OpenGL® 开放式图形系统规范：

规范（2.0版 - 2004年10月22日）

马克·西格尔 柯特·阿克利

编辑（1.1版）：克里斯·弗雷泽编辑（1.2-1.5版）：乔恩·利奇

编辑（2.0版）：乔恩·利奇与帕特·布朗

目录

图示列表

表格列表

第一章

引言

本文档描述了OpenGL图形系统：其定义、运作方式及实现要求。我们假设读者至少具备计算机图形学的基础知识，包括熟悉计算机图形算法的核心内容，以及了解基本图形硬件和相关术语。

1.1 可选功能的格式化

从OpenGL 1.2版本开始，规范中的某些特性被视为可选；OpenGL实现可选择提供或不提供这些特性（参见第3.6.2节）。

规范中可选的部分在首次定义可选功能时会特别标注（参见第3.6.2节）。可选的状态表条目以灰色背景呈现。

1.2 什么是OpenGL图形系统?

OpenGL（全称"开放图形库"）是连接图形硬件的软件接口。该接口包含数百个程序和函数，使程序员能够指定生成高质量图形图像所需的对象与操作，特别是三维物体的彩色图像。

OpenGL的大部分功能要求图形硬件包含帧缓冲区。许多OpenGL调用涉及绘制点、线、多边形和位图等对象，但某些绘制操作的实现方式（例如抗锯齿处理时）

或纹理处理时）依赖于帧缓冲区。此外，OpenGL的部分功能专门用于帧缓冲区的操作。

1.3 程序员视角下的OpenGL

对程序员而言，OpenGL是一组命令集，可用于定义二维或三维几何对象，并控制这些对象如何渲染到帧缓冲区中。OpenGL主要提供即时模式接口，这意味着定义对象后会立即进行绘制。

一个典型的使用OpenGL的程序首先调用函数打开一个窗口，该窗口将作为程序绘图的帧缓冲区。随后调用函数分配一个GL上下文并将其关联到该窗口。一旦分配了GL上下文，程序员即可自由发出OpenGL命令。 部分调用用于绘制简单几何对象（如点、线段和多边形），另一些则影响这些基本图形元素的渲染效果——包括光照与着色方式，以及它们从用户二维/三维模型空间映射到二维屏幕的过程。此外还存在直接操控帧缓冲区的调用，例如像素读写操作。

1.4 实现者视角下的OpenGL

对实现者而言，OpenGL是一组影响图形硬件运行的指令集。若硬件仅包含可寻址帧缓冲区，则OpenGL必须几乎完全在主机CPU上实现。 更常见的情况是，图形硬件可能包含不同程度的图形加速功能——从能渲染二维线条和多边形的栅格化子系统，到能对几何数据进行变换和计算的复杂浮点处理器。 OpenGL实现者的任务是提供CPU软件接口，同时将每个OpenGL命令的工作在CPU和图形硬件之间进行分配。这种分配必须根据可用的图形硬件进行定制，以在执行OpenGL调用时获得最佳性能。

OpenGL维护着大量状态信息。这些状态控制着对象如何被绘制到帧缓冲区中。其中部分状态可直接供用户访问：用户可通过调用获取其值。 然而另一些状态仅能通过其对绘制效果的影响来体现。本规范的核心目标之一，正是将OpenGL状态信息显式化——阐明其变化机制，并明确其具体作用。

1.5 我们的观点

我们将OpenGL视为控制特定绘图操作集的状态机。该模型应能产生满足程序员和实现者需求的规范。但它未必提供具体的实现模型。实现必须产生符合规范方法的结果，但执行特定计算时可能存在比规范方法更高效的途径。

1.6 配套文档

本规范应与题为*《OpenGL着色语言》*的配套文档一并阅读。 后者（下称《OpenGL着色语言规范》）定义了用于编写顶点着色器和片段着色器的编程语言的语法与语义（参见第2.15节和第3.11节）。这些章节可能引用配套文档中定义的概念和术语（如着色语言变量类型）。

OpenGL 2.0 实现保证至少支持着色语言 1.10 版；实际支持的版本可参照第 6.1.11 节所述方法进行查询。

第2章

OpenGL 操作

2.1 OpenGL 基础知识

OpenGL（以下简称“GL”）仅负责向帧缓冲区进行渲染（以及读取帧缓冲区中存储的值）。它不支持图形硬件有时关联的其他外围设备，如鼠标和键盘。程序员必须依靠其他机制获取用户输入。

GL通过多种可选模式绘制*基本图形单元*。每个单元可为点、线段、多边形或像素矩形。各模式可独立切换，设置某一模式不会影响其他模式（尽管多种模式可能相互作用以决定最终渲染到帧缓冲区的内容）。通过函数或过程调用的形式发送*命令*，即可设置模式、指定图形单元并执行其他GL操作。

基元由一个或多个*顶点*组成的集合定义。顶点定义了一个点、一条边的端点，或两条边相交的多边形角点。 数据（包含位置坐标、颜色、法线及纹理坐标）与顶点相关联，每个顶点均按顺序独立处理且采用相同方式。唯一例外是当顶点组需进行*裁剪*以使原始图形适配指定区域时，此时可修改顶点数据并创建新顶点。裁剪类型取决于顶点组所代表的原始图形类型。

命令始终按接收顺序处理，尽管命令效果显现前可能存在不确定延迟。这意味着例如：当前基元必须完全绘制完毕，后续基元才能影响帧缓冲区。同时意味着查询和像素读取操作返回的状态，均与先前所有GL命令完整执行后的状态一致（除非另有明确规定）。

通常，GL命令对GL模式或帧缓冲区的影响必须完全完成后，后续命令才能产生类似效果。

在图形语言中，数据绑定发生在调用时。这意味着传递给命令的数据在接收该命令时被解析。即使命令需要指向数据的指针，这些数据也在调用时被解析，后续对数据的任何更改都不会影响图形语言（除非后续命令使用了相同的指针）。

图形渲染管线（GL）直接控制着三维与二维图形的基础操作，包括变换矩阵、光照方程系数、抗锯齿方法及像素更新运算符等参数的设定。它并不提供描述或建模复杂几何对象的手段。 另一种表述方式是：GL提供的是描述复杂几何对象渲染方式的机制，而非描述复杂对象本身的机制。

GL命令的解释模型采用客户端-服务器模式。即程序（客户端）发出命令，这些命令由GL（服务器端）进行解释和处理。服务器端可能与客户端位于同一台计算机上，也可能位于不同计算机上。从这个意义上说，GL具有“网络透明性”。 服务器可维护多个*GL*上下文，每个上下文封装当前GL状态。客户端可选择*连接*至任意上下文。若程序未*连接*至*上下文*时发出GL命令，将导致行为未定义。

GL命令对帧缓冲区的影响最终由分配帧缓冲区资源的窗口系统控制。正是窗口系统决定GL在任意时刻可访问帧缓冲区的哪些部分，并向GL传达这些部分的结构。因此，GL中不存在用于配置帧缓冲区或初始化GL的命令。 同样地，GL也不涉及在CRT显示器上呈现帧缓冲区内容（包括通过伽马校正等技术对单个帧缓冲区值进行转换）。帧缓冲区配置在GL外部与窗口系统协同完成；当窗口系统为GL渲染分配窗口时，GL上下文即完成初始化。

GL的设计旨在支持不同图形能力和性能的多种图形平台。为适应这种多样性，我们针对某些GL操作规定了理想行为而非实际行为。 在允许偏离理想行为的情况下，我们同时规定了实现必须遵循的规则，以确保其能有效近似理想行为。这种允许的GL行为差异意味着：即使在完全相同的帧缓冲配置下运行，两个不同的GL实现面对相同输入时也可能无法实现像素级一致的输出。

最后，为避免与其他软件包的名称冲突，GL中的命令名、常量和类型均添加了前缀（在C语言中分别**以gl**、GL和GL开头）。为清晰起见，本文档省略了这些前缀。

2.1.1 浮点运算

图形渲染器在运行过程中需执行大量浮点运算。我们不规定浮点数的具体表示方式或运算实现方法，仅要求数值的浮点部分包含足够位数，且指数字段足够大，以确保每次浮点运算的结果精度达到$1^{0^{5分}}$之一。 用于表示位置、法线或纹理坐标的浮点数最大可表示量必须至少为$^{2^{32}}$；颜色值的最大可表示量必须至少为2^{10}。所有其他浮点值的最大可表示量必须至少为

2^{32} . $x \quad 0 = 0 \quad x = 0$ 对于任何非无限且非NaN*的*x。 $1 \quad x = x \quad 1 = x$。 . .

$x + 0 = 0 + x = x$。$0^0 = 1$。（偶尔会规定其他要求。）大多数单精度浮点格式都满足这些要求。

任何可表示的浮点数值均可作为需要浮点数据的GL命令的合法输入。向此类命令提供非浮点数值的结果未作规定，但不得导致GL中断或终止。例如在IEEE算术中，向GL命令提供负零或非正规化数将产生可预测结果，而提供NaN或无穷大则产生未规定结果。

某些计算涉及除法运算。在此类情况下（包括向量归一化所需的隐含除法），除以零将产生未指定结果，但不得导致GL中断或终止。

2.2 GL状态

图形渲染管线（GL）维护着大量状态信息。本文档将逐一列举每个状态变量，并说明如何修改这些变量。为便于讨论，状态变量根据其功能进行了某种程度的任意分类。尽管我们描述了GL对帧缓冲区执行的操作，但帧缓冲区本身并不属于GL状态的一部分。

我们区分两种状态类型。第一类称为*GL服务器状态*，存在于GL服务器端，绝大多数GL状态属于此类。第二类称为*GL客户端状态*，存在于GL客户端。除非另有说明，本文档提及的所有状态均为GL服务器状态；GL

客户端状态则会特别标识。每个GL上下文实例对应一整套GL服务器状态；客户端与服务器间的每次连接则同时涉及GL客户端状态与GL服务器状态。

虽然图形语言（GL）的具体实现可能依赖于硬件，但本讨论独立于GL所运行的特定硬件平台。因此，我们仅在图形硬件状态与GL状态完全对应时才关注前者。

2.3 GL命令语法

GL命令是函数或过程。不同命令组执行相同操作，但参数传递方式各异。为便捷处理这种差异，我们采用特定记法描述命令及其参数。

GL命令由*名称*及后续最多4个字符组成，具体取决于命令类型。首字符表示该命令必须接收的指定类型参数数量。第二字符或字符对表示参数的具体类型：8位整数、16位整数、32位整数、单精度浮点数或双精度浮点数。 最后一个字符（若存在）为 `v`，表示该命令接受指向数组（向量）的指针，而非一系列单独参数。以下两个具体示例取自 **Vertex** 命令：

```
void Vertex3f(float x, float y, float z);
```

以及

```
void Vertex2sv( short v[2] );
```

这些示例展示了这些命令的ANSI C声明。通常，命令声明采用以下形式[1]

rtype **Name ϵ1234** { } **ϵ b s i f d ub us ui** { } **ϵv**
(*[参数列表,] T 参数1, ..., T 参数N [, 参数列表]*) ;

rtype 是函数的返回类型。括号 { } 包含一系列字符（或字符对），其中选择其中一个。ϵ 表示无字符。方括号内括起的参数（*[args ,]* 和 *[, args]*）可有可无。

[1] 本文档所示声明适用于ANSI C。C++和Ada等允许传递参数类型信息的语言支持更简洁的声明和更少的入口点。

字母	对应GL类型
b	字节
s	短
i	`int`
f	浮点
d	`double`
ub	`ubyte`
us	`ushort`
ui	`uint`

表 2.1： 命令后缀字母与 GL 参数类型的对应关系。GL 类型的定义请参见表 2.2。

N 个参数 *arg1* 至 *argN* 的类型为 *T*，该类型对应表 2.1 中列出的类型字母或字母组合之一（若无字母，则参数类型显式给出）。若末尾字符非 **v**，则 *N* 由数字 **1**、**2**、**3** 或 **4** 表示（若无数字，则参数数量固定）。 若末尾字符为**v**，则仅存在arg1且其为指定类型的N值数组。最后，通过在类型名前缀`u`表示无符号类型（例如`unsigned char`简写为`uchar`）。

例如，

`void` **Normal3{fd}**(`T` *arg*); 表示以下两个声明

`void` **Normal3f**(`float` *arg1*, `float` *arg2*, `float` *arg3*); `void` **Normal3d**(`double` *arg1*, `double` *arg2*, `double` *arg3*);

而

`void` **Normal3{fd}v**(`T` *arg*);

则表示以下两个声明

`void` **Normal3fv**(`float` *arg[3]*); `void` **Normal3dv**(`double` *arg[3]*);

类型固定的参数（即命令中未通过后缀标明的参数）属于14种类型之一（或指向其中一种的指针）。这些类型汇总于表2.2中。

GL 类型	最小值 位宽	描述
布尔型	1	布尔
字节	8	带符号的2的补码二进制整数
ubyte	8	无符号二进制整数
char	8	字符串的组成字符
短整型	16	带符号的2的补码二进制整数
ushort	16	无符号二进制整数
int	32	有符号的2的补码二进制整数
uint	32	无符号二进制整数
sizei	32	非负二进制整数大小
枚举	32	枚举二进制整数值
intptr	*ptrbits*	带符号的2的补码二进制整数
sizeiptr	*ptrbits*	非负二进制整数大小
bitfield	32	位域
浮点	32	浮点数值
clampf	32	浮点值被限制在[0, 1]范围内
double	64	浮点数
clampd	64	浮点值，取值范围限制为[0, 1]

表 2.2： GL 数据类型。GL 类型并非 C 类型。因此，例如 GL 类型 int 在本文档之外被称为 GLint，且不一定等同于 C 类型的 int。实现可能使用比表中指示的位数更多的位来表示 GL 类型。然而，对于超出最小范围的整数值，无需进行正确解释。

ptrbits 是表示指针类型所需的位数；换言之，intptr 和 sizeiptr 类型必须足够大，以存储任意地址。

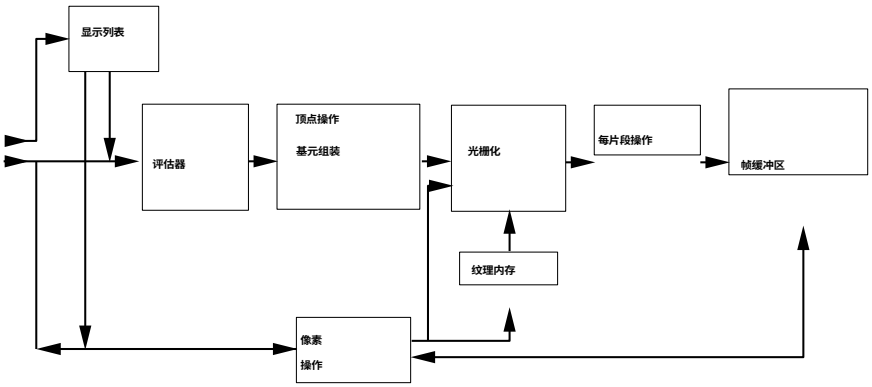


图2.1. GL的模块化结构图。

2.4 基础GL操作

图2.1展示了图形处理库（GL）的示意图。命令从左侧进入GL。部分命令指定待绘制的几何对象，其余命令则控制对象在各处理阶段的处理方式。多数命令可累积至*显示列表中*，供GL后续处理。否则，命令将通过处理管道进行实时传输。

第一阶段通过评估输入值的多项式函数，为近似曲线和曲面几何体提供高效手段。后续阶段则处理由顶点描述的几何基元：点、线段和多边形。 本阶段对顶点进行变换与光照处理，并将基元裁剪至视体范围，为后续光栅化阶段做准备。光栅化器通过二维点、线段或多边形描述生成系列帧缓冲区地址与数值。 每个生成的*片段*被传递至下一阶段，该阶段对单个片段执行操作后最终更新帧缓冲区。这些操作包括：基于输入深度值与预存深度值的条件更新（实现深度缓冲）、输入片段颜色与存储颜色的混合渲染，以及片段值的遮罩处理与逻辑运算。

最后，存在一种方法可绕过管道中的顶点处理环节，将片元块直接发送至各个片元操作单元，最终促使像素块写入帧缓冲区；同时也可从帧缓冲区读取值

或在帧缓冲区不同区域间进行复制。这些传输可能包含某种解码或编码操作。

此排序仅作为描述图形处理器（GL）的工具，并非其实现的严格规则，我们仅将其作为组织GL各项操作的手段。例如曲面等对象在转换为多边形前可能已进行变换。

2.5 GL错误

图形语言仅检测部分可视为错误的条件。这是因为在许多情况下，错误检查会对无错误程序的性能产生负面影响。

命令

```
enum GetError( void );
```

用于获取错误信息。每种可检测错误均对应唯一数值代码。错误发生时将设置标志位并记录代码。 后续发生的错误不会影响该记录代码。调用 **GetError** 时，该代码将被返回且标志位被清除，以便后续错误能重新记录其代码。若调用 **GetError** 返回 NO ERROR，则表示自上次调用 **GetError**（或自 GL 初始化）以来未检测到任何错误。

为支持分布式实现，可能存在多个标志-代码对。此时，**当GetError**调用返回非NO ERROR值后，后续每次调用将返回不同标志-代码对的非零代码（顺序未指定），直至所有非NO ERROR代码均被返回。 当不再存在非NO ERROR错误代码时，所有标志将被重置。该方案需要若干对标志位与整数的组合。所有标志的初始状态均为清零，所有代码的初始值均为NO ERROR。

表 2.3 总结了图形渲染错误。当前，当错误标志被设置时，仅当发生内存不足时图形操作的结果才未定义。在其他情况下，引发错误的命令会被忽略，因此不会影响图形状态或帧缓冲区内容。若该命令返回值，则返回零。 若生成错误的命令通过指针参数修改值，则这些值不会被更改。此错误语义仅适用于GL错误，不适用于内存访问错误等系统错误。此行为为当前规范；GL在错误状态下的处理机制可能变更。

每个GL命令的描述中都隐含着若干错误生成条件：

错误	描述	违规命令 被忽略?
无效枚举 _	枚举参数超出范围	是
无效值 _	数字参数超出范围	是
操作无效 _	当前状态下操作非法	是
栈溢出 _	该命令将导致堆栈 溢出	是
栈下溢 _	命令将导致堆栈 下溢	是
内存不足 _	内存不足，无法执行 命令	未知
表过大 _ _	指定的表过大	是

表 2.3： GL 错误汇总

- 若向需要枚举值的命令传递的符号常量不在该命令允许范围内，则会触发 INVALID ENUM 错误。即使参数是符号常量指针，若所指值不属于该命令允许范围，仍会发生此错误。
- 当指定大小为 sizei 的参数时若提供负数，将触发 INVALID VALUE 错误。
- 若命令执行导致内存耗尽，则可能触发 OUT OF MEMORY 错误。

否则，仅当本规范中明确描述的条件发生时才会生成错误。

2.6 开始/结束范式

在图形语言中，大多数几何对象通过**在**开始/结束对之间封装一系列坐标集来绘制，这些坐标集指定顶点，并可选地指定法线、纹理坐标和颜色。采用此方式绘制的几何对象共有十种：点、线段、线段环、分离线段、多边形、三角带、三角扇、分离三角形、四边形带和分离四边形。

每个顶点由两个、三个或四个坐标定义。此外，在处理每个顶点时，可能使用*当前法线*、多个*当前纹理坐标集*、多个*当前通用顶点属性*、*当前颜色*、*当前次要颜色*以及*当前雾坐标*。法线用于GL的照明计算；当前法线是一个三维向量，可通过发送三个定义它的坐标来设置。 纹理坐标决定纹理图像如何映射到基元上。 可通过多组纹理坐标指定多个纹理图像在基元上的映射方式。支持的纹理单元数量取决于具体实现，但至少为两个。可通过状态MAX TEXTURE UNITS查询支持的纹理单元数量。通用顶点属性可在顶点着色器（第2.15节）中访问，用于计算供后续处理阶段使用的值。

每个顶点都关联着主色和次色（参见第3.9节）。这些关联颜色要么基于当前颜色和当前次色，要么由光照生成——具体取决于光照功能是否启用。纹理坐标和雾坐标同样关联于每个顶点。单个顶点可关联多组纹理坐标。图2.2概括了辅助数据与变换后顶点的关联关系，最终生成*经过处理的 顶点*。

当前值属于GL状态的一部分。顶点和法线经过变换，颜色可能受光照影响或被替换，纹理坐标经过变换且可能受纹理坐标生成函数影响。针对每个当前值指定的处理流程，都会应用于发送至GL的每个顶点。

顶点、法线、纹理坐标、雾坐标、通用属性及颜色的传输机制，法线变换方式以及顶点在二维屏幕上的映射方法，将在后续章节中详细阐述。

在为顶点分配颜色之前，顶点所需的状态包括：顶点坐标、当前法线、当前边标记（参见第2.6.2节）、当前材质属性（参见第2.14.2节）、当前雾坐标、多个通用顶点属性集以及多个当前纹理坐标集。 由于颜色分配是按顶点逐个进行的，因此处理后的顶点包含该顶点的坐标、其边标记、其雾坐标、其分配的颜色以及其多个纹理坐标集。

图2.3展示了从顶点序列构建*基本图形*（点、线段或多边形）的操作流程。基本图形形成后，将被裁剪至视体。此过程可能通过改变顶点坐标、纹理坐标及颜色来修改基本图形。对于线段和多边形基本图形——

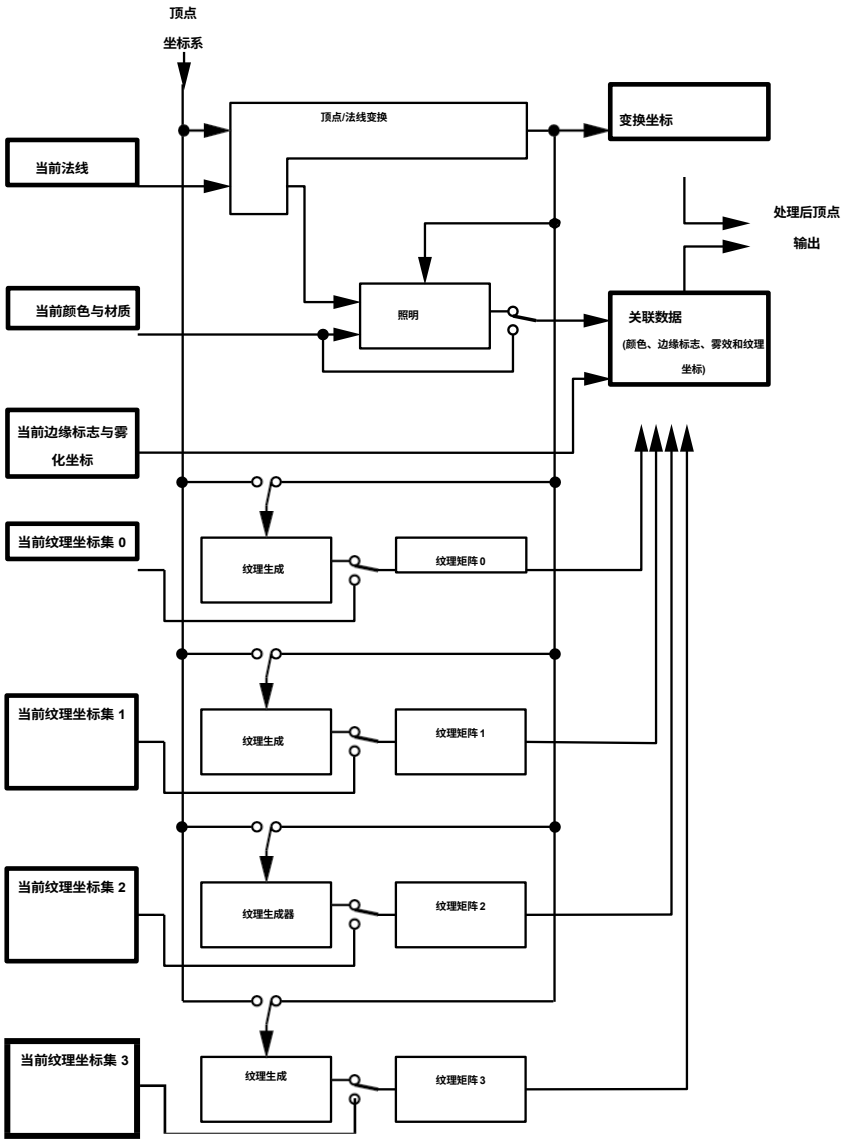


图2.2. 顶点与当前值的关联关系。粗线框代表OpenGL状态。图中展示了四个纹理单元；但多纹理渲染支持的单元数量可能因具体实现而异。

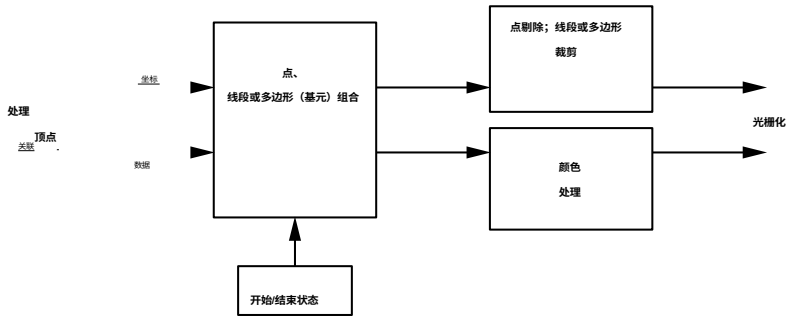


图2.3. 基本图形组装与处理。

在原始组件中，裁剪操作可能向原始组件插入新顶点。用于光栅化的原始组件定义顶点均关联有纹理坐标与颜色信息。

2.6.1 开始与结束

构成支持几何对象类型的顶点需通过两条命令定义：

```
void Begin( enum mode ); void End(
void );
```

在Begin

和End命令之间指定的顶点数量没有**限制**。

点。通过**调用**Begin并传入POINTS参数值，可指定一系列独立点。此情况下**Begin与End之间**无需保持特殊状态，因每个点均独立于前后点。

线段带。当Begin函数的参数为LINE STRIP时，通过在Begin/End对中包含两个或多个端点序列，可指定一条或多条相连线段。此时，第一个顶点标记首段起点，第二个顶点同时标记首段终点与次段起点。 一般而言，第i个顶点（*i*>1）同时定义第i条线段的起点与第i-1条线段的终点。末尾顶点则标记最后一条线段的终点。若Begin/End对间仅指定单个顶点，则不生成任何图形元素。

所需状态包含由上次发送的顶点生成的处理后顶点（以便据此生成连接至当前顶点的线段），以及一个布尔标志，用于指示当前顶点是否为首顶点。

线环。通过**向Begin传递**LINE LOOP参数值定义的线环，本质上与线带相同，区别在于会从最后指定顶点到首顶点添加最终线段。其附加状态包含已处理的首顶点。

独立线段。当**Begin**参数的值为LINES时，通过用**Begin和End**包围顶点对来生成独立线段，每个线段由一对顶点定义。此时，**Begin和End**对之间的前两个顶点定义第一个线段，后续每对顶点各定义一个新线段。若指定顶点数为奇数，则忽略最后一个顶点。 所需状态与线段相同，但使用方式不同：一个顶点存储当前线段的首顶点，一个布尔标志标记当前顶点是奇数还是偶数（即线段起始或终止点）。

多边形。多边形通过将其边界定义为一系列线段来描述。当以POLYGON参数调用Begin时，边界线段的指定方式与线环相同。根据图形库的当前状态，多边形可通过多种方式渲染，例如勾勒边界或填充内部区域。使用少于三个顶点描述的多边形不会生成原始图形。

仅凸多边形能确保由GL正确绘制。若指定多边形在投影至窗口时为非凸形状，则渲染的多边形只需位于定义其边界的投影顶点所构成的凸包内部即可。

支持多边形所需的状态至少包含两个已处理的顶点（永远不需要超过两个，尽管实现可能使用更多）；这是因为凸多边形可在顶点到达时进行光栅化，而无需等待所有顶点指定完毕。顶点顺序在光照计算和多边形光栅化中至关重要（参见第2.14.1节和第3.5.1节）。

三角带。三角带是由共享边连接的三角形序列。当**Begin**函数带TRIANGLE STRIP参数调用时，通过**在Begin**/End对之间指定定义顶点序列来定义三角带。此时前三个顶点定义首个三角形（其顺序与多边形相同，具有重要意义）。 后续每个顶点均通过自身与前三角形两个顶点共同定义新三角形。当**Begin接**收到三角带指令时，若**Begin**/End对包含少于三个顶点，则不生成原始图形。详见图2.4。

支持三角带所需的状态包含：一个标记首个三角形是否完成的标志位，以及两个已处理顶点存储器（称为顶点A

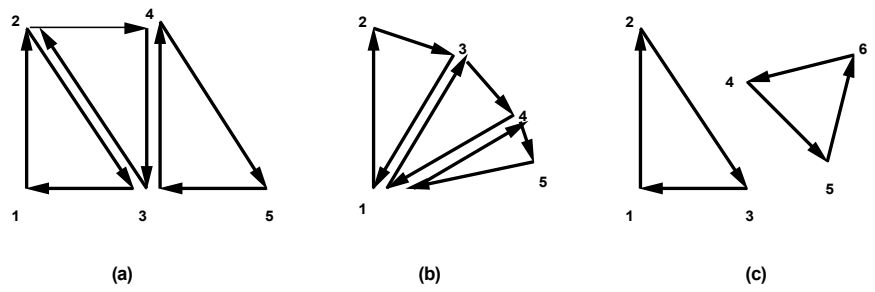


图2.4. (a) 三角带。 (b) 三角扇。 (c) 独立三角形。数字表示顶点在**Begin和End**之间的排序顺序。请注意：在(a)和(b)中，三角形边序由首个三角形决定；而在(c)中，每个三角形的边序与其他三角形互不相关。

以及顶点B），以及一个单比特指针，用于指示哪个存储顶点将被下一个顶点替换。**在Begin**(TRIANGLE STRIP)之后，该指针初始化为指向顶点A。Begin/End对之间发送的每个顶点都会翻转指针状态。因此，第一个顶点存储为顶点A，第二个存储为顶点B，第三个存储为顶点A，依此类推。 发送的第二个顶点之后，所有顶点均按顶点A、顶点B、当前顶点的顺序构成三角形。

三角扇形。三角扇形与三角带形相同，但有一个例外：除第一个顶点外，后续每个顶点都会替换两个存储顶点中的顶点B。**当Begin**参数的值为TRIANGLE FAN时，三角扇形的顶点将被包含在Begin**和End**之间。

分离三角形。当**Begin**参数的值为TRIANGLES时，通过**在Begin**与**End**之间放置顶点来指定分离三角形。 此时，第3i+1、第3i+2、第*3i*+3顶点（按此顺序）将为*每个i*=0,1,...n确定一个三角形 1，**且Begin与End**之间存在*3*n+*k*个顶点。*k*取值为0、1或2；若*k*不为零，则最后*k*个顶点将被忽略。 对于每个三角形，顶点A为顶点*3i*，顶点B为顶点3i+1。否则，独立三角形与三角带相同。

适用于多边形的规则同样适用于由三角带、三角扇或独立三角形生成的每个三角形。

四**边形（四边形）带。**当**开始点**为

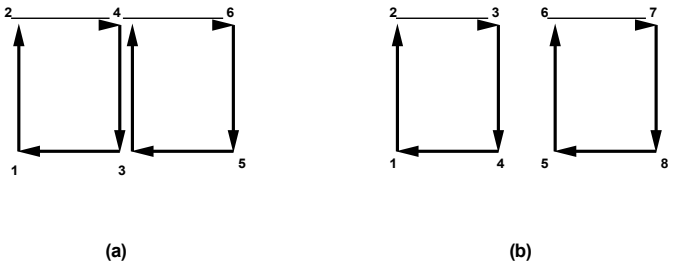


图2.5. (a) 四边形带状结构。(b) 独立四边形。编号表示顶点在**Begin**与**End**之间的序列顺序。

使用四边形带调用。若**Begin**与**End**之间的*m*个顶点为v_1、…、v_m，*其中*v_j为第j个指定顶点，则*第i个*四边形包含顶点（按顺序排列）：v_{2i}、v_{2i+1}、v_{2i+3}，以及v_{2i+2}，其中$i = 0, \ldots, m/2$。因此所需状态包含三个处理中的顶点（用于存储前一个四边形的最后两个顶点及当前四边形的第三个顶点，即第一个新顶点）、一个标记旗（用于指示首个四边形完成状态），以及一个单比特计数器（用于统计顶点对的成员数量）。参见图2.5。

若四边形带的顶点数少于四个，则不会生成任何基元。若**在Begin和End**之间指定的四边形带顶点数为奇数，则最后一个顶点将被忽略。

独立四边形独立四边形与四边形带相似，区别在于每组四个顶点——第4j+1个、第4j+2个、第4j+3个和第4j+4个顶点（j=0,1,…,n） 1——共同构成单个四边形。 **Begin**与**End**之间的顶点总数为$4n + k$，其中$0 \le k \le 3$；若*k*不为零，则最后*k*个顶点将被忽略。通过调用**Begin**并传入参数值QUADS即可生成独立四边形。

针对多边形制定的规则同样适用于四边形带中生成的每个四边形或单独四边形。

Begin和End所需的状态由一个11位整数组成，该整数指示十种可能**的Begin/End**模式之一，或表示当前未处理任何**Begin/End**模式。

2.6.2 多边形边

由多边形、三角带、三角扇、独立三角形集、四边形带或独立四边形集生成的每个基元，其每条边均被标记为边界边或*非边界边*。这些分类在多边形光栅化过程中被使用；某些模式会影响多边形边界边的解释（参见第3.5.4节）。 默认情况下所有边均为边界边，但可通过调用以下函数修改多边形、独立三角形或独立四边形的边标记：

```
void EdgeFlag( boolean flag ); void EdgeFlagv(
boolean *flag );
```

修改标志位的值。若标志为零，则标志位设为FALSE；若
*flag*非零，则标志位设为TRUE。

当**Begin**参数值为POLYGON、TRIANGLES或QUADS时，**Begin**与**End**对之间指定的每个顶点均作为边起点。若边标志位为TRUE，则每个指定顶点生成的边均标记为边界边；若标志位为FALSE，则生成的边均标记为非边界边。

用于边标记的状态由一个当前标记位组成。初始时该位为TRUE。此外，每个已处理的多边形基元顶点都必须附加一个位，用于指示从该顶点开始的边是否为边界边。

2.6.3 Begin/End 块内的 GL 命令

在任何Begin/End对之间允许使用的GL命令仅限于：指定顶点坐标、顶点颜色、法线坐标、纹理坐标、通用顶点属性及雾坐标的命令（**Vertex**、**Color**、**SecondaryColor**、**Index**、**Normal**、**TexCoord**和**MultiTexCoord**、**VertexAttrib**、**雾坐标**），**数组**元素命令（参见第2.8节），**坐标**评估与**点**评估命令（参见第5.1节），用于指定光照材质参数的命令（**材质**命令；参见第2.14.2节），显示列表调用命令（**调用**列表与**调用列表集**；参见第5.4节），以及**边界**标志命令。在**执行**Begin与**对应的End**之间执行任何其他GL命令，将导致INVALID OPERATION错误。若在已执行**Begin**但未执行**End**前再次执行**Begin**，或在未执行对应**Begin**的情况下执行**End**，均会引发INVALID OPERATION错误。

执行以下命令：**启用客户端状态**、**禁用客户端状态**、**推送客户端属性**、**弹出客户端属性**、**颜色指针**、**雾坐标指针**、**边界标志指针**

Pointer 、 **IndexPointer** 、 **NormalPointer** 、 **TexCoordPointer** 、 **SecondaryColorPointer** 、 **VertexPointer** 、 **VertexAttribPointer**、**ClientActiveTexture**、**InterleavedArrays** 和 **PixelStore** 命令的执行。若在任何 **Begin/End** 对之间执行此类命令，可能不会触发错误，也可能触发错误。 若未产生错误，则GL操作行为未定义。（相关命令详见第2.8节、3.6.1节及第6章说明。）

2.7 顶点定义

顶点通过提供其二维、三维或四维坐标进行定义。此操作需使用多种版本的Vertex命令之一：

```
void Vertex{234}{sifd}( T coords );void
Vertex{234}{sifd}v( T coords );
```

调用**任何**Vertex命令均需指定四个坐标：*x*、*y*、*z*和*w*。*其中x*坐标为首坐标，*y*为次坐标，*z*为第三坐标，*w*为第四坐标。 调用 **Vertex2** 设置 *x* 和 *y* 坐标；*z* 坐标默认设为零，*w* 坐标设为一。**Vertex3** 将 *x*、*y* 和 *z* 设为传入值，*w* 设为一。**Vertex4** 设置全部四个坐标，可指定射影三维空间中的任意点。在 **Begin/End** 对之外调用 **Vertex** 命令将导致未定义行为。

当前值用于将辅助数据关联至顶点，具体说明详见第2.6节。当前值可通过执行相应命令随时更改。相关命令如下：

```
void TexCoord{1234}{sifd}( T coords ); void
TexCoord{1234}{sifd}v( T coords );
```

指定当前的同质纹理坐标，命*名为*s、*t*、*r*和*q*。TexCo**ord**1系列命令将*s*坐标设置为提供的单个参数，同时将*t*和*r*设为0，*q*设为1。 类似地，**TexCoord2**将s和*t*设为指定值，*r*设为0，*q*设为1；**TexCoord3**设置*s*、t和*r*，*q*设为1；**TexCoord4**则设置全部四个纹理坐标。

实现必须支持至少两组纹理坐标。命令

```
void MultiTexCoord 1234 sifd (enum texture,T coords) void MultiTexCoord 1234
sifd v(enum texture,T        {       }{       }
    坐标)                    {       }{       }
```

将待修改的坐标集作为纹理参数。纹理参数采用 *TEXTUREi* 形式的符号常量，表示需修改*第 i 个*纹理坐标集。这些常量遵循 *TEXTUREi* = TEXTURE0 + *i* 的关系 （*i* 的取值范围为 0 到 k-1，其中 *k* 是由 MAX TEXTURE COORDS 定义、取决于具体实现的纹理坐标集数量）。

_ TexCoord 命令与对应的 **Multi-TexCoord** 命令完全等效，*其中纹理*设置为 TEXTURE0。

获取当前纹理坐标返回由活动纹理值定义的纹理坐标集。

若为 **Multi-TexCoord** 的*纹理*参数指定无效的纹理坐标集，将导致未定义行为。

当前法线通过以下方式设置：

```
void Normal3{bsifd}( T coords ); void
Normal3{bsifd}v( T coords );
```

传递**给法向量的**字节、短整型或整型值将按表2.9中对应（带符号）类型的说明转换为浮点值。

当前雾坐标通过以下方式设置：

```
void FogCoord{fd}( T coord );void FogCoord{fd}v( T
coord );
```

设置当前颜色和辅助颜色有多种方式。图形库同时存储当前单值*颜色索引*、当前四值RGBA颜色以及辅助颜色。根据图形库处于*颜色索引模式*还是*RGBA模式*，索引值或颜色及辅助颜色具有有效性。模式选择在图形库初始化时确定。

设置RGBA颜色的命令如下：

```
void Color{34}{bsifd ubusui}( T components );void Color{34}{bsifd ubusui}v( T
components );void SecondaryColor3{bsifd ubusui}( T components );
void 次要颜色3{bsifd ubusui}v( T components );
```

Color命令有两个主要变体：**Color3和Color4**。四值版本设置全部四个值。三值版本将R、G和B设置为给定值；A默认设为1.0。（整数颜色分量（R、G、B和A）转换为浮点值的机制详见第2.14节。）

次要颜色仅提供三值版本。次要A值始终固定为1.0。

接受浮点值的Color和SecondaryColor命令版本，其数值范围名义上介于0.0至1.0之间。0.0对应最小值，1.0对应帧缓冲区中该组件可能取到的最大值（取决于机器配置）（参见第2.14节关于颜色与着色的说明）。超出[0, 1]范围的值不会被限制。

命令

```
void Index{sifd ub}( T index ); void Index{sifd ub}v(
T index );
```

用于更新当前（单值）颜色索引。该命令接受一个参数，即当前颜色索引应设置的值。超出颜色索引（机器相关）可表示范围的值不会被限制。

顶点着色器（参见第2.15节）可编写为访问包含4个分量的通用顶点属性数组，该数组除常规属性外还包含此类属性。数组首个槽位编号为0，其大小由实现相关的常量MAX VERTEX ATTRIBS指定。

命令

```
void VertexAttrib{1234}{sfd}( uint index, T values );void VertexAttrib{123}{sfd}v( uint
index, T values );void VertexAttrib4{bsifd ubusui}v( uint index, T values );
```

可用于将给定值加载到*槽索引处的*通用属性中，该属性的分量*名为*x、*y*、*z和*w*。VertexAttrib1*系列命令*将*x坐标设置为提供的单个参数，同时*将*y和z设为0，w设为1。**类似**地，VertexAttrib2*命令*将*x*和*y*设为指定值，z设为0，w设为1；VertexAttrib3*命令设置*x*、y和*z*，并将w设为1；VertexAttrib4*命令则设置全部四个坐标。若*索引*值大于或等于MAX VERTEX ATTRIBS，将触发INVALID VALUE错误。

命令

```
void VertexAttrib4Nub( uint index, T values );
void VertexAttrib4N{bsi ubusui}v( uint index, T values );
```

还可指定采用固定点坐标的顶点属性，这些坐标根据表2.9缩放至归一化范围。

先前定义的**VertexAttrib***入口点也可用于加载顶点着色器中声明为2× ×2、3× ×3或4× ×4矩阵的属性。矩阵的每列占用通用4分量属性槽中的一个，该槽位于 ×

顶点属性可用槽位上限。 矩阵按列优先顺序加载至这些槽位。矩阵列需按槽位编号递增顺序加载。设置通用顶点属性零可指定顶点；该顶点的四个坐标值将取自属性零的数值。**Vertex2**、Vertex3或Vertex4命令与*索引*为零的对应VertexAttrib*命令完全等效。设置任何其他通用顶点属性将更新该属性的当前值。顶点属性零当前无有效值。值。顶点属性零不存在当前值。

通用属性与常规属性之间不存在别名冲突。换言之，应用程序可同时设置所有MAX VERTEX ATTRIBS通用属性及所有常规属性，无需担心某个特定属性会覆盖其他属性的值。

支持顶点指定所需的状态包含每组纹理坐标对应的四个浮点数，用于存储当前纹理*坐标*、三个浮点数用于存储当前法线的三个坐标，一个浮点数用于存储当前雾化坐标，四个浮点值用于存储当前RGBA颜色，四个浮点值用于存储当前RGBA次要颜色，一个浮点值用于存储当前颜色索引，以及MAX VERTEX ATTRIBS 1个四分量浮点向量用于存储通用顶点属性。

当前不存在顶点概念，因此不会为顶点坐标 或通用属性零分配状态。每个纹理坐标集的初始值均为$(s, t, r, q) = (0, 0, 0, 1)$。初始当前法线坐标为$(0, 0, 1)$。 初始雾度坐标为零。初始RGBA颜色为$(R, G, B, A) = (1, 1, 1, 1)$，初始RGBA辅助颜色为$(0, 0, 0, 1)$。初始颜色索引为1。所有通用顶点属性的初始值均为$(0, 0, 0, 1)$。

2.8 顶点数组

第2.7节所述的顶点规范命令几乎可接受任何格式的数据，但即使指定简单的几何体也需要执行大量命令。顶点数据也可存储于客户端地址空间中的数组中。 通过执行单条GL命令，即可利用这些数组中的数据块定义多个几何基元。客户端最多可配置七组数组（外加MAX TEXTURE COORDS与MAX VERTEX ATTRIBS数组）：分别用于存储顶点坐标、法线、颜色、次要颜色、颜色索引、边界标记、雾化坐标、两组及以上纹理坐标集，以及一个或多个通用顶点属性。相关命令如下：

void **VertexPointer**(int *size*, enum *type*, sizei *stride*, void *pointer);

void **普通指针**(enum *类型*, sizei *步长*, void *指针);

void **ColorPointer**(int *size*, enum *type*, sizei *stride*, void *pointer);

void **次要颜色指针**(int *尺寸*, enum *类型*, sizei *步长*, void *指针);

void **索引指针**(enum *类型*, sizei *步长*, void *指针);void **边标志指针**(sizei *步长*, void *指针);

void **雾坐标指针**(enum *类型*, sizei *步长*, void *指针);

void **TexCoordPointer**(int *size*, enum *type*, sizei *stride*, void *pointer);

void **VertexAttribPointer**(uint *index*, int *size*, enum *type*, boolean *normalized*, sizei *stride*, const void *指针);

描述这些数组的位置和组织结构。对于每个命令，*type* 参数指定数组中存储值的数据类型。由于边标志始终为布尔类型，**EdgeFlagPointer** 不包含*类型*参数。当存在 *size* 参数时，它表示数组中每个顶点存储的值的数量。由于法线始终由三个值指定，**NormalPointer** 不包含 *size* 参数。 同样地，由于颜色索引和边标志始终由单个值指定，**Index**Pointer 和**EdgeFlagPointer**同样不包含*size*参数。表2.4列出了*size*和*type*（当存在时）的允许取值范围。 对于*类型*参数，BYTE、SHORT、INT、FLOAT和DOUBLE分别表示字节型、短整型、整型、浮点型和双精度型；UNSIGNED BYTE、UNSIGNED SHORT和UNSIGNED INT则分别表示无符号字节型、无符号短整型和无符号整型。若size参数值超出表中范围，将触发INVALID VALUE错误。

VertexAttribPointer命令中的*index*参数用于标识所描述的通用顶点属性数组。若*index*值大于或等于MAX VERTEX ATTRIBS，则会触发INVALID VALUE错误。该命令中的*normalized*参数用于标识固定点类型是否_

命令	大小	标准化	类型
顶点指针	2,3,4	否	短整型、整型、浮点型、双精度型
NormalPointer	3	是	字节, 短整型, 整型, 浮点型, `double`
颜色指针	3,4	是	字节, 无符号字节, 短整型, `ushort`, `int`, `uint`, `float`, `double`
次要颜色指针	3	是	字节, `ubyte`, 短整型, `ushort`, `int`, `uint`, `float`, `double`
索引指针	1	否	`ubyte`, `short`, `int`, `float`, `double`
雾坐标指针	1	-	浮点数, 双精度浮点数
纹理坐标指针	1,2,3,4	否	短整型, 整型, 浮点型, 双精度浮点型
EdgeFlagPointer	1	否	布尔
顶点属性指针	1,2,3,4	*标志*	字节, 无符号字节, 短整型, `ushort`, `int`, `uint`, `float`, `double`

表 2.4： 顶点数组大小（每个顶点的值）和数据类型。"标准化"列指示固定点类型是否被直接接受，或被标准化为 [0, 1]（对于无符号类型）或 [-1, 1]（对于有符号类型）。对于通用顶点属性，固定点数据仅在 **VertexAttribPointer** *的标准化*标志被设置时才会被标准化。

转换为浮点数时应进行归一化处理。若归一化为TRUE，则按表2.9所示规则转换定点数据；否则直接转换定点值。

数组中对应单个顶点的1、2、3或4个值构成一个数组*元素*。每个数组元素内的值在内存中按顺序存储。若*步长*指定为零，则数组元素同样按顺序存储。若步长为负值，则触发INVALID VALUE错误。否则，数组第i个元素与第(i+1)个元素的指针地址间距*为步长*基本机器单位（通常为无符号字节），其中指向第(i+1)个元素的指针地址更大。对于每条命令，*指针*参数指定指定数组首个元素首个值在内存中的位置。

通过调用以下任一方法可启用或禁用单个数组：

```
void EnableClientState( enum array ); void DisableClientState(
enum array );
```

*其中array*可设置为VERTEX ARRAY、NORMAL ARRAY、COLOR ARRAY、SECONDARY COLOR ARRAY、INDEX ARRAY、EDGE_FLAG_ARRAY、雾坐标数组或纹理坐标数组，分别对应顶点、法线、颜色、次要颜色、颜色索引、边标记、雾坐标或纹理坐标数组。

通过调用以下任一函数可启用或禁用单个通用顶点属性数组：

```
void EnableVertexAttribArray( uint index );void
DisableVertexAttribArray( uint index );
```

其中 *index* 用于标识要启用或禁用的通用顶点属性数组。若 *index* 大于或等于 MAX VERTEX ATTRIBS，则生成错误 INVALID VALUE。

命令

```
void ClientActiveTexture( enum texture );
```

用于选择将由 TexCoordPointer 命令修改的顶点数组客户端状态参数，以及受 **EnableClientState** 和 **DisableClientState**命令（参数TEXTURE COORD ARRAY）影响的数组。此命令设置客户端状态变量CLIENT ACTIVE TEXTURE。每个纹理坐标集均拥有一个客户端状态向量，该向量在调用本命令时被选定。此状态向量包含顶点数组状态。 此调用同时选定用于查询客户端状态的纹理坐标集状态。

指定无效*纹理*将引发 INVALID ENUM 错误。*纹理*的有效值与第 2.7 节所述 **MultiTexCoord** 命令中的定义相同。

命令

```
void ArrayElement( int i );
```

将 将每个启用数组的第 i 个元素传输至图形渲染器。

ArrayElement(*i*) 的效果等同于命令序列

```
if （法线数组启用）
    Normal3[type]v(法线数组第i个元素); if （启用颜色数组）
    Color[size]|[type]v(颜色数组第i个元素); if （辅助颜色数组启
用）
    SecondaryColor3[type]v(次级颜色数组元素 i)； if （雾坐标数组启用）
    FogCoord[type]v(雾坐标数组元素 i)； for (j = 0; j <
textureUnits; j++)
    如果 （纹理坐标集 j 数组启用）                                    {
        MultiTexCoord[size]|[type]v(纹理坐标集 j 的纹理坐标数组元素 i 对应于纹理0 + j 位置); 若颜色索引数组启用
    Index[类型]v(颜色索引数组元素i); if （边缘标志数组启用）
    EdgeFlagv(边界标志数组元素i);
for (j = 1; j < 通用属性数组; j++) if （通用顶点属性数组启用）          {
                        如果 （通用顶点属性 j 数组归一化标志被设置，且{
            类型非浮点型或倍精度浮点型）
        VertexAttrib[size]N[type]v(j, 通用顶点属性j数组元素i); 否则
        VertexAttrib[size]|[type]v(j, 通用顶点属性j数组元素i);
    }
}
如果（通用属性数组 0 已启用）                                    {
    如果 （通用顶点属性 0 数组归一化标志被设置，且
        类型非浮点型或倍精度浮点型）
    VertexAttrib[size]N[type]v(0, 通用顶点属性0数组元素i); 否则
    VertexAttrib[size]|[type]v(0, 通用顶点属性0数组元素i);
```

```
}  else if (顶点数组启用)                    {
       顶点|大小||类型|v(顶点数组元素 i);
}
```

其中textureUnits和*generic*Attributes分别表示实现支持的纹理坐标集数量和通用顶点属性数量。"[size]"和"[type]"对应于相应数组的大小和类型。对于通用顶点属性，假定存在完整的顶点属性命令集，即使GL并未提供所有此类函数。

在 Begin 执行与对应的 **End** 执行之间对数组数据所做的更改，可能以非顺序的方式影响同一 **Begin/End** 周期内对 **ArrayElement** 的调用。也就是说，在数组数据更改之前对 **ArrayElement** 的调用可能访问到已更改的数据，而在数组数据更改之后的调用可能访问原始数据。

指定 $i < 0$ 将导致未定义行为。 此情况建议返回错误 INVALID VALUE。

命令

```
void DrawArrays( enum mode, int first, sizei count );
```

使用每个启用数组中*第 1* 到*第 1* + *count* 1 的元素构造一组几何基元序列。*mode* 参数指定构造的基元类型，其接受的标记值与 **Begin** 命令的 *mode* 参数相同。

```
DrawArrays（mode, first, count）;
```

与以下命令序列效果相同

```
if （mode 或 count 无效）
    生成相应错误
else     {
    Begin（mode）;
    for (int i = 0; i < count ; i++)
        数组元素（first+ i）;
    结束();
}
```

有一个例外：当前的法线坐标、颜色、次要颜色、颜色索引、边缘标志、雾坐标、纹理坐标和通用属性在**执行 DrawArrays后**均处于未确定状态，前提是对应数组处于

启用的数组，则其当前值将变为未定义。禁用数组的当前值不会因**DrawArrays**执行而改变。

指定first < 0将导致未定义行为。 在此情况下建议生成错误 INVALID VALUE。

命令

```
void MultiDrawArrays( enum mode, int *first, sizei *count, sizei
    primcount );
```

的行为与**DrawArrays**完全相同，区别在于指定了*primcount*个独立元素范围。其效果等同于：

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

该命令

```
void DrawElements( enum mode, sizei count, enum type, void *indices );
```

使用*索引数*中存储的索引构建由*计数*个元素组成的几何基元序列。*类型*必须为UNSIGNED BYTE、UNSIGNED SHORT或UNSIGNED INT之一，表明*索引数*中的值分别属于GL类型ubyte、ushort或uint的索引。*模式*指定构建的基元类型；其接受的标记值与Begin命令的*模式*参数相同。 效果

```
DrawElements（模式, 数量, 类型, 索引）;
```

与命令序列的效果相同

```
if （模式、计数或类型无效）
   生成相应的错误
else    {
   Begin（mode）;
   for (int i = 0; i < count ; i++)
      数组元素（索引数组[i]）;结束（）;
}
```

有一个例外：当前的法线坐标、颜色、次要颜色、颜色索引、边缘标志、雾坐标、纹理坐标和通用属性在**执行DrawElements**后均处于未确定状态，前提是对应数组处于启用状态。对于禁用的数组，其当前值不会因执行**DrawElements**而改变。

命令

```
void MultiDrawElements( enum mode, sizei *count, enum type, void **indices,
    sizei primcount );
```

的行为**与DrawElements**完全相同，区别在于指定了*primcount*个独立的元素列表。其效果等同于：

```
for (i = 0; i < primcount; i++) {
   if (count[i]) > 0)
                    DrawElements(模式, 计数[i], 类型, 索引[i]);
}
```

该命令

```
void DrawRangeElements( enum mode, uint start, uint end, sizei count,
    enum type, void *indices );
```

是 **DrawElements** 的受限形式。*mode*、*count*、*type* 和 *indices* 与 **DrawElements** 的对应参数一致，但额外限制数组 *indices* 中的所有值必须位于 *start* 和 *end* 之间（包含边界值）。

实现应标注顶点数据与索引数据的推荐最大值，可通过调用 GetIntegerv 并传入符号常量 MAX_ELEMENTS_VERTICES和MAX_ELEMENTS_INDICES获取。*若end* start+1为
若元素数量超过顶点最太元素数值，或*计数*超过索引最大元素数值，则调用可能导致性能下降。系统不强制要求必须引用[*起始, 结束*]区间内的所有顶点。但实现可能对未使用的顶点进行部分处理，从而降低性能表现，无法达到采用最优索引集时的效果。

若 *end* < *start*，则生成 INVALID VALUE 错误。*模式*、计数或类型参数无效时，将产生与调用 **DrawElements** 对应错误相同的报错。索引超出 [*start, end*] 范围属于错误，但实现方可能不进行检查。此类索引将导致实现方特定的行为。

命令

void **InterleavedArrays**(enum *format*, sizei *stride*, void *pointer);

高效初始化六个数组及其使能位，使其采用14种配置之一。 *格式*必须为14个符号常量之一： V2F, V3F, C4UBV2F, C4UBV3F, C3FV3F, N3FV3F, C4F N3F V3F, T2F V3F, T4F V4F, T2F C4UB V3F, T2F C3F V3F, T2F N3F V3F, T2F C4F N3F V3F, 或 T4F C4F N3F V4F。

交错数组的效果

交错数组 (*format, stride, pointer*) ;

与命令序列的效果相同

如果（*格式*或*步长*无效）则生成相应错误

else {

int str;

根据表2.5*和format值*设置e_t、e_c、e_n、s_t、s_c、s_v、t_c、p_c、p_n、p_v*及*s。

str = *stride*;

if (str 为零) str = s;

禁用客户端状态 (边缘标志数组) ;**禁用客户端状态** (索引数组) ;**禁用客户端状态** (辅助颜色数组) ;**禁用客户端状态** (雾坐标数组) ;

if (e_t) {

启用客户端状态 (纹理坐标数组) ; **TexCoordPointer** (s_t , 浮点数, str, *指针*) ; 否则

}

禁用客户端状态 (纹理坐标数组) ;

if (e_c) {

启用客户端状态 (COLOR ARRAY) ; **颜色指针** (s_c, t_c, str, *pointer* + p_c) ; 否则

}

禁用客户端状态 (COLOR ARRAY) ;

if (e_n) {

启用客户端状态 (法线数组) ;

NormalPointer (浮点数, str, *指针* + p_n) ;

} else

format	e_t	e_c	e_n	s_t	s_c	s_v	t_c
V2F	错误	假	假			2	
V3F	假	假	假			3	
C4UBV2F	假	真	假		4	2	无符号字节
C4UB V3F	假	真	假		4	3	无符号字节
C3FV3F	假	真	假		3	3	浮动
N3F V3F	假	假	真			3	
C4F N3F V3F	假	真	真		4	3	浮动
T2F V3F	真	假	假	2		3	
T4FV4F	真	假	假	4		4	
T2F C4UB V3F	真	真	假	2	4	3	无符号字节
T2F C3F V3F	真	真	假	2	3	3	浮动
T2F N3F V3F	真	假	真	2		3	
T2F C4F N3F V3F T4F C4F N3F V4F	真 真	真 真	真 真	2 4	4 4	3 4	FLOAT FLOAT

format	p_c	p_n	p_v	s
V2FV 3F C4UB V2F C4UB V3F	0 0		0 0 cc	$2f$ $3f$ $c + 2\pi$ $c + 3f$
C3F V3F N3F V3F C4F N3F V3F T2F V3F	0 0	0 $4f$	$3f$ $3f7$ $f2f$	$6f$ $6f1$ $0f5f$
T4F V4F T2F C4UB V3F-T2F C3F V3F T2F N3F V3F	$2f$ $2f$	$2f$	$4f$ $c +$ $2f5f5f$	$8f$ $c + 5f$ $8f$ $8f$
T2F C4F N3F V3F T4F C4F N3F V4F	$2f$ $4f$	$6f$ $8f$	$9f$ $11f$	$12f$ $15f$

表 2.5： 控制**交错数组**执行的变量。f 为 sizeof(FLOAT) 的大小。c 为 sizeof(UNSIGNED BYTE) 的 4 倍，向上取整至最接近的 f 的倍数。所有指针运算均以 sizeof(UNSIGNED BYTE) 为单位进行。

禁用客户端状态(NORMAL ARRAY); **启用客户端状态**(VERTEX ARRAY);
顶点指针(s_v, FLOAT, str, *pointer* + p_v);

}

若支持的纹理单元数量（MAX TEXTURE COORDS的值）为m，支持的通用顶点属性数量（MAX VERTEX ATTRIBS的值）为*n*，则实现顶点数组所需的客户端状态包含：用于客户端活动纹理单元选择器的整数、7 + *m* + n个布尔值、7 + *m* + n个内存指针、 7 + *m* + *n* 个整数步长值、7 + *m* + *n* 个表示数组类型的符号常量、3 + *m* + *n* 个表示元素值的整数，以及 *n* 个指示归一化的布尔值。 初始状态下，客户端活动纹理单元选择器为TEXTURE0，所有布尔值均为false，内存指针均为NULL，步长均为零，数组类型均为FLOAT，表示每个元素值的整数均为四。

2.9 缓冲区对象

第2.8节所述的顶点数据数组存储在客户端内存中。有时需要将频繁使用的客户端数据（如顶点数组数据）存储在高性能的服务器内存中。GL缓冲对象提供了一种机制，允许客户端在该内存中分配、初始化数据并进行渲染。

缓冲对象的命名空间采用无符号整数，其中零号保留给GL使用。创建缓冲对象需将未使用的名称绑定至ARRAY BUFFER，绑定操作通过调用以下函数实现：

```
void BindBuffer( enum target, uint buffer );
```

实现，其中target设为ARRAY BUFFER，buffer设为未使用的名称。生成的缓冲区对象是一个新的状态向量，初始化时包含零大小的内存缓冲区，并包含表2.6所列的状态值。

BindBuffer 也可用于绑定现有缓冲区对象。若绑定成功，则不会改变新绑定缓冲区对象的状态，且与*目标*相关的任何先前绑定关系均告解除。

当缓冲区对象处于绑定状态时，对其绑定目标执行的GL操作将影响该绑定缓冲区对象；而查询绑定缓冲区对象的目标时，将返回绑定对象的状态。

初始状态下，保留名称零绑定至数组缓冲区。由于不存在名称为零的缓冲区对象，客户端尝试修改

名称	类型	初始值	法律价值
缓冲区大小 _	整数	0	任何非负整数
缓冲区使用情况	枚举	静态绘制	流绘制、流读取、 _ 流复制、静态绘制、静态读取、静态复制、动态绘制 、动态读取、_动态复制 _ _ _ _
缓冲区访问 _	枚举	读写	只读，只写， _ 读写 _
缓冲映射 _	布尔值	假	真，假
缓冲区映射指针 _	void*	NULL	地址

表 2.6： 缓冲区对象参数及其值。

当零绑定时查询目标数组缓冲区的缓冲区对象状态将引发GL错误。 _

通过调用

```
void DeleteBuffers( sizei n, const uint *buffers );
```

buffers 包含待删除的 *n* 个缓冲区对象名称。删除后缓冲区对象将清空内容，其名称重新变为未占用状态。*buffers* 中未占用的名称将被静默忽略，零值亦同。

命令

```
void GenBuffers( sizei n, uint *buffers );
```

*返回缓冲区中n个*先前未使用的缓冲对象名称。这些名称仅在**生成缓冲区**时被标记为已使用，但它们在首次绑定时才获得缓冲区状态，如同未被使用一般。

当缓冲区对象处于绑定状态时，对其执行的任何GL操作都会影响该对象的所有其他绑定。若在绑定期间删除缓冲区对象，当前上下文（即调用**Delete-Buffers的**线程）中所有指向该对象的绑定将重置为零。 其他上下文和线程中的绑定不受影响，但在其他线程中使用已删除缓冲区将产生未定义结果，包括但不限于可能的GL错误和渲染损坏。然而，在其他上下文或线程中使用已删除缓冲区未必导致程序终止。

缓冲区对象的数据存储通过调用

```
void BufferData( enum target, sizeiptr size, const void *data, enum usage );
```

当*目标*设置为数组缓冲区、*大小*设置为数据存储的基本机器单位尺寸，且*数据*指向客户端内存中的源数据时。若*数据*不为空，则源数据将被复制到缓冲区对象的数据存储中。若*数据*为空，则缓冲区对象数据存储的内容未定义。

usage 参数需指定为九个枚举值之一，用于标识数据存储的预期应用场景。具体值如下：

`STREAM DRAW` 数据存储内容由应用程序指定一次，最多作为GL绘图命令的源使用数次。

`STREAM READ` 数据存储内容将通过从GL读取数据进行一次指定，并最多被应用程序查询几次。

`STREAM COPY` 数据存储内容将通过从GL读取数据进行一次初始化，最多作为GL绘图命令的源使用数次。

静态绘制 数据存储内容由应用程序指定一次，并多次作为GL绘制命令的源。

静态读取 数据存储内容将通过从图形层读取数据进行一次指定，并由应用程序多次查询。

静态复制数据存储内容将通过从图形渲染器读取数据进行一次性定义，并多次作为图形渲染命令的源数据。

动态绘制：应用程序将反复重新定义数据存储内容，并多次将其作为GL绘图命令的源。

动态读取 数据存储内容将通过从GL读取数据反复重新定义，并被应用程序多次查询。

动态复制数据存储内容将通过从图形层读取数据反复重新定义，并多次作为图形层绘制命令的源数据。

*使用*模式仅作为性能提示提供。指定的使用模式值不会限制数据存储的实际使用模式。

BufferData 删除任何现有的数据存储，并将缓冲对象状态变量的值设置为表 2.7 所示。

名称	值
缓冲区大小	*size*
缓冲区使用率	*使用量*
缓冲区访问	读写
缓冲映射	FALSE
缓冲映射指针	空

表 2.7： 缓冲区对象初始状态。

客户端必须使数据元素符合客户端平台的要求，并额外满足基础层级要求：缓冲区内指向*由N个*基本机器单位组成的数据项的偏移量必须是*N*的倍数。

若通用语言（GL）无法创建指定大小的数据存储，则触发
生成OUT OF MEMORY错误。

若需修改缓冲区对象数据存储中的部分或全部数据，客户端可使用命令

```
void BufferSubData( enum target, intptr offset, sizeiptr size, const void
    *data );
```

*目标*设置为数组缓冲区。偏移量和*大小*以基本机器单位为单位，指定缓冲区对象中需替换的数据范围。*数据*指定客户端内存中长度为基本机器单位的区域，该区域包含用于替换指定缓冲区范围的数据。若*偏移*量或*大小*小于零，或*偏移*量加*大小*大于缓冲区大小值，则会引发无效值错误。

通过调用

```
void *MapBuffer( enum target, enum access );
```

当*目标*设置为数组缓冲区时，若图形渲染器能够将缓冲区对象的数据存储映射到客户端地址空间，**MapBuffer**将返回指向该数据存储的指针值。若缓冲区数据存储已处于映射状态，**MapBuffer**将返回NULL并触发无效操作错误。 否则**MapBuffer** 返回 NULL 并触发 OUT OF MEMORY 错误。*访问*权限可设为 READ ONLY（只读）、WRITE ONLY（只写）或 READ WRITE（读写），用于指定数据存储映射期间客户端可通过指针执行的操作类型。

MapBuffer 设置缓冲区对象状态值，如表 2.8 所示。

名称	值
缓冲区访问	*访问*
缓冲区映射	TRUE
缓冲映射指针	指向数据存储的指针

表 2.8： **MapBuffer** 设置的缓冲区对象状态。

MapBuffer返回的非空指针可在映射有效期间，由客户端用于修改和查询缓冲区对象数据，且需符合映射访问规则。若使用该指针尝试修改只读数据存储区，或从只写数据存储区读取数据，则不会触发GL错误，但操作可能变慢并导致系统错误（可能包括程序终止）。 **MapBuffer**返回的指针值不得作为参数传递给GL命令。例如，不得用于指定数组指针，或用于指定/查询像素/纹理图像数据；此类操作将产生未定义结果，尽管出于性能考虑，实现方可能不会检查此类行为。

调用 **BufferSubData** 修改映射缓冲区的数据存储将引发 INVALID OPERATION 错误。

缓冲区对象数据存储的映射可能具有非标准性能特征。 例如，此类映射可能被标记为不可缓存的内存区域，此时读取操作可能极其缓慢。为确保最佳性能，客户端应根据BUFFER USAGE和BUFFER ACCESS的值来使用映射。若使用方式与这些值不符，其速度可能比普通内存慢多个数量级。

在客户端指定映射数据存储的内容之后，且在该存储中的数据被任何GL命令解除引用之前，必须通过调用

```
boolean UnmapBuffer( enum target );
```

此时需将*target*设为ARRAY BUFFER。解除映射将使指向该数据存储的指针失效，并将对象的BUFFER MAPPED状态设为FALSE，其BUFFER MAP POINTER状态设为NULL。

UnmapBuffer 返回 TRUE，除非缓冲区映射期间其数据存储中的数据值已损坏。此类损坏可能由屏幕分辨率变更或其他窗口系统相关因素导致。

导致系统堆（如高性能图形内存）被丢弃的事件。GL实现必须保证此类损坏仅可能发生在缓冲区数据存储映射期间。若发生此类损坏，**UnmapBuffer**将返回FALSE，且缓冲区数据存储的内容将变得未定义。

若缓冲区数据存储已处于未映射状态，**UnmapBuffer**将返回FALSE并触发INVALID OPERATION错误。但作为缓冲区删除或重新初始化副作用发生的解除映射操作不视为错误。

2.9.1 缓冲区对象中的顶点数组

顶点数组数据块可存储于缓冲对象中，其格式与布局选项与客户端顶点数组支持的相同。但预期GL实现至少应针对以下数据进行优化：所有分量均以浮点数表示的数据，以及分量以浮点数或无符号字节表示的颜色数据。

为每个顶点数组类型关联的客户端状态中添加了一个缓冲区对象绑定点。指定顶点数组位置和组织结构的命令会将绑定到数组缓冲区的缓冲区对象名称复制到与所指定类型顶点数组对应的绑定点。 例如，**NormalPointer**命令将ARRAY BUFFER BINDING的值（即目标ARRAY BUFFER对应缓冲区绑定可查询名称）复制到客户端状态变量NORMAL ARRAY BUFFER BINDING中。

渲染命令 **ArrayElement**、**DrawArrays**、**DrawElements**、**DrawRangeElements**、**MultiDrawArrays** 和 **MultiDrawElements** 的运作方式与先前定义一致，但当数组的缓冲区绑定值不为零时，启用的顶点数组和属性数组的数据将从缓冲区获取。 当数组从缓冲区对象获取时，该数组的指针值将用于计算指向缓冲区对象数据存储区的偏移量（以基本机器单位为单位）。该偏移量通过从指针值中减去空指针计算得出，此时两个指针均被视为指向基本机器单位的指针。

在单次渲染操作中，顶点数组或属性数组可由客户端内存与各类缓冲区对象的任意组合提供数据。

若尝试从当前映射的缓冲区对象中获取数据，将触发INVALID OPERATION 错误。

2.9.2 缓冲对象中的数组索引

数组索引块可采用与客户端索引数组相同的格式选项存储于缓冲区对象中。初始时ELEMENT ARRAY BUFFER绑定值为零，**这意味着**DrawElements和**DrawRangeElements**将从其*索引*参数传递的数组中获取索引，而**MultiDrawElements**则从其*索引*参数传递的数组指针数组中获取索引。

通过调用**BindBuffer**命令，将*目标*设置为ELEMENT ARRAY BUFFER，并将*缓冲区*设置为缓冲区对象的名称，即可将缓冲区对象绑定至ELEMENT ARRAY BUFFER。若不存在对应的缓冲区对象，则将根据第2.9节的定义初始化一个缓冲区对象。

BufferData、**BufferSubData**、**MapBuffer**和**UnmapBuffer**命令均可设置*目标*为ELEMENT ARRAY BUFFER使用。此时这些命令的运作方式与第2.9节所述相同，但操作对象为当前绑定至ELEMENT ARRAY BUFFER目标的缓冲区。

当非空缓冲区对象名称绑定至元素数组缓冲区时，DrawElements**与DrawRangeElements**函数将从该缓冲区对象获取索引值，其*索引*参数作为缓冲区对象内的偏移量使用，具体方式与第2.9.1节所述相同。**MultiDrawElements**函数同样从该缓冲区对象获取索引值，其*索引*参数作为指向指针数组的指针使用，该数组中的指针代表缓冲区对象内的偏移量。

通过将未使用的名称绑定到数组缓冲区和元素数组缓冲区创建的缓冲区对象在形式上等价，但图形渲染器可能根据初始绑定选择不同的存储实现方案。在某些情况下，通过将索引和数组数据存储在独立的缓冲区对象中，并使用对应的绑定点创建这些缓冲区对象，可实现性能优化。

2.10 矩形

存在一组GL命令，用于高效地通过两个角顶点来定义矩形。

```
void Rect{sifd}( T x1, T y1, T x2, T y2 );void Rect{sifd}v( T
v1[2], T v2[2] );
```

每个命令接受四种参数形式：要么是两个连续的(*x*, *y*)坐标对组合，要么是两个数组指针，每个数组包含*x*值后接y值。Rect命令的效果如下：

$$\mathbf{Rect}(x_1, y_1, x_2, y_2);$$

与以下命令序列完全等效：

```
Begin(POLYGON);
    Vertex2(x1,    y1);
    Vertex2(x2,    y1);
    Vertex2(x2,    y2);
    Vertex2(x1, y2);
结束();
```

根据发出的Rect命令类型，将调用相应的**Vertex2**命令。

Rect命令的执行情况调用相应的Vertex2命令。

2.11 坐标变换

本节及后续至2.14节的内容，将阐述采用固定功能方法转换顶点属性所需的状态值与操作流程。*可编程*顶点属性转换方法详见2.15节。

顶点、法线和纹理坐标在用于生成帧缓冲区图像前需先进行变换。我们将首先阐述顶点坐标的变换机制及其控制方式。

图2.6展示了应用于顶点的变换序列。 提交至图形渲染器的顶点坐标称为*物体坐标*。*模型*视图矩阵作用于这些坐标后生成*视点*坐标。随后通过投影矩阵对视点坐标进行变换，生成裁剪坐标。对裁剪坐标执行透视除法后，得到*归一化设备*坐标。最后通过*视口*变换将这些坐标转换为*窗口坐标*。

对象坐标、视点坐标和裁剪坐标均为四维坐标，包含*x*、*y*、z和w四个分量（按此顺序排列）。模型视图矩阵和投影矩阵均为4×4矩阵。

投影矩阵均为4×4矩阵。

若物体坐标系中的顶点由 $\begin{bmatrix} x_o \\ y_o \\ z_o \\ w_o \end{bmatrix}$ 且模型视图矩阵为 *M*，则顶点的视点坐标可表示为

$$\begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix} = M \begin{bmatrix} x_o \\ y_o \\ z_o \\ w_o \end{bmatrix}.$$

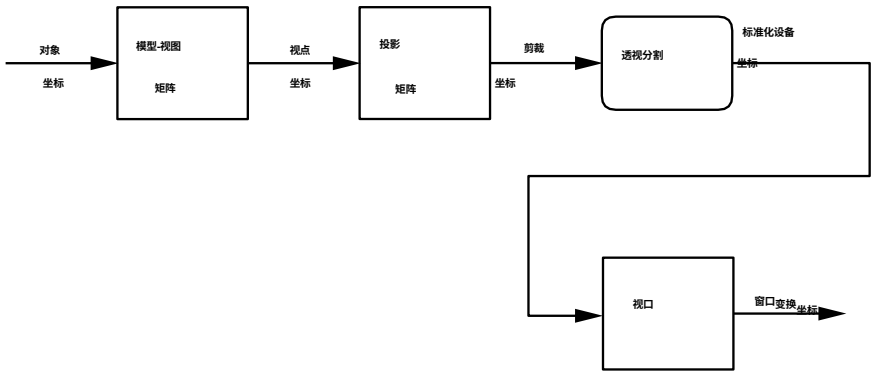


图2.6. 顶点变换序列。

同样地，若 P 为投影矩阵，则顶点的裁剪坐标为

$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix} = P \begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix}.$$

顶点的归一化设备坐标为

$$\begin{bmatrix} x_d \\ y_d \\ z_d \end{bmatrix} = \begin{bmatrix} x_c/w_c \\ y_c/w_c \\ z_c/w_c \end{bmatrix}.$$

2.11.1 视口控制

视口变换由视口的宽度和高度（分别用像素表示为 p_x和 p_y）及其中心点（o_x, o_y）（同样以像素为单位）决定。顶点的窗口坐标，$\begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix}$ 由以下公式给出：

$$\begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix} = \begin{bmatrix} (p_x/2)x_d + o_x \\ (p_y/2)y_d + o_y \\ [(f-n)/2]z_d + (n+f)/2 \end{bmatrix}.$$

应用于 z_d的因子和偏移量由 n 和 f 编码，通过以下方式设置

```
void DepthRange( clampd n, clampd f );
```

n 和 f 均被限制在 [0, 1] 范围内，所有 clampd 或 clampf 类型的参数亦如此。z_w 采用固定点表示法，其位数至少等于帧缓冲器的深度缓冲区位数。 我们假设所用的固定点表示法将每个值表示为 $k/(2^m - 1)$，其中 k 为 0, 1, ..., $2^m - 1$，即 k（例如1.0在二进制中表示为全1的字符串）。

视口变换参数通过以下函数指定：

```
void Viewport( int x, int y, sizei w, sizei h );
```

其中 x 和 y 分别表示视口左下角在窗口中的 x 和 y 坐标，w 和 h 分别表示视口的宽度和高度。上述方程中的视口参数可通过以下公式计算得出：$o_x = x + w/2$ 和 $o_y = y + h/2$；$p_x = w$，$p_y = h$。

视口宽度与高度在指定时将受限于实现相关的最大值。可通过执行相应Get命令（参见第6章）获取最大宽高值。视口最大尺寸必须大于或等于渲染显示器的可见尺寸。若 w 或 h 为负值，则返回INVALID VALUE。

实现视口变换所需的状态包含四个整数和两个受限浮点数值。初始状态下，w 和 h 分别设置为渲染窗口的宽度和高度。o_x 和 o_y 分别设置为 w/2 和 h/2。n 和 f 分别设置为 0.0 和 1.0。

2.11.2 矩阵

透视矩阵与模型视图矩阵通过多种命令进行设置与修改。受影响的矩阵由当前矩阵模式决定。当前矩阵模式通过以下命令设置：

```
void MatrixMode( enum mode );
```

该函数接受预定义常量TEXTURE、MODELVIEW、COLOR或PROJECTION作为参数值。TEXTURE将在第2.11.2节中描述，COLOR则在第3.6.3节中说明。若当前矩阵模式为MODELVIEW，则矩阵运算作用于模型视图矩阵；若为PROJECTION，则作用于投影矩阵。

影响当前矩阵的两个基本命令为

```
void LoadMatrix{fd}( T m[16] );void MultMatrix{fd}(
T m[16] );
```

LoadMatrix 接受指向一个 4×4 矩阵的指针，该矩阵以列优先顺序存储为 16 个连续的浮点数值，即：

$$\begin{bmatrix} a1 & a5 & a9 & a13 \\ a2 & a6 & a10 & a14 \\ a3 & a7 & a11 & a15 \\ a4 & a8 & a12 & a16 \end{bmatrix}.$$

（这与矩阵元素的标准行优先C语言排序不同。若采用标准排序，后续所有变换方程均需转置，此时表示向量的列将变为行。）

指定矩阵将当前矩阵替换为所指向的矩阵。**Mult-Matrix**与**LoadMatrix**采用相同类型的参数，但会将当前矩阵与所指向的矩阵相乘，并将乘积替换为当前矩阵。*若C*是当前矩阵，*M* 是 MultMatrix 参数指向的矩阵，则最终的当前矩阵 *C'*为

$$C' = C \cdot M。$$

命令

```
void LoadTransposeMatrix{fd}( T m[16] ); void
MultTransposeMatrix{fd}( T m[16] );
```

将存储在行优先顺序的4×4矩阵的指针作为16个连续的浮点数值，即作为

$$\begin{bmatrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ a9 & a10 & a11 & a12 \\ a13 & a14 & a15 & a16 \end{bmatrix}.$$

LoadTransposeMatrix[fd](m) 的效果

LoadTransposeMatrix[fd]（*m*）；

的效果与

LoadMatrix[fd]（m^T ）；

执行

MultTransposeMatrix[fd](*m*)；

的效果等同于

MultMatrix[fd](m^T)；

该命令

```
void LoadIdentity( void );
```

实际上调用**LoadMatrix函数**并传入单位矩阵：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

还有多种其他用于操作矩阵的命令。**Rotate**、**Translate**、**Scale**、**Frustum** 和 **Ortho** 均操作当前矩阵。每条命令都会计算一个矩阵，然后调用 **MultMatrix** 处理该矩阵。在

```
void Rotate{fd}( T ϑ, T x, T y, T z );
```

θ表示旋转角度（单位：度）；**向量v的**坐标**由**$\mathbf{v} = (x\ y\ z)^T$给出。计算得到的矩阵表示以指定轴为轴心、围绕通过原点的直线进行逆时针旋转（当该轴向上时，即右手定则决定旋转方向）。因此该矩阵为：

$$\begin{bmatrix} & & & 0 \\ & R & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

设 $\mathbf{u} = \mathbf{v}/||\mathbf{v}|| = (x'\ y'\ z')^T$。若

$$S = \begin{bmatrix} 0 & -z' & y' \\ z' & 0 & -x' \\ -y' & x' & 0 \end{bmatrix}$$

则

$$R = \mathbf{u}\mathbf{u}^T + \cos\vartheta(I - \mathbf{u}\mathbf{u}^T) + \sin\vartheta S.$$

函数的参数为

void 翻译(T *x*, T *y*, T *z*);

给出平移向量的坐标为 $(x\ y\ z)^T$ 。结果矩阵是指定向量的平移：

$$\begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

```
void Scale{fd}( T x, T y, T z );
```

