# The OpenGL$^{TM}$ Graphics System: A Specification (Version 1.0)

Mark Segal
Kurt Akeley

# Contents


**1 Introduction** **1**
- 1.1 What is the OpenGL Graphics System? . . . . . . . . . . . . 1
- 1.2 Programmer's View of OpenGL . . . . . . . . . . . . . . . . . 1
- 1.3 Implementor's View of OpenGL . . . . . . . . . . . . . . . . . 2
- 1.4 Our View . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

**2 OpenGL Operation** **4**
- 2.1 OpenGL Fundamentals . . . . . . . . . . . . . . . . . . . . . 4
- 2.2 GL State . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
- 2.3 GL Command Syntax . . . . . . . . . . . . . . . . . . . . . . 7
- 2.4 Basic GL Operation . . . . . . . . . . . . . . . . . . . . . . 10
- 2.5 GL Errors . . . . . . . . . . . . . . . . . . . . . . . . . . 11
- 2.6 Begin/End Paradigm . . . . . . . . . . . . . . . . . . . . . . 12
  - 2.6.1 Begin and End Objects . . . . . . . . . . . . . . . . . 14
  - 2.6.2 Polygon Edges . . . . . . . . . . . . . . . . . . . . . 18
  - 2.6.3 GL Commands within **Begin/End** . . . . . . . . . . . 19
- 2.7 Vertex Specification . . . . . . . . . . . . . . . . . . . . . 19
- 2.8 Rectangles . . . . . . . . . . . . . . . . . . . . . . . . . . 21
- 2.9 Coordinate Transformations . . . . . . . . . . . . . . . . . . 22
  - 2.9.1 Controlling the Viewport . . . . . . . . . . . . . . . . 23
  - 2.9.2 Matrices . . . . . . . . . . . . . . . . . . . . . . . . 24
  - 2.9.3 Normal Transformation . . . . . . . . . . . . . . . . . 28
  - 2.9.4 Generating texture coordinates . . . . . . . . . . . . . 29
- 2.10 Clipping . . . . . . . . . . . . . . . . . . . . . . . . . . 31
- 2.11 Current Raster Position . . . . . . . . . . . . . . . . . . . 33
- 2.12 Colors and Coloring . . . . . . . . . . . . . . . . . . . . . 36
  - 2.12.1 Lighting . . . . . . . . . . . . . . . . . . . . . . . . 37
  - 2.12.2 Lighting Parameter Specification . . . . . . . . . . . . 41

2.12.3 **ColorMaterial** . . . 42
2.12.4 Lighting State . . . 45
2.12.5 Color Index Lighting . . . 45
2.12.6 Clamping or Masking . . . 46
2.12.7 Flatshading . . . 47
2.12.8 Color and Texture Coordinate Clipping . . . 47
2.12.9 Final Color Processing . . . 48

**3 Rasterization** **49**
3.1 Invariance . . . 50
3.2 Antialiasing . . . 50
3.3 Points . . . 52
3.3.1 Point Rasterization State . . . 55
3.4 Line Segments . . . 55
3.4.1 Basic Line Segment Rasterization . . . 55
3.4.2 Other Line Segment Features . . . 58
3.4.3 Line Rasterization State . . . 62
3.5 Polygons . . . 62
3.5.1 Basic Polygon Rasterization . . . 62
3.5.2 Stippling . . . 64
3.5.3 Antialiasing . . . 65
3.5.4 Options Controlling Polygon Rasterization . . . 65
3.5.5 Polygon Rasterization State . . . 66
3.6 Pixel Rectangles . . . 66
3.6.1 Pixel Storage Modes . . . 67
3.6.2 Pixel Transfer Modes . . . 67
3.6.3 Rasterization of Pixel Rectangles . . . 69
3.7 Bitmaps . . . 76
3.8 Texturing . . . 78
3.8.1 Texture Minification . . . 82
3.8.2 Texture Magnification . . . 86
3.8.3 Texture Environments and Texture Functions . . . 86
3.8.4 Texture Application . . . 87
3.9 Fog . . . 89
3.10 Antialiasing Application . . . 91

**4 Per-Fragment Operations and the Framebuffer** **92**
4.1 Per-Fragment Operations . . . . . . . . . . . . . . . . . . . . . 93
4.1.1 Pixel Ownership Test . . . . . . . . . . . . . . . . . . . 93
4.1.2 Scissor test . . . . . . . . . . . . . . . . . . . . . . . . 94
4.1.3 Alpha test . . . . . . . . . . . . . . . . . . . . . . . . . 94
4.1.4 Stencil test . . . . . . . . . . . . . . . . . . . . . . . . 95
4.1.5 Depth buffer test . . . . . . . . . . . . . . . . . . . . . 96
4.1.6 Blending . . . . . . . . . . . . . . . . . . . . . . . . . . 97
4.1.7 Dithering . . . . . . . . . . . . . . . . . . . . . . . . . 99
4.1.8 Logical Operation . . . . . . . . . . . . . . . . . . . . . 99
4.2 Whole Framebuffer Operations . . . . . . . . . . . . . . . . . . 101
4.2.1 Selecting a Buffer for Writing . . . . . . . . . . . . . . . 101
4.2.2 Fine Control of Buffer Updates . . . . . . . . . . . . . . 102
4.2.3 Clearing the Buffers . . . . . . . . . . . . . . . . . . . . 103
4.2.4 The Accumulation Buffer . . . . . . . . . . . . . . . . . 104
4.3 Drawing, Reading, and Copying Pixels . . . . . . . . . . . . . . 106
4.3.1 Writing to the Stencil Buffer . . . . . . . . . . . . . . . 106
4.3.2 Reading Pixels . . . . . . . . . . . . . . . . . . . . . . . 106
4.3.3 Copying Pixels . . . . . . . . . . . . . . . . . . . . . . . 111
4.3.4 Pixel draw/read state . . . . . . . . . . . . . . . . . . . 112

**5 Special Functions** **113**
5.1 Evaluators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113
5.2 Selection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119
5.3 Feedback . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121
5.4 Display Lists . . . . . . . . . . . . . . . . . . . . . . . . . . . . 125
5.5 **Flush** and **Finish** . . . . . . . . . . . . . . . . . . . . . . . . 127
5.6 Hints . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 128

**6 State and State Requests** **129**

**A Invariance** **153**
A.1 Repeatability . . . . . . . . . . . . . . . . . . . . . . . . . . . . 153
A.2 Multi-pass Algorithms . . . . . . . . . . . . . . . . . . . . . . . 154
A.3 Invariance Rules . . . . . . . . . . . . . . . . . . . . . . . . . . 154
A.4 What All This Means . . . . . . . . . . . . . . . . . . . . . . . 156

**B Corollaries** **157**

**Index of OpenGL Commands** **160**

# List of Figures


2.1 Block diagram of the GL. . . . . . . . . . . . . . . . . . . . . . 10
2.2 Creation of a processed vertex from a transformed vertex and current values. . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.3 Primitive assembly and processing. . . . . . . . . . . . . . . . 14
2.4 Triangle strips, fans, and independent triangles. . . . . . . . . 16
2.5 Quadrilateral strips and independent quadrilaterals. . . . . . 17
2.6 Vertex transformation sequence. . . . . . . . . . . . . . . . . 22
2.7 Current raster position. . . . . . . . . . . . . . . . . . . . . . 34
2.8 Processing of colors. . . . . . . . . . . . . . . . . . . . . . . 36
2.9 ColorMaterial operation. . . . . . . . . . . . . . . . . . . . . 42

3.1 Rasterization. . . . . . . . . . . . . . . . . . . . . . . . . . . 49
3.2 Rasterization of non-antialiased wide points. . . . . . . . . . . 54
3.3 Rasterization of antialiased wide points. . . . . . . . . . . . . 54
3.4 Visualization of Bresenham's algorithm. . . . . . . . . . . . . 56
3.5 Rasterization of non-antialiased wide lines. . . . . . . . . . . 59
3.6 The region used in rasterizing an antialiased line segment. . . 61
3.7 Operation of **DrawPixels**. . . . . . . . . . . . . . . . . . . . 69
3.8 Selecting a subimage from an image . . . . . . . . . . . . . . 73
3.9 A bitmap and its associated parameters. . . . . . . . . . . . . 77
3.10 A texture image and the coordinates used to access it. . . . . 80

4.1 Per-fragment operations. . . . . . . . . . . . . . . . . . . . . 93
4.2 Operation of **ReadPixels**. . . . . . . . . . . . . . . . . . . . 106
4.3 Operation of **CopyPixels**. . . . . . . . . . . . . . . . . . . . 111

5.1 Map Evaluation. . . . . . . . . . . . . . . . . . . . . . . . . . 115
5.2 Feedback syntax. . . . . . . . . . . . . . . . . . . . . . . . . . 124

# List of Tables


2.1 Correspondence of command suffix letters to argument types. 8
2.2 GL argument types . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.3 Summary of GL errors . . . . . . . . . . . . . . . . . . . . . . . 12
2.4 RGBA color component conversions . . . . . . . . . . . . . . . 37
2.5 Summary of lighting parameters. . . . . . . . . . . . . . . . . . 39
2.6 Correspondence of lighting parameter symbols to names. . . . 43
2.7 Polygon flatshading color selection. . . . . . . . . . . . . . . . 47

3.1 **PixelStore** parameters pertaining to **DrawPixels**. . . . . . 67
3.2 **PixelTransfer** parameters. . . . . . . . . . . . . . . . . . . . . 68
3.3 **PixelMap** parameters. . . . . . . . . . . . . . . . . . . . . . . 69
3.4 **DrawPixels** and **ReadPixels** types. . . . . . . . . . . . . . 71
3.5 **DrawPixels** and **ReadPixels** formats. . . . . . . . . . . . . 72
3.6 Correspondence of texture components to extracted R, G, B, and A values. . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
3.7 Texture parameters and their values. . . . . . . . . . . . . . . 82
3.8 Texture functions. . . . . . . . . . . . . . . . . . . . . . . . . . 88

4.1 Values controlling the source blending function and the source blending values they compute . . . . . . . . . . . . . . . . . . 98
4.2 Values controlling the destination blending function and the destination blending values they compute . . . . . . . . . . . 98
4.3 Arguments to **LogicOp** and their corresponding operations. . 100
4.4 Arguments to **DrawBuffer** and the buffers that they indicate.102
4.5 **PixelStore** parameters pertaining to **ReadPixels**. . . . . . . 107
4.6 **ReadPixels** index masks and component conversion formulas.111

5.1 Values specified by the *target* to **Map1**. . . . . . . . . . . . . 114
5.2 Correspondence of feedback type to number of values per vertex.123

6.1 Attribute groups . . . . . . . . . . . . . . . . . . . . . . . . . . 133
6.2 State variable types . . . . . . . . . . . . . . . . . . . . . . . . 135

# Chapter 1

# Introduction

This document describes the OpenGL graphics system: what it is, how it acts, and what is required to implement it. We assume that the reader has at least a rudimentary understanding of computer graphics. This means familiarity with the essentials of computer graphics algorithms as well as familiarity with basic graphics hardware and associated terms.

## 1.1 What is the OpenGL Graphics System?

OpenGL (for "Open Graphics Library") is a software interface to graphics hardware. The interface consists of a set of several hundred procedures and functions that allow a programmer to specify the objects and operations involved in producing high-quality graphical images, specifically color images of three-dimensional objects.

Most of OpenGL requires that the graphics hardware contain a framebuffer. Many OpenGL calls pertain to drawing objects such as points, lines, polygons, and bitmaps, but the way that some of this drawing occurs (such as when antialiasing or texturing is enabled) relies on the existence of a framebuffer. Further, some of OpenGL is specifically concerned with framebuffer manipulation.

## 1.2 Programmer's View of OpenGL

To the programmer, OpenGL is a set of commands that allow the specification of geometric objects in two or three dimensions, together with commands that control how these objects are rendered into the framebuffer.

For the most part, OpenGL provides an immediate-mode interface, meaning that specifying an object causes it to be drawn.

A typical program that uses OpenGL begins with calls to open a window into the framebuffer into which the program will draw. Then, call are made to allocate a GL context and associate it with the window. Once a GL context is allocated, the programmer is free to issue OpenGL commands. Some calls are used to draw simple geometric objects (i.e. points, line segments, and polygons), while others affect the rendering of these primitives including how they are lit or colored and how they are mapped from the user's two- or three-dimensional model space to the two-dimensional screen. There are also calls to effect direct control of the framebuffer, such as reading and writing pixels.

## 1.3 Implementor's View of OpenGL

To the implementor, OpenGL is a set of commands that affect the operation of graphics hardware. If the hardware consists only of an addressable framebuffer, then OpenGL must be implemented almost entirely on the host CPU. More typically, the graphics hardware may comprise varying degrees of graphics acceleration, from a raster subsystem capable of rendering two-dimensional lines and polygons to sophisticated floating-point processors capable of transforming and computing on geometric data. The OpenGL implementor's task is to provide the CPU software interface while dividing the work for each OpenGL command between the CPU and the graphics hardware. This division must be tailored to the available graphics hardware to obtain optimum performance in carrying out OpenGL calls.

OpenGL maintains a considerable amount of state information. This state controls how objects are drawn into the framebuffer. Some of this state is directly available to the user: he or she can make calls to obtain its value. Some of it, however, is visible only by the effect it has on what is drawn. One of the main goals of this specification is to make OpenGL state information explicit, to elucidate how it changes, and to indicate what its effects are.

## 1.4 Our View

We view OpenGL as a state machine that controls a set of specific drawing operations. This model should engender a specification that satisfies the needs of both programmers and implementors. It does not, however, necessarily provide a model for implementation. An implementation must produce results conforming to those produced by the specified methods, but there may be ways to carry out a particular computation that are more efficient than the one specified.

# Chapter 2

# OpenGL Operation

## 2.1 OpenGL Fundamentals

OpenGL (henceforth, the "GL") is concerned only with rendering into a framebuffer (and reading values stored in that framebuffer). There is no support for other peripherals sometimes associated with graphics hardware, such as mice and keyboards. Programmers must rely on other mechanisms to obtain user input.

The GL draws *primitives* subject to a number of selectable modes. Each primitive is a point, line segment, polygon, or pixel rectangle. Each mode may be changed independently; the setting of one does not affect the settings of others (although many modes may interact to determine what eventually ends up in the framebuffer). Modes are set, primitives specified, and other GL operations described by sending *commands* in the form of function or procedure calls.

Primitives are defined by a group of one or more *vertices*. A vertex defines a point, an endpoint of an edge, or a corner of a polygon where two edges meet. Data (consisting of positional coordinates, colors, normals, and texture coordinates) are associated with a vertex and each vertex is processed independently, in order, and in the same way. The only exception to this rule is if the group of vertices must be *clipped* so that the indicated primitive fits within a specified region; in this case vertex data may be modified and new vertices created. The type of clipping depends on which primitive the group of vertices represents.

Commands are always processed in the order in which they are received, although there may be an indeterminate delay before the effects of a command are realized. This means, for example, that one primitive must be drawn completely before any subsequent one can affect the framebuffer. It also means that queries and pixel read operations return state consistent with complete execution of all previously invoked GL commands. In general, the effects of a GL command on either GL modes or the framebuffer must be complete before any subsequent command can have any such effects.

In the GL, data binding occurs on call. This means that data passed to a command are interpreted when that command is received. Even if the command requires a pointer to data, those data are interpreted when the call is made, and any subsequent changes to the data have no effect on the GL (unless the same pointer is used in a subsequent command).

The GL provides direct control over the fundamental operations of 3D and 2D graphics. This includes specification of such parameters as transformation matrices, lighting equation coefficients, antialiasing methods, and pixel update operators. It does not provide a means for describing or modeling complex geometric objects. Another way to describe this situation is to say that the GL provides mechanisms to describe how complex geometric objects are to be rendered rather than mechanisms to describe the complex objects themselves.

The model for interpretation of GL commands is client-server. That is, a program (the client) issues commands, and these commands are interpreted and processed by the GL (the server). The server may or may not operate on the same computer as the client. In this sense, the GL is "network-transparent." A server may maintain a number of GL *contexts*, each of which is an encapsulation of current GL state. A client may choose to *connect* to any one of these contexts.

The effects of GL commands on the framebuffer are ultimately controlled by the window system that allocates framebuffer resources. It is the window system that determines which portions of the framebuffer the GL may access at any given time and that communicates to the GL how those portions are structured. Therefore, there are no GL commands to configure the framebuffer or initialize the GL. Similarly, display of framebuffer contents on a CRT monitor (including the transformation of individual framebuffer values by such techniques as gamma correction) is not addressed by the GL. Framebuffer configuration occurs outside of the GL in conjunction with the window system; the initialization of a GL context occurs when the window system allocates a window for GL rendering.

The GL is designed to be run on a range of graphics platforms with varying graphics capabilities and performance. To accommodate this variety, we specify ideal behavior instead of actual behavior for certain GL operations. In cases where deviation from the ideal is allowed, we also specify the rules that an implementation must obey if it is to approximate the ideal behavior usefully. This allowed variation in GL behavior imples that two distinct GL implementations may not agree pixel for pixel when presented with the same input even when run on identical framebuffer configurations.

Finally, command names, constants, and types are prefixed in the GL (by **gl**, `GL_`, and `GL`, respectively in `C`) to reduce name clashes with other packages. The prefixes are omitted in this document for clarity.

