- 光源的位置保持不变。
- 光源沿一个静止的物体移动。
- 光源沿观察点移动。

光源保持静止

在最简单的情况下（如示例程序5-1），光源的位置保持不变。为了实现这个效果，需要在使用了视图和模型变换之后设置光源的位置。示例程序5-4显示了init()函数和reshape()函数可能使用的代码。

示例程序5-4 静止光源

```
glViewport(0,0,(GLsizei)w,(GLsizei)h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
if (w <=h)
   glOrtho(-1.5,1.5,-1.5*h/w,1.5*h/w,-10.0,10.0);
else
   glOrtho(-1.5*w/h,1.5*w/h,-1.5,1.5,-10.0,10.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

/*later in init()*/
GLfloat light_position [] ={1.0,1.0,1.0,1.0 };
glLightfv(GL_LIGHT0,GL_POSITION,position);
```

正如读者所看到的那样，首先确定的是视口和投影矩阵。接着，在模型视图矩阵中加载单位矩阵，然后再设置光源。由于使用了单位矩阵，在乘以模型视图矩阵之后，原先所指定的光源位置（1.0,1.0, 1.0）并没有发生变化。由于在此之后光源位置和模型矩阵都没有发生变化，因此光源的方向一直保持为（1.0, 1.0, 1.0）。

独立地移动光源

现在，假定我们想旋转或移动光源的位置，使光源相对于一个静止物体移动。完成这个任务可以使用的一种方法就是在模型变换之后设置光源位置，而这次模型变换本身就专门用于改变光源的位置。在程序早期的init()函数中，可以使用相同的函数调用序列。然后，需要执行我们所需要的模型变换（在模型视图堆栈上），并重置光源位置。这个操作很可能是在display()函数中完成的。示例程序5-5显示了display()函数可能使用的代码。

示例程序5-5 独立地移动光源

```
static GLdouble spin;

void display(void)
{
   GLfloat light_position [] ={0.0,0.0,1.5,1.0 };
   glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);

   glPushMatrix();
```

```
        gluLookAt(0.0,0.0,5.0,0.0,0.0,0.0,0.0,1.0,0.0);
        glPushMatrix();
            glRotated(spin,1.0,0.0,0.0);
            glLightfv(GL_LIGHT0,GL_POSITION,light_position);
        glPopMatrix();
        glutSolidTorus(0.275,0.85,8,15);
    glPopMatrix();
    glFlush();
}
```

spin是一个全局变量，很可能由输入设备所控制。display()函数对场景进行重绘，使光源沿一个静止的圆环面旋转spin度。注意两对函数glPushMatrix()和glPopMatrix()调用，它们用于隔离视图和模型变换，这两个变换都出现在模型视图堆栈之上。在示例程序5-5中，由于观察点的位置保持固定，因此当前矩阵被压入到堆栈中，然后程序通过gluLookAt()函数加载所需的视图变换矩阵。在指定模型变换glRotated()之前，当前矩阵再次被压入到矩阵堆栈中。然后，光源位置就在新的经过旋转的坐标系统中进行设置，使光源似乎绕着前一个位置进行了旋转（记住，光源位置是以视觉坐标的形式存储的，经过变换之后可以通过模型视图矩阵获取）。经过旋转变换的矩阵从堆栈弹出之后，这个程序接着就绘制圆环面。

在示例程序5-6中，光源绕一个物体旋转。按下鼠标左键时，光源的位置便旋转30度。这个程序绘制了一个小型的没有光照的线框立方体，表示场景中光源的位置。

示例程序5-6 使用模型视图变换移动光源：movelight.c

```
static int spin =0;

void init(void)
{
   glClearColor(0.0,0.0,0.0,0.0);
   glShadeModel(GL_SMOOTH);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
}
/* Here is where the light position is reset after the modeling
 * transformation (glRotated)is called.This places the
 * light at a new position in world coordinates.The cube
 * represents the position of the light.
   */
void display(void)
{
   GLfloat position [] ={0.0,0.0,1.5,1.0 };

   glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);
   glPushMatrix();
   glTranslatef(0.0,0.0,-5.0);

   glPushMatrix();
   glRotated((GLdouble)spin,1.0,0.0,0.0);
   glLightfv(GL_LIGHT0,GL_POSITION,position);

   glTranslated(0.0,0.0,1.5);
```

```
    glDisable(GL_LIGHTING);
    glColor3f(0.0,1.0,1.0);
    glutWireCube(0.1);
    glEnable(GL_LIGHTING);
    glPopMatrix();

    glutSolidTorus(0.275,0.85,8,15);
    glPopMatrix();
    glFlush();
}

void reshape(int w,int h)
{
    glViewport(0,0,(GLsizei)w,(GLsizei)h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(40.0,(GLfloat)w/(GLfloat)h,1.0,20.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void mouse(int button,int state,int x,int y)
{
    switch (button){
       case GLUT_LEFT_BUTTON:
          if (state ==GLUT_DOWN){
             spin =(spin +30)%360;
             glutPostRedisplay();
          }
          break;
       default:
          break;
    }
}
int main(int argc,char**argv)
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB |GLUT_DEPTH);
    glutInitWindowSize(500,500);
    glutInitWindowPosition(100,100);
    glutCreateWindow(argv [0]);
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMouseFunc(mouse);
    glutMainLoop();
    return 0;
}
```

光源和观察点一起移动

为了创建一个与观察点一起移动的光源，需要在视图变换之前设置光源位置。接着进行的视图变换

就会以相同的方式同时影响光源和观察点。记住，光源的位置是以视觉坐标的形式存储的，这也是视觉坐标系统发挥重要作用的少数几个场合之一。在示例程序5-7中，光源的位置是在init()函数中定义的，它按照视觉坐标（0,0,0）的形式存储光源位置。换句话说，光线是从照相机的镜头中发射出来的。

示例程序5-7 光源与观察点一起移动

```
GLfloat light_position [] ={0.0,0.0,0.0,1.0 };

glViewport(0,0,(GLint)w,(GLint)h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(40.0,(GLfloat)w/(GLfloat)h,1.0,100.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glLightfv(GL_LIGHT0,GL_POSITION,light_position);
```

现在，如果观察点进行了移动，光源也跟着它一起移动，它和观察点的距离仍然是（0, 0, 0）。在下面这段代码（续示例程序5-7）中，全局变量（ex, ey, ez）控制观察点的位置，(upx, upy, upz)控制朝上向量的值。display()函数是在事件循环中调用的，用于对场景进行重绘。

```
static GLdouble ex, ey, ez, upx, upy, upz;

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
        gluLookAt(ex, ey, ez, 0.0, 0.0, 0.0, upx, upy, upz);
        glutSolidTorus(0.275, 0.85, 8, 15);
    glPopMatrix();
    glFlush();
}
```

当那个被光照射的圆环面被重绘时，光源位置和观察者位置被移动到相同的位置。随着传递给gluLookAt()函数的值的变化以及观察点位置的移动，这个物体看上去将变暗，因为它始终是从观察点位置被照亮的。即使没有重新指定光源位置，光源也会进行移动，因为视觉坐标系统发生了变化。

光源的这种移动方法就好像矿工帽的照明方式。另外，手持蜡烛或灯笼也属于这种情况。通过调用glLightfv(GL_LIGHTi, GL_POSITION, postion)所指定的光源位置就是观察点与照明源的x、y和z距离。当观察点的位置移动时，光源和观察点仍然保持相同的相对位置。



尝试一下

按照下面的方式修改示例程序5-6：

- 使光源沿物体移动而不是绕着它旋转。提示：在display()函数中用glTranslated()取代第一个glRotated()调用，并选择一个适当的值来代替spin参数。
- 修改衰减因子，使光的强度随着光源和物体之间距离的增加而减弱。提示：使用glLight*()函数设置所需要的衰减参数。

Nate Robin的光源位置教程

如果读者已经下载了Nate Robin的教学程序包，现在就可以运行“lightposition”教程（关于如何下载以及从哪里下载这些程序的详细介绍，请参阅前言部分的“Nate Robin的OpenGL教程”）。在这个教程中，读者可以试验光源位置和模型视图矩阵的效果。

5.5 选择光照模型

OpenGL的光照模型概念包括下面4个部分：

- 全局环境光强度。
- 观察点的位置是位于场景还是位于无限远处。
- 物体的正面和背面是否应该执行不同的光照计算。
- 镜面颜色是否应该从环境和散射颜色中分离出来，并在纹理操作之后再应用。

本节解释如何指定光照模型，并讨论了如何启用光照，也就是告诉OpenGL需要执行光照计算。

兼容性扩展
glLightModel
GL_LIGHT_MODEL_
AMBIENT
GL_LIGHT_MODEL_
LOCAL_VIEWER
GL_LIGHT_MODEL_
TWO_SIDE
GL_LIGHT_MODEL_
COLOR_CONTROL

glLightModel*()函数用于指定光照模型的所有属性。glLightModel*()具有两个参数：光照模型属性以及这个属性需要设置的值。

void **glLightModel**{if}(GLenum *pname*, TYPE *param*);

void **glLightModel**{if}v(GLenum *pname*, const TYPE *param);

设置光照模型的属性。被设置的光照模型的特征是由pname定义的，它指定了一个命名参数（参见表5-2）。param表示pname参数被设置的值。如果是向量版本的函数，它是一个指向一组值的指数。如果是非向量版本的函数，它是参数值本身。非向量版本只能用于设置单值的光照模型属性，并且不能用于GL_LIGHT_MODEL_AMBIENT。

表5-2 glLightModel*()函数的pname参数的默认值

参数名	默认值	含 义
GL_LIGHT_MODEL_ AMBIENT	(0.2, 0.2, 0.2, 1.0)	整个场景的环境光的RGBA强度
GL_LIGHT_MODEL_LOCAL_VIEWER	0.0或GL_FALSE	镜面反射角度是如何计算的
GL_LIGHT_MODEL_TWO_SIDE	0.0或GL_FALSE	指定了单面还是双面光照
GL_LIGHT_MODEL_COLOR_CONTROL	GL_SINGLE_COLOR	镜面颜色的计算是否从环境和散射颜色中分离出来

5.5.1 全局环境光

就像前面所讨论的那样，每个光源都可以为场景的环境光做出贡献。另外，还存在来自任何特定光源的环境光。为了指定这种全局环境光的RGBA强度，可以使用GL_LIGHT_MODEL_AMBIENT参数，如下所示：

```
GLfloat lmodel_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
```

在这个例子中，lmodel_ambient使用的值是GL_LIGHT_MODEL_AMBIENT的默认值。由于这些值所产生的是少量白色的环境光，因此即使我们并没有在场景中指定任何散射光，仍然能够看到场景中的物体。彩图14显示了不同数量的全局环境光的效果。

5.5.2 局部的观察点或无限远的观察点

观察点的位置影响镜面反射所产生的亮点的计算。更具体地说，一个特定顶点上的亮点的强度计算取决于这个顶点的法线、这个顶点和光源的方向以及这个顶点和观察点的距离。记住，观察点实际

上并没有因为调用光照函数而移动（我们并不需要更改投影变换，如第3.3节所述）。反之，光照计算需要做出一些假设，例如观察点是否移动。

如果观察点位于无限远处，它和场景中任何顶点的方向都是固定的。局部的观察点能够产生更为逼真的结果。但是，由于每个顶点都必须计算它和观察点的方向，所以程序的性能会受到影响。在默认情况下，OpenGL假定使用的是无限远的观察点。可以用下面这行代码把观察点改变为局部观察点：

```
glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
```

这个调用把观察点放在视觉坐标系统的（0，0，0）位置。如果想把观察点切换到无限远处，可以通过参数传递GL_FALSE。

5.5.3 双面光照

所有的多边形都要执行光照计算，不管它们是正面还是背面。但是，由于我们在思维中所设置的光照条件一般适用于正面的多边形，因此背面的多边形所接受的光照一般并不正确。在示例程序5-1中，物体是一个球体，只有正面能够看到，因为它们位于球体的外面。因此，在这种情况下，背面多边形看上去是什么样子无关紧要的。但是，如果球体被切开，它的内部也可见，我们可能希望球体的内部也根据我们定义的光照条件接受完全的光照。我们可能还想为背面多边形提供不同的材料描述。当用下面这条命令启用双面光照时：

```
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
```

OpenGL将会反转表面法线的方向，表示背面多边形的法线。一般而言，这意味着它是可见的背面和正面多边形的表面法线，而不是一条指向观察者的直线。结果，所有的多边形都能得到正确的光照。但是，这些额外的操作通常使得双面光照比默认的单面光照要慢一些。

为了禁用双面光照，可以向前面这个函数传递GL_FALSE参数（关于如何为两个面提供材料属性的信息，请参阅第5.6节）。我们还可以用glFrontFace()函数控制多边形的哪一面为正面，详见第2.4.3节中的“反转和剔除多边形表面”。

5.5.4 镜面辅助颜色

对于典型的光照计算，环境、散射、镜面和发射成分经过计算之后简单地叠加在一起。在默认情况下，纹理贴图是在光照之后应用的。因此，镜面亮点会被削弱，纹理的效果可能和预想的不一样。为了把镜面颜色的应用推迟到纹理贴图之后，可以采用下面的做法：

```
glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL,
              GL_SEPARATE_SPECULAR_COLOR);
```

在这种模式下，光照为每个顶点产生两种颜色：一种是主颜色，它由所有非镜面光照颜色组成。另一种是辅助颜色，它是所有镜面光照颜色的总和。在进行纹理贴图时，只有主颜色与纹理颜色进行组合。在纹理贴图之后，辅助颜色被添加到主颜色和纹理颜色的最终组合颜色（参见第9.10节）。同时进行了光照和纹理处理的物体如果使用独立的镜面颜色，通常可以使镜面亮点更为明显，效果也更为逼真。

为了恢复到默认值，可以调用：

```
glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SINGLE_COLOR);
```

这样，主颜色再次包含了所有的光照成分：环境、散射、镜面和发射。在纹理贴图之后不会再添加光照成分。如果并不执行纹理贴图，把镜面颜色从其他光照成分中分离出来就毫无意义。

5.5.5 启用光照

在OpenGL中，需要显式地启用（或禁用）光照。如果未启用光照，当前的颜色就简单地映射到当前的顶点，不存在任何涉及法线、光源、光照模型和材料属性的计算。下面是启用光照的方法：

兼容性扩展
GL_LIGHTING
GL_LIGHTi

```
glEnable(GL_LIGHTING);
```

为了禁用光照，可以用GL_LIGHTING为参数调用glDisable()函数。

指定了每个光源的参数之后，还需要显式地启用每个光源。示例程序5-1只使用了一个光源，也就是GL_LIGHT0：

```
glEnable(GL_LIGHT0);
```

5.6 定义材料属性

读者已经看到了如何创建具有一定特征的光源，以及如何定义程序所需要的光照模型。本节描述如何定义场景中物体的材料属性：环境、散射和镜面颜色、光泽度以及所有发射光的颜色（关于光照和材料属性计算的方程式，请参见第5.7节）。从概念上说，绝大多数材料属性类似于我们在创建光源时所设置的属性。设置材料属性的机制也相似，唯一的区别是它使用的函数是glMaterial*()。

void **glMaterial**{if}(GLenum *face*, GLenum *pname*, TYPE *param*);

void **glMaterial**{if}v(GLenum *face*, GLenum *pname*, const TYPE **param*);

指定一种用于光照计算的当前材料属性。face可以是GL_FRONT、GL_BACK或GL_FRONT_AND_BACK，表示物体的哪些面应该接受光照。pname表示设置的特定材料属性。param提供了具体的属性值，它既可以是指向一组值的指数（向量版本），也可以是属性值本身（非向量版本）。非向量版本只用于设置GL_ SHININESS 。表5-3列出了pname可以使用的值。注意GL_AMBIENT_AND_DIFFUSE允许把环境和散射材料颜色同时设置为同一个RGBA值。

兼容性扩展
glMaterial
GL_AMBIENT
GL_DIFFUSE
GL_AMBIENT_AND_ DIFFUSE
GL_SPECULAR
GL_SHININESS
GL_COLOR_INDEXES

表5-3 glMaterial*()函数的pname参数的默认值

参数名	默认值	含 义
GL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	材料的环境颜色
GL_DIFFUSE	(0.8, 0.8, 0.8, 1.0)	材料的散射颜色
GL_AMBIENT_AND_DIFFUSE		材料的环境和散射颜色
GL_SPECULAR	(0.0, 0.0, 0.0, 1.0)	材料的镜面颜色
GL_SHININESS	0.0	镜面指数
GL_EMISSION	(0.0, 0.0, 0.0, 1.0)	材料的发射颜色
GL_COLOR_INDEXES	(0, 1, 1)	环境、散射和镜面颜色索引

正如在第5.3节中的“选择光照模型”讨论的那样，我们可以为物体的正面和背面多边形分别选择不同的光照计算方法。如果背面确实可能看到，可以使用glMaterial*()函数的face参数为正面和背面提供不同的材料属性。彩图14提供了一个例子，有一个物体的内侧和外侧使用了不同的材料属性。

为了让读者领略通过设置材料属性可以实现的效果，可以观察彩图16。这张图使用不同的材料属

性集绘制同一个物体。整张图使用了相同的光源和光照模型。接下来的各节将讨论用于绘制各个球体的特定属性。

注意，glMaterial*()设置的绝大多数属性都是颜色（R、G、B、A）。不管我们向其他参数提供的alpha值是什么，所有顶点的alpha值都是材料的散射颜色的alpha值，也就是在glMaterial*()函数中提供给GL_DIFFUSE的alpha值（具体的方法在下一节描述）。关于alpha颜色的详细讨论，请参阅第6章。另外，注意RGBA材料属性并不适用于颜色索引模型（关于颜色索引模式下相关参数的更多信息，可以参阅第5.8节）。

5.6.1 散射和环境反射

glMaterial*()设置的GL_DIFFUSE和GL_AMBIENT参数影响物体反射的散射光和环境光的颜色。当人眼感知物体的颜色时，散射颜色扮演了最为重要的一个角色。人眼对物体颜色的知觉受到入射散射光的颜色以及入射光相对于法线方向角度的影响（当入射光垂直于表面时，物体的颜色看上去最深）。观察点的位置对散射颜色没有任何影响。

环境颜色影响物体的整体颜色。当物体被直接照亮时，散射颜色占据主导地位。因此，只有当物体并没有受到直接照射时，环境颜色才会占据重要地位。物体的总体环境颜色受到全局环境光和各个光源所发出的环境光的影响。和散射颜色一样，环境颜色并不受观察点位置的影响。

对于现实世界的物体，散射和环境颜色通常是相同的。由于这个原因，OpenGL提供了一种方便的方法，用glMaterial*()函数为它们赋一个相同的值：

```
GLfloat mat_amb_diff[] = { 0.1, 0.5, 0.8, 1.0 };
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE,
             mat_amb_diff);
```

在这个例子中，RGBA颜色（0.1, 0.5, 0.8, 1.0，是一种深蓝）表示正面和背面多边形的当前环境颜色和散射颜色。

在彩图16中，第一排的球体不存在环境颜色（0.0, 0.0, 0.0, 0.0,），第二排的球体具有相当强的环境颜色（0.7, 0.7, 0.7, 1.0）。

5.6.2 镜面反射

物体的镜面反射将产生亮点。与环境和散射颜色不同，观察者所看到的镜面颜色的数量取决于观察点的位置，在反射角方向上的镜面颜色最亮。我们可以想像室外日光下的一个金属球，当我们转动脑袋时，阳光所产生的亮点将随之移动。但是，如果我们转动脑袋的幅度太大，可能完全看不到亮点。

OpenGL允许我们在不存在反射光（使用GL_SPECULAR）的情况下设置材料的效果，并控制亮点的大小和亮度（使用GL_SHININESS）。可以向GL_SHININESS赋一个范围在[0.0, 128.0]之间的值。这个值越大，亮点就更小、更亮（关于如何计算镜面亮点的细节，可以参阅第5.7节）。

在彩图16中，第一列的球体不存在镜面反射。第二列球体的GL_SPECULAR和GL_SHININESS所赋的值如下：

```
GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat low_shininess[] = { 5.0 };
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, low_shininess);
```

在第三列中，GL_SHININESS参数的值增加到100.0。

5.6.3 发射光颜色

通过为GL_EMISSION参数指定一个RGBA颜色值，可以使物体看上去好像发射出这种颜色的光。由于绝大多数现实世界的物体（除了光源之外）自己都不会发光，因此我们很可能只有在模拟场景中的台灯和其他光源时才会使用这个特性。在彩图16中，第4列球体的GL_EMISSION值是一种灰红色：

```
GLfloat mat_emission[] = {0.3, 0.2, 0.2, 0.0};
glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
```

注意，这些球体看上去有些微微发光。但是，它们实际上并不是作为光源。我们需要创建一个光源，并把它放在和球体相同的位置，以创建这种效果。

5.6.4 更改材料属性

示例程序5-1在场景中唯一的那个物体（球体）的所有顶点上使用相同的材料属性。在其他情况下，我们可能想为同一个物体的不同顶点赋以不同的材料属性值。场景中的物体很可能不止一个，并且每个物体的材料属性很可能并不相同。例如，生成彩图16的代码必须绘制12个不同的物体（均为球体），每个物体都有不同的材料属性。示例程序5-8显示了display()函数的部分代码。

示例程序5-8 不同的材料属性：material.c

```
   GLfloat no_mat [] ={0.0,0.0,0.0,1.0 };
   GLfloat mat_ambient [] ={0.7,0.7,0.7,1.0 };
   GLfloat mat_ambient_color [] ={0.8,0.8,0.2,1.0 };
   GLfloat mat_diffuse [] ={0.1,0.5,0.8,1.0 };
   GLfloat mat_specular [] ={1.0,1.0,1.0,1.0 };
   GLfloat no_shininess [] ={0.0 };
   GLfloat low_shininess []={5.0 };
   GLfloat high_shininess [] ={100.0 };
   GLfloat mat_emission [] ={0.3,0.2,0.2,0.0};

   glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);

/*draw sphere in first row,first column
 *diffuse reflection only;no ambient or specular
 */
   glPushMatrix();
   glTranslatef(-3.75,3.0,0.0);
   glMaterialfv(GL_FRONT,GL_AMBIENT,no_mat);
   glMaterialfv(GL_FRONT,GL_DIFFUSE,mat_diffuse);
   glMaterialfv(GL_FRONT,GL_SPECULAR,no_mat);
   glMaterialfv(GL_FRONT,GL_SHININESS,no_shininess);
   glMaterialfv(GL_FRONT,GL_EMISSION,no_mat);
   glutSolidSphere(1.0,16,16);
   glPopMatrix();

/*draw sphere in first row,second column
 *diffuse and specular reflection;low shininess;no ambient
 */
   glPushMatrix();
   glTranslatef(-1.25,3.0,0.0);
   glMaterialfv(GL_FRONT,GL_AMBIENT,no_mat);
   glMaterialfv(GL_FRONT,GL_DIFFUSE,mat_diffuse);
   glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
```

```
    glMaterialfv(GL_FRONT,GL_SHININESS,low_shininess);
    glMaterialfv(GL_FRONT,GL_EMISSION,no_mat);
    glutSolidSphere(1.0,16,16);
    glPopMatrix();
/* draw sphere in first row,third column
 * diffuse and specular reflection;high shininess;no ambient
 */
    glPushMatrix();
    glTranslatef(1.25,3.0,0.0);
    glMaterialfv(GL_FRONT,GL_AMBIENT,no_mat);
    glMaterialfv(GL_FRONT,GL_DIFFUSE,mat_diffuse);
    glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
    glMaterialfv(GL_FRONT,GL_SHININESS,high_shininess);
    glMaterialfv(GL_FRONT,GL_EMISSION,no_mat);
    glutSolidSphere(1.0,16,16);
    glPopMatrix();
/* draw sphere in first row,fourth column
 * diffuse reflection;emission;no ambient or specular refl.
 */
    glPushMatrix();
    glTranslatef(3.75,3.0,0.0);
    glMaterialfv(GL_FRONT,GL_AMBIENT,no_mat);
    glMaterialfv(GL_FRONT,GL_DIFFUSE,mat_diffuse);
    glMaterialfv(GL_FRONT,GL_SPECULAR,no_mat);
    glMaterialfv(GL_FRONT,GL_SHININESS,no_shininess);
    glMaterialfv(GL_FRONT,GL_EMISSION,mat_emission);
    glutSolidSphere(1.0,16,16);
    glPopMatrix();
```

正如读者所看到的那样，这段代码反复调用glMateriafv()函数，来设置每个球体的材料属性。注意，只有在需要重新指定材料属性时，才需要调用这个函数。第2、3和4个球体使用和第1个球体相同的环境和散射属性，因此这些参数无需重新指定。由于调用glMaterial*()函数需要一定的开销，因此示例程序5-8应该进行重写，尽可能少地修改材料属性。

Nate Robin的光照材料教程

如果读者已经下载了Nate Robin的教学程序包，现在可以运行“lightmaterial”教程。在这个教程中，读者可以试验不同的材料属性颜色，包括环境、散射和镜面颜色，以及光亮指数。

5.6.5 颜色材料模式

为了减少修改材料属性所带来的开销，可以采用的另一种方法是使用glColorMaterial()。

兼容性扩展
glColorMaterial

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

使表面（由face参数指定）的材料属性（由mode参数指定）总是使用当前颜色。当前颜色的变化（使用glColor*()）将会立即更新指定的材料属性。face参数可以是GL_FRONT、GL_BACK或GL_FRONT_AND_BACK（默认值）。mode参数可以是GL_AMBIENT、GL_DIFFUSE、GL_SPECULAR、GL_AMBIENT_AND_DIFFUSE（默认值）和GL_EMISSION。在任何时候，只有一种模式处于活动状态。glColorMaterial()对于颜色索引模式没有效果。

注意，glColorMaterial()指定了两个独立的值：第一个值决定哪一（或哪些）面被更新，第二个值决定了哪个（或哪些）材料属性被更新。OpenGL并没有为每个面维护一个独立的mode变量。

在调用glColorMaterial()函数之后，我们需要以GL_COLOR_MATERIAL为参数调用glEnable()函数。之后，可以根据需要使用glColor*()函数更改当前颜色（或者用glMaterial*()函数更改其他材料属性）：

```
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT, GL_DIFFUSE);
/* now glColor* changes diffuse reflection  */
glColor3f(0.2, 0.5, 0.8);
/* draw some objects here */
glColorMaterial(GL_FRONT, GL_SPECULAR);
/* glColor* no longer changes diffuse reflection
 * now glColor* changes specular reflection  */
glColor3f(0.9, 0.0, 0.2);
/* draw other objects here */
glDisable(GL_COLOR_MATERIAL);
```

对于场景中的大多数顶点，如果想更改单个材料参数的值，可以使用glColorMaterial()。如果想更改多个材料参数的值（就像彩图16那样），可以使用glMaterial *（）。如果不再需要使用glColorMaterial()的功能时，确保将其禁用，以避免得到出乎预料的材料属性，并避免与此相关的性能开销。glColorMaterial()对性能所造成的影响因不同的OpenGL实现而异。有些实现可能会对顶点函数进行优化，使它们能够根据当前颜色快速地对材料属性进行更新。

示例程序5-9显示了一个交互式的程序，它使用glColorMaterial()更改材料属性。可以通过按下3个鼠标按钮更改散射属性的颜色。

示例程序5-9 使用glColorMaterial()：colormat.c

```
GLfloat diffuseMaterial [4] ={0.5,0.5,0.5,1.0 };

void init(void)
{
   GLfloat mat_specular [] ={1.0,1.0,1.0,1.0 };
   GLfloat light_position [] ={1.0,1.0,1.0,0.0 };

   glClearColor(0.0,0.0,0.0,0.0);
   glShadeModel(GL_SMOOTH);
   glEnable(GL_DEPTH_TEST);
   glMaterialfv(GL_FRONT,GL_DIFFUSE,diffuseMaterial);
   glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
   glMaterialf(GL_FRONT,GL_SHININESS,25.0);
   glLightfv(GL_LIGHT0,GL_POSITION,light_position);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);

   glColorMaterial(GL_FRONT,GL_DIFFUSE);
   glEnable(GL_COLOR_MATERIAL);
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);
   glutSolidSphere(1.0,20,16);
```

```
    glFlush();
}

void reshape(int w,int h)
{
    glViewport(0,0,(GLsizei)w,(GLsizei)h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <=h)
       glOrtho(-1.5,1.5,-1.5*(GLfloat)h/(GLfloat)w,
          1.5*(GLfloat)h/(GLfloat)w,-10.0,10.0);
    else
       glOrtho(-1.5*(GLfloat)w/(GLfloat)h,
          1.5*(GLfloat)w/(GLfloat)h,-1.5,1.5,-10.0,10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void mouse(int button,int state,int x,int y)
{
    switch (button){
       case GLUT_LEFT_BUTTON:
          if (state ==GLUT_DOWN){   /*change red */
             diffuseMaterial [0] +=0.1;
             if (diffuseMaterial [0] >1.0)
                diffuseMaterial [0] =0.0;
             glColor4fv(diffuseMaterial);
             glutPostRedisplay();
          }
          break;
       case GLUT_MIDDLE_BUTTON:
          if (state ==GLUT_DOWN){   /*change green */
             diffuseMaterial [1] +=0.1;
             if (diffuseMaterial [1] >1.0)
                diffuseMaterial [1] =0.0;
             glColor4fv(diffuseMaterial);
             glutPostRedisplay();
          }
          break;
       case GLUT_RIGHT_BUTTON:
          if (state ==GLUT_DOWN){   /*change blue */
             diffuseMaterial [2] +=0.1;
             if (diffuseMaterial [2] >1.0)
                diffuseMaterial [2] =0.0;
             glColor4fv(diffuseMaterial);
             glutPostRedisplay();
          }
          break;
       default:
          break;
    }
}
```

```
int main(int argc,char**argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB |GLUT_DEPTH);
   glutInitWindowSize(500,500);
   glutInitWindowPosition(100,100);
   glutCreateWindow(argv [0]);
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMouseFunc(mouse);
   glutMainLoop();
   return 0;
}
```



尝试一下

按照下面的方式，对示例程序5-8进行修改：

- 更改场景中的全局环境光。提示：更改GL_LIGHT_MODEL_AMBIENT参数的值。
- 更改散射、环境和镜面反射参数、光泽度和反射颜色。提示：使用glMaterial*()函数，但是要避免过多地调用这个函数。
- 使用双面材料，并增加一个额外的裁剪平面，以便能够看到一行或一列球体的内面和外面（如果读者忘了什么是其他裁剪平面，可以参阅第3.7节）。提示：用GL_LIGHT_MODEL_TWO_SIDE参数打开双面光照，设置所需要的材料属性，然后增加一个裁剪平面。
- 删除用于设置GL_DIFFUSE值的glMaterialfv()调用，并使用效率更高的glColorMaterial()函数实现相同的光照。

5.7 和光照有关的数学知识



高级知识

本节介绍OpenGL在执行光照计算以确定RGBA模式下的颜色时所使用的方程式（有关颜色索引模式下的对应内容，请参阅第5.8节中的“和颜色索引模式有关的数学知识”）。如果读者愿意通过试验获取自己所需要的光照条件，也可以略过本节。即使读完了本节，仍然需要进行一些试验，这样读者会对光照的参数值对顶点颜色的影响有更深的了解。注意，如果并未启用光照，顶点的颜色就是当前颜色。如果启用了光照，OpenGL将在视觉坐标系统中执行本节所描述的光照计算。

在下面这些方程式中，R、G和B成分的数学运算是分别进行的。因此，如下面这个例子所示，如果3个条件加在一起，每个条件的R、G、B值是分别进行添加的，以形成最终的RGB颜色（$R_1 + R_2 + R_3, G_1 + G_2 + G_3, B_1 + B_2 + B_3$）。当这3个条件相乘时，计算方式是（$R_1 R_2 R_3, G_1 G_2 G_3, B_1 B_2 B_3$）。记住，顶点的最终A（alpha）成分等于它的材料散射alpha值。

使用光照之后，顶点的颜色是按照下面的公式计算的：

顶点的颜色= 顶点处的材料发射颜色+ 全局环境光（在顶点处根据材料环境颜色属性进行缩放）+ 经过适当衰减的来自所有光源的环境、散射和镜面光成分

在执行光照计算之后，颜色值将在[0, 1]的范围之内（RGBA模式下）进行截取。

注意，OpenGL的光照计算并不考虑其中一个物体遮挡其他物体的可能性。因此，它不会自动创

建阴影（关于创建阴影的技巧，请参阅第14.15节）。另外，在OpenGL中，被光照的物体并不会向其他物体发射光线。

5.7.1 材料的发射光

材料的发射光非常简单，就是赋值给GL_EMISSION参数的RGB值。

5.7.2 经过缩放的全局环境光

全局环境光（由GL_LIGHT_MODEL_AMBIENT参数定义）与材料的环境颜色属性（glMaterial*()所赋的GL_AMLIENT值）相乘：

$$\text{ambient}_{\text{light model}} * \text{ambient}_{\text{material}}$$

这两个参数的R、G和B值都分别相乘，以计算最终全局环境光的RGB值：$(R_1 R_2, G_1 G_2, B_1 B_2)$。

5.7.3 光源的贡献

每个光源都会对顶点的颜色做出贡献，这些贡献将会叠加在一起。下面这个公式计算每个光源所做出的贡献：

每个光源的贡献= 衰减因子 × 聚光灯效果 ×（环境光成分+ 散射光成分+ 镜面成分）

衰减因子

衰减因子是在第5.4.2一节中描述的：

$$\text{衰减因子} = \frac{1}{k_c + k_l d + k_q d^2}$$

其中：

d = 光源的位置和顶点之间的距离

k_c = GL_CONSTANT_ATTENUATION

k_l = GL_LINEAR_ATTENUATION

k_q = GL_QUADRATIC_ATTENUATION

如果光源为方向性光源，那么衰减因子为1。

聚光灯效果

聚光灯效果根据光源是否为聚光灯以及顶点是否位于聚光灯的光锥范围之内取如下3个值之一：

- 如果光源不是聚光灯（GL_SPOT_CUTOFF为180.0），则取值为1。
- 如果光源为聚光灯，但是顶点位于聚光灯光锥的外面，则取值为0。
- 其他情况为$(\max\{\mathbf{v} \cdot \mathbf{d}, 0\})^{\text{GL_SPOT_EXPONENT}}$，其中：

$\mathbf{v} = (v_x, v_y, v_z)$，它是从顶点指向聚光灯（GL_POSITION）的单位向量。

$\mathbf{d} = (d_x, d_y, d_z)$，它是聚光灯的方向（GL_SPOT_DIRECTION），它假定光源是聚光灯，并且顶点位于聚光灯所产生的光锥的内部。

v和d这两个向量的点积随着它们之间角度的余弦值而发生变化。因此，正对聚光灯方向的物体能够得到最强的光照。离光锥轴线的位置越远，物体受到照射的强度也就越弱。

为了判断一个特定的顶点是否位于光锥的内部，OpenGL将计算$(\max\{\mathbf{v} \cdot \mathbf{d}, 0\})$，其中v和d的定义如上所述。如果这个值小于聚光灯的切角（GL_SPOT_CUTOFF）的余弦值，那么这个顶点就位于光锥的外面。否则，它就位于光锥的内部。

环境光成分

环境光成分就是光源的环境颜色根据材料的环境属性进行缩放后的值：

$$\text{ambient}_{\text{light}} * \text{ambient}_{\text{material}}$$

散射光成分

散射光成分需要考虑光线是否直接照射在顶点上、光源的散射颜色以及材料的散射属性：

$$(\max\{L \cdot n, 0\}) * \text{diffuse}_{\text{light}} * \text{diffuse}_{\text{material}}$$

其中：

$L = (L_x, L_y, L_z)$，它是从顶点到光源位置（GL_POSITION）的单位向量。

$n = (n_x, n_y, n_z)$是顶点的单位法线向量。

镜面光成分

镜面光成分还取决于光线是否直接落在顶点上。如果$L \cdot n$小于等于0，顶点上就不存在镜面光（如果这个值小于0，表示光线位于表面的另一面）。如果存在镜面光成分，它的值取决于如下因素：

- 顶点（n_x, n_y, n_z）处的单位法线向量。
- 两个单位向量之和。这两个单位向量分别是：a) 从顶点指向光源位置（或光源方向）的向量；b) 从顶点指向观察点的向量（前提是GL_LIGHT_MODEL_LOCAL_VIEWER为真。否则，使用向量（0, 0, 1））。把这两个向量的和规范化（把每个成分除以向量的长度），产生$s = (s_x, s_y, s_z)$。
- 镜面指数（GL_SHININESS）。
- 光源的镜面颜色（$\text{GL_SPECULAR}_{\text{light}}$）。
- 材料的镜面属性（$\text{GL_SPECULAR}_{\text{material}}$）。

OpenGL根据这些定义来计算镜面光成分，如下所示：

$$(\max\{s \cdot n, 0\})^{\text{shininess}} \times \text{specular}_{\text{light}} \times \text{specular}_{\text{material}}$$

但是，如果$L \cdot n = 0$，则镜面光成分为0。

5.7.4 完整的光照计算公式

根据上面所描述的这些定义，下面是RGBA模式下的完整光照计算公式：

$$\begin{aligned}\text{顶点颜色} = {} & \text{emission}_{\text{material}} \\ & + \text{ambient}_{\text{light model}} \times \text{ambient}_{\text{material}} \\ & + \sum_{i=0}^{n-1}\left(\frac{1}{k_c + k_l d + k_q d^2}\right)_i \times (\text{聚光灯效果})_i \\ & \times [\text{ambient}_{\text{light}} \times \text{ambient}_{\text{material}} \\ & + (\max\{\mathbf{L} \cdot \mathbf{n}, 0\}) \times \text{diffuse}_{\text{light}} \times \text{diffuse}_{\text{material}} \\ & + (\max\{\mathbf{s} \cdot \mathbf{n}, 0\})^{\text{shininess}} \times \text{specular}_{\text{light}} \times \text{specular}_{\text{material}}]_i\end{aligned}$$

5.7.5 镜面辅助颜色

如果当前的光照模型颜色控制为GL_SEPARATE_SPECULAR_COLOR，那么每个顶点都将产生主颜色和辅助颜色，它们的计算方法如下：

主颜色= 顶点的材料发射颜色

+ 根据顶点处的材料环境颜色属性进行缩放后的全局环境光
+ 来自所有光源并进行了适当衰减的环境光和散射光成分
辅助颜色= 来自所有光源并进行了适当衰减的镜面光成分

下面这两个公式用于主颜色和辅助颜色的光照计算：

$$\begin{aligned}\text{主颜色} = {} & \text{emission}_{\text{material}} \\ & + \text{ambient}_{\text{light model}} \times \text{ambient}_{\text{material}} \\ & + \sum_{i=0}^{n-1}\left(\frac{1}{k_c + k_l d + k_q d^2}\right)_i \times (\text{聚光灯效果})_i \\ & \times [\text{ambient}_{\text{light}} \times \text{ambient}_{\text{material}} \\ & + (\max\{\mathbf{L}\cdot\mathbf{n}, 0\}) \times \text{diffuse}_{\text{light}} \times \text{diffuse}_{\text{material}}]_i\end{aligned}$$

$$\begin{aligned}\text{辅助颜色} = {} & \sum_{i=0}^{n-1}\left(\frac{1}{k_c + k_l d + k_q d^2}\right)_i \times (\text{聚光灯效果})_i \\ & \times [(\max\{\mathbf{s}\cdot\mathbf{n}, 0\})^{\text{shininess}} \times \text{specular}_{\text{light}} \times \text{specular}_{\text{material}}]_i\end{aligned}$$

在进行纹理贴图时，只有主颜色与纹理颜色进行混合。在纹理操作之后，辅助颜色被添加到主颜色和纹理颜色混合所产生的颜色之中。

5.8 颜色索引模式下的光照

在颜色索引模式下，组成RGBA值的参数或者没有效果，或者具有特殊的解释。由于在颜色索引模式下实现相应的效果要困难得多，因此应该尽可能地使用RGBA模式。事实上，在RGBA模式下所使用的光源、光照模式或材料参数中，只有光源参数GL_DIFFUSE和GL_SPECULAR以及材料参数GL_SHININESS可以用于颜色索引模式。GL_DIFFUSE和GL_SPECULAR（分别用d_i和s_i表示）用于计算散射光和镜面光强度（d_{ci}和s_{ci}）的颜色索引值，如下所示：

$$d_{ci} = 0.30\text{R}(d_i) + 0.59\text{G}(d_i) + 0.11\text{B}(d_i)$$
$$s_{ci} = 0.30\text{R}(s_i) + 0.59\text{G}(s_i) + 0.11\text{B}(s_i)$$

其中R(x)、G(x)和B(x)分别表示颜色x的红、绿和蓝色成分。权值0.30、0.59和0.11反映了人眼对红、绿、蓝色的“感知”权重。人眼对绿色最为敏感，对蓝色最不敏感。

为了在颜色索引模式下指定材料颜色，可以在glMaterial*()函数中使用特殊参数GL_COLOR_INDEXES，如下所示：

```
GLfloat mat_colormap[] = { 16.0, 47.0, 79.0 };
glMaterialfv(GL_FRONT, GL_COLOR_INDEXES, mat_colormap);
```

为GL_COLOR_INDEXES提供的3个数分别指定了环境、散射和镜面材料颜色的颜色索引值。换句话说，OpenGL把第一个索引值（此例中为16.0）相关联的颜色作为纯环境颜色，把第二个索引值（47.0）相关联的颜色作为纯散射颜色，把第三个索引值（79.0）相关联的颜色作为纯镜面颜色。在默认情况下，环境颜色索引值是0.0，散射和镜面颜色索引值都是1.0。注意，glColorMaterial()函数在颜色索引模式下没有任何效果。

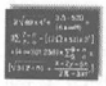
当OpenGL绘制场景时，它使用与这些数值之间的索引值相关联的颜色对场景中的物体进行着色。因此，我们必须根据指定的索引值创建一个颜色表（在此例中，是在索引值16和47之间，然后在47

和79之间）。颜色表常常是按照渐变的方式创建的，但是也可以使用其他方式来实现不同的效果。下面是一个渐变形式的颜色表例子，它从黑色的环境颜色开始，然后是洋红色的散射颜色，最后是白色的镜面颜色：

```
for (i = 0; i < 32; i++) {
   glutSetColor(16 + i, 1.0 * (i/32.0), 0.0, 1.0 * (i/32.0));
   glutSetColor(48 + i, 1.0, 1.0 * (i/32.0), 1.0);
}
```

GLUT库的glutSetColor()函数接受4个参数。它把第1个参数所表示的颜色索引值与最后3个参数所指定的RGB三元组相关联。当i = 0时，颜色索引值16就与RGB值（0.0, 0.0, 0.0）（黑色）相关联。这个颜色表是以渐变的形式创建的，直到索引值47（当i = 31时）处的散射材料颜色，它是一种纯的洋红色（RGB值为1.0, 0.0, 1.0）。第二个循环创建自洋红色的散射材料颜色到白色（索引值79）的镜面颜色之间的颜色表。彩图15显示了一个单光源照射下的球体，它所使用的就是这个颜色表。

和颜色索引光照模式有关的数学知识



高级话题

正如读者所预想的那样，由于颜色索引模式和RGBA模式允许接受的参数不同，它们的计算方法也不相同。由于颜色索引模式不存在材料发射和环境光，因此只有RGBA光照计算公式中用于计算光源的散射成分和镜面成分以及光泽度的那部分内容才适用于颜色索引模式下的光照计算。而且，即使是这些部分也需要进行修改。

首先是RGBA光照计算公式中的散射光成分和镜面光成分。在散射光成分中，我们用前一节为颜色索引模式所定义的d_{ci}来代替$\text{diffuse}_{\text{light}} \times \text{diffuse}_{\text{material}}$。类似地，在镜面光成分中，用前一节所定义的$s_{ci}$来代替$\text{specular}_{\text{light}} \times \text{specular}_{\text{material}}$。（计算衰减、聚光灯效果和所有其他成分的方式和前面相同。）我们把经过修改的散射光成分和镜面光成分分别称为d和s。现在，假定$s' = \min\{s, 1\}$，我们就可以像下面这样进行计算：

$$c = a_m + d(1 - s')(d_m - a_m) + s'(s_m - a_m)$$

其中，a_m、d_m和s_m是使用GL_COLOR_INDEXES指定的环境、散射和镜面材料颜色索引值。最终的颜色索引值是：

$$c' = \min\{c, s_m\}$$

在执行光照计算之后，颜色索引值被转换为定点形式（二进制小数点右边的位数未定）。然后，整数部分用$2^n - 1$进行屏蔽（使用位逻辑操作AND），其中n是颜色索引模式下的颜色位数。

第 6 章　混合、抗锯齿、雾和多边形偏移

本章目标

- 对颜色进行混合，实现像“半透明”这样的效果。
- 使用抗据齿技术，使直线和多边形的锯齿状边缘变得平滑。
- 创建具有逼真大气效果的场景。
- 根据与观察点的距离，渲染各种大小的圆点（经过抗锯齿处理）。
- 在相同（或接近）的深度值下绘制几何图形，但是在几何图形相交的地方避免出现不真实的人工痕迹。

前面的各章提供了创建计算机图形场景所需要的基本信息。现在，我们已经学习了如何完成下面这些任务：

- 绘制几何形状。
- 对几何形状进行变换，以便从任何观察点对它们进行观察。
- 指定场景中的物体应该使用什么颜色，应该使用哪种着色方式。
- 增加光源，并解释它们如何影响场景中的物体。

现在，我们可以深入一步。本章讨论了5种技巧，可以利用它们，为自己的场景增光添彩。这些技巧都不难使用。事实上，解释它们比使用它们的难度要大得多。这些技巧分别在下面各节中描述：

- 混合：介绍如何指定一个混合函数，把源颜色和目标颜色进行混合。混合的最终效果是使场景看上去像是半透明的。
- 抗锯齿：解释一种相对微妙的颜色更改效果，使点、直线和多边形的边缘看上去显得更加平滑一些。多重采样是一个功能强大的技巧，可以对场景中的所有图元进行抗锯齿处理。
- 雾：描述如何创建深度的幻觉。这是通过根据物体和观察点的距离来计算物体的颜色而实现的。因此，远处的物体看上去逐渐变淡，并最终融入到背景中，这和现实生活中我们所感受的情况相同。
- 点参数：讨论一种有效的技巧，根据和观察点的距离，渲染不同大小和颜色的点。点参数可以用于对微粒系统进行建模。
- 多边形偏移：如果想在一个实心物体的上面绘制一个线框轮廓，并使用和那个物体相同的顶点，读者可能已经注意到使用普通方法所产生的丑陋的人工痕迹。本节展示了如何对深度值进行偏移，使一个带轮廓的实心物体看上去更为漂亮。

OpenGL 1.1增加了多边形偏移功能。

OpenGL 1.2引入了可选择的混合方程式（glBlendEquation()）、常量混合颜色（glBlendColor()）以及下面这些混合因子：GL_CONSTANT_COLOR、GL_ONE_MINUS_ CONSTANT_COLOR、GL_CONSTANT_ALPHA和GL_ONE_MINUS_ CONSTANT_ ALPHA。

在OpenGL 1.2和1.3版本中，这些功能只在那些支持图像处理子集的OpenGL实现中得到支持。在1.4版本中，混合方程式、常量混合颜色和常量混合因子都已经成为OpenGL的核心特性。

在OpenGL 1.3版本中，多重采样也成为OpenGL的核心特性。

OpenGL 1.4版本还增加了下面这些新特性，它们也将在本章中讨论：

- 使用GL_SRC_COLOR和GL_ONE_MINUS_SRC_COLOR作为源混合因子。在OpenGL 1.4之前，GL_SRC_COLOR和GL_ONE_MINUS_SRC_COLOR都必须作为目标混合因子。
- 使用GL_DST_COLOR和GL_ONE_MINUS_DST_COLOR作为目标混合因子。在OpenGL 1.4之前，GL_DST_COLOR和GL_ONE_MINUS_DST_COLOR都必须作为源混合因子。
- 显式地指定雾坐标。
- 用于控制点图元特征的点参数。
- 使用独立的混合函数，混合RGB和alpha成分。

6.1 混合

alpha值到底是什么东西呢？读者已经看到过alpha值（RGBA中的A），到目前为止它一直都是被忽略的。当使用glClearColor()函数指定一种清除颜色，或者当我们指定像材料属性和光源强度这样的光照参数时，可以用glColor*()函数指定alpha值。正如第4章所述，显示器屏幕上的像素发出红、绿和蓝光，它们的强度是由红、绿、蓝成分的值所控制的。那么，alpha值是如何影响屏幕上绘制的物体的呢？

如果启用了混合，alpha值常常用于把被处理片断的颜色值与已经存储在帧缓冲区中的像素颜色值进行组合。混合是在场景进行了光栅化并转换为像素之后，但是在最终的像素绘制到帧缓冲区之前发生的。alpha值可以用于alpha测试，根据片断的alpha值决定接受或拒绝这个片断（关于这个处理过程的更多信息，请参阅第10章）。

如果不使用混合，每个新片断将会改写帧缓冲区中已经存在的颜色值，就好像该片断是不透明的那样。如果使用了混合，我们可以控制原来的颜色值与新片断的颜色值的组合方式。因此，可以通过alpha混合来创建半透明的片断，仍然保留一些原先存储的颜色值。颜色混合是诸如透明化、数字合成、油漆这类技巧的核心。

注意：*在颜色索引模式下不能指定alpha值，因此混合操作在颜色索引模式下是非法的。*

对于混合操作，最自然的思考方式是把片断的RGB成分看成是它的颜色，把alpha成分看成是它的透明度。透明和半透明表面的不透明性要低于不透明表面，也就是它们的alpha值要低于不透明表面。例如，如果透过绿色玻璃观察一个物体，我们看到的颜色部分来自于玻璃的绿色，部分来自于物体的颜色。这两种颜色所占的比例取决于玻璃的传播属性：如果玻璃传播撞击它的光线的80%（即不透明度为20%），我们看到的颜色就是玻璃颜色的20%加上玻璃后面那个物体的颜色的80%。

我们可以很容易地想到使用半透明表面的各种情形。例如，如果观察一辆汽车，汽车内部和观察点之间有一层玻璃。我们可以透过两层玻璃看到汽车后面的一些物体。

6.1.1 源因子和目标因子

在混合过程中，新片断（源）的颜色值与当前已经存储的对应像素（目标）的颜色值之间的组合是分两步进行的。首先，我们需要指定如何计算源和目标混合因子。这两个因子都是RGBA四元组，分别与源颜色和目标颜色的R、G、B和A值相乘。然后，这两个RGBA四元组中对应的成分再进行组合。为了用数学的方式来描述这个过程，假定源和目标混合因子分别是（S_r, S_g, S_b, S_a）和（D_r, D_g, D_b, D_a），并且下标s或d表示源颜色和目标颜色的RGBA值。最终的RGBA混合颜色是通过下面的公式

得出的：

$$(R_sS_r + R_dD_r, G_sS_g + G_dD_g, B_sS_b + B_dD_b, A_sS_a + A_dD_a)$$

这个四元组的每个成分最终都要通过截取限定在[0，1]的范围之内（除非颜色截取不可用。参见4.4.1节中的“颜色截取”）。

源片断和目标片断的默认组合方式是把它们的值加在一起（$C_sS + C_dD$）。请参见第6.1.3节，了解如何选择其他数学操作来组合片断。

可以使用两种不同的方法来选择源混合因子和目标混合因子。可以调用glBlendFunc()函数，并选择2个混合因子：第一个混合因子表示源RGBA，第二个混合因子表示目标RGBA。

也可以使用glBlendFuncSeparate()函数，并选择4个混合因子。这些混合因子允许我们为RGB选择一种混合操作，为alpha值选择另一种混合操作。

void **glBlendFunc**(GLenum *srcfactor*, GLenum *destfactor*);

控制被处理片断（源片断）的颜色值如何与已经存储在帧缓冲区中的像素（目标像素）的颜色值进行组合。表6-1解释了这个函数可以使用的参数值。参数srcfactor表示如何计算源混合因子，destfactor表示如何计算目标混合因子。

混合因子的值位于范围[0，1]之间。在源颜色和目标颜色进行混合之后，它们的值将进行截取，以限制在[0，1]的范围之内。

表6-1 源混合因子和目标混合因子

常 量	RGB混合因子	alpha混合因子
GL_ZERO	(0, 0, 0)	0
GL_ONE	(1, 1, 1)	1
GL_SRC_COLOR	(R_s, G_s, B_s)	A_s
GL_ONE_MINUS_SRC_COLOR	$(1, 1, 1) - (R_s, G_s, B_s)$	$1 - A_s$
GL_DST_COLOR	(R_d, G_d, B_d)	A_d
GL_ONE_MINUS_DST_COLOR	$(1, 1, 1) - (R_d, G_d, B_d)$	$1 - A_d$
GL_SRC_ALPHA	(A_s, A_s, A_s)	A_s
GL_ONE_MINUS_SRC_ALPHA	$(1, 1, 1) - (A_s, A_s, A_s)$	$1 - A_s$
GL_DST_ALPHA	(A_d, A_d, A_d)	A_d
GL_ONE_MINUS_DST_ALPHA	$(1, 1, 1) - (A_d, A_d, A_d)$	$1 - A_d$
GL_CONSTANT_COLOR	(R_c, G_c, B_c)	A_c
GL_ ONE_MINUS_CONSTANT_COLOR	$(1, 1, 1) - (R_c, G_c, B_c)$	$1 - A_c$
GL_ CONSTANT_ALPHA	(A_c, A_c, A_c)	A_c
GL_ ONE_MINUS_CONSTANT_ALPHA	$(1, 1, 1) - (A_c, A_c, A_c)$	$1 - A_c$
GL_SRC_ALPHA_SATURATE	$(f, f, f); f = \min(A_s, 1 - A_d)$	1

void **glBlendFuncSeparate**(GLenum *srcRGB*, GLenum *destRGB*,
GLenum *srcAlpha*, GLenum *destAlpha*);

glBlendFuncSeparate()函数的作用类似于glBlendFunc()函数，它也用于控制源颜色值（片断）与目标颜色值（帧缓冲区）的组合。**glBlendFuncSeparate()**所接受的参数与glBlendFunc()函数相同（见表6-1）。参数srcRGB表示颜色值的源混合因子，参数destRGB表示颜色值的目标混合因子。参数srcALpha表示alpha值的源混合因子，参数desTALpha表示alpha值的目标混合因子。

混合因子的值位于范围[0，1]之间。在源颜色和目标颜色进行混合之后，它们的值将进行截取，以限制在[0，1]的范围之内。

注意：在表6-1中，源、目标和常量颜色的RGBA值分别用下标s、d和c表示。RGBA四元组的减法表示逐成分地相减。GL_SRC_ALPHA_SATURATE只能作为源混合因子使用。

如果使用了其中一个GL*CONSTANT*混合函数，就需要使用glBlendColor()指定一种常量颜色。

void **glBlendColor**(GLclampf *red*, GLclampf *green*, GLclampf *blue*,
GLclampf *alpha*);

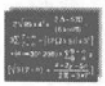
使用当前的red、green、blue和alpha值，作为混合操作的常量颜色（R_c, G_c, B_c, A_c）。

6.1.2 启用混合

不管如何指定混合函数，总是需要启用混合功能，以便使它生效：

```
glEnable(GL_BLEND);
```

以GL_BLEND为参数使用glDisable()函数可以禁用混合功能。注意，使用常量GL_ONE（源）和GL_ZERO（目标）的结果相当于禁用混合功能。它们正是默认值。



高级话题

OpenGL 2.0引入了同时渲染到多个缓冲区的功能（参见10.1.3节）。在OpenGL 3.0之前的版本中，所有的缓冲区同时打开或关闭混合（使用glEnable()和glDisable()）。然而，在OpenGL 3.0中，可以基于每个缓冲区来管理混合设置，只要使用glEnablei()和glDisablei()就可以了（参见本书10.2.7小节）。

6.1.3 使用混合方程式组合像素

在标准的混合中，帧缓冲区中的颜色与新片断的颜色进行组合（使用加法），产生新的帧缓冲区颜色。glBlendEquation()或glBlendEquationSeparate()函数都可以选择其他数学操作来计算颜色片断和帧缓冲区像素的差、最小值和最大值。

void **glBlendEquation**(GLenum *mode*);

指定帧缓冲区和源颜色如何进行混合。mode可以使用的值有GL_FUNC_ADD（默认）、GL_FUNC_SUBTRACT、GL_FUNC_REVERSE_SUBTRACT、GL_MIN、GL_MAX或GL_LOGIC_OP。表6-2描述了这些模式。

表6-2 混合方程式的数学操作

混合模式参数	数学操作	混合模式参数	数学操作
GL_FUNC_ADD	$C_sS + C_dD$	GL_MIN	$\min(C_sS, C_dD)$
GL_FUNC_SUBTRACT	$C_sS - C_dD$	GL_MAX	$\max(C_sS, C_dD)$
GL_FUNC_REVERSE_SUBTRACT	$C_dD - C_sS$	GL_LOGIC_OP	C_S *op* C_D

void **glBlendEquationSeparate**(GLenum *modeRGB*,
GLenum *modeAlpha*);

指定帧缓冲区和源颜色如何进行混合，但是它允许RGB和alpha颜色成分使用不同的混合模式。modeRGB和modeALpha参数允许接受的值和**glBlendEquation()**函数相同。

在表6-2中，C_s和C_d表示源颜色和目标颜色。表中的S和D参数表示glBlendFunc()或glBlendFuncSeparate()函数所指定的源混合因子和目标混合因子。对于GL_LOGIC_OP，逻辑操作符是通过调用glLogicOp()函数指定的。请参阅第10章的表10-4，了解OpenGL所支持的逻辑操作。

在示例程序6-1中，我们演示了不同的混合方程式模式。'a'、's'、'r'、'm'和'x'键用于选择可以使用的混合模式。我们用一个蓝色方块表示源颜色，用黄色背景表示目标颜色。每种颜色的混合因子使用glBlendFunc()函数设置为GL_ONE。

示例程序6-1 演示混合方程式模式：blendeqn.c

```
/*The following keys change the selected blend equation mode
 *
 *     'a ' -> GL_FUNC_ADD
 *     's ' -> GL_FUNC_SUBTRACT
 *     'r ' -> GL_FUNC_REVERSE_SUBTRACT
 *     'm ' -> GL_MIN
 *     'x ' -> GL_MAX
 */

void init(void)
{
   glClearColor(1.0,1.0,0.0,0.0);

   glBlendFunc(GL_ONE,GL_ONE);
   glEnable(GL_BLEND);
}
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(0.0,0.0,1.0);
   glRectf(-0.5,-0.5,0.5,0.5);
   glFlush();
}
void keyboard(unsigned char key,int x,int y)
{
   switch (key){
      case 'a ':case 'A ':
       /*Colors are added as:(1,1,0)+(0,0,1)=(1,1,1)which
        *will produce a white square on a yellow background.*/
         glBlendEquation(GL_FUNC_ADD);
         break;

      case 's ':case 'S ':
       /*Colors are subtracted as:(0,0,1)-(1,1,0)=(-1,-1,1)
        *which is clamped to (0,0,1),producing a blue square
        *on a yellow background.*/
         glBlendEquation(GL_FUNC_SUBTRACT);
         break;

      case 'r ':case 'R ':
       /*Colors are subtracted as:(1,1,0)-(0,0,1)=(1,1,-1)
        *which is clamped to (1,1,0).This produces yellow for
```

```
        *both the square and the background.*/
          glBlendEquation(GL_FUNC_REVERSE_SUBTRACT);
          break;

        case 'm ':case 'M ':
         /*The minimum of each component is computed,as
          *[min(1,0),min(1,0),min(0,1)] which equates to ((0,0,0).
          *This will produce a black square on the yellow
          *background.*/
           glBlendEquation(GL_MIN);
           break;

        case 'x ':case 'X ':
         /*The minimum of each component is computed,as
          *[max(1,0),max(1,0),max(0,1)] which equates to
          *(1,1,1).This will produce a white square on the
          *yellow background.*/
           glBlendEquation(GL_MAX);
           break;
        case 27:
           exit(0);
    }break;
    glutPostRedisplay();
}
```

6.1.4 混合的样例用法

对于源混合因子和目标混合因子，并不是所有的组合都是合理的。绝大多数应用程序只使用其中一小部分组合方式。下面各个段落描述了源混合因子和目标混合因子的各种组合方式的典型用途。有些例子只使用新片断的alpha值，因此它们在帧缓冲区不存储alpha值的情况下依然可行。注意，常常可以使用多种方式来实现相同的效果。

- 如果想绘制一幅这样的图：它的一半来自一幅图像，另一半来自另一幅图像，它们以相同的数量混合在一起。为此，可以把源混合因子设置为GL_ONE，把目标混合因子设置为GL_ZERO，并首先绘制第一幅图像。然后，把源混合因子设置为GL_SRC_ALPHA，把目标混合因子设置为GL_ONE_MINUS_SRC_ALPHA，并用0.5的alpha值绘制第二幅图像。混合因子的这种用法可以说代表了最常用的混合操作。如果需要把第一幅图像的75%与第二幅图像的25%进行混合，我们仍然按上面的方法绘制第一幅图像，然后用0.25的alpha值绘制第二幅图像。
- 为了以相同的比例混合3幅不同的图像，可以把目标混合因子设置为GL_ONE，把源混合因子设置为GL_SRC_ALPHA。用0.3333 333的alpha值绘制每幅图像。使用这种技巧，每幅图像的亮度是它原来的1/3。在图像并不重叠的地方，还是比较容易注意到这个效果的。
- 假设我们正在编写一个绘图程序，并想拥有一个能够逐渐增加颜色的画刷。用这个画刷每刷一道可以在图像的当前颜色上增加一些颜色（例如，10%的当前画刷颜色与90%的图像颜色进行混合）。为此，可以使用alpha值为10%的画刷来绘制图像，并使用GL_SRC_ALPHA（源混合因子）和GL_ONE_MINUS_SRC_ALPHA（目标混合因子）。注意，为了实现抗锯齿功能的画刷，可以使画刷在中央的颜色浓一些，在边缘的颜色淡一些（参见第6.2节）。类似地，也可以通过把清除颜色设置为背景色来实现橡皮擦的功能。

- 使用源或目标颜色的混合函数（用于源因子的GL_DST_COLOR或GL_ONE_MINUS_DST_COLOR，以及用于目标因子的GL_SRC_COLOR或GL_ONE_MINUS_SRC_COLOR）允许我们有效地对各个颜色成分进行单独的调整。这个操作相当于应用一种简单的过滤器，例如把红色成分与80%相乘，把绿色成分与40%相乘，把蓝色成分与72%相乘，以模拟透过一种能够过滤20%的红色、60%的绿色以及28%的蓝色的照相底片观察场景的效果。
- 假设我们想绘制一幅由3个半透明表面组成的图片，这几个表面相互遮挡，并且都位于一个实心背景上。假设最远的那个表面能够传播它背后80%的颜色，第二个表面能够传播它背后40%的颜色，最近的那个表面能够传播它背后90%的颜色。为了合成这幅图片，首先用默认的源因子和目标因子绘制背景色，然后把混合因子改为GL_SRC_ALPHA（源）和GL_ONE_MINUS_SRC_ALPHA（目标）。接着，用0.2的alpha值绘制最远的那个表面，然后用0.6的alpha值绘制中间那个表面，最后用0.1的alpha值绘制最近的那个表面。
- 如果系统具有alpha平面，就可以在某个时刻渲染物体（包括它们的alpha值），将它们读回，然后对完全渲染的物体进行有趣的褪光和合成操作（关于这个技巧的例子，读者可以参阅Tom Duff所著的《Compositing 3D Rendered Images》，该文发表于SIGGRAPH 1985《Proceedings》，第41～44页）。注意，用于图片合成的物体可以来自任何途径。它们可以是由OpenGL函数渲染的，也可以是由诸如光线追踪或放射扫描等技巧（这些技巧是由其他图形函数库实现的）渲染的，或者是通过扫描现有的图像获得的。
- 可以通过向图像中的单独片断分配不同的alpha值，实现非矩形光栅图像的效果。在大多数情况下，对于透明的片断，向它们分配0.0的alpha值；对于不透明的片断，向它们分配1.0的alpha值。例如，可以绘制一个树形的多边形，并应用一幅植物纹理图像。如果把矩形的纹理图像中观察者可以“看穿”的那部分像素的alpha值设置为0，它们就不会被观察者看到。这种方法有时又称“billboarding”，它比通过三维多边形来创建树形状的方法要快得多。图6-1显示了这个技巧的一个例子。在这张图中，树是一个矩形的多边形，可以沿树干的中心进行旋转（如它的轮廓所示）。因此，它始终面向观察者。关于混合纹理的更多信息，请参阅第9.5节。

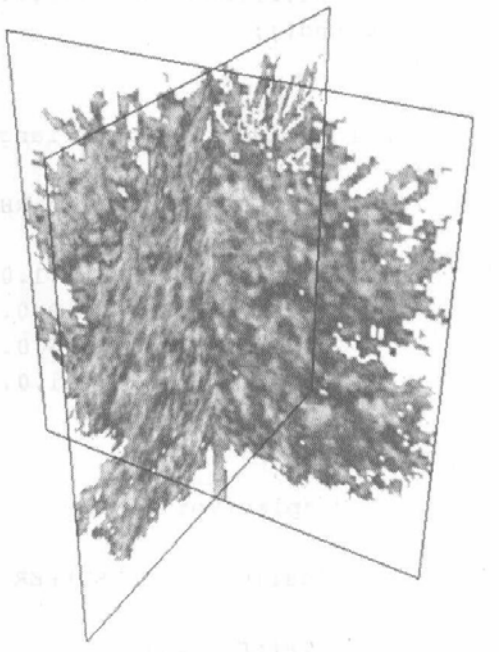


图6-1 创建一幅非矩形的光栅图像

- 混合也可以用于抗锯齿，减少光栅屏幕上所绘制的几何图元的锯齿状边缘（详见第6.2节）。

6.1.5 一个混合的例子

示例程序6-2绘制了两个重叠的着色三角形，每个三角形的alpha值均为0.75。这个程序启用了混合功能，并把源和目标混合因子分别设置为GL_SRC_ALPHA和GL_ONE_ MINUS_SRC_ALPHA。

当程序启动时，窗口的左边将绘制一个黄色的三角形，然后窗口的右边将出现一个青色的三角形。因此，在窗口的中间两个三角形重叠的地方，第二个三角形的青色就与第一个三角形的黄色进行混合。我们可以在窗口中输入“t”来改变两个三角形的绘图顺序。

示例程序6-2 混合例子：alpha.c

```
static int leftFirst =GL_TRUE;

/*Initialize alpha blending function.*/
static void init(void)
{
   glEnable(GL_BLEND);
   glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
   glShadeModel(GL_FLAT);
   glClearColor(0.0,0.0,0.0,0.0);
}

static void drawLeftTriangle(void)
{
/* draw yellow triangle on LHS of screen */
   glBegin(GL_TRIANGLES);
      glColor4f(1.0,1.0,0.0,0.75);
      glVertex3f(0.1,0.9,0.0);
      glVertex3f(0.1,0.1,0.0);
      glVertex3f(0.7,0.5,0.0);
   glEnd();
}

static void drawRightTriangle(void)
{
/* draw cyan triangle on RHS of screen */
   glBegin(GL_TRIANGLES);
      glColor4f(0.0,1.0,1.0,0.75);
      glVertex3f(0.9,0.9,0.0);
      glVertex3f(0.3,0.5,0.0);
      glVertex3f(0.9,0.1,0.0);
   glEnd();
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);

   if (leftFirst){
      drawLeftTriangle();
      drawRightTriangle();
   }
   else {
      drawRightTriangle();
      drawLeftTriangle();
   }
   glFlush();
}

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

```
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <=h)
       gluOrtho2D(0.0,1.0,0.0,1.0*(GLfloat)h/(GLfloat)w);
    else
       gluOrtho2D(0.0,1.0*(GLfloat)w/(GLfloat)h,0.0,1.0);
}

void keyboard(unsigned char key,int x,int y)
{
    switch (key){
       case 't ':
       case 'T ':
          leftFirst =!leftFirst;
          glutPostRedisplay();
          break;
       case 27:/*Escape key */
          exit(0);
          break;
       default:
          break;
   }
}

int main(int argc,char**argv)
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB);
    glutInitWindowSize(200,200);
    glutCreateWindow(argv [0]);
    init();
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

两个三角形的绘制顺序会影响重叠区域的颜色。如果首先绘制左边的三角形，青色的片断（源）与帧缓冲区中已经存在的黄色片断（目标）进行混合。如果首先绘制右边的三角形，就是黄色（源）与青色（目标）进行混合。由于alpha值都是0.75，因此源颜色的混合因子是0.75，而目标颜色的混合因子则是1.0−0.75 = 0.25。换句话说，源片断看上去是半透明的，但是它们对最终颜色的影响大于目标片断对最终颜色的影响。

6.1.6 使用深度缓冲区进行三维混合

正如我们在前面那个例子中所看到的那样，多边形的绘图顺序将极大地影响最终的混合结果。在绘制三维的半透明物体时，根据是从前向后还是从后向前绘制各个多边形，最终的结果可能大相径庭。在决定正确的绘图顺序时，还需要考虑深度缓冲区的效果（关于深度缓冲区的介绍，可以参阅第5.1节和第10.2.4节）。深度缓冲区追踪观察点与占据屏幕窗口中一个特定像素的物体之间的距离。当另

外一个物体也占据了这个像素时，它的深度值就会与存储于深度缓冲区中原先物体的深度值进行比较，只有当这个物体的距离比原先的物体更靠近观察点时，它的颜色值才会应用于这个像素。按照这种方法，被遮挡（隐藏）的表面部分就不需要绘制，因此不会用于混合操作之中。

如果想在同一个场景中同时渲染透明和半透明的物体，就需要使用深度缓冲区，对那些位于不透明物体后面的所有物体执行隐藏表面消除。如果一个不透明的物体遮挡住了一个半透明物体或另一个不透明物体，就需要利用深度缓冲区消除这些距离更远的物体。但是，如果半透明物体更靠近观察点，就需要把它与位于它后面的不透明物体进行混合。一般情况下，如果场景中的物体都是静止的，我们能够推断出多边形的绘图顺序。但是，如果物体或观察者是运动的，这个问题就会变得极为困难。

这个问题的解决方案是启用深度缓冲区，但是在绘制半透明物体时让深度缓冲区处于只读状态。首先，绘制所有的不透明物体，按正常的方式对深度缓冲区进行操作。然后，把深度缓冲区设置为只读，保持它里面的深度值。绘制半透明物体时，它们的深度值仍然与不透明物体的深度值进行比较。因此，如果它们位于不透明物体的后面，它们就不会被绘制。但是，如果它们更靠近观察点，它们并不消除那些不透明的物体，因为此时深度缓冲区的值是无法改变的。反之，它们与这些不透明的物体进行混合。为了控制深度缓冲区是否可写，可以使用glDepthMask()函数。如果向它传递GL_FALSE参数，深度缓冲区就设置为只读。如果向它传递GL_TRUE，深度缓冲区就可以进行正常的读写。

示例程序6-3显示了如何使用这种方法绘制不透明和半透明的三维物体。在这个程序中，键入“a”将会触发一个动画序列，场景中一个半透明的物体沿一个不透明的物体移动。按下“r”键可以把这个物体重置为原先的位置。为了使半透明的物体在重叠时能够达到最佳的效果，这个程序从后向前绘制所有的物体。

示例程序6-3 三维混合：alpha3D.c

```
#define MAXZ 8.0
#define MINZ -8.0
#define ZINC 0.4
static float solidZ =MAXZ;
static float transparentZ =MINZ;
static GLuint sphereList,cubeList;

static void init(void)
{
   GLfloat mat_specular [] ={1.0,1.0,1.0,0.15 };
   GLfloat mat_shininess [] ={100.0 };
   GLfloat position [] ={0.5,0.5,1.0,0.0 };

   glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
   glMaterialfv(GL_FRONT,GL_SHININESS,mat_shininess);
   glLightfv(GL_LIGHT0,GL_POSITION,position);

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);

   sphereList =glGenLists(1);
   glNewList(sphereList,GL_COMPILE);
      glutSolidSphere(0.4,16,16);
   glEndList();
```

```
    cubeList =glGenLists(1);
    glNewList(cubeList,GL_COMPILE);
       glutSolidCube(0.6);
    glEndList();
}
void display(void)
{
    GLfloat mat_solid [] ={0.75,0.75,0.0,1.0 };
    GLfloat mat_zero [] ={0.0,0.0,0.0,1.0 };
    GLfloat mat_transparent [] ={0.0,0.8,0.8,0.6 };
    GLfloat mat_emission [] ={0.0,0.3,0.3,0.6 };

    glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);

    glPushMatrix();
       glTranslatef(-0.15,-0.15,solidZ);
       glMaterialfv(GL_FRONT,GL_EMISSION,mat_zero);
       glMaterialfv(GL_FRONT,GL_DIFFUSE,mat_solid);
       glCallList(sphereList);
    glPopMatrix();

    glPushMatrix();
       glTranslatef(0.15,0.15,transparentZ);
       glRotatef(15.0,1.0,1.0,0.0);
       glRotatef(30.0,0.0,1.0,0.0);
       glMaterialfv(GL_FRONT,GL_EMISSION,mat_emission);
       glMaterialfv(GL_FRONT,GL_DIFFUSE,mat_transparent);
       glEnable(GL_BLEND);
       glDepthMask(GL_FALSE);
       glBlendFunc(GL_SRC_ALPHA,GL_ONE);
       glCallList(cubeList);
       glDepthMask(GL_TRUE);
       glDisable(GL_BLEND);
    glPopMatrix();

    glutSwapBuffers();
}

void reshape(int w,int h)
{
    glViewport(0,0,(GLint)w,(GLint)h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <=h)
       glOrtho(-1.5,1.5,-1.5*(GLfloat)h/(GLfloat)w,
              1.5*(GLfloat)h/(GLfloat)w,-10.0,10.0);
    else
       glOrtho(-1.5*(GLfloat)w/(GLfloat)h,
              1.5*(GLfloat)w/(GLfloat)h,-1.5,1.5,-10.0,10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

```
void animate(void)
{
   if (solidZ <=MINZ ||transparentZ >=MAXZ)
      glutIdleFunc(NULL);
   else {
      solidZ -=ZINC;
      transparentZ +=ZINC;
      glutPostRedisplay();
   }

}
void keyboard(unsigned char key,int x,int y)
{
   switch (key){
      case 'a ':
      case 'A ':
         solidZ =MAXZ;
         transparentZ =MINZ;
         glutIdleFunc(animate);
         break;
      case 'r ':
      case 'R ':
         solidZ =MAXZ;
         transparentZ =MINZ;
         glutPostRedisplay();
         break;
      case 27:
         exit(0);
   }     break;
}

int main(int argc,char**argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_DOUBLE |GLUT_RGB |GLUT_DEPTH);
   glutInitWindowSize(500,500);
   glutCreateWindow(argv [0]);
   init();
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;
}
```

6.2 抗锯齿

读者可能已经注意到，在有些OpenGL图像中，直线（尤其是接近水平或垂直的直线）会呈现锯齿状边缘。这种“锯齿”现象是由于理想的直线是由一系列的像素模拟的，而这些像素又必须位于它们各自的像素网格中。本节将讨论减少这种“锯齿”效果的抗锯齿技巧。图6-2显示了两条相交的直线，左边显示了锯齿状直线，右边则显示了经过抗锯齿处理后的直线。为了更好地观察它们的效果，

这张图进行了放大。

如图6-3所示，宽度为1像素的斜线所覆盖的像素比其他情况下的直线所覆盖的像素要多。事实上，在执行抗锯齿操作时，OpenGL根据片断覆盖像素方块的面积来计算覆盖比例。图6-3显示了直线的覆盖比例。在RGBA模式下，OpenGL将片断的覆盖比例与alpha值相乘，并将乘积作为alpha值把片断与帧缓冲区中的对应像素进行混合。在颜色索引模式下，OpenGL根据片断的覆盖比例来设置颜色索引的最后4位（如果没有覆盖，设置为0000；如果完全覆盖，设置为1111）。为了使用这种覆盖信息，必须加载颜色映射表，并正确地应用颜色映射表，以利用覆盖信息。



图6-2 锯齿和抗锯齿直线

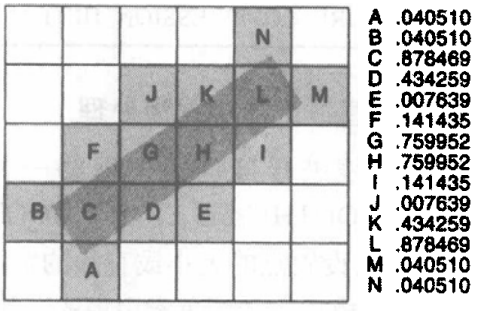


图6-3 确定覆盖值

计算覆盖值的细节非常复杂，一般情况下很难进行指定，并且可能因为具体使用的OpenGL实现而存在稍许的差异。可以使用glHint()函数对图像质量和渲染速度之间的权衡关系施加控制，但并不是所有的OpenGL实现都会遵从这种提示。

void **glHint**(GLenum target, GLenum hint);

控制OpenGL的一些行为。target参数表示需要控制什么行为。表6-3列出了它可以使用的值。hint参数可以是GL_FASTEST，表示应该选择效率最高的选项；它也可以是GL_NICEST，表示应该选择质量最高的选项。它还可以是GL_DONT_CARE，表示没什么偏向。对提示的解释因不同的OpenGL实现而异，某种OpenGL实现也可能完全忽略这种提示。关于相关主题的更多信息，请参阅本节关于采样的细节，以及第6.3节中有关雾的细节。

如果target参数是GL_PERSPECTIVE_CORRECTION_HINT，它表示如何在图元中对颜色值和纹理坐标进行插值：在屏幕空间中进行线性插值（相对较为简单）或者采用透视修正（perspectivecorrect）的方式（计算量更大一些）。对于颜色，系统通常可以使用线性插值的方式，因为虽然从技术上说这种方法所产生的结果并不是非常准确，但是它的视觉效果还是可以接受的。但是对于纹理，系统在绝大多数情况下必须使用透视修正的方式，它所产生的视觉效果才是可以接受的。因此，OpenGL实现可以使用这个参数来选择插值方式。请参阅第3章有关透视投影的讨论和第4章有关颜色的讨论，以及第9章有关纹理贴图的讨论。

兼容性扩展
GL_PERSPECTIVE_
CORRECTION_HINT
GL_FOG_HINT
GL_POINT_SMOOTH_
HINT
GL_GENERATE_MIPMAP_
HINT

表6-3 target参数的取值及其含义

参 数	含 义
GL_POINT_SMOOTH_HINT、GL_LINE_SMOOTH_HINT、GL_POLYGON_SMOOTH_HINT	在抗锯齿操作时点、直线和多边形的采样质量
GL_FOG_HINT	雾计算是根据像素（GL_NICEST）还是顶点（GL_FASTEST）进行
GL_PERSPECTIVE_CORRECTION_HINT	颜色和纹理坐标插值的质量
GL_GENERATE_MIPMAP_HINT	自动Mipmap层生成的质量和性能
GL_TEXTURE_COMPRESSION_HINT	压缩纹理图像的质量和性能

6.2.1 对点和直线进行抗锯齿处理

对点和直线进行抗锯齿处理的一种方法之一是使用glEnable()函数（参数根据需要为GL_POINT_SMOOTH或GL_LINE_SMOOTH）启用抗锯齿功能。还可以使用glHint()函数提供质量提示（记住，可以设置点的大小或直线的宽度，也可以对直线进行点画。参见第2.4.2节）。然后，根据使用的是RGBA模式还是颜色索引模式，执行下面描述的过程。

对点或直线进行抗锯齿处理的另一种方法是使用多重采样，如第6.2.2节所述。

RGBA模式下的抗锯齿

在RGBA模式下，需要启用混合功能。最常用的混合因子是GL_SRC_ALPHA（源）和GL_ONE_MINUS_SRC_ALPHA（目标）。另外，也可以使用GL_ONE来表示目标混合因子，使直线在相交的地方显得更亮一些。接着，就可以绘制需要进行抗锯齿处理的点或直线了。如果使用的alpha值较大，抗锯齿效果就更加明显。记住，由于所执行的是混合操作，因此可能需要考虑第6.1.6节描述的渲染顺序。但是，在大多数情况下，可以忽略渲染顺序，它并不会产生明显的不良效果。示例程序6-4对抗锯齿所需要的一些模式进行初始化，并绘制两条相交的斜线。当我们运行这个程序时，可以按“k”键对这两条直线进行旋转，以观察不同的斜率下直线的抗锯齿效果。注意，这个例子并没有启用深度缓冲区。

示例程序6-4 抗锯齿直线：aargb.c

```
static float rotAngle =0.;

/* Initialize antialiasing for RGBA mode,including alpha
 * blending,hint,and line width.Print out implementation-
 * specific info on line width granularity and width.
 */
void init(void)
{
   GLfloat values [2];
   glGetFloatv(GL_LINE_WIDTH_GRANULARITY,values);
   printf("GL_LINE_WIDTH_GRANULARITY value is %3.1f \n ",
      values [0]);
   glGetFloatv(GL_LINE_WIDTH_RANGE,values);
   printf("GL_LINE_WIDTH_RANGE values are %3.1f %3.1f \n ",
      values [0],values [1]);

   glEnable(GL_LINE_SMOOTH);
```

```
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
    glHint(GL_LINE_SMOOTH_HINT,GL_DONT_CARE);
    glLineWidth(1.5);

    glClearColor(0.0,0.0,0.0,0.0);
}

/*Draw 2 diagonal lines to form an X */
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);

    glColor3f(0.0,1.0,0.0);
    glPushMatrix();
    glRotatef(-rotAngle,0.0,0.0,0.1);
    glBegin(GL_LINES);
       glVertex2f(-0.5,0.5);
       glVertex2f(0.5,-0.5);
    glEnd();
    glPopMatrix();

    glColor3f(0.0,0.0,1.0);
    glPushMatrix();
    glRotatef(rotAngle,0.0,0.0,0.1);
    glBegin(GL_LINES);
       glVertex2f(0.5,0.5);
       glVertex2f(-0.5,-0.5);
    glEnd();
    glPopMatrix();

    glFlush();
}

void reshape(int w,int h)
{
    glViewport(0,0,(GLint)w,(GLint)h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <=h)
       gluOrtho2D(-1.0,1.0,
          -1.0*(GLfloat)h/(GLfloat)w,1.0*(GLfloat)h/(GLfloat)w);
    else
       gluOrtho2D(-1.0*(GLfloat)w/(GLfloat)h,
          1.0*(GLfloat)w/(GLfloat)h,-1.0,1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void keyboard(unsigned char key,int x,int y)
{
    switch (key){
       case 'r ':
```

```
        case 'R ':
            rotAngle +=20.;
            if (rotAngle >=360.)rotAngle =0.;
            glutPostRedisplay();
            break;
        case 27:/*Escape Key */
            exit(0);
            break;
        default:
            break;
    }
}

int main(int argc,char**argv)
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB);
    glutInitWindowSize(200,200);
    glutCreateWindow(argv [0]);
    init();
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

颜色索引模式下的抗锯齿

在颜色索引模式下进行抗锯齿处理的技巧就是加载和使用颜色映射表。由于颜色索引的最后4位表示覆盖值，因此需要在颜色映射表中加载16个连续的颜色索引，颜色的变化是从背影过渡到物体的颜色（第1个索引值必须是16的倍数）。然后，把颜色缓冲区清除为颜色映射表中的16种颜色，并使用这个颜色映射表中的颜色来绘制点或直线。示例程序6-5说明了在颜色索引模式下如何创建颜色表来绘制抗锯齿的直线。这个例子创建了两个颜色映射表：一个包含了各种绿色色调，另一个包含了各种蓝色色调。

示例程序6-5 颜色索引模式下的抗锯齿：aaindex.c

```
#define RAMPSIZE 16
#define RAMP1START 32
#define RAMP2START 48

static float rotAngle =0.;

/*  Initialize antialiasing for color-index mode,
 *  including loading a green color ramp starting
 *  at RAMP1START and a blue color ramp starting
 *  at RAMP2START.The ramps must be a multiple of 16.
 */
void init(void)
{
    int i;
```

```
    for (i =0;i <RAMPSIZE;i++){
        GLfloat shade;
        shade =(GLfloat)i/(GLfloat)RAMPSIZE;
        glutSetColor(RAMP1START+(GLint)i,0.,shade,0.);
        glutSetColor(RAMP2START+(GLint)i,0.,0.,shade);
    }
    glEnable(GL_LINE_SMOOTH);
    glHint(GL_LINE_SMOOTH_HINT,GL_DONT_CARE);
    glLineWidth(1.5);

    glClearIndex((GLfloat)RAMP1START);
}
/*Draw 2 diagonal lines to form an X */
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);

    glIndexi(RAMP1START);
    glPushMatrix();
    glRotatef(-rotAngle,0.0,0.0,0.1);
    glBegin(GL_LINES);
        glVertex2f(-0.5,0.5);
        glVertex2f(0.5,-0.5);
    glEnd();
    glPopMatrix();

    glIndexi(RAMP2START);
    glPushMatrix();
    glRotatef(rotAngle,0.0,0.0,0.1);
    glBegin(GL_LINES);
        glVertex2f(0.5,0.5);
        glVertex2f(-0.5,-0.5);
    glEnd();
    glPopMatrix();

    glFlush();
}
void reshape(int w,int h)
{
    glViewport(0,0,(GLsizei)w,(GLsizei)h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <=h)
        gluOrtho2D(-1.0,1.0,
            -1.0*(GLfloat)h/(GLfloat)w,1.0*(GLfloat)h/(GLfloat)w);
    else
        gluOrtho2D(-1.0*(GLfloat)w/(GLfloat)h,
            1.0*(GLfloat)w/(GLfloat)h,-1.0,1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

```
void keyboard(unsigned char key,int x,int y)
{
   switch (key){
      case 'r ':
      case 'R ':
         rotAngle +=20.;
         if (rotAngle >=360.)rotAngle =0.;
         glutPostRedisplay();
         break;
      case 27:/*Escape Key */
         exit(0);
         break;
      default:
         break;
   }
}

int main(int argc,char**argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_SINGLE |GLUT_INDEX);
   glutInitWindowSize(200,200);
   glutCreateWindow(argv [0]);
   init();
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;
}
```

由于颜色映射表的变化范围是从背景色过渡到物体的颜色，因此只有在背景色上绘制抗锯齿的直线时，其结果才是正确的。当蓝线被绘制时，它会擦除两条直线相交处的绿色像素。为了修正这个问题，我们可能需要使用一个颜色范围从绿色（帧缓冲区中的直线颜色）到蓝色（被绘制的直线颜色）的颜色表对直线相交区域进行重绘。但是，这需要额外的计算，通常并不值得这样做，因为相交区域是非常小的。注意，在RGBA模式下，这并不是一个问题，因为被绘制物体的颜色将与帧缓冲区中已经存在的颜色进行混合。

当我们在颜色索引模式下绘制抗锯齿的点和直线时，可能还需要启用深度测试。在上面这个例子里，深度测试是禁用的，因为这两条直线位于同一个z平面上。但是，如果我们所绘制的是三维场景，就应该启用深度测试，使最终像素颜色对应于“最近”的物体。

第6.1.6节描述的技巧也可以用于把抗锯齿的点和直线与锯齿的、进行了深度缓冲区测试的多边形进行混合。为此，我们首先绘制多边形，然后把深度缓冲区设置为只读，并绘制点和直线。点和直线相交的效果很好，但是它们可能会被更近的多边形遮挡。



尝试一下

取一个以前的程序，例如第3.8节描述的机器人手臂或太阳系的例子，在绘制线框物体时使用抗锯齿功能。读者可以在RGBA模式和颜色索引模式下都尝试一下。同样，也可以试试不同宽度的直线以及不同大小的点，以观察它们的效果。

6.2.2 使用多重采样对几何图元进行抗锯齿处理

多重采样技巧使用额外的颜色、深度和模板信息（样本）对OpenGL图元（点、直线、多边形、位图和图像）进行抗锯齿处理。每个片断不再只有1种颜色、1个深度值和1组纹理坐标，而是根据了像素样本的数量，具有多种颜色、多个深度值和多组纹理坐标。这些信息的计算并不是在每个像素的中心进行的（如果是单一采样，就采用这种方式），而是分布于几个采样位置。这种技巧并不是使用alpha值来表示一个图元覆盖某个像素的程度，而是根据一个多重采样缓冲区所保存的样本来计算抗锯齿覆盖信息。

多重采样特别适合对多边形的边缘进行抗锯齿处理，因为此时不需要进行排序（如果使用alpha值对多边形进行抗锯齿处理，半透明物体的绘图顺序将会影响最终的颜色）。多重采样能够解决传统上难以解决的一些难题，例如相交或相邻的多边形。

在应用程序中增加多重采样功能非常简单，只要使用下面几个步骤就行了：

1) 获取一个支持多重采样的窗口。如果使用GLUT，可以通过下面这种方法来实现：

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

2) 在打开窗口之后，需要验证一下多重采样功能是否可用。例如，GLUT所提供的窗口“几乎”就是我们需要的窗口。如果查询状态变量GL_SAMPLE_BUFFERS返回的值是1，并且查询GL_SAMPLES返回的值大于1，就可以使用多重采样功能（GL_SAMPLES返回了像素样本的数量。如果只有1个样本，多重采样就被禁用）。

```
GLint bufs, samples;
glGetIntegerv(GL_SAMPLE_BUFFERS, &bufs);
glGetIntegerv(GL_SAMPLES, &samples);
```

3) 调用下面这个函数，启用多重采样功能：

```
glEnable(GL_MULTISAMPLE);
```

示例程序6-6并排显示了两组图元，我们可以看到多重采样所实现的效果。init()函数检查多重采样的状态变量，并编译两个显示列表：一个列表包含了由不同宽度的直线和三角形（填充多边形）组成的风车，另一个列表则包含了一个棋盘背景。在display()函数中，这个风车分别在使用多重采样（左侧）和不使用多重采样（右侧）的条件下进行绘制。读者可以比较一下左右两侧的效果。对比度强的背景有时候能够凸现锯齿效果，有时候却可能掩盖锯齿效果。读者可以按下“b”键进行切换，分别在使用和不使用棋盘背景下绘制这个场景。

示例程序6-6 启用多重采样：multisamp.c

```
static int bgtoggle =1;
/*
 *  Print out state values related to multisampling.
 *  Create display list with "pinwheel"of lines and
 *  triangles.
 */
void init(void)
{
GLint buf,sbuf;
int i,j;

glClearColor(0.0,0.0,0.0,0.0);
```

```
glGetIntegerv(GL_SAMPLE_BUFFERS,&buf);
printf("number of sample buffers is %d \n",buf);
glGetIntegerv(GL_SAMPLES,&sbuf);
printf("number of samples is %d \n",sbuf);

glNewList(1,GL_COMPILE);
for (i =0;i <19;i++){
   glPushMatrix();
   glRotatef(360.0*(float)i/19.0,0.0,0.0,1.0);
   glColor3f (1.0,1.0,1.0);
   glLineWidth((i%3)+1.0);
   glBegin(GL_LINES);
      glVertex2f(0.25,0.05);
      glVertex2f(0.9,0.2);
   glEnd();
   glColor3f(0.0,1.0,1.0);
   glBegin(GL_TRIANGLES);
      glVertex2f(0.25,0.0);
      glVertex2f(0.9,0.0);
      glVertex2f(0.875,0.10);
   glEnd();
   glPopMatrix();
}

glEndList();

glNewList(2,GL_COMPILE);
glColor3f(1.0,0.5,0.0);
glBegin(GL_QUADS);
for (i =0;i <16;i++){
   for (j =0;j <16;j++){
      if (((i +j)%2)==0){
         glVertex2f(-2.0 +(i *0.25),-2.0 +(j *0.25));
         glVertex2f(-2.0 +(i *0.25),-1.75 +(j *0.25));
         glVertex2f(-1.75 +(i *0.25),-1.75 +(j *0.25));
         glVertex2f(-1.75 +(i *0.25),-2.0 +(j *0.25));
      }
   }
}
   glEnd();
   glEndList();
}
/*Draw two sets of primitives so that you can
 *compare the user of multisampling against its absence.
 *
 *This code enables antialiasing and draws one display list,
 *and then it disables and draws the other display list.
 */
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);

   if (bgtoggle)
```

```
        glCallList(2);

    glEnable(GL_MULTISAMPLE);
    glPushMatrix();
    glTranslatef(-1.0,0.0,0.0);
    glCallList(1);
    glPopMatrix();

    glDisable(GL_MULTISAMPLE);
    glPushMatrix();
    glTranslatef(1.0,0.0,0.0);
    glCallList(1);
    glPopMatrix();
    glutSwapBuffers();
}

void keyboard(unsigned char key,int x,int y)
{
    switch (key){
        case 'b ':
        case 'B ':
            bgtoggle =!bgtoggle;
            glutPostRedisplay();
            break;
        case 27:/*Escape Key */
            exit(0);
            break;
        default:
            break;
    }
}
```

在标准的OpenGL实现中，无法对多重采样进行微调。我们无法更改样本的数量，也无法指定（甚至查询）子像素采样的位置。

如果多重采样被启用，并且存在一个多重采样缓冲区，那么点、直线和多边形所产生的片断一般都进行了抗锯齿处理。例如，如果一个点很大，那么它就是圆的，并不是正方的，不管是否启用了GL_POINT_SMOOTH。GL_LINE_SMOOTH 和GL_POINT_SMOOTH的状态都是被忽略的。其他图元属性，例如点的大小和直线的宽度，也受到多重采样的支持。

alpha和多重采样覆盖

在默认情况下，多重采样计算的覆盖值是独立于alpha值的。但是，如果启用了下面这些特殊的模式，那么在计算片断的覆盖值时就会考虑alpha值因素。这些特殊模式如下：

- GL_SAMPLE_ALPHA_TO_COVERAGE：使用片断的alpha值来计算最终的覆盖值。
- GL_SAMPLE_ALPHA_TO_ONE：把片断的alpha值设置为1（即最大值），然后在计算覆盖值时使用这个值。
- GL_SAMPLE_COVERAGE：使用glSampleCoverage()函数所设置的值，这个值与经过计算产生的覆盖值进行组合（使用AND操作）。另外，也可以通过在glSampleCoverage()函数中使用invert标志来反转这个模式。

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

设置参数，用于在计算多重采样覆盖值时解释alpha值。value是一个临时覆盖值，在GL_SAMPLE_COVERAGE或GL_ALPHA_TO_COVERAGE模式被启用时使用。invert是一个布尔值，表示这个临时覆盖值与片断的覆盖值进行组合（AND操作）之前是否应该进行逐位的反转。

6.2.3 对多边形进行抗锯齿处理

对填充多边形的边缘进行抗锯齿处理类似于对点和直线进行抗锯齿处理。当不同的多边形具有重叠的边缘时，就需要对颜色值进行适当的混合。可以使用本节描述的方法，也可以使用累积缓冲区对整个场景进行抗锯齿处理。使用第10章描述的累积缓冲区，从用户的角度而言也许操作起来更简单一些。但是，这种方法需要大量的计算，因此速度较慢。不过，本节描述的方法虽然速度较快，但是使用起来多少有些笨拙。

注意：如果在绘制多边形时使用了点或轮廓线的形式（也就是在glPolygonMode()函数中使用了GL_POINT或GL_LINE参数），并且像前面所述的那样启用了抗锯齿功能，那么它所执行的就是点或直线抗锯齿。本节剩余部分所描述的是使用GL_FILL模式的多边形抗锯齿。

从理论上说，无论是RGBA模式还是颜色索引模式，都可以对多边形进行抗锯齿处理。但是，相交的物体对多边形抗锯齿处理的影响要远大于对点和直线抗锯齿处理的影响。因此，渲染的顺序和混合的准确性就显得更加关键。事实上，它们之所以显得关键的原因是当我们对多个多边形进行抗锯齿处理时，需要从前向后对多边形进行排序，并把源混合因子设置为GL_SRC_ALPHA_SATURATE，把目标混合因子设置为GL_ONE。因此，在颜色索引模式下，对多边形进行抗锯齿处理一般是不可行的。

为了在RGBA模式下对多边形进行抗锯齿处理，可以使用alpha值来表示多边形边缘的覆盖值。我们需要向glEable()函数传递GL_POLYGON_SMOOTH参数来启用多边形抗锯齿功能。这将导致多边形边缘的像素根据它的覆盖值被分配以分数形式的alpha值，就像按照直线的形式进行了抗锯齿处理一样。另外，如果愿意，可以为GL_POLYGON_SMOOTH_HINT提供一个值。

现在，需要对重叠的边缘进行正确的混合。首先，关闭深度缓冲区，以便控制重叠像素的绘制方式。然后，把混合因子设置为GL_SRC_ALPHA_SATURATE（源）和GL_ONE（目标）。使用这个特殊的混合函数，最终的颜色是目标颜色与经过缩放的源颜色的和。缩放因子是源alpha值或1减去目标alpha值中较小的那个。这意味着对于一个具有较大alpha值的像素，后续的源像素对最终颜色产生的影响很小，因为1减去目标alpha值的结果接近于零。按照这种方法，多边形边缘的像素可能最终与后来所绘制的多边形的颜色进行了混合。最后，在绘制场景中的所有多边形之前，需要按照从前到后的顺序对它们进行排序。

6.3 雾

计算机图像有时候由于过于清晰和锐利，反而显得不太逼真。抗锯齿处理使物体的边缘显得更为平滑，增加了逼真感。另外，还可以通过添加雾效果，使整幅图像变得更加逼真。所谓雾效果，就是使远处的物体看上去逐渐变得模糊。“雾”是一个通用的术语，描述了一些类似的大气效果。雾可以用于模拟模糊、薄雾、烟或污染（见彩图9）。雾在本质上是一种视觉模拟应用，用于模拟具有有限可视性的场合。飞行模拟器程序常常需要使用雾效果。

当雾启用之后，远离观察点的物体开始融入到雾颜色中。可以控制雾的浓度，它决定了物体随着距离的增加而融入到雾颜色的速度。另外，还可以设置雾的颜色。也可以在雾的距离计算中为每个顶

点指定一个雾坐标，而不是使用自动生成的深度值。

雾在RGBA和颜色索引模式下都可以使用，不过雾的计算在这两种模式下稍有不同。由于雾是在执行了矩阵变换、光照和纹理之后才应用的，因此它对经过变换的、带光照和经过纹理贴图的物体产生影响。注意，在大型的模拟程序中，雾可以提高性能，因为它可以选择不绘制那些因为雾的影响而不可见的物体。

雾可以应用于所有类型的几何图元，包括点和直线。在点和直线上应用雾效果又称为深度提示(depth-cuing)（如彩图2所示），它在分子模型和其他应用程序中极为常用。

6.3.1 使用雾

雾的使用非常简单。可以在glEnable()函数中使用GL_FOG参数来启用雾效果，并使用glFog*()函数设置用于控制雾浓度的方程式。如果需要，可以用glHint()函数提供一个GL_FOG_HINT值，见表6-3。示例程序6-7绘制了5个红色的球体，每个球体和观察点的距离各不相同。可以按“f”键在3种不同的雾方程式之间进行选择。下一节将详细描述这些雾方程式。

示例程序6-7 RGBA模式下的5个雾化球体：fog.c

```
static GLint fogMode;

static void init(void)
{
   GLfloat position [] ={0.5,0.5,3.0,0.0 };

   glEnable(GL_DEPTH_TEST);

   glLightfv(GL_LIGHT0,GL_POSITION,position);

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   {
      GLfloat mat [3] ={0.1745,0.01175,0.01175};
      glMaterialfv(GL_FRONT,GL_AMBIENT,mat);
      mat [0] =0.61424;mat [1]=0.04136;mat [2] =0.04136;
      glMaterialfv(GL_FRONT,GL_DIFFUSE,mat);
      mat [0] =0.727811;mat [1] =0.626959;mat [2] =0.626959;
      glMaterialfv(GL_FRONT,GL_SPECULAR,mat);
      glMaterialf(GL_FRONT,GL_SHININESS,0.6*128.0);
   }

   glEnable(GL_FOG);
   {
      GLfloat fogColor [4] ={0.5,0.5,0.5,1.0};

      fogMode =GL_EXP;
      glFogi(GL_FOG_MODE,fogMode);
      glFogfv(GL_FOG_COLOR,fogColor);
      glFogf(GL_FOG_DENSITY,0.35);
      glHint(GL_FOG_HINT,GL_DONT_CARE);
      glFogf(GL_FOG_START,1.0);
      glFogf(GL_FOG_END,5.0);
```

```
    }
    glClearColor(0.5,0.5,0.5,1.0);/*fog color */
}

static void renderSphere(GLfloat x,GLfloat y,GLfloat z)
{
    glPushMatrix();
    glTranslatef(x,y,z);
    glutSolidSphere(0.4,16,16);
    glPopMatrix();
}
/*display()draws 5 spheres at different z positions.
*/
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);
    renderSphere(-2.,-0.5,-1.0);
    renderSphere(-1.,-0.5,-2.0);
    renderSphere(0.,-0.5,-3.0);
    renderSphere(1.,-0.5,-4.0);
    renderSphere(2.,-0.5,-5.0);
    glFlush();
}
void reshape(int w,int h)
{
    glViewport(0,0,(GLsizei)w,(GLsizei)h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <=h)
        glOrtho(-2.5,2.5,-2.5*(GLfloat)h/(GLfloat)w,
            2.5*(GLfloat)h/(GLfloat)w,-10.0,10.0);
    else
        glOrtho(-2.5*(GLfloat)w/(GLfloat)h,
            2.5*(GLfloat)w/(GLfloat)h,-2.5,2.5,-10.0,10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void keyboard(unsigned char key,int x,int y)
{
    switch (key){
        case 'f ';
        case 'F ':
            if (fogMode ==GL_EXP){
                fogMode =GL_EXP2;
                printf("Fog mode is GL_EXP2 \n ");
            }
            else if (fogMode ==GL_EXP2){
                fogMode =GL_LINEAR;
                printf("Fog mode is GL_LINEAR \n ");
            }
            else if (fogMode ==GL_LINEAR){
```

```
            fogMode =GL_EXP;
            printf("Fog mode is GL_EXP \n ");
         }
         glFogi(GL_FOG_MODE,fogMode);
         glutPostRedisplay();
         break;
      case 27:
         exit(0);
         break;
      default:
         break;
   }
}

int main(int argc,char**argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB |GLUT_DEPTH);
   glutInitWindowSize(500,500);
   glutCreateWindow(argv [0]);
   init();
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;
}
```

6.3.2 雾方程式

雾根据雾混合因子把雾颜色与源片断的颜色进行混合。这个因子f是根据下列这3个方程式之一进行计算的，并截取在[0,1]的范围之内：

$$f = e^{-(density \cdot z)} \text{(GL_EXP)}$$

$$f = e^{-(density \cdot z)^2} \text{(GL_EXP2)}$$

$$f = \frac{end - z}{end - start} \text{(GL_LINEAR)}$$

在这3个方程式中，z是观察点和片断中心的视觉坐标距离。可以根据逐顶点的雾坐标对z视觉坐标距离施加控制（在本节的“雾坐标”中描述）。density、start和end的值都是在glFog*()函数中指定的。f因子在RGBA模式和颜色索引模式下的用法不同。

兼容性扩展
GL_FOG

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

void **glFog**{if}v(GLenum *pname*, const TYPE **params*);

设置用于雾计算的参数和函数。如果pname是GL_FOG_MODE，那么param就是GL_EXP（默认）、GL_EXP2或GL_LINEAR，它们分别用于选择3种不同的雾因子。如果pname是GL_FOG_DENSITY、GL_FOG_START或GL_FOG_END，那么param就分别是方程式中表示density、start和end的值（如果是向量版本，param就是指向这些值的指

针)。在RGBA模式下，pname可以是GL_FOG_COLOR，此时param指向包含了雾的RGBA颜色的4个值。在颜色索引模式下，对应的pname的值是GL_FOG_INDEX，此时param是指定了雾的颜色索引值的单值。

图6-4显示了在使用不同参数值情况下的雾浓度方程式。

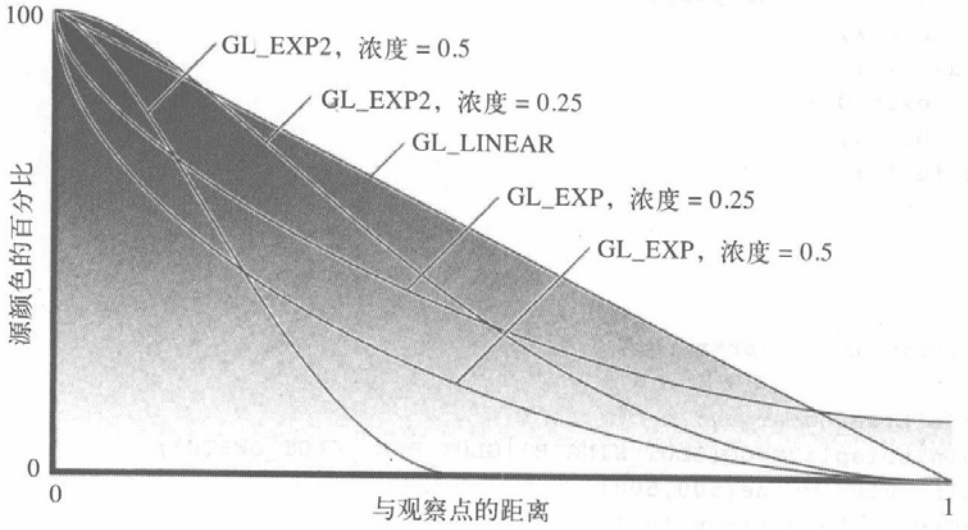


图6-4 雾浓度方程式

RGBA模式下的雾

在RGBA模式下，按照下面这种方式使用雾因子f计算最终的雾颜色：

$$C = fC_i + (1-f)\,C_f$$

其中C_i表示源片断的RGBA值，C_f表示用GL_FOG_COLOR分配的雾颜色值。

颜色索引模式下的雾

在颜色索引模式下，最终的雾颜色索引值是按照下面的公式计算的：

$$I = I_i + (1-f)\,I_f$$

其中I_i是源片断的颜色索引，I_f是GL_FOG_INDEX所指定的雾颜色索引。

为了在颜色索引模式下使用雾，必须在一个颜色表中加载适当的值。颜色表的第一种颜色是在没有雾的情况下物体的颜色，颜色表的最后一种颜色是物体完全雾化后的颜色。我们可能需要使用glClearIndex()函数对背景颜色索引进行初始化，使它对应于颜色表的最后一种颜色。按照这种方式，完全雾化的物体就与背景完全融合。类似地，在绘制物体之前，应该调用glIndex*()函数，并向它传递颜色表的第一种颜色（未雾化的颜色）。最后，为了在场景中对不同颜色的物体应用雾，需要创建几个颜色表，并在绘制每个物体之前调用glIndex*()函数把当前颜色索引设置为每个颜色表的起始颜色。示例程序6-8说明了如何对适当的条件进行初始化，并在颜色索引模式下应用雾效果。

示例程序6-8 颜色索引模式下的雾：fogindex.c

```
/*Initialize color map and fog.Set screen clear color
 *to end of color ramp.
 */
#define NUMCOLORS 32
#define RAMPSTART 16
```

```
static void init(void)
{
   int i;

   glEnable(GL_DEPTH_TEST);

   for (i =0;i <NUMCOLORS;i++){
      GLfloat shade;
      shade =(GLfloat)(NUMCOLORS-i)/(GLfloat)NUMCOLORS;
      glutSetColor(RAMPSTART +i,shade,shade,shade);
   }
   glEnable(GL_FOG);

   glFogi(GL_FOG_MODE,GL_LINEAR);
   glFogi(GL_FOG_INDEX,NUMCOLORS);
   glFogf(GL_FOG_START,1.0);
   glFogf(GL_FOG_END,6.0);
   glHint(GL_FOG_HINT,GL_NICEST);
   glClearIndex((GLfloat)(NUMCOLORS+RAMPSTART-1));
}

static void renderSphere(GLfloat x,GLfloat y,GLfloat z)
{
   glPushMatrix();
   glTranslatef(x,y,z);
   glutWireSphere(0.4,16,16);
   glPopMatrix();
}
/*display()draws 5 spheres at different z positions.
 */
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);
   glIndexi(RAMPSTART);

   renderSphere(-2.,-0.5,-1.0);
   renderSphere(-1.,-0.5,-2.0);
   renderSphere(0.,-0.5,-3.0);
   renderSphere(1.,-0.5,-4.0);
   renderSphere(2.,-0.5,-5.0);

   glFlush();
}

void reshape(int w,int h)
{
   glViewport(0,0,w,h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <=h)
      glOrtho(-2.5,2.5,-2.5*(GLfloat)h/(GLfloat)w,
         2.5*(GLfloat)h/(GLfloat)w,-10.0,10.0);
```

```
    else
       glOrtho(-2.5*(GLfloat)w/(GLfloat)h,
          2.5*(GLfloat)w/(GLfloat)h,-2.5,2.5,-10.0,10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void keyboard(unsigned char key,int x,int y)
{

    switch (key){
       case 27:
         exit(0);
    }
}

int main(int argc,char**argv)
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE |GLUT_INDEX |GLUT_DEPTH);
    glutInitWindowSize(500,500);
    glutCreateWindow(argv [0]);
    init();
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

雾坐标

如前所述，雾方程式使用雾坐标z来计算颜色值：

$$f = e^{-(density \cdot z)} \text{(GL_EXP)}$$

$$f = e^{-(density \cdot z)^2} \text{(GL_EXP2)}$$

$$f = \frac{end - z}{end - start} \text{(GL_LINEAR)}$$

在默认情况下，z是根据观察点和片断的距离自动计算产生的。但是，也可以对雾的计算施加灵活的控制。我们可能想模拟一个雾方程式，而不是沿用OpenGL提供的方程式。例如，我们可能想让一个飞行模拟器具有“基于地面”的雾，以便使用像海平面一样的浓雾。

在OpenGL 1.4版本中，可以调用glFog(GL_FOG_COORD_SRC, GL_FOG_COORD)显式地为每个顶点指定z值。在显式的雾坐标模式下，可以用glFogCoord*()函数指定每个顶点的雾坐标。

兼容性扩展

glFogCoord

void **glFogCoord**{fd}(TYPE z);

void **glFogCoord**{fd}v(const TYPE *z);

把当前的雾坐标设置为z。如果GL_FOG_COORD是当前的雾坐标来源，当前的雾方程式（GL_LINEAR、GL_EXP或GL_EXP2）就会使用当前的雾坐标来计算雾。

z值应该是正的，表示视觉坐标距离。应该避免使用负值来表示雾坐标，因为使用负的雾坐标可能会计算出奇怪的颜色。

在几何图元的内部，雾坐标可能由每个片断插值而得。

示例程序6-9对一个三角形进行渲染，它允许我们通过几个数字键对每个顶点的雾坐标进行更改。在显式雾坐标模式下，前后移动观察点（按下“f”和“b”键）并不会转换雾坐标，因此不会对顶点的颜色产生影响。如果不使用显式雾坐标（按下“c”键），移动观察点将会极大地影响由计算所产生的雾颜色。

示例程序6-9 雾坐标：fogcoord.c

```
static GLfloat f1,f2,f3;

/*Initialize fog
 */
static void init(void)
{
   GLfloat fogColor [4] ={0.0,0.25,0.25,1.0};
   f1 =1.0f;
   f2 =5.0f;
   f3 =10.0f;

   glEnable(GL_FOG);
   glFogi (GL_FOG_MODE,GL_EXP);
   glFogfv (GL_FOG_COLOR,fogColor);
   glFogf (GL_FOG_DENSITY,0.25);
   glHint (GL_FOG_HINT,GL_DONT_CARE);
   glFogi(GL_FOG_COORD_SRC,GL_FOG_COORD);
   glClearColor(0.0,0.25,0.25,1.0);/*fog color */
}
   /*display()draws a triangle at an angle.
    */
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);

   glColor3f(1.0f,0.75f,0.0f);
   glBegin(GL_TRIANGLES);
   glFogCoordf(f1);
   glVertex3f(2.0f,-2.0f,0.0f);
   glFogCoordf(f2);
   glVertex3f(-2.0f,0.0f,-5.0f);
   glFogCoordf(f3);
   glVertex3f(0.0f,2.0f,-10.0f);
   glEnd();
   glutSwapBuffers();
}

void keyboard(unsigned char key,int x,int y)
{
   switch (key){
      case 'c ':
         glFogi(GL_FOG_COORD_SRC,GL_FRAGMENT_DEPTH);
         glutPostRedisplay();
```

```
            break;
        case 'C ':
            glFogi(GL_FOG_COORD_SRC,GL_FOG_COORD);
            glutPostRedisplay();
            break;
        case '1 ':
            f1 =f1 +0.25;
            glutPostRedisplay();
            break;
        case '2 ':
            f2 =f2 +0.25;
            glutPostRedisplay();
            break;
        case '3 ':
            f3 =f3 +0.25;
            glutPostRedisplay();
            break;
        case '8 ':
            if (f1 >0.25){
                f1 =f1 -0.25;
                glutPostRedisplay();
            }
            break;
        case '9 ':
            if (f2 >0.25){
                f2 =f2 -0.25;
                glutPostRedisplay();
            }
            break;
        case '0 ':
            if (f3 >0.25){
                f3 =f3 -0.25;
                glutPostRedisplay();
            }
            break;
        case 'b ':
            glMatrixMode(GL_MODELVIEW);
            glTranslatef(0.0,0.0,-0.25);
            glutPostRedisplay();
            break;
        case 'f ':
            glMatrixMode(GL_MODELVIEW);
            glTranslatef(0.0,0.0,0.25);
            glutPostRedisplay();
            break;
        case 27:
            exit(0);
            break;
        default:
            break;
    }
}
```

6.4 点参数

在有些情况下，我们可能想对那些看上去像圆或球体的物体进行渲染，但又不想使用效率较低的多边形近似模拟法。例如，在飞行模拟器应用程序中，我们可能想对跑道上的路灯进行建模，当飞机靠近跑道时，路灯变得越来越大、越来越亮。或者，我们可能想模拟液滴（例如雨滴和游戏程序中飞溅的血滴）。我们可以用微粒系统来模拟这些现象。

路灯和液滴都可以渲染成点图元，但是需要对点进行修改，使它们具有适当的大小和亮度。可以使用glPointSize()和glEnable(GL_POINT_SMOOTH)创建较大的圆点，并使用雾来提示距离。但是，glPointSize()函数不能出现在glBegin()和glEnd()之间，因此就难以改变点的大小。我们必须随时重新计算点的大小。然后，为了获得最佳的性能，根据大小对它们进行重新分组。