实现沿x-, y-, z-轴的一般缩放。对应的矩阵为

$$\begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

对于

```
void Frustum( double l, double r, double b, double t, double n, double f );
```

坐标 $(l\ b\ n)^T$和 $(r\ t\ n)^T$ 分别指定近裁剪平面上映射到窗口左下角和右上角的点（假设视点位于 $(0\ 0\ 0)^T$ 处）。f 表示视点到远裁剪平面的距离。 若 n 或 f 小于或等于零，或 l 等于 r、b 等于 t、n 等于 f，则返回错误值 `INVALID VALUE`。对应矩阵为

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}.$$

```
void Ortho( double l, double r, double b, double t, double n, double f );
```

描述一个产生平行投影的矩阵。$(l\ b\ n)^T$和 $(r\ t\ n)^T$ 分别指定近裁剪平面上映射到窗口左下角和右上角的点。f 给出从眼睛到裁剪平面的距离

到远裁剪平面的距离。若 *l* 等于 *r*、*b* 等于 *t* 或 *n* 等于 *f*，则返回错误值 INVALID VALUE。对应矩阵为

$$\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

对于每组纹理坐标，将一个 4×4 矩阵应用于相应的纹理坐标。该矩阵应用于对应的纹理坐标。 ×4矩阵应用于对应的纹理坐标。该矩阵以

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

其中左侧矩阵为当前纹理矩阵。该矩阵将作用于纹理坐标生成所得的坐标（可能直接采用当前纹理坐标），最终转换后的坐标即成为顶点关联的纹理坐标。将矩阵模式设为TEXTURE时，前述矩阵运算将应用于纹理矩阵。

命令

```
void ActiveTexture( enum texture );
```

指定活动纹理单元选择器，即活动纹理单元。每个纹理单元最多包含两个独立子单元：纹理坐标处理单元（由纹理矩阵堆栈和纹理坐标生成状态构成）以及纹理图像单元（包含第3.8节定义的所有纹理状态）。 在支持不同数量纹理坐标集和纹理图像单元的实现中，某些纹理单元可能仅包含上述两个子单元中的一个。

活动纹理单元选择器指定了由涉及纹理坐标处理的命令所访问的纹理坐标集。 此类命令包括：访问当前矩阵堆栈的命令（当MATRIX MODE为TEXTURE时）、控制点精灵坐标替换的TexEnv命令（参见第3.3节）、**Tex-Gen**（第2.11.4节）、**启用/禁用**命令（若选择了任何纹理坐标生成枚举项），以及查询当前纹理坐标和当前光栅纹理坐标的操作。若当前活动纹理对应的纹理坐标集编号大于或等于实现相关的常量MAX TEXTURE COORDS，则任何此类命令将引发INVALID OPERATION错误。

活动纹理单元选择器还负责选择由涉及纹理图像处理的指令（参见第3.8节）所访问的纹理图像单元。此类指令包括所有**TexEnv指令**变体（控制点精灵坐标替换的指令除外）、**TexParameter**指令、TexImage指令、**BindTexture指令**、任何纹理目标（如TEXTURE 2D）的启用/**禁**用指令，以及所有此类状态的查询指令。 若当前ACTIVE TEXTURE值对应的纹理图像单元编号大于或等于实现相关的常量MAX COMBINED TEXTURE IMAGE UNITS，则将触发错误INVALID OPERATION

。

ActiveTexture在指定无效*纹理*时会引发INVALID ENUM错误。*texture*参数*采用TEXTUREi*形式的符号常量，表示需修改*第i 个*纹理单元。常量*遵循TEXTUREi* = TEXTURE0 + *i的规则*（*i*取值范围为0至*k* 1，*其中*k取值为MAX TEXTURE COORDS与MAX COMBINED TEXTURE IMAGE UNITS中的较大值）。

为向下兼容《 》及更早版本的 规范， 实现相关的常量 中的MAX TEXTURE UNITS指定了该实现支持的常规纹理单元数量。其值不得大于MAX TEXTURE COORDS与MAX COMBINED TEXTURE IMAGE UNITS中的较小值。

对于每种矩阵模式（MODELVIEW、 PROJECTION、 COLOR）及每个纹理单元，均存在独立的矩阵栈。MODELVIEW模式的栈深度至少为32（即至少包含32个模型视图矩阵）。其他模式的栈深度至少为2。所有纹理单元的纹理矩阵栈具有相同深度。 任何模式下的当前矩阵即为该模式栈顶的矩阵。

```
void PushMatrix( void );
```

将当前矩阵复制到栈顶及其下方位置，使栈中元素减少一个。

```
void PopMatrix( void );
```

从栈顶弹出顶部条目，用栈中第二个条目替换当前矩阵。入栈或出栈操作发生在与当前矩阵模式对应的栈上。当栈中仅有一个条目时弹出矩阵将引发STACK UNDERFLOW错误；向已满栈入栈矩阵将引发STACK OVERFLOW错误。

当前矩阵模式为 TEXTURE 时，操作对象为活动纹理单元对应的纹理矩阵堆栈。

实现变换所需的状态包含：用于活动纹理单元选择器的整数；指示当前矩阵模式的四值整数；

模 式 ，以及至少包含两个4× ×4矩阵的栈（分别用于COLOR、PROJECTION及每组纹理坐标TEXTURE），另有一个至少包含32个4×4矩阵的栈用于MODELVIEW。每个矩阵栈均关联一个栈指针。初始时每个栈仅含一个矩阵，且所有矩阵均设为单位矩阵。 初始活动纹理单元选择器为TEXTURE0，初始矩阵模式为MODELVIEW。

2.11.3 法线变换

最后，我们探讨模型视图矩阵与变换状态如何影响法线。在用于光照计算前，法线会通过由模型视图矩阵派生的矩阵转换为视点坐标系。转换后的法线需进行缩放与归一化操作，使其在光照计算中保持单位长度。缩放与归一化操作由

```
void Enable( enum target );
```

以及

```
void Disable( enum target );
```

，其中*目标值*可设为RESCALE NORMAL或NORMALIZE。这需要两个状态位。初始状态下法线不进行缩放或归一化。

若模型视图矩阵为 M，则法线转换为视点坐标的公式为：

$$(n_x' \quad n_y' \quad n_z' \quad q') = (n_x \quad n_y \quad n_z \quad q) \cdot M^{-1}$$

其中，若 q 为关联顶点坐标，则

$$q = \begin{cases} 0, & w = 0, \\ \dfrac{-(n_x \quad n_y \quad n_z)}{w}, & w \neq 0 \end{cases} \tag{2.1}$$

实现可选择转换为 $(n_x \quad n_y \quad n_z)$ 转换为眼睛坐标系，使用

$$(n_x' \quad n_y' \quad n_z') = (n_x \quad n_y \quad n_z) \cdot M_u^{-1}$$

其中 M_u是从 M 中取出的左上角 3x3 矩阵。

重缩放将转换后的法线乘以缩放因子

$$(n_x' \quad n_y' \quad n_z') = f (n_x' \quad n_y' \quad n_z')$$

若缩放功能禁用，*则*$f = 1$。若缩放功能启用，则f按下列公式计算：$f = m_{(ij)}$（其中m^{ij}*表示矩阵*M的*第*i*行第*j*列*元素，矩阵顶部行编号为1，最左列编号为1）

$$f = \frac{1}{\sqrt{m_{31}^2 + m_{32}^2 + m_{33}^2}}$$

请注意，若发送至GL的法线为单位长度，且模型视图矩阵均匀缩放空间，则重新缩放会使变换后的法线恢复为单位长度。

或者，实现方案也可将f定义为：

$$f = \frac{1}{\sqrt{n_x'^2 + n_y'^2 + n_z'^2}}$$

重新计算每个法线的f值。这使得所有非零长度法线都具有单位长度，无论其输入长度如何，也与模型视图矩阵的性质无关。

缩放后，用于光照的最终变换法线 n_f 计算如下：

为

$$n_f = m (n_x'' \quad n_y'' \quad n_z'')$$

如果归一化功能被禁用，则 $m = 1$。否则

$$m = \frac{1}{\sqrt{n_x''^2 + n_y''^2 + n_z''^2}}$$

由于我们既未指定浮点格式也未指定矩阵求逆方法，因此无法在条件数较差（近奇异）的模型视图矩阵 M 情况下定义行为。若矩阵为精确奇异矩阵，则变换后的法向量未定义。若 GL 实现判定模型视图矩阵不可逆，则逆矩阵的元素可任意设定。 无论何种情况，法线变换或使用变换后法线均不会导致GL中断或终止。

2.11.4 纹理坐标生成

与顶点关联的纹理坐标可取自当前纹理坐标，或根据依赖顶点坐标的函数生成。命令

```
void TexGen{ifd}( enum coord, enum pname, T param ); void TexGen{ifd}v( enum coord,
enum pname, T params );
```

控制纹理坐标生成。*coord*必须是常量S、T、R或Q之一，分别表示对应坐标为s、t、r或q坐标。在命令的第一种形式中，param是指定单值纹理生成参数的符号常量；在第二种形式中，params是指向指定纹理生成参数值数组的指针。*pname* 必须是三个符号常量之一：TEXTURE GEN MODE、OBJECT PLANE 或 EYE PLANE。若 *pname* 为 TEXTURE GEN MODE，则 *params* 需指向或直接指定为整型值，该值须为符号常量 OBJECT LINEAR、EYE LINEAR、SPHERE MAP、REFLECTION MAP 或 NORMAL MAP 之一。

若纹理生成模式为对象线性，则由坐标*coord*指示的生成函数为：

$$g = p_1 x_o + p_2 y_o + p_3 z_o + p_4 w_o$$

x_o、y_o、z_o和 w_o是顶点的物体坐标。p_1、…、p_4 通过调用 **TexGen** 并设置 *pname* 为 OBJECT PLANE 来指定，此时 *params* 指向包含 p_1、…、$p_{(4)}$的数组。 每个纹理坐标对应一组独立的平面方程系数；*coord*参数指示指定系数所关联的坐标。

若纹理生成模式为EYE LINEAR，则该函数为

$$g = p'_1 x_e + p'_2 y_e + p'_3 z_e + p'_4 w_e$$

其中

$$(p'_1 \quad p'_2 \quad p'_3 \quad p'_4) = (p_1 \quad p_2 \quad p_3 \quad p_4) M^{-1}$$

x_e、y_e、z_e和 w_e是顶点的眼坐标。p_1、…、p_4 通过调用 **TexGen** 并设置 *pname* 为 EYE PLANE 来设定，这与在 OBJECT PLANE 情况下设置系数相对应。 M 是指定 $p_1, \ldots, p_4$ 时生效的模型视图矩阵。若 M 条件差或奇异，计算出的纹理坐标可能不准确或未定义。

当配合适当构造的纹理图像使用时，通过在**纹理生成器**中调用TEXTURE GEN MODE并指定SPHERE MAP模式，可模拟球形环境在多边形表面的反射图像。 球面贴图纹理坐标生成规则如下：设从原点指向顶点（视点坐标系）的单位向量为**u**，当前法线向量经视点坐标系变换后记为$\mathbf{n}_{(e)}$。 反射向量r定义为

$$\mathbf{r} = \mathbf{u} - 2\mathbf{n}_e^T(\mathbf{n}_e\mathbf{u})$$

设 $m = 2\sqrt{r_x^2 + r_y^2 + (r_z + 1)^2}$。则赋予 s 坐标的值为

(**TexGen** 的第一个参数值为 S) 赋值为 $s = r_x/m + \frac{1}{2}$；t 坐标的赋值为 $t = r_y/m + \frac{1}{2}$。当 *pname* 指示球面映射时，若使用 R 或 Q 坐标调用 **TexGen**，将引发 INVALID ENUM 错误。

若纹理生成模式为反射贴图，则按球面贴图模式所述计算反射向量 **r**。此时s坐标值为$s = r_x$；t坐标值为$t = r_y$；r坐标值为$r = r_z$。当pname指示为反射贴图时，若以Q坐标调用**TexGen**将引发INVALID ENUM错误。

若纹理生成模式为法线贴图，则按第2.11.3节所述计算法线向量n_f。此时s坐标的赋值为$s = n_{f_x}$；t坐标的赋值为$t = n_{f_y}$；而赋值给...的值为...
r坐标为 $r = n_{f_z}$（其中 n_{f_x}、n_{f_y}和 n_{f_z}是 n_f 的分量。）
当*pname*指示为NORMAL MAP时，若使用Q坐标调用**TexGen**将产生
将返回错误INVALID ENUM。

纹理坐标生成函数**通过启用**和**禁用**指令配合参数TEXTURE GEN S、TEXTURE GEN T、TEXTURE GEN R或TEXTURE GEN Q（分别对应相应纹理坐标）来启用或禁用。 启用时，根据当前EYE_LINEAR、OBJECT_LINEAR或SPHERE MAP规范计算指定纹理坐标（取决于该坐标当前的TEXTURE GEN MODE设置）。禁用时，后续顶点将从当前纹理坐标中获取指定坐标值。

每个纹理单元生成纹理坐标所需的状态包含：每个坐标对应一个五值整数（表示坐标生成模式），以及每个坐标对应一位（用于指示纹理坐标生成功能的启用或禁用状态）。此外，对于EYE LINEAR和OBJECT LINEAR中的四个坐标，均需配置四个系数。初始状态下所有纹理坐标的生成功能均处于禁用状态。 *s*方向上p_i初始值均为0，唯p_1为1；*t*方向上p_i全为0，唯p_2为1。*r*与*q*方向的p_i值均为0。上述p_i值同时适用于EYE LINEAR与OBJECT LINEAR两种模式。初始状态下所有纹理生成模式均为EYE LINEAR。

2.12 裁剪

基元被裁剪*至裁剪体积*。在裁剪坐标系中，*视图体积*由以下关系定义：

$$-w_c \le x_c \le w_c$$
$$-w_c \le y_c \le w_c$$
$$-w_c \le z_c \le w_c$$

该视图体积可通过最多*n*个客户端定义的剪裁平面进一步限制以生成剪裁体积（n为实现相关的最大值，至少为6）。每个客户端定义的平面指定一个半空间。剪裁体积即所有此类半空间与视图体积的交集（若未启用任何客户端定义的剪裁平面，则剪裁体积即为视图体积）。

客户端定义的裁剪平面通过以下方式指定：

```
void ClipPlane(枚举 p，双精度数组 eqn[4]);
```

第一个参数 *p* 的值为符号常量 *CLIP* PLANEi，其中 *i* 是介于 0 与 *n* 之间的整数 1，表示 *n* 个客户端定义的剪裁平面之一。*eqn* 是由四个双精度浮点数组成的数组。 这些是物体坐标系中平面方程的系数：p_1、p_2、p_3和p_4（按此顺序排列）。在指定这些系数时，会应用当前模型视图矩阵的逆矩阵，从而得到

$$(p'_1 \quad p'_2 \quad p'_3 \quad p'_4) = (p_1 \quad p_2 \quad p_3 \quad p_4)\, M^{-1}$$

(*其中*M为当前模型视图矩阵；*若*M奇异则所得平面方程未定义，*若*M条件*差*则可能不准确)以获得眼坐标系中的平面方程系数。所有满足

$$(p'_1 \quad p'_2 \quad p'_3 \quad p'_4) \begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix} \ge 0$$

位于平面定义的半空间内；不满足此条件的点不位于该半空间内。

当顶点着色器处于活动状态时，向量 $(x_e, y_e, z_e, w_e)^T$ 不再被计算。取而代之的是使用 `glClipVertex` 内置变量的值。若顶点着色器未写入 `gl_ClipVertex`，其值将未定义，这意味着对任何客户端定义的裁剪平面的裁剪结果同样

未定义。用户必须确保裁剪顶点与客户端定义的裁剪平面定义在同一坐标系中。

客户端定义的剪裁平面通过通用**启**用命令启用，通过禁用命令禁用。两者命令的参数值均为 *CLIP PLANEi*，其中 *i* 是介于 0 与 *n* 之间的整数；指定 *i* 的值可启用或禁用索引为 *i* 的平面方程。常量遵循 `CLIP PLANEi` = `CLIP PLANE0` + *i* 的关系。

若待处理的基元为点，当其位于裁剪体积内部时，裁剪操作保持其不变；否则将其舍弃。 若原始元素为线段，当其完全位于裁剪体积内时保持不变；当完全位于体积外时则被舍弃。若线段部分位于体积内、部分位于体积外，则该线段将被裁剪，并为一个或两个顶点计算新的顶点坐标。被裁剪的线段端点同时位于原始线段和裁剪体积边界上。

该裁剪操作为每个裁剪顶点生成0至1之间的裁剪值t。若裁剪顶点坐标为$\mathbf{P}$，原始顶点坐标为$\mathbf{P}_1$和$\mathbf{P}_2$，则t值由以下公式计算：

$$\mathbf{P} = t\mathbf{P}_1 + (1 - t)\mathbf{P}_2。$$

t的值用于颜色、次要颜色、纹理坐标和雾坐标裁剪（第2.14.8节）。

若原始图形为多边形，则当其所有边完全位于裁剪体积内部时通过裁剪；否则将被裁剪或舍弃。多边形裁剪可能导致边被截断，但因需保持多边形连通性，这些被截断的边将通过沿裁剪体积边界的新边重新连接。 因此裁剪可能需要向多边形引入新顶点。这些顶点关联有边标记：裁剪引入的边被标记为边界边（边标记`TRUE`），而多边形原有边在这些顶点处被截断后仍保留其原始标记。

若多边形恰好与裁剪体边界的一条边相交，则裁剪后的多边形必须包含该边界边上的一点。该点必须位于边界边与原始多边形顶点凸包的交界处。我们提出此要求是因为多边形可能并非完全平面。

当线段或多边形的顶点具有符号不同的w_c值时，裁剪后可能产生多个连通分量。图形库实现无需处理此情况，即裁剪仅需生成原始图形中$w_c > 0$区域的部分。

使用裁剪平面渲染的基元必须满足互补性准则——

假设一个单剪切平面，其系数为 (p'_1 p'_2 p'_3 p'_4)（或一个数值（具有相似规格的剪裁平面）被启用，并绘制一系列基元。接下来，假设原始剪裁平面被重新指定为系数 ($-p'_1$ $-p'_2$ $-p'_3$ $-p'_4$)（其他裁剪平面相应调整）后，原始图形将被重新绘制（且图形渲染器处于相同状态）。此时，原始图形不得遗漏任何像素，且在被裁剪平面截断的区域内，任何像素均不得重复绘制。

进行裁剪所需的状态至少包含6组平面方程（每组由四个双精度浮点系数构成）以及至少6个对应位，用于指示这些客户端定义的平面方程中哪些处于启用状态。初始状态下，所有客户端定义的平面方程系数均为零，所有平面均处于禁用状态。

2.13 当前光栅位置

*当前光栅化位置*用于直接影响帧缓冲区像素的命令。这些命令会绕过顶点变换和基元组合，将在下一章中进行说明。然而，当前光栅化位置与顶点具有某些共同特性。

当前光栅位置通过以下命令之一设置：

```
void RasterPos{234}{sifd}( T coords ); void
RasterPos{234}{sifd}v( T coords );
```

RasterPos4 接受四个参数分别表示 *x*、*y*、*z* 和 *w*。**RasterPos3**（或 **RasterPos2**）原理相同，但仅设置 *x*、*y* 和 *z*（*w* 默认设为 1），或仅设置 *x* 和 *y*（*z* 默认设为 0，*w* 默认设为 1）。

获取当前光栅纹理坐标的行为受活动纹理状态设置的影响。

坐标被视为在**顶点**命令中指定的坐标。若顶点着色器处于活动状态，则*使用*x、y、z*和*w坐标作为顶点的对象坐标来执行该顶点着色器。 否则，x、y、z和w坐标将通过当前模型视图矩阵和投影矩阵进行变换。这些坐标与当前值共同用于生成主色、次色及纹理坐标，其处理方式与顶点相同。由此产生的颜色和纹理坐标将覆盖当前光栅化位置关联数据中存储的颜色和纹理坐标。 若顶点着色器处于活动状态，则当前光栅化距离将设置为着色器内置变量`gl_FogFragCoord`的值。否则，若雾源值（参见第3.10节）

若为雾坐标，则当前光栅化距离设置为当前雾坐标值。否则，当前光栅化距离设置为仅经当前模型视图矩阵变换后，从视点坐标系原点到顶点的距离。该距离可参照第3.10节所述方法进行近似计算。

由于顶点着色器可能在光栅化位置设定时执行，任何未被着色器写入的属性将导致当前光栅化位置处于未定义状态。顶点着色器应输出所有变量，这些变量将在使用当前光栅化位置进行像素基元光栅化时被使用。

变换后的坐标将作为点传递至裁剪操作。若该"点"未被剔除，则计算其到窗口坐标的投影（参见第2.11节），将其保存为当前光栅化位置并设置有效位。若该"点"被剔除，则当前光栅化位置及其关联数据将变为未定义状态，有效位被清除。 图2.7总结了当前光栅位置的行为。

此外，当前光栅位置也可通过**WindowPos命令**之一设置：

命令之一设置：

```
void WindowPos{23}{ifds}( T coords );
void WindowPos{23}{ifds}v( const T coords );
```

WindowPos3 接受三个值表示 x、y 和 z，而 **WindowPos2** 接受两个值表示 x 和 y，z 默认设为 0。当前光栅坐标位置 (x_w, y_w, z_w, w_c) 定义为：

$$x_w = x$$

$$y_w = y$$

$$z_w = \begin{cases} n, & z \le 0 \\ fn + z(f - n), & \text{否则/ } z \ge 1 \end{cases}$$

$$w_c = 1$$

其中 n 和 f 是传递给 **DepthRange** 的值（参见第 2.11.1 节）。

窗口位置函数不执行光照、纹理坐标生成与变换以及裁剪操作。相反，在RGBA模式下，当前

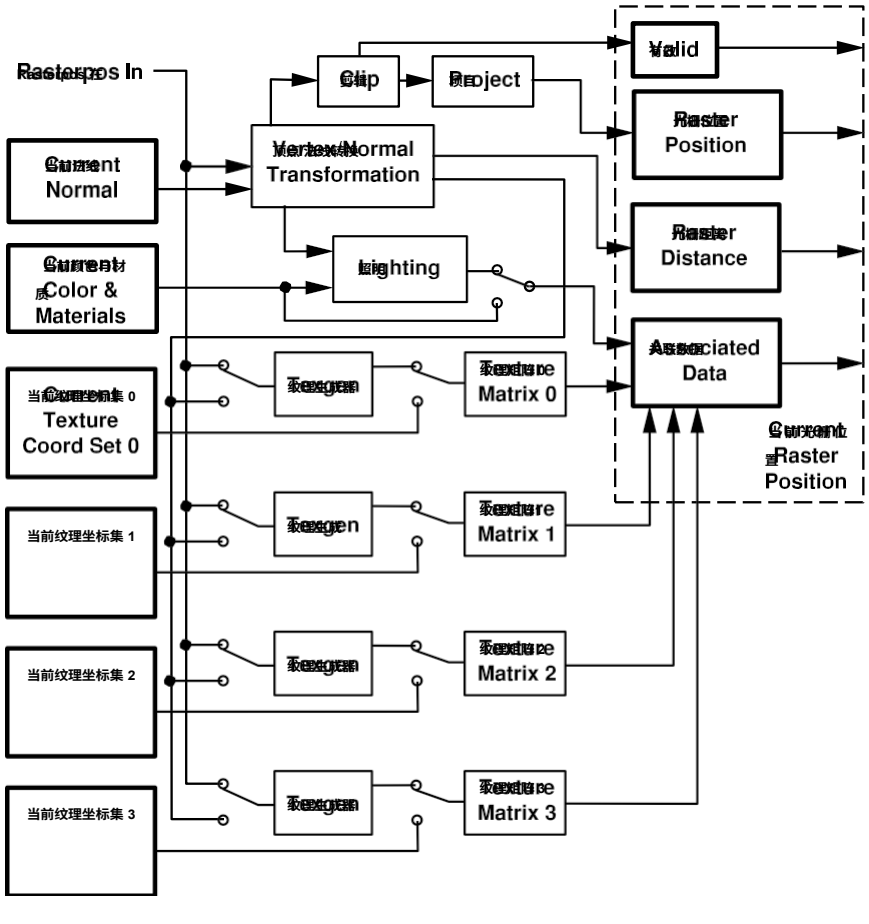


图2.7. 当前光栅位置及其设置方式。图示包含四个纹理单元；但多纹理渲染支持的单元数量可能因具体实现而异。

光栅颜色和辅助颜色分别通过将当前颜色和辅助颜色的每个分量限制在[0, 1]范围内获得。在颜色索引模式下，当前光栅颜色索引被设置为当前颜色索引。当前光栅纹理坐标被设置为当前纹理坐标，并设置有效位。

若雾源值为FOG COORD SRC，则当前光栅距离设置为当前雾坐标值；否则光栅距离设为0。

当前光栅位置需要六个单精度浮点数值来表示其x_w、$y(w)$和$z(w)$窗口坐标，其$w(c)$裁剪坐标，其光栅距离（在光栅处理中用作雾协调），一个有效位，以及四个浮点数值用于存储：$_w$ 和 z_w 窗口坐标，其 w_c 裁剪坐标，其光栅距离（在光栅处理中用作雾坐标），一个有效位，四个浮点值用于存储当前RGBA颜色，四个浮点值用于存储当前RGBA次要颜色，一个浮点值用于存储当前颜色索引，以及每个纹理单元对应的4个浮点值用于纹理坐标。 初始状态下，所有坐标与纹理坐标均为(0, 0, 0, 1)，视点坐标距离为0，雾化坐标为0，有效位被设置，关联RGBA颜色为(1, 1, 1, 1)，关联RGBA次要颜色为(0, 0, 0, 1)，关联颜色索引颜色为1。 在RGBA模式下，关联颜色索引始终保持初始值；在颜色索引模式下，RGBA颜色和次要颜色始终保持初始值。

2.14 颜色与着色

图2.8和图2.9展示了光栅化前RGBA颜色与颜色索引的处理流程。输入颜色以多种格式呈现。表2.9总结了根据调用何种版本的Color命令指定颜色分量时，R、G、B和A分量所经历的转换过程。受限于精度限制，部分转换后的值无法精确呈现。 在颜色索引模式下，单值颜色索引不会进行映射。

接下来，若启用光照效果，则生成颜色索引或主次颜色。若禁用光照，则后续处理将使用当前颜色索引或当前颜色（主色）及当前次色。光照处理后，RGBA颜色将被限制在[0, 1]范围内。 颜色索引将转换为定点数值，随后对其整数部分进行掩码处理（参见第2.14.6节）。经过钳位或掩码处理后，若原始图形采用*平面着色*，则表示所有顶点将采用相同颜色。最后，若原始图形经过裁剪，则必须在裁剪引入或修改的顶点处计算颜色（及纹理坐标）。

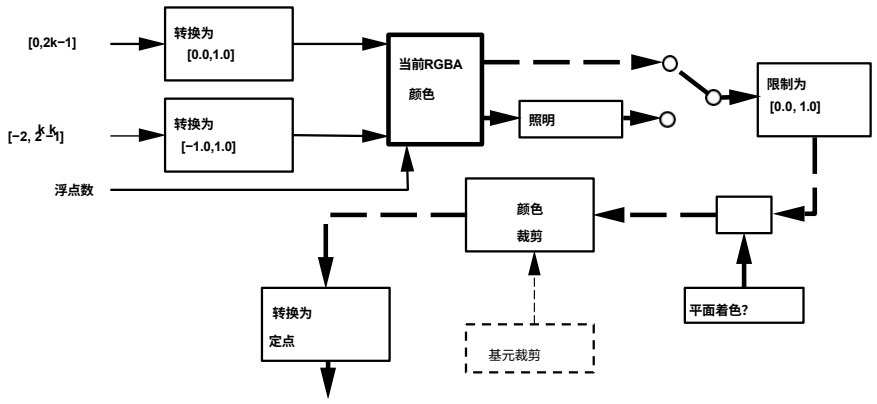


图2.8. RGBA颜色处理流程。粗虚线表示主顶点颜色与次顶点颜色，两者处理方式相同。*k值*解释详见表2.9。

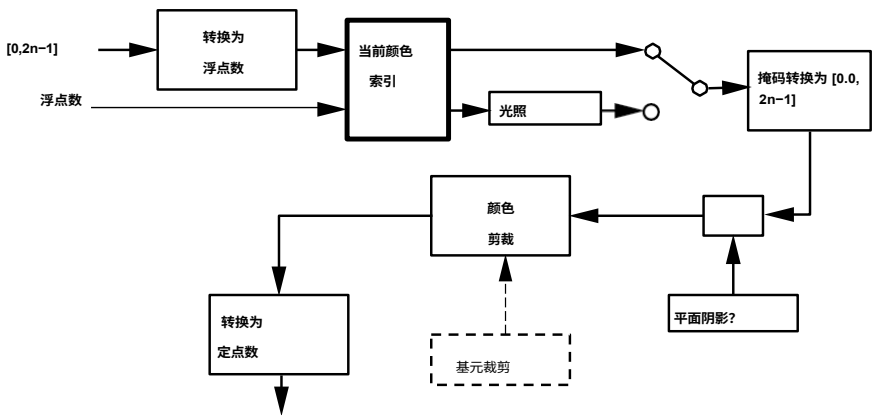


图2.9. 色彩索引处理流程。n为色彩索引的位数。

GL类型	转换
ubyte	$c/(2^8 - 1)$
字节	$(2c + 1)/(2^{(8-1)})$
ushort	$c/(2^{16} - 1)$
短整型	$(2c + 1)/(2^{16} - 1)$
uint	$c/(2^{32} - 1)$
整数	$(2c + 1)/(2^{32} - 1)$
float	c
double	c

表 2.9： 组件转换。 颜色、法线和深度分量(c)通过本表公式转换为内部浮点表示(f)。所有运算均采用内部浮点格式进行。这些转换适用于GL命令参数指定的分量及像素数据中的分量。即使GL数据类型的实现范围大于最低要求范围，公式仍保持不变。（参见表2.2）

2.14.1 光照

GL照明为发送至GL的每个顶点计算颜色。具体实现方式是将客户端指定的照明模型定义的方程应用于一组参数，该参数集可包含顶点坐标、一个或多个光源的坐标、当前法线，以及定义光源特性和当前材质的参数。以下讨论假设GL处于RGBA模式。（颜色索引照明详见第2.14.5节。）

通过通用启用或禁用命令配合符号值 `LIGHTING` 可开启或关闭光照。若光照关闭，则当前颜色与当前辅助颜色将分别赋予顶点主色与辅助色；若光照开启，则根据当前光照参数计算出的颜色将赋予顶点主色与辅助色。

光照操作

灯光参数分为五种类型：颜色、位置、方向、实数或布尔值。颜色参数由四个浮点数值组成，分别对应R、G、B和A通道，顺序依次为R→G→B→A。这些参数的取值范围不受限制。位置参数包含四个浮点坐标（x、y、z和w），用于指定物体坐标系中的位置（w可为零）。

指向由x、y和z给定方向的无限远点）。方向参数由三个浮点坐标（x、y和z）组成，用于指定物体坐标系中的方向。实数参数则是一个浮点值。 各类参数值及其类型汇总于表2.10。若参数值超出表中给定范围，则光照计算结果未定义。

存在n个光源，索引为$i = 0, \ldots, n-1$（n为实现相关的最大值，必须至少为8）。需注意$\mathbf{d}_{cli}$和$\mathbf{s}_{cli}$的默认值在$i = 0$与$i > 0$时存在差异。

在具体说明照明如何计算颜色之前，我们先引入简化相关表达式的运算符和符号。若$\mathbf{c}_1$和$\mathbf{c}_2$是无透明度的颜色，其中$c_{(1)} = (r_1, g_1, b_{(1)})$且$c_2 = (r_2, g_2, b_{(2)})$，则定义$c_{(1)} \approx c_{(2)} = (r_{(1)}, g_{(1)}, b_{(1)}) \approx (r_{(2)}, g_{(2)}, b_{(2)})$。$\mathbf{c}_1 \mathbf{c}_2 = (r_1 r_2, g_1 g_2, b_1 b_2)$，则定义$\mathbf{c}_1 \mathbf{c}_2 = (r_1 r_2, g_1 g_2, b_1 b_2)$。颜色相加通过各分量相加实现。 颜色与标量相乘即表示将每个分量与该标量相乘。若$\mathbf{d}_1$和$\mathbf{d}_2$为方向向量，则定义：

$$\mathbf{d}_1 \odot \mathbf{d}_2 = \max\{\mathbf{d}_1 \cdot \mathbf{d}_2, 0\}。$$

（方向被视为具有三个坐标。）若$\mathbf{P}_1$和$\mathbf{P}_2$是（齐次、具有四个坐标）点，则$\overrightarrow{\mathbf{P}_1\mathbf{P}_2}$为从$\mathbf{P}_1$指向P(2)的单位向量。到点$\mathbf{P}_2$的位置。注意，若点$\mathbf{P}_2$的w坐标为零而点$\mathbf{P}_1$的w坐标不为零，则$\overrightarrow{\mathbf{P}_1\mathbf{P}_2}$即为对应于点P(2)的x、y、z坐标所指定方向的单位向量；若点$\mathbf{P}_{(1)}$的w坐标为零而点$\mathbf{P}_2$的w坐标不为零，则$\overrightarrow{\mathbf{P}_1\mathbf{P}_2}$即为对应于点$P_{(2)}$的x、y、z坐标所指定方向的单位向量。
和z坐标所指定的方向；若$\mathbf{P}_1$的w坐标为零而$\mathbf{P}_2$的w坐标非零，则$-P-(1)-P \rightarrow (2)$对应于$\mathbf{P}_2$的x、y、z坐标所指定的方向。

坐标系中，$\overrightarrow{\mathbf{P}_1\mathbf{P}_2}$的单位向量是该对应方向的负值。若$\mathbf{P}_1$与$\mathbf{P}_2$的w坐标均为零，则

$\overrightarrow{\mathbf{P}_1\mathbf{P}_2}$即为通过归一化对应于$\mathbf{P}_2 - \mathbf{P}_{(1)}$的方向。

若d为任意方向，则令$\hat{d}$为指向d方向的单位向量。令$\|\mathbf{P}_1\mathbf{P}_2\|$为$\mathbf{P}_1$与$\mathbf{P}_2$之间的距离。最后，令$\mathbf{V}$为被照亮顶点对应的点，$\mathbf{n}$为对应的法线。令$\mathbf{P}_e$为视点（在视点坐标系中为(0, 0, 0, 1)）。

光照在顶点产生两种颜色：主色$\mathbf{c}_{pri}$和次色$\mathbf{c}_{sec}$。$\mathbf{c}_{pri}$与$\mathbf{c}_{sec}$的数值取决于光照模型颜色控制参数c_{es}。

若c_{es}= 单一颜色，则计算$\mathbf{c}_{pri}$和$\mathbf{c}_{sec}$的方程为

$$c_{pri} = e_{cm} + \mathbf{a}_{cm} \times \mathbf{a}_{cs}$$

参数	类型	默认值	描述
材料参数			
acm	颜色	(0.2, 0.2, 0.2, 1.0)	材质的环境色
dcm	颜色	(0.8, 0.8, 0.8, 1.0)	材质漫反射颜色
scm	颜色	(0.0, 0.0, 0.0, 1.0)	材料的镜面色
ecm	颜色	(0.0, 0.0, 0.0, 1.0)	材料的发射颜色
srm	真实	0.0	镜面反射指数 (范围：[0.0, 128.0])
a_m	实数	0.0	环境色指数
d_m	真实	1.0	漫反射色指数
s_m	实数	1.0	镜面色指数
光源参数			
acli	颜色	(0.0, 0.0, 0.0, 1.0)	光的环境强度
$\mathbf{d}_{cli}$ $(i = 0)$ $\mathbf{d}_{cli}$ $(i > 0)$	颜色 颜色	(1.0, 1.0, 1.0, 1.0) (0.0, 0.0, 0.0, 1.0)	光的漫反射强度 0 光的漫反射强度
$\mathbf{s}_{cli}$ $(i = 0)$ $\mathbf{s}_{cli}$ $(i > 0)$	颜色 颜色	(1.0, 1.0, 1.0, 1.0) (0.0, 0.0, 0.0, 1.0)	镜面反射强度 0 镜面光强度 *i*
Ppli	位置	(0.0, 0.0, 1.0, 0.0)	光点的位置
sdli	方向	(0.0, 0.0, −1.0)	光线 *i* 的聚光灯方向
srli	实数	0.0	灯光 *i* 的聚光灯指数 (范围：[0.0, 128.0])
crli	实数	180.0	聚光灯截止角（针对光线 *i*） (范围：[0.0, 90.0], 180.0)
k0i	实数	1.0	恒定衰减因子 光i（范围：[0.0, ∞))
k1i	实数	0.0	线性衰减因子 光i（范围：[0.0, ∞))
k2i	实数	0.0	二次衰减因子，适用于 光 i (范围：[0.0, ∞))
照明模型参数			
acs	颜色	(0.2, 0.2, 0.2, 1.0)	场景环境色
vbs	布尔值	FALSE	查看器假定被是位于 (0, 0, 0) 在 eye 坐标系中 (TRUE) 或 (0, 0, ∞) (FALSE)
ces	enum	单色	控制颜色的计算
tbs	布尔	FALSE	使用双面照明模式

表 2.10： 照明参数汇总。各颜色分量的取值范围为 (−∞, +∞)。

$$+ \sum_{i=0}^{n-1} (att_i)(spot_i)\,[\mathbf{a}_{cm} * \mathbf{a}_{cli} + (\mathbf{n} \odot \vec{\mathbf{P}}_{pli})\,\mathbf{d}_{厘米} * \mathbf{d}_{cli} + (f_i)\,(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}} \mathbf{s}_{cm} * \mathbf{s}_{cli}]$$

$$\mathbf{c}_{秒} = (0, 0, 0, 1)$$

若 c_{es} = 独立镜面颜色，则

$$c_{pri} = e_{cm} + a_{cm} * a_{cs} + \sum_{i=0}^{n-1} (att_i)(spot_i)\,[\mathbf{a}_{cm} * \mathbf{a}_{cli} + (\mathbf{n} \odot \vec{\mathbf{P}}_{pli})\,\mathbf{d}_{cm} * \mathbf{d}_{cli}]$$

$$c_{sec} = \sum_{i=0}^{n-1} (att_i)(spot_i)(f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}} \mathbf{s}_{cm} * \mathbf{s}_{cli}$$

其中

$$f_i = \begin{cases} 1, & \mathbf{n} \odot \vec{\mathbf{P}}_{pli} = 0, \\ 0, & \text{否则}, \end{cases} \tag{2.2}$$

$$\mathbf{h}_i = \begin{cases} \vec{\mathbf{P}}_{pli} + \vec{\mathbf{P}}_e, & v_{bs} = \texttt{TRUE}, \\ \vec{\mathbf{P}}_{pli} + (0 \quad 0 \quad 1)^T, & v_{bs} = \texttt{FALSE}, \end{cases} \tag{2.3}$$

$$att_i = \begin{cases} \dfrac{1}{k_{0i} + k_{1i} \|VP_{pli}\| + k_{2i} \|VP_{pli}\|^2}, & \text{若} \mathbf{P}_{pli}\text{'s } w \neq 0, \\ 1.0, & \text{否则。} \end{cases} \tag{2.4}$$

$$spot_i = \begin{cases} (\vec{\mathbf{P}}_{pli} \mathbf{V} \odot \hat{\mathbf{s}}_{dli})^{s_{rli}}, & c_{rli} \neq 180.0, \vec{\mathbf{P}}_{pli} \mathbf{V} \odot \hat{\mathbf{s}}_{dli} \geq \cos(c_{rli}), \\ 0.0, & c_{rli} \neq 180.0, \vec{\mathbf{P}}_{pli} \mathbf{V} \odot \hat{\mathbf{s}}_{dli} < \cos(c_{rli}), \\ 1.0, & c_{rli} = 180.0. \end{cases} \tag{2.5}$$

所有计算均在眼坐标系中进行。

照明产生的A值是**与**$\mathbf{d}_{cm}$相关的α值。A始终与**原色**$\mathbf{c}_{prl}$相关联；$\mathbf{c}_{sec}$的α分量始终为1。

若*v*的w_e坐标（w为眼坐标系中的坐标）为零，则照明结果未定义。

为零时，照明结果将不确定。

照明可采用*双面*模式运行（t_{bs}= TRUE），*其中前侧*颜色通过一组材质参数（*前侧材质*）计算，*后侧*颜色则通过另一组材质参数（*后侧材质*）计算。此二次计算将**n**替换**为n**。若t_{bs}= FALSE，则后侧颜色与前侧颜色均采用前侧材质**与n计算**所得的颜色。

此外，顶点着色器可运行于双面颜色模式。当顶点着色器处于活动状态时，其可计算前表面与后表面颜色，并写入`gl FrontColor`、`gl BackColor`、gl FrontSecondaryColor及`gl BackSecondaryColor`输出变量。若启用`VERTEX PROGRAM TWO SIDE`指令，GL将按下述规则在前表面与后表面颜色间进行选择。否则始终选用正面颜色输出。双面颜色模式通过调用**Enable**或**Disable**并传入符号值`VERTEX PROGRAM TWO SIDE`来启用或禁用。

背面与正面颜色的选择取决于原始图元的类型。

被点亮的顶点属于某个部分。若基本图形为点或线段，则始终选择前视颜色。若为多边形，则选择依据基于窗口坐标计算的多边形（裁剪或未裁剪）带符号面积的正负值。计算该面积的一种方法是

$$a = \frac{1}{2}\sum_{i=0}^{n-1} x_w^i y_w^{i\oplus 1} - x_w^{i\oplus 1} y_w^i \tag{2.6}$$

其中 x_w^i 和 y_w^i 分别是第 i 个顶点的 *x* 坐标和 *y* 坐标。

1 表示第 i 个顶点（顶点编号从零开始）的 x 和 y 窗口坐标，*其中* i 1 表示 $(i + 1) \bmod n$。该值的符号解释由

```
void FrontFace( enum dir );
```

将dir设为`CCW`（对应于窗口坐标系中投影多边形的逆时针方向）表示：若 0，则多边形每个顶点的颜色采用为该顶点计算的背景色；若$a > 0$，则选择前景色。*若*dir为`CW`，则上述不等式中的a将替换为a。这需要1位状态位；初始时该位指示`CCW`方向。

2.14.2 照明参数规范

照明参数分为三类：材料参数、光源参数和照明模型参数（见表2.10）。照明参数集通过以下方式指定：

```
void Material{if}( enum face, enum pname, T param );void Material{if}v( enum face, enum pname, T params );void Light{if}( enum light, enum pname, T param );void Light{if}v( enum light, enum pname, T params );void LightModel{if}( enum pname, T param );
void 光照模型{if}v( enum 光照名称, T 光照参数);
```

pname 是表示需设置参数的符号常量（参见表 2.11）。在向量版本的命令中，*params* 是指向需设置参数的值组的指针。所指向的值组数量取决于待设置的参数类型。 在非向量版本中，*param* 是用于设置单值参数的数值。（若 *param* 对应多值参数，则会导致 INVALID ENUM 错误。）对于 **Material** 命令，*face* 必须为 FRONT、BACK 或 FRONT AND BACK 之一，分别表示应设置前表面材料、后表面材料或两者的属性 *名称*。 在**Light命令**中，*light*是形式*为LIGHTi的*符号常量，表示需为*第*盏灯设置指定参数。常量遵循*LIGHTi* = LIGHT0 + *i*的规则。

表2.11列出了三个参数组中预定义常量名称与照明方程中名称的对应关系，以及每个参数组必须指定的值的数量。 材料**和光源中**指定的颜色参数将转换为浮点值（若以整数形式指定），具体转换规则参照表2.9中带符号整数的转换方式。若指定的照明参数超出表2.10给定的允许范围，则触发INVALID VALUE错误。（符号“ ”表示该类型参数的最大可表示数值。）

可在Begin/End对内部通过**调用Material**修改材质属性。但当顶点着色器激活时，此类属性变更无法保证立即更新材质参数（详见表2.11），需待后续**End**指令生效。

当前模型视图矩阵将应用于特定光源的**Light**标识位置参数（当该位置被指定时）。这些变换后的数值即用于光照方程的计算值。

当仅使用模型视图矩阵左上角3x3部分指定聚光灯方向时，其方向将发生转变。即若$\mathbf{M}_u$为左上角3x3矩阵

参数	名称	数值数量
材质参数（**材质**）		
acm	环境	4
dcm	漫反射	4
acm, *dcm*	环境光与漫反射光	4
scm	镜面	4
ecm	EMISSION	4
srm	SHININESS	1
a_m , d_m , s_m	色索引	3
光源参数（**光**）		
acli	环境	4
dcli	漫射	4
scli	镜面反射	4
Ppli	位置	4
sdli	SPOT DIRECTION	3
srli	SPOT EXPONENT	1
crli	SPOT CUTOFF	1
k_0	恒定衰减	1
k_1	线性衰减	1
k_2	二次衰减	1
照明模型参数 (**LightModel**)		
acs	光照模型环境光	4
vbs	光照模型局部观察者	1
tbs	LIGHT MODEL TWO SIDE	1
ces	轻型模型色彩控制	1

表 2.11: 对应关系 中的 照明 参数 符号 到 名称。

AMBIENT AND DIFFUSE 用于将 $\mathbf{a}_{cm}$和 $\mathbf{d}_{cm\ 设置}$为相同值。

若矩阵取自当前模型视图矩阵 M，则聚光灯方向

$$\begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix}$$

将变换为

$$\begin{bmatrix} d'_x \\ d'_y \\ d'_z \end{bmatrix} = M_u \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix}。$$

通过调用**Enable**或**Disable**函数并传入符号值*LIGHTi*（*i*取值范围为0至n-1，其中n为实现相关的光源数量）来启用或禁用单个光源。若禁用*光源*，则照明方程中的第i项将从求和项中有效移除。

2.14.3 ColorMaterial

可将一种或多种材质属性附加至当前颜色，使其持续追踪颜色的各色度值。通过调用**Enable**或**Disable**并传入符号值COLORMATERIAL来启用或禁用此行为。

控制这些模式选择的命令是

```
void ColorMaterial( enum face, enum mode );
```

face 取值为 FRONT、BACK 或 FRONT AND BACK，分别表示当前颜色仅影响正面材质、背面材质或两者同时影响。*mode* 取值为 EMISSION、AMBIENT、DIFFUSE、SPECULAR 或 AMBIENT AND DIFFUSE，指定哪些材质属性会追踪当前颜色。若模式为EMISSION（发射）、AMBIENT（环境光）、DIFFUSE（漫反射）或SPECULAR（镜面反射），则分别由$\mathbf{e}_{cm}$、$\mathbf{a}_{cm}$、$\mathbf{d}_{cm}$或$\mathbf{s}_{(cm)}$的值追踪当前颜色。若模式为AMBIENT AND DIFFUSE（环境光与漫反射），则$\mathbf{a}_{cm}$和$\mathbf{d}_{cm}$共同追踪当前颜色。对材质属性的替换是永久性的；被替换的值将持续生效，直至发送新颜色或在当前未启用**COLOR** MATERIAL覆盖该特定值时设置新材质值。当COLOR MATERIAL启用时，指定的参数将始终追踪当前颜色。例如调用

```
ColorMaterial (FRONT, AMBIENT)
```

当启用颜色材质时，将当前颜色的值赋予前置**材质**$\mathbf{a}_{cm}$。

材质属性可在Begin/End对内部通过启用**ColorMaterial**模式并调用**Color**函数间接修改。但当顶点

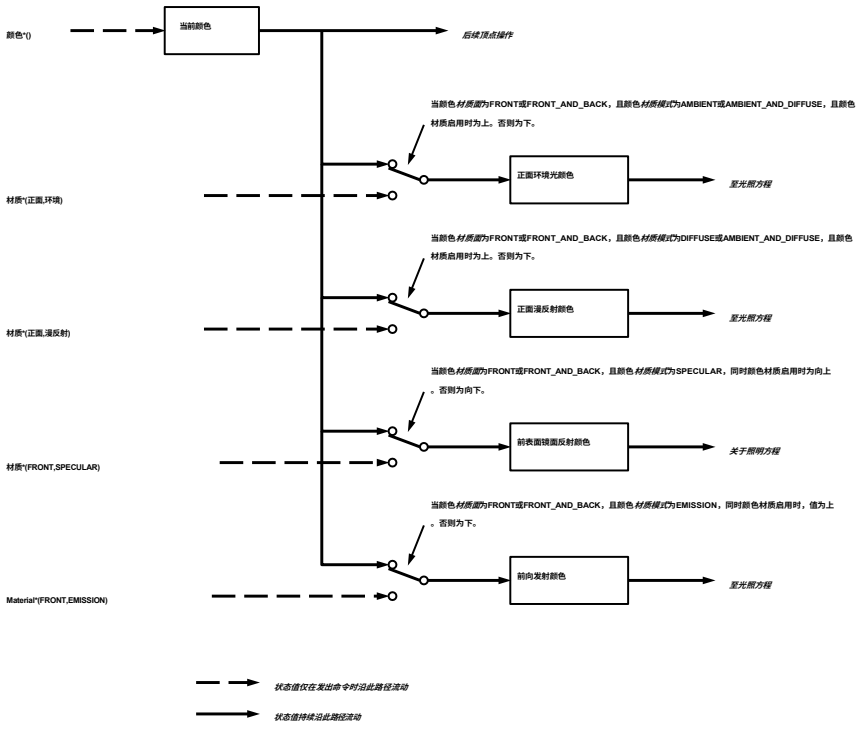


图2.10. ColorMaterial操作示意。当**Color**Material启用且模式正确时，材质属性会持续根据当前颜色更新。本图仅展示前表面材质属性，后表面材质处理方式相同，但需将*面方向*设为`BACK`或`FRONT AND BACK`。

着色器处于活动状态时，此类属性更改无法保证更新表 2.11 中定义的材质参数，直到下一个 **End** 命令。

2.14.4 照明状态

照明所需的状态包含所有照明参数（前/后材质参数、照明模型参数及至少8组光源参数），一个位用于指示是否需计算与正面颜色不同的背面颜色，至少8位用于指示哪些光源处于启用状态，一个五值变量用于指示当前的ColorMaterial模式，一个位用于指示COLOR MATERIAL是否启用，以及一个单一位用于指示照明是否启用。初始状态下，所有照明参数均采用默认值。 不进行背光颜色评估，**材质**模式为"正反面+环境光+漫反射"，且灯光与COLOR MATERIAL均处于禁用状态。

2.14.5 颜色索引照明

在颜色指数模式下采用简化照明计算，该模式使用了控制RGBA照明的多数参数，但不包含任何RGBA材质参数。首先，*光i的*RGBA漫反射强度与镜面反射强度（分别记为$\mathbf{d}_{cli}$**和**$\mathbf{s}_{cli}$）决定了颜色指数模式下的漫反射与镜面反射光*强度*d_{li}*和*s_{li}，其计算公式为：

$$d_{li} = 0.30R(\mathbf{d}_{cli}) + 0.59G(\mathbf{d}_{cli}) + 0.11B(\mathbf{d}_{(cli)}))$$

d(li) = 0.30R(d(cli))
+ 0.59

$$s_{li} = 0.30R(\mathbf{s}_{cli}) + 0.59G(\mathbf{s}_{cli}) + 0.11B(\mathbf{s}_{(cli)}))。$$

$R(\mathbf{x})$表示**颜色x**的红（R）分量，$G(\mathbf{x})$*和*$B(\mathbf{x})$的含义类似。

接下来，令

$$s = \sum_{i=0}^{n} (att_i)(spot_i)(s_{li})(f_i)(\mathbf{n} \cdot \hat{h}_i)^{srm}$$

其中 att_i和 $spot_i$ 分别由方程 2.4 和 2.5 给出，f_i和 $\hat{h}_i$ 分别由方程 2.2 和 2.3 给出。令 $s' = \min s, 1$。最后令

$$d = \sum_{i=0}^{n} (att_i)(spot_i)(d_{li})(\mathbf{n} \cdot \overrightarrow{\mathbf{V}}_{pli}). \quad \{ \quad \}$$

然后颜色索引照明产生一个值 c，由以下公式给出：

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

最终颜色索引为

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

参数a_m、d_m*和*s_m为材料属性，其定义详见表2.10和表2.11。所有环境光强度均已整合*至*$a_{(m)}$中。与RGBA照明类似，禁用光源将导致对应项从求和表达式中省略。t_{bs}的解释以及前/后颜色的计算方式，均遵循已针对RGBA照明描述的流程。

a_m、d_m和s_m的数值通过 **Material** 设置，采用 *pname* 为 `COLOR INDEXES` 的方式。它们的初始值分别为 0、1 和 1。附加状态包含三个浮点数值。这些数值对 RGBA 照明不产生影响。

2.14.6 钳位或遮罩

无论是否启用照明，主色与次色所有色度分量均会被限制在[0, 1]范围内。

对于颜色索引，首先将其转换为固定小数点数，二进制小数点右侧保留未指定位数；选择最接近的固定小数点值。 随后，二进制小数点右侧位保持不变，整数部分则与掩码 $2^{n(n)}$ 1 进行位与运算（其中 *n* 为颜色索引缓冲区中单色位数，缓冲区详见第4章）。

2.14.7 平面着色

原始图形可采用*平面着*色，即所有顶点被赋予相同的颜色索引或相同的原色与辅色。这些颜色源自生成该原始图形的顶点。对于点图形，这些颜色与该点关联；对于线段图形，则取自线段第二个（终点）顶点的颜色；对于多边形图形，则根据多边形生成方式选取特定顶点的颜色。表2.12总结了具体情况。 对于线段，这些颜色来自线段的第二个（末端）顶点。对于多边形，这些颜色取决于多边形的生成方式，可能来自选定的顶点。表 2.12 总结了各种可能性。

平面着色由

```
void ShadeModel( enum mode );
```

mode 值必须为符号常量 `SMOOTH` 或 `FLAT` 之一。若 *mode* 为 `SMOOTH`（初始状态），则顶点颜色将单独处理；若 *mode* 为 `FLAT`，则开启平面着色。因此 **ShadeModel** 仅需一位状态。

多边形的基元类型	顶点
单一多边形（$i \equiv 1$）	1
三角带	$i+2$
三角扇区	$i+2$
独立三角形	$3i$
四边形条带	$2i+2$
独立四边形	$4i$

表 2.12： 多边形平面着色颜色选择。用于对由指定**开始**/结束类型生成的第 i 个多边形进行平面着色的颜色，来源于指定顶点被指定时生效的当前颜色（若禁用光照）。若启用光照，则颜色由指定顶点的光照效果生成。顶点编号为 1 至 n，其中 n 是**开始**/**结束**对之间顶点的数量。

2.14.8 颜色与关联数据裁剪

经过光照、钳位或遮罩处理及可能的平面着色后，颜色将进行裁剪。位于裁剪体积内的顶点关联颜色不受裁剪影响。但若原始图形被裁剪，则裁剪产生的顶点所分配的颜色即为裁剪后的颜色。

设未截断边上两个顶点 $\mathbf{P}_1$和 $\mathbf{P}_2$ 分配的颜色分别为 $\mathbf{c}_1$和 $\mathbf{c}_2$。截断点 $\mathbf{P}$ 的 t 值（参见第 2.12 节）用于计算与 $\mathbf{P}$ 关联的颜色，其表达式为：

$$\mathbf{c} = t\mathbf{c}_1 + (1 - t)\mathbf{c}_2.$$

（对于索引颜色，将颜色与标量相乘意味着将索引与标量相乘。对于RGBA颜色，则意味着将R、G、B和A各自与标量相乘。主色与次色均采用相同处理方式。）多边形裁剪可能在裁剪体积边界沿线产生裁剪顶点。 该情况通过以下机制处理：多边形裁剪每次仅针对裁剪体边界的一个平面进行裁剪。颜色裁剪采用相同方式，因此裁剪点始终出现在多边形边（可能已裁剪）与裁剪体边界的交点处。

纹理和雾坐标、顶点着色器变量（第2.15.3节）以及按顶点计算的点大小，在裁剪基元时也必须进行裁剪。该方法与颜色裁剪完全相同。

2.14.9 最终颜色处理

对于RGBA颜色，每个颜色分量（取值范围为[0, 1]）通过四舍五入转换为m位的固定点数值。 我们假设所使用的固定点表示法将每个值表示为k/(2^m 1)，其中k 0, 1, ..., 2^m 1，即k的二进制表示为全1字符串（例如1.0在二进制中表示为全1字符串）。 m必须至少等于帧缓冲区对应组件的位数。若帧缓冲区不含A组件，或仅含1位A组件，则A组件的m值必须至少为2。颜色索引经四舍五入转换为固定小数点值，其位数至少等于帧缓冲区颜色索引部分的位数。

因为若干形式为 k/(2^m 1) 的数值可能无法精确表示为

对于有限精度浮点量，我们对RGBA组件的定点转换提出额外要求。假设灯光功能已禁用，顶点关联的颜色未被裁剪，且使用**Colorub**、**Colorus**或**Colorui**中任一方式指定该颜色。 当这些条件满足时，RGBA分量必须转换为与Color命令中指定分量相匹配的值：若m小于指定分量的位数b，则转换值必须等于指定值的最高m位；否则，转换值的最高b位必须等于指定值。

2.15 顶点着色器

第2.11至2.14节所述的操作序列是一种处理顶点数据的固定功能方法。应用程序可通过*顶点着色器*更通用地描述对顶点值及其关联数据执行的操作。

顶点着色器是一组字符串数组，包含针对每个处理顶点执行的操作源代码。顶点着色器使用的语言在《OpenGL着色语言规范》中有详细说明。

使用顶点着色器时，需先将着色器源代码加载至*着色器对象*并*进行编译*。随后将一个或多个顶点着色器对象附加至*程序对象*。接着对程序对象*进行链接操作*，该过程会将附加至程序的所有编译着色器对象生成可执行代码。当链接后的程序对象作为当前程序对象使用时，其包含的顶点着色器可执行代码将用于处理顶点。

除了顶点着色器外，还可创建、编译并链接*片段着色器*至程序对象。片段着色器影响光栅化过程中的片段处理，其具体说明详见第3.11节。单个程序对象可同时包含顶点着色器与片段着色器。

，具体说明详见第3.11节。单个程序对象可同时包含顶点着色器和片段着色器。

当前使用的程序对象若包含顶点着色器，则该顶点着色器被视为*活动*状态并用于处理顶点。若程序对象未包含顶点着色器，或当前未使用任何程序对象，则改用固定功能模式处理顶点。

2.15.1 着色器对象

构成程序的源代码由可编程阶段执行，该程序封装在一个或多个*着色器对象*中。

着色器对象的命名空间为无符号整数，其中零保留给GL使用。该命名空间与程序对象共享。 以下章节定义了通过名称操作着色器和程序对象的命令。接受着色器或程序对象名称的命令，若提供的名称既非着色器也非程序对象名称，则会生成错误INVALID VALUE；若提供的名称标识的对象类型不符合预期，则会生成错误INVALID OPERATION。

创建着色器对象时使用命令

```
uint CreateShader( enum type );
```

着色器对象在创建时为空。*type*参数指定要创建的着色器对象类型。对于顶点着色器，*type*必须为VERTEX SHADER。返回一个非零名称，可用于引用该着色器对象。若发生错误，则返回零。

命令

```
void ShaderSource( uint shader, sizei count, const char **string, const int
    *length );
```

将源代码加载到*名为shader的*着色器对象中。*string*是一个*包含count*个指针的数组，这些指针指向可选的以空字符结尾的字符串，这些字符串共同构成源代码。*length*参数是一个数组，包含每个字符串的字符数（即字符串长度）。如果*length数组*中的某个元素为负值，则对应的字符串以空字符结尾。如果length为NULL，则string参数中的所有字符串均视为以空字符结尾。 ShaderSource命令*将着色器的*源代码设置为*字符串*数组中的文本字符串。*若着色*器先前已加载源代码，则现有源代码将被完全替换。传递的任何长度值均不包含空终止符的计数。

加载到着色器对象中的字符串应构成符合OpenGL着色语言规范定义的有效着色器源代码。

着色器源代码加载完成后，可通过以下命令编译着色器对象：

```
void CompileShader( uint shader );
```

每个着色器对象都具有一个布尔状态——编译状态（COMPILE STATUS），该状态会因编译结果而改变。可通过**GetShaderiv函数**查询此状态（参见第6.1.14节）。*若着色器*编译无误且可供使用，该状态将设置为TRUE；否则为FALSE。 编译失败可能由多种原因导致，具体详见《OpenGL着色语言规范》。**若CompileShader**操作失败，先前编译的所有信息将丢失。因此编译失败不会恢复*着色器的*旧状态。

使用**ShaderSource**修改着色器对象的源代码不会改变其编译状态或已编译的着色器代码。

每个着色器对象都包含信息日志，该日志为文本字符串，会在编译过程中被覆盖。可通过 **Get-ShaderInfoLog** 查询此日志以获取编译尝试的详细信息（参见第 6.1.14 节）。

使用以下命令可删除着色器对象

```
void DeleteShader( uint shader );
```

若*着色*器未附加到任何程序对象，则立即删除。否则，*着色器*会被标记为待删除状态，并在不再附加到任何程序对象时被删除。若对象被标记为待删除，其布尔状态位 DELETE STATUS 将被设置为 true。可通过 **GetShaderiv** 查询 DELETE STATUS 的值（参见第 6.1.14 节）。 **DeleteShader**函数将静默忽略零值。

2.15.2 程序对象

用于GL可编程阶段的着色器对象被集合形成*程序对象*。由这些可编程阶段执行的程序称为*可执行程序*。定义可执行程序所需的所有信息均封装在程序对象中。程序对象通过命令创建：

```
uint CreateProgram( void );
```

程序对象创建时为空。该命令返回一个非零名称，可用于引用程序对象。若发生错误，则返回0。

要将着色器对象附加到程序对象，请使用命令

```
void AttachShader( uint program, uint shader );
```

若*着色器*已附加到*程序对象*，则会生成错误“INVALID OPERATION”。

着色器对象可在源代码加载至着色器对象之前，或着色器对象编译之前附加至程序对象。单个程序对象可附加多个同类型着色器对象，单个着色器对象亦可附加至多个程序对象。

要从程序对象中分离着色器对象，请使用命令

```
void DetachShader( uint program, uint shader );
```

若*着色器*未附加到*程序*，则会生成错误“INVALID OPERATION”。*若着色器*已被标记为删除且未附加到任何其他程序对象，则该着色器将被删除。

要使用程序对象中包含的着色器对象，必须先链接该程序对象。命令

```
void LinkProgram( uint program );
```

将链接名为 *program* 的程序对象。每个程序对象都具有一个布尔状态 LINK STATUS，该状态会因链接操作而改变。可通过 **GetProgramiv** 查询此状态（参见第 6.1.14 节）。若创建出有效的可执行程序，该状态将设置为 TRUE，否则为 FALSE。链接失败的原因多种多样，具体详见《OpenGL 着色语言规范》。 若*程序*关联的着色器对象存在未成功编译的情况，或*程序中*使用的活动统一变量/采样器变量数量超出允许范围（参见第2.15.3节），链接操作同样会失败。当**LinkProgram**失败时，该程序对象的先前链接信息将全部丢失。因此链接失败不会恢复*程序*的旧状态。

每个程序对象都包含一个信息日志，该日志会在链接操作后被覆盖。可通过**GetProgramInfoLog函数**查询此信息日志，以获取有关链接操作的更多细节（参见第6.1.14节）。

若生成有效可执行文件，可通过以下命令将其纳入当前渲染状态：

```
void UseProgram( uint program );
```

该命令将把可执行代码安装为当前渲染状态的一部分，前提是程序对象 *program* 包含有效的可执行代码（即已成功链接）。若调用 **UseProgram** 时 *program* 设为 0，则视为图形渲染器无可编程阶段，此时将使用固定功能路径。若 *program* 未成功链接，则生成 `INVALID OPERATION` 错误且当前渲染状态保持不变。

在程序对象处于使用状态时，应用程序可自由修改关联的着色器对象、编译关联的着色器对象、附加额外的着色器对象以及分离着色器对象。这些操作不会影响程序对象的链接状态或可执行代码。

若正在使用的程序对象重新链接成功，且该对象因先前调用**UseProgram**而处于使用状态，则LinkProgram命令将把生成的可执行代码安装为当前渲染状态的一部分。

若正在使用的程序对象重新链接失败，其链接状态将设置为`FALSE`，但现有可执行文件及关联状态仍将保留在当前渲染状态中，直至后续**调用UseProgram**将其移出使用状态。此类程序被移出使用状态后，除非成功重新链接，否则无法再次成为当前渲染状态的一部分。

程序对象可通过以下命令删除

```
void DeleteProgram( uint program );
```

如果*程序*不是任何GL上下文的当前程序，则立即删除该程序。否则，*程序*会被标记为待删除状态，并在不再是任何上下文的当前程序时被删除。当程序对象被删除时，所有附加在其上的着色器对象都会被分离。**DeleteProgram**将静默忽略零值。

2.15.3 着色器变量

顶点着色器在执行过程中可引用多个变量。*顶点属性*是第2.7节中定义的每个顶点对应的值。*统一变量*则是程序级变量，在程序执行期间保持恒定。 *采样*器是用于纹理映射的特殊统一变量（参见第3.8节）。*变量变量*存储顶点着色器执行结果，供后续渲染管线使用。后续章节将分别阐述这些变量类型。

顶点属性

顶点着色器可访问内置顶点属性变量，这些变量对应于**由Vertex、Normal、Color等**命令设置的每顶点状态。顶点着色器还可定义命名属性变量，这些变量与**通过VertexAttrib***设置的通用顶点属性绑定。该绑定可在程序链接前由应用程序指定，也可在程序链接时由GL自动分配。

当声明为浮点型、vec2、vec3 或 vec4 的属性变量绑定到泛型属性索引 *i* 时，其值分别取自泛型属性 *i* 的 *x*、(*x*, *y*)、(*x*, *y*, *z*) 或 (*x*, *y*, *z*, *w*) 分量。 当属性变量声明为 mat2 时，其矩阵列取自通用属性 *i* 和 *i* + 1 的 (*x*, *y*) 分量。当属性变量声明为 mat3 时，其矩阵列取自通用属性 *i* 至 *i* + 2 的 (*x*, *y*, *z*) 分量。 当属性变量声明为 mat4 时，其矩阵列取自通用属性 *i* 至 *i* + 3 的 (*x*, *y*, *z*, *w*) 组件。

属性变量（无论是常规属性还是通用属性）若被编译器和链接器判定为在着色器执行时可能被访问，则视为活动属性。在顶点着色器中声明但从未使用的属性变量不计入限制。当编译器和链接器无法做出明确判定时，该属性将被视为活动属性。 若活动通用属性与活动常规属性的总数超过MAX VERTEX ATTRIBS限制，程序对象将链接失败。

要确定程序使用的活动顶点属性集及其类型，请使用以下命令：

```
void GetActiveAttrib( uint program, uint index, sizei bufSize, sizei *length,
    int *size, enum *type, char *name );
```

此命令提供由*索引*选定的属性信息。*索引*值为0时选定首个活动属性，*索引*值为ACTIVE ATTRIBUTES 1时选定最后一个活动属性。ACTIVE ATTRIBUTES的值可通过**GetProgramiv**查询（参见第6.1.14节）。 若索引值大于或等于活动属性数值，将触发无效值错误。需注意*索引*值仅用于标识活动属性列表中的成员，与对应变量所绑定的通用属性无关。

参数 *program* 是指曾被执行过 **LinkProgram** 命令的程序对象名称。*该程序*对象不必

链接已成功建立。链接失败可能是因为活动属性数量超过了限制。

所选属性的名称以空字符终止的字符串形式返回*至name变量*。实际写入*name*的字符数（不含空终止符）通过*length*返回。若*length*为NULL，则不返回长度值。*bufSize*指定了可写入*name*的最大字符数（含空终止符）。 返回的属性名称可以是通用属性名称或常规属性名称（以"gl"前缀开头，完整列表参见OpenGL着色语言规范）。*程序中最*长属性名称的长度由ACTIVE ATTRIBUTE MAX LENGTH指定，**可通过GetProgramiv**查询（参见第6.1.14节）。

对于所选属性，其类型将返回至*type字段*。属性的大小将返回至*size字段*。*size字段*中的数值单位与*type字段*返回的类型一致。返回的类型可为FLOAT、FLOAT VEC2、FLOAT VEC3、FLOAT VEC4、FLOAT MAT2、FLOAT MAT3或FLOAT MAT4中的任意一种。

若发生错误，返回参数*length*、*size*、type和*name*将保持不变。

此命令将返回尽可能多的活动属性信息。若无可用信息，则*长度*将设为零，*名称*为空字符串。当链接失败后调用GetActiveAttrib时可能出现此情况。

程序对象成功链接后，可查询属性变量名与索引的绑定关系。命令

```
int GetAttribLocation( uint program, const char *name );
```

*返回*名为 *program* *的*程序对象上次链接时，名为 *name* 的属性变量所绑定的通用属性索引。*name* 必须是空终止字符串。若 *name* 为活动属性且是属性矩阵，则 **GetAttribLocation** 返回该矩阵首列的索引。若 *program* 未成功链接，则生成 INVALID OPERATION 错误。 若 *name* 非活动属性、为常规属性，或发生错误，则返回 -1。

属性变量与通用属性索引的绑定也可显式指定。命令

```
void BindAttribLocation( uint program, uint index, const char *name );
```

指定程序 *program* 中名为 *name* 的属性变量应在下次链接时绑定至通用顶点属性 *index*。若 *name*

此前已被绑定，则其原有绑定关系*将被index*取代。*name*必须为空终止字符串。*若index*值等于或大于MAX VERTEX ATTRIBS，则触发INVALID VALUE错误。**BindAttribLocation**在程序链接前不生效，尤其不会修改已链接程序中活动属性变量的绑定关系。 _ _

内置属性变量会自动绑定到常规属性，且不能被显式绑定。*若*名称以保留字符前缀"gl"开头，将生成错误"INVALID OPERATION"。 _

程序链接时，任何未指定绑定的活动属性 _
通过 **BindAttribLocation** 分配的属性绑定将由图形库自动绑定至顶点属性。此类绑定可通过 **GetAttribLocation** 命令查询。若活动属性变量的分配绑定导致图形库需引用不存在的通用属性（即数量大于或等于 MAX VERTEX ATTRIBS 的属性），则 **LinkProgram** 将失败。 **若BindAttribLocation分配**的属性绑定占用空间不足，导致无法为需多个连续通用属性的活动矩阵属性分配位置，则**程序**链接将失败。此外，当程序对象使用的顶点着色器包含（预处理阶段未移除的）对绑定至通用属性零及常规顶点位置（glVertex）的属性变量赋值时，**程序链接**同样会失败。

在任何顶点着色器对象绑定到程序对象之前，都可以调用**BindAttribLocation**。因此允许将任意名称（除以"gl"开头的名称外）绑定到索引，包括那些从未在任何顶点着色器对象中用作属性的名称。
因此允许将任意名称（除以"gl"开头的名称外）绑定至索引，包括从未在顶点着色器对象中用作属性的名称。针对不存在或未激活的属性变量所做的绑定将被忽略。

发送到通用属性索引 *i* 的通用属性值与常规属性值同属当前状态的一部分。若新程序对象被激活，GL 将追踪这些值，确保新程序对象中同样绑定到索引 *i* 的属性能观察到相同值。

应用程序可将多个属性名称绑定至同一索引，但仅当所有绑定均指向相同通用属性时才有效。
相同位置。这被称为*别名*。此机制仅在可执行程序中仅有一个别名属性处于活动状态时有效，或者当着色器中没有任何路径会消耗同一位置下别名属性集中的多个属性时才有效。 若链接器判定着色器中所有路径均消耗多重别名属性，则可能引发链接错误，但实现方在此情况下无需强制报错。编译器和链接器可假定不存在别名关系，并采用仅在无别名时有效的优化策略。泛型属性与常规属性之间不可建立别名关系。

统一变量

着色器可声明命名*统一*变量，具体说明详见《OpenGL着色语言规范》。这些统一变量的值在单个基元内保持恒定，通常在多个基元间也保持不变。 统一变量属于程序对象特有的状态。加载后其值将被保留，且只要程序对象未被重新链接，每次使用该对象时其值都会被恢复。当编译器和链接器判定某统一变量在可执行代码运行时会被实际访问时，该变量即被视为*活动状态*。若编译器和链接器无法做出明确判定，则该统一变量仍被视为活动状态。

顶点着色器可访问的统一变量存储空间大小由实现相关的常量 `MAX_VERTEX_UNIFORM_COMPONENTS` 指定。该值代表顶点着色器统一变量存储区可容纳的独立浮点数、整数或布尔值数量。若尝试使用的空间超过顶点着色器统一变量可用存储空间，将引发链接错误。

当程序成功链接后，该程序对象下所有活动统一变量均初始化为零（布尔型变量为`FALSE`）。成功链接还会为每个活动统一变量生成一个存储地址。可通过该地址配合相应的Uniform*命令（详见下文）修改活动统一变量的值。每次成功重新链接后，这些地址将失效并被重新分配。

若需查找程序对象内某个活动统一变量的存储位置，请使用命令

```
int GetUniformLocation( uint program, const char *name );
```

此命令将返回统一变量*名称*的位置。*名称*必须为以空字符结尾的字符串，且不包含空格。 若*名称*不对应*程序中*有效的统一变量名，或*名称*以保留前缀"`gl `"开头，则返回值为-1。*若程序*链接未成功，将生成错误`INVALID OPERATION`。程序链接完成后，统一变量的位置不会改变，除非重新链接程序。

有效的*名称*不能是结构体、结构体数组，也不能是单个向量或矩阵的任何部分。为识别有效*名称*，可在*名称中*使用"`.`"（点）和"`[]`"运算符来指定结构体的成员或数组的元素。

统一数组的首个元素通过在统一数组名称后附加"`[0]`"来标识。除非字符串*名称*的末尾部分表示

统一数组时，该数组首个元素的位置可通过统一数组名称或统一数组名称后缀"[0]"获取。

要确定程序使用的活动统一属性集，并
确定其大小和类型，请使用以下命令：

```
void GetActiveUniform( uint program, uint index, sizei bufSize, sizei *length,
    int *size, enum *type, char *name );
```

该命令提供*索引*所选统一变量的信息。*索引*值为0时选取首个活动统一变量，*索引*值为ACTIVE UNIFORMS 1时选取最后一个活动统一变量。ACTIVE UNIFORMS的值可通过**GetProgramiv**查询（参见第6.1.14节）。_ 若索引值大于或等于ACTIVE UNIFORMS，则触发INVALID VALUE错误。需注意*索引值*仅用于标识活动统一变量列表中的成员，与对应统一变量变量的实际存储位置无关。

参数*程序*是程序对象的名称，该对象曾被发出过**LinkProgram命令**。*程序*不必已成功链接，链接可能因活动统一变量数量超过限制而失败。

若发生错误，返回参数*长度*、*大小*、*类型*和*名称*将保持不变。

对于选定的统一变量，其名称将被返回至*名称变量*name。该字符串*名称*以空字符结尾。实际写入*名称变量name*的字符数（不含空字符结尾）将通过*length*返回。若length为NULL，则不返回长度值。可写入*名称变量name*的最大字符数（含空字符结尾）由*bufSize*指定。返回的统一变量名称也可为内置统一状态的名称。 内置统一状态的完整列表详见《OpenGL着色语言规范》第7.5节。*程序*中最长统一变量名称的长度由ACTIVE UNIFORM MAX LENGTH决定，可通过**GetProgramiv函数**查询（参见第6.1.14节）。

在着色器中声明的每个统一变量，都会通过点号（.）和方括号（[]）运算符进行拆分（如有必要），直至每个字符串均可合法地重新传递给**GetUniformLocation函数**。这些字符串各自构成一个有效的统一变量，且每个字符串都会被分配一个索引。

对于选定的制服，其类型将返回至*type字段*。制服尺寸将返回至*size字段*。 *size字段*的数值单位与*type字段*返回的类型一致。 返回的类型可为FLOAT、

浮点数向量2，浮点数向量3，浮点数向量4，整型数，整型数向量2，整型数向量3，整型数向量4，布尔型， 布尔向量2、布尔向量3、布尔向量4、浮点矩阵2、浮点矩阵3、浮点矩阵4、1D采样器、2D采样器、3D采样器、立方体采样器、1D阴影采样器或2D阴影采样器。

如果数组中存在一个或多个活动元素，**GetActiveUniform** 将根据上述限制条件将数组名称返回至 *name* 参数。数组类型将返回至 *type* 参数。*size* 参数包含已使用的最高数组元素索引值加一。编译器或链接器将确定实际使用的最高索引值。每个统一数组仅会由图形渲染器报告一个活动统一变量。

GetActiveUniform将尽可能返回所有活动统一变量的信息。若无可用信息，*length*将设为零，*name*将为空字符串。当链接失败后调用GetActiveUniform时可能出现此情况。

要将值加载到当前使用的程序对象的统一变量中，请使用以下命令：

```
void Uniform1234{if}(int 位置, T 值);
void Uniform1234{if}v(int 位置, sizei 数量,
    T 值);
void 均匀矩阵234{f}v(int 位置, sizei 数量, boolean 转置,
    const float *值);
```

给定值将被加载到由*location*标识的统一变量位置。

Uniform*f v 命令将向统一位置加载计数组的1至4个浮点值，该位置可定义为浮点数、浮点向量、浮点数组或浮点向量数组。

Uniform*i v 命令将载入 *count* 组（每组 1 至 4 个）整数值至统一位置，该位置可定义为采样器、整数、整数向量、采样器数组、整数数组或整数向量数组。仅 **Uniform1i v** 命令可用于载入采样器值（详见下文）。

UniformMatrix 234 fv 命令将向定义为矩阵或矩阵数组的统一位置加载 2×2、3×3 或 4×4 矩阵（对应命令名称中的 **2**、**3** 或 **4**），这些矩阵包含浮点数值。若 *transpose* 为 FALSE，则矩阵按列优先顺序指定；否则按行优先顺序指定。

当为声明为布尔值的统一变量加载值时，无论是布尔向量、布尔数组还是布尔向量数组，均可使用**Uniform*i v**和**Uniform*f v**命令集加载布尔值。类型转换由GL自动完成。若输入值为0或

0.0f，否则设为TRUE。使用的Uniform*命令必须与着色器中声明的统一变量大小匹配。例如，要加载声明为bvec2的统一变量，可使用**Uniform2i v**或**Uniform2f v**。若尝试使用不匹配的Uniform*命令，将引发INVALID OPERATION错误。本例中使用**Uniform1iv**将导致错误。

对于所有其他统一类型，使用的**Uniform***命令必须与着色器中声明的统一变量的大小和类型相匹配。不进行任何类型转换。 例如，加载声明为vec4的统一变量时必须使用Uniform4**f v**；加载3x3矩阵时必须使用UniformMatrix**3**fv。若尝试使用不匹配的Uniform*命令，将触发INVALID OPERATION错误。本例中**使用Uniform4i v**即会引发错误。

当从任意位置 *k* 开始向声明为数组的均匀分布中加载 *N* 个元素时，数组中从 *k* 到 *k* + *N* 的元素 1的数组元素将被替换为新值。对于任何数组元素的值若超过**GetActiveUniform**报告的最高数组元素索引，将被GL忽略。

若 *位置*参数值为-1，Uniform*指令将静默忽略传入数据，当前统一变量值保持不变。

当满足以下任一条件时，**Uniform***命令将触发INVALID OPERATION错误且不修改统一变量值：

- • 使用的Uniform*命令名称所示大小与着色器中声明的uniform大小不匹配
- 如果着色器中声明的统一变量类型不是布尔型，且使用的Uniform*命令名称所指示的类型与该统一变量的类型不匹配，
- 若 *计数*大于1，且着色器中声明的统一变量并非数组变量，
- 若当前使用的程序对象中不存在位置为*location*的变量且*location*不为-1，或
- 若当前未使用的程序对象中不存在该位置的变量。

采样器

*采样*器是OpenGL着色语言中用于标识每次纹理查找所用纹理对象的特殊统一变量。采样器的值指示正在访问的纹理图像单元。将采样器值设为*i*将选择纹理

纹理图像单元编号 *i*。*i* 的取值范围为 0 到实现支持的最大纹理图像单元数量（该数量取决于具体实现）。

采样器的类型用于标识纹理图像单元上的目标。随后将使用绑定至该目标的纹理对象进行纹理查找。例如，类型为sampler2D的变量即选定其纹理图像单元上的TEXTURE 2D目标。纹理对象与目标的绑定操作仍通过**BindTexture**实现，而选择待绑定的纹理图像单元则仍通过**ActiveTexture**完成。