### Floating-Point Computation

The GL must perform a number of floating-point operations during the course of its operation. We do not specify how floating-point numbers are to be represented or how operations on them are to be performed. We require simply that numbers' floating-point parts contain enough bits and that their exponent fields are large enough so that individual results of floating-point operations are accurate to about 1 part in $10^5$. The maximum representable magnitude of a floating-point number used to represent positional or normal coordinates must be at least $2^{32}$; the maximum representable magnitude for colors or texture coordinates must be at least $2^{10}$. The maximum representable magnitude for all other floating-point values must be at least $2^{32}$. $x \cdot 0 = 0 \cdot x = 0$ for any non-infinite and non-NaN $x$. $1 \cdot x = x \cdot 1 = x$. $x + 0 = 0 + x = x$. $0^0 = 1$. (Occasionally further requirements will be specified.) Most single-precision floating-point formats meet these requirements.

Any representable floating-point value is legal as input to a GL command that requires floating-point data. The result of providing a value that is not a floating-point number to such a command is unspecified, but must not lead to GL interruption or termination. In IEEE arithmetic, for example, providing a negative zero or a denormalized number to a GL command yields predictable results, while providing a NaN or an infinity yields unspecified results.

Some calculations require division. In such cases (including implied divisions required by vector normalizations), a division by zero produces an unspecified result but must not lead to GL interruption or termination.

## 2.2 GL State

The GL maintains considerable state. This document enumerates each state variable and describes how each variable can be changed. For purposes of discussion, state variables are categorized somewhat arbitrarily by their function. Although we describe the operations that the GL performs on the framebuffer, the framebuffer is not a part of GL state.

We distinguish two types of state. The first type of state, called GL *server state*, resides in the GL server. The majority of GL state falls into this category. The second type of state, called GL *client state*, resides in the GL client. Unless otherwise specified, all state referred to in this document is GL server state; GL client state is specifically identified. Each instance of a GL context implies one complete set of GL server state; each connection from a client to a server implies a set of both GL client state and GL server state.

While an implementation of the GL may be hardware dependent, this discussion is independent of the specific hardware on which a GL is implemented. We are therefore concerned with the state of graphics hardware only when it corresponds precisely to GL state.

## 2.3 GL Command Syntax

GL commands are functions or procedures. Various groups of commands perform the same operation but differ in how arguments are supplied to them. To conveniently accommodate this variation, we adopt a notation for describing commands and their arguments.

GL commands are formed from a *name* followed, depending on the particular command, by up to 4 characters. The first character indicates the number of values of the indicated type that must be presented to the command. The second character or character pair indicates the specific type of the arguments: 8-bit integer, 16-bit integer, 32-bit integer, single-precision floating-point, or double-precision floating-point. The final character, if present, is `v`, indicating that the command takes a pointer to an array (a vector) of values rather than a series of individual arguments. Two specific examples come from the **Vertex** command:

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

and

| Letter | Type | Corresponding `C` Type |
|---|---|---|
| **b** | 8-bit integer | `char` |
| **s** | 16-bit integer | `short` |
| **i** | 32-bit integer | `long` |
| **f** | 32-bit floating-point | `float` |
| **d** | 64-bit floating-point | `double` |
| **ub** | 8-bit unsigned integer | `unsigned char` |
| **us** | 16-bit unsigned integer | `unsigned short` |
| **ui** | 32-bit unsigned integer | `unsigned long` |

Table 2.1: Correspondence of command suffix letters to argument types.

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

These examples show the ANSI `C` declarations for these commands. In general, a command declaration has the form*

*rtype* **Name**{$\epsilon$**1234**}{$\epsilon$ **b s i f d ub us ui**}{$\epsilon$**v**}
( *[args ,] T arg1* , . . . , *T argN [, args]* ) ;

*rtype* is the return type of the function. The braces ({}) enclose a series of characters (or character pairs) of which one is selected. $\epsilon$ indicates no character. The arguments enclosed in brackets (*[args ,]* and *[, args]*) may or may not be present. The $N$ arguments *arg1* through *argN* have type $T$, which corresponds to one of the type letters or letter pairs as indicated in Table 2.1 (if there are no letters, then the arguments' type is given explicitly). If the final character is not **v**, then $N$ is given by the digit **1**, **2**, **3**, or **4** (if there is no digit, then the number of arguments is fixed). If the final character is **v**, then only *arg1* is present and it is an array of $N$ values of the indicated type. Finally, we indicate an `unsigned` type by the shorthand of prepending a `u` to the beginning of the type name (so that, for instance, `unsigned char` is abbreviated `uchar`).

For example,

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

*The declarations shown in this document apply to ANSI `C`. Languages such as `C++` and Ada that allow passing of argument type information admit simpler declarations and fewer entry points.

| GL Type | Minimum Precision | Description |
|---|---|---|
| `enum` | 32 bits | Enumerated Value |
| `boolean` | 1 bit | Boolean |
| `bitfield` | 32 bits | Bit Field |
| `byte` | 8 bits | Byte Integer |
| `short` | 16 bits | 16-bit Integer |
| `int` | 32 bits | 32-bit Integer |
| `sizei` | 32 bits | Non-negative Size |
| `ubyte` | 8 bits | Unsigned Byte |
| `ushort` | 16 bits | Unsigned 16-bit Integer |
| `uint` | 32 bits | Unsigned 32-bit Integer |
| `float` | 32 bits | 32-bit Floating-Point Value |
| `clampf` | 32 bits | 32-bit Floating-Point Value clamped to $[0,1]$ when received |
| `double` | 64 bits | 64-bit Floating-Point Value |
| `clampd` | 64 bits | 64-bit Floating-Point Value clamped to $[0,1]$ when received |

Table 2.2: GL argument types. An implementation may use more bits than the number indicated in the table to represent one of these types.

indicates the two declarations

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

while

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

means the two declarations

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

Arguments whose type is fixed (i.e. not indicated by a suffix on the command) are of one of 14 types (or pointers to one of these). These types are summarized in Table 2.2.

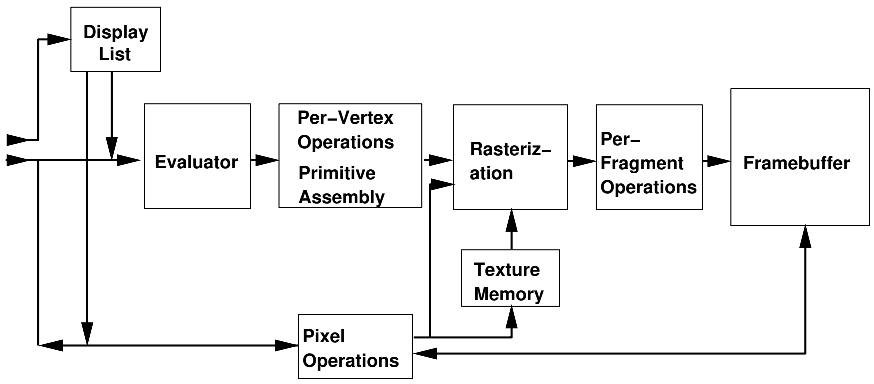


Figure 2.1. Block diagram of the GL.

## 2.4 Basic GL Operation

Figure 2.1 shows a schematic diagram of the GL. Commands enter the GL on the left. Some commands specify geometric objects to be drawn while others control how the objects are handled by the various stages. Most commands may be accumulated in a *display list* for processing by the GL at a later time. Otherwise, commands are effectively sent through a processing pipeline.

The first stage provides an efficient means for approximating curve and surface geometry by evaluating polynomial functions of input values. The next stage operates on geometric primitives described by vertices: points, line segments, and polygons. In this stage vertices are transformed and lit, and primitives are clipped to a viewing volume in preparation for the next stage, rasterization. The rasterizer produces a series of framebuffer addresses and values using a two-dimensional description of a point, line segment, or polygon. Each *fragment* so produced is fed to the next stage that performs operations on individual fragments before they finally alter the framebuffer. These operations include conditional updates into the framebuffer based on incoming and previously stored depth values (to effect depth buffering), blending of incoming fragment colors with stored colors, as well as masking and other logical operations on fragment values.

Finally, there is a way to bypass the vertex processing portion of the pipeline to send a block of fragments directly to the individual fragment operations, eventually causing a block of pixels to be written to the framebuffer; values may also be read back from the framebuffer or copied from one portion of the framebuffer to another. These transfers may include some type of decoding or encoding.

This ordering is meant only as a tool for describing the GL, not as a strict rule of how the GL is implemented, and we present it only as a means to organize the various operations of the GL. Objects such as curved surfaces, for instance, may be transformed before they are converted to polygons.

## 2.5 GL Errors

The GL detects only a subset of those conditions that could be considered errors. This is because in many cases error checking would adversely impact the performance of an error-free program.

The command

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

is used to obtain error information. Each detectable error is assigned a numeric code. When an error is detected, a flag is set and the code is recorded. Further errors, if they occur, do not affect this recorded code. When **GetError** is called, the code is returned and the flag is cleared, so that a further error will again record its code. If a call to **GetError** returns `NO_ERROR`, then there has been no detectable error since the last call to **GetError** (or since the GL was initialized).

To allow for distributed implementations, there may be several flag-code pairs. In this case, after a call to **GetError** returns a value other than `NO_ERROR` each subsequent call returns the non-zero code of a distinct flag-code pair (in unspecified order), until all non-`NO_ERROR` codes have been returned. When there are no more non-`NO_ERROR` error codes, all flags are reset. This scheme requires some positive number of pairs of a flag bit and an integer. The initial state of all flags is cleared and the initial value of all codes is `NO_ERROR`.

Table 2.3 summarizes GL errors. Currently, when an error flag is set, results of GL operation are undefined only if `OUT_OF_MEMORY` has occurred. In other cases, the command generating the error is ignored so that it has no effect on GL state or framebuffer contents. If the generating command

| Error | Description | Offending command ignored? |
|---|---|---|
| `INVALID_ENUM` | `enum` argument out of range | Yes |
| `INVALID_VALUE` | Numeric argument out of range | Yes |
| `INVALID_OPERATION` | Operation illegal in current state | Yes |
| `STACK_OVERFLOW` | Command would cause a stack overflow | Yes |
| `STACK_UNDERFLOW` | Command would cause a stack underflow | Yes |
| `OUT_OF_MEMORY` | Not enough memory left to execute command | Unknown |

Table 2.3: Summary of GL errors

returns a value, it returns zero. If the generating command modifies values through a pointer argument, no change is made to these values. These error semantics apply only to GL errors, not to system errors such as memory access errors. This behavior is the current behavior; the action of the GL in the presence of errors is subject to change.

Two error generation conditions are implicit in the description of every GL command. First, if a command that requires an enumerated value is passed an enumerant that is not one of those specified as allowable for that command, the error `INVALID_ENUM` results. This is the case even if the argument is a pointer to an enumerated value if that value is not allowable for the given command. Second, if a negative number is provided where an argument of type `sizei` is specified, the error `INVALID_VALUE` results.

## 2.6 Begin/End Paradigm

In the GL, most geometric objects are drawn by enclosing a series of coordinate sets that specify vertices and optionally normals, texture coordinates, and colors between **Begin/End** pairs. There are ten geometric objects that are drawn this way: points, line segments, line segment loops, separated line segments, polygons, triangle strips, triangle fans, separated triangles,

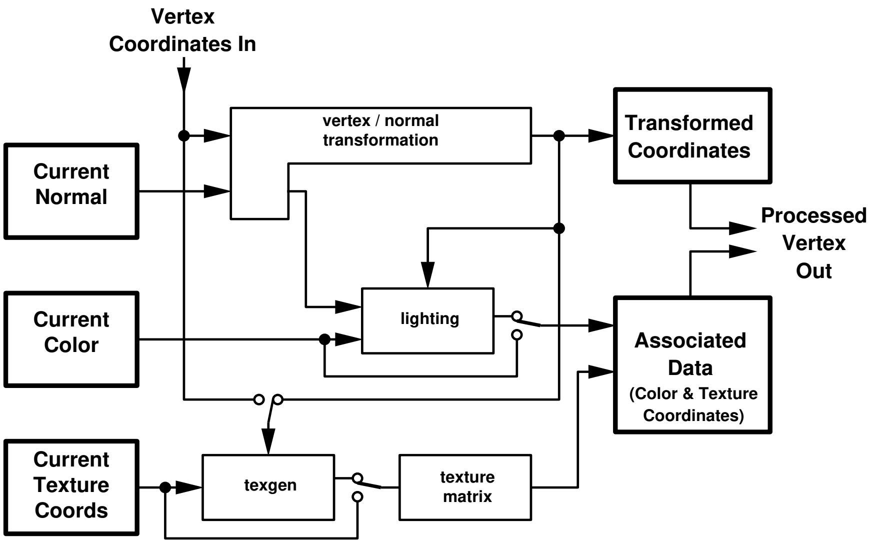


Figure 2.2. Association of current values with a vertex. The heavy lined boxes represent GL state.

quadrilateral strips, and separated quadrilaterals.

Each vertex is specified with two, three, or four coordinates. In addition, a *current normal*, *current texture coordinates*, and *current color* may be used in processing each vertex. Normals are used by the GL in lighting calculations; the current normal is a three-dimensional vector that may be set by sending three coordinates that specify it. Texture coordinates determine how a texture image is mapped onto a primitive.

A color is associated with each vertex as it is specified. This *associated* color is either the current color or a color produced by lighting depending on whether or not lighting is enabled. Texture coordinates are similarly associated with each vertex. Figure 2.2 summarizes the association of auxiliary data with a transformed vertex to produce a *processed vertex*.

The current values are part of GL state. Vertices and normals are transformed, colors may be affected or replaced by lighting, and texture coordinates are transformed and possibly affected by a texture coordinate generation function. The processing indicated for each current value is applied for

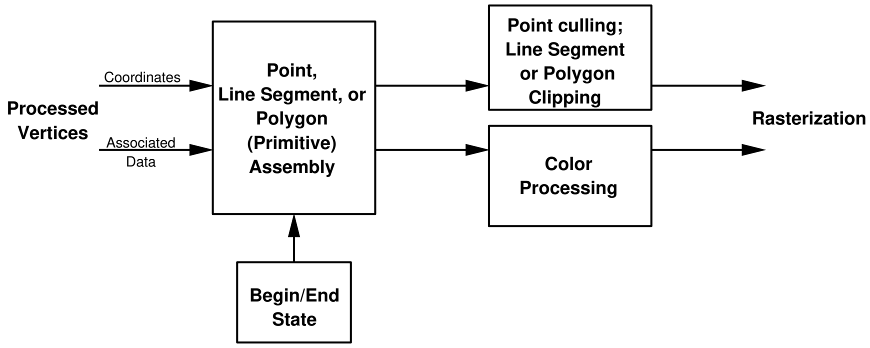


Figure 2.3. Primitive assembly and processing.

each vertex that is sent to the GL.

The methods by which vertices, normals, texture coordinates, and colors are sent to the GL, as well as how normals are transformed and how vertices are mapped to the two-dimensional screen, are discussed later.

Before a color has been assigned to a vertex, the state required by a vertex is the vertex's coordinates, the current normal, and the current texture coordinates. Once color has been assigned, however, the current normal is no longer needed. Because color assignment is done vertex-by-vertex, a processed vertex comprises the vertex's coordinates, its assigned color, and its texture coordinates.

Figure 2.3 shows the sequence of operations that builds a *primitive* (point, line segment, or polygon) from a sequence of vertices. After a primitive is formed, it is clipped to a viewing volume. This may alter the primitive by altering vertex coordinates, texture coordinates, and color. In the case of a polygon primitive, clipping may insert new vertices into the primitive. The vertices defining a primitive to be rasterized have texture coordinates and color associated with them.

### 2.6.1 Begin and End Objects

**Begin** and **End** require one state variable with eleven values: one value for each of the ten possible **Begin**/**End** objects, and one other value indicating that no **Begin/End** object is being processed. The two relevant commands are

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

There is no limit on the number of vertices that may be specified between a **Begin** and an **End**.

**Points.** A series of individual points may be specified by calling **Begin** with an argument value of `POINTS`. No special state need be kept between **Begin** and **End** in this case, since each point is independent of previous and following points.

**Line Strips.** A series of one or more connected line segments is specified by enclosing a series of two or more endpoints within a **Begin/End** pair when **Begin** is called with `LINE_STRIP`. In this case, the first vertex specifies the first segment's start point while the second vertex specifies the first segment's endpoint and the second segment's start point. In general, the $i$th vertex (for $i > 1$) specifies the beginning of the $i$th segment and the end of the $i - 1$st. The last vertex specifies the end of the last segment. If only one vertex is specified between the **Begin/End** pair, then no primitive is generated.

The required state consists of the processed vertex produced from the last vertex that was sent (so that a line segment can be generated from it to the current vertex), and a boolean flag indicating if the current vertex is the first vertex.

**Line Loops.** Line loops, specified with the `LINE_LOOP` argument value to **Begin**, are the same as line strips except that a final segment is added from the final specified vertex to the first vertex. The additional state consists of the processed first vertex.

**Separate Lines.** Individual line segments, each specified by a pair of vertices, are generated by surrounding vertex pairs with **Begin** and **End** when the value of the argument to **Begin** is `LINES`. In this case, the first two vertices between a **Begin** and **End** pair define the first segment, with subsequent pairs of vertices each defining one more segment. If the number of specified vertices is odd, then the last one is ignored. The state required is the same as for lines but it is used differently: a vertex holding the first vertex of the current segment, and a boolean flag indicating whether the current vertex is odd or even (a segment start or end).