点参数是一种自动、优雅的解决方案。它根据点和观察点的距离，对点的大小和亮度进行衰减。可以使用glPointParameterf*()函数指定衰减方程式的系数以及点的alpha成分（用于控制亮度）。

void **glPointParameter**{if}(GLenum *pname*, GLfloat *param*);

void **glPointParameter**{if}v(GLenum *pname*, const TYPE **param*);

设置与点图元的渲染相关的值。

如果pname是GL_POINT_DISTANCE_ATTENUATION，那么param就是一个包含了3个浮点值（a, b, c）的数组，分别包含了常数、线性或二次衰减系数，用于根据点与观察点的距离d来计算点的大小和亮度：

$$derivedSize = clamp\left(size \times \sqrt{\left(\frac{1}{a + b \times d + c \times d^2}\right)}\right)$$

如果pname设置为GL_POINT_SIZE_MIN或GL_POINT_SIZE_MAX，param便是一个绝对限制（分别表示最低值和最高值），用于以前的方程式中，对通过计算所产生的点大小进行截取。

如果启用了多重采样，并且pname是GL_FADE_THRESHOLD_SIZE，那么param就指定了点大小的一个不同的低限（threshold）。如果derivedSize < threshold，那么计算所得的fade因子就用于对alpha值进行调整，由此衰减它的亮度：

$$fade = \left(\frac{derivedSize}{threshold}\right)^2$$

如果pname是GL_POINT_SPRITE_COORD_ORIGIN，并且param是GL_LOWER_LEFT时，那么点块纹理上被迭代的纹理坐标的原点就是左下角的片断，t纹理坐标沿片断自底向上垂直增加。另外，如果param设置为GL_UPPER_LEFT，t纹理坐标就自顶向下垂直减少。

兼容性扩展
GL_POINT_SIZE_MIN
GL_POINT_SIZE_MAX
GL_POINT_DISTANCE_ATTENUATION

GL_POINT_DISTANCE_ATTENUATION距离的计算类似于局部光源的衰减系数计算。在示例程序6-10中，按下“c”、“l”或“q”键可以在常数、线性或二次衰减方程式之间切换。按下“f”或“b”键可以前或后移动观察点，在线性或二次衰减方程式中使点看起来显得更大或更小。

示例程序6-10 点参数：pointp.c

```
static GLfloat constant [3] ={1.0,0.0,0.0};
static GLfloat linear [3] ={0.0,0.12,0.0};
static GLfloat quadratic [3] ={0.0,0.0,0.01};
...
void keyboard(unsigned char key,int x,int y){
 switch (key){
 case 'c ':
  glPointParameterfv (GL_POINT_DISTANCE_ATTENUATION,constant);
  glutPostRedisplay();
 break;
 case 'l ':
  glPointParameterfv (GL_POINT_DISTANCE_ATTENUATION,linear);
  glutPostRedisplay();
  break;
 case 'q ':
  glPointParameterfv (GL_POINT_DISTANCE_ATTENUATION,quadratic);
  glutPostRedisplay();
  break;
 case 'b ':
  glMatrixMode (GL_MODELVIEW);
  glTranslatef (0.0,0.0,-0.5);
  glutPostRedisplay();
  break;
 case 'f ':
  glMatrixMode (GL_MODELVIEW);
  glTranslatef (0.0,0.0,0.5);
  glutPostRedisplay();
break;
...
```

如果在示例程序6-10中选择线性或二次衰减系数，并让观察点非常靠近一个点，此时这个点看上去会显得比实际上更大，因为这个点的原大小除以了一个小于1的因子。为了缓解这种情况，可以增大常数衰减因子或者用GL_POINT_MAX_SIZE设置最大值限制。

在使用点参数时，我们所需要的肯定是圆点而不是方块的点，因此需要像第6.2.1节描述的那样启用点抗锯齿功能。下面的代码就可以完成这个任务：

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

6.5 多边形偏移

如果想着重显示实心物体的边缘，可以用GL_FILL模式绘制这个物体，然后在GL_LINE模式下用一种不同的颜色再次绘制这个物体。但是，由于直线和多边形的光栅化方式并不完全相同，因此OpenGL为直线上的像素所产生的深度值和为多边形边缘上的像素所产生的深度值通常并不相同。即使是在同一个顶点，它们的值也可能不同。着重显示的直线可能受到与它重叠的多边形的影响而变得忽浓忽淡，从而产生非常不好的视觉效果。

可以使用多边形偏移来消除这种不良效果。多边形偏移增加了一个适当的偏移值，把重合的z值

适当地分开一些，使着重显示的直线与多边形的边缘清晰地分离开来（第10.2.3节描述的模板缓冲区也可以用于实现这种效果，但是多边形偏移的速度要比模板快得多）。多边形偏移也可以用于在表面上生成贴花、使用隐藏直线消除来渲染图像。除了直线和多边形之外，这个技巧也可以用于点图元。

可以使用3种方法启用多边形偏移，分别用于不同的多边形光栅化模式：GL_FILL、GL_LINE和GL_POINT。可以向glEnable ( )函数传递适当的参数（GL_POLYGON_OFFSET_FILL、GL_POLYGON_OFFSET_LINE或GL_POLYGON_OFFSET_POINT）来启用多边形偏移。还必须调用glPolygonMode() 函数设置当前的多边形光栅化方法。

---

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

如果启用多边形偏移，在执行深度测试之前，每个片断的深度值都将加上一个通过计算所产生的偏移值。这个偏移值是通过下面的公式计算的：

$$offset = m \cdot factor + r \cdot units$$

其中m是多边形的最大深度斜率（是在光栅化过程中计算的），r是保证能够产生可解析区别的窗口坐标深度值的最小值。r是一个因OpenGL实现而异的常数。

为了漂亮地渲染边缘着重显示的实心物体，避免产生不良的视觉效果，我们可以向实心物体添加一个正值（把它推向远处），或者向线框添加一个负值（把它拉到近处）。最大的问题是：为了实现我们所需要的效果，需要多大的偏移量？遗憾的是，偏移量的大小取决于许多因素，包括每个多边形的深度斜率以及线框直线的宽度。

OpenGL可以计算深度斜率（如图6-5所示）。深度斜率是在穿越多边形时z值除以x或y值的变化所得的值。深度值的范围限制在[0，1]之内，并且x和y坐标都是用窗口坐标表示的。为了估计多边形的最大深度斜率（偏移公式中的m），可以使用下面这个公式：

$$m = \sqrt{\left(\frac{\partial z}{\partial x}\right)^2 + \left(\frac{\partial z}{\partial y}\right)^2} \quad \left(\text{OpenGL实现可能会使用近似的 } m = \max\left(\frac{\partial z}{\partial x}, \frac{\partial z}{\partial y}\right)\right)$$

对于和近侧及远侧裁剪平面平行的多边形，它们的深度斜率为0。对于那些在场景中深度斜率接近于0的多边形，我们可能需要使用一个很小的常数偏移量。为了创建一个很小的常数偏移量，可以在调用glPolygonOffset()函数时向它传递factor = 0.0以及units = 1.0。

对于那些和裁剪平面存在很大角度的多边形，深度斜率可能明显大于0，此时就需要一个较大的偏移量。一个较小的、非零的factor值（例如0.75或1.0）便足以产生明显的深度区别，从而消除不良的视觉效果。

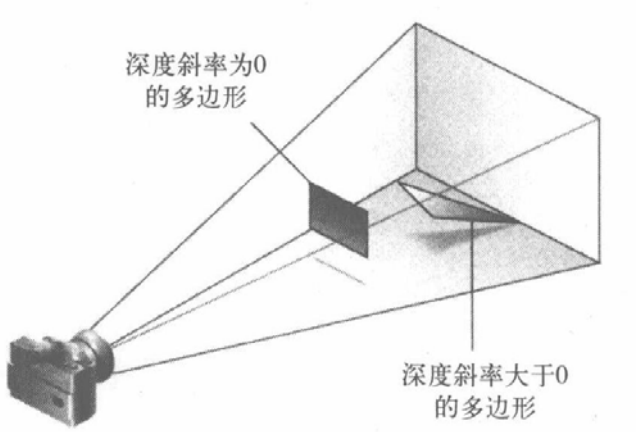


图6-5 多边形以及它们的深度斜率

示例程序6-11显示了一段代码，其中一个显示列表（用于绘制一个实心物体）首先在使用光照的条件下进行渲染。它使用默认的GL_FILL模式，并把用于产生偏移量的factor设置为1.0、units也设置为1.0。这些值可以保证场景中所有多边形的偏移量均已足够，不管它们的深度斜率有多大（这些值可能比实际上可以满足需要的最小值稍大一些，但是较大的偏移值更容易看到效果）。然后，为了着重显示第一个物体的边缘，可以在禁用多边形偏移的情况下把它渲染为不带光照的线框物体。

示例程序6-11 用于消除不良视觉效果的多边形偏移：polyoff.c

```
glEnable(GL_LIGHT0);
glEnable(GL_LIGHTING);
glPolygonOffset(1.0,1.0);

glEnable(GL_POLYGON_OFFSET_FILL);
glCallList(list);
glDisable(GL_POLYGON_OFFSET_FILL);

glDisable(GL_LIGHTING);
glDisable(GL_LIGHT0);
glColor3f(1.0,1.0,1.0);
glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
glCallList(list);
glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
```

在有些情况下，使用最简单的factor和units值（1.0和1.0）并不能满足要求。例如，如果需要着重显示的边缘的直线宽度大于1，那么我们可能需要增大factor的值。另外，由于深度值在使用透视投影时并不是均匀地变换到窗口坐标中（参见第3.4.2节），因此靠近近侧裁剪平面的多边形所需要的偏移量要小于靠近远侧裁剪平面的多边形。另外，读者可以试验一下不同的factor值，以保证获得自己所需要的效果。

第7章 显示列表

本章目标

- 理解显示列表如何与立即模式的命令一起使用，对数据进行组织，以提高性能。
- 了解如何使用显示列表以及应该在什么时候使用显示列表，最大限度地提高应用程序的性能。

注意：*在OpenGL 3.1中，本章介绍的所有技术和功能都被废弃删除了。*

显示列表是一组存储在一起的OpenGL函数，可以在以后执行。调用一个显示列表时，它所存储的函数就会按照顺序执行。绝大多数OpenGL函数既可以存储在显示列表中，也可以在立即模式下执行。可以在程序中混合使用立即模式的编程方式和显示列表。到目前为止，读者所看到的示例程序都采用了立即模式。本章介绍什么是显示列表，并讨论如何最有效地使用它们。本章的主要内容如下：

- 为什么使用显示列表：解释什么时候应该使用显示列表。
- 一个使用显示列表的例子：提供一个简单的例子，显示了使用显示列表的基本函数。
- 显示列表的设计哲学：解释一些设计选择的原因（例如显示列表是不可编辑的），并说明了使用显示列表有望实现什么程度的性能提升。
- 创建和执行显示列表：详细讨论用于创建、执行和删除显示列表的命令。
- 执行多个显示列表：显示如何连续执行几个显示列表，并使用一组小型的字符集作为例子。
- 用显示列表管理状态变量：说明如何使用显示列表保存和恢复那些用于设置状态变量的OpenGL函数。

7.1 为什么使用显示列表

显示列表可以提高性能，因为可以用它来存储OpenGL函数，供以后执行。如果需要多次重绘同一个几何图形，或者如果有一些需要多次调用的用于更改状态的函数，把这些函数存储在显示列表中是一个很好的思路。使用显示列表，可以一次定义几何图形（或状态更改），并在以后多次执行它们。

为了观察如何使用显示列表来存储几何图形，让我们考虑一个绘制三轮车的例子。三轮车的两个后轮大小相同，彼此之间相隔一定的距离。前轮比后轮要大一些，位置也不相同。绘制三轮车车轮的一种有效方法是把绘制一个轮子的操作存储在显示列表中，并3次执行这个显示列表。每次在执行显示列表之前，根据需要适当地设置模型视图矩阵，以便正确地计算每个轮子的大小和位置。

当通过网络在另一台远程机器上运行OpenGL程序时，把绘图命令保存在显示列表中具有特别重要的意义。在这种情况下，服务器更像是机器而不是宿主机（参见第1.1节关于OpenGL客户机/服务器模型的讨论）。由于显示列表是服务器状态的一部分，保存在服务器中。如果把需要重复调用的函数存储在显示列表中，显然可以大大减少通过网络传输的数据量。

当我们在本地机器上运行OpenGL程序时，可以把经常使用的函数存储在显示列表中，以提高程序的性能。有些图形硬件可能会把显示列表存储在专用的内存中，甚至以一种更为优化的形式来存储数据，使之与图形硬件或软件更为兼容（关于这些优化措施的详细讨论，请参阅第7.3节）。

7.2 一个使用显示列表的例子

显示列表是一种方便而又有效的方式，对一组OpenGL函数进行命名和组织。例如，假设我们想绘制一个圆环面，并从不同的角度观察它。完成这个任务的最有效方法就是把这个圆环面存储在一个显示列表中。然后，当我们想更改视图时，可以修改模型视图矩阵，然后执行这个显示列表来绘制这个圆环面。示例程序7-1描述了这个过程。

示例程序7-1 创建显示列表：torus.c

```
GLuint theTorus;

/*Draw a torus */
static void torus(int numc,int numt)
{
   int i,j,k;
   double s,t,x,y,z,twopi;

   twopi =2 *(double)M_PI;
   for (i =0;i <numc;i++){
      glBegin(GL_QUAD_STRIP);
   for (j =0;j <=numt;j++){
      for (k =1;k >=0;k--){
         s =(i +k)%numc +0.5;
         t =j %numt;

         x =(1+.1*cos(s*twopi/numc))*cos(t*twopi/numt);
         y =(1+.1*cos(s*twopi/numc))*sin(t*twopi/numt);
         z =.1 *sin(s *twopi /numc);
         glVertex3f(x,y,z);
      }
   }
   glEnd();
   }
}

/*Create display list with Torus and initialize state */
static void init(void)
{
   theTorus =glGenLists(1);
   glNewList(theTorus,GL_COMPILE);
   torus(8,25);
   glEndList();

   glShadeModel(GL_FLAT);
   glClearColor(0.0,0.0,0.0,0.0);
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0,1.0,1.0);
   glCallList(theTorus);
```

```
    glFlush();
}

void reshape(int w,int h)
{
    glViewport(0,0,(GLsizei)w,(GLsizei)h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(30,(GLfloat)w/(GLfloat)h,1.0,100.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0,0,10,0,0,0,0,1,0);
}
/*Rotate about x-axis when "x " typed;;rotate about y-axis
when "y " typed;"i " returns torus to original view */
void keyboard(unsigned char key,int x,int y)
{
    switch (key){
    case 'x ':
    case 'X ':
       glRotatef(30.,1.0,0.0,0.0);
       glutPostRedisplay();
       break;
    case 'y ':
    case 'Y ':
       glRotatef(30.,0.0,1.0,0.0);
       glutPostRedisplay();
       break;
    case 'i ':
    case 'I ':
       glLoadIdentity();
       gluLookAt(0,0,10,0,0,0,0,1,0);
       glutPostRedisplay();
       break;
    case 27:
       exit(0);
       break;
    }
}

int main(int argc,char **argv)
{
    glutInitWindowSize(200,200);
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB);
    glutCreateWindow(argv [0]);
    init();
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

首先让我们观察init()函数。它创建了一个用于绘制圆环面的显示列表，并对OpenGL渲染状态进行初始化。注意，用于绘制圆环面的函数torus()位于一对glNewList()和glEndList()之间，这对函数用于对显示列表进行定界。glNewList()函数的listName参数是一个整型索引值，由glGenLists()函数产生，它可以唯一地标识一个显示列表。

当窗口具有焦点时，用户可以通过按下“x”或“y”键使这个圆环面绕x或y轴旋转。每当此时，回调函数keyboard()就会被调用，它把一个30°的旋转矩阵（绕x轴或y轴）加载到当前的模型视图矩阵中。然后，glutPostRedisplay()函数被调用，它导致glutMainLoop()函数调用display()函数，并在其他事件被处理之后渲染这个圆环面。当用户按下“i”键时，keyboard()函数将恢复初始的模型视图矩阵，并把这个圆环面恢复到原先的位置。

display()函数非常简单。它清除窗口并调用glCallList()函数执行这个显示列表所存储的函数。如果我们没有使用显示列表，display()函数不得不在每次被调用时重复调用这些函数来绘制这个圆环面。

显示列表只能包含OpenGL函数。在示例程序7-1中，只有glBegin()、glVertex()和glEnd()函数存储在显示列表中。它们的参数将会进行求值，最终的值是在显示列表被创建时复制到显示列表的。用于创建这个圆环面的所有三角函数只出现一次，因此应该能够提高性能。但是，显示列表中的值不能在以后进行修改。当一个函数被存储到显示列表之后，就没有办法将它从这个显示列表中删除。定义了一个显示列表之后，没有办法再在其中插入新函数。可以删除整个显示列表并创建一个新的显示列表，但是不能对显示列表进行编辑。

注意：显示列表也适用于GLU函数，这是因为它们最终都可以分解为低层的OpenGL函数，后者可以方便地存储在显示列表中。在GLU中使用显示列表对于GLU分格化（参见第11章）和NURBS（参见第12章）的性能优化具有特别重要的意义。

7.3 显示列表的设计哲学

为了优化性能，OpenGL的显示列表更像是命令缓存器，而不是动态数据库。换句话说，当显示列表创建之后，它就无法进行修改。如果允许修改显示列表，搜索显示列表和执行内存管理所带来的开销将会降低显示列表的性能。当一个可修改的显示列表的一部分被修改时，在分配和销毁内存时可能会产生内存碎片。OpenGL实现为了提高效率对显示列表所存储的函数所做的所有修改都必须重新完成。另外，由于这种可修改的显示列表的访问难度可能加大，因此通过网络或系统总线对它们进行缓存的难度也随之加大。

显示列表存储的函数的优化方式因不同的OpenGL实现而异。例如，一个像glRotate*()这样的简单函数如果存储在显示列表中可能会有相当大的性能提升，因为用于产生旋转矩阵的计算是比较复杂的（涉及平方根和三角函数）。但是，在显示列表中，只有最终的旋转矩阵需要存储，因此存储在显示列表中的旋转函数的执行速度可能和执行glMultMatrix*()函数一样快。有些复杂的OpenGL实现甚至可以把几条相邻的转换命令组合为一个单独的矩阵乘法。

尽管我们无法保证自己使用的OpenGL实现是否会针对任何特定用途对显示列表进行优化，但是显示列表的执行速度无论如何也不会比线性地执行存储在显示列表中的函数更慢。当然，显示列表也存在一定的开销。如果显示列表非常小，使用显示列表的开销可能会超过它所带来的好处。下面列出了显示列表最能够体现优化作用的领域，并注明了这些主题在本书中的具体章节。

- 矩阵操作（第3章）。大多数矩阵操作要求OpenGL计算逆矩阵。用于计算矩阵及其逆矩阵的函数都可能被特定的OpenGL实现放在显示列表中。

- 对位图和图像进行光栅化（第8章）。程序指定的光栅化数据的格式并不一定最适合硬件。在编译显示列表时，OpenGL可能会把数据转换为最适合硬件的形式。这些光栅字符的绘图速度可能会有很大的提高，因为字符串通常是由一系列的小型位图组成的。
- 光源、材料属性和光照模型（第5章）。在复杂的光照条件下绘制场景时，可能需要更改场景中每个物体使用的材料。设置材料的速度可能非常慢，因为它可能涉及非常复杂的计算。如果把材料定义放在显示列表中，每次切换不同的材料时，就不必重新执行这些计算，因为显示列表存储的只是最终的计算结果。因此，如果使用显示列表，光照场景的渲染速度会更快（关于使用显示列表对诸如光照条件之类的值进行更改的更多细节，请参阅第7.6节中的“封装模式修改”）。
- 多边形点画模式（第2章）。

注意：为了优化纹理图像，应该把纹理数据存储在纹理对象而不是显示列表中。

这里所列出的用于指定属性的一些函数因不同的渲染环境而异，因此需要考虑这个因素，以保证实现性能的优化。例如，在启用了GL_COLOR_MATERIAL之后，有些材料属性会追踪当前颜色（参见第5章），所有设置相同材料属性的glMaterial*()调用都将被忽略。

在几何图形中存储状态设置可能会提高性能。例如，假设我们想在一些几何图形上应用一种变换，然后再绘制结果，可以使用如下代码：

```
glNewList(1, GL_COMPILE);
draw_some_geometric_objects();
glEndList();

glLoadMatrix(M);
glCallList(1);
```

但是，如果这些几何图形每次都以相同的方式进行变换，把这些矩阵存储在显示列表中更为合适。例如，如果我们像下面这样编写代码，有些OpenGL实现可能会在定义物体时（而不是在每次绘制它们时）通过对它们进行变换来提高性能：

```
glNewList(1, GL_COMPILE);
glLoadMatrix(M);
draw_some_geometric_objects();
glEndList();
glCallList(1);
```

更有可能出现的情况是在渲染图像时。正如我们将在第8章所看到的那样，可以修改像素传输状态变量，控制图像和位图的光栅化方式。如果设置这些状态变量的函数出现在显示列表中，位于定义图像或位图的函数之前，OpenGL就可以预先执行相关的操作，并对结果进行缓存。

记住，显示列表也有一些缺点。如果显示列表非常小，它的效率可能不高，因为显示列表本身也存在一定的开销。显示列表的另一个缺点是它的内容是不可修改的。为了提高性能，OpenGL不允许修改显示列表的内容，也不允许读取它的内容。如果应用程序需要独立地维护显示列表中的数据（例如，进行后续的数据处理），可能需要大量的额外内存。

7.4 创建和执行显示列表

正如读者所看到的那样，glNewList()和glEndList()用于开始和结束一个显示列表的定义。可以向glCallList()函数提供显示列表的标识索引，从而执行这个显示列表。在示例程序7-2中，我们在init()函数中创建了一个显示列表。这个显示列表包含了用于绘制红色三角形的函数。然后，在display()函

数中，这个显示列表执行了10次。另外，这个程序使用立即模式绘制了一条直线。注意，显示列表将会分配内存，用于存储这些函数以及所有必要的变量值。

示例程序7-2 使用显示列表：list.c

```
GLuint listName;

static void init(void)
{
   listName =glGenLists(1);
   glNewList(listName,GL_COMPILE);
      glColor3f(1.0,0.0,0.0);/*current color red */
      glBegin(GL_TRIANGLES);
      glVertex2f(0.0,0.0);
      glVertex2f(1.0,0.0);
      glVertex2f(0.0,1.0);
      glEnd();
      glTranslatef(1.5,0.0,0.0);/*move position */
   glEndList();
   glShadeModel(GL_FLAT);
}

static void drawLine(void)
{
   glBegin(GL_LINES);
   glVertex2f(0.0,0.5);
   glVertex2f(15.0,0.5);
   glEnd();
}

void display(void)
{
   GLuint i;
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(0.0,1.0,0.0);/*current color green */
   for (i =0;i <10;i++)   /*draw 10 triangles */
      glCallList(listName);
   drawLine();/*Is this line green?NO!*/
              /*Where is the line drawn?*/
   glFlush();
}
```

显示列表中的glTranslate()函数用于更改下一个将要绘制的物体的位置。如果没有这个函数，两次调用这个显示列表将在相同的位置绘制两个三角形。在立即模式下调用的drawLine()函数也受到在它之前的10个glTranslate()调用的影响。因此，如果在显示列表中调用了变换函数，不要忘了考虑这些函数对程序后面的影响。

一次只能创建一个显示列表。换句话说，必须使用glNewList()和glEndList()结束一个显示列表的定义之后才能定义下一个显示列表。正如读者所预想的那样，在还没有开始定义显示列表的情况下直接调用glEndList()将会导致GL_INVALID_OPERATION错误（关于错误处理的更多信息，请参阅第14.1节）。

7.4.1 命名和创建显示列表

每个显示列表都由一个整型索引值标识。在创建显示列表时，必须要小心，不要选择已经被使用的索引值，避免导致原有的显示列表被覆盖。为了避免这种意外情况，可以使用glGenLists()函数产生一个或多个未使用的索引值。

GLuint **glGenLists**(GLsizei *range*);

分配range个连续的、以前未分配的显示列表索引。这个函数所返回的整型值标识了一块连续的空显示列表索引的起始位置。它所返回的所有索引都被标记为空和已使用，因此后续的**glGenLists()**调用就不会再返回这些值，除非它们已经删除。如果系统无法满足这个函数所请求分配的索引值的数量，或者range参数本身为0，这个函数将返回0。

兼容性扩展
glGenLists

在下面这个例子中，请求了一个索引值，如果请求成功，就用它来创建一个新的显示列表：

```
listIndex = glGenLists(1);
if (listIndex != 0) {
   glNewList(listIndex,GL_COMPILE);
      ...
   glEndList();
}
```

注意：0并不是合法的显示列表索引值。

void **glNewList**(GLuint *list*, GLenum *mode*);

指定了一个显示列表的开始。在它之后的OpenGL函数（直到调用**glEndList()**结束这个显示列表定义之前）将存储在一个显示列表中，除了那些无法存储于显示列表的OpenGL函数之外（在执行显示列表时，这些函数将以立即模式执行）。list是一个非0的正整数，它唯一地标识了这个显示列表。mode参数可以使用的值是GL_COMPILE和GL_COMPILE_ AND_EXECUTE。如果把这些函数放在显示列表中时并不想立即执行它们，应该使用GL_COMPILE。如果希望立即执行这些函数并把它们放在显示列表中供以后使用，应该使用GL_COMPILE_AND_EXECUTE。

兼容性扩展
glNewList
glEndList
GL_COMPILE
GL_COMPILE_AND_EXECUTE

void glEndList(void);

标志一个显示列表定义的结束。

当显示列表被创建时，它与当前的OpenGL渲染环境一起存储。因此，当这个渲染环境销毁时，这个显示列表也销毁。有些窗口系统允许多个渲染环境共享显示列表。在这种情况下，当共享组（share group）中的最后一个渲染环境销毁时，这个显示列表才会销毁。

7.4.2 存储在显示列表里的是什么

在创建显示列表时，只有表达式的值存储在显示列表中。如果数组中的值以后发生了更改，显示列表中的值并不会发生变化。在下面这段代码中，显示列表包含了一条把当前的RGBA颜色设置为黑色（0.0, 0.0, 0.0）的命令。如果color_vector数组中的值以后被修改为红色（1.0, 0.0, 0.0），它不会对显示列表产生影响，因为显示列表包含的是当它被创建时生效的值：

```
GLfloat color_vector[3] = {0.0, 0.0, 0.0};
glNewList(1, GL_COMPILE);
   glColor3fv(color_vector);
glEndList();
color_vector[0] = 1.0;
```

并不是所有的OpenGL函数都可以存储在显示列表中并执行。例如，设置客户机状态的函数以及用于提取状态值的函数就无法存储在显示列表中（这些函数有很多极易分辨，因为它们都有返回值，或者直接返回，或者把值保存在一个传引用的参数中）。当我们在创建显示列表时调用了这些函数时，它们会立即执行。

表7-1列举了无法存储于显示列表中的OpenGL函数。注意，当使用glNewList()创建了一个显示列表后，如果再把它放在其他显示列表内部，也会产生错误。读者可能还没有看到过这张表中的许多函数，它们将在以后的章节中分别介绍。

表7-1 无法在显示列表中存储的OpenGL函数

glAreTexturesResident	glEdgeFlagPointer	glIsShader
glAttachShader	glEnableClientState	glIsTexture
glBindAttribLocation	glEnableVertexAttribArray	glLinkProgram
glBindBuffer	glFeedbackBuffer	glMapBuffer
glBufferData	glFinish	glNormalPointer
glBufferSubData	glFlush	glPixelStore
glClientActiveTexture	glFogCoordPointer	glPopClientAttrib
glColorPointer	glFogCoordPointer	glPushClientAttrib
glCompileShader	glGenBuffers	glReadPixels
glCreateProgram	glGenLists	glRenderMode
glCreateShader	glGenQueries	glSecondaryColorPointer
glDeleteBuffers	glGenTextures	glSecondaryColorPointer
glDeleteLists	glGet*	glSelectBuffer
glDeleteProgram	glIndexPointer	glShaderSource
glDeleteQueries	glInterleavedArrays	glTexCoordPointer
glDeleteShader	glIsBuffer	glUnmapBuffer
glDeleteTextures	glIsEnabled	glValidateProgram
glDetachShader	glIsList	glVertexAttribPointer
glDisableClientState	glIsProgram	glVertexPointer
glDisableVertexAttribArray	glIsQuery	

为了理解这些函数为什么无法存储在显示列表中，读者需要知道在使用OpenGL访问网络时，客户机和服务器可能在不同的机器上。当一个显示列表被创建时，它存储于服务器中，因此服务器无法依赖客户机来获取与显示列表相关的任何信息。如果显示列表允许存储像glGet*()或glIs*()这样的查询函数，调用程序将不知道如何处理通过网络返回的数据。如果在发送数据时不对显示列表进行解析，调用程序就不知道该把数据放在什么地方。因此，所有具有返回值的函数都不能存储在显示列表中。

用于修改客户状态的函数（如glPixelStore()、glSelectBuffer()）以及定义顶点数组的函数也无法存储在显示列表中。例如，用于指定顶点数组的函数（如glVertexPointer()、glColorPointer()和glInterleavedArrays()）设置客户状态指针，因此它们无法存储在显示列表中。glArrayElement()、glDrawArrays()和glDrawElements()向服务器发送数据，让服务器根据被启用的数组来创建图元（参

见第2.6节）。存储在显示列表中的顶点数组数据是通过对指针进行解引用获取的，而不是通过存储指针本身来获取的。因此，以后对顶点数组中的数据所做的修改将不会影响显示列表中的图元定义。

另外，所有使用了像素存储模式的函数也无法存储在显示列表中，因为它们使用的模式将会在把它们放入显示列表时生效（参见第8.3.2节）。其他依赖客户状态的函数（如glFlush()和glFinish()）也无法存储在显示列表中，因为它们依赖于当它们被执行时所生效的客户状态。

7.4.3 执行显示列表

在创建了显示列表之后，就可以通过调用glCallList()函数来执行它。很显然，可以多次执行同一个显示列表，也可以把显示列表与其他立即模式下的函数混合执行。

兼容性扩展
glCallList

void **glCallList**(GLuint *list*);

这个函数执行由list指定的显示列表。显示列表中的函数按照它们的存储顺序执行，就像它们并没有放在显示列表中一样。如果list并未定义，这个函数就不执行任何任务。

可以在程序中的任何地方调用glCallList()，只要有一个可以访问这个显示列表的OpenGL渲染环境（也就是这个显示列表创建时的活动渲染环境或者是同一个共享组中的渲染环境）处于活动状态。显示列表可以在一个函数中创建并在另一个函数中执行，因为它的索引能够唯一地标识它。另外，并不存在把显示列表的内容保存到数据文件的工具，也不存在根据文件创建显示列表的工具。从这种意义上说，显示列表用于临时的设计。

7.4.4 层次式显示列表

可以创建层次式显示列表（hierarchical display list），这是一种在glNewList()和glEndList()之间通过调用glCallList()执行其他显示列表的显示列表。层次式显示列表适用于那些由不同成分组成的物体，尤其是当有些成分需要多次使用的时候。例如，下面这个显示列表用于渲染一辆自行车，它调用了其他显示列表来渲染自行车的不同部分：

```
glNewList(listIndex,GL_COMPILE);
   glCallList(handlebars);
   glCallList(frame);
   glTranslatef(1.0, 0.0, 0.0);
   glCallList(wheel);
   glTranslatef(3.0, 0.0, 0.0);
   glCallList(wheel);
glEndList();
```

为了避免无限的递归，显示列表的嵌套层次具有限制。这个限制至少是64，也可能更高，取决于OpenGL实现。为了确定自己所使用的OpenGL实现的嵌套限制，可以调用：

```
glGetIntegerv(GL_MAX_LIST_NESTING, GLint *data);
```

OpenGL允许创建的显示列表调用其他尚未创建的显示列表。当一个显示列表调用另一个尚未定义的显示列表时，后者不会执行任何操作。

可以使用层次式显示列表模拟可编辑的显示列表，就是用几个低层的显示列表来包装一个显示列表。例如，为了把一个多边形放在显示列表中并方便地编辑它的顶点，可以使用示例程序7-3的代码。

示例程序7-3 层次式显示列表

```
glNewList(1,GL_COMPILE);
   glVertex3fv(v1);
glEndList();
glNewList(2,GL_COMPILE);
   glVertex3fv(v2);
glEndList();
glNewList(3,GL_COMPILE);
   glVertex3fv(v3);
glEndList();

glNewList(4,GL_COMPILE);
   glBegin(GL_POLYGON);
      glCallList(1);
      glCallList(2);
      glCallList(3);
   glEnd();
glEndList();
```

为了渲染这个多边形，可以4次调用这个显示列表。为了编辑顶点，只需要重新创建与这个顶点相对应的那个显示列表。由于一个索引数可以唯一地标识一个显示列表，因此创建一个与原有的显示列表具有相同索引的显示列表将自动删除原先的那个显示列表。记住，这种技巧并不能优化内存的使用，也不能进一步提高性能，但它是一种可以接受的做法，在有些场合具有应用价值。

7.4.5 管理显示列表索引

至此，我们推荐使用glGenLists()来获取未使用的显示列表索引。如果不想使用glGenLists()，也可以使用glIsList()来判断一个特定的索引是否已被使用。

可以使用glDeleteLists()函数显式地删除一个特定的显示列表或者一组连续范围的显示列表。使用glDeleteLists()后，那些被删除的显示列表的索引可以重新使用。

GLboolean **glIsList**(GLuint *list*);

如果list已经用于表示显示列表，这个函数返回GL_TRUE，否则返回GL_FALSE。

兼容性扩展
glIsList

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

删除range个显示列表，从list指定的索引开始。如果试图删除一个尚未创建的显示列表，这个行为将被忽略。

兼容性扩展
glDeleteLists

7.5 执行多个显示列表

OpenGL提供了一种有效的机制，可以连续执行几个显示列表。这个机制要求我们把显示列表索引放在一个数组中并调用glCallLists()函数。这个机制的一种明显用途就是当一组显示列表索引对应于有意义的值时。例如，我们创建了一种字体，每个显示列表索引都可能对应于这种字体的一个字符的ASCII值。为了拥有几种像这样的字体，我们需要为每种字体建立一个不同的初始显示列表索引。可以在调用glCallLists()之前使用glListBase()函数来指定这个初始索引。

void **glListBase**(GLuint *base*);

指定一个偏移量，它将与**glCallLists()**函数中的显示列表索引相加，以获取最终的显示列表索引。默认的显示列表基址是0。显示列表基址对于**glCallList()**函数并无效果，后者只执行显示列表。另外，显示列表基址对于**glNewList()**也没有效果。

兼容性扩展
glListBase
glCallLists

void **glCallLists**(GLsizei *n*, GLenum *type*, const GLvoid **lists*);

执行n个显示列表。被执行的显示列表的索引是通过把当前显示列表基址表示的偏移值（由**glListBase()**函数指定）与list指定的数组中的有符号整型值相加得到的。

type参数表示lists数组中值的类型。它可以设置为GL_BYTE、GL_UNSIGNED_BYTE、GL_SHORT、GL_UNSIGNED_SHORT、GL_INT、GL_ UNSIGNED_ INT或GL_FLOAT，分别表示lists应该被看作字节数组、无符号字节数组、短整型数组、无符号短整型数组、整型数组、无符号整型数组或浮点型数组。type也可以是GL_2_BYTES、GL_3_BYTES或GL_4_BYTES，它们表示从lists中连续读取2、3或4个字节，并逐字节进行移动和相加，以计算显示列表偏移值。它使用了下面这个算法（其中byte[0]是字符序列的起始位置）：

```
/* b = 2, 3, or 4; bytes are numbered 0, 1, 2, 3 in array */
offset = 0;
for (i = 0; i < b; i++) {
   offset = offset << 8;
   offset += byte[i];
}
index = offset + listbase;
```

对于多字节的数据，高端的数据首先出现，就像从数组中按顺序读取字节一样。

示例程序7-4是完整的示例程序7-5的一部分，它使用了多个显示列表。这个程序用一种笔画字体绘制字符（一组由线段组成的字母）。initStrokedFont()为每个字符设置显示列表索引，使它们对应于它们的ASCII值。

示例程序7-4 定义多个显示列表

```
void initStrokedFont(void)
{
   GLuint base;

   base =glGenLists(128);
   glListBase(base);
   glNewList(base+'A ',GL_COMPILE);
      drawLetter(Adata);glEndList();
   glNewList(base+'E ',GL_COMPILE);
      drawLetter(Edata);glEndList();
   glNewList(base+'P ',GL_COMPILE);
      drawLetter(Pdata);glEndList();
   glNewList(base+'R ',GL_COMPILE);
      drawLetter(Rdata);glEndList();
   glNewList(base+'S ',GL_COMPILE);
      drawLetter(Sdata);glEndList();
   glNewList(base+' ',GL_COMPILE);     /*space character */
      glTranslatef(8.0,0.0,0.0);
```

```
    glEndList();
}
```

glGenLists()函数分配128个连续的显示列表索引，其中第一个索引就成为显示列表基址。我们为每个字母创建了一个显示列表，每个显示列表的索引是基址与这个字母对应的ASCII值相加的结果。

在这个例子中，我们只创建了几个字母以及空格字符。

在创建了显示列表之后，就可以调用glCallLists()来执行显示列表。例如，可以向子函数printStrokedString()传递一个字符串：

```
void printStrokedString(GLbyte *s)
{
   GLint len = strlen(s);
   glCallLists(len, GL_BYTE, s);
}
```

这个字符串中每个字母的ACSII值作为显示列表索引的偏移值使用。当前的显示列表基址与每个字母的ACSII值相加，确定最终需要执行的显示列表的索引。示例程序7-5产生的输出如图7-1所示。



图7-1 定义字符A、E、P、R、S的笔画字体

示例程序7-5 定义一种笔画字体的多个显示列表：stroke.c

```
#define PT 1
#define STROKE 2
#define END 3

typedef struct charpoint {
   GLfloat x,y;
   int type;
}CP;

CP Adata [] ={
   {0,0,PT},{0,9,PT},{1,10,PT},{4,10,PT},
   {5,9,PT},{5,0,STROKE},{0,5,PT},{5,5,END}
};

CP Edata [] ={
   {5,0,PT},{0,0,PT},{0,10,PT},{5,10,STROKE},
   {0,5,PT},{4,5,END}
};

CP Pdata [] ={
   {0,0,PT},{0,10,PT},{4,10,PT},{5,9,PT},{5,6,PT},
   {4,5,PT},{0,5,END}
};

CP Rdata [] ={
   {0,0,PT},{0,10,PT},{4,10,PT},{5,9,PT},{5,6,PT},
   {4,5,PT},{0,5,STROKE},{3,5,PT},{5,0,END}
};
```

```
CP Sdata [] ={
    {0,1,PT},{1,0,PT},{4,0,PT},{5,1,PT},{5,4,PT},
    {4,5,PT},{1,5,PT},{0,6,PT},{0,9,PT},{1,10,PT},
    {4,10,PT},{5,9,END}
};

/*drawLetter()interprets the instructions from the array
*for that letter and renders the letter with line segments.
*/
static void drawLetter(CP *l)
{
   glBegin(GL_LINE_STRIP);
   while (1){
      switch (l->type){
         case PT:
            glVertex2fv(&l->x);
            break;
         case STROKE:
            glVertex2fv(&l->x);
            glEnd();
            glBegin(GL_LINE_STRIP);
            break;
         case END:
            glVertex2fv(&l->x);
            glEnd();
            glTranslatef(8.0,0.0,0.0);
            return;
      }
      l++;
   }
}
/*Create a display list for each of 6 characters.*/
static void init(void)
{
   GLuint base;

   glShadeModel(GL_FLAT);

   base =glGenLists(128);
   glListBase(base);
   glNewList(base+'A ',GL_COMPILE);
   drawLetter(Adata);
   glEndList();
   glNewList(base+'E ',GL_COMPILE);
   drawLetter(Edata);
   glEndList();
   glNewList(base+'P ',GL_COMPILE);
   drawLetter(Pdata);
   glEndList();
   glNewList(base+'R ',GL_COMPILE);
   drawLetter(Rdata);
   glEndList();
```

```
    glNewList(base+'S ',GL_COMPILE);
    drawLetter(Sdata);
    glEndList();
    glNewList(base+' ',GL_COMPILE);
    glTranslatef(8.0,0.0,0.0);
    glEndList();
}

char *test1 ="A SPARE SERAPE APPEARS AS ";
char *test2 ="APES PREPARE RARE PEPPERS ";

static void printStrokedString(char *s)
{
    GLsizei len =strlen(s);
    glCallLists(len,GL_BYTE,(GLbyte *)s);
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0,1.0,1.0);
    glPushMatrix();
    glScalef(2.0,2.0,2.0);
    glTranslatef(10.0,30.0,0.0);
    printStrokedString(test1);
    glPopMatrix();
    glPushMatrix();
    glScalef(2.0,2.0,2.0);
    glTranslatef(10.0,13.0,0.0);
    printStrokedString(test2);
    glPopMatrix();
    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);
}

void keyboard(unsigned char key,int x,int y)
{
    switch (key){
        case ' ':
            glutPostRedisplay();
            break;
        case 27:
            exit(0);
            break;
    }
```

```
}

int main(int argc,char**argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB);
   glutInitWindowSize(440,120);
   glutCreateWindow(argv [0]);
   init();
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;
}
```

7.6 用显示列表管理状态变量

显示列表可以包含用于修改OpenGL状态变量值的函数。这些值在执行显示列表时被修改，就像这些函数是在立即模式下执行一样，并且这些修改结果在显示列表完成执行后仍然会保持。正如示例程序7-2和接下来的示例程序7-6所显示的那样，在显示列表执行时所修改的当前颜色和当前矩阵在显示列表完成执行之后仍然有效。

示例程序7-6 在显示列表执行之后状态修改的持久性

```
glNewList(listIndex,GL_COMPILE);
   glColor3f(1.0,0.0,0.0);
   glBegin(GL_POLYGON);
      glVertex2f(0.0,0.0);
      glVertex2f(1.0,0.0);
      glVertex2f(0.0,1.0);
   glEnd();
   glTranslatef(1.5,0.0,0.0);
glEndList();
```

现在如果调用下面的代码序列，在显示列表之后所绘制的那条直线使用红色作为当前颜色，并移动（1.5,0.0,0.0）：

```
glCallList(listIndex);
glBegin(GL_LINES);
   glVertex2f(2.0,-1.0);
   glVertex2f(1.0, 0.0);
glEnd();
```

有时候，我们希望保持对状态的修改。但也有一些时候，我们希望在执行一个显示列表时修改状态，并在执行显示列表之后恢复原来的状态。由于无法在显示列表中使用glGet*()，因此必须使用另一种方法来查询和恢复状态变量的值。

可以使用glPushAttrib()函数保存一组状态变量，并使用glPopAttrib()函数在需要的时候恢复这些值。为了保存和恢复当前矩阵，可以使用glPushMatrix()和glPopMatrix()，就像第3.6节描述的那样。这些压入和弹出函数可以合法地存储在显示列表中。为了在示例程序7-6中恢复状态变量，可以使用示例程序7-7所示的代码。

示例程序7-7 用显示列表恢复状态变量

```
glNewList(listIndex,GL_COMPILE);
   glPushMatrix();
   glPushAttrib(GL_CURRENT_BIT);
   glColor3f(1.0,0.0,0.0);
   glBegin(GL_POLYGON);
      glVertex2f(0.0,0.0);
      glVertex2f(1.0,0.0);
      glVertex2f(0.0,1.0);
   glEnd();
   glTranslatef(1.5,0.0,0.0);
   glPopAttrib();
   glPopMatrix();
glEndList();
```

如果使用了示例程序7-7中可以恢复状态变量的显示列表，那么示例程序7-8中的代码就绘制一条绿色的、未进行移动的直线。如果使用的是示例程序7-6中不保存和恢复状态变量的显示列表，这条直线就是用红色绘制的，并且10次移动（1.5,0.0,0.0）。

示例程序7-8 显示列表可能影响也可能不影响drawLine()

```
void display(void)
{
   GLint i;

   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(0.0,1.0,0.0);        /*set current color to green */
   for (i =0;i <10;i++)
   glCallList(listIndex);         /*display list called 10 times */
   drawLine();                    /*how and where does this line appear?*/
   glFlush();
}
```

封装模式修改

可以使用显示列表对成组的函数进行组织和存储，以修改各种模式或设置各种参数。当我们想从一组设置切换到另一组设置时，使用显示列表可能比直接调用函数具有更高的效率，因为这些设置可以用某种格式进行缓存，并与我们使用的图形系统的需求相匹配。

当需要从各种不同的光照、光照模型和材料参数设置中进行切换时，使用显示列表可能比使用立即模式更为高效。还可以把显示列表用于点画模式、雾参数以及裁剪平面方程式。一般而言，执行显示列表至少不会慢于直接调用相关的函数。但是，需要记住，在切换到使用显示列表时，它还是存在一些额外的开销。

示例程序7-9显示了如何使用显示列表在3种不同的直线点画模式中进行切换。首先，它调用glGenLists()函数为每种点画模式分配一个显示列表，并为每种模式创建一个显示列表。然后，就可以使用glCallList()函数从一种点画模式切换到另一种点画模式。

示例程序7-9 用于模式变更的显示列表

```
GLuint offset;
offset =glGenLists(3);

glNewList(offset,GL_COMPILE);
```

```
    glDisable(GL_LINE_STIPPLE);
glEndList();

glNewList(offset+1,GL_COMPILE);
    glEnable(GL_LINE_STIPPLE);
    glLineStipple(1,0x0F0F);
glEndList();

glNewList(offset+2,GL_COMPILE);
    glEnable(GL_LINE_STIPPLE);
    glLineStipple(1,0x1111);
glEndList();

#define drawOneLine(x1,y1,x2,y2)glBegin(GL_LINES);\
    glVertex2f((x1),(y1));glVertex2f((x2),(y2));glEnd();

glCallList(offset);
drawOneLine(50.0,125.0,350.0,125.0);

glCallList(offset+1);
drawOneLine(50.0,100.0,350.0,100.0);

glCallList(offset+2);
drawOneLine(50.0,75.0,350.0,75.0);
```

第 8 章　绘制像素、位图、字体和图像

本章目标

- 定位和绘制位图数据。
- 把像素数据（位图和图像）从帧缓冲区读取到内存，或者从内存读取到帧缓冲区。
- 把像素数据从一个颜色缓冲区复制到另一个颜色缓冲区，或者复制到同一个颜色缓冲区的其他位置。
- 图像写入到帧缓冲区时把它放大或缩小。
- 控制像素数据格式，并在帧缓冲区中存取数据时执行其他变换。
- 使用图像处理子集对像素进行处理。
- 使用缓冲区对象存储像素数据。

注意：在OpenGL 3.0中，本章所介绍的很多功能都废弃了，并且从OpenGL 3.1中删除了。它们被使用帧缓冲区对象的更强大的功能替代，第10章将详细介绍帧缓冲区对象。

到目前为止，本书所进行的讨论都集中于几何数据（点、直线和多边形）的渲染。除此之外，OpenGL还可以渲染两种重要类型的数据：

- 位图，一般用于表示字体中的字符。
- 图像数据，可以被扫描或计算。

位图和图像数据都采用矩形的像素数组的格式。它们之间的一个区别是位图的每个像素是由单个位的信息组成的，而图像的每个像素一般包含了好几段数据（例如，完整的红、绿、蓝和alpha成分）。另外，位图就像掩码一样，因为它们一般用于覆盖其他的图像。但是，图像数据既可以简单地进行覆盖，也可以采用某种方法与帧缓冲区中的数据进行混合。

本章描述如何把像素数据（位图和图像）从内存绘制到帧缓冲区，以及如何把像素数据从帧缓冲区读取到内存。本章还描述了如何把像素数据从一个位置复制到另一个位置，即可以从一个缓冲区复制到另一个缓冲区，也可以在同一个缓冲区中进行复制。

注意：OpenGL并不支持从文件读取像素和图像以及把像素和图像保存到文件中。

本章的内容分布于下面各节中：

- 位图和字体：描述用于定位和绘制位图数据的函数。这种数据可以用来描述字体。
- 图像：描述有关绘制、读取和复制像素数据的基本信息。
- 图像管线：描述当图像和位图数据从帧缓冲区读取以及写入到帧缓冲区时所执行的各种操作。
- 读取和绘制像素矩形：描述像素数据如何存储在内存中以及当像素数据移入移出内存时如何对它进行转换的所有相关细节。
- 使用缓冲区对象存取像素矩形数据：讨论使用服务器端的缓冲区对象更为高效地存储和读取像素数据。
- 提高像素绘图速度的技巧：列出在绘制像素矩形时有助于提高性能的一些技巧。
- 图像处理子集讨论当前的OpenGL扩展所提供的另一种像素处理操作。

在大多数情况下，需要执行的像素操作非常简单。因此，在编写程序时也许只需要用到本章前三节的内容。但是，像素操作也可能非常复杂，我们可以使用多种方法把像素数据存储在内存中。当把像素数据移入或移出帧缓冲区时，也可以对它们进行一些操作，这些细节是本章第四节的主题。我们很可能在实际需要这些信息时才会阅读这一节。第8.6节提供了一些有用的技巧，可以提高位图和图像的渲染性能。

OpenGL 1.2版本添加了封装一些像素格式（例如GL_UNSIGNED_BYTE_3_3_2和GL_UNSIGNED_INT_10_10_1_2），并提供了BGR和BGRA像素格式。

在1.2版本中，有一组称为图像处理子集（Imaging Subset）的图像操作成为ARB批准的扩展，其中包括颜色矩阵变换、颜色查找表、柱状图和一些新的混合操作（例如glBlendEquation()、glBlendColor()和一些常数混合模式）。在1.4版本中，图像处理子集中的混合操作被提升为OpenGL的核心特性。

1.4版本还引入了GL_SRC_COLOR和GL_ONE_MINUS_SRC_COLOR作为源混合函数，并引入了GL_DST_COLOR和GL_ONE_MINUS_DST_COLOR作为目标混合函数。1.4版本还增加了glWindowPos*()函数，它可以用窗口坐标指定光栅位置。

OpenGL 3.0引入了本章介绍的众多额外的数据类型和像素格式。很多这样的格式都像纹理格式（参见第9章了解详细内容）和渲染缓冲区格式（它们是帧缓冲区对象的新功能的一部分，参见第10章）一样很有用。

8.1 位图和字体

位图是由1和0组成的矩形数组，作为窗口中一个矩形区域的绘图掩码。假设我们正在绘制一幅位图，并且当前的光栅颜色是红色。在位图中为1的地方，帧缓冲区中的对应像素就用红色像素（或者根据实际生效的片断操作，与红色像素进行混合）代替（参见第10.2节）。对于位图中为0的地方，就不会生成片断，像素的内容不受影响。位图的最常见用途就是在屏幕上绘制字符。

OpenGL对于字符串的绘制以及字体的操纵只提供了最低层次的支持。glRasterPos*()（或glWindowPos*()）和glBitmap()函数可以在屏幕上定位和绘制位图。另外，通过显示列表机制，可以使用一系列的字符代码建立对应位图系列的索引，以表示这些字符（关于显示列表的更多信息，请参阅第7章）。我们必须自己编写函数，以操纵位图、字体和字符串。

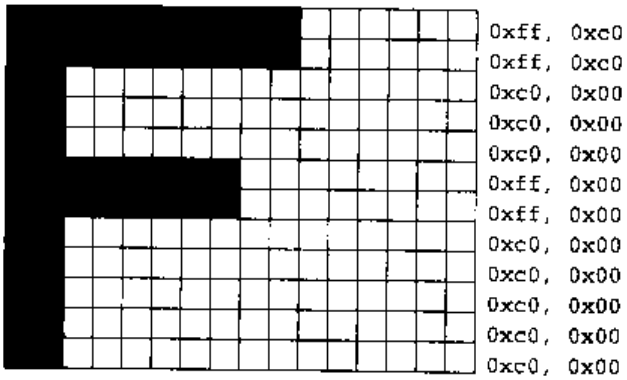


图8-1 位图F以及它的数据

请观察示例程序8-1，它在屏幕上绘制了3个F。图8-1显示了位图形式的F以及与它对应的位图数据。

示例程序8-1 绘制位图字体：drawf.c

```
GLubyte rasters [24] ={
    0xc0,0x00,0xc0,0x00,0xc0,0x00,0xc0,0x00,0xc0,0x00,
    0xff,0x00,0xff,0x00,0xc0,0x00,0xc0,0x00,0xc0,0x00,
    0xff,0xc0,0xff,0xc0};

void init(void)
{
```

```
    glPixelStorei(GL_UNPACK_ALIGNMENT,1);
    glClearColor(0.0,0.0,0.0,0.0);
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0,1.0,1.0);
    glRasterPos2i(20,20);
    glBitmap(10,12,0.0,0.0,11.0,0.0,rasters);
    glBitmap(10,12,0.0,0.0,11.0,0.0,rasters);
    glBitmap(10,12,0.0,0.0,11.0,0.0,rasters);
    glFlush();
}
```

在图8-1中，注意F字符可见部分的最大宽度是10位。位图数据总是成块存储的（8位的倍数），但是实际位图的最宽部分并不一定是8的倍数。组成位图的位是从左下角开始绘制的，首先绘制的是最底部一行，然后是倒数第二行，接下来以此类推。我们可以从代码中发现，位图在内存中是按照这样的顺序存储的：光栅数组首先存储的是0xc0、0x00、0xc0、0x00，表示最底部的两行，然后一直到0xff、0xc0、0xff、0xc0，也就是F的最顶部两行。

在这个示例程序中，我们感兴趣的函数是glRasterPos2i()和glBitmap()，它们将在接下来的几节中详细讨论。现在，可以忽略对glPixelStorei()函数的调用，它描述了位图数据是如何存储在计算机内存中的（关于这个函数的更多信息，请参阅第8.3.2节）。

8.1.1 当前光栅位置

当前光栅位置就是开始绘制下一幅位图（或图像）的屏幕位置。在前面的F例子中，光栅位置是通过坐标（20，20）参数调用glRasterPos*()函数设置的，它就是屏幕左下角开始绘制F的地方：

```
glRasterPos2i(20, 20);
```

兼容性扩展
glRasterPos

void **glRasterPos**{234}{sifd}(*TYPE x*, *TYPE y*, *TYPE z*, *TYPE w*);

void **glRasterPos**{234}{sifd}v(const *TYPE *coords*);

设置当前的光栅位置。*x*、*y*、*z*和*w*参数指定了光栅位置的坐标。如果使用的是这个函数的向量形式，coords参数包含了光栅位置的各个光标。如果使用的是**glRasterPos2*()**，*z*被隐式设置为0，*w*被隐式设置为1。类似地，如果使用的是**glRasterPos3*()**，*w*被设置为1。

光栅位置的坐标被变换为屏幕坐标，就像以它为参数调用了glVertex*()函数一样（也就是说，用模型视图和投影矩阵进行变换）。在变换之后，它或者定义了视口中一个有效的点，或者由于位于视景体的外部而裁剪掉。如果经过变换的点被裁剪掉，当前的光栅位置就是无效的。

在1.4版本之前，如果我们想根据窗口坐标（屏幕坐标）来指定光栅位置，必须设置模型视图和投影矩阵来实现简单的2D渲染。我们使用类似下面这样的代码，其中width和height表示视口的大小（以像素为单位）：

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, (GLfloat) width, 0.0, (GLfloat) height);
```

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

在1.4版本中，glWindowsPos*()函数是作为glRasterPos*()函数的替代品引入的。glWindowsPos*()用窗口坐标指定当前光栅位置，而不必把它的x和y坐标通过模型视图和投影矩阵进行变换，也不会被裁剪出视口区域。glWindowsPos*()使我们更容易混合使用2D文本和3D图形，而不必在各种变换状态之间反复切换。

兼容性扩展
glWindowPos

void **glWindowPos**{23}{sifd}(*TYPE x*, *TYPE y*, *TYPE z*);

void **glWindowPos**{23}{sifd}v(const *TYPE *coords*);

设置当前光栅位置，以窗口坐标x和y为参数，不必进行矩阵变换、裁剪或纹理坐标生成。z值被变换为（并截取为）由**glDepthRange()**函数所设置的当前近侧平面值和远侧平面值。如果使用的是向量形式，coords数组包含了光栅位置的各个坐标。如果使用的是**glWindowPos2*()**，z被隐式设置为0。

为了获取当前的光栅位置（不管由glRasterPos*()设置还是由glWindowsPos*()设置），可以用GL_CURRENT_RASTER_POSITION为第一个参数调用查询函数glGetFloatv()，它的第二个参数应该是一个指针，指向保存了（x，y，z，w）值（浮点数形式）的数组。以GL_CURRENT_RASTER_POSITION_VALID为第一个参数调用glGetBooleanv()可以确定当前的光栅位置是否有效。

8.1.2 绘制位图

在设置了光栅位置之后，可以使用glBitmap()函数来绘制数据。

兼容性扩展
glBitmap

void **glBitmap**(GLsizei *width*, GLsizei *height*,
GLfloat x_{bo}, GLfloat y_{bo},
GLfloat x_{bi}, GLfloat y_{bi},
const GLubyte *bitmap);

绘制由bitmap指定的位图，*bitmap*是一个指向位图图像的指针。位图的原点是当前光栅位置。如果当前光栅位置无效，这个函数就不会绘制任何东西，而且当前光栅位置继续保持无效。width和height参数表示位图的宽度和高度（以像素为单位）。*width*并不需要是8的倍数，尽管数据是以8位的无符号字符的形式存储的。在F例子中，第10位之后的垃圾位的数据无关紧要，因为**glBitmap()**在调用时指定了宽度为10，因此每行只有10个位被渲染。x_{bo}和y_{bo}定义了位图的原点，它是根据当前光栅位置确定的，正值在当前光栅位置的上面和右边，负值在当前光栅位置的左边和下面。x_{bi}和y_{bi}表示位图光栅化之后光栅位置的x增加值和y增加值（如图8-2所示）。

允许任意设置位图的原点，使我们可以很方便地把字符扩展到原点的下面（一般用于具有下伸部分的字符，如g、j和y），也可以把它扩展到原点的左边（用于各种花饰字符以及倾斜的字符）。

在绘制了位图之后，当前光栅位置在x和y方向分别增加x_{bi}和y_{bi}。如果只想推进当前光栅位置，而不想绘制任何东西，可以调用glBitmap()函数，把bitmap参数设置为NULL，并把width和height参

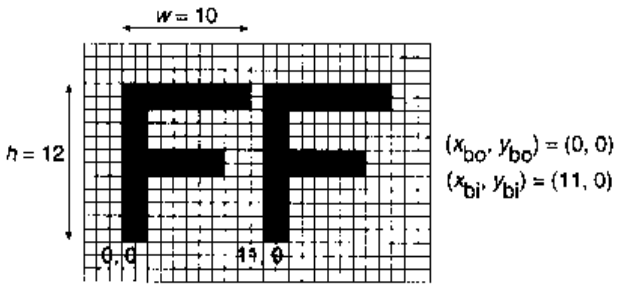


图8-2 位图和它相关的参数

数设置为0。对于标准的拉丁字体，y_{bi}一般为0.0，x_{bi}一般为正值（因为连续的字符是从左向右绘制的）。对于从右向左绘制字符的希伯来字符，x_{bi}值一般为负。对于那些以垂直方向绘制字符的字体，x_{bi}为0.0，y_{bi}则为非零值。在图8-2中，每次绘制F时，当前光栅位置前进11个像素，这样连续的字符之间有1个像素的间隙。

由于x_{bo}、y_{bo}、x_{bi}和y_{bi}都是浮点值，因此字符之间的距离并不一定是整数个像素。在实际绘制字符的时候，总是以实际的像素为边界，但是当前光栅位置总是浮点形式，因此每个像素在绘制时总是尽可能地靠近它所属的位置。例如，如果F例子中的代码进行了修改，x_{bi}是11.5而不是12，并且绘制了多个字符，那么字符之间的间距将交替地使用1和2个像素，尽可能地模拟程序所要求的1.5像素的间距。

注意：位图字体无法进行旋转，因为位图在绘制时总是与帧缓冲区的x和y轴对齐。另外，位图也无法进行缩放。

8.1.3 选择位图的颜色

在绘制几何图形时，可以用glColor*()和glIndex*()函数设置当前颜色或当前颜色索引。这两个函数还可以用于设置不同的状态变量GL_CURRENT_RASTER_COLOR和GL_CURRENT_RASTER_INDEX，用位图进行渲染。光栅颜色状态变量是在调用glRasterPos*()时根据当前颜色设置的，这很容易误导人。在下面这段代码中，位图的当前颜色是什么？

```
glColor3f(1.0, 1.0, 1.0);  /* white */
glRasterPos3fv(position);
glColor3f(1.0, 0.0, 0.0);  /* red  */
glBitmap(....);
```

位图是白色的，是不是对此感到奇怪？在这段代码中，当程序调用glRasterPos3fv()时，GL_CURRENT_RASTER_COLOR被设置为白色。第二个glColor3f()调用修改了GL_CURRENT_COLOR的值，用于以后的几何图形渲染，但是用于渲染位图的颜色并没有发生变化。

为了获取当前的光栅颜色或光栅颜色索引，可以用GL_CURRENT_RASTER_COLOR或GL_CURRENT_RASTER_INDEX为第一个参数调用glGetFloatv()或glGetIntegerv()这两个查询函数。

8.1.4 字体和显示列表

在第7章中，我们讨论了显示列表的基本概念。但是，有一些显示列表管理函数与字符串的绘制具有特殊的相关性。在阅读本节时，读者要记住本节所展示的思路同样适合于用位图数据绘制的字体以及用几何图形（点、直线和多边形）绘制的字体（第7.5节提供了一个几何图形字体的例子）。

一种字体一般由一组字符组成，其中每个字符都有一个标识码（通常是ASCII码）和一个绘制方法。对于标准的ASCII字符集，大写字母A用整数65表示，B是66，接下来以此类推。字符串“DAB”可以用3个索引值68、65和66来表示。按照这种最简单的方法，我们可以用显示列表65来绘制A，用显示列表66用来绘制B，接下来以此类推。为了绘制字符串68，65和66，只要执行对应的显示列表就可以了。

可以按照这种方法来调用glCallLists()：

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

第一个参数n表示需要绘制的字符数量，type通常是GL_BYTE，lists是一个字符代码数组。

由于许多应用程序需要以多种字体和字号来绘制字符串，上面这种最简单的方法就不是很实用。我们可以想到的一种方法是不管什么字体都用65来表示A。可以强制把字体1的A、B和C编码为1065、

1066和1067，把字体2的A、B和C编码为2065、2066和2067。但是，8位的字节无法表示大于256的数。一种更好的解决方案是在选择显示列表前向字符串中的每个字母增加一个偏移值。在此例中，字体1用1065、1066和1067分别表示A、B和C。而在字体2中，它们可能分别用2065、2066和2067表示。为了用字体1绘制字符，可以把偏移值设置为1000，并绘制显示列表65、66和67。为了用字体2绘制同一个字符串，可以把偏移值设置为2000，并绘制这些相同的显示列表。

为了设置偏移值，可以使用glListBase()函数。在前面这个例子中，可以用1000或2000作为它的唯一参数。接下来所需要的是一系列连续的未使用显示列表标识符，可以通过调用glGenLists()来获取它们：

```
GLuint glGenLists(GLsizei range);
```

这个函数返回range个显示列表标识符。它所返回的显示列表标识符都标记为“已使用”（即使它们是空的），因此后续的glGenLists()调用不会返回这些标识符（除非在此之前显式地删除了它们）。因此，如果使用4为参数，并且glGenLists()返回81，就可以使用显示列表标识符81、82、83和84来表示这些字符。如果glGenLists()无法找到一块满足请求长度的未使用标识符，它就返回0（注意，只需要调用glDeleteLists()函数1次就可以删除与一种字体相关联的所有显示列表）。

大多数欧美字体的字符数量并不多（小于256），因此很容易用不同的单字节代码来表示每个字符。但是，亚洲字体的字符数量可能非常庞大，因此用单字节来表示每个字符的做法是行不通的。OpenGL允许我们用1字节、2字节、3字节甚至4字节的字符来组成字符串，这是通过glCallLists()函数中的type参数实现的。这个参数可以是下列值之一：

GL_BYTE	GL_UNSIGNED_BYTE
GL_SHORT	GL_UNSIGNED_SHORT
GL_INT	GL_UNSIGNED_INT
GL_FLOAT	GL_2_BYTES
GL_3_BYTES	GL_4_BYTES

关于这些值的更多信息，请参阅第7.5节。

8.1.5 定义和使用一种完整的字体

glBitmap()函数以及前一节描述的显示列表机制使我们可以很方便地定义光栅字体。在示例程序8-2中，我们定义了一种ASCII字体的大写字母。在这个程序中，每个字符具有相同的宽度（但其他字体可能并不是这样）。在定义了这些字符之后，这个程序就打印出“THE QUICK BROWN FOX JUMPS OVER A LAZY DOG”这条信息。

示例程序8-2的代码类似F例子，只是每个字符的位图存储在它自己的显示列表中。当显示列表标识符与glGenLists()所返回的偏移值进行组合之后，它就相当于这个字符的ASCII码。

示例程序8-2 绘制一种完整的字体：font.c

```
GLubyte space [] =
   {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};
GLubyte letters [][13] ={
   {0x00,0x00,0xc3,0xc3,0xc3,0xc3,0xff,0xc3,0xc3,0xc3,0x66,0x3c,0x18},
   {0x00,0x00,0xfe,0xc7,0xc3,0xc3,0xc7,0xfe,0xc7,0xc3,0xc3,0xc7,0xfe},
   {0x00,0x00,0x7e,0xe7,0xc0,0xc0,0xc0,0xc0,0xc0,0xc0,0xc0,0xe7,0x7e},
   {0x00,0x00,0xfc,0xce,0xc7,0xc3,0xc3,0xc3,0xc3,0xc3,0xc7,0xce,0xfc},
   {0x00,0x00,0xff,0xc0,0xc0,0xc0,0xc0,0xfc,0xc0,0xc0,0xc0,0xc0,0xff},
```

```
    {0x00,0x00,0xc0,0xc0,0xc0,0xc0,0xc0,0xc0,0xfc,0xc0,0xc0,0xc0,0xff},
    {0x00,0x00,0x7e,0xe7,0xc3,0xc3,0xcf,0xc0,0xc0,0xc0,0xc0,0xe7,0x7e},
    {0x00,0x00,0xc3,0xc3,0xc3,0xc3,0xc3,0xff,0xc3,0xc3,0xc3,0xc3,0xc3},
    {0x00,0x00,0x7e,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x7e},
    {0x00,0x00,0x7c,0xee,0xc6,0x06,0x06,0x06,0x06,0x06,0x06,0x06,0x06},
    {0x00,0x00,0xc3,0xc6,0xcc,0xd8,0xf0,0xe0,0xf0,0xd8,0xcc,0xc6,0xc3},
    {0x00,0x00,0xff,0xc0,0xc0,0xc0,0xc0,0xc0,0xc0,0xc0,0xc0,0xc0,0xc0},
    {0x00,0x00,0xc3,0xc3,0xc3,0xc3,0xc3,0xc3,0xdb,0xff,0xff,0xe7,0xc3},
    {0x00,0x00,0xc7,0xc7,0xcf,0xcf,0xdf,0xdb,0xfb,0xf3,0xf3,0xe3,0xe3},
    {0x00,0x00,0x7e,0xe7,0xc3,0xc3,0xc3,0xc3,0xc3,0xc3,0xc3,0xe7,0x7e},
    {0x00,0x00,0xc0,0xc0,0xc0,0xc0,0xc0,0xfe,0xc7,0xc3,0xc3,0xc7,0xfe},
    {0x00,0x00,0x3f,0x6e,0xdf,0xdb,0xc3,0xc3,0xc3,0xc3,0xc3,0x66,0x3c},
    {0x00,0x00,0xc3,0xc6,0xcc,0xd8,0xf0,0xfe,0xc7,0xc3,0xc3,0xc7,0xfe},
    {0x00,0x00,0x7e,0xe7,0x03,0x03,0x07,0x7e,0xe0,0xc0,0xc0,0xe7,0x7e},
    {0x00,0x00,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0xff},
    {0x00,0x00,0x7e,0xe7,0xc3,0xc3,0xc3,0xc3,0xc3,0xc3,0xc3,0xc3,0xc3},
    {0x00,0x00,0x18,0x3c,0x3c,0x66,0x66,0xc3,0xc3,0xc3,0xc3,0xc3,0xc3},
    {0x00,0x00,0xc3,0xe7,0xff,0xff,0xdb,0xdb,0xc3,0xc3,0xc3,0xc3,0xc3},
    {0x00,0x00,0xc3,0x66,0x66,0x3c,0x3c,0x18,0x3c,0x3c,0x66,0x66,0xc3},
    {0x00,0x00,0x18,0x18,0x18,0x18,0x18,0x18,0x3c,0x3c,0x66,0x66,0xc3},
    {0x00,0x00,0xff,0xc0,0xc0,0x60,0x30,0x7e,0x0c,0x06,0x03,0x03,0xff}
};
GLuint fontOffset;
void makeRasterFont(void)
{
   GLuint i,j;
   glPixelStorei(GL_UNPACK_ALIGNMENT,1);
   fontOffset =glGenLists(128);
   for (i =0,j ='A ';i <26;i++,j++){
      glNewList(fontOffset +j,GL_COMPILE);
      glBitmap(8,13,0.0,2.0,10.0,0.0,letters [i]);
      glEndList();
   }
   glNewList(fontOffset +' ',GL_COMPILE);
   glBitmap(8,13,0.0,2.0,10.0,0.0,space);
   glEndList();
}
void init(void)
{
   glShadeModel(GL_FLAT);
   makeRasterFont();
}

void printString(char *s)
{
   glPushAttrib(GL_LIST_BIT);
   glListBase(fontOffset);
   glCallLists(strlen(s),GL_UNSIGNED_BYTE,(GLubyte *)s);
   glPopAttrib();
}

/*Everything above this line could be in a library
```

```
*that defines a font.To make it work,you've got
*to call makeRasterFont()before you start making
*calls to printString().
*/
void display(void)
{
   GLfloat white [3] ={1.0,1.0,1.0 };

   glClear(GL_COLOR_BUFFER_BIT);
   glColor3fv(white);

   glRasterPos2i(20,60);
   printString("THE QUICK BROWN FOX JUMPS ");
   glRasterPos2i(20,40);
   printString("OVER A LAZY DOG ");
   glFlush();
}
```

8.2 图像

图像与位图相似，但是屏幕矩形区域中的每个像素并不是由1个位表示的。在图像中，每个像素可以包含更多的信息。例如，图像的每个像素可以存储完整的颜色（R、G、B、A）。图像的来源很多，例如：

- 用扫描仪进行数字处理后的照片。
- 图形程序首先使用图形硬件在屏幕上生成图像，然后逐像素地读取它。
- 软件程序在内存中逐像素地生成图像。

在正常情况下，我们所想到的图像就是来自颜色缓冲区的图片。但是，也可以从深度缓冲区和模板缓冲区读取（或写入）矩形区域的像素数据（参见第10章对这些缓冲区的解释）。

除了简单地显示在屏幕上之外，图像还可以用于纹理贴图。在这种情况下，它们实际上是粘贴到按照正常方式在屏幕上所渲染的多边形的表面（关于这个技巧的更多信息，请参阅第9章）。

读取、写入和复制像素数据

OpenGL提供了3个基本的函数来操纵图像数据：

- glReadPixels()：从帧缓冲区读取一个矩形像素数组，并把数据保存在内存中。
- glDrawPixels()：把内存中保存的一个矩形像素数组写入到帧缓冲区中由glRasterPos*()指定的当前位置。
- glCopyPixels()：把一个矩形像素数组从帧缓冲区的一个部分复制到另一部分。这个函数的行为类似于在调用glReadPixels()之后再调用glDrawPixels()，但是数据并不会写入到内存中。

在上述这些函数中，像素数据处理操作的顺序如图8-3所示。

图8-3展示了基本的像素处理流程。glRasterPos*()函数指定了glDrawPixels()函数和glCopyPixels()函数所使用的当前光栅位置，它产生的坐标通过几何处理管线进行变换。glDrawPixels()函数和glCopyPixels()函数都受到光栅化和片断操作的影响（但是，在绘制或复制像素矩形时，启用雾或纹理效果是毫无意义的）。

但是，这些操作还是存在许多复杂之处。这是因为存在多种帧缓冲区数据，并且在计算机内存中存储像素信息的方法也有很多种。此外，在读取、写入和复制操作时还可能执行各种不同的数据转换。

由于存在这些可能性，所以也就存在许多不同的操作模式。如果程序所完成的任务就是把图像复制到屏幕上或者把它们临时复制到内存中（以后再把它们从内存中复制出来），我们可以忽略绝大多数操作模式。但是，如果希望修改内存中的数据（例如，我们存储的图像格式和窗口所要求的图像格式可能并不相同，或者我们想把图像数据保存到文件中，以后再把它恢复到另一个任务中或恢复到另一台图像功能显著不同的计算机中），就必须理解这些模式。

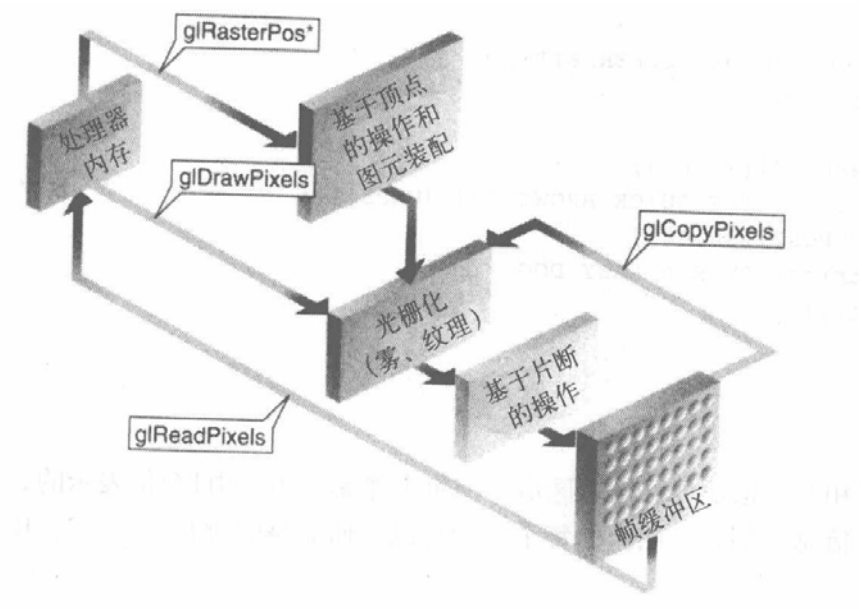


图8-3 像素数据处理流程简图

本节的剩余部分将详细描述这几个基本函数。下一节讨论组成图像管线的一系列图像操作（像素存储模式、像素传输操作和像素映射操作）的细节。

把像素数据从缓冲区读取到内存

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

从帧缓冲区中的一个矩形区域读取像素数据，这个矩形区域的左下角位置是窗口坐标（x, y），它的宽度和高度分别是width和height。读取的像素数据保存在pixels所指向的数组中。format表示这个函数读取的像素数据元素的类型（索引值或R、G、B、A成分值，见表8-1），type表示每个元素的数据类型（见表8-2）。

glReadPixels()可以产生一些OpenGL错误：如果*format*设置为GL_DEPTH并且没有深度缓冲区；如果format是GL_STENCIL并且没有模板缓冲区；如果*format*设置为GL_DEPTH_STENCIL，并且没有与帧缓冲区相关联的深度和模板缓冲区；如果type不是GL_UNSIGNED_INT_24_8，也不是GL_FLOAT_32_UNSIGNED_INT_24_8_REV，将会产生一个GL_INVALID_OPERATION错误，并且设置GL_INVALID_ENUM。

表8-1 glReadPixels()或glDrawPixels()所使用的像素格式

格式常量	像素格式
GL_COLOR_INDEX	1个颜色索引值
GL_RG或GL_RG_INTEGER	1个红色成分，后面跟着1个绿色成分
GL_RGB或GL_RGB_INTEGER	依次是1个红色成分、1个绿色成分和1个蓝色成分
GL_RGBA或GL_RGBA_INTEGER	依次是1个红色成分、1个绿色成分、1个蓝色成分和1个alpha成分

（续）

格式常量	像素格式
GL_BGR或GL_BGR_INTEGER	依次是1个蓝色成分、1个绿色成分和1个红色成分
GL_BGRA或GL_BGRA_INTEGER	依次是1个蓝色成分、1个绿色成分、1个红色成分和1个alpha成分
GL_RED或GL_RED_INTEGER	1个红色成分
GL_GREEN 或GL_GREEN_INTEGER	1个绿色成分
GL_BLUE或_INTEGER	1个蓝色成分
GL_ALPHA或GL_ALPHA_INTEGER	1个alpha成分
GL_LUMINANCE	1个亮度成分
GL_LUMINANCE_ALPHA	1个亮度成分和1个alpha成分
GL_STENCIL_INDEX	1个模板索引值
GL_DEPTH_COMPONENT	1个深度成分
GL_DEPTH_STENCIL	深度成分和模板成分的组合

表8-2 glReadPixels()或glDrawPixels()所使用的数据类型

类型常量	数据类型
GL_UNSIGNED_BTYE	无符号8位整数
GL_ BTYE	有符号8位整数
GL_BITMAP	无符号8位整数中的各个位，使用和glBitmap()相同的格式
GL_UNSIGNED_SHORT	无符号16位整数
GL_SHORT	有符号16位整数
GL_UNSIGNED_INT	无符号32位整数
GL_INT	有符号32位整数
GL_FLOAT	单精度浮点数
GL_HALF_FLOAT	一个16位浮点数值
GL_UNSIGNED_BTYE _3_3_2	包装为无符号8位整数
GL_UNSIGNED_BTYE _2_3_3_REV	包装为无符号8位整数
GL_UNSIGNED_SHORT_5_6_5	包装为无符号16位整数
GL_UNSIGNED_SHORT_5_6_5_REV	包装为无符号16位整数
GL_UNSIGNED_SHORT_4_4_4_4	包装为无符号16位整数
GL_UNSIGNED_SHORT_4_4_4_4_REV	包装为无符号16位整数
GL_UNSIGNED_SHORT_5_5_5_1	包装为无符号16位整数
GL_UNSIGNED_SHORT_1_5_5_5_REV	包装为无符号16位整数
GL_UNSIGNED_ INT_8_8_8_8	包装为无符号32位整数
GL_UNSIGNED_ INT_8_8_8_8_REV	包装为无符号32位整数
GL_UNSIGNED_ INT_10_10_10_2	包装为无符号32位整数
GL_UNSIGNED_ INT_2_10_10_10_REV	包装为无符号32位整数
GL_UNSIGNED_INT_24_8	包装为无符号32位整数（只能和GL_DEPTH_STENCIL的一个格式一起使用）
GL_UNSIGNED_INT_10F_11F_10F_11F_REV	包装为无符号32位整数的10位和11位浮点数
GL_UNSIGNED_INT_5_9_9_9_REV	共享其5位指数值的3个9位浮点值，包装为无符号32位整数
GL_FLOAT_32_UNSIGNED_INT_24_8_REV	包装为2个32位值的深度和模板值：32位浮点数的深度值和一个8位的无符号模板值（中间的24位未使用）

如果使用glReadPixels()来获取RGBA或颜色索引信息，可能需要分清自己访问的是哪个缓冲区。

例如，如果拥有一个双缓冲的窗口，就需要指定从前缓冲区还是从后缓冲区读取数据。为了控制当前读取的源缓冲区，可以调用glReadBuffer()（请参阅第10.1.3节）。

注意：GL_*_REV像素格式专门用于Microsoft的Windows操作系统。

记住，按照指定的格式，可以读取或写入1~4个元素。例如，如果格式是GL_RGBA，并且读取的是个32位的整数（即type为GL_UNSIGNED_INT或GL_INT），那么每个被读取的像素都要求16字节的存储空间（4个元素×4个字节/每个成分）。

图像的每个元素都存储在内存中，见表8-2。如果元素表示一个连续的值，如红、绿、蓝或亮度成分，每个值进行缩放，以便用适当数量的位来表示。例如，假设红色成分一开始被指定为一个位于0.0~1.0之间的浮点值。如果它需要被包装为1个无符号字节，它就只保留8位精度，即使帧缓冲区为红色成分分配了更多的位。GL_UNSIGNED_SHORT和GL_UNSIGNED_INT分别提供了16位和32位的精度。GL_BYTE、GL_SHORT和GL_INT的有符号版本分别具有7、15和31位的精度，因为负值一般并不使用。

如果元素是索引值（例如，是一个颜色索引值或模板索引值），并且类型不是GL_FLOAT，这个值就简单地对类型中的可用位进行屏蔽。有符号版本（GL_BYTE、GL_SHORT和GL_INT）的掩码位要少1位。例如，如果一个颜色索引值存储在一个有符号的8位整数中，它首先根据0x7f设置掩码。如果类型是GL_FLOAT，这个索引值就简单地转换为一个单精度浮点数（例如，索引值17被转换为浮点数17.0）。

对于包装数据类型（那些以GL_UNSIGNED_BYTE_*、GL_UNSIGNED_SHORT_*或GL_UNSIGNED_INT_*开头的常量），每个像素的所有颜色成分压缩成一种无符号数据类型：字节、短整型或标准整型。每种类型的合法格式均有限制，见表8-3。如果在一种像素包装数据类型中使用了一种非法的像素格式，就会产生一个GL_INVALID_OPERATION错误。

表8-3 包装数据类型的有效像素格式

包装类型常量	有效像素格式
GL_UNSIGNED_BYTE_3_3_2	GL_RGB
GL_UNSIGNED_BYTE_2_3_3_REV	GL_RGB
GL_UNSIGNED_SHORT_5_6_5	GL_RGB
GL_UNSIGNED_ SHORT_5_6_5_REV	GL_RGB
GL_UNSIGNED_ SHORT_4_4_4_4	GL_RGBA、GL_BGRA
GL_UNSIGNED_ SHORT_4_4_4_4_REV	GL_RGBA、GL_BGRA
GL_UNSIGNED_ SHORT_5_5_5_1	GL_RGBA、GL_BGRA
GL_UNSIGNED_ SHORT_1_5_5_5_REV	GL_RGBA、GL_BGRA
GL_UNSIGNED_INT_8_8_8_8	GL_RGBA、GL_BGRA
GL_UNSIGNED_ INT_8_8_8_8_REV	GL_RGBA、GL_BGRA
GL_UNSIGNED_ INT_10_10_10_2	GL_RGBA、GL_BGRA
GL_UNSIGNED_ INT_2_10_10_10_REV	GL_RGBA、GL_BGRA
GL_UNSIGNED_INT_24_8	GL_DEPTH_STENCIL
GL_UNSIGNED_INT_10F_11F_11F	GL_RGB
GL_UNSIGNED_INT_5_9_9_9_REV	GL_RGB
Gl_FLOAT_32_UNSIGNED_INT_24_8_REV	GL_DEPTH_STENCIL

在经过包装的像素数据中，位段位置的颜色值顺序是由像素格式以及类型常量是否包含_REV后

缀所决定的。如果没有_REV后缀，颜色成分在正常情况下被赋值为占据最高位置的第一个颜色成分。如果有_REV后缀，颜色成分的包装顺序是相反的，它被赋值为占据最低位置的第一个颜色成分。

为了说明这个概念，图8-4显示了GL_UNSIGNED_BYTE_3_3_2、GL_UNSIGNED_BYTE_2_3_3_REV和GL_UNSIGNED_SHORT_4_4_4_4（和_REV）数据类型与RGBA/BGRA像素格式的4种有效组合。其他14种包装像素数据类型与像素格式的有效组合也采用相似的模式。

每个颜色成分的最高位总是包装到最高有效位。单个成分的存储并不受任何像素存储模式的影响，尽管整个像素的存储可能受到字节交换模式的影响（关于字节交换的更多信息，请参阅第8.3.2节）。

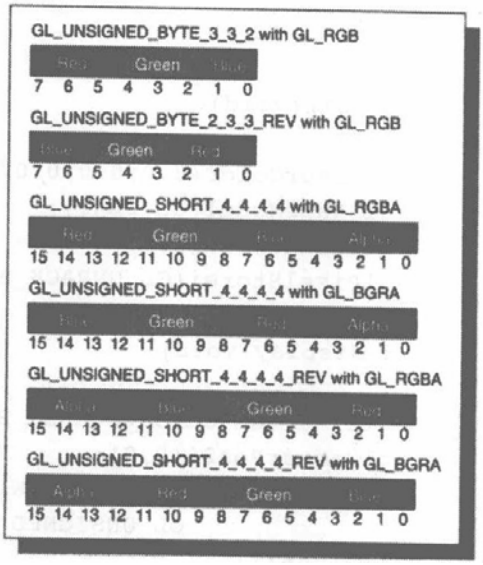


图8-4 一些数据类型和像素格式的成分顺序

把像素数据从内存写入到帧缓冲区

兼容性扩展

glDrawPixels

void **glDrawPixels**(GLsizei *width*, GLsizei *height*, GLenum *format*, GLenum *type*, const GLvoid **pixels*);

绘制一个宽度和高度分别为width和height的矩形像素数据。这个像素矩形的左下角就是当前光栅位置。format与type参数与glReadPixels()函数中的同名参数具有相同的含义（format和type参数的合法值见表8-1和表8-2）。pixels所指向的数组包含了被绘制的像素数据。如果当前光栅位置是无效的，这个函数就不会绘制任何东西，并且当前光栅位置仍然保持为无效。

如果format是GL_STENCIL并且没有和帧缓冲区相关联的模板缓冲区；或者类似的，format是GL_DEPTH_STENCIL，并且没有深度缓冲区和模板缓冲区，将会产生一个GL_INVALID_OPERATION。

示例程序8-3是一个程序的一部分，它使用glDrawPixels()函数在窗口的左下角绘制一个像素矩形。makeCheckImage()函数创建了一个64 × 64的RGB数组，表示一幅黑白交替的棋盘图像。glRasterPos2i(0, 0)用于设置图像左下角的位置。此刻，我们可以忽略glPixelStorei()函数。

示例程序8-3 使用glDrawPixels()：image.c

```
#define checkImageWidth 64
#define checkImageHeight 64
GLubyte checkImage [checkImageHeight][checkImageWidth][3];

void makeCheckImage(void)
{
   int i,j,c;

   for (i =0;i <checkImageHeight;i++){
      for (j =0;j <checkImageWidth;j++){
         c =(((i&0x8)==0)^((j&0x8)==0))*255;
         checkImage [i][j][0] =(GLubyte)c;
         checkImage [i][j][1] =(GLubyte)c;
```

```
            checkImage [i][j][2] =(GLubyte)c;
        }
    }
}
void init(void)
{
    glClearColor(0.0,0.0,0.0,0.0);
    glShadeModel(GL_FLAT);
    makeCheckImage();
    glPixelStorei(GL_UNPACK_ALIGNMENT,1);
}
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glRasterPos2i(0,0);
    glDrawPixels(checkImageWidth,checkImageHeight,GL_RGB,
                 GL_UNSIGNED_BYTE,checkImage);
    glFlush();
}
```

当使用glDrawPixels()函数写入RGBA或颜色索引信息时，可能需要使用glDrawBuffer()函数（和glReadBuffer()函数一起使用）控制当前绘图缓冲区。第10.1.3节将描述这个函数和glReadBuffer()函数。

当format参数设置为GL_STENCIL或GL_DEPTH_STENCIL的时候，glDrawPixels()的工作方式略有不同。在模板缓冲区受到影响的情况下，将要写的窗口位置更新了它们的模板值（根据当前的前模板掩码），但是颜色缓冲区中的值不会产生或受到影响（例如，不会产生片段；如果绑定了一个片段着色器，在绘制操作中不会对该位置执行它）。同样，如果还提供了一个深度，并且可以写入深度缓冲区（例如，深度掩码是GL_TRUE），那么深度值会直接写入到深度缓冲区中，然后通过设置深度测试来保持该值不受影响。

在帧缓冲区内部复制像素数据

兼容性扩展
glCopyPixels

void **glCopyPixels**(GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*,
GLenum *buffer*);

从帧缓冲区中的一个矩形区域（它的左下角位于(x, y)，它的宽度和高度分别为width和height）复制像素数据。数据被复制到帧缓冲区中的一个新位置，它的左下角是当前光栅位置。buffer是GL_COLOR、GL_STENCIL或GL_DEPTH，指定了这个函数所使用的帧缓冲区。**glCopyPixels()**函数的行为类似于在调用**glReadPixels()**之后再调用**glDrawPixels()**，其中buffer和format参数所使用的变换如下：

- 如果buffer是GL_DEPTH或GL_STENCIL，那就分别使用GL_DEPTH_COMPONENT或GL_STENCIL_INDEX。如果buffer是GL_DEPTH_STENCIL，并且这些缓冲区中的任何一个不存在的话（例如，没有和帧缓冲区相关的模板缓冲区），那么，就好像针对缺失的缓冲区的数据的通道为0。
- 如果buffer是GL_COLOR，根据系统使用的是RGBA模式还是颜色索引模式，分别使用GL_RGBA或GL_COLOR_INDEX。

注意，glCopyPixels()函数并不需要format或data参数，因为数据绝不会复制到内存中。glCopyPixels()中用于读取的源缓冲区以及目标缓冲区分别是由glReadBuffer()函数和glDrawBuffer()函数指定的。示例程序8-4使用了glDrawPixels()和glCopyPixels()函数。

对于这3个函数，写入帧缓冲区或者从帧缓冲区读取时所进行的数据转换取决于当时生效的模式，详见下一节。

OpenGL 3.0引入了一个新的像素复制命令，名为glBlitFramebuffer()，它包含了glCopyPixels() 和glPixelZoom()的功能。第10.4.2节将详细介绍这一点，因为它用到了帧缓冲区对象的某些功能。

8.3 图像管线

本节讨论完整的图像管线，像素存储模式和像素传输操作，包括如何设置一种任意的映射，对像素数据进行转换。还可以调用glPixelZoom()函数在绘制像素矩形前对它进行放大或缩小。图8-5显示了这些操作的顺序。

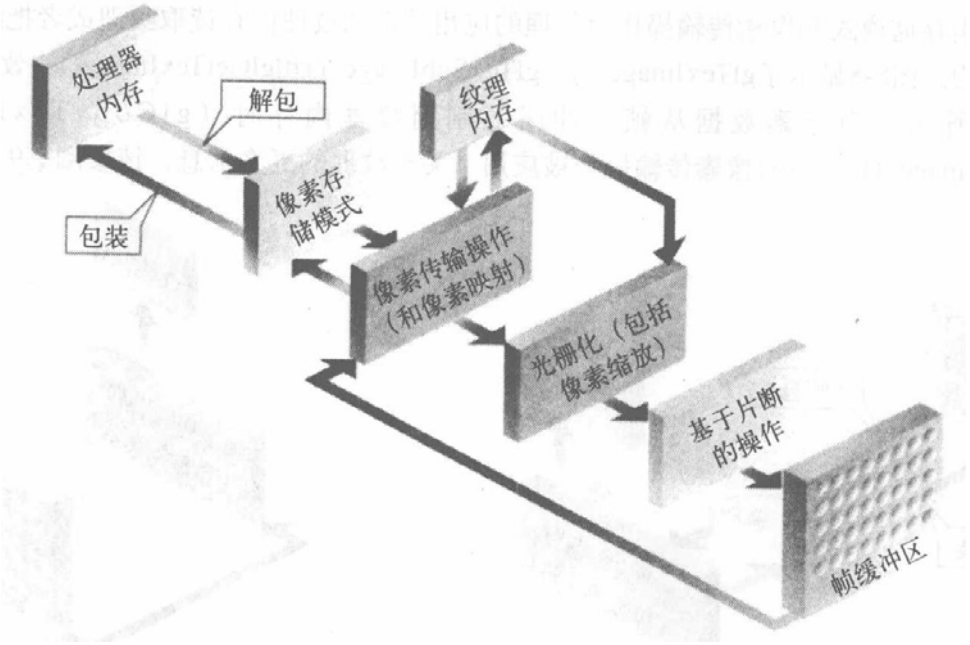


图8-5 图像管线

当glDrawPixels()函数被调用时，数据首先根据当前生效的像素存储模式从内存进行解包，接着执行像素传输操作。然后，最终的像素进行光栅化。在光栅化阶段，像素矩形可能会根据当前的状态进行放大或缩小。最后应用的是片断操作，像素被写入到帧缓冲区（关于片断操作的详细讨论，请参阅第10.2节）。

当glReadPixels()函数被调用时，数据从帧缓冲区读回，并执行像素传输操作，最终的数据包装到处理器内存中。

glCopyPixels()函数执行所有的像素传输操作（相当于glReadPixels()函数执行的操作），然后像glReadPixels()函数一样写入最终图像，但不需要进行第二次传输。图8-6显示了glCopyPixels()函数如何在帧缓冲区内部移动数据。

根据第8.1.2节以及图8-7，我们可以发现渲染位图要比渲染图像简单一些，因为前者不需要执行像素传输操作和像素缩放操作。

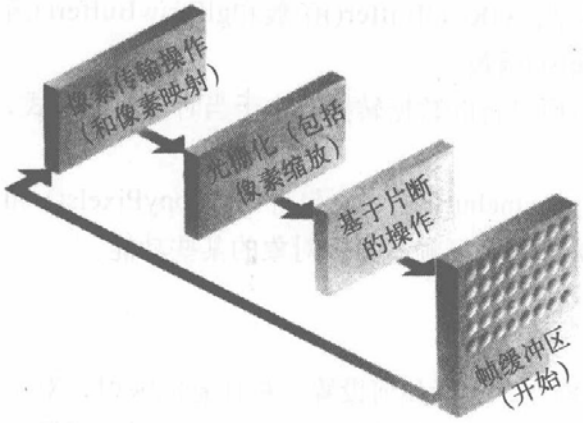


图8-6 glCopyPixels()的像素路径

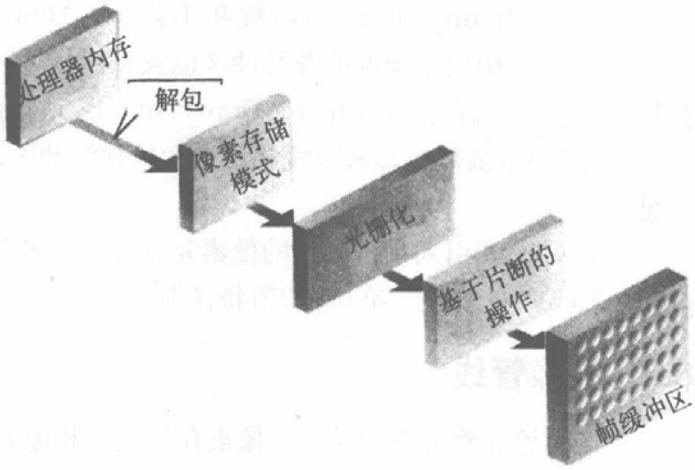


图8-7 glBitmap()的像素路径

注意，像素存储模式和像素传输操作对纹理的应用是在从纹理内存读取纹理或者把纹理写入到纹理内存时进行的。图8-8显示了glTexImage*()、glTexSubImage*()和glGetTexImage()的效果。

如图8-9所示，当像素数据从帧缓冲区复制到纹理内存时（glCopyTexImage*()或glCopyTexSubImage*()），只有像素传输操作被应用（关于纹理的更多信息，请参阅第9章）。

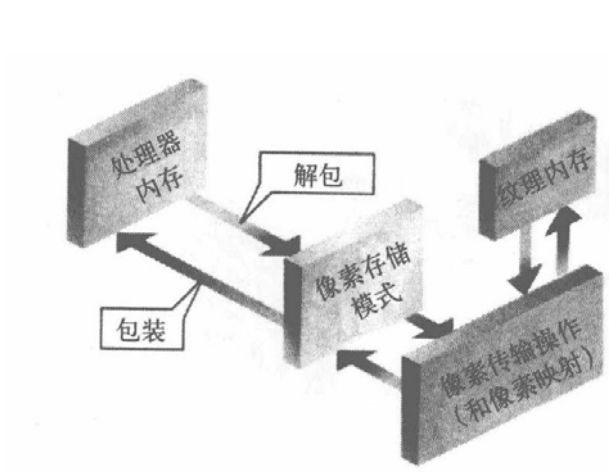


图8-8 glTexImage*()、glTexSubImage*()和glGetTexImage()的像素路径

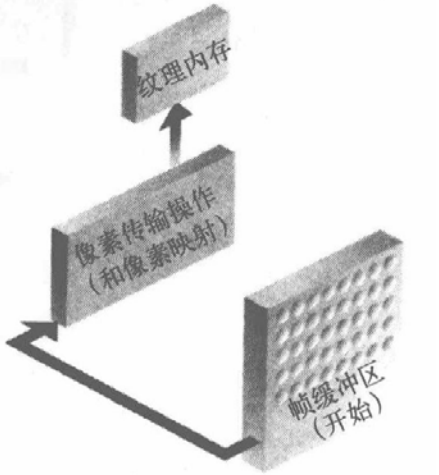


图8-9 glCopyTexImage*()和glCopyTexSubImag*()的像素路径

8.3.1 像素包装和解包

包装和解包是指像素数据写入到处理器内存以及从处理器内存读取的方式。

当图像存储在内存中时，每个像素有1~4块数据（称为元素）组成。像素数据有可能只包括颜色索引或亮度（亮度是红、绿、蓝值的加权之和），也可能包括了每个像素的红、绿、蓝和alpha成分。像素数据可能出现的排列（或称为格式）决定了每个像素存储的元素数量以及它们的存储顺序。

有些元素（如颜色索引或模板索引）是整型值，有些元素（如红、绿、蓝、alpha值，以及深度成分）则是浮点值（一般位于0.0~1.0）。存储于帧缓冲区的浮点成分通常比完整的浮点值所需要的精度低一点。例如，颜色成分可以用8位存储。表示各种成分的位数取决于程序所使用的特定硬件。因此，用32位的完整浮点数形式存储每个成分往往是非常浪费的，特别是图像有可能包含上百万个像素。

元素可以按照各种不同的类型存储于内存中，从8位的字节到32位的整数或浮点数。OpenGL明确定义了每种格式下的每种成分转换为每种可行的数据类型的方式。记住，如果试图把一个高精度的成分值转换为一种低精度的类型，可能会损失数据的精度。

8.3.2 控制像素存储模式

图像数据常常以二维或三维矩形数组的形式存储于处理器内存中。我们常常需要显示或存储一幅对应于数组某个区域的子图像。另外，需要考虑不同的计算机可能具有不同的字节顺序约定。最后，在有些计算机中，如果存储在内存中的数据能够按照2字节、4字节或8字节对齐，那么它们在内存和帧缓冲区之间的传输将具有非常高的效率。对于这类计算机，我们可能想控制字节的对齐方式。本段所描述的所有问题都是由像素存储模式控制的，我们将在下一小节详细讨论这个概念。可以使用glPixelStore*()函数指定像素存储模式。在前面的几个示例程序中，我们已经看到过这个函数的用法。

OpenGL所支持的所有像素存储模式都是由glPixelStore*()函数控制的。一般情况下，可以连续几次调用这个函数，成批地设置几个参数值。

---

void **glPixelStore**{if}(GLenum *pname*, *TYPE param*);

设置像素存储模式。这些模式将影响**glDrawPixels()**、**glReadPixels()**、**glBitmap()**、**glPolygonStipple()**、**glTexImage1D()**、**glTexImage2D()**、**glTexImage3D()**、**glTexSubImage1D()**、**glTexSubImage2D()**、**glTexSubImage3D()**、**glGetTexImage()**函数所执行的处理。如果启用了图像处理子集特性（参见第8.7节），它们还将影响glGetColorTable()、glGetConvolutionFilter()、glGetSeparableFilter()、glGetHistogram()和glGetMinmax()的操作。

pname参数可以使用的值见表8-4，表8-4还列出了它们的数据类型、初始值以及有效值范围。GL_UNPACK*参数控制数据在内存中是如何由**glDrawPixels()**、**glBitmap()**、**glPolygonStipple()**、**glTexImage1D()**、**glTexImage2D()**、**glTexImage3D()**、**glTexSubImage1D()**、**glTexSubImage2D()**、**glTexSubImage3D()**解包的。GL_PACK*参数控制数据是如何由glReadPixels()和glGetTexImage()包装到内存中的。如果启用了图像处理子集特性，这些函数还包括glGetColorTable()、glGetConvolutionFilter()、glGetSeparableFilter()、glGetHistogram()和glGetMinmax()。

GL_UNPACK_IMAGE_HEIGHT、GL_PACK_IMAGE_HEIGHT、GL_UNPACK_SKIP _IMAGES和GL_PACK_SKIP_IMAGES只影响3D纹理（**glTexImage3D()**、**glTexSubImage3D()**和**glGetTexImage(GL_TEXTURE_3D, …)**）

表8-4 glPixelStore()的参数

参数名称	类型	初始值	有效范围
GL_UNPACK_SWAP_BYTES, GL_PACK_SWAP_BYTES	GLboolean	FALSE	TRUE/FALSE
GL_UNPACK_LSB_FIRST, GL_PACK_LSB_FIRST	GLboolean	FALSE	TRUE/FALSE
GL_UNPACK_ROW_LENGTH, GL_PACK_ROW_LENGTH	GLint	0	任何非负整数
GL_UNPACK_SKIP_ROWS, GL_PACK_SKIP_ROWS	GLint	0	任何非负整数
GL_UNPACK_SKIP_PIXELS, GL_PACK_SKIP_PIXELS	GLint	0	任何非负整数
GL_UNPACK_ALIGNMENT, GL_PACK_ALIGNMENT	GLint	4	1, 2, 4, 8
GL_UNPACK_IMAGE_HEIGHT, GL_PACK_IMAGE_HEIGHT	Glint	0	任何非负整数
GL_UNPACK_SKIP_IMAGES, GL_PACK_SKIP_IMAGES	GLint	0	任何非负整数

由于用于包装和解包的对应参数具有相同的含义，因此在本节的剩余部分，我们将一起介绍它们，而不再提及前缀GL_PACK或GL_UNPACK。例如，*SWAP_BYTES表示GL_PACK_SWAP_BYTES和GL_UNPACK_SWAP_BYTES。

如果*SWAP_BYTES参数为FALSE（默认值），内存中的字符顺序就采用OpenGL客户机的自身方案。否则，就反转字节顺序。字节反转可作用于任何大小的元素，但只对多字节的元素才有意义。

字节交换在不同的OpenGL实现中可能具有不同的效果。如果在某种OpenGL实现中，GLubyte为8位，GLushort为16位，GLint为32位，图8-10说明了这几种不同的数据类型是如何进行字节交换的。注意，字节交换对于单字节数据并无效果。

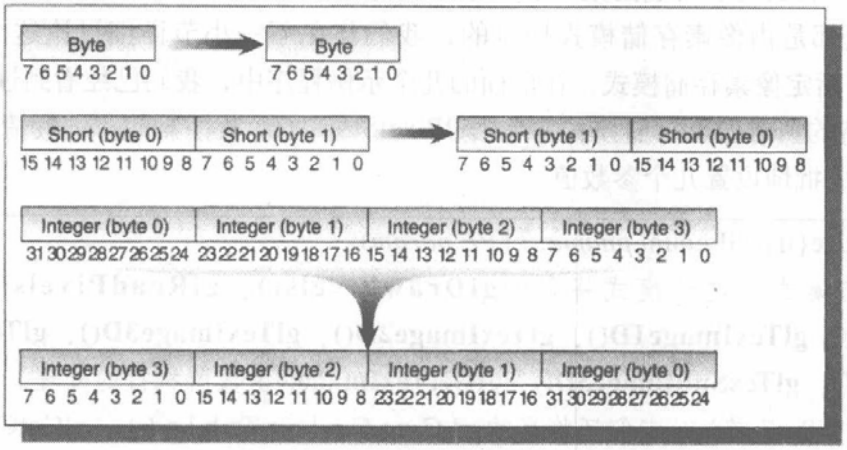


图8-10 byte、short、int数据的字节交换效果

注意：只要OpenGL应用程序并不与其他计算机共享图像，就可以忽略字节顺序的问题。如果应用程序必须渲染在不同的计算机上创建的图像，并且这些计算机具有不同的字节顺序，就可以使用*SWAP_BYTES来交换字节顺序。但是，*SWAP_BYTES并不允许重新排列元素的顺序（例如，交换红色和绿色成分的位置）。

*LSB_FIRST参数只适用于在位图上绘制或读取1位图像（位图的每个像素只保存和恢复1位数据）。如果*LSB_FIRST为FALSE（默认值），数据位就从字节的最高有效位开始提取。否则，数据位就从相反的方向开始提取。例如，如果*LSB_FIRST是FALSE，字节是0x31，位顺序是{0，0，1，1，0，0，0，1}。如果*LSB_FIRST是TRUE，位顺序就是{1，0，0，0，1，1，0，0}。

有时候，我们只想绘制或读取在内存中所存储的图像数据矩形的一个子矩形。如果内存中的矩形大于被绘制或读取的子矩形，就需要使用*ROW_LENGTH参数来指定这个更大矩形的实际长度（以像素为单位）。如果*ROW_LENGTH为0（默认值），行长度便被认为与使用glReadPixels()、glDrawPixels()或glCopyPixels()所指定的宽度相同。我们还需要指定从这个子矩形开始复制数据时需要跳过的行数和像素数。这两个值是用*SKIP_ROWS和*SKIP_PIXELS参数设置的，如图8-11所示。在默认情况下，这两个参数的值都为0，也就是从左下角开始复制数据。

特定的计算机常常会对硬件进行优化。如果数据在内存中是按照一种特定的字节对齐形式保存的，那么像素进出内存时就具有非常高的效率。例如，在一台字长32位的计算机上，如果数据在内存中按照32位的边界对齐（地址为4字节的倍数），那么硬件提取数据的速度就会快得多。类似地，在一台64位的计算机上，如果数据的地址按照8字节的边界对齐，它的数据存取效率也会非常高。但是，也有一些计算机的数据存储速度与字节对齐与否并无关系。

让我们来看一个例子，假定我们使用的计算机对于按照4字节边界对齐的像素数据具有更快的访问速度。如果我们强制图像的每行数据都从4节字的边界开始存储，它的存储效率最高。如果图像的宽度是5个像素，每个像素分别用1个字节表示它的红、绿和蓝色信息，那么它的一行就需要5 × 3 = 15字节的数据。如果第一行数据以及后续的每行数据都从4字节的边界开始存储，就具有最佳的显示效率。这样，每行数据都有1个字节的内存空间被浪费（每行16个字节能够满足4字节的边界对齐要求）。如果数据是按照这种方式存储的，就可以把*ALIGNMENT参数设置为适当的值（在此例中为4）。

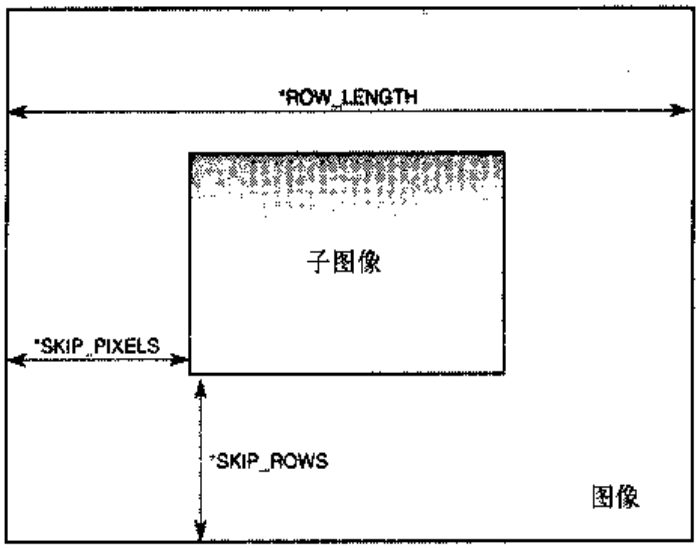


图8-11 *SKIP_ROWS、*SKIP_PIXELS和*ROW_LENGTH参数

如果*ALIGNMENT设置为1，每一行数据都是从下一个可用的字节开始存储的。如果设置为2，每一行的最后可能会跳过1个字节，使下一行数据的起始地址总是2的倍数。如果是位图（或者1位的图像），每个像素都用1个字节表示，前述的字节对齐方式对它也适用，不过必须计数单独位的数量。例如，假设每个像素保存1位的数据，如果行的长度是75，并且对齐值为4个字节，那么每行就需要75/8（或93/8）个字节。由于12是4的所有倍数中最小的大于93/8的值，因此每一行用12个字节来表示。如果对齐值为1，每行就用10个字节来表示（大于93/8的最小整数）。示例程序8-4显示了glPixelStorei()函数的简单用法。

注意：*ALIGNMENT的默认值是4。一个常见的编程错误就是以为图像数据是紧密排列并对齐的（即假设*ALIGNMENT设置为1）。

*IMAGE_HEIGHT和*SKIP_IMAGES参数只影响三维纹理的定义和查询。关于这些像素存储模式的细节，请参阅第9.2.4节中的“三维纹理的像素存储模式”。

8.3.3 像素传输操作

当图像从内存传输到帧缓冲区或者从帧缓冲区传输到内存时，OpenGL可以对它执行一些操作。例如，可以更改颜色成分的范围。一般情况下，红色成分位于0.0~1.0之间，但是我们也可以把它限定在另一个范围之内。或者，可以把来自其他图形系统的像素数据的红色成分限制在一个特定的范围之内。甚至可以创建颜色映射，在像素传输期间执行任意的颜色索引或颜色成分的转换。这种在像素传输期间所执行的转换称为像素传输操作。它们是由glPixelTransfer*()和glPixelMap*()函数控制的。

注意，尽管颜色、深度和模板缓冲区具有许多相似之处，但是它们的行为并不相同，有些模式具有一些特殊情况。本节以及接下来的数节讨论所有的模式细节，包括这些特殊情况。

像素传输函数的有些特征是用glPixelTransfer*()函数设置的，其他特征则是由glPixelMap*()函数指定的，我们将在下一节描述这个函数。

兼容性扩展
glPixelTransfer及其他函数

void **glPixelTransfer**{if}(GLenum *pname*, *TYPE param*);

设置像素传输模式。像素传输模式将影响**glDrawPixels()**、**glReadPixels()**、**glCopyPixels()**、**glTexImage1D()**、**glTexImage2D()**、

glTexImage3D()、**glCopyTexImage1D()**、**glCopyTexImage2D()**、**glTexSubImage1D()**、**glTexSubImage2D()**、**glTexSubImage3D()**、**glCopyTexSubImage1D()**、**glCopyTexSubImage2D()**、**glCopyTexSubImage3D()**和**glGetTexImage()**的操作。pname参数必须是表8-5的第一列所列出的值之一，它的值param也必须位于这张表所列出的有效范围之内。

表8-5 glPixelTransfer*()参数

参数名称	类 型	初始值	有效范围
GL_MAP_COLOR	GLboolean	FALSE	TRUE/FALSE
GL_MAP_STENCIL	GLboolean	FALSE	TRUE/FALSE
GL_INDEX_SHIFT	GLint	0	(−∞, ∞)
GL_INDEX_OFFSET	GLint	0	(−∞, ∞)
GL_RED_SCALE	GLfloat	1.0	(−∞, ∞)
GL_GREEN_SCALE	GLfloat	1.0	(−∞, ∞)
GL_BLUE_SCALE	GLfloat	1.0	(−∞, ∞)
GL_ALPHA_SCALE	GLfloat	1.0	(−∞, ∞)
GL_DEPTH_SCALE	GLfloat	1.0	(−∞, ∞)
GL_RED_BIAS	GLfloat	0.0	(−∞, ∞)
GL_GREEN_BIAS	GLfloat	0.0	(−∞, ∞)
GL_BLUE_BIAS	GLfloat	0.0	(−∞, ∞)
GL_ALPHA_BIAS	GLfloat	0.0	(−∞, ∞)
GL_DEPTH_BIAS	GLfloat	0.0	(−∞, ∞)
GL_POST_CONVOLUTION_RED_SCALE	GLfloat	1.0	(−∞, ∞)
GL_ POST_CONVOLUTION_GREEN_SCALE	GLfloat	1.0	(−∞, ∞)
GL_ POST_CONVOLUTION_BLUE_SCALE	GLfloat	1.0	(−∞, ∞)
GL_ POST_CONVOLUTION_ALPHA_SCALE	GLfloat	1.0	(−∞, ∞)
GL_ POST_CONVOLUTION_RED_BIAS	GLfloat	0.0	(−∞, ∞)
GL_ POST_CONVOLUTION_GREEN_BIAS	GLfloat	0.0	(−∞, ∞)
GL_ POST_CONVOLUTION_BLUE_BIAS	GLfloat	0.0	(−∞, ∞)
GL_ POST_CONVOLUTION_ALPHA_BIAS	GLfloat	0.0	(−∞, ∞)
GL_ POST_COLOR_MATRIX_ RED_SCALE	GLfloat	1.0	(−∞, ∞)
GL_ POST_ COLOR_MATRIX_ GREEN_SCALE	GLfloat	1.0	(−∞, ∞)
GL_ POST_ COLOR_MATRIX_ BLUE_SCALE	GLfloat	1.0	(−∞, ∞)
GL_ POST_ COLOR_MATRIX_ ALPHA_SCALE	GLfloat	1.0	(−∞, ∞)
GL_ POST_ COLOR_MATRIX_ RED_BIAS	GLfloat	0.0	(−∞, ∞)
GL_ POST_ COLOR_MATRIX_ GREEN_BIAS	GLfloat	0.0	(−∞, ∞)
GL_ POST_ COLOR_MATRIX_ BLUE_BIAS	GLfloat	0.0	(−∞, ∞)
GL_ POST_ COLOR_MATRIX_ ALPHA_BIAS	GLfloat	0.0	(−∞, ∞)

警告：GL_POST_CONVOLUTION_*和GL_POST_COLOR_MATRIX_*参数只有当OpenGL实现支持图像处理子集功能时才可用（参阅第8.7节）。

如果GL_MAP_COLOR或GL_MAP_STENCIL参数设置为TRUE，就启用了映射功能。下一小节将描述如何完成这种映射以及如何修改映射的内容。其他所有参数都会直接影响像素成分的值。

缩放和偏移可以作用于红、绿、蓝、alpha和深度成分。例如，从帧缓冲区读取红、绿、蓝成分时，我们可能希望在把它们转换为亮度格式之前在处理器内存中对它们进行缩放。亮度是通过对红、

绿、蓝成分进行求和计算而得的，因此如果使用了GL_RED_SCALE、GL_GREEN_SCALE和GL_BLUE_SCALE的默认值，这些成分将在最终的亮度值（或强度值）中具有相同的贡献。如果我们想根据NTSC标准把RGB转换为亮度值，可以把GL_RED_SCALE设置为0.30，把GL_GREEN_SCALE设置为0.59，把GL_BLUE_SCALE设置为0.11。