采样器的定位需通过**GetUniformLocation函数**查询，与任何统一变量相同。采样器值需通过**调用Uniform1i v函数**设置。禁止使用**其他Uniform***入口点加载采样器，否则将导致INVALID OPERATION错误。

在同一程序对象内，禁止不同采样器类型的变量指向同一纹理图像单元。此类情况仅能在后续渲染命令执行时被检测到，届时将触发INVALID OPERATION错误。

活动采样器是指程序对象中实际正在使用的采样器。**Link**Program命令用于判断采样器是否处于活动状态。LinkProgram命令将尝试检测程序对象所含着色器中的活动采样器数量是否超过最大允许限制。若判定活动采样器数量超限，则链接失败（不同类型着色器的限制值可能不同）。 每个活动采样器变量均计入限制额度，即使多个采样器引用同一纹理图像单元。若链接时无法确定（例如程序对象仅包含顶点着色器），则将在下次渲染命令执行时判定，并生成INVALID OPERATION错误。

可变变量

顶点着色器可定义一个或多个变量（参见OpenGL着色语言规范）。这些值将在渲染的基元上进行插值处理。OpenGL着色语言规范为顶点着色器定义了一组内置变量，这些变量对应于顶点处理后固定功能处理所需的值。

用于处理可变变量的插值器数量由实现相关的常量 MAX VARYING FLOATS 决定。该值代表可插值的独立浮点数值数量；声明为向量、矩阵和数组的可变变量均会消耗多个插值器。程序链接时，任何可变变量的所有组件都会——

由顶点着色器写入或由片段着色器读取的变量将计入此限制。变换后的顶点位置（glPosition）不属于变量变量，不计入此限制。若程序中着色器访问的变量变量超过MAX VARYING_FLOATS个组件的容量，则该程序可能无法链接，除非设备相关的优化能使其适应可用硬件资源。

2.15.4 着色器执行

若通过调用**UseProgram将**包含顶点着色器的成功链接程序对象设为当前对象，则顶点着色器的可执行版本将用于处理传入顶点值，而非第2.11至2.14节所述的固定功能顶点处理。具体而言：

- • 模型视图矩阵和投影矩阵不会应用于顶点坐标（参见第2.11节）
- 纹理矩阵不会应用于纹理坐标（第2.11.2节）。
- 法线不会转换为视点坐标，也不会进行缩放或归一化（第2.11.3节）。
- 未对AUTO NORMAL评估的法线进行归一化处理（第5.1节）。
- 纹理坐标不会自动生成（第2.11.4节）。
- 不执行顶点光照计算（第2.14.1节）。
- 颜色材质计算未执行（第2.14.3节）。
- 颜色索引光照未执行（第2.14.5节）。
- 上述所有内容均适用于设置当前光栅位置（第2.13节）。

对顶点着色器执行后得到的顶点值将应用以下操作：

- 颜色截断或遮罩处理（第2.14.6节）。
- 剪裁坐标的透视分割（第2.11节）。
- 视口映射，包含深度范围缩放（第2.11.1节）。

- 裁剪操作，包含客户端定义的裁剪平面（第2.12节）。
- 正面面判定（第2.14.1节）。
- 平面着色（第2.14.7节）。
- 颜色、纹理坐标、雾效、点大小及通用属性裁剪（第2.14.8节）。
- 最终颜色处理（第2.14.9节）。

后续章节将阐述顶点着色器执行过程中的若干特殊考量。

纹理访问

顶点着色器具备在纹理贴图中进行查找的能力（前提是GL实现支持此功能）。顶点着色器可使用的纹理图像单元最大数量为MAX VERTEX TEXTURE IMAGE UNITS；若最大值为零，则表示该GL实现不支持顶点着色器中的纹理访问操作。 GL渲染管线中片段着色器阶段可用的纹理图像单元上限为MAX TEXTURE IMAGE UNITS。顶点着色器与片段着色器处理阶段的纹理图像单元总使用量不得超过MAX COMBINED TEXTURE IMAGE UNITS。 若顶点着色器与片段着色器阶段同时访问同一纹理图像单元，则该访问将计入MAX COMBINED TEXTURE IMAGE UNITS限制中的两个纹理图像单元。

当顶点着色器中执行纹理查找时，过滤后的纹理值τ按第3.8.8节和第3.8.9节所述方式计算，并根据表3.21（第3.8.9节）转换为纹理源颜色C(s)。τ按第3.8.8节和第3.8.9节所述方式计算，并根据表3.21（第3.8.13节）将其转换为纹理源*颜色*C_s。最终返回一个四分量向量(R_s, G_s, B_s, A_s)至顶点着色器。

在顶点着色器中，无法如第3.8.8节所述，通过纹理坐标对窗口坐标的偏导数来执行自动细节等级计算。因此无法自动选择图像数组的细节等级。 纹理贴图的缩放由细节级别值控制，该值可选地作为参数传递给纹理查找函数。若纹理查找函数提供显式细节级别值 l，则预偏置细节级别值 $\lambda_{base}(x, y) = l$（替代公式 3.18）。 若纹理查找函数未提供显式细节级别值，则 $\lambda_{base}(x, y) = 0$。此时缩放因子 $\rho(x, y)$ 及其近似函数 $f(x, y)$（参见方程 3.21）将被忽略。

涉及深度分量数据的纹理查找操作，可直接返回深度数据，或返回与用于执行查找的纹理坐标进行比较后的结果，具体如第3.8.14节所述。 着色器通过使用阴影采样器类型（sampler1DShadow 或 sampler2DShadow）请求比较操作，纹理则通过 TEXTURE COMPARE MODE 参数请求。这些请求必须保持一致；若出现以下情况，纹理查找结果将未定义：

- • 纹理查找函数使用的采样器类型为 sampler1D 或 sampler2D，且纹理对象的内部格式为 DEPTH COMPONENT，同时 TEXTURE COMPARE MODE 未设置为 NONE。
-
- 纹理查找函数中使用的采样器类型为 sampler1DShadow 或 sampler2DShadow，且纹理对象的内部格式并非深度分量。

若顶点着色器使用的采样器关联的纹理对象未完成（如第3.8.10节定义），则纹理图像单元将返回(*R, G, B, A*) = (0, 0, 0, 1)。

位置不变性

如果顶点着色器使用内置函数 ftransform 生成顶点位置，则通常可确保无论使用该顶点着色器还是固定功能管道，转换后的位置都保持一致。这使得多通道渲染算法得以正确运行——某些通道采用固定功能顶点转换，其他通道则使用顶点着色器。 若顶点着色器未使用 ftransform 生成位置，则即使用于计算位置的指令序列与第 2.11 节所述的变换序列一致，也无法保证变换后的位置匹配。

验证

在链接时，并非总能确定程序对象是否实际可执行。因此，当首次发出渲染命令时会进行验证，以判断当前活动的程序对象是否可执行。若

无法执行时，则不会渲染任何片段，且**Begin**、Raster-Pos或任何执行隐式**Begin**的命令将引发INVALID OPERATION错误。

当满足以下条件时，**Begin**、**RasterPos**或执行隐式**Begin**的命令将触发此错误：

- 当前程序对象中任意两个活动采样器类型不同，但引用同一纹理图像单元；
- 当前程序对象中任何活动采样器指向的纹理图像单元，其固定功能片段处理访问的纹理目标与采样器类型不匹配，或
- 程序中活动采样器的数量与用于固定功能片段处理的纹理图像单元数量之和，超过了允许的纹理图像单元总数限制。

若当前使用的程序对象未配置片段着色器，则将执行固定功能片段处理操作。

这些渲染命令报告的"无效操作"错误可能无法提供足够信息来确定当前活动程序对象为何无法执行。对于仍可执行但因当前GL状态导致效率低下或效果欠佳的程序对象，则完全无法获取相关信息。作为开发辅助手段，请使用命令

```
void ValidateProgram( uint program );
```

对程序对象 *program* 进行验证，以检查其是否符合当前 GL 状态。每个程序对象都具有一个布尔状态 VALIDATE STATUS，该状态会在验证过程中被修改。可通过 **GetProgramiv 函数**查询此状态（参见第 6.1.14 节）。 若验证成功，该状态将设置为TRUE；否则将设置为FALSE。验证成功时，在当前GL状态下程序对象必将执行；验证失败时，在当前GL状态下程序对象必将不执行。

ValidateProgram将检查所有可能导致渲染命令执行时出现INVALID OPERATION错误的条件，并可能检查其他条件。 例如，它可能提供优化着色器代码片段的建议。*程序*的信息日志将被验证结果信息覆盖（可能为空字符串）。写入信息日志的结果通常仅在应用程序开发阶段有用；应用程序不应期望不同GL实现产生完全相同的信息。

着色器不应因指令空间不足或临时变量不足而编译失败，程序对象也不应因此类原因链接失败。实现应确保所有有效的着色器和程序对象均可成功编译、链接并执行。

未定义行为

在着色器中使用数组或矩阵变量时，可能通过运行时计算的索引访问超出变量声明范围的元素。此类越界读取将返回未定义值；越界写入将产生未定义结果，并可能破坏着色器或GL使用的其他变量。着色器对此类错误的防护程度取决于具体实现。

2.15.5 必需状态

GL维护状态以标识正在使用的着色器和程序对象名称。初始状态下不存在任何着色器或程序对象，且无名称处于使用中。

每个着色器对象所需的状态包含：

- 一个无符号整数，用于指定着色器对象名称。
- 一个整数，用于存储着色器类型的值。
- 一个布尔值，用于存储删除状态，初始值为FALSE。
- 一个布尔值，用于存储上次编译的状态，初始值为FALSE。
- 一个字符数组，用于存储信息日志，初始为空。
- 一个整数，用于存储信息日志的长度。
- 一个字符数组，用于存储拼接后的着色器字符串，初始为空。
- 一个整数，用于存储拼接后的着色器字符串长度。每个程序对象所需的状态包含：
- 一个无符号整数，用于标识程序对象的名称。
- 一个布尔值，用于存储删除状态，初始值为FALSE。
- 一个布尔值，用于存储上次链接尝试的状态，初始值为FALSE。

- 一个布尔值，用于存储上次验证尝试的状态，初始值为FALSE。
- 一个整数，用于存储已附加着色器对象的数量。
- 一个无符号整数列表，用于记录已附加着色器对象的名称。
- 一个字符数组，用于存储信息日志，初始为空。
- 一个整数，用于存储信息日志的长度。
- 一个整数，用于存储活动统一变量的数量。
- 对于每套活跃制服，包含三个整数（存储其位置、尺寸和类型）以及一个字符型数组（存储其名称）。
- 一个字符数组，存储每个活动统一变量的数值。
- 一个整数，用于存储活动属性的数量。
- 对于每个活动属性，包含三个整数（存储其位置、大小和类型）以及一个字符型数组（存储其名称）。

支持顶点着色器所需的附加状态包括：

- 一个位标记，用于指示顶点程序双面颜色模式是否启用（初始为禁用）。
- 一位用于指示顶点程序点大小模式（第3.3.1节）是否启用，初始状态为禁用。

此外，需保留一个无符号整数用于存储当前程序对象的名称（若存在）。

第3章

光栅化

光栅化是将原始图形转换为二维图像的过程。该图像的每个点都包含颜色和深度等信息。因此，原始图形的栅格化包含两个部分：首先确定原始图形占据窗口坐标系中整数网格的哪些方格；其次为每个方格分配深度值和一个或多个颜色值。 该过程的结果将传递至GL的下一阶段（片元操作），该阶段利用这些信息更新帧缓冲区中的相应位置。图3.1示意了光栅化过程。 片段的颜色值最初由光栅化操作（第3.3至3.7节）确定，随后可能通过执行第3.8、3.9和3.10节定义的纹理映射、颜色合成及雾化操作进行修改，或由第3.11节定义的片段着色器进行调整。 最终深度值最初由光栅化操作确定，可被片段着色器修改或替换。点、线、多边形、像素矩形或位图的光栅化结果均可通过片段着色器进行处理。

网格单元及其对应的颜色、z（深度）、雾化坐标和纹理坐标参数共同构成一个*片元*；这些参数统称为片元的*关联数据*。 片元的定位基于其左下角，该点位于整数网格坐标上。光栅化操作同时涉及片元中心点，该点相对于左下角偏移$(1/2, 1/2)$（故位于半整数坐标上）。

在图形学中，网格单元并不需要实际呈正方形。光栅化规则不受网格单元实际纵横比的影响。然而，非正方形网格的显示会导致光栅化后的点和线段在某个方向上显得更粗。我们假设片段是正方形的，因为这能简化抗锯齿和纹理贴图操作。

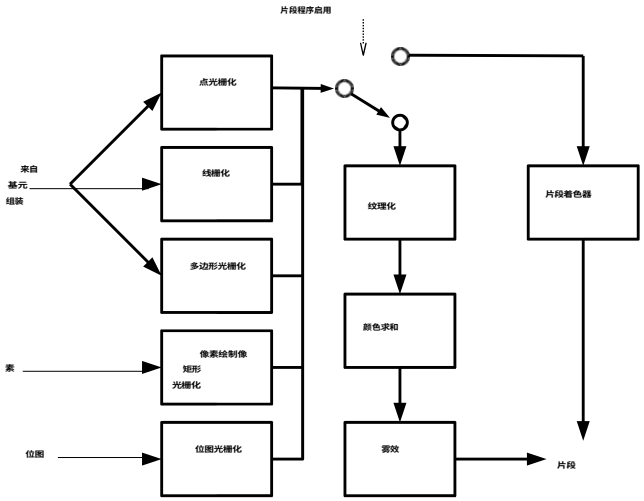

片段程序启用
点光栅化
线栅化
多边形光栅化
像素绘制像
矩形
光栅化
位图光栅化
来自
基元
组装
素
位图
纹理化
颜色求和
雾效
片段着色器
片段

图3.1. 光栅化过程。

多个因素会影响光栅化。线条和多边形可能呈现点状纹理。点可赋予不同直径，线段可设定不同宽度。点、线段或多边形均可启用抗锯齿处理。

3.1 不变性

考虑通过在窗口坐标系中将原始图形 *p* 偏移 (*x, y*) 获得的原始图形 *p'*，其中 *x* 和 *y* 为整数。 只要*原始图形 p'和原始图形 p* 均未被裁剪，则由 *p'* 生成的每个片元 *f '* 必须与*原始图形 p* 生成的对应片元 *f* 完全相同，唯一区别在于 *f '* 的中心相对于 *f* 的中心偏移了 (*x, y*) 量。

3.2 抗锯齿

点、线或多边形的抗锯齿处理取决于图形渲染器处于RGBA模式还是颜色索引模式，具体采用以下两种方式之一实现：

在RGBA模式下，光栅化片段的R、G、B值保持不变，但A值会与[0, 1]范围内的浮点数相乘——该数值描述了片段在屏幕像素上的覆盖程度。GL的片段处理阶段可配置为使用A值，将传入的片段与帧缓冲区中对应的现有像素进行混合渲染。

在颜色索引模式下，颜色索引的最低有效 *b* 位（位于二进制点左侧）用于抗锯齿处理；其中 $b = \min(4, m)$，*m* 表示帧缓冲区颜色索引部分的位数。抗锯齿过程根据片段的覆盖值设置这 *b* 位：无覆盖时设为零，完全覆盖时设为全一。

抗锯齿片段覆盖值的具体计算方式难以一概而论。原因在于，高质量抗锯齿技术不仅需考虑感知因素，还需兼顾显示帧缓冲区内容的显示器特性。此类细节超出本文档讨论范围。 此外，某原始图元片段的覆盖值计算可能不仅取决于该片段所在网格单元，还取决于其与多个邻近网格单元的关联关系。另一关键考量是：精确计算覆盖值可能导致计算成本过高，因此允许具体GL实现采用快速但非完全精确的覆盖值计算方法进行近似。

基于上述考量，我们选择在典型场景下定义精确抗锯齿的行为：当每个显示像素均为完美正方形时

均匀强度。该正方形称为*片段方块*，其左下角坐标为(x, y)，右上角坐标为$(x+1, y+1)$。我们认识到这种简单的盒式滤波器可能无法产生最理想的抗锯齿效果，但它提供了一个简单且定义明确的模型。

图形渲染实现可采用其他抗锯齿方法，但须满足以下条件：

1. 若两个片元f_1和f_2中，某个基元覆盖的f_1区域是覆盖f_2的对应区域的子集，则f_1的覆盖率计算值必须小于或等于$f_{(2)\text{ 的覆盖率}}$计算值。

2. *片段f的*覆盖计算必须是局部性的：它只能取决于f与正在进行光栅化的基元边界之间的关系，而不能取决于f的x和y坐标。

另一项可取但非必需的属性是：

3. 对特定基元进行光栅化生成的所有片段的覆盖值之和必须保持恒定，且不受窗口坐标系中任何刚性变换的影响——前提是这些片段均不位于窗口边缘。

在某些实现中，通过提供GL提示（第5.6节），可获得不同程度的抗锯齿质量，从而允许用户在图像质量与速度之间进行权衡。

3.2.1 多采样

多采样是一种对所有GL基元（点、线、多边形、位图和图像）进行抗锯齿的机制。 该技术通过在每个像素点对所有基元进行多次采样实现。每次像素更新时，颜色采样值会被解析为单一可显示颜色，因此抗锯齿效果在应用层面上呈现为自动完成。由于每次采样均包含颜色、深度和模板信息，其颜色（含纹理操作）、深度及模板功能与单采样模式具有等效性能。

在帧缓冲区中添加了一个额外的缓冲区，称为多采样缓冲区。像素采样值（包括颜色、深度和模板值）存储在此缓冲区中。 每个采样包含每个片段颜色的独立颜色值。当帧缓冲区包含多采样缓冲区时，即使该多采样缓冲区未存储深度或模板值，它也不会包含深度或模板缓冲区。然而，颜色缓冲区（左、右、前、后和辅助）确实与多采样缓冲区共存。

缓冲区（左、右、前、后及辅助缓冲区）仍可与多采样缓冲区共存。

多采样抗锯齿技术在渲染多边形时最具价值，因其无需隐藏面消除排序，且能正确处理相邻多边形、物体轮廓乃至相交多边形。 若仅渲染点或线，基础GL提供的"平滑"抗锯齿机制可能生成更高质量图像。该机制设计允许在单场景渲染过程中交替使用多采样与平滑抗锯齿技术。

若 SAMPLE BUFFERS 的值为 1，则所有基元的光栅化过程将发生改变，此过程称为多采样光栅化。否则，基元光栅化称为单采样光栅化。通过调用 **GetIntegerv** 函数并将 *pname* 设置为 SAMPLE BUFFERS 可查询该值。

在多采样渲染过程中，像素片段的内容通过两种方式改变：首先，每个片段包含一个覆盖值，该值包含 SAMPLES 位。SAMPLES 的值是实现相关的常量，可通过调用 **GetIntegerv** 并设置 *pname* 为 SAMPLES 来查询。

其次，每个片段包含多个采样深度值、颜色值和纹理坐标集，而非单采样渲染模式下维护的单一深度值、颜色值和纹理坐标集。实现方案可选择将相同的颜色值和纹理坐标集分配给多个采样点。 颜色值与纹理坐标集的评估位置可在像素内任意选取，包括片段中心或任意采样点。颜色值与纹理坐标集无需在相同位置评估。因此每个像素片段包含整数x/y网格坐标、SAMPLES个颜色值与深度值、SAMPLES组纹理坐标集，以及最高为SAMPLES位的覆盖值。

多采样光栅化通过调用**Enable**或**Disable**函数启用或禁用。
使用符号常量 MULTISAMPLE。

若禁用 MULTISAMPLE，所有基元的多采样光栅化效果等同于单采样（片段中心）光栅化，但片段覆盖值将设为全覆盖。颜色值、深度值及纹理坐标集可采用单采样光栅化分配的数值，也可按下文多采样光栅化规则进行分配。

当启用MULTISAMPLE时，所有基元的多采样光栅化与单采样光栅化存在显著差异。需理解帧缓冲区中每个像素均关联多个采样点位置。这些位置

精确位置而非区域，称为采样点。与像素关联的采样点可能位于界定该像素的单位正方形内部或外部。此外，帧缓冲区中每个像素的采样点相对位置可能相同，也可能不同。

若采样位置随像素变化，则应将其对齐至窗口边界而非屏幕边界。否则渲染结果将受窗口位置影响。第3.1节所述的不变性要求对所有多采样光栅化操作均予以放宽，因为采样位置可能取决于像素位置。

无法查询像素的实际采样位置。

3.3 点

若顶点着色器未激活，则点的光栅化由以下函数控制：

```
void PointSize( float size );
```

size 参数指定点图形的请求尺寸。默认值为 1.0。若值小于或等于零，将导致 INVALID VALUE 错误。

请求的点尺寸将乘以距离衰减因子，先限制在指定点尺寸范围内，再进一步限制在实现相关的点尺寸范围内，从而得出衍生点尺寸：

$$\text{衍生尺寸} = \text{取值范围}_{\text{尺寸}} * \left(\frac{1}{a + b * d + c * d^2} \right)$$

其中d是眼睛坐标系中从眼睛（0, 0, 0, 1）到顶点的距离，*a*、*b*和*c*是距离衰减函数系数。

如果未启用多采样，则派生尺寸将作为点宽传递至光栅化阶段。

若顶点着色器处于活动状态且顶点程序点大小模式启用，则从（可能经过裁剪的）着色器内置函数 gl PointSize 获取推导点大小，并将其限制在实现相关的点大小范围内。若写入 gl PointSize 的值小于或等于零，则结果未定义。 若顶点着色器处于活动状态且顶点程序点大小模式禁用，则点大小由PointSize命令指定的点大小状态决定。此时不执行距离衰减。顶点程序点大小模式通过调用**Enable**或**Disable**并传入符号值VERTEX PROGRAM POINT SIZE来启用或禁用。

如果启用了多采样，实现可选择对点透明度进行渐变处理（参见第3.13节），而非允许点宽度低于指定阈值。此时，光栅化点的宽度为

$$\text{宽度} = \begin{cases} \text{推导尺寸} & \text{推导尺寸} \geq \text{阈值} \\ \text{阈值} & \text{否则} \end{cases} \tag{3.1}$$

且淡出因子按以下方式计算：

$$\text{fade} = \begin{cases} 1 & \text{推导尺寸} \geq \text{阈值} \\ \left(\frac{\text{推导尺寸}}{\text{阈值}}\right)^2 & \text{否则} \end{cases} \tag{3.2}$$

距离衰减函数*系数*a、*b*和c，首个点尺寸范围的边界限制clamp，以及点衰减*阈值*，均通过以下参数指定：

```
void PointParameter{if}( enum pname, T param );
void PointParameter{if}v( enum pname, const T params );
```

若 *pname* 为 POINT SIZE MIN 或 POINT SIZE MAX，则 *param* 指定或 *params* 分别指向衍生点尺寸的下限或上限阈值。 若下限大于上限，则阈值处理后的点尺寸未定义。 若 *pname* 为 POINT DISTANCE ATTENUATION，则 *params* 指向系数 *a*、*b* 和 *c*。 若 *pname* 为 POINT FADE THRESHOLD SIZE，则 *param* 指定或 *params* 指向点*衰减阈值*。当 POINT SIZE MIN、POINT SIZE MAX 或 POINT FADE THRESHOLD SIZE 取值小于零时，将引发 INVALID VALUE 错误。

点抗锯齿功能可通过调用 **Enable** 或 **Disable** 并传入符号常量 POINT SMOOTH 来启用或禁用。默认状态为点抗锯齿功能处于禁用状态。

点精灵的启用或禁用需通过调用**Enable**或**Disable**函数并传入符号常量POINT SPRITE实现。默认状态为点精灵禁用。当点精灵启用时，点抗锯齿启用状态将被忽略。

点精灵纹理坐标替换模式通过第3.8.13节所述的TexEnv*命令设置，其中*目标*为POINT SPRITE，参数名称为COORD REPLACE。*参数param*的取值为FALSE或TRUE。每个纹理坐标集的默认值均为禁用点精灵纹理坐标替换。

点精灵纹理坐标原点通过**Point**Parameter*命令设置，*其中pname*为POINT SPRITE COORD ORIGIN，*param*为LOWER LEFT或UPPER LEFT。默认值为UPPER LEFT。

3.3.1 基础点光栅化

默认状态下，点渲染*通过将*x_w*和*y_w坐标（注意下标表示窗口坐标系的x/y轴）截断为整数实现。该(x, y)地址与对应顶点关联数据生成的信息，将作为单个片段发送至GL的片段处理阶段。

点宽度非1.0时的效果取决于点抗锯齿和点精灵的状态。若抗锯齿与点精灵均禁用，则实际宽度由以下步骤确定：先将指定宽度四舍五入至最接近的整数，再将其限制为实现相关的最大非抗锯齿点宽度。该实现相关的值必须不小于实现相关的最大抗锯齿点宽（四舍五入至最近整数值），且无论如何不得小于1。若指定宽度四舍五入后结果为0，则视为该值为1。若最终宽度为奇数，则点

$$(x, y) = (\lfloor x_w \rfloor + \frac{1}{2}, \lfloor y_w \rfloor + \frac{1}{2})$$

根据顶点的 x_w和 y_w 计算得出，以 (x, y) 为中心的奇数宽度方形网格定义了光栅化片段的中心（需注意片段中心位于半整数窗口坐标值处）。若宽度为偶数，则中心点为

$$(x, y) = (\lfloor x_w + \frac{1}{2} \rfloor, \lfloor y_w + \frac{1}{2} \rfloor);$$

光栅化片段中心位于以 (x, y)为中心的偶数宽度正方形内，其窗口坐标值为半整数。参见图3.2。

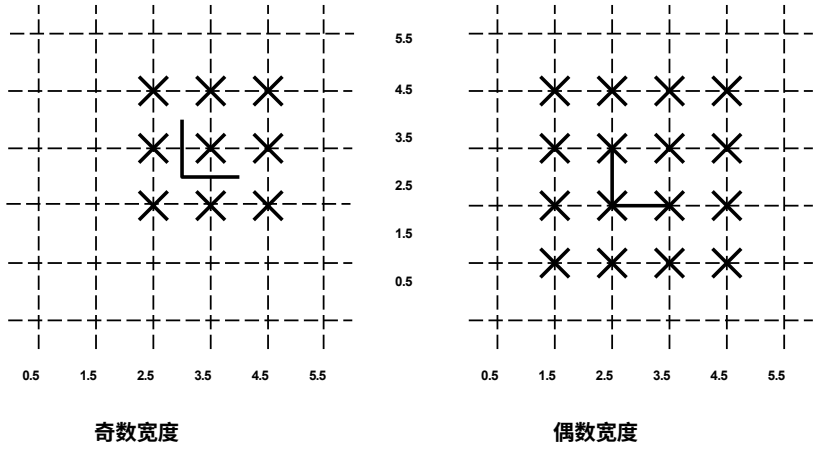


图3.2. 未抗锯齿宽点的光栅化处理。交叉符号表示光栅化过程中在阴影区域内任意点生成的片段中心。虚线网格线位于半整数坐标位置。

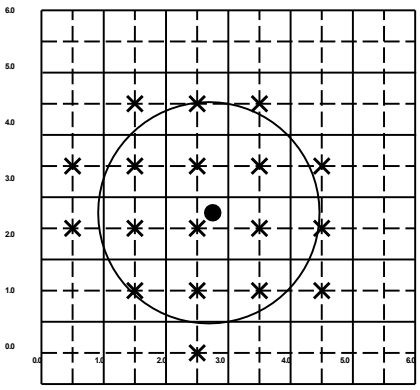


图3.3. 抗锯齿宽点的光栅化过程。黑色圆点表示待光栅化的点。阴影区域具有指定宽度。X标记表示光栅化生成的片段中心。片段的覆盖值计算基于阴影区域覆盖对应片段方块的比例。实线位于整数坐标上。

在光栅化非抗锯齿点时，所有生成的片段均分配相同的关联数据，即该点对应顶点的数据。

若启用抗锯齿且禁用点精灵，则点光栅化会为每个与圆形区域相交的片段方块生成一个片段。该圆形区域的直径等于当前点宽度，中心位于点坐标(x_w, y_w)处（图3.3）。 每个片段的覆盖值为圆形区域与对应片段方块的交集窗口坐标面积（但参见第3.2节）。该值将被保存并用于光栅化的最终步骤（第3.12节）。除覆盖值外，每个片段关联的数据均与被光栅化的点相同。

启用点抗锯齿时，并非所有宽度都需要支持，但宽度
必须提供1.0。若请求的宽度不受支持，则使用最接近的支持宽度替代。支持宽度的范围及该范围内等间距渐变的宽度取决于具体实现。 可通过第6章所述的查询机制获取宽度范围及梯度值。例如，若宽度范围为0.1至2.0且梯度间隔为0.1，则支持0.1、0.2、...、1.9、2.0等宽度值。

若启用点精灵功能，则点光栅化将为每个帧缓冲区像素生成片段，其中心位于以点坐标(x_w, y_w)为中心、边长等于当前点大小的正方形内部。

在点精灵光栅化过程中生成的所有碎片均被赋予相同的关联数据，即对应于该点的顶点数据。然而，对于每个设置为COORD REPLACE TRUE的纹理坐标集，这些纹理坐标将被替换为点精灵纹理坐标。*s*坐标在点上水平方向从左至右变化范围为0至1。 若点精灵坐标原点设置为左下（POINT SPRITE COORD ORIGIN），*则t*坐标在垂直方向上从底部到顶部变化范围为0到1。反之，若点精灵纹理坐标原点设置为左上（UPPER LEFT），则*t*坐标在垂直方向上从顶部到底部变化范围为0到1。*r*和q坐标分别被替换为常量0和1。

*计算*s和t坐标时采用以下公式:

$$s = \frac{1}{2} + \frac{x_f + 1 - x_w}{\text{尺寸}} \tag{3.3}$$

$$t = \begin{cases} \frac{1}{2} + \frac{(y_f + 1 - y_w)}{\text{尺寸}}, & \text{点精灵坐标原点} = \text{左下角} \\ \frac{1}{2} - \frac{(y_f + 1 - y_w)}{\text{尺寸}}, & \text{点精灵坐标原点} = \text{左上角} \end{cases} \tag{3.4}$$

*其中大小*表示该点的尺寸，x_f和 y_f是该片段的（整数）窗口坐标，而 x_w和 y(w) 是该点顶点精确的、未四舍五入的窗口坐标。

，x_w和 y_w 则是该点对应顶点的精确未舍入窗口坐标。

点精灵支持的宽度必须是抗锯齿点支持宽度的超集。这些宽度无需等间距分布。若请求的宽度不受支持，则使用最接近的支持宽度替代。

3.3.2 点光栅化状态

控制点光栅化的状态包含以下参数：浮点数点宽度、三个浮点数值（分别指定最小点尺寸、最大点尺寸及点淡出阈值尺寸）、三个浮点数值（指定距离衰减系数）、一个位标记（指示是否启用抗锯齿）、每个纹理坐标集对应的点精灵纹理坐标替换模式位标记，以及点精灵纹理坐标原点位标记。

3.3.3 点多采样光栅化

如果启用了MULTISAMPLE，且SAMPLE BUFFERS的值为1，则无论点抗锯齿（POINT SMOOTH）是否启用，点都将通过以下算法进行光栅化： 点光栅化为每个帧缓冲区像素生成一个片段，该像素包含一个或多个采样点（ ），这些点与以点坐标（x_w, y_w）为中心的区域相交。当POINT SPRITE禁用时，该区域为直径等于当前点宽度的圆形；当POINT SPRITE启用时，则为边长等于当前点宽度的正方形。 与该区域相交的采样点对应的覆盖位为1，其余覆盖位为0。除纹理坐标外，每个采样点关联的片段数据均来自光栅化点本身（当启用POINT SPRITE时，纹理坐标按第3.3节所述方式计算）。

点大小范围和渐变数量等同于禁用点精灵时对抗锯齿点的支持范围。支持的点大小集等同于启用点精灵且未启用多采样时的点精灵支持范围。

3.4 线段

线段可由线条带**开始**/结束对象、线环或独立线段序列生成。线段光栅化受多个变量控制。线宽可通过调用

```
void LineWidth( float width );
```

设置为适当的正浮点数值。默认宽度为1.0，小于等于0.0的值将引发INVALID VALUE错误。抗锯齿功能通过符号常量LINE SMOOTH的**启用/禁用**控制。最后，线段可采用点状填充效果，该效果由*设置点状纹理*的GL命令控制（详见下文）。

3.4.1 基本线段光栅化

线段光栅化首先将线段划*分为x主线段*或*y主线段*。x主线段的斜率位于闭区间[1, 1]内；其余线段均为y主线段（斜率由线段端点决定）。本文仅针对x主线段说明光栅化过程，除非对y主线段的修改不言自明。

理想情况下，光栅化渲染器采用"菱形退出"规则来确定由线段光栅化产生的片元。对于每个以窗口坐标 x_f和 y_f为中心的片元 f，定义一个菱形区域，该区域由四个半平面相交形成：

$$R_f = \{ (x, y) \mid |x - x_f| + |y - y_f| < 1/2. \}$$

本质上，一条起点为$\mathbf{p}_a$、终点为$\mathbf{p}_b$的线段会产生那些线段*与R_f*相交的片段 f，除非$\mathbf{p}_b$位于R_f内部。参见图3.4。

为避免端点*位于R_f*边界时产生的问题，我们（原则上）通过微小扰动调整给定端点。**设$\mathbf{p}_a$和$\mathbf{p}_b$**的窗口坐标分别为(x_a, y_a)与(x_b, y_b)。通过扰动获得**端点$\mathbf{p}'$**由 (x_a, y_a) (ϵ, ϵ^2) 给出，$\mathbf{p}'$ 由 (x_b, y_b) (ϵ, ϵ^2) 给出。将线条光栅化线段从 $\mathbf{p}_a$ 开始，到 $\mathbf{p}_b$ 结束，产生那些片段 f，其中**在$\mathbf{p}'$开始并在$\mathbf{p}'$**结束的线段*与R_f*相交，除非$\mathbf{p}'$包含在 R_f . ϵ被选取得足够小，使得当δ替代ϵ时，对线段进行光栅化处理产生的碎片与碎片，*当δ替代ϵ时*，对于任何$0 < \delta \leq \epsilon$。

当点$\mathbf{p}_a$和$\mathbf{p}_b$位于片段中心时，这种片段特征描述可简化为布雷森汉姆算法，但需作一处修改：此描述中生成的线段为"半开线段"，即对应**点$\mathbf{p}_b$**的最终片段不会被绘制。 这意味着在光栅化一系列相连线段时，共享端点仅生成一次而非两次（布雷森汉姆算法会重复生成）。

由于菱形退出规则的初始与终止条件可能难以实现，允许采用其他线段光栅化算法，但须遵循以下规则：

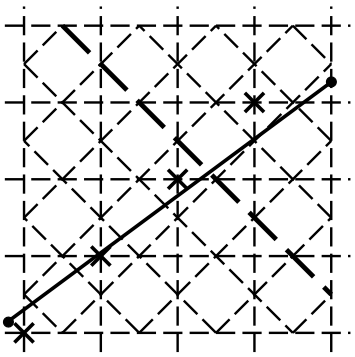


图3.4. 布雷森姆算法的可视化示意图。图中展示了一条线段的局部。每个片段中心周围都放置了一个高度为1的菱形区域；当线段穿出这些区域时，光栅化过程会生成相应的片段。

1. 算法生成的片段坐标*在x或y*窗口坐标方向上的偏差不得超过一个单位，与钻石退出规则生成的对应片段相比。

2. 算法生成的片段总数与菱形退出规则生成的片段总数差异不得超过一个。

3. 对于x主线，不得产生两个位于同一窗口坐标列的碎片（对于y主线，不得出现两个碎片位于同一行）。

4. 若两条线段共享端点，且两者均为x主线段（均由左至右或均由右至左）或y主线段（均由下至上或均由上至下），则栅格化这两条线段时既不会产生重复片段，也不会遗漏任何片段导致连接线段的连续性中断。

接下来需明确各栅格化片段数据的获取方式。设生成的片段中心窗口坐标为 $\mathbf{p}_r = (x_d, y_d)$，且 $\mathbf{p}_a = (x_a, y_a)$ 与 $\mathbf{p}_b = (x_b, y_b)$。设定

$$t = \frac{(\mathbf{p}_r - \mathbf{p}_a) \cdot (\mathbf{p}_b - \mathbf{p}_a)}{\left\| \mathbf{p}_b - \mathbf{p}_a \right\|^2} \tag{3.5}$$

(注：在$\mathbf{p}_a$处t=0，在$\mathbf{p}_b$处t=1。）对于片段关联*数据f的*值（无论是主次R、G、B或A通道（RGBA模式）还是颜色索引（颜色索引模式）），雾度坐标，s/t/r/q纹理坐标，或裁剪w坐标（深度值z需通过下方公式3.7计算），其计算方式为

$$f = \frac{(1-t)f_a/w_a + tf_b/w_b}{(1-t)/w_a + t/w_b} \tag{3.6}$$

其中 f_a和 f_b 分别是与线段起始端点和终止端点相关的数据；w_a和 w_b 分别是线段起始端点和终止端点的剪裁 w 坐标。请注意，线性插值会使用

$$f = (1-t)f_a + tf_b. \tag{3.7}$$

此公式之所以不正确（深度值除外），在于其在窗口空间内插值数据点，而该空间可能因透视效应产生畸变。实际所需的是在剪裁空间内插值时对应的值，

方程3.6正是如此实现的。图形学实现可能选择用方程3.7近似方程3.6，但在插值纹理坐标或裁剪w坐标时，这通常会导致不可接受的失真效应。

3.4.2 其他线段特性

我们刚刚描述了使用默认线条点阵$FFFF_{16}$对宽度为1的非抗锯齿线段进行光栅化的过程。接下来将说明针对线段光栅化参数取一般值时线段的光栅化处理。

线条点阵

命令

```
void LineStipple( int factor, ushort pattern );
```

定义一条*线点阵*。*pattern*是一个无符号短整数。*线点阵*取自*pattern*的最低16位。它决定了当线条进行光栅化时需要绘制的那些片段。*factor*是一个计数值，用于通过*使线点阵*中的每个位*被factor*次使用来修改有效线点阵。

*因子*值被限制在[1, 256]范围内。可通过常量LINE STIPPLE配合**启用**或**禁用指令**控制线条点状纹理效果。禁用时，效果等同于线条点状纹理采用默认值。

线点状填充掩盖了光栅化过程中生成的某些片段，使其不会被发送到图形渲染器的片段处理阶段。该掩盖机制通过三个参数实现：16位线点状*填充参数*p、线重复计*数*r以及整数点状填充*计数器*s。设

$$b = \lfloor s/r \rfloor \bmod 16,$$

若*向量* p 的第 b 位为 1，则生成一个片段；否则不生成。*向量* p 的位按从低到高依次编号，0 为最低位，15 为最高位。 *s的初*始值为零；每生成一条线段的片段后（片段按顺序生成，从起点开始向终点推进），*s*便递增一次。每当出现**Begin指令**时，以及在独立线段组（通过LINES参数调用Begin时指定）中的每条线段开始前，*s*都会重置为0。

若线段经过裁剪，则线段起始处的 *s* 值不可确定。

宽线条

非抗锯齿线的实际宽度由以下过程确定：先将给定的宽度四舍五入至最接近的整数，再将其限制在实现相关的最大非抗锯齿线宽度范围内。 该实现相关的值必须不小于实现相关的最大抗锯齿线宽（四舍五入至最近整数值），且无论如何不得小于1。若指定宽度四舍五入后结果为 0，则视为该值为1。

宽度非一的非抗锯齿线段通过在次要方向上偏移（对于x主线，次要方向为*y*；对于y主线，次要方向为*x*）并沿次要方向复制片段进行光栅化（见图3.5）。 设w为四舍五入后的宽度整数值（若w=0，则视为*w*=1）。 若线段端点在窗口坐标系中分别为(x_0, y_0)和(x_1, y_1)，则将端点为(x_0, y_0(*w* 1)/2)和(x_1, y_1(*w* 1)/2) 的线段将被光栅化，但不会生成单个片段，而是在每个 *x* 位置（y 主轴线段则为 *y* 位置）生成一列高度为 *w* 的片段（即长度为 *w* 的片段行）。该列最底部的片段即为使用修改后坐标光栅化宽度为 1 的线段所生成的片段。 若该*列*x位置的点阵位为零，则不生成整列；否则生成整列。

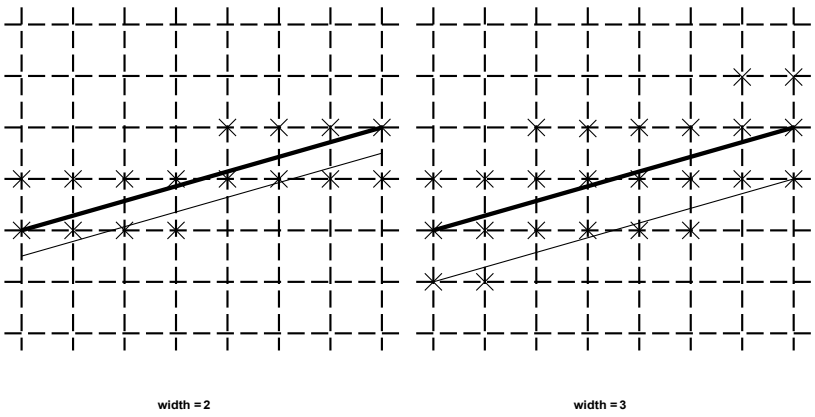


图3.5. 未抗锯齿宽线的栅格化效果。图中显示的是x方向为主的线段。粗线段为指定栅格化的线段；细线段为栅格化过程中使用的偏移线段。x标记表示栅格化生成的片段中心点。

抗锯齿

光栅化抗锯齿线段生成的片元，其片元方块与以该线段为中心的矩形相交。其中两条边与指定线段平行，每条边距线段当前宽度的一半：一条位于线段上方，一条位于下方。另外两条边穿过线段端点，并与指定线段的方向垂直。 通过计算矩形与片段正方形的交集面积来为每个片段计算覆盖值（参见图3.6；另见第3.2节）。与非抗锯齿线段相同，使用方程3.6计算关联数据值；使用方程3.5为每个被线段矩形截取的片段正方形求取*t*值。 线段抗锯齿无需支持所有宽度，但必须提供宽度为1.0的抗锯齿线段。与点宽度类似，可通过查询GL实现获取可用抗锯齿线宽的范围及分级数量。

为实现抗锯齿效果，点状线被视为以线段为中心的一系列连续矩形。每个矩形宽度等于当前线宽，长度为1像素（末个矩形可能较短）。这些矩形从0开始编号至n，起始于

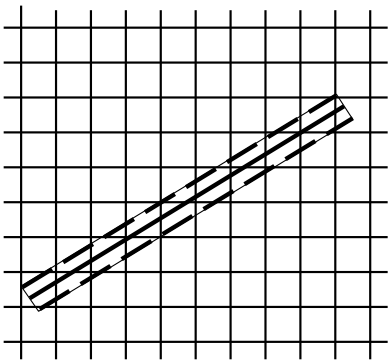


图3.6. 用于光栅化及计算抗锯齿线段对应覆盖值的区域（图示为x主轴线段）。

在线段起始端点发生的事件。每个矩形均按上述线点画法中的流程进行消除或生成，其中"片段"替换为"矩形"。生成的每个矩形均按抗锯齿多边形进行光栅化（详见下文说明），但不应用剔除、非默认PolygonMode设置及多边形点画法。

3.4.3 线条光栅化状态

用于线条光栅化的状态包含以下要素：浮点数线宽、16位线条点阵、点阵重复计数、点阵启用状态位以及抗锯齿启用状态位。此外，在光栅化过程中必须维护整数点阵计数器以实现线条点阵效果。 线宽初始值为1.0。线点阵*初始值为* $FFFF_{16}$（全1点阵）。线点阵重复计数初始值为1。线点阵初始状态为禁用。线段抗锯齿初始状态为禁用。

3.4.4 线条多采样光栅化

如果启用了MULTISAMPLE，且SAMPLE BUFFERS的值为1，则无论线抗锯齿（LINE SMOOTH）是否启用，线条都将使用以下算法进行光栅化。线条光栅化会为每个像素生成一个片元。

每个帧缓冲区像素都包含一个或多个采样点，这些采样点与第3.4.2节（其他线段特性）中抗锯齿部分描述的矩形区域相交。若启用线条点状填充，则该矩形区域会被划分为相邻的单位长度矩形，其中部分矩形将根据第3.4.2节所述流程进行剔除——该流程中"片段"被替换为"矩形"。 覆盖位对应于与保留矩形相交的采样点——光照值为1，其余覆盖位均为0。每种颜色、深度及纹理坐标集均通过将对应采样位置代入公式3.5计算，再利用结果评估公式3.7生成。 实现方案可选择将相同颜色值和纹理坐标集分配给多个采样点：通过在像素内任意位置（包括片段中心或任意采样点）计算式3.5，再代入式3.6即可。颜色值与纹理坐标集无需在相同位置计算。

线宽范围与渐变级数等效于抗锯齿线条所支持的参数。

3.5 多边形

多边形可由多边形**开始**/结束对象生成，三角形可由三角带、三角扇或独立三角形序列生成，四边形则可由四边形带、独立四边形序列或**矩形**命令生成。与点和线段类似，多边形的光栅化过程由多个变量控制。 多边形抗锯齿功能通过**启用**/**禁用**符号常量`POLYGON SMOOTH`控制。对应于线段点状填充的多边形处理方式称为多边形点状填充，详见下文。

3.5.1 基本多边形光栅化

多边形光栅化的第一步是判断该多边形属于*背面朝向*还是*正面朝向*。该判断通过检查第2.14.1节公式2.6计算出的面积符号来实现（包括根据最后一次调用**FrontFace函数**指示的符号可能反转的情况）。 若该符号为正，则多边形为正面朝向；否则为背面朝向。此判定需结合**CullFace**启用位与模式值，共同决定特定多边形是否进行光栅化。**CullFace**模式通过调用

```
void CullFace( enum mode );
```

模式为符号常量：取值为`FRONT`、`BACK`或`FRONT AND BACK`。通过符号常量启用或**禁用**剔除功能 _

CULL FACE。当前面朝向的多边形在以下任一情况下进行光栅化：剔除功能禁用，或**CullFace**模式为BACK；而背面朝向的多边形仅在以下任一情况下进行光栅化：剔除功能禁用，或**CullFace**模式为FRONT。**CullFace**模式的初始设置为BACK。初始状态下，剔除功能处于禁用状态。

确定哪些片段由多边形光栅化产生的规则称为*点采样*。通过取多边形顶点的*x*和y窗口坐标，形成二维投影。 位于该多边形内部的片段中心将通过光栅化生成。对于片段中心位于多边形边界上的情况需特殊处理：当两个多边形分别位于片段中心所在的公共边（端点相同）两侧时，要求在光栅化过程中仅其中一个多边形能生成该片段。

关于将多边形栅格化后生成的每个片段相关数据，我们首先定义三角形内片段值的生成方式。 定义三角形的重心*坐标*。重心坐标由三个数值*a*、*b*、*c*组成，每个数值均在[0, 1]区间内，且*满足*$a + b + c = 1$。这些坐标可唯一确定三角形内部或边界上的任意*点p*。

$$p = ap_a + bp_b + cp_c,$$

其中 p_a , p_b , 和 p_c 是三角形的顶点。*a*, *b*, 和 *c* 可通过以下方式求得：

$$a = \frac{A(pp_bp_c)}{A(p_ap_bp_c)} \qquad b = \frac{A(pp_ap_c)}{A(p_ap_bp_c)} \qquad c = \frac{A(pp_ap_b)}{A(p_ap_bp_c)}$$

其中 A(*lmn*) 表示以 l, m, n 为顶点的三角形在窗口坐标系中的面积。

l、*m*、*n*。

将数据点 p_a、p_b或 p_c 分别记为 f_a、f_b或 f_c。此时，数据点在三角形光栅化生成的片段上的值 *f* 由三角形光栅化生成的片段上数据点的值由以下公式给出：

$$f = \frac{af_a/w_a + bf_b/w_b + cf_c/w_c}{a/w_a + b/w_b + c/w_c} \tag{3.8}$$

其中 w_a 、w_b 和 w_c 分别是 p_a 、p_b 和 $p_{(c)}$ 的剪裁 *w* 坐标。*a*、*b* 和 *c* 是生成数据的片段的重心坐标。*a*、*b* 和 *c* 必须与片段中心的精确坐标完全对应。 换言之，与片段相关的数据必须在片段中心处采样。

与线段光栅化类似，方程 3.8 可近似为

$$f = af_a + bf_b + cf_c;$$

对于颜色值而言，此方法可能产生可接受的结果（深度值*必须*采用此方法），但若用于纹理坐标或*裁剪*w坐标，通常会导致不可接受的失真效应。

对于边数超过三条的多边形，我们仅要求通过多边形顶点处基准值的凸组合，即可获得光栅化算法生成的每个片段所对应的值。即必须满足：在每个片段

$$f = \sum_{i=1}^{n} a_i f_i$$

其中 n 为多边形顶点数，f_i为顶点处*函数值* i；对于每个 i，$0 \le a_i \le 1$ 且 $\sum_{i=1}^{n} a_i = 1$。a_i的值可能与 i 不同，但在顶点 i 上，$a_j = 0$，$j \neq i$ 且 $a_i = 1$。

实现所需行为的一种算法是将多边形三角剖分（不添加任何顶点），然后如前所述分别处理每个三角形。 一种满足限制条件的扫描线光栅化器是：沿每条边线性插值数据，再在相邻边之间的水平跨度内进行线性插值（此时方程3.8的分子与分母需独立迭代，并对每个片段执行除法运算）。

3.5.2 点描法

多边形点状填充与线状点状填充原理相似，通过屏蔽光栅化产生的特定片段，使其不被传递至GL的下一阶段。无论多边形抗锯齿状态如何，此机制均适用。点状填充通过以下函数控制：

```
void PolygonStipple( ubyte *pattern );
```

*模式*是一个指向内存的指针，其中封装了一个32× ×32的模式。该图案将根据第3.6.4节中**DrawPixels命令的**处理流程从内存中解包；其效果等同于向该命令传递*高度*与*宽度*均为32、*类型*为BITMAP、*格式*为COLOR INDEX的参数。解包后的原始值（在执行任何转换或运算前）构成由0和1组成的点阵图案。

若x_w*和*y_w是栅格化多边形片段的窗口坐标，则该片段仅当模式位$(x_w \bmod 32, y_w \bmod 32)$为1时才会被发送至GL的下一阶段。

多边形点状填充可通过常量POLYGON STIPPLE**启用**或禁用。禁用时，效果等同于点状填充图案全为1。

3.5.3 抗锯齿

多边形抗锯齿通过在多边形内部与该片段的正方形相交处生成片段来进行光栅化。 每个片段均计算覆盖值，该值将被保存以便按第3.12节所述应用。通过将数据值在片段方块与多边形内部的交界区域进行积分，并将该积分值除以交界面积，即可为片段分配关联数据。 对于完全位于多边形内部的片段正方形，可直接采用片段中心处的数据值，无需进行片段范围内的积分计算。无论多边形是否启用抗锯齿，多边形点状填充的运作方式保持一致。然而，第3.5.1节定义的多边形点采样规则则对于抗锯齿多边形，此规则不予强制执行。

3.5.4 控制多边形光栅化的选项

多边形光栅化处理的解释通过以下方式控制：

```
void PolygonMode( enum face, enum mode );
```

face 取值为 FRONT、BACK 或 FRONT AND BACK，分别表示*模式所*描述的栅格化方法将覆盖面向多边形、背向多边形或正反两面多边形的栅格化方法。*mode* 取值为符号常量 POINT、LINE 或 FILL。 调用 **Polygon-Mode** 时若指定 POINT，则多边形中某些顶点在光栅化处理时会被视为被 **Begin**(POINT) 和 **End** 指令对包围。接受此处理的顶点是那些被标记为多边形边界起始点的顶点（参见第 2.6.2 节）。LINE 模式则将标记为边界的边栅化为线段。 （多边形首个光栅化边起始处会重置线点阵计数器，后续边则不会。）FILL是多边形光栅化的默认模式，对应第3.5.1、3.5.2和3.5.3节的描述。 请注意这些模式仅影响多边形的最终光栅化：具体而言，在应用这些模式之前，多边形的顶点会进行光照计算，且多边形会先经过裁剪操作，可能还会执行剔除处理。

多边形抗锯齿仅适用于**PolygonMode**的FILL状态。对于POINT或LINE状态，则分别应用点抗锯齿或线段抗锯齿。

3.5.5 深度偏移

通过光栅化生成的所有片段的深度值，可通过为该多边形计算的单一值进行偏移。该值由以下函数确定：

```
void PolygonOffset( float factor, float units );
```

*因子*用于缩放多边形的最大深度斜率，*单位*用于缩放与深度缓冲区可用分辨率相关的实现相关常量。最终值相加产生多边形偏移量。*因子*和*单位*均可为正负值。

三角形的最大深度斜率 m 为

$$m = \sqrt{\left(\frac{\partial z_w}{\partial x_w}\right)^2 + \left(\frac{\partial z_w}{\partial y_w}\right)^2} \tag{3.9}$$

其中 (x_w, y_w, z_w) 是三角形上的一个点。m 可近似为

$$m = \max\left(\left|\frac{\partial z_w}{\partial x_w}\right|, \left|\frac{\partial z_w}{\partial y_w}\right|\right)。 \tag{3.10}$$

若多边形拥有超过三个顶点，在光栅化过程中可能使用一个或多个*m值*。每个m值可在[*min*,*max*]区间内取任意数值，其中*min*和*max*分别是通过对所有三顶点组合形成的三角形求解方程3.9或方程3.10所得的最小值与最大值。

最小可分辨差异 r 是实现常量。它是窗口坐标 z 值中保证在整个多边形光栅化过程及深度缓冲区中保持唯一性的最小差异。当两个多边形的光栅化生成的所有片元对（其顶点完全相同，但 z_w 值相差 r）都将具有不同的深度值。

多边形的偏移量*o*为

$$o = m * 因子 + r * 单位。 \tag{3.11}$$

m 根据上述描述计算得出，作为深度值在 [0,1] 范围内的函数，而 o 则应用于同一范围内的深度值。

布尔状态值 POLYGON OFFSET POINT、POLYGON OFFSET LINE 和 POLYGON OFFSET FILL 决定在点模式、线模式和填充模式下进行多边形光栅化时是否应用 *o*。这些布尔状态值通过作为 **Enable** 和 **Disable** 命令的参数值来启用或禁用。 若启用多边形偏移点，则在点模式下栅格化多边形时，*o*值将被添加至每个片段的深度值。同理，若启用多边形偏移线或多边形偏移填充，则在线模式或填充模式下栅格化多边形时，*o*值将分别被添加至对应片段的深度值。

片段深度值始终限制在[0,1]范围内，具体方式有两种：一是偏移量累加后进行截断（推荐方案），二是直接对多边形光栅化过程中使用的顶点值进行截断。

3.5.6 多边形多采样光栅化

如果启用了MULTISAMPLE且SAMPLE BUFFERS的值为1，则无论多边形抗锯齿（POLYGON SMOOTH）是否启用，多边形都将采用以下算法进行光栅化： 多边形光栅化会为每个帧缓冲区像素生成一个片段，该像素包含一个或多个满足第3.5.1节所述点采样准则的采样点，其中包含对位于多边形边界边缘的采样点的特殊处理。 若多边形基于其方向和**剔除面**模式被剔除，则光栅化过程中不会生成任何片段。片段剔除机制与带锯齿及抗锯齿多边形相同，均通过多边形点阵点进行处理。

满足点采样准则的采样点对应的覆盖位为1，其余覆盖位为0。每种颜色、深度及纹理坐标集均通过将对应采样位置代入第3.5.1节所述的重心方程生成，该过程采用省略w分量的方程3.8近似值。 实现方案可选择通过重心评估将相同颜色值和纹理坐标集分配给多个采样点，评估时可使用包含片段中心或任一采样点的像素位置。颜色值与纹理坐标集无需在同一位置进行评估。

上述光栅化仅适用于**多边形模式的**填充状态。对于点和线，则适用第3.3.3节（点多采样光栅化）和第3.4.4节（线多采样光栅化）所述的光栅化方式。

3.5.7 多边形光栅化状态

多边形光栅化所需状态包含：多边形点阵图案、点阵化启用/禁用状态、当前多边形抗锯齿状态（启用/禁用）、正向/背向多边形当前PolygonMode设置值、点/线/填充模式多边形偏移量启用/禁用状态，以及多边形偏移方程的因子与偏移量值。 初始点阵图案为全1；初始状态下点阵效果禁用。多边形抗锯齿初始设置为禁用。前后向多边形**PolygonMode**初始状态均为填充模式。初始偏移因子与偏移量均为0；所有模式下初始状态均为偏移禁用。

3.6 像素矩形

具有颜色、深度及其他特定值的矩形**可通过**DrawPixels命令（详见第3.6.4节）转换为片段。部分**控制DrawPixels操作**的参数与运算逻辑，亦适用于**ReadPixels**（用于从帧缓冲区获取像素值）**和CopyPixels**（用于在帧缓冲区不同位置间复制像素）；但关于**ReadPixels和CopyPixels**的详细说明将推迟至第4章，待帧缓冲区机制阐述完毕后再行讨论。不过在本节中，当涉及**DrawPixels**的参数与状态同样适用于ReadPixels或**CopyPixels**时，我们将予以特别标注。

若干参数控制客户端内存中像素的编码方式（用于读写操作），以及像素在放入帧缓冲区前或从帧缓冲区读取后（涉及读写与复制操作）的处理方式。这些参数通过三条指令设置：**PixelStore**、**PixelTransfer** 和 **PixelMap**。

3.6.1 像素存储模式

像素存储模式会影响执行**DrawPixels和**ReadPixels命令（以及其他命令；参见第3.5.2、3.7和3.8节）时的操作。若将命令放入显示列表（参见第5.4节），其生效时间可能与命令执行时间不同。像素存储模式通过以下方式设置：

```
void PixelStore{if}( enum pname, T param );
```

pname 是表示待设置参数的符号常量，*param* 是其设置值。表 3.1 汇总了像素存储参数的类型、初始值及允许范围。若将参数设置为超出指定范围的值，将导致 `INVALID VALUE` 错误。

参数名称	类型	初始值	有效范围
UNPACK_SWAP_BYTES	布尔值	FALSE	TRUE/FALSE
按低位优先解包	布尔值	FALSE	TRUE/FALSE
解包行长度	整数	0	[0, ∞)
解包跳过行数	整数	0	[0, ∞)
UNPACK_SKIP_PIXELS	整数	0	[0, ∞)
展开对齐	整数	4	1,2,4,8
解包图像高度	整数	0	[0, ∞)
解包跳过图像	整数	0	[0, ∞)

表 3.1： 与 **DrawPixels**、 **Col-orTable**、 **ColorSubTable**、 **ConvolutionFilter1D**、 **ConvolutionFilter2D**、 **SeparableFilter2D**、 **PolygonStipple**、 **TexImage1D**、 **TexImage2D**、 **TexImage3D**、 **Tex-SubImage1D**、 **TexSubImage2D** 和 **TexSubImage3D** 相关的一个或多个 **PixelStore** 参数。

采用浮点数版本**的PixelStore**可设置任意类型参数：若参数为布尔型，则传入值为0.0时设为FALSE，否则设为TRUE；若参数为整数型，则传入值将四舍五入至最接近的整数。 该命令的整数版本同样可用于设置任意类型的参数：若参数为布尔型，则传入值为0时设为FALSE，否则设为TRUE；若参数为浮点型，则传入值将转换为浮点数。

3.6.2 图像子集

某些像素传输和片段级操作仅在包含可选*成像子集*的GL实现中可用。 成像子集包含新增命令及现有命令可选用的新增枚举值。若实现支持该子集，则所有相关调用与枚举值均须遵循GL规范后续章节所述方式实现。若未支持该子集，调用任何未支持命令将触发INVALID OPERATION错误，使用任何新增枚举值将触发INVALID ENUM错误。

仅在成像子集可用的新操作详见第3.6.3节。成像子集操作包括：

1. 颜色表相关命令，涵盖子章节**《颜色表规范》**《**替代颜色表规范**》《**颜色表**查询》《卷积后颜色表查询》《卷积后**颜色矩阵颜色表查询**》所述的所有命令及枚举值，以及第6.1.7节所述的查询命令。

命令、**色表状态与代理状态**、**色表查找**、**卷积后色表查找**、**后色矩阵色表查找**，以及第6.1.7节所述的查询命令

2. 卷积，包括**卷积滤波器规范**、**替代卷积滤波器规范命令**及**卷积**子章节所述的所有命令与枚举值，以及第6.1.8节所述的查询命令。

3. 颜色矩阵，包括子章节《**颜色矩阵规范**》和《**颜色矩阵转换**》中描述的所有命令和枚举项，以及第6.1.6节中描述的简单查询命令。

4. 直方图与最小最大值，包括子章节**直方图表规范**、**直方图状态与代理状态**、**直方图**、**最小最大值表**规范及**最小最大值**中描述的所有命令与枚举值，以及第6.1.9节和第6.1.10节所述的查询命令。

仅当EXTENSIONS字符串包含子字符串"ARB imaging"时才支持成像子集。EXTENSIONS的查询方法详见第6.1.11节。

若未支持成像子集，相关像素传输操作将不执行；像素将保持不变传递至后续操作。

3.6.3 像素传输模式

像素传输模式会**影响DrawPixels**（第3.6.4节）、**ReadPixels**（第4.3.2节）**和CopyPixels**（第4.3.3节）在执行时的工作方式（该时间点可能与命令发出时间不同）。

某些像素传输模式通过以下方式设置：

```
void PixelTransfer{if}( enum param, T value );
```

param 是表示待设置参数的符号常量，*value* 是其设置值。表 3.2 汇总了通过 **PixelTransfer** 设置的像素传输参数、其类型、初始值及允许范围。若参数值超出指定范围，将引发 INVALID VALUE 错误。该命令与 **PixelStore** 具有相同版本，且采用相同规则接受并转换传递值以设置参数。

像素映射查找表通过以下命令设置：

```
void PixelMap{ui us f}v( enum map, sizei size, T values );
```

参数名称	类型	初始值	有效范围
地图颜色	布尔值	FALSE	TRUE/FALSE
地图模板	布尔值	FALSE	TRUE/FALSE
索引移位	整数	0	(-∞, ∞)
索引偏移	整数	0	(-∞, ∞)
x 比例	浮点	1.0	(-∞, ∞)
深度缩放	浮点	1.0	(-∞, ∞)
x 偏移量	浮点	0.0	(-∞, ∞)
深度偏移	浮点	0.0	(-∞, ∞)
卷积后缩放*因子*	浮点	1.0	(-∞, ∞)
卷积后*乘以*偏置	浮点	0.0	(-∞, ∞)
POST COLOR MATRIX *x* SCALE	浮点数	1.0	(-∞, ∞)
后处理颜色矩阵 × 偏移量	浮点	0.0	(-∞, ∞)

表 3.2：**像素**传输参数。*x* 表示红色、绿色、蓝色或透明度。

map 是符号映射名称，表示要设置的映射；*size* 表示映射的大小；*values* 是指向*大小为* map 的值的数组的指针。

表项可采用三种类型之一进行指定：单精度浮点数、无符号短整型或无符号整型，具体取决于调用的 PixelMap版本。表项在指定时将转换为对应类型。提供颜色分量值的表项按表2.9进行转换；提供颜色索引值的表项则从无符号短整型或无符号整型转换为浮点数。 提供模板索引的条目将通过四舍五入转换为整数，转换类型为单精度浮点数。各类表格及其初始大小与条目汇总于表3.3。以索引作为地址的表格必须*满足size* $=2^n$，否则将引发INVALID VALUE错误。 每个表的最大允许*大小*由实现相关的值MAX PIXEL MAP TABLE规定，但必须至少为32（所有表共享单一最大值）。若**向PixelMap**传递*的大小*超过实现最大值或小于1，则触发INVALID VALUE错误。

颜色表规范

颜色查找表通过以下方式指定：

```
void ColorTable( enum target, enum internalformat, sizei width, enum format,
    enum type, void *data );
```

映射名称	地址	值	初始大小	初始值
像素映射 I 到 I	颜色索引	颜色索引	1	0.0
像素图 S 到 S	模板索引	模板索引	1	0
像素图 I 到 R	颜色索引	R	1	0.0
像素映射 I 到 G	颜色索引	G	1	0.0
像素图 I 至 B	颜色索引	B	1	0.0
像素映射 I 到 A	颜色索引	A	1	0.0
像素映射 R 到 R	R	R	1	0.0
像素图 G 到 G	G	G	1	0.0
像素图 B 到 B	B	B	1	0.0
像素图 A 到 A	A	A	1	0.0

表 3.3： 像素映射参数。

*目标*必须是表3.4中列出的常规颜色表名称之一，用于定义该表。代理表名称是本节后文将讨论的特殊情况。*宽度*、*格式*、类型和*数据*参数用于指定内存中的图像，其含义和允许值**与DrawPixels函数的**对应参数相同（参见3.6.4节），此时*高度*默认为1。 表格的最大允许*宽度*取决于具体实现，但必须至少为32。不允许使用COLOR INDEX、DEPTH COMPONENT、STENCIL INDEX格式以及BITMAP*类型*。

指定图像从内存中读取并进行处理，其过程如同调用**DrawPixels**函数，在最终转换为RGBA格式后停止。随后，每个像素的R、G、B和A通道值将分别乘以四个COLOR TABLE SCALE参数，偏移四个COLOR TABLE BIAS参数，并限制在[0, 1]范围内。这些参数通过调用**ColorTableParameterfv函数**进行设置，具体方法如下所述。

随后从生成的R、G、B和A值中选取组件，以与纹理处理相同的方式（参见第3.8.1节），生成采用*由internalformat*指定（或由此派生）的*基础内部格式的*表格。internalformat必须是表3.15或表3.16中除DEPTH格式外的任意格式。

颜色查找表被重新定义为包含 *width* 个条目，每个条目采用指定的内部格式。该表通过索引 0 至 *width* 1 构建。表中位置 *i* 由第 i 个图像像素指定（从零开始计数）。

若*宽度*不为零或非非负二的幂次方，则生成错误INVALID VALUE。若指定的颜色查找表超出实现支持范围，则生成错误TABLE TOO LARGE。

调用以下函数可指定表的缩放系数和偏移量参数：

表名	类型
颜色表 _ 卷积后颜色表后颜色矩阵颜色表 _ _ _ _ _ _	常规
代理颜色表 _ 代理卷积后颜色表代理后颜色矩阵颜色表 _ _ _ _ _ _ _	代理

表 3.4： 颜色表名称。常规表关联图像数据。代理表不包含图像数据，仅用于判断图像能否加载至对应常规表。

```
void ColorTableParameter if v( enum target, enum pname, T params );
                                                    { }
```

*目标*必须是常规颜色表名称。*pname* 参数取值为 `COLOR TABLE SCALE` 或 `COLOR TABLE BIAS` 之一。*params* 指向包含四个值的数组：红、绿、蓝、透明度，按此顺序排列。

图形处理器实现可根据任意ColorTable参数调整内部分量分辨率的分配，但该分配不得受其他因素影响，且一旦确定便不可更改。分配规则必须保持不变：每次使用相同参数值指定颜色表时，必须采用相同的分配方案。这些分配规则同样适用于本节后文所述的代理颜色表。

替代色表指定命令

颜色表也可通过直接从帧缓冲区获取的图像数据来指定，现有表的部分内容可重新定义。

命令

```
void CopyColorTable( enum target, enum internalformat, int x, int y, sizei width );
```

定义颜色表的方式**与ColorTable**完全相同，区别在于表数据取自帧缓冲区而非客户端内存。*target*必须是常规颜色表名称。*x*、*y*和**width与CopyPixels的**对应参数完全一致（参见第4.3.3节）；它们指定图像*宽度*以及待复制帧缓冲区区域的左下角(*x, y*)坐标。

图像从帧缓冲区提取的过程，完全等同于将这些参数传递给**CopyPixels**函数时：参数*类型*设为COLOR，*高度*设为1，并在最终转换为RGBA格式后停止操作。

后续**处理与ColorTable的**描述完全相同，首先通过COLOR TABLE SCALE进行缩放。参数*target*、*internalformat*和width采用**与ColorTable**对应参数相同的数值及含义。*格式*默认为RGBA。

新增两条命令：

```
void ColorSubTable( enum target, sizei start, sizei count, enum format, enum type,
    void *data );
void CopyColorSubTable( enum target, sizei start, int x, int y, sizei count );
```

仅重新指定现有颜色表的一部分。指定颜色表的*内部*格式或*宽度*参数不会改变，表中指定部分之外的条目也不会改变。*目标*必须是常规颜色表名称。

ColorSubTable命令的*参数格式*、类型和*数据*均**与ColorTable命令的**对应参数一致，这意味着它们采用相同的数值进行指定，且具有相同含义。 同样地，**CopyColorSub**Table命令的*参数*x、*y*和count**与CopyColorTable命令的**x、y和width参数相对应。这两组ColorSubTable命令对像素组的解释和处理方式完全遵循**其ColorTable**对应命令的模式，唯一区别在于：R、G、B和A像素组值向颜色表组件的映射由表的*内部格式*控制，而非通过命令参数设定。

参数start和count用于定义颜色表的子区域，该区域从索引*start*开始，至索引*start* + count + 1结束。采用零起始计数，第n个像素组被分配至索引*为count* + *n的*表项。若*start* + *count* > *width*，则生成错误INVALID VALUE。

颜色表状态与代理状态

颜色表所需的状态可分为两类。对于三个表中的每个表，都存在一个数值数组。每个数组关联着一个宽度（描述表内部格式的整数）、六个整数值（描述表中红、绿、蓝、透明度、亮度和强度各组件的分辨率），以及两组各四个浮点数（用于存储表的缩放因子和偏移量）。 初始数组均为空数组（零宽度，内部格式为

RGBA格式且各分量为零）。缩放参数初始值为(1,1,1,1)，偏移参数初始值为(0,0,0,0)。

除颜色查找表外，系统还维护部分实例化的代理颜色查找表。每张代理表包含宽度和内部格式状态值，以及红、绿、蓝、Alpha、亮度和强度分量分辨率的状态。 代理表不包含图像数据，也不包含缩放和偏移参数。当ColorTable执行时若将*目标*指定为表3.4中列出的代理颜色表名称之一，该表的代理状态值将被重新计算并更新。若表体积过大，系统不会报错，但代理格式、宽度及各分量分辨率将被设为零。 若该颜色表可被目标参数设为对应常规表名的ColorTable调用所兼容（例如COLOR TABLE即为PROXY COLOR TABLE对应的常规名称），则代理状态值将完全按常规表指定方式设置。使用代理*目标*调用ColorTable不会影响任何实际颜色表的图像或状态。

所有代理目标均未关联图像。它们不能用作颜色表，且绝不能通过 **GetColorTable** 进行查询。若尝试此操作，将生成 INVALID ENUM 错误。

卷积滤波器规格

二维卷积滤波器图像通过调用以下函数指定：

```
void ConvolutionFilter2D( enum target, enum internalformat, sizei width, sizei height,
    enum format, enum type, void *data );
```

*目标*必须为CONVOLUTION 2D。*width*、*height*、*format*、*type*和*data*参数用于指定内存中的图像，其含义和允许值与**DrawPixels函数**中对应参数相同。禁止使用COLOR INDEX、DEPTH COMPONENT、STENCIL INDEX格式以及BITMAP*类型*。

指定图像从内存中提取并处理，如同调用**DrawPixels**函数般执行，直至最终转换为RGBA格式后停止。随后，每个像素的R、G、B和A分量将分别乘以四个二维卷积滤波器缩放参数，并偏移四个二维卷积滤波器偏置参数。这些参数通过调用**ConvolutionParameterfv函数**设置，具体方法如下所述。 整个过程中均不进行裁剪操作。

随后从生成的R、G、B、A值中选取分量，以与纹理处理（第3.8.1节）相同的方式，生成*符合internalformat*指定（或由此派生）*的基础内部格式*的数据表。internalformat必须

为表3.15或表3.16中除深度格式外的任意格式。

像素的红、绿、蓝、透明度、亮度及/或强度分量以浮点数而非整数格式存储。它们构成一个二维图像，通过坐标、*j索引，其中i*从左向右递增（起始值为零），*j*从下向上递增（起始值为零）。图像*位置*, j由第N个像素（从零开始计数）指定，其中

$$N = i + j * 宽度$$

若*宽度*或*高度*超过最大支持值，则会生成 INVALID VALUE 错误。可通过 **Get-ConvolutionParameteriv** 查询这些值，将 *target* 设为 CONVOLUTION 2D，并将 *pname* 分别设为 MAX CONVOLUTION WIDTH 或 MAX CONVOLUTION HEIGHT。

二维滤波器的缩放系数与偏置参数通过调用

```
void ConvolutionParameter if v( enum target, enum pname, T params );
                                                    { }
```

使用*目标*卷积2D。*pname*可为卷积滤波器尺度或卷积滤波器偏置。*params*指向一个包含四个值的数组：红、绿、蓝、透明度，按此顺序排列。

一维卷积滤波器通过以下方式定义：

```
void ConvolutionFilter1D( enum target, enum internalformat, sizei width, enum format,
    enum type, void *data );
```

target必须为CONVOLUTION 1D。*internalformat*、*width*、*format和type*的语义与二维对应参数完全一致，接受相同值。但data必须指向一维图像。

图像从内存中提取出来，并像调用*高度*为1的**ConvolutionFilter2D**那样进行处理，但会根据一维卷积滤波器缩放因子和卷积滤波器偏置因子进行缩放和偏置。

参数进行缩放和偏移。这些参数的指定方式与二维参数完全相同，区别在于调用ConvolutionParameterfv时需指定目标 CONVOLUTION 1D。

图像通过坐标 *i* 形成，其中 *i* 从左向右递增，

起始值为零。图像*位置*由第i个像素点指定，计数从零开始。

若*宽度*超过最大支持值，则会生成INVALID VALUE错误。该值可通过**调用GetConvolutionParameteriv**获取，需*将target*设为CONVOLUTION 1D，*并将pname*设为MAX CONVOLUTION WIDTH。

针对二维可分离滤波器的定义提供了特殊支持——此类滤波器的图像可表示为两个一维图像的乘积，而非完整的二维图像。二维可分离卷积滤波器通过以下函数定义：

```
void SeparableFilter2D( enum target, enum internalformat, sizei width, sizei height,
    enum format, enum type, void *row, void *column );
```

*目标*必须为可分离二维图像。internalformat指定将保留的两个一维图像的表项格式。row指向*指定格式*和*类型的宽度*像素宽图像。column指向指定*格式*和*类型*的高度像素高图像。

这两幅图像从内存中提取出来，并分别按单独调用**Convolu-tionFilter1D**的方式处理，但每幅图像都会通过二维可分离的卷积滤波器缩放参数和卷积滤波器偏置参数进行缩放和偏置。这些参数的指定方式与一维和二维参数完全相同，只是调用Convolution-Parameteriv时将*目标*设为SEPARABLE 2D。

替代卷积滤波器指定命令

一维和二维滤镜也可通过直接从帧缓冲区获取的图像数据进行指定。

命令

```
void CopyConvolutionFilter2D( enum target,
    enum 内部格式, int x, int y, sizei 宽度, sizei 高度 );
```

定义了一个**与ConvolutionFilter2D**完全相同的二维滤波器，区别在于图像数据取自帧缓冲区而非客户端内存。*target*必须为CONVOLUTION 2D。*x*、*y*、width和*height*与**CopyPixels**的对应参数完全一致（参见第4.3.3节）；它们指定图像的宽度和*高度*，以及要复制的帧缓冲区区域左下角的(*x, y*)坐标。图像从帧缓冲区获取的过程完全等同于将这些参数传递给**CopyPixels函数**时将参数*类型*设为COLOR，并在最终扩展为RGBA格式后停止。

后续处理**与ConvolutionFilter2D的**描述完全相同，首先通过卷积滤波器缩放因子进行缩放。*目标参数target*、*内部格式in-ternalformat*、宽度width和*高度height*采用相同数值进行指定，其含义与ConvolutionFilter2D的等效参数一致。格式默认为RGBA。

*格式*默认为RGBA。

命令

```
void CopyConvolutionFilter1D( enum target,
    enum 内部格式, int x, int y, sizei 宽度);
```

定义了一维滤波器，其方式**与ConvolutionFilter1D**完全相同，区别在于图像数据取自帧缓冲区而非客户端内存。target必须为CONVOLUTION 1D。*x*、*y*和*width*参数**与CopyPixels的**对应参数完全一致（参见第4.3.3节）；它们指定图像*宽度*及待复制帧缓冲区区域的左下角坐标(*x, y*)。图像从帧缓冲区提取的过程完全等同于将这些参数传递给**CopyPixels**函数时：参数*类型*设为COLOR，*高度*设为1，并在最终扩展为RGBA格式后停止。

后续处理**与ConvolutionFilter1D的**描述完全相同，首先通过卷积滤波器缩放因子进行缩放。*参数target*、*internalformat*和width采用与**ConvolutionFilter2D**对应参数相同的数值及含义进行指定。*格式*默认为RGBA。

卷积滤波器状态

卷积滤波器所需的状态包括：一维图像数组、可分离滤波器的两个一维图像数组以及二维图像数组。每个滤波器关联以下参数：宽度与高度（仅限二维及可分离滤波器）、描述滤波器内部格式的整数，以及两组各含四个浮点数的数组用于存储滤波器尺度和偏置值。

初始卷积滤波器均为空（宽度和高度均为零，内部格式为RGBA，各分量尺寸为零）。所有缩放参数的初始值均为(1,1,1,1)，所有偏移参数的初始值均为(0,0,0,0)。

颜色矩阵规范

将矩阵模式设为COLOR时，第2.11.2节所述的矩阵运算将作用于颜色矩阵堆栈顶层矩阵。所有矩阵运算对颜色矩阵的效果与对其他矩阵相同。

直方图表规范

直方图表通过以下方式指定：

```
void Histogram( enum target, sizei width, enum internalformat,
    boolean sink );
```

target 必须为 HISTOGRAM 才能指定直方图表。*target* 值 PROXY HISTOGRAM 是本节后文将讨论的特殊情况。*width* 指定直方图表的条目数量，*internalformat* 指定每个表条目的格式。 直方图表的最大允许*宽度*取决于具体实现，但必须至少为32。*sink*参数决定像素组在直方图操作中的处理方式：TRUE表示消耗，FALSE表示传递至minmax操作。

若直方图操作执行无误，则指定的直方图表将被重新定义为包含 *width* 个条目，每个条目均采用指定的内部格式。条目索引范围为 0 至 width 1。每个条目的各分量均被置零。先前直方图表中的值（若有）将丢失。

若*宽度*不为零或非非负的2的幂次方，则会生成错误INVALID VALUE。若指定的直方图表对实现而言过大，则会生成错误TABLE TOO LARGE。 若 *internalformat* 不属于表 3.15 或表 3.16 中的格式，或为 1、2、3、4 以及上述表格中的任何 DEPTH 或 INTENSITY 格式，则会生成错误 INVALID ENUM。

GL实现可根据**任意直方**图参数调整内部组件的分辨率分配，但该分配不得受其他因素影响，且一旦确定便不可更改。特别地，分配必须保持不变：当使用相同参数值指定直方图时，每次都必须采用相同的分配方案。这些分配规则同样适用于代理直方图（详见本节后续内容）。

直方图状态与代理状态

直方图操作所需的状态是一个数组，其中包含关联的宽度值、描述直方图内部格式的整数、描述表格中红、绿、蓝、透明度及亮度各组件分辨率的五个整数值，以及一个标记操作是否消耗像素组的标志位。初始数组为空（零宽度，内部格式为RGBA，各组件尺寸为零）。 标志的初始值为false。

除直方图表外，系统还维护一个部分实例化的代理直方图表。该表包含宽度、内部格式以及红、绿、蓝、透明度和亮度分量的分辨率。 代理表不包含图像数据或标志位。当以PROXY HISTOGRAM为*目标*执行**直方图**操作时，代理状态值将被重新计算并更新。若直方图数组过大，系统不会报错，但代理格式、宽度及各通道分辨率将

置零。若直方图表可被*目标*设为HISTOGRAM的**直方图**调用所容纳，则代理状态值将完全按实际直方图表的规格设置。调用*目标*为PROXY HISTOGRAM的**直方图**操作不会影响实际直方图表。