**Polygons.** A polygon is described by specifying its boundary as a series of line segments. When **Begin** is called with `POLYGON`, the bounding line segments are specified in the same way as line loops. Depending on the current state of the GL, a polygon may be rendered in one of several ways such as outlining its border or filling its interior. A polygon described with fewer than three vertices does not generate a primitive.

Only convex polygons are guaranteed to be drawn correctly by the GL. If a specified polygon is nonconvex (in particular, if its bounding edges, when projected onto the window, intersect anywhere other than at common endpoints), then the rendered polygon need only lie within the convex hull of the vertices defining its boundary.

The state required to support polygons consists of at least two processed vertices (more than two are never required, although an implementation may use more); this is because a convex polygon can be rasterized as its vertices arrive, before all of them have been specified. The order of the vertices is significant in lighting and polygon rasterization (see sections 2.12.1 and 3.5.1).

**Triangle strips.** A triangle strip is a series of triangles connected along shared edges. A triangle strip is specified by giving a series of defining vertices between a **Begin/End** pair when **Begin** is called with `TRIANGLE_STRIP`. In this case, the first three vertices define the first triangle (and their order is significant, just as for polygons). Each subsequent vertex defines a new triangle using that point along with two vertices from the previous triangle. A **Begin/End** pair enclosing fewer than three vertices, when `TRIANGLE_STRIP` has been supplied to **Begin**, produces no primitive. See Figure 2.4.

The state required to support triangle strips consists of a flag indicating if the first triangle has been completed, two stored processed vertices, (called vertex A and vertex B), and a one bit pointer indicating which stored vertex will be replaced with the next vertex. After a **Begin**(`TRIANGLE_STRIP`), the pointer is initialized to point to vertex A. Each vertex sent between a **Begin/End** pair toggles the pointer. Therefore, the first vertex is stored as vertex A, the second stored as vertex B, the third stored as vertex A, and so on. Any vertex after the second one sent forms a triangle from vertex A, vertex B, and the current vertex (in that order).

**Triangle fans.** A triangle fan is the same as a triangle strip with one exception: each vertex after the first always replaces vertex B of the two stored vertices. The vertices of a triangle fan are enclosed between **Begin** and **End** when the value of the argument to **Begin** is `TRIANGLE_FAN`.

**Separate Triangles.** Separate triangles are specified by placing vertices between **Begin** and **End** when the value of the argument to **Begin** is `TRIANGLES`. In this case, The $3i + 1$st, $3i + 2$nd, and $3i + 3$rd vertices (in that order) determine a triangle for each $i = 0, 1, \ldots, n-1$, where there are

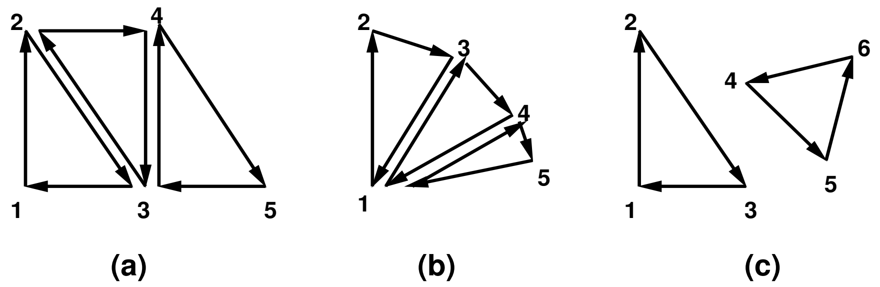


Figure 2.4. (a) A triangle strip. (b) A triangle fan. (c) Independent triangles. The numbers give the sequencing of the vertices between **Begin** and **End**. Note that the in (a) and (b) triangle edge ordering is determined by the first triangle, while in (c) the order of each triangle's edges is independent of the other triangles.

$3n+k$ vertices between the **Begin** and **End**. $k$ is either 0, 1, or 2; if $k$ is not zero, the final $k$ vertices are ignored. For each triangle, vertex A is vertex $3i$ and vertex B is vertex $3i+1$. Otherwise, separate triangles are the same as a triangle strip.

The rules given for polygons also apply to each triangle generated from a triangle strip, triangle fan or from separate triangles.

**Quadrilateral (quad) strips.** Quad strips generate a series of edge-sharing quadrilaterals from vertices appearing between **Begin** and **End**, when **Begin** is called with `QUAD_STRIP`. If the $m$ vertices between the **Begin** and **End** are $v_1, \ldots, v_m$, where $v_j$ is the $j$th specified vertex, then quad $i$ has vertices (in order) $v_{2i}$, $v_{2i+1}$, $v_{2i+3}$, and $v_{2i+2}$ with $i = 0, \ldots, \lfloor m/2 \rfloor$. The state required is thus three processed vertices, to store the last two vertices of the previous quad along with the third vertex (the first new vertex) of the current quad, a flag to indicate when the first quad has been completed, and a one-bit counter to count members of a vertex pair. See Figure 2.5.

A quad strip with fewer than four vertices generates no primitive. If the number of vertices specified for a quadrilateral strip between **Begin** and **End** is odd, the final vertex is ignored.

**Separate Quadrilaterals** Separate quads are just like quad strips except that each group of four vertices, the $4j+1$st, the $4j+2$nd, the $4j+3$rd,

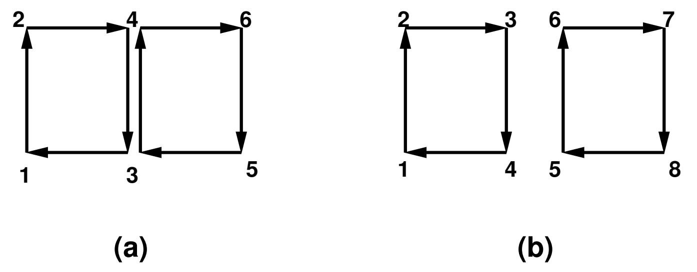


Figure 2.5. (a) A quad strip. (b) Independent quads. The numbers give the sequencing of the vertices between **Begin** and **End**.

and the $4j + 4$th, generate a single quad, for $j = 0, 1, \ldots, n - 1$. The total number of vertices between **Begin** and **End** is $4n + k$, where $0 \leq k \leq 3$; if $k$ is not zero, the final $k$ vertices are ignored. Separate quads are generated by calling **Begin** with the argument value `QUADS`.

The rules given for polygons also apply to each quad generated in a quad strip or from separate quads.

### 2.6.2 Polygon Edges

Each edge of each primitive generated from a polygon, triangle strip, triangle fan, separate triangle set, quadrilateral strip, or separate quadrilateral set, is flagged as either *boundary* or *non-boundary*. These classifications are used during polygon rasterization; some modes affect the interpretation of polygon boundary edges (see section 3.5.4). By default, all edges are boundary edges, but the default flagging of polygons, separate triangles, or separate quadrilaterals may be altered by calling

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

to change the value of a flag bit. If *flag* is zero, then the flag bit is set to `FALSE`; if *flag* is non-zero, then the flag bit is set to `TRUE`,

When **Begin** is supplied with one of the argument values `POLYGON`, `TRIANGLES`, or `QUADS`, each vertex specified within a **Begin** and **End** pair begins an edge. If the edge flag bit is `TRUE`, then each specified vertex begins an edge that is flagged as boundary. If the bit is `FALSE`, then induced edges are flagged as non-boundary.

The state required for edge flagging consists of one current flag bit. Initially, the bit is `TRUE`. In addition, each processed vertex of an assembled polygonal primitive must be augmented with a bit indicating whether or not the edge beginning on that vertex is boundary or non-boundary.

### 2.6.3 GL Commands within Begin/End

The only GL commands that are allowed within any **Begin/End** pairs are the commands for specifying vertex coordinates, vertex color, normal coordinates, and texture coordinates (**Vertex**, **Color**, **Index**, **Normal**, **TexCoord**), **EvalCoord** and **EvalPoint** commands (see section 5.1), commands for specifying lighting material parameters (**Material** commands; see section 2.12.2), display list invocation commands (**CallList** and **CallLists**; see section 5.4), and the **EdgeFlag** command. Executing **Begin** after **Begin** has already been executed but before an **End** is issued generates the `INVALID_OPERATION` error, as does executing **End** without a previous corresponding **Begin**. Executing any other GL command within **Begin/End** results in the error `INVALID_OPERATION`.

## 2.7 Vertex Specification

Vertices are specified by giving their coordinates in two, three, or four dimensions. This is done using one of several versions of the **Vertex** command:

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

A call to any **Vertex** command specifies four coordinates: $x$, $y$, $z$, and $w$. The $x$ coordinate is the first coordinate, $y$ is second, $z$ is third, and $w$ is fourth. A call to **Vertex2** sets the $x$ and $y$ coordinates; the $z$ coordinate is implicitly set to zero and the $w$ coordinate to one. **Vertex3** sets $x$, $y$, and $z$ to the provided values and $w$ to one. **Vertex4** sets all four coordinates, allowing the specification of an arbitrary point in projective three-space. Invoking a **Vertex** command outside of a **Begin/End** pair results in undefined behavior.

Current values are used in associating auxiliary data with a vertex as described in section 2.6. A current value may be changed at any time by issuing an appropriate command. The commands

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

specify the current homogeneous texture coordinates, named $s$, $t$, $r$, and $q$. The **TexCoord1** family of commands set the $s$ coordinate to the provided single argument while setting $t$ and $r$ to 0 and $q$ to 1. Similarly, **TexCoord2** sets $s$ and $t$ to the specified values, $r$ to 0 and $q$ to 1; **TexCoord3** sets $s$, $t$, and $r$, with $q$ set to 1, and **TexCoord4** sets all four texture coordinates.

The current normal is set using

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

The current normal is set to the given coordinates whenever one of these commands is issued. Byte, short, or integer values passed to **Normal** are converted to floating-point values as indicated for the corresponding (signed) type in Table 2.4.

Finally, there are several ways to set the current color. The GL stores both a current single-valued *color index*, and a current four-valued RGBA color. One or the other of these is significant depending as the GL is in *color index mode* or *RGBA mode*. The mode selection is made when the GL is initialized.

The command to set RGBA colors is

void **Color{34}{bsifd ubusui}**( T *components* ) ;
void **Color{34}{bsifd ubusui}v**( T *components* ) ;

The **Color** command has two major variants: **Color3** and **Color4**. The four value versions set all four values. The three value versions set R, G, and B to the provided values; A is set to 1.0. (The conversion of integer color components (R, G, B, and A) to floating-point values is discussed in section 2.12.)

Versions of the **Color** command that take floating-point values accept values nominally between 0.0 and 1.0. 0.0 corresponds to the minimum while 1.0 corresponds to the maximum (machine dependent) value that a component may take on in the framebuffer (see section 2.12 on colors and coloring). Values outside $[0, 1]$ are not clamped.

The command

void **Index{sifd}**( T *index* ) ;
void **Index{sifd}v**( T *index* ) ;

**Index** updates the current (single-valued) color index. It takes one argument, the value to which the current color index should be set. Values outside the (machine-dependent) representable range of color indices are not clamped.

The state required to support vertex specification consists of four floating-point numbers to store the current texture coordinates $s$, $t$, $r$, and $q$, three floating-point numbers to store the three coordinates of the current normal, four floating-point values to store the current RGBA color, and one floating-point value to store the current color index. There is no notion of a current vertex, so no state is devoted to vertex coordinates. The initial values of $s$, $t$, and $r$ of the current texture coordinates are zero; the initial value of $q$ is one. The initial current normal has coordinates $(0, 0, 1)$. The initial RGBA color is $(R, G, B, A) = (1, 1, 1, 1)$. The initial color index is 1.

## 2.8 Rectangles

There is a set of GL commands to support efficient specification of rectangles as two corner vertices.

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

Each command takes either four arguments organized as two consecutive pairs of $(x, y)$ coordinates, or two pointers to arrays each of which contains an $x$ value followed by a $y$ value. The effect of the **Rect** command

**Rect** $(x_1, y_1, x_2, y_2)$;

has exactly the same effect as the following sequence of commands:

**Begin**(POLYGON);
  **Vertex2**$(x_1, y_1)$;
  **Vertex2**$(x_2, y_1)$;
  **Vertex2**$(x_2, y_2)$;
  **Vertex2**$(x_1, y_2)$;
**End**();

The appropriate **Vertex2** command would be invoked depending on which of the **Rect** commands is issued.

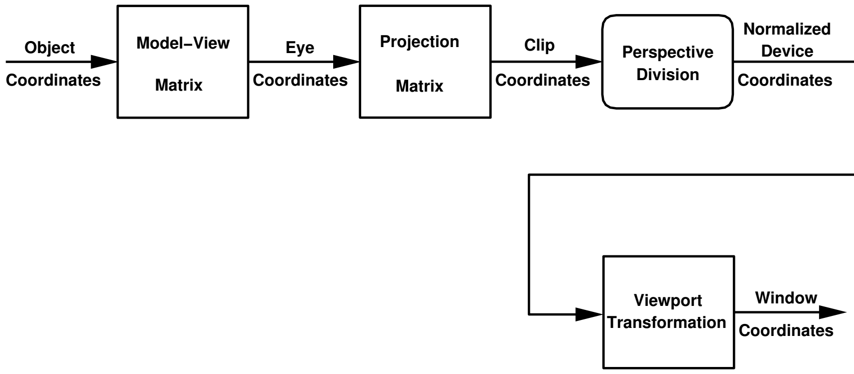


Figure 2.6. Vertex transformation sequence.

## 2.9 Coordinate Transformations

Vertices, normals, and texture coordinates are transformed before their coordinates are used to produce an image in the framebuffer. We begin with a description of how vertex coordinates are transformed and how this transformation is controlled.

Figure 2.6 diagrams the sequence of transformations that are applied to vertices. The vertex coordinates that are presented to the GL are termed *object coordinates*. The *model-view* matrix is applied to these coordinates to yield *eye* coordinates. Then another matrix, called the *projection* matrix, is applied to eye coordinates to yield *clip* coordinates. A perspective division is carried out on clip coordinates to yield *normalized device* coordinates. A final *viewport* transformation is applied to convert these coordinates into *window coordinates*.

Object coordinates, eye coordinates, and clip coordinates are four-dimensional, consisting of $x$, $y$, $z$, and $w$ coordinates (in that order). The model-view and perspective matrices are thus $4 \times 4$.

If a vertex in object coordinates is given by $\begin{pmatrix} x_o \\ y_o \\ z_o \\ w_o \end{pmatrix}$ and the model-view matrix is $M$, then the vertex's eye coordinates are found as

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

Similarly, if $P$ is the projection matrix, then the vertex's clip coordinates are

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

The vertex's normalized device coordinates are then

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

### 2.9.1 Controlling the Viewport

The viewport transformation is determined by the viewport's width and height in pixels, $p_x$ and $p_y$, respectively, and its center $(o_x, o_y)$ (also in pixels). The vertex's window coordinates, $\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix}$, are given by

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

The factor and offset applied to $z_d$ encoded by $n$ and $f$ are set using

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

Each of $n$ and $f$ are clamped to lie within $[0, 1]$, as are all arguments of type `clampd` or `clampf`. $z_w$ is taken to be represented in fixed-point with at least as many bits as there are in the depth buffer of the framebuffer. We assume that the fixed-point representation used represents each value $k/(2^m - 1)$, where $k \in \{0, 1, \ldots, 2^m - 1\}$, as $k$ (e.g. 1.0 is represented in binary as a string of all ones).

Viewport transformation parameters are specified using void **Viewport**( int $x$, int $y$, sizei $w$, sizei $h$ ) ;

where $x$ and $y$ give the $x$ and $y$ window coordinates of the viewport's lower-left corner and $w$ and $h$ give the viewport's width and height, respectively. The viewport parameters shown in the above equations are found from these values as $o_x = x + w/2$ and $o_y = y + h/2$; $p_x = w$, $p_y = h$.

Viewport width and height are clamped to implementation-dependent maximums when specified. The maximum width and height may be found by issuing an appropriate **Get** command (see Chapter 6). The maximum viewport dimensions must be greater than or equal to the visible dimensions of the display being rendered to. INVALID_VALUE is generated if either $w$ or $h$ is negative.

The state required to implement the viewport transformation is 6 integers. In the initial state, $w$ and $h$ are set to the width and height, respectively, of the window into which the GL is to do its rendering. $o_x$ and $o_y$ are set to $w/2$ and $h/2$, respectively. $n$ and $f$ are set to 0.0 and 1.0, respectively.

### 2.9.2 Matrices

The projection matrix and model-view matrix are set and modified with a variety of commands. The affected matrix is determined by the current matrix mode. The current matrix mode is set with

void **MatrixMode**( enum *mode* ) ;

which takes one of the three pre-defined constants TEXTURE, MODELVIEW, or PROJECTION as the argument value. TEXTURE is described later. If the current matrix mode is MODELVIEW, then matrix operations apply to the model-view matrix; if PROJECTION, then they apply to the projection matrix.

The two basic commands for affecting the current matrix are

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

**LoadMatrix** takes a pointer to a $4 \times 4$ matrix stored in column-major order as 16 consecutive floating-point values, i.e. as

$$\begin{pmatrix} a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \\ a_4 & a_8 & a_{12} & a_{16} \end{pmatrix}.$$

(This differs from the standard row-major C ordering for matrix elements. If the standard ordering is used, all of the subsequent transformation equations are transposed, and the columns representing vectors become rows.)