索引（颜色索引和模板索引）也可以进行转换。如果是索引，就会对它应用移位和偏移。如果想在渲染时控制颜色表的哪部分被使用，这个技巧就很实用。

8.3.4 像素映射

所有的颜色成分、颜色索引和模板索引在进入屏幕内存之前都可以通过表查找的方式进行更改。用于控制这种映射的函数是glPixelMap*()。

void **glPixelMap**{ui us f}v(GLenum *map*, GLint *mapsize*,
const TYPE **values*);

加载由map所确定的像素映射表，它共有mapsize个项。values指向它们的值。表8-6列出了映射名称和值。它们的默认大小都是1，默认值均为0。每个映射表的大小必须是2的整数次方。

兼容性扩展
glPixelMap
及其他函数

表8-6 glPixelMap*()的参数名称和值

映射名称	地址	值
GL_PIXEL_MAP_I_TO_I	颜色索引	颜色索引
GL_PIXEL_MAP_S_TO_S	模板索引	模板索引
GL_PIXEL_MAP_I_TO_R	颜色索引	R
GL_PIXEL_MAP_I_TO_G	颜色索引	G
GL_PIXEL_MAP_I_TO_B	颜色索引	B
GL_PIXEL_MAP_I_TO_A	颜色索引	A
GL_PIXEL_MAP_R_TO_R	R	R
GL_PIXEL_MAP_G_TO_G	G	G
GL_PIXEL_MAP_B_TO_B	B	B
GL_PIXEL_MAP_A_TO_A	A	A

映射表的最大大小因计算机而异。可以用glGetIntegerv()函数查询自己所使用的计算机所支持的像素映射表的大小。可以用GL_MAX_PIXEL_MAP_TABLE参数获取所有像素映射表的最大大小，也可以用GL_PIXEL_MAP_*_T()_*_SIZE参数获取指定像素映射表的当前大小。在表8-6中，地址为颜色索引或模板索引的6个映射表的大小必须是2的整数次方。4个RGBA映射表可以是1至GL_MAX_PIXEL_MAP_TABLE之间的任何大小。

为了理解像素映射表是如何工作的，让我们看一个简单的例子。假定我们想创建一个包含256个项的映射表，并使用GL_PIXEL_MAP_I_TO_I进行颜色索引与颜色的映射。可以创建一个表，表中的每个项分别表示0~255之间的一个值，并用glPixelMap*()函数对这张表进行初始化。假设我们想把这个表作为门槛值，把101以下的索引（0~100）映射为0，并把101及以上的索引映射为255。在这种情况下，这张表就是由101个0和155个255组成。为了启用像素映射，可以使用glPixelTransfer*()函数，并把GL_MAP_COLOR参数设置为TRUE。一旦加载并启用了这个像素映射表之后，所有小于101的颜色索引便被映射为0，所有101~255之间的颜色索引便被映射为255。如果像素的颜色索引值

大于255，它首先用255这个掩码进行处理，丢弃前8位，剩余的值再在映射表中进行查找。如果索引是一个浮点值（例如88.145 85），它就被四舍五入为最接近的整数（即88），然后再在映射表中查找这个值（结果为0）。

在像素映射表中，还可以映射模板索引或者把颜色索引转换为RGB（有关索引转换的相关信息，请参阅第8.4节）。

8.3.5 放大、缩小或翻转图像

在应用了像素存储模式和像素传输操作之后，图像和位图就进行光栅化。正常情况下，图像中的每个像素被写入到屏幕中的一个像素。但是，可以使用glPixelZoom()函数对图像进行任意的放大、缩小甚至翻转（反射）。

兼容性扩展

glPixelZoom

void **glPixelZoom**(GLfloat $zoom_x$, GLfloat $zoom_y$);

设置像素写入操作（**glDrawPixels()**和**glCopyPixels()**）中x和y方向的放大或缩小因子。在默认情况下，$zoom_x$和$zoom_y$都是1.0。如果它们都是2.0，图像的每个像素被绘制成4个屏幕像素。注意，可以使用分数形式的放大或缩小因子，也可以使用负的缩放因子。负的缩放因子根据当前的光栅位置对图像进行翻转。

在光栅化过程中，每个图像像素被当作一个$zoom_x \times zoom_y$的矩形，并且为中心位于这个矩形之内的所有像素生成片段。具体地说，假设当前的光栅位置是(x_{rp}, y_{rp})，如果一组特定的元素（索引或成分）属于第n行第m列，那么在窗口坐标下这个矩形边界的对角是：

$$(x_{rp} + zoom_x \cdot n,\ y_{rp} + zoom_y \cdot m) \text{ 和 } (x_{rp} + zoom_x\ (n+1),\ y_{rp} + zoom_y\ (m+1))$$

中心位于这个矩形内部（或者位于底部或左边界上）的所有片断都是根据这组特定的元素产生的。

负值的缩放因子可以用来产生翻转的图像，OpenGL从底向上（并从左向右）逐行描述图像。如果有一幅“自顶向下”的图像，例如一帧录像，就可以使用glPixelZoom(1.0, −1.0)使图像符合OpenGL所定义的顺序。另外，在必要的情况下，应该正确地重置光栅位置。

示例程序8-4显示了glPixelZoom()函数的用法。这个程序首先在窗口的左下角绘制了一幅棋盘图像。通过按下鼠标按钮并移动鼠标，可以使用glCopyPixels()函数把窗口的左下角复制到当前光标位置（如果把图像复制到它本身的位置，它看上就会显得十分古怪）。被复制的图像将进行缩放，但最初它的缩放值是默认的1.0，因此我们不会注意到它实际上进行了缩放。按下“z”和“Z”键可以把缩放因子的值增加或减少0.5。任何形式的窗口破坏都会导致窗口的内容被重绘。按下“r”键可以对图像和缩放因子进行重置。

示例程序8-4 绘制、复制和缩放像素数据：image.c

```
#define checkImageWidth 64
#define checkImageHeight 64
GLubyte checkImage [checkImageHeight][checkImageWidth][3];

static GLdouble zoomFactor =1.0;
static GLint height;

void makeCheckImage(void)
{
   int i,j,c;
```

```
    for (i =0;i <checkImageHeight;i++){
        for (j =0;j <checkImageWidth;j++){
            c =(((i&0x8)==0)^((j&0x8)==0))*255;
            checkImage [i][j][0] =(GLubyte)c;
            checkImage [i][j][1] =(GLubyte)c;
            checkImage [i][j][2] =(GLubyte)c;
        }
    }
}

void init(void)
{
    glClearColor(0.0,0.0,0.0,0.0);
    glShadeModel(GL_FLAT);
    makeCheckImage();
    glPixelStorei(GL_UNPACK_ALIGNMENT,1);
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glRasterPos2i(0,0);
    glDrawPixels(checkImageWidth,checkImageHeight,GL_RGB,
                 GL_UNSIGNED_BYTE,checkImage);
    glFlush();
}

void reshape(int w,int h)
{
    glViewport(0,0,(GLsizei)w,(GLsizei)h);
    height =(GLint)h;
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0,(GLdouble)w,0.0,(GLdouble)h);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
void motion(int x,int y)
{
    static GLint screeny;

    screeny =height -(GLint)y;
    glRasterPos2i(x,screeny);
    glPixelZoom(zoomFactor,zoomFactor);
    glCopyPixels(0,0,checkImageWidth,checkImageHeight,
                 GL_COLOR);
    glPixelZoom(1.0,1.0);
    glFlush();
}

void keyboard(unsigned char key,int x,int y)
{
```

```
    switch (key){
       case 'r ':
       case 'R ':
          zoomFactor =1.0;
          glutPostRedisplay();
          printf("zoomFactor reset to 1.0 \n ");
          break;
       case 'z ':
          zoomFactor +=0.5;
          if (zoomFactor >=3.0)
          zoomFactor =3.0;
          printf("zoomFactor is now %4.1f \n ",zoomFactor);
          break;
       case 'Z ':
          zoomFactor -=0.5;
          if (zoomFactor <=0.5)
          zoomFactor =0.5;
          printf("zoomFactor is now %4.1f \n ",zoomFactor);
          break;
       case 27:
          exit(0);
          break;
       default:
          break;
    }
}
```

8.4 读取和绘制像素矩形

本节详细描述像素的读取和绘制过程。当像素从帧缓冲区复制到内存（读取）时所进行的像素转换操作与像素从内存复制到缓冲区（绘制）时所进行的像素转换操作颇为相似，但是它们并不相同。接下来的几节将对此进行解释。如果读者初次阅读本书，可以跳过本节内容，尤其是当读者并不想马上使用像素传输操作时。

像素矩形的绘制过程

图8-12以及它后面的内容描述了把像素绘制到帧缓冲区的操作过程。

1) 如果像素并不是索引（也就是说，如果像素格式并不是GL_COLOR_INDEX或GL_STENCIL_INDEX），第一个步骤就是在必要的情况下把成分值转换为浮点形式（关于转换的细节，请参阅表4-1）。

2) 如果格式是GL_LUMINANCE或GL_LUMINANCE_ALPHA，亮度元素就转换为R、G、B成分，每个成分都使用这个亮度值。如果是GL_LUMINANCE_ALPHA格式，alpha值成为A值。如果是GL_LUMINANCE格式，A值便设置为1.0。

3) 每个成分（R、B、G、A或深度）都与适当的缩放因子相乘，并加上适当的偏移值。例如，R成分与GL_RED_SCALE对应的值相乘，并与GL_RED_BIAS对应的值相加。

4) 如果GL_MAP_COLOR为TRUE，R、B、G和A成分都截取在[0.0，1.0]的范围之内，并且与一个整数（表的长度减去1）相乘，截去小数部分，然后再在表中进行查找。关于这方面的更多细节，请参阅第8.6节。

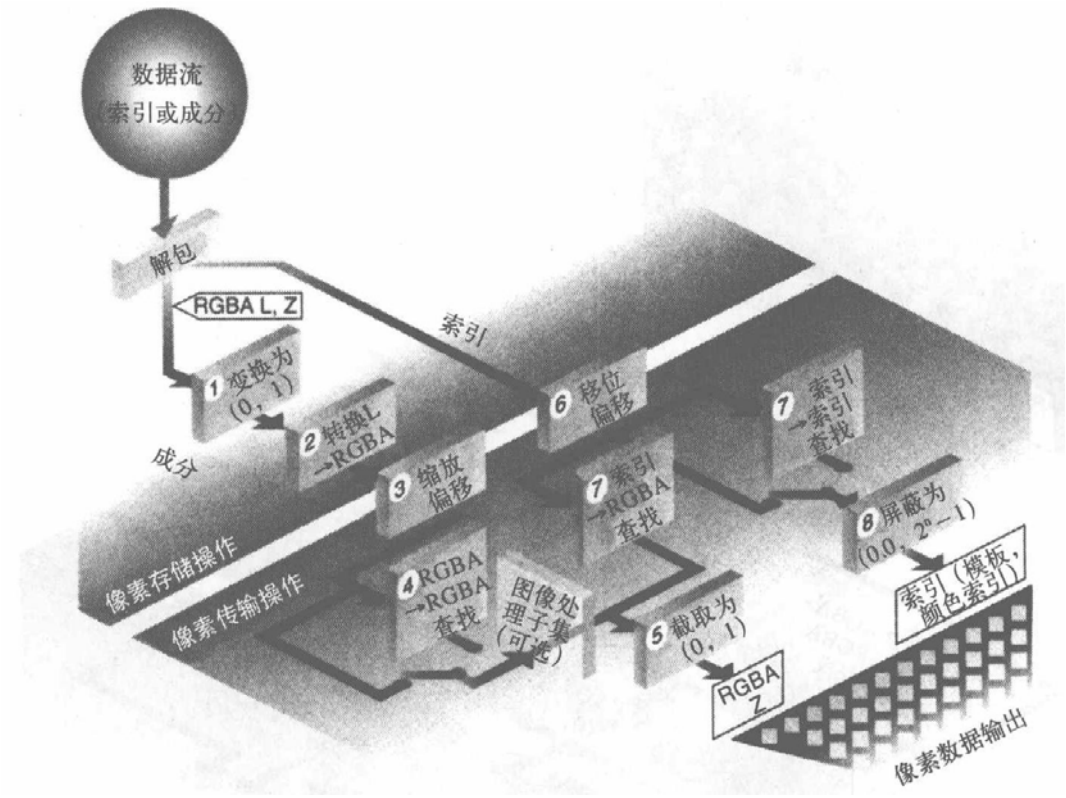


图8-12 用glDrawPixels()函数绘制像素

5) 接着，如果R、G、B、A成分的值还没有截取在[0.0，1.0]范围之内，就把它们截取在这个范围之内。然后，把它们转换为定点数，小数点左边的位数与对应的帧缓冲区成分的位数相同。

6) 如果是索引值（模板或颜色索引），首先把它们转换为定点形式（如果它们最初为浮点数），小数点右边是一些未指定的位。最初为定点形式的那些索引值仍然保持原样，小数点右边的所有位都设置为0。

随后，这个步骤所产生的索引值进行左移或右移（移动的位数是GL_INDEX_SHIFT的绝对值）。如果GL_INDEX_SHIFT大于0，它就进行左移，否则就进行右移。最后，这个索引值与GL_INDEX_OFFSET相加。

7) 处理索引值的下一个步骤取决于使用的是RGBA模式还是颜色索引模式。在RGBA模式下，颜色索引值使用GL_PIXEL_MAP_I_TO_R、GL_PIXEL_MAP_I_TO_G、GL_PIXEL_MAP_I_TO_B和GL_PIXEL_MAP_I_TO_A所指定的颜色成分转换为RGBA值（详见第8.3.4节）。在颜色索引模式下，如果GL_MAP_COLOR为GL_TRUE，它就在GL_PIXEL_MAP_I_TO_I表中查找一个颜色索引值（如果GL_MAP_COLOR为GL_FALSE，这个索引值就不发生变化）。如果图像是由模板索引而不是颜色索引组成的，并且GL_MAP_COLOR为GL_TRUE，它就在与GL_PIXEL_MAP_S_TO_S所对应的表中查找一个索引值。如果GL_MAP_STENCIL为GL_FALSE，原来的模板索引值就不会发生变化。

8) 最后，如果索引值并没有转换为RGBA值，它们就会进行屏蔽处理，掩码的长度或者是颜色索引缓冲区的长度，或者是模板缓冲区的长度。

像素矩形的读取过程

在像素矩形绘制过程中所执行的许多转换操作在像素矩形读取过程中也要执行。图8-13显示了像素矩形的读取过程，下面描述了这个过程。

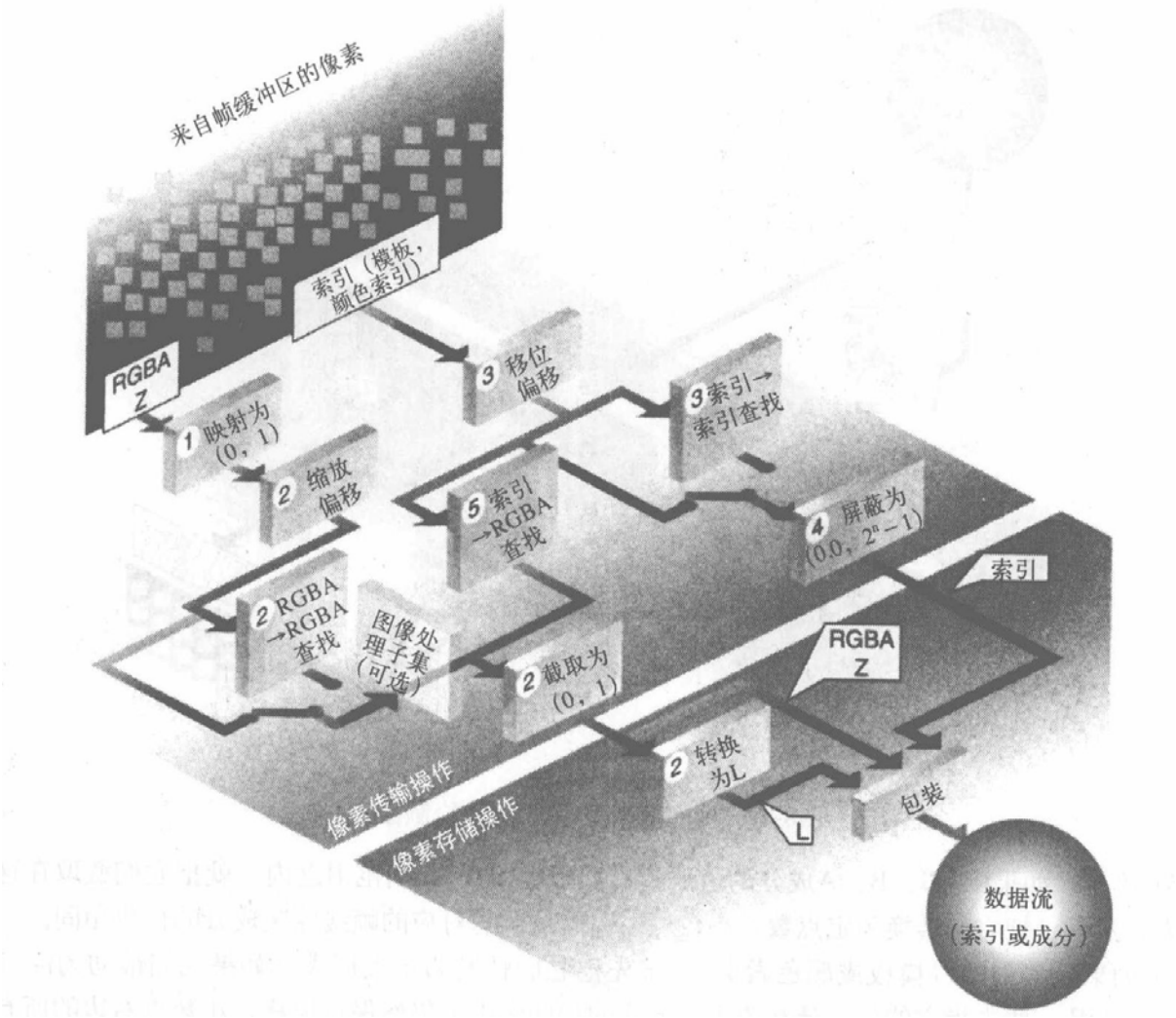


图8-13 用glReadPixels()函数读取像素

1）如果被读取的像素不是索引值（也就是说，如果格式不是GL_COLOR_INDEX或GL_STENCIL_INDEX），像素成分就被映射到[0.0, 1.0]，也就是和写入时刚好相反。

2）接着，对每个成分进行缩放和偏移。如果GL_MAP_COLOR为GL_TRUE，它们就进行映射，并再次截取在[0.0, 1.0]的范围之内。如果所需要的是亮度值而不是RGB值，那么R、G、B值就相加（L = R + G + B）。

3）如果像素是索引值（颜色索引或模板索引），它们就进行移位、偏移，并且如果GL_MAP_COLOR为GL_TRUE，它们还将进行映射。

4）如果存储格式为GL_COLOR_INDEX或GL_STENCIL_INDEX，像素索引就根据存储类型的位数（1、8、16或32）进行屏蔽，并按照前面描述的方法包装到内存中。

5）如果存储格式是某种成分类型（例如亮度或RGB），像素总是需要执行从索引到RGBA的映射。接着，它们就像一开始就是RGBA值一样进行处理（如果必要，可以转换为亮度值）。

6）最后，不管是索引数据还是成分数据，最终的结果都根据glPixelStore*()所设置的GL_PACK*模式包装到内存中。

像素读取操作使用的缩放、偏移、移位值与像素绘制操作使用的操作相同，因此如果我们既要读取像素又要绘制像素，就需要在每次读取或绘制之前把这些成分重置为适当的值。类似地，如果在读

取和绘制像素时都要使用映射表，也需要对各个映射表进行正确的重置。

注意：在像素读取和绘制操作中，亮度的处理似乎是不正确的。例如，亮度可能并不像图8-12和图8-13所示的那样均匀地依赖于R、G、B成分。例如，如果我们想根据自己的方式计算亮度值，让R、G、B成分分别占据30%、59%和11%，可以把GL_RED_SCALE设置为0.30，把GL_RED_BIAS设置为0.0，接下来以此类推。按照这种方式，亮度值L就是0.30R + 0.59G + 0.11B。

8.5 使用缓冲区对象存取像素矩形数据



高级话题

在第2.7.7节中，我们描述了如何使用缓冲区对象存储顶点数组数据，以提高应用程序的性能。与此相似，在缓冲区对象中存储像素数据也可以提高应用程序的性能。

通过把像素矩形数据存储在服务器端的缓冲区对象中，可以消除在渲染每一帧时都把数据从客户机的内存传输到OpenGL服务器的开销。例如，为了把一幅图像渲染为场景的背景，可以不调用glClear()方法，而是改用缓冲区对象。

与使用缓冲区对象存储顶点数组数据相比，像素缓冲区对象既可以读取（就像对应的顶点数组缓冲区对象一样），也可以写入。当从OpenGL提取像素数据时就会写入到缓冲区对象，例如调用glReadPixels()或者使用glGetTexImage()提取一个纹理的纹理单元时。

8.5.1 使用缓冲区对象传输像素数据

一些用于把数据从客户机应用程序的内存传输到OpenGL服务器的OpenGL函数（例如glDrawPixels()、glTexImage*D()、glCompressedTexImage*D()、glPixelMap*()）以及图像处理子集中接受像素数组为参数的类似函数，可以使用缓冲区对象在服务器中存储像素数据。

为了在缓冲区对象中存储像素矩形数据，需要在应用程序中添加一些步骤。

1)（可选）调用glGenBuffers()生成缓冲区对象标识符。

2) 以GL_PIXEL_UNPACK_BUFFER为参数调用glBindBuffer()函数，绑定一个缓冲区对象，用于对像素进行解包。

3) 使用glBufferData()函数请求数据存储，并可以初始化这些数据元素。同样，在调用这个函数时把target参数指定为GL_PIXEL_UNPACK_BUFFER。

4) 再次调用glBindBuffer()，绑定适当的缓冲区对象，在渲染中使用。

5) 调用适当的函数（例如glDrawPixels()或glTexImage2D()），使用这些数据。

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)～3)。

示例程序8-5修改了image.c程序（示例程序8-4），使用了像素缓冲区对象。

示例程序8-5 绘制、复制和缩放存储在缓冲区对象中的像素数据：pboimage.c

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

/*Create checkerboard image*/
#definecheckImageWidth 64
#definecheckImageHeight 64
GLubyte checkImage [checkImageHeight][checkImageWidth][3];

static GLdouble zoomFactor =1.0;
static GLint height;
```

```
static GLuint pixelBuffer;

void makeCheckImage(void)
{
   int i,j,c;

   for (i =0;i <checkImageHeight;i++){
      for (j =0;j <checkImageWidth;j++){
         c =(((i&0x8)==0)^((j&0x8)==0))*255;
         checkImage [i][j][0] =(GLubyte)c;
         checkImage [i][j][1] =(GLubyte)c;
         checkImage [i][j][2] =(GLubyte)c;
      }
   }
}

void init(void)
{
   glewInit();
   glClearColor (0.0,0.0,0.0,0.0);
   glShadeModel(GL_FLAT);
   makeCheckImage();
   glPixelStorei(GL_UNPACK_ALIGNMENT,1);

   glGenBuffers(1,&pixelBuffer);
   glBindBuffer(GL_PIXEL_UNPACK_BUFFER,pixelBuffer);
   glBufferData(GL_PIXEL_UNPACK_BUFFER,
                3*checkImageWidth*checkImageHeight,
                checkImage,GL_STATIC_DRAW);
}
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glRasterPos2i(0,0);
   glBindBuffer(GL_PIXEL_UNPACK_BUFFER,pixelBuffer);
   glDrawPixels(checkImageWidth,checkImageHeight,GL_RGB,
                GL_UNSIGNED_BYTE,BUFFER_OFFSET(0));
   glFlush();
}
```

8.5.2 使用缓冲区对象提取像素数据

像素缓冲区还可以作为从OpenGL缓冲区所读取的像素的操作目标，并把这些像素传递回应用程序。对于像glReadPixels()和glGetTexImage()这样的函数，可以向它们传递当前绑定像素缓冲区内的一个偏移值，并用它们提取的像素更新缓冲区的数据值。

对于那些作为像素提取操作目标的缓冲区对象，它们的初始化和用法几乎与第8.5.1节描述的步骤相同，唯一的区别是所有与缓冲区对象相关的函数调用的缓冲区参数target必须是GL_PIXEL_PACK_ BUFFER。

当OpenGL函数完成提取像素之后，就可以调用glMapBuffer()函数（参见第2.7节）或glGetBuffer SubData()访问缓冲区对象中的数据值。在有些情况下，glGetBufferSubData()可能比glMapBuffer()具

有更高的传输效率。

示例程序8-6演示了使用一个像素缓冲区对象存储和访问通过调用glReadPixels()提取图像之后所渲染的像素。

示例程序8-6 使用缓冲区对象提取像素数据

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

GLuint pixelBuffer;
GLsizei imageWidth;
GLsizei imageHeight;
GLsizei numComponents =4;/*four components for GL_RGBA */
GLsizei bufferSize;

void
init(void)
{
   bufferSize =imageWidth *imageHeight *numComponents
              *sizeof(GLfloat);/*machine storage size */
   glGenBuffers(1,&pixelBuffer);
   glBindBuffer(GL_PIXEL_PACK_BUFFER,pixelBuffer);
   glBufferData(GL_PIXEL_PACK_BUFFER,bufferSize,
                NULL,/*allocate but don't initialize data */
                GL_STREAM_READ);
}

void
display(void)
{
   int i;
   GLsizei numPixels =imageWidth *imageHeight;

   /*Draw frame */

   glReadPixels(0,0,imageWidth,imageHeight,GL_RGBA,
                GL_FLOAT,BUFFER_OFFSET(0));
   GLfloat *pixels =glMapBuffer(GL_PIXEL_PACK_BUFFER,
                                GL_READ_ONLY);
   for (i =0;i <numPixels;++i ){
      /*insert your pixel processing here
        process(&pixels [i*numComponents] );
      */
   }
   glUnmapBuffer(GL_PIXEL_PACK_BUFFER);
}
```

8.6 提高像素绘图速度的技巧

正如读者所看到的那样，OpenGL提供了一组丰富的功能，用于读取、绘制和操纵像素数据。尽管这些功能非常实用，但是它们可有会降低程序的性能。下面是一些提高像素绘制速度的技巧：

- 为了实现最佳的性能，可以把所有的像素传输参数设置为它们的默认值，并把像素缩放率设置

为（1.0, 1.0）。

- 当像素绘制到帧缓冲区时，会执行一系列的片断操作（参见第10.2节）。为了实现最优化的性能，可以禁用所有这些片断操作。
- 在执行像素操作时，可以禁用其他那些开销较大的状态，例如纹理贴图或混合。
- 如果使用了与帧缓冲区相匹配的图像格式和类型，OpenGL实现就不需要把这些像素转换为与帧缓冲区匹配的格式。例如，如果把图像写入到一个RGB帧缓冲区（每个成分占据8位），就可以在调用glDrawPixels()函数时把format参数设置为RGB，并把type参数设置为UNSIGNED_BYTE。
- 在许多OpenGL实现中，无符号整数类型的图像格式比有符号整数类型的图像格式具有更快的处理速度。
- 绘制一个大型像素矩形的速度常常要比绘制几个小型像素矩形的速度要快，因为像素的传输开销将被众多数量的像素所分摊。
- 如果可能，可以使用较小的数据类型（例如，使用GL_UNSIGNED_BYTE）和更少的成分（例如，使用GL_LUMINANCE_ALPHA格式），以减少需要复制的数据。
- 像素传输操作（包括像素映射以及使用非默认的缩放、偏移和移位值）可能会降低性能。
- 如果需要每帧渲染相同的图像（例如作为背景），可以把它渲染为纹理四边形，就好比调用glDrawPixels()。把它作为纹理存储之后，数据下载到OpenGL中就只需要1次。关于纹理贴图的详细讨论，请参见第9章。

8.7 图像处理子集

图像处理子集是一组函数的集合，它们提供了额外的像素处理功能。使用图像处理子集，可以完成如下操作：

- 使用颜色查找表，用于替换像素值。
- 使用卷积，用于过滤图像。
- 使用颜色矩阵变换，进行颜色空间变换以及其他线性变换。
- 收集柱状图统计数据，以及关于图像的最小和最大颜色成分信息。

除了glPixelTransfer*()和glPixelMap*()所提供的功能之外，如果还需要其他像素处理功能，就应该使用图像处理子集。

图像处理子集是一个OpenGL扩展。如果一种OpenGL实现定义了GL_ARB_imaging标记，它就启用了图像处理子集，就可以使用接下来各节所描述的所有功能。如果这个标记并没有被定义，我们就无法使用所有这些功能。为了判断自己所使用的OpenGL实现是否支持图像处理子集，请参见第14.3节。

注意：尽管图像处理子集一直是一个OpenGL扩展，但其功能在OpenGL 3.0中废弃了，并且从OpenGL 3.1规范中删除了。

当像素传递给OpenGL或者从OpenGL读取时，它们就由图像处理子集启用的所有功能进行处理。受图像处理子集影响的函数包括：

- 绘制和读取像素矩形的函数：glReadPixels()、glDrawPixels()和glCopyPixels()。
- 用于定义纹理的函数：glTexImage1D()、glTexImage2D()、glCopyTexImage*D()。

图8-14显示了当像素传递到OpenGL以及从OpenGL读取时所经历的图像处理子集操作。图像处理子集的绝大多数功能都可以被启用或禁用，但是颜色矩阵变换例外，它总是被启用的。

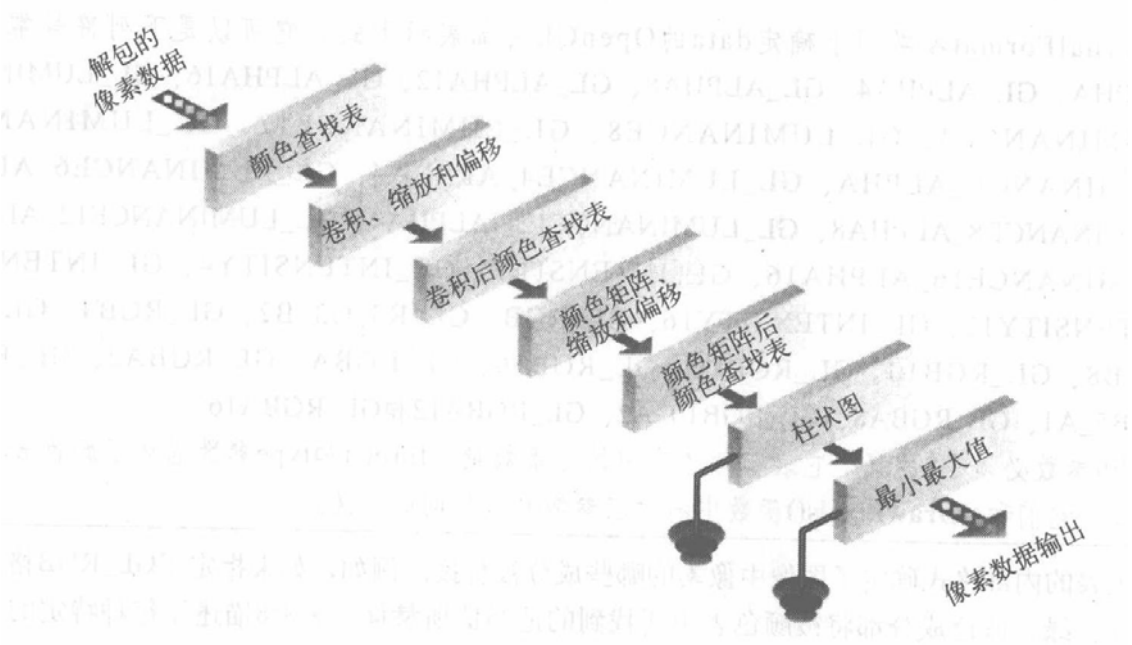


图8-14 图像处理子集操作

8.7.1 颜色表

颜色表是一种查找表，用于替换像素的颜色。在应用程序中，颜色表可以用于实现对比增强、过滤和图像均衡等效果。

可以使用3种不同类型的颜色查找表，它们是在不同的像素管线阶段使用的。表8-7显示了像素在管线中可以被各个颜色表所替换的位置。

表8-7 图像处理管线发生颜色表操作的位置

颜色表参数	在像素上的操作
GL_COLOR_TABLE	当它们进入图像处理管线时
GL_POST_CONVOLUTION_COLOR_TABLE	在卷积操作之后
GL_POST_COLOR_MATRIX_COLOR_TABLE	在颜色矩阵变换之后

可以调用glEnable()函数，使用表8-7所列出的各个参数，单独启用各个颜色表。

指定颜色表

指定颜色表的方法类似于指定一维图像。如图8-14所示，共有3种颜色表可以用于更新像素值。可以用glColorTable()来定义各个颜色表。

void **glColorTable**(GLenum *target*, GLenum *internalFormat*,
GLsizei *width*, GLenum *format*, GLenum *type*,
const GLvoid *data);

当target设置为GL_COLOR_TABLE、GL_POST_CONVOLUTION_COLOR_TABLE或GL_POST_COLOR_MATRIX_COLOR_TABLE时，这个函数就用于定义指定的颜色表。如果target设置为GL_PROXY_COLOR_TABLE、GL_PROXY_POST_CONVULOTION_ COLOR_TABLE或GL_PROXY_POST_ COLOR_MATRIX_COLOR_TABLE，glColorTable()就用于验证指定的颜色表在可用的资源中是否适用。

internalFormat参数用于确定data的OpenGL内部表示形式。它可以是下列符号常量之一：GL_ALPHA、GL_ALPHA4、GL_ALPHA8、GL_ALPHA12、GL_ALPHA16、GL_LUMINANCE、GL_LUMINANCE4、GL_LUMINANCE8、GL_LUMINANCE12、GL_LUMINANCE16、GL_LUMINANCE_ALPHA、GL_LUMINANCE4_ALPHA4、GL_LUMINANCE6_ALPHA2、GL_LUMINANCE8_ALPHA8、GL_LUMINANCE12_ ALPHA4、GL_LUMINANCE12_ALPHA12、GL_LUMINANCE16_ALPHA16、GL_INTENSITY、GL_INTENSITY4、GL_INTENSITY8、GL_INTENSITY12、GL_INTENSITY16、GL_RGB、GL_R3_G3_B2、GL_RGB4、GL_RGB5、GL_RGB8、GL_RGB10、GL_RGB12、GL_RGB16、GL_RGBA、GL_RGBA2、GL_RGBA4、GL_RGB5_A1、GL_RGBA8、GL_RGB10_A2、GL_RGBA12和GL_RGBA16。

width参数必须是2的幂，它表示颜色表中的像素数量。format和type参数描述了颜色表数据的格式和类型。它们和**glDrawPixels()**函数中的对应参数具有相同的含义。

颜色表的内部格式确定了图像中像素的哪些成分被替换。例如，如果指定了GL_RGB格式，每个像素中红、绿、蓝色成分都将被颜色表中所找到的适当值所替换。表8-8描述了每种特定的基本内部格式所替换的像素成分。

表8-8 颜色表像素替换

基本内部格式	红色成分	绿色成分	蓝色成分	alpha成分
GL_ALPHA	不改变	不改变	不改变	A_t
GL_LUMINANCE	L_t	L_t	L_t	不改变
GL_LUMINANCE_ALPHA	L_t	L_t	L_t	A_t
GL_INTENSITY	I_t	I_t	I_t	I_t
GL_RGB	R_t	G_t	B_t	不改变
GL_RGBA	R_t	G_t	B_t	A_t

在表8-8中，L_t表示已定义的颜色表的亮度项，它只影响红、绿、蓝成分。I_t表示强度项，它对红、绿、蓝和alpha成分施加相同的影响。

对图像应用了适当的颜色表之后，像素可以进行缩放和偏移。在这些操作之后，它们的值被截取在[0，1]的范围之内。可以使用glColorTableParameter*()函数，为每个颜色表设置GL_COLOR_TABLE_SCALE和GL_COLOR_TABLE_BIAS因子。

void **glColorTableParameter**{if}v(GLenum *target*, GLenum *pname*,
TYPE **param*);

为每个颜色表设置GL_COLOR_TABLE_SCALE和GL_COLOR_TABLE_BIAS参数。target可以是GL_COLOR_TABLE、GL_POST_CONVOLUTION_COLOR_TABLE或GL_POST_COLOR_MATRIX_COLOR_TABLE，它指定了需要设置缩放和偏移值的颜色表。

pname参数的值可以是GL_COLOR_TABLE_SCALE和GL_COLOR_TABLE_BIAS。param指向一个包含4个值的数组，它们分别表示红、绿、蓝和alpha成分。

示例程序8-7显示了如何使用颜色表对图像进行反转。这个程序创建颜色表，并用每个颜色成分的反转值来替换它。

示例程序8-7 使用颜色表进行像素替换：colortable.c

```
extern GLubyte*readImage(const char*,GLsizei*,GLsizei*);

GLubyte *pixels;
GLsizei width,height;

void init(void)
{
   int i;
   GLubyte colorTable [256][3];

   pixels =readImage("Data/leeds.bin ",&width,&height);
   glPixelStorei(GL_UNPACK_ALIGNMENT,1);
   glClearColor(0,0,0,0);
   /*Set up an inverting color table */
   for (i =0;i <256;++i){
      colorTable [i][0] =255 -i;
      colorTable [i][1} =255 -i;
      colorTable [i][2] =255 -i;
   }
   glColorTable(GL_COLOR_TABLE,GL_RGB,256,GL_RGB,
      GL_UNSIGNED_BYTE,colorTable);
   glEnable(GL_COLOR_TABLE);
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glRasterPos2i(1,1);
   glDrawPixels(width,height,GL_RGB,GL_UNSIGNED_BYTE,
      pixels);
   glFlush();
}
```

注意：示例程序8-5引入了一个新函数readImage()，这是为了简化这个示例程序。一般情况下，我们需要一个函数读取程序所请求的图像文件格式。下面列出了readImage()函数所理解的文件格式。在这种文件中，数据是按照线性顺序排列的。

- 图像的宽度，存储为GLsizei类型。
- 图像的高度，存储为GLsizei类型。
- width · height RGB三元组，每个颜色成分都以GLubyte类型存储。

除了在应用程序中显式地指定颜色表之外，我们可能想用帧缓冲区内创建的图像来定义颜色表。glCopyColorTable()函数允许指定从帧缓冲区读取一行像素，并用它来定义颜色表。

---

void **glCopyColorTable**(GLenum *target*, GLenum *internalFormat*,
GLint *x*, GLint *y*, GLsizei *width*);

创建一个颜色表，使用帧缓冲区的数据来定义颜色表的元素。像素是从当前的GL_READ_BUFFER读取的，它的处理方式就像调用了**glCopyPixels()**但没有执行最后的转换操作一样。**glPixelTransfer*()**执行的设置将会被应用。

target参数必须设置为**glColorTable()**函数所使用的目标之一。internalFormat参数使用和**glColorTable()**的internalFormat参数相同的符号常量。定义颜色数组的那行像素（包含了width个像素）是从帧缓冲区中（x,y）开始的。

替换颜色表的一部分或全部

如果我们想替换颜色表的一部分，可以使用颜色子表函数在颜色表中的一个任意部分重新加载新值。

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

用data存储的值替换从start到start + count−1的颜色表项。

target参数是GL_COLOR_TABLE、GL_POST_CONVOLUTION_COLOR_TABLE或GL_POST_COLOR_MATRIX_COLOR_TABLE。format和type参数与**glColorTable()**中的对应参数相同，它们描述了存储在data中的像素值。

void **glCopyColorSubTable**(GLenum *target*, GLsizei *start*, GLint *x*,
GLint *y*, GLsizei *count*);

用帧缓冲区中从位置（x,y）开始的那一行中的count个颜色像素值替换颜色表中从start + count− 1的颜色表项。这些像素将被转换为源颜色表所使用的内部格式。

查询颜色表的值

存储在颜色表中的像素值可以使用glGetColorTable()函数来提取。关于这方面的更多细节，请参阅第B.1节。

颜色表代理

颜色表代理提供了一种方式，向OpenGL查询是否有足够的资源来存储颜色表。如果glColorTable()函数被调用时使用了下面这些代理目标之一，OpenGL就判断被查询的颜色表资源是否可用：

- GL_PROXY_COLOR_TABLE
- GL_PROXY_POST_CONVOLUTION_COLOR_TABLE
- GL_PROXY_POST_COLOR_MATRIX_COLOR_TABLE

如果颜色表资源不可用，那么它的宽度、格式和成分分辨率值都设置为0。为了检查颜色表是否可用，可以查询前面所提到的状态值之一。例如：

```
glColorTable(GL_PROXY_COLOR_TABLE, GL_RGB, 1024, GL_RGB,
   GL_UNSIGNED_BYTE, null);
glGetColorTableParameteriv(GL_PROXY_COLOR_TABLE,
   GL_COLOR_TABLE_WIDTH, &width);
if (width == 0)
   /* color table didn't fit as requested */
```

关于glGetColorTableParameter*()函数的更多细节，请参阅附录B.1节。

8.7.2 卷积

卷积是一种像素过滤器，用像素本身以及邻近像素的加权平均值来替换这个像素。卷积的应用例

子包括模糊和锐化图像、查找图像边缘以及调整图像的对比度。

图8-15显示了像素P_{00}以及相关的像素是如何由一个3×3的卷积过滤器进行处理并产生像素P'_{11}的。

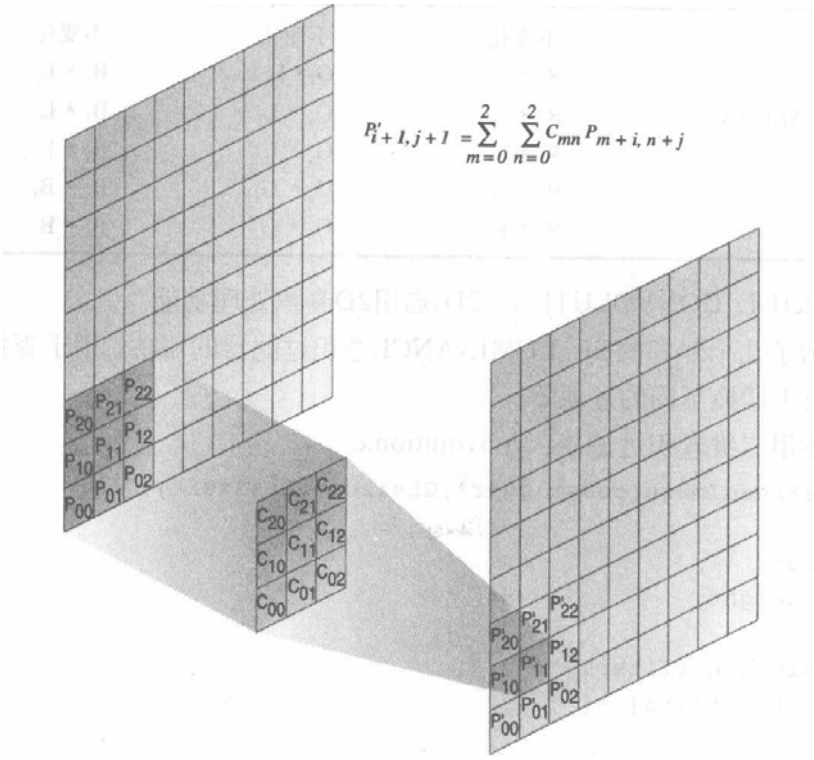


图8-15 像素卷积操作

卷积是像素权值的数组，并且只能对RGBA像素进行操作。卷积过滤器（又称卷积核）就是二维的像素权值的数组。在经过卷积处理的输出图像中，每个像素是通过把输入图像的一组像素与卷积核中的像素权值相乘后并把结果相加而创建的。例如，在图8-15中，像素P'_{11}是通过把输入图像的9个像素与卷积过滤器的9个权值相乘，然后再把各个乘积相加而得的。

---

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

定义一个二维的卷积过滤器，其中target参数必须是GL_CONVOLUTION_2D。

internalFormat参数定义了卷积操作将作用于哪个像素成分，它可以是glColorTable()的internalFormat参数可以使用的38个符号常量之一。

width和height参数指定了卷积过滤器的大小（以像素为单位）。可以用glGetConvolutionParameter*()函数查询卷积过滤器的最大宽度和最大高度。详见B.1节。

和glDrawPixels()函数一样，这个函数的format和type参数指定了image存储的像素格式。

与颜色表类似，卷积过滤器的内部格式决定了将对图像的哪些成分进行操作。表8-9描述了影响像素的一些不同的基本过滤器格式。R_s、G_s、B_s和A_s表示源像素的颜色成分，L_f表示GL_LUMINANCE过滤器的亮度值，I_f对应于GL_INTENSITY过滤器的强度值。最后，R_f、G_f、B_f和A_f分别表示卷积过滤器的红、绿、蓝和alpha成分。

表8-9 卷积过滤器如何影响RGBA像素成分

基本过滤器格式	红色结果	绿色结果	蓝色结果	alpha结果
GL_ALPHA	不变化	不变化	不变化	$A_s * A_f$
GL_LUMINANCE	$R_s * L_f$	$G_s * L_f$	$B_s * L_f$	不变化
GL_LUMINANCE_ALPHA	$R_s * L_f$	$G_s * L_f$	$B_s * L_f$	$A_s * A_f$
GL_INTENSITY	$R_s * I_f$	$G_s * I_f$	$B_s * I_f$	$A_s * I_f$
GL_RGB	$R_s * R_f$	$G_s * G_f$	$B_s * B_f$	不变化
GL_RGBA	$R_s * R_f$	$G_s * G_f$	$B_s * B_f$	$A_s * A_f$

可以使用glEnable(GL_CONVOLUTION_2D)启用2D卷积处理功能。

示例程序8-8演示了几个3×3的GL_LUMINANCE卷积过滤器的用法，用于查找图像的边缘。“h”、“l”和“v”键可以用于切换不同的过滤器。

示例程序8-8 使用二维卷积过滤器：convolution.c

```
extern GLubyte*readImage(const char*,GLsizei*,GLsizei*);

GLubyte *pixels;
GLsizei width,height;

/*Define convolution filters */
GLfloat horizontal [3][3] ={
   {0,-1,0 },
   {0,1,0 },
   {0,0,0 }
};

GLfloat vertical [3][3] ={
   {0,0,0 },
   {-1,1,0 },
   {0,0,0 }
};

GLfloat laplacian [3][3] ={
   {-0.125,-0.125,-0.125 },
   {-0.125,1.0,-0.125 },
   {-0.125,-0.125,-0.125 }
};

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glRasterPos2i(1,1);
   glDrawPixels(width,height,GL_RGB,GL_UNSIGNED_BYTE,
      pixels);
   glFlush();
}

void init(void)
{
```

```
    pixels =readImage("Data/leeds.bin ",&width,&height);
    glPixelStorei(GL_UNPACK_ALIGNMENT,1);
    glClearColor(0.0,0.0,0.0,0.0);

    printf("Using horizontal filter \n ");
    glConvolutionFilter2D(GL_CONVOLUTION_2D,GL_LUMINANCE,3,3,
       GL_LUMINANCE,GL_FLOAT,horizontal);
    glEnable(GL_CONVOLUTION_2D);
}

void keyboard(unsigned char key,int x,int y)
{
    switch (key){
       case 'h ' :
          printf("Using horizontal filter \n ");
          glConvolutionFilter2D(GL_CONVOLUTION_2D,GL_LUMINANCE,
             3,3,GL_LUMINANCE,GL_FLOAT,horizontal );
          break;

       case 'v ' :
          printf("Using vertical filter \n ");
          glConvolutionFilter2D(GL_CONVOLUTION_2D,GL_LUMINANCE,
             3,3,GL_LUMINANCE,GL_FLOAT,vertical );
          break;

       case 'l ':
          printf("Using laplacian filter \n ");
          glConvolutionFilter2D(GL_CONVOLUTION_2D,GL_LUMINANCE,
             3,3,GL_LUMINANCE,GL_FLOAT,laplacian );
          break;
       case 27:/*Escape Key */
          exit(0)
          break;
}
    glutPostRedisplay();
}
```

和颜色表一样，可以使用来自帧缓冲区的像素值指定一个卷积过滤器。glCopyConvolutionFilter2D()从当前的GL_READ_BUFFER复制一块像素矩形，用它来定义一个卷积过滤器。如果internalFormat参数指定为GL_LUMINANCE或GL_INTENSITY，这个函数就使用像素的红色成分定义这个卷积过滤器的值。

---

void **glCopyConvolutionFilter2D**(GLenum *target*,
GLenum *internalFormat*,
GLint *x*, GLint *y*,
GLsizei *width*, GLsizei *height*);

定义一个二维卷积过滤器，用颜色帧缓冲区中的像素对它进行初始化。target参数必须是GL_CONVOLUTION_2D，internalFormat必须设置为**glConvolutionFilter2D()**所使用的内部格式之一。

这个像素矩阵的左下角像素是 (x, y)，像素数量为width × height，它是从帧缓冲区读取的，并转换为指定的内部格式。

指定可分离的二维卷积过滤器

如果卷积过滤器可以用两个一维过滤器的外积（outer product）表示，那么它就可以进行分离。

glSeparableFilter2D()函数用于指定两个一维过滤器，用它们来表示一个可分离的二维卷积过滤器。和glConvolutionFilter()函数一样，卷积过滤器的内部格式决定了图像的像素是如何进行处理的。

---

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

定义了一个二维的可分离卷积过滤器。target参数必须设置为GL_SEPARABLE_2D。internalFormat参数所使用的值与**glConvolutionFilter2D()**函数的对应参数可以使用的值相同。

width指定了row数组中的像素数量。类似地，height指定了column数组中的像素数量。type和format参数定义了row和column的存储格式，其方式与**glConvolutionFilter2D()**相同。

可以用**glEnable(GL_SEPARABLE_2D)**启用卷积过滤器，使用二维可分离的卷积过滤器。如果同时指定了GL_CONVOLUTION_2D和GL_SEPARABLE_2D，则优先使用前者。

如果绑定了一个解包的像素缓冲区对象，并且width、height、format和type的组合加上绑定的缓冲区对象的指定偏移量，导致内存访问超出了创建缓冲区对象时所分配的内存范围，就会设置GL_INVALID_OPERATION错误。

例如，可以通过为row和column各指定一个一维过滤器[−1/2，1，−1/2]，创建一个3×3的卷积过滤器，表示一个可分离的GL_LUMINANCE卷积过滤器。OpenGL将使用这两个一维过滤器来计算源图像的卷积过滤器，其方式相当于根据下面的外积来计算完整的二维卷积过滤器：

$$\begin{bmatrix} -1/2 \\ 1 \\ -1/2 \end{bmatrix} \begin{bmatrix} -1/2 & 1 & -1/2 \end{bmatrix} = \begin{bmatrix} 1/4 & -1/2 & 1/4 \\ -1/2 & 1 & -1/2 \\ 1/4 & -1/2 & 1/4 \end{bmatrix}$$

从计算的角度而言，使用可分离的卷积过滤器要比使用不可分离的卷积过滤器具有更高的效率。

一维卷积过滤器

一维卷积过滤器和二维卷积过滤器基本相同，只是这种卷积过滤器的height参数设置为1。但是，它们只影响一维纹理的定义（详见第9.2.3节）。

---

void **glConvolutionFilter1D**(GLenum *target*, GLenum *internalFormat*,
GLsizei *width*, GLenum *format*,
GLenum *type*, const GLvoid **image*);

指定一个一维卷积过滤器。target必须设置为GL_CONVOLUTION_1D。width指定了过滤器中的像素数量。internalFormat、format和type参数与glConvolutionFilter2D()函数中的对应参数具有相同的含义。image指向用于定义这个卷积过滤器的一维图像。

可以使用**glEnable(GL_CONVOLUTION_1D)**来启用一维卷积过滤器。

我们可能想用帧缓冲区所产生的值来指定这个卷积过滤器。**glCopyConvolutionFilter1D()**函数从当前的GL_READ_BUFFER中复制一行像素，并把它们转换为指定的内部格式，然后用它们来定义这个卷积过滤器。

如果绑定了一个解包的像素缓冲区对象，并且width、format和type的组合加上绑定的缓冲区对象的指定偏移量，导致内存访问超出了创建缓冲区对象时所分配的内存范围，就会设置GL_INVALID_OPERATION错误。

---

void **glCopyConvolutionFilter1D**(GLenum *target*,
GLenum *internalFormat*, GLint *x*,
GLint *y*, GLsizei *width*);

使用来自帧缓冲区的像素值定义一个一维卷积过滤器。**glCopyConvolutionFilter1D()**从位置（x，y）复制width个像素，并把它们转换为指定的内部格式。

指定了一个一维卷积过滤器之后，就可以对它进行缩放和偏移。缩放和偏移值是用glConvolutionParameter*()函数指定的。在缩放或偏移之后，卷积过滤器的值不会进行截取。

---

void **glConvolutionParameter**{if}(GLenum *target*, GLenum *pname*,
TYPE param);

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

设置参数，控制卷积过滤器的执行方式。target参数必须是GL_CONVOLUTION_1D、GL_CONVOLUTION_2D或GL_SEPARABLE_2D。

pname参数必须是GL_CONVOLUTION_BORDER_MODE、GL_CONVOLUTION_FILTER_SCALE或GL_CONVOLUTION_FILTER_BIAS。如果pname指定为GL_CONVOLUTION_BORDER_MODE，这个函数就用于定义卷积边界模式。在这种情况下，params必须是GL_REDUCE、GL_CONSTANT_BORDER或GL_REPLICATE_BORDER。如果params设置为GL_CONVOLUTION_FILTER_SCALE或GL_CONVOLUTION_FILTER_ BIAS，params必须指定一个包含4个值的数组，分别表示红、绿、蓝和alpha成分。

卷积边界模式

位于图像边缘的像素卷积的处理方式和图像内部像素的处理方式不同。它们的卷积将根据卷积边界模式进行修改。计算边界卷积的选项共有3种：

- GL_REDUCE模式使最终图像在各个方向上均收缩，收缩量就是卷积过滤器的大小。最终图像的宽度是（width−W_f），高度是（height−H_f），其中W_f和H_f分别表示卷积过滤器的宽度和高度。如果按照这种方法计算出来的最终图像的宽度或高度为0或负值，就不会产生任何输出，也不会产生任何错误。
- GL_CONSTANT_BORDER用一个常量像素值来表示源图像边界之外的像素，计算边界像素的卷积。这个常量像素值是用glConvolutionParameter*()函数设置的。最终图像的大小与源图像的大小一致。
- GL_REPLICATE_BORDER计算卷积的方式与GL_CONSTANT_BORDER模式相同，除了用图像的最外面一行（或一列）的像素来计算图像边界像素的卷积。最终图像的大小与源图像的大小一致。

后卷积操作

在卷积操作完成之后，最终图像的像素可能要进行缩放和偏移，并截取在范围[0,1]之内。缩放和偏移值是通过调用glPixelTransfer*()函数指定的，分别使用GL_POST_CONVOLUTION_*_SCALE或

GL_POST_CONVOLUTION_*_BIAS。如果在glColorTable( )中指定了GL_POST_CONVOLUTION_COLOR_TABLE，它就允许我们用一个颜色查找表来替换像素成分。

8.7.3 颜色矩阵

为了对像素值进行颜色空间变换和线性变换，图像处理子集支持一个4×4的矩阵堆栈，可以通过设置glMatrixMode(GL_COLOR)来选择颜色矩阵。例如，为了从RGB颜色空间转换为CMY（青、洋红、黄）颜色空间，可以采用下面的做法：

```
GLfloat rgb2cmy[16] = {
   -1,  0,  0, 0,
    0, -1,  0, 0,
    0,  0, -1, 0,
    1,  1,  1, 1
};

glMatrixMode(GL_COLOR);  /* enter color matrix mode */
glLoadMatrixf(rgb2cmy);
glMatrixMode(GL_MODELVIEW);  /* back to modelview mode */
```

注意：记住OpenGL的矩阵是以列主序格式存储的。关于在OpenGL中使用矩阵的更多细节，请参阅第3.1.2节。

颜色矩阵变换至少拥有2个矩阵元素（关于确定颜色矩阵堆栈深度的细节，请参阅B.1节）。和图像处理子集的其他特性不同，只要启用了图像处理子集，颜色矩阵变换总是会被执行，我们无法禁用这个特性。

示例程序8-9演示了如何使用颜色矩阵交换一幅图像的红色和绿色成分。

示例程序8-9 使用颜色矩阵交换颜色成分：colormatrix.c

```
extern GLubyte*readImage(const char*,GLsizei*,GLsizei*);

GLubyte *pixels;
GLsizei width,height;
void init(void)
{
   /*Specify a color matrix to reorder a pixel 's components
   *from RGB to GBR */
   GLfloat m [16] ={
     0.0,1.0,0.0,0.0,
     0.0,0.0,1.0,0.0,
     1.0,0.0,0.0,0.0,
     0.0,0.0,0.0,1.0
   };

   pixels =readImage("Data/leeds.bin ",&width,&height);
   glPixelStorei(GL_UNPACK_ALIGNMENT,1);
   glClearColor(0.0,0.0,0.0,0.0);

   glMatrixMode(GL_COLOR);
   glLoadMatrixf(m);
   glMatrixMode(GL_MODELVIEW);
}
```

```
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glRasterPos2i(1,1);
   glDrawPixels(width,height,GL_RGB,GL_UNSIGNED_BYTE,
      pixels);
   glFlush();
}
```

后颜色矩阵变换操作

和后卷积操作相似，在这个阶段，像素也可能进行缩放和偏移。可以使用glPixelTransfer*()函数，并在参数中使用GL_POST_COLOR_MATRIX_*_SCALE或GL_POST_COLOR_MATRIX_*_BIAS，定义用于后颜色矩阵操作的缩放值或偏移值。像素值在进行缩放和偏移之后将在[0, 1]的范围之内进行截取。

8.7.4 柱状图

使用图像处理子集，可以收集和图像有关的统计信息。柱状图确定了一幅图像的颜色分布。例如，可以用这个信息决定如何平衡一幅图像的对比度。

glHistogram()函数指定了希望进行柱状图处理的图像成分，并确定了只想收集统计信息还是想继续对图像进行处理。为了收集柱状图统计信息，必须调用glEnable(GL_ HISTOGRAM)。

与第8.7.1节描述的颜色表类似，glHistogram()函数也可以使用代理机制查询是否有足够的资源来存储用户所请求的柱状图。如果资源不足，柱状图的宽度、格式和成分分辨率都设置为0。可以使用glGetHistogramParameter()函数查询柱状图代理。关于这个函数的详细信息，请参阅附录B.1节。

---

void **glHistogram**(GLenum *target*, GLsizei *width*,
GLenum *internalFormat*, GLboolean *sink*);

定义了应该如何存储一幅图像的柱状图数据。target参数必须设置为GL_HISTOGRAM或GL_PROXY_HISTOGRAM。width参数指定了柱状图表中项的数量，它的值必须是2的整数次方。

internalFormat参数定义了柱状图数据应该如何存储。它允许使用的值包括GL_ALPHA、GL_ALPHA4、GL_ALPHA8、GL_ALPHA12、GL_ALPHA16、GL_LUMINANCE、GL_LUMINANCE4、GL_LUMINANCE8、GL_LUMINANCE12、GL_LUMINANCE16、GL_LUMINANCE_ALPHA、GL_LUMINANCE4_ALPHA4、GL_LUMINANCE6_ALPHA2、GL_LUMINANCE8_ALPHA8、GL_LUMINANCE12_ ALPHA4、GL_LUMINANCE12_ ALPHA12、GL_LUMINANCE16_ALPHA16、GL_RGB、GL_RGB2、GL_RGB4、GL_RGB5、GL_RGB8、GL_RGB10、GL_RGB12、GL_RGB16、GL_RGBA、GL_RGBA2、GL_RGBA4、GL_RGB5_A1、GL_RGBA8、GL_RGB10_A2、GL_RGB12和GL_RGB16。这个列表并不包括GL_INTENSITY*值，这点与glColorTable()所接受的值列表不同。

sink参数表示这些像素应该继续进入到图像处理管线的最小最大值阶段还是应该被丢弃。

使用**glDrawPixel()**函数把像素传递给图像处理子集之后，可以使用**glGetHistogram()**函数提取柱状图的结果。**glGetHistogram()**函数除了返回柱状图的值之外，还可以用于重置柱状图的内部存储。也可以使用**glResetHistogram()**函数重置柱状图的内部存储，这个函数将在稍后解释。

---

void **glGetHistogram**(GLenum *target*, GLboolean *reset*, GLenum *format*,
GLenum *type*, GLvoid **values*);

返回已收集的柱状图统计信息。target参数必须是GL_HISTOGRAM。reset参数指定了内部的柱状图表是否应该被清除。

format参数和type参数指定了values的存储类型，以及柱状图数据应该如何返回给应用程序。它们所接受的值与**glDrawPixels()**函数的对应参数相同。

示例程序8-10计算一幅图像的柱状图信息，并在窗口中绘制分布结果。在这个例子中，“s”键显示了sink参数的效果，它控制像素是否传递给后续的图像管线操作。

示例程序8-10 计算和绘制一幅图像的柱状图信息：histogram.c

```
#define HISTOGRAM_SIZE 256 /*Must be a power of 2 */

extern GLubyte*readImage(const char*,GLsizei*,GLsizei*);

GLubyte *pixels;
GLsizei width,height;

void init(void)
{
   pixels =readImage("Data/leeds.bin ",&width,&height);
   glPixelStorei(GL_UNPACK_ALIGNMENT,1);
   glClearColor(0.0,0.0,0.0,0.0);

   glHistogram(GL_HISTOGRAM,HISTOGRAM_SIZE,GL_RGB,GL_FALSE);
   glEnable(GL_HISTOGRAM);
}

void display(void)
{
   int i;
   GLushort values [HISTOGRAM_SIZE][3];

   glClear(GL_COLOR_BUFFER_BIT);
   glRasterPos2i(1,1);
   glDrawPixels(width,height,GL_RGB,GL_UNSIGNED_BYTE,
      pixels);
   glGetHistogram(GL_HISTOGRAM,GL_TRUE,GL_RGB,
      GL_UNSIGNED_SHORT,values);
   /*Plot histogram */

   glBegin(GL_LINE_STRIP);
   glColor3f(1.0,0.0,0.0);
   for (i =0;i <HISTOGRAM_SIZE;i++)
      glVertex2s(i,values [i][0]);
   glEnd();

   glBegin(GL_LINE_STRIP);
   glColor3f(0.0,1.0,0.0);
   for (i =0;i <HISTOGRAM_SIZE;i++)
      glVertex2s(i,values [i][1]);
```

```
    glEnd();

    glBegin(GL_LINE_STRIP);
    glColor3f(0.0,0.0,1.0);
    for (i =0;i <HISTOGRAM_SIZE;i++)
       glVertex2s(i,values [i][2]);
    glEnd();
    glFlush();
}

void keyboard(unsigned char key,int x,int y)
{
    static GLboolean sink =GL_FALSE;

    switch (key){
       case 's ':
          sink =!sink;
          glHistogram(GL_HISTOGRAM,HISTOGRAM_SIZE,GL_RGB,
              sink);
          break;
       case 27:/*Escape Key */
          exit(0);
          break;
}
    glutPostRedisplay();
}
```

glResetHistogram()函数将丢弃已收集的柱状图信息，并不提取它的值。

---

void **glResetHistogram**(GLenum *target*);

把柱状图计数器重置为0。target参数必须是GL_HISTOGRAM。

8.7.5 最小最大值

glMinmax()函数用于计算一个像素矩形中最小像素成分值和最大像素成分值。和glHistogram()函数一样，在计算了最小最大值之后，可以选择继续进行图像管线处理或者选择丢弃这些像素。

---

void **glMinmax**(GLenum *target*, GLenum *internalFormat*,
GLboolean *sink*);

计算一幅图像的最小像素值和最大像素值。target参数必须为GL_MINMAX。

internalFormat参数指定了应该计算哪些颜色成分的最小值和最大值。**glMinmax**()接受的internalFormat参数和**glHistogram**()函数所接受的值相同。

如果sink参数指定为GL_TRUE，那么这些像素将被丢弃，并不会写入到帧缓冲区。如果这个参数设置为GL_FALSE，图像就会进行渲染。

---

glGetMinmax()函数用于提取计算产生的最小值和最大值。和glHistogram()函数类似，最小值和最大值的内部值可以在访问的时候重置。

---

void **glGetMinmax**(GLenum *target*, GLboolean *reset*, GLenum *format*,
GLenum *type*, GLvoid **values*);

返回最小最大值操作的结果。target参数必须是GL_MINMAX。如果reset参数设置为GL_TRUE，

最小值和最大值就重置为它们的初始值。format参数和type参数描述了values中返回的最小最大值数据的格式，它们使用的值和**glDrawPixels()**函数的对应参数使用的值相同。

示例程序8-11演示了如何使用glMinmax()函数以GL_RGB格式计算最小像素值和最大像素值。最小最大值功能必须通过调用glEnable(GL_MINMAX)来启用。

glMinMax()函数的values数组中所返回的最小值和最大值根据成分进行分组。例如，如果把format参数指定为GL_RGB，values数组的前3个值依次表示被处理像素的最小红色、绿色和蓝色值，接着是最大红色、绿色和蓝色值。

示例程序8-11 计算最小和最大像素值：minmax.c

```
extern GLubyte*readImage(const char*,GLsizei*,GLsizei*);

GLubyte *pixels;
GLsizei width,height;

void init(void)
{
   pixels =readImage("Data/leeds.bin " ,&width,&height);
   glPixelStorei(GL_UNPACK_ALIGNMENT,1);
   glClearColor(0.0,0.0,0.0,0.0);
   glMinmax(GL_MINMAX,GL_RGB,GL_FALSE);
   glEnable(GL_MINMAX);
}

void display(void)
{
   GLubyte values [6];

   glClear(GL_COLOR_BUFFER_BIT);
   glRasterPos2i(1,1);
   glDrawPixels(width,height,GL_RGB,GL_UNSIGNED_BYTE,
      pixels);
   glGetMinmax(GL_MINMAX,GL_TRUE,GL_RGB,GL_UNSIGNED_BYTE,
      values);
   glFlush();
   printf("Red :min =%d max =%d \n ",values [0],values [3]);
   printf("Green:min =%d max =%d \n ",values [1],values [4]);
   printf("Blue :min =%d max =%d \n ",values [2],values [5]);
}
```

尽管可以使用glGetMinmax()函数在提取最小最大值时重置它们的值，但是也可以在任何时候显式地调用glResetMinmax()函数来重置最小最大值内部表的值。

void **glResetMinmax**(GLenum *target*);

把最小最大值重置为它们的初始值。target参数必须为GL_MINMAX。

第9章 纹理贴图

本章目标

- 理解纹理贴图在场景渲染中的作用。
- 指定压缩和未压缩格式的纹理图像。
- 当纹理图像应用于片断时控制它的过滤。
- 用纹理对象创建和管理纹理图像，并在可行的情况下，控制这些纹理对象的高性能工作集。
- 指定纹理和片断的颜色值是如何进行组合的。
- 提供纹理坐标，描述纹理图像应该如何映射到场景中的物体。
- 自动生成纹理坐标，产生诸如轮廓图和环境纹理这样的效果。
- 使用多重纹理（线性纹理单位），执行复杂的纹理操作。
- 使用纹理组合器函数，对纹理、片断和常量颜色值执行数学运算。
- 在应用纹理之后，使用辅助颜色对片断进行处理。
- 指定用于点块纹理处理的纹理。
- 使用纹理矩阵，对纹理坐标进行变换。
- 使用深度纹理，渲染带阴影的物体。

到目前为止，每个图元在绘制时，或者使用一种颜色进行单调着色，或者根据它的各个顶点的颜色进行平滑着色。也就是说，到目前为止我们还没有使用过纹理贴图。假设有这样一个情况：我们想绘制一面很大的砖墙，其中的每块砖都画成一个单独的多边形。如果不使用纹理贴图，这面墙（本质上是一个很大的矩形）可能需要通过绘制几千个矩形来完成。并且，这些砖块看上去可能很小，显得过于平滑和规则，不像现实生活中的砖块。

纹理贴图允许把一幅砖墙图像（例如，通过对砖墙照片进行扫描而得）映射到一个多边形的表面上，并把整面砖墙画成单个多边形。纹理贴图能够保证当这个多边形进行变换或渲染时，映射到多边形表面的图像也能够表现出正确的行为。例如，假设我们在透视投影模式下观察这面墙。当它远离观察者时，墙面上的砖块看上去就显得更小一些。纹理贴图的其他用途还包括在大型多边形上描绘植被，表示飞行模拟中的地面；用纹理图像制作墙纸；利用纹理使多边形看上去就像是自然世界的物质，如大理石、木材和布等。纹理贴图的用途几乎是无穷无尽的。纹理贴图可以应用于所有的图元，如点、直线、多边形、位图和图像（尽管最为自然的想法是把它应用到多边形上）。彩图6、8、18～21以及彩图24～32都演示了纹理贴图的用途。

纹理贴图是一个相当大的主题，并且具有相当程度的复杂性。在使用纹理贴图时，必须做出一些编程选择。初学者很可能会本能地把纹理理解成二维图像，但是纹理也可以是一维的，甚至是三维的。可以把纹理映射到由一组多边形构成的表面上，也可以把它贴到曲面上，还可以在一个、二个或三个方向上（取决于纹理的维度）重复应用同一个纹理来覆盖整个表面。另外，可以把纹理图像自动映射到物体上，用它表示被观察物体的轮廓线或者其他属性。有光泽的物体也可以进行纹理贴图，当它们位于房间或其他环境的中央时，它们的表面就可以反射周围的物体。最后，纹理也可以按不同的方式

应用到物体的表面。它可以直接画到物体的表面（就像表面上的贴花一样），调整表面的颜色。或者，把纹理颜色与表面颜色进行混合。如果读者是第一次接触纹理贴图，会发觉本章的讨论进行得相当快速。关于纹理贴图的参考资料，可以参阅Alan Watt所著的《3D Computer Graphics》（Addison-Wesley，1999）中有关纹理贴图的章节。

简单地说，纹理就是矩形的数据数组。例如，颜色数据、亮度数据、颜色和alpha数据。纹理数组中的单个值常常称为纹理单元（texel）。纹理贴图之所以复杂，是因为矩形的纹理可以映射到非矩形的区域，并且必须以合理的方式实现。

图9-1显示了纹理贴图过程。左侧的图显示了整个纹理，黑色的轮廓表示一个四边形，它的各个角映射到纹理中的这些位置。当这个四边形在屏幕上显示时，它有可能因为各种变换（旋转、移动、缩放和投影）而扭曲。右侧的图显示了这个经过纹理贴图的四边形在经过各种变换之后在屏幕上可能呈现的样子（注意，这个多边形是凹的，如果事先不进行分格化，OpenGL可能无法对它进行正确的渲染。关于对多边形进行分格化的详细信息，请参阅第11章）。

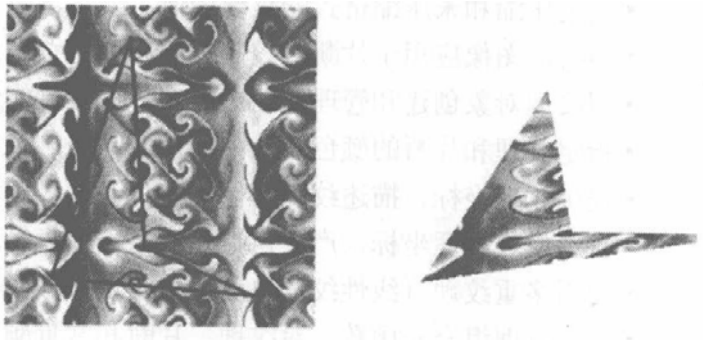


图9-1 纹理贴图过程

注意，这个纹理图像需要进行变形，以便与这个四边形的变形相匹配。在此例中，它在X方向拉伸并在Y方向收缩。除此之外，它还稍微进行了旋转，并略做了一些裁剪。根据纹理图像的大小、四边形的变形情况以及屏幕图像的大小，有些纹理单元可能会映射到多个片断上，有些片断则可能被多个纹理单元所覆盖。由于纹理是由离散的纹理单元构成的（在此例中；是由256×256个纹理单元构成），所以必须执行过滤操作，把纹理单元映射到片断上。例如，如果多个纹理单元对应于一个片断，这些纹理单元就在求得均值后再应用到这个片断上。如果纹理单元的边界位于片断的边界上，OpenGL就会对所有相关的纹理单元求加权平均值。由于存在这些计算，纹理贴图从计算的角度而言是比较昂贵的，这也是为什么有这么多的专用图像系统（包括硬件）对纹理贴图提供支持的原因。

应用程序可以创建纹理对象，每个纹理对象都表示一个单独的纹理（并可能与mipmap相关联）。有些OpenGL实现可能支持一种特别的纹理对象工作集（working set），位于工作集内部的纹理对象比位于工作集外部的纹理单元具有更高的性能。这些高性能的纹理对象称为常驻纹理对象，它们可能使用专用的硬件（或软件）加速工具。可以使用OpenGL创建和删除纹理对象，并确定由哪些纹理对象来组成工作集。

本章介绍OpenGL的纹理贴图功能，其内容分布于下面这几节中：

- 概述和示例：简单介绍执行纹理贴图所需要的步骤。本节还提供了一个相对较为简单的纹理贴图例子。
- 指定纹理：解释如何指定一维、二维和三维纹理。本节还讨论了如何使用纹理的边界、如何提供一系列大小不同的相关纹理、如何控制用于把纹理映射到屏幕坐标的过滤方法。
- 过滤：详细解释当纹理应用于多边形的像素时是如何放大或缩小的。本节还介绍了一种实现纹理缩小的特殊方法——mipmap。
- 纹理对象：描述如何把纹理图像放在纹理对象中，以便同时控制几个纹理。根据具体使用的OpenGL实现，有可能创建高性能纹理的工作集，这类纹理对象又称为常驻纹理对象。还可以

更改纹理对象的优先级，使它们增加或减少成为常驻纹理对象的机会。

- 纹理函数：讨论在表面上绘制纹理的方法。可以选择让纹理颜色替换表面的原来颜色，也可以选择由纹理颜色和表面原来的颜色组合形成最终颜色。
- 分配纹理坐标：描述如何计算纹理坐标，并为物体的顶点分配适当的纹理坐标。本节还解释了如何控制当坐标位于默认范围之外的行为。也就是如何对表面边缘的纹理进行重复或截取。
- 纹理坐标自动生成：解释如何让OpenGL自动生成纹理坐标，以实现像轮廓线或环境图这样的效果。
- 多重纹理：详细解释如何在一系列连续的纹理操作管线中应用纹理。
- 纹理组合器函数：解释如何控制对纹理的RGB和alpha值、常量颜色和片断所进行的数学运算（乘法、加法、减法、插值以及求点积）。纹理组合器函数提供了灵活、可编程的片断处理功能。
- 在纹理之后应用辅助颜色：解释在使用纹理之后如何对片断应用辅助颜色。
- 点块纹理：讨论如何对大点应用纹理，以提高它们的视觉质量。
- 纹理矩阵栈：解释如何操纵纹理矩阵栈以及如何使用q纹理坐标。
- 深度纹理：描述使用存储在深度缓冲区中的值作为纹理以确定场景阴影的过程。

OpenGL 1.1版本引入了下面几个纹理特性：

- 增加了一些纹理图像内部格式。
- 纹理代理，用于查询是否有足够的资源容纳一幅特定的纹理图像。
- 纹理子图像，用于替换原有纹理图像的部分或全部，而不是完全删除原来的纹理再创建一个新纹理来实现相同的效果。
- 根据帧缓冲区（以及系统内存）指定纹理数据。
- 纹理对象，包括常驻纹理以及纹理的优先级。

OpenGL 1.2版本又引入了下面这些纹理特性：

- 3D纹理图像。
- 一种新的纹理坐标包装模式GL_CLAMP_TO_EDGE，它使用的纹理单元取自纹理图像的边缘，而不是边框。
- 增强了对mipmap纹理的控制，用于表示不同的细节层（levels of detail, LOD）。
- 在纹理处理之后计算镜面亮点（根据光照）。

OpenGL 1.3版本又增加了下面这些纹理特性：

- 压缩纹理。
- 立方图纹理。
- 多重纹理，在一个图元上应用几个纹理。
- 纹理包装模式GL_CLAMP_TO_BORDER。
- 纹理环境模式：GL_ADD和GL_COMBINE（包括点积组合函数）。

OpenGL 1.4增加的纹理特性包括：

- 纹理包装模式GL_MIRRORED_REPEAT。
- 用GL_GENERATE_MIPMAP自动生成mipmap。
- 纹理参数GL_TEXTURE_LOD_BIAS，用于更改mipmap细节层的选择。
- 在纹理处理之后应用辅助颜色（由glSecondaryColor*()指定）。
- 处于纹理组合环境模式时，能够把不同的纹理单位作为纹理颜色的来源，用于纹理组合器函数。

- 使用深度（r坐标）作为一种内部纹理格式，以及对深度纹理单元进行比较的模式，以决定纹理的应用。

OpenGL 1.5增加的纹理特性包括：

- 新增了一些纹理比较模式，用于实现阴影贴图的纹理。

OpenGL 2.0增加的纹理特性包括：

- 在纹理图像中消除了宽度和高度必须为2的整数次方的限制。
- 点块纹理的迭代式纹理坐标。

OpenGL 2.1所增加的支持包括：

- 用sRGB格式指定纹理，它接受经过gamma校正的红、绿和蓝纹理成分。
- 在服务器端缓冲区对象中指定和提取像素矩形数据。关于使用像素缓冲区对象的细节，参见第8.5节。

OpenGL 3.0甚至包含了更多的纹理功能：

- 在未经规范化的浮点数、带符号整数和无符号整数格式中存储纹理单元（分别映射到范围[−1, 1]或[0, 1]）。
- 一维和二维的纹理数组，它们允许使用更高维度的纹理坐标来索引一维或二维的纹理图像的数组。
- 一种标准的纹理格式RGTC，用于单成分和双成分的纹理。

如果读者想使用上述的纹理功能，却又无法找到它，可以检查一下自己使用的OpenGL的版本号，看看它是不是支持这个功能（参见第14.2节）。有些OpenGL实现可能以扩展的形式支持上述的部分功能。

例如，在OpenGL 1.2版本中，OpenGL体系结构审查委员会（ARB，OpenGL的官方组织）批准了多重纹理作为一种可选的扩展。支持多重纹理的OpenGL 1.2实现需要提供以ARB为后缀的函数和常量名，例如glActiveTextureARB(GL_TEXTURE1_ARB)。在OpenGL 1.3中，多重纹理成为必须实现的功能，所以去掉了ARB后缀。

9.1 概述和示例

本节简单介绍了执行纹理贴图所需要的一些步骤，并且提供了一个相对较为简单的例子。当然，读者应该明白纹理贴图所涉及的内容是非常多的。

9.1.1 纹理贴图的步骤

为了使用纹理贴图，需要执行如下步骤：

1) 创建纹理对象，并为它指定一个纹理。
2) 确定纹理如何应用到每个像素上。
3) 启用纹理贴图功能。
4) 绘制场景，提供纹理坐标和几何图形坐标。

记住，纹理坐标必须在RGBA模式下才能使用。在颜色索引模式下使用纹理贴图的结果是难以预料的。

创建纹理对象，并为它指定一个纹理

纹理通常被认为是二维的，就像绝大多数图像一样。但是，它也可以是一维的，甚至可以是三维的。在每个纹理单元中，描述纹理的数据可以由1个、2个、3个或4个元素组成，用于表示RGBA四元组、调整常量或深度成分。

示例程序9-1是一个非常简单的程序，它创建了一个纹理对象，用于维护一个未压缩的二维纹理。这个程序并没有判断是否有足够的内存可以容纳这个纹理。由于只创建了1个纹理，所以它并没有设置纹理的优先级，也不必管理纹理对象工作集。除此之外，这个简单例子也没有包括诸如纹理边框、mipmap或立方图纹理等高级技巧。

确定纹理如何应用到每个像素上

我们可以在4种方法中进行选择，根据片断颜色和纹理图像数据来计算最终的RGBA值。其中一种方法就是简单地使用纹理颜色作为最终的颜色，这种方式称为替换（replace）模式，纹理被涂到片断的顶部，就像进行贴花一样（示例程序9-1就使用了替换模式）。另一种方法是用纹理来调整（modulate）（或缩放）片断的颜色，这个技巧在组合纹理和光照效果时非常有用。最后，我们还可以根据纹理值，用一种常量颜色与片断的颜色进行组合。

启用纹理贴图功能

我们需要在绘制场景之前启用纹理贴图功能。可以使用glEnable()和glDisable()函数来启用和禁用纹理贴图功能，使用GL_TEXTURE_1D、GL_TEXTURE_2D、GL_ TEXTURE_3D或GL_TEXTURE_CUBE_MAP常量为参数，分别表示一维、二维、三维或立方图纹理。如果同时启用了二维和三维纹理，则以后者为准。如果启用了GL_TEXTURE_CUBE_ MAP，则其他被启用的纹理均不再有效。为了使程序更为清晰，应该只启用一种类型的纹理。

绘制场景、提供纹理和几何坐标

我们需要确定纹理在进行“粘贴”之前应该如何根据它所应用的片断进行排列。也就是说，在场景中指定物体时，需要同时提供纹理坐标和几何坐标。例如，对于二维的纹理图，纹理坐标在X和Y方向上的范围都是从0.0～1.0，但是进行纹理贴图的物体的坐标却没有限制。例如，为了把砖块纹理应用到墙面上，假如墙面是正方形的，并且我们只想粘贴1份纹理，可以把纹理坐标（0，0）、（1，0）、（1，1）和（0，1）分别分配给这面墙的4个角。如果这面墙非常大，我们可能想在它上面粘贴多份纹理图像。如果想采用这样的做法，必须对纹理的排列进行精心设计，使砖块的左边缘与它左边砖块的右边缘准确对齐，下边缘与它下边砖块的上边缘准确对齐，构成精美的砖墙图案。

我们还必须确定位于范围[0.0，1.0]之外的纹理坐标应该如何处理，使这些纹理重复覆盖这个物体，或者把它们截取在一个边界值之内。

9.1.2 一个示例程序

使用简单程序演示纹理贴图时，存在的一个问题是，有趣的纹理一般都比较大。一般情况下，纹理是从图像文件读取的，这是因为如果直接在程序中指定纹理，可能需要几百行的额外代码。在示例程序9-1中，纹理（由交替的黑白方块组成，就像国际象棋盘一样）是由程序产生的。这个程序把这个纹理应用到两个正方形上，然后在透视投影模式下对它们进行渲染。其中一个正方形正面朝向观察者，另一个正方形与观察者呈45度倾斜，如图9-2所示。在物体坐标中，这两个正方形的大小相同。

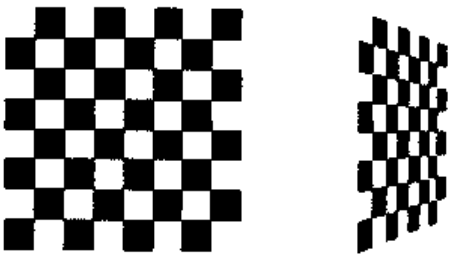


图9-2 纹理贴图的正方形

示例程序9-1 经过纹理贴图的棋盘：checker.c

```
/*Create checkerboard texture */
#define checkImageWidth 64
#define checkImageHeight 64
static GLubyte checkImage [checkImageHeight][checkImageWidth][4];

static GLuint texName;

void makeCheckImage(void)
{
   int i,j,c;
   for (i =0;i <checkImageHeight;i++){
      for (j =0;j <checkImageWidth;j++){
         c =(((i&0x8)==0)^((j&0x8))==0)*255;
         checkImage [i][j][0] =(GLubyte)c;
         checkImage [i][j][1] =(GLubyte)c;
         checkImage [i][j][2] =(GLubyte)c;
         checkImage [i][j][3] =(GLubyte)255;
      }
   }
}

void init(void)
{
   glClearColor(0.0,0.0,0.0,0.0);
   glShadeModel(GL_FLAT);
   glEnable(GL_DEPTH_TEST);
   makeCheckImage();
   glPixelStorei(GL_UNPACK_ALIGNMENT,1);

   glGenTextures(1,&texName);
   glBindTexture(GL_TEXTURE_2D,texName);

   glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
   glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
   glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,
                   GL_NEAREST);
   glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
                   GL_NEAREST);
   glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,checkImageWidth,
                   checkImageHeight,0,GL_RGBA,GL_UNSIGNED_BYTE,
                   checkImage);
   }

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);
   glEnable(GL_TEXTURE_2D);
   glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_REPLACE);
   glBindTexture(GL_TEXTURE_2D,texName);
   glBegin(GL_QUADS);
```

```
        glTexCoord2f(0.0,0.0);glVertex3f(-2.0,-1.0,0.0);
        glTexCoord2f(0.0,1.0);glVertex3f(-2.0,1.0,0.0);
        glTexCoord2f(1.0,1.0);glVertex3f(0.0,1.0,0.0);
        glTexCoord2f(1.0,0.0);glVertex3f(0.0,-1.0,0.0);

        glTexCoord2f(0.0,0.0);glVertex3f(1.0,-1.0,0.0);
        glTexCoord2f(0.0,1.0);glVertex3f(1.0,1.0,0.0);
        glTexCoord2f(1.0,1.0);glVertex3f(2.41421,1.0,-1.41421);
        glTexCoord2f(1.0,0.0);glVertex3f(2.41421,-1.0,-1.41421);
        glEnd();
        glFlush();
        glDisable(GL_TEXTURE_2D);
    }

    void reshape(int w,int h)
    {
        glViewport(0,0,(GLsizei)w,(GLsizei)h);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(60.0,(GLfloat)w/(GLfloat)h,1.0,30.0);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        glTranslatef(0.0,0.0,-3.6);
    }
    /*keyboard()and main()deleted to reduce printing */
```

棋盘纹理是在makeCheckImage()函数中生成的，所有的纹理贴图初始化是在init()函数中进行的。glGenTextures()和glBindTexture()函数分别用于命名纹理图像以及创建纹理对象（参见第9.4节）。这个单幅的全分辨率纹理图是由glTexImage2D()函数指定的，这个函数的参数指定了这幅纹理图像的大小、类型、位置以及其他属性（关于glTexImage2D()函数的更多信息，请参阅第9.2节）。

对glTexParameter*()函数的4次调用指定了这个纹理的包装形式，并且指定了当纹理图像中的纹理单元与屏幕上的像素并不完全匹配时，纹理的颜色应该如何进行过滤（参见第9.3节和第9.6.2节）。

在display()函数的内部，glEnable()函数启用了纹理功能。glTexEnv*()函数把绘图模式设置为GL_REPLACE。这样，经过纹理贴图的多边形在绘制时就使用来自纹理图像的颜色，而不再考虑它的原来颜色。

随后，这个程序绘制了两个多边形。注意，纹理坐标和顶点坐标是一起指定的。glTexCoord*()函数的行为与glNormal()函数相似。glTexCoord*()函数设置了当前的纹理坐标。所有后续的顶点函数将把这些纹理坐标与它们相关联，直到再次调用glTexCoord*()函数。

注意：当读者在自己的计算机上编译和运行这个程序时，那个倾斜正方形上的棋盘图像看上去可能是错误的。例如，它看上去就像是2个三角形，这幅棋盘图像以不同的投影方式投影到这2个三角形的表面。如果是这样，可以试着把GL_PERSPECTIVE_CORRECTION_HINT参数设置为GL_NICESET，然后再次运行这个程序。为此，还需要调用glHint()。

9.2. 指定纹理

glTexImage2D()函数用于定义二维纹理。这里简单介绍一下它的几个参数，接下来的几节将会详细介绍它们。用于定义一维和三维纹理的相关函数是glTexImage1D()和glTexImage3D()，我们分别在

第9.2.3节和第9.2.4节中介绍。

void **glTexImage2D**(GLenum *target*, GLint *level*, GLint *internalFormat*,
GLsizei *width*, GLsizei *height*, GLint *border*,
GLenum *format*, GLenum *type*, const GLvoid **texels*);

定义了一个二维纹理，或者一个一维纹理数组。target参数设置为下列常量之一：GL_TEXTURE_2D、GL_PROXY_T E X T URE_2D、GL_TEXTURE_CUBE_MAP_POSITIVE_X、GL_TEXTURE_CUBE_MAP_NEGATIVE_X、GL_TEXTURE_CUBE_MAP_POSITIVE_Y、GL_TEXTURE_CUBE_MAP_NEGATIVE_Y、GL_TEXTURE_CUBE_MAP_POSITIVE_Z、GL_TEXTURE_CUBE_MAP_NEGATIVE_Z或GL_PROXY_TEXTURE_CUBE_MAP（关于**glTexImage2D()**以及相关函数所使用的GL_*CUBE_MAP*常量的信息，请参阅第9.7.3节）用于定义二维纹理，以及GL_TEXTURE_1D_ARRAY和GL_PROXY_TEXTURE_1D_ARRAY用于定义一维纹理数组（只有在OpenGL 3.0及其以上的版本中才可用，参见本章9.2.5节），或者GL_TEXTURE_RECTANGLE和GL_PROXY_TEXTURE_RECTANGLE。

如果提供了多种分辨率的纹理图像，可以使用level参数。如果只提供了一种分辨率的纹理图像，level参数应该设置为0（关于使用多种分辨率的纹理图像的更多信息，请参阅第9.2.8节）。

接下来的一个参数internalFormat确定了哪些成分（RGBA、深度、亮度或强度）被选定为图像的纹理单元。有3组internalFormat。首先，以下用于internalFormat的符号常量明确了纹理单元值应该规范化（映射到范围[0,1]）并且存储到一个定点的表示中（如果标记名中包含了一个数值，它指定了位数）：

GL_ALPHA、GL_ALPHA4、GL_ALPHA8、GL_ALPHA12、GL_ALPHA16、GL_COMPRESSED_ALPHA、GL_COMPRESSED_LUMINANCE、GL_COMPRESSED_LUMINANCE_ALPHA、GL_COMPRESSED_INTENSITY、GL_COMPRESSED_RGB、GL_COMPRESSED_RGBA、GL_DEPTH_COMPONENT、GL_DEPTH_COMPONENT16、GL_DEPTH_COMPONENT24、GL_DEPTH_COMPONENT32、GL_DEPTH_STENCIL、GL_INTENSITY、GL_INTENSITY4、GL_INTENSITY8、GL_INTENSITY12、GL_INTENSITY16、GL_LUMINANCE、GL_LUMINANCE4、GL_LUMINANCE8、GL_LUMINANCE12、GL_LUMINANCE16、GL_LUMINANCE_ALPHA、GL_LUMINANCE4_ALPHA4、GL_LUMINANCE6_ALPHA2、GL_LUMINANCE8_ALPHA8、GL_LUMINANCE12_ALPHA4、GL_LUMINANCE12_ALPHA12、GL_LUMINANCE16_ALPHA16、GL_RED、GL_R8、GL_R16、GL_RG、GL_RG8、GL_RG16、GL_RGB、GL_R3_G3_B2、GL_RGB4、GL_RGB5、GL_RGB8、GL_RGB10、GL_RGB12、GL_RGB16、GL_RGBA、GL_RGBA2、GL_RGBA4、GL_RGB5_A1、GL_RGBA8、GL_RGB10_A2、GL_RGBA12、GL_RGBA16、GL_SRGB、GL_SRGB8、GL_SRGB_ALPHA、GL_SRGB8_ALPHA8、GL_SLUMINANCE_ALPHA、GL_SLUMINANCE8_ALPHA8、GL_SLUMINANCE、GL_SLUMINANCE8、GL_COMPRESSED_SRGB、GL_COMPRESSED_SRGB_ALPHA、GL_COMPRESSED_SLUMINANCE或者GL_COMPRESSED_SLUMINANCE_ALPHA。关于如何应用这些被选择的成分的信息，请参阅第9.5节。关于如何处理压缩纹理的信息，请参阅第9.2.6节。

internalFormat的下一组符号常量是在OpenGL 3.0中添加的，并且指定了浮点像素格式，它不是规范化的（并且存储在具有指定位数的浮点值中）：GL_R16F、GL_R32F、GL_RG16F、GL_RG32F、GL_RGB16F、GL_RGB32F、GL_RGBA16F、GL_RGBA32F、GL_R11F_G11F_B10F和

GL_RGB9_E5。另一组符号常量接受带符号的和无符号的（标记中增加一个“U”来表示）整数格式（分别存储在指定位数的整型中）的表示：GL_R8I、GL_R8UI、GL_R16I、GL_R16UI、GL_R32I、GL_R32UI、GL_RG8I、GL_RG8UI、GL_RG16I、GL_RG16UI、GL_RG32I、GL_RG32UI、GL_RGB8I、GL_RGB8UI、GL_RGB16I、GL_RGB16UI、GL_RGB32I、GL_RGB32UI、GL_RGBA8I、GL_RGBA8UI、GL_RGBA16I、GL_RGBA16UI、GL_RGBA32I和GL_RGBA32UI。此外，如果internalFormat是GL_COMPRESSED_RED和GL_COMPRESSED_RG之一，纹理可以以一种压缩的格式存储，如果是GL_COMPRESSED_RED_RGTC1、GL_COMPRESSED_SIGNED_RED_RGTC1、GL_COMPRESSED_RG_RGTC2和GL_COMPRESSED_SIGNED_RG_RGTC2之一，可以存储为特定的压缩纹理格式；对于确定大小的深度和模板格式，OpenGL 3.0增加了：GL_DEPTH_COMPONENT16、GL_DEPTH_COMPONENT24、GL_DEPTH_COMPONENT32F、GL_DEPTH24_STENCIL8和GL_DEPTH32F_STENCIL8，用于包装的模板-深度双通道纹理单元。

OpenGL 3.1增加了对带符号规范化值（映射到范围[−1，1]）的支持，用如下标记为internalFormat指定：GL_R8_SNORM、GL_R16_SNORM、GL_RG8_SNORM、GL_RG16_SNORM、GL_RGB8_SNORM、GL_RGB16_SNORM、GL_RGBA8_SNORM和GL_RGBA16_SNORM。

internalFormat参数可以请求一种特定的成分分辨率。例如，如果internalFormat设置为GL_R3_G3_B2，这个函数所请求的纹理单元就由3位红、3位绿和2位蓝色组成。根据定义，GL_INTENSITY、GL_LUMINANCE、GL_LUMINANCE_ALPHA、GL_DEPTH_COMPONENT、GL_RGB、GL_RGBA、GL_SRGB、GL_SRGB_ALPHA、GL_SLUMINANCE和GL_SLUMINANCE_ALPHA以及上述标记的压缩格式都不是很严格，因为它们并不要求一种特定的分辨率（为了与OpenGL 1.0版本兼容，表示internalFormat参数的值1、2、3和4分别对应于常量值GL_LUMINANCE、GL_LUMINANCE_ALPHA、GL_DEPTH_COMPONENT、GL_RGB和GL_RGBA）。

width和height参数表示纹理图像的宽度和高度。border参数表示边框的宽度，它可以是0（没有边框）或者是1（对于OpenGL 3.1实现，必须是0）。对于不支持OpenGL 2.0及其以上版本的OpenGL实现，width和height的值都必须是2m + 2b，其中m是一个非负的整数（width和height可以使用不同的m值），b是border的值。纹理图像的最大值取决于OpenGL实现，但它至少是64×64（如果有边框，至少是66×66）。另外，OpenGL 2.0及其以上版本的OpenGL实现，纹理可以是任意大小。format和type参数描述了纹理图像数据的格式和数据类型。它们和glDrawPixels()函数中的对应参数具有相同的含义（参见第8.3节）。事实上，纹理数据和glDrawPixels()函数所使用的数据具有相同的格式，因此glPixelStore*()和glPixelTransfer*()函数所进行的设置对它也起作用。示例程序9-1执行了下面这个调用：

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

这是因为这个示例程序中的数据在每行的末尾并没有进行填充。format参数可以是GL_COLOR_INDEX、GL_DEPTH_COMPONENT、GL_RGB、GL_RGBA、GL_RED、GL_GREEN、GL_BLUE、GL_ALPHA、GL_LUMINANCE或GL_LUMINANCE_ALPHA。也就是说，它可以是glDrawPixels()函数的format参数可以使用的值中除了GL_STENCIL_INDEX之外的所有值。

类似地，type参数可以是GL_BYTE、GL_UNSIGNED_BYTE、GL_SHORT、GL_UNSIGNED_SHORT、GL_INT、GL_UNSIGNED_INT、GL_FLOAT、GL_BITMAP或其中一种像素包装数据类型。

最后，texels参数包含了纹理图像数据。这个数据描述了纹理图像本身以及它的边框。

纹理图像的内部格式可能会影响纹理操作的性能。例如，有些OpenGL实现在执行纹理操作时使用GL_RGBA格式比使用GL_RGB格式更快，因为前者的颜色成分正好在处理器内存中满足4字节的对齐。由于情况各不相同，所以应该检查一下自己所使用的OpenGL实现的特定情况。

纹理图像的内部格式也可能影响纹理图像所消耗的内存。例如，对于内部格式为GL_RGBA8的纹理，每个纹理单元占据32位；而对于内部格式为GL_R3_G3_B2的纹理，每个纹理单元只使用8位。当然，在内存消耗和颜色分辨率之间也存在对应的权衡关系。

GL_DEPTH_COMPONENT纹理存储了深度值，与颜色相比，它更常用来渲染阴影（参阅9.13节）。类似地，GL_DEPTH_STENCIL纹理存储了相同纹理中的深度和模板值。

内部格式指定为GL_SRGB、GL_SRGB8、GL_SRGB_ALPHA、GL_SRGB8_ALPHA8、GL_SLUMINANCE_ALPHA、GL_SLUMINANCE8_ALPHA8、GL_SLUMINANCE、GL_SLUMINANCE8、GL_COMPRESSED_SRGB、GL_COMPRESSED_SRGB_ALPHA、GL_COMPRESSED_SLUMINANCE或GL_COMPRESSED_SLUMINANCE_ALPHA)的纹理，会拥有在sRGB颜色空间（正式的名称叫做国际电工委员会标准61966-2-1）中指定的红色、绿色和蓝色成分。sRGB颜色空间近似地与2.2 gamma校正线性RGB颜色空间相同。对于sRGB纹理，纹理中的alpha值不应该是gamma校正的。

对于后缀为“F”，“I”和“UI”的内部格式，纹理单元的格式分别存储在指定位数的浮点值、带符号整数值和无符号整数值中（例如，GL_R16F将存储一个单通道的纹理，每个纹理单元都是一个16位的浮点值）。所有这些格式都是OpenGL 3.0中增加的。对于这些格式，这些值并不是映射到范围[0,1]，而是允许它们的完整数字精度。涉及浮点值的另一种特殊的格式是包装的共享指数格式，它把红色、绿色和蓝色值指定为浮点值，所有这些值都具有同样的指数值。在线的附录J中说明了所有这些格式。该附录可以通过http://www.opengl-redbook.com/appendices/访问。

整数的内部纹理格式（带有“I”或“UI”后缀的那些）要求输入数据和指定的整数大小一致。

用包含GL_*_SNORM的内部格式指定的带符号的规范化值都转换到[−1, 1]范围中。

尽管在颜色索引模式下使用纹理贴图的后果是不可知的，但是我们仍然可以指定一个GL_COLOR_INDEX图像的纹理。在这种情况下，OpenGL会执行像素传输操作，在形成纹理图像之前通过颜色查找表把颜色索引值转换为RGBA值。

如果OpenGL实现支持图像处理子集，并且已经启用了一些图像处理子集功能，纹理图像就会受到这些功能的影响。例如，如果启用了二维卷积过滤器，那么在纹理图像上也会进行卷积过滤处理（卷积过滤可能会改变图像的宽度和高度）。

纹理图像的宽度和高度（不包括可选的边框宽度）上的纹理单元的数量必须是2的整数次方（OpenGL 2.0则无此要求）。如果源图像不满足这个要求，可以使用OpenGL工具库函数gluScaleImage()更改纹理图像的大小。

---

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

使用适当的像素存储模式对图像进行缩放，对存储在datain中的数据进行解包。format、typein和typeout参数可以使用**glDrawPixels()**函数所支持的任何格式或数据类型。图像是通过线性插值和箱过滤（根据从widthin和heightin到widthout和widthout所提示的大小），最终的图像写入到dataout，使用

像素GL_PACK*存储模式。**gluScaleImage()**的调用者必须分配足够的内存空间来容纳输出缓冲区。如果函数执行成功，它就返回0。如果执行失败，它就返回一个GLU错误代码。

注意：在GLU 1.3中，gluScaleImage()支持包装像素格式（以及相关的数据类型），但是OpenGL 3.0及其以后的版本很可能不再支持这些。

帧缓冲区本身也可以作为纹理数据使用。glCopyTexImage2D()函数从帧缓冲区读取一块矩形像素，并且把它作为一个新纹理的纹理单元。

void **glCopyTexImage2D**(GLenum *target*, GLint *level*,
GLint *internalFormat*,GLint *x*, GLint *y*,
GLsizei *width*, GLsizei *height*, GLint *border*);

创建一个二维纹理，使用帧缓冲区的数据来定义纹理单元。像素是从当前的GL_READ_BUFFER读取的。它的处理过程就像调用**glcopyPixels()**函数一样，但像素并不是放在帧缓冲区中，而是放入纹理内存中。**glPixelTransfer***()和其他像素传输操作的设置也会对它产生作用。

target参数必须是下列常量之一：GL_TEXTURE_2D、GL_TEXTURE_CUBE_MAP_POSITIVE_X、GL_TEXTURE_CUBE_MAP_NEGATIVE_X、GL_TEXTURE_CUBE_MAP_POSITIVE_Y、GL_TEXTURE_CUBE_MAP_NEGATIVE_Y、GL_TEXTURE_CUBE_MAP_POSITIVE_Z或GL_TEXTURE_CUBE_MAP_NEGATIVE_Z（关于这些*CUBE_MAP*常量的用法，请参阅第9.7.3节）。level、internalFormat和border参数与glTexImage2D()函数中的对应参数具有相同的含义。纹理数组是从屏幕对齐的像素矩形读取的，它的左下角是（x, y）表示的坐标。width和height参数指定了这个像素矩形的宽度和高度。width和height的值都必须是2m + 2b，其中m是一个非负的整数（width和height可以使用不同的m值），b是border的值。另外，在支持OpenGL 2.0或更高版本的实现中，纹理图像的宽度和高度可以是任何大小。

如果OpenGL实现是3.0及其以后的版本，可以使用帧缓冲对象直接渲染到纹理内存，从而高效地执行和glCopyPixels2D()相同的操作。第10.4节详细介绍了这个过程。

接下来的几节将进一步讨论纹理的细节，包括target、border和level参数的使用。target参数可以查询纹理的实际大小（通过glTexImage*D()创建一个纹理代理）以及纹理是否可以被当前的OpenGL实现的纹理资源使用。第9.2.2节描述了如何重新定义纹理的一部分。第9.2.3节和第9.2.4节分别介绍了一维和三维纹理。第9.2.6节讨论了纹理边框，它是由border参数控制的。level参数用于指定不同分辨率的纹理，它被并入到mipmap这个特殊技巧中。我们将在第9.2.8节中对它进行解释。mipmap要求我们理解如何对纹理进行过滤，纹理过滤也将在这一节中描述。

9.2.1 纹理代理

对于使用纹理贴图的OpenGL程序员而言，纹理的大小是非常重要的。纹理资源一般都是有限的，而纹理格式的限制也因不同的OpenGL实现而异。OpenGL提供了一种特殊的纹理目标，称为纹理代理（Texture Proxy），可用于判断当前的OpenGL实现在某种特定的纹理大小下是否支持某种特定的纹理格式。

可以使用glGetIntegerv(GL_MAX_TEXTURE_SIZE, ...)查询纹理图像的最大宽度和最大高度（不包括边框）的下界。一般情况下，它就是当前的OpenGL实现所支持的最大正方形纹理的大小。对于3D纹理，可以使用GL_MAX_3D_TEXTURE_SIZE参数查询3D纹理图像的最大允许大小（不包括边

框的宽度、高度或深度)。对于立方图纹理，可以使用类似的GL_MAX_CUBE_MAP_TEXTURE_SIZE参数。

但是，使用GL_MAX*TEXTURE_SIZE所进行的查询并没有考虑纹理的内部格式以及其他因素的效果。一幅以GL_RGBA16内部格式存储纹理单元的纹理图像可能用64位表示每个纹理单元，因此它的最大大小可能要比使用GL_LUMINANCE4内部格式的纹理图像要小16倍。那些要求边框或mipmap的纹理会进一步减少可用内存的数量。

纹理代理是一种特殊的占位符，它允许更精确地查询OpenGL是否可以容纳某种内部格式的纹理图像。

例如，为了判断是否有足够的资源容纳一个标准的2D纹理，可以在调用glTexImage2D()函数时把target参数设置为GL_PROXY_TEXTURE_2D，并设置相关的level、internalFormat、width、height、border、format和type参数值。对于纹理代理，应该传递NULL作为texels数组的指针。对于立方图纹理，在调用glTexImage2D()函数时应该把target参数设置为GL_PROXY_TEXTURE_CUBE_MAP。对于一维或三维纹理，可以使用对应的1D或3D函数以及对应的符号常量。

在创建了纹理代理之后，就可以使用glGetTexLevelParameter*()函数查询纹理状态变量的值。如果没有足够的资源容纳这个纹理代理，表示宽度、高度、边框宽度以及成分分辨率的纹理状态变量都设置为0。

---

void **glGetTexLevelParameter**{if}v(GLenum *target*, GLint *level*,
GLenum *pname*, TYPE *params);

在params参数中返回一个特定细节层的纹理参数值，这个细节层是由level参数指定的。target参数定义了目标纹理，它的值可以是GL_TEXTURE_1D、GL_TEXTURE_2D、GL_TEXTURE_3D、GL_TEXTURE_CUBE_MAP_POSITIVE_X、GL_TEXTURE_CUBE_MAP_NEGATIVE_X、GL_TEXTURE_CUBE_MAP_POSITIVE_Y、GL_TEXTURE_CUBE_MAP_NEGATIVE_Y、GL_TEXTURE_CUBE_MAP_POSITIVE_Z、GL_TEXTURE_CUBE_MAP_NEGATIVE_Z、GL_TEXTURE_ID_ARRAY、GL_TEXTURE_2D_ARRAY、GL_TEXTURE_RECTANGLE、GL_PROXY_TEXTURE_ID、GL_PROXY_TEXTURE_ID_ARRAY、GL_PROXY_TEXTURE_2D、GL_PROXY_TEXTURE_2D_ARRAY、GL_PROXY_TEXTURE_3D、GL_PROXY_TEXTURE_CUBE_MAP或GL-PROXY_TEXTURE_RECTANGLE。(GL_TEXTURE_ CUBE_MAP是无效的，因为它并没有指定立方图的一个特定表面)。pname参数可以接受的值为GL_TEXTURE_WIDTH、GL_TEXTURE_HEIGHT、GL_TEXTURE_DEPTH、GL_TEXTURE_BORDER、GL_TEXTURE_INTERNAL_FORMAT、GL_TEXTURE_RED_SIZE、GL_TEXTURE_GREEN_SIZE、GL_TEXTURE_BLUE_SIZE、GL_TEXTURE_ALPHA_SIZE、GL_TEXTURE_LUMINANCE_SIZE和GL_TEXTURE_ INTENSITY_SIZE。

示例程序9-2显示了如何使用纹理代理来判断是否有足够的资源创建一个64×64纹理单元、使用RGBA成分、分辨率为8位的纹理。如果可以，它就调用glGetTexLevelParameteriv()函数把这种内部格式（此例中为GL_RGBA8）存储到变量format中。

示例程序9-2 使用纹理代理查询纹理资源

```
GLint width;

glTexImage2D(GL_PROXY_TEXTURE_2D,0,GL_RGBA8,
```

```
                        64,64,0,GL_RGBA,GL_UNSIGNED_BYTE,NULL);
glGetTexLevelParameteriv(GL_PROXY_TEXTURE_2D,0,
                        GL_TEXTURE_WIDTH,&width);
```

注意：纹理代理存在一个主要的限制：纹理对象回答的问题是一个纹理是否能够被加载到纹理内存中。不管纹理单元当时是否正在使用，纹理代理都返回相同的答案。如果其他纹理正在使用纹理资源，那么纹理代理的查询所得到的也可能是肯定的结果，但实际上此时并没有足够的资源使这个被查询的纹理成为常驻纹理（也就是成为高性能纹理工作集的一部分）。纹理代理查询并不回答“是否有足够的能力处理所请求的纹理”这样的问题。关于管理常驻纹理的更多信息，请参阅第9.4节。

9.2.2 替换纹理图像的全部或一部分

和修改原有纹理相比，创建新纹理的开销更大一些。用新的信息修改一个现有纹理的全部或一部分常常比从头创建一个新纹理更合适。在有些应用场合，例如使用实时捕捉的视频图像作为纹理图像的应用程序，这个技巧非常实用。在这种应用程序中，可以创建一个纹理，然后反复调用glTexSubImage2D()函数，用新的视频纹理作为纹理数据。另外，在使用glTexSubImage2D()函数时，纹理图像的宽度和高度并不一定需要是2的整数次方。在处理视频图像时，少了这个限制是非常实用的，因为它的宽度和高度一般并不是2的整数次方。但是，必须把视频图像加载到一幅更大的初始图像中，后者的宽度和高度必须是2^n个纹理单元（OpenGL 2.0则无此限制），并调整子图像的纹理坐标。

---

void **glTexSubImage2D**(GLenum *target*, GLint *level*, GLint *xoffset*,
GLint *yoffset*, GLsizei *width*, GLsizei *height*,
GLenum *format*, GLenum *type*,
const GLvoid **texels*);

定义一个二维纹理图像，替换当前一幅现有的二维图像的全部或一块连续的区域（用2D的术语，就是一个矩形）。target参数必须设置为**glCopyTexImage2D()**函数的对应参数可以使用的值之一。

level、format和type参数类似于**glTexImage2D()**使用的对应参数。level是mipmap细节层序号。把子图像的宽度或高度指定为0并不会产生错误，但它显然不会产生任何效果。format和type描述了纹理图像数据的格式和数据类型。子图像也受到**glPixelStore*()**和**glPixelTransfer*()**以及其他像素传输操作所设置模式的影响。

texels包含了表示子图像的纹理数据。width和height参数是当前纹理图像中全部或部分被替换的子区域的宽度和高度。xoffset和yoffset指定了*x*和*y*方向上的纹理单元偏移量（(0, 0) 表示纹理的左下角），并且指定了子图像应该从现有纹理数组中的什么地方开始替换。这块区域并不包括位于原先定义的纹理数组的范围之外的任何纹理单元。

在示例程序9-3中，对有些来自示例程序9-1的代码做了修改，按下“s”键的效果是用一幅更小的棋盘图像替换原来的图像（最终的纹理如图9-3所示）。按下“r”键可以恢复原先的图像。示例程序9-3显示了两个函数makeCheckImages()和keyboard()，它们都进行了重大修改。关于glBind Texture()函数的更多信息，请参阅第9.4节。

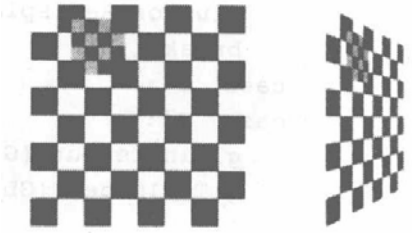


图9-3 添加了子图像的纹理

示例程序9-3 替换纹理子图像：texsub.c

```
/*Create checkerboard textures */
#define checkImageWidth 64
#define checkImageHeight 64
#define subImageWidth 16
#define subImageHeight 16
static GLubyte checkImage [checkImageHeight][checkImageWidth][4];
static GLubyte subImage [subImageHeight][subImageWidth][4];

void makeCheckImages(void)
{
   int i,j,c;

   for (i =0;i <checkImageHeight;i++){
      for (j =0;j <checkImageWidth;j++){
         c =(((i&0x8)==0)^ (((j&0x8)==0))*255;
         checkImage [i][j][0] =(GLubyte)c;
         checkImage [i][j][1] =(GLubyte)c;
         checkImage [i][j][2] =(GLubyte)c;
         checkImage [i][j][3] =(GLubyte)255;
      }
   }
   for (i =0;i <subImageHeight;i++){
      for (j =0;j <subImageWidth;j++){
         c =(((i&0x4)==0)^ (((j&0x4)==0))*255;
         subImage [i][j][0] =(GLubyte)c;
         subImage [i][j][1] =(GLubyte)0;
         subImage [i][j][2] =(GLubyte)0;
         subImage [i][j][3] =(GLubyte)255;
      }
   }
}

void keyboard(unsigned char key,int x,int y)
{
   switch (key){
      case 's':
      case 'S':
         glBindTexture(GL_TEXTURE_2D,texName);
         glTexSubImage2D(GL_TEXTURE_2D,0,12,44,
                         subImageWidth,subImageHeight,GL_RGBA,
                         GL_UNSIGNED_BYTE,subImage);
         glutPostRedisplay();
         break;
      case 'r':
      case 'R':
         glBindTexture(GL_TEXTURE_2D,texName);
         glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,
                      checkImageWidth,checkImageHeight,0,
                      GL_RGBA,GL_UNSIGNED_BYTE,checkImage);
         glutPostRedisplay();
```

```
            break;
         case 27:
            exit(0);
            break;
         default:
            break;
      }
   }
```

另外，帧缓冲区本身也可以作为纹理数据的来源，这次它所扮演的角色是纹理子图像。glCopyTexSubImage2D()函数从帧缓冲区读取一块像素矩形，并替换一个现有纹理数组的一部分。glCopyTexSubImage2D()相当于集成了glCopyTexImage2D()和glTexSubImage2D()的功能。

---

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

用来自帧缓冲区的图像数据替换当前现有的一幅二维纹理图像的全部或一块连续的子区域。像素是从当前的GL_READ_BUFFER读取的，其处理方式就像调用**glCopyPixels()**函数，但并不把像素读取到帧缓冲区，而是放在纹理内存中。**glPixelTransfer*()**和其他像素传输操作所设置的模式也会对它产生影响。

target参数必须设置为**glCopyTexImage2D()**函数的对应参数可以使用的值之一。level是mipmap细节层的序号。xoffset和yoffset指定了x和y方向上的纹理单元偏移量（(0, 0) 表示纹理的左下角），并且指定了子图像应该从纹理数组中的什么地方开始替换。子图像纹理数组取自一个屏幕对齐的像素矩形，它的左下角为（x, y）参数所指定的坐标。width和height参数指定了子图像矩形的宽度和高度。