PROXY HISTOGRAM 没有关联的图像。它不能作为直方图使用，且其图像绝不能通过 **GetHistogram** 进行查询。若尝试此操作，将导致 INVALID ENUM 错误。

–

最小最大值表规范

最小最大表通过以下方式指定：

```
void Minmax( enum target, enum internalformat, boolean sink );
```

*target*必须为MINMAX。*internalformat*指定表项的格式。*sink*指定像素组是否被minmax操作消耗（TRUE）或传递至最终转换（FALSE）。

若 *internalformat* 不属于表 3.15 或表 3.16 中的格式，或为 1、2、3、4，或为这些表格中的任何 DEPTH 或 INTENSITY 格式，则会生成 INVALID ENUM 错误。生成的表格始终包含 2 个条目，每个条目的值仅对应内部格式的组件。

最小最大操作所需的初始状态包含：一个存储两个元素的表（第一个元素存储最小值，第二个存储最大值）、描述表内部格式的整数，以及指示操作是否消耗像素组的标志位。初始状态为：最小值表条目设为可表示的最大值，最大值表条目设为可表示的最小值。 内部格式默认设置为RGBA，标志初始值为false。

3.6.4 像素矩形的渲染过程

图3.7展示了绘制主机内存中编码像素的过程。我们将按发生顺序描述该过程的各个阶段。

像素绘制使用

```
void DrawPixels( sizei width, sizei height, enum format, enum type, void *data );
```

format 是表示内存中值含义的符号常量。

width 和 *height* 分别表示要进行颜色表查找的像素矩形的宽度和高度。

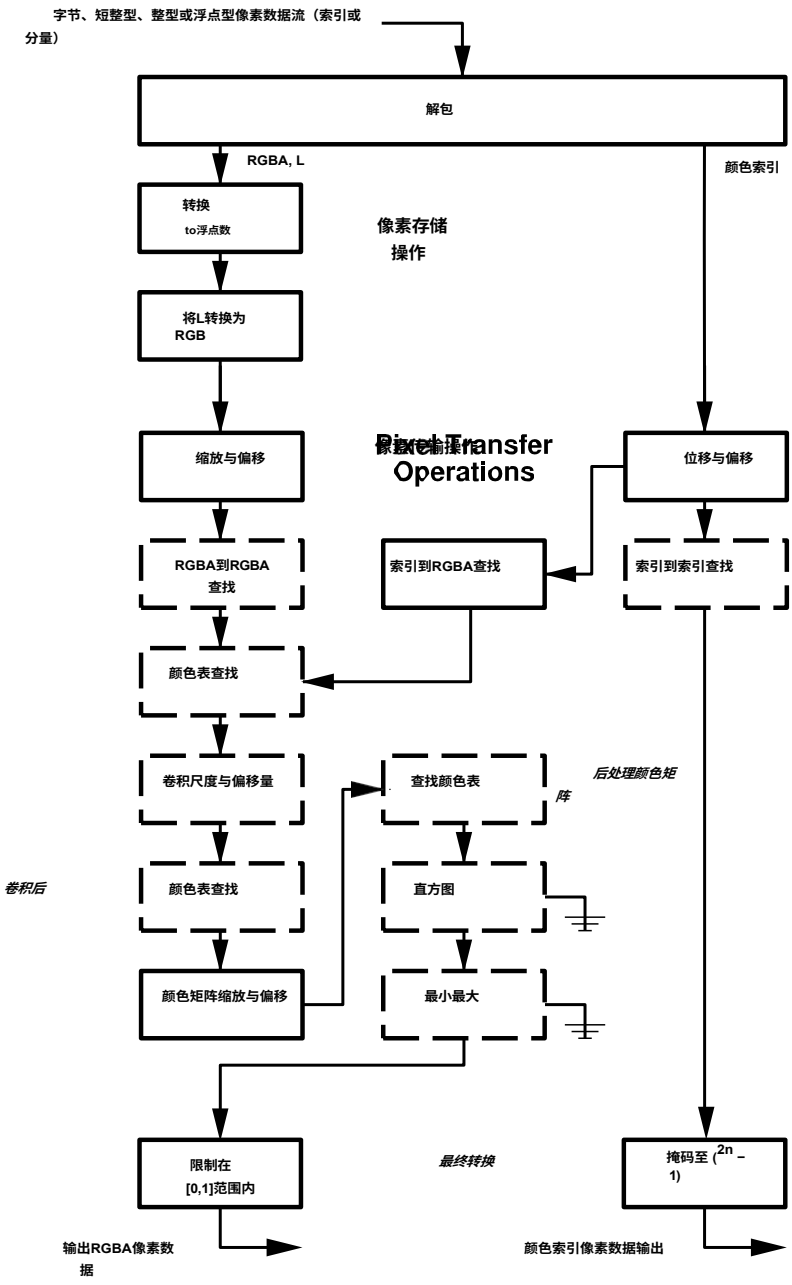


图3.7. **DrawPixels的操作**流程。若图形渲染器处于RGBA模式，则输出RGBA像素；否则输出颜色索引像素。虚线框内的操作可启用或禁用。RGBA与color index pixel paths, shown in深度与模板像素路径未显示。

*类型*参数 标记名称	对应 GL 数据类型	特殊 解释
无符号字节 _	ubyte	无
位图	ubyte	是
字节	字节	否
无符号短整型 _	ushort	无
SHORT	短	无
无符号整型 _	uint	无
INT	int	无
浮点	浮点	无
无符号字节 3 3_2 _ _ _	ubyte	是
无符号字节 2 3_3 反转 _ _ _ _	ubyte	是
无符号短整型 5_6_5 _ _ _	ushort	是
无符号短整型 5_6_5 反转 _ _ _ _	ushort	是
无符号短整型 4_4_4_4 _ _ _ _	ushort	是
无符号短整型 4_4_4_4 反转 _ _ _ _	ushort	是
无符号短整型 5_5_5_1 _ _ _ _	ushort	是
无符号短整型 1_5_5_5 反转 _ _ _ _	ushort	是
无符号整型 8_8_8_8 _ _ _ _	uint	是
无符号整型 8_8_8_8 反向 _ _ _ _	uint	是
无符号整型 10_10_10_2 _ _ _	uint	是
无符号整数 2_10_10_10 反相 _ _ _	uint	是

表 3.5： **中DrawPixels和ReadPixels的***类型*参数值及其对应的GL数据类型。GL数据类型的定义请参见表2.2。特殊解释详见第3.6.4节末尾。

绘制*数据*是一个指向待绘制数据的指针。这些数据采用七种GL数据类型之一表示，具体类型由*type参数*指定。 二十*种类型*标记值与其所指示的GL数据类型的对应关系见表3.5。若GL处于彩色索引模式，且*格式*值既非COLOR INDEX、STENCIL INDEX也非DEPTH COMPONENT，则会触发INVALID OPERATION错误。

*若*type为BITMAP而*format*非COLOR INDEX或STENCIL INDEX，则触发INVALID ENUM错误。*格式*与类型值组合的额外限制将在下文讨论。 _ _ _

格式名称	元素含义与顺序	目标缓冲区
色索引 _	颜色索引	颜色
模板索引 _	模板索引	模板
深度组件 _	深度	深度
红色	R	红色
绿色	G	颜色
蓝色	B	颜色
ALPHA	A	颜色
RGB	红、绿、蓝	颜色
RGBA	红、绿、蓝、透明	颜色
BGR	B, G, R	颜色
BGRA	B、G、R、A	色彩
亮度	Luminance	色度
亮度阿尔法 _	亮度，A	颜色

表 3.6： 的DrawPixels和ReadPixels格式。第二列给出了对元素组的描述及其数量和顺序。除非指定为索引，否则格式返回组件。

解包

数据从主机内存中读取时，以带符号或无符号字节序列（GL数据类型byte和ubyte）、带符号或无符号短整数（GL数据类型short和ushort）、带符号或无符号整数（GL数据类型int和uint）或浮点数（GL数据类型float）的形式呈现。 这些元素根据*格式*被分组为单值、双值、三值或四值集合，从而构成数据组。表3.6总结了从内存获取的数据组格式，同时标明了哪些格式生成索引值、哪些生成组件值。

默认情况下，每种GL数据类型的值均按客户端GL绑定语言的规范进行解释。 但若启用UNPACK SWAP BYTES指令，则值的解释需按表3.7修改位序。修改后的位序仅在GL数据类型ubyte为8位时定义，且对于每个具体GL数据类型，仅当该类型以8、16或32位表示时才生效。

内存中的组被视为排列在一个矩形中。该矩形由一系列*行*组成，其中第一行第一组的首个元素由传递**给DrawPixels的**指针所指向。若UNPACK ROW LENGTH的值非正，则每行包含的组数*为width*；

_ _

元素大小	默认位序	修改位序
8 位	[7..0]	[7..0]
16位	[15..0]	[7..0][15..8]
32 位	[31..0]	[7..0][15..8][23..16][31..24]

表 3.7：启用 UNPACK SWAP BYTES 时元素的位序修改。这些重新排序仅在 GL 数据类型 ubyte 为 8 位时定义，且仅适用于 8、16 或 32 位的 GL 数据类型。位 0 为最低有效位。

否则组数为UNPACK ROW LENGTH。若p表示第一行首个元素在内存中的位置，则第N行首个元素由

$$p + Nk \tag{3.12}$$

其中 N 是行号（从零开始计数），k 定义为

$$k = \begin{cases} nl & s \geq a, \\ a/s\,\lceil nl/a \rceil & s < a \end{cases} \tag{3.13}$$

其中 n 为组内元素数量，l 为行内组数，a 为 UNPACK ALIGNMENT 的值，s 为元素大小（单位为 GL ubyte）。若单个元素的位数并非 GL ubyte 位数的 1、2、4 或 8 倍，则对于所有 a 的值，$k = nl$。

存在一种机制，用于从更大的包含矩形中选择一组子矩形。该机制依赖于三个整数参数：UNPACK ROW LENGTH（解包行长度）、UNPACK SKIP ROWS（解包跳过行数）和UNPACK SKIP PIXELS（解包跳过像素数）。从内存中获取第一组数据后，传递**给DrawPixels函数**的指针实际向前移动了(UNPACK_SKIP PIXELS)n+(UNPACK SKIP ROWS)k个元素。随后从内存中连续元素处获取width组数据（不移动指针），之后指针向前*移动*k个元素。通过这种方式共获取height组width个值的数据。 参见图3.8。

调用 **DrawPixels** 时 一个 *类型* 类型 UNSIGNED BYTE_3 3 2, UNSIGNED BYTE 2 3 3 REV, UNSIGNED SHORT 5 6 5, UNSIGNED SHORT 5 6 5 REV, 无符号短整型 4 4 4 4, 无符号短整型 4 4 4 4 反转, 无符号短整型 5 5 5 1, 无符号短整型 1 5 5 5 反转, 无符号整型 8 8 8 8, 无符号整型 8 8 8 8 反转, 无符号整型 10 10 10 2, 或 无符号整型 2 10 10 10 反序 是一种特殊情况，其中每组的所有组件都被打包到单个无符号字节、无符号短整型或

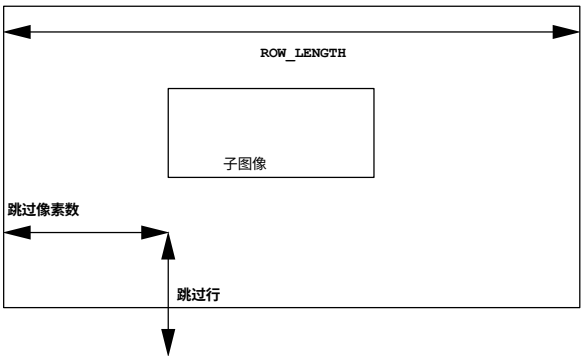


图3.8 从图像中选取子图像。所示参数名称**在DrawPixels中**以UNPACK为前缀，在**ReadPixels**中以PACK为前缀。

无符号整型，具体取决于类型。每组像素的组件数量由类型固定，且必须与*格式*参数所指示的每组组件数量匹配，具体参见表3.8。若发生不匹配，则生成错误INVALID OPERATION。此约束同样适用于所有其他接受或返回像素数据的函数——这些*函数*通过类型和*格式*参数来定义数据的类型与格式。

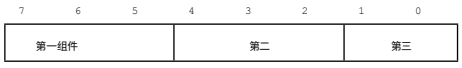
各打包像素类型的第一、第二、第三和第四分量在位域中的位置如表3.9、3.10和3.11所示。每个位域均被解释为无符号整数值。若基础GL类型支持高于最小精度的存储（例如9位字节），则打包分量在像素中右对齐。

组件通常按从最高有效位到最低有效位的顺序打包，后续组件依次占据较低有效位位置。标记名称以REV结尾的类型将组件打包顺序逆转为从最低有效位到最高有效位。所有情况下，每个组件的最高有效位均打包在其位域位置的最高有效位。

类型 参数 标记名称	GL数据 类型	数量 组件	匹配 像素格式
无符号字节 3 3_2 _ _ _	ubyte	3	RGB
无符号字节 2 3_3 反转 _ _ _ _	ubyte	3	RGB
无符号短整型 5_6_5 _ _ _	ushort	3	RGB
无符号短整型 5_6_5 反转 _ _ _ _	ushort	3	RGB
无符号短整型 4_4_4_4 _ _ _ _	ushort	4	RGBA,BGRA
无符号短整型 4_4_4_4 反转 _ _ _ _	ushort	4	RGBA,BGRA
无符号短整型 5_5_5_1 _ _ _ _	ushort	4	RGBA,BGRA
无符号短整型 1_5_5_5 反转 _ _ _ _	ushort	4	RGBA,BGRA
无符号整型 8_8_8_8 _ _ _ _	uint	4	RGBA、BGRA
无符号整型 8_8_8_8 反转 _ _ _ _	uint	4	RGBA,BGRA
无符号整型 10_10_10_2 _ _ _	uint	4	RGBA、BGRA
无符号整数 2_10_10_10 反转 _ _ _	uint	4	RGBA、BGRA

表 3.8： 压缩像素格式。

无符号字节 3 3_2: _ _ _



无符号字节 2 3_3 反转: _ _ _ _

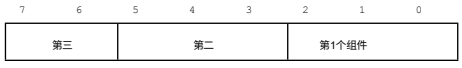
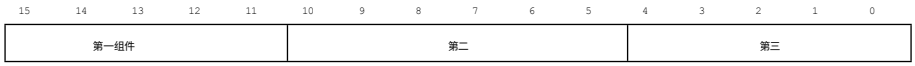
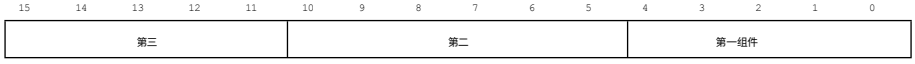


表 3.9： 无符号字节格式。各组件的位号均有标注。

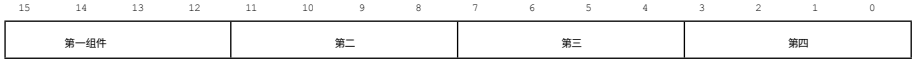
无符号短整型 5_6_5: _ _ _



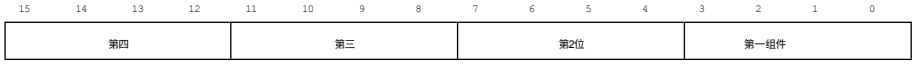
无符号短整型 5_6_5 反转: _ _ _ _



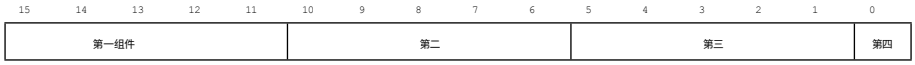
无符号短整型 4_4_4_4: _ _ _ _



无符号短整型 4_4_4_4 反转: _ _ _ _



无符号短整型 5_5_5_1: _ _ _ _



无符号短整型 1_5_5_5 反转: _ _ _ _

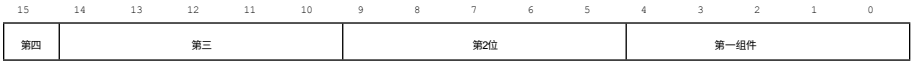


表 3.10: 无符号短整型格式 _

无符号整型 8 8_8 8: _ _ _ _

无符号整型 8 8_8 8 反转: _ _ _

无符号整数 10 10 10 2: _ _ _

无符号整数 2 10 10 10 反向: _ _ _

表 3.11: UNSIGNED INT 格式 _

格式	第一 组件	第二 组件	第三 组件	第四 组件
RGB	红色	绿色	蓝色	
RGBA	红色	绿色	蓝色	alpha
BGRA	蓝色	绿色	红色	alpha

表 3.12： 打包像素字段分配。

压缩像素中各组件与字段的映射关系如表3.12所示

若启用字节交换功能，将在从每个像素中提取组件前执行该操作。上述关于行长度和图像提取的讨论同样适用于紧凑像素——此时需将"组件"替换为"组"，且每组组件数量默认为1。

以BITMAP类型调用DrawPixels属于特殊情况，此时数据为一系列GL ubyte值。每个ubyte值通过其8个最低有效位指定8个1位元素。 若UNPACK LSB FIRST的值为FALSE，则8个单比特元素按从最高位到最低位顺序排列；否则按从最低位到最高位顺序排列。每个ubyte中除最低8位外的其他位值均不重要。

第一行的第一个元素是ubyte（如上定义）的第一个位，该ubyte由传递给DrawPixels的指针所指向。第二行的第一个元素是位于p + k位置的ubyte的第一个位（同样按上述定义），其中k由传递给DrawPixels的指针计算得出。

。第二行的第一个元素是位于位置 $p + k$ 的 ubyte 的第一个位（同样按上述定义），*其中* k 计算为

$$k = a \frac{l}{8a} \tag{3.14}$$

同样存在一种机制，可从位图图像中选取子矩形区域的元素。 在从内存获取首个元素前，传递**给DrawPixels的**指针会先向前移动UNPACK SKIP ROWS *个*k字节。随后忽略UNPACK SKIP PIXELS个1位元素，获取后续*宽度*为1位元素时不移动字节指针，之后指针再向前移动k字节。通过此方式可获取*高度为*高度元素*的*组。_ *

浮点数转换

此步骤仅适用于组件组，不作用于索引。组内每个元素均按ap-

采用表2.9（第2.14节）中的相应公式。对于打包像素类型，通过*计算c / (2^N 1)*对组内每个元素进行转换，*其中*c是包含该元素的位域的无符号整数值，N是位域的位数。

转换为RGB

此步骤仅在*格式*为亮度或亮度Alpha时生效。若*格式*为亮度，则将每个单元素组转换为R、G、B（三）元素组，具体操作是将原始单元素复制到三个新元素中。 若格式为LUMINANCE ALPHA，则将每组两个元素转换为包含R、G、B和A（四个）元素的组：将原始组的首个元素复制到新组的前三个元素中，并将原始组的第二个元素复制到A（第四个）新元素中。

最终扩展为RGBA格式

此步骤仅针对非深度组件组执行。每个组按以下方式转换为包含4个元素的组：若组中不包含A元素，则添加A元素并将其值设为1.0。若组中缺少R、G或B中的任意元素，则添加每个缺失元素并赋值为0.0。

像素传输操作

此步骤实为一系列操作。由于像素传输操作在绘制、复制、读取像素以及指定纹理图像（无论是从内存还是帧缓冲区）时均等效执行，故在第3.6.5节中单独描述。完成该节所述处理后，将按后续章节所述方式处理组。

最终转换

对于颜色索引，最终转换包括将索引中二进制点左侧的位按 2^n 进行掩码处理，其中 n 是索引缓冲区中的位数。对于 RGBA 组件，每个元素都被限制在 [0, 1] 范围内。最终值根据第 2.14.9 节（最终颜色处理）中的规则转换为定点数。

对于深度分量，元素首先被限制在[0, 1]范围内，然后转换为定点数，如同窗口z值一般（参见第2.11.1节《控制视口》）。

模板索引通过 $2^n - 1$ 进行掩码处理，其中 n 为模板缓冲区位数。

转换为片段

组件向片元的转换由以下函数控制：

```
void PixelZoom( float zx , float zy );
```

设 (x_{rp}, y_{rp}) 为当前光栅坐标（参见第2.13节）。若当前光栅坐标无效，则忽略DrawPixels操作；像素传输操作不会更新直方图或最小最大值表，且不生成任何片段。但即使后续像素所有权测试（第4.1.1节）或剪裁测试（第4.1.2节）拒绝了对应片段，直方图与最小最大值表仍会更新。）若某组（索引或组件）在某行中为第n个且属于第m行，则在窗口坐标系中视为由以下矩形边界限定的区域：

$(x_{rp} + z_x n, y_{rp} + z_y m)$ 以及 $(x_{rp} + z_x(n+1), y_{rp} + z_y(m+1))$

（z_x或 z_y 可能为负值）。对于矩形内部、底部或左边界上的每个帧缓冲区像素，都会生成一个表示组 (n, m) 的片段。

源自颜色数据组的片段会采用该组的颜色索引或颜色分量，以及当前光栅位置关联的深度值；而源自深度分量的片段则采用该分量的深度值，以及当前光栅位置关联的颜色索引或颜色分量。两种情况下，雾坐标均取自当前光栅位置关联的光栅距离，纹理坐标则取自当前光栅位置关联的纹理坐标。格式为STENCIL INDEX的**DrawPixels**生成的组将特殊处理，详见第4.3.1节。

3.6.5 像素传输操作

GL定义了四种像素组：

1. *RGBA分量：*每个组包含四个颜色分量：红、绿、蓝和透明度。
2. *深度组件：*每组包含单个深度组件。

3. *颜色索引：* 每个组包含单一颜色索引。

4. *模板索引：* 每个组包含一个模板索引。

本节所述的每项操作都会依次应用于图像中的每个像素组。许多操作仅适用于特定类型的像素组；若某项操作不适用于给定组，则跳过该操作。

分量运算

此步骤仅适用于RGBA分量组和深度分量组。 每个分量均乘以相应的有符号缩放因子：R分量乘以RED SCALE，G分量乘以GREEN SCALE，B分量乘以BLUE SCALE，A分量乘以ALPHA SCALE，深度分量乘以DEPTH SCALE。随后将结果与对应的有符号偏移量相加：RED BIAS、GREEN BIAS、BLUE BIAS、ALPHA BIAS或DEPTH BIAS。

索引运算

此步骤仅适用于颜色索引和模板索引组。若索引为浮点数，则将其转换为定点数，二进制小数点右侧位数未指定，二进制小数点左侧至少保留$\log_2 \times$ (MAX PIXEL MAP TABLE)位。原为整数的索引保持不变；转换后定点数中的小数位均设为零。

随后将固定点索引按索引移位位数进行移位：若INDEX SHIFT > 0则左移，否则右移。无论哪种情况移位后均补零。最后将带符号整数偏移量INDEX OFFSET加至索引值。

RGBA到RGBA查找

此步骤仅适用于RGBA组件组，且当MAP COLOR为FALSE时跳过。首先，每个组件值会被限制在[0, 1]范围内。每个R、G、B、A组件元素均关联一张映射表：R通道对应PIXEL MAP R TO R，G通道对应PIXEL MAP G TO G，B通道对应PIXEL MAP B TO B，A通道对应PIXEL MAP A TO A。

对于 A. 每个元素都乘以一个比对应表大小少 1 的整数，然后通过将该值四舍五入到最接近的整数来为每个元素找到一个地址。对于每个元素，用对应表中该地址处的值替换该元素。

颜色索引查找

此步骤仅适用于颜色索引组。若调用像素传输操作的GL命令要求生成RGBA分量像素组，则在此步骤执行转换。当满足以下条件时需生成RGBA分量像素组：

1. 当组将被光栅化且GL处于RGBA模式时，或
2. 这些组将作为图像加载到纹理内存中，或
3. 这些组将以非 COLOR INDEX 格式。

若需使用RGBA分量组，则索引的整数部分用于引用4张颜色分量表：PIXEL MAP I TO R、PIXEL MAP I TO G、PIXEL MAP I TO B和PIXEL MAP I TO A。每张表必须包含2^n条条目，其中n为某个整数值（n可能因不同情况而异）。每个表）。对于每个表，索引值首先被四舍五入至最接近的整数；该结果与 $2^n - 1$ 进行按位与运算，所得值作为表内地址使用。索引值将转换为相应的 R、G、B 或 A 值。由此获得的四个元素组将替换索引值，并将该组类型转换为 RGBA 颜色分量。

若无需RGBA分量组且启用了MAP COLOR功能，则在PIXEL MAP I TO I表中查找索引（否则不进行查找）。 同样，该表必须包含2^n条目（*n为整数*）。索引值先四舍五入至最接近的整数，与2^n 1进行按位与运算，所得值作为表内地址。表中值替换原索引值。浮点表值先转换为精度未指定的定点值，该组类型保持为颜色索引。

模板索引查找

此步骤仅适用于模板索引组。若启用MAP STENCIL功能，则索引将在PIXEL MAP S TO S表中进行查找（否则不进行查找）。该表必须包含2^n条目，其中*n为*整数。整数索引与2^n 1进行按位与运算，所得值作为表内地址使用。表中对应的整数值将替换原始索引。

颜色表查找

此步骤仅适用于RGBA组件组。仅当启用COLOR TABLE时才执行颜色表查找。若启用零宽度表则不进行查找

基准内部格式	R	G	B	A
ALPHA				A_t
LUMINANCE	L_t	L_t	L_t	
LUMINANCE_ALPHA	L_t	L_t	L_t	A_t
强度	I_t	I_t	I_t	I_t
RGB	R_t	G_t	B_t	
RGBA	R_t	G_t	B_t	A_t

表 3.13： 颜色表查找。R_t、G_t、B_t、A_t、L_t和I_t是根据表格格式分配给像素分量 R、G、B 和 A 的颜色表值。当没有分配时，分量值在查找后保持不变。

执行。

表的内部格式决定了组中哪些组件将被替换（参见表3.13）。待替换的组件通过以下方式转换为索引：先限制在[0, 1]范围内，再乘以小于表宽度的整数，最后舍入为最接近的整数。组件将被索引对应的表项替换。

所需状态为一个位，用于指示颜色表查找功能的启用或禁用状态。初始状态下查找功能处于禁用状态。

卷积

此步骤仅适用于RGBA组件组。 若启用CONVOLUTION 1D，则一维卷积滤波器仅应用于传递给**TexImage1D**、**TexSubImage1D**、Copy-TexImage1D和**CopyTexSubImage1D**的一维纹理图像。 如果启用了 CONVOLUTION 2D，则二维卷积滤波器仅应用于传递给 **DrawPixels**、**CopyPixels**、**ReadPixels**、**TexImage2D**、**TexSubImage2D**、**CopyTexImage2D**、**CopyTexSubImage2D** 和 **CopyTexSubImage3D** 的二维图像。 如果启用了 SEPARABLE 2D 且禁用了 CONVOLUTION 2D，则对这些图像应用可分离的二维卷积滤波器。

卷积运算是源图像像素与卷积滤波器像素的乘积之和。源图像像素始终包含四个分量：红、绿、蓝和透明度，在下述方程中分别用 R_s、G_s、B_s 和 $A_{(s)}$ 表示。 滤波器像素可采用五种存储格式之一，包含1、2、3或4个分量。这些分量在下式中分别表示为R_f、G_f、B_f、A_f、L_f和I_f。卷积运算的结果为四元组R,G,B,A。

基础过滤器格式	R	G	B	A
ALPHA	R_s	G_s	B_s	$A_s * A_f$
LUMINANCE	$R_s * L_f$	$G_s * L_f$	$B_s * L_f$	A_s
亮度阿尔法	$R_s * L_f$	$G_s * L_f$	$B_s * L_f$	$A_s * A_f$
光强	$R_s * I_f$	$G_s * I_f$	$B_s * I_f$	$A_s * I_f$
RGB	$R_s * R_f$	$G_s * G_f$	$B_s * B_f$	A_s
RGBA	$R_s * R_f$	$G_s * G_f$	$B_s * B_f$	$A_s * A_f$

表 3.14：根据滤镜图像格式计算滤波后的颜色分量。$C * F$ 表示图像分量 C 与滤镜 F 的卷积。

根据滤波器的内部格式，每个源图像像素的单独颜色分量要么与一个滤波器分量进行卷积，要么保持不变。相关规则定义于表 3.14。

三种卷积滤波器的卷积运算定义各不相同。*变量W_f和H_f*表示卷积滤波器的尺寸，*变量W_s和H_s*表示源像素图像的尺寸。

卷积方程定义如下，其中C表示滤波结果，C_f表示一维或二维卷积滤波器，$C_{行}$和$C_{列}$表示构成二维不可分离滤波器的两个一维滤波器。滤波输出图像C(r)取决于*原始图像颜色C(s)*以及下文所述的卷积边界模式。
可分离滤波器。C' 取决于源图像颜色 C_s 及下文所述的卷积边界模式。滤波输出图像 C_r 则同时依赖于上述所有参数。
变量并在每种边界模式下分别描述。像素索引命名法详见第3.6.3节中的**卷积滤波器规范**子章节。

一维滤波器：

$$C[i'] = \sum_{n=0}^{W_f-1} C'_s[i' + n] * C_f[n]$$

二维滤波器：

$$C[i', j'] = \sum_{n=0}^{W_f-1} \sum_{m=0}^{H_f-1} {}_sC'[i' + n, j' + m] * C_f[n, m]$$

二维可分离滤波器：

$$C[i', j'] = \sum_{n=0}^{W_f-1} \sum_{m=0}^{H_f-1} {}_s C'[i' + n, j' + m] * C_{行}[n] * C_{列}[m]$$

若一维滤波器的 W_f 为零，则 $C[i]$ 始终设为零。同理，若二维滤波器的 W_f 或 H_f 为零，则 $C[i, j]$ 始终设为零。

特定卷积滤波器的卷积边界模式通过调用

```
void ConvolutionParameterif( enum target, enum pname, T param );
```

其中 *target* 是滤波器的名称，*pname* 是卷积边界模式，*param* 参数取值为 REDUCE、CONSTANT BORDER 或 REPLICATE BORDER。

边界模式 REDUCE

采用边界模式REDUCE进行卷积时，源图像的宽度和高度将分别缩减为 $W_f - 1$ 和 $H_f - 1$。若缩减后导致结果图像的宽度和/或高度为零或负值，则输出直接为空值，且不触发错误。卷积后图像的坐标采用边界模式REDUCE的卷积在宽度方向上为零到 $W_s - W_f$，在高度方向上为零到 $H_s - H_f$。当规格可能导致误差时导致错误时，实际测试的是由此产生的尺寸而非源图像尺寸。（具体示例为 **TexImage1D** 和 TexImage2D，它们规定了图像尺寸约束。即使以空像素指针调用 **TexImage1D** 或 TexImage2D，生成的纹理图像尺寸仍遵循指定图像卷积计算的结果）。

当边界模式为REDUCE时，C 等于源图像颜色 $C_{(s)}$，且 C_r 等于滤波结果 C。

对于其余边界模式，定义 $C_w = W_f/2$ 且 $C_h = H_f/2$。坐标 (C_w, C_h) 定义卷积滤波器的中心位置。

边界模式 常数边界

若卷积边界模式为恒定边界，则输出图像的尺寸与源图像相同。卷积结果等同于将源图像四周填充为当前卷积边界颜色像素的效果。 当卷积滤波器超出源图像边缘时，将使用恒定边界像素作为滤波器输入。当前卷积边界颜色通过调用ConvolutionParameterfv或**ConvolutionParameteriv**设置，*其中*pname设为CONVOLUTION BORDER COLOR，*参数数组* params包含四个值：

作为图像边界的RGBA颜色值。整数颜色分量按线性方式解释：最大正整数映射为1.0，最大负整数映射为-1.0。浮点颜色分量在指定时不进行截断。

对于一维滤波器，结果颜色由以下公式定义：

$$C_r[i] = C[i - C_w]$$

其中$C[i]$通过以下方程计算得出：

$$C_s[i] = \begin{cases} C_s[i'], & 0 \le i' < W_s \\ C_c, & \text{否则} \end{cases}$$

且 C_c 为卷积边界颜色。

对于二维或可分离的二维滤波器，结果颜色由以下公式定义：

$$C_r[i, j] = C[i - C_w, j - C_h]$$

其中$C[i', j']$通过以下方程计算得出：

$$C_s[i, j] = \begin{cases} C_s[i', j'], & 0 \le i' < W_s, 0 \le j' < H_s \\ C_c, & \text{否则} \end{cases}$$

边界模式 `REPLICATE BORDER`

卷积边界模式`REPLICATE BORDER`同样生成与源图像尺寸相同的输出图像。该模式的行为与`CONSTANT BORDER`模式完全一致，仅在卷积滤波器超出源图像边缘的像素位置处理方式上有所不同。对于这些位置，其处理效果如同复制了源图像最外侧的单像素边界。 概念上，源图像最左侧单像素列的每个像素会被*复制*C_w次以在左边缘提供额外图像数据；最右侧单像素列的每个像素同样*复制*C_w次以在右边缘提供数据；顶部与底部单像素行的每个像素值均被复制，从而在顶部和*底部边缘生成*C_h行图像数据。每个角点的像素值同样被复制，以提供源图像每个角点卷积运算所需的数据。

对于一维滤波器，结果颜色由以下公式定义：

$$C_r[i] = C[i - C_w]$$

其中 $C[i']$ 通过以下 $C_s'[i']$ 的计算公式求得：

$$C_s'[i'] = C_s[\text{clamp}(i', W_s)]$$

且钳位函数 clamp(*val, max*) 定义为

$$\text{clamp}(val, max) = \begin{cases} 0, & val < 0 \\ val, & 0 \le val < max \\ max - 1, & val \ge max \end{cases}$$

对于二维或可分离的二维滤波器，结果颜色由以下公式定义：

$$C_r[i, j] = C[i - C_w, j - C_h]$$

其中$C[i', j']$通过*以下*$C_s'[i', j']$方程计算：

$$C_s'[i', j'] = C_s[\text{clamp}(i', W_s), \text{clamp}(j', H_s)]$$

若执行卷积运算（ ）与卷积后红尺度（ convolution operation ），则对卷积后红尺度（ performed, ）进行处理。每个红（ ）分量（ ）的卷积后红尺度（）在卷积后图像（ resulting image ）中，通过对应的卷积后红尺度参数（ scaled by the **corresponding** PixelTrans-fer parameters）进行缩放：卷积后红尺度（POST CONVOLUTION RED SCALE）用于红（ ）分量（ R com-ponent, POST CONVOLUTION GREEN SCALE for a G compo-nent, POST CONVOLUTION BLUE SCALE for a B component, ）卷积后蓝尺度（POST CONVOLUTION BLUE SCALE）用于蓝（a B component, ）卷积后阿尔法尺度（POST CONVOLUTION ALPHA SCALE）用于阿尔法（ an A component. ） 结果将添加至对应的偏置值： （后卷积红偏置）、POST CONVOLUTION GREEN BIAS（后卷积绿偏置）、 （后卷积蓝偏置）、（后卷积红偏置）或POST CONVOLUTION ALPHA BIAS（后卷积Alpha偏置）。

所需状态包含三个位，分别指示一维卷积、二维卷积或可分离二维卷积的启用/禁用状态；一个描述当前卷积边界模式的整数；以及四个指定卷积边界颜色的浮点数值。初始状态下所有卷积操作均禁用，边界模式为REDUCE，边界颜色为(0, 0, 0, 0)。

卷积后颜色表查找

此步骤仅适用于RGBA组件组。通过调用**Enable**或**Disable**并传入符号常量POST CONVOLUTION COLOR TABLE来启用或禁用卷积后颜色表查找。卷积后颜色表通过调用**ColorTable**并传入*目标*参数定义：

卷积后颜色表。除该特性外，其操作与第3.6.5节中定义的颜色表查找完全相同。

所需状态为一位，用于指示后卷积表查找功能的启用或禁用状态。初始状态下该功能处于禁用状态。

色彩矩阵转换

此步骤仅适用于RGBA组件组。各组件通过颜色矩阵进行变换，每个变换后的组件再乘以相应的有符号缩放因子：红色组件对应POST COLOR MATRIX RED SCALE，绿色组件对应POST COLOR MATRIX GREEN SCALE，蓝色组件对应，B组件对应，A组件对应POST COLOR MATRIX ALPHA SCALE。结果为：添加至作为有符号偏移量：POST COLOR MATRIX RED BIAS, POST COLOR MATRIX GREEN BIAS, POST COLOR MATRIX BLUE BIAS, 或 后处理颜色矩阵阿尔法偏移量。所得的各分量将替换原始组的每个分量。

即若 M_c 为色彩矩阵，s 下标表示组分的缩放项，b 下标表示偏移项，则组分 R G B A 被转换为

$$\begin{bmatrix} R' \\ G' \\ B' \\ A' \end{bmatrix} = \begin{bmatrix} R_s & 0 & 0 & 0 \\ 0 & G_s & 0 & 0 \\ 0 & 0 & B_s & 0 \\ 0 & 0 & 0 & A_s \end{bmatrix} M_c \begin{bmatrix} R \\ G \\ B \\ A \end{bmatrix} + \begin{bmatrix} R_b \\ G_b \\ B_b \\ A_b \end{bmatrix}.$$

邮政颜色矩阵颜色表查找

此步骤仅适用于RGBA组件组。后置色彩矩阵颜色表查找通过调用**Enable**或**Disable**并传入符号常量POST COLOR MATRIX COLOR TABLE来启用或禁用。后置色彩矩阵表通过**调用ColorTable**并以POST COLOR MATRIX COLOR TABLE作为*目标*参数来定义。除上述差异外，其操作机制与第3.6.5节定义的颜色表查找完全一致。

所需状态为单比特值，用于指示后处理色彩矩阵查找功能的启用或禁用状态。初始状态下查找功能处于禁用状态。

直方图

此步骤仅适用于RGBA组件组。通过调用**Enable**或**Disable**函数并传入符号常量HISTOGRAM来启用或禁用直方图操作。

若表格宽度不为零，则索引 R_i、G_i、B_i和 A_i 被移除。
通过将每个像素组的红、绿、蓝和透明度分量（不修改这些分量）限制在[0,1]范围内，乘以小于直方图表宽度的数值，并舍入到最接近的整数，从而获得这些分量。若直方图表格式包含红色或亮度分量，则直方图*条目*R_i的红色或亮度分量加一。 若直方图表格式包含绿色通道，则*直方图条目*G_i的绿色分量加一。直方图*条目*B_i*和*A_i的蓝色与透明度分量亦按相同方式递增。若直方图条目分量递增至超出其最大值，该值将变为未定义状态；此情况不视为错误。

若**直方图***接收器*参数为FALSE，则直方图操作对正在处理的像素组流不产生影响。否则，所有RGBA像素组将在直方图操作完成后立即被丢弃。由于直方图操作优先于最小最大操作，因此不会执行最小最大操作。不会生成像素片段，不会修改纹理内存内容，也不会返回像素值。但无论是否丢弃像素组，纹理对象状态都会被修改。

最小最大值

此步骤仅适用于RGBA分量组。通过调用**Enable**或**Disable**并传入符号常量MINMAX来启用或禁用最小最大值操作。

若最小值表的格式包含红色或亮度值，则红色分量值仅当其小于该分量时，才会替换最小值表元素中的红色或亮度值。 同样地，若格式包含红色或亮度通道，且组内红色分量大于最大值元素中的红色或亮度值，则红色组分量将替换红色或亮度最大值分量。若表格格式包含绿色通道，当绿色组分量小于绿色最小值或大于绿色最大值时，将分别有条件地替换绿色最小值和/或最大值。 若表格格式包含蓝色和/或透明度，则蓝色与透明度组分将按类似方式进行测试与替换。最小值与最大值组分的内部类型为浮点数，其可表示范围至少等同于用于表示颜色的浮点数（参见2.1.1节）。对于超出可表示范围的组分值，未定义任何语义处理机制。

超出可表示范围的组分量值的处理语义。

如果 **Minmax** *接收器*参数为 FALSE，则 minmax 操作对正在处理的像素组流不产生任何影响。否则，所有 RGBA 像素组将在 minmax 操作完成后立即被丢弃。不会生成像素片段，不会修改纹理内存内容，也不会返回像素值。然而，无论像素组是否被丢弃，纹理对象状态都会被修改。

3.6.6 像素矩形多采样光栅化

如果启用了MULTISAMPLE，且SAMPLE BUFFERS的值为1，则像素矩形将通过以下算法进行光栅化。 设 (X_{rp}, Y_{rp}) 为当前光栅化位置。（若当前位置无效，则忽略 **DrawPixels** 操作。）若某组像素（索引或分量）位于第 n 列且属于第 m 行，则考虑窗口坐标系中由以下矩形边界限定的区域：

$$(X_{rp} + Z_x * n,\ Y_{rp} + Z_y * m)$$

和

$$(X_{rp} + Z_x \text{ 乘以 } (n + 1),\ Y_{rp} + Z_y \text{ 乘以 } (m + 1))$$

其中 Z_x和 Z_y 是由像素缩放因子 **PixelZoom** 指定的像素缩放因子，各自可为正值或负值。对于每个帧缓冲区像素，若其包含一个或多个采样点位于该矩形内部、底部边界或左侧边界上，则生成代表组 (n, m) 的片元。 每个生成的片段从组和当前光栅化位置获取关联数据，其方式符合第3.6.4节"**转换为片段**"小节的讨论。所有深度和颜色采样值均被赋予相同数值，该数值取自其所属组（针对深度组和颜色组）或当前光栅化位置（若不属于组）。 所有采样值均采用相同的雾坐标及纹理坐标集，该坐标集取自当前光栅化位置。

根据像素缩放因子，单个像素矩形可能为同一帧缓冲区像素生成多个（甚至大量）片段。

3.7 位图

位图是由0和1组成的矩形，用于指定要生成的特定片段模式。每个片段都关联着相同的数据，这些数据与*当前光栅位置*相关联。

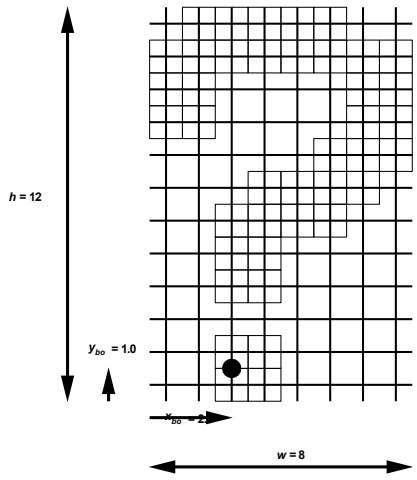


图3.9. 一幅位图及其关联参数。x_{bi}和 y_{bi} 未显示。

位图通过以下方式发送：

```
void Bitmap( sizei w, sizei h, float xbo , float ybo , float xbi , float ybi ,
    ubyte *data );
```

w 和 *h* 分别表示矩形位图的整数宽度和高度。 (x_{bo}, y_{bo}) 表示位图原点的浮点坐标值。(x_{bi}, y_{bi}) 表示位图光栅化后添加至光栅化位置的浮点坐标增量。*data* 是指向位图的指针。

如同多边形图案，位图会根据第3.6.4节中**为DrawPixels命令**指定的流程从内存中展开；这相当于向该命令传递的宽度和*高度*分别等于w和*h*，*类型*为BITMAP，*格式*为COLOR INDEX。展开后的原始值（在执行任何转换或运算之前）构成由0和1组成的点阵图案。 参见图3.9。

通过**Bitmap**发送的位图按以下方式进行光栅化：首先，若当前光栅位置无效（有效位被重置），则忽略该位图。否则，构建一个矩形片段数组，其左下角位于

$$(x_{ll}, y_{ll}) \rightleftharpoons (\lfloor x_{rp} - x_{bo} \rfloor, \lfloor y_{rp} - y_{bo} \rfloor)$$

右上角位于($x_{ll}+w$, $y_{ll}+h$)，其中w和h分别是位图的宽度和高度。数组中的片段在位图对应位为1时生成，否则不生成。每个片段关联的数据是当前光栅位置对应的数据。生成片段后，更新当前光栅位置：

$$(x_{rp}, y_{rp}) \leftarrow (x_{rp}+x_{bi}, y_{rp}+y_{bi})。$$

当前光栅位置的 z 和 w 值保持不变。

位图多采样光栅化

如果启用了MULTISAMPLE功能，且SAMPLE BUFFERS的值为1，则位图将通过以下算法进行光栅化。若当前光栅化位置无效，则忽略该位图。 否则，将构建一个屏幕对齐的像素尺寸矩形数组，其左下角位于(X_{rp}, Y_{rp})，右上角位于($X_{rp}+w$, $Y_{rp}+h$)，其中w和h分别是位图的宽度和高度。 若位图中对应位为0，则数组内矩形被剔除；否则保留。位图光栅化过程为每个帧缓冲区像素生成片段，当该像素的采样点位于保留矩形内部或其底部/左侧边界时即产生片段。

覆盖位对应于保留矩形内部、底部或左侧边缘的采样点时为1，其余覆盖位均为0。每个采样的关联数据均与当前光栅化位置相关联。生成片段后，当前光栅化位置的更新方式与单采样光栅化情况完全一致。

3.8 纹理映射

纹理映射将一个或多个指定图像的局部区域映射到启用纹理功能的每个基元上。该映射通过使用图像在片段(s, t, r, q)坐标位置处的颜色来修改片段的主RGBA颜色实现。纹理映射不影响次要颜色。实现必须支持同时使用至少两张图像进行纹理映射。

该片段携带多组纹理坐标（s, t, r, q），用于索引独立图像以生成颜色值，这些颜色值共同用于修改片段的RGBA颜色。纹理映射仅适用于RGBA模式；在颜色索引模式下的使用行为未定义。以下子章节（直至第3.8.8节）规定了单纹理的GL操作规范，第3.8.15节则详细说明了多纹理单元间的交互机制。

本规范提供两种方式指定基元纹理映射的具体实现：第一种称为固定功能模式（本节描述），第二种称为片段着色器（第3.11节描述）。纹理映射图像的指定方式及应用于基元时的过滤机制为两种方法所共用，本节将对此进行讨论。 本节同时阐述了通过固定功能机制确定RGBA值的生成方式。若片段着色器处于活动状态，则RGBA值的确定方法由应用程序提供的片段着色器决定，具体规范详见《OpenGL着色语言规范》。

当没有片段着色器处于活动状态时，纹理映射使用的坐标为(*s/q, t/q, r/q*)，该坐标由原始纹理坐标(*s, t, r, q*)推导而来。*若*q纹理坐标小于或等于零，则纹理映射使用的坐标未定义。 当片段着色器处于活动状态时，(*s, t, r, q*)坐标可供片段着色器使用。片段着色器中用于纹理映射的坐标由OpenGL着色语言规范定义。

3.8.1 纹理图像规范

命令

```
void TexImage3D( enum target, int level, int internalformat, sizei width, sizei height,
    sizei depth, int border, enum format, enum type, void *data );
```

用于指定三维纹理图像。 *目标*必须是TEXTURE 3D，或在第3.8.11节所述的特殊情况下为PROXY TEXTURE 3D。*格式*、类型*和数据*需与Draw-Pixels的对应参数匹配（参见第3.6.4节）； 它们分别指定图像数据的格式、数据类型以及指向主机内存中图像数据的指针。*格式* STENCIL INDEX 不被允许。

内存中的组被视为按相邻矩形序列排列。每个矩形均为二维图像，其尺寸与组织结构由TexImage3D的宽度和*高度*参数指定。UNPACK ROW LENGTH与UNPACK ALIGNMENT的值控制着这些图像中行与行之间的间距，其作用方式与DrawPixels相同。 若整型参数UNPACK IMAGE HEIGHT的值非正数，则每个二维图像的行数*为height*；否则行数即为UNPACK IMAGE HEIGHT。每个二维图像包含整数行数，并与相邻图像完全相邻。

选择三维图像子卷的机制依赖于整数参数UNPACK SKIP IMAGES。 若UNPACK SKIP IMAGES为正值，则在从内存获取首个组前，指针将按UNPACK SKIP IMAGES乘以单个二维图像元素数量的步长递增。随后处理*深度*二维图像，每个图像均以与**DrawPixels**相同的方式提取子图像。

选定组的处理流程完全**遵循DrawPixels规范**，仅在最终转换前停止。生成的每个R、G、B、A或深度值均被限制在[0, 1]区间内。

随后从生成的R、G、B、A或深度值中选取组件，以获得具有*由internalformat*指定（或由此派生）的*基础内部格式的*纹理。表3.15总结了R、G、B、A和深度值映射至纹理组件的关系，该关系取决于纹理图像的基础内部格式。*internalformat* 可指定为：- 表 3.15 所列七种内部格式符号常量之一- 表 3.16 所列带*尺寸内部*格式符号常量之一- 表 3.17 所列特定压缩内部格式符号常量之一- 表 3.18 所列六种通用压缩内部格式符号常量之一 为兼容GL 1.0版本，*internalformat*亦可取整数值1、2、3、4，分别等效于符号常量LUMINANCE、LUMINANCE ALPHA、RGB和RGBA。若*指定的internalformat*值不在上述范围，将引发INVALID VALUE错误。

仅当*目标*为TEXTURE 1D、TEXTURE 2D、PROXY TEXTURE 1D或PROXY TEXTURE 2D时，纹理图像规范命令才支持基础内部格式为DEPTH COMPONENT的纹理。若将此格式与其他*目标*结合使用，将导致INVALID OPERATION错误。

基础内部格式为深度组件的纹理需要深度组件数据；其他基础内部格式的纹理需要RGBA组件数据。若基础内部格式为深度组件而格式非深度组件，或基础内部格式非深度组件而格式为深度组件，则会触发无效操作错误。

GL未提供特定的压缩内部格式，但提供了获取扩展功能所支持的此类格式标记值的机制。可通过查询NUM COMPRESSED TEXTURE FORMATS的值获取渲染器支持的特定压缩内部格式数量。通过查询COMPRESSED TEXTURE FORMATS的值可获取渲染器支持的特定压缩内部格式集合。 此查询仅返回适用于通用场景的格式值。渲染器不会枚举那些需在使用前特别理解其限制条件的格式。

通用压缩内部格式绝不会直接用作纹理图像的内部格式。若 *internalformat* 为六种通用压缩内部格式之一，其值将被替换为 GL 选择的、具有相同基础内部格式的特定压缩内部格式符号常量。若无可用特定压缩格式，则 *internalformat* 将被替换为对应的基础内部格式。 若 *internalformat* 被指定或映射为特定压缩内部格式，但因故无法支持该格式（例如压缩格式不支持3D纹理或边界），则 *internalformat* 将被替换为对应的基础内部格式，纹理图像将不会被GL压缩。

*内部组件分辨率*是指纹理图像中分配给每个值的位数。若将*internalformat*指定为基础内部格式，则GL会以自行选择的内部组件分辨率存储最终纹理。 若指定了固定尺寸的内部格式，则R、G、B、A及深度值与纹理组件的映射关系等同于表3.15中对应基础内部格式的组件映射，且GL会尽可能将每个纹理组件的内存分配值调整至与表3.16所列分配值最接近。 ("尽可能接近"的定义由实现决定。但对于表3.16中分配值非零的组件，必须分配非零位数；其余组件则必须分配零位数。实现必须为每个基础内部格式支持至少一种内部组件分辨率分配方案。 若指定压缩内部格式，则R、G、B、A及深度值与纹理组件的映射关系等同于表3.15中对应基础内部格式的组件映射。指定图像将采用由实现方选择的（可能有损的）压缩算法进行压缩。GL。

GL实现**可根据TexImage3D、TexImage2D**（见下文）**或TexImage1D**（见下文）的任意参数（*目标除外*）调整内部组件分辨率或压缩内部格式的分配，但该分配及选定的压缩图像格式不得受其他状态影响，且一旦确定便不可更改。 此外，压缩图像格式的选择不得受*数据*参数影响。分配必须保持不变：每次使用相同参数值指定纹理图像时，必须选择相同的分配方案和压缩图像格式。这些分配规则同样适用于第3.8.11节所述的代理纹理。

图像本身（由*数据*指向）是一系列值组。第一组是纹理图像左下后角的位置。后续各组依次填充宽度*为width*的行，从左至右排列；*高度为height*的行则从底部开始堆叠。

基础内部格式	RGBA与深度值	内部组件
ALPHA	A	*A*
深度分量	深度	*D*
亮度	R	*L*
亮度阿尔法	R,A	*L,A*
光强	R	*I*
RGB	R,G,B	*R,G,B*
RGBA	R,G,B,A	*R,G,B,A*

表 3.15： 将 RGBA 和深度像素分量转换为内部纹理、表或滤波器分量。有关纹理分量 *R*、*G*、*B*、*A*、*L*、*I* 和 *D* 的描述，请参见第 3.8.13 节。

至顶部形成单个二维图像切片；*深度*切片则从后向前堆叠。当某组的最终R、G、B和A分量计算完成后，它们将按表3.15所述分配至*纹素*分量。从零开始计数，每个生成的第N个纹素被分配内部整数坐标(*i, j, k*)，其中

$$i = (N \bmod \textit{宽度}) - b_s$$

$$j = \left(\left\lfloor \frac{N}{\textit{宽度}} \right\rfloor \bmod \textit{高度}\right) - b_s$$

$$k = \left(\left\lfloor \frac{N}{\textit{宽度} \times \textit{高度}} \right\rfloor \bmod \textit{深度}\right) - b_s$$

其中 b_s 是指定的*边框*宽度。因此，三维图像的最后一个二维图像切片被赋予最高的 *k* 值。

每个颜色分量通过四舍五入转换为n位固定小数值，其中n是图像数组中该分量分配的存储位数。 我们假设所用的定点表示法将每个值*表示为*k/(2^n 1)，*其中*k 0, 1, ..., 2^n 1，即*k的*二进制表示为全1字符串（例如1.0在二进制中表示为全1字符串）。

TexImage3D 的 *level* 参数是一个整数*级别的细节*数值。细节级别将在下文的 **Mipmapping** 部分进行讨论。主纹理图像的细节级别数值为 0。若指定的细节级别小于零，则会生成 INVALID VALUE 错误。

TexImage3D 的*边界*参数表示边界宽度。边界的重要性将在下文阐述。边界宽度影响纹理图像的尺寸：设

Sized 内部格式	基础 内部格式	*R* 位	*G* 位	*B* 位	*A* 位	*L* 位	*I* 位	*D* 位
ALPHA4	ALPHA				4			
ALPHA8	ALPHA				8			
ALPHA12	ALPHA				12			
ALPHA16	ALPHA				16			
深度分量16	深度组件							16
深度分量24	深度组件							24
深度组件32	深度组件							32
亮度4	亮度					4		
亮度8	LUMINANCE					8		
LUMINANCE12	LUMINANCE					12		
LUMINANCE16	LUMINANCE					16		
亮度4Alpha4	亮度阿尔法				4	4		
亮度6阿尔法2	LUMINANCE ALPHA				2	6		
亮度8阿尔法8	LUMINANCE ALPHA				8	8		
亮度12阿尔法4	LUMINANCE ALPHA				4	12		
亮度12阿尔法12	LUMINANCE ALPHA				12	12		
亮度16阿尔法16	LUMINANCE ALPHA				16	16		
强度4	强度						4	
强度8	强度						8	
强度12	强度						12	
强度16	强度						16	
R3 G3 B2	RGB	3	3	2				
RGB4	RGB	4	4	4				
RGB5	RGB	5	5	5				
RGB8	RGB	8	8	8				
RGB10	RGB	10	10	10				
RGB12	RGB	12	12	12				
RGB16	RGB	16	16	16				
RGBA2	RGBA	2	2	2	2			
RGBA4	RGBA	4	4	4	4			
RGB5 A1	RGBA	5	5	5	1			
RGBA8	RGBA	8	8	8	8			
RGB10 A2	RGBA	10	10	10	2			
RGBA12	RGBA	12	12	12	12			
RGBA16	RGBA	16	16	16	16			

表 3.16： 不同尺寸的内部格式与基本内部格式的对应关系，以及
每种尺寸内部格式的*期望*组件分辨率。

压缩内部格式	基础内部格式
(无)	

表 3.17： 特定压缩内部格式。OpenGL 1.3 未定义任何格式

1.3未定义任何压缩格式，但GL扩展中定义了若干特定压缩类型。

通用压缩内部格式	基础内部格式
压缩Alpha	ALPHA
压缩亮度	亮度
压缩亮度Alpha	亮度阿尔法
压缩强度	强度
压缩RGB	RGB
压缩RGBA	RGBA

表 3.18： 通用压缩内部格式。

$$w_s = w_t + 2b_s \tag{3.15}$$

$$h_s = h_t + 2b_s \tag{3.16}$$

$$d_s = d_t + 2b_s \tag{3.17}$$

其中 w_s、 h_s和 d_s 分别表示指定图像的*宽度*、*高度和深度*，而 w_t、 h_t和 d_t 表示边界内部纹理图像的尺寸。若 w_t、 h_t或 d_t小于零，则生成错误 INVALID VALUE。

宽度、高度或深度为零的图像表示空纹理。若在纹理参数TEXTURE BASE LEVEL（参见第3.8.4节）指定的细节级别中使用空纹理，则效果等同于禁用纹理映射。

目前，边界*宽度*b_s的最大值为1。*若*b_s小于零，或大于 b_t，则会生成错误INVALID VALUE。

三维纹理图像的最大允许宽度、高度或深度，取决于细节级别和内部格式相关的函数。对于图像数组，其值必须至少为 $2^{k-lod} + 2b_t$。细节级别为0*至*k（*其中*k为MAX 3D TEXTURE SIZE的2的对数），lod是图像数组的细节级别，b_t是最大边界宽度。对于任何细节级别大于*k*的图像数组，该值可能为零。若指定图像过大以致无法存储，则生成错误

INVALID VALUE 错误，表示指定图像过大，在任何条件下均无法存储。

同样地，对于0至*k*级别的图像数组，一维或二维纹理图像的最大允许宽度，以及二维纹理图像的最大允许高度，必须至少满足$2^{k-lod} + 2b_t$的条件，其中k是MAX TEXTURE SIZE的以2为底的对数。 立方贴图纹理的最大允许宽度和高度必须相同，且对于级别0至*k*的图像数组，其值必须至少满足$2^{k-lod} + 2b_{(t)}$的条件，其中k为MAX CUBE MAP TEXTURE SIZE的2进制对数。

实现可能仅允许创建级别为0的图像数组，前提是该单一图像数组能够被支持。关于创建级别为1或更高的图像数组的额外限制，将在第3.8.10节中详细说明。

命令

```
void TexImage2D( enum target, int level,
    int internalformat, sizei width, sizei height, int border, enum format,
    enum type, void *data );
```

用于到指定二维纹理图像。*目标*必须是 TEXTURE 2D（用于二维纹理），或 TEXTURE CUBE MAP POSITIVE X、TEXTURE CUBE MAP NEGATIVE X、TEXTURE CUBE MAP POSITIVE Y、TEXTURE CUBE MAP NEGATIVE Y、TEXTURE CUBE MAP POSITIVE Z、或 TEXTURE CUBE MAP NEGATIVE Z（用于三维纹理）之一。立方贴图纹理。此外，*目标*参数在特殊情况下（详见第3.8.11节）可为PROXY TEXTURE 2D（表示二维代理纹理）或PROXY TEXTURE CUBE MAP（表示立方贴图代理纹理）。其余**参数与TexImage3D的**对应参数一致。

在解码纹理图像时，**TexImage2D**等效于调用**TexImage3D**并传入对应参数及*深度*值1，但存在以下差异：

- 图像*深度*始终为1，与*边界*参数值无关。图像将执行卷积操作（可能改变其*宽度*
-
- 忽略UNPACK_SKIP IMAGES设置。

二维纹理由单张二维纹理图像构成。立方体贴图纹理则由六张二维纹理图像组成。这六个立方体贴图目标共同构成单个立方体贴图纹理，但每个目标分别对应立方体贴图的独立面。上述列出的TEXTURE CUBE MAP *目标会更新其对应的立方体贴图面二维纹理图像。请注意，这六个立方体贴图

在指定、更新或查询立方贴图六个二维图像之一时，会使用诸如TEXTURE CUBE MAP POSITIVE X之类的二维图像标记；但当启用立方贴图纹理化或绑定至立方贴图纹理对象时（即访问整个立方贴图而非特定二维图像），则需指定TEXTURE CUBE MAP目标。

当TexImage2D的*target***参数为六种立方体贴图二维图像目标之一时，若***width***与***height***参数不等，将触发**INVALID VALUE错误。

最后，命令

```
void TexImage1D( enum target, int level,
    int 内部格式, sizei 宽度, int 边框, enum 格式, enum 类型, void
    *数据 );
```

用于指定一维纹理图像。*目标*必须为TEXTURE 1D，或在第3.8.11节所述特殊情况下为PROXY TEXTURE 1D。

为解码纹理图像之目的，TexImage1D等效于**调用TexImage2D**并传入对应参数及高度1，但需注意：

- 图像高度始终为1，与*边界*值无关。
- 仅当启用 CONVOLUTION 1D 时，才会对图像执行卷积操作（可能改变其*宽度*）。

图像指针指向的图像被解码并复制到图形处理器的内部存储器中。 此复制操作将解码后的图像置于最大允许宽度 b_t的边框内，无论是否指定了边框（参见图 3.10）[1]。若未指定边框或指定的边框宽度小于最大允许值，图像仍按最大可能宽度的边框进行存储。 任何超出边界（包括已指定边界本身）的区域将被赋予未指定值。二维纹理仅在其左、右、上、下四端具有边界，而一维纹理仅在其左右两端具有边界。

我们将把（可能经过边界增强的）解码图像称为*纹理数组*。三维纹理数组具有宽度、高度和深度 w_s、 h_s和 d_s，分别由方程 3.15、3.16 和 3.17 定义。 二维纹理数组的*深度*$d_s = 1$，*高度*h_s*与宽度*w_s如上所述；一维纹理数组的*深度* $d_s = 1$，*高度*$h_s = 1$，*宽度*w_s如上所述。

[1] 图3.10需展示三维纹理图像。

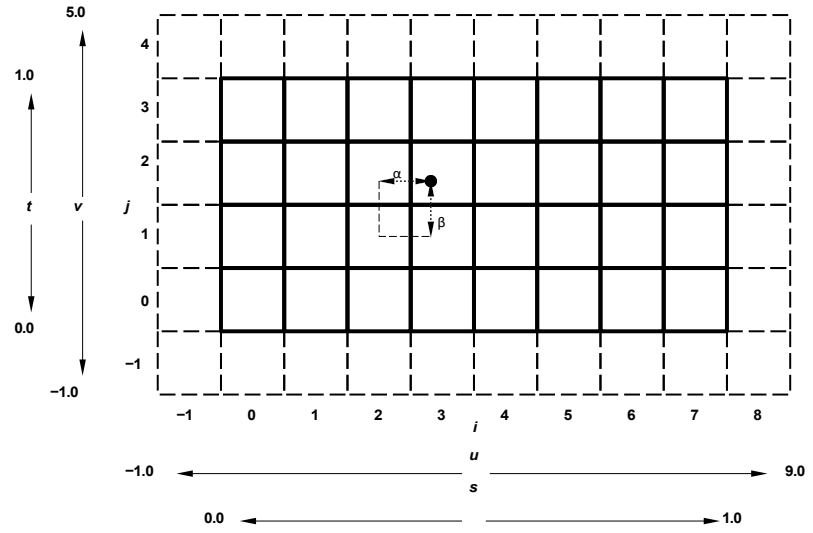


图3.10. 纹理图像及其访问坐标。此为二维纹理，尺寸为n=3×m=2。一维纹理仅由单条水平纹理带构成。图中同时标注了α和β值——用于混合相邻纹素以获取纹理值的参数。

纹理数组中的元素 (*i, j, k*) *被称为纹素*（对于二维纹理，*k* 无关紧要；对于一维纹理，*j* 和 *k* 都无关紧要）。用于对片元进行纹理映射的*纹理*值由该片元关联的 (*s, t, r*) 坐标决定，但可能与任何实际纹素都不对应。参见图 3.10 。

若 **TexImage1D**、**TexImage2D** 或 **TexImage3D** 的 *data* 参数为空指针（C 实现中为零值指针），则会创建一个一维、二维或三维纹理数组，*其目标*、*层级*、*内部格式*、*宽度*、*高度和深度*均按指定创建，但图像内容未指定。此时客户端内存中不会访问任何像素值，也不会执行像素处理。 然而，系统仍会生成错误，其表现完全如同*数据*指针有效时的情形。

3.8.2 替代纹理图像指定命令

二维和一维纹理图像也可通过直接取用帧缓冲区中的图像数据来指定，同时可重新指定现有纹理图像的矩形子区域。

命令

```
void CopyTexImage2D( enum target, int level, enum internalformat, int x,
    int y, sizei width, sizei height, int border );
```

定义了一个二维纹理数组，其方式与**TexImage2D**完全相同，区别在于图像数据取自帧缓冲区而非客户端内存。目前，*目标*必须为以下类型之一：TEXTURE_2D、TEXTURE_CUBE_MAP_POSITIVE_X、TEXTURE_CUBE_MAP_NEGATIVE_X、TEXTURE_CUBE_MAP_POSITIVE_Y、TEXTURE_CUBE_MAP_NEGATIVE_Y、TEXTURE_CUBE_MAP_POSITIVE_Z 或 TEXTURE_CUBE_MAP_NEGATIVE_Z 之一。*x*, *y*, *width* 和 *height* 完全对应于 **CopyPixels** 的相应参数（参见第 4.3.3 节）；它们指定图像的宽度和*高度*，以及待复制帧缓冲区区域的左下角 (*x*, *y*) 坐标。图像从帧缓冲区提取的过程，完全等同于将这些参数传递给**CopyPixels函数**时——根据*内部格式*将参数*类型*设为COLOR或DEPTH，并在像素传输处理完成后停止。 RGBA数据取自当前颜色缓冲区，深度分量数据则取自深度缓冲区。若需深度分量数据而未存在深度缓冲区，则触发INVALID_OPERATION错误。后续处理流程**与TexImage2D**完全一致，首先对生成的像素组进行R、G、B、A或深度值的裁剪。 *参数level*、*internalformat*和*border*采用**与TexImage2D**对应参数相同的取值范围及含义，但*internalformat*不可设为1、2、3或4。*若internalformat*指定无效值，则触发INVALID_ENUM错误。*width*、*height*和*border*的约束条件完全**遵循TexImage2D**对应参数的规范。

当CopyTexImage2D的目标参数为六种立方体贴图二维图像目标之一时，若宽度和*高度*参数不等，则会生成INVALID_VALUE错误。

命令

```
void CopyTexImage1D( enum target, int level, enum internalformat, int x,
    int y, sizei width, int border );
```

定义了一维纹理数组，其方式与 **TexImage1D** 完全相同，区别在于图像数据取自帧缓冲区而非客户端内存。目前，*目标*必须为 `TEXTURE 1D`。为解码纹理图像，**CopyTexImage1D** 等同于调用 **CopyTexImage2D** 并传入对应参数及*高度*为 1 的值，但无论*边界参数*取值如何，图像*高度*始终为 1。 *level*、*internalformat* 和 *border* 的取值及含义与 **TexImage1D** 的对应参数完全一致，但 *internalformat* 不可设为 `1`、`2`、`3` 或 `4`。*width* 和 *border* 的约束条件完全遵循 **TexImage1D** 对应参数的规范。

六个新增命令，

```
void TexSubImage3D( enum target, int level, int xoffset, int yoffset, int zoffset,
    sizei width, sizei height, sizei depth, enum format, enum type, void *data );
void TexSubImage2D( enum target, int level, int xoffset,
    int yoffset, sizei width, sizei height, enum format, enum type, void *data );
void TexSubImage1D( enum target, int level, int xoffset, sizei width, enum format,
    enum type, void *data );
void CopyTexSubImage3D( enum 目标, int 层级, int x偏移, int y偏移, int z
    偏移, int x坐标, int y坐标, sizei 宽度, sizei 高度);
void CopyTexSubImage2D( enum 目标, int 层级, int x偏移, int y偏移, int x
    坐标, int y坐标, sizei 宽度, sizei 高度);
void CopyTexSubImage1D( enum 目标, int 层级, int x偏移量, int x坐标,
    int y坐标, sizei 宽度 );
```

仅重新指定现有纹理数组中的矩形子区域。指定纹理数组的*内部格式*、*宽度*、*高度*、*深度*或*边界*参数均保持不变，且指定子区域外的纹素值亦不受影响。 目前**TexSubImage1D**和**CopyTexSubImage1D**的目标参数必须为`TEXTURE 1D`，**TexSubImage2D** 和 **CopyTexSubImage2D** *的目标*参数必须为 `TEXTURE 2D`、`TEXTURE CUBE MAP POSITIVE X`、`TEXTURE CUBE MAP NEGATIVE X`、`TEXTURE CUBE MAP POSITIVE Y`、`TEXTURE CUBE MAP NEGATIVE Y`、`TEXTURE CUBE MAP POSITIVE Z` 或 `TEXTURE CUBE MAP NEGATIVE Z` 之一，且 **TexSubImage3D** 和 **CopyTexSubImage3D** 的*目标*参数必须为 `TEXTURE 3D`。每个命令的*level*参数指定修改的纹理数组的层级。

。若*层级*小于零或大于纹理最大宽度、高度或深度的2进制对数，则会触发INVALID VALUE错误。

TexSubImage3D 的参数 *width*、*height*、*depth*、*format*、*type* 和 *data* 与 **TexImage3D** 的对应参数一致，即它们使用相同的数值进行指定，且具有相同含义。 同样地，**TexSubImage2D** 的参数 *width*、*height*、*format*、*type* 和 *data* 与 **TexImage2D** 的对应参数一致；**TexSubImage1D** 的参数 *width*、*format*、*type* 和 *data* 则与 **TexImage1D** 的对应参数一致。

CopyTexSubImage3D 和 **CopyTexSubImage2D** 的参数 *x*、*y*、*width* 和 *height* 与 **CopyTexImage2D**[2]的对应参数一致。**CopyTexSubImage1D** 的参数 *x*、*y* 和 *width* 与 **CopyTexImage1D** 的对应参数一致。每条 **TexSubImage** 命令对像素组的解释和处理方式完全与对应的 **TexImage** 命令相同，但 R、 G、B、A 和深度像素组值的赋值操作由纹理数组的*内部格式*控制，而非命令参数。**TexSubImage** 命令的*参数*格式与被重指定纹理数组的*内部格式*所适用的约束和错误，与对应 **TexImage** 命令的*格式*和*内部*格式参数完全一致。

TexSubImage3D 和 **CopyTexSubImage3D** 的参数 *xoffset*、*yoffset* 和 *zoffset* 指定纹理数组中一个矩形子区域的左下纹素坐标，该子区域的宽度为 width，高度为 height，深度为 depth。 **CopyTexSubImage3D** 关联的*深度*参数始终为 1，因为帧缓冲区内存是二维的——**CopyTexSubImage3D** 仅替换三维纹理单个 *s,t* 切片的一部分。

xoffset、*yoffset* 和 *zoffset* 的负值对应边界纹理单元的坐标，其寻址方式如图 3.10 所示。 设 w_s 、h_s 、d_s 和 b_s 分别为纹理数组的指定宽度、高度、深度及边界宽度，*x*、*y*、*z*、*w*、*h*、*d* 分别为 *xoffset*、*yoffset*、*zoffset*、*宽度*、高度*和*深度参数值，当满足以下任一关系时将触发 INVALID VALUE 错误：

$$x < -b_s$$
$$x + w > w_s - b_s \quad y < -b_s$$
$$y + h > h_s - b_s$$
$$z < -b_s$$

[2] 由于帧缓冲区本质上是二维的，因此不存在**CopyTexImage3D**命令。

$$z + d > d_s - b_s$$

从零开始计数，第n个像素组被分配给具有内部整数坐标[*i, j, k*]的纹理像素，其中

$$i = x + (n \bmod w)$$

$$j = y + (\lfloor \frac{n}{w} \rfloor \bmod h)$$

$$k = z + (\lfloor \frac{n}{宽度 \times 高度} \rfloor 模 d$$

TexSubImage2D 和 **CopyTexSubImage2D** 的参数 *xoffset* 和 *yoffset* 指定纹理数组中宽度为 width、高度为 height 的矩形子区域左下角纹素坐标。*xoffset* 和 *yoffset* 的负值对应边界纹素坐标，其寻址方式如图 3.10 所示。 设 w_s 、h_s 和 b_s 分别为纹理数组的指定宽度、高度及边界宽度，*x*、*y*、*w* 和 *h* 分别为 *xoffset*、*yoffset*、*width* 和 *height* 的参数值，当满足以下任一关系时将触发 `INVALID VALUE` 错误：

$$x < -b_s$$

$$x + w > w_s - b_s \quad y < -b_s$$

$$y + h > h_s - b_s$$

从零开始计数，第n个像素组被分配给内部整数坐标为[*i, j*]的纹理像素，其中

$$i = x + (n \bmod w)$$

$$j = y + (\lfloor \frac{n}{w} \rfloor \bmod h)$$

TexSubImage1D 和 **CopyTexSubImage1D** 的 *xoffset* 参数指定纹理数组中宽度方向子区域的左侧纹素坐标。*xoffset* 的负值对应边界纹素的坐标。设 w_s和 b_s 分别为纹理数组的指定宽度和边界宽度，*x* 和 *w* 分别为 *xoffset* 和 *width* 的参数值，则以下任一关系将引发 `INVALID VALUE` 错误：

$$x < -b_s$$

$$x + w > w_s - b_s$$

从零开始计数，第n个像素组被分配给内部整数坐标为[*i*]的纹理像素，其中

$$i = x + (n \bmod w)$$

采用压缩内部格式的纹理图像可能以特定方式存储，导致无法在不解压并重新压缩纹理图像的情况下，使用子图像命令修改图像。即使通过此方式修改图像，也可能无法保留被修改区域之外部分纹素的内容。为避免此类复杂情况，GL不支持对采用压缩内部格式的纹理图像进行任意修改。 若**调用TexSubImage3D**、**CopyTexSubImage3D**、**TexSubImage2D**、**CopyTexSubImage2D**、TexSubImage1D或**CopyTexSubImage1D**时，*x偏移量*、*y偏移量*或*z偏移量*不等于b_s（边界宽度），将导致INVALID OPERATION错误。 此外，此类调用修改区域外的纹素内容将未定义。针对特定场景可放宽这些限制。

压缩的内部格式，其图像易于修改。

3.8.3 压缩纹理图像

纹理图像也可通过已存储于已知压缩图像格式的图像数据进行指定或修改。当前GL未定义此类格式，但为支持此功能的GL扩展提供了机制。

命令

```
void CompressedTexImage1D( enum target, int level, enum internalformat, sizei
    width, int border, sizei imageSize, void *data );
void CompressedTexImage2D( enum target, int level, enum internalformat, sizei
    width, sizei height, int border, sizei imageSize, void *data );
void CompressedTexImage3D( enum 目标, int 级别, enum 内部格式, sizei 宽度,
    sizei 高度,
    sizei depth, int border, sizei imageSize, void *data );
```

分别定义一维、二维和三维纹理图像，其输入数据以特定压缩图像格式存储。*目标*、*级别*、*内部格式*、*宽度*、*高度*、*深度*和边界参数**与TexImage1D**、TexImage2D和**TexImage3D**中的含义相同。*data*指向*以internalformat*对应的压缩图像格式存储的压缩图像数据。由于

由于GL未提供特定图像格式，若*将internalformat*设为六种通用压缩格式中的任意一种，将导致INVALID ENUM错误。

对于所有其他压缩内部格式，压缩图像将根据定义*内部格式*标记的规范进行解码。压缩纹理图像被视为从地址*data*开始的*imageSize*个字节数组。 解码压缩纹理图像时，所有像素存储模式和像素传输模式均被忽略。若 *imageSize* 参数与压缩图像的格式、尺寸及内容不一致，将引发 INVALID VALUE 错误。若压缩图像未按定义的图像格式编码，调用结果将不可预测。

特定压缩内部格式可能对压缩图像规范调用或参数的使用施加格式专属限制。例如，压缩图像格式可能仅支持2D纹理，或禁止使用非零边界值。此类限制将在定义压缩内部格式的扩展规范中明确记载；违反限制将导致INVALID OPERATION错误。

特定压缩内部格式施加的任何限制均为不变的，这意味着如果图形渲染器接受并以压缩形式存储纹理图像，那么向**CompressedTexImage1D**、**CompressedTexImage2D**或**CompressedTexImage3D**提供相同图像时，只要满足以下限制条件，就不会引发INVALID OPERATION错误：

- *数据*指向由**GetCompressedTexImage**（第6.1.4节）返回的压缩纹理图像。
- *目标*、层级和*内部格式*与返回*数据*的**GetCompressed**TexImage调用中提供的*目标*、*层级*和*格式*参数完全匹配。
- *width*、*height*、*depth*、*border*、*internalformat*、和 *image-Size*与TEXTURE WIDTH、TEXTURE HEIGHT、TEXTURE DEPTH、TEXTURE BORDER、TEXTURE INTERNAL FORMAT、以及在调用**GetCompressedTexImage***返回数据*时生效的图像*级别*对应的纹理压缩图像尺寸。

此保证不仅适用于**GetCompressedTexImage**返回的图像，也适用于任何其他正确编码的、具有相同尺寸和格式的压缩纹理图像。

命令

```
void CompressedTexSubImage1D( enum target, int level, int xoffset, sizei width, enum
    format, sizei imageSize, void *data );
```

```
void CompressedTexSubImage2D( enum target, int level, int xoffset, int yoffset,
    sizei width, sizei height, enum format, sizei imageSize, void *data );
void 压缩纹理子图像三维( enum 目标, int 层级, int x偏移, int y偏移, int z偏移,
    sizei 宽度, sizei 高度, sizei 深度, enum 格式,
    sizei imageSize, void *data );
```

仅重新指定现有纹理数组中的矩形区域，传入数据存储于已知的压缩图像格式中。*目标*、*层级*、*x偏移*、*y偏移*、*z偏移*、*宽度*、*高度*和深度参数与**TexSubImage1D**、TexSubImage2D及**TexSubImage3D**中的含义相同。*data*指向*以format*对应的压缩图像格式存储的压缩图像数据。 由于核心GL未提供特定图像格式，若将*格式参数*设为这六种通用压缩内部格式中的任意一种，将导致INVALID ENUM错误。

由*data*参数和*imageSize*参数指向的图像，其解释方式等同于提供给**CompressedTexImage1D**、CompressedTexImage2D和**CompressedTexImage3D函数**的情况。这些命令不支持图像格式转换，因此若*格式*与被修改纹理图像的内部格式不匹配，则会导致INVALID OPERATION错误。 若 *image*Size 参数与压缩图像的格式、尺寸及内容不符（数据过少或过多），则会引发 INVALID VALUE 错误。与 **CompressedTexImage** 调用类似，压缩内部格式可能对压缩图像规范调用或参数的使用存在额外限制。此类限制将在定义压缩内部格式的规范中记录；违反这些限制将导致 INVALID OPERATION 错误。

特定压缩内部格式施加的任何限制均为不变的，这意味着若GL接受并以压缩形式存储纹理图像，在满足以下限制条件时，将相同图像提供给 **CompressedTexSubImage1D** 、 **CompressedTexSubImage2D** 、CompressedTexSubImage3D不会导致INVALID OPERATION错误：

- *数据*指向由**GetCompressedTexImage**（第6.1.4节）返回的压缩纹理图像。
- *目标*、*层级*和*格式*参数需与返回*data*的**GetCompressedTexImage**调用中提供的参数完全匹配。
- *宽度*、*高度*、*深度*、*格式*以及*图像尺寸*匹配该val-ues的纹理宽度、纹理高度、纹理深度、

纹理内部格式，以及纹理压缩图像尺寸

针对**GetCompressedTexImage**调用时生效的图像级别*level*

调用返回*数据*时生效的图像级别。

- *宽度*、*高度*、*深度*、格式分别对应纹理宽度、纹理高度、纹理深度及纹理内部格式的数值
  当前生效于图像层级 *level*。

- *xoffset*、 、*yoffset*、 和 *zoffset* 均为 b, 其中 *b* 为
  当前图像*层级*生效的纹理边框。

此保证不仅适用于**GetCompressedTexture函数**返回的图像，也适用于任何其他正确编码且尺寸相同的压缩纹理图像。

调用**CompressedTexSubImage3D**、Compressed**TexSubImage2D**或**CompressedTexSubImage1D**时，若*x偏移量*、*y偏移量*或*z偏移量*不等于 b_s（边界宽度），或*宽度*、高度和*深度*分别与TEXTURE WIDTH、TEXTURE HEIGHT或TEXTURE DEPTH的值不匹配，则将引发INVALID OPERATION错误。调用修改区域外的纹素内容定义为未定义。对于易于修改图像的特定压缩内部格式，这些限制可酌情放宽。