The specified matrix replaces the current matrix with the one pointed to. **MultMatrix** takes the same type argument as **LoadMatrix**, but multiplies the current matrix by the one pointed to and replaces the current matrix with the product. If $C$ is the current matrix and $M$ is the matrix pointed to by **MultMatrix**'s argument, then the resulting current matrix, $C'$, is

$$C' = C \cdot M.$$

The command

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

effectively calls **LoadMatrix** with the identity matrix:

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

There are a variety of other commands that manipulate matrices. **Rotate**, **Translate**, **Scale**, **Frustum**, and **Ortho** manipulate the current matrix. Each computes a matrix and then invokes **MultMatrix** with this matrix. In the case of

`void` **Rotate{fd}**( `T` $\theta$, `T` $x$, `T` $y$, `T` $z$ ) ;

$\theta$ gives an angle of rotation in degrees; the coordinates of a vector $v$ are given by $v = (x\ y\ z)^T$. The computed matrix is a counter-clockwise rotation about the line through the origin with the specified axis when that axis is pointing up (i.e. the right-hand rule determines the sense of the rotation angle). The matrix is thus

$$\begin{pmatrix} & & & 0 \\ & R & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Let $u = v/||v|| = ( x' \quad y' \quad z' )^T$. If

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

then

$$R = uu^T + \cos\theta(I - uu^T) + \sin\theta S.$$

The arguments to

void **Translate{fd}**( T $x$, T $y$, T $z$ ) ;

give the coordinates of a translation vector as $(x\ y\ z)^T$. The resulting matrix is a translation by the specified vector:

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

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

produces a general scaling along the $x$-, $y$-, and $z$- axes. The corresponding matrix is

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

For

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

the coordinates $(l\ b\ -n)^T$ and $(r\ t\ -n)^T$ specify the points on the near clipping plane that are mapped to the lower-left and upper-right corners of the window, respectively (assuming that the eye is located at $(0\ 0\ 0)^T$). $f$ gives the distance from the eye to the far clipping plane. If either $n$ or $f$ is less than or equal to zero, the error INVALID_VALUE results. The corresponding matrix is

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

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

describes a perspective matrix that produces parallel projection. $(l\ b\ -n)^T$ and $(r\ t\ -n)^T$ specify the points on the near clipping plane that are mapped to the lower-left and upper-right corners of the window, respectively. $f$ gives the distance from the eye to the far clipping plane. The corresponding matrix is

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

There is another $4\times4$ matrix that is applied to texture coordinates. This matrix is applied as

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

where the left matrix is the current texture matrix. The matrix is applied to the coordinates resulting from texture coordinate generation (which may simply be the current texture coordinates), and the resulting transformed coordinates become the texture coordinates associated with a vertex. Setting the matrix mode to `TEXTURE` causes the already described matrix operations to apply to the texture matrix.

There is a stack of matrices for each of the matrix modes. For `MODELVIEW` mode, the stack depth is at least 32 (that is, there is a stack of at least 32 model-view matrices). For the other modes, the depth is at least 2. The current matrix in any mode is the matrix on the top of the stack for that mode.

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

pushes the stack down by one, duplicating the current matrix in both the top of the stack and the entry below it.

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

pops the top entry off of the stack, replacing the current matrix with the matrix that was the second entry in the stack. The pushing or popping takes place on the stack corresponding to the current matrix mode. Popping a matrix off a stack with only one entry generates the error `STACK_UNDERFLOW`; pushing a matrix onto a full stack generates `STACK_OVERFLOW`.

The state required to implement transformations consists of a three-valued integer indicating the current matrix mode, a stack of at least two $4 \times 4$ matrices for each of `PROJECTION` and `TEXTURE` with associated stack pointers, and a stack of at least 32 $4 \times 4$ matrices with an associated stack pointer for `MODELVIEW`. Initially, there is only one matrix on each stack, and all matrices are set to the identity. The initial matrix mode is `MODELVIEW`.

### 2.9.3 Normal Transformation

Finally, we consider how the model-view matrix affects normals. Normals are of interest only in eye coordinates, so the rules governing their transformation to other coordinate systems are not examined.

Normals sent to the GL may or may not have unit length. If normalization is enabled, then normals specified with the **Normal3** command are normalized after transformation. Normalization is controlled with

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

and

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

with *target* equal to `NORMALIZE`. This requires one bit of state. The initial state is for normals not to be normalized.

A normal at a point defines a plane at that point. If the normal is $\begin{pmatrix} n_x & n_y & n_z \end{pmatrix}$ and the point is $\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$, then for the point to satisfy the plane equation we must have

$$\begin{pmatrix} n_x & n_y & n_z & q \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = 0$$

whence

$$q = \frac{-\begin{pmatrix} n_x & n_y & n_z \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}}{w}, \qquad w \neq 0$$

or $q = 0$ if $w = 0$. Therefore, if the model-view matrix is $M$, then the transformed plane equation is

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

and the transformed normal is

$$\frac{1}{\sqrt{n_x{}'^2 + n_y{}'^2 + n_z{}'^2}} \begin{pmatrix} n_x{}' \\ n_y{}' \\ n_z{}' \end{pmatrix}. \tag{2.1}$$

If normalization is disabled, then the square root in equation 2.1 is replaced with 1. Otherwise, the square root remains as written.

Because we specify neither the floating-point format nor the means for matrix inversion, we cannot specify behavior in the case of a poorly-conditioned (nearly singular) model-view matrix $M$. In case of an exactly singular matrix, the transformed normal is undefined. If the GL implementation determines that the model-view matrix is uninvertible, then the entries in the inverted matrix are arbitrary. In any case, neither normal transformation nor use of the transformed normal may lead to GL interruption or termination.

### 2.9.4 Generating texture coordinates

Texture coordinates associated with a vertex may either be taken from the current texture coordinates or generated according to a function dependent on vertex coordinates. The command

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

controls texture coordinate generation. *coord* must be one of the constants `S`, `T`, `R`, or `Q`, indicating that the pertinent coordinate is the $s$, $t$, $r$, or $q$ coordinate, respectively. In the first form of the command, *params* is a pointer to an array of values that specify texture generation parameters; in the second form, *param* must be a value specifying a single-valued texture generation parameter. *pname* must be one of the three symbolic constants `TEXTURE_GEN_MODE`, `OBJECT_PLANE`, or `EYE_PLANE`. If *pname* is `TEXTURE_GEN_MODE`, then either *params* points to or *param* is an integer that is one of the symbolic constants `OBJECT_LINEAR`, `EYE_LINEAR`, or `SPHERE_MAP`.

If TEXTURE_GEN_MODE indicates OBJECT_LINEAR, then the generation function for the coordinate indicated by *coord* is

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

$x_o$, $y_o$, $z_o$, and $w_o$ are the object coordinates of the vertex. $p_1, \ldots, p_4$ are specified by calling **TexGen** with *pname* set to OBJECT_PLANE in which case *param* points to an array containing $p_1, \ldots, p_4$. There is a distinct group of plane equation coefficients for each texture coordinate; *coord* indicates the coordinate to which the specified coefficients pertain.

If TEXTURE_GEN_MODE indicates EYE_LINEAR, then the function is

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

where

$$\begin{pmatrix} p'_1 & p'_2 & p'_3 & p'_4 \end{pmatrix} = \begin{pmatrix} p_1 & p_2 & p_3 & p_4 \end{pmatrix} M^{-1}$$

$x_e$, $y_e$, $z_e$, and $w_e$ are the eye coordinates of the vertex. $p_1, \ldots, p_4$ are set by calling **TexGen** with *pname* set to EYE_PLANE in correspondence with setting the coefficients in the OBJECT_PLANE case. $M$ is the model-view matrix in effect when $p_1, \ldots, p_4$ are specified. Computed texture coordinates may be inaccurate or undefined if $M$ is poorly conditioned or singular.

When used with a suitably constructed texture image, calling **TexGen** with TEXTURE_GEN_MODE indicating SPHERE_MAP can simulate the reflected image of a spherical environment on a polygon. SPHERE_MAP texture coordinates are generated as follows. Denote the unit vector pointing from the origin to the vertex (in eye coordinates) by $u$. Denote the current normal, after transformation to eye coordinates, by $n'$. Let $r = \begin{pmatrix} r_x & r_y & r_z \end{pmatrix}^T$, the reflection vector, be given by

$$r = u - 2n'n'^T u,$$

and let $m = 2\sqrt{r_x^2 + r_y^2 + (r_z + 1)^2}$. Then the value assigned to an $s$ coordinate (the first **TexGen** argument value is S) is $s = r_x/m + \frac{1}{2}$; the value assigned to a $t$ coordinate is $t = r_y/m + \frac{1}{2}$. Calling **TexGen** with a *coord* of either R or Q when *pname* indicates SPHERE_MAP generates the error INVALID_ENUM.

A texture coordinate generation function is enabled or disabled using **Enable** and **Disable** with an argument of TEXTURE_GEN_S, TEXTURE_GEN_T, TEXTURE_GEN_R, or TEXTURE_GEN_Q (each indicates the corresponding texture coordinate). When enabled, the specified texture coordinate is computed according to the current EYE_LINEAR, OBJECT_LINEAR or SPHERE_MAP specification, depending on the current setting of TEXTURE_GEN_MODE for that coordinate. When disabled, subsequent vertices will take the indicated texture coordinate from the current texture coordinates.

The state required for texture coordinate generation comprises a three-valued integer for each coordinate indicating coordinate generation mode, and a bit for each coordinate to indicate whether texture coordinate generation is enabled or disabled. In addition, four coefficients are required for the four coordinates for each of EYE_LINEAR and OBJECT_LINEAR. The initial state has the texture generation function disabled for all texture coordinates. The initial values of $p_i$ for $s$ are all 0 except $p_1$ which is one; for $t$ all the $p_i$ are zero except $p_2$, which is 1. The values of $p_i$ for $r$ and $q$ are all 0. These values of $p_i$ apply for both the EYE_LINEAR and OBJECT_LINEAR versions. Initially all texture generation modes are EYE_LINEAR.

## 2.10 Clipping

Primitives are clipped to the *clip volume*. In clip coordinates, the *view volume* is defined by

$$\begin{array}{c} -w_c \leq x_c \leq w_c \\ -w_c \leq y_c \leq w_c \\ -w_c \leq z_c \leq w_c \end{array} .$$

This view volume may be further restricted by as many as $n$ client-defined clip planes to generate the clip volume. ($n$ is an implementation dependent maximum that must be at least 6.) Each client-defined plane specifies a half-space. The clip volume is the intersection of all such half-spaces with the view volume (if there no client-defined clip planes are enabled, the clip volume is the view volume).

A client-defined clip plane is specified with

void **ClipPlane**( enum *p*, double *eqn[4]* ) ;

The value of the first argument, $p$, is a symbolic constant, CLIP_PLANE$i$, where $i$ is an integer between 0 and $n-1$, indicating one of $n$ client-defined clip planes. *eqn* is an array of four double-precision floating-point values. These are the coefficients of a plane equation in object coordinates: $p_1$, $p_2$, $p_3$, and $p_4$ (in that order). The inverse of the current model-view matrix is applied to these coefficients, at the time they are specified, yielding

$$\begin{pmatrix} p'_1 & p'_2 & p'_3 & p'_4 \end{pmatrix} = \begin{pmatrix} p_1 & p_2 & p_3 & p_4 \end{pmatrix} M^{-1}$$

(where $M$ is the current model-view matrix; the resulting plane equation is undefined if $M$ is singular and may be inaccurate if $M$ is poorly-conditioned) to obtain the plane equation coefficients in eye coordinates. All points with eye coordinates $( x_e \quad y_e \quad z_e \quad w_e )^T$ that satisfy

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

lie in the half-space defined by the plane; points that do not satisfy this condition do not lie in the half-space.

Client-defined clip planes are enabled with the generic **Enable** command and disabled with the **Disable** command. The value of the argument to either command is `CLIP_PLANE`$i$ where $i$ is an integer between 0 and $n$; specifying a value of $i$ enables or disables the plane equation with index $i$. The constants obey `CLIP_PLANE`$i$ = `CLIP_PLANE0` + $i$.

If the primitive under consideration is a point, then clipping passes it unchanged if it lies within the clip volume; otherwise, it is discarded. If the primitive is a line segment, then clipping does nothing to it if it lies entirely within the clip volume and discards it if it lies entirely outside the volume. If part of the line segment lies in the volume and part lies outside, then the line segment is clipped and new vertex coordinates are computed for one or both vertices. A clipped line segment endpoint lies on both the original line segment and the boundary of the clip volume.

This clipping produces a value, $0 \leq t \leq 1$, for each clipped vertex. If the coordinates of a clipped vertex are $P$ and the original vertices' coordinates are $P_1$ and $P_2$, then $t$ is given by

$$P = tP_1 + (1 - t)P_2.$$

The value of $t$ is used in color and texture coordinate clipping (section 2.12.8).

If the primitive is a polygon, then it is passed if every one of its edges lies entirely inside the clip volume and either clipped or discarded otherwise. Polygon clipping may cause polygon edges to be clipped, but because polygon connectivity must be maintained, these clipped edges are connected by new edges that lie along the clip volume's boundary. Thus, clipping may require the introduction of new vertices into a polygon. Edge flags are associated with these vertices so that edges introduced by clipping are flagged as boundary (edge flag `TRUE`), and so that original edges of the polygon that become cut off at these vertices retain their original flags.

If it happens that a polygon intersects an edge of the clip volume's boundary, then the clipped polygon must include a point on this boundary edge. This point must lie in the intersection of the boundary edge and the convex hull of the vertices of the original polygon. We impose this requirement because the polygon may not be exactly planar.

A line segment or polygon whose vertices have $w_c$ values of differing signs may generate multiple connected components after clipping. GL implementations are not required to handle this situation. That is, only the portion of the primitive that lies in the region of $w_c > 0$ need be produced by clipping.

Primitives rendered with clip planes must satisfy a complementarity criterion. Suppose a single clip plane with coefficients $( p'_1 \quad p'_2 \quad p'_3 \quad p'_4 )$ (or a number of similarly specified clip planes) is enabled and a series of primitives are drawn. Next, suppose that the original clip plane is respecified with coefficients $( -p'_1 \quad -p'_2 \quad -p'_3 \quad p'_4 )$ (and correspondingly for any other clip planes) and the primitives are drawn again (and the GL is otherwise in the same state). In this case, primitives must not be missing any pixels, nor may any pixels be drawn twice in regions where those primitives are cut by the clip planes.

Clipping requires at least 6 sets of plane equations (each consisting of four double-precision floating-point coefficients) and at least 6 corresponding bits indicating which of these client-defined plane equations are enabled. In the initial state, all client-defined plane equation coefficients are zero and all planes are disabled.

## 2.11 Current Raster Position

The *current raster position* is used by commands that directly affect pixels in the framebuffer. These commands, which bypass vertex transformation and primitive assembly, are described in the next chapter. The current raster position, however, shares some of the characteristics of a vertex.

The current raster position consists of three window coordinates ($x$, $y$, and $z$), a clip coordinate $w$ value, an eye coordinate distance, a valid bit, and associated data consisting of a color and texture coordinates. It is set using one of the **RasterPos** commands:

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

**RasterPos4** takes four values indicating $x$, $y$, $z$, and $w$. **RasterPos3** (or **RasterPos2**) is analogous, but sets only $x$, $y$, and $z$ with $w$ implicitly set to 1 (or only $x$ and $y$ with $z$ implicitly set to 0 and $w$ implicitly set to 1).

The coordinates are treated as if they were specified in a **Vertex** command. The $x$, $y$, $z$, and $w$ coordinates are transformed by the current model-view and perspective matrices. These coordinates, along with current values, are used to generate a color and texture coordinates just as is done for a vertex. The color and texture coordinates so produced replace the color and texture coordinates stored in the current raster position's associated data. The distance from the origin of the eye coordinate system to the vertex as transformed by only the current model-view matrix replaces the current raster distance. This distance can be approximated (see section 3.9).

The transformed coordinates are passed to clipping as if they represented a point. If the "point" is not culled, then the projection to window coordinates is computed (section 2.9) and saved as the current raster position, and the valid bit is set. If the "point" is culled, the current raster position and its associated data become indeterminate and the valid bit is cleared. Figure 2.7 summarizes the behavior of the current raster position.

The current raster position requires five single-precision floating-point values for its $x$, $y$, and $z$ window coordinates, its $w$ clip coordinate, and its eye coordinate distance, a single valid bit, a color (RGBA and color index), and texture coordinates for associated data. In the initial state, the coordinates and texture coordinates are both $(0,0,0,1)$, the eye coordinate distance is 0, the valid bit is set, the associated RGBA color is $(1,1,1,1)$ and the associated color index color is 1. In RGBA mode, the associated color index always has its initial value; in color index mode, the RGBA color always maintains its initial value.

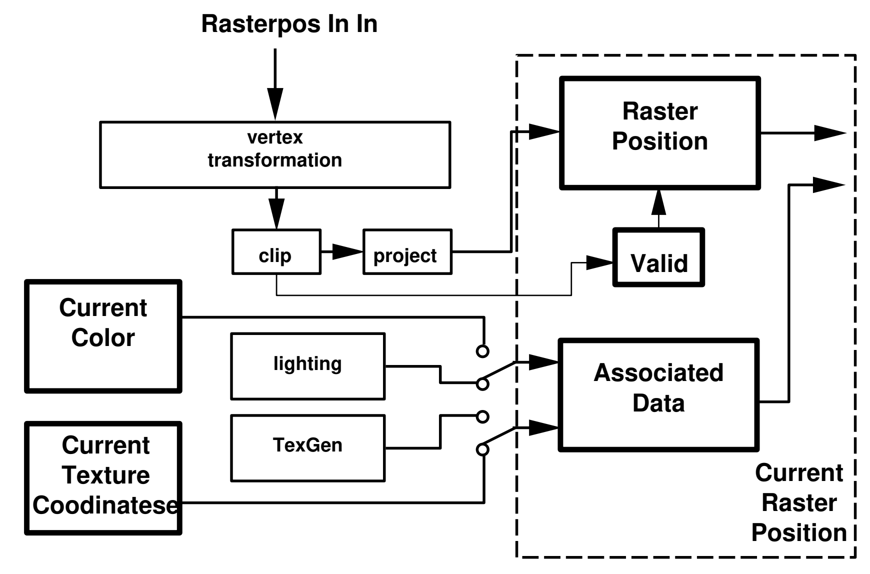


Figure 2.7. The current raster position and how it is set.