对于OpenGL 3.1，如果target是GL_TEXTURE_RECTANGLE，而level不是0，会产生一个GL_INVALID_VALUE错误。

纹理矩形

OpenGL 3.1增加了通过其纹理单元位置而不是规范化的纹理坐标来定位的纹理。这些纹理通过GL_TEXTURE_RECTANGLE的target来指定，若想要在渲染的时候直接把纹理单元映射为像素，这么做就很有用。使用纹理矩形的时候有一些限制，正如其名字所示，不能进行基于mipmap的过滤（例如，纹理矩形不能有mipmap），也不能是压缩的。

9.2.3 一维纹理

有时候，使用一维纹理便已足够了。例如，程序所绘制的带纹理的镶条的所有变化可能都发生在同一个方向。一维纹理就像是一个高度为1的二维图像，并且它的顶部和底部没有边框。所有的二维纹理和子纹理定义函数都存在对应的一维版本。为了创建简单的一维纹理，可以使用glTexImage1D()函数。

---

void **glTexImage1D**(GLenum *target*, GLint *level*, GLint *internalFormat*,
GLsizei *width*, GLint *border*, GLenum *format*,
GLenum *type*, const GLvoid **texels*);

定义了一个一维纹理。它的所有参数都和**glTexImage2D()**函数的对应参数具有相同的含义，只是texels现在是一个一维数组。和以前一样，width的值必须是2^m（如果带边框，必须是2^{m+2}），其中m是一个非负整数。可以提供mipmap和纹理代理（把target设置为GL_PROXY_TEXTURE_1D），也可以使用相同的过滤操作。

示例程序9-8提供了一个使用一维纹理的例子。

如果OpenGL实现支持图像处理子集，并且启用了一维卷积过滤器（GL_ CONVOLUTION_1D），那么它就会在纹理图像上执行卷积操作（卷积可能会更改纹理图像的大小）。其他像素操作也可能会对纹理图像产生影响。

为了替换一维纹理图像的全部或一部分，可以使用glTexSubImage1D()函数。

---

void **glTexSubImage1D**(GLenum *target*, GLint *level*, GLint *xoffset*,
GLsizei *width*, GLenum *format*,
GLenum *type*, const GLvoid **texels*);

定义了一个一维纹理数组，它将替换当前现有的一幅一维纹理图像的全部或一块连续的子区域（用1D的术语表示，就是一行）。target参数必须设置为GL_TEXTURE_1D。

level、format和type参数和**glTexImage1D()**函数的对应参数相似。level是mipmap细节层序号。format和type参数描述了纹理图像数据的格式和数据类型。子图像还受到**glPixelStore*()**、**glPixelTransfer*()**以及其他像素传输操作所设置的模式的影响。

texels参数包含了子图像的纹理数据。width是当前纹理图像中被替换的全部或部分纹理单元的数量。xoffset指定了现有纹理数组中开始被子图像替换的偏移量。

为了使用帧缓冲区作为新的一维纹理的来源，或者用它来替换一个现有的一维纹理，可以使用glCopyTexImage1D()函数或glCopyTexSubImage1D()函数。

---

void **glCopyTexImage1D**(GLenum *target*, GLint *level*,
GLint *internalFormat*, GLivnt *x*, GLint *y*,
GLsizei *width*, GLint *border*);

创建一个一维纹理，并使用来自帧缓冲区的数据定义纹理单元。像素是从当前的GL_READ_BUFFER读取的，其处理过程就像调用**glCopyPixels()**函数一样，但像素并不是放在帧缓冲区中，而是放入纹理内存中。**glPixelStore*()**和**glPixelTransfer*()**所设置的模式也会对它产生影响。

target参数必须设置为GL_TEXTURE_1D。level、internalFormat和border参数的效果和它们在**glCopyTexImage2D()**中相同。纹理数组是从一行像素中读取的，它的左下角坐标由参数（x, y）指定。width参数指定了这一行的像素数量。width的值必须是2^m（如果带边框，必须是2^{m+2}），其中m是一个非负整数。对于不支持OpenGL 2.0的OpenGL实现，width的值是2^m（如果有边框，是2^{m+2}），其中m是一个非负的整数。

---

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

使用来自帧缓冲区的图像数据替换当前现有的一幅一维纹理图像的全部或一块连续的子区域。像素是从当前的GL_READ_BUFFER读取的，其处理方式就像调用**glCopyPixels()**函数，但并不把像素读取到帧缓冲区，而是放在纹理内存中。**glPixelTransfer*()**和其他像素传输操作所做的设置会对它产生影响。

target参数必须设置为GL_TEXTURE_1D。level是mipmap细节层的标号。xoffset指定了纹理单元偏移量以及在这个纹理数组的什么地方开始旋转纹理子图像。子图像纹理数组来源于一行像素，它的左下角坐标由参数（x, y）指定。width参数指定了这一行的像素数量。

9.2.4 三维纹理



高级话题

三维纹理最常见的应用是医学和地球科学领域的渲染。在医学应用程序中，三维纹理可以用于表示一系列的断层计算成像系统（CT）或核磁共振（MRI）图像。对于石油和天然气研究人员，三维纹理可以用来对岩石地层进行建模。三维纹理是一大类应用范畴的一部分，称为体渲染(volume rendering)。有些高级的体渲染应用程序需要处理体纹理单元（voxel），它表示基于体渲染实体的数据。

由于三维纹理可能非常大，它所消耗的纹理资源可能相当多。即使是一个相对较为粗糙的三维纹理所使用的纹理内存也可能比一个二维纹理使用的纹理内存多16或32倍（绝大多数二维纹理和子纹理定义函数都具有对应的三维版本）。

三维纹理图像可以看成是由一层层的二维子图像矩形构成的。在内存中，这些矩形按顺序排列在一起。为了创建简单的三维纹理，可以使用glTexImage3D()函数。

注意：图像处理子集并不存在三维卷积。但是，二维卷积过滤器也可以作用于三维纹理图像。

---

void **glTexImage3D**(GLenum *target*, GLint *level*, GLint *internalFormat*,
GLsizei *width*, GLsizei *height*, GLsizei *depth*,
GLint *border*, GLenum *format*, GLenum *type*,
const GLvoid **texels*);

定义一个三维纹理或者一个二维纹理的数组。所有的参数都和**glTexImage2D()**函数中的对应参数具有相同的含义，只不过现在texels是一个三维数组，并且增加了用于3D纹理的depth参数。对于GL_TEXTURE_2D_ARRAY，depth参数表示纹理数组的长度。如果OpenGL 实现不支持OpenGL 2.0，depth参数的值是2^m（如果带边框，必须是2^m+2），其中m是个非负整数。对于OpenGL 2.0实现，宽度和高度必须为2的整数次方的限制的需求都消除了。可以提供mipmap和纹理代理（把target设置为GL_PROXY_TEXTURE_3D），并且可以应用相同的过滤操作。

示例程序9-4是一个完整程序的一部分，它提供了一个使用三维纹理的例子。

示例程序9-4 三维纹理：texture3d.c

```
#define iWidth 16
#define iHeight 16
#define iDepth 16

static GLubyte image [iDepth][iHeight][iWidth][3];
static GLuint texName;

/*Create a 16x16x16x3 array with different color values in
*each array element [r,g,b].Values range from 0 to 255.
*/
void makeImage(void)
{
   int s,t,r;

   for (s =0 ;s <16 ;s++)
      for (t =0 ;t <16 ;t++)
        for (r =0 ;r <16 ;r++){
            image [r][t][s][0] =s *17;
```

```
            image [r][t][s][1] =t *17;
            image [r][t][s][2] =r *17;
        }
    }
}
/*Initialize state:the 3D texture object and its image
*/
void init(void)
{
    glClearColor(0.0,0.0,0.0,0.0);
    glShadeModel(GL_FLAT);
    glEnable(GL_DEPTH_TEST);

    makeImage();
    glPixelStorei(GL_UNPACK_ALIGNMENT,1);

    glGenTextures(1,&texName);
    glBindTexture(GL_TEXTURE_3D,texName);

    glTexParameteri(GL_TEXTURE_3D,GL_TEXTURE_WRAP_S,GL_CLAMP);
    glTexParameteri(GL_TEXTURE_3D,GL_TEXTURE_WRAP_T,GL_CLAMP);
    glTexParameteri(GL_TEXTURE_3D,GL_TEXTURE_WRAP_R,GL_CLAMP);

    glTexParameteri(GL_TEXTURE_3D,GL_TEXTURE_MAG_FILTER,
                    GL_NEAREST);
    glTexParameteri(GL_TEXTURE_3D,GL_TEXTURE_MIN_FILTER,
                    GL_NEAREST);
    glTexImage3D(GL_TEXTURE_3D,0,GL_RGB,iWidth,iHeight,
                 iDepth,0,GL_RGB,GL_UNSIGNED_BYTE,image);
}
```

为了替换三维纹理的全部或部分纹理单元，可以使用glTexSubImage3D()函数。

---

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

定义一个三维纹理数组，替换一个当前现有的三维纹理图像的全部或一块连续的子区域。target参数必须设置为GL_TEXTURE_3D。

level、format和type参数类似于**glTexImage3D()**函数使用的参数。level是mipmap细节层编号。format和type描述了纹理图像数据的格式和数据类型。子图像还受到**glPixelStore* ()**、**glPixelTransfer*()**以及其他像素传输操作所设置的模式的影响。

---

texels包含了这幅子图像的纹理数据。width、height和depth指定了这幅子图像的大小（以纹理单元为单位）。xoffset、yoffset和zoffset指定了纹理单元偏移量，表示从现有纹理数组的什么地方开始放置这幅子图像。

为了使用帧缓冲区作为子图像的来源，替换一个现有三维纹理的一部分，可以使用glCopySubImage3D()函数。

void **glCopyTexSubImage3D**(GLenum *target*, GLint *level*, GLint *xoffset*,
GLint *yoffset*, GLint *zoffset*, GLint *x*,
GLint *y*, GLsizei *width*, GLsizei *height*);

使用来自帧缓冲区的图像数据替换一幅当前现有的三维纹理图像的一块连续的子区域。像素是从当前的GL_READ_BUFFER读取的，其处理方式就像调用**glCopyPixels**()函数，但并不把像素读取到帧缓冲区，而是放在纹理内存中。**glPixelTransfer***()和其他像素传输操作所做的设置会对它产生影响。

target参数必须设置为GL_TEXTURE_3D。level是mipmap细节层序号。子图像纹理数组取自一个屏幕对齐的像素矩形，它的左下角坐标由参数（x, y）指定。width和height参数指定了这个子图像矩形的宽度和高度。xoffset、yoffset和zoffset指定了纹理单元偏移量，也就是从纹理数组中的什么地方开始放置这个子图像。由于这个子图像是一个二维的矩形，因此原来的三维纹理只有其中的一片（位于zoffset的那一片）被替换。

三维纹理的像素存储模式

像素存储值控制每层（也就是每个2D矩形）的行与行之间的空间。glPixelStore*()函数可以用于设置像素存储模式，使用诸如*ROW_LENGTH、*ALIGNMENT、*SKIP_PIXELS和*SKIP_ROWS这样的参数（其中*可以是GL_UNPACK或GL_PACK），这个函数用于控制对一个完整的像素或纹理单元数据矩形中的一个子矩形的引用。在第8.3.2节中，我们已经讨论了这些模式。

前面讨论的像素存储模式对于描述三维中的其中两维是很实用的，但是我们需要增加一种像素存储模式，支持对三维纹理图像数据的各个子体的引用。新参数*IMAGE_SKIP和*SKIP_IMAGES允许使用glTexImage3D()、glTexSubImage3D()和glGetTexImage()函数对需要访问的任何纹理子体进行定界和访问。

如果内存中的三维纹理大于被定义的子体，就需要使用*IMAGE_HEIGHT参数指定单个子图像的高度。另外，如果这个子体并不是从第一层开始的，就需要设置*SKIP_IMAGES参数。

*IMAGE_HEIGHT是一个像素存储参数，它定义了一幅三维纹理图像的其中一层的高度。如果*IMAGE_HEIGHT的值为0（负值是无效的），那么每个二维矩形的行数就是height的值，后者是传递给glTexImage3D()或glTexSubImage3D()的参数（这种做法是极为常见的，因为*IMAGE_HEIGHT在默认情况下是0）。否则，单层的高度就是这个*IMAGE_HEIGHT值。

图9-4显示了如何用*IMAGE_HEIGHT参数确定图像的高度（当height参数只决定子图像的高度时）。这张图显示了一个只有两层的三维纹理。

*SKIP_IMAGES参数定义了在访问这个子体的第一个数据之后需要跳过多少层。如果*SKIP_IMAGES的值是一个正整数（称之为n），那么纹理图像数据中的指针就向前推进多层（n×每层中纹理单元的数量）。最终的子体是从第n层开始，并且具有若干层的深度（深度的层次是由传递给glTexImage3D()或glTexSubImage3D()的depth参数决定的）。如果*SKIP_IMAGES的值为0（默认值），那么对纹理单元数据的访问就是从纹理单元数组描述的第一层开始的。

图9-5显示了*SKIP_IMAGES如何跳过几层，到达实际需要访问的子体。在这个例子中，*SKIP_IMAGES == 3，被访问的子体是从第3层开始的。

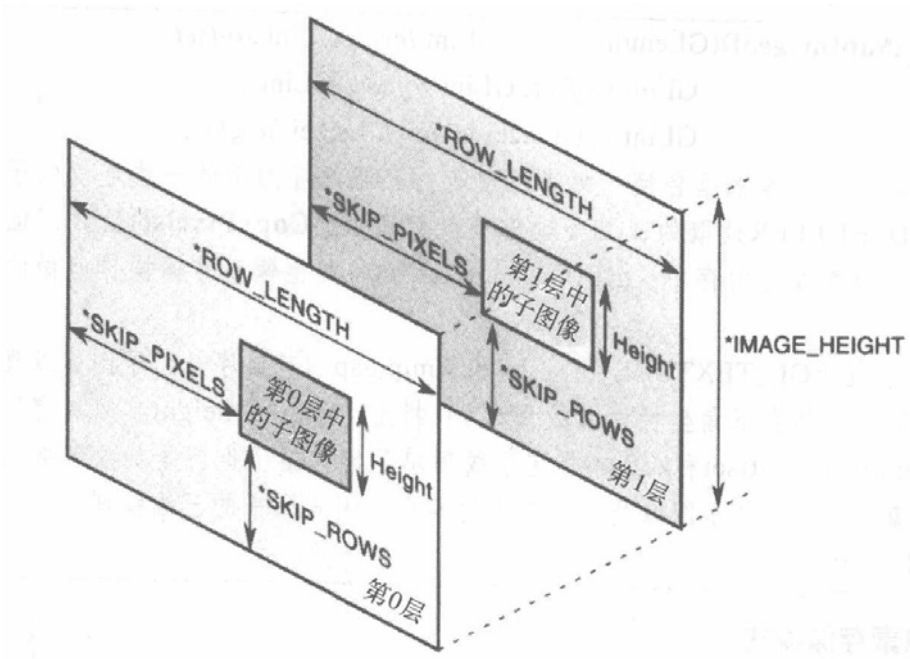


图9-4 *IMAGE_HEIGHT像素存储模式

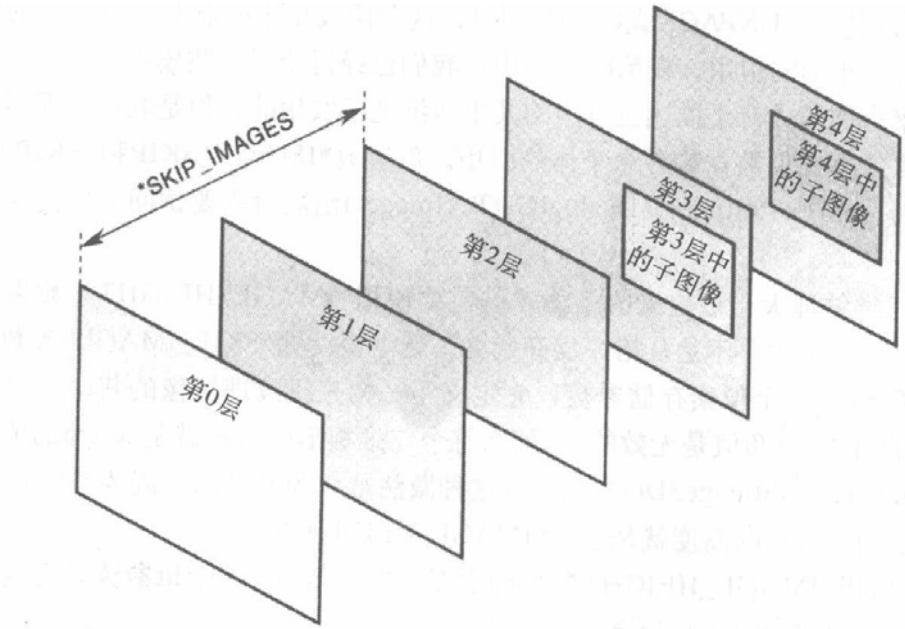


图9-5 *SKIP_IMAGES像素存储模式

9.2.5 纹理数组



高级话题

对于某些应用程序，可能需要在一次绘制调用的范围内同时访问多个一维或二维的纹理。例如，假设编写一款游戏，它显示几何形状基本相同的多个角色，但是，每个角色都有自己的装束。通过OpenGL 3.0的纹理数组的功能，可以使用一组调用，这类似于示例程序9-5所采用的方法。

如果纹理对象需要在OpenGL服务器中更新，为每次绘制调用而调用glBindTexture()可能会影响到应用程序的性能（因此，这可能是纹理存储资源的一个短处）。纹理数组允许把一维或二维纹理的

一个集合组织起来，所有的纹理都具有相同的大小，都位于更高维度的一个纹理之中（例如，二维纹理的数组是三维纹理的一部分）。如果想要使用一个三维纹理来存储二维纹理的一个集合，可能会遇到一些不便之处：纹理坐标的索引（在这个例子中是r）规范化到了范围[0, 1]。要访问一组7个纹理中的第3个纹理，可能需要遍历.35714（或者大概这么多）才能访问到“2”（纹理也是从0开始索引的，就像C语言一样）。纹理数组允许这种纹理选择。

此外，纹理数组允许在通过索引访问的纹理中进行合适的mipmap过滤。相比较而言，一个三维纹理可能在纹理“薄片”之间过滤，这种方式可能无法返回想要的结果。

9.2.6 压缩纹理图像

纹理图像在内部可以用一种压缩格式存储，以减少它所使用的内存数量。纹理图像可以在加载时进行压缩，也可以直接以一种压缩格式加载。

在加载时压缩纹理图像

为了让OpenGL在加载纹理图像时对它进行压缩，可以把internalFormat参数设置为其中一种GL_COMPRESSED_*枚举值。当纹理单元被任何活动的像素存储模式（参见第8.3.2节）或像素传输模式（参见第8.3.3节）处理之后，图像就会自动进行压缩。

当图像被加载之后，可以使用下面的方法判断它是否被压缩以及被压缩为什么格式：

```
GLboolean compressed;
GLenum    textureFormat;
GLsizei   imageSize;

glGetTexLevelParameteriv(GL_TEXTURE_2D, GL_TEXTURE_COMPRESSED,
   &compressed);
if (compressed == GL_TRUE) {
   glGetTexLevelParameteriv(GL_TEXTURE_2D,
      GL_TEXTURE_INTERNAL_FORMAT, &textureFormat);
   glGetTexLevelParameteriv(GL_TEXTURE_2D,
      GL_TEXTURE_COMPRESSED_IMAGE_SIZE, &imageSize);
}
```

加载经过压缩的纹理图像

OpenGL并没有指定纹理压缩应该使用的内部格式。每种OpenGL实现都可以指定一组OpenGL扩展，实现一种特定的纹理压缩格式。对于直接加载的压缩纹理，知道它们的存储格式并在OpenGL实现中验证这种纹理格式的有效性是非常重要的。

为了加载一个以压缩格式存储的纹理，可以使用glCompressedTexImage*()函数。

---

void **glCompressedTexImage1D**(GLenum *target*, GLint *level*,
GLenum *internalformat*, GLsizei *width*,
GLint *border*, GLsizei *imageSize*,
const GLvoid **texels*);

void **glCompressedTexImage2D**(GLenum *target*, GLint *level*,
GLenum *internalformat*, GLsizei *width*,
GLsizei *height*, GLint *border*,
GLsizei *imageSize*, const GLvoid **texels*);

void **glCompressedTexImage3D**(GLenum *target*, GLint *level*,
GLenum *internalformat*, GLsizei *width*,
GLsizei *height*, GLsizei *depth*,
GLint *border*, GLsizei *imageSize*,
const GLvoid **texels*);

根据一幅已压缩的纹理图像定义一个一维、二维或三维纹理。

如果所提供的是多个分辨率的纹理图像，需要设置level参数。如果只使用一种分辨率，level的值应该为0。关于使用多个分辨率的更多信息，请参阅第9.2.8节。

internalformat参数指定了经过压缩的纹理图像的格式。它必须是加载这幅纹理图像的OpenGL实现所支持的压缩格式之一，否则就会产生GL_INVALID_ENUM错误。为了判断哪些压缩纹理格式是受到支持的，可以参阅附录B。

width、height和depth参数分别表示一维、二维和三维纹理图像的维度。和未压缩的纹理一样，border表示纹理边框的宽度，它或者是0（即没有边框），或者是1。每个参数的值都必须是$2^m + 2b$，其中m是一个非负整数，b是border的值。对于OpenGL 2.0，宽度和高度必须是2的整数次方的限制都取消了。

对于OpenGL 3.1，如果**glCompressedTexImage2D()**的target是GL_TEXTURE_RECTANGLE或GL_PROXY_TEXTURE_RECTANGLE，会产生一个GL_INVALID_ENUM错误。

另外，就像未压缩的纹理一样，经过压缩的纹理也可以替换一个已经加载的纹理的全部或一部分。为此，可以使用glCompressedTexSubImage*D()函数。

void **glCompressedTexSubImage1D**(GLenum *target*, GLint *level*,
GLint *xoffset*, GLsizei *width*,
GLenum *format*, GLsizei *imageSize*,
const GLvoid **texels*);

void **glCompressedTexSubImage2D**(GLenum *target*, GLint *level*,
GLint *xoffset*, GLint *yoffet*,
GLsizei *width*, GLsizei *height*,
GLsizei *imageSize*,
const GLvoid **texels*);

void **glCompressedTexSubImage3D**(GLenum *target*, GLint *level*,
GLint *xoffset* GLint *yoffset*,
GLint *zoffset*, GLsizei *width*,
GLsizei *height*, GLsizei *depth*,
GLsizei *imageSize*,
const GLvoid **texels*);

根据一个已经压缩的纹理图像定义一个一维、二维或三维纹理。

xoffset、yoffset和zoffset参数指定了各个纹理分量的像素偏移量，以便在纹理数组中放置这幅新图像。

width、height和depth参数指定了用于更新纹理图像的这幅新的一维、二维或三维纹理图像的大小。

imageSize参数指定了存储于*texels*数组的字节数量。

9.2.7 使用纹理边框



高级话题

如果需要使用一幅很大的纹理图像，它的大小超过了OpenGL实现所允许的范围。只要稍微花点心思，就可以把几个不同的纹理有效地拼合成一个更大的纹理。例如，如果我们所需要的纹理的大小是允许映射到一个正方形上的最大图像的两倍，可以把这个正方形画成4个子正方形，然后在绘制每个子正方形之前加载一个不同的纹理。

由于每一时刻只有一个纹理图像能够处于有效状态，因此这个方法可能会在纹理的边缘上产生一些问题，尤其是在启用了某些形式的线性过滤的情况下。用于边缘像素的纹理值必须与边界之外的其他东西进行匀和，最理想的情况就是取它与相邻的纹理图像的纹理值的平均值。如果为每个纹理定义了一个边框，它的纹理单元值等于邻近纹理图像边缘的纹理单元值，那么在使用线性过滤时，这种做法就会产生正确的结果。

为了正确地执行这个任务，首先注意每幅纹理图像都有8个邻居。每条边和每个角都有一个对应的邻居。边框角上的纹理单元需要与和这个角接触的那些纹理图像的纹理单元相对应。如果使用的纹理是一整块纹理的边缘或角，就需要决定用哪个值合理地表示边框。最为简单合理的方法是使用glTexSubImage2D()函数复制纹理图像中邻近纹理单元的值。

如果纹理在应用时只是覆盖图元的一部分，也需要使用纹理的边框颜色（关于这方面的更多信息，请参阅第9.6.2节）。

注意：OpenGL 3.1及其以后的版本不支持纹理边框。当在那些实现中指定一个纹理的时候，边框的值必须是0。

9.2.8 mipmap：多重细节层



高级话题

和场景中的其他任何物体一样，可以在多个不同的位置观察纹理对象。在动态场景中，当一个纹理对象迅速地远离观察点而去时，纹理图像必须随被投影的图像一起缩小。为了实现这个效果，OpenGL必须对纹理图像进行过滤，适当地对它进行缩小，使它在映射到物体的表面时不会产生令人不快的人工视觉效果，如闪烁或抖动等。例如，当渲染一面砖墙时，如果它离观察者很近，可以使用较大的纹理图像（例如128×128纹理单元）。但是，当这面砖墙与观察者的距离迅速加大时，在它缩小为屏幕上的单个像素之前，在经过一些过渡点时，经过过滤的纹理图像可能会出现突然的变化。

为了避免这种人工痕迹，可以指定一系列预先过滤的分辨率递减的纹理图像，称为mipmap，如图9-6所示。mipmap这个术语是由Lance williams创造的，最早出现在他的论文“Pyramidal Parametrics”（SIGGRAPH 1983 Proceedings）中。mip表示拉丁语multum in parvo，意思是“一块小地方有很多东西”。mipmap使用了一些精巧的方法把图像数据挑选到内存中。

注意：如果读者想深入地了解mipmap，需要理解缩小过滤器（minification filter），它将在第9.3节中介绍。

当OpenGL使用mipmap时，它会根据被贴图的物体的大小（以像素为单位）自动确定应该使用哪个纹理。通过这种方法，纹理图像的细节层就能够适应绘制到屏幕上的图像。当物体的图像变小时，纹理图像也随之变小。mipmap要求一些额外的计算，并需要一些纹理存储区域。但是，如果不使用mipmap，当纹理映射到更小的物体上时，当物体移动时，就会产生闪烁或抖动现象。

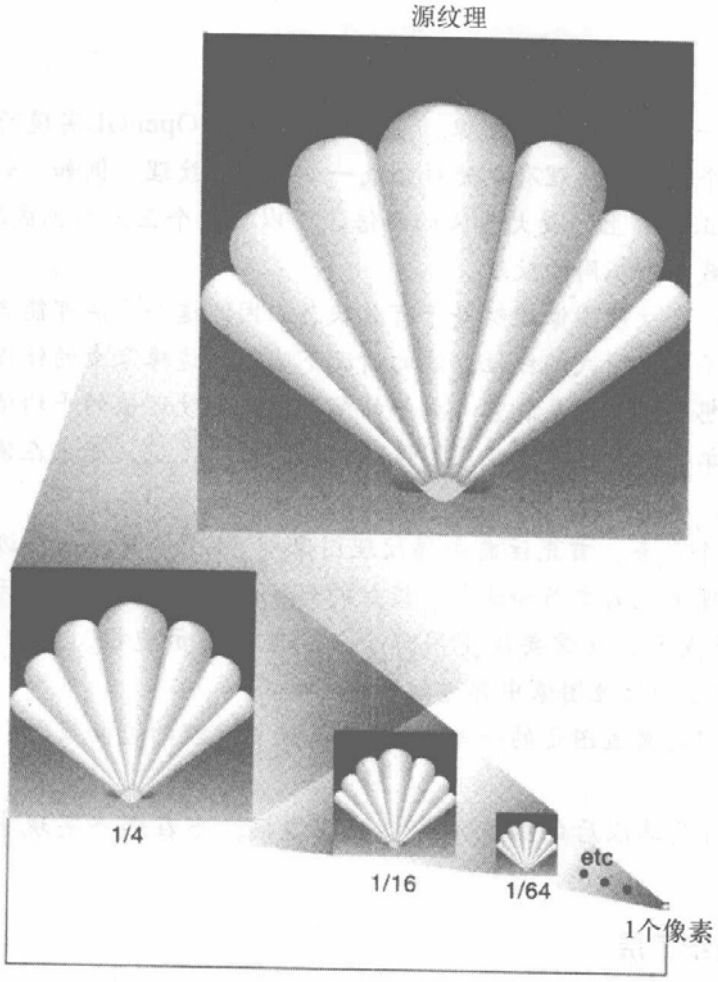


图9-6 mipmap

为了使用mipmap，必须提供全系列的大小为2的整数次方的纹理图像，其范围从最大值直到1×1纹理单元。例如，如果最高分辨率的纹理图像是64×16纹理单元，还必须提供大小分别为32×8、16×4、8×2、4×1、2×1和1×1的纹理图像。较小的纹理图像通常是进行了过滤的版本，是对最大的纹理图像进行适当匀缩之后的结果。较小纹理图像的每个纹理单元是更高一级分辨率的纹理图像的4个纹理单元的平均值。由于OpenGL并没有规定使用任何特定的方法来计算低分辨率的纹理图像，因此不同大小的纹理可以是完全不相关的。在实际使用中，不相关的纹理图像会使得mipmap层之间的过渡变得极为醒目，如彩图20所示。

为了指定这些纹理，可以依次调用glTexImage2D()函数生成每种分辨率的纹理图像，每次调用时使用不同的level、width、height和image参数。level参数以0为起点，这个参数用于标识整个系列的所有纹理图像。在前面那个例子中，大小为64×16的最高分辨率的纹理图像声明为level = 0，分辨率为32×8的纹理图像声明为level = 1，接下来以此类推。另外，为了使mipmap纹理生效，需要选择一种适当的过滤模式（参见第9.3节）。

注意：这里对OpenGL mipmap的描述避免了详细讨论纹理单元大小和多边形大小之间的缩放因子（称为λ）。这里还假设使用了与mipmap相关的默认参数值。关于λ和mipmap参数的解释，可以参阅本节稍后的“计算mipmap层”和“mipmap层的细节控制层”。

示例程序9-5显示了一系列的6幅纹理图像的用法，它们的大小从32×32逐渐缩小为1×1。这个程序绘制了一个矩形，它从前景位置延伸至极远处，最终在某个点消失，如彩图20所示。注意，纹理坐标的范围从0.0到8.0，因此铺满这个矩形需要64份纹理图像的拷贝，每个方向需要8份。为了更清楚地显示一幅纹理图像是如何与另一幅纹理图像相连的，我们在每幅纹理图像中使用了不同的颜色。

示例程序9-5 mipmap纹理：mipmap.c

```
GLubyte mipmapImage32 [32][32][4];
GLubyte mipmapImage16 [16][16][4];
GLubyte mipmapImage8 [8][8][4];
GLubyte mipmapImage4 [4][4][4];
GLubyte mipmapImage2 [2][2][4];
GLubyte mipmapImage1 [1][1][4];

static GLuint texName;

void makeImages(void)
{
   int i,j;

   for (i =0;i <32;i++){
      for (j =0;j <32;j++){
          mipmapImage32 [i][j][0] =255;
          mipmapImage32 [i][j][1] =255;
          mipmapImage32 [i][j][2] =0;
          mipmapImage32 [i][j][3] =255;
      }
   }
   for (i =0;i <16;i++){
      for (j =0;j <16;j++){
          mipmapImage16 [i][j][0] =255;
          mipmapImage16 [i][j][1] =0;
          mipmapImage16 [i][j][2] =255;
          mipmapImage16 [i][j][3] =255;
      }
   }
   for (i =0;i <8;i++){
      for (j =0;j <8;j++){
          mipmapImage8 [i][j][0]=255;
          mipmapImage8 [i][j][1]=0;
          mipmapImage8 [i][j][2]=0;
          mipmapImage8 [i][j][3]=255;
      }
   }
   for (i =0;i <4;i++){
      for (j =0;j <4;j++){
          mipmapImage4 [i][j][0]=0;
          mipmapImage4 [i][j][1]=255;
          mipmapImage4 [i][j][2]=0;
          mipmapImage4 [i][j][3]=255;
      }
   }
```

```
    for (i =0;i <2;i++){
       for (j =0;j <2;j++){
          mipmapImage2 [i][j][0] =0;
          mipmapImage2 [i][j][1] =0;
          mipmapImage2 [i][j][2] =255;
          mipmapImage2 [i][j][3] =255;
       }
    }
    mipmapImage1 [0][0][0] =255;
    mipmapImage1 [0][0][1] =255;
    mipmapImage1 [0][0][2] =255;
    mipmapImage1 [0][0][3] =255;
}
void init(void)
{
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_FLAT);

    glTranslatef(0.0,0.0,-3.6);
    makeImages();
    glPixelStorei(GL_UNPACK_ALIGNMENT,1);

    glGenTextures(1,&texName);
    glBindTexture(GL_TEXTURE_2D,texName);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,
                    GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
                    GL_NEAREST_MIPMAP_NEAREST);
    glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,32,32,0,
                 GL_RGBA,GL_UNSIGNED_BYTE,mipmapImage32);
    glTexImage2D(GL_TEXTURE_2D,1,GL_RGBA,16,16,0,
                 GL_RGBA,GL_UNSIGNED_BYTE,mipmapImage16);
    glTexImage2D(GL_TEXTURE_2D,2,GL_RGBA,8,8,0,
                 GL_RGBA,GL_UNSIGNED_BYTE,mipmapImage8);
    glTexImage2D(GL_TEXTURE_2D,3,GL_RGBA,4,4,0,
                 GL_RGBA,GL_UNSIGNED_BYTE,mipmapImage4);
    glTexImage2D(GL_TEXTURE_2D,4,GL_RGBA,2,2,0,
                 GL_RGBA,GL_UNSIGNED_BYTE,mipmapImage2);
    glTexImage2D(GL_TEXTURE_2D,5,GL_RGBA,1,1,0,
                 GL_RGBA,GL_UNSIGNED_BYTE,mipmapImage1);

    glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_REPLACE);
    glEnable(GL_TEXTURE_2D);
}
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);
    glBindTexture(GL_TEXTURE_2D,texName);
    glBegin(GL_QUADS);
    glTexCoord2f(0.0,0.0);glVertex3f(-2.0,-1.0,0.0);
```

```
    glTexCoord2f(0.0,8.0);glVertex3f(-2.0,1.0,0.0);
    glTexCoord2f(8.0,8.0);glVertex3f(2000.0,1.0,-6000.0);
    glTexCoord2f(8.0,0.0);glVertex3f(2000.0,-1.0,-6000.0);
    glEnd();
    glFlush();
}
```

在实际应用程序中构建Mipmap

示例程序9-5通过对每个mipmap层使用不同颜色的方法来说明mipmap的处理过程，当一幅纹理图像被另一幅纹理图像替换时，我们就可以清楚地觉察到这个过程。在现实应用中，需要精心定义mipmap纹理图像，使它们之间的过渡尽量显得平滑。因此，较低分辨率的纹理图像通常是原先未过滤的高分辨率纹理图像的过滤版本。

使用OpenGL功能来创建mipmap有几种方法。在OpenGL的最新版本中（例如，OpenGL 3.0及其以后的版本），我们使用glGenerateMipmap()，它将为绑定到一个特定的纹理目标（参阅第9.4节及其对glBindTexture()的介绍）的当前纹理图像构建mipmap栈。

---

int **glGenerateMipmap**(GLenum *target*);

为与target相关联的纹理图像生成一组完整的mipmap，其中，target必须是如下之一：GL_TEXTURE_1D、GL_TEXTURE_2D、GL_TEXTURE_3D、GL_TEXTURE_1D_ARRAY、GL_TEXTURE_2D_ARRAY或GL_TEXTURE_CUBE_MAP。

所构建的mipmap层由GL_TEXTURE_BASE_LEVEL和GL_TEXTURE_MAX_LEVEL控制（参阅本节稍后的“Mipmap层的细节控制”，了解这些值的详细说明）。如果这些值保留为默认值，所创建的整个mipmap栈其层级下降为一个单个纹理单元的纹理图像。每个连续层级创建过程中采用的过滤方法是和实现相关的。

如果target是GL_TEXTURE_CUBE_MAP，并且并非所有的立方图表面都是已初始化的和一致的，将会产生一个GL_INVALID_OPERATION错误。

---

glGenerateMipmap()的使用明确了你喜欢哪一个mipmap，并且将它们置于OpenGL实现的控制之下。如果你还没有用到OpenGL 3.0及其以后的实现，仍然可以让OpenGL产生mipmap。使用glTexParameter*()把GL_GENERATE_MIPMAP设置为GL_TRUE，然后对于具有BASE_LEVEL的一个mipmap的纹理单元（内部或边框）的任何改变，都将会自动引起从BASE_LEVEL+1到MAX_LEVEL的所有mipmap层级的所有纹理被重新计算和替换。而所有其他mipmap层级的纹理，包括BASE_LEVEL层级的，都将保持不变。

注意：在OpenGL 3.1及其以后的版本中，GL_GENERATE_MIPMAP的用法已经由更明确的glGenerateMipmap()函数所替代。在OpenGL 3.1中，试图在一个纹理对象上设置GL_GENERATE_MIPMAP将会产生一个GL_INVALID_OPERATION错误。

最后，如果在使用一个较早的OpenGL实现（1.4之前的任何版本），可能需要手动地构建mipmap栈，而得不到OpenGL的协助。然而，由于mipmap构建是如此重要的一个操作，OpenGL Utility Library包含了一些程序，可以帮助我们操作那些要用做mipmap的纹理的图像。

假如已经创建了第0层（最高分辨率）的mipmap，可以使用gluBuild1DMipmaps()、gluBuild2DMipmaps()或gluBuild3DMipmaps()函数创建和定义一系列大小递减的mipmap，直到1×1纹理单元（如果是1维纹理，为1纹理单元；如果是3维纹理，为1×1×1纹理单元）。如果源纹理的大

小并不是2的整数次方，可以使用gluBuild*DMipmaps()函数把纹理图像缩放为最邻近的2的整数次方。

同样，如果纹理太大，可以使用gluBuild*DMipmaps()函数把纹理图像缩小为适当的大小（通过GL_PROXY_TEXTURE机制来测量）。

int **gluBuild1DMipmaps**(GLenum *target*, GLint *internalFormat*,
GLint *width*, GLenum *format*, GLenum *type*,
const void **texels*);
int **gluBuild2DMipmaps**(GLenum *target*, GLint *internalFormat*,
GLint *width*, GLint *height*, GLenum *format*,
GLenum *type*, const void **texels*);
int **gluBuild3DMipmaps**(GLenum *target*, GLint *internalFormat*,
GLint *width*, GLint *height*, GLint *depth*,
GLenum *format*, GLenum *type*, const void **texels*);

创建一系列的mapmap，并调用**glTexImages*D()**加载这些纹理图像。target、internalFormat、width、height、depth、format、type和texels参数的含义和**glTexImage1D()**、**glTexImage2D()**、**glTexImage3D()**函数的对应参数完全相同。如果所有的mipmap均成功创建，这些函数就返回0。否则，它们返回一个GLU错误代码。

随着对细节层控制的增加（使用BASE_LEVEL、MAX_LEVEL、MIN_LOD和MAX_LOD），可能只需要创建gluBuild*DMipmaps()函数所创建的mipmap的一个子集。例如，需要使用的最小纹理图像可能是4×4纹理单元，而不是1×1纹理单元。为了计算和加载mipmap层的一个子集，可以调用gluBuild*DMipmapLevels()函数。

int **gluBuild1DMipmapLevels**(GLenum *target*, GLint *internalFormat*,
GLint *width*, GLenum *format*,
GLenum *type*, GLint *level*, GLint *base*,
GLint *max*, const void **texels*);
int **gluBuild2DMipmapLevels**(GLenum *target*, GLint *internalFormat*,
GLint *width*, GLint *height*, GLenum *format*,
GLenum *type*, GLint *level*, GLint *base*,
GLint *max*, const void **texels*);
int **gluBuild3DMipmapLevels**(GLenum *target*, GLint *internalFormat*,
GLint *width*, GLint *height*, GLint *depth*,
GLenum *format*, GLenum *type*,
GLint *level*, GLint *base*, GLint *max*,
const void **texels*);

创建一系列的mipmap，并调用**glTexImage*D()**函数加载这些纹理图像。level表示texels图像的mipmap层。basc和max确定要从texels中提取哪些mipmap层。target、internalFormat、width、height、depth、format、type和texels参数和**glTexImage1D()**、**glTexImage2D()**和**glTexImage3D()**函数的对应参数完全相同。如果所有的mipmap图像均成功创建，这些函数返回0，否则返回一个GLU错误代码。

计算Mipmap层

哪个mipmap层将作为一个特定多边形的纹理取决于纹理图像的大小和被贴图多边形的大小（以像素为单位）之间的缩放因子。我们把这个缩放因子称为ρ，另外再定义一个λ值，其中$\lambda = \log_2\rho + \text{lod}_{\text{bias}}$。（由于纹理图像可以是多维的，因此必须确认$\rho$是所有各维中的最大缩放因子，这一点非常重要）。

lod_{bias}表示偏移细节层，它是glTexEnv*()函数设置的一个常量值，用于对λ值进行调整（关于如何使用glTexEnv*()函数设置偏移细节层的细节，请参阅第9.5节）。在默认情况下，$\text{lod}_{\text{bias}} = 0.0$（即没有效果）。一开始最好使用这个默认值，然后根据需要进行微量的调整。

如果$\lambda \leqslant 0.0$，纹理就小于多边形，因此需要使用放大过滤器。如果$\lambda > 0.0$，就需要使用缩小过滤器。如果所选择的缩小过滤器使用了mipmap，那么λ就表示mipmap层（通常，放大和缩小过滤器的切换点是$\lambda = 0.0$，但这并不确定，mipmap过滤器的选择可能会影响切换点）。

例如，如果纹理图像的大小是64×64纹理单元，多边形的大小为32×32像素，那么$\rho = 2.0$（而不是4.0），因此$\lambda = 1.0$。如果纹理图像的大小是64×32纹理单元，多边形的大小是8×16像素，那么$\rho = 8.0$（x的缩放率为8.0，y的缩放率为2.0，使用最大值），因此$\lambda = 3.0$。

Mipmap层的细节控制

在默认情况下，必须为各个维的每种分辨率都提供一个mipmap（最小分辨率为1个纹理单元）。在有些实际使用中，我们可能想避免使用非常小的mipmap来表示数据。例如，我们可能想使用一种称为嵌拼（mosaicing）的技巧，就是用几幅更小的图像组合成一幅纹理图像。图9-7显示了一个嵌拼的例子，在单个纹理上有许多字符，它显然比为每个字符都创建一个纹理要高效得多。在使用这个纹理时，为了只把一个字符贴到物体上，可以灵活地使用纹理坐标提取需要的字符。

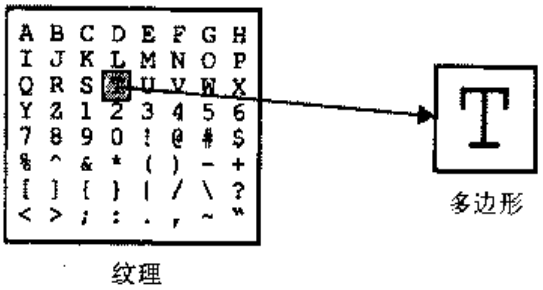


图9-7 使用嵌拼纹理

如果必须提供非常小的纹理，组成低分辨率的嵌拼mipmap的不同字符可能会粘在一起。因此，我们可能想限制最低分辨率。一般情况下，我们希望能够根据需要增加或删除mipmap层。

mipmap存在的另一个问题是“跳跃（poping）”，也就是当被贴图的多边形变大或变小时所引起的不同分辨率的mipmap层的突然切换。

注意：*许多mipmap功能是在OpenGL的后期版本中引进的。读者应该检查一下自己使用的OpenGL实现，看看它是不是支持某个特定的功能。有些版本可能会以扩展的形式提供某些功能。*

为了控制mipmap层，可以向glTexParameter*()传递GL_TEXTURE_BASE_LEVEL、GL_TEXTURE_MAX_LEVEL、GL_TEXTURE_MIN_LOD和GL_TEXTURE_MAX_LOD常量。前两个常量（为了简单起见，在本节的剩余内容中，我们将用BASE_LEVEL和MAX_LEVEL来表示它们）控制哪些mipmap层将被使用，并因此确定需要指定哪些mipmap层。另外两个常量（简写为MIN_LOD和MAX_LOD）用于控制前面提到的缩放因子λ的活动范围。

这些纹理参数可以解决前面描述的一些问题。有效地使用BASE_LEVEL和MAX_LEVEL可以减少应用程序需要指定的mipmap层的数量，并因此提高纹理资源的使用效率。选择性地使用MAX_LOD可以保持嵌拼纹理的有效性，MIN_LOD可以减少使用高分辨率纹理时产生的跳跃效果。

BASE_LEVEL和MAX_LEVEL用于设置mipmap层的边框。BASE_LEVEL是所使用的最高分辨率的mipmap层（最大的纹理）。BASE_LEVEL的默认值是0 。但是，可以在以后更改BASE_LEVEL的值，

因此可以随时添加更高分辨率的纹理。类似地，MAX_LEVEL用于限制可以使用的最低分辨率的mipmap层。MAX_LEVEL的默认值是1000，这基本上意味着最小分辨率纹理的大小总是1个纹理单元。

为了设置mipmap基层和mipmap最大层，可以调用glTexParameter*()函数，根据使用的纹理，把函数的第一个参数设置为GL_TEXTURE_1D、GL_TEXTURE_2D、GL_TEXTURE_3D或GL_TEXTURE_CUBE_MAP。第二个参数所取的值是表9-1所描述的参数之一。第三个参数提供了具体的参数值。

表9-1 mipmap层参数控制

参 数	描 述	值
GL_TEXTURE_BASE_LEVEL	可以使用的最高分辨率的纹理层（标号最小的mipmap层）	任何非负整数
GL_TEXTURE_MAX_LEVEL	可以使用的最小分辨率的纹理层（标号最大的mipmap层）	任何非负整数

示例程序9-6的代码把mipmap基层和mipmap最大层分别设置为2和5。由于纹理基层图像的分辨率为64×32，因此第3、4、5层的mipmap的分辨率必须相应地递减。

示例程序9-6 设置mipmap基层和mipmap最大层

```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_BASE_LEVEL,2);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAX_LEVEL,5);
glTexImage2D(GL_TEXTURE_2D,2,GL_RGBA,64,32,0,GL_RGBA,
             GL_UNSIGNED_BYTE,image1);
glTexImage2D(GL_TEXTURE_2D,3,GL_RGBA,32,16,0,GL_RGBA,
             GL_UNSIGNED_BYTE,image2);
glTexImage2D(GL_TEXTURE_2D,4,GL_RGBA,16,8,0,GL_RGBA,
             GL_UNSIGNED_BYTE,image3);
glTexImage2D(GL_TEXTURE_2D,5,GL_RGBA,8,4,0,GL_RGBA,
             GL_UNSIGNED_BYTE,image4);
```

以后，我们可能想添加其他具有更高或更低分辨率的mipmap层。例如，可以添加一个128×64的纹理，并把它设置为mipmap第1层，但是必须记得重新设置BASE_LEVEL。

注意：为了使mipmap生效，BASE_LEVEL和“最大可能层”之间的所有mipmap层都必须加载到应用程序中。最大可能层就是MAX_LEVEL的值和大小为1纹理单元（1、1×1或1×1×1）的mipmap层编号中较小的那个。如果并没有加载所有必要的mipmap层，那么纹理功能可能会神秘地消失。如果使用了mipmap功能但是纹理并没有出现，可以检查一下是否加载了所有必需的mipmap层。

和设置BASE_LEVEL和MAX_LEVEL参数一样，glTexParameter*()还可以设置MIN_LOD和MAX_LOD参数。表9-2对它们进行了描述。

表9-2 mipmap细节层的参数控制

参 数	描 述	值
GL_TEXTURE_MIN_LOD	λ（纹理图像和多边形之间的缩放值）的最小值	任何值
GL_TEXTURE_MAX_LOD	λ的最大值	任何值

下面的代码使用glTexParameter*()参数指定细节层参数：

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_LOD, 2.5);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAX_LOD, 4.5);
```

MIN_LOD和MAX_LOD提供了λ（纹理图像和多边形之间的缩放因子）的最小值和最大值，用于mipmap层的缩放。后者间接指定了需要使用的mipmap层。

如果有一个64×64像素的多边形，并且MIN_LOD为默认值0.0，那么第0层64×64纹理单元的纹理图像可能会被选中（前提是BASE_LEVEL = 0；BASE_LEVEL必须小于等于MAX_LOD）。但是，如果MIN_LOD设置为2.0，那么可以被选择的最大纹理图像就是16×16纹理单元，它对应于λ= 2.0。

MAX_LOD只有当它小于λ的最大值（或者是MAX_LEVEL，或者是大小为1纹理单元的mipmap层的编号）时才有效。对于一个64×64纹理单元的纹理图像，λ= 6.0对应于1×1纹理单元的mipmap层的编号。如果MAX_LOD是4.0，就无法选择小于4×4纹理单元的mipmap层。

读者可以发现，稍微大于BASE_LEVEL的MIN_LOD或者稍微小于MAX_LEVEL的MAX_LOD能够最为有效地减少与mipmap层过渡相关的视觉效果（例如跳跃）。

9.3 过滤

纹理图像是正方形或长方形的，但是当它们映射到多边形的表面并转换为屏幕坐标之后，纹理图像中的单个纹理单元很少与最终屏幕图像中的单个像素形成一一对应关系。根据物体使用的变换以及它所应用的纹理图像，屏幕上的一个像素可以对应很广的范围，可能是一个纹理单元的一小部分（放大），也可能是多个纹理单元（缩小），如图9-8所示。无论是哪种情况，我们很难知道具体使用的是哪些纹理值以及它们是如何匀和或插值的。为此，OpenGL允许指定几种过滤选项，确定这些计算的细节。这些选项提供了速度和图像质量之间的权衡。另外，可以分别为放大和缩小指定不同的过滤方法。

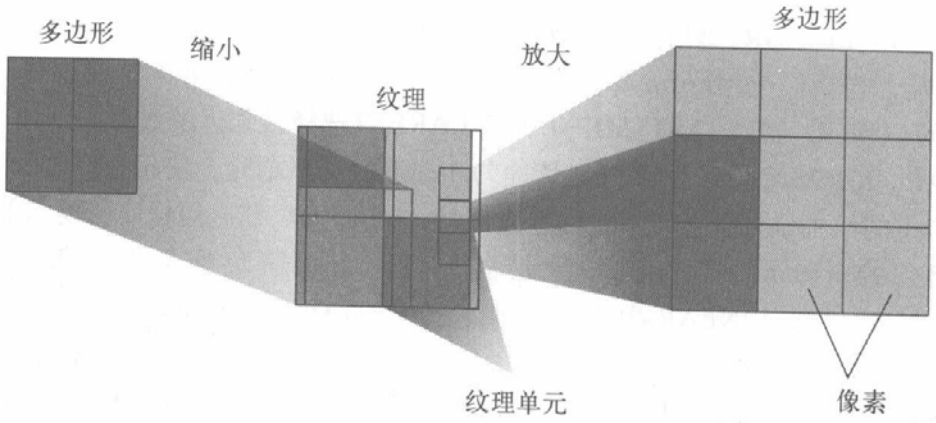


图9-8 纹理的放大和缩小

有些情况下，很难确定需要进行缩小还是放大。如果纹理图像需要在x轴和y轴上都进行拉伸（或收缩），那么它就需要放大（或缩小）。如果纹理图像需要在一个方向进行拉伸而在另一个方向进行收缩，OpenGL就需要在放大和缩小之间做出选择，尽可能获得最好的结果。避免这种问题的最好办法就是避免使用这类将会产生扭曲的纹理坐标（参见第9.6.1节）。

下面的代码显示了如何使用glTexParameter*()函数指定放大和缩小过滤方法：

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                GL_NEAREST);
```

glTexParameter*()函数的第一个参数可以是GL_TEXTURE_1D、GL_TEXTURE_2D、

GL_TEXTURE_3D或GL_TEXTURE_CUBE_MAP。由于我们当前所讨论的主题是过滤，因此第二个参数使用GL_TEXTURE_MAG_FILTER或GL_TEXTURE_MIN_FILTER，表示指定的过滤方法用于放大还是缩小。第三个参数指定了具体的过滤方法。表9-3列出了这两个参数可以取的值。

表9-3 用于放大和缩小的过滤方法

参 数	值
GL_TEXTURE_MAG_FILTER	GL_NEAREST或GL_LINEAR
GL_TEXTURE_MIN_FILTER	GL_NEAREST、GL_LINEAR、GL_NEAREST_MIPMAP_NEAREST、GL_NEAREST_MIPMAP_ LINEAR、GL_LINEAR _MIPMAP_NEAREST或GL_LINEAR _MIPMAP_LINEAR

如果选择了GL_NEAREST，那么最靠近像素中心的那个纹理单元将用于放大和缩小，这可能导致锯齿状的人工痕迹（有时候还比较严重）。如果选择GL_LINEAR，那么OpenGL就会对靠近像素中心的一块2×2纹理单元取加权平均值，用于放大和缩小（如果是三维纹理，就是对一块2×2×2纹理单元取加权平均值；如果是一维纹理，它就对2个纹理单元取平均值）。当纹理坐标靠近纹理图像的边缘时，最邻近的2×2纹理单元可能包含了纹理图像之外的内容。在这种情况下，OpenGL使用的纹理单元值取决于当前生效的环绕模式以及纹理是否具有边框（参见第9.6.2节）。GL_NEAREST所需要的计算量要少于GL_LINEAR，因此它的执行速度更快一些，但是GL_LINEAR能够提供更为平滑的结果。

在放大时，即使已经提供了mipmap层，OpenGL也只使用基层的纹理图像。在缩小时，可以选择一种过滤方法，使用最合适的一个或两个mipmap层（如下一段所述）。如果在缩小时指定了GL_NEAREST或GL_LINEAR，只有基层的纹理图像被使用。

如表9-3所示，当我们使用mipmap实现缩小时，可以使用4种额外的过滤操作。在一个单独的mipmap内部，可以用GL_NEAREST_MIPMAP_NEAREST选择最邻近的纹理单元值，或者可以通过指定GL_LINEAR _MIPMAP_NEAREST线性地勾和邻近的纹理单元。使用最邻近纹理单元的速度更快，但是它所产生的结果质量较差。系统所选择的特定mipmap层是应用程序所要求的缩小量的函数，并且在选择相邻的mipmap时存在一种权衡关系。为了避免突然的过渡效果，可以使用GL_NEAREST_MIPMAP_ LINEAR或GL_LINEAR_ MIPMAP_LINEAR取两个相邻的mipmap层的纹理单元勾和之后的最佳值。GL_NEAREST_MIPMAP_LINEAR从两个纹理中选择最邻近的纹理单元，并取它们之间的线性勾和值。GL_ LINEAR_MIPMAP_ LINEAR使用线性勾和计算出两个纹理各自的值，然后对两个计算结果再次进行线性勾和。正如读者所预料的那样，一般而言，GL_LINEAR_MIPMAP_ LINEAR能够产生质量最好的结果，但是它需要更多的计算，因此也是最慢的。

警告：如果请求了一个mipmap纹理过滤器，但是并没有提供完整一致的mipmap集(GL_TEXTURE_BASE_LEVEL和GL_TEXTURE_MAX_LEVEL之间所有具有正确大小的纹理图像)，OpenGL将会在不产生任何错误的情况下隐式地禁用纹理功能。如果使用了mipmap，却没有看到任何纹理，可以对所有层次的mipmap纹理进行检查。

有些纹理过滤器还具有一些更为人所熟知的名称。GL_NEAREST常常称为点采样（point sampling）。GL_LINEAR常常称为双线性采样，这是因为在二维纹理中，它是在一个2×2纹理单元数组中进行采样的。GL_LINEAR_MIPMAP_LINEAR有时候称作三线性采样（trilinear sampling），因为它是对两个双线性采样的mipmap进行线性勾和的。

注意：放大和缩小的切换点通常是在λ= 0.0，但是它受到所选择的缩小过滤器类型的影响。

如果当前的放大过滤器是GL_LINEAR并且缩小过滤器是GL_NEAREST_MIPMAP_NEAREST或GL_NEAREST_MIPMAP_LINEAR，那么它们之间的切换点就出现在λ= 0.5。这就防止了缩小的纹理看上去比放大之后的图像更为锐利。

Nate Robin的纹理教程

如果已经下载了Nate Robin的教学程序包，现在就可以运行“texture教程”（关于如何下载这些程序的信息，请参阅前言部分的“Nate Robin的OpenGL教程”）。在这个教程中，读者可以试验纹理贴图的过滤方法，在GL_NEAREST和GL_LINEAR之间进行切换。

9.4 纹理对象

纹理对象用于存储纹理数据，以便随时使用它们。我们可以控制多个纹理，并可以返回到以前加载到纹理资源的纹理。使用纹理对象通常是应用纹理的最快方法，它能够大幅度地提高应用程序的性能，因为绑定（复用）一个原有的纹理对象要比使用glTexImage*D()函数重新加载一幅纹理图像快得多。

另外，有些OpenGL实现支持一种容量有限的高性能纹理工作集。可以使用纹理对象，把最常用的纹理加载到这种容量有限的区域中。

为了使用纹理对象控制纹理数据，需要执行下面的步骤：

1) 生成纹理名称。

2) 初次把纹理对象绑定（创建）到纹理数据上，包括图像数组和纹理属性。

3) 如果OpenGL实现支持高性能纹理工作集，可以检查一下是否有足够的空间容纳所有的纹理对象。如果空间不足，可以为各个纹理对象建立优先级，把最常使用的纹理对象保存在这个工作集中。

4) 绑定和重新绑定纹理对象，使它们的数据当前可以用于渲染纹理模型。

9.4.1 命名纹理对象

任何非零的无符号整数都可以用来表示纹理对象的名称。为了避免意外的重名，应该坚持使用glGenTextures()函数来提供未使用的纹理对象名称。

---

void **glGenTextures**(GLsizei *n*, GLuint **textureNames*);

在数组*textureNames*中返回n个当前未使用的值，表示纹理对象的名称。*textureNames*中返回的值并不一定是连续的整数。

*textureNames*中的名称被标记为已使用，但是它们只有在第一次绑定时才获得纹理状态和维属性（1D、2D或3D）。

零作为一个保留的纹理对象名，它不会被**glGenTextures()**函数当作纹理对象名称而返回。

---

glIsTexture()函数用于判断一个纹理名称是否处于实际使用中。如果一个纹理名称是由glGenTextures()函数返回的，但是它还没有被绑定（绑定是指至少以这个名称为参数调用了glBindTexture()函数1次），那么glIsTexture()函数就返回GL_FALSE。

GLboolean **glIsTexture**(GLuint *textureName*);

如果*textureName*是一个已绑定的纹理对象名称，并且还没有删除，那么这个函数就返回GL_TRUE。如果*textureName*为零或者不是一个现有的纹理对象的名称，这个函数就返回GL_FALSE。

9.4.2 创建和使用纹理对象

glBindTexture()函数可以创建和使用纹理对象。当一个纹理对象名称初次绑定时（使用glBindTexture()函数），OpenGL就会创建一个新的纹理对象，并把纹理图像和纹理属性设置为默认值。glTexImage*()、glTexSubImage*()、glCopyTexImage*()、glCopyTexSubImage*()、glTexParameter*()和glPrioritizeTextures()函数的后续调用将把数据存储在这个纹理对象中。纹理对象可以包含一幅纹理图像以及相关的mipmap图像（如果有的话），并包括相关的数据，例如宽度、高度、边框宽度、内部格式、成分的分辨率和纹理属性等。被保存的纹理属性包括缩小和放大过滤器、环绕模式、边框颜色和纹理优先级。

当一个纹理对象以后再次绑定时，它的数据就成为当前的纹理状态（以前绑定的纹理对象的状态被替换）。

---

void **glBindTexture**(GLenum *target*, GLuint *textureName*);

glBindTexture()可以完成3个不同的任务。当*textureName*是一个非零的无符号整数，并且应用程序第一次在这个函数中使用这个值时，这个函数将会创建一个新的纹理对象，并把这个名称分配给它。当*textureName*是一个以前已经创建的纹理对象时，这个纹理对象就成为活动纹理对象。如果*textureName*为零，OpenGL就停止使用纹理对象，并返回到无名称的默认纹理。

当一个纹理对象初次绑定（即被创建）时，它的维数由*target*参数指定，也就是GL_TEXTURE_1D、GL_TEXTURE_2D、GL_TEXTURE_3D或GL_TEXTURE_CUBE_MAP。在初次绑定之后，这个纹理对象的状态立即就等于OpenGL在初始化时target的默认状态。在处于初始状态时，像缩小和放大过滤器、环绕模式、边框颜色和纹理优先级这样的纹理属性都设置为它们的默认值。

在示例程序9-7中，init()函数创建了两个纹理对象。在display()函数中，每个纹理对象用于渲染一个不同的四边形。

示例程序9-7 绑定纹理对象：texbind.c

```
#define checkImageWidth 64
#define checkImageHeight 64
static GLubyte checkImage [checkImageHeight][checkImageWidth][4];
static GLubyte otherImage [checkImageHeight][checkImageWidth][4];

static GLuint texName [2];

void makeCheckImages(void)
{
   int i,j,c;

   for (i =0;i <checkImageHeight;i++){
      for (j =0;j <checkImageWidth;j++){
         c =(((i&0x8)==0)^((j&0x8)==0))*255;
         checkImage [i][j][0] =(GLubyte)c;
         checkImage [i][j][1] =(GLubyte)c;
         checkImage [i][j][2] =(GLubyte)c;
         checkImage [i][j][3] =(GLubyte)255;
         c =(((i&0x10)==0)^((j&0x10)==0))*255;
         otherImage [i][j][0] =(GLubyte)c;
         otherImage [i][j][1] =(GLubyte)0;
         otherImage [i][j][2] =(GLubyte)0;
```

```
            otherImage [i][j][3] =(GLubyte)255;
        }
    }
}

void init(void)
{
    glClearColor(0.0,0.0,0.0,0.0);
    glShadeModel(GL_FLAT);
    glEnable(GL_DEPTH_TEST);

    makeCheckImages();
    glPixelStorei(GL_UNPACK_ALIGNMENT,1);

    glGenTextures(2,texName);
    glBindTexture(GL_TEXTURE_2D,texName [0]);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,
                    GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
                    GL_NEAREST);
    glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,checkImageWidth,
                 checkImageHeight,0,GL_RGBA,GL_UNSIGNED_BYTE,
                 checkImage);

    glBindTexture(GL_TEXTURE_2D,texName [1]);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,
                    GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
                    GL_NEAREST);
    glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_REPLACE);
    glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,checkImageWidth,
                 checkImageHeight,0,GL_RGBA,GL_UNSIGNED_BYTE,
                 otherImage);
    glEnable(GL_TEXTURE_2D);
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);
    glBindTexture(GL_TEXTURE_2D,texName [0]);
    glBegin(GL_QUADS);
    glTexCoord2f(0.0,0.0);glVertex3f(-2.0,-1.0,0.0);
    glTexCoord2f(0.0,1.0);glVertex3f(-2.0,1.0,0.0);
    glTexCoord2f(1.0,1.0);glVertex3f(0.0,1.0,0.0);
    glTexCoord2f(1.0,0.0);glVertex3f(0.0,-1.0,0.0);
    glEnd();
    glBindTexture(GL_TEXTURE_2D,texName [1]);
    glBegin(GL_QUADS);
    glTexCoord2f(0.0,0.0);glVertex3f(1.0,-1.0,0.0);
```

```
    glTexCoord2f(0.0,1.0);glVertex3f(1.0,1.0,0.0);
    glTexCoord2f(1.0,1.0);glVertex3f(2.41421,1.0,-1.41421);
    glTexCoord2f(1.0,0.0);glVertex3f(2.41421,-1.0,-1.41421);
    glEnd();
    glFlush();
}
```

当一个纹理对象再次绑定时，可以编辑这个绑定的纹理对象的内容。我们所调用的能够修改纹理图像或其他属性的函数也能够修改当前绑定的纹理对象的内容以及当前的纹理状态。

在示例程序9-7中，在display()函数之后，我们仍然绑定到texName[1]所表示的纹理名称上。注意，我们并没有调用一个虚构的纹理函数来修改这个纹理对象的数据。

如果使用了mipmap，纹理图像中的所有相关的mipmap层都必须放到一个纹理对象中。在示例程序9-5中，一幅纹理图像的第0层到第5层的mipmap图像都放在一个称为texName的纹理对象中。

9.4.3 清除纹理对象

绑定纹理对象或解除纹理对象的绑定时，它们的数据保存在纹理资源的某个位置中。如果纹理资源有限，删除纹理显然是释放纹理资源的有效方法之一。

---

void **glDeleteTextures**(GLsizei *n*, const GLuint **textureNames*);

删除n个纹理对象，它们的名称由*textureNames*数组提供。被释放的纹理对象名称以后可以重新使用（例如，调用**glGenTextures()**函数可能会重新返回这些名字）。

如果一个当前绑定的纹理对象被删除，这个绑定就会恢复到默认的纹理，就像以0为*textureName*参数调用了**glBindTexture()**函数一样。如果试图删除不存在的纹理对象或名称为0的纹理对象，这个函数将忽略这类操作，并不会产生错误。

9.4.4 常驻纹理工作集

有些OpenGL实现支持高性能的纹理工作集，称为常驻纹理。一般情况下，这些OpenGL实现具有专用的硬件来执行纹理操作，并提供有限的硬件缓存来存储纹理图像。在这种情况下，应该尽可能地使用纹理对象，因为可以把许多纹理加载到这个工作集中，并对它们加以控制。

如果应用程序所需要的所有纹理的总大小超出了缓存的大小，有些纹理就无法放在这个工作集中。如果想判断一个纹理当前是否为常驻纹理，可以绑定这个对象，然后调用glGetTexParameter*v()函数判断与GL_TEXTURE_RESIDENT状态相关联的值。如果想知道多个纹理的纹理常驻状态，可以使用glAreTexturesResident()函数。

GLboolean **glAreTexturesResident**(GLsizei *n*,
const GLuint **textureNames*,
GLboolean **residences*);

查询n个纹理对象的纹理常驻状态。这些纹理对象的名称由*textureNames*参数提供。函数返回的常驻状态保存在residences数组中，后者与*textureNames*数组中的纹理对象一一对应。如果*textureNames*数组中的所有纹理对象都是常驻的，**glAreTextureResident()**函数就返回GL_TRUE。如果*textureNames*数组中有任何一个纹理对象不是常驻纹理，这个函数就返回GL_FALSE，并且residences数组中与textureNames中的非常驻纹理对象相对应的元素也设置为GL_FALSE。

兼容性扩展
glAreTexturesResident

注意，glAreTextureResident()函数返回当前的常驻状态。纹理资源的动态性非常强，纹理资源状态可能会在任何时候发生变化。有些OpenGL实现在第一次使用纹理时会把它们保存在缓存里。在绘制纹理之前检查它的常驻状态是非常必要的。

如果OpenGL实现并没有建立高性能纹理工作集，那么纹理对象总被认为是常驻的。在这种情况下，glAreTextureResident()函数总是返回GL_TRUE，并且基本上不会提供任何信息。

纹理常驻策略

如果可以创建一个纹理工作集，并尽可能地实现最佳的纹理性能，就必须考虑自己使用的OpenGL实现和应用程序的具体情况。例如，在视觉模拟或视频游戏中，在任何情况下都必须坚持性能优先的原则。在这种情况下，绝对不应该访问非常驻纹理。在这类应用程序中，我们可能想在初始化阶段加载所有的纹理，并使它们成为常驻纹理。如果并不拥有足够的纹理内存，可能需要减少所使用的纹理图像的大小、分辨率以及mipmap层数量。或者，可以使用glTexSubImage*()函数反复地复用同一块纹理内存。

注意：如果有几个大小相同并且生命周期较短的纹理，可以使用glTexSubImage*()函数在现有的纹理对象中加载不同的图像。这个技巧较之删除纹理并从头创建新纹理要快速得多。

对于那些需要随时创建纹理图像的应用程序，可能无法避免使用非常驻纹理。如果有些纹理较之其他纹理使用得更加频繁一些，可以向它们分配更高的优先级，增加它们成为常驻纹理的机会。删除纹理对象也可以释放空间。简而言之，向纹理对象分配较低的优先级可能会使这个纹理在纹理资源紧张时被移出高性能纹理工作集。glPrioritizeTextures()函数可以为纹理对象分配优先级。

兼容性扩展
glPrioritizeTextures

void **glPrioritizeTextures**(GLsizei *n*, const GLuint **textureNames*,
const GLclampf **priorities*);

向n个纹理对象分配纹理常驻优先级。这些纹理对象的名称由textureNames数组提供，相应的优先级在priorities数组中设置。在分配之前，priorities数组中的优先级值被截取在[0.0, 1.0]的范围之内。0表示最低的优先级（成为常驻纹理的可能性最低），1表示最高的优先级。

glPrioritizeTextures()函数并不要求textureNames数组中的任何纹理被绑定。但是，在纹理对象绑定之前，这个函数所分配的优先级并不会产生效果。

glTexParameter*()也可以用于设置单个纹理的优先级，但是这只有在这个纹理当前被绑定的情况下才可以。事实上，使用glTexParameter*()是设置默认纹理优先级的唯一方法。

如果几个纹理对象具有相同的优先级，OpenGL实现一般会采用最近最少使用（least recently used，LRU）策略来决定哪些纹理对象应该移出纹理工作集。如果知道自己所使用的OpenGL实现使用了这种策略，可以为所有的纹理对象分配相同的优先级，这可以建立一种比较合理的LRU系统来实现纹理资源的重新分配。

如果OpenGL实现并不使用相同优先级纹理对象的LRU策略，或者并不知道它是不是采用这种策略，可以自己实现LRU策略，精心维护纹理对象的优先级。当一个纹理对象被使用（绑定）时，可以把它的优先级设置为最高，表示它最近正在使用。然后，在一定的时间间隔之后，降低所有纹理对象的优先级。

注意：纹理内存碎片可能会成为一个问题，尤其是多次删除和创建新纹理时。可以按照顺序把所有的纹理对象进行绑定，并把它们全部加载到纹理工作集中。但是，如果按照一种不同的顺序绑定这些纹理对象，可能会导致有些纹理对象成为非常驻纹理。

9.5 纹理函数

本章到目前为止所讨论的例子中，纹理图像表面的颜色值都直接作为被贴图表面的颜色。我们还可以使用纹理图像对物体表面的颜色进行调整，而不是直接贴图。也可以把纹理图像的颜色与物体表面的原先颜色进行组合。可以通过向glTexEnv*()函数提供适当的参数来选择纹理函数。

兼容性扩展

glTexEnv 及其他函数

void **glTexEnv**{if}(GLenum *target*, GLenum *pname*, *TYPE param*);

void **glTexEnv**{if}v(GLenum *target*, GLenum *pname*, const *TYPE* **param*);

设置当前的纹理函数。target必须是GL_TEXTURE_FILTER_CONTROL或GL_TEXTURE_ENV。

如果*target*是GL_TEXTURE_FILTER_CONTROL，那么*pname*必须是GL_TEXTURE_LOD_BIAS，*param*必须是一个浮点值，作为*mipmap*细节层参数的偏移值使用。

如果*target*是GL_TEXTURE_ENV，并且*pname*是GL_TEXTURE_ENV_MODE，那么*param*必须是GL_DECAL、GL_REPLACE、GL_MODULATE、GL_BLEND、GL_ADD或GL_COMBINE之一，它们表示纹理值应该如何与被处理的片断颜色值进行组合。如果*pname*为GL_TEXTURE_ENV_COLOR，那么*param*就是包含了4个浮点值（R、G、B、A）的数组，表示用于GL_BLEND操作的颜色。

如果*target*为GL_POINT_SPRITE，并且*pname*是GL_COORD_REPLACE，那么把*param*设置为GL_TRUE可以启用围绕一个点块纹理的坐标进行迭代。如果*param*设置为GL_FALSE，图元周围的纹理坐标保持不变。

注意：这只是glTexEnv*()函数可以接受的一部分值，并未包括纹理组合器函数。关于GL_COMBINE的详细信息以及glTexEnv*()函数的pname和param参数的完整列表，请参阅第9.9节和表9-8。

纹理函数以及基本内部格式的组合决定了纹理的每个成分是如何应用的。纹理函数是在纹理中被选择的纹理成分以及片断的颜色值（在不使用纹理时的颜色）上进行操作的（注意，纹理成分的选择是在应用了像素传输函数后进行的）。记住，在使用glTexImage*D()函数指定纹理图像时，这个函数的第三个参数就是为每个纹理单元所选择的内部格式。

一共有6种基本的内部格式：GL_ALPHA、GL_LUMINANCE、GL_LUMINANCE_ ALPHA、GL_INTENSITY、GL_RGB和GL_RGBA。可以使用其他内部格式（例如GL_LUMINANCE6_ALPHA2或GL_R3_G3_B2）为纹理成分指定分辨率，这些格式都与6种基本格式之一相匹配。

纹理计算最终是在RGBA模式下进行的，但是有些内部格式并不属于RGB模式。表9-4显示了如何从不同的纹理格式中提取RGBA值，其中包括一些不太直观的方法。

表9-5和表9-6显示了纹理函数（除GL_COMBINE之外）和基本的内部格式

表9-4 从不同的纹理格式中提取颜色值

基本内部格式	所提取的源颜色（R，G，B，A）
GL_ALPHA	(0，0，0，A)
GL_LUMINANCE	(L，L，L，1)
GL_LUMINANCE_ALPHA	(L，L，L，A)
GL_INTENSITY	(I，I，I，I)
GL_RGB	(R，G，B，1)
GL_RGBA	(R，G，B，A)

如何决定纹理的每个成分所应用的纹理应用公式。

在表9-5和表9-6中，注意下面这些下标：

- s表示纹理源颜色，由表9-4确定。
- f表示新的片断值。
- c表示用GL_TEXTURE_ENV_COLOR分配的值。
- 如果没有下标，表示最终计算所得的颜色。

在这两个表中，颜色三元组（R、G、B）与一个标量值的乘法表示R、G、B成分都与这个标量值相乘。两个颜色三元组相乘或相加表示第一种颜色的每个成分与第二种颜色的对应成分相乘或相加。

表9-5 替换、调整和贴花纹理函数

基本内部格式	GL_REPLACE函数	GL_MODULATE函数	GL_DECAL函数
GL_ALPHA	$C = C_f$ $A = A_s$	$C = C_f$ $A = A_f A_s$	未定义
GL_LUMINANCE	$C = C_s$ $A = A_f$	$C = C_f C_s$ $A = A_f$	未定义
GL_LUMINANCE_ALPHA	$C = C_s$ $A = A_s$	$C = C_f C_s$ $A = A_f A_s$	未定义
GL_INTENSITY	$C = C_s$ $A = C_s$	$C = C_f C_s$ $A = A_f C_s$	未定义
GL_RGB	$C = C_s$ $A = A_f$	$C = C_f C_s$ $A = A_f$	$C = C_s$ $A = A_f$
GL_RGBA	$C = C_s$ $A = A_s$	$C = C_f C_s$ $A = A_f A_s$	$C = C_f(1 - A_s) + C_s A_s$ $A = A_f$

表9-6 混合和添加纹理函数

基本内部格式	GL_BLEND函数	GL_ADD函数
GL_ALPHA	$C = C_f$ $A = A_f A_s$	$C = C_f$ $A = A_f A_s$
GL_LUMINANCE	$C = C_f(1 - C_s) + C_c C_s$ $A = A_f$	$C = C_f + C_s$ $A = A_f$
GL_LUMINANCE_ALPHA	$C = C_f(1 - C_s) + C_c C_s$ $A = A_f A_s$	$C = C_f + C_s$ $A = A_f A_s$
GL_INTENSITY	$C = C_f(1 - C_s) + C_c C_s$ $A = A_f(1 - A_s) + A_c A_s$	$C = C_f + C_s$ $A = A_f + A_s$
GL_RGB	$C = C_f(1 - C_s) + C_c C_s$ $A = A_f$	$C = C_f + C_s$ $A = A_f$
GL_RGBA	$C = C_f(1 - C_s) + C_c C_s$ $A = A_f A_s$	$C = C_f + C_s$ $A = A_f A_s$

“替换”纹理函数在不进行纹理贴图的情况下简单地取片断的颜色，然后将其丢弃，并用纹理的颜色取而代之。如果希望在物体上应用不透明的纹理，就可以使用替换纹理函数。例如，绘制一块带有一个不透明标签的肥皂。

“贴花”纹理函数与替换纹理函数类似，但是它只适用于RGB或RGBA内部格式，并且处理alpha值的方式是不同的。在RGBA内部格式中，片断颜色与纹理颜色的混合比例是由纹理的alpha值决定的，片断的alpha值并不会产生影响。贴花纹理函数可以用于应用alpha混合纹理，例如机翼上的徽章。

对于“调整”纹理函数而言，片断的颜色将根据纹理图像的内容进行调整。如果基本内部格式是GL_LUMINANCE、GL_LUMINANCE_ALPHA或GL_INTENSITY，颜色值就与相同的值相乘，因此纹理图像对片断颜色的调整结果就是片断的原来颜色（如果亮度或强度值为1）和黑色（如果亮度或强度值为0）之间的颜色。对于GL_RGB和GL_RGBA内部格式，源颜色的每种成分与纹理中的对应值（有可能不同）相乘。如果不存在alpha值，它就与片断的alpha值相乘。调整纹理函数非常适用于光照，因为带光照的多边形的颜色可以用于衰减纹理的颜色。由于这个原因，本书所附彩图中的绝大多数纹理贴图例子都使用了调整函数。白色、镜面多边形常常用于渲染受光照的纹理物体，并且由纹理图像提供散射颜色。

“添加”纹理函数简单地把纹理颜色与片断颜色相加。如果纹理存在alpha值，它就与片断的alpha值相乘，但GL_INTENSITY格式除外，此时是纹理的强度值与片断的alpha值相加。除非精心选择了纹理颜色和片断颜色，否则使用添加纹理函数很容易导致过度饱和的颜色或者导致颜色值被截取。

“混合”纹理函数是唯一使用GL_TEXTURE_ENV_COLOR所指定颜色的函数。亮度、强度或颜色值的使用方式有点像在GL_TEXTURE_ENV_COLOR下把一个alpha值与片断的颜色值进行混合（第6章提供了一个billboarding的例子，它使用了混合纹理）。

Nate Robin的纹理教程

如果读者已经下载了Nate Robin的教学程序包，现在可以运行“Texture教程”。可以改变纹理贴图环境属性，并观察几个纹理函数的效果。如果使用GL_MODULATE，注意glColor4f()所指定的颜色的效果。如果选择GL_BLEND，观察一下，如果改变env_color，数组所指定的颜色会发生什么情况。

9.6 分配纹理坐标

当我们绘制进行了纹理贴图的场景时，必须为每个顶点提供物体坐标和纹理坐标。经过变换之后，物体坐标决定了应该在屏幕上的哪个地点渲染每个特定的顶点。纹理坐标决定了纹理图像中的哪个纹理单元将分配给这个顶点。就像着色多边形和直线中两个顶点之间的插值匀和一样，顶点之间的纹理坐标也会进行插值匀和（记住，纹理是矩形的数据数组）。

纹理坐标可以由1个、2个、3个或4个坐标组成。它们通常用s、t、r和q坐标来表示，以便与物体坐标（x、y、z和w）和求值器坐标（u、v）进行区分。对于二维纹理，使用的是s和t；对于三维纹理，使用的是s、t和r。q坐标与w坐标类似，一般取值为1，可以用于创建齐次坐标，它将在“q坐标”这个高级特性中讲述。用于指定纹理坐标的函数glTexCoord*()与glVertex*()、glColor*()和glNormal*()函数类似，它具有几个相似的变型，并且也是在一对glBegin()和glEnd()中使用的。通常，纹理坐标值的范围是从0到1。但是，也可以向它分配这个范围之外的值，其结果将第9.6.2节中描述。

兼容性扩展
glTexCoord

void **glTexCoord**{1234}{sifd}(*TYPE coords*);

void **glTexCoord**{1234}{sifd}v(const *TYPE *coords*);

设置当前纹理坐标（*s*, *t*, *r*, *q*）。接下来对**glVertex***()的调用将导致当前纹理坐标被分配给这个函数所指定的顶点。在**glTexCoord1***()中，*s*坐标设置为指定的值，*t*和*r*坐标设置为0，*q*坐标设置为1。**glTexCoord2***()允许指定*s*和*t*坐标，而*r*和*q*则分别设置为0和1。在**glTexCoord3***()中，*q*设置为1，其他坐标则设置为指定的值。可以用**glTexCoord4***()指定所有4个坐标的值。使用适当的后缀（*s*, *i*, *f*或*d*）以及对应的TYPE（GLshort、GLint、GLfloat或GLdouble），可以指定坐标的数据类型。可以单独提供各个坐标值，也可以使用这个函数的向量版本通过一个数组提供所有的坐标。在进行任何纹理贴图之前，纹理坐标值与一个4×4的纹理矩阵相乘（参见第9.12节）。注意，整型的纹理坐标值是直接使用的，而不是像普通的坐标一样被映射到[−1, 1]的范围之内。

下一节将讨论如何计算适当的纹理坐标。我们可能不必显式地分配它们，而是选择通过顶点坐标函数，由OpenGL自动计算纹理坐标（参见第9.7节）。

Nate Robin的纹理教程

如果读者已经下载了Nate Robin的教学程序包，现在可以运行“Texture教程”，并在4个不同的顶

点上试验glTexCoord2f()的参数，观察如何使用整个纹理图像的一部分进行贴图。（如果把纹理坐标设置为小于0或大于1会发生什么情况？）

9.6.1 计算正确的纹理坐标

二维纹理是正方形或矩形的图像，一般映射到多边形模型上。在最简单的情况下，一幅矩形纹理图像映射到一个矩形的模型上。例如，我们所使用的纹理可能是一幅扫描下来的砖墙图像，并且用矩形表示建筑物的砖墙。假设砖墙和纹理都是正方形的，并且我们希望把整幅纹理图像映射到整面砖墙上。这个纹理正方形的纹理坐标按照逆时针方向依次是（0，0）、（1，0）、（1，1）和（0，1）。当绘制砖墙时，只要在指定砖墙的顶点时（按照逆时针顺序）把这4个坐标设置为纹理坐标就可以了。

现在，假设这面砖墙的高度只有宽度的2/3，而纹理仍然是正方形的。为了避免使纹理产生扭曲，需要把砖墙映射到纹理的一个子区域，使纹理和砖墙保持相同的纵横比。假设我们决定把纹理左边2/3部分映射到砖墙上，需要用（0，0）、（1，0）、（1，2/3）和（0，2/3）来表示纹理坐标（逆时针方向）。

下面来看一个稍微复杂一点的例子，假设想在屏幕上显示一个罐头，这个罐头的四周裹着一个标签。为了获取纹理图像，可以买一个罐头，并取下标签，然后把它扫描下来。假设这个标签的高度为4个单位，宽度为12个单位，它的纵横比就是3：1。由于纹理的纵横比必须是从2"到1，因此我们可以简单地去掉上面1/3的部分，也可以通过适当的剪切和粘贴实现正确的纵横比。现在，假设我们决定去掉上面1/3的部分，并且用长度为4单位（罐头的高度）、宽度为12/30单位（罐头周长的1/30）的30个多边形来模拟表示罐头的圆柱体，可以向这30个矩形分配下面的纹理坐标：

1: (0, 0), (1/30, 0), (1/30, 2/3), (0, 2/3)

2: (1/30, 0), (2/30, 0), (2/30, 2/3), (1/30, 2/3)

3: (2/30, 0), (3/30, 0), (3/30, 2/3), (2/30, 2/3)

...

30: (29/30, 0), (1, 0), (1, 2/3), (29/30, 2/3)

只有少数曲面（例如圆锥和圆柱）可以在不产生扭曲的情况下映射到平表面上。其他所有曲面在映射到平表面时都会产生一定程度的扭曲。一般而言，表面的曲率越大，纹理所需要的扭曲度也就越大。

如果并不介意纹理的扭曲，寻找合适的映射方法还是相当容易的。例如，考虑一个表面坐标由$(\cos\theta\cos\phi, \cos\theta\sin\phi, \sin\theta)$确定的球体，其中$0\leqslant\theta\leqslant2\pi$，并且$0\leqslant\phi\leqslant\pi$。$\theta-\phi$矩形可以直接映射到一个矩形的纹理图像，但是越靠近球体的两极，纹理的扭曲度就越大。纹理图像的整个顶部边缘被映射到北极，整个底部边缘被映射到南极。在其他表面中，例如中间有一个大洞的圆环面，当它的自然表面坐标映射到纹理坐标时，只需要极少的扭曲，因此它适用于许多应用程序。图9-9显示了两个圆环面，其中一个圆环面中间的洞较小（因此在靠近中心的地方存在很大的扭曲），另一个圆环面中间的洞较大（因此只需要很少的扭曲）。

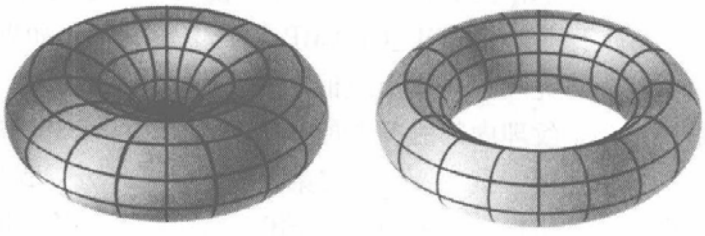


图9-9 纹理映射的扭曲

如果对由求值器（参见第12章）所产生的样条表面进行纹理贴图，表面的u和v参数有时候可以作为纹理坐标使用。一般情况下，把纹理图像映射到由多边形模拟的曲面需要相当程度的艺术想像力。

9.6.2 重复和截取纹理

可以分配[0，1]范围之外的纹理坐标，并且在纹理图像中对它们进行截取或重复。在重复纹理模式下，如果有一个很大的平面，它的纹理坐标范围在各个方向上都是从0.0到10.0，那么这个平面上将出现100份纹理拷贝，它们将平铺在屏幕上。在进行纹理重复时，纹理坐标的整数部分被忽略，纹理图像的拷贝将平铺在表面上。在大部分纹理需要重复的应用场合，纹理顶部的纹理单元应该与底部的纹理单元相匹配，左边缘和右边缘的纹理单元也应该匹配。

可以使用“镜像”重复，也就是表面上相邻的平铺纹理图像呈镜面反转状。例如，在纹理坐标范围[0，1]内部，纹理的方向可能是从左到右（或自底向下，或从近到远），但“镜像”重复可以在纹理坐标范围[1，2]内部把方向重置为从右向左，然后在坐标范围[2，3]内再回到从左向右，接下来以此类推。

另一种可以使用的方法是对纹理坐标进行截取。所有大于1.0的值都设置为1.0，所有小于0.0的值都设置为0.0。截取适用于想在一个大型表面上只出现一份纹理拷贝的场合。如果表面的纹理坐标在各个方向上都是从0.0到10.0，那么只会在表面的左下角出现一份纹理拷贝。

如果使用的是具有边框的纹理，或者指定了一种纹理边框颜色，环绕模式和过滤方法（参见第9.3节）都会影响边框信息的使用方式。

如果使用了过滤模式GL_NEAREST，纹理中最邻近的纹理单元将被使用。在绝大多数环绕模式下，边框（或边框颜色）是被忽略的。但是，如果纹理坐标位于[0，1]的范围之外，并且环绕模式为GL_CLAMP_TO_BORDER，那么最邻近的边框纹理单元会被选中。如果不存在边框，就使用常量边框颜色。

如果已经选择了GL_LINEAR作为过滤方法，OpenGL就使用一个2×2的包含纹理数据的数组的加权组合作为纹理（对于二维数组）。如果存在边框或边框颜色，纹理和边框颜色就一起使用，如下所示：

- 对于环绕模式GL_REPEAT，边框总是被忽略。2×2的加权纹理单元会环绕到纹理相反一侧的边缘上。因此，右边缘的纹理单元会与左边缘的纹理单元求均值，顶部和底部的纹理单元也会求均值。
- 对于环绕模式GL_CLAMP，OpenGL就在一个2×2的加权纹理单元数组中使用取自边框的纹理单元（或GL_TEXTURE_BORDER_COLOR）。
- 对于环绕模式GL_CLAMP_TO_EDGE，边框始终被忽略。位于纹理边缘或靠近纹理边缘的纹理单元将用于纹理计算，但并不使用纹理边框上的纹理单元。
- 对于环绕模式GL_CLAMP_TO_BORDER，如果纹理坐标位于范围[0，1]之外，那么只有边框纹理单元（如果没有边框，则使用常量边框颜色）用于纹理。在靠近纹理坐标边缘的地方，无论是边框还是纹理内部的纹理单元都可能根据一个2×2的数组进行采样。

如果使用了截取，可以避免表面的剩余部分受到纹理的影响。为此，可以在纹理的边缘（或边框，如果指定了边框）上使用alpha值0。贴花纹理函数在计算中直接使用纹理的alpha值。我们可能还需要用适当的源和目标混合因子来启用混合功能（参见第6.1节）。

为了观察环绕的效果，必须使纹理坐标超出[0.0，1.0]的范围。在示例程序9-1中我们可以修改各个正方形的纹理坐标，使纹理坐标从0.0映射到4.0，如下所示：

```
glBegin(GL_QUADS);
  glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
  glTexCoord2f(0.0, 4.0); glVertex3f(-2.0, 1.0, 0.0);
```

```
    glTexCoord2f(4.0, 4.0); glVertex3f(0.0, 1.0, 0.0);
    glTexCoord2f(4.0, 0.0); glVertex3f(0.0, -1.0, 0.0);

    glTexCoord2f(0.0, 0.0); glVertex3f(1.0, -1.0, 0.0);
    glTexCoord2f(0.0, 4.0); glVertex3f(1.0, 1.0, 0.0);
    glTexCoord2f(4.0, 4.0); glVertex3f(2.41421, 1.0, -1.41421);
    glTexCoord2f(4.0, 0.0); glVertex3f(2.41421, -1.0, -1.41421);
  glEnd();
```

在GL_REPEAT环绕模式下，程序的结果如图9-10所示。

在这个例子里，纹理在s和t方向上都进行了重复，这是下面这两个glTexParameter*()调用的结果：

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

有些OpenGL实现支持GL_MIRRORED_REPEAT环绕模式，它会在每个整型纹理坐标边框上反转纹理的方向。图9-11显示了普通的重复环绕模式（左侧）和镜像重复模式（右侧）的对比。

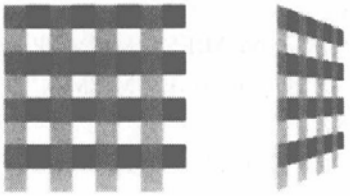


图9-10 重复纹理图

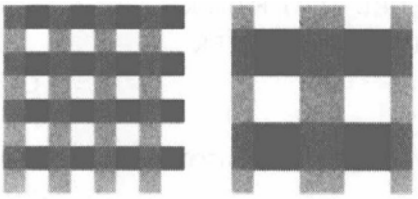


图9-11 GL_REPEAT与GL_MIRRORED_REPEAT的比较

在图9-12中，每个方向都使用了GL_CLAMP。当纹理坐标s或t大于1时，OpenGL就使用纹理坐标正好为1的纹理单元。

环绕模式在每个方向上都是独立的。可以在一个方向上使用截取，而在另一个方向上使用重复，如图9-13所示。

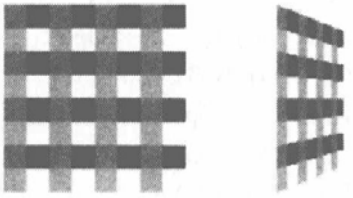


图9-12 截取纹理



图9-13 重复和截取纹理

现在我们已经看到了glTexParameter*()的几个参数，这个函数的说明如下：

void **glTexParameter**{if}(GLenum *target*, GLenum *pname*, *TYPE param*);

void **glTexParameter**{if}**v**(GLenum *target*, GLenum *pname*, const *TYPE* **param*);

void **glTexParameterI**{i ui}**v**(GLenum *target*, GLenum *pname*, const *TYPE* **param*);

设置不同的参数，当纹理应用于片断或存储于纹理对象时控制它的处理方式。target参数是GL_TEXTURE_1D、GL_TEXTURE_2D、GL_TEXTURE_3D或GL_TEXTURE_CUBE_MAP，分别对应于各种纹理类型。*pname*和*param*参数可以取的值见表9-7。可以使用向量版本

兼容性扩展
GL_CLAMP
GL_TEXTURE_BORDER_COLOR
GL_GENERATE_MIPMAP
GL_TEXTURE_PRIORITY

的函数，提供一个数组来表示GL_TEXTURE_ BORDER_COLOR，或者使用非向量版本单独为其他参数提供值。如果传递给**glTexParameterf*()**的值以整型的方式提供，它们就根据表4-1转换为浮点数。此外，它们还将截取在[0，1]的范围之内。传递给**glTexParameterI*()**的整数值不会转换。类似地，如果把整数值传递给**glTexParameterI*()**来设置其浮点值参数，它们将会根据表4-1转换。

表9-7 glTexParameter*()函数的参数

参 数	值
GL_TEXTURE_WRAP_S	GL_CLAMP、GL_CLAMP_TO_EDGE、GL_CLAMP_TO_BORDER、GL_REPEAT、GL_MIRRORED_REPEAT
GL_TEXTURE_WRAP_T	GL_CLAMP、GL_CLAMP_TO_EDGE、GL_CLAMP_TO_BORDER、GL_REPEAT、GL_MIRRORED_REPEAT
GL_TEXTURE_ WRAP_R	GL_CLAMP、GL_CLAMP_TO_EDGE、GL_CLAMP_TO_BORDER、GL_REPEAT、GL_MIRRORED_REPEAT
GL_TEXTURE_MAG_FILTER	GL_NEAREST、GL_LINEAR
GL_TEXTURE_MIN_FILTER	GL_NEAREST、GL_LINEAR、GL_NEAREST_MIPMAP_NEAREST、GL_NEAREST_MIPMAP_LINEAR、GL_LINEAR_MIPMAP_NEAREST、GL_LINEAR_MIPMAP_LINEAR
GL_TEXTURE_BORDER_COLOR	[0.0，1.0]之间的4个任意值（对于非整数纹理格式，）或者带符号或无符号的整数值（对于整数纹理格式）
GL_TEXTURE_PRIORITY	[0.0，1.0]用于当前的纹理对象
GL_TEXTURE_MIN_LOD	任何浮点值
GL_TEXTURE_MAX_LOD	任何浮点值
GL_TEXTURE_BASE_LEVEL	任何非负整数
GL_TEXTURE_MAX_LEVEL	任何非负整数
GL_TEXTURE_LOD_BIAS	任何浮点值
GL_DEPTH_TEXTURE_MODE	GL_RED、GL_LUMINANCE、GL_INTENSITY、GL_ALPHA
GL_TEXTURE_COMPARE_MODE	GL_TEXTURE_COMPARE_MODE、GL_NONE，GL_COMPARE_REF_TO_TEXTURE（针对OpenGL3.0及其以上版本），或者GL_COMPARE_R_TO_TEXTURE（针对OpenGL2.1以下版本，包括2.1）
GL_TEXTURE_COMPARE_FUNC	GL_LEQUAL、GL_GEQUAL、GL_LESS、GL_GREATER、GL_EQUAL、GL_NOTEQUAL、GL_ALWAYS、GL_NEVER
GL_GENERATE_MIPMAP	GL_TRUE、GL_FALSE



尝试一下

图9-12和图9-13在绘图时使用了GL_NEAREST作为缩小和放大过滤器。如果把过滤器的值修改为GL_LINEAR会发生什么情况呢？最终的图像看上去应该更为模糊。

也可以在计算纹理时使用边框信息。为了进行最简单的演示，可以把GL_TEXTURE_BORDER_COLOR设置为一种引人注目的颜色。把过滤器设置为GL_NEAREST，并把环绕模式设置为GL_CLAMP_TO_BORDER之后，边框颜色就会影响被纹理贴图的物体（影响[0，1]范围之外的纹理坐标）。如果过滤器设置为GL_LINEAR并且环绕模式设置为GL_CLAMP，边框也会对纹理产生影响。

如果把环绕模式切换为GL_CLAMP_TO_EDGE或GL_REPEAT，又会发生什么情况呢？不管是哪种情况，边框颜色都将被忽略。

Nate Robin的纹理教程

读者可以运行Nate Robin的“Texture”教程，观察环绕参数GL_REPEAT和GL_CLAMP的效果。需要把顶点（glTexCoord2f()的参数）的纹理坐标设置为小于0或大于1，以观察重复或截取的效果。

9.7 纹理坐标自动生成

可以使用纹理贴图生成模型的轮廓线，或者模拟具有光泽的模型对任意环境的反射。为了实现这些效果，可以让OpenGL自动生成纹理坐标，而不是使用glTexCoord*()函数显式地分配纹理坐标。

为了自动生成纹理坐标，可以使用glTexGen()函数。

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

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

指定用于自动生成纹理坐标的函数。第一个参数coord必须是GL_S、GL_T、GL_R或GL_Q，分别表示需要自动生成s、t、r或q坐标。*pname*参数为GL_TEXTURE_GEN_MODE、GL_OBJECT_PLANE或GL_EYE_PLANE。如果是GL_TEXTURE_GEN_MODE，*param*就是一个整数（或者，在向量版本中，是一个整型指针），是GL_OBJECT_PLANE、GL_EYE_PLANE、GL_SPHERE_MAP、GL_REFLECTION_MAP或GL_NORMAL_MAP之一。这些符号常量决定了使用哪个函数生成纹理坐标。如果*pname*取其他可能的值，*param*就是一个指向数组的指针（向量版本），这个数组指定了纹理生成函数的参数。

兼容性扩展
glTexGen and all accpted tokens.