3.8.4 纹理参数

多种参数控制纹理数组在指定/修改时的处理方式及其应用于片段时的行为。每个参数均通过调用

```
void TexParameter if( enum 目标, enum 参数名, T 参数 );void TexParameter if v( enum 目标, enum 参数名,
                                                    { }
    T params );                                     { }
```

target 是目标对象，可以是 TEXTURE 1D、TEXTURE 2D、TEXTURE 3D 或 TEXTURE_CUBE MAP。*pname* 是表示待设置参数的符号常量；可能的常量及其对应参数汇总于表 3.19。 在命令的第一种形式中，*param* 是用于设置单值参数的数值；在命令的第二种形式中，*params* 是参数数组，其类型取决于所设置的参数。 若 TEXTURE BORDER COLOR 的值或 TEXTURE PRIORITY 的值被指定为整数，则应用表 2.9 中有符号整数的转换规则将其转换为浮点数，随后将每个值限制在 [0, 1] 范围内。

在第3.8节的剩余部分中，*用*lod_{min}、lod_{max}、$level_{base}$*和*$level_{(max)}$*表示* 的 值 ，其中 是纹理贴图

名称	类型	合法值
纹理卷绕模式 – –	整数	裁剪，裁剪到边缘，重复，裁剪到边界， 镜像重复 –
纹理包裹 T – –	整数	裁剪，裁剪至边缘，重复， – 边界裁剪，镜像重复 –
纹理包裹 R – –	整数	CLAMP, CLAMP TO EDGE, REPEAT, 紧贴边框，镜像重复 –
纹理最小滤波 – –	整数	最近邻， 线性， 最近邻MIP贴图最近邻，最近邻MIP贴图线性，线性MIP贴图最近邻，线性MIP贴图线性， – – – –
纹理最大过滤 – –	整数	最近邻， 线性
纹理边界颜色 – –	4 个浮点数	任意4个值在[0, 1]范围内
纹理优先级 –	浮点数	[0, 1] 范围内的任意值
纹理最小LOD – –	浮点数	任意值
纹理最大LOD – –	浮点	任意值
纹理基础级别 – –	整数	任意非负整数
纹理最高级别 – –	整数	任何非负整数
纹理LOD偏移量 – –	浮点数	任意值
深度纹理模式 –	枚举	亮度、强度、透明度
纹理比较模式 – –	枚举	无、将R与纹理比较 – – –
纹理比较函数 – –	枚举	LEQUAL, GEQUAL 小于、大于、等于、不等于、总是 、从不
生成MIP贴图 –	布尔	TRUE 或 FALSE

表 3.19： 纹理参数及其值。

主轴方向	目标	s_c	t_c	m_a
$+r_x$	纹理立方体贴图正X	$-r_z$	$-r_y$	r_x
$-r_x$	纹理立方体贴图负X轴	r_z	$-r_y$	r_x
$+r_y$	纹理立方体贴图正Y	r_x	r_z	r_y
$-r_y$	纹理立方贴图负Y轴	r_x	$-r_z$	r_y
$+r_z$	纹理立方贴图正Z轴	r_x	$-r_y$	r_z
$-r_z$	纹理立方贴图负Z轴	$-r_x$	$-r_y$	r_z

表 3.20：基于纹理坐标主轴方向选择的立方体贴图图像。

纹理最大LOD、_纹理基础级别和纹理最大级别分别 _ _ _
分别。

立方贴图纹理的参数适用于整个立方贴图；六个独立的二维纹理图像使用立方贴图本身的纹理参数。

若纹理参数 GENERATE MIPMAP 的值为 TRUE，则指定或更改纹理数组可能产生副作用，具体详见第 3.8.8 节中关于**自动生成 Mipmap** 的讨论。

3.8.5 深度分量纹理

深度纹理在纹理过滤和应用过程中可被视为亮度、强度或透明度纹理。深度纹理的初始状态默认按亮度纹理处理。

3.8.6 立方体贴图选择

当启用立方体贴图时，(*s t r*) 纹理坐标被视为一个方向向量$(r_x r_y r_z)$，该向量从立方体中心点发出（q坐标可忽略，因其仅改变向量尺度而不影响方向）。在纹理应用时，通过插值得到的片元方向向量将根据最大模数坐标方向（主轴方向）选择立方贴图某一面的二维图像。若两个或多个坐标模数相等，具体实现可定义规则以消除歧义。 该规则必须确定性且仅依赖于$(r_x \quad r_y \quad r_z)$。表3.20的目标列说明了主轴方向如何映射到特定立方体贴图目标的二维图像。

根据表3.20中规定的长轴方向确定的s_c、t_c和m_a，更新后的(s　　t）按下列公式计算：

$$s = \frac{1}{2}\frac{s_c}{|m_a|} + 1$$

$$t = \frac{1}{2}\frac{t_c}{|m_a|} + 1$$

这个新的 (s t) 用于根据第 3.8.7 节至第 3.8.9 节给出的规则，在确定的面二维纹理图像中查找纹理值。

3.8.7 纹理包裹模式

由TEXTURE WRAP S、TEXTURE WRAP T或TEXTURE WRAP R值定义的包裹模式分别影响s、t和r纹理坐标的解释。每种模式的效果如下所述。_

循环模式 REPEAT

包裹模式REPEAT忽略纹理坐标的整数部分，仅使用小数部分。（对于数值f，小数部分*定义为*f　f，*无论f的正负*；需注意　　函数会向　　方向截断。）

REPEAT 是所有纹理坐标的默认行为。

折返模式 CLAMP

包裹模式 CLAMP 将纹理坐标限制在 [0, 1] 范围内。

包裹模式 CLAMP TO EDGE　_　_

包裹模式 CLAMP TO EDGE 在所有米普级上对纹理坐标进行裁剪，确保纹理过滤器永远不会采样边界纹素。裁剪后返回的颜色仅取自纹理图像边缘的纹素。

纹理坐标被限制在[*min, max*]范围内。最小值定义为

$$min = \frac{1}{2N}$$

*其中*N为纹理图像在夹边方向上的维度（一维/二维/三维）。最大值定义为

$$max = 1 - min$$

确保纹理坐标映射范围[0, 1]始终保持对称约束。

包裹模式：边界裁剪

包裹模式 CLAMP TO BORDER 会在所有米波图上对纹理坐标进行裁剪，使得当对应纹理坐标远超出 [0, 1] 范围时，纹理过滤器始终采样边界纹素。裁剪后返回的颜色仅来源于纹理图像的边界纹素，若纹理图像无边界则取常量边界颜色。

纹理坐标被限制在[*min, max*]范围内。最小值定义为

$$min = \frac{-1}{2N}$$

其中 *N* 表示一维、二维或三维纹理图像在夹紧方向上的尺寸（不含边界）。最大值定义为

$$max = 1 - min$$

确保纹理坐标映射范围[0, 1]内始终保持对称裁剪。

镜像重复**包裹模式**

镜像重复包裹模式首先对纹理坐标进行镜像处理，其中镜像 *f*值的计算方式为

$$\text{镜像}() = \begin{cases} f - [f\,], & [f\,] \text{ 为偶数} \\ 1 - (f - [f\,]), & [f\,] \text{ 为奇数} \end{cases}$$

镜像坐标随后按上述包裹模式描述进行夹持 CLAMP TO_EDGE。

3.8.8 纹理缩放

将纹理应用于基元意味着从纹理图像空间到帧缓冲区图像空间的映射。通常，该映射涉及对采样纹理图像的重建，随后进行由

映射到帧缓冲区空间，接着进行过滤，最后对经过过滤、变形和重建的图像进行重采样，然后将其应用于片段。在GL中，这种映射通过两种简单的过滤方案之一进行近似。根据纹理空间到帧缓冲区空间的映射是否被认为会*放大*或*缩小*纹理图像，选择其中一种方案。

缩放因子与细节级别

该选择由缩放*因子*$\rho(x, y)$和*细节*级别参数共同决定。

$\lambda(x, y)$ 定义为

$$\lambda_{基数}(x, y) = \log_2[\rho(x, y)] \tag{3.18}$$

$$\lambda'(x, y) = \lambda_{基准}(x, y) + clamp(偏移_{纹理对象} + 偏移_{纹理单元} + 偏移_{着色器}) \tag{3.19}$$

$$\lambda = \begin{cases} lod_{max}, & \lambda' > lod_{max} \\ \lambda', & log_{min} \le \lambda' \le log_{max} \\ 未定义, & 最小对数密度 > 最大对数密度 \end{cases} \tag{3.20}$$

$bias_{texobj}$是绑定纹理对象的纹理LOD偏移值（详见第3.8.4节）。$bias_{texunit}$是当前纹理单元的纹理LOD偏移值（详见第3.8.13节）。$bias_{shader}$是片段着色器可用的纹理查找函数中可选偏移参数的值。若纹理访问在未提供偏移量的片段着色器中执行，或在片段着色器外部执行，则*偏移量*$_{着色器}$为零。这些值的总和将被限制在[*偏移量*$_{最大值}$, *偏移量*$_{最大值}$]范围内，其中*偏移量*$_{最大值}$是实现定义常量MAX TEXTURE LOD BIAS的值。

若 $\lambda(x, y)$ 小于或等于常数 c（详见下文第 3.8.9 节），则纹理被视为放大；若大于该常数，则纹理被视为缩小。初始值 lod_{min}和 lod_{max}的选择需确保永远不会限制 λ 的正常范围。可通过调用 **TexParameter[if]** 并分别将 *pname* 设为 TEXTURE MIN LOD 或 TEXTURE MAX LOD 来为特定纹理重新指定这些值。

设 $s(x, y)$ 为将原始图形内部的每个窗口坐标组 (x, y) 映射至 s 个纹理坐标的函数；$t(x, y)$ 和 $r(x, y)$ 类比定义。设 $u(x, y) = w_t \times s(x, y)$，$v(x, y) = h_t \times t(x, y)$，且 $w(x, y) = d_t \times r(x, y)$，其中 w_t、h_t和 d_t 分别由方程 3.15、3.16 和 3.17 定义。

其中 w_s、h_s 和 d_s 分别等于图像数组的宽度、高度和深度

其级别为$level_{base}$。对于一维纹理，定义$v(x, y) \quad 0$和$w(x, y) \quad 0$；对于二维纹理，定义$w(x, y) \quad 0$。对于多边形，在窗口坐标为(x, y)的片元处，ρ由以下公式给出：

$$\rho = \max\left(\sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial w}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 + \left(\frac{\partial w}{\partial y}\right)^2} \right) \tag{3.21}$$

其中 $\partial u/\partial x$ 表示 u 对窗口 x 的导数，其余导数同理。

对于一条线，公式为

$$\rho = \frac{\sqrt{\left(\frac{\partial u}{\partial x}\Delta x + \frac{\partial u}{\partial y}\Delta y\right)^2 + \left(\frac{\partial v}{\partial x}\Delta x + \frac{\partial v}{\partial y}\Delta y\right)^2 + \left(\frac{\partial w}{\partial x}\Delta x + \frac{\partial w}{\partial y}\Delta y\right)^2}}{l}, \tag{3.22}$$

其中 $\Delta x = \sqrt{x_2 - x_1}$ 且 $\Delta y = y_2 - y_1$ 其中 (x_1, y_1) 和 (x_2, y_2) 为线段窗口坐标的端点，且 $l = \sqrt{\Delta x^2 + \Delta y^2}$。对于点、像素矩形或位图时，$\rho \equiv 1$。

虽然普遍认为方程3.21和3.22在纹理映射时能获得最佳效果，但它们往往难以实现。因此，实际实现中可采用*函数*$f(x, y)$来近似*理想密度*ρ，该函数需满足以下条件：

1. $f(x, y)$ 在$|\partial u/\partial x|$、$|\partial u/\partial y|$、$|\partial v/\partial x|$、$|\partial v/\partial y|$、$|\partial w/\partial x|$、$|\partial w/\partial y|$
2. 令

$$m_u = \max\left(\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}\right)$$

$$m_v = \max\left(\frac{\partial v}{\partial x}, \frac{\partial v}{\partial y}\right)$$

$$m_w = \max\left(\frac{\partial w}{\partial x}, \frac{\partial w}{\partial y}\right).$$

则 $\max\{m_u, m_v, m_w\} \le f(x, y) \le m_u + m_v + m_w$。

当 λ 表示最小化时，`TEXTURE_MIN_FILTER` 的赋值

纹理最小过滤器为最近邻时，将获取纹理像素数组中*距离*(*s,t, r*)点最近（曼哈顿距离）的纹理像素。这意味着位置(*i, j, k*)处的纹理像素将成为纹理值，其中i由

$$i = \begin{cases} \lfloor u \rfloor, & s < 1 \\ w_t - 1, & s = 1 \end{cases} \tag{3.23}$$

（请注意，若纹理卷绕模式为重复，则 $0 \le s < 1$。）同样地，*j* 的计算方式为

$$j = \begin{cases} \lfloor v \rfloor, & t < 1 \\ h_t - 1, & t = 1 \end{cases} \tag{3.24}$$

，*k* 则为

$$k = \begin{cases} \lfloor w \rfloor, & r < 1 \\ d_t - 1, & r = 1 \end{cases} \tag{3.25}$$

对于一维纹理，*j* 和 *k* 无关紧要；位置 *i* 处的纹素即为纹理值。对于二维纹理，*k* 无关紧要；位置 (*i, j*) 处的纹素即为纹理值。

当纹理最小化过滤模式为线性时，将从基纹理层级 $level_{base}$ 的图像数组中选取 2×2×2 的纹素立方体。该立方体通过以下步骤生成：首先按第 3.8.7 节所述方法包裹纹理坐标，然后计算

纹理坐标进行折返处理，再计算

$$i_0 = \begin{cases} \lfloor u - 1/2 \rfloor \bmod w_t, & \text{纹理循环模式为重复} \\ \lfloor u - 1/2 \rfloor, & \textit{否则} \end{cases}$$

$$j_0 = \begin{cases} \lfloor v - 1/2 \rfloor \bmod h_t, & \texttt{TEXTURE\_WRAP\_T} \text{ is } \texttt{REPEAT} \\ \lfloor v - 1/2 \rfloor, & \textit{否则} \end{cases}$$

以及

$$k_0 = \begin{cases} \lfloor w - 1/2 \rfloor \bmod d_t, & \text{纹理循环 } \texttt{R} \text{ 为重复} \\ \lfloor w - 1/2 \rfloor, & \textit{否则} \end{cases}$$

然后

$$i_1 = \begin{cases} (i_0 + 1) \bmod w_t, & \text{纹理卷绕 } \texttt{S} \text{ 为重复} \\ i_0 + 1, & \textit{否则} \end{cases}$$

$$j_1 = \begin{cases} (j_0 + 1) \bmod h_t, & \text{纹理循环方向为重复} \\ j_0 + 1, & \textit{否则} \end{cases}$$

且

$$k_1 = \begin{cases} (k_0 + 1) \bmod d_s, & \text{纹理卷绕 R 为重复} \\ k_0 + 1, & \text{否则} \end{cases}$$

令

$$\alpha = \text{frac}(u - 1/2) \quad \beta = \text{frac}(v - 1/2) \quad \gamma = \text{frac}(w - 1/2)$$

其中 frac(x) 表示 x 的小数部分。

对于三维纹理，纹理值τ可通过以下公式计算：

$$\begin{aligned} \tau = {} & (1-\alpha)(1-\beta)(1-\gamma)\tau_{i_0 j_0 k_0} + \alpha(1-\beta)(1-\gamma)\tau_{i_1 j_0 k_0} \\ & + (1-\alpha)\beta(1-\gamma)\tau_{i_0 j_1 k_0} + \alpha\beta(1-\gamma)\tau_{i_1 j_1 k_0} \\ & + (1-\alpha)(1-\beta)\gamma\tau_{i_0 j_0 k_1} + \alpha(1-\beta)\gamma\tau_{i_1 j_0 k_1} \\ & + (1-\alpha)\beta\gamma\tau_{i_0 j_1 k_1} + \alpha\beta\gamma\tau_{i_1 j_1 k_1} \end{aligned}$$

其中 τ_{ijk}是三维纹理图像中位置 (i, j, k) 的纹理像素。

对于二维纹理，

$$\tau = (1-\alpha)(1-\beta)\tau_{i_0 j_0} + \alpha(1-\beta)\tau_{i_1 j_0} + (1-\alpha)\beta\tau_{i_0 j_1} + \alpha\beta\tau_{i_1 j_1} \quad (3.26)$$

其中 τ_{ij} 是二维纹理图像中位置 (i, j) 的纹理像素。

对于一维纹理，

$$\tau = (1-\alpha)\tau_{i_0} + \alpha\tau_{i_1}$$

其中 τ_i是单维纹理中位置 i 的纹理像素。

若上述方程中选取的τ_{ijk}、τ_{ij}或τ_i涉及边界纹素，且满足以下任一条件：$i < -b_s$、$j < -b_s$、$k < -b_s$、$i \geq w_s + b_s$、$j \geq h_s + b_s$或$d \geq d_s + b_s$，则使用由 TEXTURE BORDER COLOR 定义的边界值代替未指定的值。若纹理包含颜色分量，TEXTURE BORDER COLOR 的值将被解释为 RGBA 颜色值，以符合表 3.15 所示的纹理内部格式。若纹理包含深度分量，TEXTURE BORDER COLOR 的首个分量将被解释为深度值。

Mip映射

纹理最小化过滤 _ 值 最近米普贴图最近，最近米普贴图线性，
_ _ 线性MIP贴图最近， _
以及线性MIP贴图线性模式均需使用*MIP贴图*。MIP贴图是由多个数组组成的有序集合，每个数组代表同一图像的不同分辨率层级。

低于前一级。若级别 $level_{base}$的图像数组（不含边界）尺寸为 $w_b \times h_b \times d_b$，则该级MIP图包含 $[\log_2(\max(w_b, h_b, d_b)) + 1)$ 个图像数组。 将级别编号为：当级别 $level_{base}$为第 0 级时，第 i 级数组的尺寸为 w(i) × h(i) × d(i)。
第0级，则第i级数组的尺寸为

$$\max(1, \lfloor \frac{w_b}{2^i} \rfloor) \times \max(1, \lfloor \frac{h_b}{2^i} \rfloor) \times \max(1, \lfloor \frac{d_b}{2^i} \rfloor)$$

直到遇到维度为1的最后一个数组 $1^{(2)}$ 1.
× ×

每张Mipmap中的数组均通过**TexImage3D**、**TexImage2D**、**Copy-TexImage2D**、**TexImage1D**或**CopyTexImage1D**定义；设置的数组通过细节级别*参数level*标识。 细节级别编号从

$level_{base}$ 用于原始纹理数组，其中 $p = [\log_2(\max(w_b, h_b, d_b))] +$
$level_{base}$ 每次单位增加表示一个数组，其维度为
（小数部分向下取整为最近整数），如前所述。所有从 $level_{base}$ 到 $q = \min p, level_{max}$的数组都必须定义，详见第 3.8.10 节。 { }

对于特定纹理，可通过调用 **TexParameter[if]** 并分别将 *pname* 设置为 TEXTURE BASE LEVEL 或 TEXTURE MAX LEVEL 来重新指定 $level_{base}$和 $level_{max}$的值。 _ _

若任一值为负数，则会触发 INVALID VALUE 错误。

Mipmap与细节级别协同工作，用于近似应用经过适当过滤的纹理到片元。设c为*λ值*，当*λ大于c*时，从缩小到放大的过渡发生（由于本讨论涉及缩小，我们仅关注$\lambda > c$的情况）。

对于 mipmap 滤镜 最近邻MIPMAP最近邻 _ 以及
线性MIPMAP最近，最近MIPMAP最近，将选择第d个MIPMAP数组，其中

$$d = \begin{cases} level_{base}, & \lambda \le \frac{1}{2} \\ [层级_{(基数)} + \lambda + 1] - 1, & \lambda > \frac{1}{2}, 层级_{基数} + \lambda \le q + \frac{1}{2} \\ q, & \lambda > \frac{1}{2}, level_{base} + \lambda > q + \frac{1}{2} \end{cases} \quad (3.27)$$

随后将NEAREST或LINEAR过滤规则应用于所选数组。

对于 mipmap 滤波器 NEAREST MIPMAP LINEAR 和 LINEAR MIPMAP LINEAR，将选择级别 d_1和 d_2的mipmap 数组，其中

$$= \begin{cases} q, & \text{层级}_{基准} + \lambda \geq q \\ [level_{base} + \lambda], & \text{否则} \end{cases} \tag{3.28}$$

$$= \begin{cases} q, & level_{base} + \lambda \geq q \\ d_1 + 1, & \text{否则} \end{cases} \tag{3.29}$$

随后将最近邻或线性滤波规则应用于每个选定数组，得到两个对应的纹理值 τ_1和 τ_2。最终纹理值通过以下方式确定：

$$\tau = [1 - \lambda/\tau]\tau_1 + \lambda/\tau\tau_2.$$

自动生成Mipmap

如果纹理参数 GENERATE MIPMAP 的值为 TRUE，则对 MIP 贴图的*级别*$_{基准}$数组内部或边界纹素进行任何修改时，系统也会根据修改后的*级别*$_{基准}$数组计算出完整的衍生 MIP 贴图数组集（如第 3.8.10 节所定义）。 数组级别 $level_{base}$+ 1 至 p 将被替换为派生数组，无论其先前内容如何。所有其他 MIP 贴图数组（包括 $level_{base}$ 数组）均不受此计算影响。

派生Mipmap数组的内部格式和边界宽度均*与*$level_{base}$数组一致，且派生数组的维度需符合第3.8.10节所述要求。

衍生数组的内容通过对$_{基}$级数组进行重复过滤缩减计算得出。虽未强制要求特定过滤算法，但建议采用盒式滤波器作为默认滤波器。在某些实现中，过滤质量可能受提示信息影响（参见第5.6节）。

自动生成Mipmap的功能仅适用于非代理纹理图像目标。

3.8.9 纹理放大

当 λ 表示放大倍数时，TEXTURE MAG FILTER 的取值决定了纹理值的获取方式。TEXTURE MAG FILTER_有两种可能的取值：NEAREST 和 LINEAR。NEAREST的行为完全等同于TEXTURE MIN FILTER中的NEAREST（使用公式3.23、3.24和3.25）；LINEAR的行为完全等同于TEXTURE MIN FILTER中的LINEAR（使用公式3.26）。放大操作始终使用细节*级别*$level_{base}$的纹理数组。

最后是*c值*的选择，即缩放与放大切换点。若放大滤镜采用LINEAR算法，而缩小滤镜采用NEAREST MIPMAP NEAREST或NEAREST MIPMAP LINEAR算法，*则*$c = 0.5$。此设定旨在确保缩小后的纹理不会比放大后的纹理显得"更锐利"。*否则*$c = 0$。

3.8.10 纹理完整性

当所有用于纹理应用所需的图像数组和纹理参数均被完整定义时，该纹理即为完整。完整性的定义因纹理维度而异。

对于一维、二维或三维纹理，当以下条件均成立时，该纹理即为*完整*:

- 从*基级*$level_{base}$*到*q*级*（其中q在第3.8.8节的《Mipmapping讨论》中定义）的Mipmap数组集，均采用相同的内部格式进行指定。
- 各数组的边界宽度均相同。
- 数组维度遵循第3.8.8节**Mipmap**讨论中描述的序列。
- $level_{base} \leq level_{max}$
- *基准级*数组的每个维度均为正值。

数组层级k若满足$k < level_{base}$或$k > q$，则对完整性定义不具意义。

对于立方体贴图，当以下条件全部成立时，该贴图即为*立方体完整*:

- 构成立方体贴图的六个纹理图像各自的*基准*数组具有相同的正方形尺寸。
- 每个$level_{base}$数组均采用相同的内部格式定义。
- 每个*基础级别的*数组具有相同的边界宽度。

最后，若除满足立方体完整性外，六张纹理图像在单独考察时均达到完整性，则该立方体贴图纹理即为*完整Mipmap立方体*。

完整性对纹理应用的影响

当原始图形进行光栅化时，若纹理单元启用了单维、二维或三维纹理映射（但不包括立方体贴图），且TEXTURE MIN FILTER设置为需要米普贴图的模式，同时绑定至该启用纹理目标的纹理图像不完整，则该纹理单元将视为禁用了纹理映射功能。– –

如果在原始图形光栅化时启用了纹理单元的立方贴图纹理映射，且绑定的立方贴图纹理不满足立方完整性要求，则该纹理单元将视为禁用了纹理映射。此外，若TEXTURE MIN FILTER需要使用MIP贴图，而纹理不满足MIP立方完整性要求，则该纹理单元同样视为禁用了纹理映射。

– –

完整性对纹理图像规格的影响

实现仅允许创建级别为1或更高的纹理图像数组，前提是能够支持与请求数组一致的*完整米波*图图像数组集。 一个完整的米普图数组集等同于满足以下条件的完整数组集：$_{基准}$*级别*$_{(level_{base})}$= 0 且$_{最大}$*级别*$_{(level_{max})}$= 1000。其中，创建的图像数组在排除边界条件下，其维度被视为相邻较低编号数组对应维度的一半（若存在小数部分则向下取整至最接近的整数）。

3.8.11 纹理状态与代理状态

纹理所需的状态可分为两类。首先是九组Mipmap数组（分别对应一维、二维和三维纹理目标各一组，以及六组用于立方体贴图纹理目标）及其数量。每个数组关联以下属性：宽度、高度（仅限二维、三维及立方体贴图）、深度（仅限三维）、 边界宽度、描述图像内部格式的整数、描述图像红/绿/蓝/透明度/亮度/强度各通道分辨率的六个整数值、描述图像是否压缩的布尔值，以及压缩图像的整数尺寸。 初始纹理数组均为空（宽度、高度、深度均为零，边界宽度为零，内部格式为1，压缩标志设为FALSE，压缩后尺寸为零，各分量值均为零）。 接下来是两组纹理属性：每组包含选定的缩放与放大滤镜、*s*轴*与t轴*的包络模式（仅限二维/三维及立方贴图）、*r轴的*包络模式（仅限三维）、纹理边界颜色，以及描述最小值与最大值的两个整数。

– –

细节级别，两个描述基础和最大Mipmap数组的整数，一个布尔标志用于指示纹理是否驻留，一个布尔值用于指示是否应执行自动Mipmap生成，三个描述深度纹理模式、比较模式和比较函数的整数，以及与每组属性关联的优先级。驻留标志的值由GL确定，并可能因其他GL操作而改变。 该标志仅可由应用程序查询而不可设置（参见第3.8.12节）。初始状态下，TEXTURE MIN FILTER值设为NEAREST MIPMAP LINEAR，TEXTURE MAG FILTER值设为LINEAR。*s*、*t*和*r*的包裹模式均设为REPEAT。 TEXTURE MIN LOD与TEXTURE MAX LOD值分别为-1000和1000。TEXTURE BASE LEVEL与TEXTURE MAX LEVEL值分别为0和1000。TEXTURE PRIORITY为1.0，TEXTURE BORDER COLOR为(0,0,0,0)。 GENERATE MIPMAP的值是假的。DEPTH TEXTURE MODE、TEXTURE COMPARE MODE和 TEXTURE COMPARE FUNC 的值分别是 LUMINANCE、NONE 和 LEQUAL。TEXTURE RESIDENT 的初始值由 GL 确定。

除了一维、二维、三维及六组立方体贴图集的图像数组外，还维护着部分实例化的一维、二维、三维及一组立方体贴图集的代理图像数组。 每个代理数组包含宽度、高度（仅限二维和三维数组）、深度（仅限三维数组）、边界宽度及内部格式状态值，同时包含红、绿、蓝、Alpha、亮度和强度各通道的分辨率状态。代理数组不包含图像数据，也不包含纹理属性。**当**TexImage3D以PROXY TEXTURE 3D为*目标*执行时，指定细节等级的三维代理状态值将被重新计算并更新。 若图像数组无法通过*目标*设为TEXTURE 3D的TexImage3D调用获得支持，则不会产生错误，但代理宽度、高度、深度、边界宽度及各分量分辨率将被设为零。若图像数组可通过**此类TexImage3D**调用获得支持，则代理状态值将完全按实际图像数组被指定时的设置进行配置。两种情况下均不会传输或处理像素数据。

当TexImage1D执行时将*目标*指定为PROXY TEXTURE 1D，**或TexImage2D**执行时将*目标*指定为PROXY TEXTURE 2D时，一维和二维代理数组的操作方式相同。

当TexIm-age2D执行时，若*目标*字段指定为PROXY TEXTURE CUBE_MAP，则立方体贴图代理数组将以相同方式进行操作。需要补充说明的是：若通过PROXY TEXTURE CUBE MAP判定某立方体贴图纹理受支持，则意味着其全部六个立方体贴图二维图像均受支持。同样地，若指定的代理纹理立方贴图不受支持，则该立方贴图的全部六个二维图像均不被支持。

所有代理纹理均未关联图像。 因此，PROXY TEXTURE 1D、PROXY TEXTURE 2D、PROXY TEXTURE 3D以及PROXY TEXTURE_CUBE_MAP无法作为纹理使用，且其图像绝不能通过 **GetTexImage** 进行查询。若尝试此操作，将生成 INVALID ENUM 错误。同样地，代理纹理不存在与关卡无关的状态，因此不得使用代理纹理*目标*调用 **GetTexParameteriv** 或 **GetTexParameterfv**。若尝试此操作，将生成 INVALID ENUM 错误。

3.8.12 纹理对象

除默认纹理TEXTURE 1D、TEXTURE 2D、TEXTURE 3D和TEXTURE CUBE MAP外，还可创建并操作命名的一维、二维、三维及立方体贴图纹理对象。纹理对象的命名空间为无符号整数，其中零由GL保留。

纹理对象通过将未使用的名称*绑定*至TEXTURE 1D、TEXTURE 2D、TEXTURE 3D或TEXTURE CUBE MAP创建。绑定操作通过调用以下函数实现：

```
void BindTexture( enum target, uint texture );
```

将*目标*设置为所需的纹理目标，*并将纹理*设置为未使用的名称。生成的纹理对象是一个新的状态向量，包含第3.8.11节中列出的所有状态值，且初始值相同。若新纹理对象绑定至TEXTURE 1D、TEXTURE 2D、TEXTURE 3D或TEXTURE CUBE MAP，则
并保持为一维、二维、三维或立方体贴图纹理，直至被删除。

BindTexture 亦可用于将现有纹理对象绑定至 TEXTURE 1D、TEXTURE 2D、TEXTURE 3D 或 TEXTURE CUBE MAP。错误
若尝试绑定与指定*目标*维度不同的纹理对象，将触发无效操作。若绑定成功，则绑定纹理对象的状态保持不变，且此前与*该目标的*任何绑定关系均告解除。

当纹理对象处于绑定状态时，对其绑定目标执行的GL操作将影响该绑定对象，而查询其绑定目标将返回绑定对象的状态。若启用了与绑定目标维度匹配的纹理映射，则绑定纹理对象的状态将指导纹理操作。

在 中， 初始 状态， TEXTURE 1D, TEXTURE 2D, TEXTURE 3D, 以及TEXTURE_CUBE MAP分别关联着一维、二维、三维及立方贴图纹理状态向量。为避免初始纹理丢失，它们被视为名称均为0的纹理对象。因此初始的一维、二维、三维及立方贴图纹理均以TEXTURE 1、TEXTURE 2、TEXTURE 3和TEXTURE CUBE MAP形式进行操作、查询和应用。

初始纹理的访问权限不致丢失，它们被视为名称均为0的纹理对象。因此，初始的一维、二维、三维及立方体贴图纹理将分别以TEXTURE 1D、TEXTURE 2D、TEXTURE 3D或TEXTURE CUBE MAP的形式进行操作、查询和应用，同时0将绑定至对应的目标。

纹理对象通过调用

```
void DeleteTextures( sizei n, uint *textures );
```

textures 包含待删除的 *n* 个纹理对象名称。纹理对象删除后将失去内容与维度属性，其名称重新归为未分配状态。 若当前绑定至TEXTURE 1D、TEXTURE 2D、TEXTURE 3D或TEXTURE CUBE MAP目标的纹理被删除，其效果等同于执行BindTexture时指定相同*目标*并*设置*纹理为零。*textures*中未使用的名称将被静默忽略，数值零亦同。

命令

```
void GenTextures( sizei n, uint *textures );
```

*返回纹理中n个*先前未使用的纹理对象名称。这些名称仅在**生成纹理**时被标记为已使用，但它们在首次绑定时才会获取纹理状态和维度，如同未被使用时一样。

实现方可选择建立工作集，对其中纹理对象执行绑定操作时能获得更高性能。当前属于工作集的纹理对象称为*驻留对象*。命令

```
boolean AreTexturesResident( sizei n, uint *textures, boolean *residences );
```

*若textures中*列出的*所有n个*纹理对象均驻留内存，或实现未区分工作集，则返回TRUE。*若textures中*至少有一个纹理对象未驻留内存，则返回FALSE，并将每个纹理对象的驻留状态返回*至residences*。*否则residences*内容保持不变。 *若textures中的*名称存在未用或为零的情况，则返回FALSE，并生成INVALID VALUE错误，*residences*内容将不可预测。也可通过调用**Get**TexParameteriv**或GetTexParameterfv**查询单个绑定纹理对象的驻留状态：将target设为纹理对象绑定的目标，并将pname设为TEXTURE RESIDENT。

AreTexturesResident 仅指示纹理对象当前是否为驻留状态，而非其是否无法被设为驻留。实现方可选择

仅在首次使用时将纹理对象设为驻留。客户端可通过为每个纹理对象指定优先级，引导GL实现确定哪些纹理对象应驻留。命令

```
void PrioritizeTextures( sizei n, uint *textures, clampf *priorities );
```

将*textures*中命名的*n*个纹理对象的优先级设置为*priori-ties*中的值。每个优先级值在赋值前会被限制在[0,1]范围内。零表示最低优先级，驻留概率最低；一表示最高优先级，驻留概率最高。 单个绑定纹理对象的优先级也可通过**调用TexParameteri、TexParameterf、TexParameteriv或TexParameterfv**进行修改：将target设为该纹理对象的绑定目标，*p*name设为TEXTURE PRIORITY，*param*或*params*参数指定新优先级值（该值在赋值前会被限制在[0,1]范围内）。**PrioritizeTextures**会自动忽略对未使用的纹理对象名称或零（默认纹理）进行优先级设置的尝试。

纹理对象命名空间（包括初始的一维、二维和三维纹理对象）在所有纹理单元间共享。单个纹理对象可同时绑定至多个纹理单元。纹理对象绑定后，对该目标对象执行的任何GL操作都会影响所有绑定该纹理对象的其他纹理单元。

纹理绑定受状态设置ACTIVE TEXTURE的影响。

若删除某个纹理对象，则相当于将所有绑定该对象的纹理单元重新绑定至纹理对象零。

3.8.13 纹理环境与纹理函数

命令

```
void TexEnv{if}( enum target, enum pname, T param ); void TexEnv{if}v( enum
target, enum pname, T params );
```

设置*纹理环境*参数，该环境规定了在对片段进行纹理映射时如何解释纹理值，或设置每个纹理单元的过滤参数。*目标*必须为为以下之一点精灵、纹理环境或纹理过滤控制。*pname*是表示待设置参数的符号常量。在命令的第一种形式中，param是用于设置单值参数的数值；在第二种形式中，*params*是参数数组的指针：该数组可包含单个符号常量、单个数值或需设置参数的数值组。

。

当*目标*为点精灵时，点精灵光栅化行为将受到第3.3节所述的影响。

当*目标*为纹理过滤控制时，*pname*必须为纹理LOD偏移量。此时该参数为单精度浮点值*偏移量*$_{texunit}$，用于偏移细节级别*参数*k，具体行为参见第3.8.8节。

当*目标*为时，TEXTURE ENV参数为：可能的环境参数包括TEXTURE ENV MODE、TEXTURE ENV COLOR、COMBINE RGB和COMBINE ALPHA。TEXTURE ENV MODE可设置为REPLACE、MODULATE、DECAL、BLEND、ADD或COMBINE之一。TEXTURE ENV COLOR通过提供四个单精度浮点值（范围[0,1]，超出范围值将被限制在此范围内）设置为RGBA颜色。若为TEXTURE ENV COLOR提供整数值，则参数为：COMBINE ALPHA。
通过提供四个单精度浮点值（取值范围为[0, 1]，超出范围的值将被截断）来设置RGBA颜色。若为TEXTURE ENV COLOR提供整数值，则按表2.9中带符号整数的转换规则将其转换为浮点数。

TEXTURE ENV_MODE_的值指定*了一种纹理函数*。该函数的结果取决于片段和纹理数组的值。函数的具体形式取决于上次指定的纹理数组的基础内部格式。

C_f和A_f[3]是传入片元的主色分量；C_s和A_s是纹理源颜色的分量，由表 3.21 所示的滤波纹理值R_t、G_t、B_t、A_t、L_t和I_t推导得出；C_c和A_c是纹理环境颜色的分量；C_p和A_p是来自先前纹理环境的分量（对于纹理环境 0，C_p和A_p分别等同于C_f和A_f）；C_v和A_v是由纹理函数计算得出的主色分量。

所有这些颜色值均在[0, 1]范围内。纹理函数在表3.22、3.23和3.24中给出。

若纹理环境模式（TEXTURE ENV MODE）的值为组合（COMBINE），则纹理函数的形式取决于组合RGB（COMBINE RGB）和组合Alpha（COMBINE ALPHA）的值，具体参照表3.24。随后，纹理函数的RGB与Alpha结果将分别乘以RGB缩放（RGB SCALE）和Alpha缩放（ALPHA SCALE）的值，最终结果被限制在[0, 1]范围内。

参数Arg0、Arg1和Arg2由以下值决定：
SRCn RGB、*SRCn* ALPHA、*OPERANDn* RGB 和 *OPERANDn* ALPHA，其中 $n = 0$、1 或 2，如表 3.25 和 3.26 所示。C_s^n和A_s^n分别表示绑定到纹理单元 n 的纹理图像中的纹理源颜色和透明度。

[3] 在第3.8.13节的剩余部分中，符号C_x用于表示由x指定的颜色中三个分量R_x、G_x和B_x中的任意一个。对C_x的操作对每个颜色分量独立执行。颜色的A分量通常采用不同方式处理，因此单独用A_x表示。

纹理基底 内部格式	纹理源颜色 C_s	A_s
ALPHA	(0, 0, 0)	A_t
LUMINANCE	(L_t, L_t, L_t)	1
亮度阿尔法 _	(L_t, L_t, L_t)	A_t
强度	(I_t, I_t, I_t)	I_t
RGB	(R_t, G_t, B_t)	1
RGBA	(R_t, G_t, B_t)	A_t

表 3.21： 过滤后的纹理组件与纹理源组件的对应关系。

纹理基准 内部格式	REPLACE 功能	MODULATE 功能	DECAL 功能
ALPHA	$C_v = C_p$ $A_v = A_s$	$C_v = C_p$ $A_v = A_p A_s$	*未定义*
亮度 (或 1)	$C_v = C_s$ $A_v = A_p$	$C_v = C_p C_s$ $A_v = A_p$	*未定义*
亮度阿尔法 _ (或 2)	$C_v = C_s$ $A_v = A_s$	$C_v = C_p C_s$ $A_v = A_p A_s$	*未定义*
强度	$C_v = C_s$ $A_v = A_s$	$C_v = C_p C_s$ $A_v = A_p A_s$	*未定义*
RGB (或 3)	$C_v = C_s$ $A_v = A_p$	$C_v = C_p C_s$ $A_v = A_p$	$C_v = C_s$ $A_v = A_p$
RGBA (或 4)	$C_v = C_s$ $A_v = A_s$	$C_v = C_p C_s$ $A_v = A_p A_s$	$C_v = C_p (1 - A_s) + C_s A_s$ $A_v = A_p$

表 3.22： 纹理函数 REPLACE、MODULATE 和 DECAL。

纹理基底 内部格式	混合 函数	ADD 功能
ALPHA	$C_v = C_p$ $A_v = A_p A_s$	$C_v = C_p$ $A_v = A_p A_s$
亮度 (或 1)	$C_v = C_p (1 - C_s) + C_c C_s$ $A_v = A_p$	$C_v = C_p + C_s$ $A_v = A_p$
亮度阿尔法 (或 2)	$C_v = C_p (1 - C_s) + C_c C_s$ $A_v = A_p A_s$	$C_v = C_p + C_s$ $A_v = A_p A_s$
强度	$C_v = C_p (1 - C_s) + C_c C_s$ $A_v = A_p (1 - A_s) + A_c A_s$	$C_v = C_p + C_s$ $A_v = A_p + A_s$
RGB (或 3)	$C_v = C_p (1 - C_s) + C_c C_s$ $A_v = A_p$	$C_v = C_p + C_s$ $A_v = A_p$
RGBA (或 4)	$C_v = C_p (1 - C_s) + C_c C_s$ $A_v = A_p A_s$	$C_v = C_p + C_s$ $A_v = A_p A_s$

表 3.23： 纹理函数 BLEND 和 ADD。

当前纹理环境所需的状态，对于每个纹理单元，包含：- 一个六值整数，指示纹理函数；- 一个八值整数，指示RGB组合器函数；- 一个六值整数，指示ALPHA组合器函数；- 六个四值整数，指示组合器RGB和ALPHA源参数；- 三个四值整数，指示组合器RGB操作数；- 三个二值整数，指示组合器ALPHA操作数；- 四个浮点环境颜色值。三个二值整数表示合成器ALPHA操作数，以及四个浮点环境颜色值。初始状态下，纹理与合成器函数均为MODULATE，合成器RGB与ALPHA源分别为TEXTURE、 PREVIOUS和CONSTANT（分别对应源0、1、2），组合器RGB操作数（源0和1）均为SRC COLOR，组合器RGB操作数（源2）及组合器ALPHA操作数均为SRC ALPHA，环境颜色为(0, 0, 0, 0)。

每个纹理单元所需的纹理过滤参数状态包含一个浮点细节级别偏移量。该偏移量的初始值为0.0。

3.8.14 纹理比较模式

纹理值也可根据指定的比较函数计算。参数TEXTURE COMPARE MODE指定比较操作数，参数TEXTURE COMPARE FUNC指定比较函数。最终纹理采样的格式由

COMBINE RGB	纹理函数
替换	Arg0
调制	Arg0 * Arg1
ADD	Arg0 + Arg1
有符号加法	Arg0 + Arg1 − 0.5
插值	Arg0 乘以 Arg2 加 Arg1 乘以 (1 - Arg2)
SUBTRACT	Arg0 − Arg1
点3 RGB	$4 \times ((Arg0_r - 0.5) \times (Arg1_r - 0.5) + (Arg0_g - 0.5) \times (Arg1_g - 0.5) + (Arg0_b - 0.5) * (Arg1_b - 0.5))$
点阵3 RGBA	$4 \times ((_{r的}$相位角 − 0.5) 乘以 ($_{g的相位角}$ − 0.5) 加上 ($_{r的}$相位角 − 0.5) 乘以 ($_{g的相位角}$ − 0.5) 加上 $(Arg0_b - 0.5) * (Arg1_b - 0.5))$

COMBINE ALPHA	纹理函数
替换	Arg0
调制	Arg0 * Arg1
ADD	Arg0 + Arg1
有符号加法	Arg0 + Arg1 − 0.5
插值	Arg0 乘以 Arg2 加 Arg1 乘以 (1 - Arg2)
减法	Arg0 − Arg1

表 3.24：COMBINE 纹理函数。DOT3 RGB 和 DOT3 RGBA 函数计算出的标量表达式会被分别置入输出结果的 3 个（RGB）或 4 个（RGBA）分量中。对于 DOT3 RGBA，COMBINE ALPHA 生成的结果会被忽略。

SRCn RGB	*操作数n* RGB	参数
纹理	SRC COLOR 源颜色减一源透明度 1减去源颜色	C_s $1 - C_s$ A_s $1 - A_s$
纹理n	SRC COLOR SRC 减去颜色 SRC 减去透明度 1减去SRC透明度	C_s^n $1 - C_s^n A_s^n$ $1 - A_s^n$
常数	SRC 色彩 源颜色减一源透明度 1减去SRC透明度	C_c $1 - C_c$ Ac $1 - A_c$
主色	SRC 颜色 减去源颜色源颜色透明度 1减去SRC阿尔法	C_f $1 - C_f$ A_f $1 - A_f$
PREVIOUS	SRC COLOR ONE MINUS SRC COLOR SRC ALPHA 一减源图Alpha	C_p $1 - C_p$ A_p $1 - A_p$

表 3.25：COMBINE RGB 函数的参数。

SRCn ALPHA	*操作数n* 透明度	参数
纹理	源Alpha 减去源Alpha	A_s $1 - A_s$
纹理n	SRC ALPHA 源Alpha减一	A_s^n $1 - A_s^n$
常数	SRC ALPHA 1减去SRC阿尔法	A_c $1 - A_c$
原色	SRC 阿尔法 一减SRC阿尔法	A_f $1 - A_f$
PREVIOUS	SRC ALPHA 减去 SRC ALPHA	A_p $1 - A_p$

表 3.26：COMBINE ALPHA 函数的参数。

深度纹理模式。

深度纹理比较模式

若当前绑定纹理的基础内部格式为深度分量，则
纹理比较模式、纹理比较函数和深度纹理模式
将按下述方式控制纹理单元的输出。否则，纹理单元将按常规方式运行，并跳过纹理比较操作。

设 D_t为深度纹理值，取值范围为 [0, 1]，R 为插值后的纹理坐标（限制在 [0, 1] 范围内）。则有效纹理值 L_t、I_t或 A_t的计算方式如下：

若纹理比较模式为NONE，则

$$r = D_t$$

若纹理比较模式为COMPARE R TO TEXTURE，则 $r=D(t)$
取决于表3.27所示的纹理比较函数。

纹理比较函数	计算结果 r
LEQUAL	$r=\begin{cases}1.0, & R \le D_t \\ 0.0, & R > D_t\end{cases}$
GEQUAL	$r=\begin{cases}1.0, & R \ge D_t \\ 0.0, & R < D_t\end{cases}$
LESS	$r=\begin{cases}1.0, & R < D_t \\ 0.0, & R \ge D_t\end{cases}$
大于	$r=\begin{cases}1.0, & R > D_t \\ 0.0, & R \le D_t\end{cases}$
等于	$r=\begin{cases}1.0, & R = D_t \\ 0.0, & R \ne D_t\end{cases}$
NOTEQUAL	$r=\begin{cases}1.0, & R = D_t \\ 0.0, & R = D_t\end{cases}$
始终	$r = 1.0$
永不	$r = 0.0$

表 3.27： 深度纹理比较函数。

最终的 r 被赋值给 L_t， I_t， 或 A_t 。
深度纹理模式分别为亮度、强度或透明度。

若纹理放大滤波器值非最近邻，或纹理最小滤波器值非最近邻或最近邻MIP贴图最近邻，则纹理最小过滤值不是最近邻或最近邻MIP贴图最近邻，则r 可能通过将多个深度纹理值与纹理R坐标进行比较来计算。具体实现方式取决于具体实现，但r应为[0, 1]范围内的值，且该值与比较通过或失败的次数成正比。

3.8.15 纹理应用

纹理功能通过通用**启用**和禁用命令分别开启或关闭，可使用符号常量TEXTURE 1D、TEXTURE 2D、TEXTURE 3D或TEXTURE CUBE MAP分别启用一维、二维、三维或立方体贴图纹理。若同时启用二维与一维纹理，则采用二维纹理。若三维纹理与二维或一维纹理中任一启用，则采用三维纹理。若立方贴图纹理与三维、二维或一维纹理中任一启用，则采用立方贴图纹理。若所有纹理均禁用，则将未修改的光栅化片段传递至GL下一阶段（尽管其纹理坐标可能被丢弃）。否则，将根据第3.8.6至3.8.9节规则，依据当前绑定纹理图像的参数值，从对应维度中查找纹理值。该纹理值将与输入片段共同用于计算当前绑定纹理环境指定的纹理函数。该函数的结果将替换传入片段的主R、G、B和A值。这些颜色值将传递给后续操作。除纹理坐标可能被丢弃外，与传入片段相关的其他数据保持不变。

每个纹理单元均独立于其他纹理单元启用并绑定至纹理对象。

。每个纹理单元遵循针对单维、双维三维及立方体贴图的优先级规则。因此不同维度的贴图映射可由各单元并行执行。每个单元均拥有独立的启用状态与绑定状态。

每个纹理单元都配有一个环境函数，如图3.11所示。第二个纹理函数通过以下方式计算：使用第二个纹理的纹理值、第一个纹理函数计算得到的片段以及第二个纹理单元的环境函数。若存在第三个纹理，则将第二个纹理函数生成的片段与第三个纹理值结合，并使用第三个纹理单元的环境函数进行处理，依此类推。**由ActiveTexture选定的**纹理单元决定了TexEnv调用将修改哪个纹理单元的环境。

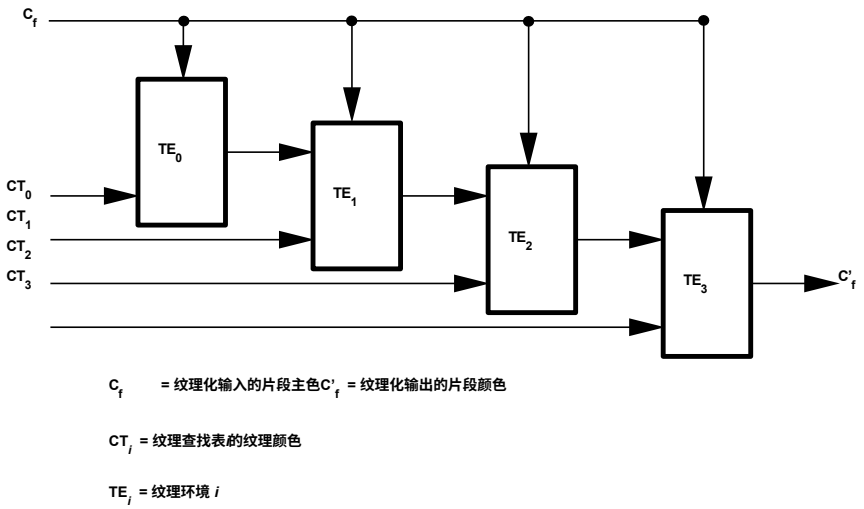


图3.11. 多纹理处理流水线。图中展示了四个纹理单元；但多纹理处理支持的单元数量可能因具体实现而异。输入的片段颜色会根据对应纹理环境的状态，依次与每个纹理进行组合，最终生成的片段颜色作为输入传递至流水线中的下一个纹理单元。

若纹理环境模式（TEXTURE ENV MODE）的值为组合（COMBINE），则*通过*SRCn RGB、*SRCn* ALPHA、*OPERANDn RGB*和*OPERANDn* ALPHA指定的值来计算与特定纹理单元关联的纹理函数。若将*TEXTUREn*指定为*SRCn* RGB或*SRCn*_ALPHA，则计算该纹理单元的纹理函数时将使用纹理*单元n*的纹理值。

纹理处理功能可针对每个纹理单元单独启用或禁用。若某单元禁用纹理处理，则前一单元生成的片段将保持原样传递至后续单元。超过 MAX TEXTURE UNITS参数指定数量的纹理单元将始终被视为禁用状态。

若纹理单元被禁用，或其绑定的纹理存在无效或不完整的情况（如第3.8.10节所定义），则该纹理单元的混合功能将被禁用。当某个已启用的纹理单元所对应的纹理环境引用了被禁用的纹理单元，或引用了绑定至其他单元的无效/不完整纹理时，

纹理混合的结果将无法定义。

每个纹理单元所需的状态为四位，用于指示一维、二维、三维或立方贴图纹理化功能的启用或禁用状态。初始状态下，所有纹理单元的所有纹理化功能均处于禁用状态。

3.9 颜色求和

在颜色合成开始时，每个片段拥有两个RGBA颜色值：主颜色 $\mathbf{c}_{pri}$（若启用纹理映射则可能被修改）和次要颜色 $\mathbf{c}_{sec}$。

若启用颜色相加功能，则将这两种颜色的R、G、B分量相加，生成单一后纹理化RGBA**颜色c**。其中**c的**A分量取自$\mathbf{c}_{pri}$的A分量；$\mathbf{c}_{sec}$的A分量不予使用。随后将**c的**各分量限制在[0, 1]范围内。若禁用颜色相加功能，则直接将$\mathbf{c}_{pri}$赋值**给c**。

颜色求和功能通过通用**启用**和**禁用**命令分别启用或禁用，使用符号常量COLOR SUM进行控制。当灯光功能启用且顶点着色器未激活时，颜色求和阶段始终生效，此时COLOR SUM的值将被忽略。

所需状态为单比特位，用于指示颜色合成是否启用。初始状态下颜色合成处于禁用状态。

在颜色索引模式下，或当片段着色器处于活动状态时，颜色合成功能无效。

3.10 雾效

若启用雾效，系统将通过混合*因子f将*雾色与光栅化片段的后纹理颜色进行融合。雾效的启用与禁用需通过**启用**和**禁用**命令配合符号常量FOG实现。

该*因子f*根据以下三种方程之一计算：

$$f = \exp(-d \cdot c), \tag{3.30}$$

$$f = \exp(-(d \cdot c)^2), \text{ 或} \tag{3.31}$$

$$f = \frac{e\hat{}(-c)}{e - s} \tag{3.32}$$

若顶点着色器处于活动状态，或雾源（如下定义）为 FOG COORD，则 c 为该片段雾坐标的插值值。否则，若雾源为 FRAGMENT DEPTH，则 c 为从视点坐标距离（在视点坐标系中为 (0, 0, 0, 1)）到片段中心的距离。该方程与雾源，以及 d 或 e 和 s 共同决定了

（0, 0, 0, 1）到片段中心的距离。该方程与雾源，以及*d*或*e*和*s*的组合，通过以下方式指定：

```
void Fog{if}(枚举 pname, T 参数);void Fog{if}v(枚举 pname, T
参数);
```

若参数名称为FOG MODE，则*参数*必须为，或*参数*必须指向一个整数，该整数应为符号常量EXP、EXP2或LINEAR之一。此时，雾计算将分别采用公式3.30、3.31或3.32（若选择3.32时且*e=s*，则结果未定义）。 若 *pname* 为 FOG COORD SRC，则 *param* 必须为，或 *params* 必须指向一个整数，该整数为符号常量 FRAGMENT DEPTH 或 FOG COORD 之一。若 *pname* 为 FOG DENSITY、FOG START 或 FOG END，则 *param* 或 *params* 指向的值分别对应 *d*、*s* 或 *e*。若指定的 *d* 小于零，则返回错误 INVALID VALUE。

实现方案可选择通过z_e近似计算从视点到每个片段中心的视点坐标距离。此外，*函数f*无需在每个片段处计算，而可在每个顶点处计算后，如同其他数据般进行插值处理。

无论采用何种方程与近似法*计算*，结果均需限制在[0, 1]区间内以获得*最终f值*。

f 的使用方式取决于图形渲染器处于 RGBA 模式还是颜色索引模式。在 RGBA 模式下，若 C_r 表示光栅化片段的 R、G 或 B 值，则雾效生成的对应值为：

$$C = f\,C_r + (1 - f)\,C_f \text{。}$$

(栅格化片段的A值不受雾混合影响。) C_f的R、G、B和A值**通过调用Fog**函数并设置*p*name为FOG COLOR来指定；此时*params*指向构成$C_{(f)\,89}$四个值。若这些值非浮点数，则按表2.9中带符号整数的转换规则转换为浮点数。 C_f的每个分量在指定时会被限制在 [0, 1] 范围内。

在颜色索引模式下，雾混合的计算公式为：

$$I = i_r + (1 - f)\,i_f$$

其中 i_r是光栅化片段的颜色索引，i_f是单精度浮点数。 $(1 - f)\,i_f$ 需四舍五入至与 i_r 相同位数的二进制点右侧固定点值，并将 *I* 的整数部分与 $2^n - 1$ 进行掩码（位与运算），其中 n 为

颜色索引缓冲区中的颜色（缓冲区详见第4章）。通过**调用Fog**函数设置pname为FOG INDEX，并将param或指向雾度索引单一值的*params作为参数*传递，即可设定i_f的值。i_f的整数部分需进行掩码处理：$2^n - 1$。

雾效所需的状态参数包括：用于选择雾效方程的三值整数、三个浮点*数d*、*e和s*、RGBA雾色值及雾色索引、用于选择雾坐标源的二值整数，以及一个用于指示雾效是否启用的单比特位。 初始状态下雾效禁用，雾坐标源为片段深度，雾模式为指数模式，d=1.0，e=1.0，s=0.0；$C_f = (0, 0, 0, 0)$*且*$i_f = 0$。

若片段着色器处于活动状态，雾效将失效。

3.11 片段着色器

如第3.8至3.10节所述，对点、线段、多边形、像素矩形或位图进行光栅化后生成的片元所执行的操作序列，属于处理此类片元的固定功能方法。应用程序可通过*片元着色器*更通用地描述对这些片元执行的操作。

片段着色器是一组字符串数组，包含针对每个片段执行的操作源代码。这些操作针对光栅化后的点、线段、多边形、像素矩形或位图所生成的片段进行处理。片段着色器使用的语言在《OpenGL着色语言规范》中有详细描述。

片段着色器仅在图形渲染器处于RGBA模式时生效。其在颜色索引模式下的行为未定义。

片段着色器需遵循第2.15.1节所述方式创建，其*类型*参数为FRAGMENT SHADER。其附加与使用方式详见第2.15.2节对程序对象的描述。

当当前使用的程序对象包含片段着色器时，其片段着色器被视为*活动*状态，并用于处理片段。若程序对象未包含片段着色器，或当前未使用任何程序对象，则采用前文所述的固定功能片段处理操作。

3.11.1 着色器变量

片段着色器可访问当前着色器对象所属的统一变量。片段着色器统一变量可用的存储空间由实现相关的常量MAX_FRAGMENT_UNIFORM_COMPONENTS 指定。该值代表片段着色器统一变量存储器中可容纳的独立浮点型、整型或布尔型

值的数量。若尝试使用的空间超过片段着色器统一变量存储的可用空间，则会引发链接错误。

片段着色器可读取与光栅化生成的片段属性相对应的变量。 OpenGL着色语言规范定义了一组可由片段着色器访问的内置变量。这些内置变量包含用于固定功能片段处理的数据，例如片段位置、颜色、次要颜色、纹理坐标、雾坐标以及视点Z坐标。

此外，当顶点着色器处于活动状态时，可定义一个或多个*顶点变量*
变量（详见第2.15.3节及OpenGL着色语言规范）。这些值将在渲染的基元上进行插值计算。当片段着色器中定义了同名变量时，即可获取这些插值结果。

用户定义的变量在当前光栅化位置不会被保存。处理像素矩形或位图光栅化生成的片段时，用户定义变量的值为未定义状态。内置变量则具有明确定义的值。

3.11.2 着色器执行

若片段着色器处于活动状态，则使用其可执行版本处理来自点、线段、多边形、像素矩形或位图光栅化的输入片段值，而非第3.8至3.10节所述的固定功能片段处理。特别地，

- 第3.8.13节所述的纹理环境和纹理函数将不被应用。
- 第3.8.15节所述的纹理应用不被采用。
- 第3.9节所述的颜色求和操作不予执行。
- 第3.10节所述的雾效果未应用。

纹理访问

当片段着色器执行纹理查找时，GL会按第3.8.8节和第3.8.9节所述方式计算滤波后的纹理*值*r，并根据表3.21（第3.8.13节）将其转换为纹理源*颜色*C_s。

GL向片段着色器返回一个四分量向量(R_s, G_s, B_s, A_s)。为细节级别计算之目的，导数$\frac{du}{dx}$, $\frac{du}{dy}$, $\frac{dv}{dx}$, $\frac{dv}{dy}$, $\frac{dw}{dx}$, $\frac{dw}{dy}$可通过差分算法进行近似计算，具体方法详见《OpenGL着色语言规范》第8.8节。

涉及深度分量数据的纹理查找可直接返回深度数据，或返回与用于执行查找的纹理坐标的比较结果。 着色器通过使用阴影采样器类型（sampler1DShadow 或 sampler2DShadow）请求比较操作，纹理则通过 TEXTURE COMPARE MODE 参数请求。这些请求必须保持一致；若满足以下条件，纹理查找结果将未定义：

- • 纹理查找函数使用的采样器类型为 sampler1D 或 sampler2D，且纹理对象的内部格式为深度分量，同时纹理比较模式未设置为 NONE。
- 在纹理查找函数中使用的采样器类型为sampler1DShadow或sampler2DShadow，且纹理对象的内部格式为 DEPTH COMPONENT，TEXTURE COMPARE MODE设置为NONE。
- 纹理查找函数中使用的采样器类型为 sampler1DShadow 或 sampler2DShadow，且纹理对象的内部格式并非 DEPTH COMPONENT。

如果片段着色器使用了采样器，而该采样器关联的纹理对象未完成（如第3.8.10节所定义），则纹理图像单元将返回(*R, G, B, A*)
= (0, 0, 0, 1)。

在渲染单个基元期间，片段着色器内部可访问的独立纹理单元数量由实现相关的常量MAX TEXTURE IMAGE UNITS规定。

着色器输入

OpenGL着色语言规范描述了可作为片段着色器输入的值。

内置变量 glFragCoord 存储窗口坐标 *x*、*y*、z，
以及 $\frac{1}{w}$ 用于该片段。gl FragCoord 的 *z* 分量经历了...
转换为浮点数。该转换必须保持0和1的值不变。需注意 *该z*分量已包含多边形偏移量（若
启用时（参见第3.5.5节）。该$\frac{1}{w}$值由w_c坐标计算得出（参见

第2.11节），该坐标是投影矩阵与顶点视点坐标的乘积结果。

内置变量 glColor 和 glSecondaryColor 分别存储片段颜色和次级颜色的 R、G、B 和 A 颜色分量。每个定点颜色分量都会隐式转换为浮点数。该转换必须保持值 0 和 1 不变。

内置变量 glFrontFacing 若片段由正面基元生成则设为 TRUE，否则设为 FALSE。 对于由多边形、三角形或四边形基元生成的片元（包括由点或线渲染的多边形），其判定依据是通过检查第2.14.1节方程2.6计算的面积符号（包括由**FrontFace**控制的符号反转）。若符号为正，则该基元生成的片元为正面朝向；否则为背面朝向。 其余所有片段均视为正面片段。

着色器输出

OpenGL着色语言规范描述了片段着色器可能输出的值，包括gl FragColor、gl FragData[n]和gl FragDepth。片段着色器写入的最终片段颜色值或最终片段数据值将被限制在[0, 1]范围内，随后按第2.14.9节所述转换为定点数值。 片段着色器写入的最终片段深度值，首先会被限制在[0, 1]范围内，随后如同窗口Z值般转换为固定小数点数（参见第2.11.1节）。需注意此处仅执行固定小数点转换，不进行深度范围计算。

向 gl FragColor 写入数据将指定后续渲染管线阶段使用的片段颜色（颜色编号为零）。 向 gl FragData[n] 写入数据将指定片段颜色编号 *n* 的值。任何未由片段着色器写入的、与片段相关的颜色或颜色分量均未定义。 片段着色器不得同时对 gl FragColor 和 gl FragData 进行静态赋值。此类情况将导致编译或链接错误。若着色器在预处理后包含会写入变量的语句（无论运行时控制流是否触发该语句执行），则视为对变量进行了静态赋值。

向 gl-FragDepth 写入值可指定当前处理的片段的深度值。若活动片段着色器未对 gl FragDepth 进行静态赋值，则后续渲染管线阶段将使用光栅化过程中生成的深度值。 否则将采用gl FragDepth的赋值结果，但对于未执行 gl EragDepth赋值语句的片段，该值将处于未定义状态。因此，若着色器对gl FragDepth进行了静态赋值

gl_FragDepth，则必须始终负责写入该值。

3.12 抗锯齿应用

如果生成光栅化片段的原始图形启用了抗锯齿功能，则计算出的覆盖值将应用于该片段。在RGBA模式下，该值将与片段的Alpha（A）值相乘以获得最终Alpha值。在颜色索引模式下，该值将用于设置颜色索引值的低位，具体操作如第3.2节所述。覆盖值将分别应用于每个片段的颜色。

3.13 多采样点淡出

最后，若启用多采样且光栅化片段源自点基元，则将公式3.2计算的淡出因子应用于该片段。在RGBA模式下，淡出因子与片段的Alpha值相乘以得出最终Alpha值；在颜色索引模式下，淡出因子无效。淡出因子需分别应用于每个片段颜色。

第4章

片段级操作与帧缓冲区

帧缓冲区由一组像素构成，这些像素以二维数组的形式排列。该数组的高度和宽度可能因不同的GL实现而有所不同。在本讨论中，帧缓冲区中的每个像素仅被视为若干位组成的集合。每个像素所占用的位数同样可能因具体的GL实现或上下文而变化。

帧缓冲区中每个像素的对应位被组合成一个*位平面*；每个位平面包含每个像素的单个位。这些位平面被分组为若干*逻辑缓冲区*，即*颜色*缓冲区、*深度*缓冲区、模板缓冲区*和*累加缓冲区。颜色缓冲区实际上由多个缓冲区构成：*左前*缓冲区、*右前*缓冲区、*左后*缓冲区、*右后*缓冲区以及若干*辅助*缓冲区。 通常前置缓冲区的内容会显示在彩色显示器上，而后置缓冲区的内容不可见。（单视场景仅显示左前缓冲区；立体场景同时显示左前和右前缓冲区。）辅助缓冲区的内容永远不可见。 所有颜色缓冲区必须具有相同数量的位平面，但具体实现或上下文可选择完全不提供右缓冲区、后缓冲区或辅助缓冲区。此外，具体实现或上下文也可不提供深度缓冲区、模板缓冲区或累加缓冲区。

颜色缓冲区由无符号整数颜色索引或R、G、B（以及可选的A）无符号整数值构成。每个颜色缓冲区、深度缓冲区、模板缓冲区和累加缓冲区的位平面数量是固定的且取决于窗口。若提供累加缓冲区，其每个R、G、B颜色组件的位平面数量必须至少等于颜色缓冲区的位平面数量。

所有提供的位平面初始状态均未定义。

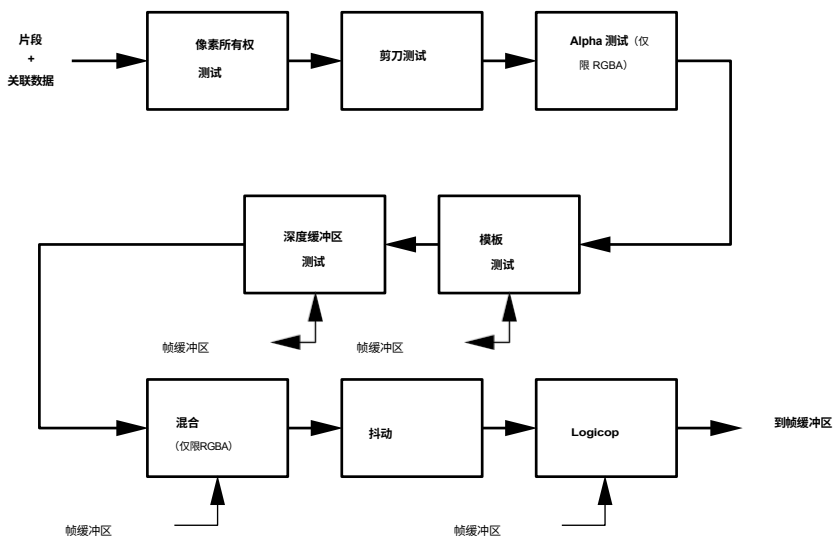


图4.1. 片段级操作。

4.1 片段操作

通过光栅化生成的片段（其窗口坐标为(x_w, y_w)）会根据若干参数和条件，修改帧缓冲区中该位置的像素。我们将按执行顺序描述这些修改与测试操作（如图4.1所示）。图4.1展示了这些 修改与测试流程。

4.1.1 像素所有权测试

首次测试用于确定帧缓冲区中坐标为(x_w, y_w)的像素当前是否由GL（更精确地说，由当前GL上下文）拥有。 若非如此，窗口系统将决定传入片段的处理方式。可能的结果包括：片段被丢弃，或后续部分片段操作被应用于该片段。此检测机制使窗口系统能够控制GL的行为，例如当GL窗口被遮挡时。

4.1.2 剪裁测试

剪刀测试用于判定点 (x_w, y_w) 是否位于由四个数值定义的剪刀矩形区域内。这些数值通过以下函数设置：

```
void Scissor( int left, int bottom, sizei width, sizei height );
```

若*左侧* x_w < *左边* + *宽度* 且 *底部* y_w < *底部* + *高度*，则剪刀测试通过。否则测试失败，该片段被丢弃。该测试通过常量 SCISSOR TEST 的 **Enable** 或 **Disable** 启用或禁用。禁用时，效果等同于剪刀测试始终通过。若*宽度*或*高度*任一值小于零，则生成 INVALID VALUE 错误。 所需状态包含四个整数值及一个标记测试启用的位。初始状态下 *left* = *bottom* = 0；*width* 和 *height* 由 GL 窗口尺寸决定。剪裁测试默认禁用。

4.1.3 多采样片段操作

此步骤根据SAMPLE ALPHA TO COVERAGE、SAMPLE ALPHA TO ONE、SAMPLE COVERAGE、SAMPLE COVERAGE VALUE和SAMPLE COVERAGE INVERT的数值修改片段透明度与覆盖值。若MULTISAMPLE功能禁用，或SAMPLE BUFFERS值非1，则本步骤不会生成

若多采样功能被禁用，或采样缓冲区数量值非1，则本步骤不修改片段透明度或覆盖值。

SAMPLE ALPHA TO COVERAGE、 采样α值转为1、 以及样本覆盖率通过**调用Enable和Disable**启用或禁用，*其中cap*参数需指定为三个标记值之一。调用IsEnabled并设置cap为目标标记值可查询全部三个值的状态。若启用"样本Alpha值覆盖率"功能，将生成临时覆盖率值，其中每位取决于对应采样位置的Alpha值。该临时值随后与片段覆盖率值进行与运算。 否则此时片段覆盖值保持不变。若片段着色器写入多组颜色，则采用片段颜色零的透明度值来确定临时覆盖值。

将采样alpha值转换为临时覆盖值时无需特定算法。设计意图是使临时覆盖值中的1的数量与片段的alpha值集成正比：全部为1对应所有alpha值的最大值，全部为0则表示所有alpha值均为0。该算法还应具有伪随机特性，以避免因覆盖采样位置规律性导致的图像伪影。 该算法在不同像素位置可以且应当有所差异。若存在差异，则应相对于窗口坐标而非屏幕坐标进行定义，以确保渲染结果不受窗口位置影响。

采用差异化算法时，应基于窗口坐标而非屏幕坐标进行定义，确保渲染结果不受窗口位置影响。

接下来，若启用 SAMPLE ALPHA TO ONE，则每个 alpha 值将替换为可表示的最大 alpha 值。否则，alpha 值保持不变。最后，若启用 SAMPLE COVERAGE，则将片段覆盖率与另一个临时覆盖率进行与运算。 该临时覆盖率的生成方式与前述相同，但作为SAMPLE COVERAGE VALUE值的函数进行计算。函数形式不必完全一致，但必须具备相同的比例性和不变性特性。 若SAMPLE COVERAGE INVERT为TRUE，则临时覆盖率将取反（所有位值被反转）后，再与片段覆盖率进行与运算。

通过调用 通过调用

```
void SampleCoverage( clampf value, boolean invert );
```

将*value*设置为所需覆盖值，并将*invert*设为TRUE或FALSE。*value*在存储为SAMPLE COVERAGE VALUE前会被限制在[0,1]范围内。 通过调用 **GetFloatv** 并设置 *pname* 为 SAMPLE COVERAGE VALUE 可查询 SAMPLE COVERAGE VALUE。通过调用 **GetBooleanv** 并设置 *pname* 为 SAMPLE COVERAGE INVERT 可查询 SAMPLE COVERAGE INVERT。

4.1.4 Alpha测试

此步骤仅适用于RGBA模式。在颜色索引模式下，请跳至下一操作。透明度测试会根据输入片段的透明度值与常量值的比较结果，有条件地丢弃片段。 若片段着色器写入多组颜色，则以片段颜色零的Alpha值作为Alpha测试判定依据。该比较功能可通过通用**启用/禁用**指令配合符号常量ALPHA TEST进行控制。禁用时，相当于始终通过比较。
测试通过以下函数控制：

```
void AlphaFunc( enum func, clampf ref );
```

func 是表示透明度测试函数的符号常量；*ref* 是参考值。*ref* 值会被限制在 [0, 1] 区间内，随后根据第 2.14.9 节中 A 组件的规则转换为定点数值。为进行透明度测试，片段的透明度值也会四舍五入至最接近的整数。 指定测试函数的可能常量包括：NEVER（失败）、ALWAYS（通过）、LESS（小于）、LEQUAL（小于等于）、EQUAL（等于）、GEQUAL（大于等于）、GREATER（大于）或NOTEQUAL（忽略），分别表示通过或失败的片段。

永远，总是，如果片段的alpha值分别小于、小于等于、等于、大于等于、大于或不等于参考值。

所需状态包含浮点参考值、指示比较函数的八位整数，以及控制比较启用的位标记。初始状态设定为参考值0且函数为ALWAYS。默认情况下透明度测试处于禁用状态。

4.1.5 模板测试

模板测试会根据模板缓冲区中位置(x_w, y_w)处的值与参考值的比较结果，有条件地丢弃片段。该测试通过**启用**和禁用命令启用或禁用，使用符号常量STENCIL TEST。禁用时，模板测试及相关修改均不执行，片段始终通过。

模板测试通过以下函数控制：

```
void StencilFunc( enum func, int ref, uint mask ); void StencilFuncSeparate( enum face, enum func, int ref,
    uint mask );
void StencilOp( enum sfail, enum dpfail, enum dppass ); void StencilOpSeparate( enum face, enum sfail, enum dpfail,
    enum dppass );
```

存在两组与模板相关的状态：前模板状态组和后模板状态组。当处理来自非多边形基元（点、线、位图、图像矩形）以及正面多边形基元的光栅化片段时，模板测试和写入操作使用前模板状态集；而处理来自背面多边形基元的光栅化片段时，则使用后模板状态集。在模板测试中，即使因当前多边形模式需以点或线形式光栅化，该原始图形仍被视为多边形。多边形的前向/后向判定方式与双面光照及面剔除相同（参见第2.14.1节和第3.5.1节）。

StencilFuncSeparate 和 **StencilOpSeparate** 接受一个*面部*参数，该参数可以是 FRONT、BACK 或 FRONT AND BACK，用于指示受影响的状态集。**StencilFunc** 和 **StencilOp** 将正面和背面的模板状态设置为相同的值。

StencilFunc 和 **StencilFuncSeparate** 接受三个参数来控制模板测试的通过或失败。*ref* 是用于无符号模板比较的整数参考值，其取值范围被限制在 $[0, 2^s - 1]$ 之间，其中 s 是模板缓冲区的位数。该值的 s 个最低有效位

*掩码*分别与参考值和存储的模板值进行位与运算，所得的掩码值即参与由*func*控制的比较操作。func是决定模板比较函数的符号常量；八个符号常量分别为NEVER（从不）、ALWAYS（始终）、LESS（小于）、LEQUAL（小于等于）、EQUAL（等于）、GEQUAL（大于等于）、GREATER（大于）或NOTEQUAL（不等于）。 因此，当掩码后的参考值小于、小于等于、等于、大于等于、大于或不等于模板缓冲区中掩码后的存储值时，模板测试将分别通过NEVER、ALWAYS、LESS、LEQUAL、EQUAL、GEQUAL、GREATER或NOTEQUAL。

StencilOp 和 **StencilOpSeparate** 接受三个参数，用于指定当前或后续特定测试失败或通过时存储的模板值将如何处理。*sfail* 参数决定模板测试失败时的操作。其符号常量包括：KEEP（保持）、ZERO（清零）、REPLACE（替换）、INCR（饱和增量）、DECR（饱和递减）、INVERT（饱和反转）、INCR WRAP（饱和增量循环）和 DECR WRAP（饱和递减循环）。 – –
这些分别对应：保留当前值、设为零、替换为参考值、饱和递增、饱和递减、位反转、无饱和递增、无饱和递减。

在增量与减量操作中，模板位被视为无符号整数。饱和增减会将模板值限制在0与最大可表示值之间。非饱和增减则采用循环递增机制：当最大可表示值被增量时结果为0，当0被减量时结果为最大可表示值。

当深度缓冲区测试失败（参见第4.1.6节）时（*dpfail*），或测试通过时（*dppass*），均采用相同的符号值表示模板操作状态。

若模板测试失败，则丢弃传入的片段。所需状态包含传递给 **StencilFunc** 或 **StencilFuncSeparate** 以及 **StencilOp** 或 **StencilOpSeparate** 的最新值，以及一个指示模板测试是否启用的位。 初始状态下，模板功能处于禁用状态，前后模板参考值均为零，前后模板比较函数均为ALWAYS，前后模板掩码均为全1。初始时，所有三组前后模板操作均为KEEP。

若无模板缓冲区，则无法进行模板修改，此时无论调用何种**模板函数**，模板测试均视为通过。

4.1.6 深度缓冲区测试

深度缓冲区测试会在深度比较失败时丢弃传入的片段。该比较功能可通过通用**启用**和**禁**用命令配合符号常量DEPTH TEST启用或禁用。禁用时，深度比较及后续可能的深度缓冲区值更新操作将被跳过，片段直接传递至后续处理。然而模板值将 –

如下方所示进行修改，如同深度缓冲区测试通过。若启用，则执行比较操作，深度缓冲区和模板值随后可能被修改。

比较操作通过

```
void DepthFunc( enum func );
```

此命令接受单个符号常量：NEVER、ALWAYS、LESS、LEQUAL、EQUAL、GREATER、GEQUAL、NOTEQUAL 中的任意一个。 因此，深度缓冲区测试通过的条件为：当输入片段的 z_w 值小于、小于等于、等于、大于、大于等于或不等于由输入片段 (x_w, y_w) 坐标指定位置存储的深度值时，测试结果分别为 never、always。

若深度缓冲区测试失败，则丢弃该光片。根据当前深度缓冲区测试失败时生效的函数，更新光片坐标(x_w, y_w)处的模板值。 否则，片段继续执行后续操作，并将深度缓冲区在片段坐标(x_w, y_w)处的值设为该片段*的*z_w值。此时模板值将根据当前深度缓冲区测试成功的有效函数进行更新。

所需状态包含一个八值整数及单比特位，用于指示深度缓冲功能的启用状态。初始状态下函数为LESS且测试功能禁用。

如果没有深度缓冲区，则相当于深度缓冲区测试始终通过。

4.1.7 遮挡查询

遮挡查询可用于追踪通过深度测试的片段或采样数量。

遮挡查询与查询对象相关联。

通过调用以下函数可启动和结束遮挡查询：

```
void BeginQuery( enum target, uint id ); void EndQuery(
enum target );
```

其中*target*为SAMPLES PASSED。若**BeginQuery**调用时使用未分配的*id*，该名称将被标记为已用并关联至新查询对象。

以*目标*值为SAMPLES PASSED的**BeginQuery操作**将当前通过样本计数重置为零，并将查询活动状态设为TRUE，同时将活动查询ID设为*id*。 以SAMPLES PASSED为目标调用**End**Query时，将当前通过样本计数初始化复制到活动遮挡查询对象的结果值中，将活动遮挡查询对象的结果可用状态设为FALSE，将查询活动状态设为FALSE，并将活动查询ID设为0。

若在存在相同*目标*的其他查询运行期间，或当*id*为当前运行中查询名称时调用**Begin**Query且*id*为零，则触发INVALID OPERATION错误。

若在无相同*目标*的查询运行时调用**EndQuery**，则会触发INVALID OPERATION错误。

当遮挡查询处于活动状态时，每个通过深度测试的片段都会使通过样本计数增加特定数量。若SAMPLE BUFFERS的值为0，则每个片段使通过样本计数增加1；若SAMPLE BUFFERS的值为1，则通过样本计数增加的数量等于覆盖位被设置的样本数量。 但实现方可酌情选择：若片段内存在任何被覆盖的样本，则允许直接按SAMPLES值增加已通过样本计数。

若样本通过计数发生溢出（即超过$2^n - 1$的值，*其中n*（其中 samples-passed count 为样本计数中的位数），其值将变得未定义。建议（但非强制要求）实现方案通过在 2^n 处饱和处理此溢出情况： 1 并停止进一步递增。