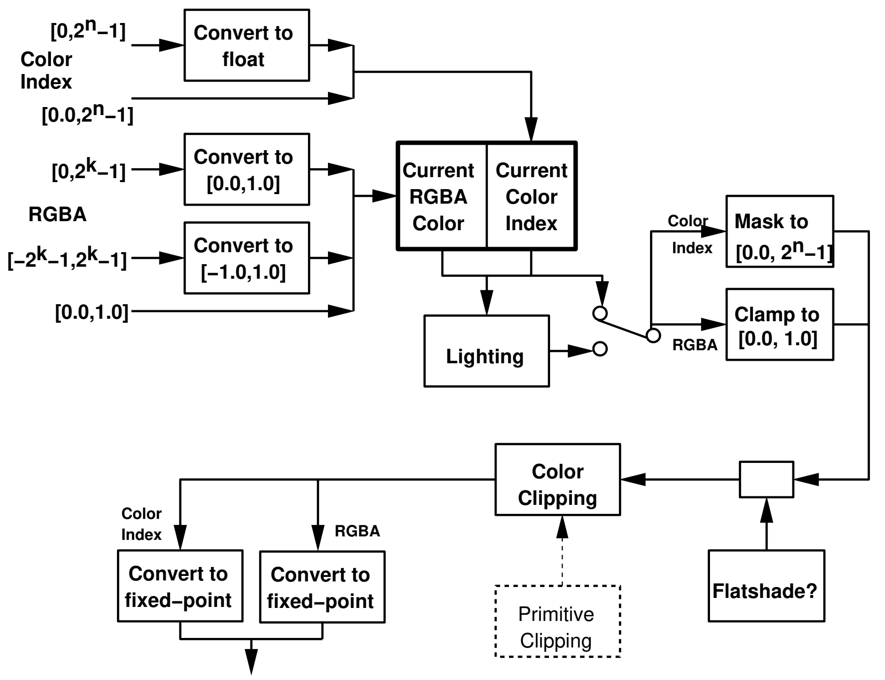


Figure 2.8. Processing of colors. $n$ is the number of bits in a color index; $m$ is the number of bits an R, G, B, or A component. See Table 2.4 for the interpretation of $k$.

## 2.12 Colors and Coloring

Figure 2.8 diagrams the processing of colors before rasterization. Incoming colors arrive in one of several formats. Table 2.4 summarizes the conversions that take place on R, G, B, and A components depending on which version of the **Color** command was invoked to specify the components. As a result of limited precision, some converted values will not be represented exactly. In color index mode, a single-valued color index is not mapped.

Next, lighting, if enabled, produces a color. If lighting is disabled, the current color is used in further processing. After lighting, RGBA colors are clamped to the range $[0, 1]$. A color index is converted to fixed-point

| Command | Element Type | Conversion Formula |
|---|---|---|
| **Colorub** | Unsigned 8-bit integer | $c/(2^8 - 1)$ |
| **Colorb** | Signed 8-bit integer | $(2c + 1)/(2^8 - 1)$ |
| **Colorus** | Unsigned 16-bit integer | $c/(2^{16} - 1)$ |
| **Colors** | Signed 16-bit integer | $(2c + 1)/(2^{16} - 1)$ |
| **Colorui** | Unsigned 32-bit integer | $c/(2^{32} - 1)$ |
| **Colori** | Signed 32-bit integer | $(2c + 1)/(2^{32} - 1)$ |
| **Colorf** | Floating-point | $c$ |

Table 2.4: RGBA Color component conversions. The leftmost column indicates the correspondence between color setting commands and conversions. $c$ represents the value of the component to be converted.

and then its integer portion is masked (see section 2.12.6). After clamping or masking, a primitive may be *flatshaded*, indicating that all vertices of the primitive are to have the same color. Finally, if a primitive is clipped, then colors (and texture coordinates) must be computed at the vertices introduced or modified by clipping.

### 2.12.1 Lighting

GL lighting computes a color for each vertex sent to the GL. This is accomplished by applying an equation defined by a client-specified lighting model to a collection of parameters that can include the vertex coordinates, the coordinates of one or more light sources, the current normal, and parameters defining the characteristics of the light sources and a current material. The following discussion assumes that the GL is in RGBA mode. (Color index lighting is described in section 2.12.5.)

Lighting may be in one of two states:

1. Lighting Off. In this state the color assigned to a vertex is the current color.

2. Lighting On. In this state, a vertex's color is found by computing a value given the current lighting parameters.

Lighting is turned either on or off using the generic **Enable** or **Disable** commands with the symbolic value `LIGHTING`.

### Lighting Operation

A lighting parameter is of one of five types: color, position, direction, real, or boolean. A color parameter consists of four floating-point elements, one for each of R, G, B, and A, in that order. There are no restrictions on the allowable values for these parameters. A position parameter consists of four floating-point coordinates ($x$, $y$, $z$, and $w$) that specify a position in object coordinates ($w$ may, in some cases, be zero, indicating a point at infinity in the direction given by $x$, $y$, and $z$). A direction parameter consists of three floating-point coordinates ($x$, $y$, and $z$) that specify a direction in object coordinates. A real parameter is one floating-point value. The various values and their types are summarized in Table 2.5. The result of a lighting computation is undefined if a value for a parameter is specified that is outside the range given for that parameter in the table.

There are $n$ light sources, indexed by $i = 0, \ldots, n-1$. ($n$ is an implementation dependent maximum that must be at least 8.) Note that the default values for $d_{cli}$ and $s_{cli}$ differ for $i = 0$ and $i > 0$.

Before specifying the way that lighting computes colors, we introduce operators and notation that simplify the expressions involved. If $c_1$ and $c_2$ are colors without alpha where $c_1 = (r_1, g_1, b_1)$ and $c_2 = (r_2, g_2, b_2)$, then define $c_1 * c_2 = (r_1 r_2, g_1 g_2, b_1 b_2)$. Addition of colors is accomplished by addition of the components. Multiplication of colors by a scalar means multiplying each component by that scalar. If $d_1$ and $d_2$ are directions, then define

$$d_1 \odot d_2 = \max\{d_1 \cdot d_2, 0\}.$$

(Directions are taken to have three coordinates.) If $P_1$ and $P_2$ are (homogeneous, with four coordinates) points then let $\overrightarrow{P_1 P_2}$ be the unit vector that points from $P_1$ to $P_2$. Note that if $P_2$ has a zero $w$ coordinate and $P_1$ has non-zero $w$ coordinate, then $\overrightarrow{P_1 P_2}$ is the unit vector corresponding to the direction specified by the $x$, $y$, and $z$ coordinates of $P_2$; if $P_1$ has a zero $w$ coordinate and $P_2$ has a non-zero $w$ coordinate then $\overrightarrow{P_1 P_2}$ is the unit vector that is the negative of that corresponding to the direction specified by $P_1$. If both $P_1$ and $P_2$ have zero $w$ coordinates, then $\overrightarrow{P_1 P_2}$ is the unit vector obtained by normalizing the direction corresponding to $P_2 - P_1$.

If $d$ is an arbitrary direction, then let $\hat{d}$ be the unit vector in $d$'s direction. Let $\|P_1 P_2\|$ be the distance between $P_1$ and $P_2$. Finally, let $V$ be the point corresponding to the vertex being lit, and $n$ be the corresponding normal. Let $P_e$ be the eyepoint ($(0, 0, 0, 1)$ in eye coordinates).

| Parameter | Type | Default Value | Description |
|---|---|---|---|
| Material Parameters | | | |
| $a_{cm}$ | color | $(0.2, 0.2, 0.2, 1.0)$ | ambient color of material |
| $d_{cm}$ | color | $(0.8, 0.8, 0.8, 1.0)$ | diffuse color of material |
| $s_{cm}$ | color | $(0.0, 0.0, 0.0, 1.0)$ | specular color of material |
| $e_{cm}$ | color | $(0.0, 0.0, 0.0, 1.0)$ | emissive color of material |
| $s_{rm}$ | real | 0.0 | specular exponent (range: $[0.0, 128.0]$) |
| $a_m$ | real | 0.0 | ambient color index |
| $d_m$ | real | 1.0 | diffuse color index |
| $s_m$ | real | 1.0 | specular color index |
| Light Source Parameters | | | |
| $a_{cli}$ | color | $(0.0, 0.0, 0.0, 1.0)$ | ambient intensity of light $i$ |
| $d_{cli}(i = 0)$<br>$d_{cli}(i > 0)$ | color<br>color | $(1.0, 1.0, 1.0, 1.0)$<br>$(0.0, 0.0, 0.0, 1.0)$ | diffuse intensity of light 0<br>diffuse intensity of light $i$ |
| $s_{cli}(i = 0)$<br>$s_{cli}(i > 0)$ | color<br>color | $(1.0, 1.0, 1.0, 1.0)$<br>$(0.0, 0.0, 0.0, 1.0)$ | specular intensity of light 0<br>specular intensity of light $i$ |
| $P_{pli}$ | position | $(0.0, 0.0, 1.0, 0.0)$ | position of light $i$ |
| $s_{dli}$ | direction | $(0.0, 0.0, -1.0)$ | direction of spotlight for light $i$ |
| $s_{rli}$ | real | 0.0 | spotlight exponent for light $i$ (range: $[0.0, 128.0]$) |
| $c_{rli}$ | real | 180.0 | spotlight cutoff angle for light $i$ (range: $[0.0, 90.0]$, $180.0$) |
| $k_{0i}$ | real | 1.0 | constant attenuation factor for light i (range: $[0.0, \infty)$) |
| $k_{1i}$ | real | 0.0 | linear attenuation factor for light i (range: $[0.0, \infty)$) |
| $k_{2i}$ | real | 0.0 | quadratic attenuation factor for light i (range: $[0.0, \infty)$) |
| Lighting Model Parameters | | | |
| $a_{cs}$ | color | $(0.2, 0.2, 0.2, 1.0)$ | ambient color of scene |
| $v_{bs}$ | boolean | `FALSE` | viewer assumed to be at $(0, 0, 0)$ in eye coordinates (`TRUE`) or $(0, 0, \infty)$ (`FALSE`) |
| $t_{bs}$ | boolean | `FALSE` | use two-sided lighting mode |

Table 2.5: Summary of lighting parameters. The range of individual color components is $(-\infty, +\infty)$.

The color $c$ produced by lighting a vertex is given by

$$\begin{aligned} c &= e_{cm} \\ &+ a_{cm} * a_{cs} \\ &+ \sum_{i=0}^{n-1} (att_i)(spot_i)\,[a_{cm} * a_{cli} \\ &\qquad\qquad + (n \odot \overrightarrow{VP}_{pli}) d_{cm} * d_{cli} \\ &\qquad\qquad + (f_i)(n \odot \hat{h}_i)^{s_{rm}} s_{cm} * s_{cli}] \end{aligned}$$

where

$$f_i = \begin{cases} 1, & n \odot \overrightarrow{VP}_{pli} \neq 0, \\ 0, & \text{otherwise}, \end{cases} \tag{2.2}$$

$$h_i = \begin{cases} \overrightarrow{VP}_{pli} + \overrightarrow{VP}_e, & v_{bs} = TRUE, \\ \overrightarrow{VP}_{pli} + \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}^T, & v_{bs} = FALSE, \end{cases} \tag{2.3}$$

$$att_i = \begin{cases} \dfrac{1}{k_{0i} + k_{1i}\|VP_{pli}\| + k_{2i}\|VP_{pli}\|^2}, & \text{if } P_{pli}\text{'s } w \neq 0, \\ 1.0, & \text{otherwise}, \end{cases} \tag{2.4}$$

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

All computations are carried out in eye coordinates.

The value of A produced by lighting is the alpha value associated with $d_{cm}$. Results of lighting are undefined if the $w_e$ coordinate ($w$ in eye coordinates) of $V$ is zero.

Lighting may operate in *two-sided* mode ($t_{bs} = TRUE$), in which a *front* color is computed with one set of material parameters (the *front material*) and a *back* color is computed with a second set of material parameters (the *back material*). This second computation replaces $n$ with $-n$. If $t_{bs} =$

$FALSE$, then the back color and front color are both assigned the color computed using the front material with $n$.

The selection between back color and front color depends on the primitive of which the vertex being lit is a part. If the primitive is a point or a line segment, the front color is always selected. If it is a polygon, then the selection is based on the sign of the (clipped or unclipped) polygon's signed area computed in window coordinates. One way to compute this area is

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

where $x_w^i$ and $y_w^i$ are the $x$ and $y$ window coordinates of the $i$th vertex of the $n$-vertex polygon (vertices are numbered starting at zero for purposes of this computation) and $i \oplus 1$ is $(i+1)$ mod $n$. The interpretation of the sign of this value is controlled with

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

Setting *dir* to `CCW` (corresponding to counter-clockwise orientation of the projected polygon in window coordinates) indicates that if $a \leq 0$, then the color of each vertex of the polygon becomes the back color computed for that vertex while if $a > 0$, then the front color is selected. If *dir* is `CW`, then $a$ is replaced by $-a$ in the above inequalities. This requires one bit of state; initially, it indicates `CCW`.

### 2.12.2 Lighting Parameter Specification

Lighting parameters are divided into three categories: material parameters, light source parameters, and lighting model parameters (see Table 2.5). Sets of lighting parameters are specified with

`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` **LightModel{if}v**( `enum` *pname*, `T` *params* ) ;

*pname* is a symbolic constant indicating which parameter is to be set (see Table 2.6). In the vector versions of the commands, *params* is a pointer to a group of values to which to set the indicated parameter. The number of values pointed to depends on the parameter being set. In the non-vector versions, *param* is a value to which to set a single-valued parameter. (If *param* corresponds to a multi-valued parameter, the error INVALID_ENUM results.) For the **Material** command, *face* must be one of FRONT, BACK, or FRONT_AND_BACK, indicating that the property *name* of the front or back material, or both, respectively, should be set. In the case of **Light**, *light* is a symbolic constant of the form LIGHT$i$, indicating that light $i$ is to have the specified parameter set. The constants obey LIGHT$i$ = LIGHT0 + $i$.

Table 2.6 gives, for each of the three parameter groups, the correspondence between the pre-defined constant names and their names in the lighting equations, along with the number of values that must be specified with each. Color parameters specified with **Material** and **Light** are converted to floating-point values (if specified as integers) as indicated in Table 2.4 for signed integers. The error INVALID_VALUE occurs if a specified lighting parameter lies outside the allowable range given in Table 2.5. (The symbol "$\infty$" indicates the maximum representable magnitude for the indicated type.)

The current model-view matrix is applied to the position parameter indicated with **Light** for a particular light source when that position is specified. These transformed values are the values used in the lighting equation. The spotlight direction is transformed when it is specified into a value of $s_{dli}$ using the rules given for transforming normals at the end of section 2.9.3.

An individual light is enabled or disabled by calling **Enable** or **Disable** with the symbolic value LIGHT$i$ ($i$ is in the range 0 to $n-1$, where $n$ is the implementation-dependent number of lights). If light $i$ is disabled, the $i$th term in the lighting equation is effectively removed from the summation.

### 2.12.3 ColorMaterial

It is possible to attach one or more material properties to the current color, so that they continuously track its component values. This behavior is enabled and disabled by calling **Enable** or **Disable** with the symbolic value COLOR_MATERIAL.

The command that controls which of these modes is selected is

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

*face* is one of FRONT, BACK, or FRONT_AND_BACK, indicating whether the front material, back material, or both are affected by the current color. *mode* is one of EMISSION, AMBIENT, DIFFUSE, SPECULAR, or AMBIENT_AND_DIFFUSE and

| Parameter | Name | Number of values |
|---|---|---|
| Material Parameters (**Material**) | | |
| $a_{cm}$ | `AMBIENT` | 4 |
| $d_{cm}$ | `DIFFUSE` | 4 |
| $a_{cm}, d_{cm}$ | `AMBIENT_AND_DIFFUSE` | 4 |
| $s_{cm}$ | `SPECULAR` | 4 |
| $e_{cm}$ | `EMISSION` | 4 |
| $s_{rm}$ | `SHININESS` | 1 |
| $a_m, d_m, s_m$ | `COLOR_INDEXES` | 3 |
| Light Source Parameters (**Light**) | | |
| $a_{cli}$ | `AMBIENT` | 4 |
| $d_{cli}$ | `DIFFUSE` | 4 |
| $s_{cli}$ | `SPECULAR` | 4 |
| $P_{pli}$ | `POSITION` | 4 |
| $s_{dli}$ | `SPOT_DIRECTION` | 3 |
| $s_{rli}$ | `SPOT_EXPONENT` | 1 |
| $c_{rli}$ | `SPOT_CUTOFF` | 1 |
| $k_0$ | `CONSTANT_ATTENUATION` | 1 |
| $k_1$ | `LINEAR_ATTENUATION` | 1 |
| $k_2$ | `QUADRATIC_ATTENUATION` | 1 |
| Lighting Model Parameters (**LightModel**) | | |
| $a_{cs}$ | `LIGHT_MODEL_AMBIENT` | 4 |
| $v_{bs}$ | `LIGHT_MODEL_LOCAL_VIEWER` | 1 |
| $t_{bs}$ | `LIGHT_MODEL_TWO_SIDE` | 1 |

Table 2.6: Correspondence of lighting parameter symbols to names. `AMBIENT_AND_DIFFUSE` is used to set $a_{cm}$ and $d_{cm}$ to the same value.