不同的纹理坐标生成方法具有不同的用途。当纹理图像与移动的物体保持固定时，在物体坐标中指定参考平面是最为合适的。因此，可以使用GL_OBJECT_LINEAR模式把木纹图像映射到桌子的表面。为了产生移动物体的动态轮廓线，在视觉坐标（使用GL_EYE_LINEAR模式）中指定参考平面是最为合适的。地质测量专家用钻井勘测石油或天然气时，也常常使用GL_EYE_LINEAR模式。钻井可以用不同的颜色进行渲染，表示不同深度的岩石层。GL_SPHERE_MAP和GL_REFLECTION_MAP模式主要用于球体环境的贴图，GL_NORMAL_MAP模式则用于立方图纹理（请参见第9.7.2节和第9.7.3节）。

9.7.1 创建轮廓线

当指定了GL_TEXTURE_GEN_MODE和GL_OBJECT_LINEAR时，纹理生成函数就是顶点（x_0, y_0, z_0, w_0）的物体坐标的线性组合：

$$\text{生成的坐标} = p_1x_0 + p_2y_0 + p_3z_0 + p_4w_0$$

p_1，……，p_4的值是由glTexGen*v()函数的param参数提供的，此时pname参数设置为GL_OBJECT_PLANE。如果p_1，……，p_4已经进行了正确的规范化，这个函数将会给出从这个顶点到一个平面的距离。例如，如果$p_2 = p_3 = p_4 = 0$并且$p_1 = 1$，这个函数将会给出这个顶点和$x = 0$平面之间的距离。这个距离在这个平面的其中一侧是正的，在另一侧是负的。如果这个顶点正好就在这个平面上，这个距离就是0。

示例程序9-8首先在一个茶壶上绘制分隔均匀的轮廓线。这些直线提示了与$x = 0$平面的距离。$x = 0$平面的系数位于下面这个数组中：

```
static GLfloat xequalzero[] = {1.0, 0.0, 0.0, 0.0};
```

由于一共只显示1个属性（与平面的距离），所以使用一维纹理图像就足够了。这个纹理图像为一

种绿色常量颜色，只是它每隔一定的间隔包含了一个红色的标记。由于这个茶壶位于xy平面上，因此这些轮廓线都垂直于它的底部。彩图18显示了这个程序所绘制的图像。

在这个程序中，按下“s”键可以把参考平面的参数修改为：

```
static GLfloat slanted[] = {1.0, 1.0, 1.0, 0.0};
```

这些轮廓线与$x + y + z = 0$平面平行，并沿一个角度切割茶壶，如彩图18所示。为了把参考平面恢复为它的初始值$x = 0$，可以按“x”键。

示例程序9-8 自动生成纹理坐标：texgen.c

```
#define stripeImageWidth 32
GLubyte stripeImage [4*stripeImageWidth];

static GLuint texName;

void makeStripeImage(void)
{
   int j;

   for (j =0;j <stripeImageWidth;j++){
      stripeImage [4*j] =(GLubyte)((j<=4)?255 :0);
      stripeImage [4*j+1] =(GLubyte)((j>4)?255 :0);
      stripeImage [4*j+2] =(GLubyte)0;
      stripeImage [4*j+3] =(GLubyte)255;
   }
}

/*planes for texture-coordinate generation */
static GLfloat xequalzero [] ={1.0,0.0,0.0,0.0};
static GLfloat slanted [] ={1.0,1.0,1.0,0.0};
static GLfloat *currentCoeff;
static GLenum currentPlane;
static GLint currentGenMode;

void init(void)
{
   glClearColor(0.0,0.0,0.0,0.0);
   glEnable(GL_DEPTH_TEST);
   glShadeModel(GL_SMOOTH);

   makeStripeImage();
   glPixelStorei(GL_UNPACK_ALIGNMENT,1);

   glGenTextures(1,&texName);
   glBindTexture(GL_TEXTURE_1D,texName);
   glTexParameteri(GL_TEXTURE_1D,GL_TEXTURE_WRAP_S,GL_REPEAT);
   glTexParameteri(GL_TEXTURE_1D,GL_TEXTURE_MAG_FILTER,
                   GL_LINEAR);
   glTexParameteri(GL_TEXTURE_1D,GL_TEXTURE_MIN_FILTER,
                   GL_LINEAR);
   glTexImage1D(GL_TEXTURE_1D,0,GL_RGBA,stripeImageWidth,0,
                GL_RGBA,GL_UNSIGNED_BYTE,stripeImage);
```

```
    glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_MODULATE);
    currentCoeff =xequalzero;
    currentGenMode =GL_OBJECT_LINEAR;
    currentPlane =GL_OBJECT_PLANE;
    glTexGeni(GL_S,GL_TEXTURE_GEN_MODE,currentGenMode);
    glTexGenfv(GL_S,currentPlane,currentCoeff);

    glEnable(GL_TEXTURE_GEN_S);
    glEnable(GL_TEXTURE_1D);
    glEnable(GL_CULL_FACE);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_NORMALIZE);
    glFrontFace(GL_CW);
    glCullFace(GL_BACK);
    glMaterialf(GL_FRONT,GL_SHININESS,64.0);
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);

    glPushMatrix();
    glRotatef(45.0,0.0,0.0,1.0);
    glBindTexture(GL_TEXTURE_1D,texName);
    glutSolidTeapot(2.0);
    glPopMatrix();
    glFlush();
}

void reshape(int w,int h)
{
    glViewport(0,0,(GLsizei)w,(GLsizei)h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <=h)
       glOrtho(-3.5,3.5,-3.5*(GLfloat)h/(GLfloat)w,
               3.5*(GLfloat)h/(GLfloat)w,-3.5,3.5);
    else
       glOrtho(-3.5*(GLfloat)w/(GLfloat)h,
               3.5*(GLfloat)w/(GLfloat)h,-3.5,3.5,-3.5,3.5);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void keyboard(unsigned char key,int x,int y)
{
    switch (key){
       case 'e':
       case 'E':
          currentGenMode =GL_EYE_LINEAR;
```

```
            currentPlane =GL_EYE_PLANE;
            glTexGeni(GL_S,GL_TEXTURE_GEN_MODE,currentGenMode);
            glTexGenfv(GL_S,currentPlane,currentCoeff);
            glutPostRedisplay();
            break;
         case 'o':
         case 'O':
            currentGenMode =GL_OBJECT_LINEAR;
            currentPlane =GL_OBJECT_PLANE;
            glTexGeni(GL_S,GL_TEXTURE_GEN_MODE,currentGenMode);
            glTexGenfv(GL_S,currentPlane,currentCoeff);
            glutPostRedisplay();
            break;
         case 's':
         case 'S':
            currentCoeff =slanted;
            glTexGenfv(GL_S,currentPlane,currentCoeff);
            glutPostRedisplay();
            break;
         case 'x':
         case 'X':
            currentCoeff =xequalzero;
            glTexGenfv(GL_S,currentPlane,currentCoeff);
            glutPostRedisplay();
            break;
         case 27:
            exit(0);
            break;
         default:
            break;
      }
   }

   int main(int argc,char**argv)
   {
      glutInit(&argc,argv);
      glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB |GLUT_DEPTH);
      glutInitWindowSize(256,256);
      glutInitWindowPosition(100,100);
      glutCreateWindow(argv [0]);
      init();
      glutDisplayFunc(display);
      glutReshapeFunc(reshape);
      glutKeyboardFunc(keyboard);
      glutMainLoop();
      return 0;
   }
```

可以通过向glEnable()函数传递GL_TEXTURE_GEN_S来启用s坐标的纹理坐标生成。为了生成其他坐标，可以使用GL_TEXTURE_GEN_T、GL_TEXTURE_GEN_R或GL_TEXTURE_GEN_Q参数。在glDisable()函数中使用对应的常量值可以禁用纹理坐标生成。另外，注意使用GL_REPEAT模式可

以沿茶壶重复生成轮廓线。

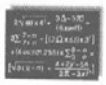
GL_OBJECT_LINEAR函数在模型的坐标系统中计算纹理坐标。示例程序9-8首先使用了GL_OBJECT_LINEAR函数，因此轮廓线保持与茶壶的底座垂直，不管这个茶壶如何旋转，也不管是从哪个方向观察它。但是，如果按下了“e”键，纹理生成模式就会从GL_OBJECT_LINEAR更改为GL_EYE_LINEAR，轮廓线就会相对于视觉坐标系统进行计算（按下“o”键可以把纹理生成模式恢复为GL_OBJECT_LINEAR）。如果参考平面为x = 0，从观察点看过去，茶壶上有几条与xy平面平行的红带，如彩图18所示。从数学的角度而言，这相当于把向量 (p_1, p_2, p_3, p_4) 与模型视图矩阵的逆矩阵相乘，其结果可以用于计算从顶点到平面的距离。纹理坐标是用下面的函数生成的：

$$\text{生成的坐标} = p_1' x_e + p_2' y_e + p_3' z_e + p_4' w_e$$
$$\text{其中 } (p_1' p_2' p_3' p_4') = (p_1 p_2 p_3 p_4)\mathbf{M}^{-1}$$

在此例中，(x_e, y_e, z_e, w_e) 是顶点的视觉坐标，p_1，……，p_4的值是由glTexGen*()函数的param参数提供的，此时pname参数设置为GL_EYE_PLANE。由于主要的值只有到了被指定的时候才会计算，所以这个操作的计算开销并不像看上去那么大。

在所有这些例子中，我们只使用了一个纹理坐标来生成轮廓线。但是，如果需要，也可以独立地生成s、t和r纹理坐标，表示这个顶点到2个或3个不同平面的距离。只要适当地构建一个二维或三维纹理图像，就可以同时看到最终生成的2组或3组轮廓线。为了增加复杂度，可以混合使用纹理生成函数。例如，可以使用GL_OBJECT_LINEAR计算s坐标，并用GL_EYE_LINEAR生成t坐标。

9.7.2 球体纹理



高级话题

环境纹理的目标是渲染具有完美反射能力的物体，它的表面颜色就是反射到人眼周围环境的颜色。换句话说，如果我们在房间里观察一件擦得锃亮、具有完美反射能力的银器时，在银器表面上所看到的就是墙壁、地板和房间内其他物体的反射图像（环境纹理的一个经典例子就是电影《终结者2》中邪恶的液体机械人）。在银器表面上能够看到的环境反射物体取决于眼睛的位置以及银器的位置和摆放角度。为了实现环境纹理贴图，我们需要做的就是创建一幅适当的纹理图像，并让OpenGL生成纹理坐标。

环境纹理是一种近似模拟，它基于这样的假设：和光亮的物体表面相比，环境中其他物体都相距甚远。也就是说，这些物体都可以看成是一个很大的房间内所放置的小物体。进行了这样的假设之后，为了确定光亮表面上某个点的颜色，我们需要取一条从观察点到这个点的光线，然后把这条光线从表面反射出来。这条反射光线的方向完全决定了这个点需要绘制的颜色。在一幅平面纹理图像中对各个方向的颜色进行编码相当于把一个擦得锃亮的完美球体放在环境的中央，然后在极远处用长焦镜头对它进行拍照。用数学语言描述，就是镜头具有无限长的焦距，并且相机位于无限远处。因此，需要编码的区域就是覆盖整个纹理图像的一个圆形区域，它与纹理图像的顶、底、左、右边缘相切。这个圆形区域之外的纹理值不会对结果产生影响，因为它们不会在环境纹理中使用。

为了创建正确的环境纹理图像，需要一个很大的银面球体，然后在无限远处使用无限长焦距的照相机拍摄这个球体的照片，最后把照片扫描下来。为了近似地实现这种效果，可以使用角度极大的广角镜头（鱼眼镜头）拍摄照片，并把它扫描下来。彩图21显示了一幅使用这种镜头拍摄的照片以及把这幅图像作为环境纹理图像使用的结果。

创建了环境纹理图像之后，需要调用OpenGL的环境纹理贴图算法。这个算法在球体表面上找到一个与被渲染的物体上的点具有相同正切面的点，并且把球体上这个点的颜色绘制成那个物体上对应点的颜色。

为了自动生成纹理坐标，对环境纹理贴图提供支持，可以在程序中使用如下代码：

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

GL_SPHERE_MAP常量为环境纹理生成适当的纹理坐标。如上述代码所示，我们需要指定s和t方向的纹理坐标。但是，我们并不需要为纹理坐标生成函数指定任何参数。

GL_SPHERE_MAP纹理函数使用如下数学步骤来生成纹理坐标：

1) **u**是从原点指向顶点的单位向量（在视觉坐标中）。

2) **n**'是当前法线向量，在变换为视觉坐标之后。

3) **r**是反射向量 $(r_x, r_y, r_z)^{\mathrm{T}}$，它是通过$\mathbf{u} - 2\mathbf{n}'\mathbf{n}'^{\mathrm{T}}\mathbf{u}$这个公式计算的。

4) 中间值m是通过下面这个公式计算的：

$$m = 2\sqrt{r_x^2 + r_y^2 + (r_z + 1)^2}$$

5) 最后，可以使用下面这两个公式计算s和t纹理坐标：

$$s = r_x / m + \frac{1}{2}$$

和

$$t = r_y / m + \frac{1}{2}$$

9.7.3 立方图纹理



高级话题

立方图纹理是一种特殊的纹理技术，它用6幅二维纹理图像构成一个以原点为中心的纹理立方体。对于每个片断，纹理坐标（s, t, r）被当作方向向量看待，每个纹理单元都表示从原点所看到的纹理立方体上的图像。立方图纹理非常适用于实现环境、反射和光照效果。立方图纹理还可以把纹理环绕到球体物体上，使纹理单元均匀地分布于它的各个面上。

可以调用glTexImage2D()函数6次，分别使用target参数表示立方体的各个面（+X，−X，+Y，−Y，+Z，−Z），从而创建一个立方图纹理。顾名思义，每个立方图纹理必须具有相同的维度，使立方体的每个面都由相同数量的纹理单元组成，如下面这段代码所示。其中，imageSize设置为2的整数次方：

```
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X, 0, GL_RGBA,
 imageSize, imageSize, 0, GL_RGBA, GL_UNSIGNED_BYTE, image1);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_X, 0, GL_RGBA,
 imageSize, imageSize, 0, GL_RGBA, GL_UNSIGNED_BYTE, image4);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Y, 0, GL_RGBA,
 imageSize, imageSize, 0, GL_RGBA, GL_UNSIGNED_BYTE, image2);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, 0, GL_RGBA,
 imageSize, imageSize, 0, GL_RGBA, GL_UNSIGNED_BYTE, image5);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Z, 0, GL_RGBA,
```

```
 imageSize, imageSize, 0, GL_RGBA, GL_UNSIGNED_BYTE, image3);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, 0, GL_RGBA,
 imageSize, imageSize, 0, GL_RGBA, GL_UNSIGNED_BYTE, image6);
```

为了创建立方图纹理，可以在场景的原点上放置一架照相机（真实的或合成的），然后把相机依次对准各个轴的正方向和负方向，拍摄6幅视野为90度的“快照”。这些“快照”把3D空间划分为6个在原点相交的平截头体（即视景体）。

立方图纹理的功能与其他许多纹理操作是正交的，因此立方图纹理可以使用标准的纹理特性，如纹理边框、mipmap、复制图像、子图像以及多重纹理等。可以使用一种特殊的纹理代理表示立方图纹理（GL_PROXY_TEXTURE_CUBE_MAP），因为立方图纹理所占用的内存常常是普通2D纹理的6倍。应该把立方图纹理视为一个整体，为它指定纹理参数并创建纹理对象，而不是为6个立方体表面分别指定纹理参数以及创建纹理对象。下面的代码用于设置立方图纹理的环绕模式和过滤方法，它把target参数的值设置为GL_TEXTURE_CUBE_MAP：

```
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S,
                GL_REPEAT);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T,
                GL_REPEAT);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R,
                GL_REPEAT);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER,
                GL_NEAREST);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER,
                GL_NEAREST);
```

为了确定某个特定片断所使用的纹理（以及纹理单元），当前纹理坐标（s, t, r）首先根据s、t和r的最大绝对值（主轴）以及它的符号（方向），在6个纹理中选择其一。剩余的两个坐标除以坐标的最大值，得到新的坐标（s', t'），以查找立方图纹理中那个被选择的纹理的对应纹理单元。

虽然可以显式地计算和指定纹理坐标，但这样做不仅繁琐，而且毫无必要。通常，可以调用glTexGen*()函数自动生成立方图纹理的坐标，这个函数可以使用两种特殊的纹理生成模式：GL_REFLECTION_MAP或GL_NORMAL_MAP。

GL_REFLECTION_MAP所使用的计算方式和GL_SPHERE_MAP相同（直到球体纹理坐标计算的第3个步骤，参见第9.7.2节），它把计算产生的（r_x, r_y, r_z）作为纹理坐标（s, t, r）使用。它非常适用于环境纹理贴图，可以替代GL_SPHERE_MAP。

GL_NORMAL_MAP适用于渲染具有无限远处光源（或远处的局部光源）以及散射反射的场景。GL_NORMAL_MAP使用模型视图矩阵把顶点的法线变换为视觉坐标。它产生的（n_x, n_y, n_z）就作为纹理坐标（s, t, r）。示例程序9-9使用GL_NORMAL_MAP进行纹理坐标生成，并且启用了立方图纹理。

示例程序9-9 生成立方图纹理坐标：cubemap.c

```
glTexGeni(GL_S,GL_TEXTURE_GEN_MODE,GL_NORMAL_MAP);
glTexGeni(GL_T,GL_TEXTURE_GEN_MODE,GL_NORMAL_MAP);
glTexGeni(GL_R,GL_TEXTURE_GEN_MODE,GL_NORMAL_MAP);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
glEnable(GL_TEXTURE_GEN_R);
/*  turn on cube map texturing */
glEnable(GL_TEXTURE_CUBE_MAP);
```

9.8 多重纹理

在进行标准的纹理处理时，一次是把1幅图像应用到1个多边形上。多重纹理允许应用几个纹理，在纹理操作管线中把它们逐个应用到同一个多边形上。多重纹理存在一系列的纹理单位（texture unit），每个纹理单位执行各自的纹理操作，并把结果传递给下一个纹理单位，直到所有纹理单位的操作均完成为止。图9-14显示了一个片断是如何经历4次纹理操作的，它在经历每个纹理单位时都要执行一次纹理操作。

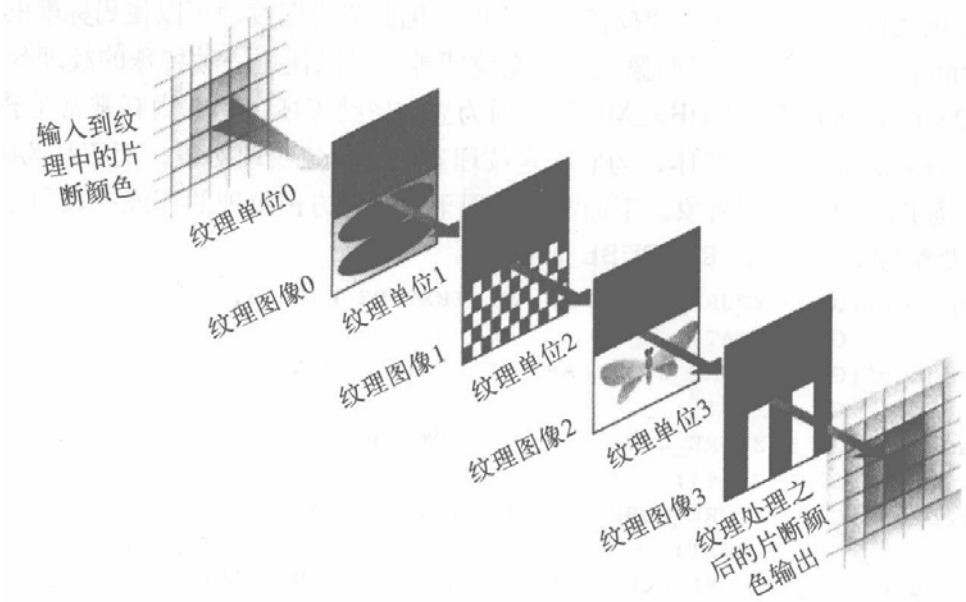


图9-14 多重纹理处理管线

多重纹理能够实现一些高级的渲染技巧，例如光照效果、贴花、合成和细节纹理（detail texture）等。

多重纹理的步骤

为了使用多重纹理，可以执行下面这两个步骤：

注意：在反馈模式下，除了第一个纹理单位之外，多重纹理的行为是未定义的。

1) 对于每个纹理单位，建立相关的纹理状态，包括纹理图像、过滤器、纹理环境、纹理坐标生成和纹理矩阵等。可以使用glActiveTexture()函数更改当前的纹理单位。下面的“建立纹理单位”将详细讨论这个话题。还可以调用glGetIntegerv(GL_MAX_ TEXTURE_UNITS, …)查询当前OpenGL实现所支持的纹理单位的数量。任何OpenGL实现至少必须支持2个纹理单位。

2) 在指定顶点时，可以使用glMultiTexCoord*()函数为每个顶点指定多个纹理坐标，分别用于不同的纹理单位。每个纹理坐标分别用于每个纹理单位的处理。纹理坐标自动生成以及在顶点数组中指定纹理数组属于特殊情况。这些特殊情况将在稍后的“指定纹理坐标的其他方法”中描述。

建立纹理单位

多重纹理引入了纹理单位，用于在应用程序中进行多道纹理处理。每个纹理单位都具有相同的功能，并且具有自己的纹理状态，包括：

- 纹理图像。
- 过滤参数。
- 纹理环境应用。
- 纹理坐标自动生成。

• 顶点数组指定（如果需要的话）。

每个纹理单位根据它的纹理状态，把原先的片断颜色与纹理图像进行组合。然后，把上述操作产生的片断颜色传递给下一个纹理单位（如果后者处于活动状态）。

为了向每个纹理单位分配纹理信息，可以使用glActiveTexture()函数选择需要修改的当前纹理单位。在此之后，对glTexImage*()、glTexParameter*()、glTexEnv*()、glTexGen*()和glBindTexture()函数的调用就像查询当前纹理坐标和当前光栅纹理坐标一样，只影响当前纹理单位。

---

void **glActiveTexture**(GLenum *texUnit*);

选择可以由纹理函数进行修改的当前纹理单位。*texUnit*是一个符号常量，其形式为GL_TEXTUREi，其中i的范围是从0至k−1，k是纹理单位的最大数量。

如果使用纹理对象，可以把一个纹理对象绑定到当前纹理单位。当前纹理单位就具有这个纹理对象所包含的纹理状态的值（包括纹理图像）。

示例程序9-10的代码段分成两个部分。第一部分代码创建了2个普通的纹理对象（假设texels0和texels1定义了纹理图像），第二部分代码使用这2个纹理对象来设置2个纹理单位。

示例程序9-10 在多重纹理下对纹理单位进行初始化：multiex.c

```
/*Two ordinary texture objects are created */
GLuint texNames [2];
glGenTextures(2,texNames);
glBindTexture(GL_TEXTURE_2D,texNames [0]);
glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,32,32,0,GL_RGBA,
             GL_UNSIGNED_BYTE,texels0);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
glBindTexture(GL_TEXTURE_2D,texNames [1]);
glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,16,16,0,GL_RGBA,
             GL_UNSIGNED_BYTE,texels1);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,
                GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,
                GL_CLAMP_TO_EDGE);
/* Use the two texture objects to define two texture units
 * for use in multitexturing.*/
glActiveTexture(GL_TEXTURE0);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D,texNames [0]);
glTexEnvi(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_REPLACE);
glMatrixMode(GL_TEXTURE);
   glLoadIdentity();
   glTranslatef(0.5f,0.5f,0.0f);
   glRotatef(45.0f,0.0f,0.0f,1.0f);
   glTranslatef(-0.5f,-0.5f,0.0f);
glMatrixMode(GL_MODELVIEW);
glActiveTexture(GL_TEXTURE1);
```

```
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D,texNames [1]);
glTexEnvi(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_MODULATE);
```

这样，当一个需要进行纹理处理的多边形被渲染时，就会在渲染中使用这两个纹理单位。在第一个纹理单位中，纹理图像texels0使用了最邻近过滤、重复环绕和替换纹理环境，并根据一个纹理矩阵进行旋转。在第一个纹理单位渲染完成后，这个经过纹理处理的多边形便发送到第二个纹理单位（GL_TEXTURE1），在线性过滤、边缘截取、调整纹理环境以及默认的单位纹理矩阵下应用纹理图像texels1。

注意：对纹理属性组（使用glPushAttrib()、glPushClientAttrib()、glPopAttrib()或glPopClientAttrib()）的操作将会保存或恢复所有纹理单位的纹理状态（除了纹理矩阵状态之外）。

指定顶点以及它们的纹理坐标

在多重纹理中，每个顶点只有一组纹理坐标是不够的，需要为每个顶点的每个纹理单位都指定一组纹理坐标。我们不再使用glTexCoord*()函数，而是必须使用glMultiTexCoord*()函数。这个函数除了指定纹理坐标之外，还指定了纹理单位。

兼容性扩展
glMultiTexCoord

void **glMultiTexCoord{1234}{sifd}**(GLenum *texUnit*, *TYPE coords*);

void **glMultiTexCoord{1234}{sifd}v**(GLenum *texUnit*, const *TYPE* **coords*);

将参数coords中的纹理坐标数据（*s*, *t*, *r*, *q*）用于纹理单位*texUnit*。*texUnit*参数所使用的枚举值与**glActiveTexture()**函数相同。

在示例程序9-11中，一个三角形需要两组纹理坐标，因为多重纹理需要两个活动的纹理单位。

示例程序9-11 为多重纹理指定顶点

```
glBegin(GL_TRIANGLES);
glMultiTexCoord2f(GL_TEXTURE0,0.0,0.0);
glMultiTexCoord2f(GL_TEXTURE1,1.0,0.0);
glVertex2f(0.0,0.0);
glMultiTexCoord2f(GL_TEXTURE0,0.5,1.0);
glMultiTexCoord2f(GL_TEXTURE1,0.5,0.0);
glVertex2f(50.0,100.0);
glMultiTexCoord2f(GL_TEXTURE0,1.0,0.0);
glMultiTexCoord2f(GL_TEXTURE1,1.0,1.0);
glVertex2f(100.0,0.0);
glEnd();
```

注意：如果在多重纹理下使用了glTexCoord*()，它所设置的就是第一个纹理单位的纹理坐标。换句话说，使用glTexCoord*()相当于使用glMultiTexCoord*(GL_TEXTURE0, ...)。

一种非常少见的情形是对位图或图像矩形进行多重纹理处理，此时需要把每个光栅位置与几个纹理坐标相关联。因此，必须多次调用glMultiTexCoord*()函数，分别用于每个纹理单位、每个glRasterPos*()和glWindowPos*()调用（由于整个位图或图像矩阵只有一个当前光栅位置，每个纹理单元只有一个对应的纹理坐标，因此在这种情况下应用多重纹理所增加的美感非常有限）。

指定纹理坐标的其他方法

在使用多重纹理时，显式调用glMultiTexCoord*()函数只是指定纹理坐标的3种方法之一。另两种方法是使用纹理坐标自动生成（使用glTexGen*()函数）和顶点数组（使用glTexCoordPointer()函数）。

如果在多重纹理下使用了纹理坐标自动生成功能，可以使用glActiveTexture()函数确定下面这些纹理坐标自动生成函数应用于哪个纹理单位：

- glTexGen*(...)
- glEnable(GL_TEXTURE_GEN_*)
- glDisable(GL_TEXTURE_GEN_*)

如果在多重纹理下使用顶点数组来指定纹理坐标，可以使用glClientActiveTexture()函数确定glTexCoordPointer()函数所指定的纹理坐标数据应用于哪个纹理单位。

void **glClientActiveTexture**(GLenum *texUnit*);

选择当前纹理单位，以便用顶点数组指定纹理坐标数据。*texUnit*是一个符号常量，其形式为GL_TEXTUREi，它与**glActiveTexture()**函数的对应参数所使用的值相同。

兼容性扩展
glClientActive Texture

恢复使用单个纹理单位

在使用多重纹理时，如果想恢复为使用单个纹理单位，需要禁用除纹理单位0之外的所有纹理单位，如示例程序9-12所示。

示例程序9-12 恢复为使用纹理单位0

```
/*disable texturing for other texture units */
glActiveTexture (GL_TEXTURE1);
glDisable (GL_TEXTURE_2D);
glActiveTexture (GL_TEXTURE2);
glDisable (GL_TEXTURE_2D);
/*make texture unit 0 current */
glActiveTexture (GL_TEXTURE0);
```

9.9 纹理组合器函数



高级话题

随着OpenGL的演变，它的重心逐渐从原先的顶点处理（变换、裁剪）过渡到光栅化和片断操作。程序员所接触到的纹理细节越来越多，片断的处理能力也越来越强。

除了多重纹理技术之外，OpenGL还提供了一些灵活的纹理组合器函数，允许程序员对片断与纹理值或其他颜色值的混合施加更精细的控制。纹理组合器函数支持高质量的纹理效果，例如凹凸贴图、更灵活的真实镜面光照以及纹理淡出效果（例如对2种纹理进行插值）等。纹理组合器函数可以从3种来源接收颜色和alpha数据，并对它们进行处理，生成RGBA颜色作为它的输出，用于后续的操作。

可以广泛使用glTexEnv*()函数配置纹理组合器函数。第9.5节已经对glTexEnv*()函数进行了简单的描述，下面是它的完整描述：

void **glTexEnv**{if}(GLenum *target*, GLenum *pname*, *TYPE param*);

void **glTexEnv**{if}**v**(GLenum *target*, GLenum *pname*, const *TYPE* **param*);

设置当前的纹理函数。target必须是GL_TEXTURE_FILTER_CONTROL或GL_ TEXTURE_ENV。

如果*target*是GL_TEXTURE_FILTER_CONTROL，*pname*必须是GL_TEXTURE_LOD_BIAS，*param*必须是一个单精度浮点值，表示mipmap细节层的偏移值。

如果*target*是GL_TEXTURE_ENV，*pname*和*param*参数可以接受的值见表9-8。如果*pname*是GL_TEXTURE_ENV_MODE，*param*指定了纹理值如何与被

兼容性扩展
glTexEnv and all accepted tokens

处理片断的颜色值进行组合。几种环境模式（GL_BLEND、GL_COMBINE、GL_COMBINE_RGB和GL_COMBINE_ ALPHA）决定了是否使用其他环境模式。

如果纹理环境模式为GL_BLEND，就使用GL_TEXTURE_ENV_COLOR设置。

如果纹理环境模式为GL_COMBINE，就可以使用GL_COMBINE_RGB、GL_COMBINE_ALPHA、GL_COMBINE_SCALE或GL_ALPHA_SCALE参数。对于GL_COMBINE_RGB函数，可以指定GL_SOURCEi_RGB和GL_OPERANDi_RGB参数（其中i是0、1或2）。GL_COMBINE_ALPHA函数也类似，可以指定GL_SOURCEi_ALPHA和GL_OPERANDi_ALPHA参数。

表9-8 target参数为GL_TEXTURE_ENV情况下的纹理环境参数

glTexEnv*()函数的pname参数	glTexEnv*()函数的param参数
GL_TEXTURE_ENV_MODE	GL_DECAL、GL_REPLACE、GL_MODULATE、GL_BLEND、GL_ADD或GL_COMBINE
GL_TEXTURE_ENV_COLOR	包含了4个浮点值（R、G、B、A）的数组
GL_COMBINE_RGB	GL_REPLACE、GL_MODULATE、GL_ADD、GL_ADD_SIGNED、GL_INTERPOLATE、GL_SUBTRACT、GL_DOT3_RGB或GL_DOT3_RGBA
GL_COMBINE_ALPHA	GL_REPLACE、GL_MODULATE、GL_ADD、GL_ADD_SIGNED、GL_INTERPOLATE或GL_SUBTRACT
GL_SRCi_RGB或GL_SRCi_ ALPHA（其中i是0、1或2）	GL_TEXTURE、GL_TEXTUREn（其中n表示启用了多重纹理情况下的第n个纹理单位）、GL_CONSTANT、GL_PRIMARY_COLOR或GL_PREVIOUS
GL_OPERANDi_RGB（其中i是0、1或2）	GL_SRC_COLOR、GL_ONE_MINUS_SRC_COLOR、GL_SRC_ALPHA或GL_ONE_MINUS_SRC_ALPHA
GL_OPERANDi_ALPHA（其中i是0、1或2）	GL_SRC_ALPHA或GL_ONE_MINUS_SRC_ALPHA
GL_RGB_SCALE	浮点型颜色缩放因子
GL_ALPHA_SCALE	浮点型alpha缩放因子

下面是使用纹理组合器函数的步骤。如果使用的是多重纹理，可以为每个纹理单位使用不同的纹理组合器函数，因此需要对每个纹理单位重复这些步骤。

- 为了使用纹理组合器函数，必须调用：

```
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE);
```

- 指定在纹理组合中怎样使用RGB或alpha值（见表9-9）。例如，示例程序9-13表示在当前纹理单位中把其中一种来源的RGB和alpha值减去另一种来源的RGB和alpha值。

表9-9 GL_COMBINE_RGB和GL_COMBINE_ALPHA函数

glTexEnv*()函数的param参数	组合器函数
GL_REPLACE	*Arg0*
GL_MODULATE（默认）	*Arg0* * *Arg1*
GL_ADD	*Arg0* + *Arg1*
GL_ADD_SIGNED	*Arg0* + *Arg1* − 0.5
GL_INTERPOLATE	*Arg0* * *Arg2* + *Arg1* * (1 − *Arg2*)
GL_SUBTRACT	*Arg0* − *Arg1*
GL_DOT3_RGB GL_DOT3_RGBA	$4 * ((Arg0_r - 0.5) * (Arg1_r - 0.5) + (Arg0_g - 0.5) * (Arg1_g - 0.5) + (Arg0_b - 0.5) * (Arg1_b - 0.5))$

示例程序9-13 设置可编程纹理组合器函数

```
glTexEnvf(GL_TEXTURE_ENV,GL_COMBINE_RGB,GL_SUBTRACT);
glTexEnvf(GL_TEXTURE_ENV,GL_COMBINE_ALPHA,GL_SUBTRACT);
```

注意：GL_DOT3_RGB和GL_DOT3_RGBA只用于GL_COMBINE_RGB，不能用于GL_COMBINE_ALPHA。

GL_DOT3_RGB和GL_DOT3_RGBA模式存在微小的区别。在GL_DOT3_RGB模式中，所有3个值（R、G、B）设置为同一个计算结果（点积）。而在GL_DOT3_RGBA模式下，所有4个值（R、G、B、A）设置为同一个计算结果。

- 使用GL_SOURCEi_RGB常量指定纹理组合器函数的第i个参数来自何处。参数的数量（最多为3个）取决于选择的函数类型。如表9-9所示，GL_SUBTRACT要求2个参数，可以用下面的代码进行设置：

示例程序9-14 设置纹理组合器函数的来源

```
glTexEnvf(GL_TEXTURE_ENV,GL_SRC0_RGB,GL_TEXTURE);
glTexEnvf(GL_TEXTURE_ENV,GL_SRC1_RGB,GL_PREVIOUS);
```

当pname是GL_SOURCEi_RGB，param参数就指定了纹理组合器函数的第i个参数的来源：

- GL_TEXTURE：第i个参数的来源是当前纹理单位的纹理
- GL_TEXTUREn：与纹理单位n相关联的纹理（如果使用这个来源，纹理单位n必须启用并且有效。否则，其结果就是未定义的）。
- GL_CONSTANT：GL_TEXTURE_ENV_COLOR设置的常量颜色。
- GL_PRIMARY_COLOR：片断主颜色用于纹理单位0，也就是进行纹理处理之前的片断颜色。
- GL_PREVIOUS：使用前一个纹理单位的片断（对于纹理单位0，和GL_PRIMARY_COLOR相同）。

假设在示例程序9-14中，为纹理单位2设置了GL_SUBTRACT组合器代码，那么计算方法就是把纹理单元2的输出（GL_TEXTURE, Arg0）减去纹理单位1的输出（GL_PREVIOUS, Arg1）。

- 指定需要使用哪些源值（RGB或alpha）以及如何使用它们：
- GL_OPERANDi_RGB与对应的GL_SOURCEi_RGB相匹配，并确定当前的GL_COMBINE_RGB函数的颜色值。如果pname为GL_OPERANDi_RGB，那么param必须是下列值之一：GL_SRC_COLOR、GL_ONE_MINUS_SRC_COLOR、GL_SRC_ALPHA或GL_ONE_MINUS_SRC_ALPHA。
- 类似地，GL_OPERANDi_ALPHA与对应的GL_SOURCEi_ALPHA相匹配，并确定当前的GL_COMBINE_ALPHA函数的颜色值。但是，param参数此时只能设置为GL_SRC_ALPHA或GL_ONE_MINUS_SRC_ALPHA。

当GL_SRC_ALPHA用于GL_COMBINE_RGB函数时，组合器来源的alpha值就解释为R、G、B。在示例程序9-15中，Arg2的R、G、B成分为（0.4, 0.4, 0.4）。

示例程序9-15 在RGB组合器操作中使用alpha值

```
static GLfloat constColor [4] ={0.1,0.2,0.3,0.4};
glTexEnvfv(GL_TEXTURE_ENV,GL_TEXTURE_ENV_COLOR,constColor);
glTexEnvf(GL_TEXTURE_ENV,GL_SRC2_RGB,GL_CONSTANT);
glTexEnvf(GL_TEXTURE_ENV,GL_OPERAND2_RGB,GL_SRC_ALPHA);
```

在示例程序9-15中，如果操作数是GL_SRC_COLOR，RGB成分将是（0.1, 0.2, 0.3）。对于GL_ONE_MINUS*模式，组合器的计算方式就是对值进行求补（1−color或1−alpha）。在示例程序

9-15中，如果操作数是GL_ONE_MINUS_SRC_COLOR，RGB成分就是（0.9, 0.8, 0.7）。对于GL_ONE_MINUS_SRC_ALPHA，计算结果是（0.6, 0.6, 0.6）。

- 选择RGB或alpha缩放因子（可选）。默认的方法是：

```
glTexEnvf(GL_TEXTURE_ENV, GL_RGB_SCALE, 1.0);
glTexEnvf(GL_TEXTURE_ENV, GL_ALPHA_SCALE, 1.0);
```

- 最后，绘制几何图形，并确保所有的顶点都拥有与之相关联的纹理坐标。

插值组合器函数

插值函数可以帮助我们演示纹理组合器函数的用法，因为它使用了最多数量的参数，并使用了几种来源和操作模式。示例程序9-16是combiner.c程序的一部分。

示例程序9-16 插值组合器函数：combiner.c

```
/*for use as constant texture color */
static GLfloat constColor [4] ={0.0,0.0,0.0,0.0};

constColor [3] =0.2;
glTexEnvfv(GL_TEXTURE_ENV,GL_TEXTURE_ENV_COLOR,constColor);
glBindTexture(GL_TEXTURE_2D,texName [0]);
glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_COMBINE);
glTexEnvf(GL_TEXTURE_ENV,GL_COMBINE_RGB,GL_INTERPOLATE);
glTexEnvf(GL_TEXTURE_ENV,GL_SRC0_RGB,GL_TEXTURE);
glTexEnvf(GL_TEXTURE_ENV,GL_OPERAND0_RGB,GL_SRC_COLOR);
glTexEnvf(GL_TEXTURE_ENV,GL_SRC1_RGB,GL_PREVIOUS);
glTexEnvf(GL_TEXTURE_ENV,GL_OPERAND1_RGB,GL_SRC_COLOR);
glTexEnvf(GL_TEXTURE_ENV,GL_SRC2_RGB,GL_CONSTANT);
glTexEnvf(GL_TEXTURE_ENV,GL_OPERAND2_RGB,GL_SRC_ALPHA);

/*geometry is now rendered */
```

在示例程序9-16中，只有一个活动纹理单位。由于纹理组合器函数设置为GL_INTERPOLATE，因此它使用了3个参数，并按照下面的公式进行组合：Arg0 * Arg1 + Arg1 * (1−Arg2)。这3个参数如下所示：

- Arg0：GL_TEXTURE，与当前绑定的纹理对象（texName[0]）相关联的纹理图像。
- Arg1：GL_PREVIOUS，前一个纹理单位的结果，但是由于这是纹理单位0，因此GL_PREVIOUS就是进行纹理处理之前的片断。
- Arg2：GL_CONSTANT，一个常量值，当前设置为（0.0, 0.0, 0.0, 0.2）。

我们获得的插值结果是纹理图像和未进行纹理处理时的片断的加权混合操作的结果。由于GL_OPERAND2_RGB被指定为GL_SRC_ALPHA，因此常量颜色（Arg2）的alpha值便作为加权值使用。

如果运行示例程序9-16，将会看到20%的纹理与80%的平滑着色多边形进行混合。conbiner.c程序还会修改常量颜色的alpha值，因此可以看到不同权重的结果。

可以观察一下插值函数，它解释了为什么

表9-10 一些纹理环境模式的默认值

glTexEnv*()函数的pname参数	param参数的初始值
GL_SRC0_RGB	GL_TEXTURE
GL_SRC1_RGB	GL_PREVIOUS
GL_SRC2_RGB	GL_CONSTANT
GL_ORERAND0_RGB	GL_SRC_COLOR
GL_ORERAND1_RGB	GL_SRC_COLOR
GL_ORERAND2_RGB	GL_SRC_ALPHA

要选择多个OpenGL默认值。插值函数的第3个参数可以作为另2个参数的权值使用。由于插值函数是唯一使用3个参数的纹理组合器函数，因此把GL_CONSTANT作为Arg2的默认值是安全的。初看之下，把GL_OPERAND2_RGB的默认设置为GL_SRC_ALPHA似乎很奇怪。但是，插值函数的加权值在3个成分中通常是一样的，因此使用单个值也是合理的，并且把alpha值作为这个常量值是比较方便的。

9.10 在纹理之后应用辅助颜色

把一个纹理应用于一个典型的片断时，只有主颜色与纹理单元的颜色进行组合。主颜色可能是光照计算的结果，也可能是由glColor*()函数所指定的。

在纹理处理之后，在执行雾计算之前，可以对片断应用一种辅助颜色。应用辅助颜色可以使经过纹理贴图的物体具有更真实的镜面亮点效果。

9.10.1 在禁用光照时使用辅助颜色

如果并没有启用光照，并且启用了颜色求和模式（使用glEnable(GL_COLOR_SUM)，那么当前的辅助颜色（由glSecondaryColor*()设置）便添加到经过纹理处理的片断颜色上。

void **glSecondaryColor3**{b s i f d ub us ui}(*TYPE r*, *TYPE g*, *TYPE b*);

void **glSecondaryColor3**{b s i f d ub us ui}v(const *TYPE *values*);

设置当前辅助颜色的红、绿、蓝成分。第一个后缀表示参数的数据类型：byte、short、int、float、double、unsigned byte、unsigned short或unsigned int。如果存在第二个后缀v，那么values就是一个指向特定类型的数组的指针。

兼容性扩展
glSecondaryColor
GL_COLOR_SUM

glSecondaryColor*()函数接受的参数和glColor*()函数相同，参数的解释也相同（参见表4-1）。辅助颜色也可以在顶点数组中指定。

9.10.2 启用光照后的辅助镜面颜色

纹理操作是在光照之后进行的，但是把镜面亮点和纹理的颜色进行混合通常会减弱光照的效果。如第5.5节所述，可以为每个顶点计算2种颜色：一种是主颜色，由所有的非镜面成分组成，另一种是辅助颜色，由所有镜面成分组成。如果把镜面颜色分离出来，辅助（镜面）颜色就可以在纹理计算之后再添加到片断中。

注意：如果启用了光照，就会应用辅助镜面颜色，而不管是否为GL_COLOR_SUM模式，并且glSecondaryColor*()设置的任何辅助颜色都会忽略。

9.11 点块纹理

虽然OpenGL支持对大小超过1个像素的大点（用glPointSize()设置）进行抗锯齿处理，但是它的视觉效果未必能够达到应用程序的要求。点块纹理允许对大点的着色施加更好的控制。在默认情况下，在启用了点块纹理后，纹理中的每个片断都分配一个相同的状态，与被渲染点的顶点的初始状态相同。点块纹理通过对这个大点的所有片断的纹理坐标进行迭代，修改片断数据的生成方式。

为了启用点块纹理，可以用GL_POINT_SPRITE为参数调用glEnable()函数。这将导致OpenGL忽略当前的点抗锯齿设置。大点中的每个片断将分配相关的顶点数据，这些值将会在着色中使用。

注意：OpenGL 3.1渲染所有的点，就好像支持点块纹理一样。glTexEnv()以及和点块纹理相关的所有参数都已经删除了。

为了根据点块纹理对纹理坐标进行迭代，需要调用glTexEnv*(GL_POINT_SPRITE, GL_COORD_REPLACE,GL_TURE)。在多重纹理中，需要为每个需要应用点块纹理的纹理单位都调用这个函数。抗锯齿点和纹理贴图点块纹理的区别如图9-15所示。左侧是一幅10个像素大小的经过抗锯齿处理的点的图像，右侧是一幅10个像素大小的纹理贴图点块纹理的图像。

点块纹理的纹理坐标是由OpenGL在光栅化过程中自动分配的。图9-16显示了如何为点块纹理分配纹理坐标。s纹理坐标从左向右沿点块纹理的片断从0增加到1。但是，t纹理坐标的值是由点块纹理指定的纹理坐标的原点决定的。而点块纹理坐标原点又是通过调用glPointParameter()函数指定的（第一个参数为GL_POINT_SPRITE_COORD_ORIGIN，第二个参数的值或者设置为GL_LOWER_LEFT，或者设置为GL_UPPER_LEFT）。



图9-15 抗锯齿点和点块纹理的区别

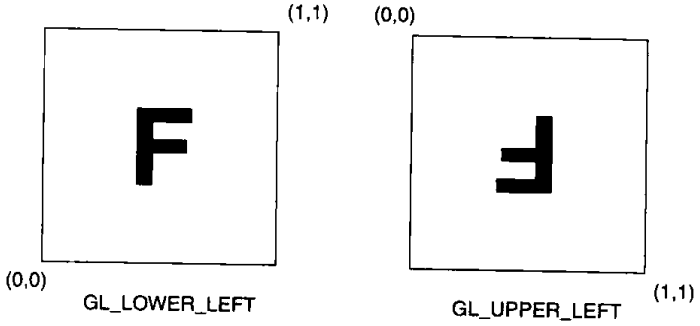


图9-16 根据GL_POINT_SPRITE_COORD_ORIGIN的设置分配纹理坐标

当点块纹理的原点指定为GL_LOWER_LEFT时，t坐标从点块纹理的底部向上自0增加到1。反之，如果原点指定为GL_UPPER_LEFT时，t坐标自顶向下从0增加到1。

示例程序9-17显示了一个使用了纹理贴图的10个像素大小的点块纹理。

示例程序9-17 配置用于纹理贴图的点块纹理：sprite.c

```
glPointSize(10.0);
glTexEnvi(GL_POINT_SPRITE,GL_COORD_REPLACE,GL_TRUE);
glPointParameteri(GL_POINT_SPRITE_COORD_ORIGIN,GL_LOWER_LEFT);
glEnable(GL_POINT_SPRITE);
```

9.12 纹理矩阵堆栈



高级话题

就像物体的坐标在进行渲染之前需要根据矩阵进行变换一样，纹理坐标在用于纹理贴图之前也需要乘以一个4×4的矩阵。在默认情况下，纹理矩阵是单位矩阵，因此显式指定或自动生成的纹理坐标并不会发生变化。但是，在重绘物体时通过修改纹理矩阵，可以实现使纹理沿表面滑动、绕表面旋转、收缩或放大的效果，或者是上述3种效果的组合。事实上，由于纹理矩阵完全是一个普通的4×4矩阵，因此可以实现诸如透视这样的效果。

纹理矩阵实际上是一个位于堆栈顶部的矩阵，这个堆栈的深度至少能够容纳2个矩阵。所有像glPushMatrix()、glPopMatrix()、glMultMatrix()和glRotate*()这样的矩阵操纵函数都可以在纹理矩阵上使用。为了更改当前的纹理矩阵，需要把矩阵模式设置为GL_TEXTURE，如下所示：

```
glMatrixMode(GL_TEXTURE); /* enter texture matrix mode */
glRotated(...);
/* ... other matrix manipulations ... */
glMatrixMode(GL_MODELVIEW); /* back to modelview mode */
```

兼容性扩展
GL_TEXTURE

q坐标

第4个纹理坐标q的数学意义相当于物体坐标（x, y, z, w）的w坐标。当4个纹理坐标（s, t, r, q）与纹理矩阵相乘时，它所产生的向量（s', t', r', q'）解释为齐次纹理坐标。换句话说，纹理图像是通过坐标值s'/q'、t'/q' 和r'/q' 进行索引的。

如果需要多个投影或透视变换，可能就要用到q坐标。例如，如果想对强度不匀（例如中间更亮，或者由于灯罩或镜头的影响使光束为非圆锥形）的聚光灯进行建模。为了模拟光线照射到平面上的情况，可以创建一个与光线形状和强度相对应的纹理，然后使用投影变换把纹理投影到平面上。当我们在场景中把光锥投影到平面上时，需要使用透视变换（$q \neq 1$），因为光线可能并不垂直于被照射的表面。当观察者从一个不同的观察点观察场景（使用透视方式）时，就可能出现第二个透视变换。（例子参见彩图28。关于这个主题的详细内容，可以参阅Mark Segal、Carl Korobkin、Rolf van Widenfelt、Jim Foran和Paul Haeberli所著的《Fast Shadows and Lighting Effects Using Texture Mapping》，SIGGRAPH 1992 Proceedings[Computer Graphics, 26:2, July 1992，第249-252页]）

使用q坐标的另一个例子是当纹理图像是以透视方式拍摄的照片时。和聚光灯一样，最终的视图取决于2个投影变换的组合。

Nate Robin的纹理教程

在Nate Robin的“texture教程”中，可以使用弹出菜单来观察4×4的纹理堆栈，可以对矩阵的值进行修改，然后再观察它们的效果。

9.13 深度纹理



高级话题

当一个表面接受光照之后（参见第5章），我们会注意到OpenGL光源并没有在表面上投射出阴影。每个顶点的颜色在计算时并没有考虑场景中的其他物体。为了产生阴影，需要确定并记录哪些表面（或表面的部分）和光源的直线路径上具有遮挡物。

一种使用深度纹理进行多道渲染的技巧可以提供一种渲染阴影的解决方案。如果把观察点临时移动到光源位置，我们就会注意到自己所看到的一切都被光所照射，也就是从这个角度观察不存在阴影。深度纹理提供了一种机制，把所有“无阴影”的片断的深度值保存在一个阴影图（shadow map）中。当我们对场景进行渲染时，如果把每个源片断与阴影图中的对应深度值进行比较，就可以根据每个片断是否罩有阴影来选择如何对它进行渲染。这个思路类似于深度测试，只不过它是把光源作为观察点来进行操作的。

下面简单描述这种操作：

1) 从光源的角度对场景进行渲染。场景看上去是什么样子无关紧要，只需要深度值。然后创建阴影图，也就是捕捉深度缓冲区的值，并把它们存储在一个纹理图像（即阴影图）中。

2) 生成纹理坐标，其中（s, t）坐标引用阴影图中的位置，第三个纹理坐标r则表示与光源的距离。然后再次绘制场景，把每个片断的r值与对应的深度纹理值进行比较，以确定这个片断是被光所照射还是被阴影所笼罩。

下面各节提供了更详细的讨论，包括用于说明各个步骤的示例代码。

9.13.1 创建阴影图

第一个步骤是创建包含深度值的阴影图。可以把观察点设置为光源的位置，然后对场景进行渲染，从而创建这个阴影图。示例程序9-18调用了glGetLightfv()函数获取当前光源位置，并计算一个朝上向量，然后用它进行视图变换。

示例程序9-18首先设置视口的大小，使之与阴影图的大小相匹配。然后，它设置适当的投影和视图矩阵。场景中的物体被渲染，所产生的深度图像被复制到纹理内存中，作为阴影图使用。最后，视口被重置为原先的大小和位置。

注意如下一些要点：

- 投影矩阵控制光源“灯罩”的形状。gluPerspective()函数中的变量lightfovy和lightAspect用于控制这个灯罩的大小。lightFovy值越小，光源就越像聚光灯。lightFovy越大，光源就越像泛光灯。
- 光源的近侧和远侧裁剪平面（lightNearPlane和lightFarPlane）用于控制深度值的精度。应该尽量使近侧和远侧裁剪平面之间的距离保持很小，以提高深度值的精度。
- 在深度缓冲区中保存了深度值之后，需要捕捉它们，并把它们放在一个GL_DEPTH_COMPONENT格式的纹理图像中。示例程序9-18使用glCopyTexImage2D()函数根据深度缓冲区的内容创建一幅纹理图像。和其他所有纹理图像一样，需要保证这幅纹理图像的宽度和高度是2的整数次方。

示例程序9-18 以光源为观察点渲染场景：shadowmap.c

```
GLint viewport [4];
GLfloat lightPos [4];

glGetLightfv(GL_LIGHT0,GL_POSITION,lightPos);
glGetIntegerv(GL_VIEWPORT,viewport);

glViewport(0,0,SHADOW_MAP_WIDTH,SHADOW_MAP_HEIGHT);

glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);

glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
gluPerspective(lightFovy,lightAspect,lightNearPlane,
                lightFarPlane);
glMatrixMode(GL_MODELVIEW);

glPushMatrix();
glLoadIdentity();
gluLookAt(lightPos [0],lightPos [1],lightPos [2],
    lookat [0],lookat [1],lookat [2],
    up [0],up [1],up [2]);
drawObjects();
glPopMatrix();

glMatrixMode(GL_PROJECTION);
glPopMatrix();
```

```
glMatrixMode(GL_MODELVIEW);

glCopyTexImage2D(GL_TEXTURE_2D,0,GL_DEPTH_COMPONENT,0,0,
    SHADOW_MAP_WIDTH,SHADOW_MAP_HEIGHT,0);

glViewport(viewport [0],viewport [1],
    viewport [2],viewport [3]);
```

9.13.2 生成纹理坐标并进行渲染

现在，可以使用glTexGen*()函数自动生成纹理坐标，计算与光源的视觉空间距离。r坐标的值对应于图元与光源的距离。为此，可以使用和创建与阴影图相同的投影和视图变换。示例程序9-19使用GL_MODELVIEW矩阵堆栈执行所有的矩阵计算。

注意，生成的（s, t, r, q）纹理坐标和阴影图中的深度值并没有进行类似的缩放。纹理坐标是在视觉坐标下产生的，因此它们的范围位于[−1，1]。纹理单元的深度值的范围位于[0，1]。因此，应该进行移动和缩放，把纹理坐标的值映射到和阴影图相同的范围。

示例程序9-19 计算纹理坐标：shadowmap.c

```
GLfloat tmpMatrix [16];

glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glLoadIdentity();
glTranslatef(0.5,0.5,0.0);
glScalef(0.5,0.5,1.0);
gluPerspective(lightFovy,lightAspect,
    lightNearPlane,lightFarPlane);
gluLookAt(lightPos [0],lightPos [1],lightPos [2],
     lookat [0],lookat [1],lookat [2],
     up [0] ,up [1],up [2]);
glGetFloatv(GL_MODELVIEW_MATRIX,tmpMatrix);
glPopMatrix();

transposeMatrix(tmpMatrix);

glTexGeni(GL_S,GL_TEXTURE_GEN_MODE,GL_OBJECT_LINEAR);
glTexGeni(GL_T,GL_TEXTURE_GEN_MODE,GL_OBJECT_LINEAR);
glTexGeni(GL_R,GL_TEXTURE_GEN_MODE,GL_OBJECT_LINEAR);
glTexGeni(GL_Q,GL_TEXTURE_GEN_MODE,GL_OBJECT_LINEAR);

glTexGenfv(GL_S,GL_OBJECT_PLANE,&tmpMatrix [0]);
glTexGenfv(GL_T,GL_OBJECT_PLANE,&tmpMatrix [4]);
glTexGenfv(GL_R,GL_OBJECT_PLANE,&tmpMatrix [8]);
glTexGenfv(GL_Q,GL_OBJECT_PLANE,&tmpMatrix [12]);

glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
glEnable(GL_TEXTURE_GEN_R);
glEnable(GL_TEXTURE_GEN_Q);
```

在示例程序9-19中，在场景的第二次和最后一次渲染之前，纹理比较模式GL_ COMPARE_R_

TO_TEXTURE指示OpenGL把片断的r坐标与纹理单元值进行比较。如果r的距离小于或等于（比较函数为GL_LEQUAL）纹理单元值，这个片断和光源之间就没有其他物体，它就是被光源直接照射的，其有效亮度值为1。如果比较结果失败，那么这个片断和光源之间还存在其他图元，因此这个片断就被阴影所笼罩，它的有效亮度值为0。参见示例程序9-20。

示例程序9-20 渲染场景，比较r坐标：shadowmap.c

```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,
    GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,
    GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
    GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,
    GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_COMPARE_FUNC,
    GL_LEQUAL);
glTexParameteri(GL_TEXTURE_2D,GL_DEPTH_TEXTURE_MODE,
    GL_LUMINANCE);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_COMPARE_MODE,
    GL_COMPARE_R_TO_TEXTURE);
glEnable(GL_TEXTURE_2D);
```

这种技巧也可能产生意想不到的人工视觉效果：

- 自身阴影，也就是一个物体不正确地把阴影投影到自身，这是一个常见的问题。
- 可能会出现投影纹理的锯齿效果，特别是那些离光源非常远的区域。使用高分辨率的阴影图可以帮助减少这种现象。
- GL_MODULATE模式应用于深度纹理时会导致阴影和非阴影区域之间的急剧过渡。

遗憾的是，这些问题并没有通用而有效的解决方法。可以多进行试验，以产生具有最佳视觉效果的图像。

第10章 帧缓冲区

本章目标

- 了解帧缓冲区是由哪些缓冲区组成的，以及如何使用它们。
- 清除选中的缓冲区，并启用它们，以便写入。
- 控制用于像素的裁剪测试、alpha测试、模板测试和深度测试的参数。
- 使用遮挡查询判断物体是否可见。
- 执行抖动和逻辑操作。
- 使用累积缓冲区实现如场景抗锯齿这样的效果。
- 创建和使用帧缓冲区对象，以使用高级技术，并且尽可能少地在缓冲区之间复制数据。

注意： *在OpenGL 3.1中，本章所介绍的一些技术和函数已经废弃并删除了，但是概念仍然是相关的，并且有更加现代的功能可供使用。*

几乎每个图形程序的重要目标之一都是在屏幕上绘制图形。屏幕是由一个矩形的像素数组组成的，每个像素都可以在图像的某一个点上显示一个某种颜色的微小方块。在光栅化阶段（包括纹理和雾）之后，数据就不再是像素，而是成为片断。每个片断都具有与像素对应的坐标数据以及颜色值和深度值。然后，每个片断都经历一系列的测试和操作，其中有些操作已经在前面描述（参见第6.1节），还有一些操作将在本章中讨论。

如果顺利通过了这些测试和操作，片断值便可以转换为像素。为了绘制这些像素，我们需要知道它们的颜色（即存储在颜色缓冲区中的信息）。当每个像素的数据按照统一的方式存储时，存储所有像素的存储空间就叫做缓冲区。不同的缓冲区为每个像素存储的数据量可能不同。但是，在一个特定的缓冲区内，为每个像素存储的数据量是相同的。为每个像素存储了1位信息的缓冲区又称为位平面（bitplane）。

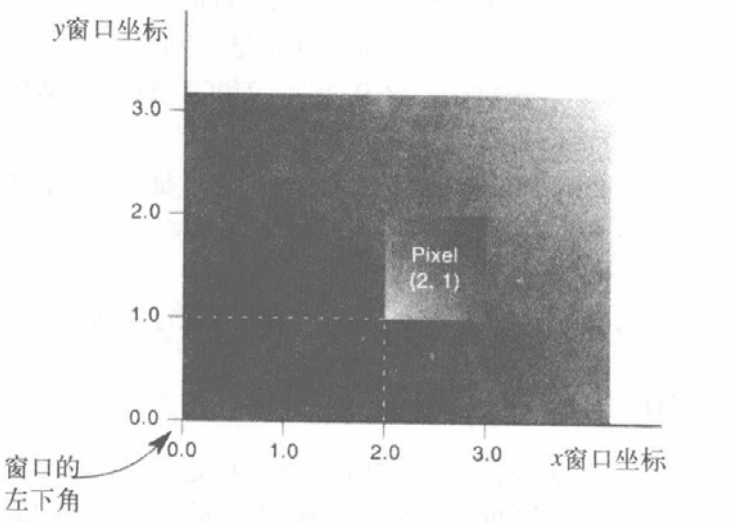


图10-1 一个像素所占据的区域

如图10-1所示，OpenGL窗口的左下角像素为像素（0，0），对应于窗口左下角像素所占据的一个1×1区域。一般而言，像素(x, y)填充的区域的左边界为x，右边界为$x+1$，下边界为y，上边界为$y+1$。

颜色缓冲区就是缓冲区的一个例子，它用于保存屏幕上所显示的颜色信息。假设屏幕的宽度为1280个像素，高度为1024个像素，并使用24位的完整颜色。换句话说，这个屏幕可以显示2^{24}（16 777 216）种不同的颜色。由于24位相当于3个字节（每个字节8位），因此这个颜色缓冲区必须为屏幕上的1 310 720（1280×1024）个像素的每一个都存储3个字节的数据。在特定的硬件系统中，物

理屏幕所显示的像素可能更多或更少。另外，每个像素的数据量也可能更多或更少。但是，对于一个特定的颜色缓冲区而言，它为屏幕上每个像素所存储的数据量是相同的。

颜色缓冲区只是用于存储像素信息的许多缓冲区之一，还有很多其他缓冲区。例如，深度缓冲区存储了每个像素的深度信息。颜色缓冲区本身也可能包括几个子缓冲区。系统的帧缓冲区是由所有这些缓冲区组成的。除了颜色缓冲区之外，其他缓冲区都无法直接查看，它们用于实现诸如隐藏表面消除、场景抗锯齿、模板测试、绘制平滑的运动图像等功能。

本章描述OpenGL实现可能拥有的所有缓冲区以及它们的用法。本章还将讨论数据在写入到可查看的颜色缓冲区之前所执行的一系列测试和像素操作。最后，本章还解释了如何使用累积缓冲区，它用于保存要绘制到颜色缓冲区中的图像。最初，累积缓冲区在存储高精度颜色值方面能力稍强。OpenGL 3.0引入了支持本地浮点值的颜色缓冲区，尽管这不会使累积缓冲区技术变得没必要，但是它却显得有些过时了。这些事情很容易在浮点缓冲区上实现。

本章主要由下面几节组成：

- 缓冲区及其用途：介绍各种缓冲区及其用途，包括如何清除缓冲区以及如何允许在缓冲区中写入信息。
- 片断测试和操作：描述像素的位置和颜色在计算产生之后却还没有绘制到屏幕之前所进行的裁剪测试、alpha测试、模板测试和深度测试。在片断更新到屏幕之前，还可能会进行一些操作，例如混合、抖动和逻辑操作。
- 累积缓冲区：描述如何使用累积缓冲区执行一些高级的操作。这些技巧包括对整个场景进行抗锯齿处理、使用运动模糊以及模拟照片景深的效果。
- 帧缓冲区对象：介绍OpenGL 3.0中的一种新的对象类型，它能够在缓冲区而不是显示在屏幕上的颜色缓冲区中渲染。它支持离屏渲染的功能以及使用渲染到纹理来更新纹理图像的功能。

OpenGL 1.4版本增加了下面这个功能：

- 模板操作GL_INCR_WRAP和GL_DECR_WRAP。

OpenGL 3.0版本增加了如下功能：

- 帧缓冲区对象用于离屏渲染和渲染到纹理的功能。
- 用于渲染的sRGB格式的帧缓冲区。

OpenGL 3.1版本修改了这一功能：

- 取消了对累积缓冲区和辅助缓冲区的支持。

10.1 缓冲区及其用途

OpenGL系统可以对如下缓冲区进行操作：

- 颜色缓冲区：前-左、前-右、后-左、后-右以及任何数量的辅助颜色缓冲区。
- 深度缓冲区。
- 模板缓冲区。
- 累积缓冲区。

具体使用的OpenGL实现决定了可以使用哪些缓冲区，并决定了每个缓冲区为每个像素保留多少位。另外，应用程序可能具有多种画面（或窗口类型），它们分别具有不同的缓冲区。表10-1列出了调用glGetIntegerv()函数时所使用的参数，它们可以查询特定的画面可以使用的基于像素的缓冲区存储信息。

表10-1 基于像素的缓冲区存储的查询参数

参 数	含 义
GL_RED_BITS、GL_GREEN_BITS、GL_BLUE_BITS、GL_ALPHA_BITS	颜色缓冲区中每个R、G、B或A成分的位数
GL_INDEX_BITS	颜色缓冲区中每个颜色索引值的位数
GL_DEPTH_BITS	深度缓冲区中每个像素的位数
GL_STENCIL_BITS	模板缓冲区中每个像素的位数
GL_ACCUM_RED_BITS、GL_ACCUM_GREEN_BITS、GL_ACCUM_BLUE_BITS、GL_ACCUM_ALPHA_BITS	累积缓冲区中每个R、G、B或A成分的位数

注意：如果使用的是X窗口系统，至少要保证可以在RGBA模式下使用1个颜色缓冲区以及各个非零颜色成分相关的模板缓冲区、深度缓冲区和累积缓冲区。另外，如果使用的X窗口系统实现支持伪彩画面，至少要保证可以在颜色索引模式下使用1个颜色缓冲区，以及相关的深度和模板缓冲区。可以使用glXGetConfig()函数查询自己所使用的画面。有关这个函数的更多信息，请参阅附录C和《OpenGL Reference Manual》。

10.1.1 颜色缓冲区

颜色缓冲区通常用于绘图的缓冲区。它们包含了颜色索引或RGB颜色数据，还可能包含alpha数据。支持立体画面的OpenGL实现提供了左、右颜色缓冲区，分别包含了左、右立体图像。如果OpenGL实现不支持立体图像，那就只使用左缓冲区。类似地，双缓冲系统提供了前、后缓冲区，而单缓冲系统只提供了前缓冲区。每个OpenGL实现都必须提供一个前-左缓冲区。

另外，OpenGL实现还可能支持不可显示的辅助颜色缓冲区。OpenGL并没有指定这类缓冲区的特定用途，因此可以按照自己的想法定义和使用它们。例如，可以使用它们保存一幅想要重复使用的图像。这样，我们就不必对这幅图像进行重绘，只要把它从辅助缓冲区复制到普通的颜色缓冲区就可以了。有关这方面的细节，请参阅第8.2节中对glCopyPixels()函数的描述。

注意：在OpenGL 3.0中，颜色索引模式渲染和辅助缓冲区废弃了；这些功能完全从OpenGL 3.1中删除了。如果你的OpenGL实现支持这一扩展的话，它们可以作为GL_ARB_compatibility扩展的一部分使用。

可以在glGetBooleanv()函数中使用GL_STEREO或GL_DOUBLEBUFFER参数来判断自己使用的系统是否支持立体画面（即提供了左、右缓冲区）或双缓冲（即提供了前、后缓冲区）。为了判断系统支持多少个辅助缓冲区（如果有的话），可以用GL_AUX_BUFFERS为参数调用glGetIntegerv()函数。

深度缓冲区

深度缓冲区存储每个像素的深度值。如第5.1节所述，深度通常是根据物体和观察点的距离来测量的，因此具有较大深度值的像素有可能会被具有较小深度值的像素覆盖。但是，这只是一种通常的约定，我们完全可以使用第10.2.4节描述的方法更改深度缓冲区的行为。深度缓冲区有时又称为z缓冲区（x表示屏幕的水平方向，y表示屏幕的垂直方向，z表示从观察点垂直于屏幕的方向）。

模板缓冲区

模板缓冲区的用途之一就是把绘图限制在屏幕中的某个区域，就像使用纸板和喷漆实现精确的绘图一样。例如，如果想绘制一幅在形状怪异的挡风玻璃上出现的图像，可以在模板缓冲区内存储一幅

挡风玻璃形状的图像，然后再绘制整个场景。模板缓冲区可以防止那些透过挡风玻璃无法看到的物体被绘制出来。因此，如果应用程序是一个模拟驾驶的程序，可以只绘制车内的仪表和其他物体1次。当汽车开动时，只有车窗外的场景需要更新。

累积缓冲区

与RGBA模式下的颜色缓冲区一样，累积缓冲区也用于存储RGBA颜色数据（在颜色索引模式下，使用累积缓冲区的结果是未定义的）。累积缓冲区通常用于把一系列的图像合成为一幅图像。通过这种方法，可以对图像进行超量采样（supersampling），然后对样本求平均值，并且将结果写入到颜色缓冲区中，从而实现场景的抗锯齿处理。不能把数据直接写入到累积缓冲区，累积操作总是以矩形块为单位对数据进行操作，通常是把数据移入或移出颜色缓冲区。

10.1.2 清除缓冲区

在图形程序中，清除屏幕（以及所有的缓冲区）是开销最大的操作之一。一个1280×1024的屏幕包含了上百万的像素。对于简单的图形应用程序，清除操作花费的时间可能比接下来进行的绘图操作的时间还要多。如果需要清除的不仅仅是颜色缓冲区，还需要清除深度和模板缓冲区，清除操作所花费的时间可能还要成倍地增加。

为了解决这个问题，有些计算机提供了能够同时清除多个缓冲区的硬件。OpenGL的清除命令能够充分利用这类硬件的优势。首先，指定需要写入到每个将要清除的缓冲区的值，然后，发布一条命令执行清除操作。在这个命令中，需要传递将要清除的缓冲区列表。如果硬件支持同时清除多个缓冲区，这些清除操作就会同时进行。否则，这些缓冲区就会按顺序逐个清除。

下面这些函数可以用于设置每个缓冲区的清除值：

兼容性扩展
glClearIndex
glClearAccum

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

void **glClearIndex**(GLfloat *index*);

void **glClearDepth**(GLclampd *depth*);

void **glClearStencil**(GLint *s*);

void **glClearAccum**(GLfloat *red*, GLfloat *green*, GLfloat *blue*, GLfloat *alpha*);

指定RGBA模式下的颜色缓冲区、颜色索引模式下的颜色缓冲区、深度缓冲区、模板缓冲区和累积缓冲区的当前清除值。GLclampf和GLclampd类型（经过截取的GLfloat和GLdouble）都限定在0.0~1.0的范围之内。默认的深度缓冲区清除值是1.0，其他所有缓冲区的默认清除值都是0。用清除命令设置的值一直保持有效，直到再次用同一个命令对它们进行修改。

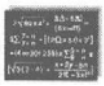
选择了清除值，并准备清除缓冲区时，可以使用glClear()函数：

兼容性扩展
GL_ACCUM_
BUFFER_BIT

void **glClear**(GLbitfield *mask*);

清除指定的缓冲区。mask的值是GL_COLOR_BUFFER_BIT、GL_DEPTH_BUFFER_BIT、GL_STENCIL_BUFFER_BIT和GL_ACCUM_BUFFER_BIT的一些逻辑OR操作的组合结果，标识了哪些缓冲区需要被清除。GL_COLOR_BUFFER_BIT用于清除RGBA颜色缓冲区或颜色

索引缓冲区（取决于系统所使用的是哪种颜色模式）。在清除RGBA颜色缓冲区或颜色索引缓冲区时，所有启用的可写入的颜色缓冲区（参见下一节）都被清除。如果启用了像素所有权测试、裁剪测试和抖动操作，它们都会在清除操作中执行。屏蔽操作（例如**glColorMask()**和**glIndexMask()**）也会生效。alpha测试、模板测试和深度测试并不会影响**glClear()**函数的操作。



高级话题

如果使用多个绘制缓冲区（尤其是具有浮点或非规范化的整数像素格式的缓冲区），可以使用OpenGL 3.0中引入的glClearBuffer*()函数来清除每个单个的绑定缓冲区。与glClearColor()和glClearDepth()函数（它们在OpenGL中设置一个清除值，当调用glClear()的时候使用这个值）不同，glClearBuffer*()使用传递给它的值直接清除绑定的绘制缓冲区。

void **glClearBuffer**{fi ui}**v**(GLenum *buffer*, GLint *drawbuffer*,
const *TYPE* **value*);

将由*drawbuffer*索引的与*buffer*相关联的缓冲区清除为*value*。*buffer*必须是GL_COLOR、GL_DEPTH或GL_STENCIL之一。

如果*buffer*是GL_COLOR，*drawbuffer*为一个特定绘制缓冲区指定了索引，并且value是包含清除颜色的一个四元素数组。如果*drawbuffer*索引的缓冲区有多个绘制缓冲区（如调用**glDrawBuffers()**的特定情况），所有的绘制缓冲区都清除为*value*。

如果*buffer*是GL_DEPTH或GL_STENCIL，*drawbuffer*必须是0，而*value*是包含一个恰当的清除值（根据深度值的截取和类型转换，以及模板值的屏蔽和类型转换）的单元素数组。

如果*buffer*不是上面列出的可接受值中的一个，将会产生一个GL_INVALID_ENUM错误。如果*buffer*是GL_COLOR，并且*drawbuffer*小于0或者大于或等于GL_MAX_DRAW_BUFFERS；亦或如果*buffer*是GL_DEPTH或GL_STENCIL，并且*drawbuffer*不为0，将会产生GL_INVALID_VALUE。最后，如果buffer是GL_COLOR，并且应用于一个颜色索引模式的缓冲区，将会设置GL_INVALID_ OPERATION。

为了减少与使用多绘制缓冲区相关的函数调用的次数，可以调用glClearBufferfi()同时清除深度和模板缓冲区（这等同于调用glClearBuffer*()两次，一次针对深度缓冲区调用，一次针对模板缓冲区调用，但是前者的效率更高）：

void **glClearBufferfi**(GLenum *buffer*, GLint *drawbuffer*,
GLfloat *depth*, GLint *stencil*);

同时清除当前绑定的帧缓冲区的深度和模板缓冲区（这在某些实现上可能会更快）。*buffer*必须是GL_DEPTH_STENCIL，并且*drawbuffer*必须是0。*depth*和*stencil*将分别用做各自缓冲区的清除值。

如果*buffer*不是GL_DEPTH_STENCIL，将会产生GL_INVALID_ENUM；并且，如果*drawbuffer*不等于0，将会产生GL_INVALID_VALUE。

10.1.3 选择用于读取和写入的颜色缓冲区

绘图和读取操作的目标都可以是任何颜色缓冲区：前、后、前左、后左、前右、后右，以及任何辅助缓冲区。可以选择一个单独的缓冲区作为绘图或读取操作的目标。对于绘图操作，还可以通过设置把绘图结果同时写入到多个缓冲区。可以使用glDrawBuffer()函数选择需要写入的缓冲区，也可以使用glReadBuffer()函数选择用于读取的缓冲区，作为glReadPixels()、glCopyPixels()、

glCopyTexImage*()和glCopyTexSubImage*()的数据来源。

如果使用的是双缓冲系统，通常只想在后缓冲区中进行绘图（并在完成绘图后交换前后缓冲区）。在有些情况下，我们可能想把双缓冲区系统当作单缓冲系统一样使用。可以调用glDrawBuffer()函数实现这个目的，它允许同时在前后2个缓冲区中进行绘图。glDrawBuffer()函数还可以选择渲染立体图像的缓冲区（GL*LEFT和GL*RIGHT），还可以选择渲染到辅助缓冲区（GL_AUX*i*）。

兼容性扩展
GL_AUX*i*

void **glDrawBuffer**(GLenum *mode*);

void **glDrawBuffers**(GLsizei *n*, const GLenum **buffers*);

选择用于写入或清除的颜色缓冲区，并禁用以前的glDrawBuffer()调用所启用的缓冲区。这个函数可以同时启用超过1个的缓冲区。mode的值可以是下列之一：

GL_FRONT	GL_FRONT_LEFT	GL_AUX*i*
GL_BACK	GL_FRONT_RIGHT	GL_FRONT_AND_BACK
GL_LEFT	GL_BACK_LEFT	GL_NONE
GL_RIGHT	GL_BACK_RIGHT	GL_COLOR_ATTACHMENT*i*

其中，省略了LEFT或RIGHT的参数表示同时包括左右立体缓冲区。类似地，省略了FRONT和BACK的参数表示同时包括前后缓冲区。GL_AUX*i*中的*i*用于标识某个特定的辅助缓冲区。

在默认情况下，mode在单缓冲环境下为GL_FRONT，在双缓冲环境下为GL_BACK。

OpenGL 2.0增加了**glDrawBuffers()**函数，它指定了用于接收颜色值的多个颜色缓冲区。*buffer*是缓冲区枚举值的数组，这个数组只允许包含GL_NONE、GL_FRONT_LEFT、GL_FRONT_RIGHT、GL_BACK_LEFT、GL_BACK_RIGHT和GL_AUX*i*。

OpenGL 3.0添加了帧缓冲区对象，当OpenGL绑定到一个用户定义的帧缓冲对象的时候，增加了GL_COLOR_ATTACHMENTi来指定哪个颜色渲染缓冲区是绘制目标。

如果利用固定功能的管线来生成片断的颜色，每个指定的缓冲区会接收相同的颜色值。如果使用了片断着色器并指定输出到多个缓冲区，那么每个缓冲区将写入这个着色器的输出所指定的颜色值。相关细节，请参阅第15.11节。

注意：当把绘图设置为写入到多个缓冲区时，只要系统中至少存在一个缓冲区，就不会出现错误。如果所有的缓冲区均不存在，就会发生错误。

兼容性扩展
GL_AUX*i*

void **glReadBuffer**(GLenum *mode*);

选择颜色缓冲区，作为以后调用**glReadPixels()**、**glCopyPixels()**、**glCopyTexImage***()、**glCopyTexSubImage***()和**glCopyConvolutionFilter***()的像素读取来源，并禁用以前调用glReadBuffer()时所启用的缓冲区。mode的值可以是下列之一：

GL_FRONT	GL_FRONT_LEFT	GL_AUX*i*
GL_BACK	GL_FRONT_RIGHT	GL_COLOR_ATTACHMENTi
GL_LEFT	GL_BACK_LEFT	
GL_RIGHT	GL_BACK_RIGHT	

glReadBuffer()函数设置的缓冲区与**glDrawBuffer()**函数设置的缓冲区相同。在默认情况下，mode在单缓冲环境下为GL_FRONT，在双缓冲环境下为GL_BACK。

当从一个用户定义的帧缓冲区对象读取数据时，使用OpenGL 3.0的GL_COLOR_ATTACHMENT*i*。*i*的值必须在0和GL_MAX_COLOR_ ATTACHMENTS（作为调用**glGetIntegerv()**的返回值）之间。

注意：必须从确实存在的缓冲区中进行读取，否则就会出错。

10.1.4 缓冲区的屏蔽

在OpenGL把数据写入到已启用的颜色、深度或模板缓冲区之前，它首先会对数据执行屏蔽操作（通过下面所描述的函数之一）。所有的掩码都使用逻辑AND操作进行组合，以写入对应的数据。

兼容性扩展
glIndexMask

void **glIndexMask**(GLuint *mask*);

void **glColorMask**(GLboolean *red*, GLboolean *green*, GLboolean *blue*, GLboolean *alpha*);

void **glColorMaski**(GLuint *buf*, GLboolean *red*, GLboolean *green*, GLboolean *blue*, GLboolean *alpha*);

void **glDepthMask**(GLboolean *flag*);

void **glStencilMask**(GLuint *mask*);

void **glStencilMaskSeparate**(GLenum *face*, GLuint *mask*);

设置掩码，把写入控制在指定的缓冲区。**glIndexMask()**设置的掩码只在颜色索引模式下起作用。如果mask中出现了1，颜色索引缓冲区中对应的位就被写入。如果mask中出现了0，颜色索引缓冲区中对应的位就不会被写入。类似地，glColorMask()只影响RGBA模式下的绘图。red、green、blue和alpha值用于控制对应的成分是否写入（GL_TRUE表示写入）。

如果**glDepthMask()**的*flag*参数设置为GL_TRUE，深度缓冲区可以写入，否则它被禁用。**glStencilMask()**的掩码用于设置模板数据，其方式与**glIndexMask()**设置颜色索引数据的方法相同。所有布尔型掩码的默认值均为GL_TRUE，两个GLuint掩码的默认值是全1。

OpenGL 2.0 增加了**glStencilMaskSeparate()**函数，允许对正面和背面多边形独立地使用掩码值。

OpenGL 3.0增加了**glColorMaski()**函数，允许在渲染到多个颜色缓冲区时为buf指定的单个缓冲区设置颜色掩码（参见第10.1.3节了解关于使用多个缓冲区的详细内容）。

在颜色索引模式下，可以通过颜色屏蔽实现大量的技巧。例如，可以把索引中的每个位当作一个不同的层，并且根据适当的颜色表设置，设置任意层之间的交互。可以创建顶层和底层效果，以及所谓的颜色表动画效果（有关颜色屏蔽的例子，请参阅第14章）。RGBA模式下的屏蔽用得相对较少，但是仍有一些应用。例如，可以用它把独立的图像分别加载到红、绿和蓝位平面。

我们已经在第6.1.6节中看到了禁用深度缓冲区的一种用途。如果一个通用的背景适用于多个帧，

并且想添加一些可能被背景的某些部分遮挡的物体，也可以禁用深度缓冲区的写入。例如，假设背景是一片森林，我们想重复绘制一些具有相同树木的帧，但是有一些物体在树木之间移动。当这些树木被绘制并且它们的深度值记录在深度缓冲区之后，保存树木的图像，然后在禁用深度缓冲区的情况下绘制这些新的物体。只要这些新物体相互之间不重叠，图像就是正确的。为了绘制下一帧，可以恢复树木的图像，并在此基础上继续绘图，并不需要恢复深度缓冲区的值。在背景极为复杂的情况下（复杂到把图像重新复制到颜色缓冲区要比根据几何描述从头渲染图像快得多），这个技巧是非常实用的。

对模板缓冲区进行屏蔽，可以使用一个多位模板缓冲区保存多个模板（每个模板1个位）。可以根据这个技巧，执行第10.2.3节所解释的加盖技术，或者实现第14.20节所描述的生存游戏。

注意：glStencilMask()函数指定的掩码控制哪些位平面可以进行写入。这个掩码与glStencilFunc()函数的第3个参数所指定的掩码并无关联，后者只是指定了哪些位平面需要被模板函数考虑。

10.2 片断测试和操作

在屏幕上绘制几何图形、文本或图像时，OpenGL会执行一些计算，进行旋转、移动和缩放，以便确定光照效果以及物体投影到窗口坐标的方式，并确定哪些像素的颜色应该绘制。本书前面很多章节提供了一些有关如何控制这些操作的信息。在OpenGL确定了每个片断是否应该生成以及应该使用什么颜色之后，这个片断仍然需要经历一些处理阶段，这些操作控制是否需要这个片断以及如何把它写入到帧缓冲区中。例如，如果一个片断位于一个矩形区域的外部，或者它和帧缓冲区中已经存在的像素相比距离观察点更远，它就不会被绘制。在另一个阶段，片断的颜色与帧缓冲区中已经存在的像素颜色进行混合。

本节描述片断在进入帧缓冲区之前必须经历的所有测试，以及当片断写入到帧缓冲区时可能对它进行的最终操作。这些测试和操作是按照下面的顺序进行的。如果片断在其中某个测试中被排除，后面的测试或操作就不再进行。

1) 裁剪测试。
2) alpha测试。
3) 模板测试。
4) 深度测试。
5) 混合。
6) 抖动。
7) 逻辑操作。

下面各小节将详细描述这些测试和操作。

10.2.1 裁剪测试

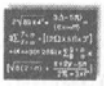
可以使用glScissor()函数，在窗口中定义一个矩形区域，并把绘图限制在这个区域之内。如果一个片断位于这个区域的内部，那么它就能够通过裁剪测试。

---

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

设置裁剪矩形（又称为裁剪框）的位置和大小。这个函数的参数定义了这个裁剪矩形的左下角（*x*、*y*）以及它的宽度和高度。位于这个矩形内部的像素可以通过裁剪测试。可以通过向**glEnable()**和**glDisable()**函数传递GL_SCISSOR_TEST参数，分别启用和禁用裁剪测试。在默认情况下，裁剪框就是窗口的大小，并且裁剪测试是被禁用的。

裁剪测试只是模板测试的一个版本，它使用的是屏幕上的一个矩形区域。用硬件实现裁剪测试相当容易，而模板测试的实现则可能相当慢，也许是因为模板测试是用软件执行的。



高级话题

裁剪测试的一个高级应用是执行非线性投影。首先，把窗口划分为常规的子区域网格，指定视口和裁剪参数，并在某一时刻把渲染限制在一个区域内部。然后，使用不同的投影矩阵，把整个场景投影到每个区域中。

为了判断裁剪测试是否被启用，并获取已定义的裁剪矩形的值，可以分别在glIsEnable()函数中使用GL_SCISSOR_TEST参数以及在调用glGetIntegerv()时使用GL_SCISSOR_BOX参数。

10.2.2 alpha测试

兼容性扩展
GL_ALPHA_TEST

在RGBA模式下，alpha测试允许根据一个片断的alpha值接受或拒绝它。可以通过向glEnable()和glDisable()函数传递GL_ALPHA_TEST，分别启用和禁用alpha测试。为了判断alpha测试是否启用，可以调用glIsEnabled()函数，并以GL_ALPHA_TEST为参数。

注意：alpha片段测试在OpenGL 3.0中废弃了，并且在OpenGL 3.1中，替代为通过使用discard操作在片段着色器中丢弃片段。参见第15.5.4节中的“流控制语句”的介绍。

如果alpha测试被启用，它就把源片断的alpha值与一个参考值进行比较，并根据比较结果接受或拒绝这个片断。可以用glAlphaFunc()函数设置参考值和比较函数。在默认情况下，这个参考值是0，比较函数是GL_ALWAYS，alpha测试被禁用。为了获取alpha比较函数或参考值，可以使用glGetIntegerv()函数，并分别以GL_ALPHA_TEST_FUNC或GL_ALPHA_TEST_REF为参数。

兼容性扩展
glAlphaFunc

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

设置用于alpha测试的参考值和比较函数。参考值*ref*的范围被限定在0~1之间。表10-2列出了func可能使用的值以及它们的含义。

表10-2 glAlphaFunc()的参数值

参 数	含 义
GL_NEVER	决不接受这个片断
GL_ALWAYS	总是接受这个片断
GL_LESS	如果片断的alpha小于参考值便接受它
GL_LEQUAL	如果片断的alpha小于等于参考值便接受它
GL_EQUAL	如果片断的alpha等于参考值便接受它
GL_GEQUAL	如果片断的alpha大于等于参考值便接受它
GL_GREATER	如果片断的alpha大于参考值便接受它
GL_NOTEQUAL	如果片断的alpha不等于参考值便接受它

alpha测试的一个应用就是实现一种透明算法。可以分两次渲染整个场景，第一次只接受alpha值为1的片断，第二次接受alpha值不为1的片断。在这两次渲染时都打开深度缓冲区，但是在第二道渲染时禁止写入到深度缓冲区。

alpha测试的另一种用途是用纹理图像制作贴花，并且可以看透帖花的某些部分。可以把能够看透的贴花部分的alpha值设置为0.0，其他地方设置为1.0，并且把参考值设置为0.5（或者0.0~1.0之间

的任何值），并且把比较函数设置为GL_GREATER。这样，贴花就具有能够看透的部分，并且深度缓冲区中的值也不会受到影响。这个技巧又称“billboarding”，在第6.1.4节中有所介绍。

10.2.3 模板测试

模板测试只有在存在模板缓冲区的情况下才会执行（如果不存在模板缓冲区，模板测试总是能够通过）。模板测试把像素存储在模板缓冲区的值与一个参考值进行比较。根据测试结果，对模板缓冲区中的这个值进行相应的修改。可以使用glStencilFunc()和glStencilOp()函数选择需要使用的特定比较函数、参考值以及对模板缓冲区所执行的修改操作。

---

void **glStencilFunc**(GLenum *func*, GLint *ref*, GLuint *mask*);

void **glStencilFuncSeparate**(GLenum *face*, GLenum *func*, GLint *ref*, GLuint *mask*);

设置模板测试所使用的比较函数（*func*）、参考值（*ref*）和掩码（*mask*）。这个函数根据比较函数把参考值与模板缓冲区中的值进行比较，但比较只在那些对应的掩码设置为1的位上进行。比较函数可以是GL_NEVER、GL_ALWAYS、GL_LESS、GL_LEQUAL、GL_QUAL、GL_GEQUAL、GL_GREATER、GL_NOTEQUAL。例如，如果比较函数是GL_LESS，那么只有当*ref*小于模板缓冲区中对应的值时才通过模板测试。如果模板缓冲区包含了s个位平面，那么在执行比较之前，*mask*中低端的s个位都与模板缓冲区中的值以及参考值进行AND操作。被屏蔽的值都解释为非负值。我们可以向**glEnable()**和**glDisable()**函数传递GL_STENCIL_TEST参数，启用或禁用模板测试。在默认情况下，*func*是GL_ALWAYS，*ref*是0，*mask*是全1，模板测试被禁用。

OpenGL 2.0添加了**glStencilFuncSeparate()**函数，允许为多边形的正面和背面分别指定模板函数参数。

---

void **glStencilOp**(GLenum *fail*, GLenum *zfail*, GLenum *zpass*);

void **glStencilOpSeparate**(GLenum *face*, GLenum *fail*, GLenum *zfail*, GLenum *zpass*);

指定了当一个片断通过或未通过模板测试时，模板缓冲区中的数据如何进行修改。*fail*、*zfail*和*zpass*这3个函数可以是GL_KEEP、GL_ZERO、GL_REPLACE、GL_INCR、GL_INCR_WRAP、GL_DECR、GL_DECR_WRAP或GL_INVERT。它们分别对应于保持当前值、用0替换当前值、用参考值替换当前值、在使用饱和的情况下将当前值加1、在不使用饱和的情况下将当前值加1、在使用饱和的情况下将当前值减1、在不使用饱和的情况下将当前值减1以及对当前值进行逐位的反转。增值和减值函数的结果被限制在0和最大无符号整型值（例如模板缓冲区保存了s个位，那么这个最大值是2^s-1）之间。

如果片断无法通过测试，*fail*函数就会应用。如果通过测试，在深度测试失败的情况下应用zfail函数，在深度测试也通过或者未执行深度测试的情况下应用*zpass*函数（参见第10.2.4节）。在默认情况下，所有3个模板测试都设置为GL_KEEP。

OpenGL 2.0增加了glStencilOpSeparate()函数，允许为多边形的正面和背面指定独立的模板测试。

“使用饱和”表示模板值将会限制在极端值之下。如果在使用饱和的情况下对模板值0进行减1操作，它的值仍然保持为0。“不使用饱和”表示模板值在超出了限定范围后会回绕。如果在不使用饱和的情况下对模板值0执行减1操作，它的值便会回绕为最大的无符号整数值（这将是一个相当大的值）。

模板查询

可以使用查询函数glGetInteger()，获取与全部6个模板相关的参数值。表10-3列出了这些值。可以调用glIsEnabled()函数（以GL_STENCIL_TEST为参数）判断模板测试是否启用。

表10-3 模板测试的的查询值

参 数	含 义
GL_STENCIL_FUNC	模板函数
GL_STENCIL_REF	模板参考值
GL_STENCIL_VALUE_MASK	模板掩码
GL_STENCIL_FAIL	模板测试失败后的操作
GL_STENCIL_PASS_DEPTH_FAIL	模板测试通过但深度测试失败后的操作
GL_STENCIL_PASS_DEPTH_PASS	模板测试通过并且深度测试通过后的操作

模板示例

模板测试最常见的用途是屏蔽掉屏幕中一些不规则的区域，以免在这些区域中进行绘图（如第10.1节中的挡风玻璃例子）。为此，可以把模板掩码设置为全0，并且在模板缓冲区中用1绘制所需要的形状。我们无法把几何图形直接绘制到模板缓冲区，但是可以通过把它写入到颜色缓冲区，并为zpass参数选择一个适当的值（例如GL_REPLACE）来实现这个目的。（可以使用glDrawPixels()函数把像素数据直接绘制到模板缓冲区。）在进行绘图时，模板缓冲区中也会写入一个值（在此例中为参考值）。为了防止模板缓冲区的绘图影响颜色缓冲区，可以把颜色掩码设置为0（或GL_FALSE）。另外，还可以禁止对深度缓冲区的写入。

在定义了模板区域之后，就可以把参考值设置为1，并设置比较函数，如果模板平面值等于参考值就通过测试。在绘图过程中，不要对模板平面的内容进行修改。

示例程序10-1显示了如何按照这种方式使用模板测试。这个程序绘制了两个圆环面，场景的中央有一个钻石形状的镂空。在钻石形状的模板掩码内部绘制了一个球体。在这个例子中，只有当窗口进行重绘时才会在模板缓冲区中进行绘图，因此颜色缓冲区的内容在模板掩码创建之后被清除。

示例程序10-1 使用模板缓冲区：stencil.c

```
#define YELLOWMAT 1
#define BLUEMAT 2

void init(void)
{
   GLfloat yellow_diffuse [] ={0.7,0.7,0.0,1.0 };
   GLfloat yellow_specular [] ==={1.0,1.0,1.0,1.0 };

   GLfloat blue_diffuse [] ==={0.1,0.1,0.7,1.0 };
   GLfloat blue_specular []={0.1,1.0,1.0,1.0 };

   GLfloat position_one [] ==={1.0,1.0,1.0,0.0 };

   glNewList(YELLOWMAT,GL_COMPILE);
   glMaterialfv(GL_FRONT,GL_DIFFUSE,yellow_diffuse);
   glMaterialfv(GL_FRONT,GL_SPECULAR,yellow_specular);
   glMaterialf(GL_FRONT,GL_SHININESS,64.0);
```

```
    glEndList();

    glNewList(BLUEMAT,GL_COMPILE);
    glMaterialfv(GL_FRONT,GL_DIFFUSE,blue_diffuse);
    glMaterialfv(GL_FRONT,GL_SPECULAR,blue_specular);
    glMaterialf(GL_FRONT,GL_SHININESS,45.0);
    glEndList();

    glLightfv(GL_LIGHT0,GL_POSITION,position_one);

    glEnable(GL_LIGHT0);
    glEnable(GL_LIGHTING);
    glEnable(GL_DEPTH_TEST);

    glClearStencil(0x0);
    glEnable(GL_STENCIL_TEST);
}

/*Draw a sphere in a diamond-shaped section in the
 *middle of a window with 2 tori.
 */
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);

/*draw blue sphere where the stencil is 1 */
    glStencilFunc(GL_EQUAL,0x1,0x1);
    glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP);
    glCallList(BLUEMAT);
    glutSolidSphere(0.5,15,15);
/*draw the tori where the stencil is not 1 */
    glStencilFunc(GL_NOTEQUAL,0x1,0x1);
    glPushMatrix();
       glRotatef(45.0,0.0,0.0,1.0);
       glRotatef(45.0,0.0,1.0,0.0);
       glCallList(YELLOWMAT);
       glutSolidTorus(0.275,0.85,15,15);
       glPushMatrix();
          glRotatef(90.0,1.0,0.0,0.0);
          glutSolidTorus(0.275,0.85,15,15);
       glPopMatrix();
    glPopMatrix();
}

/*Whenever the window is reshaped,redefine the
 *coordinate system and redraw the stencil area.
 */
void reshape(int w,int h)
{
    glViewport(0,0,(GLsizei)w,(GLsizei)h);

/*create a diamond shaped stencil area */
```

```
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <=h)
       gluOrtho2D(-3.0,3.0,-3.0*(GLfloat)h/(GLfloat)w,
                  3.0*(GLfloat)h/(GLfloat)w);
    else
       gluOrtho2D(-3.0*(GLfloat)w/(GLfloat)h,
                  3.0*(GLfloat)w/(GLfloat)h,-3.0,3.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    glClear(GL_STENCIL_BUFFER_BIT);
    glStencilFunc(GL_ALWAYS,0x1,0x1);
    glStencilOp(GL_REPLACE,GL_REPLACE,GL_REPLACE);
    glBegin(GL_QUADS);
       glVertex2f(-1.0,0.0);
       glVertex2f(0.0,1.0);
       glVertex2f(1.0,0.0);
       glVertex2f(0.0,-1.0);
    glEnd();

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0,(GLfloat)w/(GLfloat)h,3.0,7.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0,0.0,-5.0);
}
/*Main Loop
*Be certain to request stencil bits.
*/
int main(int argc,char**argv)
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB |
                        GLUT_DEPTH |GLUT_STENCIL);
    glutInitWindowSize(400,400);
    glutInitWindowPosition(100,100);
    glutCreateWindow(argv [0]);
    init();
    glutReshapeFunc(reshape);
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

下面这些例子说明了模板测试的其他用途。第14章还提供了一些使用模板测试的其他思路。

- 加盖：假如绘制的是一个闭合的由几个多边形组成的凸形物体（或者几个，只要它们彼此并不相交或包围），并且裁剪平面可能会切除物体的一部分，也可能并不切除。假设裁剪平面与物体相交，我们想用某种颜色为物体加上盖，以免看到物体的内部。为此，可以用0清除模板缓冲区，然后在绘图之前启用模板测试，并把模板比较函数设置为GL_ALWAYS，并且在片断通

过模板测试时将模板缓冲区的值取反。在绘制完所有的物体之后，不需要加盖的屏幕区域对应的模板缓冲区值为0，需要加盖的区域则为非零值。接着，可以重新设置模板函数，只绘制模板缓冲区值不为0的区域，并使用加盖颜色绘制一个穿越整个屏幕的多边形。

- 重叠的半透明多边形：假设有一个半透明表面，它由多个稍微重叠的多边形组成。如果简单地使用alpha混合，底层的那部分物体被多个半透明表面所覆盖，看起来效果不佳。可以使用模板缓冲区，确保每个片断最多只被组成半透明表面的1个多边形覆盖。为此，可以使用0清除模板缓冲区，并且在模板平面为0的地方才绘制片断，在绘制片断之后将模板缓冲区的值增加1。
- 点画模式：假设我们想使用点画模式来绘制图像（有关点画模式的详细信息，请参阅第2.4节）。为此，可以把点画模式写入到模板缓冲区，然后根据模板缓冲区的内容进行绘图。写入点画模式之后，在绘制图像时，模板缓冲区的内容不会改变，因此物体将根据模板缓冲区中的点画模式进行绘制。

10.2.4 深度测试

对于屏幕上的每个像素，深度缓冲区负责记录观察点和占据这个像素的物体之间的距离。然后，如果能够通过指定的深度测试，源片断的深度值就会替换深度缓冲区中已经存在的值。

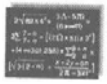
深度缓冲区一般用于隐藏表面的消除。如果像素出现了一种新的候选颜色，只有当对应的新物体比原来的物体更靠近观察点时，它才会被绘制。按照这种方式，在整个场景被渲染之后，只有那些没有被其他物体遮挡的物体才会保留。最初，深度缓冲区的清除值表示一个与观察点尽可能远的距离，因此任何物体的深度值都要小于它。如果这就是我们所需要的深度缓冲区使用方式，只要简单地向glEnable()函数传递GL_DEPTH_TEST，并记得在重绘每一帧之前清除深度缓冲区（参见第10.1.2节）。另外，还可以使用glDepthFunc()函数为深度测试选择一个不同的比较函数。

---

void **glDepthFunc**(GLenum *func*);

为深度测试设置比较函数。*func*的值必须是GL_NEVER、GL_ALWAYS、GL_LESS、GL_LEQUAL、GL_QUAL、GL_GEQUAL、GL_GREATER或GL_NOTEQUAL。当源片断的z值与已经存储于深度缓冲区中的对应z值满足指定的关系时，这个片断就通过了深度测试。比较函数的默认值是GL_LESS，表示源片断的z值如果小于深度缓冲区中已经存储的值，这个片断便通过深度测试。在这种情况下，z值表示物体和观察点之间的距离。它的值越小，表示物体和观察点的距离越短。

10.2.5 遮挡查询



高级话题

深度缓冲区决定了每个像素是否可见。为了提高性能，在渲染几何物体（有可能非常复杂）之前判断它是否可见是一个不错的思路（如果不可见就不用绘制它）。遮挡查询（occlusion query）允许我们判断一组几何图形在进行了深度测试之后是否可见。

对于由大量多边形组成的复杂几何形状，这个技巧显得尤其重要。使用这种方法，可能并不需要绘制复杂物体的所有几何形状，只需要绘制它的边框以及其他较为简单的部分（所需要的渲染资源也要少很多）。如果遮挡查询的结果表示某部分几何图形的渲染不会对当前可见的片断或样本产生影响，那么这部分几何图形就在场景中不可见，也就不需要进行绘制。

为了使用遮挡查询，需要执行如下步骤：

1) 为每个所需的遮挡查询生成一个查询ID（可选）。

2) 调用glBeginQuery()，表示开始一个遮挡查询。

3) 渲染需要进行遮挡查询的几何图形。

4) 调用glEndQuery()，表示已经完成了遮挡查询。

5) 提取通过了遮挡查询的样本数量。

为了使遮挡查询的处理尽量保持高效，可以禁用所有可能增加渲染时间但不会影响像素可见性的渲染模式。

生成查询对象

遮挡查询对象标识符只不过是一个无符号整数。尽管不是严格必需，但是让OpenGL生成一组ID仍然是一种很好的做法。glGenQueries()函数可以生成一组未使用的查询ID，供接下来使用。

---

void **glGenQueries**(GLsizei *n*, GLuint **ids*);

返回*n*个当前未使用的ID，可用于表示遮挡查询对象的名称。这些名称是在*ids*数组中返回的，它们并不一定是连续的整数。

这个函数返回的名称被标记为已使用，以便分配其他查询对象。但是它们只有在**glBeginQuery()**函数中被指定之后才处于有效的状态。

0是一个被保留的遮挡查询名称，决不会出现在**glGenQueries()**查询返回的名称列表中。

可以调用glIsQuery()函数，判断一个标识符当前是否由一个遮挡查询对象所使用。

GLboolean **glIsQuery**(GLuint *id*);

如果*id*是一个遮挡查询对象的名称，这个函数返回GL_TRUE。如果*id*是0或者它不是一个遮挡查询对象的名称，这个函数返回GL_FALSE。

对遮挡查询对象进行初始化

为了指定一个遮挡查询所使用的几何图形，只要在glBeginQuery()和glEndQuery()之间包括需要进行的渲染操作，如示例程序10-2所示。

示例程序10-2 用遮挡查询渲染几何图形：occquery.c

```
glBeginQuery(GL_SAMPLES_PASSED,Query);
glBegin(GL_TRIANGLES);
glVertex3f(0.0,0.0,1.0);
glVertex3f(-1.0,0.0,0.0);
glVertex3f(1.0,0.0,0.0);
glEnd();
glEndQuery(GL_SAMPLES_PASSED);
```

在使用遮挡查询时，可以使用除了glGenQueries()和glDeleteQueries()之外，几乎所有的OpenGL操作，它们会导致GL_INVALID_OPERATION错误。

---

void **glBeginQuery**(GLenum *target*, GLuint *id*);

指定一次遮挡查询操作的开始。*target*必须是GL_SAMPLES_PASSED。*id*是一个无符号整数标识符，标识了这个遮挡查询操作。

---

void **glEndQuery**(GLenum *target*);

结束一次遮挡查询操作。*target*必须是GL_SAMPLES_PASSED。

判断遮挡查询的结果

在完成对需要进行遮挡查询的几何物体的渲染之后，需要提取遮挡查询的结果。可以通过调用glGetQueryObject[u]iv()函数来完成，如示例程序10-3所示。这个函数将返回片断或样本（如果使用了多重采样，详见第6.2.2节中的“alpha和多重采样覆盖”）的数量。

---

void **glGetQueryObjectiv**(GLenum *id*, GLenum *pname*, GLint **params*);

void **glGetQueryObjectuiv**(GLenum *id*, GLenum *pname*, GLuint **params*);

查询一个遮挡查询对象的状态。*id*是这个遮挡查询对象的名称。如果*pname*是GL_QUERY_RESULT，*params*将包含通过了深度测试的片断或样本（如果启用了多重采样）的数量。如果数量为0，表示这个对象被完全遮挡。

在完成遮挡查询操作时可能会产生延迟。如果*pname*是GL_QUERY_RESULT_ AVAILABLE，并且查询id的结果有效，*params*将包含GL_TRUE，否则将包含GL_FALSE。

示例程序10-3 提取遮挡查询的结果occquery.c

```
count =1000;/*counter to avoid a possible infinite loop */
while (!queryReady &&count--){
    glGetQueryObjectiv(Query,GL_QUERY_RESULT_AVAILABLE,
       &queryReady);
}
if (queryReady){
   glGetQueryObjectiv(Query,GL_QUERY_RESULT,&samples);
   fprintf(stderr,"Samples rendered:%d \n",samples);
}
else {
   fprintf(stderr,"Result not ready ...rendering
      anyways \n");
   samples =1;/*make sure we render */
}

if (samples >0)
   glDrawArrays(GL_TRIANGLE_FAN,0,NumVertices);
```

清除遮挡查询对象

在完成了遮挡查询测试之后，就可以调用glDeleteQueries()函数，释放与遮挡查询对象相关的资源。

---

void **glDeleteQueries**(GLsizei *n*, const GLuint **ids*);

删除*n*个遮挡查询对象，它们的名称由*ids*数组提供。被释放的遮挡查询对象可以被复用（例如，通过**glGenQueris()**函数重新生成）。

10.2.6 条件渲染



高级话题

遮挡查询的问题之一，是它们需要OpenGL暂停对几何图元和片段的处理，计算深度缓冲区中受影响的样本的数目，并把这个值返回给应用程序。在性能敏感的应用程序中，以这种方式停止现

代的图形硬件，通常会给性能带来灾难性的影响。为了不必再暂停OpenGL的操作，条件渲染允许图形服务器（硬件）判断一个遮挡查询是否产生任何片段以及是否渲染所涉及到的命令。使用glGetQuery*()的形式把想要条件执行的渲染操作包围起来，这样就可以打开条件渲染。

void **glBeginConditionalRender**(GLuint *id*, GLenum *mode*);

void **glEndConditionalRender**(void);

将那些根据遮挡查询对象id的结果可能丢弃的一系列OpenGL渲染命令删除掉。*mode*指定了OpenGL实现如何使用遮挡查询的结果，并且它必须是如下之一：GL_QUERY_WAIT、GL_QUERY_NO_WAIT、GL_QUERY_BY_REGION_WAIT或GL_QUERY_BY_REGION_NO_WAIT。

如果*id*不是一个已有的遮挡查询，将会设置GL_INVALID_VALUE。如果当一个条件渲染序列在执行中的时候调用了**glBeginConditionalRender()**，或者如果没有条件渲染在执行中的时候调用了**glEndConditionalRender()**，或者如果*id*是一个遮挡查询对象的名字，而*target*不是GL_SAMPLES_PASSED，或者如果*id*是一个处理中的遮挡查询的名字，将会产生GL_INVALID_OPERATION。

示例程序10-4的代码完全替代了示例程序10-3中的代码行。这种方法不仅代码更加紧凑，而且由于把执行查询完全移到了OpenGL服务器，消除了性能的最大障碍，也很高效。

示例程序10-4 使用条件渲染：condrender.c

```
glBeginConditionalRender(Query,GL_QUERY_WAIT);
glDrawArrays(GL_TRIANGLE_FAN,0,NumVertices);
glEndConditionalRender();
```

10.2.7 混合、抖动和逻辑操作

当一个源片断通过了第10.2节描述的所有测试之后，它就可以按照不同的方式与颜色缓冲区的当前内容进行组合。最简单的方式（也是默认的方式）就是覆盖原先的值。另外，如果使用的是RGBA模式，并且想对片断进行半透明或抗锯齿处理，可以把它的值与缓冲区中已有的值求平均（混合）。在可用颜色数量较少的系统中，可能需要对颜色值进行抖动，在适当损失颜色质量的情况下增加可使用的颜色数量。最后，可以使用各种位操作符，对源片断和已经写入的片断进行组合。

混合

混合操作把源片断的R、G、B和alpha值与已经存储在这个位置的像素的对应值进行组合。可以使用不同的混合操作，混合的执行取决于源片断的alpha值和这个像素已经存储的alpha值（如果有的话）。关于这个话题的深入讨论，请参阅第6.1节。

从OpenGL 3.0开始，可以使用如下命令，在每个缓冲区的基础上控制混合：

void **glEnablei**(GLenum *target*, GLuint *index*);

void **glDisablei**(GLenum *target*, GLuint *index*);

为缓冲区*index*打开或关闭混合。*target*必须是GL_BLEND。如果*index*大于或等于GL_MAX_DRAW_BUFFERS，将会产生A GL_INVALID_VALUE。

要确定对于一个特定的缓冲区是否打开了混合，使用glIsEnabledi()。

GLboolean **glIsEnabledi**(GLenum *target*, GLuint *index*);

确定对于缓冲区*index*来说，*target*是否打开。

对于OpenGL 3.0，*target*必须是GL_BLEND，否则，产生一个GL_INVALID_ENUM。如果*index*超出了*target*支持的范围，将会产生一个GL_INVALID_VALUE。

抖动

在那些只有少量颜色位平面的系统中，可以通过对图像中的颜色进行抖动，在略微损失颜色质量的前提下，增加可用颜色的数量。抖动类似于报纸所使用的半调（halftone）技巧。虽然《New York Times》只使用两种颜色（黑和白），但是它可以使用由黑点和白色所组成的灰色来显示照片。报纸上的照片图像（没有灰色调）与真实的照片（具有灰度）相比，分辨率的损失是显而易见的。类似地，那些只有较少数量位平面的系统可以对邻近像素的红、绿、蓝值进行抖动，产生范围更广的颜色值。

抖动操作是与硬件相关的。所有的OpenGL实现都允许打开或关闭这个操作。事实上，在有些计算机上，启用抖动根本不会引发抖动操作，因为计算机如果已经具有很高的颜色分辨率，根本不需要通过抖动来增加颜色。为了启用和禁用抖动，可以分别向glEnable()和glDisable()函数传递GL_DITHER参数。在默认情况下，抖动是被启用的。

抖动在RGBA模式和颜色索引模式下均可使用。OpenGL以某种与硬件相关的方式交替使用两个最接近的颜色值或颜色索引值。例如，在颜色索引模式下，如果启用了抖动，并且被绘制的颜色索引是4.4，OpenGL就使用索引4绘制60%的像素，用索引5绘制其余40%的像素（抖动算法有很多，但是任何一种抖动算法都只依赖于源片断的颜色值以及它的x坐标和y坐标）。在RGBA模式下，OpenGL需要对每个成分（包括alpha成分）进行抖动。在颜色索引模式下，OpenGL通常需要在颜色表中以适当的梯度来指定颜色，否则可能会得到奇怪的图像。

逻辑操作

片断执行的最后一种操作是逻辑操作，例如OR、XOR或INVERT，它们作用于源片断值和（或）颜色缓冲区中的目标片断。这类逻辑操作在bit-blt类型的计算机上尤其实用。在这种类型的计算机上，主要的图形操作就是把一块数据矩形从窗口上的一个地方复制到另一个地方，从窗口复制到处理器内存，或者从内存复制到窗口。一般而言，这种复制并不是把数据直接写入到内存，而是允许在源数据和已经存在的数据之间执行任意的逻辑操作。然后，再用操作结果、替换原有的数据。

由于这个过程可以相当方便地用硬件实现，因此这种类型的计算机数量很多。下面举例说明逻辑操作的用途。XOR可以实现一种可撤销的方式来绘制图像。简单地对同一幅图像连续执行两次XOR操作，所得到的结果就是最初的图像。另外，在使用颜色索引模式时，颜色索引可以解释为位模式。然后，可以把一幅图像合成为多个层面上的绘图组合，使用写入掩码把绘图限制在不同组的位平面中，并执行逻辑操作来修改不同的层面。

可以向glEnable()和glDisable()函数传递GL_INDEX_LOGIC_OP或GL_COLOR_ LOGIC_OP参数，分别在颜色索引模式和RGBA模式下启用和禁用逻辑操作。还必须用glLogicOp()函数在16种逻辑操作中做出选择，或者使用默认值GL_COPY。为了与OpenGL 1.0版本保持向后兼容，也可以调用glEnable(GL_LOGIC_OP)，在颜色索引模式下启用逻辑操作。

void **glLogicOp**(GLenum *opcode*);

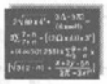
选择需要执行的逻辑操作，用于组合源片断和当前存储于颜色缓冲区中的像素（目标像素）。表10-4显示了*opcode*可以使用的值以及它们的含义（*s*表示源片断，*d*表示目标像素）。默认的逻辑操作是GL_COPY。

表10-4 16种逻辑操作

参数	操作	参数	操作
GL_CLEAR	0	GL_AND	$s \wedge d$
GL_COPY	s	GL_OR	$s \vee d$
GL_NOOP	d	GL_NAND	$\neg(s \wedge d)$
GL_SET	1	GL_NOR	$\neg(s \vee d)$
GL_COPY_INVERTED	$\neg s$	GL_XOR	s *XOR* d
GL_INVERT	$\neg d$	GL_EQUIV	$\neg(s$ *XOR* $d)$
GL_AND_REVERSE	$s \wedge \neg d$	GL_AND_INVERTED	$\neg s \wedge d$
GL_OR_REVERSE	$s \vee \neg d$	GL_OR_INVERTED	$\neg s \vee d$

10.3 累积缓冲区

注意：累计缓冲区已经从OpenGL 3.1中废弃删除。本书以前版本的本章中所介绍的一些技术（主要是全景抗锯齿），已经为其他的技术所替代（参见6.2.2节中的“alpha和多重采样覆盖”）。其他的技术使用帧缓冲区（OpenGL 3.0新增的功能）中的浮点像素格式很容易实现。这些技术的概念和这里所介绍的是相似的。



高级主题

累积缓冲区可以用于场景抗锯齿、运动模糊、景深模拟以及计算多个光源所产生的柔和阴影等。此外，它还有一些其他应用，尤其是当它与其他缓冲区联合使用时（关于如何使用累积缓冲区的更多细节，请参阅《The Accumulation Buffer: Hardware Support for High-Quality Rendering》，作者Paul Haeberli和Kurt Akeley[SIGGRPH 1990 Proceedings，第309～318页]）。

OpenGL的图形操作并不是直接写入到累积缓冲区。一般情况下，OpenGL在一个标准颜色缓冲区中产生一系列的图像，然后以每次一幅的方式把它们累积到累积缓冲区中。在累积操作完成后，再把结果复制到颜色缓冲区中以便查看。为了减少舍入误差，累积缓冲区的精度（每种颜色成分的位数）可能比标准的颜色缓冲区更高。显然，多次渲染场景所需要的时间比一次渲染要长，但是它的渲染质量更高。在应用程序中，需要在渲染质量和渲染速度之间进行合理的权衡。

可以按照摄像师对胶片进行多次曝光的方式使用累积缓冲区。通常，摄像师在未前推胶卷的情况下对同一场景进行多次拍摄，以实现多次曝光。如果场景中存在移动的物体，它们看上去就会显得比较模糊。与摄像师使用相机所做的事情相比，我们可以使用计算机对图像进行更多的处理。例如，计算机可以非常精确地控制观察点，但摄像师就无法精确地移动相机的位置（关于如何清除累积缓冲区的细节，请参阅第10.1.2节。glAccum()函数可以对累积缓冲区进行控制）。

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

兼容性扩展
glAccum and all accepted tokens

控制累积缓冲区。*op*参数选择操作，*value*是该操作的使用数量。可以使用的操作有：GL_ACCUM、GL_LOAD、GL_RETURN、GL_ADD和GL_MULT。

- GL_ACCUM从**glReadBuffer()**选择的用于读取的当前缓冲区中读取每个像素，把R、G、B、*alphha*值与*value*相乘，然后把结果添加到累积缓冲区中。
- GL_LOAD与GL_ACCUM基本相同，区别在于用计算所得的值替换累积缓冲区中的值，而不是添加到累积缓冲区。
- GL_RETURN从累积缓冲区接收值，把它们与*value*相乘，然后把结果放在

可以进行写入的颜色缓冲区中。

- GL_ADD和GL_MULT分别简单地把累积缓冲区中每个像素的值与*value*相加或相乘，并把它返回到累积缓冲区中。对于GL_MULT，value被截取在[−1.0，1.0]的范围之内，而GL_ADD则不进行这种截取。

10.3.1 运动模糊

可以使用类似的方法来模拟运动模糊效果，如彩图7和图10-2所示。假设场景由一些静止的物体和一些移动的物体组成，我们想绘制一幅运动模糊图像，表示场景在一小段时间内的情况。可以按照相同的方式设置累积缓冲区，不是对图像进行空间上的微移，而是进行时间上的微移。可以按照如下方式调用glAccum()函数：

```
glAccum(GL_MULT, decayFactor);
```

这样，随着场景被绘制到累积缓冲区中，整个场景将越来越模糊。其中decayFactor是一个0.0~1.0之间的数字，其值越小，物体的运动速度就越快。然后，可以使用下面这个调用，将完成后的图像（显示了物体的当前位置及以前位置的“汽化痕迹”）从累积缓冲区转移到标准的颜色缓冲区：

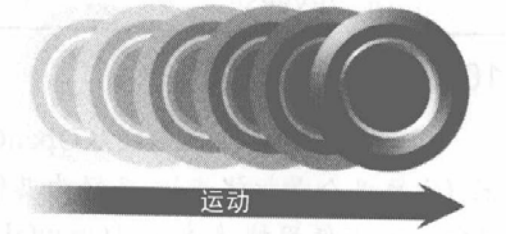


图10-2 运行模糊的物体

```
glAccum(GL_RETURN, 1.0);
```

即使物体以不同的速度移动，或者有些物体进行的是加速运动，图像看上去仍是正确的。和前面一样，微移点（在此例中是时间性的）越多，最终图像的质量就越高，直到由于累积缓冲区的有限精度而无法进一步提高分辨率为止。可以同时在时间和空间上进行微移，同时实现运动模糊和场景抗锯齿效果。但是，图像渲染质量的提高总是要以增加渲染时间为代价的。

10.3.2 景深

照相机在拍摄照片时，存在一个和胶卷保持特定距离的平面，这个平面上的物体在照片中显得最为清晰。离这个平面越远，物体在照片上就越模糊。照相机的景深是以聚焦面为中心的一个非常小的区域，在这个区域中，物体能够实现完美的聚焦效果。

在正常情况下，OpenGL绘制的所有物体都进行了正确的聚焦（除非用户使用的显示器坏了，在这种情况下，所有的物体看上去都很模糊）。累积缓冲区可以模拟我们在照片中所看到的效果，物体和聚焦平面的距离越远，它们看上去就越模糊。这个技巧并不是对照相机所产生的这种效果进行准确模拟，但是它的结果看上去有点类似。

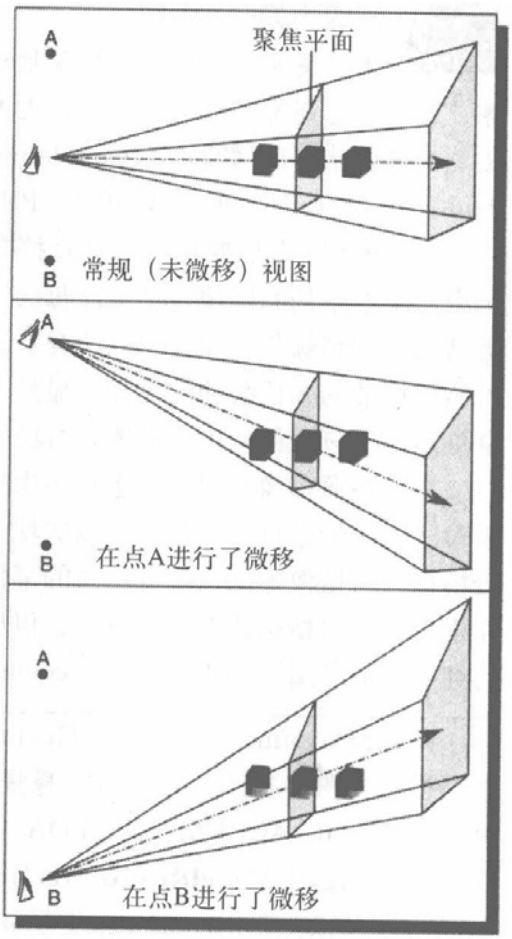


图10-3 对视景体进行微移，实现景深效果

为了实现这个效果，可以通过在glFrustum()函数中使用不同的参数值，对场景进行反复绘制。可以选择这样的参数：当观察点的位置发生稍微的变化时，每个视景体都覆盖了聚焦平面上的同一个矩形，如图10-3所示。

另外，可以使用累积缓冲区，按照常规的方法，对所有的渲染结果求平均值。

彩图10是一幅使用景深效果绘制的图像。这幅图像共包括5个茶壶，其中金色茶壶（左数第二个）正好位于聚焦平面上。对于其他茶壶，距离聚焦平面（金色茶壶）越远，茶壶就越模糊。示例程序10-7显示了用于绘制这幅图像的代码，其中accPerspective()和accFrustum()函数的定义如示例程序10-4所示。场景共绘制8次，每次都调用accPerspective()函数对视景体进行微移。为了实现场景抗锯齿，这个函数的第5个和第6个参数分别是在x和y方向上对视景体进行微移。但是，为了实现景深效果，视景体在进行微移时需要始终覆盖整个聚焦平面。聚焦平面是由accPerspective()函数的第9个（最后一个）参数定义的，在此例中为0.5。模糊程度是由一个常数和x、y方向上的微移值（accPerspective()函数的第7个和第8个参数）的乘积决定的。确定这个常数值并不存在理论上的最优方法，需要不断地试验，直到实现满意的效果。注意，在示例程序10-7中，accPerspective()函数的第5个和第6个参数被设置为0.0，也就是关闭了场景的抗锯齿处理功能。

示例程序10-5 景深效果：dof.c

```
void init(void)
{
   GLfloat ambient [] ={0.0,0.0,0.0,1.0 };
   GLfloat diffuse [] ={1.0,1.0,1.0,1.0 };
   GLfloat specular [] ={1.0,1.0,1.0,1.0 };
   GLfloat position [] ={0.0,3.0,3.0,0.0 };
   GLfloat lmodel_ambient [] ={0.2,0.2,0.2,1.0 };
   GLfloat local_view [] ={0.0 };

   glLightfv(GL_LIGHT0,GL_AMBIENT,ambient);
   glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuse);
   glLightfv(GL_LIGHT0,GL_POSITION,position);

   glLightModelfv(GL_LIGHT_MODEL_AMBIENT,lmodel_ambient);
   glLightModelfv(GL_LIGHT_MODEL_LOCAL_VIEWER,local_view);

   glFrontFace(GL_CW);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_AUTO_NORMAL);
   glEnable(GL_NORMALIZE);
   glEnable(GL_DEPTH_TEST);

   glClearColor(0.0,0.0,0.0,0.0);
   glClearAccum(0.0,0.0,0.0,0.0);
/*make teapot display list */
   teapotList =glGenLists(1);
   glNewList(teapotList,GL_COMPILE);
   glutSolidTeapot(0.5);
   glEndList();
}

void renderTeapot(GLfloat x,GLfloat y,GLfloat z,
   GLfloat ambr,GLfloat ambg,GLfloat ambb,
   GLfloat difr,GLfloat difg,GLfloat difb,
   GLfloat specr,GLfloat specg,GLfloat specb,GLfloat shine)
```

```
{
   GLfloat mat [4];

   glPushMatrix();
   glTranslatef(x,y,z);
   mat [0] =ambr;mat [1] =ambg;mat [2] =ambb;mat [3] =1.0;
   glMaterialfv(GL_FRONT,GL_AMBIENT,mat);
   mat [0] =difr;mat [1] =difg;mat [2] =difb;
   glMaterialfv(GL_FRONT,GL_DIFFUSE,mat);
   mat [0] =specr;mat [1] =specg;mat [2] =specb;
   glMaterialfv(GL_FRONT,GL_SPECULAR,mat);
   glMaterialf(GL_FRONT,GL_SHININESS,shine*128.0);
   glCallList(teapotList);
   glPopMatrix();
}

void display(void)
{
   int jitter;
   GLint viewport [4];

   glGetIntegerv(GL_VIEWPORT,viewport);
   glClear(GL_ACCUM_BUFFER_BIT);

   for (jitter =0;jitter <8;jitter++){
      glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);
      accPerspective(45.0,
         (GLdouble)viewport [2]/(GLdouble)viewport [3] ,
         1.0,15.0,0.0,0.0,
         0.33*j8 [jitter].x,0.33*j8 [jitter].y,5.0);
/*    ruby,gold,silver,emerald,and cyan teapots */
      renderTeapot(-1.1,-0.5,-4.5,0.1745,0.01175,
                   0.01175,0.61424,0.04136,0.04136,
                   0.727811,0.626959,0.626959,0.6);
      renderTeapot(-0.5,-0.5,-5.0,0.24725,0.1995,
                   0.0745,0.75164,0.60648,0.22648,
                   0.628281,0.555802,0.366065,0.4);
      renderTeapot(0.2,-0.5,-5.5,0.19225,0.19225,
                  0.19225,0.50754,0.50754,0.50754,
                  0.508273,0.508273,0.508273,0.4);
      renderTeapot(1.0,-0.5,-6.0,0.0215,0.1745,0.0215,
                  0.07568,0.61424,0.07568,0.633,
                  0.727811,0.633,0.6);
      renderTeapot(1.8,-0.5,-6.5,0.0,0.1,0.06,0.0,
                  0.50980392,0.50980392,0.50196078,
                  0.50196078,0.50196078,.25);
      glAccum(GL_ACCUM,0.125);
   }
   glAccum(GL_RETURN,1.0);
   glFlush();
}
```