命令

```
void GenQueries( sizei n, uint *ids );
```

将返回*n*个先前未使用的查询对象名称*至ids*数组。这些名称会被标记为已使用，但在首次通过**BeginQuery**调用前不会关联实际对象。查询对象包含单一状态字段——整型结果值。该值在对象创建时初始化为零。除保留给GL的零值外，任何正整数均可作为有效查询对象名称。

查询对象通过调用

```
void DeleteQueries( sizei n, const uint *ids );
```

ids 包含待删除的 *n* 个查询对象名称。删除查询对象后，其名称将重新变为未被使用。*ids* 中未被使用的名称将被静默忽略。

当任何目标的查询处于活动状态时调用 **GenQueries** 或 **DeleteQueries** 都会引发 INVALID OPERATION 错误。

必要状态包含：单比特位标识遮挡查询是否活跃、当前活跃遮挡查询的标识符，以及记录已通过样本数量的计数器。

4.1.8 混合

混合操作将*输入源*片段的R、G、B、A值与帧缓冲区中片段位置(x_w, y_w)处存储的目标R、G、B、A值进行组合。

源值与目标值根据*混合方程*进行组合，结合由*混合函数*确定的源值与目标值权重因子四元组，以及常量*混合颜色*，从而获得新的R、G、B和A值，具体过程如下所述。 每个浮点值均被限制在[0, 1]范围内，并按第2.14.9节所述方式转换回定点值。最终生成的四个值将传递至后续操作。

混合操作取决于输入片段的透明度值与当前存储像素对应的透明度值。混合仅在RGBA模式下生效；在颜色索引模式下将被跳过。通过符号常量BLEND配合En**able或Disable**指令启用或禁用混合功能。若混合功能禁用，或启用了颜色值的逻辑运算（参见第4.1.10节），则直接进入下一操作。

如果多个片段颜色被写入多个缓冲区（参见第4.2.1节），则混合运算将针对每个片段颜色及其对应的缓冲区分别计算并应用。

混合方程

混合效果由*混合方程*控制，通过以下命令定义：

```
void BlendEquation( enum mode );
void BlendEquationSeparate( enum modeRGB, enum modeAlpha );
```

BlendEquationSeparate的参数*modeRGB*决定RGB混合函数，*而modeAlpha*决定Alpha混合方程。 的参数*mode*同时决定RGB和Alpha混合方程。modeRGB和*modeAlpha*必须分别取值于FUNC ADD、FUNC SUBTRACT、FUNC REVERSE SUBTRACT、MIN、MAX或LOGIC OP。

目标（帧缓冲区）分量与源（片段）分量均按第2.14.9节（最终颜色处理）所述方案表示为定点值。常量颜色分量则视为浮点值。 – –

在混合操作之前，每个固定点颜色分量都会经历隐式转换为浮点数。该转换必须保持值0和1不变。混合分量被视为以浮点形式执行。

表4.1给出了每种模式下对应的分量级混合方程，无论是在*RGB模式*下作用于RGB分量，还是*在Alpha模式*下作用于Alpha分量。

表中，颜色分量缩写（R、G、B或A）后的s下标表示输入片段的源颜色分量，d下标

模式	RGB 组件	Alpha 成分
FUNC_ADD	$R_c = R_s * S_r + R_d * D_r$ $G_c = G_s * S_g + G_d * D_g$ $B_c = B_s$ 乘以 S_b 加 B_d 乘以 D_b	$A_c = A_s$ 乘以 S_a 加 A_d 乘以 D_a
FUNC_SUBTRACT	$R_c = R_s * S_r - R_d * D_r$ $G_c = G_s * S_g - G_d * D_g$ $B_c = B_s * S_b - B_d * D_b$	$A_c = A_s$ 乘以 S_a 减去 A_d 乘以 D_a
函数反向减法	$R_c = R_d * S_r - R_s * D_r$ $G_c = G_d * S_g - G_s * D_g$ $B_c = B_d * S_b - B_s * D_b$	$A_c = A_d * S_a - A_s * D_a$
MIN	$R_c = \min(R_s, R_d)$ $G_c = \min(G_s, G_d) B_c = \min(B_s, B_d)$	$A_c = \min(A_s, A_d)$
MAX	$R_c = \max(R_s, R_d)$ $G_c = \max(G_s, G_d) B_c = \max(B_s, B_d)$	$A_c = \max(A_s, A_d)$
逻辑运算	$R_c = R_s$ *运算符* R_d $G_c = G_s$ *运算符* G_d $B_c = B_s$ *运算符* B_d	$A_c = A_s$ *OP* A_d

表 4.1： RGB 与 alpha 混合方程。OP 表示由 **LogicOp** 指定的逻辑运算（参见表 4.3；RGB 与 alpha 组件均采用相同的逻辑运算）。

颜色分量缩写中的上标指代对应帧缓冲区位置的目标颜色分量，而颜色分量缩写中的c下标指代常量混合颜色分量。无下标的颜色分量缩写表示混合后产生的新颜色分量。此外，S_r、S_b、*S(a)分别表示由源混合函数确定的源权重因子中的红、绿*、蓝及透明度分量，D(r)、$S(b)$、$S(a)$则表示混合后生成的红、绿、蓝及透明度分量。$_g$、S_b和 S_a 分别是源混合函数确定的源权重因子中的红、绿、蓝和透明度分量，而 D_r、D_g、D_b和 D_a 则是目标混合函数确定的目标权重因子中的红、绿、蓝和透明度分量。混合函数将在下文描述。

混合函数

混合方程使用的权重因子由混合函数决定。混合函数通过以下命令指定：

函数	RGB混合因子 (S_r, S_g, S_b) 或 (D_r, D_g, D_b)	Alpha混合因子 S_a或 D_a
零	(0, 0, 0)	0
ONE	(1, 1, 1)	1
SRC COLOR	(R_s, G_s, B_s)	A_s
源颜色减一	$(1, 1, 1) - (R_s, G_s, B_s)$	$1 - A_s$
DST 颜色	(R_d, G_d, B_d)	A_d
减去 DST 颜色	$(1, 1, 1) - (R_d, G_d, B_d)$	$1 - A_d$
SRC 透明度	(A_s, A_s, A_s)	A_s
一减去SRC阿尔法	$(1, 1, 1) - (A_s, A_s, A_s)$	$1 - A_s$
DST ALPHA	(A_d, A_d, A_d)	A_d
1 减去 DST ALPHA	$(1, 1, 1) - (A_d, A_d, A_d)$	$1 - A_d$
恒定色彩	(R_c, G_c, B_c)	A_c
常量颜色减一	$(1, 1, 1) - (R_c, G_c, B_c)$	$1 - A_c$
常数阿尔法	(A_c, A_c, A_c)	A_c
1 减去常数α	$(1, 1, 1) - (A_c, A_c, A_c)$	$1 - A_c$
SRC ALPHA SATURATE[1]	(f, f, f)[2]	1

表 4.2： RGB 和 ALPHA 源与目标混合函数及对应混合因子。三元组的加减运算按分量进行。

[1] SRC ALPHA SATURATE 仅适用于源RGB和Alpha混合功能。

。

[2] $f = \min(A_s, 1 - A_d)$。

```
void BlendFuncSeparate( enum srcRGB, enum dstRGB, enum srcAlpha, enum dstAlpha );
void BlendFunc( enum src, enum dst );
```

BlendFuncSeparate参数srcRGB和*dstRGB*分别决定源和目的RGB混合函数，而srcAlpha和*dstAlpha*则决定源和目的Alpha混合函数。BlendFunc参数*src*同时决定RGB和Alpha源函数，而*dst*则同时决定RGB和Alpha目的函数。

可能的源与目标混合函数及其对应的计算混合因子汇总于表 4.2。

混合颜色

混合操作中使用的常量颜色C_c通过命令指定

```
void BlendColor( clampf red, clampf green, clampf blue, clampf alpha );
```

四个参数存储前会被限制在[0,1]范围内。该常量颜色可同时用于源和目标混合函数

混合状态

混合所需的状态包括：用于RGB和Alpha混合方程的两个整数，指示源和目标RGB及Alpha混合函数的四个整数，用于存储RGBA常量混合颜色的四个浮点值，以及一个指示混合功能启用或禁用的位。初始RGB和Alpha混合方程均为FUNC_ADD。初始混合函数为：源RGB和Alpha函数设为ONE，目标RGB和Alpha函数设为ZERO。初始常量混合颜色为(R, G, B, A) = (0, 0, 0, 0)。混合功能默认处于禁用状态。

混合操作针对当前启用写入的每个颜色缓冲区（参见第4.2.1节）执行一次，使用每个缓冲区的颜色$_{作为C(d)}$。若某个颜色缓冲区无A*值，则*A_d取值为1。

4.1.9 抖动

抖动是在两个颜色值或索引值之间进行选择。在RGBA模式下，将任意颜色分量的值视为一个固定小数点值，其二进制小数点左侧有m位，*其中*m是帧缓冲区中分配给该分量的位数；将每个这样的值*称为*c。对于*每个*c，抖动会选择一个值

c_1 使得 $c_1 \in \{\max\{0, \lfloor c \rfloor - 1\}, \lfloor c \rfloor\}$（选定后，将 c_1 视为 [0,1] 区间内具有 m 位精度的固定点值）。该选定可能取决于像素的 x_w和 y_w 坐标。在颜色索引模式下，同样规则适用，但 c 变为单一颜色索引值。c 的值不得*超过帧缓冲区中该颜色组件或索引所能表示的最大值。*

坐标。在颜色索引模式下，同样规则适用，*此时*c为单一颜色索引值。*c*值不得超过帧缓冲区中对应分量或索引的最大可表示值。

可采用多种抖动算法，但任何算法生成的抖动值仅取决于输入值及*片段的*x、y窗口坐标。若禁用抖动，则每个颜色分量将截断为固定小数点值，其位数与帧缓冲区中对应分量的位数相同；颜色索引值将舍入至帧缓冲区颜色索引部分可表示的最近整数。

启用抖动需使用符号常量DITHER的**Enable**指令，禁用则使用**Disable**指令。因此所需状态仅需单比特。初始状态为抖动启用。

4.1.10 逻辑运算

最后，对传入片段的颜色值或索引值与帧缓冲区中对应位置存储的颜色值或索引值进行逻辑运算。运算结果将替换帧缓冲区中该片段坐标 (x_w, y_w) 处的值。 颜色索引的逻辑运算可通过符号常量INDEX LOGIC OP启用或禁用（**启用/禁用**）。（为兼容GL 1.0版本，亦可使用符号常量LOGIC OP。） 颜色值的逻辑运算通过**启用**或**禁用**符号常量COLOR LOGIC OP进行控制。若启用颜色值逻辑运算，则无论BLEND值为何，效果均等同于禁用混合。当多个片段颜色写入多个缓冲区时（参见第4.2.1节），逻辑运算将针对每个片段颜色及其对应缓冲区分别计算并应用。

逻辑运算符通过

```
void LogicOp( enum op );
```

op 为符号常量；可能的常量及其对应操作详见表 4.3。表中 *s* 表示输入片段值，*d* 表示帧缓冲区存储值。符号常量的数值与 X 窗口系统中对应符号值的数值相同。

逻辑运算对每个选定写入的颜色索引缓冲区独立执行，或对每个选定写入的颜色缓冲区中的红、绿、蓝及Alpha通道值分别独立执行。所需状态包含：表示逻辑运算的整数值，以及两个位元用于标识该运算是否启用。初始状态默认采用COPY逻辑运算且处于禁用状态。

4.1.11 附加的多采样片段操作

若绘制缓冲区模式为NONE，则不会对任何多采样或颜色缓冲区进行修改。否则，片段处理过程如下所述。

若启用MULTISAMPLE且SAMPLE BUFFERS值为1，则对每个像素采样执行透明度测试、模板测试、深度测试、混合及抖动操作，而非仅对每个片段执行一次。透明度、

参数值	操作
清除与	0
并反转复制	$s \wedge d$ $s \wedge \neg d$ s
与反转空操作	$\neg s \wedge d$ d
异或 或	s xor d $s \vee d$
且非 等效	$\neg(s \vee d)$
反转_	$\neg(s$ xor $d)$
或反相复制反相或反相与非	$\neg d$
设置 –	$s \vee \neg d$
	$\neg s$
	$\neg s \vee d$
	$\neg(s \wedge d)$
	全为1

表 4.3： **LogicOp** 的参数及其对应的运算。

模板测试或深度测试的结果将导致该采样点的处理终止，而非直接丢弃该片段。所有操作均作用于多采样缓冲区（将在后续章节中描述）中存储的颜色、深度和模板值。此时颜色缓冲区的内容不会被修改。

仅当像素样本的片段覆盖位值为1时，才会对其执行模板、深度、混合及抖动操作。若对应覆盖位为0，则该样本不执行任何操作。

若禁用 `MULTISAMPLE` 且 `SAMPLE BUFFERS` 值为 1，则片段可完全按上述方式处理，由于片段覆盖率必须设置为全覆盖，因此可能进行优化。但允许进一步优化：实现方案可选择识别最中心采样点，仅对该采样点执行透明度、模板和深度测试。 无论模板测试结果如何，所有多采样缓冲区的模板采样值均设置为相应的新模板值。若深度测试通过，则所有多采样缓冲区的深度采样值均设置为片段中心采样点的深度值，所有颜色采样值均设置为传入片段的颜色值。否则，多采样缓冲区的颜色值与深度值均保持不变。

在多采样缓冲区上完成所有操作后，多采样缓冲区中每种颜色的采样值会被组合生成单一颜色值，该值随后会被写入由DrawBuffer或DrawBuffers函数选定的对应颜色缓冲区中。

值被组合为单一颜色值，该值将写入由**DrawBuffer**或**DrawBuffers**选定的对应颜色缓冲区。 实现可延迟颜色缓冲区的写入操作，但帧缓冲区状态必须表现得如同每个片段处理时颜色缓冲区均已更新。组合方法未作规定，但建议对每个颜色分量独立计算简单平均值。

4.2 全帧缓冲区操作

前文描述了将单个片段发送到帧缓冲区时发生的操作。本节将介绍控制或影响整个帧缓冲区的操作。

4.2.1 选择写入缓冲区

首要操作是控制各片段颜色写入的目标颜色缓冲区。此功能可通过**DrawBuffer**或**DrawBuffers命令**实现。

命令

```
void DrawBuffer( enum buf );
```

定义了用于写入零片段颜色的颜色缓冲区集合。*buf* 是指定写入零、一、二或四个缓冲区的符号常量。这些常量包括 NONE、FRONT LEFT、FRONT RIGHT、BACK LEFT、BACK RIGHT、FRONT、后左、后右、前右、前后、AUX0至AUXm，其中*m*+1 – – – 表示可用辅助缓冲区数量。

常量指代四个潜在可见缓冲区：*左前*、*右前*、*左后*、*右后*，以及*辅助*缓冲区。除*AUXi*外，省略LEFT或RIGHT的参数同时指代左右缓冲区；除*AUXi*外，省略FRONT或BACK的参数同时指代前后缓冲区。 AUXi仅启用*辅助*缓冲区*i*的绘制功能。每个*AUXi*均*遵循AUXi* = AUX0 + *i*的规则。常量及其对应缓冲区详见表4.4。**若DrawBuffer**函数传入的常量（除NONE外）未指向GL上下文中分配的任何颜色缓冲区，则返回错误INVALID OPERATION。

DrawBuffer 将把非零的片段颜色绘制缓冲区设置为 NONE。

命令

–

符号常量	front 左	前 右	后 左	后 右	辅助 *i*
无					
前左	•				
右前		•			
后左			•		
后右	•	•		•	
前部					
后	•		•	•	
左			•		
右	•	•		•	
前后		•	•	•	
AUXi					•

表 4.4：**DrawBuffer** 的参数及其所指的缓冲区。

```
void DrawBuffers( sizei n, const enum *bufs );
```

定义所有片段颜色写入的绘制缓冲区。*n* 指定 *bufs* 中缓冲区的数量。*bufs* 是指向符号常量数组的指针，该数组指定每个片段颜色写入的缓冲区。常量值可以是 NONE、FRONT LEFT、FRONT RIGHT、BACK LEFT、BACK RIGHT 以及 AUX0 到 AUXm（其中 m + 1 是可用辅助缓冲区的数量）。

AUXm，其中m + 1为可用辅助缓冲区数量。定义的绘制缓冲区– –
按顺序对应各自的片段颜色。超过*n的*片段颜色对应的绘制缓冲区将设置为NONE。

除 NONE 外，同一缓冲区在 *bufs* 指向的数组中不得重复出现。重复指定同一缓冲区将导致 INVALID OPERATION 错误。

如果正在执行固定功能片段着色，**DrawBuffers** 指定一组用于写入片段颜色的绘制缓冲区。

若片段着色器向 gl FragColor 写入数据，**DrawBuffers** 则指定一组绘制缓冲区，其中 gl FragColor 定义的单片段颜色将写入至这些缓冲区。 若片段着色器写入 gl FragData，则 **DrawBuffers** 指定一组绘制缓冲区，其中由 gl FragData 定义的多重片段颜色将分别写入这些缓冲区。若片段着色器既未写入 gl FragColor 也未写入 gl FragData，则着色器执行后片段颜色的值未定义，且每个片段颜色可能不同。

最大绘制缓冲区数量取决于具体实现，且必须至少为1。可通过调用

– –

GetIntegerv函数配合符号常量MAX DRAW BUFFERS进行查询。

常量FRONT、BACK、LEFT、RIGHT以及FRONT AND BACK在传递**给DrawBuffers函数的**bufs数组中无效，将导致INVALID OPERATION错误。此限制源于这些常量本身可能指向多个缓冲区，具体关系如表4.4所示。

若向**DrawBuffers**传递的常量（除NONE外）未指向GL上下文中分配的任何颜色缓冲区，则会引发INVALID OPERATION错误。*若n*大于MAX DRAW BUFFERS的值，则会引发INVALID VALUE错误。

使用DrawBuffer**或DrawBuffers**指定缓冲区后，后续像素颜色值的写入操作将影响所指定的缓冲区。

若将NONE指定为片段颜色的绘制缓冲区，则该片段颜色将不会写入任何缓冲区。

单视图上下文仅包含左缓冲区，而立体视图上下文包含左右两个缓冲区。同样，单缓冲上下文仅包含前缓冲区，双缓冲上下文则包含前后两个缓冲区。上下文类型在GL初始化时选定。

处理颜色缓冲区选择所需的状态是每个支持的片段颜色对应的一个整数。初始状态下，若无后缓冲区，片段颜色为零的绘制缓冲区为FRONT；否则为BACK。片段颜色非零的绘制缓冲区初始状态均为NONE。

4.2.2 缓冲区更新的精细控制

在完成所有片段操作后，通过四条命令对逻辑帧缓冲区的位写入进行掩码控制。命令如下：

```
void IndexMask( uint mask );
void ColorMask(boolean r, boolean g, boolean b, boolean a);
```

控制颜色缓冲区或缓冲区（取决于当前指定用于写入的缓冲区）。*掩码*的最低*有效*n位（*其中*n为颜色索引缓冲区中的位数）指定掩码。当掩码中出现1时，对应的颜色索引缓冲区（或缓冲区）位将被写入；当出现0时，该位则不被写入。 该掩码仅在颜色索引模式下生效。RGBA模式中，ColorMask用于控制R、G、B、A值向颜色缓冲区（或多个缓冲区）的写入。*r*、*g*、*b*、a分别指示R、G、B、A值是否写入（TRUE表示对应值被写入）。 初始状态下，所有位（彩色索引模式）及所有颜色值（RGBA模式）均处于可写入状态。

深度缓冲区可通过以下方式启用或禁用 z_w 值的写入：

```
void DepthMask( boolean mask );
```

若掩码值非零，则深度缓冲区启用写入；否则禁用。初始状态下深度缓冲区默认启用写入。

命令

```
void StencilMask( uint mask );
void StencilMaskSeparate( enum face, uint mask );
```

控制特定位元写入模板平面。

*掩码*的最低有效s位构成整数掩码（s为模板缓冲区位数），与**IndexMask**相同。**Stencil-MaskSeparate**的*face*参数可取值为FRONT、BACK或FRONT AND BACK，用于指示影响前模板掩码状态还是后模板掩码状态。**Stencil**Mask将前模板掩码状态与后模板掩码状态均设置为相同值。

由正面基元生成的片元使用正面掩码，背面基元生成的片元使用背面掩码（参见第4.1.5节）。清除操作在清空模板缓冲区时始终使用正面模板写入掩码。

各类遮罩操作所需状态包含三个整数和一位位元：整数控制颜色索引，整数控制正反面模板值，位元控制深度值。另需四位位元组指示RGBA值中应写入的颜色分量。初始状态下，整数遮罩均为全1，控制深度值及RGBA分量写入的位元亦为全1。

多采样缓冲区更新的精细控制

当SAMPLE BUFFERS的值为1时，**ColorMask**、DepthMask以及**StencilMask**或**StencilMaskSeparate**将控制多采样缓冲区中值的修改。颜色遮罩对颜色缓冲区的修改无效。若完全禁用颜色遮罩，仍需按前述方式合并颜色采样值，并将结果用于替换由**DrawBuffer**启用的缓冲区中的颜色值。

4.2.3 清除缓冲区

GL提供了一种将特定缓冲区内每个像素的部分区域设置为相同值的机制。

```
void Clear( bitfield buf );
```

该值是多个位值的按位或运算结果，用于指示需要清除的缓冲区。这些位值分别为：COLOR BUFFER BIT（颜色缓冲区位）、DEPTH BUFFER BIT（深度缓冲区位）、STENCIL BUFFER BIT（模板缓冲区位）和ACCUM BUFFER BIT（累加缓冲区位），分别表示当前启用的颜色写入缓冲区、深度缓冲区、模板缓冲区和累加缓冲区（详见下文）。每个缓冲区被清除后的值取决于该缓冲区的清除值设置。若掩码并非指定值的位或运算结果，则会触发INVALID VALUE错误。

```
void ClearColor( clampf r, clampf g, clampf b, clampf a );
```

在RGBA模式下为颜色缓冲区设置清除值。每个指定的分量都会被限制在[0, 1]范围内，并根据第2.14.9节的规则转换为定点数。

```
void ClearIndex( float index );
```

设置清除颜色索引。将*index*转换为二进制点左侧精度未指定的定点值，然后用$2^m - 1$掩码该值的整数部分，其中m为帧缓冲区中存储的颜色索引值位数。

```
void ClearDepth( clampd d );
```

接受一个浮点值，该值被限制在[0, 1]范围内，并根据第2.11.1节中给出的窗口*z*值规则转换为定点数。同样地，

```
void ClearStencil( int s );
```

接受单个整数参数，该参数用于将模板缓冲区清零至指定值。
s 值将按模板缓冲区位平面数量进行掩码处理。

```
void ClearAccum( float r, float g, float b, float a );
```

该函数接受四个浮点数参数，依次用于设置累加缓冲区的R、G、B和A值（详见下一节）。这些值在指定时会被限制在[0, 1]范围内。

调用 **Clear** 时，仅会应用以下片段级操作（若已启用）：像素所有权测试、剪裁测试和抖动处理。

操作同样生效。若缓冲区不存在，则针对该缓冲区的**清除操作**无效。

清除所需的状态是为每个颜色缓冲区、深度缓冲区、模板缓冲区和累加缓冲区设置明确的清除值。初始时，RGBA颜色清除值为(0,0,0,0)，清除颜色索引为0，模板缓冲区和累加缓冲区的清除值均为0。深度缓冲区的初始清除值为1.0。

清除多采样缓冲区

当清空一个或多个颜色缓冲区时，多采样缓冲区的颜色采样值将被清除，具体由**清除**掩码位COLOR BUFFER BIT**和**DrawBuffer模式决定。**若**DrawBuffer模式为NONE，则无法清除多采样缓冲区的颜色采样值。

如果**清除**掩码位深度缓冲位或模板缓冲位被设置，则分别清除对应的深度或模板采样。

4.2.4 累加缓冲区

累加缓冲区中每个像素的组成部分包含四个值：分别对应R、G、B和A通道。累加缓冲区仅通过调用

```
void Accum( enum op, float value );
```

（清除操作除外）。*op* 是表示累加缓冲区操作的符号常量，*value* 是该操作中使用的浮点值。可选操作包括累加（ACCUM）、加载（LOAD）、返回（RETURN）、乘法（MULT）和加法（ADD）。

当剪裁测试启用时（参见第4.1.2节），任何**累加**操作仅更新当前剪裁框内的像素；否则将更新窗口内所有像素。 累加缓冲区操作对每个受影响像素均采用相同方式处理，因此我们描述每种操作对单个像素的影响。累加缓冲区值视为[0, 1]范围内的有符号值。使用ACCUM操作可从当前选定的读取缓冲区（参见第4.3.2节）获取R、G、B和A通道值。 每个分量作为[0, 1]范围内的定点值（参见第2.14.9节）转换为浮点数。随后将各结果乘以*value*参数，并将乘积与累加缓冲区中对应颜色分量相加，最终用该合成值替换当前累加缓冲区颜色值。

LOAD操作与ACCUM操作效果相同，但计算出的值会替换对应的累加缓冲区组件，而非累加至其中。

RETURN操作从累加缓冲区中提取每个颜色值，将R、G、B和A各通道与*指定数值*相乘，并将结果限制在[0, 1]范围内。 最终生成的颜色值会被写入当前启用颜色写入的缓冲区，如同栅格化生成的片元数据。但需注意：此时仅会应用以下片元级操作（若已启用）：像素所有权测试、剪裁测试（参见4.1.2节）及抖动处理（参见4.1.9节）。同时也会执行颜色遮罩处理（参见4.2.2节）。

MULT操作将累加缓冲区中的每个R、G、B和A值与*指定数值*相乘，然后将缩放后的颜色分量返回至对应的累加缓冲区位置。ADD操作与MULT相同，区别在于将*指定数值*直接加至每个颜色分量。

仅当操作类型为RETURN时，**Accum**操作的颜色分量才需进行钳位处理。此时，发送至已启用颜色缓冲区的值将首先被钳制在[0, 1]范围内。否则，若颜色分量操作结果超出[0, 1]范围，则结果未定义。

如果没有累加缓冲区，或者图形渲染器处于颜色索引模式，**则累加操作**将触发无效操作错误。

累积缓冲区本身之外无需任何状态设置。

4.3 绘制、读取和复制像素

可通过DrawPixels和ReadPixels命令向帧缓冲区写入或读取像素。**CopyPixels**命令可将像素块从帧缓冲区某区域复制至另一区域。

4.3.1 向模板缓冲区写入

DrawPixels的操作已在第3.6.4节中描述，但当*格式*参数为STENCIL INDEX时除外。此时，所有针对**DrawPixels**描述的操作均会执行，但最终输出的将是带有对应模板索引的窗口坐标(*x, y*)，而非片段。 每个坐标-模板索引对将直接传递至片段级操作，跳过光栅化过程中的纹理、雾效及抗锯齿应用阶段。随后每对数据将作为片段进行像素所有权与剪裁测试；其余片段级操作均被跳过。 最后，每个模板索引将写入其对应的

在帧缓冲区中的位置，受当前前栅模版（通过**StencilMask**或**StencilMaskSeparate**设置）影响。若存在深度分量且**DepthMask**设置非FALSE，则深度分量也将写入帧缓冲区；此时忽略**DepthTest**的设置。

若未存在模板缓冲区，则返回错误INVALID OPERATION。

4.3.2 读取像素

从帧缓冲区读取像素并将其放置在客户端内存中的方法如图4.2所示。我们将按发生顺序描述像素读取过程的各个阶段。

像素读取使用

```
void ReadPixels( int x, int y, sizei width, sizei height, enum format, enum type,
    void *data );
```

ReadPixels函数中x和y之后的参数**与DrawPixels函数**的参数对应。适用于**ReadPixels**及其他图像查询命令（参见第6.1节）的像素存储模式汇总于表4.5。

从帧缓冲区获取像素

若格式为深度分量模式，则值从深度缓冲区获取。若无深度缓冲区，则触发无效操作错误。

如果存在多采样缓冲区（即SAMPLE BUFFERS的值为1），则值将从该缓冲区的深度采样中获取。建议使用最中心采样的深度值，但具体实现可选择在每个像素上对深度采样值进行任意函数运算。

若格式为模板索引（STENCIL INDEX），则值取自模板缓冲区；同样地，若不存在模板缓冲区，则会发生无效操作（INVALID OPERATION）错误。

若存在多采样缓冲区，则值将从该缓冲区中的模板采样中获取。建议使用中心采样的模板值，但具体实现可选择每个像素处模板采样值的任意函数。

对于所有其他格式，取值缓冲区均为颜色缓冲区之一；颜色缓冲区的选择由**ReadBuffer命令**控制。

命令

```
void ReadBuffer( enum src );
```

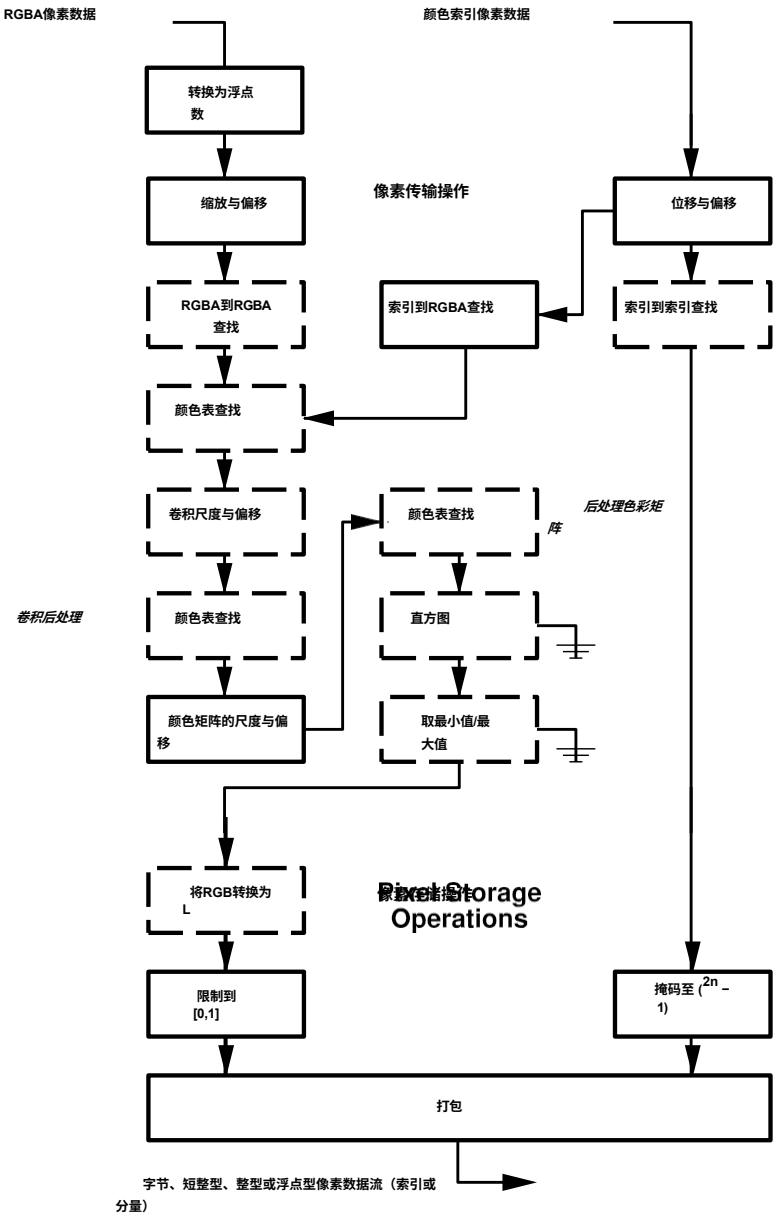


图4.2. **ReadPixels操作**示意图。虚线框内操作可启用或禁用。图示包含RGBA与颜色索引像素路径，深度及模板像素路径未显示。

参数名称	类型	初始值	有效范围
PACK_SWAP_BYTES	布尔值	FALSE	TRUE/FALSE
先打包低位字节	布尔值	FALSE	TRUE/FALSE
PACK_ROW_LENGTH	整数	0	[0, ∞)
跳过行数	整数	0	[0, ∞)
跳过像素	整数	0	[0, ∞)
PACK 对齐	整数	4	1,2,4,8
图像高度	整数	0	[0, ∞)
打包跳过图片	整数	0	[0, ∞)

表 4.5：与 **ReadPixels**、**GetColorTable**、**Get-ConvolutionFilter**、**GetSeparableFilter**、**GetHistogram**、**GetMinmax**、**GetPoly-gonStipple** 和 **GetTexImage** 相关的 **PixelStore** 参数。

接受符号常量作为参数。可能值包括：FRONT LEFT、FRONT RIGHT、BACK LEFT、BACK RIGHT、FRONT、BACK、LEFT、RIGHT 及 AUX0 *通过AUXn*。FRONT和LEFT指代前左缓冲区，BACK指代后左缓冲区，RIGHT指代前右缓冲区。其余常量直接对应其所命名的缓冲区。若请求的缓冲区缺失，则生成INVALID OPERATION错误。若无后缓冲区，则**读取缓冲区的**初始设置为FRONT；否则为BACK。

ReadPixels从选定缓冲区中获取每个像素的值，该像素的左下角坐标为$(x + i, y + j)$，*其中*0 且$i <$ width，0 *且*$j <$ *height*；该像素被称为第j行中的第i个像素。若这些像素中的任何一个超出当前GL上下文分配的窗口范围，则其获取的值为未定义。 对于不属于当前上下文的独立像素，其结果同样未定义。否则，**ReadPixels**将从选定缓冲区获取值，无论这些值如何被写入该缓冲区。

若图形渲染器处于RGBA模式，且*格式*为RED、GREEN、BLUE、ALPHA、RGB、RGBA、BGR、BGRA、LUMINANCE或LUMINANCE ALPHA之一，则在每个像素位置从选定缓冲区获取红色、绿色、蓝色、和透明度值将从选定缓冲区中获取。若帧缓冲区不支持透明度值，则获取的A值为1.0. 若*格式*为COLOR INDEX且图形渲染器处于RGBA模式，则会发生INVALID OPERATION错误。若图形渲染器处于颜色索引模式，且*格式*非DEPTH COMPONENT或STENCIL INDEX，则在每个像素位置获取颜色索引。

RGBA值转换

此步骤仅在图形渲染器处于RGBA模式且*格式*既非STENCIL INDEX也非DEPTH COMPONENT时生效。R、G、B、A值构成元素组。每个元素视为[0,1]区间内的固定点值，位数为m，其中m为选定缓冲区对应颜色组件的位数（参见2.14.9节）。

深度值的转换

此步骤仅在*格式*为深度分量时适用。元素被视为[0,1]区间内的固定点值，其*精度*为m位，其中m为深度缓冲区中的位数（参见第2.11.1节）。

像素传输操作

此步骤实为第3.6.5节单独描述的步骤序列。完成该节所述处理后，将按后续章节所述方式处理各组。

转换为L

此步骤仅适用于RGBA组件组，且仅当格式为
LUMINANCE 或 LUMINANCE ALPHA。L值计算为：

$$L = R + G + B$$

*其中R、G和B*分别是红、绿、蓝通道的数值。计算得到的单一L通道值将替换该组中的R、G和B通道值。

最终转换

对于索引，若*类型*非浮点型，最终转换需按表4.6给定值对索引进行掩码处理；若*类型*为浮点型，则将整数索引转换为GL浮点数据值。

对于RGBA颜色，首先将每个分量限制在[0, 1]范围内，然后应用表4.7中的相应转换公式处理该分量。

客户端内存布局

元素组在内存中的放置方式与从内存中提取用于
即j行第i组元素（对应于

*类型*参数	索引掩码
无符号字节 _	$2^8 - 1$
位图	1
字节	$2^7 - 1$
无符号短整型 _	$2^{16} - 1$
SHORT	$2^{15} - 1$
无符号整型 _	$2^{32} - 1$
整型	$2^{31} - 1$

表 4.6：**ReadPixels** 使用的索引掩码。浮点数据不进行掩码处理。

第 j 行）被放置在内存中，其位置恰好是 **DrawPixels** 函数提取第 j 行第 i 组像素的位置。详见第 3.6.4 节中的**解包说明**。唯一区别在于使用以PACK开头的存储模式参数替代以UNPACK开头的参数。若格式为RED、GREEN、BLUE、ALPHA或LUMINANCE，则仅写入对应的单一元素；若*格式*为LUMINANCE ALPHA、RGB或BGR，则仅写入对应的两个或三个元素；其余情况均写入各组的所有元素。 _

4.3.3 像素复制 _

CopyPixels函数将帧缓冲区中一个区域的矩形像素值复制到另一个区域。像素复制过程如图4.3所示。

```
void CopyPixels( int x, int y, sizei width, sizei height, enum type );
```

type 为符号常量，必须为 COLOR、STENCIL 或 DEPTH 之一，分别表示传输值为颜色值、模板值或深度值。前四个参数的含义与 **ReadPixels** 函数的对应参数相同。

值从帧缓冲区获取，经转换（如适用）后，将执行第3.6.5节所述的像素传输操作，如同调用**ReadPixels**函数并传入对应参数一般。若*类型*为STENCIL或DEPTH，则分别视为**ReadPixels**的*格式*为STENCIL INDEX或DEPTH COMPONENT。若*类型*为COLOR，当GL处于RGBA模式时，其*格式*等同于RGBA；当GL处于颜色索引模式时，其*格式*等同于COLOR INDEX。 _

_

_

*类型*参数	GL 数据类型	组件 转换公式
无符号字节	无符号字节	$c = (2^{\wedge 8} - 1)f$
BYTE	字节	$c = [(2^{8} - 1)f - 1]/2$
无符号短整型	ushort	$c = (2^{16} - 1)f$
SHORT	short	$c = [(2^{16} - 1)f - 1]/2$
无符号整数	uint	$c = (2^{32} - 1)f$
INT	int	$c = [(2^{32} - 1)f - 1]/2$
浮点	浮点	$c = f$
无符号字节 3 3_2	ubyte	$c = (2^{\wedge N} - 1)f$
无符号字节 2 3_3 反转	ubyte	$c = (^{2^{\wedge}N} - 1)f$
无符号短整型 5_6_5	ushort	$c = (2^{N} - 1)f$
无符号短整型 5_6_5 反向	ushort	$c = (^{2^{\wedge}N} - 1)f$
无符号短整型 4_4_4_4	ushort	$c = (^{2^{\wedge}N} - 1)f$
无符号短整型 4_4_4_4 反向	ushort	$c = (2^{\wedge N} - 1)f$
UNSIGNED SHORT 5_5_5_1	ushort	$c = (^{2^{\wedge}N} - 1)f$
无符号短整型 1_5_5_5 反向	ushort	$c = (^{2^{\wedge}N} - 1)f$
无符号整型 8_8_8_8	uint	$c = (2^{N} - 1)f$
无符号整型 8_8_8_8 反向	uint	$c = (2^{N} - 1)f$
无符号整型 10_10_10_2	uint	$c = (2^{\wedge N} - 1)f$
无符号整数 2_10_10_10 反转	uint	$c = (2^{\wedge N} - 1)f$

表 4.7：反向分量转换，用于将分量数据返回至客户端内存时。颜色、法线和深度分量将通过指定方程，从内部浮点表示形式（*f*）转换为指定 GL 数据类型（*c*）的数值。 所有运算均采用内部浮点格式执行。这些转换适用于GL查询命令返回的组件数据，以及返回至客户端内存的像素数据组件。即使GL数据类型的实现范围大于最低要求范围，方程仍保持不变（参见表2.2）。以N为指数的方程需对打包数据类型的每个位域分别执行，*其中*N取值为该位域的位数。

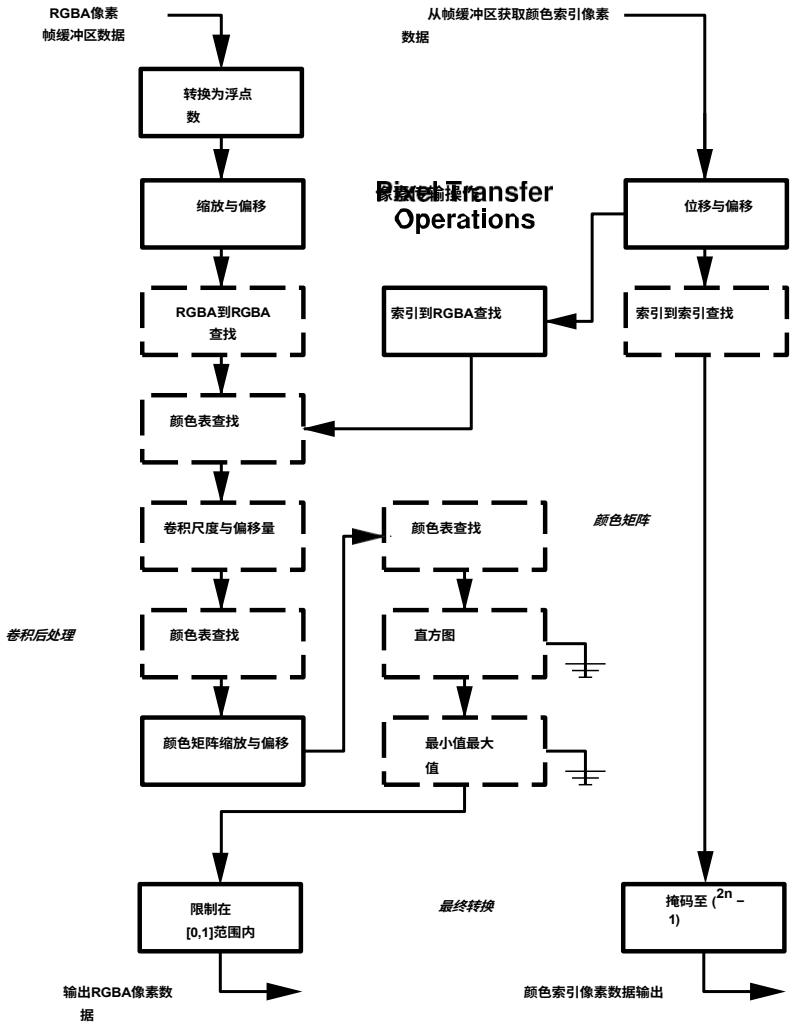


图4.3. **CopyPixels**操作示意图。虚线框内操作可启用或禁用。当前不执行索引到RGBA的查找操作。图中显示RGBA和颜色索引像素路径；深度和模板像素路径未显示。

由此获得的元素组随后被写入帧缓冲区，如同**向DrawPixels**函数传入了宽度和*高度参数*一般，操作始于元素的最终转换。其有效*格式*与先前描述的格式相同。

4.3.4 像素绘制/读取状态

像素操作所需的状态由**PixelStore、PixelTransfer和PixelMap设置**的参数构成。该状态已在表3.1、3.2和3.3中汇总说明。同时需提供整型变量**ReadBuffer的**当前设置值及当前光栅化位置（参见第2.13节）。通过**PixelStore**设置的状态属于GL客户端状态。

第五章

特殊函数

本章介绍了其他GL功能，这些功能难以归入前几章的任何一类。 这些功能包括：评估器（用于建模曲线和曲面）、选择器（用于定位渲染后的原始图形元素）、反馈机制（在光栅化前返回GL结果）、显示列表（用于指定GL命令组以便后续执行）、刷新与终止（用于同步GL命令流）以及提示信息。

5.1 评估器

评估器提供了一种使用多项式或有理多项式映射生成顶点、法线、纹理坐标及颜色的手段。生成的值会被传递至GL的后续阶段，如同直接由客户端提供一般。变换、光照、基元组合、光栅化及像素级操作均不受评估器使用的影响。

考虑由以下公式定义的R^k值多项式$\mathbf{p}(u)$：

$$\mathbf{p}(u) = \sum_{i=0}^{n} B^n_i(u)\mathbf{R}_i \tag{5.1}$$

其中 $\mathbf{R}_i \in R^k$ 且

$$B^n_i(u) = \binom{n}{i} u^i (1-u)^{n-i},$$

第 i 个伯恩斯坦多项式，次数为 n（注意 $0^0 \equiv 1$ 且 $\binom{n}{0} \equiv 1$）。每个 $\mathbf{R}_i$ *均为控制点*。相关命令为

```
void Map1fd( enum target, T u1, T u2, int stride, int order, T points
    );                  { }
```

目标	*k*	值
MAP1 顶点 3	3	*x*、*y*、z顶点坐标
MAP1 顶点 4	4	*x*, *y*, *z*, *w* 顶点坐标
MAP1索引	1	颜色索引
MAP1 颜色 4	4	R、G、B、A
MAP1 正常	3	*x*, *y*, *z* 法线坐标
MAP1 纹理坐标 1	1	*s* 纹理坐标
MAP1 纹理坐标 2	2	*s*, *t* 纹理坐标
MAP1 纹理坐标 3	3	*s*, *t*, *r* 纹理坐标
MAP1 纹理坐标 4	4	*s*, *t*, *r*, *q* 纹理坐标

表 5.1： *目标*为 **Map1** 指定的值。值按其获取顺序给出。

target 是表示定义多项式范围的符号常量。其可能取值及对应的评估次数详见表 5.1。*order* 等于 *n* + 1；若 *order* 小于 1 或大于 MAX EVAL ORDER，则生成 INVALID VALUE 错误。*points* 是指向 *n* + 1 个存储区块集合的指针。 每个存储区块分别以*k*个单精度浮点数或双精度浮点数开头，其余空间可填充任意数据。表5.1说明了*k*值如何随*目标值*变化，以及每种情况下k值的具体含义。

步长是每个存储块中单精度或双精度值（视情况而定）的数量。*若步长*小于*k*时将引发错误INVALID VALUE。多项式*阶数order*同时代表包含控制点的存储区块数量。包含控制点的存储区块数。

u_1和 u_2 提供两个浮点数值，定义映射原像的端点。当给定值 u' 进行计算时，使用的公式为

$$\mathbf{p}'(u') = \mathbf{p}\left(\frac{u' - u_1}{u_2 - u_1}\right).$$

当 $u_1 = u_2$ 时，将出现错误 INVALID VALUE。

Map2 与 **Map1** 类似，但描述的是形式

$$\mathbf{p}(u, v) = \sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(u) B_j^m(v) \mathbf{R}_{ij}.$$

Map2命令的形式为

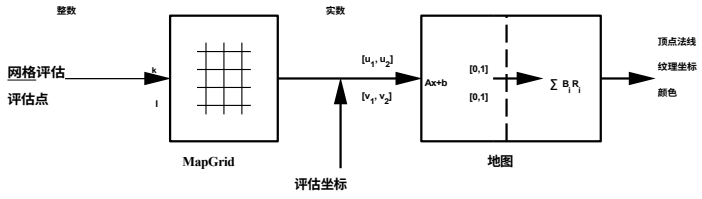


图5.1. 映射评估。

```
void Map2{fd}( enum target, T u1 , T u2 , int ustride,
    int uorder, T v1 , T v2 , int vstride, int vorder, T points );
```

*目标*类型为范围类型，选自**与Map1**相同的组，但字符串MAP1替换为MAP2。*points*指向$(n+1)(m+1)$个存储区块($uorder = n+1$ 且 $vorder = m+1$；若 *uorder* 或 *vorder* 小于 1 或大于 MAX EVAL ORDER，则触发 INVALID VALUE 错误)。构成 $\mathbf{R}_{ij}$的值位于

$$(u\text{步长})i + (v\text{步长})j$$

值（根据需要为单精度或双精度浮点数）位于由*points*指向的首个值之后。u_1、u_2、v_1和v_2定义了映射前的矩形区域；域点(u、v)*的计算结果为:*

；域点 (u', v') 的计算结果为

$$\mathbf{p}'(u', v') = \mathbf{p}\left(\frac{u' - u_1}{u_2 - u_1}, \frac{v' - v_1}{v_2 - v_1}\right).$$

通过上述描述中对应于该贴图的常量，使用**Enable**和**Disable**启用或禁用定义贴图的评估。 评估器映射仅为纹理单元TEXTURE0生成坐标。若ustride或*vstride小于k*，或u_1等于u2，或v_1等于v_2，则返回错误INVALID VALUE。若ACTIVE TEXTURE值非TEXTURE0，**调用Map 12**将引发 错误INVALID OPERATION。

图5.1示意性描述了贴图评估过程；启用贴图的评估可通过两种方式实现。第一种方式是使用

```
void EvalCoord{12}{fd}( T arg ); void
EvalCoord{12}{fd}v( T arg );
```

EvalCoord1 用于评估已启用的单维映射。其参数是表示域坐标 *u*ˆ的值（或指向该值的指针）。**Eval-Coord2** 用于评估已启用的二维映射。两个参数依次指定域坐标 *u*ˆ和 *v*ˆ。

当发出任一 **EvalCoord** 命令时，所有当前启用的映射
指定维度的评估值将被计算。随后，对于每个启用的贴图，系统会以计算出的坐标执行对应的GL命令，但存在一个关键差异：当进行评估时，GL会使用评估值而非当前值（若评估未启用则使用当前值）。有效命令的执行顺序无关紧要，但顶点坐标评估必须最后执行。 评估器的使用不会影响当前颜色、法线或纹理坐标。若启用ColorMaterial，评估的颜色值将影响光照方程结果，如同修改了当前颜色，但不会改变追踪光照参数或当前颜色值。

若对应贴图（所指维度）未启用，则任何指令均无法生效。当特定维度启用多个评估项时（例如`MAP1 TEXTURE COORD 1`与`MAP1 TEXTURE COORD 2`），仅采用坐标数最多的贴图评估结果。

最后，若启用了`MAP2 VERTEX 3`或`MAP2 VERTEX 4`，则计算表面法线。解析计算（有时会产生长度为零的法线）是一种可选方法。若启用了自动法线生成，则将此计算出的法线作为生成顶点的关联法线。 自动法线生成通过符号常量`AUTO NORMAL`的**启用/禁**用进行控制。若自动生成禁用，则启用的法线贴图将用于生成法线。若两者均未启用，则评估生成的顶点不携带法线（实际效果为沿用当前法线）。

对于图映射顶点3，**令q** = **p**。对于图映射顶点4，**令q** = (*x*/*w*, *y*/*w*, *z*/*w*)，其中(*x*, *y*, *z*, *w*) = **p**。然后令

$$\mathbf{m} = \frac{\partial \mathbf{q}}{\partial u} \times \frac{\partial \mathbf{q}}{\partial v}.$$

然后生成的解析法线 **n** 由以下公式给出：**n** = **m**（如果顶点着色器处于活动状态），否则由 **n** = **m**/‖**m**‖**给出**。

第二种评估方式是使用一组命令来实现——
vide用于高效指定待映射的一系列等间隔值。该方法分为两个步骤。第一步是在定义域中定义网格。

使用以下方法实现：

```
void MapGrid1{fd}( int n, T u'1 , T u'2 );
```

用于一维映射，或

```
void MapGrid2{fd}( int n_u , T u'1 , T u'2 , int n_v , T v'1 ,
      T v'2 );
```

对于二维地图而言。在**MapGrid1**的情况下，*u'1和u'2*描述了区间，而n描述该区间的划分数量。 当n ≤ 0时，将返回错误

当 *n* ≤ 0 时将导致 INVALID VALUE 错误。对于 **MapGrid2**，(*u'1* , *v'1*) 指定一个两维点，(*u'2* , *v'2*) 指定另一个点。n_u 给出划分数量 *u'1* 与 *u'2* 之间，n_v 给出 *v'1* 与 *v'2* 之间的划分数。若 n_u ≤ 0 或 n_v ≤ 0，则会发生错误 INVALID VALUE。

定义网格后，可通过调用

```
void EvalMesh1( enum mode, int p1 , int p2 );
```

模式参数取值为 POINT 或 LINE。其效果等同于执行以下操作：
代码片段，其中 *Δu'* = (*u'2* − *u'1*)/*n*：

```
Begin (type);
    for i = p1 to p2 step 1.0
        EvalCoord1 (i * Δu' + u'1 );
结束();
```

其中EvalCoord1f或**EvalCoord1d**将根据需要替换为**EvalCoord1**。若*模式*为POINT，则*类型*为POINTS；若*模式*为LINE，则*类型*为LINE STRIP。唯一要求是：当i=0或*i=n*时，根据i * Δ*u*(') + *u(')*计算的值应分别恰好等于u(')或u(')。

*计算结果i * Δu' + u'1* 分别*精确等于u'1或u'2*。

二维地图的对应命令为

```
void EvalMesh2( enum mode, int p1 , int p2 , int q1 , int q2 );
```

*mode*必须为FILL、LINE或POINT。当mode为FILL时，这些命令等效于以下形式（*其中Δu'*= (*u'2*− *u'1*)/*n且Δv'*= (*v'2*− *v'1*)/*m*）：

```
for i = q1 to q2                    1 步 1.0
    开始(QUAD STRIP);
        for j = p1 to p2 step 1.0 EvalCoord2 (j * Δu'
            + u' EvalCoord2 (j * Δu'    + u'
                                                  1 , i * Δv'    + v'    ); 1
                                                  1 , (i + 1) * Δv'    + v'    ); 1
    结束();
```

若*模式*为LINE，则调用E**valMesh2**等同于

```
for i = q1 to q2 step 1.0
    Begin(LINE STRIP);
    for j = p1 to p2 step 1.0
        EvalCoord2 (j * Δu'                + u'1 ,       i * Δv'       + v'1 );
    结束();;
for i = p1 to p2 step 1.0
    开始(线段);
    for j = q1 to q2 step 1.0
        评估坐标2 (i * Δu'                 + u'1 ,       j * Δv'       + v'1 );
    结束();
```

若*模式*为POINT，则调用**EvalMesh2**等同于

```
Begin(POINTS);
    for i = q1 to q2 step 1.0
        for j = p1 to p2 step 1.0
            EvalCoord2 (j * Δu'    + u'    , i * Δv'1   + v'    );
                                                                  1
结束();
```

同样地，在所有三种情况下，都存在这样的要求：0 *Δu'+ u'= u'，n *Δu'+ u'≠ u'，0 ≤ Δv'+ v'= v'，以及 m * Δv'+ v'= v'。

也可对网格上的单个点进行评估：

```
void EvalPoint1( int p );
```

调用该函数等同于

执行命令

```
EvalCoord1 (p * Δu' + u'    );
```

其中Δu'和u'的定义如上所述。

```
void EvalPoint2( int p, int q );
```

等同于命令

EvalCoord2($p * \Delta u + u_1$, $q * \Delta v + v_1$);

评估器所需的状态可能包含9个一维映射规范和9个二维映射规范，以及每个规范对应的标志位以指示哪些规范处于启用状态。每个映射规范包含一个或两个顺序，一个尺寸适配的控制点数组，以及一组描述域的两个值（针对一维映射）或四个值（针对二维映射）。 *u*或*v*轴的最大可能阶数取决于具体实现（u和*v*共享同一最大值），但必须至少为8。每个控制点包含1至4个浮点数（取决于贴图类型）。初始时所有贴图阶数均为1（即常量贴图）。 所有顶点坐标贴图均生成坐标(0, 0, 0, 1)（或其子集）；所有法线坐标贴图生成(0, 0, 1)；RGBA贴图生成(1, 1, 1, 1)；颜色索引贴图生成1.0；纹理坐标贴图生成(0, 0, 0, 1)。 初始状态下所有映射均处于禁用状态。二维映射需通过标志位指示是否启用自动法线生成功能，初始状态下该功能默认禁用。此外需提供：一维网格规格需两个浮点数及整数网格划分数；二维网格规格需四个浮点数及两个整数网格划分数。 初始状态下，一维域区间的边界分别为0和1.0；二维域区间的边界分别为(0, 0)和(1.0, 1.0)。 网格划分数量为：1维为1，2维为两个方向各1。若在未启用顶点映射的维度上执行评估命令，则不会产生任何效果。

5.2 选择

选择用于确定哪些基本图形将绘制到窗口的某个区域中。该区域由当前的模型视图矩阵和透视矩阵定义。选择通过返回一个整数值*名称*数组来实现，该数组代表*名称堆栈*的当前内容。此堆栈通过命令进行控制

```
void InitNames( void ); void PopName(
void ); void PushName( uint name ); void
LoadName( uint name );
```

InitNames 清空（清除）名称堆栈。**PopName** 从名称堆栈顶部弹出一个名称。**PushName** 将*名称*压入名称堆栈。

LoadName将栈顶值替换为*name*。向空栈加载名称将引发INVALID OPERATION错误。从空栈弹出名称将引发STACK UNDERFLOW；向满栈压入名称将引发STACK OVERFLOW。名称栈的最大允许深度取决于具体实现，但必须至少为64。

在选择模式下，第4章所述的帧缓冲区更新操作不会执行。通过以下方式将GL置于选择模式：

```
int RenderMode( enum mode );
```

模式为符号常量：RENDER、SELECT 或 FEEDBACK 之一。RENDER 为默认值，对应前文所述的渲染模式。SELECT 指定选择模式，FEEDBACK 指定反馈模式（详见下文）。当图形渲染器未处于选择模式时，使用任何名称堆栈操作命令均无效。

选择模式通过以下命令控制：

```
void SelectBuffer( sizei n, uint *buffer );
```

*缓冲区*是一个指向无符号整数数组（称为选择数组）的指针，该数组可能被填充名称，而*n*是一个整数，表示该数组中可存储的最大值数量。在调用SelectBuffer之前将GL置于选择模式将导致INVALID OPERATION错误，同样，在选择模式下调用**SelectBuffer**也会导致此错误。

在选择模式下，若点、线、多边形**或**RasterPos命令生成的有效坐标与裁剪体积（参见2.12节）相交，则该基本图形（或RasterPos命令）将触发选择*命中*。WindowPos命令始终会产生选择命中，因为其生成的光栅坐标位置永远有效。 对于多边形，若该多边形本应被剔除则不产生命中，但选择判断基于多边形本身，与**PolygonMode**设置无关。在选择模式下，每次执行名称堆栈操作命令或调用RenderMode时，若自上次操作堆栈或调用**RenderMode**以来存在命中，则会将*命中记录*写入选择数组。

命中记录按以下顺序包含下列项目：一个非负整数，表示命中时名称堆栈中的元素数量；最小深度值；最大深度值；以及名称堆栈（按底层元素优先顺序排列）。 最小深度值与最大深度值分别取自自上次命中记录以来，所有与裁剪体相交的基元（裁剪后）顶点在窗口坐标系z轴上的最小值与最大值。

写入。最小值和最大值（均位于[0, 1]区间内）分别乘以2^{32}， 1，并舍入至最接近的无符号整数，所得值写入命中记录。这些值不执行深度偏移运算_(参见第3.5.5节）。

命中记录通过指向选择数组的指针存储于该数组中。进入选择模式时，该指针初始化为数组开头。 每次复制命中记录时，指针会更新为指向名称堆栈顶部元素存储位置之后的数组元素。若将命中记录复制到选择数组会导致总值数量*超过n*，则仅写入数组能容纳的部分记录并设置溢出标志。

通过调用 **RenderMode** 并传入非 `SELECT` 的参数值可退出选择模式。在选择模式下调用时，**RenderMode** 会返回复制到选择数组中的命中记录数，并将 **SelectBuffer** 指针重置为其最后指定的值。 在调用**RenderMode**之前，无法保证值已写入选择数组。若选择数组溢出标志被设置，则**RenderMode**返回1并清除溢出标志。每次调用**RenderMode**时，名称堆栈都会被清空，堆栈指针也会重置。

选择所需的状态包含：选择数组地址及其最大容量、名称堆栈及其关联指针、最小√最大深度值以及若干标志位。其中一个标志位指示当前**渲染**模式值。初始状态下，图形渲染器处于渲染模式。 另一标志用于指示上次名称堆栈操作后是否发生命中。该标志在进入选择模式时及每次名称堆栈操作时重置。最后一个标志用于指示是否将超过最大复制名称数量。该标志在进入选择模式时重置。此标志、选择数组地址及其最大容量均属于GL客户端状态。

5.3 反馈

通过调用**RenderMode**并传入`FEEDBACK`参数，可将GL置于反馈模式。在此模式下，不会执行第4章所述的帧缓冲区更新操作。取而代之的是，本应进行光栅化的基元信息将通过*反馈缓冲区*返回给应用程序。

反馈功能通过以下函数控制：

```
void FeedbackBuffer( sizei n, enum type, float *buffer );
```

buffer 是指向浮点数值数组的指针，反馈信息将写入该数组；*n* 是表示可写入该数组的最大值数量的数字。*type* 是描述每个顶点需反馈信息的符号常量（参见图 5.2）。 若在调用 **FeedbackBuffer** 之前将图形渲染器置于反馈模式，或在反馈模式下调用 **FeedbackBuffer**，则会引发 INVALID OPERATION 错误。

在反馈模式下，每个将被光栅化的原始图形（或位图，或调用**DrawPixels/CopyPixels**操作——前提是光栅化位置有效）都会生成一组数值块，这些数值会被复制到反馈数组中。若此操作会导致条目数量超出上限，则该数值块将被部分写入，以填满数组（若仍有剩余空间）。 GL进入反馈模式后生成的首个值块置于反馈数组开头，后续块依次排列。每个块以标识基元类型的代码开头，随后是描述基元顶点及关联数据的值。 位图和像素矩形也会写入条目。反馈发生在多边形剔除（第3.5.1节）**和**多边形模式解析（第3.5.4节）之后。若GL实现通过分解多边形进行渲染，则在将三边以上多边形拆分为三角形后也可能触发反馈。 反馈返回的*x*、*y*、z坐标均为窗口坐标；若返回w值，则为裁剪坐标。*z*值不执行深度偏移运算（参见3.5.5节）。对于位图和像素矩形，返回坐标为当前光栅化位置的坐标。

返回的纹理坐标和颜色是根据第2.14.8节所述的裁剪操作生成的结果。即使在支持多纹理单元的实现中，也仅返回纹理单元TEXTURE0的坐标。返回的颜色为原色。

GL命令解释的排序规则同样适用于反馈模式。每个命令必须被完整解析，且其对GL状态及反馈缓冲区写入值的影响必须完成后，后续命令方可执行。

通过调用 **RenderMode** 并传入非 FEEDBACK 的参数值可退出反馈模式。在反馈模式下调用 **RenderMode** 时，**该函数**将返回反馈数组中存储的值的数量，并将反馈数组指针重置为 *buffer*。返回值永远不会超过传递给 **FeedbackBuffer** 的最大值数量。

若向反馈缓冲区写入某个值会导致写入的值超过指定的最大值数量，则该值不会被写入，且溢出标志会被设置。此时调用RenderMode会返回1，之后溢出标志会被重置。在反馈模式下，值的完整性无法保证

类型	坐标	颜色	纹理	总值
二维	*x*, *y*	–	–	2
3D	*x*, *y*, *z*	–	–	3
3D 颜色	*x*, *y*, *z*	*k*	–	3 + *k*
– 3D彩色纹理	*x*, *y*, *z*	*k*	4	7 + *k*
– 4D 颜色纹理	*x*, *y*, *z*, *w*	*k*	4	8 + *k*

表5.2： 反馈类型与每个顶点值数的对应关系。k值为1
在颜色索引模式下，RGBA模式下为4。

在调用**RenderMode**之前需写入反馈缓冲区。

图5.2给出了反馈生成的数组语法。每个基本单元由唯一标识值标记，后跟若干顶点。顶点通过反馈*类型*确定的浮点值数量进行反馈。表5.2列出了反馈*缓冲区*与每个顶点返回值数量的对应关系。

命令

```
void PassThrough( float token );
```

可在反馈模式下作为标记使用。*token*将作为原始图形返回，并以其专属标识值标记。反馈机制会保持PassThrough命令相对于原始图形定义的顺序。**该命令**不得出现在Begin**与End**之间，且当图形渲染器未处于反馈模式时无效。

反馈所需的状态包括：指向反馈数组的指针、该数组可容纳的最大值数量以及反馈*类型*。需设置溢出标志以指示是否已写入最大允许数量的反馈值；初始时该标志为清零状态。这些状态变量属于GL客户端状态。反馈机制还依赖与选择相同的模式标志，用于指示GL当前处于反馈模式、选择模式或正常渲染模式。

5.4 显示列表

显示列表本质上是一组存储起来以备后续执行的GL命令及其参数。通过提供唯一标识该列表的编号，可指示GL处理特定显示列表（可能重复处理）。此操作将触发列表内命令的执行，其效果等同于常规命令调用。唯一例外是依赖客户端反馈列表的命令。

反馈列表：

反馈项 反馈列表 反馈项

反馈项：

点

线段 多边形 位图

像素矩形 穿透

点：

点标记顶点

线段：

线段标记 顶点 顶点

线段重置标记 顶点 顶点 多边形：

多边形标记 *n* 多边形规格 多边形规格:

多边形规格 顶点 顶点 顶点 顶点

位图:

位图标记 顶点

像素矩形：

绘制像素令牌顶点 _

复制像素令牌 顶点 穿透令牌：

传递令牌 *f* _

顶点:

二维:

f f

3D：

f f f

3D颜色：

f f f 颜色

3D 颜色纹理： _

f f f 颜色纹理

4D 颜色纹理： _

f f f f 颜色纹理

颜色：

f f f f f

纹理：

f f f f

图5.2： 反馈语法。*f* 为浮点数。n为浮点整数，表示多边形顶点数量。以TOKEN结尾的符号均为符号化浮点常量。"vertex"规则下的标签显示了根据反馈*类型*返回的不同顶点数据。LINE TOKEN与LINE RESET TOKEN完全相同，区别在于后者仅在线段的线点阵重置时返回。

当此类命令被累积到显示列表中时（即发出时而非执行时），当时生效的客户端状态将应用于该命令。仅当命令执行时，服务器状态才会受到影响。 与往常一样，作为命令参数传递的指针在命令发出时被解引用。（**当ArrayElement、DrawArrays、DrawElements或DrawRangeElements**命令被累积到显示列表时，顶点数组指针会被解引用。）

显示列表通过调用

```
void NewList( uint n, enum mode );
```

n 是正整数，用于存储后续的显示列表；*mode* 是控制图形渲染器在创建显示列表时行为的符号常量。若 *mode* 为 COMPILE，则命令在写入显示列表时不会执行；若 *mode* 为 COMPILE AND EXECUTE，则命令在遇到时立即执行，随后写入显示列表。若 $n = 0$，则会触发 INVALID VALUE 错误。

调用 **NewList** 后，所有后续的 GL 命令都会按发出顺序放入显示列表，直到调用

```
void EndList( void );
```

发生后，图形渲染器将恢复到正常的命令执行状态。只有当调用EndList时，指定的显示列表才会实际关联到**NewList**所指示的索引。若在未调用匹配**的NewList**前调用**EndList**，或在**调用EndList**前再次调用**NewList**，则会生成INVALID OPERATION错误。 若调用EndList时因内存不足无法存储指定显示列表，则会触发OUT OF MEMORY错误。此时，版本1.1及以上的GL实现将确保：若显示列表存在先前内容，则不作任何修改；除显示列表模式为COMPILE AND EXECUTE时由GL命令执行的状态变更外，不改变GL状态的其他部分。

显示列表定义完成后，通过调用

```
void CallList( uint n );
```

参数n指定要调用的显示列表索引。这将按顺序执行显示列表中保存的命令，如同未使用显示列表直接发出指令。若n=0，则生成错误INVALID VALUE。

命令

```
void CallLists( sizei n, enum type, void *lists );
```

提供了一种高效执行多个显示列表的方法。*n* 是表示要调用的显示列表数量的整数，*lists* 是指向偏移量数组的指针。每个偏移量根据*列表的*定义构造如下：首先，*类型*参数可取常量BYTE、UNSIGNED BYTE、SHORT、UNSIGNED SHORT、INT、UNSIGNED INT或FLOAT，分别表示*列表*指向的数组类型为字节数组、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。 此时每个偏移量通过将数组元素转换为整数获得（浮点数值将被截断）。此外，*type* 还可取值为 2 BYTES、3 BYTES 或 4 BYTES，表示数组包含 2、3 或 4 个无符号字节序列，此时每个整数偏移量按以下算法构造：

offset ← 0
for *i* = 1 **to** *b*
    offset ← *offset* 向左移位 8 位
    offset ← *offset* + *字节*
    在数组中前进至下一个*字节*

b 取值为 2、3 或 4（由*类型*决定）。若 *n* = **0**，**CallLists** 不执行任何操作。

依次取 *n* 个构造偏移量，将其加至显示列表基址以获取显示列表编号。针对每个编号，执行对应显示列表。基址通过调用

```
void ListBase( uint base ); 指定偏移量。
```

指定与任何显示列表都不对应的显示列表索引将不起作用。**CallList** 或 **CallLists** 可能出现在显示列表内部。（若向 **NewList** 传递的*模式*为 COMPILE AND EXECUTE，则执行相应列表，但构建中的列表将包含 **CallList** 或 **CallLists** 本身，而非其构成命令。）为避免显示列表相互调用导致无限递归，在执行过程中会设置实现相关的嵌套层级限制，该限制值至少为 64。

系统提供两条命令用于管理显示列表索引。

```
uint GenLists( sizei s );
```

返回一个整数 *n*，使得索引 *n, ...,* n+s-1 此前未被使用（即从 *n* 开始存在 *s* 个此前未使用的显示列表索引）。**GenLists** 还具有

为每个索引 *n*, ..., *n* + *s* 1 创建一个空显示列表的效果，使得这些索引均被使用。若不存在一组 *s* 个连续的先前未使用的显示列表索引，或当 *s* = 0 时，**GenLists** 返回 0。

布尔值 **IsList**(整数*列表*);

若*列表*属于某个显示列表的索引，则返回 `TRUE`。

可通过调用以下函数删除一组连续的显示列表：

`void` **DeleteLists**(`uint` *list*, `sizei` *range*);

其中 *list* 是待删除首项显示列表的索引，*range* 是待删除的显示列表数量。所有显示列表信息将被清除，对应索引将失效。若索引对应的显示列表不存在，则该索引将被忽略。当 *range* = 0 时，程序不执行任何操作。

某些命令在编译显示列表时被调用时，不会被堆叠到显示列表中，而是立即执行。这些命令分为几类，包括：编译显示列表时调用某些命令时，这些命令不会被编译到显示列表中，而是立即执行。这些命令分为几类，包括

显示列表：**生成列表**和**删除列表**。

渲染模式：**反馈缓冲区**、**选择缓冲区**和**渲染模式**。

顶点数组：**客户端活动纹理**、**颜色指针**、**边缘标志指针**、**雾坐标指针**、**索引指针**、**交错数组**、**法线指针**、**次要颜色指针**、**纹理坐标指针**、**顶点属性指针**和**顶点指针**。

客户端状态：**EnableClientState**、**DisableClientState**、**EnableVertexAttribArray**、**DisableVertexAttribArray**、**PushClientAttrib**、**PopClientAttrib**。

像素与纹理：**PixelStore**、**ReadPixels**、**GenTextures**、**DeleteTextures** 和 **AreTexturesResident**。

遮挡查询：**生成**查询和**删除查询**。

顶点缓冲对象：**生成缓冲区**、**删除缓冲区**、**绑定缓冲区**、**缓冲区数据**、**缓冲区子数据**、**映射**缓冲区和**取消映射缓冲区**。

程序和着色器对象：**CreateProgram**、**CreateShader**、**DeleteProgram**、**DeleteShader**、**AttachShader**、**DetachShader**、**BindAttribLocation**、**CompileShader**、**ShaderSource**、**LinkProgram** 和 **ValidateProgram**。

GL命令流管理：**Finish**和**Flush**。

其他查询：所有名称以 **Get** 和 **Is** 开头的查询命令（详见第 6 章）。

GL命令要求从缓冲区对象获取源数据时，应在显示列表编译时直接引用相关缓冲区对象数据，而非将缓冲区ID和偏移量编码到显示列表中。仅允许立即执行的GL命令（而非编译到显示列表中的命令）将缓冲区对象用作数据接收端。

TexImage3D、**TexImage2D**、**TexImage1D**、Histogram和Col-orTable在调用时会立即执行，使用对应的 代理参数 PROXY TEXTURE 3D; PROXY TEXTURE 2D 或 PROXY TEXTURE CUBE MAP; PROXY TEXTURE 1D ; PROXY HISTOGRAM; 以及 PROXY COLOR TABLE、PROXY_POST CONVOLUTION COLOR TABLE 或 PROXY POST COLOR MATRIX COLOR TABLE。

当程序对象处于使用状态时，可能执行显示列表，其顶点属性调用与该程序对象所含顶点着色器预期的内容不完全匹配。此类不匹配情况的处理方法详见第2.15.3节。

显示列表需要一个状态位来指示图形命令应立即执行还是放入显示列表。初始状态下命令立即执行。若该位指示创建显示列表，则需提供索引以标识当前定义的显示列表。另一个位用于在创建显示列表期间指示命令是否应在编译进入列表时立即执行。当前**列表基准**设置需一个整数，其初始值为零。 最后，需维护状态以标识当前作为显示列表索引使用的整数。初始状态下无索引被使用。

5.5 清空与结束

命令

```
void Flush( void );
```

表示所有先前发送至图形渲染器的命令都必须在有限时间内完成。

命令

```
void Finish( void );
```

强制所有先前GL命令完成执行。**Finish**不会返回，直至先前发出的命令对GL客户端/服务器状态及帧缓冲区产生的所有效果完全实现。

5.6 提示

当存在变量空间时，可通过提示控制GL行为的某些方面。提示使用

目标	提示描述
透视校正提示	参数插值质量
点平滑提示	点采样质量
线平滑提示	线条采样质量
多边形平滑提示	多边形采样质量
雾效提示	雾效质量 （按像素或按顶点计算）
生成MIP贴图提示	质量与性能 自动生成MIP贴图层级
纹理压缩提示	质量与性能 纹理图像压缩
片段着色器导数提示	片段着色器求导的精度 处理内置函数 dFdx、dFdy 和 fwidth

表 5.3： 提示目标与描述。

```
void Hint( enum target, enum hint );
```

target 是表示待控制行为的符号常量，*hint* 是表示期望行为类型的符号常量。可能*的目标*值详见表 5.3；对于*每个目标*，*hint* 必须是以下选项之一：FASTEST 表示应选择最高效的选项；NICEST 表示应选择最高质量的选项；DONT CARE 表示对此不作偏好选择。

对于纹理压缩提示，FASTEST 表示应尽可能快速压缩纹理图像，而 NICEST 表示应以最小图像失真进行压缩。FASTEST 适用于一次性纹理压缩，若需通过 **GetCompressedTextureImage**（第 6.1.4 节）检索压缩结果以供复用，则应使用 NICEST。

提示的具体解释取决于实现方式。某些实现可能完全忽略这些提示。

所有提示的初始值均为 DONT CARE。

第六章

状态与状态请求

描述GL机器所需的状态在第6.2节中列举。大多数状态通过前几章描述的调用设置，并可使用第6.1节描述的调用进行查询。

6.1 查询GL状态

6.1.1 简单查询

多数GL状态可通过符号常量完全标识。这些状态变量的值可通过一组Get命令获取。用于获取简单状态变量的命令包括：

```
void GetBooleanv( enum value, boolean *data );void GetIntegerv( enum
value, int *data );void GetFloatv( enum value, float *data );void
GetDoublev( enum value, double *data );
```

这些命令获取布尔型、整型、浮点型或双精度状态变量。*value* 是表示要返回的状态变量的符号常量。*data* 是指向标量或数组的指针，用于存放返回的数据，其类型与指定类型一致。此外

布尔型 **IsEnabled**(枚举*值*);

可用于判断*值*当前是否启用（与**Enable功能**相同）或禁用。

6.1.2 数据转换

若发出的Get命令返回的值类型与所获取值的类型不同，则会进行类型转换。当调用GetBooleanv时，浮点数或整数值仅在为零时转换为FALSE（否则转换为TRUE）。若调用**GetIntegerv**（或下文任何Get命令），布尔值将被解释为1或0，浮点值将四舍五入至最接近的整数——除非该值属于RGBA颜色分量、**深度**范围值、深度缓冲区清除值或法线坐标。在上述情况下，**Get**命令将根据表4.7中的INT条目将浮点数转换为整数；若值不在[1, 1]范围内，则转换为未定义值。若调用**GetFloatv**，布尔值将被解释为1.0或0.0，整数将强制转换为浮点数，双精度浮点数将转换为单精度。**GetDoublev调用**时执行类似转换。若数值绝对值过大无法用请求类型表示，则返回该类型能表示的最近值。

除非另有说明，多值状态变量返回其多个值的顺序与设置命令中作为参数传递的顺序一致。例如，两个**DepthRange**参数按 *n* 接着 *f* 的顺序返回。同样，评估器映射的点按传递给 **Map1** 时的出现顺序返回。**Map2** 在第 $[(uorder)i + j]$ 个值块中返回 $\mathbf{R}_{ij}$（关于 *i*、*j*、*uorder* 和 $\mathbf{R}_{ij}$的说明见第 228 页）。

通过调用 **GetBooleanv**、**GetIntegerv**、**GetFloatv** 和 **GetDoublev** 时，将 *pname* 设置为 TRANSPOSE_MODELVIEW_MATRIX、TRANSPOSE_PROJECTION_MATRIX、TRANSPOSE_TEXTURE_MATRIX 或 TRANSPOSE_COLOR_MATRIX 之一。

```
GetFloatv(TRANSPOSE_MODELVIEW_MATRIX, m);
```

与命令序列的效果相同

```
GetFloatv(MODELVIEW_MATRIX, m);
m = m^T ;
```

查询转置投影矩阵、转置纹理矩阵和转置颜色矩阵时也会发生类似的转换。

大多数纹理状态变量由ACTIVE_TEXTURE的值限定，以确定查询哪个服务器纹理状态向量。客户端纹理状态变量（如纹理坐标数组指针）由CLIENT_ACTIVE_TEXTURE的值限定。表6.5、6.6、6.9、6.15、6.18、

6.33 指明在状态查询期间由 ACTIVE TEXTURE 或 CLIENT ACTIVE TEXTURE 限定的状态变量。 若纹理状态变量对应于纹理坐标处理单元（即TexGen状态、使能位及矩阵），且ACTIVE TEXTURE值大于或等于MAX TEXTURE COORDS时，查询操作将引发INVALID OPERATION错误。 若活动纹理值大于或等于最大组合纹理图像单元数，则所有其他纹理状态查询均将导致无效操作错误。

6.1.3 枚举查询

其他命令用于获取通过类别（裁剪平面、光源、材质等）及符号常量标识的状态变量。这些命令包括：

```
void GetClipPlane( enum plane, double eqn[4] );void GetLight if v( enum light,
enum value, T data );void GetMaterial if v( enum face, enum value, T data );void
GetTexEnv if v( enum env, enum value, T data );
void GetTexGen(ifd v, enum coord, enum value, T data);void GetTexParameter(if v,
enum target, enum value,
    T data );
void GetTexLevelParameter(enum target, int lod, enum value, T data);
void GetPixelMap ui us f v( enum map, T data );void GetMap ifd v( enum map,
enum value, T data );void GetBufferParameteriv( enum target, enum value,
    T data );
```

GetClipPlane 始终将四个双精度值返回至 *eqn*；这些值是平面在视点坐标系中*平面*方程的系数（这些坐标是在指定平面时计算得出的）。

GetLight 将*光照*（同样为符号常量）的*数值*（符号常量）信息存*入数据中*。POSITION 或 SPOT DIRECTION 返回以视点坐标系表示的数值（这些坐标系是在指定位置或方向时计算得出的）。

GetMaterial、GetTexGen、GetTexEnv、GetTexParameter和GetBuffer-Parameter与GetLight类似，将第一个参数指定的目标的*值*信息写入*数据*。GetMaterial的*face*参数必须为FRONT或BACK，分别表示正面或背面材质。GetTexEnv的*env*参数必须为TEXTURE ENV或

纹理过滤控制。**GetTexGen函数**的*坐标*参数必须为S、T、R或Q之一。对于**GetTexGen**，EYE LINEAR系数以平面指定时计算的眼坐标返回；OBJECT LINEAR系数则以物体坐标返回。