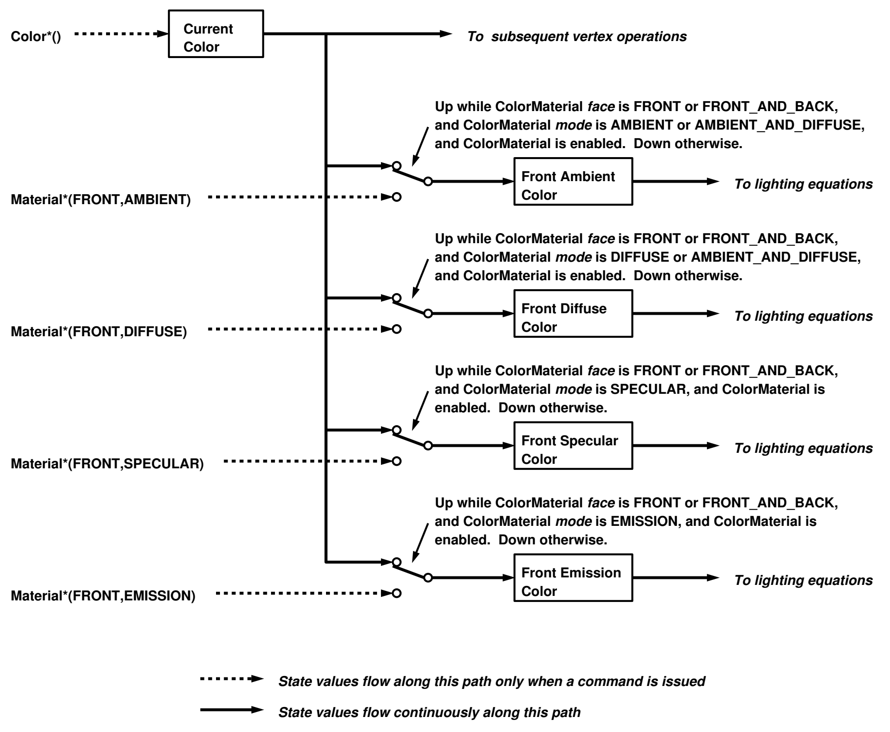


Figure 2.9. **ColorMaterial** operation. Material properties are continuously updated from the current color while **ColorMaterial** is enabled and has the appropriate mode. Only the front material properties are included in this figure. The back material properties are treated identically.

specifies which material property or properties track the current color. If *mode* is `EMISSION`, `AMBIENT`, `DIFFUSE`, or `SPECULAR`, then the value of $e_{cm}$, $a_{cm}$, $d_{cm}$ or $s_{cm}$, respectively, will track the current color. If *mode* is `AMBIENT_AND_DIFFUSE`, both $a_{cm}$ and $d_{cm}$ track the current color. The replacements made to material properties are permanent; the replaced values remain until changed by either sending a new color or by setting a new material value when **ColorMaterial** is not currently enabled to override that particular value. When `COLOR_MATERIAL` is enabled, the indicated parameter or parameters always track the current color. For instance, calling

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

while `COLOR_MATERIAL` is enabled sets the front material $a_{cm}$ to the value of the current color.

### 2.12.4 Lighting State

The state required for lighting consists of all of the lighting parameters (front and back material parameters, lighting model parameters, and at least 8 sets of light parameters), a bit indicating whether a back color distinct from the front color should be computed, at least 8 bits to indicate which lights are enabled, a five-valued variable indicating the current **ColorMaterial** mode, a bit indicating whether or not `COLOR_MATERIAL` is enabled, and a single bit to indicate whether lighting is enabled or disabled. In the initial state, all lighting parameters have their default values. Back color evaluation does not take place, **ColorMaterial** is `FRONT_AND_BACK` and `AMBIENT_AND_DIFFUSE`, and both lighting and `COLOR_MATERIAL` are disabled.

### 2.12.5 Color Index Lighting

A simplified lighting computation applies in color index mode that uses many of the parameters controlling RGBA lighting, but none of the RGBA material parameters. First, the RGBA diffuse and specular intensities of light $i$ ($d_{cli}$ and $s_{cli}$, respectively) determine color index diffuse and specular light intensities, $d_{li}$ and $s_{li}$ from

$$d_{li} = (.30)R(d_{cli}) + (.59)G(d_{cli}) + (.11)B(d_{cli})$$

and

$$s_{li} = (.30)R(s_{cli}) + (.59)G(s_{cli}) + (.11)B(s_{cli}).$$

$R(x)$ indicates the R component of the color $x$ and similarly for $G(x)$ and $B(x)$.

Next, let

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

where $att_i$ and $spot_i$ are given by equations 2.4 and 2.5, respectively, and $f_i$ and $\hat{h}_i$ are given by equations 2.2 and 2.3, respectively. Let $s' = \min\{s, 1\}$. Finally, let

$$d = \sum_{i=0}^{n} (att_i)(spot_i)(d_{li})(n \odot \overrightarrow{VP}_{pli}).$$

Then color index lighting produces a value $c$, given by

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

The final color index is

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

The values $a_m$, $d_m$ and $s_m$ are material properties described in Tables 2.5 and 2.6. Any ambient light intensities are incorporated into $a_m$. As with RGBA lighting, disabled lights cause the corresponding terms from the summations to be omitted. The interpretation of $t_{bs}$ and the calculation of front and back colors is carried out as has already been described for RGBA lighting.

The values $a_m$, $d_m$, and $s_m$ are set with **Material** using *prop* of `COLOR_INDEXES`. Their initial values are 0, 1, and 1, respectively. The additional state consists of three floating-point values. These values have no effect on RGBA lighting.

### 2.12.6 Clamping or Masking

After lighting, RGBA colors are clamped to the range $[0, 1]$. For a color index, the index is first converted to fixed-point with an unspecified number of bits to the right of the binary point; the nearest fixed-point value is selected. Then, the bits to the right of the binary point are left alone while the integer portion is masked (bitwise ANDed) with $2^n - 1$, where $n$ is the number of bits in a color in the color index buffer (buffers are discussed in chapter 4).

| Primitive type of polygon $i$ | Vertex |
|---|---|
| single polygon ($i \equiv 1$) | 1 |
| triangle strip | $i+2$ |
| triangle fan | $i+2$ |
| independent triangle | $3i$ |
| quad strip | $2i+2$ |
| independent quad | $4i$ |

Table 2.7: Polygon flatshading color selection. The color used for flatshading the $i$th polygon generated by the indicated **Begin/End** type is the current color (if lighting is disabled) in effect when the indicated vertex is specified. If lighting is enabled, the color is produced by lighting the indicated vertex. Vertices are numbered 1 through $n$, where $n$ is the number of vertices between the **Begin/End** pair.

### 2.12.7 Flatshading

A primitive may be *flatshaded*, meaning that all vertices of the primitive are assigned the same color. This color is the color of the vertex that spawned the primitive. For a point, this is the color associated with the point. For a line segment, it is the color of the second (final) vertex of the segment. For a polygon, the selected color depends on how the polygon was generated. Table 2.7 summarizes the possibilities.

Flatshading is controlled by

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

*mode* value must be either of the symbolic constants `SMOOTH` or `FLAT`. If *mode* is `SMOOTH` (the initial state), vertex colors are treated individually. If *mode* is `FLAT`, flatshading is turned on. **ShadeModel** thus requires one bit of state.

### 2.12.8 Color and Texture Coordinate Clipping

After lighting, clamping or masking and possible flatshading, colors are clipped. If the color is associated with a vertex that lies within the clip volume, it is unaffected by clipping. If a primitive is clipped, however, the colors assigned to vertices produced by clipping are clipped colors.

Let the color assigned to the two vertices $P_1$ and $P_2$ of an unclipped edge be $c_1$ and $c_2$. The value of $t$ (Section 2.10) for a clipped point $P$ is used to obtain the color associated with $P$ as

$$c = tc_1 + (1 - t)c_2.$$

(For a color index color, multiplying a color by a scalar means multiplying the index by the scalar. For an RGBA color, it means multiplying each of R, G, B, and A by the scalar.) Polygon clipping may create a clipped vertex along an edge of the clip volume's boundary. This situation is handled by noting that polygon clipping proceeds by clipping against one plane of the clip volume's boundary at a time. Color clipping is done in the same way, so that clipped points always occur at the intersection of polygon edges (possibly already clipped) with the clip volume's boundary.

Texture coordinates must also be clipped when a primitive is clipped. The method is exactly analogous to that used for color clipping.

### 2.12.9 Final Color Processing

For an RGBA color, each color component (which lies in [0,1]) is converted (by rounding to nearest) to a fixed-point value with $m$ bits. We assume that the fixed-point representation used represents each value $k/(2^m - 1)$, where $k \in \{0, 1, \ldots, 2^m - 1\}$, as $k$ (e.g. 1.0 is represented in binary as a string of all ones). $m$ must be at least as large as the number of bits in the corresponding component of the framebuffer. If the framebuffer does not contain an A component, then $m$ must be at least 2 for A. A color index is converted (by rounding to nearest) to a fixed-point value with at least as many bits as there are in the color index portion of the framebuffer.

Because a number of the form $k/(2^m - 1)$ may not be represented exactly as a limited-precision floating-point quantity, we place a further requirement on the fixed-point conversion of RGBA components. Suppose that lighting is disabled, the color associated with a vertex has not been clipped, and one of **Colorub**, **Colorus**, or **Colorui** was used to specify that color. When these conditions are satisfied, an RGBA component must convert to a value that matches the component as specified in the **Color** command: if $m$ is less than the number of bits $b$ with which the component was specified, then the converted value must equal the most significant $m$ bits of the specified value; otherwise, the most significant $b$ bits of the converted value must equal the specified value.

# Chapter 3

# Rasterization

Rasterization is the process by which a primitive is converted to a two-dimensional image. Each point of this image contains such information as color and depth. Thus, rasterizing a primitive consists of two parts. The first is to determine which squares of an integer grid in window coordinates are occupied by the primitive. The second is assigning a color and a depth value to each such square. The results of this process are passed on to the next stage of the GL (per-fragment operations), which uses the information to update the appropriate locations in the framebuffer. Figure 3.1 diagrams the rasterization process.

A grid square along with its parameters of assigned color, $z$ (depth), and texture coordinates is called a *fragment*; the parameters are collectively dubbed the fragment's *associated data*. A fragment is located by its lower-left corner, which lies on integer grid coordinates. Rasterization operations also refer to a fragment's *center*, which is offset by $(1/2, 1/2)$ from its lower-left corner (and so lies on half-integer coordinates).

Grid squares need not actually be square in the GL. Rasterization rules are not affected by the actual aspect ratio of the grid squares. Display of non-square grids, however, will cause rasterized points and line segments to appear fatter in one direction than the other. We assume that fragments are square, since it simplifies antialiasing and texturing.

Several factors affect rasterization. Lines and polygons may be stippled. Points may be given differing diameters and line segments differing widths. A point, line segment, or polygon may be antialiased.

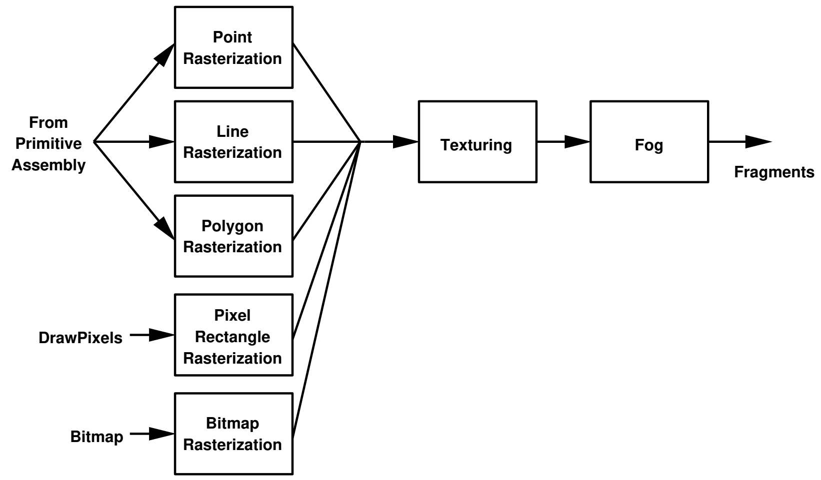


Figure 3.1. Rasterization.

## 3.1 Invariance

Consider a primitive $p'$ obtained by translating a primitive $p$ through an offset $(x, y)$ in window coordinates, where $x$ and $y$ are integers. As long as neither $p'$ nor $p$ is clipped, it must be the case that each fragment $f'$ produced from $p'$ is identical to a corresponding fragment $f$ from $p$ except that the center of $f'$ is offset by $(x, y)$ from the center of $f$.

## 3.2 Antialiasing

Antialiasing of a point, line, or polygon is effected in one of two ways depending on whether the GL is in RGBA or color index mode.

In RGBA mode, the R, G, and B values of the rasterized fragment are left unaffected, but the A value is multiplied by a floating-point value in the range $[0, 1]$ that describes a fragment's screen pixel coverage. The per-fragment stage of the GL can be set up to use the A value to blend the incoming fragment with the corresponding pixel already present in the framebuffer.

In color index mode, the least significant $b$ bits (to the left of the binary point) of the color index are used for antialiasing; $b = \min\{4, m\}$, where $m$ is the number of bits in the color index portion of the framebuffer. The antialiasing process sets these $b$ bits based on the fragment's coverage value: the bits are set to zero for no coverage and to all ones for complete coverage.

The details of how antialiased fragment coverage values are computed are difficult to specify in general. The reason is that high-quality antialiasing may take into account perceptual issues as well as characteristics of the monitor on which the contents of the framebuffer are displayed. Such details cannot be addressed within the scope of this document. Further, the coverage value computed for a fragment of some primitive may depend on the primitive's relationship to a number of grid squares neighboring the one corresponding to the fragment, and not just on the fragment's grid square. Another consideration is that accurate calculation of coverage values may be computationally expensive; consequently we allow a given GL implementation to approximate true coverage values by using a fast but not entirely accurate coverage computation.

In light of these considerations, we chose to specify the behavior of exact antialiasing in the prototypical case that each displayed pixel is a perfect square of uniform intensity. The square is called a *fragment square* and has lower left corner $(x, y)$ and upper right corner $(x+1, y+1)$. We recognize that this simple box filter may not produce the most favorable antialiasing results, but it provides a simple, well-defined model.

A GL implementation may use other methods to perform antialiasing, subject to the following conditions:

1. If $f_1$ and $f_2$ are two fragments, and the portion of $f_1$ covered by some primitive is a subset of the corresponding portion of $f_2$ covered by the primitive, then the coverage computed for $f_1$ must be less than or equal to that computed for $f_2$.

2. The coverage computation for a fragment $f$ must be local: it may depend only on $f$'s relationship to the boundary of the primitive being rasterized. It may not depend on $f$'s $x$ and $y$ coordinates.

Another property that is desirable, but not required, is:

3. The sum of the coverage values for all fragments produced by rasterizing a particular primitive must be constant, independent of any rigid motions in window coordinates, as long as none of those fragments lies along window edges.

In some implementations, varying degrees of antialiasing quality may be obtained by providing GL hints (section 5.6), allowing a user to make an image quality versus speed tradeoff.

## 3.3 Points

The rasterization of points is controlled with

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

*size* specifies the width or diameter of a point. The default value is 1.0. A value less than or equal to zero results in the error `INVALID_VALUE`.

Point antialiasing is enabled or disabled by calling **Enable** or **Disable** with the symbolic constant `POINT_SMOOTH`. The default state is for point antialiasing to be disabled.

In the default state, a point is rasterized by truncating its $x_w$ and $y_w$ coordinates (recall that the subscripts indicate that these are $x$ and $y$ window coordinates) to integers. This $(x, y)$ address, along with the data associated with the vertex corresponding to the point, is sent as a single fragment to the per-fragment stage of the GL.

The effect of a point width other than 1.0 depends on the state of point antialiasing. If antialiasing is disabled, the actual width is determined by rounding the supplied width to the nearest integer, then clamping it to the implementation-dependent maximum non-antialiased point width. Though this implementation-dependent value cannot be queried, it must be no less than the implementation-dependent maximum antialiased point width, rounded to the nearest integer value, and in any event no less than 1. If rounding the specified width results in the value 0, then it is as if the value were 1. If the resulting width is odd, then the point

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

is computed from the vertex's $x_w$ and $y_w$, and a square grid of the odd width centered at $(x, y)$ defines the centers of the rasterized fragments (recall that fragment centers lie at half-integer window coordinate values). If the width is even, then the center point is

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

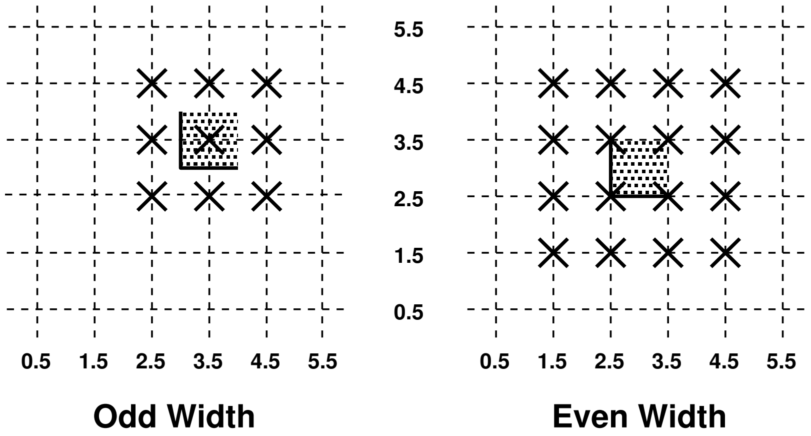


Figure 3.2. Rasterization of non-antialiased wide points. The crosses show fragment centers produced by rasterization for any point that lies within the shaded region. The dotted grid lines lie on half-integer coordinates.