```
void reshape(int w,int h)
{
   glViewport(0,0,(GLsizei)w,(GLsizei)h);
}
/*Main Loop
 *Be certain you request an accumulation buffer.
 */
int main(int argc,char**argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB |
                       GLUT_ACCUM |GLUT_DEPTH);
   glutInitWindowSize(400,400);
   glutInitWindowPosition(100,100);
   glutCreateWindow(argv [0]);
   init();
   glutReshapeFunc(reshape);
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;
}
```

10.3.3 柔和阴影

为了累积多个光源所产生的柔和阴影，可以多次渲染场景，每次只打开一个光源，并将渲染结果累积起来。这个操作可以与使用空间微移的场景抗锯齿操作一起进行（有关绘制阴影的信息，请参阅第14.15节）。

10.3.4 微移

如果打算选取9个或16个样本对图像进行抗锯齿，我们可能会认为这些样本均匀地分布在图像中是最好的选择。令人吃惊的是，这种想法并不一定正确。事实上，有时候在相邻的像素上进行采样的效果更好。我们可能希望采样点在像素中心的周围均匀分布（规范化分布）。表10-5列出了在一些样本数量下的合理微移点集合。这张表的大部分例子都采用了均匀分布，并且都位于像素内部。

表10-5 样本微移值

样本数量	微移值
2	{0.25, 0.75}, {0.75, 0.25}
3	{0.5033922635, 0.8317967229}, {0.7806016275, 0.2504380877}, {0.2261828938, 0.4131553612}
4	{0.375, 0.25}, {0.125, 0.75}, {0.875, 0.25}, {0.625, 0.75}
5	{0.5, 0.5}, {0.3, 0.1}, {0.7, 0.9}, {0.9, 0.3}, {0.1, 0.7}
6	{0.4646464646, 0.4646464646}, {0.1313131313, 0.7979797979}, {0.5353535353, 0.8686868686}, {0.8686868686, 0.5353535353}, {0.7979797979, 0.1313131313}, {0.2020202020, 0.2020202020}
8	{0.5625, 0.4375}, {0.0625, 0.9375}, {0.3125, 0.6875}, {0.6875, 0.8125}, {0.8125, 0.1875}, {0.9375, 0.5625}, {0.4375, 0.0625}, {0.1875, 0.3125}
9	{0.5, 0.5}, {0.1666666666, 0.9444444444}, {0.5, 0.1666666666}, {0.5, 0.8333333333}, {0.1666666666, 0.2777777777}, {0.8333333333, 0.3888888888}, {0.1666666666, 0.6111111111}, {0.8333333333, 0.7222222222}, {0.8333333333, 0.0555555555}

（续）

样本数量	微移值
12	{0.4166666666, 0.625}，{0.9166666666, 0.875}，{0.25, 0.375}，{0.4166666666, 0.125}，{0.75, 0.125}，{0.0833333333, 0.125}，{0.75, 0.625}，{0.25, 0.875}，{0.5833333333. 0.375}，{0.9166666666, 0.375}，{0.0833333333, 0.625}，{0.58333333, 0.875}
16	{0.375, 0.4375}，{0.625, 0.0625}，{0.875, 0.1875}，{0.125, 0.0625}，{0.375, 0.6875}，{0.875, 0.4375}，{0.625, 0.5625}，{0.375, 0.9375}，{0.625, 0.3125}，{0.125, 0.5625}，{0.125, 0.8125}，{0.375, 0.1875}，{0.875, 0.9375}，{0.875, 0.6875}，{0.125, 0.3125}，{0.625, 0.8125}

10.4 帧缓冲区对象



高级话题

到目前为止，关于缓冲区的所有讨论都集中在窗口系统所提供的缓冲区上，也就是调用glutCreateWindow()的时候所请求的缓冲区（以及调用glutInitDisplayMode()所配置的缓冲区）。尽管可以只对这些缓冲区相当成功地使用任何技术，但是各种操作常常需要在缓冲区之间大量移动数据。这就是引入帧缓冲区对象（作为OpenGL 3.0的一部分）的原因所在，可以创建自己的帧缓冲区，并且使用它们附带的渲染缓冲区来最小化数据复制并优化性能。

对于执行离屏渲染、更新纹理图像和进行缓冲区ping-ponging（GPGPU中用到的一种数据传输技术），帧缓冲区对象都非常有用。

窗口系统所提供的帧缓冲区是你的图形服务器的显示系统可用的唯一的帧缓冲区，也就是说，它是你在自己的屏幕上可以看到的唯一的帧缓冲区。当窗口打开的时候所创建的缓冲区，其使用上也有所限制。相比较而言，应用程序创建的帧缓冲区不能在显示器上显示，它们只支持离屏渲染。

窗口系统提供的帧缓冲区和我们所创建的帧缓冲区的另一个不同之处在于，当窗口创建的时候，窗口系统管理的那些缓冲区就分配它们的缓冲区——颜色、深度、模板和累计等。当创建应用管理的帧缓冲区对象的时候，需要创建与所创建的帧缓冲区对象相关的其他渲染缓冲区。带有窗口系统提供的缓冲区的那些缓冲区，不能与应用程序创建的缓冲区对象相关联，反之亦然。

要分配一个应用程序生成的帧缓冲区对象的名字，需要调用glGenFramebuffers()，它将为帧缓冲区对象生成一个未使用的标识符。和OpenGL中的一些其他对象相比较（例如，纹理对象和显示列表），总是需要使用glGenFramebuffers()返回的一个名字。

void **glGenFramebuffers**(GLsize *n*, GLuint **ids*);

分配*n*个未使用的帧缓冲区对象名字，并且在*ids*中返回这些名字。

分配一个帧缓冲区对象名字并不会真正地创建帧缓冲区对象或者为其分配任何存储空间。这些任务都是通过调用glBindFramebuffer()来完成的。glBindFramebuffer()操作与我们在OpenGL中所见到的众多其他glBind*()函数工作方式相似。当初次针对一个特定的帧缓冲区调用它的时候，它会为该对象分配存储空间并初始化。任何后续的调用，将会把所提供的帧缓冲区对象名字作为一个活跃的对象绑定。

void **glBindFramebuffer**(GLenum *target*, GLuint *framebuffer*);

指定一个帧缓冲区对象用于读取或写入。当*target*是GL_DRAW_FRAMEBUFFER的时候，*framebuffer*指定了帧缓冲区渲染的目标。类似地，当*target*是GL_READ_FRAMEBUFFER的时候，

*framebuffer*指定了读取操作的源。

*framebuffer*必须要么是0（这把*target*绑定为默认值），要么是窗口系统提供的一个帧缓冲区，要么是调用**glGenFramebuffers()**所产生的一个帧缓冲区对象。

如果*framebuffer*不是0，也不是之前通过调用**glGenFramebuffers()**所产生的、却没有通过调用**glDeleteFramebuffers()**删除掉的一个有效的帧缓冲区对象，将会产生一个GL_INVALID_OPERATION错误。

与我们在OpenGL中遇到的所有其他对象一样，可以通过调用glDeleteFramebuffers()来释放应用程序分配的一个帧缓冲区。这个函数将会把该帧缓冲区对象的名字标记为未分配，并释放与该帧缓冲区相关的任何资源。

void **glDeleteFramebuffers**(GLsize *n*, const GLuint **ids*);

回收与*ids*所提供的名字相关的*n*个帧缓冲区对象。如果一个帧缓冲区对象是当前绑定的（如它的名字传递给了最近的**glBindFramebuffer()**调用）并且被删除了，帧缓冲区目标立即绑定到*id* 0（窗口系统提供的帧缓冲区），并且帧缓冲区对象将释放。

glDeleteFramebuffers()不会产生错误。未使用的名字或0会被直接忽略。

为了完整起见，可以通过调用glIsFramebuffer()来判定一个特定的无符号整数是否是应用程序分配的一个帧缓冲区对象。

GLboolean **glIsFramebuffer**(GLuint *framebuffer*);

如果*framebuffer*是**glGenFramebuffers()**返回的一个帧缓冲区的名字，将返回GL_TRUE。如果**framebuffer**是0（窗口系统默认帧缓冲区），或者是一个未分配的或通过调用**glDeleteFramebuffers()**删除的值，返回GL_FALSE。

一旦创建了一个帧缓冲区对象，仍然不能用它做太多事情。需要提供一个地方，作为绘制的目的地和读取操作的源，这些地方叫做帧缓冲区附加（framebuffer attachment）。我们将在介绍完渲染缓冲区之后更详细地讨论它，渲染缓冲区是可以附加到一个帧缓冲区对象的类型之一。

10.4.1 渲染缓冲区

渲染缓冲区是OpenGL管理的有效内存，其中包含了格式化的图像数据。假设图像缓冲区的格式与OpenGL期望向其中渲染的格式一致（例如，不能向深度缓冲区中渲染颜色），一旦渲染缓冲区附加到一个帧缓冲区对象，它所保存的数据就变得有意义了。

与OpenGL中的很多其他缓冲区一样，分配和删除缓冲区的过程也和我们前面见到的类似。要创建一个新的渲染缓冲区，可以调用glGenRenderbuffers()。

void **glGenRenderbuffers**(GLsizei *n*, GLuint **ids*);

分配*n*个未使用的渲染缓冲区对象名，并且把这些名字返回到*ids*中。这些名字一直是未使用的，直到调用**glBindRenderbuffer()**来绑定。

类似地，调用glDeleteRenderbuffers()将释放和一个渲染缓冲区相关的存储空间。

void **glDeleteRenderbuffers**(GLsizei *n*, const GLuint **ids*);

回收和*ids*所提供的名字相关的*n*个渲染缓冲区。如果渲染缓冲区之一是当前绑定的并且传递给**glDeleteRenderbuffers()**了，一个到0的绑定会替代在当前帧缓冲区附加点的绑定，此外，将会释放

该渲染缓冲区。

glDeleteRenderbuffers()不会产生错误。未使用的名字或0会被直接忽略。

类似地，可以调用glIsRenderbuffer()来判断一个名字是否表示一个有效的渲染缓冲区。

void **glIsRenderbuffer**(GLuint *renderbuffer*);

如果*renderbuffer*是**glGenRenderbuffers()**返回的一个渲染缓冲区的名字，返回GL_TRUE。如果renderbuffer是0（窗口系统默认帧缓冲区），或者是一个未分配的或一个已经通过调用**glDeleteRenderbuffers()**删除的值，返回GL_FALSE。

绑定一个帧缓冲区对象以便可以修改其状态，类似地，调用glBindRenderbuffer()来影响一个渲染缓冲区的创建并修改与它相关的状态，包括它所包含的图像数据的格式。

void **glBindRenderbuffer**(GLenum *target*, GLuint *renderbuffer*);

创建一个渲染缓冲区，并将其与名字*renderbuffer*关联起来。*target*必须是GL_RENDERBUFFER。*renderbuffer*必须要么是0（它删除任何绑定的渲染缓冲区），要么是调用**glGenRenderbuffers()**所产生的一个名字，否则将产生一个GL_INVALID_OPERATION错误。

创建渲染缓冲区存储

当初次针对一个未使用的渲染缓冲区名字调用glBindRenderbuffer()的时候，OpenGL服务器创建一个渲染缓冲区，其所有状态信息都设置为默认值。在这个配置过程中，并没有分配存储空间来存储图像数据。我们需要分配存储空间并指定其图像格式，然后才可以把渲染缓冲区附加到一个帧缓冲区并向其中渲染。这通过调用glRenderbufferStorage()或glRenderbufferStorageMultisample()来完成。

void **glRenderbufferStorage**(GLenum *target*, GLenum *internalformat*,
GLsizei *width*, GLsizei *height*);

void **glRenderbufferStorageMultisample**(GLenum *target*,
GLsizei *samples*, GLenum *internalformat*, GLsizei *width*,
GLsizei *height*);

为绑定的渲染缓冲区分配存储空间用来存储图像数据。target必须是GL_RENDERBUFFER。对于可以渲染颜色的缓冲区，internalformat必须是如下之一：GL_RED、GL_R8、GL_R16、GL_RG、GL_RG8、GL_RG16、GL_RGB、GL_R3_G3_B2、GL_RGB4、GL_RGB5、GL_RGB8、GL_RGB10、GL_RGB12、GL_RGB16、GL_RGBA、GL_RGBA2、GL_RGBA4、GL_RGB5_A1、GL_RGBA8、GL_RGB10_A2、GL_RGBA12、GL_RGBA16、GL_SRGB、GL_SRGB8、GL_SRGB_ALPHA、GL_SRGB8_ALPHA8、GL_R16F、GL_R32F、GL_RG16F、GL_RG32F、GL_RGB16F、GL_RGB32F、GL_RGBA16F、GL_RGBA32F、GL_R11F_G11F_B10F、GL_RGB9_E5、GL_R8I、GL_R8UI、GL_R16I、GL_R16UI、GL_R32I、GL_R32UI、GL_RG8I、GL_RG8UI、GL_RG16I、GL_RG16UI、GL_RG32I、GL_RG32UI、GL_RGB8I、GL_RGB8UI、GL_RGB16I、GL_RGB16UI、GL_RGB32I、GL_RGB32UI、GL_RGBA8I、GL_RGBA8UI、GL_RGBA16I、GL_RGBA16UI或GL_RGBA32I。OpenGL 3.1添加了如下的格式：GL_R8_SNORM、GL_R16_SNORM、GL_RG8_SNORM、GL_RG16_SNORM、GL_RGB8_SNORM、GL_RGB16_SNORM、GL_RGBA8_SNORM或GL_RGBA16_SNORM。

要把一个渲染缓冲区用做深度缓冲区，它必须是深度可渲染的，这通过把*internalformat*设置为如下之一来指定：GL_DEPTH_COMPONENT、GL_DEPTH_COMPONENT16、GL_DEPTH_COMPONENT32、GL_DEPTH_COMPONENT32或GL_DEPTH_COMPONENT32F。

对于专门用作模板缓冲区的情况，*internalformat*应该指定为如下之一：GL_STENCIL_INDEX、GL_STENCIL_INDEX1、GL_STENCIL_INDEX4、GL_STENCIL_INDEX8或GL_STENCIL_INDEX16。

对于包装的深度-模板存储，*internalformat*必须是GL_DEPTH_STENCIL，它允许渲染缓冲区作为深度缓冲区、模板缓冲区或组合的深度-模板附加点来附加。

*width*和*height*以像素为单位指定了渲染缓冲区的大小，*samples*指定了每个像素多重采样样本的数目。在**glRenderbufferStorageMultisample()**调用中把*samples*设置为0，就等同于调用**glRenderbufferStorage()**。

如果*width*和*height*大于查询GL_MAX_RENDERBUFFER_SIZE时候的返回值，或者*samples*大于查询GL_MAX_SAMPLES时候的返回值，将产生一个GL_INVALID_VALUE。如果*internalformat*是一个带符号的或无符号的整数格式（例如，标记中包含一个“I”或“UI”的格式），并且*samples*不为0，而且该实现不支持多重采样的整数缓冲区，会产生一个GL_INVALID_OPERATION。最后，如果渲染缓冲区大小和格式的组合超过了可用来分配的内存，会产生一个GL_OUT_OF_MEMORY错误。

示例程序10-6 创建一个RGBA颜色渲染缓冲区：fbo.c

```
glGenRenderbuffers(1,&color);
glBindRenderbuffer(GL_RENDERBUFFER,color);
glRenderbufferStorage(GL_RENDERBUFFER,GL_RGBA,256,256);
```

一旦为渲染缓冲区创建了存储空间，需要将其附加到一个帧缓冲区对象，然后才能够向其中渲染。

帧缓冲区附加

渲染的时候，可以把渲染结果发送到如下位置：

- 发送到颜色缓冲区以创建图像，甚至发送到多个颜色缓冲区（如果使用多个渲染目标，参见第15.11节中的“特殊的输出值”）。
- 发送到深度缓冲区以存储遮挡信息。
- 用来存储基于像素掩码的模板缓冲区，以控制渲染。

这些缓冲区中的每一个都表示为一个帧缓冲区附加，我们可以向其附加合适的图像缓冲区，以便稍后向其中渲染或者从其中读取。可能的帧缓冲区附加点如表10-6所示。

表10-6 帧缓冲区附加

附加名称	说 明
GL_COLOR_ATTACHMENTi	第i个颜色缓冲区。i的范围从0（默认的颜色缓冲区）到GL_MAX_COLOR_ATTACHMENTS-1
GL_DEPTH_ATTACHMENT	深度缓冲区
GL_STENCIL_ATTACHMENT	模板缓冲区
GL_DEPTH_STENCIL_ATTACHMENT	包装的深度-模板缓冲区的一个特殊附加（这要求渲染缓冲区已经按照一个GL_DEPTH_STENCIL像素格式分配）

当前，有两种类型的渲染表面可以与这些附加之一关联起来：渲染缓冲区和纹理图像的一层。

我们首先介绍把一个渲染缓冲区附加到帧缓冲区对象，这通过调用glFramebufferRenderbuffer()来实现。

void **glFramebufferRenderbuffer**(GLenum *target*, GLenum *attachment*,
GLenum *renderbuffertarget*, GLuint *renderbuffer*);

把*renderbuffer*和当前绑定的帧缓冲区对象的附加联系起来。*target*必须是GL_READ_FRAMEBUFFER、GL_DRAW_FRAMEBUFFER或GL_FRAMEBUFFER(它等同于GL_DRAW_FRAMEBUFFER)。

*attachment*是如下之一：GL_COLOR_ATTACHMENTi、GL_DEPTH_ATTACHMENT、GL_STENCIL_ATTACHMENT或GL_DEPTH_STENCIL_ATTACHMENT。

*renderbuffertarget*必须是GL_RENDERBUFFER，并且*renderbuffer*必须是0（这会删除附加点的任何渲染缓冲区附加），或者是**glGenRenderbuffers()**所返回的一个渲染缓冲区的名字，否则将产生一个GL_INVALID_OPERATION错误。

在示例程序10-7中，我们创建并附加了两个渲染缓冲区：一个用于颜色，另一个用于深度。然后，我们处理渲染，最后，把结果复制回窗口系统提供的帧缓冲区以显示结果。可以使用这种技术为一个视频的离屏渲染生成帧，这样，就不必担心由于窗口重叠或者某人调整窗口大小并中断渲染而导致可见的帧缓冲区被破坏。

要记住重要的一点是，在渲染之前可能需要为每个帧缓冲区设置视口，特别是应用程序定义的帧缓冲区和窗口系统提供的帧缓冲区的大小不同的时候。

示例程序10-7 附加一个渲染缓冲区以便渲染：fbo.c

```
enum {Color,Depth,NumRenderbuffers };
GLuint framebuffer,renderbuffer [NumRenderbuffers]

void
init()
{
   glGenRenderbuffers(NumRenderbuffers,renderbuffer );
   glBindRenderbuffer(GL_RENDERBUFFER,renderbuffer [Color]);
   glRenderbufferStorage(GL_RENDERBUFFER,GL_RGBA,256,256 );
   glBindRenderbuffer(GL_RENDERBUFFER,renderbuffer [Depth] );
   glRenderbufferStorage(GL_RENDERBUFFER,GL_DEPTH_COMPONENT24,
       256,256 );

   glGenFramebuffers(1,&framebuffer );
   glBindFramebuffer(GL_DRAW_FRAMEBUFFER,framebuffer );
   glFramebufferRenderbuffer(GL_DRAW_FRAMEBUFFER,
      GL_COLOR_ATTACHMENT0,GL_RENDERBUFFER,
      renderbuffer [Color] );
   glFramebufferRenderbuffer(GL_DRAW_FRAMEBUFFER,
      GL_DEPTH_ATTACHMENT,GL_RENDERBUFFER,
      renderbuffer [Depth] );

   glEnable(GL_DEPTH_TEST );
}

void
display()
{
   /*Prepare to render into the renderbuffer */
```

```
    glBindFramebuffer(GL_DRAW_FRAMEBUFFER,framebuffer );
    glViewport(0,0,256,256 );

    /*Render into renderbuffer */
    glClearColor(1.0,0.0,0.0,1.0 );
    glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT );
    /*Do other rendering */

    /*Set up to read from the renderbuffer and draw to
    **window-system framebuffer */
    glBindFramebuffer(GL_READ_FRAMEBUFFER,framebuffer );
    glBindFramebuffer(GL_DRAW_FRAMEBUFFER,0 );
    glViewport(0,0,windowWidth,windowHeight);
    glClearColor(0.0,0.0,1.0,1.0 );
    glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT );

    /*Do the copy */
    glBlitFramebuffer(0,0,255,255,0,0,255,255,
       GL_COLOR_BUFFER_BIT,GL_NEAREST );
    glutSwapBuffers();
}
```

帧缓冲对象的另一个常见用法是动态地更新纹理。你可能想要表示表面外观的变化（例如，游戏中墙上的弹孔），或者在进行类似GPGPU的计算，要更新一个查找表中的值。在这些情况下，和渲染缓冲区相对比，把纹理图像的一个层绑定为帧缓冲附加。渲染之后，纹理图像可以从帧缓冲区解除，以便它可以用于后续的渲染。

注意：可以从一个同时为写入而绑定为帧缓冲区附加的纹理中读取，这种情况叫做帧缓冲区渲染循环（framebuffer rendering loop）。对于两种操作来说，结果都是未定义的。也就是说，取样绑定的纹理图像所返回的值，也是绑定的时候写入到纹理层的值，这可能会出错。

---

void **glFramebufferTexture1D**(GLenum *target*, GLenum *attachment*,
GLenum *texturetarget*, GLuint *texture*, GLint *level*);
void **glFramebufferTexture2D**(GLenum *target*, GLenum *attachment*,
GLenum *texturetarget*, GLuint *texture*, GLint *level*);
void **glFramebufferTexture3D**(GLenum *target*, GLenum *attachment*,
GLenum *texturetarget*, GLuint *texture*, GLint *level*,
GLint *layer*);

把纹理对象的一层作为渲染附加添加到一个帧缓冲区对象。*target*必须是GL_READ_FRAMEBUFFER、GL_DRAW_FRAMEBUFFER或GL_FRAMEBUFFER（等同于GL_DRAW_FRAMEBUFFER）之一。*attachment*必须是以下附加点之一：GL_COLOR_ATTACHMENTi、GL_DEPTH_ATTACHMENT、GL_STENCIL_ATTACHMENT或GL_DEPTH_STENCIL_ATTACHMENT（在这种情况下，纹理的内部格式必须是GL_DEPTH_STENCIL）。

对于**glFramebufferTexture1D**()，如果*texture*不为0，*texturetarget*必须是GL_TEXTURE_1D。对于**glFramebufferTexture2D**()，*texturetarget*必须是GL_TEXTURE_2D、GL_TEXTURE_RECTANGLE、GL_TEXTURE_CUBE_MAP_POSITIVE_X、GL_TEXTURE_CUBE_MAP_POSITIVE_Y、GL_TEXTURE_CUBE_MAP_POSITIVE_Z、GL_TEXTURE_CUBE_MAP_NEGATIVE_X、

GL_TEXTURE_CUBE_MAP_NEGATIVE_Y或 GL_TEXTURE_CUBE_MAP_NEGATIVE_Z。对于**glFramebufferTexture3D()**，*texturetarget*必须是GL_TEXTURE_3D。

如果*texture*为0，表示绑定到附加的任何纹理都释放了，并且没有对附加进行后续的绑定。在这种情况下，*texturetarget*、*level*和*layer*都忽略了。

如果*texture*不等于0，它必须是一个已有的纹理对象（通过**glGenTextures()**创建）的名字，而*texturetarget*匹配与纹理对象相关的纹理类型（例如，GL_TEXTURE_1D等），或者如果纹理是一个立方图，那么*texturetarget*必须是立方图纹理表面目标之一：GL_TEXTURE_CUBE_MAP_POSITIVE_X、GL_TEXTURE_CUBE_MAP_POSITIVE_Y、GL_TEXTURE_CUBE_MAP_POSITIVE_Z、GL_TEXTURE_CUBE_MAP_NEGATIVE_X、GL_TEXTURE_CUBE_MAP_NEGATIVE_Y或GL_TEXTURE_CUBE_MAP_NEGATIVE_Z，否则，会产生一个GL_INVALID_OPERATION错误。

*level*表示作为渲染目标附加的相关纹理图像的mipmap层，并且，对于三维纹理，*layer*表示所使用纹理的层。如果*texturetarget*是GL_TEXTURE_RECTANGLE（来自OpenGL 3.1），那么*level*必须是0。

和前面的例子类似，示例程序10-8展示了这样的一个过程：动态更新纹理，在纹理更新完成之后使用它，然后用它渲染。

示例程序10-8 添加一个纹理层作为帧缓冲区附加: fbotexture.c

```
GLuint framebuffer,texture;

void
init()
{
   GLuint renderbuffer;

   /*Create an empty texture */
   glGenTextures(1,&texture );
   glBindTexture(GL_TEXTURE_2D,texture );
   glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA8,TexWidth,
      TexHeight,0,GL_RGBA,GL_UNSIGNED_BYTE,NULL );

   /*Create a depth buffer for our framebuffer */
   glGenRenderbuffers(1,&renderbuffer );
   glBindRenderbuffer(GL_RENDERBUFFER,renderbuffer );
   glRenderbufferStorage(GL_RENDERBUFFER,GL_DEPTH_COMPONENT24,
      TexWidth,TexHeight );

   /*Attach the texture and depth buffer to the framebuffer */
   glGenFramebuffers(1,&framebuffer );
   glBindFramebuffer(GL_DRAW_FRAMEBUFFER,framebuffer );
   glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER,
      GL_COLOR_ATTACHMENT0,GL_TEXTURE_2D,texture,0 );
   glFramebufferRenderbuffer(GL_DRAW_FRAMEBUFFER,
      GL_DEPTH_ATTACHMENT,GL_RENDERBUFFER,renderbuffer );

   glEnable(GL_DEPTH_TEST );
}
```

```
void
display()
{
   /*Render into the renderbuffer */
   glBindFramebuffer(GL_DRAW_FRAMEBUFFER,framebuffer );
   glViewport(0,0,TexWidth,TexHeight );
   glClearColor(1.0,0.0,1.0,1.0 );
   glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT );
   /*Do other rendering */
   /*Generate mipmaps of our texture */

   glGenerateMipmap(GL_TEXTURE_2D );

   /*Bind to the window-system framebuffer,unbinding from
   **the texture,which we can use to texture other objects */
   glBindFramebuffer(GL_FRAMEBUFFER,0 );
   glViewport(0,0,windowWidth,windowHeight );
   glClearColor(0.0,0.0,1.0,1.0 );
   glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT );

   /*Render using the texture */
   glEnable(GL_TEXTURE_2D );
   ...

   glutSwapBuffers();
}
```

void **glFramebufferTextureLayer**(GLenum *target*, GLenum *attachment*,
GLuint *texture*, GLint *level*, GLint *layer*);

附加一个三维纹理的一层或者一个一维数组纹理或二维数组纹理作为一个帧缓冲区附加，类似于**glFramebufferTexture3D()**的方式。

*target*必须是如下之一：GL_READ_FRAMEBUFFER、GL_DRAW_FRAMEBUFFER或GL_FRAMEBUFFER（等同于GL_DRAW_FRAMEBUFFER）。*attachment*必须是如下之一：GL_COLOR_ATTACHMENT*i*、GL_DEPTH_ATTACHMENT、GL_STENCIL_ATTACHMENT或GL_DEPTH_STENCIL_ATTACHMENT。

*texture*要么是0，表示该附加的当前绑定应该释放；或者是一个纹理对象名称（作为**glGenTextures()**的返回值）。*level*表示纹理对象的mipmap层，*layer*表示纹理的哪一层应该作为附加绑定。

帧缓冲区完整性

由于纹理和缓冲区格式以及帧缓冲区附加之间有无数种组合，当使用应用定义的帧缓冲区对象的时候，可能出现各种阻碍完成渲染的情况。在修改了到一个帧缓冲区对象的附加之后，最好通过调用glCheckFramebufferStatus()来检查帧缓冲区的状态。

GLenum **glCheckFramebufferStatus**(GLenum *target*);

返回表10-7列出的帧缓冲区完整性状态之一。*target*必须是GL_READ_FRAMEBUFFER、GL_DRAW_FRAMEBUFFER或GL_FRAMEBUFFER（等同于GL_DRAW_FRAMEBUFFER）。

如果glCheckFramebufferStatus()产生一个错误，返回0。

表10-7中列出了表示各种违反帧缓冲区配置规则情况的错误。

表10-7 glCheckFramebufferStatus()返回的错误

帧缓冲区完整性状态枚举值	说 明
GL_FRAMEBUFFER_COMPLETE	帧缓冲区及其附加匹配所需的渲染或读取状态
GL_FRAMEBUFFER_UNDEFINED	绑定的帧缓冲区指定为默认帧缓冲区（例如，glBindFramebuffer()和0调用指定为帧缓冲区），并且，默认的帧缓冲区不存在
GL_FRAMEBUFFER_INCOMPLETE_ATTACHMENT	绑定的帧缓冲区所必须的附加未初始化
GL_FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT	没有图像（例如，纹理层或渲染缓冲区）附加到帧缓冲区
GL_FRAMEBUFFER_INCOMPLETE_DRAW_BUFFER	每个绘制缓冲区（例如，glDrawBuffers()所指定的GL_DRAW_BUFFERi）拥有一个附加
GL_FRAMEBUFFER_INCOMPLETE_READ_BUFFER	glReadBuffer()所指定的缓冲区存在一个附加
GL_FRAMEBUFFER_UNSUPPORTED	附加到帧缓冲区对象的图像的组合与OpenGL实现的需求不兼容
GL_FRAMEBUFFER_INCOMPLETE_MULTISAMPLE	整个帧缓冲区附加中的所有图像的采样数目不匹配

在列出的错误中，GL_FRAMEBUFFER_UNSUPPORTED是和实现密切相关的，并且可能是最难于解决的一个。

10.4.2 复制像素矩形

尽管从OpenGL 1.0开始，glCopyPixels()就已经是复制像素块的默认函数了，但是随着OpenGL扩展其渲染工具，还是需要更多有用的像素复制程序。下面介绍的glBlitFramebuffer()，以单个的、增强的调用方式，包含了glCopyPixels()和glPixelZoom()的操作。glBlitFramebuffer()允许在复制操作中更大的像素过滤，和纹理映射的方式有很大的相同之处（实际上，在复制过程中，GL_NEAREST和GL_LINEAR使用同样的过滤操作）。此外，这个程序对多重采样缓冲区也有意义，并且支持在不同帧缓冲区之间复制（由帧缓冲区对象控制）。

void **glBlitFramebuffer**(GLint *srcX0*, GLint *srcY0*,
GLint *srcX1*, GLint *srcY1*, GLint *dstX0*, GLint *dstY0*,
GLint *dstX1*, GLint *dstY1*, GLbitfield *buffers*,
GLenum *filter*);

从读取帧缓冲区的一个区域、复制像素值的一个矩形到绘制缓冲区的另一个区域，在这一过程中，可能潜在地对像素调整大小、反转、转换和过滤。*srcX0*, *srcY0*, *srcX1*, *srcY1*表示像素来自的源区域，写入的矩形区域由*dstX0*, *dstY0*, *dstX1*, *dstY1*指定。*buffer*是一个逻辑运算符，或者是GL_COLOR_BUFFER_BIT、GL_DEPTH_BUFFER_BIT或GL_STENCIL_BUFFER_BIT之一，它们分别表示复制应该发生在哪个缓冲区中。最后，*filter*指定了两个矩形区域具有不同大小时候的插值方法，并且它必须是GL_NEAREST或GL_LINEAR之一；如果两个区域具有相同的大小，就不应用过滤。

如果有多个颜色绘制缓冲区（参见第15.11节的“渲染到多个输出缓冲区”），每个缓冲区接受源区域的一个副本。

如果srcX1 < srcX0或dstX1 < dstX0，图像在水平方向上翻转。类似地，如果srcY1 < srcY0或

dstY1 < dstY0，图像在垂直方向上翻转。然而，如果源大小和目标大小在同样的方向上都是相反的，那就不翻转。

如果源缓冲区和目标缓冲区具有不同的格式，在大多数情况下，要转换像素值。然而，如果读取颜色缓冲区是一个浮点格式，并且任何写入颜色缓冲区不是这一格式；或者反之亦然；如果读取缓冲区是一个带符号（无符号的）整数格式，并且不是所有绘制缓冲区都是带符号（无符号的）整数值，这一调用将会产生一个GL_INVALID_OPERATION，并且将不会复制像素。

多重采样缓冲区对于复制像素也有一个影响。如果源缓冲区是多重采样的，并且目标缓冲区不是，样本将针对目标缓冲区解析为单个像素。相反，如果源缓冲区不是多重采样的，并且目标缓冲区是，源像素数据将针对每个样本复制。最终，如果两个缓冲区都是多重采样的，并且每个缓冲区的样本数目都相同，样本会不做任何修改地复制。然而，如果两个缓冲区拥有不同数目的样本，就不会复制像素，并且产生一个GL_INVALID_OPERATION错误。

如果两个缓冲区设置了允许的位以外的其他位，或者如果*filter*不是GL_LINEAR或GL_NEAREST，将产生一个GL_INVALID_VALUE错误。

第11章　分格化和二次方程表面

本章目标

- 把凹形的填充多边形分格化为多个凸多边形，以便用标准的OpenGL函数对它们进行渲染。
- 使用OpenGL工具函数库创建二次方程表面，对球体和圆柱体的表面进行建模，并对圆盘或不完整的圆盘（弧形物体）进行分格化。

注意：在OpenGL 3.1中，本章所介绍的一些技术和函数（尤其是和二次方程表面相关的内容）可能将会废弃。尽管很多这样的功能可以在GLU库中找到，但是它们依赖于OpenGL中删除的那些功能。

OpenGL函数库（GL）面向低层操作，不管是流水线式的软件操作还是使用了硬件加速的操作。OpenGL工具函数库（GLU）对OpenGL函数库进行了补充，支持一些高层操作。本书的其他章节已经介绍了许多GLU操作。mipmap（gluBuild*DMipmaps()）和图像缩放（gluScaleImage()）作为纹理贴图的一部分在第9章进行了讨论。第3章描述了一些GLU矩阵变换函数（gluOrtho2D()、gluPerspective()、gluLookAt()、gluProject()、gluUnProject()和gluUnProject4()）。第13章将讨论gluPickMatrix()函数的应用。第12章将解释建立在OpenGL求值器基础之上的GLU NURBS工具。本章介绍剩余的两个GLU主题：多边形分格化和二次方程表面。

由于性能的原因，基本的OpenGL只能渲染凸多边形，但是GLU包含了一些函数，把凹多边形分格化为多个凸多边形，然后再用基本的OpenGL函数进行渲染。基本的OpenGL只能对简单的图元进行操作，例如点、直线和填充多边形。但是，GLU可以创建高层的物体，例如球体、圆柱体和锥体的表面等。

本章主要由下面两节组成：

- 多边形分格化：解释如何把凹多边形分格化为容易进行渲染的凸多边形。
- 二次方程表面：渲染球体、圆柱体和圆盘：描述了如何生成球体、圆柱体、圆和弧形，包括诸如表面法线和纹理坐标这样的数据。

11.1　多边形分格化

正如第2.2节所述，OpenGL只能直接显示简单的凸多边形。所谓简单多边形，是指多边形的边只在顶点处相交，没有重复的顶点，并且任何顶点都只有两条边相遇。如果应用程序需要显示凹多边形、中间有洞的多边形或者具有相交边的多边形，这些多边形在显示之前首先必须分解为简单的凸多边形。这种划分方法称为多边形分格化（tessellation），GLU提供了一组函数来执行分格化操作。这些函数所接受的参数就是描述这些难以渲染的复杂多边形的轮廓线，并返回一些三角形、三角形网、三角形扇以及直线的组合。

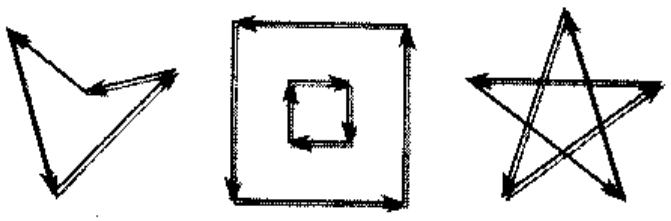


图11-1　需要进行分格化的轮廓线

图11-1显示了一些需要进行分格化的多边

形轮廓线，从左到右分别是凹多边形、中间有洞的多边形以及自身相交的多边形。

如果一个多边形需要进行分格化，需要执行如下几个典型步骤：

1) 用gluNewTess()函数创建一个新的分格化对象。

2) 调用几次gluTessCallback()函数，注册一些回调函数，在分格化时执行必要的操作。其中，最为复杂的情况是当分格化算法检测到多边形存在相交并且必须调用在GLU_TESS_COMBINE回调函数中所注册的函数的时候。

3) 调用gluTessProperty()函数指定分格化属性。其中最重要的属性是环绕规则，它确定了多边形的哪些区域应该填充，哪些区域不应该着色。

4) 通过指定一个或多个闭合多边形的轮廓线来渲染经过分格化的多边形。如果物体的数据是静态的，可以把经过分格化的多边形放在显示列表中（如果并不需要反复重新进行分格化计算，使用显示列表可以提高效率）。

5) 如果需要对其他物体进行分格化，可以复用原来的分格化对象。如果已经完成了对分格化对象的操作，可以使用gluDeleteTess()函数删除它。

注意：本章所描述的分格化工具是在GLU 1.2版本中引入的。如果读者使用的是更早版本的GLU，必须使用第11.1.7节介绍的函数。为了查询自己使用的是哪个版本的GLU，如果读者使用的GLU提供了gluGetString(GLU_VERSION)函数，可以调用这个函数，它会返回一个包含GLU版本号的字符串。如果读者使用的GLU没有提供这个函数，那么它就是GLU 1.0，因为这个版本尚未提供gluGetString()函数。

11.1.1 创建分格化对象

对一个复杂的多边形进行描述和分格化时，它具有一些相关的数据，例如顶点、边以及回调函数等。所有这些数据都连接到一个分格化对象中。为了执行分格化操作，首先需要使用gluNewTess()函数创建一个分格化对象。

GLUtesselator* **gluNewTess**(void);

创建一个新的分格化对象，并返回一个指向它的指针。如果创建失败，这个函数将返回一个NULL指针。

可以在所有的分格化任务中复用同一个分格化对象。应用程序之所以需要这个对象的原因是库函数可能需要执行它们自己的分格化操作，而这些操作应该能够在不涉及应用程序分格化任务的情况下完成。如果想在不同的分格化任务中使用不同的回调函数集合，也可以使用多个分格化对象。在一般情况下，应用程序总是分配一个分格化对象，并把它用于所有的分格化任务。释放分格化对象并不是一件急迫的事情，因为它所占据的内存非常少。当然，在完成分格化任务之后释放分格化对象肯定也是正确的做法。

11.1.2 分格化回调函数

在创建了分格化对象之后，必须提供一系列的回调函数，以便在执行分格化任务的适当时候调用它们。在指定了回调函数之后，就可以使用GLU函数描述一个或多个多边形的轮廓线了。当轮廓线的描述完成之后，分格化工具就会在必要的时候调用这些回调函数。

如果省略了任何函数，它们就不会在分格化操作时被调用，它们可能返回的任何信息也随之丢失。可以在单个gluTessCallback()调用中指定所有的回调函数。

void **gluTessCallback**(GLUtesselator **tessobj*, GLenum *type*, void (**fn*)());

把回调函数*fn*与分格化对象*tessobj*相关联。回调函数的类型是由参数*type*决定的，它可以是GLU_TESS_BEGIN、GLU_TESS_BEGIN_DATA、GLU_TESS_EDGE_FLAG、GLU_TESS_EDGE_FLAG_DATA、GLU_TESS_VERTEX、GLU_TESS_VERTEX_DATA、GLU_TESS_END、GLU_TESS_END_DATA、GLU_TESS_COMBINE、GLU_TESS_COMBINE_DATA、GLU_TESS_ERROR或GLU_TESS_ERROR_DATA。这12个回调函数具有以下原型：

GLU_TESS_BEGIN	void **begin**(GLenum *type*);
GLU_TESS_BEGIN_DATA	void **begin**(GLenum *type*, void **user_data*);
GLU_TESS_EDGE_FLAG	void **edgeFlag**(GLboolean *flag*);
GLU_TESS_EDGE_FLAG_DATA	void **edgeFlag**(GLboolean *flag*, void **user_data*);
GLU_TESS_VERTEX	void **vertex**(void **vertex_data*);
GLU_TESS_VERTEX_DATA	void **vertex**(void **vertex_data*, void **user_data*);
GLU_TESS_END	void **end**(void);
GLU_TESS_END_DATA	void **end**(void **user_data*);
GLU_TESS_COMBINE	void **combine**(GLdouble *coords*[3], void **vertex_data*[4], GLfloat *weight*[4], void ***outData*);
GLU_TESS_COMBINE_DATA	void **combine**(GLdouble *coords*[3], void **vertex_data*[4], GLfloat *weight*[4], void ***outData*, void **user_data*);
GLU_TESS_ERROR	void **error**(GLenum *errno*);
GLU_TESS_ERROR_DATA	void **error**(GLenum *errno*, void **user_data*);

为了更改回调函数，只要简单地调用gluTessCallback()，并以新的回调函数作为参数。为了删除一个原有的回调函数却并不用新的回调函数替换它，可以在gluTessCallback()函数中为相应的回调函数传递一个NULL指针。

随着分格化操作的进行，这些回调函数会在适当的时候被调用，其方式类似于使用OpenGL函数glBegin()、glEdgeFlag*()、glVertex*()和glEnd()（有关glEdgeFlag*()函数的更多信息，请参阅第2.4.3节）。combine回调函数用于在边相交的地方创建新的顶点。error回调函数则是在分格化过程遇到错误的时候被调用的。

在每个分格化对象被创建之后，GLU_TESS_BEGIN回调函数就会被调用，并使用下面这4个参数值之一：GL_TRIANGLE_FAN、GL_TRIANGLE_STRIP、GL_TRIANGLES或GL_LINE_LOOP。当分格化对象对多边形进行分解时，分格化算法决定了使用哪种类型的三角形图元的效率最高（如果启用了GLU_TESS_BOUNDARY_ONLY属性，就使用GL_LINE_LOOP进行渲染）。

由于边界标志在三角形扇或三角形带中并无意义，因此如果有一个与GLU_TESS_ EDGE_FLAG相关联的回调函数启用了边界标志，那么GLU_TESS_BEGIN回调函数只能用于GL_TRIANGLES。GLU_TESS_EDGE_FLAG回调函数的工作方式与OpenGL的glEdgeFlag*()函数完全相同。

在GLU_TESS_BEGIN回调函数被调用之后，并在GLU_TESS_END回调函数被调用之前，GLU_TESS_EDGE_FLAG和GLU_TESS_VERTEX回调函数的有些组合将会被调用（通常是通过调用gluTessVertex()函数，在第11.1.4节中描述）。OpenGL对相关的边界标志和顶点的解释方式与它们在

glBegin()和glEnd()之间时相同。

如果在分格化过程中出现了错误，error回调函数就会传递一个GLU错误号。可以使用gluErrorString()函数获得一个描述错误信息的字符串（关于这个函数的更多信息，请参阅第11.1.7节）。

示例程序11-1显示了tess.c程序的一部分，它创建了一个分格化对象，并注册了一些回调函数。

示例程序11-1 注册分格化回调函数：tess.c

```
#ifndef CALLBACK
#define CALLBACK
#endif
/*a portion of init()*/

tobj =gluNewTess();
gluTessCallback(tobj,GLU_TESS_VERTEX,glVertex3dv);
gluTessCallback(tobj,GLU_TESS_BEGIN,beginCallback);
gluTessCallback(tobj,GLU_TESS_END,endCallback);
gluTessCallback(tobj,GLU_TESS_ERROR,errorCallback);

/*the callback routines registered by gluTessCallback()*/
void CALLBACK beginCallback(GLenum which)
{
   glBegin(which);
}
void CALLBACK endCallback(void)
{
   glEnd();
}
void CALLBACK errorCallback(GLenum errorCode)
{
   const GLubyte *estring;

   estring =gluErrorString(errorCode);
   fprintf(stderr,"Tessellation Error:%s \n",estring);
   exit(0);
}
```

注意：回调函数的类型转换是非常严格的，尤其是需要程序在Microsoft Windows和Linux系统中都能以相同的方式运行时。为了在Microsoft Windows上运行，像tess.c这种声明了回调函数的程序需要在声明函数时加上符号CALLBACK。可以在CALLBACK中使用空白的定义（如下所示），这个技巧可以使程序在Microsoft Windows和Linux系统中都能运行。

```
#ifndef CALLBACK
#define CALLBACK
#endif

void CALLBACK callbackFunction(...) {
   ....
}
```

示例程序11-1注册的GLU_TESS_VERTEX回调函数就是OpenGL自身提供的glVertex3dv()函数，并且它接受的参数就是每个顶点的坐标。但是，如果想为每个顶点指定更多的信息，例如颜色值、表面法线向量或纹理坐标，必须使用一个更为复杂的回调函数。示例程序11-2显示了另一个分格化对象的开始部分，这段代码也位于tess.c程序中。被注册的回调函数vertexCallback()需要接受的参数是一

个指向6个双精度浮点值的指针，即这个顶点的x、y、z坐标以及它的红、绿、蓝颜色值。

示例程序11-2 vertex和combine回调函数：tess.c

```
/*a different portion of init()*/
   gluTessCallback(tobj,GLU_TESS_VERTEX,vertexCallback);
   gluTessCallback(tobj,GLU_TESS_BEGIN,beginCallback);
   gluTessCallback(tobj,GLU_TESS_END,endCallback);
   gluTessCallback(tobj,GLU_TESS_ERROR,errorCallback);
   gluTessCallback(tobj,GLU_TESS_COMBINE,combineCallback);

/*new callback routines registered by these calls */
void CALLBACK vertexCallback(GLvoid *vertex)
{
   const GLdouble *pointer;

   pointer =(GLdouble *)vertex;
   glColor3dv(pointer+3);
   glVertex3dv(vertex);
}
void CALLBACK combineCallback(GLdouble coords [3],
                     GLdouble *vertex_data [4],
                     GLfloat weight [4] ,GLdouble **dataOut )
{
   GLdouble *vertex;
   int i;

   vertex =(GLdouble *)malloc(6 *sizeof(GLdouble));
   vertex [0] =coords [0];
   vertex [1] =coords [1];
   vertex [2] =coords [2];
   for (i =3;i <6;i++)
      vertex [i] =weight [0] *vertex_data [0][i]
                  +weight [1] *vertex_data [1][i]
                  +weight [2] *vertex_data [2][i]
                  +weight [3] *vertex_data [3][i];
   *dataOut =vertex;
}
```

示例程序11-2还显示了GL_TESS_COMBINE回调函数的用法。当分格化算法检查输入轮廓线的相交情况，以决定是否创建新顶点时，GL_TESS_COMBINE回调函数就会被调用。当分格化对象决定把两个非常靠近的顶点合并为一个顶点时，这个回调函数也会被调用。新创建的顶点是最多可达4个的原有顶点的线性组合，它们在示例程序11-2中是用vertex_data[0..3]引用的。线性组合的系数是由weight[0..3]提供的。这些系数之和为1.0。coords提供了这个新顶点的位置。

这个被注册的回调函数必须为另一个顶点分配内存，使用vertex_data和weight数组中的数据进行加权插值，并在dataOut中返回这个新顶点的指针。示例程序11-2中的combineCallback()函数对RGB颜色值进行插值。这个函数分配了一个包含6个元素的数组，把x、y、z坐标放在前3个元素中，然后把经过加权插值的颜色值放在最后3个元素中。

用户指定的数据

一共可以注册6种类型的回调函数。每种类型的回调函数都有2个版本，因此一共有12个回调函数。

对于每种类型的回调函数，都有一个具有用户指定数据的版本和一个非用户指定数据的版本。用户指定的数据是由应用程序向gluTessBeginPolygon()函数提供的，然后不经修改地传递给每个*DATA回调函数。使用GLU_TESS_BEGIN_DATA，用户指定的数据可以用于表示“每个多边形”的数据。如果指定了一个特定回调函数的全部两个版本，具有user_data的那个版本就会被实际使用，另一个则被忽略。因此，尽管一共存在12个回调函数，但是在任一时刻实际能够使用的回调函数的最大数量为6。

例如，示例程序11-2使用了平滑着色，因此vertexCallback()函数为每个顶点指定了一种RGB颜色。如果想使用光照和平滑着色，回调函数应该为每个顶点指定一条法线向量。但是，如果想使用光照和单调着色，可以只为整个多边形（而不是为每个顶点）指定一条表面法线。在这种情况下，可以选择使用GLU_TESS_BEGIN_DATA回调函数，并且在user_data指针中传递顶点坐标和表面法线的值。

11.1.3 分格化属性

在进行分格化和渲染之前，可以使用gluTessProperty()函数设置几个属性，对分格化算法进行控制。在这些属性中，最重要也是最复杂的属性是环绕规则，它决定了多边形的“内部”和“外部”。

---

void **gluTessProperty**(GLUtesselator **tessobj*, GLenum *property*,
GLdouble *value*);

对于分格化对象*tessobj*，*property*的当前值被设置为*value*。*property*是GLU_TESS_BOUNDARY_ONLY、GLU_TESS_TOLERANCE或GLU_TESS_WINDING_RULE。

如果property是GLU_TESS_BOUNDARY_ONLY，则*value*是GL_TRUE或GL_FALSE。当它设置为GL_TRUE时，多边形不再分格化为填充多边形，而是用线框来绘制多边形的轮廓，区分多边形的内部和外部。这个属性的默认值是GL_FALSE（参阅**gluTessNormal()**函数，了解如何控制轮廓线的环绕方向）。

如果*property*的值是GLU_TESS_TOLERANCE，*value*是一个表示距离的公差值，表示两个顶点是否足够靠近，可以由GLU_TESS_COMBINE回调函数进行合并。这个公差值与一个输入顶点的最大坐标值相乘，以决定任何顶点作为合并操作的结果可以移动的最大距离。读者使用的OpenGL实现可能并不支持顶点合并，这个公差值只是作为一个提示值。默认的公差值是0。

GLU_TESS_WINDING_RULE属性决定了多边形的哪部分位于内部，哪部分位于外部（因此不必进行填充）。*value*可以是GLU_TESS_WINDING_ODD（默认值）、GLU_TESS_WINDING_NONZERO、GLU_TESS_WINDING_POSITIVE、GLU_TESS_WINDING_ NEGATIVE或GLU_TESS_WINDING_ABS_GEQ_TWO。

环绕数和环绕规则

对于一条简单的轮廓线，每个点的环绕数就是环绕这个点的所有轮廓线的代数和（用一个有符号整数表示，求和规则是：逆时针环绕的轮廓线加1，顺时针环绕的轮廓线减1）。这个过程把一个有符号整数值与平面上的每个点相关联。注意，对于同一个区域中的所有点，它们的环绕数都是相同的。

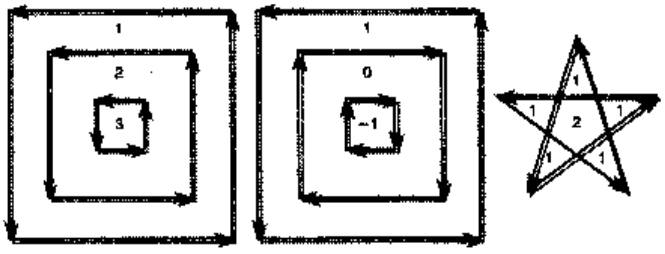


图11-2 示例轮廓线的环绕数

图11-2显示了3组轮廓线以及这些轮廓线内部点的环绕数。在左边那组轮廓线中，3条轮廓

线都是逆时针方向的，因此每个嵌套的内部区域的环绕数都要加上1。在中间那条轮廓线中，两条轮廓线都是以顺时针方向绘制的，因此环绕数将会递减，结果是负值。

如果一个区域的环绕数属于环绕规则选中的类型（奇数、非零、正数、负数、绝对值大于或等于2），那么它就是内部区域。通常，环绕规则把具有奇数和非零环绕数的区域定义为内部区域。正值、负值和“绝对值大于2”这些环绕规则在多边形CSG（计算实体几何图形，computational solidgeometry）操作中具有一定的用途。

tesswind.c程序说明了环绕规则的效果。这个程序对图11-3显示的4组轮廓线进行渲染。用户可以选择不同的环绕规则来观察它们的效果。对于每个环绕规则，黑色表示内部区域。注意顺时针环绕和逆时针环绕的效果。

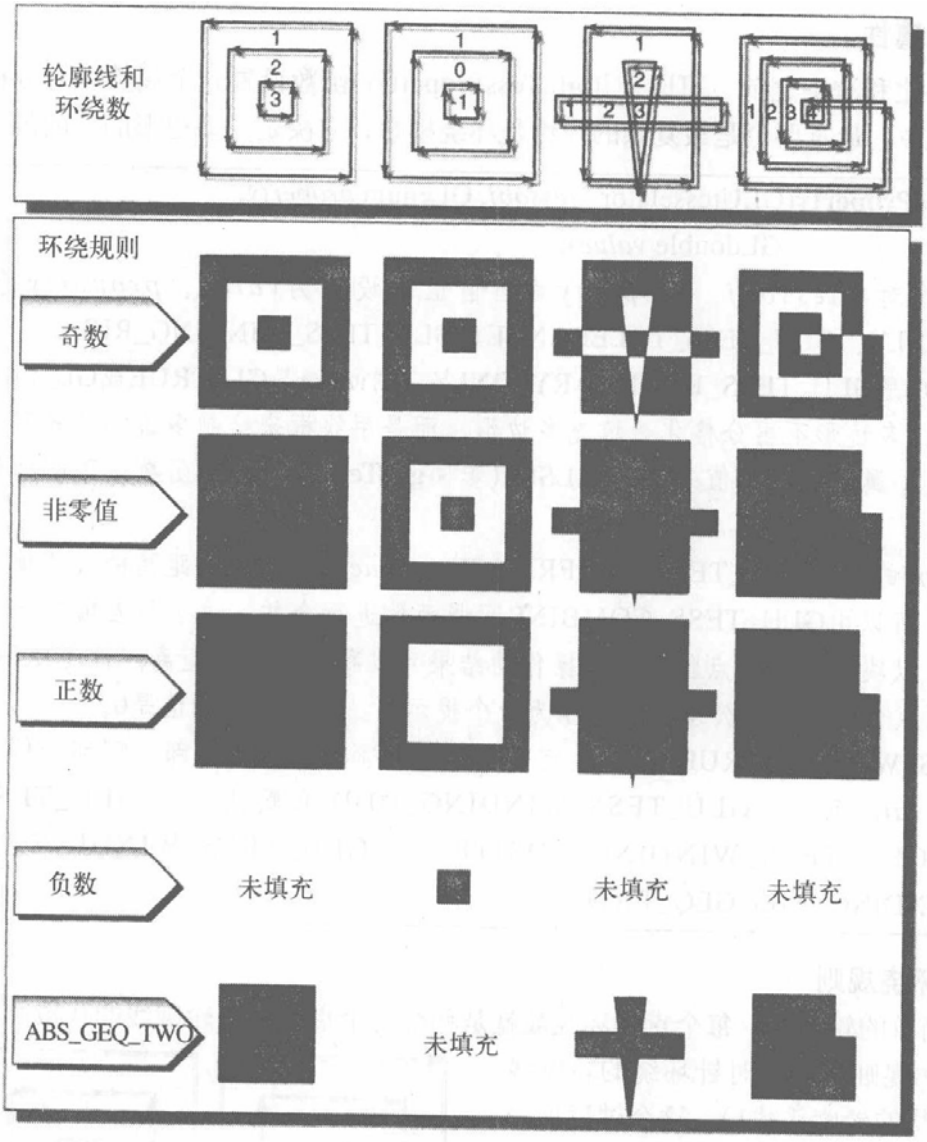


图11-3 用环绕规则定义内部区域

环绕规则的CSG用法

GLU_TESS_WINDING_ODD和GLU_TESS_WINDING_NONZERO是最常用的环绕规则，它们适用于绝大多数情况下的多边形着色。

环绕规则还可以用于CSG操作，以便于找出几条轮廓线的并、差、交集（布尔型操作）。

首先，假设每条轮廓线已经进行了定义，每个外部区域的环绕数为0，每个内部区域的环绕数为1（每条轮廓线自身不得相交）。使用这种模型，逆时针轮廓线定义了多边形的外边界，顺时针轮廓线定义了空洞。轮廓线也可以嵌套，但被嵌套的轮廓线的方向必须与嵌套它的轮廓线的方向相反。

如果原多边形并不满足这个描述，可以在打开GLU_TESS_BOUNDARY_ONLY属性的情况下运行分格化对象，把它们转换为这种形式。这个函数将会返回一个满足上面所描述规则的轮廓线列表。通过创建2个分格化对象，其中一个分格化对象的回调函数的输出可以直接作为另一个分格化对象的回调函数的输入。

假如有2个或多个满足上述条件的多边形，可以像下面这样实现CSG操作：

- 并集（UNION）：为了计算几条轮廓线的并集，可以把所有的输入轮廓线画成单个多边形。绘图产生的每个区域的环绕数就是覆盖它的原多边形的数量。可以使用GLU_TESS_WINDING_NONZERO或GLU_TESS_WINDING_POSITIVE环绕规则来提取并集。注意，如果使用非零值规则，即使把所有的轮廓线的方向取反，得到的并集结果仍然和原来相同。
- 交集（INTERSECTION）：这个操作一次只能用于2条轮廓线。用2条轮廓线绘制一个多边形，然后使用GLU_TESS_WINDING_ABS_GEQ_TWO来提取结果。
- 差集（DEFFERENCE）：假如想计算“A”减去“B并C并D”，可以绘制一个依次使用未经修改的轮廓线A、B、C和D所组成的多边形（它们的顶点顺序相反）。为了提取这个操作的结果，可以使用GLU_TESS_WINDING_POSITIVE环绕规则。（如果B、C和D是GLU_TESS_WINDING_ONLY操作的结果，反转顶点顺序的另一种方法是使用gluTessNormal()函数反转法线的符号。）

其他分格化属性函数

GLU还提供了许多其他函数，它们可以与gluTessProperty()函数协同工作。gluGetTessProperty()用于提取分格化属性的当前值。如果被使用的分格化对象用于生成线框轮廓而不是填充多边形，gluTessNormal()可以用于确定被分格化的多边形的环绕方向。

---

void **gluGetTessProperty**(GLUtesselator **tessobj*, GLenum *property*,
GLdouble **value*);

对于分格化对象*tessobj*，*property*的当前值返回到*value*。*property*和*value*的值与**gluTessProperty()**函数的对应值相同。

---

void **gluTessNormal**(GLUtesselator **tessobj*, GLdouble *x*, GLdouble *y*,
GLdouble *z*);

对于分格化对象*tessobj*，**gluTessNormal()**函数定义了一条法线向量，用于控制生成多边形的环绕方向。在分格化之前，所有的输入数据都投影到一个与法线垂直的平面上。然后，所有输出三角形的方向都根据法线设置为逆时针方向（可以通过反转这条法线向量的符号来获取顺时针方向的环绕）。默认的法线为（0,0,0）。

如果读者对输入数据的位置和方向有所了解，可以使用gluNormal()函数提高分格化操作的速度。例如，如果知道所有的多边形都位于xy平面，可以调用gluNormal(tessobj, 0, 0, 1)。

如上所述，默认的法线为（0，0，0），它的效果并不是很明显。在这种情况下，它期望输入数

据落在同一个平面附近，并根据这些顶点拟合出一个平面，而不管它们实际上是如何连接的。在选择法线的符号时，要让所有输入轮廓线的有符号区域总和为非负值（逆时针轮廓线所包围的区域为正）。但是，如果输入数据并不位于同一平面，把数据投影到垂直于法线的平面再进行计算可能会极大地改变几何形状。

11.1.4 多边形定义

在设置了所有的分格化属性，并注册了相关的回调函数之后，就进入了描述组成输入轮廓线的顶点并对多边形进行分格化的最后阶段。

void **gluTessBeginPolygon**(GLUtesselator **tessobj*, void **user_data*);

void **gluTessEndPolygon**(GLUtesselator **tessobj*);

开始和结束对需要进行分格化的多边形的指定，并把分格化对象*tessobj*与它相关联。*user_data*指向一个用户定义的数据结构，传递给所有在GLU_TESS_*_DATA回调函数中注册的函数。

gluTessBeginPolygon()和gluTessEndPolygon()之间包括了一条或多条轮廓线的定义。当gluTessEndPolygon()函数被调用时，OpenGL就实现分格化算法，并生成和渲染经过分格化的多边形。回调函数和分格化属性是通过gluTessCallback()和gluTessProperty()函数绑定并设置到分格化对象。

void **gluTessBeginContour**(GLUtesselator **tessobj*);

void **gluTessEndContour**(GLUtesselator **tessobj*);

开始和结束一条闭合的轮廓线的指定，这条轮廓线是一个多边形的一部分。闭合的轮廓线是通过零次或多次调用**gluTessVertex()**函数（用于定义顶点）形成的。每条轮廓线的最后一个顶点会自动连接到第一个顶点。

在实际使用中，为了创建一条有意义的轮廓线，至少需要使用3个顶点。

void **gluTessVertex**(GLUtesselator **tessobj*, GLdouble *coords*[3],
void **vertex_data*);

指定分格化对象的当前轮廓线的一个顶点。*coords*包含了三维顶点坐标，*vertex_data*是一个指针，它发送到与GLU_TESS_VERTEX或GLU_TESS_VERTEX_DATA相关联的回调函数。一般情况下，*vertex_data*包含了顶点坐标、表面法线、纹理坐标、颜色信息或应用程序可能需要的其他信息。

示例程序11-3显示了tess.c程序的一部分，它定义了2个多边形。一个多边形由一条矩形的轮廓线构成，中间有一个三角形的洞。另一个多边形则是一个平滑着色的、自身相交的五角星。为了提高效率，这2个多边形都存储在显示列表中。第一个多边形由两条轮廓线组成，外面那条是逆时针方向环绕的，形成“空洞”的那条则是顺时针方向环绕的。至于第二个多边形，star数组包含了坐标和颜色数据，它的分格化回调函数vertexCallback()需要使用这两种数据。

每个顶点必须位于不同的内存位置，这一点非常重要，因为gluTessVertex()函数并不复制顶点数据，它只保存指针（vertex_data）。如果程序在相同的内存位置复用几个顶点，它可能无法产生预期的结果。

注意：在gluTessVertex()函数中，为coords和vertex_data参数两次指定相同的顶点坐标数据似乎是冗余的。但是，这种做法是必要的。coords只表示顶点坐标。vertex_data使用顶点坐标，但它还可能使用顶点的其他信息。

示例程序11-3 多边形定义：tess.c

```
GLdouble rect [4][3]={50.0,50.0,0.0,
                      200.0,50.0,0.0,
                      200.0,200.0,0.0,
                      50.0,200.0,0.0};
GLdouble tri [3][3] ={75.0,75.0,0.0,
                       125.0,175.0,0.0,
                       175.0,75.0,0.0};
GLdouble star [5][6]={250.0,50.0,0.0,1.0,0.0,1.0,
                      325.0,200.0,0.0,1.0,1.0,0.0,
                      400.0,50.0,0.0,0.0,1.0,1.0,
                      250.0,150.0,0.0,1.0,0.0,0.0,
                      400.0,150.0,0.0,0.0,1.0,0.0};
startList =glGenLists(2);
tobj =gluNewTess();
gluTessCallback(tobj,GLU_TESS_VERTEX,glVertex3dv);
gluTessCallback(tobj,GLU_TESS_BEGIN,beginCallback);
gluTessCallback(tobj,GLU_TESS_END,endCallback);
gluTessCallback(tobj,GLU_TESS_ERROR,errorCallback);

glNewList(startList,GL_COMPILE);
glShadeModel(GL_FLAT);
gluTessBeginPolygon(tobj,NULL);
   gluTessBeginContour(tobj);
      gluTessVertex(tobj,rect [0],rect [0]);
      gluTessVertex(tobj,rect [1],rect [1]);
      gluTessVertex(tobj,rect [2],rect [2]);
      gluTessVertex(tobj,rect [3],rect [3]);
   gluTessEndContour(tobj);
   gluTessBeginContour(tobj);
      gluTessVertex(tobj,tri [0],tri [0]);
      gluTessVertex(tobj,tri [1],tri [1]);
      gluTessVertex(tobj,tri [2],tri [2]);
   gluTessEndContour(tobj);
gluTessEndPolygon(tobj);
glEndList();

gluTessCallback(tobj,GLU_TESS_VERTEX,vertexCallback);
gluTessCallback(tobj,GLU_TESS_BEGIN,beginCallback);
gluTessCallback(tobj,GLU_TESS_END,endCallback);
gluTessCallback(tobj,GLU_TESS_ERROR,errorCallback);
gluTessCallback(tobj,GLU_TESS_COMBINE,combineCallback);

glNewList(startList +1,GL_COMPILE);
glShadeModel(GL_SMOOTH);
gluTessProperty(tobj,GLU_TESS_WINDING_RULE,
                GLU_TESS_WINDING_POSITIVE);
gluTessBeginPolygon(tobj,NULL);
   gluTessBeginContour(tobj);
      gluTessVertex(tobj,star [0],star [0]);
      gluTessVertex(tobj,star [1],star [1]);
```

```
        gluTessVertex(tobj,star [2],star [2]);
        gluTessVertex(tobj,star [3],star [3]);
        gluTessVertex(tobj,star [4],star [4]);
    gluTessEndContour(tobj);
gluTessEndPolygon(tobj);
glEndList();
```

11.1.5 删除分格化对象

如果不再需要一个分格化对象，可以使用gluDeleteTess()函数删除它，并释放与它相关联的所有内存。

---

void **gluDeleteTess**(GLUtesselator **tessobj*);

删除指定的分格化对象*tessobj*，并释放与它相关联的所有内存。

11.1.6 提高分格化性能的建议

为了实现最佳的性能，可以参考下面这些规则：

- 把分格化操作的输出存储在显示列表或其他用户数据结构中。为了获取经过分格化之后的顶点数据，可以在反馈模式下对多边形进行分格化（参见第13.2节）。
- 使用gluTessNormal()函数提供多边形法线。
- 使用同一个分格化对象来渲染大量多边形，而不是为每个多边形分配一个新的分格化对象。（在多线程、多处理器的环境下，可以通过使用多个分格化对象获得更高的性能。）

11.1.7 描述GLU错误

GLU提供了一个函数，可以生成一个用于描述错误代码的字符串。这个函数并不限于分格化操作，也可以用于NURBS和二次方程表面产生的错误，甚至是基本GL错误。关于OpenGL的错误处理工具的更多信息，请参阅第14章。

11.1.8 向后兼容性

如果使用的是1.0或1.1版本的GLU，它的分格化对象的功能要弱得多。1.0/1.1版本的分格化对象只能处理简单的非凸多边形或者中间有孔的简单多边形。它无法对具有相交轮廓线的多边形进行分格化（没有COMBINE回调函数），也不能处理基于多边形的数据。1.0/1.1版本的分格化对象仍然能在GLU 1.2或1.3版本中使用，但并不推荐使用。

1.0/1.1版本的分格化对象与当前的分格化对象具有相似之处。这两种分格化对象都使用gluNewTess()和gluDeleteTess()函数。主要的顶点指定函数仍然是gluTessCallback()。回调机制仍然是由gluTessCallback()函数控制的，但是它只能注册5个回调函数，是目前能够注册的12个回调函数的一个子集。

下面是1.0/1.1版本的分格化对象的原型：

---

void **gluBeginPolygon**(GLUtriangulatorObj **tessobj*);

void **gluNextContour**(GLUtriangulatorObj **tessobj*, GLenum *type*);

void **gluEndPolygon**(GLUtriangulatorObj **tessobj*);

最外层的轮廓线必须首先指定，但在此之前并不需要调用**gluNextContour()**函数。对于中间有洞的多边形，只需要定义一条轮廓线，因此无须调用**gluNextContour()**。如果一个多边形有多条轮廓线（即多边形中间有洞，甚至洞中还有洞），这些轮廓线就需要逐条指定，在指定每条轮廓线之前都需要

调用**gluNextContour()**函数。轮廓线的每个顶点都需要调用**gluTessVertex()**函数。

对于**gluNextContour()**函数，*type*参数可以是GLU_EXTERIOR、GLU_INTERIOR、GLU_CCW、GLU_CW或GLU_UNKNOWN。这些只是作为分格化操作的提示。如果设置了正确的提示，分格化操作的速度可能会更快一些。即使设置的提示有误，它们也会被忽略，分格化操作仍然会进行。对于中间有洞的多边形，有一条轮廓线为外部轮廓线，其余为内部轮廓线。第一条轮廓线的类型被设定为GLU_EXTERIOR。在三维空间中，可以任意选择顺时针方向或逆时针方向。但是，任何平面都存在两种不同的方向，因此应该一致地选用GLU_CCW和GLU_CW类型。如果并不了解相关的信息，可以使用GLU_UNKNOWN。

强烈建议读者把GLU1.0/1.1的代码转换为使用GLU 1.2的新分格化接口。只要执行如下步骤就可以实现这个目的：

1) 将主数据结构类型从GLUtriangularObj修改为GLUTesslator。在GLU 1.2中，GLUtriangularObj和GLUTesslator定义为相同的类型。

2) 把gluBeginPolygon()转换为2个函数：gluTessBeginPolygon()和gluTessBeginContour()。所有的轮廓线都必须显式地开始定义，包括第一条。

3) 把gluNextContour()转换为gluTessEndContour()和gluTessBeginContour()。在开始定义新的轮廓线之前，必须结束前一条轮廓线的定义。

4) 把gluEndPolygon()转换为gluTessEndContour()和gluTessEndPolygon()。最终的轮廓线必须闭合。

5) 修改gluTessCallback()函数使用的常量。在GLU 1.2中，GLU_BEGIN、GLU_VERTEX、GLU_END、GLU_ERROR和GLU_EDGE_FLAG定义为与GLU_TESS_BEGIN、GLU_TESS_VERTEX、GLU_TESS_END、GLU_TESS_ERROR和GLU_TESS_EDGE_ FLAG相同。

11.2 二次方程表面：渲染球体、圆柱体和圆盘

基本的OpenGL函数库只支持对简单的点、直线和填充凸多边形进行建模和渲染。无论是3D物体还是像圆这样的常用2D物体都无法直接进行渲染。

通过本书前面的学习，读者已经知道了如何使用GLUT创建一些3D物体。GLU还提供了一些函数，通过多边形分格化，提供了对各种通过多边形近似模拟的并可以使用二次方程计算的2D和3D形状（球体、圆柱体、圆盘以及不完整的圆盘）进行建模和渲染。这些函数包括以各种风格和方向绘制二次方程表面的函数。二次方程表面是通过下面这个通用的二次方程定义的：

$$a_1x^2 + a_2y^2 + a_3z^2 + a_4xy + a_5yz + a_6xz + a_7x + a_8y + a_9z + a_{10} = 0$$

（参见David Roger的《Procedural Elements for Computer Graphics》New York，NY：McGraw-Hill，1985）创建和渲染二次方程表面类似于使用分格化对象。为了创建和渲染二次方程表面，可以使用如下步骤：

1) 使用gluNewQuadric()函数，创建一个二次方程对象。

2) 指定这个二次方程对象的属性（除非对它们的默认值感到满意）。

①使用gluQuadricOrientation()函数控制环绕方向，区分内部区域和外部区域。

②使用gluQuadricDrawStyle()函数选择把物体渲染为点、直线或填充多边形。

③ 对于使用了光照的二次方程表面，可以使用gluQuadricNormal()函数为每个顶点或每个面指定一条法线向量。在默认情况下不会生成任何法线。

④对于进行了纹理贴图的二次方程表面，可以使用gluQuadricTexture()函数生成纹理坐标。

3) 使用gluQuadricCallback ()函数注册一个错误处理函数。如果在渲染过程中发生了错误，这个被注册的函数就会被调用。

4) 根据需要调用相应的二次方程表面渲染函数：gluSphere()、gluCylinder()、gluDisk()或gluPartialDisk()。为了提供静态数据的性能，可以把二次方程对象封装到显示列表中。

5) 完成了操作之后，可以用gluDeleteQuadric()函数销毁这个二次方程对象。如果需要创建其他的二次方程表面，最好复用原来的二次方程对象。

11.2.1 管理二次方程对象

二次方程对象由参数、属性和回调函数组成，它们存储在一个GLUquadricObj类型的数据结构中。二次方程对象可以用于生成顶点、法线、纹理坐标以及其他数据。这些数据可以直接使用，也可以存储在显示列表中。下面这些函数用于创建和销毁二次方程对象，并报告在此期间所产生的错误。

GLUquadricObj* **gluNewQuadric**(void);

创建一个新的二次方程对象，并返回一个指向它的指针。如果操作失败，它就返回一个NULL指针。

void **gluDeleteQuadric**(GLUquadricObj **qobj*);

销毁二次方程对象*qobj*，并释放它所使用的内存。

void **gluQuadricCallback**(GLUquadricObj **qobj*, GLenum *which*,
void (**fn*)());

定义了一个函数*fn*，以便在特定的情况下调用。GLU_ERROR是*which*参数唯一合法的值，因此*fn*函数是在出现错误的情况下调用的。如果*fn*为NULL，任何原有的回调函数都会被删除。

对于GLU_ERROR，*fn*函数在调用时使用1个参数，也就是错误代码。可以使用**gluErrorString**()函数把这个错误代码转换为ACSII字符串。

11.2.2 控制二次方程对象的属性

下面这些函数影响二次方程对象生成的数据的类型。我们可以在实际指定图元之前调用这些函数。

稍后的示例程序11.4的quadric.c除了说明如何创建二次方程对象、进行错误处理以及绘制图元之外，还解释了如何更改绘图风格和法线类型。

void **gluQuadricDrawStyle**(GLUquadricObj **qobj*, GLenum *drawStyle*);

对于二次方程对象*qobj*，*drawStyle*参数用于控制它的绘图风格。这个参数的合法值是GLU_POINT、GLU_LINE、GLU_SILHOUETTE或GLU_FILL。

GLU_POINT和GLU_LINE分别指定了图元应该被渲染为点（在每个顶点处）的形式还是每对连接顶点之间的直线形式。

GLU_SILHOUETTE指定了使用直线来渲染图元。但是，如果两个面位于同一个平面上，它们之间的边便不被绘制。这种绘图风格常常用于gluDisk()和gluPartialDisk()函数。

GLU_FILL表示以填充方式对多边形进行渲染，其中多边形根据它们的法线按照逆时针方向进行绘制。这个操作可能受到gluQuadricOrientation()函数的影响。

void **gluQuadricOrientation**(GLUquadricObj *qobj, GLenum *orientation*);

对于二次方程对象*qobj*，*orientation*是GL_OUTSIDE（默认值）或GL_INDISE，这个参数用于控制法线向量的方向。

对于gluSphere()和gluCylinder()，外部和内部的定义是显而易见的。对于gluDisk()和gluPatrialDisk()，z轴正方向的那侧被认为是外部。

void **gluQuadricNormals**(GLUquadricObj *qobj, GLenum *normals*);

对于二次方程对象*qobj*，*normal*参数的值可以是GLU_NONE（默认值）、GLU_FLAT或GLU_SMOOTH。

gluQuadricNormals()函数用于指定何时生成法线向量。GLU_NONE表示不生成法线向量，适用于不使用光照的场合。GLU_FLAT为每个面生成一条法线，常用于单调着色情况下的光照场景。GLU_SMOOTH为二次方程表面的每个顶点生成一条法线向量，适用于平滑着色情况下的光照场景。

void **gluQuadricTexture**(GLUquadricObj *qobj,
GLboolean *textureCoords*);

对于二次方程对象*qobj*，*textureCoords*的值可以是GL_FALSE（默认值）或GL_TRUE。如果*textureCoords*的值是GL_TRUE，就会为这个二次方程表面生成纹理坐标。纹理坐标的生成方式因被渲染物体的类型而异。

11.2.3 二次方程图元

下面这些函数用于产生顶点以及组成二次方程表面所需的其他数据。在每个函数说明中，qobj表示一个由gluNewQuadric()函数创建的二次方程对象。

void **gluSphere**(GLUquadricObj *qobj, GLdouble *radius*, GLint *slices*,
GLint *stacks*);

绘制一个半径为*radius*的球体，球心位于原点（0，0，0）。这个球体被垂直划分为*slices*瓣（类似于经线），并被水平划分为*stacks*层（类似于纬线）。

如果启用了纹理坐标生成，纹理坐标*t*从$z = -$radius处的0.0变化到z = radius处的1.0。*t*坐标的值沿经线线性地增加。另外，纹理坐标*s*从+*y*轴的0.0变化到+*x*轴的0.25，再到−*y*轴的0.5，再到−轴的0.75，最后回到+*y*轴的1.0。

void **gluCylinder**(GLUquadricObj *qobj, GLdouble *baseRadius*,
GLdouble *topRadius*, GLdouble *height*,
GLint *slices*, GLint *stacks*);

绘制一个沿z轴的圆柱体。这个圆柱体的底面位于$z = 0$的平面，顶面位于z = height的平面。和球体一样，圆柱体被垂直划分为*slices*瓣（类似于经线），并被水平划分为*stacks*层（类似于纬线）。*baseRadius*是$z = 0$处的半径，*topRadius*是z = height处的半径。如果z = height处的半径为零，那么这个圆柱体就是一个圆锥体。

如果启用了纹理坐标生成，那么纹理坐标*t*就从$z = 0$处的0.0线性地变化到z = *height*处的1.0。纹理坐标*s*的变化范围和球体相同。

注意：圆柱体在顶部或底部并不闭合。位于圆柱体顶部和底部的圆盘并不会被绘制。

void **gluDisk**(GLUquadricObj **qobj*, GLdouble *innerRadius*,
GLdouble *outerRadius*, GLint *slices*, GLint *rings*);

绘制一个位于$z = 0$平面的圆盘，*outRadius*指定了圆盘的外半径，*innerRadius*指定了圆盘的内半径。如果*innerRadius*为0，圆盘就没有孔。圆盘绕z轴分割为*slices*个瓣（类似于切比萨饼），并沿z轴划分为*rings*个同心圆。

对于圆盘，+z轴那边为“外部”。也就是说，生成的法线都指向+z方向。否则，法线都指向−z方向。

如果启用了纹理坐标生成，纹理坐标是按线性方式生成的，当R = *outerRadius*时，纹理坐标s和t在（R，0，0）处是（1，0.5），在（0，R，0）处是（0.5，1），在（−R，0，0）处是（0，0.5），在（0，−R，0）处是（0.5，0）。

void **gluPartialDisk**(GLUquadricObj **qobj*, GLdouble *innerRadius*,
GLdouble *outerRadius*, GLint *slices*, GLint *rings*,
GLdouble *startAngle*, GLdouble *sweepAngle*);

绘制一个位于$z = 0$平面的不完整圆盘。不完整圆盘的*outRadius*、*innerRadius*、*slices*、*rings*参数的含义与完整圆盘相同。区别在于不完整圆盘只绘制圆盘的一部分，从*startAngle*开始，共绘制*startAngle* + *sweepAngle*度（其中*startAngle*和*sweepAngle*是用角度来表示的，0度表示沿+y轴方向，90度表示沿+x轴方向，180度表示沿−y轴方向，270度表示沿−x轴方向）。

不完整圆盘处理方向和纹理坐标的方式与完整圆盘相同。

注意：对于所有的二次方程表面，使用*Radius、height和类似的参数对它们进行缩放要比使用glScale*()函数更好，因为前者不需要对法线向量进行重新规范化。可以把rings和stacks参数设置为不等于1的值，强制进行粒度更细的光照计算，尤其是在材料的镜面成分非常大的时候。

示例程序11-4显示了如何绘制各种二次方程表面，并显示了不同绘图风格的效果。

示例程序11-4 二次方程表面：quadric.c

```
#ifndef CALLBACK
#define CALLBACK
#endif

GLuint startList;

void CALLBACK errorCallback(GLenum errorCode)
{
   const GLubyte *estring;

   estring = gluErrorString(errorCode);
   fprintf(stderr,"Quadric Error:%s \n",estring);
   exit(0);
}

void init(void)
{
   GLUquadricObj *qobj;
   GLfloat mat_ambient [] ={0.5,0.5,0.5,1.0 };
```

```
    GLfloat mat_specular [] ={1.0,1.0,1.0,1.0 };
    GLfloat mat_shininess []={50.0 };
    GLfloat light_position [] ={1.0,1.0,1.0,0.0 };
    GLfloat model_ambient []={0.5,0.5,0.5,1.0 };

    glClearColor(0.0,0.0,0.0,0.0);

    glMaterialfv(GL_FRONT,GL_AMBIENT,mat_ambient);
    glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
    glMaterialfv(GL_FRONT,GL_SHININESS,mat_shininess);
    glLightfv(GL_LIGHT0,GL_POSITION,light_position);
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT,model_ambient);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_DEPTH_TEST);

/* Create 4 display lists,each with a different quadric object.
 *Different drawing styles and surface normal specifications
 *are demonstrated.
 */
    startList =glGenLists(4);
    qobj =gluNewQuadric();
    gluQuadricCallback(qobj,GLU_ERROR,errorCallback);

    gluQuadricDrawStyle(qobj,GLU_FILL);/*smooth shaded */
    gluQuadricNormals(qobj,GLU_SMOOTH);
    glNewList(startList,GL_COMPILE);
       gluSphere(qobj,0.75,15,10);
    glEndList();

    gluQuadricDrawStyle(qobj,GLU_FILL);/*flat shaded */
    gluQuadricNormals(qobj,GLU_FLAT);
    glNewList(startList+1,GL_COMPILE);
       gluCylinder(qobj,0.5,0.3,1.0,15,5);
    glEndList();

    gluQuadricDrawStyle(qobj,GLU_LINE);/*wireframe */
    gluQuadricNormals(qobj,GLU_NONE);
    glNewList(startList+2,GL_COMPILE);
       gluDisk(qobj,0.25,1.0,20,4);
    glEndList();

    gluQuadricDrawStyle(qobj,GLU_SILHOUETTE);
    gluQuadricNormals(qobj,GLU_NONE);
    glNewList(startList+3,GL_COMPILE);
       gluPartialDisk(qobj,0.0,1.0,20,4,0.0,225.0);
    glEndList();
}

void display(void)
```

```
{
    glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);
    glPushMatrix();

    glEnable(GL_LIGHTING);
    glShadeModel(GL_SMOOTH);
    glTranslatef(-1.0,-1.0,0.0);
    glCallList(startList);

    glShadeModel(GL_FLAT);
    glTranslatef(0.0,2.0,0.0);
    glPushMatrix();
    glRotatef(300.0,1.0,0.0,0.0);
    glCallList(startList+1);
    glPopMatrix();

    glDisable(GL_LIGHTING);
    glColor3f(0.0,1.0,1.0);
    glTranslatef(2.0,-2.0,0.0);
    glCallList(startList+2);
    glColor3f(1.0,1.0,0.0);
    glTranslatef(0.0,2.0,0.0);
    glCallList(startList+3);

    glPopMatrix();
    glFlush();
}
void reshape(int w,int h)
{
    glViewport(0,0,(GLsizei)w,(GLsizei)h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <=h)
       glOrtho(-2.5,2.5,-2.5*(GLfloat)h/(GLfloat)w,
          2.5*(GLfloat)h/(GLfloat)w,-10.0,10.0);
    else
       glOrtho(-2.5*(GLfloat)w/(GLfloat)h,
          2.5*(GLfloat)w/(GLfloat)h,-2.5,2.5,-10.0,10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void keyboard(unsigned char key,int x,int y)
{
    switch (key){
       case 27:
          exit(0);
          break;
    }
}
```

```
int main(int argc,char**argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB |GLUT_DEPTH);
   glutInitWindowSize(500,500);
   glutInitWindowPosition(100,100);
   glutCreateWindow(argv [0]);
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
   return 0;
}
```

第12章　求值器和NURBS

本章目标



高级知识

- 使用OpenGL的求值器函数绘制基本的曲线和表面。
- 使用GLU的高层NURBS工具绘制复杂的曲线和表面，并返回几何物体的顶点数据。这些几何图形是对曲线和表面进行分格化操作产生的。

注意，本章假设了一些前提条件，第12.1节列出了这些前提。

注意：在OpenGL 3.1中，本章所介绍的所有技术和函数都废弃删除了。即便其中一些功能是GLU库的一部分，它还是依赖于已经从核心OpenGL库删除的那些功能。

在最底层，图形硬件所绘制的是点、直线和多边形（通常是三角形和四边形）。平滑的曲线或表面是通过使用大量的微小线段或多边形模拟的。但是，从数学角度而言，许多非常实用的曲线和表面可以用少许几个参数（例如控制点）来描述。保存一个表面的16个控制点要比保存1000个三角形以及这些三角形每个顶点的法线向量信息所需要的空间要少得多。另外，这1000个三角形只能对真正的表面进行近似的模拟，而这些控制点却能够准确地描述真正的表面。

求值器提供了一种方式，只用少数的控制点来指定曲线或表面（或其中的一部分）上的点。然后，就可以按照任意的精度来渲染曲线或表面。此外，它还可以自动计算表面的法线向量。可以按照多种方式使用求值器返回的点，例如绘制点状表面、绘制线框表面，也可以绘制进行了完全的光照、着色甚至纹理处理的表面。

可以使用求值器描述任何角度的多项式或有理多项式样条或表面，它们几乎包括了如今所有常见的样条或样条表面，包括B-样条、NURBS（非均匀有理B-样条）表面、Bézier曲线和表面，以及Hermite样条。由于求值器只提供了对曲线或表面的低层描述，因此它们一般存在于低层的工具函数库中。程序员使用的一般是更高层次的接口，GLU的NURBS工具就是这样一种高层接口。NURBS函数封装了大量的复杂代码。NURBS所完成的最终渲染大部分是由求值器完成的。但是有些情况下（例如，修剪曲线），NURBS函数使用平面多边形进行渲染。

本章主要由下面各节组成：

- 前提条件：描述学习本章内容所需要的一些预备知识，本节还提供了一些参考，帮助获得这方面的信息。
- 求值器：解释求值器的工作原理，并介绍如何通过控制求值器来使用适当的OpenGL函数。
- GLU的NURBS接口：描述用于创建、渲染和返回NURBS曲线和表面的分格化顶点的GLU函数。

12.1　前提条件

求值器可以创建以Bézier（或Bernstein）为基础的样条和表面。本章提供了这些函数的定义公式。但是，本章的讨论并不涉及推导过程，也不介绍它们的数学性质。如果想使用求值器绘制基于其他算法的曲线和表面，必须知道如何把它们转换为Bézier曲线和表面。另外，如果想使用求值器来渲染

Bézier曲线和表面的一部分，需要对它们的分割粒度做出决定。在做出这些决定时，需要在图像质量和渲染速度之间进行权衡。由于分割策略极为复杂，我们在此不作讨论。

同样，对NURBS进行详细讨论也超出了本书的范围。本章的目标是向已经理解这个主题的读者介绍GLU函数库的NURBS接口，并提供一些示例程序。在阅读本章之前，读者应该对NURBS控制点、节点序列和曲线修剪的概念有所了解。

如果尚不熟悉这些知识，可以参考如下书籍：

- Farin, Gerald E., Curves and Surfaces for Computer-Aided Geometric Design, Fourth Edition. San Diego, CA: Academic Press, 1996.
- Farin, Gerald E., NURB Curves and Surfaces: From Projective Geometry to Practical Use. Wellesley, MA: A. K. Peters Ltd., 1995.
- Farin, Gerald E., editor, NURBS for Curve and Surface Design, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1991.
- Hoschek, Josef, and Dieter Lasser, Fundamentals of Computer Aided Geometric Design. Wellesley, MA: A. K. Peters Ltd., 1993.
- Piegl, Les, and Wayne Tiller, The NURBS Book. New York, NY: Springer-Verlag, 1995.

注意：由于样条和样条表面的术语并未统一，因此本章使用的一些术语的含义可能与其他著作使用的术语有所不同。一般而言，OpenGL对术语的限制更为严格。例如，在OpenGL中，求值器总是以Bézier为基础。但是，在其他场合中，求值器可能只是表示同一个概念，它们所使用的可能是其他算法。

12.2 求值器

Bézier曲线是单变量的向量值函数：

$$\mathbf{C}(u) = [\ \mathbf{X}(u)\ \mathbf{Y}(u)\ \mathbf{Z}(u)\ ]$$

其中u在某个定义域（如[0,1]）中变化。Bézier曲面是双变量的向量值函数：

$$\mathbf{S}(u, v) = [\ \mathbf{X}(u, v)\ \mathbf{Y}(u, v)\ \mathbf{Z}(u, v)\ ]$$

其中u和v都在某个定义域中变化。它的输出并不一定是这里所显示的三维形式。我们可能希望为平面上的曲线或纹理坐标生成二维输出，或者可能想使用四维的输出来表示RGBA信息。在使用灰度的情况下，甚至可能使用一维的输出。

对于每个u（如果是曲面，则是u和v），C()（或S()）公式计算曲线（或曲面）上的一个点。为了使用求值器，首先需要定义函数C()或S()，然后启用它，并使用glEvalCoord1()或glEvalCoord2()函数代替glVertex()函数。按照这种方式，我们可以像使用其他顶点一样使用曲线或曲面上的顶点，例如用它们来构成点或直线。另外，有一些函数会自动生成一系列的顶点，组成一个沿u（或u和v）方向均匀排列的网格。一维和二维求值器非常相似，但是一维求值器更易于描述，因此我们首先讨论一维求值器。

12.2.1 一维求值器

本节首先提供了一个示例程序。这个程序使用一维求值器绘制了一条曲线。然后，我们描述用于控制求值器的函数和公式。

一维求值器例子：一条简单的Bézier曲线

示例程序12-1使用4个控制点绘制了一条三次Bézier曲线，如图12-1所示。



图12-1 Bézier曲线

示例程序12-1 具有4个控制点的Bézier曲线：bezcurve.c

```
GLfloat ctrlpoints [4][3] ={
        {-4.0,-4.0,0.0},{-2.0,4.0,0.0},
        {2.0,-4.0,0.0},{4.0,4.0,0.0}};
void init(void)
{
   glClearColor(0.0,0.0,0.0,0.0);
   glShadeModel(GL_FLAT);
   glMap1f(GL_MAP1_VERTEX_3,0.0,1.0,3,4,&ctrlpoints [0][0]);
   glEnable(GL_MAP1_VERTEX_3);
}

void display(void)
{
   int i;

   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0,1.0,1.0);
   glBegin(GL_LINE_STRIP);
      for (i =0;i <=30;i++)
         glEvalCoord1f((GLfloat)i/30.0);
   glEnd();
   /*The following code displays the control points as dots.*/
   glPointSize(5.0);
   glColor3f(1.0,1.0,0.0);
   glBegin(GL_POINTS);
      for (i= 0;i <4;i++)
        glVertex3fv(&ctrlpoints [i][0]);
   glEnd();
   glFlush();
}
void reshape(int w,int h)
{
   glViewport(0,0,(GLsizei)w,(GLsizei)h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <=h)
      glOrtho(-5.0,5.0,-5.0*(GLfloat)h/(GLfloat)w,
              5.0*(GLfloat)h/(GLfloat)w,-5.0,5.0);
   else
      glOrtho(-5.0*(GLfloat)w/(GLfloat)h,
              5.0*(GLfloat)w/(GLfloat)h,-5.0,5.0,-5.0,5.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

int main(int argc,char**argv)
```

```
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB);
    glutInitWindowSize(500,500);
    glutInitWindowPosition(100,100);
    glutCreateWindow(argv [0]);
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}
```

在这个例子中，这条三次Bézier曲线是通过4个控制点描述的。ctrlpoints[][]数组包含了这些控制点。这个数组是glMap1f()函数的一个参数，这个函数的其他参数是：

GL_MAP_VERTEX_3	指定了应用程序提供的三维控制点以及应该产生的三维顶点
0.0	参数u的低值
1.0	参数u的高值
3	从一个控制点跳到下一个控制点时应该推进多少个浮点值
4	样条的阶数，也就是度加1：在这个例子中，度为3（因为它是一条三次曲线）
&ctrlpoints[0][0]	指向第一个控制点的数据的指针

注意，第二个和第三个参数用于控制曲线的参数方程。当变量u从0.0变化到1.0时，曲线就从一端变化到另一端。可以调用glEnable()函数为三维顶点启用一维求值器。

这条曲线是由位于glBegin()和glEnd()之间的display()函数绘制的。由于启用了求值器，调用glEvalCoord1f()函数就像调用glVertex()函数一样，曲线上的一个顶点的坐标对应于输入参数u。

定义和计算一维求值器

度为n（或阶为$n+1$）的Bernstein多项式是通过下面的公式给出的：

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i}$$

如果P_i表示一组控制点（一维、二维、三维，甚至四维），那么下面这个公式：

$$C(u) = \sum_{i=0}^{n} B_i^n(u) P_i$$

表示一条u从0.0变化到1.0的Bézier曲线。为了表示一条与此相似，但是u从u_1变化到u_2（而不是从0.0到1.0）的Bézier曲线，可以像按如下公式进行计算：

$$C\left(\frac{u-u_1}{u_2-u_1}\right)$$

glMap1()函数使用上面这些公式定义一维求值器。

---

void **glMap1**{fd}(GLenum *target*, *TYPE u1*, *TYPE u2*, GLint *stride*, GLint *order*, const *TYPE* **points*);

定义了一维求值器。*target*参数指定了控制点表示什么，如表12-1所示，并因此决定了在*points*中应该提供多少个值。这些点可以表示顶点、RGBA颜

兼容性扩展

glMap1 and any tokens accepted by it

色数据、法线向量或纹理坐标。例如，如果这个参数是GL_MAP1_COLOR_4，这个求值器将生成沿四维（RGBA）颜色空间的一条曲线的颜色数据。也可以使用表12-1所列的参数值，在调用相应的求值器之前先启用它们。可以通过向**glEnable()**或**glDisable()**函数传递适当的值来启用或禁用求值器。

glMap1*()函数的接下来的两个参数u_1和u_2表示变量u的变化范围。变量*stride*是每个存储块之间单精度或双精度浮点数的数量。因此，它是一个控制点的起始位置和下一个控制点的起始位置之间的偏移量。

*order*参数的值就是度加1，它应该与控制点的数量一致。*points*参数指向第一个控制点的第一个坐标。对于这个示例程序所使用的数据结构，可以把*points*参数设置为：

```
(GLfloat *)(&ctlpoints[0].x)
```

表12-1 glMap1*()函数的控制点的类型

参数	含义	参数	含义
GL_MAP1_VERTEX_3	x, y, z顶点坐标	GL_MAP1_TEXTURE_COORD_1	s纹理坐标
GL_MAP1_VERTEX_4	x, y, z, w顶点坐标	GL_MAP1_TEXTURE_COORD_2	s, t纹理坐标
GL_MAP1_INDEX	颜色索引	GL_MAP1_TEXTURE_COORD_3	s, t, r纹理坐标
GL_MAP1_COLOR_4	R、G、B、A	GL_MAP1_TEXTURE_COORD_4	s, t, r, q纹理坐标
GL_MAP1_NORMAL	法线坐标		

可以一次使用多个求值器进行计算。例如，如果已经定义并启用了一个GL_MAP1_VERTEX_3和一个GL_MAP1_COLOR_4求值器，就可以调用glEvalCoord1()函数同时生成一个位置和一种颜色。对于两个顶点求值器，一次只能启用一个，尽管可能同时对它们进行了定义。类似地，只有一个纹理求值器可以处于活动状态。但是，在其他情况下，可以使用求值器来产生顶点、法线、颜色和纹理坐标数据的任何组合。如果定义和启用了超过一个的同种类型的求值器，实际使用的是那个具有最高维数的求值器。

可以使用glEvalCoord*1()计算一个已经定义并启用的一维求值器。

void **glEvalCoord1**{fd}(*TYPE u*);

void **glEvalCoord1**{fd}v(const *TYPE *u*);

使已启用的一维求值器执行计算。参数u是定义域坐标的值（如果是向量版本，是指向值的指针）。

兼容性扩展

glEvalCoord1

对于被求值的顶点，颜色、颜色索引、法线向量和纹理坐标的值是通过求值计算产生的。调用glEvalCoord*()函数并不会使用当前的颜色、颜色索引、法线向量和纹理坐标值。另外，glEvalCoord*()函数并不会对这些值进行修改。

定义均匀分布的一维坐标值

可以在glEvalCoord1()函数中使用任何u值，但目前最常用的是均匀分布的值，如示例程序12-1所示。为了获取均匀分布的值，可以用glMapGrid1*()函数定义一个一维网格，然后使用glEvalMesh1()函数应用这个网格。

void **glMapGrid1**{fd}(GLint *n*, *TYPE* u_1, *TYPE* u_2);

定义一个均匀分布的从u_1到u_2的网格，中间有n个阶段。

兼容性扩展

glMapGrid1

void **glEvalMesh1**(GLenum *mode*, GLint *p1*, GLint *p2*);

兼容性扩展
glEvalMesh1

对所有已启用的求值器应用当前定义的一维网格。*mode*参数的值可以是GL_POINT或GL_LINE，取决于想沿这条曲线绘制点还是绘制相连的直线。这个调用相当于为从*p1*到*p2*的每一步骤调用**glEvalCoord1()**函数，其中$0 \leqslant p1$，$p2 \leqslant n$。从编程的角度而言，它相当于：

```
glBegin(GL_POINTS);     /* OR glBegin(GL_LINE_STRIP); */
 for (i = p1; i <= p2; i++)
    glEvalCoord1(u1 + i*(u2-u1)/n);
glEnd();
```

例外的是情况是$i=0$或$i=n$，此时相当于以u_1和u_2为参数调用**glEvalCoord1()**函数。

12.2.2 二维求值器

二维求值器与一维求值器极为相似，区别仅仅是所有的函数都要考虑两个参数：*u*和*v*。点、颜色、法线或纹理坐标必须通过表面而不是曲线来提供。从数学的角度而言，Bézier表面的定义是由下面的公式给出的：

$$S(u,\ v)=\sum_{i=0}^{n}\sum_{j=0}^{m}B_i^n(u)B_j^m(v)P_{ij}$$

其中，P_{ij}的值是一组$m \cdot n$控制点，*Bi*函数是和一维求值器相同的Bernstein多项式。和前面一样，P_{ij}可以表示顶点、法线、颜色或纹理坐标。

使用二维求值器的过程与使用一维求值器相似：

1) 用glMap2*()函数定义一个或多个求值器。

2) 通过向glEnable()函数传递适当的值，启用这些求值器。

3) 调用这些求值器。可以使用两种方法，一种方法是在一对glBegin()和glEnd()之间调用glEvalCoord2()，另一种方法是用glMapGrid2()和glEvalMesh2()函数指定并应用一个网格。

定义和计算二维求值器

可以使用glMap2*()和glEvalCoord2*()函数定义并调用二维求值器。

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

兼容性扩展
glMap2 and any tokens accepted by it

*target*参数可以是表12-1列出的任何值之一，只不过是把其中的MAP1换成MAP2。和前面一样，我们也可以在**glEnable()**中使用这些值，启用对应的求值器。*u1*、*u2*、*v1*、*v2*分别提供了*u*和*v*的最小值和最大值。*ustride*和*vstride*参数表示这些值中不同设置之间的单精度或双精度浮点值的个数，允许用户从一个较大的数组中选择一个控制点子矩形。例如，如果数据是以下面的形式出现：

```
GLfloat ctlpoints[100][100][3];
```

并且想使用一个从ctlpoins[20][30]开始的4×4子集，可以选择*ustride*为100*3，选择*vstride*为3，而起始点*points*应该设置为&ctlpoints[20][30][0]。最后是阶参数，*uorder*和*vorder*可以不同，这可以提供更大的灵活性。例如，求值器可以在一个方向是三次的，在另一个方向是四次的。

兼容性扩展
glEvalCoord2

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

void **glEvalCoord2**{fd}v(const *TYPE *values*);

对已启用的二维求值器进行求值。参数*u*和*v*（如果是向量版本，则是指向u和v值的指针）是定义域坐标的值。如果启用了顶点求值器（GL_MAP2_VERTEX_3或GL_MAP2_ VERTEX_4），那么这个函数会随之生成表面的法线。如果通过向**glEnable()**函数传递GL_AUTO_NORMAL启用了自动法线生成，这条法线就会与被生成的顶点相关联。如果禁用了自动法线生成，OpenGL就使用对应的已被启用的法线图来产生一条法线。如果不存在这样的法线图，就使用当前法线。

二维求值器例子：一个Bézier曲面

示例程序12-2使用求值器绘制了一个线框Bézier表面，如图12-2所示。在这个例子中，这个表面是由9条各个方向的曲线绘制的。每条曲线都画成30段。为了构成完整的程序，可以在这个程序中添加示例程序12-1的reshape()和main()函数。

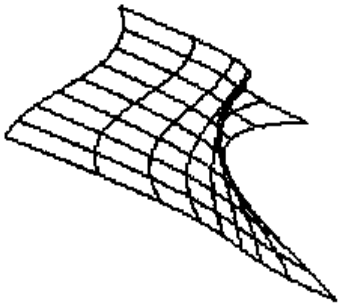


图12-2 Bézier曲面

示例程序12-2 Bézier曲面：bezsurf.c

```
GLfloat ctrlpoints [4][4][3] ={
    {{-1.5,-1.5,4.0},{-0.5,-1.5,2.0},
     {0.5,-1.5,-1.0},{1.5,-1.5,2.0}},
    {{-1.5,-0.5,1.0},{-0.5,-0.5,3.0},
     {0.5,-0.5,0.0},{1.5,-0.5,-1.0}},
    {{-1.5,0.5,4.0},{-0.5,0.5,0.0},
     {0.5,0.5,3.0},{1.5,0.5,4.0}},
    {{-1.5,1.5,-2.0},{-0.5,1.5,-2.0},
     {0.5,1.5,0.0},{1.5,1.5,-1.0}}
};

void display(void)
{
    int i,j;

    glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);
    glColor3f(1.0,1.0,1.0);
    glPushMatrix();
    glRotatef(85.0,1.0,1.0,1.0);
    for (j =0;j <=8;j++){
       glBegin(GL_LINE_STRIP);
    for (i =0;i <=30;i++)
       glEvalCoord2f((GLfloat)i/30.0,(GLfloat)j/8.0);
    glEnd();
    glBegin(GL_LINE_STRIP);
    for (i =0;i <=30;i++)
       glEvalCoord2f((GLfloat)j/8.0,(GLfloat)i/30.0);
    glEnd();
    }
    glPopMatrix();
    glFlush();
}
```

```
void init(void)
{
   glClearColor(0.0,0.0,0.0,0.0);
   glMap2f(GL_MAP2_VERTEX_3,0,1,3,4,
           0,1,12,4,&ctrlpoints [0][0][0]);
   glEnable(GL_MAP2_VERTEX_3);
   glMapGrid2f(20,0.0,1.0,20,0.0,1.0);
   glEnable(GL_DEPTH_TEST);
   glShadeModel(GL_FLAT);
}
```

定义均匀分布的二维坐标值

在二维求值器中，glMapGrid2*()和glEvalMesh2()函数与对应的一维版本的函数相似，只是现在必须同时包含u和v信息。

兼容性扩展
glMapGrid2
glEvalMesh2

void **glMapGrid2**{fd}(GLint *nu*, *TYPE u1*, *TYPE u2*,
GLint *nv*, *TYPE v1*, *TYPE v2*);

void **glEvalMesh2**(GLenum *mode*, GLint *i1*,
GLint *i2*, GLint *j1*, GLint *j2*);

定义二维图形网格，从*u1*到*u2*经历*nu*个均匀分布的阶段，从*v1*和*v2*经历*nv*个均匀分布的阶段（**glMapGrid2*()**），并把这个网格应用于所有被启用的求值器（**glEvalMesh2()**）。这两个函数与它们对应的一维版本的唯一区别是**glEvalMesh2()**函数的mode参数在GL_POINT和GL_LINE的基础上又增加了GL_FILL。GL_FILL使用四边形网格图元生成填充多边形。准确地说，**glEvalMesh2()**几乎与下面这三段代码相同（之所以说几乎相同，是因为当*i*等于*nu*或*j*等于*nv*时，这个参数就完全与*u2*或*v2*相同，而不是与*u1* + *nu**(*u2*−*u1*) / *nu*相同，也就是由于四舍五入的原因存在稍微的区别）。

```
glBegin(GL_POINTS);                    /* mode == GL_POINT */
for (i = nu1; i <= nu2; i++)
    for (j = nv1; j <= nv2; j++)
        glEvalCoord2(u1 + i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
glEnd();
```

或

```
for (i = nu1; i <= nu2; i++) {         /* mode == GL_LINE */
    glBegin(GL_LINES);
        for (j = nv1; j <= nv2; j++)
            glEvalCoord2(u1 + i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
    glEnd();
}
for (j = nv1; j <= nv2; j++) {
    glBegin(GL_LINES);
    for (i = nu1; i <= nu2; i++)
        glEvalCoord2(u1 + i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
    glEnd();
}
```

或

```
for (i = nu1; i < nu2; i++) {      /* mode == GL_FILL */
    glBegin(GL_QUAD_STRIP);
    for (j = nv1; j <= nv2; j++) {
        glEvalCoord2(u1 + i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
        glEvalCoord2(u1 + (i+1)*(u2-u1)/nu, v1+j*(v2-v1)/nv);
    glEnd();
}
```

示例程序12-3显示了使用glMapGrid2()和glEvalMesh2()函数绘制与示例程序12-2相同的Bézier表面所需要做的改动。这个程序把正方形的区域划分为大小一致的8×8网格，还添加了光照和着色效果，如图12-3所示。

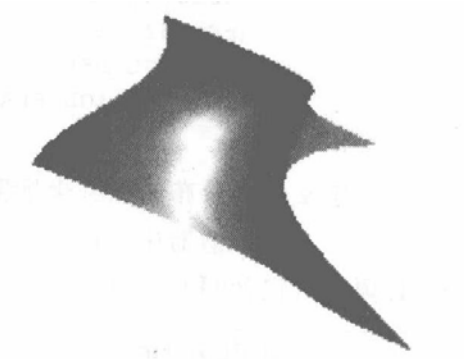


图12-3 用网格所绘制的具有光照、着色的Bézier表面

示例程序12-3 使用网格的具有光照、着色的Bézier表面：bezmesh.c

```
void initlights(void)
{
    GLfloat ambient [] ={0.2,0.2,0.2,1.0};
    GLfloat position [] ={0.0,0.0,2.0,1.0};
    GLfloat mat_diffuse [] ={0.6,0.6,0.6,1.0};
    GLfloat mat_specular [] ={1.0,1.0,1.0,1.0};
    GLfloat mat_shininess [] ={50.0};

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glLightfv(GL_LIGHT0,GL_AMBIENT,ambient);
    glLightfv(GL_LIGHT0,GL_POSITION,position);

    glMaterialfv(GL_FRONT,GL_DIFFUSE,mat_diffuse);
    glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
    glMaterialfv(GL_FRONT,GL_SHININESS,mat_shininess);
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    glRotatef(85.0,1.0,1.0,1.0);
    glEvalMesh2(GL_FILL,0,20,0,20);
    glPopMatrix();
    glFlush();
}

void init(void)
{
    glClearColor(0.0,0.0,0.0,0.0);
    glEnable(GL_DEPTH_TEST);
    glMap2f(GL_MAP2_VERTEX_3,0,1,3,4,
            0,1,12,4,&ctrlpoints [0][0][0]);
    glEnable(GL_MAP2_VERTEX_3);
    glEnable(GL_AUTO_NORMAL);
    glMapGrid2f(20,0.0,1.0,20,0.0,1.0);
    initlights();
}
```

12.2.3 使用求值器进行纹理处理

示例程序12-4同时启用了两个求值器：第一个求值器在一个Bézier表面（与示例程序12-3产生的表面相同）上生成三维点，第二个求值器则用于生成纹理坐标。在这个例子中，纹理坐标与表面的u和v坐标相同。但是，必须创建一个平面的Bézier表面来完成这个任务。

这个平面的Bézier表面是通过一个正方形定义的，它的各个角分别位于（0，0）、（0，1）、（1，0）和（1，1）。它在角（0，0）生成（0，0），在角（0，1）生成（0，1），接下来以此类推。由于它的阶是2（线性的度加上1），在点（u，v）处计算纹理产生纹理坐标（s，t）。这个程序还启用了顶点求值器，因此也将在绘制表面时生效（参见彩图19）。如果想在各个方向上使纹理重复3次，可以把texpts[][][]数组中的1.0修改为3.0。这样，这个程序所绘制的表面上将会出现纹理图像的9份拷贝。

示例程序12-4 使用求值器进行纹理处理：texturesurf.c

```
GLfloat ctrlpoints [4][4][3] ={
   {{-1.5,-1.5,4.0},{-0.5,-1.5,2.0},
    {0.5,-1.5,-1.0},{1.5,-1.5,2.0}},
   {{-1.5,-0.5,1.0},{-0.5,-0.5,3.0},
    {0.5,-0.5,0.0},{1.5,-0.5,-1.0}},
   {{-1.5,0.5,4.0},{-0.5,0.5,0.0},
    {0.5,0.5,3.0},{1.5,0.5,4.0}},
   {{-1.5,1.5,-2.0},{-0.5,1.5,-2.0},
    {0.5,1.5,0.0},{1.5,1.5,-1.0}}
};
GLfloat texpts [2][2][2] ={{{0.0,0.0},{0.0,1.0}},
                           {{1.0,0.0},{1.0,1.0}}};
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);
   glColor3f(1.0,1.0,1.0);
   glEvalMesh2(GL_FILL,0,20,0,20);
   glFlush();
}
#define imageWidth 64
#define imageHeight 64
GLubyte image [3*imageWidth*imageHeight];
void makeImage(void)
{
   int i,j;
   float ti,tj;

   for (i =0;i <imageWidth;i++){
      ti =2.0*3.14159265*i/imageWidth;
      for (j =0;j <imageHeight;j++){
         tj =2.0*3.14159265*j/imageHeight;
         image [3*(imageHeight*i+j)]=
               (GLubyte)127*(1.0+sin(ti));
         image [3*(imageHeight*i+j)+1] =
               (GLubyte)127*(1.0+cos(2*tj));
         image [3*(imageHeight*i+j)+2] =
               (GLubyte)127*(1.0+cos(ti+tj));
      }
```

```
    }
}

void init(void)
{
    glMap2f(GL_MAP2_VERTEX_3,0,1,3,4,
            0,1,12,4,&ctrlpoints [0][0][0]);
    glMap2f(GL_MAP2_TEXTURE_COORD_2,0,1,2,2,
            0,1,4,2,&texpts [0][0][0]);
    glEnable(GL_MAP2_TEXTURE_COORD_2);
    glEnable(GL_MAP2_VERTEX_3);
    glMapGrid2f(20,0.0,1.0,20,0.0,1.0);
    makeImage();
    glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_DECAL);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,
                    GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
                    GL_NEAREST);
    glTexImage2D(GL_TEXTURE_2D,0,3,imageWidth,imageHeight,0,
                    GL_RGB,GL_UNSIGNED_BYTE,image);
    glEnable(GL_TEXTURE_2D);
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_FLAT);
}
void reshape(int w,int h)
{
    glViewport(0,0,(GLsizei)w,(GLsizei)h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <=h)
       glOrtho(-4.0,4.0,-4.0*(GLfloat)h/(GLfloat)w,
               4.0*(GLfloat)h/(GLfloat)w,-4.0,4.0);
    else
       glOrtho(-4.0*(GLfloat)w/(GLfloat)h,
               4.0*(GLfloat)w/(GLfloat)h,-4.0,4.0,-4.0,4.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glRotatef(85.0,1.0,1.0,1.0);
}

int main(int argc,char**argv)
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB |GLUT_DEPTH);
    glutInitWindowSize(500,500);
    glutInitWindowPosition(100,100);
    glutCreateWindow(argv [0]);
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
```

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

12.3 GLU的NURBS接口

求值器是能够直接绘制曲线和表面的唯一OpenGL图元，虽然它们可以非常高效地用硬件来实现，但是我们在应用程序中通常还是通过高层函数库来访问它们。GLU函数库提供了NURBS接口，它们都是建立在OpenGL的求值器函数的基础之上。

12.3.1 一个简单的NURBS例子

如果理解了NURBS，编写OpenGL代码操纵NURBS曲线和表面还是相对较为容易的，即使涉及光照和纹理贴图。可以通过如下步骤绘制NURBS曲线和未修剪的NURBS表面。关于修剪表面的详细信息，请参阅第12.3.4节。

1) 如果想对NURBS表面应用光照，可以用GL_AUTO_NORMAL为参数调用glEnable()函数，自动生成法线向量（也可以自己计算法线向量）。

2) 使用gluNewNurbsRenderer()函数创建一个指向NURBS对象的指针。在创建自己的NURBS曲线或表面时，将会用到这个指针。

3) 如果需要，可以调用gluNurbsProperty()函数选择渲染值。例如，用于渲染NURBS对象的直线或多边形的最大数量。gluNurbsProperty()函数还可以启用一种模式，使用户可以在这种模式下通过回调接口提取分格化的几何数据。

4) 如果想在遇到错误时得到通知，可以调用gluNurbsCallback()函数（错误检查可能会稍稍降低性能，但我们还是强烈建议使用这种机制）。gluNurbsCallback()函数还可以注册用于提取分格化后的几何图形数据的函数。

5) 调用gluBeginCurve()或gluBeginSurface()函数，开始绘制曲线或表面。

6) 生成和渲染曲线或表面。至少需要调用1次gluNurbsCurve()或gluNurbsSurface()函数，以NURBS对象的控制点（有理或非有理的）、节点序列以及多项式基函数的阶数为参数。可以多次调用这些函数，指定法线向量和（或）纹理坐标。

7) 调用gluEndCurve()或gluEndSurface()函数完成曲线或表面的绘制。

示例程序12-5渲染了一个NURBS表面，它的形状像一座对称的小山，它的控制点范围从-3.0到3.0。基函数是一个三次B样条，但是节点序列的分布是不均匀的，每个终点的阶数为4，这就使得基函数在各个方向上都类似于Bézier曲线。这个表面进行了光照，具有暗灰色的散射光和白色的镜面亮点。图12-4显示了这个表面的线框形式。

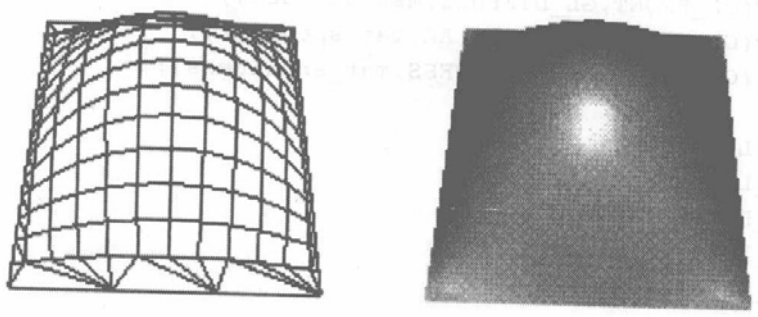


图12-4 NURBS表面

示例程序12-5 NURBS表面：surface.c

```
#ifndef CALLBACK
#define CALLBACK
#endif

GLfloat ctlpoints [4][4][3];
int showPoints = 0;

GLUnurbsObj *theNurb;

void init_surface(void)
{
   int u,v;
   for (u = 0;u <4;u++){
      for (v = 0;v <4;v++){
         ctlpoints [u][v][0] =2.0*((GLfloat)u -1.5);
         ctlpoints [u][v][1] =2.0*((GLfloat)v -1.5);

         if ((u ==1 ||u ==2)&&(v ==1 ||v ==2))
            ctlpoints [u][v][2]=3.0;
         else
            ctlpoints [u][v][2]=-3.0;
      }
   }
}

void CALLBACK nurbsError(GLenum errorCode)
{
   const GLubyte *estring;
   estring =gluErrorString(errorCode);
   fprintf(stderr,"Nurbs Error:%s \n ",estring);
   exit(0);
}

void init(void)
{
   GLfloat mat_diffuse [] ={0.7,0.7,0.7,1.0 };
   GLfloat mat_specular [] ={1.0,1.0,1.0,1.0 };
   GLfloat mat_shininess []={100.0 };

   glClearColor(0.0,0.0,0.0,0.0);
   glMaterialfv(GL_FRONT,GL_DIFFUSE,mat_diffuse);
   glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
   glMaterialfv(GL_FRONT,GL_SHININESS,mat_shininess);

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
   glEnable(GL_AUTO_NORMAL);
   glEnable(GL_NORMALIZE);

   init_surface();
```

```
    theNurb =gluNewNurbsRenderer();
    gluNurbsProperty(theNurb,GLU_SAMPLING_TOLERANCE,25.0);
    gluNurbsProperty(theNurb,GLU_DISPLAY_MODE,GLU_FILL);
    gluNurbsCallback(theNurb,GLU_ERROR,nurbsError);
}

void display(void)
{
    GLfloat knots [8] ={0.0,0.0,0.0,0.0,1.0,1.0,1.0,1.0};
    int i,j;

    glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);

    glPushMatrix();
    glRotatef(330.0,1.,0.,0.);
    glScalef(0.5,0.5,0.5);

    gluBeginSurface(theNurb);
    gluNurbsSurface(theNurb,
                    8,knots,8,knots,
                    4 * 3,3,&ctlpoints [0][0][0],
                    4,4,GL_MAP2_VERTEX_3);
    gluEndSurface(theNurb);

    if (showPoints){
       glPointSize(5.0);
       glDisable(GL_LIGHTING);
       glColor3f(1.0,1.0,0.0);
       glBegin(GL_POINTS);
       for (i =0;i <4;i++){
          for (j =0;j <4;j++){
             glVertex3f(ctlpoints [i][j][0],
                        ctlpoints [i][j][1],ctlpoints [i][j][2]);
         }
       }
       glEnd();
       glEnable(GL_LIGHTING);
    }
    glPopMatrix();
    glFlush();
}

void reshape(int w,int h)
{
    glViewport(0,0,(GLsizei)w,(GLsizei)h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0,(GLdouble)w/(GLdouble)h,3.0,8.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0,0.0,-5.0);
```