GetTexParameter

参数*目标*可以是TEXTURE 1D、TEXTURE 2D、TEXTURE 3D或TEXTURE CUBE MAP之一，表示当前绑定的单维、二维、三维或立方贴图纹理对象。**GetTex**LevelParameter参数*目标*可以是TEXTURE 1D、TEXTURE 2D、TEXTURE 3D、TEXTURE CUBE MAP POSITIVE X、TEXTURE CUBE MAP NEGATIVE X、TEXTURE CUBE MAP POSITIVE Y、TEXTURE CUBE MAP NEGATIVE Y、TEXTURE CUBE MAP POSITIVE Z、TEXTURE CUBE MAP NEGATIVE Z、PROXY TEXTURE 1D、PROXY TEXTURE 2D、代理纹理3D，或代理纹理立方体贴图，表示一维、二维或三维纹理对象，或构成立方体贴图纹理对象的六个独立2D图像之一，或一维、二维、三维或立方体贴图代理状态向量。 请注意，TEXTURE CUBE MAP 并非 **GetTexLevelParameter** 的有效*目标*参数，因为它未指定特定的立方体贴图面。*value* 是表示要获取哪个纹理参数的符号值。 **对于GetTexParameter**，*value*必须为TEXTURE RESIDENT或表3.19中的符号值之一。**GetTexLevelParameter的***lod*参数决定返回哪个细节等级的状态。若lod小于零或大于最大允许细节等级，则触发INVALID VALUE错误。

针对纹理查询： 纹理 图像 带 未压缩 内部 格式， 查询 的 TEXTURE RED SIZE、TEXTURE GREEN SIZE、TEXTURE BLUE SIZE、TEXTURE ALPHA SIZE、TEXTURE LUMINANCE SIZE、 TEXTURE DEPTH SIZE 的值， 以及 TEXTURE INTENSITY SIZE 返回存储图像数组组件的实际分辨率，而非定义图像数组时指定的分辨率。 对于采用压缩内部格式的纹理图像，返回的分辨率值对应于未压缩内部格式下各组件的分辨率，该格式生成的图像质量大致等同于所指压缩图像。由于实现方案的压缩算法质量可能因数据而异，返回的组件尺寸值仅应视为粗略近似值。

查询 *值* TEXTURE COMPRESSED IMAGE SIZE返回 该压缩纹理图像的大小（单位为微字节），该图像将由 **GetCompressedTexImage** 返回（参见章节 6.1.4）。 不允许对采用未压缩内部格式的纹理图像或代理目标查询 TEXTURE COMPRESSED IMAGE SIZE，尝试操作将导致 INVALID OPERATION 错误。

对 TEXTURE_WIDTH、TEXTURE_HEIGHT、TEXTURE_DEPTH 及 TEXTURE_BORDER 查询返回图像数组创建时指定的宽度、高度、深度和边界。图像数组的内部格式通过 TEXTURE_INTERNAL_FORMAT 查询，或为兼容 GL 1.0 版本而通过 TEXTURE_COMPONENTS 查询。

对于**GetPixelMap**，*映射*必须是表3.3中的映射名称。对于**GetMap**，*映射*必须是第5.1节所述的映射类型之一，且*值*必须是ORDER、COEFF或DOMAIN之一。

6.1.4 纹理查询

命令

```
void GetTexImage( enum tex, int lod, enum format, enum type, void *img
    );
```

用于获取纹理图像。它与其他get命令略有不同；*tex*是一个符号值，用于指示要获取的纹理（或立方体贴图纹理目标名称中的纹理面）。 TEXTURE_1D、TEXTURE_2D 和 TEXTURE_3D 分别表示一维、二维或三维纹理，而 TEXTURE_CUBE_MAP_POSITIVE_X、 TEXTURE_CUBE_MAP_NEGATIVE_X、TEXTURE_CUBE_MAP_POSITIVE_Y、 TEXTURE_CUBE_MAP_NEGATIVE_Y、TEXTURE_CUBE_MAP_POSITIVE_Z 以及 TEXTURE_CUBE_MAP_NEGATIVE_Z 则分别表示立方贴图负X轴、立方贴图正Z轴、立方贴图负Y轴和立方贴图负Z轴。 将立方体贴图纹理的相应面进行分类。*lod*是细节级别数值，*format*是表3.6中的像素格式，*type*是表3.5中的像素类型，*img*是内存块的指针。

GetTexImage从指定细节级别的纹理图像中获取组件组。组件按表6.1分配至R、G、B和A通道，从第一行首个组开始，依次获取每行组件，并按行顺序自首行至末行获取；对于三维纹理，则按图像顺序自首张至末张获取。 这些组随后被压缩并存入客户端内存。该图像不执行像素传输操作，但会应用适用于**读取像素**的像素存储模式。

对于三维纹理，像素存储操作的执行方式与二维图像相同，但需额外应用像素存储状态值 PACK_IMAGE_HEIGHT 和 PACK_SKIP_IMAGES。纹理像素与内存位置的对应关系遵循第 3.8.1 节中 **TexImage3D** 的定义。

行长度、行数、图像深度及图像数量由纹理图像尺寸（含边界）决定。当调用Get-TexImage时，若纹理深度（*lod*）小于零或超出最大允许值，将导致

基础内部格式	R	G	B	A
ALPHA	0	0	0	A_i
亮度（或 1）	L_i	0	0	1
亮度阿尔法（或2）	L_i	0	0	A_i
强度	I_i	0	0	1
RGB（或3）	R_i	G_i	B_i	1
RGBA（或4）	R_i	G_i	B_i	A_i

表 6.1： 纹理、表和滤波器的返回值。R_i、G_i、B_i、A_i、L_i和I_i是内部格式的组件，分别对应像素值 R、G、B 和 A。若请求的像素值在内部格式中不存在，则使用指定的常量值。

错误：无效值。调用 **GetTexImage** 时使用 COLOR INDEX、STENCIL INDEX 或 DEPTH COMPONENT 格式会导致无效枚举错误。

命令

```
void GetCompressedTexImage( enum target, int lod, void *img );
```

用于获取以压缩形式存储的纹理图像。参数 *target*、*lod* 和 *img* 的解释方式与 **GetTexImage** 相同。调用 **GetCompressedTexImage** 时，该函数将 TEXTURE COMPRESSED IMAGE SIZE 个字节的压缩图像数据写入由 *img* 指向的内存区域。压缩图像数据的格式遵循纹理内部格式的定义。 返回压缩纹理图像时，所有像素存储模式和像素传输模式均被忽略。

调用 **GetCompressedTexImage** 时，若 *lod* 值小于零或大于最大允许值，将引发 INVALID VALUE 错误。调用 **GetCompressedTexImage** 时，若纹理图像以未压缩的内部格式存储，将引发 INVALID OPERATION 错误。

命令

```
boolean IsTexture( uint texture );
```

如果*纹理*是纹理对象的名称，则返回 TRUE。*如果纹理*为零，或为非零值但不是纹理对象的名称，或发生错误情况，则 **IsTexture** 返回 FALSE。由 **GenTextures** 返回但尚未绑定的名称不属于纹理对象的名称。

6.1.5 点状查询

命令

```
void GetPolygonStipple( void *pattern );
```

获取多边形点状纹理。该图案按第4.3.2节中**ReadPixels命令的**处理流程打包至内存：其效果等同于向该命令传递*高度*和*宽度*均为32、*类型*为BITMAP、*格式*为COLOR INDEX的参数。

6.1.6 颜色矩阵查询

通过将*pname*设置为相应变量名称调用**GetFloatv**可查询缩放因子和偏移量变量。 调用**GetFloatv**函数时*将*pname设为COLOR MATRIX或TRANSPOSE COLOR MATRIX，即可返回颜色矩阵堆栈顶部的矩阵。通过**GetIntegerv函数**查询颜色矩阵堆栈的当前深度及最大深度，分别*将pname*设为COLOR MATRIX STACK DEPTH和MAX COLOR MATRIX STACK DEPTH。

。

6.1.7 颜色表查询

当前颜色表的内容可通过以下函数查询：

```
void GetColorTable( enum target, enum format, enum type, void *table );
```

*目标*必须是表3.4中列出的*常规*颜色表名称之一。格式和*类型*参数接受与**GetTexImage**对应参数相同的取值。一维颜色表图像将*从表指定的*起始位置返回至客户端内存。 此图像不执行像素传输操作，但会应用**适用于ReadPixels的**像素存储模式。若请求的颜色分量*格式*未包含在颜色查找表的内部格式中，则返回零值。内部颜色分量与*格式*请求分量的映射关系详见表6.1。

相关函数

```
void GetColorTableParameter(enum target, enum pname, T params);
                                                    { }
```

用于整数和浮点查询。

*目标*必须是表3.4中列出的常规或代理颜色表名称之一。pname可选值包括：COLOR TABLE SCALE（颜色表比例）、COLOR TABLE BIAS（颜色表偏移）、COLOR TABLE FORMAT（颜色表格式）、（颜色表宽度）、（颜色表红色尺寸）、COLOR TABLE GREEN SIZE（颜色表绿色尺寸）、COLOR TABLE BLUE SIZE、COLOR TABLE ALPHA SIZE、COLOR TABLE LUMINANCE SIZE、或颜色表强度大小。指定参数的值将返回在*参数中*。

6.1.8 卷积查询

通过命令查询卷积滤波器图像的当前内容

```
void GetConvolutionFilter( enum target, enum format, enum type, void *image );
```

*target*必须为CONVOLUTION 1D或CONVOLUTION 2D。format和*type*接受与**GetTexImage**对应参数相同的取值。一维或二维图像将*从image*起始位置返回至客户端内存。像素处理与组件映射机制与**GetTexImage**完全一致。

使用以下函数查询可分离滤波器图像的当前内容：

```
void GetSeparableFilter( enum target, enum format, enum type, void *row, void
    *column, void *span );
```

*target*必须为SEPARABLE 2D。format和*type*接受与**GetTexImage**对应参数相同的取值。行和列图像分别从*行*和*列*开始返回至客户端内存。*span*当前未使用。像素处理和组件映射与**GetTexImage**完全一致。

函数

```
void GetConvolutionParameter(enum target, enum pname, T params);
{ }
```

用于整数和浮点查询。*目标*必须为 CONVOLUTION 1D，目标必须为 CONVOLUTION 2D，或目标必须为 SEPARABLE 2D。*pname* 参数可选值包括 CONVOLUTION BORDER COLOR、CONVOLUTION BORDER MODE、CONVOLUTION FILTER SCALE、CONVOLUTION FILTER BIAS、CONVOLUTION FORMAT、CONVOLUTION WIDTH、CONVOLUTION HEIGHT、MAX CONVOLUTION WIDTH 或 MAX CONVOLUTION HEIGHT 之一。

指定参数的值将返回在 *params* 中。

6.1.9 直方图查询

当前直方图表的内容通过以下函数进行查询：

```
void GetHistogram( enum target, boolean reset, enum format, enum type,
    void* values );
```

*目标*必须为HISTOGRAM。*type*和*format*参数接受与**GetTexImage**对应参数相同的取值。一维直方图表图像将返回至*values*。像素处理与组件映射与**GetTexImage**完全一致，但不应用最终转换像素存储模式，而是直接将组件值限制在目标数据类型的取值范围内。

若*重置*为TRUE，则直方图所有元素的计数器均重置为零。无论计数器是否被返回，均执行重置操作。

若*重置*为FALSE，则不修改任何计数器。

调用

```
void ResetHistogram( enum target );
```

将直方图表中所有元素的计数器重置为零。*目标*必须是HISTOGRAM。

重置或查询零条目直方图表的内容不会导致错误。

函数

```
void GetHistogramParameter if v( enum target, enum pname, T params );
                                                    { }
```

用于整数和浮点查询。*目标*必须是HISTOGRAM或PROXY HISTOGRAM。*pname*可选值包括HISTOGRAM FORMAT、HISTOGRAM WIDTH、HISTOGRAM RED SIZE、HISTOGRAM GREEN SIZE、HISTOGRAM BLUE SIZE、HISTOGRAM ALPHA SIZE或HISTOGRAM LUMINANCE SIZE。*pname*仅限于目标HISTOGRAM时可为仅当*目标*为HISTOGRAM时，pname可为HISTOGRAM SINK。指定参数的值将通过*params*返回。

6.1.10 最小最大值查询

通过以下命令查询minmax表的当前内容：

```
void GetMinmax( enum target, boolean reset, enum format, enum type, void* values );
```

target 必须为 MINMAX。*type* 和 *format* 的取值范围与 **GetTexImage** 对应参数相同。返回值 *values* 为宽度为 2 的单维图像。像素处理与分量映射机制与 **GetTexImage** 完全一致。

若重置为TRUE，则每个最小值将重置为可表示的最大值，每个最大值将重置为可表示的最小值。无论是否返回，所有值均将重置。

若*重置*为FALSE，则不修改任何值。

调用

```
void ResetMinmax( enum target );
```

*将目标枚举*的所有最小值和最大值分别重置为其可表示的最大值和最小值，*目标枚举*必须为MINMAX。

函数

```
void GetMinmaxParameter(enum target, enum pname, T params);
                                                  { }
```

用于整数和浮点查询。*target*必须为MINMAX。*pname*为MINMAX FORMAT或MINMAX SINK。指定参数的值将通过*params*返回。 _ _

6.1.11 指针与字符串查询

命令

```
void GetPointerv( enum pname, void **params );
```

获取 指针 或 指针 名为 *pname* 在 数组 *params* 中。 *pname* 的可能取值包括：选择缓冲区指针、 反馈缓冲区指针、顶点数组指针、法线数组指针、颜色数组指针、次要颜色数组指针、 _ _ INDEX ARRAY POINTER、TEXTURE COORD ARRAY POINTER、 _ 、FOG COORD ARRAY POINTER、 _ 以及 _ _ EDGE FLAG数组指针。每个指针返回单一指针值。 _ _

最后，_ _ _ _ _ _

```
ubyte *GetString( enum name );
```

返回一个指向静态字符串的指针，该字符串描述当前GL连接[1]的某个方面。*name*的可能值包括VENDOR、RENDERER、VERSION、SHADING LANGUAGE VERSION和EXTENSIONS。RENDERER和VENDOR字符串的格式取决于具体实现。EXTENSIONS字符串包含以空格分隔的扩展名称列表（扩展名称本身不包含空格）。VERSION和SHADING LANGUAGE VERSION字符串的格式取决于具体实现。

和VENDOR字符串的格式取决于具体实现。EXTENSIONS字符串包含以空格分隔的扩展名列表（扩展名本身不包含空格）。VERSION和SHADING LANGUAGE VERSION字符串的布局如下：

<版本号><空格><供应商特定信息>

版本号采用以下格式之一：*主版本号.次版本号* 或 *主版本号.次版本号.发布号*，其中所有数字均包含一个或多个位数。*发布号*和供应商特定信息为可选项。但若存在，则与服务器相关，其格式和内容取决于具体实现。

GetString函数返回连接所支持的版本号（通过VERSION字符串返回）及扩展名列表（通过EXTENSIONS字符串返回）。因此，若客户端与服务器支持不同版本和/或扩展名，则返回兼容版本及扩展名列表。

6.1.12 遮挡查询

命令

```
boolean IsQuery( uint id );
```

若 *id* 为查询对象名称则返回 TRUE。若 *id* 为零，或 *id* 为非零值但非查询对象名称，则 **IsQuery** 返回 FALSE。

可通过以下命令查询目标信息：

```
void GetQueryiv( enum target, enum pname, int *params );
```

若 *pname* 为 CURRENT_QUERY，则 *params* 将存入*目标*当前活动的查询名称；若无活动查询，则存入零值。

如果 *pname* 为 QUERY_COUNTER_BITS，*则目标计数器的位数*将被存入 *params*。查询计数位的数量可能为零，此时计数器不包含有效信息。否则，最小值为

[1] 复制这些静态字符串的应用程序绝不应使用固定长度缓冲区，因为字符串在不同版本间可能不可预测地增长，导致复制时发生缓冲区溢出。此问题在某些GL实现中尤为突出，其中EXTENSIONS字符串已变得极其冗长。

允许的位数取决于实现方案的最大视口尺寸（MAX VIEWPORT DIMS）。在此情况下，计数器必须能够为视口中的每个像素表示至少两次重绘。计算允许最小值的公式如下（其中n为最小位数）：

$$n = \min\{32, \lceil \log_2(\textit{最大视口宽度} \times \textit{最大视口高度} \times 2) \rceil\}$$

可通过以下命令查询查询对象的状态：

```
void GetQueryObjectiv( uint id, enum pname, int *params );
void GetQueryObjectuiv( uint id, enum pname, uint *params );
```

如果 *id* 不是查询对象的名称，或者由 *id* 指定的查询对象当前处于活动状态，则会生成 INVALID OPERATION 错误。

若 *pname* 为 QUERY RESULT,_则查询对象的结果值将存入 *params*。若*目标*查询计数器的位数为零，则结果值始终为 0。

上述查询返回前可能存在不可预测的延迟。若 *pname* 为 QUERY RESULT AVAILABLE，当存在此类延迟时立即返回 FALSE，否则返回 TRUE。必须始终满足：若任何查询对象返回结果可用状态为 TRUE，则该查询之前发出的所有查询也必须返回 TRUE。

查询任意查询对象的状态将强制该遮蔽查询在有限时间内完成。

若在调用 **Get-QueryObject[u]iv** 之前对同一目标和 ID 执行了多个查询，返回结果始终来自最后一次查询。若未在对同一目标和 ID 发起新查询前检索先前查询结果，则所有前置查询结果将丢失。

6.1.13 缓冲区对象查询

命令

```
boolean IsBuffer( uint buffer );
```

*若buffer*是缓冲区对象的名称，则返回TRUE。若*buffer*为零，或*buffer*为非零值但并非缓冲区对象名称，则IsBuffer返回FALSE。

命令

```
void GetBufferSubData( enum target, intptr offset, sizeiptr size, void
    *data );
```

查询缓冲区对象的数据内容。*target*为数组缓冲区或元素数组缓冲区。*offset*和*size*以基本机器单位为单位，指定要查询的缓冲区对象中的数据范围。*data*指定客户端内存中一个区域，其长度为*size*个基本机器单位，用于存取查询到的数据。

若对当前已映射的缓冲区对象执行**GetBufferSubData**操作，将引发错误。

当缓冲区对象的数据存储区处于映射状态时，可通过调用

```
void GetBufferPointerv( enum target, enum pname, void **params );
```

当*目标*设置为数组缓冲区或元素数组缓冲区，*且*pname设置为缓冲区映射指针时。单个缓冲区映射指针将通过**params*返回。若缓冲区数据存储当前未被映射，或请求客户端未映射缓冲区对象的数据存储，且实现无法支持多客户端映射时，**GetBufferPointerv**将返回NULL指针值。

6.1.14 着色器与程序查询

存储在着色器或程序对象中的状态可通过接受着色器或程序对象名称的命令进行查询。若提供的名称既非着色器名称也非程序对象名称，这些命令将生成错误INVALID VALUE；若提供的名称指向另一类型的着色器，则生成错误INVALID OPERATION。若发生错误，用于存储返回值的变量将保持不变。

命令

```
boolean IsShader( uint shader );
```

若 *shader* 是着色器对象名称则返回 TRUE。若 *shader* 为零或非零值但非着色器对象名称，则返回 FALSE。当 *shader* 不是有效着色器对象名称时不产生错误。

命令

```
void GetShaderiv( uint shader, enum pname, int *params );
```

返回*参数*中名为 *shader* 的着色器对象的属性。返回的参数值由 *pname* 指定。

如果 *pname* 是着色器类型，*当着色器*对象为顶点着色器时返回 VERTEX SHADER，当*着色器*对象为片段着色器时返回 FRAGMENT SHADER。如果 *pname* 是删除状态，当着色器已被标记为待删除时返回 TRUE，否则返回 FALSE。 若 *pname* 为 COMPILE STATUS，当着色器上次编译成功时返回 TRUE，否则返回 FALSE。 若 *pname* 为 INFO LOG LENGTH，则返回包含空终止符的信息日志长度。若无信息日志则返回零。若 *pname* 为 SHADER SOURCE LENGTH，则返回构成着色器源代码的字符串拼接长度（含空终止符）。若未定义源代码则返回零。

该命令

布尔型 **IsProgram**(整型 *program*);

若 *program* 是程序对象的名称，则返回 TRUE。若 *program* 为零，或为非零值但非程序对象名称，则 **IsProgram** 返回 FALSE。当 *program* 不是有效的程序对象名称时，不会产生错误。

命令

```
void GetProgramiv( uint program, enum pname, int *params );
```

返回*参数数组*中名为 *program* 的程序对象属性。返回的参数值由 *pname* 指定。

若 *pname* 为 DELETE STATUS，当着色器已被标记为待删除时返回 TRUE，否则返回 FALSE。若 *pname* 为 LINK STATUS，当着色器上次编译成功时返回 TRUE，否则返回 FALSE。若 *pname* 为 VALIDATE STATUS，当最后一次调用 **ValidateProgram** 成功时返回 TRUE，否则返回 FALSE。若 *pname* 为 INFO LOG LENGTH，则返回包含空终止符的信息日志长度。若无信息日志则返回 0。若 *pname* 为 ATTACHED SHADERS，则返回已附加对象的数量。 若 *pname* 为 ACTIVE ATTRIBUTES，则返回*程序中*活动属性的数量。若不存在活动属性，则返回 0。若 *pname* 为 ACTIVE ATTRIBUTE MAX LENGTH，则返回最长活动属性名称的长度（含空终止符）。若不存在活动属性，则返回 0。 若 *pname* 为 ACTIVE UNIFORMS，则返回活动统一变量数量。若无活动统一变量，则返回 0。若 *pname* 为 ACTIVE UNIFORM MAX LENGTH，则返回最长活动统一变量名称的长度（含空终止符）。若无活动统一变量，则返回 0。

命令

```
void GetAttachedShaders( uint program, sizei maxCount, sizei *count, uint *shaders
    );
```

返回附加到*程序中的*着色器对象名称，存储于*着色器数组中*。实际写入*着色器*名称的数量通过*count*返回。若未附加着色器，则*count*设为零。若*count*为NULL，则忽略该参数。 可写入*shaders*的着色器名称最大数量由*maxCount*限定。通过调用**GetProgramiv**并指定ATTACHED SHADERS参数，可查询*程序*中附加的对象数量。

可通过以下命令获取包含着色器对象最后一次编译尝试或程序对象最后一次链接/验证尝试信息的字符串（*称为信息日志*）：

```
void GetShaderInfoLog( uint shader, sizei bufSize, sizei *length, char *infoLog );
void GetProgramInfoLog( uint program, sizei bufSize, sizei *length, char *infoLog );
```

这些命令将信息日志字符串返回*至infoLog*。该字符串以空字符结尾。实际写入*infoLog*的字符数（不含空字符）将通过*length*返回。若*length*为NULL，则不返回长度值。可写入*infoLog*的最大字符数（含空字符）由*bufSize*指定。 可通过**GetShaderiv** 或 **GetProgramiv** 配合 INFO LOG LENGTH 参数查询信息日志的字符数。若 *program* 为着色器对象，返回的信息日志要么为空字符串，要么包含该对象最近一次编译尝试的相关信息。若 *program* 为程序对象，返回的信息日志要么为空字符串，要么包含该对象最近一次链接尝试或验证尝试的相关信息。

信息日志通常仅在应用程序开发期间有用，应用程序不应期望不同的通用逻辑实现会生成完全相同的信息日志。

命令

```
void GetShaderSource( uint shader, sizei bufSize, sizei *length, char
    *source );
```

将着色器*对象shader*的源代码字符串返回至*source参数*。该字符串以空字符终止。实际写入的字符数

将*字符串写入源*（不包括空终止符）的*长度*将返回。若长度为NULL，则不返回长度。可写入*源*的最大字符数（包括空终止符）*由bufSize*指定。*字符串源*是通过**ShaderSource**传递给GL的字符串的连接结果。该连接字符串的长度由SHADER SOURCE LENGTH给出，可通过**GetShaderiv**查询。

命令

```
void GetVertexAttribdv( uint index, enum pname, double *params );
void GetVertexAttribfv( uint index, enum pname, float *params );
void GetVertexAttribiv( uint index, enum pname, int *params );
```

获取名为 *pname* 的顶点属性状态，该属性对应编号为 *index* 的通用顶点属性，并将信息存入数组 *params*。*pname* 必须为 VERTEX ATTRIB ARRAY ENABLED、VERTEX ATTRIB ARRAY SIZE、VERTEX ATTRIB ARRAY STRIDE、顶点属性数组类型、顶点属性数组归一化或当前顶点属性。注意：除CURRENT VERTEX ATTRIB外，所有查询均返回客户端状态。若*索引*值大于或等于MAX VERTEX ATTRIBS，则生成INVALID VALUE错误。

除CURRENT VERTEX ATTRIB外，其余查询均返回通用顶点属性数组信息。通用顶点属性数组的启用状态由**EnableVertexAttribArray命令**设置，通过**DisableVertexAttribArray命令**清除。数组大小、步长、类型及归一化标志由**VertexAttribPointer命令**设定。查询CURRENT VERTEX ATTRIB返回通用属性*索引*的当前值。若*索引*为零，将触发INVALID OPERATION错误，因通用属性零不存在当前值。

命令

```
void GetVertexAttribPointerv( uint index, enum pname, void **pointer );
```

获取名为 *pname* 的指针，用于存储编号为 *index* 的顶点属性，并将信息存入数组*指针*。*pname* 必须是顶点属性数组指针。若 *in-dex* 大于或等于 MAX VERTEX ATTRIBS，则触发 INVALID VALUE 错误。

命令

```
void GetUniformfv( uint program, int location, float *params );
void GetUniformiv( uint program, int location, int *params );
```

返回数组 *params* 中程序对象 *program* 在位置 *location* 的统一变量值。位置 *location* 的统一变量类型决定了返回的值的数量。 若*程序*未成功链接，或*位置*对*程序*无效，则生成错误INVALID OPERATION。查询统一变量数组值时，需对每个数组元素分别发出GetUniform*命令。若查询的统一变量为矩阵，则按列优先顺序返回矩阵值。若发生错误，返回参数*params*将保持不变。

6.1.15 状态保存与恢复

除获取状态变量值外，GL还提供批量保存与恢复状态变量组的功能。**PushAttrib**、**PushClientAttrib**、PopAttrib及PopClientAttrib命令用于实现此目的。命令

```
void PushAttrib( bitfield mask ); void PushClientAttrib(
bitfield mask );
```

对符号常量进行按位或运算，以指示应将哪些状态变量组压入属性堆栈。**PushAttrib** 使用服务器属性堆栈，而 **PushClientAttrib** 使用客户端属性堆栈。每个常量代表一组状态变量。各变量所属组别的分类信息详见下表中的状态变量分类表。 若在堆栈深度达到MAX ATTRIB STACK DEPTH（属性堆栈最大深度）或MAX CLIENT ATTRIB STACK DEPTH（客户端属性堆栈最大深度）时执行PushAttrib或**PushClientAttrib**，将触发STACK OVERFLOW（堆栈溢出）错误。

分别。*掩码*中设置的位若不对应属性组则被忽略。特殊掩码值ALL ATTRIB BITS和CLIENT ALL ATTRIB BITS可分别用于推送所有可堆叠的服务器和客户端状态。

命令

```
void PopAttrib( void ); void
PopClientAttrib( void );
```

将重置上次通过**对应的PushAttrib或PopClientAttrib**保存的状态变量值。未保存的变量保持不变。若在对应栈为空时执行**PopAttrib**或**PopClientAttrib**，将触发STACK UNDERFLOW错误。

表 6.2 列出了属性组及其对应的符号常量名称和堆栈。

当以设置了TEXTURE BIT的PushAttrib调用时，当前绑定纹理对象的优先级、边框颜色、过滤模式和包络模式，以及当前的纹理绑定和启用状态，都会被压入属性堆栈。（未绑定的纹理对象不会被压入或恢复。） 当包含纹理信息的属性集被弹出时，首先将绑定状态和启用状态恢复至其压入值，随后将已绑定纹理对象的优先级、边框颜色、过滤模式和包裹模式恢复至其压入值。

属性组操作会为该组内的所有纹理单元压入或弹出纹理状态。当某个组的状态被压入时，首先压入所有对应于TEXTURE0的状态，接着是对应于TEXTURE1的状态，依此类推直至包含对应于*TEXTUREk*的状态（*其中*k+1为最大纹理单元数值）。 当弹出组状态时，纹理状态将按与压入相反的顺序恢复，从对应*于TEXTUREk*的状态开始，以TEXTURE0结束。客户端纹理状态的压入和弹出操作遵循相同规则。矩阵堆栈绝不会通过**PushAttrib**、**PushClientAttrib**、**PopAttrib**或**PopClientAttrib**进行压入或弹出操作。

每个属性堆栈的深度取决于具体实现，但必须至少为16。每个属性堆栈所需的状态可能包括：每个状态变量的16份副本、16个标记每个堆栈条目中存储变量组的掩码，以及一个属性堆栈指针。初始状态下，两个属性堆栈均为空。

在以下表格中，每个变量都标注了类型。表6.3解释了这些类型。类型实际上标识了与所示描述相关的所有状态；某些情况下仅返回部分状态。 矩阵即属此类，仅返回栈顶元素；裁剪平面仅返回选定平面；灯光参数仅返回对应选定光源的数值；纹理仅返回选定纹理或纹理参数；评估器贴图仅返回选定贴图。 最后，属性列中的“–”表示该值未包含在任何属性组中（因此无法**通过PushAttrib**、**PushClientAttrib**、**Pop**Attrib或**PopClientAttrib**进行压入或弹出操作）。

初始最小最大表值的 *M* 和 *m* 条目分别代表可表示的最大值和最小值。

堆栈	属性	常量
服务器	累加缓冲区	累加缓冲位
服务器	颜色缓冲区	颜色缓冲位
服务器	当前	当前位
服务器	深度缓冲区	深度缓冲位
服务器	启用	启用位
服务器	评估	评估位
服务器	雾	雾位
服务器	提示	提示位
服务器	照明	照明位
服务器	线	线位
服务器	list	LIST_BIT
服务器	多采样	MULTISAMPLE_BIT
服务器	像素	像素模式位
服务器	点	点位位
服务器	多边形	多边形位
服务器	多边形点画	多边形点画笔
服务器	剪刀钻头	剪刀钻头
server	模板缓冲钻头	模板缓冲位
server	纹理	纹理位
服务器	转换	TRANSFORM_BIT
服务器	视口	视口位
服务器		所有属性位
客户端	顶点数组	客户端顶点数组位
客户端	像素存储	客户端像素存储位
客户端	选择	不可入栈或出栈
客户端	反馈	无法被压入或弹出
客户端		客户端所有属性位

表 6.2： 属性组

类型代码	说明
B	布尔值
BMU	基本机器单元
C	颜色（浮点 R、G、B 和 A 值）
CI	颜色索引（浮点索引值）
T	纹理坐标（浮点 *s*、*t*、*r*、*q* 值） 值）
N	法线坐标（浮点数 *x*、*y*、*z* 值）
V	顶点，包括关联数据
Z	整数
Z^+	非负整数
Z_k , Z_{k*}	k值整数（k*表示k为最小值）
R	浮点数
R^+	非负浮点数
$R^{[a,b]}$	[*a, b*] 范围内的浮点数
R^k	k元组的浮点数
P	位置（*x*, *y*, *z*, *w* 浮点坐标）
D	方向（*x*, *y*, *z* 浮点坐标）
M^4	4×4 浮点矩阵
S	NULL 结尾字符串
I	图像
A	属性堆栈条目，包括掩码
Y	指针（数据类型未指定）
n × *类型*	*n*个类型为*type*的副本（n*表示*n*为最小值）

表 6.3： 状态变量类型

6.2 状态表

以下各页表格列出了通过不同命令可获取的状态变量。对于**可通过GetBooleanv、GetIntegerv、GetFloatv或GetDoublev任一命令**获取的状态变量，仅列出其中最适合返回数据类型的命令。此类状态变量无法通过**IsEnabled命令**获取。 然而，对于查询命令列为**IsEnabled**的状态变量，也可通过**GetBooleanv、GetIntegerv、GetFloatv和GetDoublev**获取。若查询命令为其他类型，则必须使用该特定命令获取对应状态变量。

仅由成像子集（参见第3.6.2节）要求的状态表条目以灰色背景显示。

表 6.4. GL 内部开始-结束状态变量（不可访问）

获取价值	类型	获取命令	初始值	描述	Sec.	属性
–	Z11	–	0	当 /= 0 时，表示**开始/结束**对象	2.6.1	–
–	*V*	–	–	**开始/结束**行中的前一个顶点	2.6.1	–
–	*B*	–	–	*指示线顶点*是否为首项	2.6.1	–
–	*V*	–	–	**开始/结束线的**第一个顶点 **循环**	2.6.1	–
–	Z^+	–	–	线点阵计数器	3.4	–
–	$n \times V$	–	–	顶点位于**开始/结束**区域内 **多边形**	2.6.1	–
–	Z^+	–	–	*多边形顶点*数	2.6.1	–
–	$2 \times V$	–	–	三角带中的前两个顶点 **开始/结束三角带**	2.6.1	–
–	Z_3	–	–	迄今为止三角带中的顶点数 三角带：0、1 或更多	2.6.1	–
–	Z_2	–	–	三角带A/B顶点指针	2.6.1	–
–	$3 \times V$	–	–	四边形顶点 构造	2.6.1	–
–	Z_4	–	–	四边形中当前顶点数 带状区域：0、1、2 或更多	2.6.1	–

表 6.5: 当前值及关联数据

获取数值	类型	获取命令	初始值	描述	Sec.	属性
当前颜色 _	*C*	**GetIntegerv, GetFloatv**	1,1,1,1	当前颜色	2.7	当前
当前辅助色 _ _	*C*	**GetIntegerv, GetFloatv**	0,0,0,1	当前辅助颜色	2.7	当前
当前索引 _	*CI*	**GetIntegerv、GetFloatv**	1	当前颜色索引	2.7	当前
当前纹理坐标 _ _	2 * *xT*	**GetFloatv**	0,0,0,1	当前纹理坐标	2.7	当前
当前法线 _	*N*	**GetFloatv**	0,0,1	当前法线	2.7	current
当前雾坐标 _ _	*R*	**GetIntegerv、GetFloatv**	0	当前雾坐标	2.7	当前
–	*C*	–	-	与最后一个顶点关联的颜色	2.6	–
–	*CI*	–	-	与最后一个顶点相关的颜色索引 顶点	2.6	–
–	*T*	–	-	与 最后一个顶点	2.6	–
当前光栅位置 _ _	R^4	**获取浮点值**	0,0,0,1	当前光栅位置	2.13	当前
当前光栅距离 _ _	R^+	**GetFloatv**	0	当前光栅距离	2.13	当前
当前光栅颜色 _ _	*C*	**GetIntegerv、GetFloatv**	1,1,1,1	与光栅位置关联的颜色	2.13	当前
当前光栅索引 _ _	*CI*	**GetIntegerv、GetFloatv**	1	与栅格位置关联的颜色索引	2.13	当前
当前光栅纹理坐标 _ _	2 * *xT*	**GetFloatv**	0,0,0,1	与纹理相关的坐标 光栅位置	2.13	当前
当前光栅位置有效 _ _	*B*	**GetBooleanv**	*真*	栅格位置有效位	2.13	当前
EDGE FLAG _	*B*	**获取布尔值v**	*True*	边缘标志	2.6.2	current

表 6.6. 顶点数组数据

获取值	类型	获取命令	初始值	描述	Sec.	属性
客户端活动纹理	Z_{2^*}	**GetIntegerv**	纹理0	客户端活动纹理单元选择器	2.7	顶点数组
顶点数组	B	**IsEnabled**	*false*	顶点数组启用	2.8	顶点数组
顶点数组大小	Z^+	**GetIntegerv**	4	每个顶点的坐标数	2.8	顶点数组
顶点数组类型	Z_4	**GetIntegerv**	FLOAT	顶点坐标类型	2.8	顶点数组
顶点数组步长	Z^+	**GetIntegerv**	0	顶点间步长	2.8	顶点数组
顶点数组指针	Y	**GetPointerv**	0	顶点数组指针	2.8	顶点数组
法线数组	B	**IsEnabled**	*false*	法线数组启用	2.8	顶点数组
法线数组类型	Z_5	**GetIntegerv**	FLOAT	法线坐标类型	2.8	顶点数组
法线数组步长	Z^+	**GetIntegerv**	0	法线阵列步长	2.8	顶点数组
法线数组指针	Y	**GetPointerv**	0	法线数组指针	2.8	顶点数组
雾化坐标数组	B	**IsEnabled**	*False*	雾坐标数组启用	2.8	顶点数组
雾坐标数组类型	Z_2	**GetIntegerv**	FLOAT	雾坐标分量类型	2.8	顶点数组
雾坐标数组步长	Z^+	**GetIntegerv**	0	雾坐标步长	2.8	顶点数组
雾坐标数组指针	Y	**获取指针v**	0	雾坐标数组指针	2.8	顶点数组
COLOR ARRAY	B	**IsEnabled**	*false*	颜色数组启用	2.8	顶点数组
颜色数组大小	Z^+	**GetIntegerv**	4	每个顶点的颜色分量	2.8	顶点数组
颜色数组类型	Z_8	**GetIntegerv**	FLOAT	颜色组件类型	2.8	顶点数组
颜色数组步长	Z^+	**GetIntegerv**	0	颜色间步长	2.8	顶点数组
颜色数组指针	Y	**GetPointerv**	0	指向颜色数组的指针	2.8	顶点数组
次要颜色数组	B	**IsEnabled**	*false*	辅助颜色数组启用	2.8	顶点数组
次要颜色数组大小	Z^+	**GetIntegerv**	3	每个次要颜色组件 顶点	2.8	顶点数组
次级颜色数组类型	Z_8	**GetIntegerv**	FLOAT	次要颜色的类型 components	2.8	顶点数组
次级颜色数组步长	Z^+	**GetIntegerv**	0	次级颜色间步长	2.8	顶点数组
次要颜色数组指针	Y	**GetPointerv**	0	指向次要颜色数组的指针	2.8	顶点数组
索引数组	B	**IsEnabled**	*false*	索引数组启用	2.8	顶点数组
索引数组类型	Z_4	**GetIntegerv**	FLOAT	索引类型	2.8	顶点数组
索引数组步长	Z^+	**GetIntegerv**	0	索引间步长	2.8	顶点数组
索引数组指针	Y	**GetPointerv**	0	索引数组指针	2.8	顶点数组

表 6.7. 顶点数组数据（续）

获取值	类型	获取命令	初始值	描述	Sec.	属性
纹理坐标数组 纹理坐标数组大小 纹理坐标数组类型 纹理	$2 * \times B$	IsEnabled GetIntegerv GetIntegerv GetIntegerv	*假*	纹理坐标数组启用 每个元素的坐标	2.8 2.8	顶点数组 顶点数
坐标数组步长 纹理坐标数组指针 –	$2 * \times Z^{+}$	GetPointerv	4 FLOAT	纹理坐标类型	2.8	组 顶点数组 顶点
– – –	$2 * \times Z_4$		0	纹理坐标间步长 纹理坐标数组指针	2.8 2.8	数组 顶点数组
顶点属性数组启用顶点属性数组大小顶点属性数组步长顶点	$2 * \times Z^{+}$	GetVertexAttrib GetVertexAttrib GetVertexAttrib GetVertexAttrib	0	顶点属性数组启用 顶点属性数组大小 顶		
属性数组类型 – – –	$2 \times Y$	GetVertexAttrib GetVertex-AttribPointer IsEnabled GetIntegerv	*假*	点属性数组步长 顶点属性数组类型	2.8	顶点数组 顶点数
顶点属性数组归一化顶点属性数组 指针 边标志数组			4	顶点属性数组归一化顶点属性数组指针边标	2.8	组 顶点数组 顶点
– – –	$16 + \times B$	GetPointerv	0	志数组启用	2.8	数组 顶点数组 顶
边标志数组步长 边标志数组指针数组缓冲区绑定	$16 + \times Z$	GetIntegerv	浮点		2.8	
– – –		GetIntegerv		边界标志间步长 指向边界标志数组的指	2.8	点数组 顶点数组
顶点数组缓冲区绑定 正常数组缓冲区绑定 颜色数组缓冲区绑	$16 + \times Z^{+}$	GetIntegerv	*假*		2.8	
– – –		GetIntegerv		针 当前缓冲区绑定 顶点数组缓冲区绑定	2.8	顶点数组 顶点数
定 索引数组缓冲区绑定	$16 + \times Z_4$	GetIntegerv	NULL		2.8	
– – –		GetIntegerv		法线数组缓冲区绑定 颜色数组缓冲区绑	2.8	组 顶点数组 顶点
纹理坐标数组缓冲区绑定 边缘标志数组缓冲区绑定 次要颜色数组缓冲区绑定	$16 + \times B$	GetIntegerv	*假*		2.9	
– – –	$16 + \times P\ B$	GetIntegerv	0	定 索引数组缓冲区绑定	2.9	数组 顶点数组 顶
– – –	$Z^{+} Y$	GetIntegerv	0		2.9	
雾坐标数组缓冲绑定 元素数组缓冲绑定	$Z^{+} Z$		0	纹理坐标数组缓冲绑定 边缘标志数组缓冲绑	2.9	点数组 顶点数组
– – –	$^{+} Z^{+}$	GetIntegerv GetIntegerv	0	定 次要颜色数组缓冲绑定	2.9	
– – –	$Z^{+} Z$		0	雾坐标数组缓冲区绑定 元素数组缓冲区绑定	2.9	顶点数组 顶点数
– –	$^{+}$		0		2.9	组 顶点数组 顶点
– –	$2 * \times Z^{+}$		0		2.9	数组
– – –	Z^{+}		0			
– – –	Z^{+}		0		2.9 2.9.2	顶点数组 顶点数
– – –						组
– – –	$Z^{+} Z$		0			
– – – –	$^{+}$		0			
– – – –						
– – – –						
– – – –						
– – – –						
– – –						

表 6.8. 缓冲区对象状态

获取值	类型	获取命令	初始值	描述	Sec.	属性
	n × *BMU*	**获取缓冲区子数据**	-	缓冲区数据	2.9	-
缓冲区大小 _	*n* × *Z*	**GetBufferParameteriv**	0	缓冲区数据大小	2.9	-
缓冲区使用情况	*n* × *Z*	**GetBufferParameteriv**	静态绘制	缓冲区使用模式	2.9	-
缓冲区访问 _	*n* × *Z*	**GetBufferParameteriv**	读写	缓冲区访问标志	2.9	-
缓冲区映射 _	*n* × *B*	**获取缓冲区参数iv**	FALSE	缓冲区映射标志	2.9	-
缓冲区映射指针 _	*n* × *Y*	**获取缓冲区指针**	NULL	映射缓冲区指针	2.9	-

表 6.9. 转换状态

获取值	类型	获取命令	初始值	描述	Sec.	属性
颜色矩阵 (转置颜色矩阵)	$2 * \times M^4$	**GetFloatv**	单位矩阵	颜色矩阵堆栈	3.6.3	–
模型视图矩阵 (转置模型视图矩阵)	$32 * \times M^4$	**获取浮点v**	单位矩阵	模型视图矩阵堆栈	2.11.2	–
投影矩阵 (转置投影矩阵)	$2 * \times M^4$	**GetFloatv**	单位矩阵	投影矩阵堆栈	2.11.2	–
纹理矩阵 (转置纹理矩阵)	$2 \times 2 \times M^4$	**GetFloatv**	单位矩阵	纹理矩阵堆栈	2.11.2	–
视口	$4 \times Z$	**GetIntegerv**	参见 2.11.1	视口原点与范围	2.11.1	视口
深度范围	$2 \times R^+$	**GetFloatv**	0,1	近远深度范围	2.11.1	视口
颜色矩阵堆栈深度	Z^+	**GetIntegerv**	1	颜色矩阵堆栈 指针	3.6.3	–
模型视图堆栈深度	Z^+	**GetIntegerv**	1	模型视图矩阵堆栈 指针	2.11.2	–
投影堆栈深度	Z^+	**GetIntegerv**	1	投影矩阵堆栈 指针	2.11.2	–
纹理堆栈深度	$2 * \times Z^+$	**GetIntegerv**	1	纹理矩阵堆栈 指针	2.11.2	–
矩阵模式	Z_4	**GetIntegerv**	模型视图	当前矩阵模式	2.11.2	转换
NORMALIZE	B	**IsEnabled**	*false*	当前归一化 标准化开关	2.11.3	转换/启用
重新缩放 正常	B	**IsEnabled**	*false*	当前法线重缩放 开启/关闭	2.11.3	转换/启用
CLIP *PLANE i*	$6 * \times R^4$	**获取剪切平面**	0,0,0,0	用户剪裁平面 系数	2.12	转换
CLIP *PLANEi*	$6 * \times B$	**启用状态**	*false*	用户剪切平面 启用	2.12	转换/启用

表 6.10. 着色

获取值	类型	获取命令	初始值	描述	Sec.	属性
雾色	*C*	**GetFloatv**	0,0,0,0	雾色	3.10	雾
雾指数	*CI*	**GetFloatv**	0	雾指数	3.10	雾
雾密度	*R*	**GetFloatv**	1.0	指数雾密度	3.10	雾
雾起始	*R*	**GetFloatv**	0.0	线性雾起始	3.10	雾
雾结束	*R*	**GetFloatv**	1.0	线性雾结束	3.10	雾
雾模式	Z_3	**GetIntegerv**	EXP	雾模式	3.10	雾
FOG	*B*	**IsEnabled**	*false*	启用雾效时为真	3.10	雾/启用
雾协调源	Z_2	**GetIntegerv**	片段深度	雾的坐标源计算	3.10	雾
颜色总和	*B*	**启用**	*false*	启用颜色总和时为真	3.9	雾/启用
阴影模型	Z^+	**GetIntegerv**	平滑	**阴影模型**设置	2.14.7	照明

表 6.11. 照明（默认值参见表 2.10）

获取值	类型	获取命令	初始值	描述	Sec.	属性
照明	*B*	**是否启用**	*false*	若启用照明则为真 启用时为真	2.14.1	lighting/enable
颜色材质 _	*B*	**IsEnabled**	*false*	若颜色 启用	2.14.3	照明/启用
颜色材质参数 _	Z_5	**GetIntegerv**	环境与漫反射	材质 属性追踪当前颜色	2.14.3	照明
颜色材质面 _ _	Z_3	**GetIntegerv**	正面和背面	受影响面 通过颜色追踪	2.14.3	照明
环境	$2 \times C$	**GetMaterialfv**	(0.2,0.2,0.2,1.0)	环境材质 颜色	2.14.1	照明
漫反射	$2 \times C$	**获取材质fv**	(0.8,0.8,0.8,1.0)	漫反射材质 颜色	2.14.1	照明
镜面	$2 \times C$	**获取材质fv**	(0.0,0.0,0.0,1.0)	镜面材质 颜色	2.14.1	照明
发射	$2 \times C$	**获取材质fv**	(0.0,0.0,0.0,1.0)	发光材质 颜色	2.14.1	照明
光泽度	$2 \times R$	**获取材质fv**	0.0	镜面反射 材质指数	2.14.1	光照
光照模型环境光 _	*C*	**GetFloatv**	(0.2,0.2,0.2,1.0)	环境场景 颜色	2.14.1	照明
光照模型本地查看器 _ _	*B*	**GetBooleanv**	*假*	查看器为本地	2.14.1	照明
双面光照模型 _ _	*B*	**GetBooleanv**	*假*	使用双面 照明	2.14.1	照明
灯光模型色彩控制 _ _	Z_2	**GetIntegerv**	单色	色彩控制	2.14.1	照明

表 6.12. 照明（续）

获取值	类型	获取命令	初始值	描述	Sec.	属性
环境	8 * ×C	**GetLightfv**	(0.0,0.0,0.0,1.0)	环境光强度 *i*	2.14.1	照明
漫反射	8 * ×C	**GetLightfv**	参见表 2.10	光的漫反射强度	2.14.1	照明
镜面	8 * ×C	**GetLightfv**	参见表 2.10	光的镜面强度	2.14.1	照明
位置	8 * ×P	**GetLightfv**	(0.0,0.0,1.0,0.0)	灯光的位置	2.14.1	照明
恒定衰减	8 * ×R+	**GetLightfv**	1.0	恒定衰减因子	2.14.1	照明
线性衰减	8 * ×R+	**GetLightfv**	0.0	线性衰减系数	2.14.1	照明
二次衰减	8 * ×R+	**GetLightfv**	0.0	二次衰减因子	2.14.1	照明
光斑方向	8 * ×D	**GetLightfv**	(0.0,0.0,-1.0)	光点方向为光 *i*	2.14.1	照明
聚光指数	8 * ×R+	**获取Lightfv**	0.0	光的聚光指数	2.14.1	照明
光斑截止	8 * ×R+	**GetLightfv**	180.0	光斑角度 *i*	2.14.1	照明
LIGHTi	8 * ×B	**启用**	*false*	启用的为真	2.14.1	lighting/enable
颜色索引	2 × 3 × 红	**GetMaterialfv**	0,1,1	a_m , d_m , 和 s_m 用于颜色指数照明	2.14.1	照明

获取值	类型	获取命令	初始值	描述	Sec.	属性
字号 _	R^+	**GetFloatv**	1.0	字号	3.3	点
点平滑 _	*B*	**启用**	*false*	启用点抗锯齿	3.3	点/启用
点精灵 _	*B*	**IsEnabled**	*false*	点精灵启用	3.3	点/启用
点大小最小值 _	R^+	**GetFloatv**	0.0	衰减最小点尺寸	3.3	点
点尺寸最大值 _	R^+	**GetFloatv**	[1]	衰减最大点尺寸。[1] 实现相关的最大混叠点尺寸与平滑点尺寸的最大值。	3.3	点
点淡出阈值尺寸 _ _	R^+	**GetFloatv**	1.0	α衰减阈值	3.3	点
点距离衰减 _	$3 \times R^+$	**获取浮点值**	1,0,0	衰减系数	3.3	点
点精灵坐标原点 _ _	Z_2	**GetIntegerv**	左上角	点精灵的原点方向	3.3	点
线宽 _	R^+	**GetFloatv**	1.0	线宽	3.4	线
线平滑 _	*B*	**启用**	*false*	启用线条抗锯齿	3.4	线/启用
线条点状图案 _	Z^+	**GetIntegerv**	1's	线点画	3.4.2	线条
线点重复 _	Z^+	**GetIntegerv**	1	线点重复	3.4.2	线
线点画 _	*B*	**IsEnabled**	*false*	线点画启用	3.4.2	线条/启用
CULL FACE _	*B*	**启用**	*假*	多边形剔除启用	3.5.1	多边形/启用
面剔除模式 _	Z_3	**GetIntegerv**	BACK	剔除正面/背面多边形	3.5.1	polygon
正面 _	Z_2	**GetIntegerv**	逆时针	多边形前表面顺时针/逆时针指示器	3.5.1	多边形
多边形平滑 _	*B*	**启用**	*false*	多边形抗锯齿开启	3.5	多边形/启用
多边形模式 _	$2 \times Z_3$	**GetIntegerv**	填充	多边形光栅化模式（前&后）	3.5.4	多边形
多边形偏移因子 _ _	*R*	**GetFloatv**	0	多边形偏移因子	3.5.5	polygon
多边形偏移单位 _ _	*R*	**GetFloatv**	0	多边形偏移单位	3.5.5	polygon
多边形偏移点 _ _	*B*	**IsEnabled**	*false*	启用点多边形偏移模式光栅化	3.5.5	多边形/启用
多边形偏移线 _ _	*B*	**IsEnabled**	*false*	启用线段的多边形偏移模式光栅化	3.5.5	polygon/启用
多边形偏移填充 _ _	*B*	**IsEnabled**	*false*	多边形偏移填充启用模式光栅化	3.5.5	polygon/enable
_	*I*	**获取多边形点状填充**	1's	多边形点画	3.5	polygon-stipple
多边形点画 _	*B*	**IsEnabled**	*false*	多边形点画启用	3.5.2	polygon/启用

表 6.13. 光栅化

表 6.14. 多重采样

获取值	类型	获取命令	初始值	描述	Sec.	属性
多采样	*B*	**是否启用**	*True*	多采样光栅化	3.2.1	多采样/启用
将Alpha采样到覆盖层	*B*	**IsEnabled**	*false*	从alpha修改覆盖率	4.1.3	多采样/启用
将 alpha 采样为 1	*B*	**IsEnabled**	*假*	将透明度设为最大值	4.1.3	多采样/启用
采样覆盖率	*B*	**IsEnabled**	*false*	用于修改覆盖率的掩码	4.1.3	多采样/启用
采样覆盖值	R^{+}	**GetFloatv**	1	覆盖掩码值	4.1.3	多采样
覆盖采样反转	*B*	**GetBooleanv**	*假*	覆盖掩码值反转	4.1.3	multisample

表 6.15. 纹理（按纹理单元和绑定点划分的状态）

获取值	类型	获取命令	初始值	描述	Sec.	属性
纹理 xD –	2 * ×3 × *B*	**启用**	*false*	当 xD 纹理化启用时为真 启用时为真；*x* 为 1、2 或 3	3.8.15	纹理/启用
纹理立方体贴图_ –	2 * ×*B*	**IsEnabled**	*false*	若启用立方体贴图纹理化，则为真 纹理映射已启用	3.8.13	纹理/启用
纹理绑定 xD – –	2 * ×3 × *Z*+	**GetIntegerv**	0	绑定到纹理对象 纹理 xD _	3.8.12	纹理
纹理绑定立方体贴图 – –	2 * ×*Z*+	**GetIntegerv**	0	绑定到纹理对象 纹理立方体贴图_ _	3.8.11	纹理
纹理 xD –	*n* × *l*	**GetTexImage**	参见 3.8	xD 纹理图像在 l.o.d. *i*	3.8	–
纹理立方体贴图正向X – – –	*n* × *l*	**GetTexImage**	参见 3.8.1	+*x* 面体贴图 纹理图像在细节 *等级*	3.8.1	–
纹理立方贴图负X轴 – – –	*n* × *l*	**GetTexImage**	参见 3.8.1	–*x* 面立方体贴图 在细节等级 *i* 下的纹理图像	3.8.1	–
纹理立方贴图正y轴 – – –	*n* × *l*	**GetTexImage**	参见 3.8.1	+*y*面立方体贴图 在细节等级 *i* 下的纹理图像	3.8.1	–
纹理立方贴图负y轴 – – –	*n* × *l*	**GetTexImage**	参见 3.8.1	–*y*面立方体贴图 在细节等级 *i* 下的纹理图像	3.8.1	–
纹理立方贴图 正Z轴 – – –	*n* × *l*	**GetTexImage**	参见 3.8.1	+*z*面立方体贴图 在细节等级 *i* 下的纹理图像	3.8.1	–
纹理立方贴图负Z轴 – – –	*n* × *l*	**GetTexImage**	参见 3.8.1	–*z*面立方体贴图 在细节等级 *i* 下的纹理图像	3.8.1	–

表 6.16. 纹理（每个纹理对象的状态）

获取值	类型	获取命令	初始值	描述	第	属性
纹理边框颜色 _ _	$n \times C$	**GetTexParameter**	0,0,0,0	纹理边框颜色	3.8	纹理
纹理最小滤波 _ _	$n \times Z_6$	**GetTexParameter**	参见 3.8	纹理最小化函数	3.8.8	纹理
纹理磁滤镜 _ _	$n \times Z_2$	**GetTexParameter**	参见 3.8	纹理放大函数	3.8.9	纹理
纹理卷绕 S _ _	$n \times Z_5$	**GetTexParameter**	重复	纹理坐标 *s* 包裹模式	3.8.7	纹理
纹理卷绕 T _ _	$n \times Z_5$	**GetTexParameter**	REPEAT	纹理坐标 *t* 包裹模式 (仅限2D、3D、立方体贴图纹理)	3.8.7	纹理
纹理卷绕 R _ _	$n \times Z_5$	**GetTexParameter**	重复	纹理坐标 *r* 包裹模式 (仅限3D纹理)	3.8.7	纹理
纹理优先级 _	$n \times R^{[0,1]}$	**GetTexParameterfv**	1	纹理对象优先级	3.8.12	纹理
纹理驻留 _	$n \times B$	**GetTexParameteriv**	参见 3.8.12	纹理驻留	3.8.12	纹理
纹理最小LOD _ _	$n \times R$	**GetTexParameterfv**	-1000	最小细节级别	3.8	纹理
纹理最大LOD _ _	$n \times R$	**GetTexParameterfv**	1000	最大细节级别	3.8	纹理
纹理基础级别 _ _	$n \times Z$	**GetTexParameterfv**	0	基底纹理数组	3.8	纹理
纹理最大层级 _ _	$n \times Z$	**GetTexParameterfv**	1000	最大纹理数组级别	3.8	纹理
纹理LOD偏移量_ _	$n \times R$	**GetTexParameterfv**	0.0	纹理细节级别偏移量 $偏移量_{纹理对象}$	3.8.8	纹理
深度纹理模式 _	$n \times Z_3$	**GetTexParameteriv**	LUMINANCE	深度纹理模式	3.8.5	纹理
纹理比较模式 _ _	$n \times Z_2$	**GetTexParameteriv**	无	纹理比较模式	3.8.14	纹理
纹理比较函数 _ _	$n \times Z_8$	**GetTexParameteriv**	LEQUAL	纹理比较函数	3.8.14	纹理
生成MIPMAP _	$n \times B$	**获取文本参数**	FALSE	自动 Mipmap 生成	3.8.8	纹理

表 6.17：纹理（每张纹理图像的状态）

获取值	类型	获取命令	初始值	描述	Sec.	属性
纹理宽度 –	*n* × *Z*	**获取纹理层参数**	0	纹理图像的指定宽度	3.8	–
纹理高度 –	*n* × *Z*	**获取纹理层参数**	0	2D/3D纹理图像的指定高度	3.8	–
纹理深度 –	*n* × *Z*	**获取纹理层参数**	0	3D纹理图像的指定深度	3.8	–
纹理边界 –	*n* × *Z*	**获取纹理层参数**	0	纹理图像的指定边框宽度	3.8	–
纹理内部格式 – – (纹理组件) –	*n* × Z_{42*}	**获取纹理层参数**	1	纹理图像的内部图像格式	3.8	–
纹理红色尺寸 – –	*n* × *Z*	**获取纹理级别参数**	0	纹理图像的红色分辨率	3.8	–
纹理绿色尺寸 – –	*n* × *Z*	**获取纹理级别参数**	0	纹理图像的绿色分辨率	3.8	–
蓝色纹理尺寸 – –	*n* × *Z*	**获取文本级别参数**	0	纹理图像的蓝色分辨率	3.8	–
纹理透明尺寸 – –	*n* × *Z*	**获取纹理层参数**	0	纹理图像的透明度分辨率	3.8	–
纹理亮度尺寸 – –	*n* × *Z*	**获取纹理级别参数**	0	纹理图像的亮度分辨率	3.8	–
纹理强度尺寸 – –	*n* × *Z*	**获取纹理级别参数**	0	纹理图像的强度分辨率	3.8	–
纹理深度尺寸 – –	*n* × *Z*	**获取纹理层参数**	0	纹理图像的深度分辨率	3.8	–
纹理压缩 –	*n* × *B*	**获取纹理级别参数**	*假*	若纹理图像具有压缩内部格式则为真压缩内部格式时返回真	3.8.3	-
纹理压缩图像尺寸 – –	*n* × *Z*	**获取纹理级别参数**	0	纹理压缩后的图像尺寸（单位：微字节）压缩纹理图像	3.8.3	-

表 6.18. 纹理环境与生成

获取值	类型	获取命令	初始值	描述	Sec.	属性
坐标替换 _	2 * ×B	**GetTexEnviv**	*false*	坐标替换启用	3.3	点
活动纹理 _	Z2*	**GetIntegerv**	TEXTURE0	活动纹理单元选择器	2.7	纹理
纹理环境模式 _ _	2 * ×Z_6	**GetTexEnviv**	MODULATE	纹理应用函数	3.8.13	纹理
纹理环境颜色 _ _	2 * ×C	**GetTexEnvfv**	0,0,0,0	纹理环境颜色	3.8.13	纹理
纹理 LOD 偏移 _ _	2 * ×R	**GetTexEnvfv**	0.0	纹理细节级别偏移量 *纹理偏移单位*	3.8.8	纹理
纹理生成器 *x* _ _	2 * ×4 × B	**启用**	*false*	纹理生成启用（*x* 为 S、T、R 或 Q）	2.11.4	纹理/启用
EYE PLANE	2 * ×4 × R^4	**GetTexGenfv**	参见2.11.4	Texgen平面方程系数 （适用于S、T、R和Q）	2.11.4	纹理
对象平面 _	2 * ×4 × R^4	**GetTexGenfv**	参见 2.11.4	纹理生成对象线性系数 （适用于S、T、R和Q）	2.11.4	纹理
纹理生成模式 _ _	2 * ×4 × Z_5	**GetTexGeniv**	EYE_LINEAR	用于纹理生成（适用于S、T、 R和Q）	2.11.4	纹理
组合RGB _	2 * ×Z_8	**GetTexEnviv**	调制	RGB 组合器功能	3.8.13	纹理
组合阿尔法 _	2 * ×Z_6	**GetTexEnviv**	调制	Alpha组合器功能	3.8.13	纹理
SRC0 RGB	2 * ×Z_3	**GetTexEnviv**	纹理	RGB 源 0	3.8.13	纹理
SRC1 RGB	2 * ×Z_3	**GetTexEnviv**	PREVIOUS	RGB 源 1	3.8.13	纹理
SRC2 RGB	2 * ×Z_3	**GetTexEnviv**	常量	RGB 源 2	3.8.13	纹理
SRC0ALPHA	2 * ×Z_3	**GetTexEnviv**	纹理	Alpha 源 0	3.8.13	纹理
SRC1 ALPHA	2 * ×Z_3	**GetTexEnviv**	PREVIOUS	Alpha 源 1	3.8.13	纹理
SRC2 ALPHA	2 * ×Z_3	**GetTexEnviv**	常量	Alpha 源 2	3.8.13	纹理
操作数0RGB _	2 * ×Z_4	**GetTexEnviv**	SRC_COLOR	RGB 操作数 0	3.8.13	纹理
操作数1 RGB _	2 * ×Z_4	**GetTexEnviv**	SRC_COLOR	RGB 操作数 1	3.8.13	纹理
操作数2 RGB _	2 * ×Z_4	**GetTexEnviv**	SRC_ALPHA	RGB 操作数 2	3.8.13	纹理
操作数0 透明度 _	2 * ×Z_2	**GetTexEnviv**	SRC_ALPHA	Alpha 操作数 0	3.8.13	纹理
操作数1阿尔法 _	2 乘以 ×Z_2	**GetTexEnviv**	SRC_ALPHA	Alpha 操作数 1	3.8.13	纹理
操作数2阿尔法 _	2 乘以 ×Z_2	**GetTexEnviv**	SRC_ALPHA	Alpha 操作数 2	3.8.13	纹理
RGB 标度	2 * ×R^3	**GetTexEnvfv**	1.0	RGB后组合器缩放	3.8.13	纹理
ALPHA SCALE	2 * ×R^3	**GetTexEnvfv**	1.0	Alpha后组合器缩放	3.8.13	纹理

表 6.19: 像素操作

获取值	类型	获取命令	初始值	描述	Sec.	属性
剪刀测试	B	**是否启用**	*假*	剪刀测试启用	4.1.2	剪刀/启用
剪刀盒	$4 \times Z$	**GetIntegerv**	参见 4.1.2	剪刀盒	4.1.2	剪刀
ALPHA TEST	B	**IsEnabled**	*false*	启用 Alpha 测试	4.1.4	颜色缓冲区/启用
ALPHA TEST FUNC	Z_8	**GetIntegerv**	ALWAYS	Alpha 测试函数	4.1.4	颜色缓冲区
ALPHA TEST REF	R^+	**GetIntegerv**	0	Alpha测试参考值	4.1.4	颜色缓冲区
模板测试	B	**启用**	*false*	模板启用	4.1.5	模板缓冲区/启用
模板功能	Z_8	**GetIntegerv**	ALWAYS	前模板功能	4.1.5	模板缓冲区
模板值掩码	Z^+	**GetIntegerv**	1's	前模板掩码	4.1.5	模板缓冲区
模板引用	Z^+	**GetIntegerv**	0	前模板参考值	4.1.5	模板缓冲区
模板故障	Z_8	**GetIntegerv**	保留	前模板失败操作	4.1.5	模板缓冲区
模板通道深度失败	Z_8	**GetIntegerv**	保留	前模板深度缓冲区失败操作	4.1.5	模板缓冲区
模板通道深度通道	Z_8	**GetIntegerv**	保留	前模板深度缓冲区通道操作	4.1.5	模板缓冲区
模板后功能	Z_8	**GetIntegerv**	ALWAYS	后模板函数	4.1.5	模板缓冲区
模板后值掩码	Z^+	**GetIntegerv**	1's	背板模板掩码	4.1.5	模板缓冲区
模板背面引用	Z^+	**GetIntegerv**	0	后模板引用值	4.1.5	模板缓冲区
模板背面故障	Z_8	**GetIntegerv**	保持	背板失败操作	4.1.5	模板缓冲区
模板背部通过深度失败	Z_8	**获取整数值**	保持	后模板深度缓冲区失败操作	4.1.5	模板缓冲区
模板后通道深度通道	Z_8	**GetIntegerv**	保持	后向模板深度缓冲区传递操作	4.1.5	模板缓冲区
深度测试	B	**启用**	*false*	深度缓冲启用	4.1.6	深度缓冲区/启用
深度函数	Z_8	**GetIntegerv**	LESS	深度缓冲区测试函数	4.1.6	深度缓冲区

表 6.20. 像素操作（续）

获取值	类型	获取命令	初始值	描述	第	属性
混合	*B*	**是否启用**	*false*	混合启用	4.1.8	颜色缓冲区/启用
混合源 RGB (v1.3:_混合源)	Z15	**GetIntegerv**	ONE	混合源 RGB 功能	4.1.8	颜色缓冲区
混合源阿尔法	Z15	**GetIntegerv**	ONE	混合源A函数	4.1.8	颜色缓冲区
BLEND DST RGB (v1.3:BLEND DST)	Z14	**GetIntegerv**	ZERO	混合目标 RGB 函数	4.1.8	颜色缓冲区
混合目标Alpha	Z14	**GetIntegerv**	ZERO	混合目标函数	4.1.8	颜色缓冲区
混合方程 RGB (v1.5: 混合方程)	*Z*	**GetIntegerv**	FUNC_ADD	RGB 混合方程式	4.1.8	颜色缓冲区
混合方程式阿尔法	*Z*	**GetIntegerv**	FUNC_ADD	Alpha 混合方程式	4.1.8	颜色缓冲区
混合颜色	*C*	**GetFloatv**	0,0,0,0	常量混合颜色	4.1.8	颜色缓冲区
DITHER	*B*	**IsEnabled**	*True*	启用抖动	4.1.9	颜色缓冲区/启用
索引逻辑运算 (v1.0:逻辑运算)	*B*	**IsEnabled**	*false*	索引逻辑操作启用	4.1.10	颜色缓冲区/启用
颜色逻辑操作	*B*	**IsEnabled**	*false*	颜色逻辑操作启用	4.1.10	颜色缓冲区/启用
逻辑运算模式	Z16	**GetIntegerv**	COPY	逻辑运算函数	4.1.10	颜色缓冲区

表 6.21：帧缓冲区控制

获取价值	类型	获取命令	初始值	描述	Sec.	属性
绘制*缓冲区*	$1 + \times Z_{10*}$	**GetIntegerv**	参见 4.2.1	选定输出绘图缓冲区 颜色 *i*	4.2.1	颜色缓冲区
索引写入掩码	Z^{+}	**GetIntegerv**	1's	颜色索引写入掩码	4.2.2	颜色缓冲区
COLOR WRITEMASK	$4 \times B$	**GetBooleanv**	*真*	颜色写入启用；R、G、B 或 A	4.2.2	颜色缓冲区
深度写入掩码	B	**GetBooleanv**	*True*	深度缓冲区启用写入	4.2.2	深度缓冲区
模板写入掩码	Z^{+}	**GetIntegerv**	1's	前模板缓冲区写入掩码	4.2.2	模板缓冲区
模板后写入掩码	Z^{+}	**GetIntegerv**	1's	后模板缓冲区写入掩码	4.2.2	模板缓冲区
清除颜色值	C	**GetFloatv**	0,0,0,0	颜色缓冲区清除值（RGBA 模式）	4.2.3	色缓冲区
索引清除值	CI	**GetFloatv**	0	颜色缓冲区清除值（颜色索引模式）	4.2.3	色缓冲区
深度清除值	R^{+}	**GetIntegerv**	1	深度缓冲区清除值	4.2.3	深度缓冲区
模板清除值	Z^{+}	**GetIntegerv**	0	模板清除值	4.2.3	模板缓冲区
累加器清除值	$4 \times R^{+}$	**GetFloatv**	0	累加缓冲区清除值	4.2.3	累加缓冲区

表 6.22. 像素

获取值	类型	获取命令	初始值	描述	Sec.	属性
拆包交换字节	*B*	**获取布尔值v**	*假*	UNPACK SWAP BYTES 的值	3.6.1	像素商店
先解压LSB	*B*	**GetBooleanv**	*假*	UNPACK LSB FIRST 的值	3.6.1	像素存储
UNPACK IMAGE HEIGHT	Z^+	**GetIntegerv**	0	值 UNPACK IMAGE HEIGHT	3.6.1	像素存储
解包跳过图像	Z^+	**GetIntegerv**	0	UNPACK SKIP IMAGES 的值	3.6.1	像素存储
UNPACK ROW LENGTH	Z^+	**GetIntegerv**	0	UNPACK ROW LENGTH 的值	3.6.1	像素存储
UNPACK 跳过行数	Z^+	**GetIntegerv**	0	UNPACK SKIP ROWS 的值	3.6.1	像素存储
UNPACK SKIP PIXELS	Z^+	**GetIntegerv**	0	UNPACK SKIP PIXELS 的值	3.6.1	像素存储
UNPACK 对齐	Z^+	**GetIntegerv**	4	UNPACK对齐值	3.6.1	像素存储
PACK SWAP BYTES	*B*	**GetBooleanv**	*false*	PACK SWAP BYTES 的值	4.3.2	像素存储
PACK LSB FIRST	*B*	**GetBooleanv**	*假*	PACK LSB FIRST 的值	4.3.2	像素存储
PACK IMAGE HEIGHT	Z^+	**GetIntegerv**	0	PACK IMAGE HEIGHT 的值	4.3.2	像素存储
PACK SKIP IMAGES	Z^+	**GetIntegerv**	0	PACK SKIP IMAGES 的值	4.3.2	像素存储
PACK ROW LENGTH	Z^+	**GetIntegerv**	0	PACK ROW LENGTH 的值	4.3.2	像素存储
PACK SKIP ROWS	Z^+	**GetIntegerv**	0	PACK SKIP ROWS 的值	4.3.2	像素存储
PACK SKIP PIXELS	Z^+	**GetIntegerv**	0	PACK SKIP PIXELS 的值	4.3.2	像素存储
PACK ALIGNMENT	Z^+	**GetIntegerv**	4	PACK ALIGNMENT 的值	4.3.2	像素存储
MAP COLOR	*B*	**GetBooleanv**	*false*	如果颜色已映射则为真	3.6.3	像素
映射模板	*B*	**GetBooleanv**	*false*	若模板值已映射则为真	3.6.3	像素
索引偏移	*Z*	**GetIntegerv**	0	INDEX SHIFT 的值	3.6.3	像素
索引偏移	*Z*	**GetIntegerv**	0	索引偏移值	3.6.3	像素
x 比例尺	*R*	**获取浮点值**	1	*x* SCALE 的数值；*x* 代表红色、绿色、蓝色、透明度或深度	3.6.3	像素
x 偏移量	*R*	**GetFloatv**	0	*x* BIAS 的值	3.6.3	像素

表 6.23. 像素（续）

获取值	类型	获取命令	初始值	描述	Sec.	属性
颜色表 _	*B*	**是否启用**	*false*	若为颜色表则为真 查找已完成	3.6.3	像素/启用
卷积后颜色表 _ _	*B*	**启用**	*false*	若后卷积 进行颜色表查找时为真	3.6.3	像素/启用
后处理颜色矩阵颜色表 _ _	*B*	**IsEnabled**	*false*	若后处理色彩矩阵 颜色表查找	3.6.3	像素/启用
颜色表 _	*I*	**获取颜色表**	空	颜色表	3.6.3	–
卷积后颜色表 _ _	*I*	**获取颜色表**	空	卷积后颜色 表	3.6.3	–
后处理颜色矩阵颜色表 _ _	*I*	**获取颜色表**	空	后处理色彩矩阵色彩 表	3.6.3	–
颜色表格式 _ _	$2 \times 3 \times Z_{42}$	**获取颜色表- 参数iv**	RGBA	颜色表的内部 图像格式	3.6.3	–
颜色表宽度 _ _	$2 \times 3 \times Z^{+}$	**GetColorTable- 参数iv**	0	颜色表的指定值 宽度	3.6.3	–
颜色表 × 尺寸 _ _	$6 \times 2 \times 3 \times Z^{+}$	**获取颜色表- 参数iv**	0	颜色表组件 分辨率；*x* 为红色、绿色、蓝色、透明度、亮度或强度	3.6.3	–
颜色表比例 _ _	$3 \times R^{4}$	**GetColorTable- 参数fv**	1,1,1,1	应用于 颜色表条目	3.6.3	像素
颜色表偏移 _ _	$3 \times R^{4}$	**获取颜色表- 参数fv**	0,0,0,0	应用于 颜色表条目	3.6.3	像素

表 6.24. 像素（续）

获取值		类型	获取命令	初始值	描述	Sec.	属性
卷积 1D	–	*B*	**是否启用**	*false*	若启用一维卷积则为真	3.6.3	像素/启用
卷积 2D	–	*B*	**启用**	*false*	若启用二维卷积则为真	3.6.3	像素/启用
可分离2D	–	*B*	**启用**	*false*	若可分离的二维卷积成立则为真 卷积	3.6.3	像素/启用
卷积 xD	–	2 × *I*	**获取卷积- 滤波器**	*空*	卷积滤波器；*x* 为 1 或 2	3.6.3	–
可分离二维	–	2 × *I*	**GetSeparable- Fil- ter**	*空*	可分离卷积 滤波器	3.6.3	–
卷积边框颜色	– –	3 × *C*	**获取卷积- Parameterfv**	0,0,0,0	卷积边框颜色	3.6.5	像素
卷积边框模式	– –	3 × Z_4	**获取卷积- Parameteriv**	缩减	卷积边界 模式	3.6.5	像素
卷积滤波器比例	– –	3 × R^4	**获取卷积- 参数fv**	1,1,1,1	应用于卷积滤波器的缩放因子 卷积滤波器条目	3.6.3	像素
卷积滤波器偏置	– –	3 × R^4	**获取卷积- 参数fv**	0,0,0,0	应用于卷积滤波器的偏置因子 卷积滤波器条目	3.6.3	像素
卷积格式	–	3 × Z_{42}	**获取卷积参数 参数iv**	RGBA	卷积滤波器 内部格式	3.6.5	–
卷积宽度	–	3 × Z^+	**获取卷积- Parameteriv**	0	卷积滤波器宽度	3.6.5	–
卷积高度	–	2 × Z^+	**获取卷积- Parameteriv**	0	卷积滤波器高度	3.6.5	–

表 6.25. 像素（续）

获取值	类型	获取命令	初始值	描述	Sec.	属性
卷积后 *x* 缩放 _ _	*R*	**GetFloatv**	1	卷积后各通道缩放因子 卷积后；*x* 代表红、绿、蓝或透明度通道	3.6.3	像素
卷积后 *x* 偏置 _ _	*R*	**GetFloatv**	0	组件偏置因子 卷积后	3.6.3	像素
后处理色彩矩阵 × 缩放因子 _ _	*R*	**GetFloatv**	1	组件缩放因子 色彩矩阵之后	3.6.3	像素
后置色彩矩阵 × 偏置 _ _ _	*R*	**获取浮点值**	0	分量偏置因子 色彩矩阵之后	3.6.3	像素
直方图	*B*	**启用**	false	若直方图功能启用则为真 启用时为真	3.6.3	像素/启用
直方图	*I*	**获取直方图**	空	直方图表	3.6.3	–
直方图宽度 _	$2 \times Z^{+}$	**获取直方图-参数iv**	0	直方图表宽度	3.6.3	–
直方图格式 _	$2 \times Z_{42}$	**获取直方图-参数iv**	RGBA	直方图表内部格式	3.6.3	–
直方图 × 尺寸 _ _	$5 \times 2 \times Z^{+}$	**获取直方图-参数iv**	0	直方图表 分量分辨率；*x* 为红色、绿色、蓝色、透明度或亮度	3.6.3	–
直方图下沉 _	*B*	**获取直方图-Parameteriv**	假	若直方图计算消耗像素组时	3.6.3	–

表 6.26. 像素（续）

获取数值	类型	获取命令	初始值	描述	Sec.	属性
MINMAX	*B*	**是否启用**	false	如果minmax为真，则为真 启用时为真	3.6.3	像素/启用
MINMAX	*R*″	**获取最小最大值**	(M,M,M,M),(m,m,m,m)	最小最大表	3.6.3	–
MINMAX格式 _	Z42	**GetMinmax-** **参数iv**	RGBA	最小最大表内部 格式	3.6.3	–
MINMAX 接收器	*B*	**GetMinmax-** **参数iv**	假	若minmax 消耗像素组时	3.6.3	–
变焦 X _	*R*	**获取浮点值**	1.0	*x* 缩放系数	3.6.4	像素
缩放 Y _	*R*	**GetFloatv**	1.0	*y* 缩放因子	3.6.4	像素
x	8 × 32 * ×*R*	**GetPixelMap**	0	RGBA像素**映射** 转换表；*x* 是表 3.3 中的映射名称	3.6.3	–
x	2 × 32 * ×*Z*	**GetPixelMap**	0's	索引**像素图** 转换表；*x* 是表 3.3 中的映射名称	3.6.3	–
x 尺寸	Z^{+}	**GetIntegerv**	1	表 *x* 的大小	3.6.3	–
读取缓冲区	Z_3	**GetIntegerv**	参见4.3.2	读取源缓冲区	4.3.2	像素

表 6.27: 评估器（GetMap 接受地图名称）

获取值	类型	获取命令	初始值	描述	Sec.	属性
顺序	$9 \times Z_{8*}$	**GetMapiv**	1	1d 地图顺序	5.1	–
顺序	$9 \times 2 \times Z_{8*}$	**GetMapiv**	1,1	二维地图顺序	5.1	–
系数	$9 \times 8 * \times R^n$	**GetMapfv**	参见5.1	1d 控制点	5.1	–
系数	$9 \times 8 * \times 8 * \times R^n$	**GetMapfv**	参见5.1	二维控制点	5.1	–
DOMAIN	$9 \times 2 \times R$	**GetMapfv**	参见 5.1	1d域端点	5.1	–
域	$9 \times 4 \times R$	**GetMapfv**	参见 5.1	二维域端点	5.1	–
MAP1 × _	$9 \times B$	**是否启用**	*False*	1d 地图启用：*x* 为地图类型	5.1	评估/启用
MAP2 *x* _	$9 \times B$	**IsEnabled**	*false*	2D地图支持：*x*为地图类型	5.1	评估/启用
MAP1 网格域 _	$2 \times R$	**GetFloatv**	0,1	1d 网格端点	5.1	eval
MAP2 网格域 _	$4 \times R$	**GetFloatv**	0,1;0,1	二维网格端点	5.1	eval
MAP1 网格分段 _	Z^+	**GetFloatv**	1	1维网格划分	5.1	评估
MAP2 网格分段 _	$2 \times Z^+$	**GetFloatv**	1,1	二维网格划分	5.1	eval
自动 正常_	B	**启用**	*false*	自动生成正常值时为真 启用	5.1	eval/启用

表 6.28. 着色器对象状态

获取值	类型	获取命令	初始值	描述	Sec.	属性
着色器类型 _	Z_2	**获取着色器**	-	着色器类型（顶点或片段）	2.15.1	–
删除状态 _	*B*	**获取着色器索引**	*假*	着色器已标记为删除	2.15.1	–
编译状态 _	*B*	**GetShaderiv**	*假*	上次编译成功	2.15.1	–
-	0 + ×字符	**获取着色器信息日志**	空字符串	着色器对象的信息日志	6.1.14	–
信息日志长度 _	Z^+	**获取着色器值**	0	信息日志长度	6.1.14	–
-	0 + ×字符	**获取着色器源代码**	空字符串	着色器的源代码	2.15.1	–
着色器源代码长度 _	Z^+	**GetShaderiv**	0	源代码长度	6.1.14	–