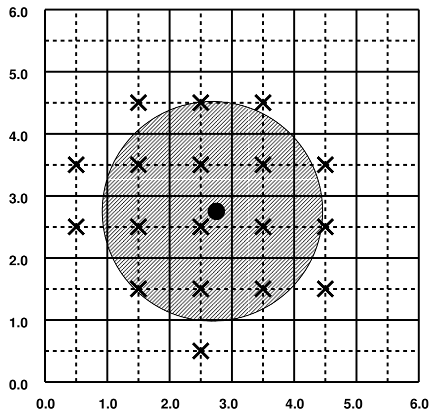


Figure 3.3. Rasterization of antialiased wide points. The black dot indicates the point to be rasterized. The shaded region has the specified width. The X marks indicate those fragment centers produced by rasterization. A fragment's computed coverage value is based on the portion of the shaded region that covers the corresponding fragment square. Solid lines lie on integer coordinates.

the rasterized fragment centers are the half-integer window coordinate values within the square of the even width centered on $(x, y)$. See Figure 3.2.

All fragments produced in rasterizing a non-antialiased point are assigned the same associated data, which are those of the vertex corresponding to the point.

If antialiasing is enabled, then point rasterization produces a fragment for each fragment square that intersects the region lying within the circle having diameter equal to the current point width and centered at the point's $(x_w, y_w)$ (Figure 3.3). The coverage value for each fragment is the window coordinate area of the intersection of the circular region with the corresponding fragment square (but see section 3.2). This value is saved and used in the final step of rasterization (section 3.10). The data associated with each fragment are otherwise the data associated with the point being rasterized.

Not all widths need be supported when point antialiasing is on, but the width 1.0 must be provided. If an unsupported width is requested, the nearest supported width is used instead. The range of supported widths and the width of evenly-spaced gradations within that range are implementation dependent. The range and gradations may be obtained using the query mechanism described in Chapter 6. If, for instance, the width range is from 0.1 to 2.0 and the gradation width is 0.1, then the widths $0.1, 0.2, \ldots, 1.9, 2.0$ are supported.

### 3.3.1 Point Rasterization State

The state required to control point rasterization consists of the floating-point point width and a bit indicating whether or not antialiasing is enabled.

## 3.4 Line Segments

A line segment results from a line strip **Begin/End** object, a line loop, or a series of separate line segments. Line segment rasterization is controlled by several variables. Line width, which may be set by calling

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

with an appropriate positive floating-point width, controls the width of rasterized line segments. The default width is 1.0. Values less than or equal to 0.0 generate the error `INVALID_VALUE`. Antialiasing is controlled with **Enable** and **Disable** using the symbolic constant `LINE_SMOOTH`. Finally, line segments may be stippled. Stippling is controlled by a GL command that sets a *stipple pattern* (see below).

### 3.4.1 Basic Line Segment Rasterization

Line segment rasterization begins by characterizing the segment as either *$x$-major* or *$y$-major*. $x$-major line segments have slope in the closed interval $[-1, 1]$; all other line segments are $y$-major (slope is determined by the segment's endpoints). We shall specify rasterization only for $x$-major segments except in cases where the modifications for $y$-major segments are not self-evident.

Ideally, the GL uses a "diamond-exit" rule to determine those fragments that are produced by rasterizing a line segment. For each fragment $f$ with center at window coordinates $x_f$ and $y_f$, define a diamond-shaped region that is the intersection of four half-planes:

$$R_f = S_{f1} \cap S_{f2} \cap S_{f3} \cap S_{f4}$$

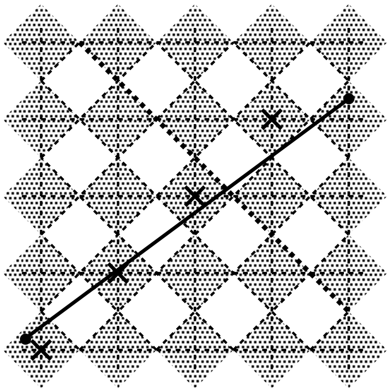

Figure 3.4. Visualization of Bresenham's algorithm. A portion of a line segment is shown. A diamond shaped region of height 1 is placed around each fragment center; those regions that the line segment exits cause rasterization to produce corresponding fragments.

where

$$
\begin{aligned}
S_{f1} &= \{(x,y) | x + y > x_f + y_f - 0.5\} \\
S_{f2} &= \{(x,y) | x + y \leq x_f + y_f + 0.5\} \\
S_{f3} &= \{(x,y) | x - y \geq x_f - y_f - 0.5\} \\
S_{f4} &= \{(x,y) | x - y < x_f - y_f + 0.5\}
\end{aligned}
$$

A line segment starting at $p_a$ and ending at $p_b$ produces those fragments $f$ for which the segment intersects $R_f$, except if $p_b$ is contained in $R_f$. See Figure 3.4.

When $p_a$ and $p_b$ lie on fragment centers, this characterization of fragments reduces to Bresenham's algorithm with one modification: lines produced in this description are "half-open," meaning that the final fragment (corresponding to $p_b$) is not drawn. This means that when rasterizing a series of connected line segments, shared endpoints will be produced only once rather than twice (as would occur with Bresenham's algorithm).

Because the initial and final conditions of the diamond-exit rule may be difficult to implement, other line segment rasterization algorithms are allowed, subject to the following rules:

1. The coordinates of a fragment produced by the algorithm may not deviate by more than one unit in either $x$ or $y$ window coordinates from a corresponding fragment produced by the diamond-exit rule.

2. The total number of fragments produced by the algorithm may differ from that produced by the diamond-exit rule by no more than one.

3. For an $x$-major line, no two fragments may be produced that lie in the same window-coordinate column (for a $y$-major line, no two fragments may appear in the same row).

4. If two line segments share a common endpoint, and both segments are either $x$-major (both left-to-right or both right-to-left) or $y$-major (both bottom-to-top or both top-to-bottom), then rasterizing both segments may not produce duplicate fragments, nor may any fragments be omitted so as to interrupt continuity of the connected segments.

Next we must specify how the data associated with each rasterized fragment are obtained. Let the window coordinates of a produced fragment center be given by $p_r = (x_d, y_d)$ and let $p_a = (x_a, y_a)$ and $p_b = (x_b, y_b)$. Set

$$t = \frac{(p_r - p_a) \cdot (p_b - p_a)}{\|p_b - p_a\|^2}. \tag{3.1}$$

(Note that $t = 0$ at $p_a$ and $t = 1$ at $p_b$.) The value of an associated datum $f$ for the fragment, whether it be R, G, B, or A (in RGBA mode) or a color index (in color index mode), or the $s$, $t$, or $r$ texture coordinate (the depth value, window $z$, must be found using equation 3.3, below), is found as

$$f = \frac{(1-t)f_a/w_a + tf_b/w_b}{(1-t)\alpha_a/w_a + t\alpha_b/w_b} \tag{3.2}$$

where $f_a$ and $f_b$ are the data associated with the starting and ending endpoints of the segment, respectively; $w_a$ and $w_b$ are the clip $w$ coordinates of the starting and ending endpoints of the segments, respectively. $\alpha_a = \alpha_b = 1$ for all data except texture coordinates, in which case $\alpha_a = q_a$ and $\alpha_b = q_b$ ($q_a$ and $q_b$ are the homogeneous texture coordinates at the starting and ending endpoints of the segment; results are undefined if either of these is 0). Note that linear interpolation would use

$$f = (1-t)f_a/\alpha_a + tf_b/\alpha_b. \tag{3.3}$$

The reason that this formula is incorrect (except for the depth value) is that it interpolates a datum in window space, which may be distorted by perspective. What is actually desired is to find the corresponding value when interpolated in eye space, which equation 3.2 does. A GL implementation may choose to approximate equation 3.2 with 3.3, but this will normally lead to unacceptable distortion effects when interpolating texture coordinates.

### 3.4.2 Other Line Segment Features

We have just described the rasterization of non-antialiased line segments of width one using the default line stipple of $FFFF_{16}$. We now describe the rasterization of line segments for general values of the line segment rasterization parameters.

#### Line Stipple

The command

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

defines a *line stipple*. *pattern* is a 16-bit integer whose bit pattern determines those fragments that are to be drawn when the line is rasterized, and *factor* is a count that is used to modify the effective line stipple by causing each bit in *pattern* to be used *factor* times. $factor$ is clamped to the range $[1, 256]$. Line stippling may be enabled or disabled using **Enable** or **Disable** with the constant `LINE_STIPPLE`. When disabled, it is as if the line stipple has its default value.

Line stippling masks certain fragments that are produced by rasterization so that they are not sent to the per-fragment stage of the GL. The masking is achieved using three parameters: the 16-bit line stipple $p$, the line repeat count $r$, and an integer stipple counter $s$. Let

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

Then a fragment is produced if the $b$th bit of $p$ is 1, and not produced otherwise. The bits of $p$ are numbered with 0 being the least significant and 15 being the most significant. The initial value of $s$ is zero; $s$ is incremented after production of each fragment of a line segment (fragments are produced in order, beginning at the starting point and working towards the ending point). $s$ is reset to 0 whenever a **Begin** occurs, and before every line segment in a group of independent segments (as specified when **Begin** is invoked with `LINES`).

If the line segment has been clipped, then the value of $s$ at the beginning of the line segment is indeterminate.

### Wide Lines

The actual width of non-antialiased lines is determined by rounding the supplied width to the nearest integer, then clamping it to the implementation-dependent maximum non-antialiased line width. Though this implementation-dependent value cannot be queried, it must be no less than the implementation-dependent maximum antialiased line width, rounded to the nearest integer value, and in any event no less than 1. If rounding the specified width results in the value 0, then it is as if the value were 1.

Non-antialiased line segments of width other than one are rasterized by offsetting them in the minor direction (for an $x$-major line, the minor direction is $y$, and for a $y$-major line, the minor direction is $x$) and replicating fragments in the minor direction (see Figure 3.5). Let $w$ be the width rounded to the nearest integer (if $w = 0$, then it is as if $w = 1$). If the line segment has endpoints given by $(x_0, y_0)$ and $(x_1, y_1)$ in window coordinates, the segment with endpoints $(x_0, y_0 - (w-1)/2)$ and $(x_1, y_1 - (w-1)/2)$ is rasterized, but instead of a single fragment, a column of fragments of height $w$ (a row of fragments of length $w$ for a $y$-major segment) is produced at each $x$ ($y$ for $y$-major) location. The lowest fragment of this column is the fragment that would be produced by rasterizing the segment of width 1 with the modified coordinates. The whole column is not produced if the stipple bit for the column's $x$ location is zero; otherwise, the whole column is produced.

### Antialiasing

Rasterized antialiased line segments produce fragments whose fragment squares intersect a rectangle centered on the line segment. Two of the edges are parallel to the specified line segment; each is at a distance of one-half the current width from that segment: one above the segment and one below it. The other two edges pass through the line endpoints and are perpendicular to the direction of the specified line segment. Coverage values are computed for each fragment by computing the area of the intersection of

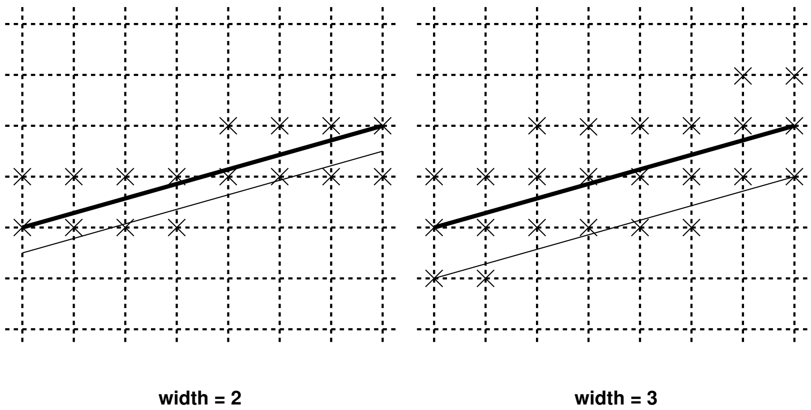


Figure 3.5. Rasterization of non-antialiased wide lines. x-major line segments are shown. The heavy line segment is the one specified to be rasterized; the light segment is the offset segment used for rasterization. x marks indicate the fragment centers produced by rasterization.

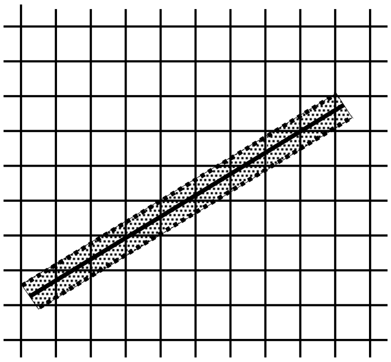

Figure 3.6. The region used in rasterizing and finding corresponding coverage values for an antialiased line segment (an x-major line segment is shown).

the rectangle with the fragment square (see Figure 3.6; see also section 3.2).

Equation 3.2 is used to compute associated data values just as with non-antialiased lines; equation 3.1 is used to find the value of $t$ for each fragment whose square is intersected by the line segment's rectangle. Not all widths need be supported for line segment antialiasing, but width 1.0 antialiased segments must be provided. As with the point width, a GL implementation may be queried for the range and number of gradations of available antialiased line widths.

For purposes of antialiasing, a stippled line is considered to be a sequence of contiguous rectangles centered on the line segment. Each rectangle has width equal to the current line width and length equal to 1 pixel (except the last, which may be shorter). These rectangles are numbered from 0 to $n$, starting with the rectangle incident on the starting endpoint of the segment. Each of these rectangles is either eliminated or produced according to the procedure given under **Line Stipple**, above, where "fragment" is replaced with "rectangle." Each rectangle so produced is rasterized as if it were an antialiased polygon, described below (but culling, non-default settings of **PolygonMode**, and polygon stippling are not applied).

### 3.4.3 Line Rasterization State

The state required for line rasterization consists of the floating-point line width, a 16-bit line stipple, the line stipple repeat count, a bit indicating whether stippling is enabled or disabled, and a bit indicating whether line antialiasing is on or off. In addition, during rasterization, an integer stipple counter must be maintained to implement line stippling. The initial value of the line width is 1.0. The initial value of the line stipple is *0xFFFF* (a stipple of all ones). The initial value of the line stipple repeat count is one. The initial state of line stippling is disabled. The initial state of line segment antialiasing is disabled.

## 3.5 Polygons

A polygon results from a polygon **Begin/End** object, a triangle resulting from a triangle strip, triangle fan, or series of separate triangles, or a quadrilateral arising from a quadrilateral strip, series of separate quadrilaterals, or a **Rect** command. Like points and line segments, polygon rasterization is controlled by several variables. Polygon antialiasing is controlled with **Enable** and **Disable** with the symbolic constant `POLYGON_SMOOTH`. The analog to line segment stippling for polygons is polygon stippling, described below.

### 3.5.1 Basic Polygon Rasterization

The first step of polygon rasterization is to determine if the polygon is *back facing* or *front facing*. This determination is made by examining the sign of the area computed by equation 2.6 of section 2.12.1 (including the possible reversal of this sign as indicated by the last call to **FrontFace**). If this sign is positive, the polygon is frontfacing; otherwise, it is back facing. This determination is used in conjunction with the **CullFace** enable bit and mode value to decide whether or not a particular polygon is rasterized. The **CullFace** mode is set by calling

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

*mode* is a symbolic constant: one of `FRONT`, `BACK` or `FRONT_AND_BACK`. Culling is enabled or disabled with **Enable** or **Disable** using the symbolic constant `CULL_FACE`. Front facing polygons are rasterized if either culling is disabled or the **CullFace** mode is `BACK` while back facing polygons are rasterized only if either culling is disabled or the **CullFace** mode is `FRONT`. The initial setting of the **CullFace** mode is `BACK`. Initially, culling is disabled.

The rule for determining which fragments are produced by polygon rasterization is called *point sampling*. The two-dimensional projection obtained by taking the $x$ and $y$ window coordinates of the polygon's vertices is formed. Fragment centers that lie inside of this polygon are produced by rasterization. Special treatment is given to a fragment whose center lies on a polygon boundary edge. In such a case we require that if two polygons lie on either side of a common edge (with identical endpoints) on which a fragment center lies, then exactly one of the polygons results in the production of the fragment during rasterization.

As for the data associated with each fragment produced by rasterizing a polygon, we begin by specifying how these values are produced for fragments in a triangle. Define *barycentric coordinates* for a triangle. Barycentric coordinates are a set of three numbers, $a$, $b$, and $c$, each in the range $[0, 1]$, with $a + b + c = 1$. These coordinates uniquely specify any point $p$ within the triangle or on the triangle's boundary as

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

where $p_a$, $p_b$, and $p_c$ are the vertices of the triangle. $a$, $b$, and $c$ can be found as

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

where $A(lmn)$ denotes the area in window coordinates of the triangle with vertices $l$, $m$, and $n$.