```
}
void keyboard(unsigned char key,int x,int y)
{
   switch (key){
      case 'c':
      case 'C':
         showPoints =!showPoints;
         glutPostRedisplay();
         break;
      case 27:
         exit(0);
         break;
      default:
         break;
   }
}
int main(int argc,char**argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB |GLUT_DEPTH);
   glutInitWindowSize(500,500);
   glutInitWindowPosition(100,100);
   glutCreateWindow(argv [0]);
   init();
   glutReshapeFunc(reshape);
   glutDisplayFunc(display);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
   return 0;
}
```

12.3.2 管理NURBS对象

如示例程序12-5所示，gluNewNurbsRender()函数返回一个新的NURBS对象，它的类型是一个指向GLUnurbsObj结构的指针。在使用其他任何NURBS函数之前，必须创建一个NURBS对象。在完成了对NURBS对象的使用之后，可以使用gluDeleteNurbsRenderer()函数释放它所占用的内存。

GLUnurbsObj* **gluNewNurbsRenderer**(void);

创建一个新的NURBS对象*nobj*，并返回一个指向这个新对象的指针。如果OpenGL无法为一个新的NURBS对象分配内存，这个函数就返回0。

---

void **gluDeleteNurbsRenderer**(GLUnurbsObj **nobj*);

销毁NURBS对象*nobj*。

控制NURBS的渲染属性

与NURBS对象相关联的一组属性可以影响物体的渲染方式。这些属性包括表面如何被光栅化（例如，填充还是线框）、显示还是返回分格化的顶点以及分格化的精度。

---

void **gluNurbsProperty**(GLUnurbsObj **nobj*, GLenum *property*,
GLfloat *value*);

控制NURBS对象*nobj*的属性。*property*参数指定了属性，它可以是GLU_DISPLAY_ MODE、GLU_NURBS_MODE、GLU_CULLING、GLU_SAMPLING_METHOD、GLU_SAMPLING_TOLERANCE、GLU_PARAMETRIC_TOLERANCE、GLU_U_STEP、GLU_V_STEP或GLU_AUTO_LOAD_MATRIX。*value*参数表示这个属性应该是什么值。

如果*property*参数是GLU_DISPLAY_MODE，它的默认值是GLU_FILL，也就是把表面渲染为多边形。如果显示模式属性为GLU_OUTLINE_POLYGON，只有分格化所创建的多边形轮廓才会被渲染。如果显示模式为GLU_OUTLINE_PATCH，它将渲染表面的轮廓，并对曲线进行修剪（参见第12.3.3节）。

如果*property*参数是GLU_NURBS_MODE，它控制被分格化的顶点是简单地进行渲染（如果*value*是默认值GLU_NURBS_RENDERER），还是允许返回分格化数据（如果*value*是GL_NURBS_TESSELLATOR）。有关这方面的细节，请参阅第12.3.3节中的“从NURBS分格化对象获取图元”。

如果这个NURBS对象完全位于视景体的外部，可以使用GLU_CULLING，不执行分格化操作，从而提高性能。把这个属性设置为GL_TRUE可以启用剔除功能（默认值为GL_FALSE）。

由于NURBS对象是作为图元渲染的，因此它是在参数值（u和v）不同的情况下进行采样的，并分解为很小的线段或多边形进行渲染。如果property属性为GLU_SAMPLING_METHOD，则value设置为GLU_PATH_LENTH（这是默认值）、GLU_PARAMETRIC_ERROR、GLU_DOMAIN_DISTANCE、GLU_OBJECT_PATH_LENGTH或GLU_OBJECT_PARAMETRIC_ERROR，这个参数指定了NURBS曲线或表面是如何分格化的。当value设置为GLU_PATH_LENTH时，表面在进行渲染时被分格化的多边形的边的最大长度（以像素为单位）不会超过GLU_ SAMPLING_TOLERANCE所指定的值。当value设置为GLU_PARAMETRIC_ERROR时，GLU_PARAMETRIC_TOLERANCE所指定的值就是被分格化的多边形和它们近似模拟的表面之间的最大距离（以像素为单位）。

如果*value*的值为GLU_OBJECT_PATH_LENGTH，它类似于GLU_PATH_LENTH，只是被分格化的图元的最大长度（GLU_ SAMPLING_TOLERANCE的值）是在物体空间中测量的，而不是以像素为单位。类似地，GLU_OBJECT_PARAMETRIC_ERROR与GLU_ PARAMETRIC_ERROR相似，区别在于GLU_OBJECT_PARAMETRIC_ERROR在测量分格化的多边形和它们近似模拟的表面之间的最大距离时是在物体空间中进行的，而不是以像素为单位。

当*value*设置为GLU_DOMAIN_DISTANCE时，应用程序以参数方程坐标的形式指定了u和v方向上每单位长度上具有多少个采样点，这两个值是用GLU_U_STEP和GLU_V_STEP指定的。

如果*property*属性是GLU_SAMPLING_TOLERANCE，并且采样方法是GLU_PATH_LENTH或GLU_OBJECT_PATH_LENGTH，value参数（使用像素作为单位或者物体空间的单位）用于控制分格化多边形的最大长度。例如，默认值50.0表示线段或多边形边缘的最大采样长度是50.0个像素或者是物体空间中的50.0个单位。如果*property*参数是GLU_ PARAMETRIC_TOLERANCE，并且采样方法是GLU_PARAMETRIC_ERROR或GLU_OBJECT_PARAMETRIC_ERROR，value参数（使用像素作为单位或者物体空间的单位）用于控制分格化多边形和它们近似模拟的表面之间的最大距离。对于这两种采样方法，GLU_PARAMETRIC_TOLERANCE的默认值是0.5。对于GLU_PARAMETRIC_ERROR而言，这意味着分格化的多边形与它们模拟的表面之间的最大距离不超过0.5个像素（对于GLU_OBJECT_PARAMETRIC_ERROR而言，0.5这个默认值并不具有相同的明显含义）。

如果采样方法是GLU_DOMAIN_DISTANCE，并且*property*参数是GLU_U_STEP或者GLU_V_STEP，则*value*以参数方程坐标的形式分别表示沿*u*或*v*方向的单位长度的采样点数量。GLU_U_STEP和GLU_V_STEP的默认值都是100。

GLU_AUTO_LOAD_MATRIX属性决定了应用程序是需要从OpenGL服务器下载投影矩阵、模型视图矩阵和视口（如果这个属性设置为GL_TRUE，也就是默认值），还是必须用**gluLoadSamplingMatrices()**函数提供这些矩阵（如果这个属性设置为GL_FALSE）。

注意：有些NURBS属性（GLU_NURBS_MODE和它的非默认值GLU_NURBS_TESSELLATOR，以及物体空间采样方法GLU_OBJECT_PATH_LENGTH或GLU_OBJECT_PARAMETRIC_ERROR）是在GLU 1.3版中引入的。在GLU 1.3之前，这些NURBS属性或者不存在，或者以厂商特定扩展的形式存在。因此，在使用这些属性之前，需要检查一下GLU版本。

void **gluLoadSamplingMatrices**(GLUnurbsObj **nobj*,
const GLfloat *modelMatrix*[16],
const GLfloat *projMatrix*[16],
const GLint *viewport*[4]);

如果禁用了GLU_AUTO_LOAD_MATRIX，模型视图矩阵、投影矩阵以及视口是通过**gluLoadSamplingMatrices()**函数指定的，为每个NURBS曲线或表面计算采样和剔除矩阵。

如果需要查询NURBS属性的当前值，可以使用gluGetNurbsProperty()函数。

void **gluGetNurbsProperty**(GLUnurbsObj **nobj*, GLenum *property*,
GLfloat **value*);

返回NURBS对象*nobj*的*property*属性的当前值，返回值放在value数组中。

处理NURBS错误

由于GLU共包含了37个NURBS函数特有的错误，因此注册一个错误回调函数是一个很好的做法。在程序遇到错误时，这个回调函数能够返回相关的错误信息。在示例程序12-5中，这个回调函数是用如下方法注册的：

```
gluNurbsCallback(theNurb, GLU_ERROR, (GLvoid (*)()) nurbsError);
```

void **gluNurbsCallback**(GLUnurbsObj **nobj*, GLenum *which*,
void (**fn*)(GLenum *errorCode*));

*which*是回调函数的类型。如果用于错误检查，它必须是GLU_ERROR。当一个NURBS函数检测到一个错误条件时，*fn*函数就会被调用，这个错误代码就是它的唯一参数。*errorCode*是37个错误条件（从GLU_NURBS_ERROR1到GLU_NURBS_ERROR37）之一。可以使用**gluErrorString()**来描述这些错误代码的含义。

在示例程序12-5中，nurbsError()函数是作为错误回调函数注册的：

```
void CALLBACK nurbsError(GLenum errorCode)
{
   const GLubyte *estring;

   estring = gluErrorString(errorCode);
```

```
    fprintf(stderr, "Nurbs Error: %s\n", estring);
    exit(0);
}
```

GLU 1.3添加了一些额外的NURBS回调函数，用于向程序返回后分格化的值（而不是渲染它们）。关于这些新的回调函数的更多信息，请参阅第12.3.3节中的“从NURBS分格化对象获取图元”。

12.3.3 创建NURBS曲线或表面

为了渲染NURBS表面，可以在一对gluBeginSurface()和gluEndSurface()函数之间调用gluNurbsSurface()函数。这对函数用于保存和恢复求值器的状态。

---

void **gluBeginSurface**(GLUnurbsObj **nobj*);

void **gluEndSurface**(GLUnurbsObj **nobj*);

在**gluBeginSurface()**之后，可以调用**gluNurbsSurface()**函数（一次或多次）定义表面的属性。在所有被调用的**gluEndSurface()**中，有且只有1个调用的表面类型是GL_MAP2_VERTEX_3或GL_MAP2_VERTEX_4，它用于生成顶点。也可以在**gluBeginSurface()**和**gluEndSurface()**之间进行NURBS表面的修剪（参见第12.3.4节）。

---

void **gluNurbsSurface**(GLUnurbsObj **nobj*, GLint *uknot_count*,
GLfloat **uknot*, GLint *vknot_count*, GLfloat **vknot*,
GLint *u_stride*, GLint *v_stride*, GLfloat **ctlarray*,
GLint *uorder*, GLint *vorder*, GLenum *type*);

描述NURBS表面*nobj*的顶点（或表面法线，或纹理坐标）。我们必须为*u*和*v*参数化方向指定几个值，例如节点序列（*uknot*和*vknot*）、节点数量（*uknot_count*和*vknot_count*）以及这个NURBS表面的多项式的阶数（*uorder*和*vorder*）。注意，这个函数并没有指定控制点的数量。但是，可以通过下述方法推导出控制点的数量：在每个参数化方向上，控制点的数量等于节点数减去阶数。整个表面的控制点数量等于两个参数化方向上控制点数量的乘积。*ctlarray*参数指向包含了控制点的数组。

最后一个参数*type*表示二维求值器类型。一般情况下，可以使用GL_MAP2_ VERTEX_3表示非有理类型的求值器，使用GL_MAP2_VERTEX_4表示有理类型的求值器。还可以使用其他类型，如用GL_MAP2_TEXTURE_COORD_*或GL_MAP2_NORMAL计算和分配纹理坐标或表面法线。例如，为了创建一个带光照（具有表面法线）和经过纹理处理的NURBS表面，可能需要执行下面的代码：

```
gluBeginSurface(nobj);
   gluNurbsSurface(nobj, ..., GL_MAP2_TEXTURE_COORD_2);
   gluNurbsSurface(nobj, ..., GL_MAP2_NORMAL);
   gluNurbsSurface(nobj, ..., GL_MAP2_VERTEX_3);
gluEndSurface(nobj);
```

*u_stride*和*v_stride*参数表示每个参数化方向上控制点之间的浮点值的数量。求值器的类型以及求值器的阶数都会影响*u_stride*和*v_stride*的值。在示例程序12-5中，*u_stride*是12（4*3），因为每个顶点有3个坐标（由GL_MAP2_VERTEX_3设置），v参数化方向有4个控制点。*v_stride*为3，因为每个顶点有3个坐标，并且v控制点在内存中是紧密相邻的。

绘制NURBS曲线与绘制NURBS表面类似，区别在于现在只需要对一个参数u进行计算，而不必对参数v进行计算。同样，为了绘制NURBS曲线，也必须在一对gluBeginCurve()和gluEndCurve()函数

之间调用绘制曲线的函数。

void **gluBeginCurve**(GLUnurbsObj **nobj*);

void **gluEndCurve**(GLUnurbsObj **nobj*);

在**gluBeginCurve()**之后，可以调用**gluNurbsCurve()**函数（一次或多次）定义这条曲线的属性。在所有被调用的**gluNurbsCurve()**中，有且只有1个调用的曲线类型是GL_MAP1_VERTEX_3或GL_MAP1_VERTEX_4，用于生成顶点。gluEndCurve()用于结束曲线的定义。

void **gluNurbsCurve**(GLUnurbsObj **nobj*, GLint *uknot_count*,
GLfloat **uknot*, GLint *u_stride*, GLfloat **ctlarray*,
GLint *uorder*, GLenum *type*);

定义NURBS曲线*nobj*。这个函数的参数与**gluNurbsSuface()**函数的对应参数具有相同的含义。注意，这个函数只需要1个节点序列以及NURBS对象的阶数的1个声明。如果这条曲线是在一对**gluBeginCurve()**和**gluEndCurve()**函数之间定义的，那么它的类型可以是任何有效的一维求值器类型（例如GL_MAP1_VERTEX_3或GL_MAP1_VERTEX_4）。

从NURBS分格化对象获取图元

在默认情况下，NURBS分格化对象把NURBS对象分解为几何直线和多边形，然后再对它们进行渲染。GLU 1.3增加了额外的回调函数，因此可以不再渲染后分格化的值，而是把它们返回给应用程序。

为此，可以调用gluNurbsProperty()函数，把GLU_NURBS_MODE属性的值设置为GLU_NURBS_TESSELLATOR。需要执行的其他步骤是调用几次gluNurbsCallback()函数，注册回调函数，以便于NURBS分格化对象调用它们。

void **gluNurbsCallback**(GLUnurbsObj **nobj*, GLenum *which*, void(**fn*)());

*nobj*是进行分格化的NURBS对象。*which*是用于标识回调函数的枚举值。如果GLU_NURBS_MODE属性的值设置为GLU_NURBS_TESSELLATOR，那么除了GLU_ERROR外，还可以使用12个回调函数。如果GLU_ NURBS_MODE属性设置为默认的GLU_NURBS_RENDERER，那么就只有GLU_ERROR是活动的。这12个回调函数的枚举值以及它们的原型如下：

GLU_NURBS_BEGIN	void **begin**(GLenum *type*);
GLU_NURBS_BEGIN_DATA	void **begin**(GLenum *type*, void **userData*);
GLU_NURBS_TEXTURE_COORD	void **texCoord**(GLfloat **tCrd*);
GLU_NURBS_TEXTURE_COORD_DATA	void **texCoord**(GLfloat **tCrd*, void **userData*);
GLU_NURBS_COLOR	void **color**(GLfloat **color*);
GLU_NURBS_COLOR_DATA	void **color**(GLfloat **color*, void **userData*);
GLU_NURBS_NORMAL	void **normal**(GLfloat **nml*);
GLU_NURBS_NORMAL_DATA	void **normal**(GLfloat **nml*, void **userData*);
GLU_NURBS_VERTEX	void **vertex**(GLfloat **vertex*);
GLU_NURBS_VERTEX_DATA	void **vertex**(GLfloat **vertex*, void **userData*);
GLU_NURBS_END	void **end**(void);
GLU_NURBS_END_DATA	void **end**(void **userData*);

为了更改回调函数，可以用新的回调函数为参数调用gluNurbsCallback()函数。为了删除一个回调函数而不是替换它，可以在调用gluNurbsCallback()函数时为相应的函数传递一个NULL指针。在这些回调函数中，有6个回调函数允许用户向它传递数据。为了指定用户数据，必须调用gluNurbsCallbackData()。

void **gluNurbsCallbackData**(GLUnurbsObj **nobj*,
void **userData*);

*nobj*是需要分格化的NURBS对象。*userData*是需要传递给回调函数的数据。

随着分格化的进行，这些回调函数的调用方式类似于调用OpenGL函数glBegin()、glTexCoord*()、glColor*()、glNormal*()、glVertex*()和glEnd()。当程序处于GLU_NURBS_ TESSELLATOR模式时，曲线或表面并不是直接渲染的，可以捕捉顶点数据，把它们当作参数传递给回调函数。

为了演示这些新的回调函数，示例程序12-6和12-7显示了surfpoints.c程序（这是前面的surface.c程序的修改版本）的部分代码。示例程序12-6显示了init()函数。这个函数创建NURBS对象，设置GLU_NURBS_MODE以执行分格化，并注册了回调函数。

示例程序12-6 注册NURBS分格化回调函数：surfpoints.c

```
void init(void)
{
/*only a portion of init()shown here--several lines deleted */
  theNurb =gluNewNurbsRenderer();
  gluNurbsProperty(theNurb,GLU_NURBS_MODE,
                   GLU_NURBS_TESSELLATOR);
  gluNurbsProperty(theNurb,GLU_SAMPLING_TOLERANCE,100.0);
  gluNurbsProperty(theNurb,GLU_DISPLAY_MODE,GLU_FILL);
  gluNurbsCallback(theNurb,GLU_ERROR,nurbsError);
  gluNurbsCallback(theNurb,GLU_NURBS_BEGIN,beginCallback);
  gluNurbsCallback(theNurb,GLU_NURBS_VERTEX,vertexCallback);
  gluNurbsCallback(theNurb,GLU_NURBS_NORMAL,normalCallback);
  gluNurbsCallback(theNurb,GLU_NURBS_END,endCallback);
}
```

示例程序12-7显示了surfpoints.c的回调函数。在这些回调函数中，printf()语句执行诊断功能，显示分格化对象返回的是什么渲染指令和顶点数据。另外，经过分格化的数据被重新递交给管线，以进行正常的渲染。

示例程序12-7 NURBS分格化回调函数：surfpoints.c

```
void CALLBACK beginCallback(GLenum whichType)
{
   glBegin(whichType);/*resubmit rendering directive */
   printf("glBegin(");
   switch (whichType){/*print diagnostic message */
      case GL_LINES:
         printf("GL_LINES)\n");break;
      case GL_LINE_LOOP:
         printf("GL_LINE_LOOP)\n");break;
      case GL_LINE_STRIP:
         printf("GL_LINE_STRIP)\n");break;
      case GL_TRIANGLES:
         printf("GL_TRIANGLES)\n");break;
```

```
        case GL_TRIANGLE_STRIP:
           printf("GL_TRIANGLE_STRIP)\n");break;
        case GL_TRIANGLE_FAN:
           printf("GL_TRIANGLE_FAN)\n");break;
        case GL_QUADS:
           printf("GL_QUADS)\n");break;
        case GL_QUAD_STRIP:
           printf("GL_QUAD_STRIP)\n");break;
        case GL_POLYGON:
           printf("GL_POLYGON)\n");break;
        default:
           break;
     }
}
void CALLBACK endCallback()
{
   glEnd();/*resubmit rendering directive */
   printf("glEnd()\n");
}

void CALLBACK vertexCallback(GLfloat *vertex)
{
   glVertex3fv(vertex);/*resubmit tessellated vertex */
   printf("glVertex3f (%5.3f,%5.3f,%5.3f)\n",
          vertex [0],vertex [1] ,vertex [2]);
}
void CALLBACK normalCallback(GLfloat *normal)
{
   glNormal3fv(normal);/*resubmit tessellated normal */
   printf("glNormal3f (%5.3f,%5.3f,%5.3f)",
          normal [0],normal [1] ,normal [2]);
}
```

12.3.4 修剪NURBS表面

为了用OpenGL创建经过修剪的表面，一开始执行的步骤与创建未修剪的表面相同。在调用gluBeginSurface()和gluNurbsSurface()之后，但在调用gluEndSurface()之前，可以调用gluBeginTrim()函数对表面进行修剪。

---

void **gluBeginTrim**(GLUnurbsObj **nobj*);

void **gluEndTrim**(GLUnurbsObj **nobj*);

标识一个修剪环路的开始和结束。修剪环路是一组有方向的修剪曲线段的集合（形成一条闭合曲线），定义了一个NURBS表面的边界。

可以创建两种类型的修剪曲线：用gluPwlCurve()函数创建一条分段的线性曲线或者用giuNurbsCurve()函数创建一条NURBS曲线。分段的线性曲线看上去并不像我们平常所说的“曲线”，因为它只是一系列的直线。用于修剪的NURBS曲线必须位于参数化空间 (u, v) 的一个单位正方形内。NURBS修剪曲线的类型通常是GLU_MAP1_TRIM2。在较为少见的情况下，它的类型也可以是GLU_MAP1_TRIM3，这种NURBS曲线在二维齐次空间 (u', v', w') 中是由 $(u, v) = (u'/w', v'/w')$ 描述的。

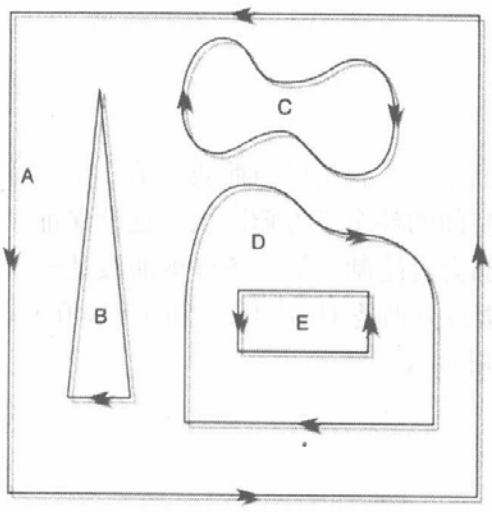
void **gluPwlCurve**(GLUnurbsObj **nobj*, GLint *count*, GLfloat **array*,
GLint *stride*, GLenum *type*);

描述了一条分段的线性修剪曲线，用于NURBS对象*nobj*。这条曲线上有*count*个点，它们是在*array*数组中提供的。*type*可以是GLU_MAP1_TRIM_2（最常见）或GLU_MAP1_TRIM_3（(u, v, w)齐次参数空间）。*type*参数的值决定了*stride*是2还是3。*stride*表示*array*数组中两个相邻顶点之间的浮点值的个数。

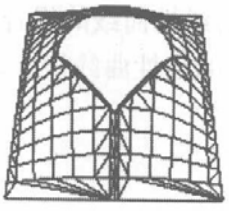
我们需要考虑修剪曲线的方向（也就是说，它们是顺时针方向还是逆时针方向），以确信已经包含了所需的表面部分。可以想象沿着这条曲线行走，曲线左边的任何东西都被保留，曲线右边的任何东西都被修剪掉。例如，如果根据一条逆时针方向的修剪环路进行修剪，这个环路内部的所有东西都被保留下来。如果根据两条互不相交的逆时针修剪环路（不存在相交的内部区域）进行修剪，这两个环路内部的所有东西都被保留。如果使用的修剪环路由一条逆时针方向的环路以及它内部的两条顺时针环路组成，这个修剪区域就会在中间形成两个洞。最外层的修剪环路必须是逆时针方向的。通常，可以使用一条包围整个单位正方形的修剪环路包含所有的东西，其结果就相当于不使用任何修剪的默认情况。

修剪曲线必须闭合并且互不相交。可以任意组合修剪曲线，只要修剪曲线能够形成环路。可以嵌套修剪曲线，创建空间中漂浮的“岛屿”。但是，必须确保修剪曲线的方向是正确的。例如，如果用两条嵌套的逆时针曲线指定修剪区域，就会产生错误，因为这两条曲线之间的区域在一条曲线的左侧，并在另一条曲线的右侧，因此无法决定应该保留这个区域还是修剪这个区域。图12-5显示了一些有效的修剪曲线组合。

图12-6显示了和图12-4相同的小山，但是它使用修剪曲线（由分段的线性曲线和NURBS曲线组成）进行了修剪。创建这幅图像的程序与示例程序12-5相似，区别在于示例程序12-8所显示的函数。




```
gluBeginSurface();
gluNurbsSurface(...);
gluBeginTrim();
gluPwlCurve(...); /* A */
gluEndTrim();
gluBeginTrim();
gluPwlCurve(...); /* B */
gluEndTrim();
gluBeginTrim();
gluNurbsCurve(...); */ C */
gluEndTrim();
gluBeginTrim();
gluNurbsCurve(...); */ D */
gluPwlCurve(...); /* D */
gluEndTrim();
gluBeginTrim();
gluPwlCurve(...); /* E */
gluEndTrim();
gluEndSurface();
```



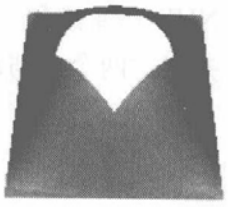


图12-5 参数化修剪曲线

图12-6 经过修剪的NURBS表面

示例程序12-8 修剪NURBS表面：trim.c

```
void display(void)
{
   GLfloat knots [8] ={0.0,0.0,0.0,0.0,1.0,1.0,1.0,1.0};
   GLfloat edgePt [5][2] =/*counter clockwise */
```

```
        {{0.0,0.0},{1.0,0.0},{1.0,1.0},{0.0,1.0},
         {0.0,0.0}};
    GLfloat curvePt [4][2] =/*clockwise */
        {{0.25,0.5},{0.25,0.75},{0.75,0.75},{0.75,0.5}};
    GLfloat curveKnots [8] =
         {0.0,0.0,0.0,0.0,1.0,1.0,1.0,1.0};
    GLfloat pwlPt [3][2] =/*clockwise */
        {{0.75,0.5},{0.5,0.25},{0.25,0.5}};

    glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    glRotatef(330.0,1.,0.,0.);
    glScalef(0.5,0.5,0.5);

    gluBeginSurface(theNurb);
    gluNurbsSurface(theNurb,8,knots,8,knots,
                    4 *3,3,&ctlpoints [0][0][0],
                    4,4,GL_MAP2_VERTEX_3);
gluBeginTrim(theNurb);
    gluPwlCurve(theNurb,5,&edgePt [0][0] ,2,
                GLU_MAP1_TRIM_2);
gluEndTrim(theNurb);
gluBeginTrim(theNurb);
    gluNurbsCurve(theNurb,8,curveKnots,2,
                  &curvePt [0][0],4,GLU_MAP1_TRIM_2);
    gluPwlCurve(theNurb,3,&pwlPt [0][0],2,
                GLU_MAP1_TRIM_2);
    gluEndTrim(theNurb);
    gluEndSurface(theNurb);

    glPopMatrix();
    glFlush();
}
```

在示例程序12-8中，gluBeginTrim()和gluEndTrim()函数之间定义了一条修剪曲线。第一条修剪曲线的顶点由数组edgePt[][]定义，以逆时针方向包围了参数化空间的整个正方形区域。这就保证了只要它的内部没有使用顺时针的修剪曲线，整个区域内的物体都会被绘制。第二条修剪曲线是一条NURBS曲线和一条分段的线性曲线的组合。这条NURBS曲线的端点分别是 (0.9, 0.5) 和 (0.1, 0.5)，并在这两个点与那条分段的线性曲线接合，形成一条闭合的顺时针曲线。

第13章　选择和反馈

本章目标

- 创建应用程序，以允许用户选择屏幕上的一块区域或者挑选屏幕上绘制的一个物体。
- 使用OpenGL的反馈模式获取渲染计算的结果。

注意：在OpenGL 3.1中，本章介绍的所有技术和函数都已经废弃删除了。即便其中一些功能是GLU库的一部分，它还是依赖于已经从核心OpenGL库删除的那些功能。

有些图形应用程序只是简单地绘制静态的二维或三维物体图像。另外还有一些应用程序允许用户标识屏幕上的物体，并且移动、更改或删除这些物体（或者进行其他操作）。支持这类具有交互式操作的应用程序本来就是OpenGL的设计目标之一。由于屏幕上绘制的物体一般会经历多次旋转、移动和透视变换，因此在三维场景中判断用户所选择的是哪个物体会比较困难。为了帮助实现这个目的，OpenGL提供了选择机制，它可以告诉我们哪个物体位于窗口中一个特定区域的内部。可以利用这种机制，并使用一个特殊的工具函数，确定用户在某个区域中所指定的是哪个物体，或者允许用户通过光标挑选物体。

选择实际上是OpenGL的一种操作模式，反馈则是另一种模式。在反馈模式中，我们使用图形硬件和OpenGL来执行通常的渲染计算。但是，OpenGL并不是根据计算结果在屏幕上绘制图像，而是返回（或反馈）绘图信息。例如，如果想在绘图仪而不是屏幕上绘制三维物体，可以在反馈模式下绘制物体、收集绘图命令，然后把它们转换为绘图仪可以理解的命令。

无论是在选择还是反馈模式下，绘图信息总是返回到应用程序而不是像渲染模式那样发送到帧缓冲区。因此，在选择或反馈模式下，屏幕冻结，不会进行绘图。在这两种模式下，颜色、深度、模板和累积，缓冲区的内容并不会受到影响。本章将分别介绍这两种模式。

- “选择”：讨论了如何使用选择模式以及相关的函数，允许应用程序的用户挑选屏幕上所绘制的物体。
- “反馈”：描述了如何获取与屏幕上所绘制的物体有关的信息，以及这些信息是如何进行格式化的。

13.1　选择

一般情况下，当我们打算使用OpenGL的选择机制时，首先把整个场景绘制到帧缓冲区，然后进入选择模式，并对场景进行重绘。但是，在进入选择模式之后，帧缓冲区的内容将不会修改，除非退出了选择模式。当退出选择模式时，OpenGL返回与视景体相交的图元列表（记住，视景体是由当前的模型视图矩阵、投影矩阵以及所有用户定义的裁剪平面决定的，如第3章所述）。与视景体相交的每个图元都会产生一个选择点击。这个图元列表实际上是以点击记录（hit record）的形式返回的，它包含了一个整型数组names以及相关的数据，对应于名字栈中的当前内容。在选择模式下，当调用用于绘制图元的函数时，可以把这些名称加载到名字栈中，从而完成对名字栈的创建。因此，当名称列表返回时，可以用它来确定屏幕上的哪些图元可能被用户选择。

除了这种选择机制之外，OpenGL还提供了一个工具函数，把绘图限制在视口中的一个小型区域内，它在某些情况下可以简化选择过程。一般情况下，可以使用这个函数确定哪个物体靠近光标，由此判断用户挑选的是哪个物体（还可以通过指定用户定义的裁剪平面限制选择区域。记住，这些平面是在全局空间中有效，而不是在屏幕空间中有效）。由于挑选是一种特殊的选择，因此我们在本章中首先描述选择，然后再讨论挑选。

13.1.1 基本步骤

为了使用选择机制，需要执行下面这些步骤：

1) 用glSelectBuffer()函数指定用于返回点击记录的数组。

2) 用glRenderMode()指定GL_SELECT，进入选择模式。

3) 使用glInitNames()和glPushName()对名字栈进行初始化。

4) 定义用于选择的视景体。通常，这个视景体与最初用于绘制场景的视景体不同，因此很可能需要用glPushMatrix()和glPopMatrix()函数保存和恢复当前的变换状态。

5) 交替调用绘制图元的函数和操纵名字栈的函数，为每个相关的图元分配一个适当的名称。

6) 退出选择模式，并处理返回的选择数据（点击记录）。

下面两段内容描述了glSelectBuffer()和glRenderMode()函数。在下一小节中，我们介绍用于操作名字栈的函数。

void **glSelectBuffer**(GLsizei *size*, GLuint **buffer*);

指定用于返回选择数据的数组。*buffer*参数是一个无符号整数类型的数组指针，它所指向的数组用于存放选择数据。*size*表示这个数组可以存储的值的最大数量。在进入选择模式之前，需要调用**glSelectBuffer()**。

兼容性扩展
glSelectBuffer
glRenderMode

GLint **glRenderMode**(GLenum *mode*);

控制应用程序是处于渲染模式、选择模式还是反馈模式。*mode*参数可以是GL_RENDER（默认）、GL_SELECT、GL_FEEDBACK。在使用一个不同的*mode*参数再次调用这个函数之前，应用程序将一直保持这次调用所设置的模式。在进入选择模式之前，必须调用**glSelectBuffer()**函数指定用于保存选择数据的数组。类似地，在进入反馈模式之前，必须调用**glFeedBackBuffer()**函数指定用于保存反馈数据的数组。如果当前的渲染模式（而不是*mode*参数）是GL_SELECT或GL_FEEDBACK，**glRenderMode()**的返回值才有意义。这个返回值是选择点击记录的数量或者反馈数组所存储的值的个数（分别在各自的模式下）。如果这个函数返回负值，表示选择或反馈数组已溢出。可以向**glGetIntegerv()**函数传递GL_RENDER_MODE参数查询当前是什么模式。

13.1.2 创建名字栈

如前一小节所述，名字堆栈形成了返回选择信息的基础。为了创建名字堆栈，首先需要用glInitNames()函数对它进行初始化，这个函数只是简单地清除这个堆栈。然后，在调用绘图函数时，向这个堆栈添加整数类型的名称。正如读者所预想的那样，用于操纵名字堆栈的函数包括一个把名称压入到名字堆栈中的函数（glPushName()），也包括一个从名字堆栈弹出名称的函数（glPopName()），还包括一个用不同的名称替换名字堆栈顶部元素的函数（glLoadName()）。示例程序13-1显示了一些相关的名字堆栈操纵函数。

示例程序13-1 创建名字栈

```
glInitNames();
glPushName(0);

glPushMatrix();/*save the current transformation state */

    /*create your desired viewing volume here */

   glLoadName(1);
   drawSomeObject();
   glLoadName(2);
   drawAnotherObject();
   glLoadName(3);
   drawYetAnotherObject();
   drawJustOneMoreObject();
glPopMatrix();/*restore the previous transformation state*/
```

在这个例子中，前两个被绘制的物体有它们各自的名称，第三和第四个物体共享一个名称。在这种设置下，如果第三个或第四个物体导致了一次选择点击，只会返回一个点击记录。在处理点击记录时，如果不想区分它们，可以让多个物体共享同一个名字。

兼容性扩展
glInitNames
glPushName
glPopName
glLoadName

void **glInitNames**(void);

清除名字堆栈，使它成为空堆栈。

void **glPushName**(GLuint *name*);

把*name*压入到名字堆栈中。如果试图压入到一个已满的堆栈，就会产生GL_STACK_OVERFLOW错误。名字堆栈的深度因不同的OpenGL实现而异，但它至少能够容纳64个名称。可以用GL_NAME_STACK_DEPTH为参数调用**glGetIntegerv()**函数，查询名字堆栈的深度。

void **glPopName**(void);

从名字堆栈弹出一个名称。试图从空堆栈弹出名称会导致GL_STACK_UNDERFLOW错误。

void **glLoadName**(GLuint *name*);

用*name*替换名字堆栈顶部的那个值。如果堆栈是空的，也就是刚刚调用了**glInitNames()**之后，**glLoadName()**将会产生GL_INVALID_OPERATION错误。为了避免出现这种情况，如果名字堆栈最初是空的，应该至少调用1次**glPushName()**函数在堆栈中压入一些东西，然后再调用**glLoadName()**。

如果应用程序并不是处于选择模式下，对glPushName()、glPopName()和glLoadName()的调用会被忽略。读者可能会发现，在绘图代码中使用这些函数，然后在选择和渲染模式下使用相同的绘图代码，可以简化应用程序的代码。

13.1.3 点击记录

在选择模式下，与视景体相交的图元会导致一次选择点击。在执行用于操纵名字堆栈的函数或调用glRenderMode()函数之后，如果出现了一次点击，OpenGL就会在选择数组中写入一个点击记录。按照这种处理方式，共享相同名称的对象（例如被多个图元使用的对象）不会产生多条点击记录。另

外，在调用glRenderMode()之前，点击记录并不能保证被写入到选择数组中。

注意：除了几何图元之外，glRasterPos()或glWindowPos()所产生的有效光栅位置坐标也会导致选择点击。另外，在选择模式下，如果启用了剔除功能，并且多边形被处理并剔除，就不会产生点击。

每个点击记录都由4个项目组成，按顺序分别如下所示：

- 当点击发生时名字堆栈中的名称数量。
- 自上一个点击记录之后，与视景体相交的图元的所有顶点的最小和最大窗口坐标z值。这两个值的范围在[0，1]之内，它们都与$2^{32}-1$相乘，然后四舍五入到最接近的无符号整数。
- 在点击发生时名字堆栈的内容，从最底部的元素开始。

在进入选择模式时，OpenGL会对一个指针进行初始化，使它指向选择数组的起始位置。每次当一条点击记录写入到这个数组时，这个指针也会相应地进行更新。如果写入一条点击记录会导致数组中值的数量超过glSelectBuffer()函数所指定的size参数，OpenGL将在这个数组中写入尽可能多的记录，并设置一个溢出标志。当使用glRenderMode()函数退出选择模式时，这个函数会返回它所写入的点击记录的数量（在溢出的情况下，还可能包含一条不完整的记录），并清除名字堆栈，重置溢出标志和堆栈指针。如果设置了溢出标志，这个函数的返回值为-1。

13.1.4 一个选择例子

示例程序13-2在屏幕上绘制了4个三角形（1个绿色、1个红色和2个黄色，通过调用drawTrinagle()函数绘制）和1个表示视景体的线框盒子（通过调用drawViewVolume()函数绘制）。然后，在选择模式下，再次对这些三角形进行渲染（selectObjects()）。对应的点击记录是在processHits()函数中处理的，并且选择数组也被打印出来。第一个三角形产生一次点击，第二个三角形不产生点击，第三个和第四个三角形共同产生一次点击。

示例程序13-2 选择示例：select.c

```
void drawTriangle(GLfloat x1,GLfloat y1,GLfloat x2,
     GLfloat y2,GLfloat x3,GLfloat y3,GLfloat z)
{
   glBegin(GL_TRIANGLES);
   glVertex3f(x1,y1,z);
   glVertex3f(x2,y2,z);
   glVertex3f(x3,y3,z);
   glEnd();
}

void drawViewVolume(GLfloat x1,GLfloat x2,GLfloat y1,
                    GLfloat y2,GLfloat z1,GLfloat z2)
{
   glColor3f(1.0,1.0,1.0);
   glBegin(GL_LINE_LOOP);
   glVertex3f(x1,y1,-z1);
   glVertex3f(x2,y1,-z1);
   glVertex3f(x2,y2,-z1);
   glVertex3f(x1,y2,-z1);
   glEnd();
   glBegin(GL_LINE_LOOP);
```

```
    glVertex3f(x1,y1,-z2);
    glVertex3f(x2,y1,-z2);
    glVertex3f(x2,y2,-z2);
    glVertex3f(x1,y2,-z2);
    glEnd();

    glBegin(GL_LINES); /*4 lines */
    glVertex3f(x1,y1,-z1);
    glVertex3f(x1,y1,-z2);
    glVertex3f(x1,y2,-z1);
    glVertex3f(x1,y2,-z2);
    glVertex3f(x2,y1,-z1);
    glVertex3f(x2,y1,-z2);
    glVertex3f(x2,y2,-z1);
    glVertex3f(x2,y2,-z2);
    glEnd();
}

void drawScene(void)
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(40.0,4.0/3.0,1.0,100.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(7.5,7.5,12.5,2.5,2.5,-5.0,0.0,1.0,0.0);
    glColor3f(0.0,1.0,0.0);/*green triangle     */
    drawTriangle(2.0,2.0,3.0,2.0,2.5,3.0,-5.0);
    glColor3f(1.0,0.0,0.0);/*red triangle       */
    drawTriangle(2.0,7.0,3.0,7.0,2.5,8.0,-5.0);
    glColor3f(1.0,1.0,0.0);/*yellow triangles   */
    drawTriangle(2.0,2.0,3.0,2.0,2.5,3.0,0.0);
    drawTriangle(2.0,2.0,3.0,2.0,2.5,3.0,-10.0);
    drawViewVolume(0.0,5.0,0.0,5.0,0.0,10.0);
}

void processHits(GLint hits,GLuint buffer [])
{
    unsigned int i,j;
    GLuint names,*ptr;

    printf("hits =%d \n ",hits);
    ptr =(GLuint *)buffer;
    for (i =0;i <hits;i++){/*for each hit */
       names =*ptr;
       printf(" number of names for hit =%d \n ",names);ptr++;
       printf(" z1 is %%g;" ,(float)*ptr/0x7fffffff);ptr++;
       printf(" z2 is %%g \n " ,(float)*ptr/0x7fffffff);ptr++;
       printf(" the name is ");
       for (j =0;j <names;j++){/*for each name */
          printf("%d ",*ptr);ptr++;
```

```
        }
        printf("\n ");
    }
}

#define BUFSIZE 512

void selectObjects(void)
{
    GLuint selectBuf [BUFSIZE];
    GLint hits;

    glSelectBuffer(BUFSIZE,selectBuf);
    (void)glRenderMode(GL_SELECT);

    glInitNames();
    glPushName(0);

    glPushMatrix();
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0,5.0,0.0,5.0,0.0,10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glLoadName(1);
    drawTriangle(2.0,2.0,3.0,2.0,2.5,3.0,-5.0);
    glLoadName(2);
    drawTriangle(2.0,7.0,3.0,7.0,2.5,8.0,-5.0);
    glLoadName(3);
    drawTriangle(2.0,2.0,3.0,2.0,2.5,3.0,0.0);
    drawTriangle(2.0,2.0,3.0,2.0,2.5,3.0,-10.0);
    glPopMatrix();
    glFlush();

    hits = glRenderMode(GL_RENDER);
    processHits(hits,selectBuf);
}
void init(void)
{
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_FLAT);
}

void display(void)
{
    glClearColor(0.0,0.0,0.0,0.0);
    glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);
    drawScene();
    selectObjects();
    glFlush();
}
```

```
int main(int argc,char**argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB |GLUT_DEPTH);
   glutInitWindowSize(200,200);
   glutInitWindowPosition(100,100);
   glutCreateWindow(argv [0]);
   init();
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;
}
```

13.1.5 挑选

作为前一节描述的处理过程的延伸，可以通过选择模式来确定一个物体是否被挑选。为了实现这个目的，可以使用一个特殊的挑选矩阵，并协同使用投影矩阵，把绘图限制在视口的一个小区域内，一般是在靠近光标的位置。然后，允许某些形式的输入（例如点击鼠标按钮），对选择模式进行初始化。在确立了选择模式并使用了特殊的挑选矩阵之后，在靠近光标位置处绘制物体就会导致选择点击。因此，挑选一般是用于判断哪些物体是在靠近光标的位置绘制的。

挑选的设置基本上和常规的选择相同，主要有下面几点区别：

- 挑选通常是由输入设备触发的。在下面这段示例代码中，按下鼠标左键能够调用一个执行挑选操作的函数。
- 可以使用工具函数gluPickMatrix()，把当前的投影矩阵与一个特殊的挑选矩阵相乘。这个函数应该在乘以标准的投影矩阵（例如调用gluPerspective()或glOrtho()）之前调用。我们很可能需要首先保存投影矩阵的内容，因此操作顺序大致如下所示：

```
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
gluPickMatrix(...);
gluPerspective, glOrtho, gluOrtho2D, or glFrustum
   /* ... draw scene for picking ; perform picking ... */
glPopMatrix();
```

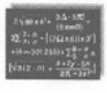
第14.6节描述了另一种完全不同的执行挑选的方法。这个技巧使用颜色值标识物体的不同成分。

---

void **gluPickMatrix**(GLdouble *x*, GLdouble *y*, GLdouble *width*,
GLdouble *height*, GLint *viewport[4]*);

创建一个投影矩阵，把绘图限制在视口中的一个小区域内，并把这个矩阵与当前的投影矩阵相乘。挑选区域的中心是窗口坐标 (x, y)，一般就是光标的位置。*width*和*height*定义了屏幕坐标下这个挑选区域的大小（可以把宽度和高度看成是挑选设备的敏感度）。*viewport[]*表示当前的视口边界，可以通过下面的调用来获取：

```
glGetIntegerv(GL_VIEWPORT, GLint *viewport);
```



高级话题

gluPickMatrix()创建的矩阵的净效果是把裁剪区域变换为单位立方体$-1 \leqslant (x, y, z) \leqslant 1$

(或$-w \leqslant (wx, wy, wz) \leqslant w$)。挑选矩阵有效地执行一次正交变换，把裁剪区域映射到单位立方体上。由于转换是任意的，因此可以让挑选作用于不同类型的区域，例如，可以把它作用于经过旋转的窗口矩形部分。在有些情况下，读者可能会发现，指定用户定义的裁剪平面来定义挑选区域会更简单一些。

示例程序13-3演示了一个简单的挑选例子。它还说明了如何使用多个名称标识一个图元的不同成分（在这个例子中，就是一个被选择物体的行和列）。这个程序绘制了一个3×3的方块网格，每个方块使用一种不同的颜色。board[3][3]数组维护每个方块的当前蓝色成分的数量。当用户点击鼠标左键时，pickSquares()函数就会被调用，以确认鼠标点击的是哪个方块。这个程序使用两个名称来标识网格中的每个方块，一个表示行，另一个表示列。另外，当用户按下鼠标左键时，光标位置下的所有方块的颜色都会改变。

示例程序13-3 挑选例子：picksquare.c

```
int board [3][3];/*amount of color for each square */

/*Clear color value for every square on the board */
void init(void)
{
   int i,j;
   for (i =0;i <3;i++)
      for (j =0;j <3;j ++)
        board [i][j] =0;
   glClearColor(0.0,0.0,0.0,0.0);
}

void drawSquares(GLenum mode)
{
   GLuint i,j;
   for (i =0;i <3;i++){
      if (mode ==GL_SELECT)
         glLoadName(i);
      for (j =0;j <3;j ++){
        if (mode ==GL_SELECT)
          glPushName(j);
        glColor3f((GLfloat)i/3.0,(GLfloat)j/3.0,
                  (GLfloat)board [i][j]/3.0);
        glRecti(i,j,i+1,j+1);
        if (mode ==GL_SELECT)
           glPopName();
      }
   }
}

/*processHits prints out the contents of the
 *selection array.
 */
void processHits(GLint hits,GLuint buffer [])
{
   unsigned int i,j;
   GLuint ii,jj,names,*ptr;
```

```
    printf("hits =%d \n ",hits);
    ptr =(GLuint *)buffer;
    for (i =0;i <hits;i++){/*for each hit */
       names =*ptr;
       printf(" number of names for this hit =%d \n " ,names);
          ptr++;
       printf(" z1 is %%g;",(float)*ptr/0x7fffffff);ptr++;
       printf(" z2 is %%g \n " ,(float)*ptr/0x7fffffff);ptr++;
       printf(" names are ");
       for (j =0;j <names;j++){/*for each name */
          printf("%d ",*ptr);
          if (j ==0)/*set row and column */
              ii =*ptr;
          else if (j ==1)
             jj =*ptr;
          ptr++;
       }
       printf("\n ");
       board [ii][jj] =(board [ii][jj] +1)%3;
    }
}

#define BUFSIZE 512

void pickSquares(int button,int state,int x,int y)
{
   GLuint selectBuf [BUFSIZE];
   GLint hits;
   GLint viewport [4];
   if (button !=GLUT_LEFT_BUTTON ||state !=GLUT_DOWN)
      return;

   glGetIntegerv(GL_VIEWPORT,viewport);

   glSelectBuffer(BUFSIZE,selectBuf);
   (void)glRenderMode(GL_SELECT);

   glInitNames();
   glPushName(0);

   glMatrixMode(GL_PROJECTION);
   glPushMatrix();
   glLoadIdentity();
/*create 5x5 pixel picking region near cursor location */
   gluPickMatrix((GLdouble)x,(GLdouble)(viewport [3]-y),
                 5.0,5.0,viewport);
   gluOrtho2D(0.0,3.0,0.0,3.0);
   drawSquares(GL_SELECT);

   glMatrixMode(GL_PROJECTION);
   glPopMatrix();
   glFlush();
```

```
    hits =glRenderMode(GL_RENDER);
    processHits(hits,selectBuf);
    glutPostRedisplay();
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    drawSquares(GL_RENDER);
    glFlush();
}

void reshape(int w,int h)
{
    glViewport(0,0,w,h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0,3.0,0.0,3.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
int main(int argc,char**argv)
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB);
    glutInitWindowSize(100,100);
    glutInitWindowPosition(100,100);
    glutCreateWindow(argv [0]);
    init();
    glutMouseFunc(pickSquares);
    glutReshapeFunc(reshape);
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

使用多个名字和层次式模型的挑选

可以使用多个名称来选择场景中一个层次式物体的不同部分。例如，如果渲染的是一条汽车装配线，我们可能想让用户通过移动鼠标来挑选装配线上第三辆汽车的左前轮的第三颗螺钉。可以使用不同的名称来标识每个层次：哪辆汽车、哪个轮子、哪颗螺钉。另举一个例子，我们可以用一个名称表示分子模型中的其中一个分子，用另一个名称表示这个分子中的其中一个原子。

示例程序13-4对示例程序3-4进行了修改，这个程序绘制了一辆具有4个相同轮子的汽车，每个轮子都有5颗相同的螺钉。这个程序对以前的代码进行了修改，根据物体的层次来操纵名字堆栈。

示例程序13-4 创建多个名称

```
draw_wheel_and_bolts()
{
    long i;

    draw_wheel_body();
```

```
    for (i =0;i <5;i++){
        glPushMatrix();
            glRotate(72.0*i,0.0,0.0,1.0);
            glTranslatef(3.0,0.0,0.0);
            glPushName(i);
                draw_bolt_body();
            glPopName();
        glPopMatrix();
    }
}
draw_body_and_wheel_and_bolts()
{
    draw_car_body();
    glPushMatrix();
        glTranslate(40,0,20);/*first wheel position*/
        glPushName(1);       /*name of wheel number 1 */
            draw_wheel_and_bolts();
        glPopName();
    glPopMatrix();
    glPushMatrix();
        glTranslate(40,0,-20);/*second wheel position */
        glPushName(2);/*name of wheel number 2 */
            draw_wheel_and_bolts();
        glPopName();
    glPopMatrix();

    /*draw last two wheels similarly */
}
```

示例程序13-5使用了示例程序13-4的函数绘制了三辆不同的汽车，编号分别为1、2和3。

示例程序13-5 使用多个名称

```
draw_three_cars()
{
    glInitNames();
    glPushMatrix();
        translate_to_first_car_position();
        glPushName(1);
            draw_body_and_wheel_and_bolts();
        glPopName();
    glPopMatrix();

    glPushMatrix();
     translate_to_second_car_position();
     glPushName(2);
         draw_body_and_wheel_and_bolts();
     glPopName();
glPopMatrix();

glPushMatrix();
    translate_to_third_car_position();
    glPushName(3);
        draw_body_and_wheel_and_bolts();
```

```
    glPopName();
    glPopMatrix();
}
```

假如执行了挑选，下面是一些可能出现的名字堆栈返回值以及它们的解释。在这些例子中，最多只返回一条点击记录。另外，*d1*和*d2*都是深度值。

2 *d1 d2* 2 1	汽车2，轮子1
1 *d1 d2* 3	汽车3（车体）
3 *d1 d2* 1 1 0	汽车1上的轮子1的螺钉0
空	挑选的位置在所有汽车之外

最后一个解释假定螺钉和轮子并没有占据相同的挑选区域。用户可能会同时挑选轮子和螺钉，从而产生两次点击。如果接收了多次点击，必须决定处理哪一次点击，也许要通过深度值来确定挑选最靠近观察点的物体。下一小节将更深入地探讨深度值的使用。

挑选和深度值

示例程序13-6说明了如何在挑选时使用深度值来确定需要挑选的物体。这个程序在常规的渲染模式下绘制了3个重叠的矩形。当用户按下鼠标左键时，pickRects()函数就会被调用。这个函数返回光标位置、进入选择模式、对名字堆栈进行初始化、把挑选矩阵与当前的正投影矩阵相乘。如果用户按下鼠标时光标位于这些矩形的上面，这个程序就会为每个矩形产生选择点击。最后，这个程序对选择缓冲区的内容进行检查，确认挑选区域中的哪些有名称的物体靠近光标。

这个程序中的矩形是用不同的深度值（z值）绘制的。由于这个程序只使用了一个名称标识所有3个矩形，因此它只记录了一次点击。但是，如果有超过一个的矩形被挑选，这个唯一的点击就具有不同的最小和最大z值。

示例程序13-6 在挑选时使用深度值

```
void init(void)
{
   glClearColor(0.0,0.0,0.0,0.0);
   glEnable(GL_DEPTH_TEST);
   glShadeModel(GL_FLAT);
   glDepthRange(0.0,1.0);/*The default z mapping */
}
void drawRects(GLenum mode)
{
   if (mode ==GL_SELECT)
      glLoadName(1);
   glBegin(GL_QUADS);
   glColor3f(1.0,1.0,0.0);
   glVertex3i(2,0,0);
   glVertex3i(2,6,0);
   glVertex3i(6,6,0);
   glVertex3i(6,0,0);
   glEnd();
   if (mode ==GL_SELECT)
      glLoadName(2);
   glBegin(GL_QUADS);
   glColor3f(0.0,1.0,1.0);
   glVertex3i(3,2,-1);
```

```
    glVertex3i(3,8,-1);
    glVertex3i(8,8,-1);
    glVertex3i(8,2,-1);
    glEnd();
    if (mode ==GL_SELECT)
      glLoadName(3);
    glBegin(GL_QUADS);
    glColor3f(1.0,0.0,1.0);
    glVertex3i(0,2,-2);
    glVertex3i(0,7,-2);
    glVertex3i(5,7,-2);
    glVertex3i(5,2,-2);
    glEnd();
}

void processHits(GLint hits,GLuint buffer [])
{
    unsigned int i,j;
    GLuint names,*ptr;

    printf("hits =%d \n ",hits);
    ptr =(GLuint *)buffer;
    for (i =0;i <hits;i++){/*for each hit */
       names =*ptr;
       printf("number of names for hit =%d \n " ,names);ptr++;
       printf("z1 is %g;",(float)*ptr/0x7fffffff);ptr++;
       printf(" z2 is %%g \n " ,(float)*ptr/0x7fffffff);ptr++;
       printf(" the name is ");
       for (j =0;j <names;j++){/*for each name */
          printf("%d ",*ptr);ptr++;
       }
       printf("\n ");
    }
}

#define BUFSIZE 512

void pickRects(int button,int state,int x,int y)
{
    GLuint selectBuf [BUFSIZE];
    GLint hits;
    GLint viewport [4];

    if (button !=GLUT_LEFT_BUTTON ||state !=GLUT_DOWN)
       return;
    glGetIntegerv(GL_VIEWPORT,viewport);

    glSelectBuffer(BUFSIZE,selectBuf);
    (void)glRenderMode(GL_SELECT);

    glInitNames();
    glPushName(0);
```

```
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
/* create 5x5 pixel picking region near cursor location */
    gluPickMatrix((GLdouble)x,(GLdouble)(viewport [3] -y),
                  5.0,5.0,viewport);
    glOrtho(0.0,8.0,0.0,8.0,-0.5,2.5);
    drawRects(GL_SELECT);
    glPopMatrix();
    glFlush();

    hits =glRenderMode(GL_RENDER);
    processHits(hits,selectBuf);
}
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);
    drawRects(GL_RENDER);
    glFlush();
}

void reshape(int w,int h)
{
    glViewport(0,0,(GLsizei)w,(GLsizei)h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0,8.0,0.0,8.0,-0.5,2.5);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

int main(int argc,char **argv)
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB |GLUT_DEPTH);
    glutInitWindowSize(200,200);
    glutInitWindowPosition(100,100);
    glutCreateWindow(argv [0]);
    init();
    glutMouseFunc(pickRects);
    glutReshapeFunc(reshape);
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```



尝试一下

- 对示例程序13-6进行修改，增加一些glPushName()调用。这样，在发生选择点击时能够有多个名字被压入到堆栈中。选择缓冲区的内容将会是什么？
- 在默认情况下，glDepthRange()函数把z值映射到[0.0，1.0]的范围。尝试对glDepthRange()的值进行修改，并观察这些修改是如何影响选择缓冲区所返回的z值的。

13.1.6 编写使用选择的程序的一些建议

绝大多数允许用户对几何物体进行交互性编辑的程序都提供了一种机制，允许用户挑选可以进行编辑的项或项组。对于二维绘图程序（例如，文本编辑器、页面设置程序以及电路设计程序），执行自己的挑选计算可能比使用OpenGL的挑选机制方便得多。寻找二维物体的边框，并把它们组织成层次式的数据结构以提高搜索速度常常比较方便。例如，在包含几百万个矩形的VLSL电路程序中使用OpenGL风格的挑选机制可能相对较慢。但是，如果这些矩形在屏幕中对齐，使用简单的边框信息进行挑选可能会快得多。并且，这种方法所使用的代码也更为简单。

另举一个例子，由于只有几何图形会产生点击，因此我们可能想创建自己的方法来挑选文本。设置当前光栅位置是一种几何操作，但它只是在当前光栅位置创建一个可挑选的点，通常是在文本的左下角。如果编辑器需要在字符串内部操纵单个的字符，就必须使用其他的挑选机制。在挑选模式下，可以在每个字符周围绘制更少的矩形，但是把文本作为一种特殊情况来处理显然会容易很多。

如果决定使用OpenGL的挑选机制，应该对程序以及它所使用的数据结构进行组织，使OpenGL无论在选择模式下还是在正常的渲染模式下都更容易绘制正确的物体。这样，当用户进行了挑选时，可以在挑选操作中使用相同的数据结构，把物体显示在屏幕上。另外，还需要考虑是否允许用户挑选多个物体。其中一种方法是为每个项存储1个位，表示它是否已经被选择（但是，使用这种方法要求对所有的物体进行遍历，以便找出被选择的项）。读者可能会发现，维护一个指针列表（每个指针指向一个被选择的项）会非常有用，它可以加速搜索操作。为每个项保存1个选择位也是一种很好的方法，因为在绘制整幅图像时，可能想把被选择的项画成不同的样子（例如，使用不同的颜色，或者给它们加上一个选择框）。最后，还需要考虑用于选择的用户界面，可能需要允许用户：

- 选择一个项。
- 快速选择一组项。
- 在选择缓冲区中增加一个项。
- 在当前的选择缓冲区中添加一组被选择的物体。
- 从选择缓冲区中删除一个项。
- 在一组重叠的物体中选择一个项。

对于二维绘图程序，一种典型的解决方法可能像下面这样：

1) 所有的选择都是通过点击鼠标左键来完成的。在接下来的步骤中，光标表示与鼠标相关联的光标，按钮表示鼠标的左键。

2) 点击一个物体就选择了它，并同时释放当前选择的其他所有物体。如果光标位于多个物体的顶部，最小的那个物体被选择。在三维中，可以使用其他许多策略来消除选择歧义。

3) 在没有物体的地方按下鼠标，摁住并拖动光标，然后再释放鼠标按钮，将会选中一个屏幕对齐的矩形中的所有物体。这个矩形的位置和大小是由按下鼠标和释放鼠标时的位置决定的。这个操作称为连片选择（sweep selection），在连片选择区域之外的物体都会成为未选中状态。我们必须做出决定，物体必须是整个部分都位于这个连片区域内时才被选中，还是只要有部分位于这个区域就可以被选中。选择“完全位于”策略通常更好一些。

4) 如果Shift键被按下，并且用户点击了一个当前尚未选中的物体，这个物体就会添加到选择列表中。如果这个被点击的物体原先已经被选中，那么它将从选择列表中删除。

5) 如果是在按下Shift键的情况下执行了连片选择，连片区域内的物体便添加到当前的选择列表中。

6) 在物体极端聚集的区域，进行连片选择通常非常困难。当鼠标按钮按下时，光标可能会位于

某个物体的上面，在正常情况下这个物体会被选中。可以执行连片选择，但在用户界面中，当鼠标点击一个物体并移动光标时所发生的最常见的行为通常是拖动这个物体。为了解决这个问题，可以通过在拖曳光标的同时按下某个键（例如Alt）来强制进行连片选择。通过这种处理方法，下面的操作顺序就构成了一次连片选择：按下Alt键，拖动光标，释放鼠标按钮。在这个操作中，当按钮被按下时位于光标下面的物体将被忽略。

7) 如果在这种类型的连片选择操作过程中还按下了Shift键，包围这个连片区域的物体会添加到当前的选择列表中。

8) 最后，如果用户在多个物体上进行点击，只有一个物体会被选择。如果在光标没有移动（或者移动量不超过1个像素）的情况下，用户在同一个地方再次进行点击，将使原先选中的物体取消选择，并选择光标下的另一个物体。通过在同一个点进行重复点击，可以选择光标下的任何一个物体。

在特殊的情况下可以应用不同的规则。在文本编辑器中，很可能并不用担心每个字符上面是不是还有别的字符，因此选择多个字符总是等于选择文档中连续的字符。因此，只需要选择第一个和最后一个字符，便可以确定需要选择的所有字符。在处理文本时，处理选择的更好方法往往是确认字符之间的位置，而不是字符本身。这样，如果起始选择和最后选择所选择的是同一对字符，这就相当于进行了一次空的选择。它还允许把光标放在文档的第一个字符之前和最后一个字符之后，而不需要使用特殊的代码。

在三维编辑器中，可以提供一些方法对所选择的物体进行旋转和缩放，因此可能并无必要进行复杂的循环式选择。另一方面，在三维中进行选择通常非常困难，因为屏幕上光标的位置并不能表示它的深度。

13.2 反馈

反馈和选择具有一个相似之处：在这两种模式下，不会产生任何像素，并且屏幕被冻结。在这两种模式下，不会发生实际的绘图。反之，与渲染图元有关的信息被发送到应用程序。选择和反馈模式的关键区别在于它们所返回的信息。在选择模式下，一些已分配的名称返回到一个整型数组中。在反馈模式下，与经过变换的图元有关的一些信息返回到一个浮点型数组中。发送到反馈数组的值包括一些标记（指定了被处理和变换的图元类型，如点、直线、多边形、图像或位图），然后是图元的顶点、颜色或其他数据。这些返回值都经过了完整的光照和视图转换。为了进入反馈模式，可以用GL_FEEDBACK为参数调用glRenderMode()函数。

下面是进入和退出反馈模式的一些步骤：

1) 调用glFeedbackBuffer()函数，指定用于保存反馈信息的数组。这个函数的参数描述了数据的类型以及写入到数组的数据量。

2) 以GL_FEEDBACK为参数调用glRenderMode()函数，进入反馈模式（在这个步骤中，可以忽略glRenderMode()函数的返回值）。在此之后，在退出反馈模式之前，图元不会通过光栅化产生像素，帧缓冲区的内容也不会发生变化。

3) 绘制图元。在发布绘图命令时，可以多次调用glPassThrough()函数在返回的反馈数据中插入标记，以方便对反馈数据进行解析。

4) 如果想返回到常规的绘图模式，可以用GL_RENDER为参数调用glRenderMode()函数，退出反馈模式。glRenderMode()函数的返回值就是反馈数组中所存储的值的数量。

5) 对反馈数组中的数据进行解析。

void **glFeedbackBuffer**(GLsizei *size*, GLenum *type*, GLfloat **buffer*);

建立一个用于保存反馈数据的缓冲区：*buffer*是一个数组指针，指向包含反馈数据的数组。*size*参数表示这个数组可以存储的值的最大数量。*type*参数描述了反馈数组中每个反馈顶点的信息，表13-1列出了可能出现的值以及具体的含义。必须在进入反馈模式后才能调用**glFeedbackBuffer()**函数。在表13-1中，k在颜色索引模式下为1，在RGBA模式下为4。

兼容性扩展
glFeedbackBuffer and all associated tokens

表13-1 glFeedbackBuffer()的type参数

type参数	坐标	颜色	纹理	总值
GL_2D	x, y	—	—	2
GL_3D	x, y, z	—	—	3
GL_3D_COLOR	x, y, z	k	—	$3 + k$
GL_3D_COLOR_TEXTURE	x, y, z	k	4	$7 + k$
GL_4D_COLOR_TEXTURE	x, y, z, w	k	4	$8 + k$

注意：如果支持多重纹理，反馈只返回纹理单位0的纹理坐标。

13.2.1 反馈数组

在反馈模式下，每个将要光栅化（如果光栅位置有效，也包括每次对glBitmap()、glDrawPixels()或glCopyPixels()函数的调用）的图元都会生成一块数据值，并复制到反馈数组中。数据值的数量是由glFeedbackBuffer()函数的type参数决定的，如表13-1所示。根据图元的类型，应该为这个参数设置适当的值：对于没有光照的二维或三维图元，使用GL_2D或GL_3D；对于具有光照的三维图元，使用GL_3D_COLOR；对于具有光照、并进行了纹理贴图的三维或四维图元，使用GL_3D_COLOR_TEXTURE或GL_4D_COLOR_TEXTURE。

每块反馈数据都是从一个表示图元类型的代码开始，紧接着是描述图元顶点以及相关信息的值。此外，它还写入了像素矩形的信息。另外，可以反馈数组返回显式创建的过渡标记，下一小节将详细解释这些标记。表13-2显示了反馈数组的语法。记住，表13-1已经描述了与每个返回顶点相关的数据。注意，一个多边形可能返回n个顶点。另外，反馈所返回的x、y和z坐标都是窗口坐标。如果返回了w，它就是在裁剪坐标下。对于位图和像素矩形，反馈所返回的坐标是当前光栅位置的坐标。在表13-2中，注意只有在重置了线段的直线点画模式之后，才会返回GL_LINE_RESET_TOKENS。

表13-2 反馈数组的语法

图元类型	代　码	相关的数据
点	GL_POINT_TOKEN	顶点
直线	GL_LINE_TOKEN或GL_LINE_RESET_TOKEN	顶点顶点
多边形	GL_POLYGON_TOKEN	n顶点顶点…顶点
位图	GL_BITMAP_TOKEN	顶点
像素矩形	GL_DRAW_PIXEL_TOKEN或GL_COPY_PIXEL_TOKEN	顶点
过渡标记	GL_PASS_THROUGH_TOKEN	1个浮点值

13.2.2 在反馈模式下使用标记

反馈是在变换、光照、多边形剔除以及使用glPolygonMode()设置多边形模式之后发生的。当把一个由超过3条边组成的多边形分解为几个三角形（如果我们所使用的OpenGL实现通过进行这种分解法来执行多边形渲染）后，也可能会发生反馈。因此，我们可能难以理解接收到的反馈数据。为了便于对反馈数据进行解析，可以根据需要在绘图命令序列中添加对glPassThrough()函数的调用，插入一些过渡标记。例如，可以用过渡标记分隔不同图元返回的反馈值。这个函数把GL_PASS_THROUGH_TOKEN写入到反馈数组，紧随其后的就是通过这个函数的参数传递的浮点值。

兼容性扩展

glPassThrough

void **glPassThrough**(GLfloat *token*);

如果在反馈模式下调用这个函数，它会在写入到反馈数组的数据流中插入一个过渡标记。这个标记是由代码GL_PASS_THROUGH_TOKEN和一个浮点值*token*组成。如果不是在反馈模式下调用这个函数，不会产生任何效果。在**glBegin()**和**glEnd()**之间调用**glPassThrough()**函数会产生GL_INVALID_OPERATION错误。

13.2.3 一个反馈例子

示例程序13-7显示了反馈模式的应用。这个程序在常规的渲染模式下绘制一个具有光照的三维场景，然后进入反馈模式，并对场景进行重绘。由于这个程序所绘制的是具有光照的、未经纹理处理的三维物体，因此反馈数据的类型是GL_3D_COLOR。由于这个程序使用了RGBA模式，因此每个未裁剪的顶点都在反馈缓冲区中生成7个值：x, y, z, r, g, b和a。

在反馈模式下，这个程序绘制了两条直线，作为一条线段的一部分，并插入一个过渡标记。接着，它在（−100.0 −100.0 −100.0）位置上绘制一个点。这个位置位于正投影下的视景体的外部，因此不会在反馈数组中写入任何值。最后，这个程序又插入了另一个过滤标记，并绘制了另一个点。

示例程序13-7 反馈模式：feedback.c

```
void init(void)
{
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
}
void drawGeometry(GLenum mode)
{
   glBegin(GL_LINE_STRIP);
   glNormal3f(0.0,0.0,1.0);
   glVertex3f(30.0,30.0,0.0);
   glVertex3f(50.0,60.0,0.0);
   glVertex3f(70.0,40.0,0.0);
   glEnd();
   if (mode ==GL_FEEDBACK)
      glPassThrough(1.0);
   glBegin(GL_POINTS);
   glVertex3f(-100.0,-100.0,-100.0);/*will be clipped */
   glEnd();
   if (mode ==GL_FEEDBACK)
      glPassThrough(2.0);
   glBegin(GL_POINTS);
   glNormal3f(0.0,0.0,1.0);
```

```
    glVertex3f(50.0,50.0,0.0);
    glEnd();
    glFlush();
}

void print3DcolorVertex(GLint size,GLint *count,
                        GLfloat *buffer)
{
    int i;

    printf(" ");
    for (i =0;i <7;i++){
       printf("%4.2f ",buffer [size-(*count)]);
        *count =*count -1;
    }
    printf("\n ");
}
void printBuffer(GLint size,GLfloat *buffer)
{
    GLint count;
    GLfloat token;

    count =size;
    while (count){
       token =buffer [size-count];count--;
       if (token ==GL_PASS_THROUGH_TOKEN){
          printf("GL_PASS_THROUGH_TOKEN \n ");
          printf(" %%4.2f \n ",buffer [size-count]);
          count--;
       }
       else if (token ==GL_POINT_TOKEN){
          printf("GL_POINT_TOKEN \n ");
          print3DcolorVertex(size,&count,buffer);
       }
       else if (token ==GL_LINE_TOKEN){
          printf("GL_LINE_TOKEN \n ");
          print3DcolorVertex(size,&count,buffer);
          print3DcolorVertex(size,&count,buffer);
       }
       else if (token ==GL_LINE_RESET_TOKEN){
          printf("GL_LINE_RESET_TOKEN \n ");
          print3DcolorVertex(size,&count,buffer);
          print3DcolorVertex(size,&count,buffer);
       }
    }
}

void display(void)
{
    GLfloat feedBuffer [1024];
    GLint size;
```

```
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0,100.0,0.0,100.0,0.0,1.0);

    glClearColor(0.0,0.0,0.0,0.0);
    glClear(GL_COLOR_BUFFER_BIT);
    drawGeometry(GL_RENDER);

    glFeedbackBuffer(1024,GL_3D_COLOR,feedBuffer);
    (void)glRenderMode(GL_FEEDBACK);
    drawGeometry(GL_FEEDBACK);
    size =glRenderMode(GL_RENDER);
    printBuffer(size,feedBuffer);
}

int main(int argc,char**argv)
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB);
    glutInitWindowSize(100,100);
    glutInitWindowPosition(100,100);
    glutCreateWindow(argv [0]);
    init();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

运行这个程序将产生下面的输出：

```
GL_LINE_RESET_TOKEN
 30.00 30.00 0.00 0.84 0.84 0.84 1.00
 50.00 60.00 0.00 0.84 0.84 0.84 1.00
GL_LINE_TOKEN
 50.00 60.00 0.00 0.84 0.84 0.84 1.00
 70.00 40.00 0.00 0.84 0.84 0.84 1.00
GL_PASS_THROUGH_TOKEN
 1.00
GL_PASS_THROUGH_TOKEN
 2.00
GL_POINT_TOKEN
 50.00 50.00 0.00 0.84 0.84 0.84 1.00
```

因此，用这些绘图命令所绘制的线段将产生两个图元：

```
glBegin(GL_LINE_STRIP);
    glNormal3f(0.0, 0.0, 1.0);
    glVertex3f(30.0, 30.0, 0.0);
    glVertex3f(50.0, 60.0, 0.0);
    glVertex3f(70.0, 40.0, 0.0);
glEnd();
```

第一个图元从GL_LINE_RESET_TOKEN开始，这个标记表示这个图元是一条线段，并且对直线点画模式进行重置。第二个图元从GL_LINE_TOKEN开始，因此它也是一条线段，但是并没有对点

画模式进行重置，因此沿用原先那条线段所遗留的模式。这两条线段的每一个顶点都在反馈数组中产生7个值。注意，这两条线段的所有4个顶点的RGBA值都是（0.84，0.84，0.84，1.0），这是一种非常淡的灰色，并且具有最高的alpha值。这些颜色值是表面法线与光照参数的交互结果。

由于在第一个和第二个过滤标记之间并没有产生任何反馈数据，因此可以去掉前两次glPassThrough()调用之间所绘制的任何图元（它们将被裁剪出视景体）。最后，这个程序在（50.0，50.0，0.0）的位置绘制了一个点，与它相关的数据将被复制到反馈数组中。

注意：无论是在反馈模式还是选择模式下，物体的信息总是在任何片断测试之前返回。

因此，在反馈模式或选择模式下，那些无法通过裁剪测试、alpha测试或模板测试的物体仍然会对它们的数据进行处理和返回。



尝试一下

对示例程序13-7进行修改，观察它们是如何影响物体所返回的反馈值的。例如，可以修改glOrtho()的坐标值。可以修改光照变量，或者干脆取消光照并把反馈类型修改为GL_3D。或者，增加更多的图元，观察其他几何物体（例如填充多边形）对反馈数组所产生的影响。

第14章 OpenGL高级技巧

本章目标

和其他各章不同，本章并没有具体的学习目标。本章列出了一系列的主题，提供了一些思路，帮助用户编写应用程序。本章讨论的有些主题（例如错误处理）并不适合在其他章节内描述，但是它们的内容又太少，也不适合开辟专门的章节来讲述。

OpenGL是一些低层工具的集合。通过前面各章的学习，读者已经了解了这些工具。现在，可以利用这些工具实现更高层次的功能。本章提供了这种高层功能的一些例子。

本章讨论了基于OpenGL函数的各种技巧，展示了这些函数可以实现的一些用途。这些用途可能并不是读者轻易能够想到的。本章所显示的例子并没有特定的顺序，相互之间也没有什么关联。阅读本章的基本方法是翻阅本节的标题，然后转到自己感兴趣的主题。为了方便起见，下面列出各节的标题，并对它们进行了简单的解释。

注意：本书提供的大多数示例程序都是完整的程序，可以编译和运行。但是，在本章中，所有的例子都不是完整的程序。为了使它们能够运行，必须做一些额外的工作。

- 错误处理：介绍如何检查OpenGL的错误情况。
- OpenGL版本：描述如何了解OpenGL实现的有关细节，包括OpenGL版本。如果需要编写能够在OpenGL早期版本上运行的应用程序，本节的内容可能非常重要。
- 标准的扩展：讨论一些技巧，用于确认和使用OpenGL标准的扩展（可能是厂商专门提供的扩展，也可能是OpenGL体系结构审核委员会批准的扩展）。
- 实现半透明效果：解释如何使用多边形点画模式来实现半透明效果。这个技巧在计算机未提供混合硬件时显得尤其重要。
- 轻松实现淡出效果：描述如何使用多边形点画模式来创建从背景淡出的效果。
- 使用后缓冲区进行物体选择：描述在双缓冲系统中如何使用后缓冲区处理简单的物体挑选。
- 低开销的图像转换：讨论如何通过把每个像素绘成四边形来实现位图图像的扭曲效果。
- 显示层次：解释如何显示不同材料的多个层次，并提示材料的重叠地点。
- 抗锯齿字符：描述如何绘制更为平滑的字体。
- 绘制圆点：描述如何绘制接近于圆形的点。
- 图像插值：显示如何把两幅图像进行平滑的混合。
- 制作贴花：解释如何绘制两幅图像，其中一幅为贴花类型，始终应该出现在另一幅图像的顶部。
- 使用模板缓冲区绘制填充的凹多边形：描述如何使用模板缓冲区绘制凹多边形、非简单多边形和中间有洞的多边形。
- 寻找冲突区域：描述如何确定三维空间中的重叠区域。
- 阴影：描述如何绘制光照物体的阴影。
- 隐藏直线消除：讨论如何绘制线框物体，并用模板缓冲区消除被隐藏的直线。
- 纹理贴图应用：描述纹理贴图的一些有趣应用，例如对图像进行旋转和扭曲。

- 绘制深度缓冲的图像：描述如何在深度缓冲环境中对图像进行组合。
- Dirichlet域：解释如何使用深度缓冲区，寻找一组点的Dirichlet域。
- 使用模板缓冲区实现生存游戏：解释如何使用模板缓冲区实现生存游戏。
- glDrawPixels()和glCopyPixels()的其他应用：描述如何使用这两个函数实现诸如仿视频、油漆喷雾和图像变换等效果。

14.1 错误处理

程序总会出错，这是亘古不变的真理。在开发过程中，使用错误处理函数是一项基本的工作。并且，在商业发行的应用程序中，我们也强烈建议使用错误处理机制（除非能保证自己的程序百分之百不会遇到OpenGL错误情况，但这种可能性太小了）。OpenGL提供了一些简单的错误处理函数，可用于基本GL和GLU函数库。

当OpenGL检测到错误时（基本GL或GLU），它会记录当前的错误代码，使这个错误的命令被忽略，因此它对OpenGL的状态或帧缓冲区的内容不会产生影响。但是，如果被记录的错误是GL_OUT_OF_MEMORY，这条命令的结果是未定义的。当一个错误代码被记录之后，当前的错误代码便不会清除，也就是说不会再记录其他错误代码，直到调用查询函数glGetError()为止。这个函数返回当前的错误代码。在查询并清除了错误代码之后，或者如果一直没有遇到错误，调用glGetError()函数将会返回GL_NO_ERROR。

GLenum **glGetError**(void);

返回错误标志的值。当GL或GLU出现错误时，这个错误标志就会设置为适当的错误代码值。如果这个函数返回GL_NO_ERROR，那么自从上一次调用**glGetError()**函数以来或自从GL初始化以来并没有出现可以检测到的错误。在调用**glGetError()**之前，不会记录其他错误。这个函数返回错误代码，并把错误标志设置为GL_NO_ERROR。

强烈建议在每个display()函数内部至少调用一次glGetError()。表14-1列出了预先定义的OpenGL基本错误代码。

OpenGL错误包括37种GLU NURBS错误（具有非描述性的常量名GLU_NURBS_ ERROR1、GLU_NURBS_ERROR2等）、14种分格化错误（GLU_TESS_MISSING_BEGIN_POLYGON、GLU_TESS_MISSING_BEGIN_CONTOUR、GLU_TESS_MISSING_END_CONTOUR、GLU_TESS_COORD_TOO_LARGE、GLU_TESS_NEED_COMBINE_ CALLBACK以及8个名称为GLU_TESS_ERROR*的错误）和GLU_INCOMPATIBLE_GL_VERSION。另外，GLU定义了错误代码GLU_INVALID_ENUM、GLU_INVALID_VALUE和GLU_OUT_OF_MEMORY，它们与相关的OpenGL代码具有相同的含义。

为了获取与GL或GLU错误代码相对应的可打印的描述性字符串，可以使用gluErrorString()函数。

表14-1 OpenGL错误代码

错误代码	描　述
GL_INVALID_ENUM	GLenum参数超出范围
GL_INVALID_VALUE	数值参数超出范围
GL_INVALID_OPERATION	在当前状态下这个操作是非法的
Gl_INVALID_FRAMEBUFFER_OPERATION	帧缓冲区对象不完整
GL_STACK_OVERFLOW	该命令将导致堆栈上溢
GL_STACK_UNDERFLOW	该命令将导致堆栈下溢
GL_OUT_OF_MEMORY	没有足够的内存执行这条命令
Gl_TABLE_TOO_LARGE	指定的表太大

const GLubyte* **gluErrorString**(GLenum *errorCode*);

返回一个指向描述性字符串的指针，这个字符串与OpenGL或GLU错误代码*errorCode*相对应。

示例程序14-1显示了一个简单的错误处理函数。

示例程序14-1 查询和打印错误

```
GLenum errCode;
const GLubyte *errString;

while ((errCode =glGetError())!=GL_NO_ERROR){
  errString =gluErrorString(errCode);
  fprintf (stderr,"OpenGL Error:%s \n ",errString);
}
```

注意：gluErrorString()函数返回的字符串无法被应用程序修改或释放。

14.2 OpenGL版本

OpenGL应用程序的可移植性是OpenGL的一个重要特性。但是，新版本的OpenGL会引入新的特性。如果在编写应用程序时使用了新特性，那么当这个应用程序在旧版本的OpenGL上运行时就会遇到问题。另外，我们可能希望自己的应用程序在各种不同的OpenGL实现上都具有相同的行为。例如，我们可能想在一台计算机上把纹理贴图作为默认的渲染模式，而在另一台计算机上使用默认的单调着色模式。可以使用glGetString()函数获取和OpenGL实现有关的版本信息。

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

返回一个指向字符串的指针，这个字符串描述了当前OpenGL实现的某个方面的信息。*name*可以是下面常量值之一：GL_VENDOR、GL_RENDERER、GL_VERSION、Gl_SHADING_LANCUAGE_VERSION或GL_EXTENSIONS。

兼容性扩展
GL_EXTENSIONS

GL_VENDOR返回提供了当前OpenGL实现的厂商的名称。GL_RENDERER返回一个渲染器标识符，通常是一个硬件平台。关于GL_ EXTENSIONS的更多信息，请参阅第14.3节。

GL_VERSION和GL_SHADING_LANGUAGE_VERSION返回一个字符串，标识当前OpenGL实现的版本号。这个版本字符串的格式如下所示：

```
<version number><space><vedor-specific information>
```

版本号的格式是：

```
major_number.minor_number
```

或者是：

```
major_number.minor_number.release_number
```

其中所有的号码都由一个或多个数字组成。厂商特定的信息是可选的。例如，如果当前的OpenGL实现是由一家虚构的XYZ公司提供的，那么这个返回的字符串可能是：

```
1.2.4 XYZ-OS 3.2
```

它表示这个OpenGL实现是XYZ公司创建的OpenGL函数的第4次发布，它遵循OpenGL 1.2规范。另外，这条信息很可能还表示这是YXZ公司的专用操作系统的3.2发布。

在运行时，查询OpenGL版本号的另一种方法是用GL_MAJOR_VERSION和

GL_MINOR_VERSION 调用glGetIntegerv()，它们将分别返回主版本号和次版本号，就好像glGetString()会返回字符串形式一样。在编译时，可以使用预处理指令#ifdef，查找名为GL_VERSION_m_n这样的常量，其中m和n分别对应主版本号和次版本号。例如，预处理器标记GL_VERSION_3_0表示OpenGL 3.0。如果出现一个特定的标记，表示在其之上的所有版本，包括该版本在内都是可用的，至少对于编译器是这样。在运行时，总是应该检查版本和扩展字符串，以确保所需要的功能在正在使用的实现中是存在的。

注意：*在客户端/服务器模式下，例如使用X窗口系统的OpenGL扩展进行间接渲染时，客户端和服务器所使用的OpenGL版本可能并不相同。如果客户端的版本比服务器高，那么客户端的有些请求可能无法得到服务器的支持。*

14.2.1 工具函数库版本

gluGetString()是一个GLU查询函数，它与glGetString()相似。

---

const GLubyte* **gluGetString**(GLenum *name*);

返回一个指向字符串的指针，这个字符串描述了OpenGL实现的某个方面的信息。*name*可以是GLU_VERSION或GLU_EXTENSIONS。

注意，gluGetString()在GLU 1.0中不可用。查询GLU版本号的另一种方法是查找符号常量GLU_VERSION_1_3。如果不存在常量GLU_VERSION_1_3，表示当前使用的是GLU 1.2版本或更早的版本。对于GLU_VERSION_1_1和GLU_VERSION_1_2，也是采用相同的原则。

另外，注意GLU扩展和OpenGL扩展不同。

14.2.2 窗口系统扩展版本

对于OpenGL的窗口系统扩展，例如GLX、WGL、PGL和AGL，有一些类似的函数（例如glXQueryExtensionString()）可以查询它们的版本信息。关于这个话题的更多信息，请参阅附录D。

（以前所提到的glXQueryExtensionString()以及相关的函数是在GLX 1.1中引入的，GLX 1.0对它们并不提供支持。）

14.3 标准的扩展

OpenGL具有正式的书面规范，描述了组成OpenGL函数库的各种操作。各个厂商可以在它们发布的OpenGL实现中增加其他功能。

函数和符号常量清楚地表示了一个特性是OpenGL标准还是厂商特定的扩展，或者是OpenGL ARB（体系结构审核委员会）所批准的OpenGL扩展。

为了创建一个厂商特定的名称，厂商必须添加一个公司标识符（大写形式），并且在必要的情况下添加一些额外的信息，例如机器名称。例如，XYZ公司想增加一个新函数glCommandXYZ()以及常量GL_DEFINITION，如果XYZ公司希望这个扩展只能在它的FooBar图形卡上使用，这个扩展就可以取名为glCommandXYZfb()和GL_DEFINITION_XYZ_FB。

如果两个或更多的厂商同意实现相同的扩展，那么函数和常量就使用更为通用的EXT后缀（glCommandEXT()和GL_DEFINITION_EXT）。

类似地，如果这个扩展通过了OpenGL ARB的批准，那么它的函数和常量就以ARB为后缀（glCommandARB()和GL_DEFINITION_ARB）。

确定支持的扩展

有多种方式来确定OpenGL实现支持哪些扩展：

- 如果OpenGL的版本小于3.0，调用glGetString(GL_EXTENSIONS)，它返回一个列表包括实现所支持的每个扩展，各扩展之间用空格隔开。
- 要查看是否支持一个特定的扩展，如果你使用的是GLU 1.3或更高版本，可以调用gluCheckExtension()，它返回一个布尔值，表示是否支持该扩展。

GLboolean **gluCheckExtension**(char **extName*,
const GLubyte **extString*);

如果在extString中可以找到*extName*，这个函数返回GL_TRUE，否则返回GL_FALSE。

如果你使用的是OpenGL 3.0或更高的版本，可以调用glGetStringi()，它返回一个单个字符串，对应与所提供的index相关的扩展字符串。

---

const GLubyte ***glGetStringi**(GLenum *target*, GLuint *index*);

返回与表示索引的状态目标的index相关的字符串。和使用**glGetIntegerv()**查询时一样，*target*必须是GL_EXTENSIONS。*index*必须是0到GL_NUM_EXTENSIONS-1之间的值。

扩展字符串没有特定的顺序；顺序可能根据OpenGL的实现的不同而不同，或者根据单个实现中的不同环境而不同。

如果index小于0，或者大于或等于GL_NUM_EXTENSIONS，将产生一个GL_INVALID_VALUE错误。

注意：在OpenGL 3.1中，glGetString()将不再接受GL_EXTENSIONS。必须使用glGetStringi()来获取扩展的列表。

如果使用GLEW库来管理扩展，它提供一种方便的方法来验证扩展：使用与扩展字符串条目相同的名字定义一个布尔变量（例如，“GL_ARB_multitexture”）。

如果当前使用的OpenGL实现支持GLU 1.3，可以使用gluCheckExtension()函数观察它是否支持指定的扩展。

gluCheckExtension()函数可以检查OpenGL和GLX扩展字符串，以及GLU扩展字符串。

在GLU 1.3之前，为了判断一个特定的扩展是否得到支持，可以使用示例程序14-2所示的代码来搜索字符串列表，寻找是否存在与扩展名称匹配的字符串。如果能够找到这个字符串，QueryExtension()函数就返回GL_TRUE，否则就返回GL_FALSE。

示例程序14-2 判断一个特定的扩展是否得到支持（在GLU 1.3之前使用的方法）

```
static GLboolean QueryExtension(char *extName)
{
   char *p =(char *)glGetString(GL_EXTENSIONS);
   char *end;
   if (p ==NULL)
     return GL_FALSE;
   end =p +strlen(p);
   while (p <end){
     int n =strcspn(p," ");
     if ((strlen(extName)==n)&&(strncmp(extName,p,n)==0)){
      return GL_TRUE;
```

```
        }
        p +=(n +1);
      }
      return GL_FALSE;
}
```

Microsoft Windows的标准扩展（WGL）

如果需要访问Microsoft Windows平台的一个扩展函数，应该做到：

- 根据标识扩展的符号常量来创建条件代码。
- 在运行时查询前面所讨论的扩展字符串。
- 使用wglGetProcAddress()函数寻找指向扩展函数的指针。

示例程序14-3说明了如何查找一个名称为glSpecialEXT()的虚构扩展。注意，条件代码（#ifdef）用于验证GL_EXT_special的定义是否存在。如果这个扩展已经存在，就为这个函数指针特别定义一种数据类型（在此例中为PENGLSPECIALEXTPROC）。最后，调用wglGetProcAddress()函数获取这个函数指针。

示例程序14-3 用wglGetProcAddress()函数查找OpenGL扩展

```
#ifdef GL_EXT_special
   PFNGLSPECIALEXTPROC glSpecialEXT;
#endif

/*somewhere later in the code */

#ifdef GL_EXT_special
   glSpecialEXT =(PFNGLSPECIALEXTPROC)
                  wglGetProcAddress("glSpecialEXT");
   assert(glSpecialEXT !=NULL);
#endif
```

14.4 实现半透明效果

可以使用多边形点画模式模拟半透明的材料。对于那些并不支持混合硬件的系统，这是一种非常有效的解决方案。由于多边形点画模式的大小是32×32位（1024位），因此可以在透明和不透明之间实现1023种不同的半透明效果（远远超过了实际所需）。例如，如果需要一个能够透过29%的光线的表面，可以简单地创建一个点画模式，把其中29%的像素（大约297个）的掩码设置为0，把其余像素的掩码设置为1。如果多个表面具有相同的半透明性质，不要使用相同的点画模式来表示每个表面，因为它们将覆盖屏幕上完全相同的位。可以随机选择适当数量的像素，把它们的掩码设置为0，从而为不同的表面创建不同的点画模式（关于多边形点画模式的更多信息，请参阅第2.4节）。

如果不喜欢使用随机选择的像素来组成点画模式，可以使用有规律的模式。但是，当多个按照这种方式创建的半透明表面叠在一起时，它们所形成的效果就不太好。通常，这并不会产生问题，因为绝大多数场景很少存在相互叠在一起的半透明区域。在一幅具有半透明车窗的汽车图片中，视线最多能够穿透两层车窗，通常仅仅是一层。

14.5 轻松实现淡出效果

假如希望实现一幅逐渐淡出为背景色的图像，可以定义一系列的多边形点画模式。每个模式依次

有更多打开的位，这样它们就能表示更深和更稠的模式。然后，可以对一个足以覆盖淡出区域的大型多边形反复应用这些模式。例如，如果想经过16个步骤淡出到黑色背景。首先，可以定义16个不同的模式数组：

```
GLubyte stips[16][4*32];
```

接着，将每个32×32模式数组中1/16的像素值设置为1，但每个模式所设置的像素均不相同，以确保对所有的点画模式执行按位OR操作时，最终得到的点画模式的所有位均为1。然后，就可以使用下面的代码来实现淡出效果：

```
draw_the_picture();
glColor3f(0.0, 0.0; 0.0);    /* set color to black */
for (i = 0; i < 16; i++) {
    glPolygonStipple(&stips[i][0]);
    draw_a_polygon_large_enough_to_cover_the_whole_region();
}
```

在有些OpenGL实现中，可以把点画模式编译到显示列表中，从而提高程序的性能。在初始化时，可以采用下面的做法：

```
#define STIP_OFFSET 100
for (i = 0; i < 16; i++) {
    glNewList(i+STIP_OFFSET, GL_COMPILE);
    glPolygonStipple(&stips[i][0]);
    glEndList();
}
```

然后，把前面第一段代码的下面这行：

```
glPolygonStipple(&stips[i][0]);
```

替换为：

```
glCallList(i);
```

通过把设置点画模式的函数编译为显示列表，OpenGL可以对stips[][]数组中的数据进行进行合理的排列，符合硬件最高效率设置点画模式所要求的格式。

这个技巧还有另一种应用，在绘制运动图像时，可能想在物体后面留下逐渐变淡的尾迹，以显示它在此之前的运动轨迹。例如，在模拟太阳系时，可以通过尾迹来显示行星的运动轨迹。在此，假定仍然按照16个步骤完成淡出，并使用显示列表来建立点画模式。程序的主循环如下所示：

```
current_stipple = 0;
while (1) {                          /* loop forever */
    draw_the_next_frame();
    glCallList(current_stipple++);
    if (current_stipple == 16) current_stipple = 0;
    glColor3f(0.0, 0.0, 0.0);        /* set color to black */
    draw_a_polygon_large_enough_to_cover_the_whole_region();
}
```

每经过一次循环，就有1/16的像素成为黑色。在16帧之内没有被行星覆盖的像素都将被清除为黑色。当然，如果使用的系统提供了对混合操作的硬件支持，在每一帧中混合一定数量的背景颜色就更加容易一些。关于多边形点画的更多信息，请参阅第2.4节。关于显示列表的更多信息，请参阅第7章。关于混合的更多信息，请参阅第6.1节。

14.6 使用后缓冲区进行物体选择

尽管OpenGL的选择机制（参见第13.1.5节）功能强大并且非常灵活，但使用起来较为繁琐。需要处理的情况往往非常简单：应用程序绘制一个由大量物体组成的场景，用户通过鼠标点击一个物体，应用程序判断鼠标光标下面的是哪个物体。

完成这个任务的其中一种方法就是利用双缓冲模式。当用户挑选一个物体时，应用程序就在后缓冲区内对整个场景进行重绘，但它并不使用常规的物体颜色，而是用某种类型的对象标识符来表示每个物体的颜色。然后，应用程序简单地读取光标下的像素，这个像素的值能够提示被挑选的物体的编号。如果希望在一幅静态的图像上进行大量的挑选，可以一次读入整个颜色缓冲区并在其中查找需要挑选的每种颜色，而不是单独读取每个像素。

注意，和标准的选择特性相比，这个技巧有一个优越之处，就是它会挑选出现在同一个像素上的多个物体中最前面的那个。由于用户在后缓冲区中是用伪颜色来绘制图像的，因此他们看不到图像。在交换缓冲区之前，可以对后缓冲区进行重绘（或者把它复制到前缓冲区）。在颜色索引模式下，编码非常简单：把物体的标识符当作索引。在RGBA模式下，需要把标识符的各个位分别编码为R、G、B和A成分。

注意，如果场景中有太多的物体，标识符可能会不够用。例如，假设我们所使用的是一个用4位的缓冲区来表示每个颜色缓冲区的颜色索引信息（可以表示16个索引值）的系统，并且处于颜色索引模式下，如果场景是由数千种可挑选的物体所组成，那么就会出现这个问题。为了解决这个问题，挑选可以通过几次来完成。为了更形象地思考这个问题，假设场景中的物体数量不超过4096个，这样就可以用12位来表示所有物体的标识符。在第一次挑选时，使用由4个高端的位组成的索引值来绘制场景，然后在第二次和第三次分别使用中间的4位和低端的4位。在每次挑选之后，读取光标下的像素，提供位信息，并把它们包装在一起，以获取物体的标识符。

通过这种方法，挑选需要进行3次，但这通常是可以接受的。注意，有了4个高端的位后，就消除了所有物体的15/16，因此在第二次挑选时只需要绘制1/16的物体。类似地，在第二次挑选之后，前面所剩的256个物体有255个已经消除。第一次挑选所需要的时间可能和绘制整个帧差不多，但是第二次和第三次挑选的速度则分别要快上16倍和256倍。

如果想编写能够在不同系统中都可以运行的代码，需要把物体标识符分解为几块，符合这些系统的最小公分母。另外，记住系统在RGB模式下也许会执行自动的颜色抖动，因此需要关闭颜色抖动功能。

14.7 低开销的图像转换

如果想绘制一幅经过变形的位图图像（也许是简单的拉伸或旋转，也许是根据某些数学函数进行剧烈的变动），可以使用很多种方法。可以把图像作为纹理图像，并据此对它进行缩放、旋转或变形。如果只是想对图像进行缩放，可以使用glPixelZoom()函数。

在许多情况下，可以通过把图像的每个像素画成四边形来实现非常好的效果。尽管使用这种方法所产生的图像不像使用复杂的过滤算法产生的图像那么漂亮，并且对于高级用户而言可能显得功能不够强大，但是它已经能够向我们提供很大的帮助。

为了更形象地说明问题，假设原图像的大小是m个像素乘以n个像素，所选择的坐标是从[0, m−1] × [0, n−1]。假设变形函数是x(m, n) 和y(m, n)。例如，如果这个变形只是简单地使用放大因子为3.2

的缩放，那么x(m, n) = 3.2m，y(m, n) = 3.2 .n。下面的代码可以用于绘制经过变形的图像：

```
glShadeModel(GL_FLAT);
glScale(3.2, 3.2, 1.0);
for (j=0; j < n; j++) {
    glBegin(GL_QUAD_STRIP);
    for (i=0; i <= m; i++) {
        glVertex2i(i,j);
        glVertex2i(i, j+1);
        set_color(i,j);
    }
    glEnd();
}
```

这段代码把原先的图像放大3.2倍，并用原先的像素颜色绘制每个经过变换之后的像素。set_color()函数表示用于设置图像像素颜色的适当OpenGL函数。

下面这段代码稍微复杂些，它使用函数x(i, j) 和y(i, j) 对图像进行变形：

```
glShadeModel(GL_FLAT);
for (j=0; j < n; j++) {
    glBegin(GL_QUAD_STRIP);
    for (i=0; i <= m; i++) {
        glVertex2i(x(i,j), y(i,j));
        glVertex2i(x(i,j+1), y(i,j+1));
        set_color(i,j);
    }
    glEnd();
}
```

可以使用下面这段代码绘制一幅变形度更大的图像：

```
glShadeModel(GL_SMOOTH);
for (j=0; j < (n-1); j++) {
    glBegin(GL_QUAD_STRIP);
    for (i=0; i < m; i++) {
        set_color(i,j);
        glVertex2i(x(i,j), y(i,j));
        set_color(i,j+1);
        glVertex2i(x(i,j+1), y(i,j+1));
    }
    glEnd();
}
```

这段代码对每个四边形的颜色进行平滑的插值。注意，这段代码在每个方向上产生的四边形较之单调着色的版本更少一些，因此这幅彩色图像用于在四边形的顶点上指定颜色。另外，可以用适当的混合函数（GL_SRC_ALPHA、GL_ONE）对多边形进行抗锯齿处理，以实现更为漂亮的图像。

14.8 显示层次

有些应用程序，例如半导体电路设计程序，需要显示多层不同的材料，并指定各层材料在什么地方重叠。下面是一个简单的例子，假设有3种不同的物质可以进行排列。在任何一个点，可以出现8种层次的组合，见表14-2。

表14-2 8种层次组合

	层次1	层次2	层次3	颜色
0	无	无	无	黑
1	有	无	无	红
2	无	有	无	绿
3	有	有	无	蓝
4	无	无	有	粉红
5	有	无	有	黄
6	无	有	有	白
7	有	有	有	灰

我们可能想让自己的程序根据层次显示8种不同的颜色。这张表的最后一列显示了一种任意的可能性。为了使用这种方法，可以使用颜色索引模式，并加载颜色图，使元素0为黑色，元素1为红色，元素2为绿色，接下来以此类推。注意，如果数字0到7是用二进制来表示的，当出现第3层时，第4位为1；当出现第2层时，第2位为1；当出现第1层时，第1位为1。

为了清除窗口，可以把写入掩码设置为7（全部3层），并把清除颜色设置为0。为了绘制图像，可以把绘图颜色设置为7。然后，想在第n层绘制物体时，把写入掩码设置为n。在其他类型的应用程序中，可能需要在一个层中进行选择性的擦除。在这种情况下，可以使用刚刚讨论的写入掩码，但是把绘图颜色设置为0而不是7（关于写入掩码的更多信息，请参阅第10.1.4节）。

14.9 抗锯齿字符

标准的字符绘制技巧是使用glBitmap()函数。这个函数在绘制一个字符的每个像素时，采取要么绘制要么不绘制的方式。例如，如果在白色背景上绘制黑色字符，最终的像素要么是黑色的，要么是白色的，不会出现中间状态的灰色。如果在渲染字符时能够使用中间颜色（在这个例子中，是灰色），就可以实现更为平滑的高质量图像。

假设在白色背景上绘制黑色字符，可以想像一下屏幕上像素高度放大后的像素格局，如图14-1的左图所示。

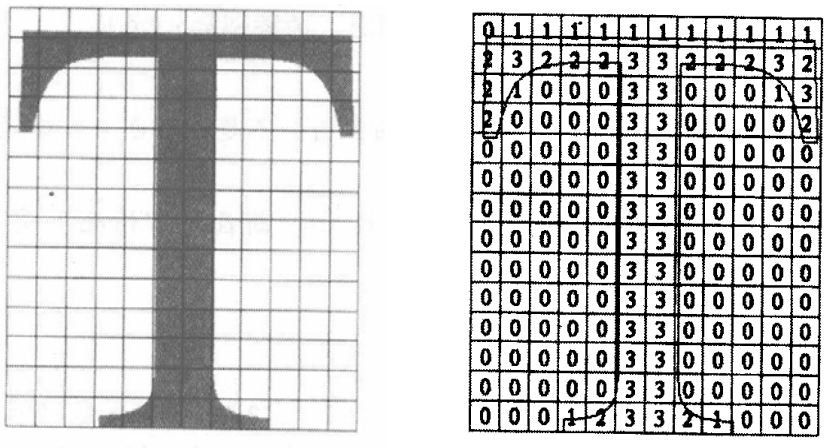


图14-1 抗锯齿字符

注意，有些像素完全位于字符的轮廓之内，因此应该绘制成黑色。有些像素完全于字符的轮廓之外，因此应该绘制成白色。但是，有些像素在理想情况下最好绘制成一定程度的灰色，灰色的程度对

应于这个像素中黑色所占的比重。如果使用这种技巧，屏幕上显示的图像将更为漂亮。

如果无需顾及速度和内存要求，可以用一幅小型的图像而不是位图来绘制每个字符。但是，如果使用的是RGBA模式，这个方法可能要求为每个像素存储32位的信息，而不是标准字符的每个像素所存储的1位信息，这显然造成了巨大的空间浪费。可以使用8位的颜色索引值来表示每个像素，并在实际处理时通过颜色表查找把它转换为RGBA值。在许多情况下，可以用黑色和白色之间的几种（例如2～3种）灰色来绘制字符。这样，在最终的字符中，每个像素只需要使用2或3个位来存储信息。

图14-1右侧的图显示了每个像素的近似覆盖比例：0表示近似为空，1表示大约1/3的覆盖，2表示大约3/2的覆盖，3表示完全覆盖。如果标号为0的像素绘制成白色，标号为3的像素绘制成黑色，标号为1和2的像素分别绘制成1/3和3/2的黑色，最终字符看上去的效果就比较好。在这种情况下，存储数字0、1、2和3只需要2位信息。因此，只要为每个像素使用2个位，就可以保存4个层次的灰度信息。

基本上，可以采用两种方法来实现抗锯齿的字符，取决于具体使用的是RGBA模式还是颜色索引模式。

在RGBA模式下，可以定义3个不同的字符位图，分别对应于图14-1中1、2和3出现的地方。然后，把颜色设置为白色，并清除背景。把颜色设置为1/3的灰色（RGB = （0.666，0.666，0.666)），并绘制所有值为1的像素。然后，把RGB设置为（0.333，0.333，0.333)，绘制所有标号为2的像素，再用RGB = (0.0，0.0，0.0）绘制标号为3的像素。我们所做的事情就是定义3种不同的字体，并对字符串进行3次重绘，在每次重绘时填充具有适当颜色深度的位。

在颜色索引模式下，可以执行相同的操作。但是，如果愿意正确地设置颜色映射表，并使用写入掩码，可以实现每个字符只用2个位图，并且每个字符串只需要进行两遍渲染。在前面那个例子中，可以设置一个位图，把字符中1或3出现的地方设置为1。然后，再设置第二个位图，把字符中2或3出现的地方设置为1。然后加载颜色表，0表示白色，1表示淡灰色，2表示深灰色，3表示黑色。把颜色设置为3（用二进制表示是11)，并把写入掩码设置为1，然后绘制第一个位图。随后，把写入掩码设置为2，并绘制第二个位图。在图14-1中出现0的地方，帧缓冲区中不会绘制任何东西。而在出现1、2和3的地方，帧缓冲区中会分别出现1、2和3。

由于这个例子只使用了4个灰色层次，因此后面这种方法所带来的好处比较有限，只不过把3遍渲染减少到2遍而已。但是，如果使用了8个灰色层次，在RGBA模式下就需要进行7遍渲染，而颜色表掩码技巧只需要使用3遍就可以了。如果是16个灰度层次，两者的对比就是15：4（关于写入掩码的更多信息，请参阅第10.1.4节。关于绘制位图的更多信息，请参阅第8.1节)



尝试一下

- 如何在RGBA模式下绘制字符，并且所使用的位图数不超过颜色索引模式下的优化技巧所需要的位图数量？

提示：可以回想一下，在进行抗锯齿处理时，RGB片断在正常情况下是如何合并到颜色缓冲区中的。

14.10 绘制圆点

可以通过启用点抗锯齿功能，关闭混合功能，并使用一个alpha测试函数（只通过那些alpha值大于0.5的片断)，来绘制近似于圆的点（关于这些主题的更多信息，请参阅第6.2节和第6.1节)。

14.11 图像插值

假设有两幅图像（这里的图像既可以是位图，也可以是通过几何图形渲染得到的图片)，我们想

均匀地从其中一幅过渡到另一幅。可以使用alpha成分和适当的混合操作方便地实现这个目的。假设想通过10步来完成过渡，第0帧显示图像A，第9帧显示图像B。最显而易见的方法是在第i帧用(9-i)/9的alpha值绘制图像A，用i/9的alpha值绘制图像B。

这个方法存在的问题是每一帧都必须绘制这两幅图像。一种更快的方法是在第0帧绘制图像A。在绘制第1帧时，用1/9的图像B与当前图像的8/9进行混合。在绘制第2帧时，用1/8的图像B与当前图像的7/8进行混合。在绘制第3帧时，用1/7的图像B与6/7的当前图像进行混合。接下来依此类推。在最后一步，就用1/1的图像B与0/1的当前图像进行混合，产生的结果就是图像B。

为了观察这个方法的工作方式，如果第i帧为：

$$\frac{(9-i)A}{9}+\frac{iB}{9}$$

用B/(9−i) 与(8−i)/(9−i) 的当前图像进行混合，所得结果是：

$$\frac{B}{9-i}+\frac{8-i}{9-i}\left[\frac{(9-i)A}{9}+\frac{iB}{9}\right]=\frac{9-(i+1)A}{9}+\frac{(i+1)B}{9}$$

(关于这个主题的一些细节，请参阅第6章。)

14.12 制作贴花

假设我们正在绘制一幅复杂的三维图片，并使用了深度缓冲技巧来消除隐藏表面。我们进一步假设这张图片的一部分是由共面的图像A和B所组成，其中图像B是一种贴花，它始终应该出现在图像A的表面上。

首先想到的方法可能是在绘制了图像A之后再绘制图片B，并设置深度缓冲过滤函数，替换那些深度值大于等于深度缓冲区中已存储之值的那些片断（GL_GEQUAL)。但是，由于顶点的浮点表示形式的精度有限，四舍五入所产生的误差可能会导致图像B有时候比图像A稍微靠前一点，有时候又比图像A稍微靠后一点。这个问题的解决方法如下：

1) 禁止深度缓冲区的写入，并渲染A。

2) 启用深度缓冲区的写入，并渲染B。

3) 禁止颜色缓冲区的写入，并再次渲染A。

4) 启用颜色缓冲区的写入。

注意，在整个过程中，深度缓冲区测试都是被启用的。在步骤1)中，A被渲染在它应该出现的位置，但深度缓冲区中的值并不会修改。因此，在步骤2)中，当B出现在A上面时，B保证能够被绘制。步骤3)简单地保证A下面的所有深度值都被正确地更新。但是，由于RGBA的写入被禁止，因此颜色像素并不会受到影响。最后，步骤4)把系统恢复为默认的状态（深度缓冲区和颜色缓冲区都允许写入)。

如果启用了模板缓冲区，可以使用下面这些更为简单的步骤：

1) 对模板缓冲区进行配置，在深度测试通过的地方写入1，在未通过的地方写入0。然后渲染A。

2) 对模板缓冲区进行配置，使模板值不会变化，但只渲染模板值为1的地方。禁止深度缓冲区的测试和更新。然后，渲染B。

通过这种方法，在任何时候对模板缓冲区的内容进行初始化都是不必要的，因为我们感兴趣的所有像素（也就是由A所渲染的像素）的模板值是在A被渲染的时候设置的。另外，确信在绘制其他多边形之前重新启用深度测试并禁用模板测试（相关主题的细节可以参阅第10.1.3节、第10.2.3节和第10.2.4节)。

14.13 使用模板缓冲区绘制填充的凹多边形

考虑图14-2所示的凹多边形1234567。可以想象它是由一系列的三角形组成的：123、134、145、156和167（均在图中显示）。粗线表示原多边形的边界。绘制所有这些三角形将把缓冲区划分为9个区域A、B、C、…、I，其中区域I位于所有三角形的外面。

在图14-2的文字部分，每个区域名称的后面是覆盖这个区域的三角形列表。区域A、D和F组成了原多边形。注意，这三个区域是由奇数个三角形覆盖的。其他每个区域都是由偶数个三角形覆盖的（可能是零个）。因此，为了渲染这个凹多边形的内部，只需要渲染由奇数个三角形覆盖的区域。这可以通过模板缓冲区来实现，使用一种两道渲染的算法。

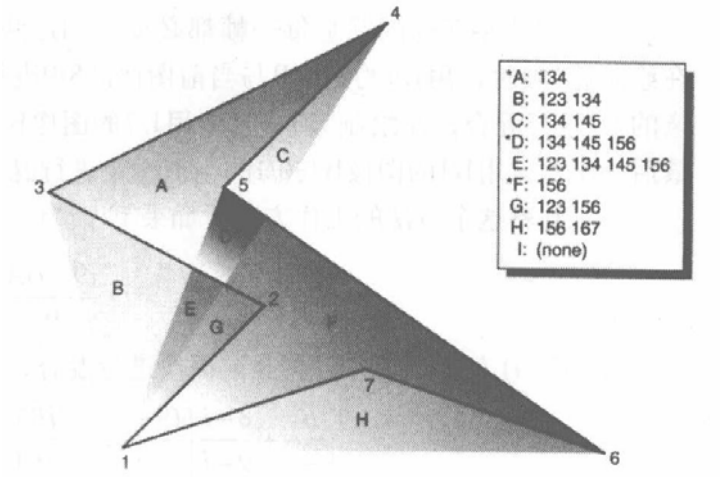


图14-2 凹多边形

首先，清除模板缓冲区，并禁止颜色缓冲区的写入。接着，使用模板缓冲区中的GL_INVERT函数，依次绘制每个三角形（为了实现最佳的性能，可以使用三角形扇）。每当绘制覆盖一个像素的三角形时，相应的模板缓冲区值就在零和非零值之间切换。在绘制完所有的三角形之后，如果像素被覆盖的次数为偶数次，则相应的模板缓冲区值为零，否则为非零值。最后，可以绘制一个覆盖所有区域的大型多边形（或重新绘制所有的三角形），但这次只绘制模板缓冲区值为非零的像素。

注意：上面这种技巧更为通用，因为我们不需要从一个多边形顶点开始进行操作。在1234567这个例子里，假定P是这个多边形上面或外面的一个任意的点。绘制三角形：P12、P23、P34、P45、P56、P67和P71。由奇数个三角形覆盖的区域位于这个多边形的内部，而其他区域则位于这个多边形的外部。这种方法之所以更为通用的原因是即使当P恰好位于一个三角形的某条边上，其结果也只是少了一个三角形而已，不会对这个多边形造成影响。

这个技巧还可用于填充非简单多边形（存在相交边的多边形）和中间有洞的多边形。下面这个例子说明了如何处理一个具有两个区域的复杂多边形（其中一个为四边形，另一个为五边形）。然后，进一步想象它们分别有一个三角形和四边形的洞（至于哪个洞位于哪个区域内部则无关紧要）。假设这两个区域分别是abcd和efghi，那两个洞分别是jkl和mnop。此外，z是这个平面上的任意一点。绘制下面这些三角形：

zab zbc zcd zda zef zfg zgh zhi zie zjk zkl zlj zmn zno zop zpm

把那些由奇数个三角形覆盖的区域标记为in，由偶数个三角形覆盖的区别标记为out（关于模板缓冲区的更多细节，请参阅第10.2.3节）。

14.14 寻找冲突区域

如果我们正式设计一种由更小的三维部件构成的机械零件，常常需要显示部位重叠的区域。在许多情况下，如果一台机器的不同部位彼此冲突，常常意味着出现了设计错误。对于运动的零件，显示重叠区域常常更有价值，因为冲突区域的查找可能贯穿整个机械设计周期。完成这个任务的方法非常复杂，这里所作的描述可能过于简单。关于这方面的完整细节，读者可以参阅Jarek Rossignac、Abe

Megahed和Bengt-Olaf Schnelder所著的论文《Interactive Inspection of Solids: Cross-sections and Interferences》(SIGGRAPH 1992 Proceedings)。

这个方法与第10.2.3节描述的加盖技术有关。它的基本思路是：指定一个裁剪平面，它穿过需要检测冲突的物体，然后判断裁剪平面的某部分是否位于多个物体的内部。对于静态图像，可以通过手工方式移动裁剪平面，以检测冲突区域。对于动态图像，可以使用裁剪平面网格，以简化对冲突区域的检测。

绘制每个需要进行检测的物体，并根据裁剪平面对它们进行裁剪。注意，可以使用第14.13节描述的奇偶规则来判断哪些像素位于物体的内部（对于形状正确（properly formed）的物体，如果从一个点到眼睛的光线与这个物体的奇数个表面相交，那么这个点就位于这个物体的内部）。为了查找冲突区域，需要在帧缓冲区中查找这样的像素：裁剪平面同时位于两个或多个区域的内部。换句话说，就是在任何一对物体内部的相交处。

如果需要测试多个物体之间是否存在相交的情况，可以在每次出现相交时存储1个位，并在裁剪缓冲区位于任何物体内部（各物体内部的并集）时存储另1个位。对于每个新物体，需要确定它的内部，找到它与已经测试过的物体的内部并集的相交处，并记录相交的点。然后把这个新物体的内部点添加到其他物体内部的并集中。

可以使用模板缓冲区中的不同位以及各种掩码操作来执行上面描述的操作。模板缓冲区需要为每个像素提供3个位：一个用于切换，以确定是否位于每个物体的内部；一个用于表示是否位于目前发现的所有物体内部的并集中；一个用于表示目前是否已经出现了冲突区域。为了更形象地说明这个原理，假定模板缓冲区的第1位用于在内部和外部之间进行切换，第2位表示是否位于当前内部的并集中，第4位表示当前是否存在冲突区域。对于每个打算渲染的物体，首先清除第1位（使用模板掩码1，并清除为0），然后使模板掩码为1，并使用GL_INVERT模板操作，对第1位进行切换。

可以使用模板操作查找模板缓冲区中各个位的交和并。例如，为了使缓冲区2中的位是缓冲区1和缓冲区2中位的并集，可以使用模板对这2个位进行屏蔽，并设置模板函数：如果模板缓冲区的值不为零，则通过模板测试。然后，在整个物体上面绘制一些东西。如果缓冲区1、缓冲区2对应的位至少有一个为1，则通过上面这个测试，在缓冲区2的相应位上写入1。另外，需要禁止颜色缓冲区的写入。交集的计算与此类似，模板函数的设置是：如果两个缓冲区的值是3（即缓冲区1和缓冲区2的对应位都为1）时通过模板测试，然后把结果写入到适当的缓冲区中（参见第10.2.3节）。

14.15 阴影

三维空间到三维空间的所有可行的投影都可以通过一个适当的4×4可逆矩阵和齐次坐标来实现。如果这个矩阵并不可逆，但它的秩（rank）为3，它可以从三维空间投影到二维平面上。每个可行的从三维空间到二维平面的投影都可以用一个秩为3的4×4矩阵来实现。为了找到一个任意的物体在一个任意的平面上由于一个任意的光源（可能位于无限远处）照射所造成的阴影，需要找到一个表示这个投影的矩阵，并把它与矩阵堆栈的顶部矩阵相乘，然后用阴影颜色绘制物体。记住，需要投影到每个被称为“地面”的平面上。

为了简单起见，假设光源位于原点，“地面”平面的方程式是$ax + by + cz + d = 0$。给定一个顶点$S = (sx, sy, sz, 1)$，从光源透过S的直线包括所有的点αS，其中α是一个任意的实数。这条直线与这个平面的交点满足下面的条件：

$$\alpha(a \cdot sx + b \cdot sy + c \cdot sz) + d = 0$$

因此，

$$\alpha = -d/(a \cdot sx + b \cdot sy + c \cdot sz)$$

将其代入直线方程，得到交点坐标为：

$$-d(sx, sy, sz) / (a \cdot sx + b \cdot sy + c \cdot sz)$$

下面这个矩阵把任意一点S映射到地面上：

$$\begin{bmatrix} -d & 0 & 0 & 0 \\ 0 & -d & 0 & 0 \\ 0 & 0 & -d & 0 \\ a & b & c & 0 \end{bmatrix}$$

如果我们首先对场景进行变换，使光源产于原点，可以使用这个矩阵。

如果光源来自无限远处，此时只有一个点S和一个方向$D = (dx, dy, dz)$。这条直线上的点是由下面这个方程式确定的：

$$S + \alpha D$$

继续像前面一样操作，这条直线与平面的交点是由下面这个方程式确定的：

$$a(sx + \alpha \cdot dx) + b(sy + \alpha \cdot dy) + c(sz + \alpha \cdot dz) + d = 0$$