表 6.29. 程序对象状态

获取值	类型	获取命令	初始值	描述	Sec.	属性
当前程序	Z^+	**GetIntegerv**	0	当前程序对象名称	2.15.2	–
删除状态	*B*	**GetProgramiv**	*假*	程序对象已删除	2.15.2	–
链接状态	*B*	**GetProgramiv**	*假*	最后一次链接尝试成功	2.15.2	–
状态验证	*B*	**GetProgramiv**	*假*	上次验证尝试成功	2.15.2	–
附加着色器	Z^+	**GetProgramiv**	0	已附加着色器对象数量	6.1.14	–
-	0 + ×*H*	**获取附加着色器**	空	已附加着色器对象	6.1.14	–
-	0 + ×字符	**获取程序信息日志**	空	程序对象信息日志	6.1.14	–
信息日志长度	Z^+	**GetProgramiv**	0	信息日志长度	2.15.3	–
ACTIVE UNIFORMS	Z^+	**获取程序**	0	活动制服数量	2.15.3	–
-	0 + ×*Z*	**获取制服位置**	–	活动制服的位置	6.1.14	–
-	0 + ×Z^+	**获取活动制服**	–	活动统一大小	2.15.3	–
-	0 + ×Z^+	**获取活动统一体**	–	活动统一类型	2.15.3	–
-	0 + ×字符	**获取活动统一**	空	活动统一名称	2.15.3	–
活动统一最大长度	Z^+	**GetProgramiv**	0	最大活动统一名称长度	6.1.14	–
	512 + ×*R*	**GetUniform**	0	统一值	2.15.3	–
活动属性	Z^+	**GetProgramiv**	0	活动属性数量	2.15.3	–
-	0 + ×*Z*	**获取属性位置**	–	活动通用属性的位置	2.15.3	–
-	0 + ×Z^+	**GetActiveAttrib**	–	活动属性的大小	2.15.3	–
-	0 + ×Z^+	**获取活动属性**	–	活动属性的类型	2.15.3	–
-	0 + ×字符	**GetActiveAttrib**	空	活动属性的名称	2.15.3	–
活动属性最大长度	Z^+	**GetProgramiv**	0	最大活动属性名称长度	6.1.14	–

表 6.30. 顶点着色器状态

获取值	类型	获取命令	初始值	描述	Sec.	属性
顶点程序双面_ _ _	*B*	**是否启用**	*假*	双面彩色模式	2.14.1	启用
当前顶点属性 _ _	16 + ×R4	**获取顶点属性**	0,0,0,1	通用顶点属性	2.7	当前
顶点程序点大小 _ _	*B*	**启用**	*false*	点尺寸模式	3.3	启用

表 6.31. 提示

获取值	类型	获取命令	初始值	描述	Sec.	属性
透视校正提示	Z_3	**GetIntegerv**	不关心	透视校正提示	5.6	提示
点平滑提示	Z_3	**GetIntegerv**	无所谓	点平滑提示	5.6	提示
线平滑提示	Z_3	**GetIntegerv**	不关心	线平滑提示	5.6	提示
多边形平滑提示	Z_3	**GetIntegerv**	不关心	多边形平滑提示	5.6	提示
雾提示	Z_3	**GetIntegerv**	无所谓	雾提示	5.6	提示
生成MIPMAP提示	Z_3	**GetIntegerv**	不关心	生成Mipmap提示	5.6	提示
纹理压缩提示	Z_3	**GetIntegerv**	无所谓	纹理压缩质量提示	5.6	提示
片段着色器导数提示	Z_3	**GetIntegerv**	不关心	片段着色器导数精度提示	5.6	提示

表 6.32. 实现相关的值

获取值	类型	获取命令	最小值	描述	Sec.	属性
最大灯光数	Z^+	**GetIntegerv**	8	最大灯数	2.14.1	–
最大剪切平面数	Z^+	**GetIntegerv**	6	用户剪切最大数量 平面	2.12	–
最大颜色矩阵堆栈深度	Z^+	**GetIntegerv**	2	最大颜色矩阵堆栈深度	3.6.3	–
最大模型视图堆栈深度	Z^+	**GetIntegerv**	32	最大模型视图堆栈深度	2.11.2	–
最大投影堆栈深度	Z^+	**GetIntegerv**	2	最大投影矩阵堆栈深度 深度	2.11.2	–
最大纹理堆栈深度	Z^+	**GetIntegerv**	2	纹理最大层数 矩阵堆栈	2.11.2	–
亚像素位数	Z^+	**GetIntegerv**	4	子像素位数 在屏幕x_w 和y_w轴上的精度	3	–
最大3D纹理尺寸	Z^+	**GetIntegerv**	16	最大3D纹理图像 尺寸	3.8.1	–
最大纹理尺寸	Z^+	**GetIntegerv**	64	最大2D/1D纹理图像 尺寸	3.8.1	–
最大纹理LOD偏移量	R^+	**GetFloatv**	2.0	纹理级距的最大绝对值 细节偏移量	3.8.8	–
最大立方体贴图纹理尺寸	Z^+	**GetIntegerv**	16	最大立方体贴图纹理图像 尺寸	3.8.1	–
最大像素贴图表	Z^+	**GetIntegerv**	32	**像素映射**表的最大尺寸 转换表	3.6.3	–
最大名称堆栈深度	Z^+	**GetIntegerv**	64	最大选择名称堆栈深度 深度	5.2	–
最大列表嵌套层级	Z^+	**GetIntegerv**	64	最大显示列表调用嵌套层级	5.4	–
最大评估顺序	Z^+	**GetIntegerv**	8	最大评估多项式 阶	5.1	–
最大视口尺寸	$2 \times Z^+$	**GetIntegerv**	参见2.11.1	最大视口尺寸	2.11.1	–

表 6.33：实现相关的值（续）

获取值	类型	获取命令	最小值	描述	Sec.	属性
属性堆栈最大深度 _ _	Z^+	**GetIntegerv**	16	服务器属性堆栈的最大深度 服务器属性堆栈	6	–
最大客户端属性堆栈深度 _ _	Z^+	**GetIntegerv**	16	客户端属性的最大堆栈深度 客户端属性堆栈深度	6	–
–	$3 \times Z^+$	-	32	颜色最大尺寸 表	3.6.3	–
–	Z^+	-	32	最大尺寸的 直方图表最大尺寸	3.6.3	–
辅助缓冲区	Z^+	**GetIntegerv**	0	辅助缓冲区数量 缓冲区	4.2.1	–
RGBA模式	B	**获取布尔值v**	–	若颜色缓冲区存储RGBA数据则返回 True rgba	2.7	–
索引模式 _	B	**GetBooleanv**	–	若颜色缓冲区存储 索引值则返回真	2.7	–
双缓冲	B	**获取布尔值v**	–	若前端与后端 缓冲区存在	4.2.1	–
立体声	B	**获取布尔值v**	–	若左、右缓冲区 存在	6	–
别名点尺寸范围 _ _	$2 \times R^+$	**获取浮点值**	1,1	别名字体的字号范围（最小至最大） 点尺寸	3.3	–
平滑点尺寸范围 _ _ (v1.1: 点尺寸范围) _ _	$2 \times R^+$	**GetFloatv**	1,1	范围（低至高） 抗锯齿点尺寸	3.3	–
平滑点尺寸粒度 _ _ (v1.1: 点尺寸粒度) _ _	R^+	**GetFloatv**	–	抗锯齿点尺寸 粒度	3.3	–
锯齿线宽范围 _ _ _	$2 \times R^+$	**获取浮点值**	1,1	混叠线宽范围（低至高） 线宽	3.4	–
平滑线宽范围 _ _ _ (v1.1: 线宽范围) _ _	$2 \times R^+$	**获取浮点值v**	1,1	范围（低至高） 抗锯齿线宽	3.4	–
平滑线宽粒度 _ _ _ (v1.1: 线宽粒度) _ _	R^+	**GetFloatv**	–	抗锯齿线宽 粒度	3.4	–

表 6.34. 实现相关值（续）

获取值	类型	获取命令	最小值	描述	Sec.	属性
最大卷积宽度 -	$3 \times Z^{+}$	**获取卷积-Parameteriv**	3	最大卷积宽度 卷积滤波器	4.3	–
最大卷积高度 -	$2 \times Z^{+}$	**获取卷积-Parameteriv**	3	卷积滤波器的最大高度 卷积滤波器	4.3	–
最大元素索引 -	Z^{+}	**GetIntegerv**	–	推荐 **DrawRangeElement**索引的最大数量	2.8	–
顶点最大元素数 -	Z^{+}	**GetIntegerv**	–	推荐 **DrawRangeElement**顶点最大数量	2.8	–
采样缓冲区 -	Z^{+}	**GetIntegerv**	0	多采样数 缓冲区	3.2.1	–
采样	Z^{+}	**GetIntegerv**	0	覆盖掩码尺寸	3.2.1	–
压缩纹理格式 - -	$0 \times Z$	**GetIntegerv**	-	枚举压缩 纹理格式	3.8.3	–
NUM 压缩纹理格式 - -	Z	**GetIntegerv**	0	枚举的压缩纹理格式数量 压缩纹理格式	3.8.3	–
查询计数位 - -	Z^{+}	**GetQueryiv**	参见 6.1.12	遮挡查询计数器 位	6.1.12	–

表 6.35. 实现相关的值（续）

获取值	类型	获取命令	最小值	描述	Sec.	属性
扩展	S	**GetString**	–	支持的扩展	6.1.11	–
渲染器	S	**GetString**	–	渲染器字符串	6.1.11	–
着色语言版本 _ _	S	**GetString**	–	着色语言 支持版本	6.1.11	–
供应商	S	**GetString**	–	供应商字符串	6.1.11	–
版本	S	**获取字符串**	–	OpenGL版本 支持	6.1.11	–
最大纹理单元数 _	Z^+	**GetIntegerv**	2	纹理单元数量 固定功能纹理单元数量	2.6	–
顶点属性最大值 _	Z^+	**GetIntegerv**	16	活动顶点属性数量 属性	2.7	–
顶点统一组件最大值 _ _	Z^+	**GetIntegerv**	512	顶点着色器中 顶点着色器统一变量	2.15.3	–
最大变化浮点数 _	Z^+	**GetIntegerv**	32	浮点变量数量 浮点变量数量	2.15.3	–
最大组合纹理图像单元 _ _ _	Z^+	**GetIntegerv**	2	纹理单元总数 可由GL访问的纹理单元总数	2.15.4	–
顶点纹理图像单元最大值 _ _	Z^+	**GetIntegerv**	0	顶点着色器可访问的纹理图像数量 顶点着色器可访问的纹理图像 单元数	2.15.4	–
最大纹理图像单元数 _ _	Z^+	**GetIntegerv**	2	可通过片段处理访问的纹理图像 可供片段处理访问的纹理图像单元数	2.15.4	–
最大纹理坐标 _	Z^+	**GetIntegerv**	2	纹理数量 坐标系	2.7	–
最大片段统一分量 _ _	Z^+	**GetIntegerv**	64	单词数量用于 frag. 着色器统一变量	3.11.1	–
最大绘制缓冲区 _	Z^+	**GetIntegerv**	1+	最大活动绘图缓冲区数 活动绘制缓冲区数量	4.2.1	–

表 6.36. 实现相关的像素深度

获取值	类型	获取命令	初始值	描述	Sec.	属性
x 位	Z^+	**GetIntegerv**	-	x颜色缓冲区中的位数 *x* 为 RED、 GREEN、BLUE、ALPHA 或 INDEX	4	–
深度位 _	Z^+	**GetIntegerv**	-	深度缓冲区平面数	4	–
模板位 _	Z^+	**GetIntegerv**	-	模板平面数量	4	–
累加器 × 位数 _	Z^+	**GetIntegerv**	-	*x*累加器中的位数 缓冲区组件（*x*为红色、 绿色、蓝色或透明度	4	–

表 6.37. 杂项

获取值	类型	获取命令	初始值	描述	Sec.	属性
列表基数	Z^+	**GetIntegerv**	0	**列表**基准设置	5.4	list
列表索引	Z^+	**GetIntegerv**	0	显示列表的数量 ；若无则为 0	5.4	–
列表模式	Z^+	**GetIntegerv**	0	在 ；若未构建则未定义	5.4	–
–	16 * ×A	–	空	服务器属性堆栈	6	–
属性堆栈深度 _	Z^+	**GetIntegerv**	0	服务器属性堆栈指针	6	–
–	16 * ×A	–	空	客户端属性堆栈	6	–
_ _ 客户端属性堆栈深度	Z^+	**GetIntegerv**	0	客户端属性堆栈指针	6	–
名称 堆栈深度 _	Z^+	**GetIntegerv**	0	名称堆栈深度	5.2	–
渲染模式 _	Z_3	**GetIntegerv**	渲染	**渲染模式**设置	5.2	–
_ 选择缓冲区指针	*Y*	**GetPointerv**	0	选择缓冲区指针	5.2	选择
选择缓冲区大小 _ _	Z^+	**GetIntegerv**	0	选择缓冲区大小	5.2	选择
_ 反馈缓冲区指针	*Y*	**GetPointerv**	0	反馈缓冲区指针	5.3	反馈
反馈缓冲区大小 _ _	Z^+	**GetIntegerv**	0	反馈缓冲区大小	5.3	反馈
反馈缓冲区类型 _ _	Z_5	**GetIntegerv**	2D	反馈类型	5.3	反馈
–	$n \times Z_8$	**获取错误**	0	当前错误代码	2.5	–
–	$n \times B$	–	*假*	存在对应项时为真 错误	2.5	–
	B	–	*假*	闭塞查询激活	4.1.7	–
当前查询 _	Z^+	**获取查询iv**	0	活动遮挡查询ID	4.1.7	–
	Z^+	–	0	闭合采样通过计数	4.1.7	–

附录A

不变性

OpenGL规范并非像素级精确，因此无法保证不同GL实现生成的图像完全一致。然而，规范在某些情况下确实规定了同一实现生成的图像必须完全匹配。本附录旨在识别并阐明需要精确匹配的场景及其依据。

A.1 可重复性

最明显且最基础的情况是重复执行一系列GL命令。对于任何给定的GL和帧缓冲状态*向量*，以及任何GL命令，当在初始GL和帧缓冲状态下执行该命令时，生成的GL和帧缓冲状态必须完全一致。

可重复性的首要目标是避免双缓冲场景重绘时的视觉伪像。若渲染不可重复，切换两个采用相同命令序列渲染的缓冲区可能导致图像出现可见变化。此类虚假运动会干扰观者体验。可重复性的另一重要意义在于提升可测试性。

可重复性虽重要，却是个弱要求。若仅以可重复性为标准，两个场景中仅改变一个（小）多边形位置的渲染结果，可能每个像素都存在差异。这种差异虽符合可重复性法则，却显然违背其精神。为确保有效运作，需补充不变性规则。

A.2 多通道算法

不变性是实现整套实用多通道算法的基础。此类算法通过多次渲染（每次采用不同的GL模式向量）最终在帧缓冲区生成结果，典型算法包括：

- 通过重新绘制原始图形（采用不同颜色或异或逻辑运算）实现从帧缓冲区"擦除"
- ..

另一方面，不变性规则会大幅增加GL高性能实现的复杂度。即使是弱可重复性要求，也会显著限制GL的并行实现。由于GL实现必须支持全部功能（而非仅实现便捷子集），采用硬件加速的实现方案需根据当前GL模式向量在硬件与软件模块间切换。 强不变性要求迫使硬件与软件模块的行为完全一致，这可能极难实现（例如当硬件与软件采用不同精度进行浮点运算时）。理想方案是达成折中，既能产生众多兼容的高性能实现方案，又能促使众多软件供应商选择移植至OpenGL.

A.3 不变性规则

对于给定的OpenGL渲染上下文实例化：

规则1 *对于任何给定的GL和帧缓冲状态向量，以及任何给定的GL命令，每次在初始GL和帧缓冲状态上执行该命令时，生成的GL和帧缓冲状态必须完全相同。*

规则2： *对下列状态值的修改不产生副作用（即修改不会影响其他状态值的使用）：*

必需项：

- *帧缓冲区内容（所有位平面）*
- *启用写入的颜色缓冲区*
- *栈顶矩阵以外的矩阵值*

- 剪裁参数（启用状态除外）
- 写入掩码（颜色、索引、深度、模板）
- 清除值（颜色、索引、深度、模板、累积）
- 当前值（颜色、索引、法线、纹理坐标、边缘标志）
- 当前光栅颜色、索引和纹理坐标。
- 材质属性（环境光、漫反射、镜面反射、发光、光泽度）

强烈建议：

- 矩阵模式
- 矩阵堆栈深度
- Alpha测试参数（除启用状态外）
- 模板参数（启用除外）
- 深度测试参数（启用除外）
- 混合参数（启用除外）
- 逻辑运算参数（启用除外）
- 像素存储与传输状态
- 评估器状态（除非其影响评估器生成的顶点数据）
- 多边形偏移参数（除启用状态外，且不影响片段的深度值）

推论 1 片段生成过程对规则 2 中标记为 • 的状态值保持不变。

推论2 生成的片段窗口坐标（x, y, z）同样不依赖于

必备条件：

- 当前值（颜色、颜色索引、法线、纹理坐标、边缘标记）
- 当前光栅颜色、颜色索引及纹理坐标
- 材质属性（环境光、漫反射、镜面反射、发光、光泽度）

规则3 每个片段操作的算术运算保持不变，但直接控制该操作的参数除外（例如控制透明测试的参数包括透明测试启用状态、透明测试函数和透明测试参考值）。

推论3： *通过相同命令序列（无论同时或分别）渲染至共享同一帧缓冲区的不同颜色缓冲区中的图像，其像素完全一致。*

规则4 *当使用相同输入多次运行时，相同的顶点着色器或片段着色器将产生相同的结果。术语"相同的着色器"指的是由相同源字符串构成的程序对象，该对象经过编译并可能多次链接，随后使用相同的GL状态向量执行。*

规则5 *所有将*`gl FragCoord.z`*条件或无条件赋值给*`gl FragDepth`*的片段着色器，在实际执行*`gl FragDepth`*赋值的片段中，彼此之间具有深度不变性。*

A.4 以上规则的综合含义

硬件加速的GL实现遇到某些GL状态向量时，应默认切换至软件操作模式。即使是弱可重复性要求也意味着，例如OpenGL实现不能在此切换中应用迟滞效应，而必须确保给定模式向量能保证后续命令*始终*在硬件或软件机器中执行。

更严格的不变性规则限制了从硬件渲染切换到软件渲染的时机，因为软件渲染器和硬件渲染器无法实现像素级精确匹配。例如，当启用或禁用混合功能时可以进行切换，但当修改混合参数时则不应进行切换。

由于浮点数在不同渲染器（硬件和软件）中可能采用不同格式表示，当切换渲染器时，许多OpenGL状态值可能会发生细微变化。规则1正是旨在避免此类状态值的改变。

附录B

推论

以下观察结论源自规范正文及其他附录。未列入本列表的观察结论并不影响其真实性。

1. 当前光栅纹理坐标必须始终保持正确，包括纹理映射未启用期间以及图形渲染器处于颜色索引模式时。
2. 在反馈模式下请求返回的纹理坐标始终有效，包括纹理映射未启用期间以及图形渲染器处于颜色索引模式时。
3. 向上兼容的OpenGL版本其错误语义可能发生变更。除此之外，向上兼容版本仅允许新增功能。
4. GL查询命令无需满足Flush或Finish命令的语义要求。唯一要求是查询到的状态必须与先前所有已执行GL命令的完整执行结果保持一致。
5. 应用程序指定的点尺寸和线宽在查询时必须按指定值返回。实现相关的截断仅在使用期间影响这些值。
6. 位图和像素传输不会触发选择命中。
7. **作为StencilFunc第三**参数指定的掩码会影响模板比较函数的操作数，但不直接影响模板缓冲区的更新。**由StencilMask**指定的掩码对模板比较函数无效，其作用仅限于限制模板缓冲区更新的效果。

8. 多边形着色在多边形模式解析之前完成。若着色模型为FLAT，则单个多边形生成的所有点或线将具有相同颜色。

9. 显示列表本质上是一组命令和参数，因此列表中命令产生的错误必须在执行时触发。若在编译模式下创建列表，则创建过程中不应产生错误。

10. 在RGBA模式的GL上下文中，**RasterPos**不会改变当前光栅索引的默认值。同样地，在颜色索引的GL上下文中，**RasterPos**也不会改变当前光栅颜色的默认值。然而，无论GL上下文的颜色模式如何，当前光栅索引和当前光栅颜色均可被查询。

11. 通过 **ColorMaterial** 关联到当前颜色的材质属性始终采用当前颜色的值。尝试通过 **Material** 调用修改该材质属性将无效。

12. 即使在颜色索引上下文中，**Material**和**ColorMaterial**仍可用于修改RGBA材质属性；反之，即使在RGBA上下文中，**Material**也可用于修改颜色索引材质属性。

13. OpenGL渲染命令不存在原子性要求，即使在片段级别亦然。

14. 由于非抗锯齿多边形的光栅化采用点采样方式，在填充模式下渲染无面积的多边形时不会生成片段；而"窄"多边形光栅化生成的片段可能无法形成连续数组。

15. OpenGL 并未强制要求其任何坐标系采用左旋或右旋规则。但需考虑以下条件：(1) 物体坐标系为右旋；(2) 仅**使用缩放**（仅限正缩放值）、**旋转和平移命令**操作模型视图矩阵；(3) 仅使用 **Frustum** 或 **Ortho** 其中一种模式设置投影矩阵；(4) **DepthRange** 的近值小于远值。若同时满足上述所有条件，则视点坐标系为右手系，而裁剪坐标系、归一化设备坐标系及窗口坐标系均为左手系。

16. ColorMaterial对色标照明无影响。

17. （无像素丢失或重复。） 假设两个多边形共享一条完全相同的边（即：存在多边形A的边上顶点A和B，以及多边形B的边上顶点C和D，且顶点A（或B）的坐标与顶点C（或D）完全一致，当同时指定A、B、C、D时，其坐标变换状态完全相同）。 此时，当两多边形光栅化生成的片元合并时，每个与共享边内部相交的片元仅生成一次。

18. OpenGL状态在FEEDBACK模式和SELECT模式下持续更新
模式下。帧缓冲区内容不会被修改。

19. 当前光栅位置、用户定义的裁剪平面、光点方向、*LIGHTi*的光源位置以及纹理生成器的视平面在指定时会进行变换。在**PopAt-trib操作**期间或复制上下文时，这些元素不会发生变换。

20. 不同组件可能采用不同的抖动算法。特别是，Alpha通道可能与红、绿、蓝通道采用不同的抖动方式，某些实现甚至可能完全不执行Alpha通道的抖动。

21. 对于任何GL状态与帧缓冲状态组合，以及任意GL命令参数组合，无论通过常规执行还是显示列表执行，最终生成的GL状态与帧缓冲状态均完全一致。

附录 C

版本 1.1

OpenGL 1.1版本是自1992年7月1日发布原始1.0版本后的首次修订。该版本向上兼容1.0版本，意味着任何能在1.0 GL实现环境下运行的程序，在1.1 GL实现环境中亦可直接运行。 本次更新主要增强了纹理映射功能，同时扩展了几何体与片段操作能力。以下简要说明新增特性：

C.1 顶点数组

顶点数据数组的传输命令数量大幅减少。系统定义了六种数组：分别存储顶点位置、法线坐标、颜色、颜色索引、纹理坐标及边界标志。这些数组可独立指定并启用，亦可通过单条命令选择预定义配置方案。

主要目标是减少将非显示列表几何数据传输至图形库所需的子程序调用次数。 次要目标是提升传输效率，特别是允许使用直接内存访问（DMA）硬件实现传输。新增功能与EXT顶点数组扩展基本一致，但不支持静态数组数据（因其复杂化接口且未被使用），并添加了预定义配置（既进一步减少子程序数量，又可实现数组数据的高效传输）。

C.2 多边形偏移

多边形光栅化生成的片段深度值可根据多边形窗口坐标深度斜率的仿射函数向原点方向偏移或远离原点偏移。偏移后的深度值可实现共面几何体（尤其是面轮廓）的深度缓冲区伪影消除渲染，未来还可应用于阴影生成算法。

新增功能与EXT多边形偏移扩展功能一致，但存在两处例外。首先，偏移量需分别针对点、线和填充光栅化模式单独启用，且三者共享同一仿射函数定义。（通过偏移轮廓片段而非填充片段的深度值，可确保深度缓冲区内容正确保存。）其次，偏移量偏移值以深度缓冲区分辨率为单位指定，而非[0,1]深度范围内的数值。

C.3 逻辑运算

RGBA渲染生成的片段可通过逻辑运算合并至帧缓冲区，其机制与GL 1.0版本中颜色索引片段的处理方式相同。此类操作期间将禁用混合功能，因其需求罕见、多数系统无法支持，且需与EXT混合逻辑运算扩展的语义保持一致——本扩展功能即基于该扩展进行松散设计。

C.4 纹理图像格式

存储的纹理数组具有一种称为*内部格式的结构*，而非简单的组件计数。内部格式以单一枚举值表示，既指示图像数据的组织方式（如LUMINANCE、RGB等），也规定了每个图像组件的存储位数。客户端可通过内部格式规范指定所需的纹理图像存储精度。新增*的基础格式*ALPHA和INTENSITY提供了新的纹理环境操作。这些新增功能与EXT纹理扩展子集的功能相匹配。

C.5 纹理替换环境

纹理映射的常见用途是将生成的片段颜色值替换为纹理颜色数据。在GL 1.0版本中，此功能仅能通过间接方式实现——要求客户端指定"白色"几何体进行调制。

通过纹理实现。GL 1.1 版本允许显式指定此类替换，可能提升性能。这些新增功能与EXT纹理扩展子集中的功能相匹配。

C.6 纹理代理

纹理代理机制允许GL实现根据其他纹理参数（尤其是内部图像格式）动态声明不同的最大纹理图像尺寸。 客户端可通过代理查询机制在运行时动态调整纹理资源使用策略。该代理接口的设计无需在GL接口中新增例程即可实现此类查询。这些新增功能与EXT纹理扩展子集的实现相匹配，但实现方需返回与完整Mipmap数组支持一致的分配信息。

C.7 复制纹理与子纹理

纹理数组数据既可从帧缓冲区内存指定，也可从客户端内存指定；纹理数组的矩形子区域既可从客户端内存重新定义，也可从帧缓冲区内存重新定义。这些新增功能与EXT copy texture和EXT subtexture扩展定义的功能相匹配。

C.8 纹理对象

一组纹理数组及其相关纹理状态可被视为单一对象。此处理方式在使用多个数组时能提升实现效率。结合子纹理功能，它还允许客户端对现有纹理数组进行渐进式修改，而非完全重新定义。这些新增功能与EXT纹理对象扩展一致，并在纹理驻留语义方面略有补充。

C.9 其他变更

1. 颜色索引现可指定为无符号字节。
2. 在点、像素矩形和位图的光栅化过程中，纹理坐标s、*t和r将被*q除。此除法操作在1.0版本中仅针对线条和多边形进行了文档说明。

3. 线条光栅化算法已调整，确保像素边界处的垂直线能正确光栅化。

4. 原分散于第3章与第4章的像素传输说明已整合至第3章统一阐述。

5. 若纹理数组中不存在Alpha通道，则纹理Alpha值将返回为1.0。此行为在1.0版本中未作规定，且参考手册中存在错误说明。

6. 雾效起始值与终止值现允许取负数。

7. 当启用**Col-orMaterial时**，评估的颜色值将指导光照方程的计算。

C.10 鸣谢

OpenGL 1.1 是众多人士贡献的结晶，代表了计算机行业的横截面。以下是部分贡献者名单，包括其贡献时所代表的公司：

库尔特·阿克利，硅图公司

Bill Armstrong， Evans & Sutherland公司Andy Bigos

，3Dlabs公司

帕特·布朗，IBM

吉姆·科布，埃文斯-萨瑟兰公司迪克·库尔特，

数字设备公司

布鲁斯·达莫拉，通用电气医疗系统约翰·丹尼斯，数字设备

公司

弗雷德·费舍尔，Accel Graphics克里斯·弗雷泽，

硅图公司托德·弗雷泽，埃文斯与萨瑟兰公司蒂姆

·弗里斯，NCD公司

肯·加内特，NCD

迈克·赫克，模板图形软件公司戴夫·希金斯，IBM

菲尔·赫胥黎，3Dlabs

戴尔·柯克兰，英特格拉夫公司

凯文·勒费布尔，惠普公司吉姆·米勒，IBM

蒂姆·米斯纳，SunSoft

杰里米·莫里斯，3Dlabs以色列·平
卡斯，英特尔比马尔·波达尔，
IBM
莱尔·拉姆肖，数字设备公司兰迪·罗斯特，惠普公
司
约翰·辛普夫，硅图公司马克·西格尔，硅图公
司伊戈尔·西尼亚克，英特尔
杰夫·史蒂文森，惠普公司；比尔·斯威尼，
SunSoft公司
凯尔文·汤普森，便携图形公司尼尔·特雷维特，
3Dlabs公司
利纳斯·维普斯塔斯，IBM
安迪·维斯珀，数字设备公司亨利·沃伦，梅格泰
克公司
宝拉·沃马克，硅图公司梅森·吴，硅图公司史蒂夫·
赖特，微软公司

附录 D

版本 1.2

OpenGL 1.2版于1998年3月16日发布，是自原始1.0版后的第二次修订。1.2版向上兼容1.1版，这意味着任何能在1.1版GL实现环境下运行的程序，在1.2版GL实现环境中亦可直接运行。

GL进行了若干功能扩展，尤其在纹理映射能力和像素处理管道方面。以下是对各项新增功能的简要说明。

D.1 三维纹理化

现可定义并使用三维纹理。同时定义了三维图像的内存格式及其对应的像素存储模式，这些新增功能与EXT texture3D扩展完全兼容。

三维纹理的重要应用之一是渲染体积图像数据。

D.2 BGRA像素格式

BGRA扩展了主机内存颜色格式列表。具体而言，它提供了与Windows平台常见文件和帧缓冲格式相匹配的分量顺序。这些新增功能与EXT bgra扩展完全一致。

D.3 压缩像素格式

主机内存中的压缩像素完全由一个无符号字节、一个无符号短整型或一个无符号整型表示。包含压缩像素的字段并非标准机器类型，但像素整体符合规范。因此像素存储模式及其解压缩对应模式均能正确处理压缩像素。

新增功能与EXT压缩像素扩展一致，并额外支持反向分量顺序的压缩格式。

D.4 法线重缩放

法线可通过模型视图矩阵推导出的常数因子进行缩放。在多数情况下，缩放操作比重新规范化更高效，且能生成相同单位长度的法线。

新增功能基于EXT法线重缩放扩展。 _ _

D.5 独立镜面反射颜色

光照计算经过修改，产生由发光项、环境项和漫反射项组成的主色（采用常规GL光照方程），以及由镜面反射项组成的次色。仅主色受纹理环境影响；次色则叠加至纹理处理结果，最终生成单一后处理纹理颜色。此机制使高光颜色基于生成光源而非表面属性。

此扩展功能与EXT独立镜面反射颜色扩展功能相匹配。 _ _ _

D.6 纹理坐标边缘裁剪

GL通常采用钳位处理，使纹理坐标精确限制在[0, 1]范围内。当使用此算法钳位纹理坐标时，纹理采样滤波器会跨过纹理图像的边缘，其中一半采样值来自纹理图像内部，另一半来自纹理边界。有时需要在不引入边界且不使用固定边界颜色的情况下对纹理进行钳位。

新型纹理裁剪算法 CLAMP TO EDGE 可在所有米普级上对纹理坐标进行裁剪，确保纹理过滤器永远不会采样边界纹素。裁剪时返回的颜色仅取自纹理图像边缘的纹素。

新增内容与SGIS纹理边缘裁剪扩展完全一致。 _ _ _

D.7 纹理细节级别控制

新增两项与纹理细节级别*参数*λ相关的约束。一项约束*将*λ限制在指定浮点数范围内，另一项则将Mipmap图像数组的选择范围限定为原本可选数组的子集。

这些约束共同实现以下效果：初始加载大型纹理时采用低分辨率，并在需要更高分辨率或资源可用时逐步提升分辨率。图像数组规格必须是整数而非连续值。通过为λ参数提供独立的连续钳位机制，可在提供更高分辨率图像时避免"画面跳跃"的视觉瑕疵。

新增功能与SGIS纹理LOD扩展完全对应。 _ _

D.8 顶点数组绘制元素范围

新增**了一种DrawElements**形式，该形式明确提供了索引集所引用的顶点范围信息。实现方案可利用此附加信息处理顶点数据，而无需扫描索引数据来确定被引用的顶点。

新增内容与EXT绘制范围元素扩展功能一致。 _ _ _

D.9 成像子集

其余新增特性主要面向高级图像处理应用，并非所有GL实现都包含这些功能。这些特性统称为*成像子集*。

D.9.1 颜色表

在像素传输过程中定义了一种新的RGBA格式颜色查找机制，该机制在现有查找功能之外提供了额外的查找能力。其关键区别在于：新查找表被视为具有内部格式的单维图像，类似于纹理图像和卷积滤波器图像。因此，新查找表可对通过像素组的子集进行操作。 例如，采用ALPHA内部格式的查找表仅修改每个像素组的A分量，而R、G、B分量保持不变。

可执行三项独立查找操作：卷积前；卷积后且色彩矩阵转换前；色彩矩阵转换后且管道统计数据收集前。

除标准内存源机制外，还提供了从帧缓冲区初始化颜色查找表的方法。

可重新定义颜色查找表的部分内容而无需重新初始化整个表。受影响的部分可从主机内存或帧缓冲区指定。

扩展 匹配 那些 的 扩展 以及 _ _ EXT颜色子表扩展。

D.9.2 卷积

在像素传输过程中，首次颜色表查找之后将执行一维或二维卷积运算。卷积核本身被视为一维或二维图像，可从应用程序内存或帧缓冲区加载。

卷积框架设计支持三维卷积，但相关API留待未来扩展。

扩展 匹配 那些 来自 的 EXT卷积 以及 _ HP卷积边界模式扩展。 _ _

D.9.3 色彩矩阵

在像素传输路径中添加了4x4矩阵变换及相关矩阵堆栈。该矩阵作用于RGBA像素组，使用方程

$$C' = MC,$$

其中

$$C = \begin{bmatrix} R \\ G \\ B \\ A \end{bmatrix}$$

其中 M 是位于颜色矩阵堆栈顶部的 4× ×4 矩阵。矩阵乘法后，每个生成的颜色分量都会按预设量进行缩放和偏移。颜色矩阵乘法遵循卷积运算。

该色彩矩阵可用于重新分配与复制色彩分量，亦可实现基础色彩空间转换。

新增功能与SGI色彩矩阵扩展功能一致。 _ _

D.9.4 像素管道统计

在像素传输管道末端执行以下操作：统计特定颜色分量值的出现次数（直方图）以及追踪颜色分量的最小值与最大值（最小-最大）。可选模式允许在完成直方图和/或最小-最大操作后丢弃像素数据。否则像素数据将不受影响地继续传递至下一操作。

新增功能与EXT直方图扩展功能一致。 _

D.9.5 常量混合颜色

可定义一个用于确定混合权重系数的常量颜色。典型应用场景是混合两张RGB图像。若不使用常量混合系数，则其中一张图像必须具备Alpha通道，且每个像素点需设置为所需的混合系数值。

新增功能与EXT混合颜色扩展功能一致。 _ _

D.9.6 新型混合运算式

除常规源目标组件加权求和外，还可采用其他混合运算式。

其中两项新方程可生成源图像与目标图像颜色分量的最小值（或最大值）。取最大值在医学成像中的最大投影等应用场景中尤为实用。

另外两个方程与默认混合方程类似，但生成的是其左右两侧的差值而非和。图像差值在许多图像处理应用中很有用。

新增项 匹配 那些 的 EXT blend minmax 和 _ _ EXT blend 减去扩展。

D.10 鸣谢

OpenGL 1.2 是众多人士贡献的结晶，代表了计算机行业的横截面。以下是部分贡献者名单，包括其贡献时所代表的公司：

库尔特·阿克利，硅图公司

Bill Armstrong，Evans & SutherlandOtto Berkes，微软

皮埃尔-吕克·比萨永，Matrox Graphics公司德鲁·布利斯

，微软公司

David Blythe，硅图公司Jon Brewster，惠普公司Dan Brokenshire，IBM公司帕特·布朗，IBM公司牛顿·张，S3公司比尔·克利福德，数字设备公司吉姆·科布，帕拉梅特克公司；布鲁斯·达莫拉，IBM凯文·达拉斯，微软公司 马赫什·丹达帕尼，Rendition公司 丹尼尔·道姆，AccelGraphics公司 苏西·德菲斯，IBM公司彼得·多伊尔，英特尔公司杰伊·杜卢克，雷瑟公司克雷格·邓伍迪，硅图公司；戴夫·厄布，IBM弗雷德·费舍尔，AccelGraphics/动态图片公司艾伦·加洛塔，ATI公司；肯·加内特，NCD公司迈克尔·戈尔德，英伟达/硅图公司克雷格·格罗谢尔，Metro Link扬·哈登伯格，三菱电机迈克·赫克，模板图形软件公司迪克·赫塞尔，雷瑟图形公司保罗·何，硅图公司肖恩·霍普伍德，硅图公司吉姆·赫尔利，英特尔公司菲尔·赫胥黎，3Dlabs迪克·杰伊，模板图形软件公司保罗·詹森，3Dfx公司布雷特·约翰逊，惠普公司迈克尔·琼斯，硅图公司蒂姆·凯利，Real3D公司乔恩·卡赞，英特尔戴尔·柯克兰，英特尔克里斯·基特里克，雷瑟唐·郭，S3赫伯·库塔，Quantum 3D

菲尔·拉克鲁特，硅图公司 普拉卡什·拉迪亚，
S3公司
乔恩·利奇，硅图公司凯文·勒菲弗，惠普公司大卫
·利贡，雷瑟图形公司肯特·林，S3公司
丹·麦凯布，S3公司；杰克·米
德尔顿，太阳微系统公司；
蒂姆·米斯纳，英特尔公司
比尔·米切尔，美国国家标准研究院
吉恩·蒙斯，英特尔威廉·纽霍尔，
Real3D
马修·帕帕基波斯，英伟达/雷瑟加里·帕克西诺斯，地
铁链接汉斯彼得·菲斯特，三菱电机
理查德·皮门特尔，参数技术公司比马尔·波达尔，IBM/英特
尔
罗布·普特尼，IBM公司迈克·昆
兰，Real3D公司
内特·罗宾斯，犹他大学德特勒夫·罗特格，艾
尔莎
兰迪·罗斯特，惠普公司凯文·拉什福斯，
太阳微系统公司理查德·S·赖特，Real3D
公司李学山，微软公司
约翰·辛普夫，硅图公司 斯特凡·西博斯，艾尔
莎公司
马克·西格尔，硅图公司；鲍勃·塞辛格，
S3公司
肖敏志，S3公司；科林·夏普，
Rendition公司；伊戈尔·西尼亚克
，英特尔公司
比尔·斯威尼，Sun威廉·斯威尼，
Sun内森·塔克，Raycer道格·特威
伦格，Sun约翰·泰恩菲尔德，
3dfx
卡蒂克·文卡塔拉曼，英特尔 安迪·维斯珀，数
字设备

亨利·沃伦，数字设备公司/梅格泰克 保拉·沃马克，硅图公司

史蒂夫·赖特，微软 大卫·于，硅图公司 兰

迪·赵，S3公司

附录 E

版本 1.2.1

OpenGL 1.2.1 版本于 1998 年 10 月 14 日发布，引入了 ARB 扩展（参见附录 J）。该版本中定义的唯一 ARB 扩展是多纹理（multitexture），允许在单次渲染过程中将多个纹理应用于片段。多纹理扩展基于 `SGIS` 多纹理扩展，通过移除将纹理坐标集路由至任意纹理单元的功能得以简化。

1999年4月1日，附录B新增一条讨论显示列表与即时模式不变性的推论。

附录F

版本 1.3

OpenGL 1.3版本于2001年8月14日发布，是自原始1.0版本以来的第三次修订。该版本向上兼容早期版本，意味着任何能在1.2、1.1或1.0 GL实现中运行的程序，在1.3 GL实现中亦可无修改运行。

本次更新主要增强了纹理映射功能（此前由ARB扩展定义），具体新增特性如下：

F.1 压缩纹理

压缩纹理图像可降低纹理内存占用，并提升带纹理基元的渲染性能。GL提供了一个框架，用于构建支持特定压缩图像格式的扩展，同时提供一组通用压缩内部格式，允许应用程序指定纹理图像应以压缩形式存储，而无需为特定压缩格式编写代码（如S3TC或FXT1等特定压缩格式可通过扩展实现）。

纹理 压缩 被 推广 从 GL ARB纹理压缩扩展 GL ARB纹理压缩扩展。

F2 立方体贴图

立方体贴图提供了一种新的纹理生成方案，通过六张二维图像（代表立方体的六个面）来查找纹理。纹理坐标(*str*)被视为从立方体中心发出的方向向量。在纹理生成时，通过插值得到的每个片段(*str*)选择

基于最大模数坐标（主轴）生成立方体单面二维图像。通过将另外两个坐标（次轴值）除以主轴值计算出新的(*st*)值，该值用于在立方体贴图中选定二维纹理图像面进行查找。

新增两种纹理坐标生成模式，可配合立方贴图纹理化使用。反射贴图模式生成与顶点视点空间反射向量匹配的纹理坐标(*str*)，适用于环境贴图，可避免球面贴图固有的奇异性问题。法线贴图模式生成与顶点变换后视点空间法线匹配的纹理坐标，适用于基于纹理的漫反射光照模型。

立方贴图功能已从GL ARB纹理立方贴图扩展中正式纳入标准。

F3 多采样

多采样提供了一种抗锯齿机制，在每个像素点对所有基元进行多次采样。每次像素更新时，颜色采样值都会被解析为单一可显示颜色，因此抗锯齿在应用程序层面上看似自动完成。由于每次采样都包含深度和模板信息，深度与模板功能的执行效果等同于单采样模式。

当支持多采样时，帧缓冲区会新增一个称为多采样缓冲区的额外缓冲区。像素采样值（包括颜色、深度和模板值）均存储于此缓冲区中。

多采样通常是耗费资源的操作，因此并非所有上下文都支持该功能。应用程序必须通过GLX 1.4提供的全新接口或WGL ARB多采样扩展获取支持多采样的上下文。该功能由GL ARB多采样扩展升级而来；扩展定义略有调整，以同时支持多采样与超采样实现。

F4 多纹理

多纹理单元功能支持多个纹理单元。除仅支持纹理单元0进行评估与反馈外，各纹理单元功能完全相同。每个纹理单元拥有独立的状态向量，包含纹理顶点数组规范、纹理图像与过滤参数以及纹理环境应用。

纹理单元的纹理环境以流水线方式应用：前一纹理环境的输出将作为后续纹理环境的输入片段颜色。

用于下一个纹理环境。对纹理客户端状态和纹理服务器状态的修改，分别通过两个选择器之一进行路由，这些选择器控制受影响的纹理状态实例。

多纹理功能由GL ARB多纹理扩展升级而来。 _ _

F.5 纹理加法环境模式

TEXTURE ENV MODE 纹理环境函数 ADD 提供了一种纹理功能，用于添加传入的片段和纹理源颜色。

纹理添加模式是从 GL ARB 纹理环境添加扩展升级而来。 _ _ _ _

F.6 纹理组合环境模式

纹理环境函数COMBINE的TEXTURE ENV MODE模式提供多种可编程组合功能，其参数可包含输入片段颜色、纹理源颜色、纹理常量颜色以及前一纹理环境阶段的结果。

组合运算包括：透明传递、乘法、加法与偏移加法、减法，以及指定参数的线性插值。RGB与A通道可分别选择不同组合运算，最终结果可按1、2或4倍进行缩放。

纹理组合功能由GL ARB纹理环境组合扩展功能升级而来。 _ _ _ _

F.7 纹理点三环境模式

TEXTURE ENV MODE_COMBINE 操作还提供指定参数的三元点积运算，其标量结果将复制到输出颜色的RGB或RGBA分量中。点积采用伪有符号算术实现，以支持像素级光照计算。

纹理DOT3模式是从GL ARB纹理环境dot3扩展中提升而来的。 _ _ _ _

F.8 纹理边界裁剪

纹理卷绕参数中的"边界裁剪"模式会在所有Mipmap级别上对纹理坐标进行裁剪，使得当纹理过滤器跨越纹理图像边缘时，返回的颜色仅来源于边界纹素。

时，仅从边界纹理像素中获取颜色。该行为与 OpenGL 1.2 引入的纹理边缘钳位模式一致。

纹理 边界 裁剪 被 提升 从 GL ARB 纹理边界裁剪 扩展

GL ARB 纹理边界裁剪扩展中 _

F9 转置矩阵

新增功能与标记符，支持将采用行优先顺序（而非列优先顺序）存储的应用程序矩阵传输至实现层。这使得应用程序能够使用标准C语言二维数组，且数组索引与预期矩阵行/列索引保持一致。此类数组被称为转置矩阵，因其是传递至OpenGL的标准矩阵的转置形式。

转置矩阵为数据在OpenGL管道间的传输提供了接口，但不改变任何OpenGL处理流程，也不暗示状态表示方式的变更。

转置矩阵功能源自GL ARB转置矩阵扩展规范。 _ _ _

F.10 鸣谢

OpenGL 1.3 是众多人士共同努力的成果。以下是部分贡献者名单，包括其贡献时所代表的公司：

Adrian Muntianu，ATIAl Reyes，
3dfx
阿兰·布夏尔，Matrox公司艾伦·科
米克，SGI公司艾伦·海里希，康柏
公司亚历克斯·埃雷拉，SP3D公司
艾伦·阿金，VA Linux公司艾伦·加
洛塔，ATI公司
Alligator Descartes，Arcane公司Andy Vesper
，MERL公司
安迪·沃尔夫，钻石多媒体公司阿克塞尔·希尔丹，
S3公司
巴托尔德·利希滕贝尔特，3Dlabs本杰明·利普查
克，康柏
Bill Armstrong，Evans & Sutherland

Bill Clifford，英特尔 Bill
Mannel，SGI Bimal Poddar，英
特尔 Bob Beretta，苹果 Brent
Insko，英伟达 Brian Goldiez，
UCF
布莱恩·格林斯通，苹果公司布莱恩·
保罗，VA Linux布莱恩·夏普，
GLSetup布鲁斯·达莫拉，IBM布鲁斯·
斯托克韦尔，康柏克里斯·布雷迪，
Alt.software克里斯·弗雷泽，Raycer克
里斯·霍尔，3dlabs
克里斯·赫克，GLSetup克里斯·莱恩，
英特尔
克里斯·索恩博罗，像素融合公司
Chuck Smith，Intelligraphics公司；Craig
Dunwoody，SGI公司；Dairsie Latimer，
PixelFusion公司
戴尔·柯克兰，3Dlabs / Intergraph丹·布罗肯希尔，
IBM
丹·金斯伯格，ATI丹·麦凯布，
S3
戴夫·阿隆森，微软公司 戴夫·戈塞林
，ATI公司
戴夫·施赖纳，SGI戴夫·曾兹，戴
尔
戴维·阿伦森，微软公司；戴维·布莱斯，SGI
公司
戴维·柯克，英伟达公司；戴维
·斯托里，SGI公司；戴维·于，
SGI公司；迪安娜·霍恩，3dfx
公司
迪克·库尔特，硅魔公司 唐·穆利斯，3dfx
埃蒙·奥迪亚，像素融合公司 爱德华（奇普）·
希尔，像素融合公司 小畑英二，日本电气公
司

埃利奥·德尔·朱迪斯，Matrox公司；

埃里克·杨，S3公司

埃文·哈特，ATI 弗雷德·费舍

尔，3dLabs

加里·帕克西诺斯，Metro Link加里·塔罗利，
3dfx

乔治·基里亚齐斯，英伟达；格雷厄姆·

康纳，IMG；赫伯·库塔，Quantum3D

；霍华德·米勒，苹果；伊戈尔·西尼亚

克，英特尔

杰克·米德尔顿，Sun公司

扬·C·哈登伯格，MERL杰森·米切尔，ATI

杰夫·韦曼，ATI杰弗里·纽奎斯特

，3dfx

延斯·欧文，精密洞察公司；杰里米·莫里斯，
3Dlabs

吉姆·布什内尔，金字塔峰公司

约翰·斯托弗，苹果公司；约翰·泰南

，像素融合公司；约翰·W·波利克，

日本电气公司

乔恩·利奇，SGI

乔恩·保罗·谢尔特，Matrox卡尔·希莱

斯拉德，NVIDIA凯尔文·汤普森

肯·卡梅隆，Pixelfusion肯·戴克，苹果

肯·尼科尔森，SGI肯特·林，

英特尔凯文·勒菲弗，惠普

凯文·马丁，VA Linux

莱斯·西尔文，日本电气

马赫什·丹迪帕尼，Rendition马克·基尔加德，英

伟达

马丁·阿蒙，3dfx公司；玛蒂娜·苏拉

达，ATI公司；马特·拉沃伊，

Pixelfusion公司；马特·鲁索，Matrox

公司

马修·帕帕基波斯，英伟达

摩根·冯·埃森，Metro Link

内森·塔克，Raycer Graphics尼尔·特雷维特，
3Dlabs

牛顿·张，S3尼克·特里安托斯，英

伟达帕特里克·布朗，英特尔保罗·

詹森，3dfx

保罗·凯勒，英伟达 保罗·马茨

，惠普 保拉·沃马克，3dfx

彼得·多恩格斯，埃文斯与萨瑟兰公司 彼得·格拉法尼

诺，苹果公司

菲尔·赫胥黎，3Dlabs 拉尔夫·比尔

曼，Elsa AG 兰迪·罗斯特，3Dlabs

蕾妮·拉希德，美光科技 里奇·约翰

逊，惠普 理查德·皮门特尔，PTC

理查德·施莱因，苹果公司 里克·哈

默斯通，ATI 里克·费斯，VA Linux

罗布·格利登，Sun

罗布·惠勒，3dfx

莎莉·彼得森，Rendition肖恩·霍普伍

德，SGI

史蒂夫·格利克曼，硅魔术公司

史蒂夫·赖特，微软斯图尔特·安德森，
Metro Link

T. C. 赵，MERL

Teri Morrison，惠普公司

Thomas Fox，IBM公司

Tim Kelley，Real 3DTom Frisinger，
ATIVictor Vedovato，美光科技
Vikram Simha，MERL张彦军，
SunZahid Hussain，德州仪器

附录 G

版本 1.4

OpenGL 1.4版于2002年7月24日发布，是自原始1.0版以来的第四次修订。该版本向上兼容早期版本，意味着任何能在1.3、1.2、1.1或1.0版GL实现中运行的程序，在1.4版GL实现中亦可无修改运行。

除对经典固定功能GL管道进行大量扩展外，OpenGL ARB还批准了ARB顶点程序扩展，该扩展支持可编程顶点处理。以下简要说明OpenGL 1.4的各项新增功能；ARB顶点程序的详细说明请参见J章。

G.1 自动生成Mipmap

将纹理参数GENERATE MIPMAP设置为TRUE会对MIPMAP数组的任何*基础级别*修改产生副作用，即通过对基础级别数组进行连续过滤，自动重新计算MIPMAP金字塔的所有更高级别。

自动生成 级MIP贴图 生成 被 提升至 ，该功能源自 的 SGIS生成Mipmap扩展。

G.2 混合平方

混合平方扩展了支持的源和目标混合函数集，允许在混合过程中对RGB和Alpha值进行平方运算。允许的源混合函数新增SRC COLOR和ONE MINUS SRC COLOR函数，允许的目标混合函数新增DST COLOR和ONE MINUS DST COLOR函数。

混合平方功能由GL_NV混合平方扩展升级而来。

G.3 成像子集的变更

由 **BlendEquation**、**BlendColor** 以及 *BlendFunc 模式*（包括CONSTANT_COLOR、ONE_MINUS_CONSTANT_COLOR、CONSTANT_ALPHA和ONE_MINUS_CONSTANT_ALPHA）描述的混合特性子集现已获得支持。这些特性在GL 1.2和1.3版本中仅作为可选成像子集提供。

G.4 深度纹理与阴影

深度纹理定义了一种新的纹理内部格式DEPTH，通常用于表示深度值。其应用场景包括基于图像的投射阴影、位移贴图以及基于图像的渲染。

基于图像的阴影渲染通过参数TEXTURE_COMPARE_MODE定义的新纹理应用模式实现。该模式支持将纹理r坐标与深度纹理值进行比较，从而生成布尔结果。

深度纹理与阴影功能已从GL_ARB深度纹理和GL_ARB阴影扩展中提升而来。

G.5 雾坐标

一种新的关联顶点和片段数据——*雾坐标*，可用于计算片段的雾效，替代基于视点到片段距离的计算方式。通过FogCoord命令指定该坐标并设置FOG_COORDINATE_SOURCE雾参数即可实现。雾坐标在计算更复杂的雾模型时尤为有效。

雾坐标功能源自GL_EXT雾坐标扩展。

G.6 多重绘制数组

可通过 **MultiDrawArrays** 和 MultiDrawElements 命令在单次调用中绘制多个基元。MultiDrawElements命令。

多重绘制数组是从GL_EXT多重绘制数组扩展中提升而来的扩展。

G.7 点参数

由PointParameter命令定义的点参数支持点的附加几何特性，允许点大小受线性或二次距离衰减影响，并增强点大小到光栅点面积及点透明度的映射控制。此效果可用于粒子或光点的渲染距离衰减。

点参数是从GL ARB点参数扩展中提升而来的。 _ _ _

G.8 次要颜色

即使禁用光照时，仍可通过SecondaryColor命令将其设为顶点参数来改变次要颜色。

辅助颜色是从 GL EXT 辅助颜色扩展中提升而来的。 _ _ _

G.9 独立混合函数

通过BlendFuncSeparate扩展混合功能，允许独立设置RGB和alpha混合函数，以满足需要源混合因子和目标混合因子的混合操作需求。

独立的混合函数（ from GL EXT blend func separate 扩展。

G.10 模板包裹

新增的模板运算符INCR WRAP和DECR WRAP允许模板值在递增或递减时绕过模板值范围进行循环，而非饱和至最小值或最大值。对于使用模板缓冲区进行片段级原始计算（如内部/外部检测）的算法，模板值循环功能不可或缺。

模板环绕功能源自GL EXT模板环绕扩展。 _ _ _

G.11 纹理交叉开关环境模式

纹理交叉功能扩展了纹理组合环境模式COMBINE，允许使用来自不同纹理单元的纹理颜色作为纹理组合函数的输入源。

纹理 环境 交叉点

ARB纹理环境交叉点扩展。 _

G.12 纹理LOD偏移量

纹理过滤控制参数TEXTURE LOD BIAS可用于偏移纹理化中用于选择Mipmap细节等级的计算λ参数，从而实现纹理的模糊或锐化效果。该LOD偏移值还可应用于景深及其他特殊视觉效果，以及某些图像处理类型。

纹理LOD偏移基于EXT纹理LOD偏移扩展实现，并新增了针对每个纹理对象的第二项偏移项。

G.13 纹理镜像重复

纹理镜像重复扩展了纹理包裹模式集，新增MIRRORED REPEAT模式。该模式实质上定义了一张尺寸为原始纹理两倍的贴图，其中每个镜像坐标对应的额外半幅区域均为原始纹理的镜像。镜像重复可用于实现表面无缝平铺。

纹理 mirrored repeat was promoted from the

ARB纹理镜像重复扩展。 _

G.14 窗口光栅位置

可通过WindowPos命令将光栅位置直接设置为指定窗口坐标，从而绕过**对RasterPos**施加的变换。窗口光栅位置在成像及其他二维操作中尤为实用。

窗口光栅位置是从 GL ARB 窗口位置扩展中提升而来的。 _ _ _

G.15 鸣谢

OpenGL 1.4 是众多人士共同努力的成果。以下列出了部分贡献者名单，包括其贡献时所代表的公司。编辑特别感谢 Bob Beretta 和 Pat Brown 长期领导 ARB 顶点程序工作组所做的持续努力，若无此项关键扩展的定义与批准，OpenGL 1.4 便无法问世。

_ _

库尔特·阿克利，英伟达公司艾伦·
阿金
Bill Armstrong, Evans & SutherlandBen Ashbaugh, Intel
克里斯·本特利，ATI公司鲍
勃·贝雷塔，苹果公司
丹尼尔·布罗肯希尔，IBM公司帕特·
布朗，英伟达公司
Bill Clifford，英特尔
格雷厄姆·康纳，Videologic公司
让-吕克·德里，迪斯克里特公司肯尼
斯·戴克，苹果公司卡斯·埃弗里特，
英伟达公司艾伦·加洛塔，ATI公司
李·格罗斯，IBM埃文·
哈特，ATI
克里斯·赫克，定义6艾伦·海里希，康柏/
惠普加雷斯·休斯，VA Linux迈克尔·I·戈
尔德，英伟达里奇·约翰逊，惠普
马克·基尔加德，英伟达戴尔·柯克
兰，3Dlabs大卫·柯克，英伟达
克里斯蒂安·拉福特，Alias—Wavefront公司吕克·勒布
朗，Discreet公司
乔恩·利奇，SGI
比尔·利西亚-凯恩，ATI巴托尔德·利希滕
贝尔特，3Dlabs杰克·米德尔顿，Sun霍华
德·米勒，苹果杰里米·莫里斯，3Dlabs
乔恩·保罗·谢尔特，Matrox
布莱恩·保罗，VA Linux/钨石图形比马尔·波达尔，英特尔
托马斯·罗尔，Xi Graphics兰迪·罗斯特，3Dlabs
杰里米·桑德梅尔，ATI

约翰·斯托弗，苹果尼克·特里安托斯，英伟达

丹尼尔·沃格尔，Epic Games梅森·吴，World Wide Woo戴夫·曾茨，戴尔

附录 H

版本 1.5

OpenGL 1.5版本于2003年7月29日发布，是自原始1.0版本以来的第五次修订。该版本向上兼容早期版本，这意味着任何能在1.4、1.3、1.2、1.1或1.0 GL实现环境下运行的程序，在1.5 GL实现环境中同样无需修改即可运行。

除在OpenGL 1.5中对经典固定功能GL管道进行扩展外，OpenGL ARB还批准了一系列相关扩展，包括OpenGL着色语言规范以及ARB着色器对象、ARB顶点着色器和ARB片段着色器扩展。通过这些扩展，可加载高级着色语言程序并替代固定功能管道使用。

以下是对OpenGL 1.5新增功能的简要说明。低级和高级着色语言是OpenGL核心的重要补充，其详细说明见附录J，对应的ARB扩展规范可通过该附录所述方式在线获取。

H.1 缓冲对象

缓冲对象允许将各类数据（尤其是顶点数组数据）缓存于服务器端的高性能图形内存中，从而提升向GL传输数据的速率。

缓冲对象是从 ARB 顶点缓冲对象扩展中提升而来的。

H.2 遮挡查询

遮挡查询是一种机制，应用程序可通过它查询某个基元或基元组绘制的像素数（更准确地说，是采样数）。遮挡查询的主要目的是确定对象的可见性。

遮挡查询是从ARB遮挡查询扩展中提升而来的功能。

H.3 阴影函数

纹理比较函数已扩展为支持全部八种二元运算，而不仅限于LEQUAL和GEQUAL。

纹理比较函数已从EXT阴影函数扩展中提升
扩展中提升。

H.4 变更标记

为符合OpenGL函数与标记名称的语法规范，引入新标记名称替代旧有不一致名称。但为兼容旧版OpenGL代码，旧标记名称仍被支持。新名称及其替代的旧名称详见表H.1。

H.5 鸣谢

OpenGL 1.5 是众多人士共同努力的成果。编辑部特别感谢以下人士在领导 ARB 工作组过程中所作的持续贡献，这些工作组对 OpenGL 1.5 的成功以及与 OpenGL 1.5 一起批准的 ARB 扩展至关重要：

Matt Craighead 领导了 工作组 该组 创建了 ARB 顶点缓冲对象扩展及 OpenGL 1.5 核心特性。Kurt Akeley 为该组撰写了初始规范。

Daniel Ginsburg与Matt Craighead领导的工作组创建了
ARB遮挡查询扩展与OpenGL 1.5核心特性。

Benjamin Lipchak领导的片段程序工作组创建了ARB片段程序扩展，完成了低级可编程着色接口。

比尔·利塞亚-凯恩领导的GL2工作组创建了高级可编程着色接口，包括ARB片段着色器、

新令牌名称	旧令牌名称
雾协调源 _	雾坐标源 _
雾坐标_	雾坐标_
当前雾坐标 _ _	当前雾坐标 _ _
雾坐标数组类型 _ _	雾协调数阵列类型 _ _
雾坐标数组步长 _ _	雾坐标数组步长 _ _
雾坐标数组指针 _ _	雾坐标数组指针 _ _
雾坐标数组 _	雾坐标数组 _
雾坐标数组缓冲区绑定 _ _	雾坐标数组缓冲区绑定 _ _ _
SRC0 RGB	SOURCE0 RGB _
SRC1 RGB	源1 RGB _
SRC2 RGB	SOURCE2 RGB _
SRC0 透明度	源0 透明度 _
SRC1 透明度	SOURCE1 ALPHA
SRC2 透明度	SOURCE2 ALPHA

表 H.1： 新令牌名称及其替换的旧名称。

ARB着色器对象、ARB顶点着色器扩展及OpenGL着色语言。 _

John Kessenich 是 GL2 工作组中 OpenGL 着色语言规范的主要编辑，该规范源于 John、Dave Baldwin 和 Randi Rost 共同撰写的初始 glslang 提案。

其他贡献者部分名单（含贡献时所代表公司）如下：

库尔特·阿克利，英伟达公司艾伦·

阿金

查德·安森，戴尔计算机

Bill Armstrong，Evans & Sutherland公司Ben Ashbaugh

，英特尔公司

戴夫·鲍德温，3Dlabs克里斯·本特

利，ATI鲍勃·贝雷塔，苹果公司

大卫·布莱斯

阿兰·布夏尔，Matrox丹尼尔·布罗肯

希尔，IBM帕特·布朗，英伟达

约翰·卡马克，id软件公司

保罗·卡迈克尔，英伟达鲍勃·卡威尔，
IBM
保罗·克拉克，IBM比尔·
克利福德，英特尔罗杰·克
劳德，SGI
格雷厄姆·康纳，PowerVR 马特·克雷格黑德
，英伟达 道格·克里斯曼，SGI
马特·克鲁克香克，Vital Images 德伦·丹·约翰
逊，Sun 苏西·德菲斯，IBM
史蒂夫·德姆洛，Vital Images 乔·邓，SiS
让-吕克·德里，Discreet公司；肯尼斯
·戴克，苹果公司；布莱恩·恩伯林，
Sun公司；卡斯·埃弗里特，英伟达公
司；布兰登·弗利夫莱特，英特尔公
司；艾伦·加洛塔，ATI公司；丹尼尔
·金斯伯格，ATI公司；史蒂夫·格兰维
尔，英伟达公司；彼得·格拉法尼诺
，苹果公司；李·格罗斯，IBM公司
里克·哈默斯通，ATI
克里斯·赫克，Definition 6公司
加雷斯·休斯，英伟达 迈克尔·I·戈尔
德，英伟达 约翰·贾维斯，
Alt.software 里奇·约翰逊，惠普
约翰·凯塞尼奇，3Dlabs马克·基尔
加德，英伟达戴尔·柯克兰，3Dlabs
雷蒙德·克拉森，英特尔杰森·奈普
，Bioware贾扬特·科尔赫，英伟达
史蒂夫·科伦，3Dlabs鲍勃·库恩，
SGI克里斯蒂安·拉福特，Alias

吕克·勒布朗，Discreet 乔恩·利
奇，SGI
凯文·勒菲弗尔，惠普比尔·利
西亚·凯恩，ATI
巴托尔德·利希滕贝尔特，3Dlabs肯特·林，英特
尔
本杰明·利普查克，ATI罗布·梅斯，
ATI
比尔·马克，英伟达
迈克尔·麦库尔，滑铁卢大学杰克·米德尔顿，太
阳报
霍华德·米勒，苹果
泰瑞·莫里森，惠普/3Dlabs马克·奥拉诺，SGI/
马里兰大学让-弗朗索瓦·帕尼塞，Discreet乔恩·
保罗·谢尔特，Matrox
布莱恩·保罗，钨图形公司
比马尔·波达尔，英特尔托马斯·罗尔，
Xi Graphics菲尔·罗杰斯，ATI
伊恩·罗曼尼克，IBM约翰·罗萨斯
科，苹果兰迪·罗斯特，3Dlabs马
特·鲁索，Matrox杰里米·桑德梅尔
，ATI保罗·萨金特，3Dlabs
福尔克·沙梅尔，Spinor GMBH公司；迈克尔·
舒尔曼，Sun公司
John Scott，Raven Software Avinash
Seetharamaiah，英特尔 John Spitzer，英伟达
弗拉德·斯塔马特，PowerVR公司 米
歇尔·斯塔姆内斯，英特尔公司 约翰·
斯托弗，苹果公司
埃斯基尔·斯滕伯格，Obsession公司；布鲁斯
·斯托克韦尔，惠普公司；克里斯托弗·谭，
IBM公司
雷·泰斯，Avid
皮埃尔·P·特伦布莱，Discreet

尼尔·特雷维特，3Dlabs 尼克·特里安
托斯，英伟达 道格拉斯·特威利格，
太阳微系统 肖恩·安德伍德，SGI
史蒂夫·厄克哈特，英特利图技维克多·维多瓦托
，ATI
Daniel Vogel，Epic GamesMik Wells，
SoftimageHelene Workman，AppleDave
Zenz，Dell
卡雷尔·祖德费尔德，Vital Images

附录 I

版本 2.0

OpenGL 2.0版本于2004年9月7日发布，是自原始1.0版本以来的第六次修订。 尽管主版本号有所提升（以表明对高级可编程着色器的支持），但2.0版本仍与早期版本向上兼容，这意味着任何能在1.5、1.4、1.3、1.2、1.1或1.0版GL实现中运行的程序，在2.0版GL实现中同样无需修改即可运行。

以下是对OpenGL 2.0新增功能的简要说明。

I.1 可编程着色

OpenGL着色语言及其相关API（用于创建、管理和使用基于着色语言编写的可编程着色器）在OpenGL 2.0中被提升为核心特性。与可编程着色相关的完整特性列表包括：

I.1.1 着色器对象

着色器对象提供了管理着色器和程序对象所需的机制。着色器对象是从ARB着色器对象扩展中提升而来的。

I.1.2 着色器程序

顶点着色器和片段着色器程序可采用高级OpenGL着色语言编写，分别替代固定功能顶点处理和片段处理。顶点着色器与片段着色器程序源自ARB顶点着色器扩展和ARB片段着色器扩展。

I.1.3 OpenGL着色语言

OpenGL着色语言是一种类似C语言的高级语言，用于编程顶点和片段处理流水线。着色语言规范定义了语言本身，而OpenGL API特性则控制顶点和片段程序如何与固定功能OpenGL流水线交互，以及应用程序如何管理这些程序。

OpenGL 2.0 实现 必须 支持 至少 修订版 1.10

实现可查询SHADING LANGUAGE VERSION字符串以确定所支持语言的确切版本。 OpenGL着色语言是从ARB着色语言100扩展升级而来（着色语言本身在配套文档中规定；由于其编写方式，该文档无需因可编程着色升级为OpenGL核心而更改）。

– – –

I.1.4 着色器 API 的变更

在将着色器扩展功能纳入OpenGL 2.0核心的过程中，对管理着色器和程序对象的API进行了细微调整。这些变更不影响着色器API的功能，但包括：使用现有的uint核心GL类型替代扩展功能引入的新handleARB类型，以及部分函数名称的变更——例如将扩展函数**CreateShaderObjectARB**映射至核心函数**CreateShader**。

I.2 多重渲染目标

可编程着色器可在单次渲染通道中向多个输出颜色缓冲区写入不同颜色。 多渲染目标功能源自ARB绘制缓冲区扩展。

– –

I.3 非2的幂次方纹理

所有纹理目标均已放宽对纹理尺寸为2的幂次方的限制，因此可指定非2的幂次方纹理而不会引发错误。 非2的幂次方纹理功能已从ARB纹理非2的幂次方扩展中正式纳入标准。

– – – – –

I.4 点精灵

点精灵通过在点上插值计算纹理坐标来替代点纹理坐标。这允许将点绘制为自定义纹理，对粒子系统尤为实用。

点精灵是从ARB点精灵扩展中提升而来的，并进一步增加了POINT SPRITE COORD ORIGIN参数，用于控制纹理坐标*t*的增长方向。

I.5 独立模板

可为原始图元的正面和背面分别定义独立模板功能，从而提升阴影体积和构造实体几何渲染算法的性能。

独立模板功能基于ATI独立模板扩展的API实现，并通过类似的EXT模板双面扩展定义了额外状态。

I.6 其他变更

对OpenGL 1.5规范进行了若干细微修订与修正：

- 在第2.7节中，将**SecondaryColor3**的A通道值修改为1.0（原为0.0），以便恢复初始GL状态。
- 在第2.13节中，将变换操作添加至**WindowPos**未执行的步骤列表。
- 第3.8.1节已明确规定：选择纹理内部格式时，必须为该格式定义的所有组件分配非零位数，其余组件则分配零位数。
- 通过将表3.22和3.23*中的*C_f替换为C_{fp})

  替换*为*C_p。
- 在第6.1.9节中，明确指出GetHistogram在存储直方图计数时不应用最终转换像素存储模式。
- 在表H.1中添加了FOG COORD ARRAY BUFFER BINDING枚举别名。

在OpenGL 2.0初始版本发布后，2004年10月22日批准的规范修订版进行了若干次细微修正：

- 在第2.13节中，将雾源名称从FOG COORD SRC修正为FOG COORD。
- 修正**UniformMatrix***声明中最后一个参数类型
  命令的声明中，将参数类型修正为`const float *`*value*，详见第2.15.3节。
- 修改了第3.6.4节中"**转换为片段**"小节第二段的结尾部分，以更清晰地描述生成的片段集合。
- 在第3.10节中，将旧版FOG坐标转换为新版FOG坐标表示法。
- 在表6.13中新增了POINT SPRITE COORD ORIGIN状态。
- 将表6.34中MAX TEXTURE UNITS的描述修改为体现其遗留状态（指固定功能纹理单元数量），并将其移至表6.35。
- 移除了表项中重复的MAX TEXTURE IMAGE UNITS和MAX TEXTURE COORDS 来自表 6.35。
- 在OpenGL 2.0致谢部分新增Victor Vedovato。
- 各类排版错误修正。

I.7 致谢

OpenGL 2.0是众多人士贡献的结晶。编者特别感谢由Bill Licea-Kane领导、John Kessenich与Barthold Lichtenbelt负责规范编辑的ARB GL2工作组持续开展的工作，他们为推动OpenGL着色语言成为OpenGL核心特性所做的必要工作。

其他贡献者名单（含贡献时所代表公司）如下：

库尔特·阿克利，英伟达公司艾伦·

阿金

戴夫·鲍德温，3Dlabs公司鲍勃·贝

雷塔，苹果公司

帕特·布朗，英伟达马特·克雷格海德
，英伟达苏西·德菲斯，IBM
肯·戴克，苹果公司卡斯·埃弗里
特，英伟达
史蒂夫·格兰维尔，英伟达迈克尔·I·戈
尔德，英伟达埃文·哈特，ATI
菲尔·赫克斯利，3Dlabs 德伦·丹·约翰
逊，Sun 约翰·凯塞尼奇，3Dlabs 马克·
基尔加德，NVIDIA 戴尔·柯克兰，
3Dlabs 史蒂夫·科伦，3Dlabs
乔恩·利奇，SGI
Bill Licea-Kane，ATIBarthold Lichtenbelt，
3DlabsKent Lin，英特尔
本杰明·利普查克，ATI罗布·梅斯，
ATI
Michael McCool，滑铁卢大学Jack Middleton，
Sun
Jeremy Morris，3DlabsTeri
Morrison，3Dlabs
马克·奥拉诺，SGI / 马里兰大学格伦·奥特纳，
ATI
布莱恩·保罗，钨图形公司比马尔·波达尔，英特
尔
菲尔·罗杰斯，ATI公司；伊
恩·罗曼尼克，IBM公司；兰
迪·罗斯特，3Dlabs公司
杰里米·桑德梅尔，ATI
福尔克·沙梅尔，Spinor GMBH杰夫·斯塔尔，
苹果
埃斯基尔·斯滕伯格，Obsession尼尔·特雷维
特，3Dlabs
维克多·维多瓦托，ATI公司；米
克·韦尔斯，软影公司；埃森·伊尔
马兹，英特尔公司；戴夫·曾茨，
戴尔公司

附录 J

ARB扩展

本章描述了经OpenGL架构审查委员会（ARB）批准的OpenGL扩展功能。这些扩展并非符合规范的OpenGL实现所必需支持的内容，但预计将被广泛采用；它们定义的功能很可能在未来规范修订中纳入必备特性集。

为避免影响核心规范的可读性，ARB扩展并未整合到核心语言中；而是通过在线方式*发布于OpenGL扩展注册库*（该库还收录了大量厂商专属扩展，以及GLX和WGL的扩展）。扩展以规范变更的形式进行文档化。该注册库可通过以下网址访问：

http://oss.sgi.com/projects/ogl-sample/registry/

以下提供 ARB 扩展的简要说明。

J.1 命名规范

为区分ARB扩展与OpenGL核心特性及厂商专属扩展，采用以下命名规范：

- 每个扩展均关联一个"`GL_ARB`*名称*"格式的唯一*名称字符串*。若扩展被实现支持，该字符串将出现在第6.1.11节所述的EXTENSIONS字符串中。
- 扩展定义的所有函数名称都将采用 **Func-tionARB** 的形式

- 扩展定义的所有枚举项名称将采用以下形式
  名称 ARB.

J.2 将扩展功能提升为核心特性

ARB扩展可在OpenGL后续版本中被*提升*为核心必备功能。此时扩展规范将并入核心规范。此类提升扩展中的函数与枚举项将移除**ARB**后缀。

后续版本的GL实现应继续在EXTENSIONS字符串中导出已提升扩展的名称字符串，并继续支持带ARB后缀的函数与枚举类型版本，以作为过渡期辅助措施。

有关在OpenGL 1.3及更高版本中被提升为核心功能的扩展说明，请分别参见附录F、G、H和I。

J.3 多纹理

多纹理的名称字符串为GL ARB多纹理。该功能在OpenGL 1.3中被提升为核心特性。

J.4 转置矩阵

转置矩阵的名称字符串为GL ARB转置矩阵。该功能在OpenGL 1.3中被提升为核心特性。

J.5 多采样

多采样的名称字符串为GL ARB多采样。该功能在OpenGL 1.3中被提升为核心特性。

J.6 纹理添加环境模式

纹理添加模式的名称字符串为GL ARB texture env add。该功能在OpenGL 1.3中被提升为核心特性。

J.7 立方体贴图

立方贴图的名称字符串为GL ARB texture cube map。该功能在OpenGL 1.3中被提升为核心特性。

J.8 压缩纹理

压缩纹理的名称字符串为GL ARB纹理压缩。该功能在OpenGL 1.3中被提升为核心特性。

J.9 纹理边界裁剪

纹理边界裁剪的名称字符串为GL ARB纹理边界裁剪。该功能在OpenGL 1.3中被提升为核心特性。

J.10 点参数

点参数的名称字符串为GL ARB点参数。该功能在OpenGL 1.4中被提升为核心特性。

J.11 顶点混合

顶点混合通过多个顶点单元替代单一的模型视图变换。每个单元拥有独立的变换矩阵及关联的当前权重。顶点经所有启用单元变换后，按各自权重进行缩放，最终求和生成视点空间顶点。法线则通过模型视图矩阵的逆转置矩阵进行类似变换。

顶点混合的名称字符串为GL ARB顶点混合。

J.12 矩阵调色板

矩阵调色板将顶点混合功能扩展至包含模型视图矩阵调色板。每个顶点可通过从调色板中选取的不同矩阵集进行变换。

矩阵调色板的名称字符串为GL ARB矩阵调色板。

J.13 纹理组合环境模式

纹理组合模式的名称字符串为 GL ARB texture env combine。该功能在 OpenGL 1.3 中被提升为核心特性。

J.14 纹理交叉开关环境模式

纹理交叉器的名称字符串为GL ARB texture env crossbar。该功能在OpenGL 1.4中被提升为核心特性。

J.15 纹理点3环境模式

DOT3的名称字符串为GL ARB texture env dot3。该功能在OpenGL 1.3中被提升为核心特性。

J.16 纹理镜像重复

纹理镜像重复功能的名称字符串为GL ARB texture mirrored repeat，对应于OpenGL 1.4版本中核心功能的名称字符串 ，该功能在OpenGL 1.4版本中被提升为核心功能。

J.17 深度纹理

深度纹理的名称字符串为GL ARB深度纹理。该功能在OpenGL 1.4中被提升为核心特性。

J.18 阴影

阴影的名称字符串为GL ARB阴影。该功能在OpenGL 1.4中被提升为核心特性。

J.19 环境光遮蔽

环境光阴影通过允许在纹理比较失败时返回由TEXTURE COMPARE FAIL VALUE ARB纹理参数指定的纹理值，扩展了基础图像阴影功能。该特性可用于阴影化片段的环境光照及其他高级光照效果。

阴影环境光的名称字符串为 GL ARB 阴影环境光。

J.20 窗口光栅化位置

窗口光栅化位置的名称字符串为GL ARB window pos。该功能在OpenGL 1.4中被提升为核心特性。

J.21 低级顶点编程

应用程序定义*的顶点程序*可采用新型低级编程语言编写，取代标准的固定功能顶点变换、光照及纹理坐标生成管道。顶点程序实现了诸多全新特效，并为未来图形管道发展奠定重要基础——届时图形管道将通过无限制的高级着色语言实现完全可编程化。

低级顶点编程的名称字符串为ARB顶点程序。 _ _

J.22 低级片段编程

应用程序定义的*片段程序*可采用与ARB顶点程序相同的低级语言编写，从而取代标准的固定功能顶点纹理映射、雾效及颜色求和运算。 _ _

名称 字符串 用于 低级 片段 编程 是
ARB 片段程序。 _

J.23 缓冲对象

缓冲对象的名称字符串为 ARB 顶点缓冲对象。它在 OpenGL 1.5 中被提升为核心功能。_

J.24 遮挡查询

遮挡查询的名称字符串为ARB遮挡查询。该功能在OpenGL 1.5中被提升为核心特性。_

J.25 着色器对象

着色器对象的名称字符串为ARB着色器对象。该功能在OpenGL 2.0中被提升为核心特性。

J.26 高级顶点编程

用于高级顶点编程的名称字符串是ARB顶点着色器。它在OpenGL 2.0中被提升为核心功能。

J.27 高级片段编程

高阶片段编程的名称字符串为ARB片段着色器。该功能在OpenGL 2.0中被提升为核心特性。

J.28 OpenGL着色语言

名称字符串用于OpenGL Shading语言是ARB着色语言100。该扩展字符串的存在表明，使用着色语言版本1编写的程序被OpenGL所接受。

它在OpenGL 2.0中被提升为核心功能。

J.29 非2的幂次方纹理

该名称字符串用于非2的幂次方纹理的名称为ARB纹理非2的幂。该功能在OpenGL 2.0中被提升为核心特性。

J.30 点精灵

点精灵的名称字符串为ARB点精灵。该功能在OpenGL 2.0中被提升为核心特性。

J.31 片段程序阴影

片段程序阴影扩展了通过ARB片段程序定义的低级片段程序，添加了1D、2D和3D纹理目标阴影功能，并移除了与ARB阴影的交互。

名称字符串用于片段程序_阴影

ARB 片段程序阴影。

J.32 多重渲染目标

多渲染目标的名称字符串为ARB绘制缓冲区。该功能在OpenGL 2.0中被提升为核心特性。

J.33 矩形纹理

矩形纹理定义了新的纹理目标TEXTURE RECTANGLE ARB，支持无需2的幂次方尺寸的2D纹理。 矩形纹理适用于存储非2的幂次方尺寸（POTS）的视频图像，可避免重采样伪影并减少纹理内存需求。该特性同样适用于阴影贴图和窗口空间纹理映射。此类纹理通过维度依赖型（即非归一化）纹理坐标进行访问。

矩形纹理是非2的幂纹理的受限版本。其差异在于：矩形纹理仅支持2D应用；需要新的纹理目标；且新目标使用非归一化纹理坐标。

纹理矩形的名称字符串为ARB纹理矩形。

索引