求出α，把它代入到直线方程，然后确定下面这个投影矩阵：

$$\begin{bmatrix} b \times dy + c \times dz & -b \times dx & -c \times dx & -d \times dx \\ -a \times dy & a \times dx + c \times dz & -c \times dy & -d \times dy \\ -a \times dz & -b \times dz & a \times dx + b \times dy & -d \times dz \\ 0 & 0 & 0 & a \times dx + b \times dy + c \times dz \end{bmatrix}$$

只要提供了平面以及一个任意的方向向量，就可以使用这个矩阵。在此之前，并不需要执行任何变换（参见第3章和附录C）。

14.16 隐藏直线消除

如果想绘制一个消除了隐藏直线的线框物体，可以使用的一种方法是用直线绘制它的轮廓，然后用具有背景色的多边形来填充组成物体表面的多边形的内部。启用了深度缓冲区之后，这种内部填充将会覆盖被更靠近眼睛的表面所遮挡的所有轮廓。这种方法确实能够达到目的，但是它无法保证物体的内部完整地位于多边形的轮廓内。事实上，它可能在许多地方出现重叠。

我们可以使用一种简单地使用两道渲染的方法，既可以用多边形偏移来实现，也可以通过模板缓冲区来实现。多边形偏移通常是优先考虑的方法，因为它的速度要比模板缓冲区方法快。本节描述了这两种方法，读者可以看看它们分别是怎样解决问题的。

14.16.1 使用多边形偏移实现隐藏直线消除

为了使用多边形偏移来实现隐藏直线消除，物体需要绘制两次。首先用前景色绘制物体的边缘，使用填充多边形，但是把多边形模式设置为GL_LINE，把多边形光栅化为线框形式。然后，用默认的多边形模式绘制填充多边形，填充线框的内部，使用足够的多边形偏移，把填充多边形稍稍推向远处。在使用多边形偏移时，填充多边形内部的后退要恰到好处，使多边形的边缘看起来并没有不适的

视觉效果。

```
glEnable(GL_DEPTH_TEST);
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
set_color(foreground);
draw_object_with_filled_polygons();

glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
glEnable(GL_POLYGON_OFFSET_FILL);
glPolygonOffset(1.0, 1.0);
set_color(background);
draw_object_with_filled_polygons();
glDisable(GL_POLYGON_OFFSET_FILL);
```

我们可能还需要对多边形偏移的数量进行调整（例如，如果直线较宽的情况下）。关于这方面的更多信息，请参阅第6章。

14.16.2 使用模板缓冲区实现隐藏直线消除

使用模板缓冲区实现隐藏直线消除是一种比较复杂的方法。对于每个多边形，需要清除模板缓冲区，并同时在帧缓冲区和模板缓冲区中绘制多边形的轮廓。然后，在填充它的内部时，只有在模板缓冲区仍然清空的地方才启用绘图。为了避免对每个多边形都进行完整的模板缓冲区清除，一种比较方便的方法是使用同一个多边形的轮廓把0写入到模板缓冲区。按照这种方式，只需要完整地清除模板缓冲区1次。

例如，下面这段代码表示用于执行这种隐藏直线消除的内部循环。每个多边形的轮廓是用前景色绘制的，并用背景色进行填充，然后再次用前景色进行绘制。模板缓冲区用于追踪每个多边形在重新绘制轮廓时所使用的填充颜色。为了优化性能，可以在绘制多边形轮廓时使用相同的模板和颜色值。这样，它们在每次循环时只需要修改两次。

```
glEnable(GL_STENCIL_TEST);
glEnable(GL_DEPTH_TEST);
glClear(GL_STENCIL_BUFFER_BIT);
glStencilFunc(GL_ALWAYS, 0, 1);
glStencilOp(GL_INVERT, GL_INVERT, GL_INVERT);
set_color(foreground);
for (i=0; i < max; i++) {
    outline_polygon(i);
    set_color(background);
    glStencilFunc(GL_EQUAL, 0, 1);
    glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
    fill_polygon(i);
    set_color(foreground);
    glStencilFunc(GL_ALWAYS, 0, 1);
    glStencilOp(GL_INVERT, GL_INVERT, GL_INVERT);
    outline_polygon(i);
}
```

相关的信息请参阅第10.2.3节。

14.17 纹理贴图的应用

纹理贴图的功能非常强大，它们有一些非常有趣的应用。下面是纹理贴图的一些高级应用：

- 抗锯齿文本：在相对较高的分辨率下为每个字符定义一幅纹理图像，并使用纹理所提供的过滤功能把它们映射到较小的区域中。通过这种方法，即使文本所在的表面与屏幕并不对齐，文本看上去也是正确的，只是有点倾斜，并存在一定程度的透视变形。
- 抗锯齿直线：方法与抗锯齿文本类似，使直线在纹理图像中的宽度为几个像素，然后使用纹理过滤对直线进行抗锯齿处理。
- 图像缩放和旋转：把一幅图像作为纹理图像，并把它映射到一个多边形上。对多边形进行旋转和缩放时，图像也会随之进行旋转和缩放。
- 图像的变形：把图像作为纹理图像，但是把它映射到样条曲面（使用求值器）。当曲面变形时，图像也会随之变形。
- 图像的投影：把图像作为纹理图像，并把它作为一个聚光灯进行投影，实现幻灯片的效果。关于如何使用纹理来模拟聚光灯，请参阅第9.12节中的“q坐标”。

（有关旋转和缩放的详细信息，请参阅第3章。有关创建纹理的详细信息，请参阅第9章。有关求值器的细节，请参阅第12章。）

14.18 绘制深度缓冲的图像

对于复杂的静态背景，根据几何描述对背景进行渲染所需要的时间有可能比把需要渲染的背景作为像素图像来绘制所需要的时间还要多。如果背景是固定的，并且前景的变化也相对较为简单，我们可能想把背景（以及相关的深度缓冲版本）绘制成图像，而不是根据它们的几何描述进行渲染。前景中可能包含一些需要较长渲染时间的物体，但可以使用它们的帧缓冲区图像和深度缓冲区。我们可以使用一种两道渲染的算法，把这些物体渲染到一个深度缓冲的环境中。

例如，如果我们所绘制的是一个由球体组成的分子模型，我们可能拥有一幅精美渲染的球体图像以及相关的深度缓冲区值，并且它们是通过Phong着色技术、光线追踪或其他OpenGL无法直接提供的方法计算得到的。在绘制复杂的模型时，可能需要绘制数百个球体，它们显然应该进行深度缓冲处理。

为了在场景中添加一幅深度缓冲的图像，首先可以使用glDrawPixels()函数把图像的深度值写入到深度缓冲区。然后启用深度缓冲，把写入掩码设置为0，禁止进行绘图，并启用模板功能，在深度缓冲区写入值的时候对模板缓冲区进行绘制。

然后，把图像绘制到颜色缓冲区，并使用前面的模板缓冲区作为掩码，只有在模板缓冲区中的值为1时，才将相应的像素写入到颜色缓冲区中。在写入时，需要把模板测试函数设置为GL_ZERO，以确保在下一幅图像加入到场景之前，模板缓冲区会自动清除。如果物体会移动，更靠近或远离观察点，就需要使用正投影。在这些情况下，可以使用glPixelTransfer*()函数（以G_DEPTH_BIAS为参数）来移动深度图像（有关glDrawPixels()和glPixelTransfer*()函数的细节，请参阅第2.1.4节、第10.2.4节和第10.2.3节以及第8章）。

14.19 Dirichlet域

假设有一个点集S，其中某个点的Dirichlet域（Voronoi多边形）是由这样的一组点所组成：它们离该点的距离比它们离点集S中其他任何点的距离都要近。对于计算机几何学中的很多问题，都可以使用Dirichlet域来解决。图14-3显示了一个Dirichlet域。

如果对于S中的每个点，都绘制一个以该点为顶点、颜色与S中其他任何一个点不同的深度缓冲的锥体，则每个点的Dirichlet域都将用这种颜色绘制。为此，最简单的方法就是预先计算一幅图像中一个锥体的深度，并像第14.18节所介绍的那样将这幅图像作为深度缓冲图像。我们并不需要像绘制着色球体一样把图像绘制到帧缓冲区。当在深度缓冲区进行绘图时，可以使用模板缓冲区来记录应该进行绘图的像素。为此，首先需要清除模板缓冲区，然后在通过深度测试的地方写入非零值。为了绘制Dirichlet域，可以绘制一个占据整个窗口的多边形，但是只有当模板缓冲区中的值为非零时才启用绘图。

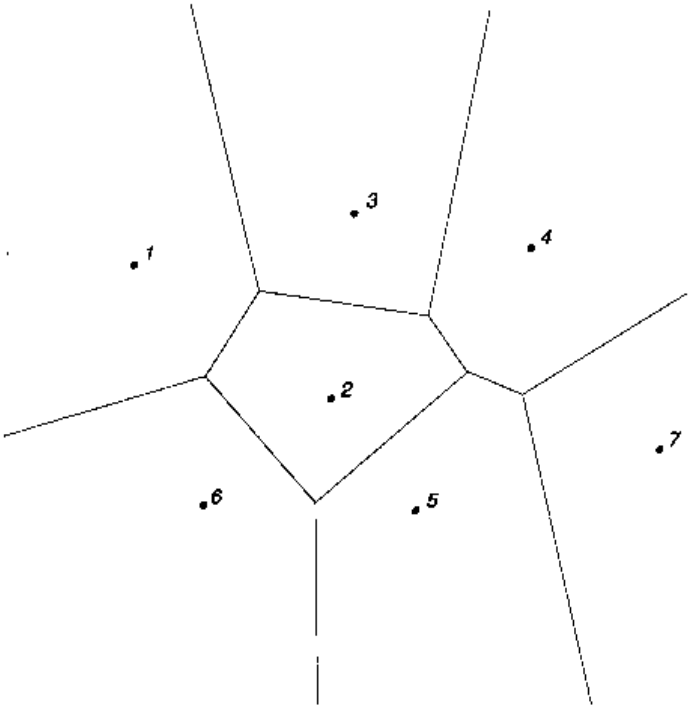


图14-3 Dirichlet域

最简单的做法是，使用一个简单的深度缓冲区，并用同一种颜色来渲染锥体。但是，如果想绘制质量较高的锥体，可能需要使用数千个多边形。使用本节介绍的技术，可以用更快的速度绘制出质量更高的锥体（相关的信息请参阅第5.1节和第10.2.4节）。

14.20 使用模板缓冲区实现生存游戏

生存游戏是由John Conway发明的，它的玩法是在一个矩形的网格上，每一个单元格的状态或者是“生存”，或者是“死亡”。为了根据当前的情况计算下一轮的情况，需要计算每一单元格的邻居有多少处于存活状态（每一单元格都有8个邻居）。如果在第n代，某一单元格是存活的，并且它有2个或3个邻居也是存活的，那么它在第$n+1$代也是存活的。或者，如果它在第n代是死亡的，但它正好有3个邻居是存活的，那么它在第$n+1$代也是存活的。在其他任何情况下，它在第$n+1$代都是死亡的。根据不同的初始配置，这个游戏能够产生令人难以置信的有趣图案（参见Martin Gardner的“Mathematical Games”，Scientific American，第223卷，1970年10月第4号，第120~123页）。图14-4显示了一个经历了6代的生存游戏。

使用OpenGL创建这个游戏的其中一种方法是使用一种多道渲染算法。把数据保存在颜色缓冲区中，一个像素表示一个单元格。假定背景色是黑色（全0），存活像素的颜色为非零值。在初始化时，把深度缓冲区和模板缓冲区清除为零，把深度缓冲区的写入掩码设置为0，并对深度比较函数进行设置，在两者不相等时才通过测试。在迭代时，从屏幕读取图像，允许在深度缓冲区中进行绘图，并调置模板函数，如果通过了深度测试，模板缓冲区的值就加上1，否则就保持不变。然后，禁止在颜色缓冲区中进行绘图。

接着，分8次绘制图像，每次在垂直、水平和倾斜方向上各偏移一个像素。完成绘图之后，模板缓冲区中就包含了每个像素的存活邻居的数量。然后，允许在颜色缓冲区中进行绘图，把颜色设置为存活单元格的颜色，并设置模板函数，只有当模板缓冲区的值为3时（3个存活邻居）才进行绘图。另外，在进行绘图时，把模板缓冲区中的值减1。然后，绘制一个覆盖图像的矩形。这样，正好具有3个存活邻居的单元格就是用“存活”颜色绘制的。

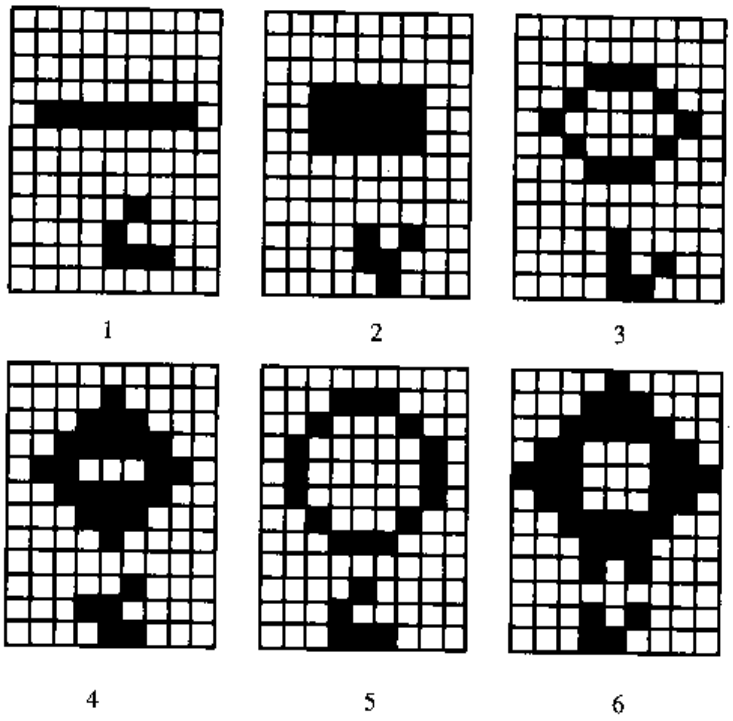


图14-4 一个经历了6代的生存游戏

此时，模板缓冲区包含了0、1、2、4、5、6、7和8，模板缓冲值为2的像素有可能存活，而模板缓冲值为0、1、4、5、6、7和8的像素则必须清除为“死亡”颜色。设置模板函数，对模板值不为2的像素进行绘制，并把模板值清除为0。然后，用“死亡”颜色绘制一个覆盖整幅图像的多边形。至此，大功告成。

为了创建一个实用的演示程序，可能需要放大单元格，而不是把它设置为单个像素那么大。如果每个单元格只用一个像素表示，那么就很难看到生存游戏所产生的详细图案（相关信息，请参阅第5.1节和第10.2.4节）。

14.21 glDrawPixels()和glCopyPixels()的其他应用

可以把glDrawPixels()函数看成是一个在屏幕上绘制一块矩形区域的函数。尽管它常常是按照这样的方式使用的，但它还有一些其他有趣的应用：

- 视频：即使计算机没有特殊的视频硬件，仍然可以用glDrawPixels()函数实现较短的电影片段的播放。这是通过在后缓冲区的一块区域中用glDrawPixels()函数反复绘制各个帧，然后把它交换到前缓冲区实现的。在性能可以被接受的前提下，使用这种方法可以显示的帧的大小取决于硬件的绘图速度。因此，如果想实现流畅的播放效果，可能不得不把电影的大小限制在100×100像素。
- 喷枪：在绘图程序中，可以用alpha值来模拟喷枪（或画刷）的形状。颜料的颜色则用颜色值来表示。为了模拟蓝色的圆形画刷，可以重复使用glDrawPixels()函数绘制蓝色正方形。alpha值在正方形的中心最大，越向外越小。在到达正方形的内接圆时，alpha值减少到0。在绘图时，混合函数这样设置：源混合因子为alpha值，目标混合因子为1 - alpha值。如果画刷的alpha值均远小于1，可能需要在一个地方重复进行绘图才能得到实心的颜色。如果alpha值接近于1，只要用画刷绘制一次就足以覆盖原来的颜色。

- 过滤放大：如果一幅像素图像的放大倍数是一个非整数，OpenGL会使用箱式过滤器，这可以导致严重的锯齿效果。为了改善过滤效果，可以将放大后的图像在小于1个像素的范围内进行微移，并重绘多次。在绘图时使用alpha混合对最终的像素求平均值。这种方法就是过滤放大。
- 图像交换：可以使用glCopyPixels()函数（使用XOR操作）交换相同大小的图像。通过这种方法，可以避免把图像读回到处理器内存中。如果A和B表示两幅图像，图像交换操作大致如下：

  a. A = A XOR B

  b. B = A XOR B

  c. A = A XOR B

第15章 OpenGL着色语言

本章目标

本章讨论OpenGL着色语言（OpenGL shading Language）。GLSL是OpenGL着色语言的通称，它是一种编程语言，用于创建可编程的着色器。作为OpenGL 2.0的一部分，OpenGL着色语言允许应用程序显式地指定在处理顶点和片断时所执行的操作。另外，GLSL可以进一步发掘OpenGL的现代硬件实现的计算威力。

本章的学习目标包括：

- 理解用OpenGL 2.0着色语言编写的可编程着色器的结构和内容。
- 把可编程着色器集成到OpenGL程序。
- 使用基于着色器的专门技术，包括变换反馈、uniform块和纹理缓冲区。

本章讨论了OpenGL着色语言的1.30版和1.40版，它们分别和OpenGL 3.0和OpenGL 3.1版本相关联。就像是OpenGL 3.1删除了OpenGL 3.0中废弃的功能一样，GLSL 1.40也从GLSL 1.30中删除了功能。如果你的实现支持GL_ARB_compatibility扩展，这些功能还能够使用。本章讨论的着色器基于GLSL 1.30的环境。

15.1 OpenGL图形管线和可编程着色器

可以把OpenGL操作看成是两台机器，其中一台对顶点进行处理，另一台对片断进行处理。机器的前面具有拔号盘和开关，可以通过切换开关和拔号来控制机器的操作，但是机器的内部操作是“固定”的。也就是说，我们无法更改机器内部的操作顺序，并且只能利用机器提供的特性。OpenGL的这种操作模式常常又称为“固定功能的管线”，附录E将对它进行详细的描述。图15-1显示了OpenGL固定功能管线的基本架构。

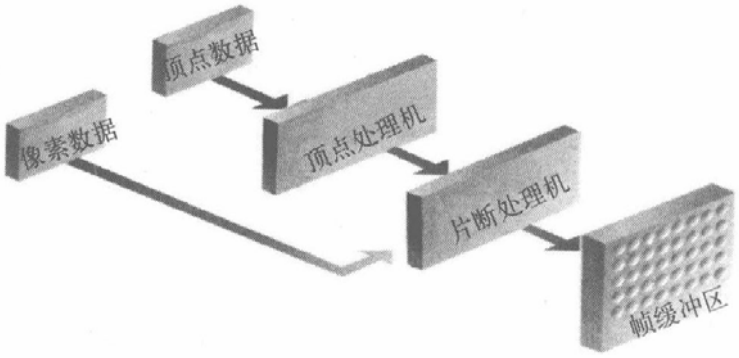


图15-1 OpenGL固定功能管线一览

除了固定功能的管线之外，OpenGL 2.0还增加了一种“可编程着色管线”，可以用小型程序控制顶点和片断的处理。这种小型程序是用OpenGL着色语言（简称GLSL）编写的。在应用程序中，既可以使用固定功能的管线处理数据，也可以使用可编程着色管线处理数据，但是不能同时兼用两者。使用着色器进行数据处理时，对应的固定功能的管线就被禁用。

在OpenGL 3.1中，固定功能的管线废弃并删除了。如果你的实现支持GL_ARB_compatibility扩展，此功能可以在该扩展中使用。

在计算机图形中使用可编程着色器并不是一件新鲜事，但是在交互性图形函数库（例如OpenGL）

中提供这个功能则是最近才有的事。就像计算机图形领域本身一样，可编程着色器本身就是一个很大的主题。本章并不想介绍可编程着色器背后所隐藏的理论，而是着重介绍在OpenGL中使用着色器所需要使用的语言以及如何把着色器集成到OpenGL应用程序中。如果想了解可编程着色管线的更多信息，可以参阅以下著作：

- 《The RenderMan Companion：A Programmer's Guide to Realistic Computer Graphics》，作者Steve Upstill（Addison-Wesley，1990）。本书描述了Rixar的RenderMan着色器语言（电影界用于生成图像的主要可编程着色器语言之一），它描述了可编程着色管线的基础知识，并提供了使用着色器生成图像的许多理论和实践。
- 《Real-Time Shading》，作者John C.Hart、Wolfgang Heidrich、Michael McCool和Marc Olano（AK Peters, Ltd.，2002）。这本较新的书籍描述了图形硬件的可编程着色领域的最新进展，并提供了可编程着色器的不同使用技巧和实现的概述。
- 《The GPU Gems》和《GPU Gems 2》（Addison-Wesley），这两本书描述了着色器实现的许多技术和技巧。

本节首先讨论着色器对顶点和片断处理的影响，接着描述把着色器集成到OpenGL应用程序中需要符合什么条件，然后介绍如何用GLSL编写着色器。有关OpenGL着色语言的细节，请参阅：

- 《The OpenGL Shading Language Specification》，作者John Kessenich、Dave Baldwin和Randi Rost（可以通过http://www.opengl.org下载）。
- 《The OpenGL Shading Language》，第3版，作者Randi Rost和Bill Licea-kane，撰稿者Dan Ginsburg、John M. Kessenich、Barthold Lichtenbelt、Hugh Malan和Mike Weiblen（Addison-Wesley，2009）。

15.1.1 顶点处理

图15-2显示了OpenGL管线的顶点处理过程。中间的着色区域表示顶点处理管线的这部分功能可以由顶点着色器来代替。

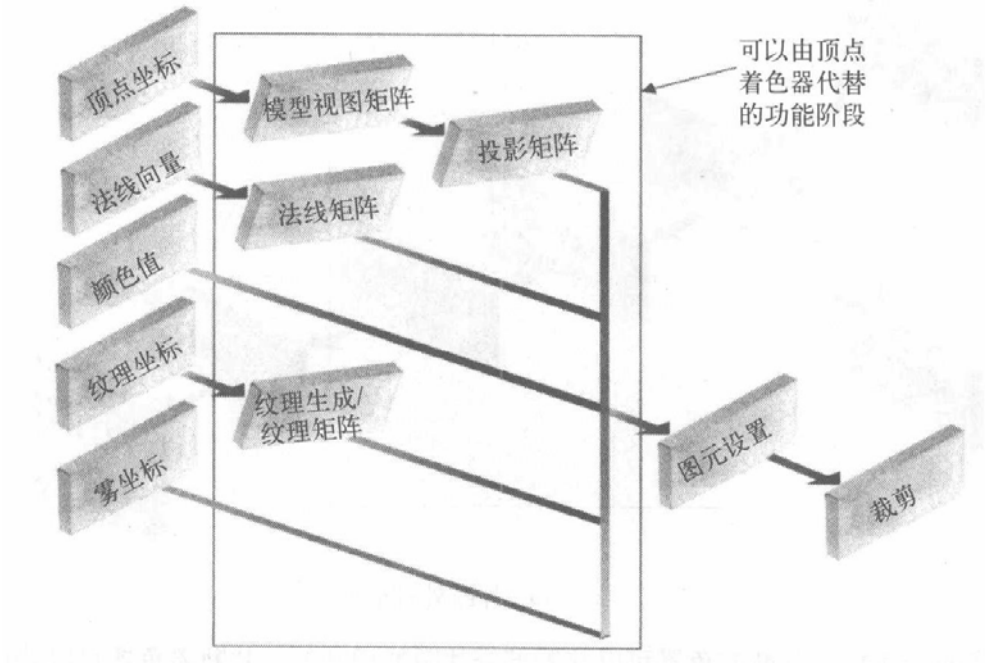


图15-2 顶点处理管线

当OpenGL使用固定功能的管线处理顶点时，它负责提供下面这些值，用于后面的光栅化处理：

- 视觉空间坐标。
- 主颜色和辅助颜色。
- 纹理坐标。
- 雾坐标。
- 点的大小。

根据应用程序启用的特性，顶点管线可能不会对所有上面这些值进行更新。这些值都是应用程序根据glVertex*()和其他顶点数据调用所输入的数据进行计算的。

并不是顶点管线中的所有操作都可以由顶点着色器代替。在执行着色器之后，下面这些操作仍然会出现，就像之前的顶点处理是由固定功能的管线完成的那样：

- 透视除法。
- 视口映射。
- 图元装配。
- 平截头体（视景体）和用户裁剪。
- 背面剔除。
- 双面光照选择。
- 多边形模式处理。
- 多边形偏移。
- 深度范围截取。

15.1.2 片断处理

与图15-2类似，图15-3显示了OpenGL管线中负责像素处理部分的内容。同样，图中着色区域的那部分功能可以由片断着色器所代替。

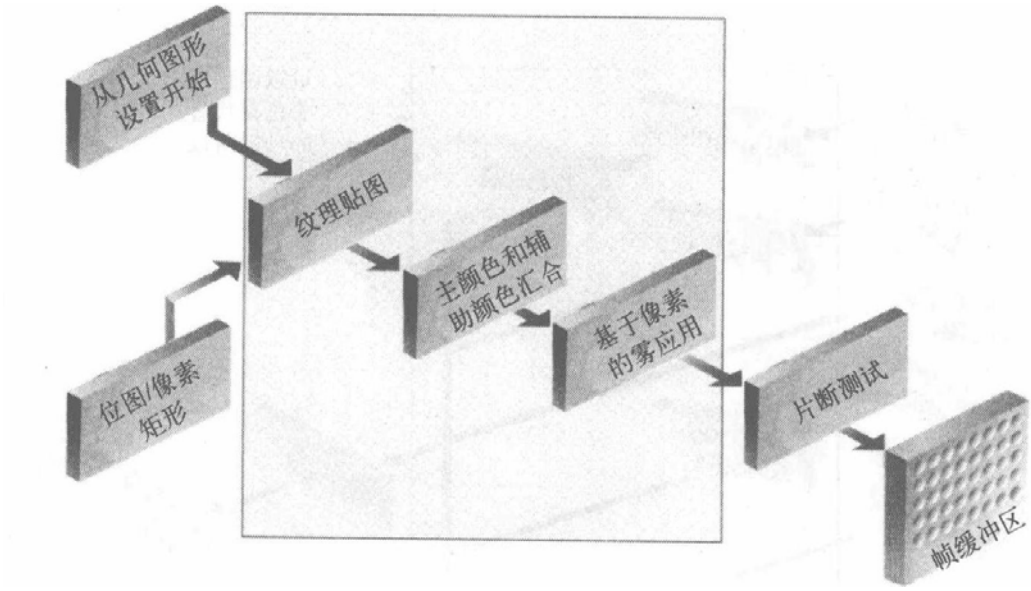


图15-3 片断处理管线

与顶点着色器类似，片断着色器可以接管部分片断处理功能。片断着色器可以处理的操作包括：

- 提取纹理单元，用于纹理贴图。

- 纹理应用。
- 雾。
- 主颜色和辅助颜色汇合。

不论是否使用片断着色器，OpenGL总会执行下面这些操作：

- 单调或平滑着色（控制片断之间的插值）。
- 像素覆盖计算。
- 像素所有权测试。
- 裁剪操作。
- 点画模式应用（在OpenGL 3.0中废弃，在OpenGL 3.1中删除）。
- alpha测试（直到OpenGL 3.0中都存在；OpenGL 3.1删除了这一测试，并用片段着色器中的可用功能替代了它）。
- 深度测试。
- 模板测试。
- alpha混合。
- 对像素进行逻辑操作。
- 颜色值的抖动。
- 颜色掩码操作。

15.2 使用GLSL着色器

15.2.1 着色器示例

示例程序15-1显示了一个简单的顶点着色器。它通过模拟重力加速度，对一个点的位置进行变换。虽然读者此刻可能还不明白许多细节，但是最好能够了解它的基本流程。

示例程序15-1 GLSL顶点着色器示例（1.30版）

```
#version 130
uniform float t;    //Time (passed in from the application)
attribute vec4 vel;//Particle velocity

const vec4 g =vec4(0.0,-9.80,0.0 );

void main()
{
   vec4 position =gl_Vertex;
   position +=t*vel +t*t*g;

   gl_Position =gl_ModelViewProjectionMatrix *position;
}
```

上面的例子是使用GLSL 1.30编写的（可以通过#version 130看出来），它是和OpenGL 3.0兼容的。对于1.40版本的着色器，需要略做修改，参见示例程序15-2。

示例程序15-2 同一GLSL顶点着色器（1.40版）

```
#version 140

uniform float t; //Time (passed in from the application)
```

```
uniform mat4 MVP;//Combined modelview and projection matrices
in vec4 pos;      //Particle position (as a homogenous coordinate)
in vec4 vel;      //Particle velocity

const vec4 g =vec4(0.0,-9.80,0.0 );

void main()
{
   vec4 position =pos;
   position +=t*vel +t*t*g;

   gl_Position =MVP *position;
}
```

15.2.2 OpenGL/GLSL接口

编写OpenGL程序所使用的着色器类似于使用基于编译器的语言（例如C语言）编写程序。我们需要用编译器来分析程序，检查它存在的错误，并把它转换为目标代码。接着，在链接阶段，链接器把一组目标文件组合在一起，形成一个可执行程序。在OpenGL程序中使用GLSL着色器是一个相似的过程，只不过编译器和链接器是OpenGL驱动程序的一部分。

图15-4显示了创建GLSL着色器对象并把它们链接起来创建可执行着色器程序所需要的步骤。

如果想在应用程序中使用顶点或片断着色器，需要按照顺序执行下面的步骤：

对于每个着色器对象：

1) 创建一个着色器对象。

2) 把着色器源代码编译为目标代码。

3) 验证这个着色器已成功通过编译。

然后，为了把多个着色器对象链接到一个着色器程序中，需要：

1) 创建一个着色器程序。

2) 把适当的着色器对象链接到这个着色器程序中。

3) 链接着色器程序。

4) 验证着色器链接阶段已经成功完成。

5) 使用着色器进行顶点或片断处理。

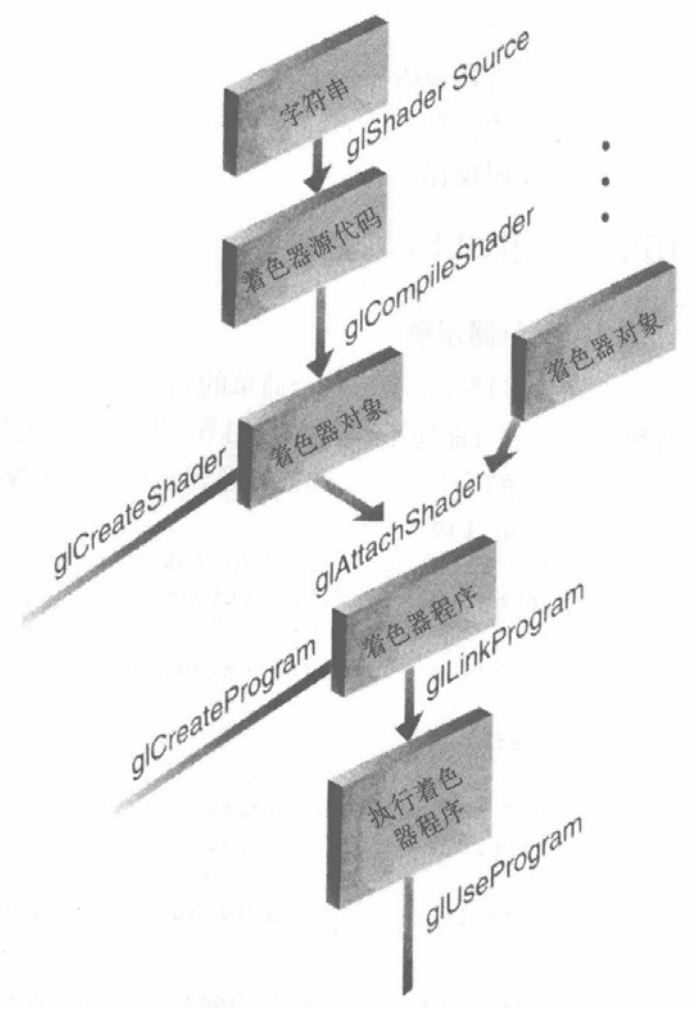


图15-4 着色器创建流程

为什么要创建多个着色器对象？就像可能在不同的程序中复用同一个函数一样，GLSL程序也沿用了这个思路。我们所创建的普通着色器对象可以在多个着色器程序中使用。不必在多个着色器程序中重复使用大量的通用代码，只要把几个适当的着色器对象链接到着色器程序就可以了。

为了创建着色器对象，可以调用glCreateShader()函数。

GLuint **glCreateShader**(GLenum *type*);

创建一个着色器对象。*type*参数的值必须是GL_VERTEX_SHADER或GL_FRAGMENT_SHADER。这个函数的返回值或者是一个非零的整数，或者是零（在出错的情况下）。

创建了着色器对象之后，需要把着色器的源代码与glCreateShader()函数创建的着色器对象相关联。这是通过调用glShaderSource()函数实现的。

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

把着色器源代码与着色器对象*shader*相关联。*string*是一个字符串数组，包含了*count*个GLchar类型的字符串，它们组成了这个着色器的源代码。*string*中的字符串可以是以NULL结尾的。*length*可以是3个值之一。如果*length*为NULL，那么它就假定*string*所使用的每个字符串都是以NULL结尾的。否则，*length*就具有*count*个元素，每个元素分别指定了*string*中对应字符串的长度。如果*length*数组中的某个元素是一个正整数，这个值表示*string*数组中对应字符串的字符数量。如果某个特定元素的值是一个负数，表示*string*数组中的对应字符串是以NULL结尾的。

为了编译着色器对象的源代码，可以使用glCompileShader()函数。

void **glCompileShader**(GLuint *shader*);

编译着色器对象*shader*的源代码。可以调用**glGetShaderiv()**函数（以GL_COMPILE_ STATUS为参数）查询编译结果。

和编译C程序相似，需要判断编译是否成功完成。以GL_COMPILE_STATUS为参数调用glGetShaderiv()函数可以返回编译过程的状态。如果这个函数返回GL_TRUE，编译过程就是成功的，这个对象就可以链接到着色器程序中。如果编译过程失败，可以查看编译日志，确定错误的原因。glGetShaderInfoLog()函数将返回一组因OpenGL实现而异的信息，描述编译过程出现的错误。可以用GL_INFO_LOG_LENGTH为参数调用glGetShaderiv()函数，查询错误日志的大小。

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

返回与着色器对象的最后一次编译相关联的日志。*shader*参数指定了需要查询的着色器对象。日志是以一个NULL结尾的字符串形式返回的，它的长度为*length*个字符，并保存在缓冲区*infoLog*中。这个函数能够返回的最大日志长度是由bufSize参数指定的。

如果*length*参数为NULL，这个函数将不会返回任何字符串。

一旦创建并编译了所有必要的着色器对象，就需要把它们链接起来，创建一个可执行的着色器程序。从本质上说，这个过程与创建着色器对象相似。首先，需要创建一个着色器程序，并把着色器对象链接到这个程序中。可以调用glCreateProgram()函数，它将返回一个着色器程序，供以后处理。

GLuint **glCreateProgram**();

创建一个空的着色器程序。返回值要么是一个非零的整数，要么发生错误的时候返回0。

创建了着色器程序之后，需要把必要的着色器对象链接到这个程序中，以创建可执行程序。这是

通过调用glAttachShader()函数把着色器对象链接到着色器程序实现的。

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

把着色器对象*shader*与着色器程序*program*相关联。着色器对象可以在任何时候链接到着色器程序，但是它的功能只有在成功链接到着色器程序后才可以使用。一个着色器对象可以同时链接到多个着色器程序。

如果需要把一个着色器对象从一个着色器程序中分离出来，以更改着色器的操作，可以调用glDetachShader()函数，并提供适当的着色器对象标识符作为参数。

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

删除着色器对象*shader*与着色器程序*program*的关联。如果*shader*从*program*中分离出来时已经被标记为删除（通过调用**glDeleteShader**()函数），它就会在此时被删除。

在所有必要的着色器对象都被链接到着色器程序之后，就需要把这些对象链接成一个可执行程序。这是通过调用glLinkProgram()函数实现的。

void **glLinkProgram**(GLuint *program*);

对链接到着色器程序*program*的所有着色器对象进行处理，生成一个完整的着色器程序。可以用GL_LINK_STATUS为参数调用**glGetProgramiv**()函数，查询链接操作的结果。如果链接成功，这个函数返回GL_TRUE，否则返回GL_FALSE。

和着色器对象一样，在链接着色器程序时也可能出现错误。可以用GL_LINK_STATUS为参数调用glGetProgramiv()函数，查询链接操作的结果。如果这个函数返回GL_TRUE，表示链接成功，然后就可以用这个程序处理顶点或片断。如果链接失败（即这个函数返回GL_FALSE），可以调用glGetProgramLog()函数获取程序链接信息，了解链接失败的原因。

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

返回与一个着色器程序的最后一次编译相关联的日志。*program*指定了需要查询的着色器程序。日志是以NULL结尾的字符串形式返回的，长度为*length*个字符，并保存在*infoLog*缓冲区中。日志的最大长度是由*bufSize*参数指定的。

如果*length*为NULL，就不会返回任何字符串。

在程序链接成功之后，可以调用glUseProgram()函数，并以这个程序的对象句柄为参数，启动这个顶点或片断着色器程序。为了恢复使用固定功能的管线，可以向这个函数传递0作为program参数。

void **glUseProgram**(GLuint *program*);

使用着色器程序*program*进行顶点或片断处理（取决于**glCreateShader**()函数创建的这个着色器的类型）。如果*program*参数为0，就不使用任何着色器进行处理，OpenGL就会恢复为固定功能管线的操作。（在OpenGL 3.1中，由于没有固定功能管线，结果是未定义的，但是不会产生错误）。

当一个着色器程序处于使用中时，它可以链接新的着色器对象、编译已链接的着色器对象，或分离以前链接的着色器对象。它还可以重新链接。如果链接成功，新的成功链接的着色器程序就会取代原先的活动程序。如果链接失败，当前绑定的着色器程序就仍然处于活动状态，并不会被替换，除非用**glUseProgram**()函数指定一个新程序，或者这个程序被成功地进行了重新链接。

示例程序15-3显示了如何用GLSL创建和链接顶点着色器。创建和链接片断着色器的过程实际上与之相同。

示例程序15-3 创建和链接GLSL着色器

```
GLuint shader,program;
GLint compiled,linked;
const GLchar*shaderSrc [] ={
   "void main()"
   "{"
   "   gl_Position =gl_ModelViewProjectionMatrix *gl_Vertex;"
   "}"
};

shader =glCreateShader(GL_VERTEX_SHADER);

glShaderSource(shader,1,shaderSrc,NULL);
glCompileShader(shader);

glGetShaderiv(shader,GL_COMPILE_STATUS,&compiled );

if (!compiled){
   GLint length;
   GLchar*log;
   glGetShaderiv(shader,GL_INFO_LOG_LENGTH,
      &length );
   log =(GLchar*)malloc(length);
   glGetShaderInfoLog(shader,length,&length,log);
   fprintf(stderr,"compile log ='%s '\n ",log);
   free(log);
}

program =glCreateProgram();

glAttachShader(program,shader);
glLinkProgram(program);

glGetProgramiv(program,GL_LINK_STATUS,&linked );

if (linked){
   glUseProgram(program);
}else {
   GLint length;
   GLchar*log;
   glGetProgramiv(progam,GL_INFO_LOG_LENGTH,&length );

   log =(GLchar*)malloc(length);
   glGetProgramInfoLog(program,length,&length,log);
   fprintf(stderr,"link log ='%s '\n " ,log);
   free(log);
}
```

为了释放与着色器对象和着色器程序相关联的资源，可以通过调用对应的删除函数来删除这些对象。如果着色器对象或着色器程序在删除时处于活动状态，这个对象就只是标记为删除。当着色器程序不再被使用或着色器对象从所有相关的着色器程序中分离出来时，它们才会实际删除。

void **glDeleteShader**(GLuint *shader*);

删除着色器对象*shader*。如果*shader*当前链接到一个或多个活动的着色器程序，这个对象就标记为已删除，一旦这个着色器对象不再由任何着色器程序使用，它就会实际删除。

void **glDeleteProgram**(GLuint *program*);

如果着色器程序*program*当前并未在任何渲染环境中使用，这个函数将立即删除这个着色器程序。否则，它就把这个着色器程序标记为删除，当它不再使用时再进行实际删除。

如果着色器程序program当前并未在任何渲染环境中使用，这个函数将立即删除这个着色器程序。否则，它就把这个着色器程序标记为删除，当它不再使用时再进行实际删除。

GLboolean **glIsProgram**(GLuint *program*);

如果*program*是一个着色器程序的名称，这个函数返回GL_TRUE。如果*program*是0或者并不是某个着色器程序的名称，这个函数返回GL_FALSE。

GLboolean **glIsShader**(GLuint *shader*);

如果*shader*是一个着色器对象的名称，这个函数返回GL_TRUE。如果*shader*是0或者不是某个着色器对象的名称，这个函数返回GL_FALSE。

为了帮助程序员开发着色器，OpenGL函数glValidateProgram()可用于验证一个着色器是否可以在当前的OpenGL状态下执行。根据底层的OpenGL实现，这个函数所执行的程序验证操作还可能返回相关的性能特征信息或者其他着色器在这种OpenGL实现上执行的特定信息。程序验证的操作方式就像编译着色器程序一样，在所有的着色器对象都链接到着色器程序之后再简单地调用glValidateProgram()函数即可。类似地，可以调用glGetProgramiv()函数（以GL_VALIDATE_STATUS为参数），查询程序验证操作的结果。

void **glValidateProgram**(GLuint *program*);

根据当前的OpenGL状态设置，对着色器程序*program*进行验证。如果验证通过，GL_VALIDATE_STATUS的值将设置为GL_TRUE，表示这个程序可以在当前的OpenGL环境中运行。否则，它就被设置为GL_FALSE。可以调用**glGetProgramiv**()函数查询GL_VALIDATE_STATUS状态的值。

15.3 OpenGL着色语言

本节简单概括OpenGL所使用的着色语言，一般称为GLSL。GLSL着色器有许多特性与C++和Java相似，它既可用于顶点着色器，也可用于片断着色器。不过，它的有些特性只适用于其中一种着色器。我们首先描述GLSL的需求、类型以及顶点和片断着色器所共享的一些语言结构，然后分别讨论每种着色器特有的特性。

15.4 使用GLSL创建着色器

15.4.1 程序起点

着色器程序就像C程序一样，是从main()函数开始执行的。每个GLSL着色器程序都是从下面的结构开始执行的：

```
void
main()
{
    // Your code goes here
}
```

“//”表示注释，从它出现之处到当前行的行末之间的所有内容都属于注释。也可以使用C类型的多行注释符“/*”和“*/”。但是，和ANSI C不同，main()函数并不返回一个整型值，它的类型被声明为void。

尽管这是一个完全合法的GLSL顶点或片断着色器程序，可以顺利地编译和运行，但是它基本上没有实现什么功能。接下来，我们将描述GLSL的变量以及它们的操作。

另外，和C以及它的派生语言一样，着色器程序的语句也是以分号结尾的。

15.4.2 声明变量

GLSL是一种强类型语言。也就是说，每个变量都必须进行声明，并具有相关的类型。变量名的规则与C相同：可以使用字母、数字和下划线（_）来组成变量名，但是变量名的第一个字符不能是数字。

表15-1显示了GLSL可以使用的3种基本数据类型。

表15-1 GLSL的基本数据类型

类型	描述
Float	类似IEEE的浮点值
int	有符号整型值
uint	无符号的整型值
bool	布尔型

GLSL还提供了一种额外的类型，称为采样器(sampler)，作为访问纹理图像的不透明句柄。各种类型的采样器以及它们的用法将在第15.6节中描述。

注意：有些OpenGL实现可能并不会严格地实现这些类型，而有些则会严格实现。只要它们的操作在语法上和逻辑上都是正确的，底层的实现可以有所不同。例如，整数也可以存储在浮点寄存器中。因此，我们在程序中不要理所当然地做出某些假设，例如以为最大整数就一定是2^{15}。

变量的作用域

所有的变量都必须进行声明，但是它们可以在使用之前的任何时候进行声明（这一点和C不同，C的变量必须在一个代码块的起始部分声明）。GLSL的作用域规则与C++的规则非常相似：

- 在所有函数定义之外声明的变量具有全局作用域，它们在着色器程序的所有函数中均可见。
- 在一对花括号内（例如函数定义、if语句等）声明的变量只在这对花括号所决定的作用域中有效。
- 循环迭代变量，例如循环中的i：

```
for (int i = 0; i < 10; ++i) {
    // loop body
}
```

它们的作用域仅限于循环体内。

变量的初始化

变量可以在声明的时候进行初始化。例如：

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

整型常量可以用八进制、十进制或十六进制来表示。如果数值前面有一个负号（可选的），表示这个常量取反；后面有一个“u”或“U”，表示是一个无符号整型值。

浮点值必须包括一个小数点，除非它是用科学记数法表示的（例如3E-7），并且可以像C语言一样包含一个可选的“f”或“F”后缀。

布尔型为true或false，可以初始化为这两个值之一，也可以作为布尔型表示式的求值结果。

构造函数

如前所述，GLSL是一种强类型语言，甚至比C++更为严格。不同类型的值不存在隐式类型转换。例如：

```
int f = 10.0;
```

将会导致编译错误，因为它试图向一个浮点变量赋一个整型值，整型值和整型向量将会被隐式地转换为等价的浮点值或浮点向量；值的任何转换都需要通过转换函数（类似C++的构造函数）来进行。例如：

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

上面使用了int()函数来完成转换。其他类型也具有类似的转换函数：float()和bool()。这些函数显示了GLSL的另一个特性：操作符重载，每个函数接受各种不同的输入类型，但使用相同的基本函数名称。我们将在稍后再讨论这个问题的细节。

15.4.3 聚合类型

GLSL的3种基本类型可以进行组合，更好地与OpenGL的数据值相匹配，并且能够简化计算方法。

首先，GLSL支持每种基本类型的二维、三维或四维向量。另外，它还支持2×2、3×3和4×4的浮点矩阵。表15-2列出了合法的向量和矩阵类型。

表15-2 GLSL向量和矩阵类型

基本类型	二维向量	三维向量	四维向量	矩阵类型
float	*vec2*	*vec3*	*vec4*	*mat2, mat3, mat4* *mat2x2, mat2x3, mat2x4* *mat3x2, mat3x3, mat3x4* *mat4x2, mat4x3, mat4x4*
int	*ivec2*	*ivec3*	*ivec4*	—
uint	*uvec2*	*uvec3*	*uvec4*	—
bool	*bvec2*	*bvec3*	*bvec4*	—

矩阵类型列出了所有各维，例如mat4x3，第一个值指定了列的数量，第二个值指定了行的数量。这些类型的变量可以像对应的标量变量一样进行初始化：

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

并且也可以进行类型转换：

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

还可以使用向量构造函数对向量进行截短或拉长。如果一个较长的向量传递给一个较短向量的构造函数，这个向量便被截短为适当的长度。

```
vec4 color;
vec3 RGB = vec3(color); // now RGB only has three elements
```

向量拉长的情况与此类似。标量值也可以提升为向量，如下所示：

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

矩阵的构建方式与此相似，可以初始化为对角矩阵或完整的矩阵。

在对角矩阵的情况下，可以向矩阵的构造函数传递单个值，这个矩阵的对角元素便会设置为这个值，矩阵中的其他所有元素都设置为0。如下所示：

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

也可以通过在矩阵的构造函数中指定每个元素的值来创建一个矩阵。可以组合使用标量和向量来指定元素的值，只要提供了足够的值，并且每个列的值以相同的方式指定。另外，矩阵是按照列主序的顺序指定的，意味着这些值首先用于填充列，然后再用于填充行（与C语言对二维数组的初始化顺序相反）。

例如，可以使用下面的任何一种方法对一个3×3的矩阵进行初始化：

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

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

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

甚至使用下面的方法：

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

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

它们都产生同一个矩阵：

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

访问向量和矩阵中的元素

可以对向量和矩阵中的单独元素进行访问和赋值。向量提供了两种类型的访问方式：名称成分方

法和类似数组的方法。矩阵则使用类似二维数组的方法进行访问。

可以通过名称来访问一个向量的成分，如下所示：

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

也可以使用下标方式。下面两行代码产生的结果与上面两行代码的结果相同：

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

事实上，如表15-3所示，可以使用3组成分名称，它们的作用相同。之所以提供多组名称的原因是便于区分所使用的是什么操作。

表15-3 向量成分访问名称

成分访问名称	描 述
(x, y, z, w)	与位置相关的成分
(r, g, b, a)	与颜色相关的成分
(s, t, p, q)	与纹理坐标相关的成分

向量成分还有一种常见的访问方式，称为搅拌式（swizzling）成分访问，可以用于颜色值以及颜色空间转换。例如，可以使用下面的方法根据输入颜色的红色成分来指定一个亮度值：

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

类似地，如果需要在向量中移动成分，可以像下面这样：

```
color = color.abgr; // reverse the components of a color
```

唯一的限制是一条语句中的一个变量只能使用一组成分。也就是说，不能像下面这样做：

```
vec4 color = otherColor.rgz;// Error: 'z' is from a
// different group
```

另外，如果试图访问一个类型所提供的范围之外的元素，编译器就会报错。例如：

```
vec2 pos;
float zPos = pos.z; // Error: no 'z' component in 2D vectors
```

可以使用数组形式来访问矩阵的元素。可以访问单个标量值，也可以访问一个元素数组：

```
mat4 m = mat4(2.0);
vec4 zVec = m[2];
float yScale = m[1][1]; // or m[1].y works as well
```

结构

可以在一个结构中把一组不同的类型从逻辑上组合在一起。结构可以方便地把一组相关的数据传递给函数。当定义结构时，它会自动创建一种新类型，并隐式地定义一个构造函数，将组成这个结构的元素类型作为参数。

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

Particle p = Particle(10.0, pos, vel); // pos, vel are vec3's
```

类似地，为了引用结构中的元素，可以使用熟知的点操作符“.”。

数组

GLSL还支持任何类型的一维数组，包括结构。和C语言一样，下标是用一对方括号（[]）表示的。

一个长度为n的数组的元素范围是从0~n-1。但是，和C语言不同，GLSL不允许使用负值的下标，也不允许使用二维数组。

数组可以声明为指定了长度或未指定长度。可以使用未指定长度的数组作为一个数组变量的前向声明，以后再把它声明给适当的长度。数组声明使用方括号记法，如下所示：

```
float coeff[3]; // an array of 3 floats
float[3] coeff; // same thing
int indices[];   // unsized. Redeclare later with a size
```

在GLSL中，数组是第一类的类型，意味着具有构造函数，可能作为函数的参数和返回类型使用。为了静态地初始化一个数组的值，可以按照下面的方式使用构造函数：

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

在构造函数中，表示维数的值是可选的。

另外，与Java相似，GLSL数组提供了一个隐式的方法length()，用于报道它们的元素数量。如果需要对一个数组中的所有值进行操作，可能需要像下面这样使用length()方法：

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

类型限定符

类型也可以使用限定符，对它们的行为进行限定。GLSL定义了4种限定符，见表15-4。

注意：在GLSL 1.30版以前，顶点着色器的输入变量用关键字“attribute”来限定。类似地，片段着色器的输入变量（它和顶点着色器的输出变量相对应）用关键字“varying”来限定。预料到潜在地增加更多的着色阶段，这些关键字都由更加通用的“in”和“out”形式替代。在GLSL 1.40中，“attribute”和“varying”都删除了（尽管在使用GL_ARB_compatibility扩展的时候仍然可以用它们）。

表15-4 GLSL类型限定符

类型限定符	描 述
const	把变量标记为只读的编译期常量
in	指定变量为着色器阶段的一个输入
out	指定变量为着色器阶段的一个输出
uniform	指定这个值从应用程序传递给着色器，并在一个特定的图元中保持为常量值

所有的存储类型限定符都适用于全局作用域的变量。另外，const还适用于局部变量和函数的形参。

const类型限定符

和C--样，const类型限定符表示变量是只读的。例如，下面这条语句：

```
const float Pi = 3.141529
```

把变量*Pi*设置为的π近似值。有了const限定符之后，在声明之后再修改这个变量的值便是错误的。因此，这个变量必须在声明的同时进行初始化。

in类型限定符

in限定符用来限定一个着色器阶段的输入。这些输入可能是顶点属性（用于顶点着色器），或者是插值变量（用于片段着色器）。

片段着色器可以进一步限定其输入值，这要使用那些只有和in关键字组合使用才有效的额外关键字。这些关键字见表15-5。

表15-5 额外的in关键字限定符（用于片段着色器输入）

in关键字限定符	说 明
centroid	在打开多点采样的时候，强迫一个片段输入变量的采样位于图元像素覆盖的区域内
smooth	以透视校正的方式插值片段输入变量
flat	不对片段输入插值（例如，对于所有的片段，输入都是相同的，就像在单调着色中一样）
noperspective	线性插值片段变量

例如，如果想要一个flat着色、centroid采样的片段输入，应该在片段着色器中指定：

```
flat centroid in fragment;
```

out类型限定符

out限定符用来限定一个着色器阶段的输出，例如，来自一个顶点着色器的齐次坐标变换，或者来自一个片段着色器的最终片段颜色。

顶点着色器可以使用centroid关键字限定其输出值，该关键字在这里的含义与其对片段输入的含义相同。此外，centroid限定的任何顶点着色器输出，都必须有一个匹配的片段着色器输入也被centroid限定（例如，片段着色器必须有一个变量具有与顶点着色器中相同的声明）。

uniform类型限定符

uniform限定了表示一个变量的值将由应用程序在着色器执行之前指定，并且在图元的处理过程中不会发生变化。uniform变量是由顶点着色器和片断着色器共享的，它们必须声明为全局变量。任何类型的变量，包括结构和数组，都可以声明为uniform变量。

考虑这样一个着色器：它在对一个图元进行着色时使用了一种额外的颜色。可以声明一个uniform变量，把这个信息传递给着色器。在这个着色器中，可以声明下面的变量：

```
uniform vec4 BaseColor;
```

在着色器内部，可以通过名字来引用BaseColor，但是为了在应用程序中设置它的值，需要做一些额外的工作。当GLSL编译器链接到着色器程序后，它会创建一个表格，其中包含了所有的uniform变量。为了在应用程序中设置BaseColor的值，需要获取BaseColor在表中的索引值，这是通过glGetUniformLocation()函数实现的。

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

返回与着色器程序*program*相关联的uniform变量*name*的索引值。*name*是一个以NULL结尾的字符串（中间没有空格）。如果*name*并不对应于活动着色器程序*program*的其中一个uniform变量，或者它是一个保留的着色器变量名称（具有gl_前缀），这个函数就返回−1。

*name*可以是单个变量名称，也可以是一个数组元素（在名称中包含带方括号的下标），或者是某个结构的一个字段（由*name*指定，中间是点号，后面是字段名，和着色器中的用法相同）。对于uniform变量的数组，查询数组第一个元素的索引时，可以只指定数组名（例如arrayName），或者指定数组第一个元素的索引（例如arrayName[0]）。

这个函数的返回值将不会修改，除非这个着色器程序被重新链接（参见**glLinkProgram**()函数的说明）。

一旦获取了一个uniform变量的相关索引值，就可以使用glUniform*()或glUniformMatrix*()函数设置这个uniform变量的值。

void **glUniform**{1234}{if ui}(GLint *location*, *TYPE value*);

void **glUniform**{1234}{if ui}**v**(GLint *location*, GLsizei *count*,
const *TYPE* **values*);

void **glUniformMatrix**{234}fv(GLint *location*, GLsizei *count*,
GLboolean *transpose*, const GLfloat **values*);

void **glUniformMatrix**{2X3,2X4,3X2,3X4,4X2,4X3}fv(GLint *location*,
GLsizei *count*, GLboolean *transpose*,
const GLfloat **values*);

设置与索引*location*相关联的uniform变量的值。这个函数的向量形式把count（1~4个值，取决于所调用的是哪个**glUniform*()**函数）组值加载到从*location*开始的*uniform*变量中。如果这个位置是一个数组的起始位置，那么这个数组的*count*个元素就按顺序加载。

这个函数的浮点形式可以加载单个浮点值、一个浮点向量、一个浮点数组或者一个包含了浮点向量的数组。

这个函数的整型形式可以用于更新单个整型值、一个整型向量、一个整型数组或者一个包含整型向量的数组。另外，它还可以用于加载单个纹理采样器或加载纹理采样器数组。

对于**glUniformMatrix**(){234}fv()，将从values加载count组2×2、3×3或4×4的矩阵。

对于**glUniformMatrix**{2x3, 2x4, 3x2, 3x4, 4x2, 4x3}fv()，将从*values*加载*count*组类似维度的矩阵。如果*transpose*是GL_TRUE，那么values是按行主序指定的（就像C语言的数组一样），如果是GL_FALSE，那么*values*是按列主序指定的（和glLoadMatrix()使用的顺序相同）。

示例程序15-4显示了如何获取一个uniform变量的索引值以及如何对它进行赋值。

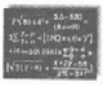
示例程序15-4 获取一个uniform变量的索引值，并对它进行赋值

```
GLint timeLoc;/*Uniform index for variable "time " in shader */
GLfloat timeValue;/*Application time */

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

uniform变量可以在指定的uniform块中声明，uniform块支持着色器的共享及其他功能。使用我们刚刚讨论的那些程序是无法访问那些uniform变量的，要访问它们，需要使用接下来描述的其他程序。

15.5 uniform块



高级话题

随着着色器程序变得越来越复杂，它们用到的uniform变量的数目将会增加。经常会出现几个着色器程序中使用同一个uniform值的情况。由于在连接着色器的时候（例如，调用glLinkProgram()的时候）产生uniform位置，索引可能会变化，即便（对我们来说）uniform变量的值是相同的。统一缓冲区对象（uniform buffer object）提供了一种方法，既优化了uniform变量的访问，又使得可以跨着色器程序共享uniform值。

可以想象的到，给定的uniform变量在应用程序和着色器中都可以存在，我们既要修改着色器，也要使用OpenGL程序来设置统一缓冲区对象。

注意：uniform块将添加到OpenGL 3.1中。

15.5.1 在着色器中指定uniform变量

要使用glMapBuffer()这样的程序来访问uniform变量的一个集合（参见第2.7节了解详细内容），需要在着色器中稍微修改一下它们的声明。在一个uniform块中，我们成组地声明uniform变量，而不是单独地声明每个uniform变量，这就像在结构中的做法一样。用uniform关键字指定一个uniform块。然后，用一对花括号把想要的所有变量都包含到一个块中，如示例程序15-5所示。

示例程序15-5 声明一个uniform变量块

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

所有的类型，除了采样器，都允许放到一个uniform块中。此外，uniform块必须声明为全局作用域。

uniform块布局控制

有各种限定符可以用来指定在一个uniform块中如何布局变量。这些限定符可以针对每一个单个的uniform块使用，或者用来指定所有后续的uniform块如何安排（在指定一个布局声明之后）。可能的限定符见表15-6。

表15-6 uniform块的布局限定符

布局限定符	说　明
shared	指定uniform块在多个程序之间共享（这是默认的共享设置）
packed	布局uniform块以使其使用的内存最小化；然而，这通常不允许跨程序共享
std140	为uniform块使用OpenGL规范中描述的默认布局
row_major	使得uniform块中的矩阵按照行主序的方式存储
column_major	指定矩阵应该按照列主序的方式存储（这是默认的排序）

例如，要指定一个单个的uniform块共享，并且按照行主序方式存储矩阵，应该按照如下方式声明它：

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

括号中的多个限定选项必须用逗号隔开。要影响到所有后续uniform块的布局，使用如下的结构：

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

通过这样的指定，该行之后声明的所有uniform块都将使用该布局，直到全局布局修改，或者除非它们自己包含了一个布局覆盖了对其声明的指定。

15.5.2 访问在uniform块中声明的uniform变量

尽管命名了uniform块，在其中声明的uniform变量并没有被该名称所限定。也就是说，uniform块并不能作为一个uniform变量的作用域，因此，在两个uniform块中声明具有相同名称的两个变量将会导致错误。然而，访问一个uniform变量的时候，没必要使用uniform块的名字。

从应用程序中访问uniform块

由于uniform构成了一座桥梁，以便在着色器和应用程序之间共享数据，需要找到指定的uniform块中的各个uniform变量在着色器中的偏移量。一旦知道了这些变量的位置，可以用数据把它们初始

化，就像可以对任何类型的缓冲区对象所做的那样（例如，使用glBufferData()这样的调用）。

首先，假设已经知道了在应用程序的着色器中使用的uniform块的名字。初始化uniform块中uniform变量的第一步，是针对一个给定的程序获取块的索引。调用glGetUniformBlockIndex()，将返回完成把uniform变量映射到应用程序的地址空间所需的一段基本信息。

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

返回和program相关的*uniformBlockName*所指定的具名uniform的索引。如果*uniformBlockName*不是*program*的一个有效uniform块，将返回GL_INVALID_INDEX。

要初始化与uniform块相关的一个缓冲区对象，需要使用glBindBuffer()函数把缓冲区对象绑定到一个GL_UNIFORM_BUFFER目标（参见第2.7.1节了解详细内容）。

一旦初始化了一个缓冲区对象，需要确定从着色器留出多大空间来容纳指定的uniform块中的变量。为了做到这一点，我们使用函数glGetActiveUniformBlockiv()来请求GL_UNIFORM_BLOCK_DATA_SIZE，它返回了编译器生成的块的大小（根据选择何种uniform块布局，编译器可能确定删除着色器中不使用的uniform变量）。glGetActiveUniformBlockiv()可以用来获取和一个指定的uniform块相关的其他参数。B.1节给出了一个完整的选项列表。

在获取了uniform块的索引之后，需要把一个帧缓冲区对象和该块联系起来。这么做的最常见的方法是调用glBindBufferRange()，或者，如果所有的缓冲区存储都用于uniform块，调用glBindBufferBase()。

---

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

void **glBindBufferBase**(GLenum *target*, GLuint *index*, GLuint *buffer*);

将缓冲区对象*buffer*和与*index*相关的uniform块关联起来。*target*可以是GL_UNIFORM_BUFFER（用于uniform块）或者GL_TRANSFORM_FEEDBACK_BUFFER（用于变换反馈，参见第15.10节）。index是和一个uniform块相关联的索引。*offset*和*size*指定了用来匹配*uniform*缓冲区的*buffer*的起始索引和范围。

调用**glBindBufferBase()**，等同于使用*offset*等于0和size等于缓冲区对象的大小来调用**glBindBufferRange()**。

这些调用会产生各种OpenGL错误：如果size小于0，如果*offset* + *size*大于缓冲区的大小，如果*offset*或*size*不是4的倍数，或者如果*index*小于0，或者大于或等于查询GL_MAX_UNIFORM_BUFFER_BINDINGS时返回的值，将会产生一个GL_INVALID_VALUE。

一旦一个具名的uniform块和一个缓冲区对象建立了关联，可以使用会影响到缓冲区值的任何命令来初始化或修改该块中的值，参见第2.7.4节的介绍。

与允许连接器分配一个块绑定然后查询分配的值这一过程相比较，你可能也想指定把一个特定名称的uniform块绑定到一个缓冲区对象。如果有多个着色器程序要共享一个uniform块，可能要采用这种方法。它避免了为每个程序分配一个不同的块索引。要显式地控制一个uniform块的绑定，在调用glLinkProgram()之前调用glUniformBlockBinding()。

GLint **gUniformBlockBinding**(GLuint *program*,
GLuint *uniformBlockIndex*,
GLuint *uniformBlockBinding*)

为*program*显式地把*uniformBlockIndex*分配给*uniformBlockBinding*。

uniform变量在一个具名的uniform块中的布局，由指定的布局限定符来控制，而这在编译和连接uniform块的时候进行。如果使用默认的布局指定，需要确定uniform块中的每个变量的offset和数据存储size。为了做到这一点，我们将使用一对调用：调用glGetUniformIndices()获取一个特定的具名uniform变量的索引，调用glGetActiveUniformsiv()获取这个特定索引的offset和size，参见示例程序15-6。

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

为*program*返回数组*uniformNames*中的名字所指定的*uniformCount*个*uniform*变量的索引，返回的索引在*uniformIndices*中。假设*uniformNames*中的每个名字都以NULL结束，并且，*uniformNames*和*uniformIndice*这两个数组中都有*uniformCount*个元素。

如果*uniformNames*中列出的名字不是一个有效的*uniform*变量，在*uniformIndices*对应的元素中返回GL_INVALID_INDEX值。

示例程序15-6 初始化一个具名的uniform块中的uniform变量：ubo.c

```
/*Vertex and fragment shaders that share a block of uniforms
**named "Uniforms " */

const char*vShader ={
   "#version 140 \n "
   "uniform Uniforms {"
   "   vec3 translation;;"
   "   float scale;;"
   "   vec4 rotation;;"
   "   bool enabled;;"
   "};"
   "in vec2 vPos;"
   "in vec3 vColor;"
   "out vec4 fColor;"
   "void main()"
   "{"
   "   vec3   pos =vec3(vPos,0.0 );"
   "   float  angle =radians(rotation [0] ));"
   "   vec3   axis =normalize(rotation.yzw );"
   "   mat3   I =mat3(1.0 );"

   "   mat3   S =mat3(0,-axis.z,axis.y,"
   "                  axis.z,0,-axis.x,"
   "                 -axis.y,axis.x,0 );"
   "   mat3 uuT =outerProduct(axis,axis );"
   "   mat3 rot =uuT +cos(angle)*(I -uuT)+sin(angle)*S;"
   "   pos *=scale;"
   "   pos *=rot;"
```

```
    "    pos +=translation;"
    "    fColor =vec4(scale,scale,scale,1 );"
    "    gl__Position =vec4(pos,1 );"
    "}"
};

const char*fShader ={
    "#version 140 \n "
    "uniform Uniforms {"
    "    vec3 translation;;"
    "    float scale;;"
    "    vec4 rotation;;"
    "    bool enabled;;"
    "};"
    "in vec4 fColor;"
    "out vec4 color;"
    "void main()"
    "{"
    " color =fColor;"
    "}"
};

/*Helper function to convert GLSL types to storage sizes */
size_t
TypeSize(GLenum type )
{
    size_t size;

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

    switch(type ){
        CASE(GL_FLOAT,               1,GLfloat );
        CASE(GL_FLOAT_VEC2,          2,GLfloat );
        CASE(GL_FLOAT_VEC3,          3,GLfloat );
        CASE(GL_FLOAT_VEC4,          4,GLfloat );
        CASE(GL_INT,                 1,GLint );
        CASE(GL_INT_VEC2,            2,GLint );
        CASE(GL_INT_VEC3,            3,GLint );
        CASE(GL_INT_VEC4,            4,GLint );
        CASE(GL_UNSIGNED_INT,        1,GLuint );
        CASE(GL_UNSIGNED_INT_VEC2,   2,GLuint );
        CASE(GL_UNSIGNED_INT_VEC3,   3,GLuint );
        CASE(GL_UNSIGNED_INT_VEC4,   4,GLuint );
        CASE(GL_BOOL,                1,GLboolean );
        CASE(GL_BOOL_VEC2,           2,GLboolean );
        CASE(GL_BOOL_VEC3,           3,GLboolean );
        CASE(GL_BOOL_VEC4,           4,GLboolean );
        CASE(GL_FLOAT_MAT2,          4,GLfloat );
        CASE(GL_FLOAT_MAT2x3,        6,GLfloat );
        CASE(GL_FLOAT_MAT2x4,        8,GLfloat );
        CASE(GL_FLOAT_MAT3,          9,GLfloat );
```

```
        CASE(GL_FLOAT_MAT3x2,       6,GLfloat );
        CASE(GL_FLOAT_MAT3x4,       12,GLfloat );
        CASE(GL_FLOAT_MAT4,         16,GLfloat );
        CASE(GL_FLOAT_MAT4x2,       8,GLfloat );
        CASE(GL_FLOAT_MAT4x3,       12,GLfloat );
        default:
           fprintf(stderr,"Unknown type:0x%x \n ",type );
           exit(EXIT_FAILURE );
           break;
}
#undef CASE

   return size;
}

void
init()
{
   GLuint program;

   glClearColor(1,0,0,1 );

   /*Compile and load vertex and fragment shaders (see
   **LoadProgram.c */
   program =LoadProgram(vShader,fShader );
   glUseProgram(program );

   /*Initialize uniform values in uniform block "Uniforms " */
   {
     GLuint   uboIndex;
     GLint    uboSize;
     GLuint   ubo;
     GLvoid   *buffer;

     /*Find the uniform buffer index for "Uniforms ",and
     **determine the block 's sizse*/
     uboIndex =glGetUniformBlockIndex(program,"Uniforms " );

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

     buffer =malloc(uboSize );

     if (buffer ==NULL ){
        fprintf(stderr,"Unable to allocate buffer \n " );
        exit(EXIT_FAILURE );
     }
     else {
        enum {Translation,Scale,Rotation,
           Enabled,NumUniforms };

        /*Values to be stored in the buffer object */
```

```
        GLfloat   scale =0.5;
        GLfloat   translation [] ={0.1,0.1,0.0 };
        GLfloat   rotation [] ={90,0.0,0.0,1.0 };
        GLboolean enabled =GL_TRUE;

        /*Since we know the names of the uniforms
        **in our block,make an array of those values */
        const char*names [NumUniforms]={
          "translation ",
          "scale ",
          "rotation ",
          "enabled "
        };

        /*Query the necessary attributes to determine
        **where in the buffer we should write
        **the values */
        GLuint    indices [NumUniforms];
        GLint     size [NumUniforms];
        GLint     offset [NumUniforms];
        GLint     type [NumUniforms];

        glGetUniformIndices(program,NumUniforms,
           names,indices );
        glGetActiveUniformsiv(program,NumUniforms,indices,
           GL_UNIFORM_OFFSET,offset );
        glGetActiveUniformsiv(program,NumUniforms,indices,
           GL_UNIFORM_SIZE,size );
        glGetActiveUniformsiv(program,NumUniforms,indices,
           GL_UNIFORM_TYPE,type );
        /*Copy the uniform values into the buffer */
        memcpy(buffer +offset [Scale],&scale,
           size [Scale] *TypeSize(type [Scale]));
        memcpy(buffer +offset [Translation],&translation,
           size [Translation] *TypeSize(type [Translation]));
        memcpy(buffer +offset [Rotation],&rotation,
           size [Rotation] *TypeSize(type [Rotation] ));
        memcpy(buffer +offset [Enabled],&enabled,
           size [Enabled] *TypeSize(type [Enabled]));

        /*Create the uniform buffer object,initialize
        **its storage,and associated it with the shader
        **program */
        glGenBuffers(1,&ubo );
        glBindBuffer(GL_UNIFORM_BUFFER,ubo );
        glBufferData(GL_UNIFORM_BUFFER,uboSize,
           buffer,GL_STATIC_RAW );

        glBindBufferBase(GL_UNIFORM_BUFFER,uboIndex,ubo );
        }
    }
    ...
}
```

15.5.3 计算不变性

GLSL并不保证不同的着色器中的两个相同的计算将产生完全相同的结果。这种情况与在CPU上执行的计算性应用程序并无不同。在计算性应用程序中，由于指令顺序的累积差别，编译后的指令顺序可能会产生微小的差别。有些多道渲染算法要求在每道着色器渲染时计算的位置完全相同，此时这些微小的错误就可能会造成问题。GLSL提供了一种方法，使用invariant关键字，强制在着色器之间实行这种类型的不变性。

invariant类型限定符

invariant限定符可以应用于顶点着色器的任何输出varying变量。这种变量可以是内置的变量，也可以是用户定义的变量。例如：

```
invariant gl_Position;

invariant centroid varying vec3 Color;
```

读者可能还记得，varying变量用于把顶点着色器的数据传递给片断着色器。不变性变量必须在顶点和片断着色器中都声明为invariant。可以在着色器中使用变量之前的任何时候都能对它应用invariant关键字，并可以用它修改以前声明的变量。

为了便于调试，可以对着色器中的所有varying变量施加不变性限制。这可以通过使用顶点着色器预处理pragma指令来实现。

```
#pragma STDGL invariant(all)
```

按照这种方式使用的全局不变性就可以用于调试。但是，它可能会对着色器的性能产生影响。保证不变性通常会禁用GLSL编译器所执行的那些优化。

15.5.4 语句

着色器的真正工作是通过对值进行计算以及做出决策来完成的。和C++一样，GLSL提供了一组丰富的操作符，可以用于创建各种算术操作来计算各种值。另外，GLSL还包括了一组标准的逻辑结构，用于控制着色器的执行。

算术操作

如果没有一张表来描述各种操作符的优先级，对语言的文本描述总显得不够完整。在表15-7中，操作符的顺序是按降序排列的。一般而言，操作数的类型必须是相同的。另外，对于向量和矩阵，操作数必须具有相同的维数。

表15-7 GLSL的操作符以及它们的优先级

1	()	—	对操作进行聚组
2	[]	数组	数组下标
	f()	函数	函数调用和构造器
	.(点号)	结构	结构字段或方法访问
	++ --	int、float、vec*、mat*	后缀的自增或自减
3	++ --	int、float、vec*、mat*	前缀的自增或自减
	+ - !	int、float、vec*、mat*	单目操作符：显示式的正值或负值，逐位反转，求反
4	* /	int、float、vec*、mat*	乘除操作
5	+ -	int、float、vec*、mat*	加减操作
6	< > <= >=	int、float、vec*、mat*	关系操作

（续）

7	== !=	int、float、vec*、mat*	相等测试操作
8	&&	bool	逻辑与操作
9	^^	bool	逻辑异或操作
10	||	bool	逻辑或操作
11	a?b:c	bool int、float、vec*、mat*	条件操作（内置的if操作，如果(a)，则(b)，否则(c)）
12	=	int、float、vec*、mat*	赋值
	+= - =		算术赋值
	*= /=		
13	,（逗号）	—	操作序列

注意：这张表列出了GLSL当前所实现的所有操作符。C++的有些操作符（例如求模操作符%）当前被保留，但尚未在GLSL中实现。

操作符重载

GLSL中的绝大多数操作符都进行了重载。也就是说，它们可以对各种类型的数据进行操作。尤其是用于向量和矩阵操作的算术操作符（包括前缀和后缀形式的自增和自减操作），在GLSL中都经过了精心的定义。例如，为了把一个向量与一个矩阵相乘（注意，这种操作的顺序是非常重要的，因为矩阵乘法并不满足交换律），可以使用下面的操作：

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

在这类操作中，正常的限制仍然存在，例如矩阵和向量的维度必须匹配。另外，标量值与向量或矩阵的乘法可以产生预期的结果。有一个重要的例外是两个向量的乘法将进行逐个成分的相乘。但是，两个矩阵相乘所产生的结果和常规的矩阵相乘一样。

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

GLSL还通过函数调用来支持其他一些常见的向量操作（例如数量积和向量积），以及在向量和矩阵上进行各种成分操作。

逻辑操作

GLSL仅有的逻辑控制结构就是if-then-else语句。和C语言一样，else子句是可选的，多条语句则需要包括在代码块中。

```
if ( truth ) {
    // true clause
} else {
    // false clause
}
```

和C语言中的情况类似，switch语句也可以以类似的形式使用（从GLSL 1.30开始）：

```
switch( int_value ) {
  case n:
        // statements
```

```
        break;

    case m:
      // statements
      break;

    default:
       // statements
      break;
}
```

GLSL switch也支持“贯穿”情况，即一条case语句末尾不以一条break语句结束。但对于语句块（结束花括号之前）的最后一个case，确实需要一条break语句。

循环结构

GLSL支持我们在C语言中所熟悉的for、while和do…while循环。

for循环允许在初始化子句中声明循环迭代变量。按照这种方式声明的迭代变量的作用域仅限于这个循环内部。

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

while (n < 10) {
    ...
}

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

流控制语句

在GLSL中，除了条件和循环之外，还存在其他控制语句。表15-8描述了可用的流控制语句。

表15-8 GLSL流控制语句

语　句	描　述
break	终止循环块的执行，并接着执行循环块之后的代码
continue	终止当前的那次循环，然后继续执行下一次循环
return [result]	从当前的子程序（函数）返回，可以同时返回一个值（假定返回值与调用程序的返回类型匹配）
discard	丢弃当前的片断并终止着色器执行。discard语句只在片断着色器程序中有效

discard语句只能用于片断着色器程序中。片断着色器程序的执行可能会在执行discard语句时终止，而具体的行为因实现而异。

15.5.5 函数

函数允许使用一个函数调用来代替一段经常出现的代码，从而实现更短小精悍的代码，并减少了出错的机会。GLSL定义了一些内置的函数，它们在附录I（可通过http://www.opengl-redbook.com/appendices/访问该附录的在线版）中列出。此外，GLSL也支持用户定义的函数。用户定义的函数可以在单个着色器对象内部定义，并在多个着色器程序中使用。

声明

函数声明的语法与C语言非常相似，唯一的例外是变量的访问限定符：

```
returnType functionName([accessModifier] type1 variable1,
                         [accessModifier] type2 varaible2,
                         ... )
{
    // function body
    return returnValue; // unless returnType is void
}
```

函数名可以是字母、数字和下划线的任意组合，但是它不能以数字或gl_开头。

返回类型可以是任何内置的GLSL类型以及用户定义的结构。返回值不能是数组。如果函数并不返回值，它的返回类型便是void。

函数的参数可以是任何类型，包括数组（必须指定长度）。

函数在使用之前必须进行声明，或者提供原型。和C++一样，编译器在使用函数之前必须看到它的定义，否则就会产生错误。如果一个函数是在一个着色器对象中使用，但是这个着色器对象并不是定义这个函数的着色器对象，那么它就必须声明这个函数的原型。原型仅仅是函数的签名，不包括它的函数体。下面是一个简单的例子：

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

参数访问限定符

虽然GLSL的函数可以修改数据并在执行完成后返回数据，但是它并不存在C和C++中指针或引用的概念。反之，函数的参数具有相关的访问限定符，表示这个值是否应该复制到函数中以及在函数完成执行后是否可以提取这个值。表15-9表示了GLSL所存在的各种参数访问限定符。

表15-9 GLSL函数的参数访问限定符

访问限定符	描　述
in	值复制到函数中（如果未指定限定符，此为默认）
const in	只读的值，复制到函数中
out	值从函数中复制出来（在传递给函数之前未初始化）
inout	值复制到函数中，并从函数中复制出来

in关键字是可选的。如果变量没有包含访问限定符，那么，参数声明中隐式地添加了一个“in”限定符。然而，如果变量的值需要从一个函数中复制出来，它必须或者标记为一个“out”（用于只写的变量），或者一个“inout”（用于可读写的变量）。向不带这些限定符之一的一个变量写入，将会产生一个编译时错误。

此外，要在编译时验证一个函数不会修改一个仅输入的变量，添加一个“const in”修饰词将会引发编译器检查该变量在函数中没有被写入过。

15.5.6 在GLSL程序中使用OpenGL状态值

使用OpenGL API可以设置的所有值几乎都可以在顶点和片断着色器中访问。附录I提供了GLSL内置变量的完整列表。

15.6 在着色器中访问纹理图像

GLSL支持在顶点和片断着色器使用纹理图像。OpenGL实现可以选择是否在顶点着色器中使用纹理图像（只要底层的OpenGL实现支持这个功能），但是它必须支持在片断着色器中使用纹理图像。

为了使用纹理图像，GLSL把一个在OpenGL应用程序中配置的活动纹理单元（参见第9.8节中的“多重纹理的步骤”）与着色器中的一个变量相关联。这个变量使用表15-10所显示的采样器数据类型之一，它允许着色器程序访问纹理图像的数据。相关的纹理图像的维数必须与采样器的类型相匹配。

表15-10 片断着色器纹理采样器类型

采样器名称	描 述	采样器名称	描 述
sampler1D isampler1D usample1D	访问1D纹理图像	sampler2DArray isampler2DArray usampler2DArray	访问2D纹理图像的一个数组
sampler2D isampler2D usampler2D	访问2D纹理图像	sampler2DRect isampler2DRect usampler2DRect	访问一个2D纹理矩形
sampler3D isampler3D usampler3D	访问3D纹理图像	sampler1DShadow	访问1D阴影纹理
samplerCube isamplerCube usamplerCube	访问立方体纹理图像 （对于反射贴图）	sampler2DShadow	访问2D阴影纹理
		samplerCubeShadow	访问立方体阴影纹理
		sampler1DArrayShadow	访问1D阴影纹理的一个数组
sampler1DArray isampler1DArray usampler1DArray	访问1D纹理图像的一个数组	sampler2DArrayShadow	访问2D阴影纹理的一个数组
		sampler2DRectShadow	访问2D阴影纹理矩形
		samplerBuffer isamplerBuffer usamplerBuffer	访问纹理缓冲区

采样器必须在着色器中声明为uniform变量，并且它们的赋值必须来自OpenGL应用程序。采样器也可以作为函数的参数，但必须是类型匹配的采样器。

采样器在着色器中使用之前必须分配一个纹理单位，并且只能通过glUniform1i()、glUniform1iv()进行初始化（把采样器应该使用的纹理单位的索引作为参数），如示例程序15-7所示。

示例程序15-7 与采样器变量相关的纹理单位

```
GLint texSampler;/*sampler index for shader variable "tex " */

texSampler =glGetUniformLocation(program,"tex ");
glUniform1i(texSampler,2);/*Set "tex " to use GL__TEXTURE2 */
```

在GLSL中使用纹理

当在GLSL着色器内部对一幅纹理图像进行采样时，需要使用已经声明并与一个纹理单位相关联的采样器变量。可以使用一些与纹理相关的函数（在1.2.7中描述）来访问OpenGL所支持的所有纹理图像类型。

在示例程序15-8中，我们对一个与Sampler 2D变量tex相关联的二维纹理图像进行采样，并把采样结果与片断的颜色进行组合，提供与在纹理环境模式下使用GL_MODULATE模式相同的结果。

示例程序15-8 在一个GLSL着色器内部对纹理进行采样

```
uniform sampler2D tex;

void main()
{
```

```
    gl_FragColor =gl_Color *texture2D(tex,gl_TexCoord [0].st);
}
```

即使纹理看上去很简单，实际上需要做的事情仍然很多。但是，在许多有趣的应用程序中，纹理图像中的数据并不一定用来表示颜色，应用程序可以把纹理图像的读取结果用于后续的计算中，尤其是当应用程序把一幅纹理图像的值作为索引来访问另一幅纹理图像的时候（参见稍后的“依赖性纹理读取”）。同样的结果也可以使用组合器纹理环境来实现（参见第9.9节），但是用着色器来实现要简单的多。

对纹理进行采样之后所进行的计算是由在着色器中编写的代码控制的，但是纹理图像的采样方法仍然是由应用程序控制的。例如，可以控制一幅纹理图像是否包含mipmap层、这些mipmap层是如何进行采样的，以及用于解析返回纹理单元值（基本上是glTexParameter*()函数所设置的参数）的过滤器。在着色器内部，可以控制mipmap选择的偏移值，并使用投影纹理技巧（参见附录I对相关函数的描述）。

依赖性纹理读取

在一个使用纹理贴图的着色器的执行过程中，需要使用纹理坐标在纹理图像中指定位置，并提取相应的纹理单元的值。纹理坐标是为每个活动纹理单位提供的，可以使用你所希望的任何值（前提是维数要匹配）来表示纹理坐标（包括刚刚从其他纹理图像采集的数据），供采样器使用。把一个纹理读取的结果作为索引值来读取另一个纹理的操作通常称为依赖性纹理读取，意即第二个操作的结果依赖于第一个操作。

在GLSL中，这种操作的实现是相当容易的，如示例程序15-9所示。

示例程序15-9 在GLSL中实现依赖性纹理读取

```
uniform sampler1D coords;
uniform sampler3D volume;

void main()
{
    vec3 texCoords =texture1D(coords,gl_TexCoord [0].s);
    vec3 volumeColor =texture3D(volume,texCoords);
    ...
}
```

纹理缓冲区



高级话题

尽管GLSL使得数组可用，不论是在着色器中用作静态初始化的值，还是作为值的集合呈现为uniform变量中的一个数组，在这两种情况下，我们都可能偶尔需要超出可用的大小限制的数组。在OpenGL 3.1之前，我们可能把这样的一个值的表存储在一个纹理图像中，然后，在纹理中操作纹理坐标来访问想要的值，这通常不是太直截了当的方式。对于这一数据存储问题，另一个更加直接的解决方案是纹理缓冲区。纹理缓冲区是缓冲区对象的一种特定类型，类似于一维纹理，可以在着色器中使用一个整数值来索引（就像一个常规的数组索引一样），但是，它提供了较为昂贵的纹理内存的资源，因此也支持较大的数据集合。我们像创建任何其他的缓冲区对象那样，来创建一个纹理缓冲区。首先调用glBufferData()（例如）初始化其数据。要把该缓冲区绑定到一个纹理缓冲区，调用glTexBuffer()。

---

void **glTexBuffer**(GLenum *target*, GLenum *internalFormat*, GLuint *buffer*);

把缓冲区对象*buffer*和*target*关联起来，这导致*buffer*中的数据的格式被解释为拥有*internaFormat*的格式。*target*必须是GL_TEXTURE_BUFFER，*internalFormat*可能是任何已知大小的纹理格式：G、

GL_R8、GL_R16、GL_R16F、GL_R32F、GL_R8I、GL_R16I、GL_R32I、GL_R8UI、GL_R16UI、GL_R32UI、GL_RG8、GL_RG16、GL_RG16F、GL_RG32F、GL_RG8I、GL_RG16I、GL_RG32I、GL_RG8UI、GL_RG16UI、GL_RG32UI、GL_RGBA8、GL_RGBA16、GL_RGBA16F、GL_RGBA32F、GL_RGBA8I、GL_RGBA16I、GL_RGBA32I、GL_RGBA8UI、GL_RGBA16UI和GL_RGBA32UI。

类似于其他的纹理图像，通过调用glActiveTexture()来指定哪个纹理单位和纹理缓冲区相关联。

15.7 着色器预处理器

编译GLSL着色器的第一个步骤是由预处理器进行解析。与C语言的预处理器相似，GLSL提供了一些指令，用于创建条件编译代码块以及定义一些值。但是，和C语言的预处理器不同，GLSL并没有提供文件包含指令。

15.7.1 预处理器指令

GLSL预处理器所接受的预处理器指令以及它们的功能见表15-11。

表15-11 GLSL预处理器指令

预处理器指令	描　述
#define #undef	控制常量和宏的定义，与C语言预处理器相似
#if #ifdef #ifndef	条件代码管理，与C语言预处理器相似，包括defined操作符
#else #elif #endif	条件表达式，只对整型表达式和已定义的值（例如由#define指定）进行求值
#error text	使编译器在着色器信息日志中插入text（直到第一个换行符）
#pragma options	控制编译器的特定选项
#extension options	根据指定的GLSL扩展，指定编译器操作
#version number	强制要求支持一个特定版本的GLSL版本
#line options	控制诊断行的编号

15.7.2 宏定义

GLSL预处理器允许使用宏定义，它与C语言的预处理器所采用的方式极为相似，只是它并不提供字符串替换和连接工具。宏可以定义单个值，如下所示：

```
#define NUM_ELEMENTS 10
```

或者像下面这样使用参数：

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

另外，GLSL还提供了几个预定义的宏，用于协助诊断信息（例如通过#error指令所产生的错误信息），见表15-12。

表15-12 GLSL预处理器的预处理宏

宏	定 义
__LINE__	由#line指令处理和修改的换行符的数量所定义的行号
__FILE__	当前被处理的源文件的字符串编号
__VERSION__ OpenGL	着色语言版本的整数表示形式

与此类似，可以使用#undef指令取消宏（不包括GLSL定义的宏）的定义。例如：

```
#undef LPos
```

15.7.3 预处理器条件

与C语言的预处理器的处理相同，GLSL预处理器提供了根据宏定义和整型常量求值结果的条件代码包含。

宏定义可以按照两种方式来确定。第一种方法是使用#ifdef指令：

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

另一种方法是在#if或#elif指令中使用defined操作符：

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

15.7.4 编译器控制

#pragma指令向编译器提供了关于希望如何对着色器进行编译的额外信息。

优化编译器选项

optinize选项指示编译器在着色器源文件中这条指令开始的位置开始启用或禁用着色器的优化。下面这两条命令分别用于启用和禁用优化。

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

和

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

这些选项只能在函数定义的外面使用。在默认情况下，所有着色器的优化选项都是启用的。

调试编译器选项

debug选项启用或禁用着色器的额外诊断输出。可以使用下面这两条指令分别启用或禁用调试：

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

或

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

与optimize选项相似，这些选项只能在函数定义的外面使用。在默认情况下，所有着色器的调试选项都是禁用的。

全局着色器编译选项

最后还有一条#pragma指令是STDGL。这个选项当前用于启用varying值输出的不变性。关于这方面的详细信息，请参阅第15.4.3节。

15.8 扩展处理

和OpenGL一样，GLSL可以通过扩展进行增强。由于开发商可能会包含与它们的OpenGL实现特定的扩展，因此根据着色器可能使用的扩展，对着色器的编译施加一个控制是非常有用的。

GLSL预处理器使用#extension指令向着色器编译器提供指令，说明在编译器应该如何处理可用的扩展。对于任何（或所有）扩展，可以指定编译器在编译时如何对它们进行处理。

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

其中，extensions_name与调用glGetString(GL_EXTENSION)所返回的扩展名称相同。可以使用下面这条指令，它将影响所有扩展的行为。

```
#extension all : <directive>
```

CLSL扩展指令所有可用的选项见表15-13。

表15-13 GLSL扩展指令限定符

指 令	描 述
require	如果不支持这个扩展或者使用了all扩展规范，则会产生一个错误
enable	如果指定的特定扩展不受支持，就会产生一个警告。如果使用了all扩展规范，则产生一个错误
warn	如果指定的特定扩展不受支持，就会产生一个警告。如果在编译时检测到使用了任何扩展，就会产生一个警告
disable	禁用对所列出的特定扩展的支持（也就是说，即使这些扩展实际是支持的，编译器就当它们是不受支持的）。如果使用了all，则禁用所有扩展。如果扩展并不存在，则产生警告和错误

15.9 顶点着色器的细节

可以使用如下几种机制把数据从应用程序发送到顶点着色器程序：

- 通过使用标准的OpenGL顶点数据接口（这些函数在glBegin()和glEnd()之间进行调用才是合法的），例如glVertex*()、glNormal*()等。这些值因顶点而异，被认为是内置的属性值。
- 通过声明uniform变量。这些值在几何图形中保持常量性质。
- 通过声明attribute变量，除了标准的顶点状态外，它们还可以根据顶点进行更新。（只有在OpenGL 3.1中指定顶点属性的时候，这种方法才有效，除非可以使用GL_ARB_compatibility扩展。

类似地，顶点着色器程序必须输出一些数据（还可以更新这些变量），以便OpenGL管线执行剩余的顶点处理操作。另外，顶点着色器程序常常和片断着色器程序一起执行。

需要写入的输出包括：

- gl_position，它必须由顶点着色器程序进行更新，并包括顶点在进行模型视图变换和投影变换之后的齐次坐标。
- 其他各个声明为varying的内置变量，用于向片断管线传递数据。这些数据包括颜色、纹理坐标和其他基于片断的数据。它们将在稍后的“varying输出变量”中描述。
- 用户定义的varying变量。

图15-5显示了一个顶点着色器程序的输入和输出。

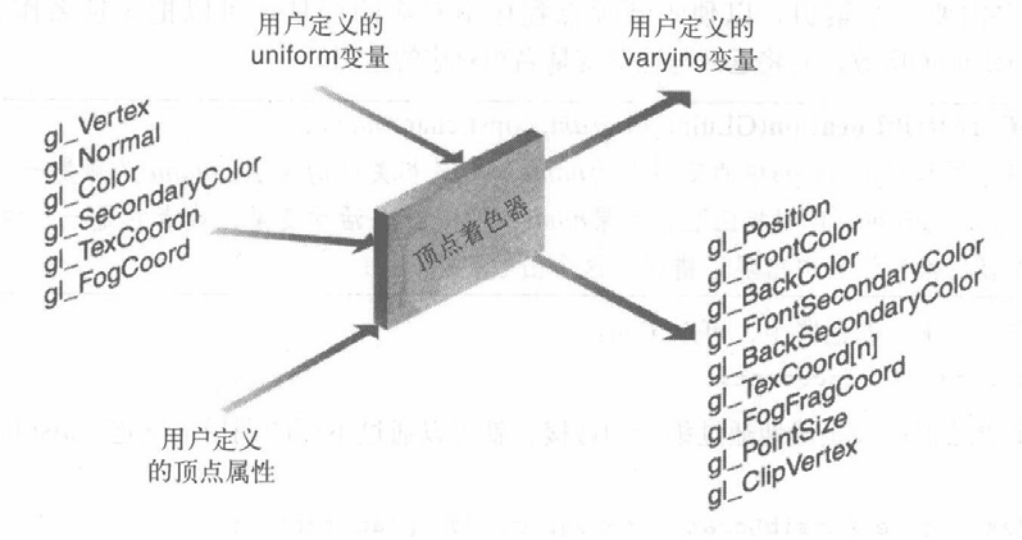


图15-5 GLSL顶点着色器输入和输出变量

内置的属性输入变量

在顶点着色器中全局可用的变量见表15-14。这些变量反映了当前的OpenGL状态（由对应的函数设置）。

表15-14 顶点着色器属性全局变量

变 量	类 型	指定函数	描 述
gl_Vertex	vec4	glVertex	顶点的全局空间坐标
gl_Color	vec4	glColor	主颜色值
gl_SecondaryColor	vec4	glSecondaryColor	辅助颜色值
gl_Normal	vec4	glNormal	光照法线
gl_MultiTexCoordn	vec4	glMultiTexCoord(n, ...);	纹理单位n的纹理坐标，n 的取值范围是0~7
gl_FogCoord	float	glFogCoord	雾坐标
gl_VertexID	int	-	自上次渲染调用开始，当前顶点的索引
gl_InstanceID	int	glDrawArraysInstanced glDrawElementsInstanced	相关图元的实例ID

用户定义的属性变量

用户定义的属性变量都是全局变量，与OpenGL应用程序传递给在OpenGL实现内部执行的顶点着色器的值相关联。

属性变量可以定义为float、float类型的向量（vec*）或矩阵（mat*）。

提示：一般情况下，属性变量在OpenGL内部是以vec4的形式实现的。如果想把一组单精度浮点值作为顶点属性使用，可以考虑把它们组合成一个或多个vec*结构。虽然把属性变量声明为单个float值也是可以的，但是OpenGL将使用一个完整的vec4来表示这个值。

为了使用用户定义的属性变量，OpenGL需要知道如何把在着色器程序中指定的变量名与传递给着色器的值进行匹配。类似于OpenGL处理uniform变量的方式，当一个着色器程序被链接时，链接器会生成一个变量名表格，用于表示属性变量。为了确定用户定义的顶点属性的最大数量，可以用

GL_MAX_VERTEX_ATTRIBS为参数调用glGetIntegerv()函数。

为了确定需要哪个索引，以便更新顶点程序中对应的变量，可以把变量名作为参数调用glGetAttribLocation()函数，它将返回与这个变量名相对应的索引。

GLint **glGetAttribLocation**(GLuint *program*, const char **name*);

返回与着色器程序*program*中的变量名为*name*的变量相关联的索引。*name*必须是一个以NULL结尾的字符串，与*program*的声明相匹配。如果*name*并不是一个活动变量，或者它是一个内置的属性变量的名称，或者在操作过程中出现了错误，这个函数将返回-1。

例如，在一个顶点着色器中，可以声明：

```
varying vec3 displacement;
```

假设这个着色器能够正确地通过编译和链接，就可以通过下面的调用来确定“displacement”变量的索引：

```
int index = glGetAttribLocation(program, "displacement");
```

可以调用gluBindAttribLocation()函数显式地设置一个属性变量与一个索引的绑定。但是，这个操作必须在着色器程序被链接之前执行。

---

void **glBindAttribLocation**(GLint *program*, GLuint *index*,
const char **name*);

显式地指定当*program*下一次被链接时，哪个*index*应该与*name*相关联。*index*必须是一个0到GL_MAX_VERTEX_ATTRIBS-1的整数，*name*必须是一个以NULL结尾的字符串。如果*name*是一个内置的属性变量（以gl_开头），这个函数将会产生一个GL_INVALID_ OPERATION错误。

为了设置与返回的索引相关联的值，可以使用某个版本的glVertexAttrib*()函数。

---

void **glVertexAttrib**{1234}{sfd}(GLuint *index*, *TYPE values*);

void **glVertexAttrib**{123}{sfd}v(GLuint *index*, const *TYPE *values*);

void **glVertexAttrib4**{bsifd ub us ui}v(GLuint *index*, *TYPE values*);

void **glVertexAttrib4Nub**(GLuint *index*, *TYPE values*);

void **glVertexAttrib4N**{bsi ub us ui}v(GLuint *index*, const *TYPE *values*);

void **glVertexAttribI**{1234}{i ui}(GLuint *index*, *TYPE values*);

void **glVertexAttribI4**{bsi ub us ui}v(GLint *index*, const *TYPE *values*);

把与*index*相关联的顶点属性变量的值设置为values。如果没有显式地设置所有4个值，y和z坐标将使用默认值0.0，w坐标将使用默认值1.0。

指定*index* 0的值相当于以相同的值调用glVertex*()。

这个函数的规范化版本将根据表4-1指定的映射把整型输入值转换为0~1的范围之内。

矩阵是通过为*index*指定连续的值来更新的。指定的*values*将用于更新矩阵的各个列。

虽然属性变量都是浮点值，但是OpenGL并没有限制对它们进行初始化的输入值的类型。特别是整型输入值，它们可以在赋值给属性变量之前通过glVertexAttrib4*N()函数调整在0~1的规范范围之内。

标量（单值）、向量和矩阵属性值可以用glVertexAttrib*()进行设置。如果是矩阵，则需要多次调用这个函数。具体地说，对于一个维度为n的矩阵，需要分别以索引index、index+1、……、index+

n-1调用glVertexAttrib*()函数。

顶点着色器还可以利用顶点数组这个工具（参见第2.6节）。和其他类型的顶点数据一样，顶点属性变量的值也可以存储在顶点数组中，并通过glDrawArrays()、glArrayElement()等函数进行更新。为了指定用于更新变量值的数组，可以调用glVertexAttribPointer()函数。

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

指定*index*的数据值可以在什么地方访问。*pointer*是一个指针，指向数组的第一组值的内存地址。*size*表示需要为每个顶点所更新的成分的数量。*type*指定了数组中每个元素的数据类型（GL_SHORT、GL_INT、GL_FLOAT或GL_DOUBLE）。*normalized*表示顶点数据在存储之前是否应该进行规范化（与**glVertexAttrib4N***()函数所使用的方式相同）。*stride*是数组中相邻元素之间的字节偏移量。如果*stride*是0，数据在数组中就是紧密相邻的。

和其他类型的顶点数组一样，指定数组只是整个操作的一部分。每个客户端的数组都需要启用。与使用glEnableClientState()函数相比，顶点属性数组是通过调用glEnableVertexAttribArray()函数启用的。

void **glEnableVertexAttribArray**(GLuint *index*);

void **glDisableVertexAttribArray**(GLuint *index*);

启用或禁用与*index*相关联的顶点数组。*index*必须是0和GL_MAX_VERTEX_ATTRIBS -1之间的一个值。

特殊的输出变量

表15-15所显示的值可以写入到顶点着色器中（并在写入后读取）。gl_position指定了顶点着色器在退出执行之后顶点的最终位置。这个变量必须写入到着色器中。

表15-15 顶点着色器特殊全局变量

变量名	类型	描述
gl_Position	vec4	经过变换的顶点位置（在视觉坐标中）
gl_PointSize	float	顶点的点大小
gl_ClipVertex	vec4	在用户定义的裁剪平面中使用的顶点位置。这个值必须与裁剪平面使用相同的坐标系统：视觉坐标或物体坐标

虽然可以把输出顶点的位置设置为希望的任何齐次坐标值，但是gl_position的最终值在顶点着色器程序中通常是用下面这种方法计算的：

```
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
```

如果使用了一种多道渲染算法（根据相同的几何图形渲染多幅图像），它需要让顶点着色器的结果与固定功能的管线保持一致，可以使用下面的代码设置gl_position：

```
gl_Position = ftransform();
```

gl_PointSize用于控制点的输出大小，与glPointSize()函数类似，只是它的操作是基于顶点进行的。为了在顶点着色器程序内部控制点的大小，可以用1个GL_VERTEX_ PROGRAM_POINT_SIZE值为参数调用glEnable()函数，它将覆盖当前可能已指定的任何点大小值。

用户定义的裁剪平面（例如通过glClipPlane()函数指定）可以通过编写一个齐次坐标写入到gl_ClipVertex变量中。为了正确地处理裁剪，被指定的裁剪平面以及写入到gl_ClipVertex的坐标值必须位于相同的坐标空间中。普通的裁剪空间是在视觉坐标中。可以把当前的顶点变换到视觉坐标中，以便执行裁剪：

```
gl_ClipVertex = gl_ModelViewMatrix * gl_Vertex;
```

varying输出变量

表15-16显示了可以写入到顶点着色器中并且它们的值可以在片断着色器中读取的变量。片断着色器中的值是根据图元的片断（如果使用的是多重采样，则为样本）进行迭代的。

表15-16 顶点着色器varying全局变量

变量名	类型	描述
gl_FrontColor	vec4	用于图元正面的主颜色
gl_BackColor	vec4	用于图元背面的主颜色
gl_FrontSecondaryColor	vec4	用于图元正面的辅助颜色
gl_BackSecondaryColor	vec4	用于图元背面的辅助颜色
gl_TexCoord[*n*]	vec4	第n个纹理坐标值
gl_FogFragCoord	vec4	片断雾坐标值

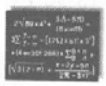
顶点着色器能够为顶点设置正面和背面颜色值。在默认情况下，不管顶点着色器内部怎么设置这些值，实际选择的将是正面颜色。这个行为可以通过调用glEnable()函数（以1个GL_VERTEX_PROGRAM_TWO_SIDE值为参数），它使OpenGL根据底层图元的方向来选择颜色。

顶点着色器中的纹理贴图

可以在顶点着色器中使用纹理贴图，但在写作本书时，这个特性尚不是必需的。然而，在顶点着色器中使用纹理与第15.6节中的“在GLSL中访问纹理”所描述的过程基本相同，仅有一处例外。顶点着色器不能实现自动mipmap选择。但是，可以使用GLSL的texture*Lod函数手工选择mipmap层。

为了确定使用的OpenGL实现是否支持在顶点着色器中使用纹理贴图，可以用glIntegerv()函数查询GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS。如果这个函数返回一个非零值，那么使用的OpenGL实现就支持在顶点着色器中使用纹理贴图。

15.10 变换反馈



高级话题

变换反馈允许一个应用程序把使用一个顶点着色器的渲染所产生的变换的图元记录到一个缓冲区对象中，这多少和第13.2节所描述的有些类似，但是，对数据的记录有更加灵活的控制。

使用变换反馈有两个步骤：

1) 指定一个顶点着色器的输出映射到一个或多个缓冲区对象。

2) 在变换反馈模式下渲染。

要完成步骤1)，在连接顶点着色器之前调用glTransformFeedbackVaryings()。这个函数将设置想要捕获的变化的输出顺序，并指定如何写出数据。

void **glTransformFeedbackVaryings**(GLuint *program*, GLsizei *count*,
const char ***varyings*, GLenum *bufferMode*);

为*program*指定计算数组*varyings*中的名字（以Null结尾）所指定的*varying*变量的顺序。*bufferMode*必须是GL_SEPARATE_ATTRIBS（指定了这些变化应该写入到与缓冲对象分隔开的count），或者GL_INTERLEAVED_ATTRIBS（把计数的变化值连续地写入到单个缓冲对象）。

如果*count*超出了可用作输出的*varying*变量的数目，*program*将会连接失败。如果*program*不是一个有效的程序对象，或者如果*bufferMode*设置为GL_SEPARATE_ATTRIBS，并且*count*大于一个GL_MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS查询所返回的值，将会产生一个GL_INVALID_VALUE。

一旦程序成功地连接了，并且设置了varying变量的顺序，可以附加缓冲区对象以接受变换的输出。要绑定一个缓冲区，使用glBindBufferRange()（或者glBindBufferBase()）；如果bufferMode设置为GL_SEPARATE_ATTRIBS，关联相应数目的缓冲区；如果bufferMode设置为GL_INTERLEAVED_ATTRIBS，关联到一个单个的缓冲区。缓冲区的大小（例如，由glBindBufferRange()指定，或者如果缓冲区大小固定，通过调用glBufferData()指定）表示为特定的图元记录的属性值的数目。当可用缓冲区空间耗尽，不再记录更多的图元。

表15-17 变换反馈图元以及它们允许的OpenGL渲染类型

变换反馈图元类型	允许的OpenGL 图元类型
GL_POINTS	GL_POINTS
GL_LINES	GL_LINES
	GL_LINE_LOOP
	GL_LINE_STRIP
GL_TRIANGLES	GL_TRIANGLES
	GL_TRIANGLE_FAN
	GL_TRAINGLE_STRIP
	GL_QUADS
	GL_QUAD_STRIP
	GL_POLYGONS

要捕获变化输出，通过调用glBeginTransformFeedback()进入变换反馈模式，并且指定想要记录的图元的类型，见表15-17。指定的varying变量的完整集合写入到每个顶点（直到达到了缓冲区中的空间限制）。

然后发布渲染命令。着色器变换顶点，然后在步骤1)中指定的varying变量被记录了下来。

void **glBeginTransformFeedback**(GLenum *primitiveMode*);

void **glEndTransformFeedback**(void);

进入并退出变换反馈模式。*primitiveMode*必须是GL_POINTS、GL_LINES或GL_TRIANGLES之一，它们表示写入到相关联的缓冲区对象的输出类型。

如果一条命令所渲染的OpenGL图元类型与当前变换反馈模式不兼容，将产生一个GL_INVALID_OPERATION错误。

示例程序15-10说明了使用变换反馈捕获顶点值、表面法线和纹理坐标的过程。注意，输入只包括顶点位置，其他的所有值都是顶点着色器生成的。

示例程序15-10 使用变换反馈捕获几何图元：xfb.c

```
GLuint
LoadTransformFeedbackShader(const char*vShader,GLsizei count,
   const GLchar**varyings)
{
   GLuint shader,program;
```

```
   GLint completed;
program =glCreateProgram();

/*
**---Load and compile the vertex shader ---
*/

if (vShader !=NULL){
   shader =glCreateShader(GL_VERTEX_SHADER);
   glShaderSource(shader,1,&vShader,NULL);
   glCompileShader(shader);
   glGetShaderiv(shader,GL_COMPILE_STATUS,&completed);

   if (!completed){
      GLint len;
      char*msg;

      glGetShaderiv(shader,GL_INFO_LOG_LENGTH,&len);
      msg =(char*)malloc(len);
      glGetShaderInfoLog(shader,len,&len,msg);
      fprintf(stderr,"Vertex shader compilation "
      "failure:\n%s \n",msg);
      free(msg);

      glDeleteProgram(program);

      exit(EXIT_FAILURE);
   }

   glAttachShader(program,shader);
}

glTransformFeedbackVaryings(program,count,varyings,
   GL_INTERLEAVED_ATTRIBS);

/*
**---Link program ---
*/
glLinkProgram(program);
glGetProgramiv(program,GL_LINK_STATUS,&completed);

if (!completed){
   GLint len;
   char*msg;

   glGetProgramiv(program,GL_INFO_LOG_LENGTH,&len);
   msg =(char*)malloc(len);
   glGetProgramInfoLog(program,len,&len,msg);
      fprintf(stderr,"Program link failure:\n%s \n",msg);
      free(msg);

      glDeleteProgram(program);
```

```
        exit(EXIT_FAILURE);

    }

    return program;
}

void
init()
{
    /*Vertex shader generating our output values */
    const char vShader [] ={
    "#version 140 \n""in vec2 coords;"
    "out vec2 texCoords;"
    "out vec3 normal;"
    "void main(){"
    "    float angle =radians(coords [0]);"
    "    normal =vec3(cos(angle),sin(angle),0.0);"
    "    texCoords =normal.xy;"
    "    gl_Position =vec4(normal.xy,coords [1],1.0);"
    "}"
    };

    /*List (and ordering)of varying values written to the
    **transform buffer */
    const char *varyings [] ={
       "texCoords",
       "normal",
       "gl_Position"
    };

    GLuint program;
    GLuint query;
    GLint count;

    /*Load shader program and set up varyings */
    program =LoadTransformFeedbackShader(vShader,3,
       varyings);

    glUseProgram(program);

    glGenQueries(1,&query);

    /*Bind to transform-feedback buffer */

    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER,0,xfb);

    glBeginQuery(GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN,

       query);
    glBeginTransformFeedback(GL_POINTS);
    glDrawArrays(GL_POINTS,0,2*(NumSlices+1));
```

```
    glEndTransformFeedback();
    glEndQuery(GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN);

    glGetQueryObjectiv(query,GL_QUERY_RESULT,&count);

    fprintf(stderr,"%d primitives written \n",count);
    ...
}
```

15.11 片断着色器

和顶点着色器程序一样，OpenGL应用程序也可以直接向片断着色器程序发送数据，并且让OpenGL在程序内部提供这些值。所有这些数据，包括输入到片断着色器的数据，用于产生片断的最终颜色和深度值。在执行片断着色器之后，经过计算所产生的值继续经过OpenGL的片断管线，进入片断测试和混合阶段。

图15-6描述了片断着色器的输入和输出。

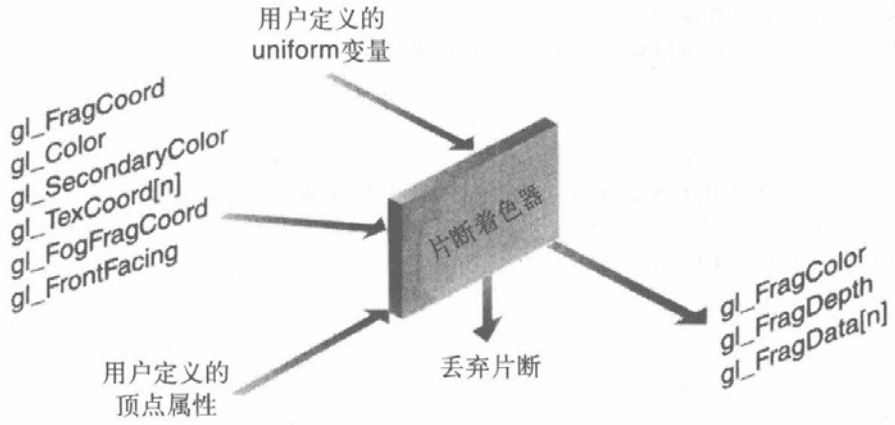


图15-6 片断着色器的内置变量

输入值

片断着色器接收顶点管线最终输出的迭代值。这些值包括片断的位置、已解析的主颜色和辅助颜色、一系列的纹理坐标以及片断的雾坐标距离。所有这些值见表15-18。

表15-18 片断着色器的varying全局变量

变 量	类 型	描 述
gl_FragCoord	vec4	（只读）片断的位置，包括z成分，它表示固定功能所计算的深度值
glFrontFacing	bool	（只读）指定了这个片断是否属于一个正面图元
gl_Color	vec4	片断的主颜色
gl_SecondColor	vec4	片断的辅助颜色
gl_TexCoord[n]	vec4	片断的第n纹理坐标
gl_FogFragCoord	float	片断的雾坐标，要么指定为视觉空间中的图元的z坐标，或者插值雾坐标
gl_PointCoord	vec2	一个点块纹理的片断位置在[0.0, 1.0]的范围中。如果当前图元并不是点块纹理或点块纹理被禁用，这个值就是未定义的

特殊的输出值

在片断着色器程序中，输入值经过组合，产生片断的最终值，见表15-19。

gl_Fragcolor是片断的最终颜色。虽然gl_FragColor并不一定要从片断着色器程序输出，但是如果把最终的片断颜色写入到颜色缓冲区，其结果是未定义的。

gl_FragDepth是片断的最终深度，可以在深度测试中使用。虽然无法修改片断的（x，y）坐标，但它的深度值却是可以修改的。

表15-19 片断着色器的输出全局变量

变 量	类 型	描 述
gl_FragColor	vec4	片断的最终颜色
gl_FragDepth	float	片断的最终深度
gl_FragData[*n*]	vec4	写入到第n个数据缓冲区的数据值

最后，片断着色器程序还可以写入其他数据。gl_FragData数组允许把各种数据写入到额外的缓冲区中，如下面的“渲染到多个输出缓冲区”所述。

渲染到多个输出缓冲区

片断着色器可以使用gl_FragData数组，把值同时输出到多个缓冲区。在数组元素gl_FragData[n]中写入一个值将导致这个颜色被写入到缓冲区中一个适当的片断中，这个片断位于传递给glDrawBuffers()函数的数组的第n个元素中。

片断着色器可以把值写入到gl_FragColor或gl_FragData中，但是不能同时写入到两者中。

用户定义的片段着色器输出

读者可能会发现，gl_FragColor这样相对缺乏描述性的名字，不太适合用来指定着色器的输出。尽管GLSL不会在意，但是下一个阅读你的代码（而你可能忘了写代码注释）的程序员可能无法立即理解你的着色器不是输出一个颜色，而是输出某种贝塞尔函数的求值结果（或者某种其他的数学计算）。

有了GLSL 1.30，我们可以给片段着色器的输出起更合适的名字，并且可以控制这些输出到各种绑定的绘制缓冲区的映射。可以在连接之前指定片段输出名到一个缓冲区的映射，也可以在连接之后询问程序来确定变量的布局。

要使用第一种方法，在调用glLinkProgram()之前调用glBindFragDataLocation()。

void **glBindFragDataLocation**(GLuint *program*, GLuint *colorNumber*,
const char **name*);

指定片段变量名的输出应该针对GLSL *program*写到颜色缓冲区*colorNumber*。这只对还没有连接的程序有效。

如果*colorNumber*小于0或者大于查询GL_MAX_DRAW_BUFFERS时返回的值，将会产生一个GL_INVALID_VALUE错误。如果*name*以gl_开头，将会产生一个GL_INVALID_OPERATION错误。

连接之后，可以通过调用glGetFragDataLocation()来获取映射：

GLint **glGetFragDataLocation**(GLuint *program*, const char **name*);

返回和来自GLSL *program*的*name*输出相关的颜色缓冲区索引。如果*program*没有成功地连接，并且调用了**glGetFragDataLocation()**，将产生一个GL_INVALID_OPERATION错误。如果*name*不是和*program*相关的一个片段程序输出，或者发生了其他的错误，返回一个-1。