Denote a datum at $p_a$, $p_b$, or $p_c$ as $f_a$, $f_b$, or $f_c$, respectively. Then the value $f$ of a datum at a fragment produced by rasterizing a triangle is given by

$$f = \frac{af_a/w_a + bf_b/w_b + cf_c/w_c}{a\alpha_a/w_a + b\alpha_b/w_b + c\alpha_c/w_c} \tag{3.4}$$

where $w_a$, $w_b$ and $w_c$ are the clip $w$ coordinates of $p_a$, $p_b$, and $p_c$, respectively. $a$, $b$, and $c$ are the barycentric coordinates of the fragment for which the data are produced. $\alpha_a = \alpha_b = \alpha_c = 1$ except for texture $s$, $t$, and $r$ coordinates, for which $\alpha_a = q_a$, $\alpha_b = q_b$, and $\alpha_c = q_c$ (if any of $q_a$, $q_b$, or $q_c$ are zero, results are undefined). $a$, $b$, and $c$ must correspond precisely to the exact coordinates of the center of the fragment. Another way of saying this is that the data associated with a fragment must be sampled at the fragment's center.

Just as with line segment rasterization, equation 3.4 may be approximated by

$$f = af_a/\alpha_a + bf_b/\alpha_b + cf_c/\alpha_c;$$

this may yield acceptable results for color values (it *must* be used for depth values), but will normally lead to unacceptable distortion effects if used for texture coordinates.

For a polygon with more than three edges, we require only that a convex combination of the values of the datum at the polygon's vertices can be used to obtain the value assigned to each fragment produced by the rasterization algorithm. That is, it must be the case that at every fragment

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

where $n$ is the number of vertices in the polygon, $f_i$ is the value of the $f$ at vertex $i$; for each $i$ $0 \le a_i \le 1$ and $\sum_{i=1}^{n} a_i = 1$. The values of the $a_i$ may differ from fragment to fragment, but at vertex $i$, $a_j = 0, j \neq i$ and $a_i = 1$.

One algorithm that achieves the required behavior is to triangulate a polygon (without adding any vertices) and then treat each triangle individually as already discussed. A scan-line rasterizer that linearly interpolates data along each edge and then linearly interpolates data across each horizontal span from edge to edge also satisfies the restrictions (in this case, the numerator and denominator of equation 3.4 should be iterated independently and a division performed for each fragment).

### 3.5.2 Stippling

Polygon stippling works much the same way as line stippling, masking out certain fragments produced by rasterization so that they are not sent to the next stage of the GL. This is the case regardless of the state of polygon antialiasing. Stippling is controlled with

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

*pattern* is a pointer to memory into which a $32 \times 32$ pattern is packed. The pattern is unpacked from memory according to the procedure given in section 3.6.3 for **DrawPixels**; it is as if the *height* and *width* passed to that command were both equal to 32, the *type* were `BITMAP`, and the *format* were `COLOR_INDEX`. The unpacked values (before any conversion or arithmetic would have been performed) are bitwise ANDed with 1 to obtain a stipple pattern of zeros and ones.

If $x_w$ and $y_w$ are the window coordinates of a rasterized polygon fragment, then that fragment is sent to the next stage of the GL if and only if the bit of the pattern $(x_w \bmod 32, y_w \bmod 32)$ is 1.

Polygon stippling may be enabled or disabled with **Enable** or **Disable** using the constant `POLYGON_STIPPLE`. When disabled, it is as if the stipple pattern were all ones.

### 3.5.3 Antialiasing

Polygon antialiasing rasterizes a polygon by producing a fragment wherever the interior of the polygon intersects that fragment's square. A coverage value is computed at each such fragment, and this value is saved to be applied as described in section 3.10. An associated datum is assigned to a fragment by integrating the datum's value over the region of the intersection of the fragment square with the polygon's interior and dividing this integrated value by the area of the intersection. For a fragment square lying entirely within the polygon, the value of a datum at the fragment's center may be used instead of integrating the value across the fragment.

Polygon stippling operates in the same way whether polygon antialiasing is enabled or not. The polygon point sampling rule defined in section 3.5.1, however, is not enforced for antialiased polygons.

### 3.5.4 Options Controlling Polygon Rasterization

The interpretation of polygons for rasterization is controlled using

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

*face* is one of `FRONT`, `BACK`, or `FRONT_AND_BACK`, indicating that the rasterizing method described by *mode* replaces the rasterizing method for front facing polygons, back facing polygons, or both front and back facing polygons, respectively. *mode* is one of the symbolic constants `POINT`, `LINE`, or `FILL`. Calling **PolygonMode** with `POINT` causes certain vertices of a polygon to be treated, for rasterization purposes, just as if they were enclosed within a **Begin**(`POINT`) and **End** pair. The vertices selected for this treatment are those that have been tagged as having a polygon boundary edge beginning on them (see section 2.6.2). `LINE` causes edges that are tagged as boundary to be rasterized as line segments. (The line stipple counter is reset at the beginning of the first rasterized edge of the polygon, but not for subsequent edges.) `FILL` is the default mode of polygon rasterization, corresponding to the description in sections 3.5.1, 3.5.2, and 3.5.3. Note that these modes affect only the final rasterization of polygons: in particular, a polygon's vertices are lit, and the polygon is clipped and possibly culled before these modes are applied.

Polygon antialiasing applies only to the `FILL` state of **PolygonMode**. For `POINT` or `LINE`, point antialiasing or line segment antialiasing, respectively, apply.

### 3.5.5 Polygon Rasterization State

The state required for polygon rasterization consists of a polygon stipple pattern, whether stippling is enabled or disabled, the current state of polygon antialiasing (enabled or disabled), and the current values of the **PolygonMode** setting for each of front and back facing polygons. The initial stipple pattern is all ones; initially stippling is disabled. The initial setting of polygon antialiasing is disabled. The initial state for **PolygonMode** is `FILL` for both front and back facing polygons.

## 3.6 Pixel Rectangles

Rectangles of color, depth, and certain other values may be converted to fragments using the **DrawPixels** command. Some of the parameters and operations governing the operation of **DrawPixels** are shared by **ReadPixels** (used to obtain pixel values from the framebuffer) and **CopyPixels** (used to copy pixels from one framebuffer location to another); the discussion of **ReadPixels** and **CopyPixels**, however, is deferred until Chapter 4 after the framebuffer has been discussed in detail. Nevertheless, we note in this section when parameters and state pertaining to **DrawPixels** also pertain to **ReadPixels** or **CopyPixels**.

A number of parameters control the encoding of pixels in client memory (for reading and writing) and how pixels are processed before being placed in or after being read from the framebuffer (for reading, writing, and copying). These parameters are set with three commands: **PixelStore**, **PixelTransfer**, and **PixelMap**.

| Parameter Name | Type | Initial Value | Valid Range |
|---|---|---|---|
| UNPACK_SWAP_BYTES | boolean | FALSE | TRUE/FALSE |
| UNPACK_LSB_FIRST | boolean | FALSE | TRUE/FALSE |
| UNPACK_ROW_LENGTH | integer | 0 | $[0, \infty)$ |
| UNPACK_SKIP_ROWS | integer | 0 | $[0, \infty)$ |
| UNPACK_SKIP_PIXELS | integer | 0 | $[0, \infty)$ |
| UNPACK_ALIGNMENT | integer | 4 | 1,2,4,8 |

Table 3.1: **PixelStore** parameters pertaining to **DrawPixels**.

### 3.6.1 Pixel Storage Modes

Pixel storage modes affect the operation of **DrawPixels** and **ReadPixels** (as well as other commands; see sections 3.5.2, 3.7, and 3.8) when one of these commands is issued. This may differ from the time that the command is executed if the command is placed in a display list (see section 5.4). Pixel storage modes are set with

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

*pname* is a symbolic constant indicating a parameter to be set, and *param* is the value to set it to. Table 3.1 summarizes the pixel storage parameters, their types, their initial values, and their allowable ranges. Setting a parameter to a value outside the given range results in the error INVALID_VALUE.

The version of **PixelStore** that takes a floating-point value may be used to set any type of parameter; if the parameter is boolean, then it is set to FALSE if the passed value is 0.0 and TRUE otherwise, while if the parameter is an integer, then the passed value is rounded to the nearest integer. The integer version of the command may also be used to set any type of parameter; if the parameter is boolean, then it is set to FALSE if the passed value is 0 and TRUE otherwise, while if the parameter is a floating-point value, then the passed value is converted to floating-point.

### 3.6.2 Pixel Transfer Modes

Pixel transfer modes affect the operation of **DrawPixels**, **ReadPixels**, and **CopyPixels** at the time when one of these commands is executed (which

| Parameter Name | Type | Initial Value | Valid Range |
|---|---|---|---|
| `MAP_COLOR` | boolean | `FALSE` | `TRUE`/`FALSE` |
| `MAP_STENCIL` | boolean | `FALSE` | `TRUE`/`FALSE` |
| `INDEX_SHIFT` | integer | 0 | $(-\infty, \infty)$ |
| `INDEX_OFFSET` | integer | 0 | $(-\infty, \infty)$ |
| `RED_SCALE` | float | 1.0 | $(-\infty, \infty)$ |
| `GREEN_SCALE` | float | 1.0 | $(-\infty, \infty)$ |
| `BLUE_SCALE` | float | 1.0 | $(-\infty, \infty)$ |
| `ALPHA_SCALE` | float | 1.0 | $(-\infty, \infty)$ |
| `DEPTH_SCALE` | float | 1.0 | $(-\infty, \infty)$ |
| `RED_BIAS` | float | 0.0 | $(-\infty, \infty)$ |
| `GREEN_BIAS` | float | 0.0 | $(-\infty, \infty)$ |
| `BLUE_BIAS` | float | 0.0 | $(-\infty, \infty)$ |
| `ALPHA_BIAS` | float | 0.0 | $(-\infty, \infty)$ |
| `DEPTH_BIAS` | float | 0.0 | $(-\infty, \infty)$ |

Table 3.2: **PixelTransfer** parameters.

may differ from the time the command is issued). Some pixel transfer modes are set with

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

*param* is a symbolic constant indicating a parameter to be set, and *value* is the value to set it to. Table 3.2 summarizes the pixel transfer parameters that are set with **PixelTransfer**, their types, their initial values, and their allowable ranges. Setting a parameter to a value outside the given range results in the error `INVALID_VALUE`. The same versions of the command exist as for **PixelStore**, and the same rules apply to accepting and converting passed values to set parameters.

The other pixel transfer modes are the various lookup tables used by **DrawPixels**, **ReadPixels**, and **CopyPixels**. These are set with

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

*map* is a symbolic map name, indicating the map to set, *size* indicates the size of the map, and *values* is a pointer to an array of *size* map values.

| Map Name | Address | Value | Init. Size | Init. Value |
|---|---|---|---|---|
| PIXEL_MAP_I_TO_I | color idx | color idx | 1 | 0 |
| PIXEL_MAP_S_TO_S | stencil idx | stencil idx | 1 | 0 |
| PIXEL_MAP_I_TO_R | color idx | R | 1 | 0.0 |
| PIXEL_MAP_I_TO_G | color idx | G | 1 | 0.0 |
| PIXEL_MAP_I_TO_B | color idx | B | 1 | 0.0 |
| PIXEL_MAP_I_TO_A | color idx | A | 1 | 0.0 |
| PIXEL_MAP_R_TO_R | R | R | 1 | 0.0 |
| PIXEL_MAP_G_TO_G | G | G | 1 | 0.0 |
| PIXEL_MAP_B_TO_B | B | B | 1 | 0.0 |
| PIXEL_MAP_A_TO_A | A | A | 1 | 0.0 |

Table 3.3: **PixelMap** parameters.

The entries of a table may be specified using one of three types: single-precision floating-point, 16-bit unsigned integer, or 32-bit unsigned integer, depending on which of the three versions of **PixelMap** is called. A table entry is converted to the appropriate type when it is specified. An entry giving a color component value is converted according to Table 2.4. An entry giving a color index value is converted from 16-bit or 32-bit unsigned integer to floating-point. An entry giving a stencil index is converted from single-precision floating-point to an integer by rounding to nearest. The various tables and their initial sizes and entries are summarized in Table 3.3. A table that takes an index as an address must have $size = 2^n$ or the error `INVALID_VALUE` results. The maximum allowable *size* of each table is implementation dependent, but must be at least 32 (a single maximum applies to all tables). The error `INVALID_VALUE` is generated if a *size* larger than the implemented maximum, or less than zero, is given to **PixelMap**.

### 3.6.3 Rasterization of Pixel Rectangles

The process of drawing pixels encoded in host memory is diagrammed in Figure 3.7. We describe the stages of this process in the order in which they occur.

Pixels are drawn using

void **DrawPixels**( sizei *width*, sizei *height*, enum *format*,

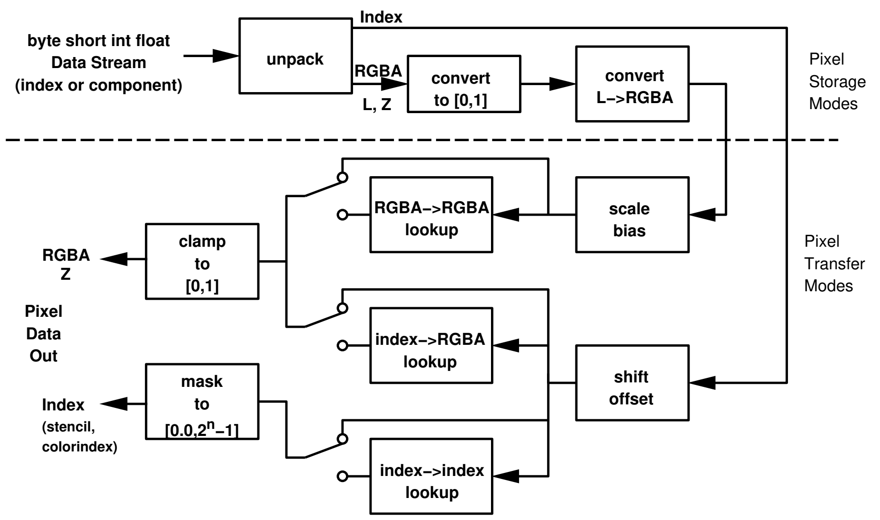


Figure 3.7. Operation of **DrawPixels**. The parameters controlling the stages above the dotted line are set with **PixelStore** while those controlling the stages below the line are set with **PixelTransfer** or **PixelMap**.

| *type* | Corresponding Type |
|---|---|
| UNSIGNED_BYTE | unsigned 8-bit integer |
| BYTE | signed 8-bit integer |
| BITMAP | single bits in unsigned 8-bit integers |
| UNSIGNED_SHORT | unsigned 16-bit integer |
| SHORT | signed 16-bit integer |
| UNSIGNED_INT | unsigned 32-bit integer |
| INT | 32-bit integer |
| FLOAT | single-precision floating-point |

Table 3.4: **DrawPixels** and **ReadPixels** types.

`enum` *type*, `void` **data* ) ;

*format* is a symbolic constant indicating what the values in memory represent. *width* and *height* are the width and height, respectively, of the pixel rectangle to be drawn. *data* is a pointer to the data to be drawn. These data are signed or unsigned bytes, 16-bit integers, or 32-bit integers, or single-precision floating-point values, depending on the value of *type*. The possible values of *type* and the types they indicate are given in Table 3.4. If the GL is in color index mode and *format* is not one of COLOR_INDEX, STENCIL_INDEX, or DEPTH_COMPONENT, then the error INVALID_OPERATION occurs.

### Unpacking

Data are taken from host memory as a sequence of signed or unsigned bytes, 16-bit integers, 32-bit integers, or single-precision floating-point *elements*. These elements are grouped into sets of one, two, three, or four values, depending on the *format*, to form a *group*. Table 3.5 summarizes the format of groups obtained from memory. It also indicates those formats that yield indices and those that yield components.

The byte-ordering of the bytes that constitute each element in memory is whatever is native to the GL client if UNPACK_SWAP_BYTES is FALSE. If it is TRUE, then byte-ordering is reversed for each element. In this case, there is no effect on a one-byte element, but the constituent bytes of a two-byte or four-byte element are reversed when those bytes are read to form the element. If the four bytes making up a four-byte element are stored in order

| Name | Type | Elements | Target Buffer |
|---|---|---|---|
| COLOR_INDEX | Index | Color Index | Color |
| STENCIL_INDEX | Index | Stencil value | Stencil |
| DEPTH_COMPONENT | Component | Depth value | Depth |
| RED | Component | R | Color |
| GREEN | Component | G | Color |
| BLUE | Component | B | Color |
| ALPHA | Component | A | Color |
| RGB | Components | R, G, B | Color |
| RGBA | Components | R, G, B, A | Color |
| LUMINANCE | Component | Luminance value | Color |
| LUMINANCE_ALPHA | Components | Luminance value, A | Color |

Table 3.5: **DrawPixels** and **ReadPixels** formats. The third column gives a description of and the number and order of elements in a group.

in memory as $b_1$, $b_2$, $b_3$, and $b_4$, then the reverse order is $b_4$, $b_3$, $b_2$, and $b_1$.

Calling **DrawPixels** with a *type* of BITMAP is a special case in which the data are a series of unsigned bytes. In this case, the only allowable *format*s are COLOR_INDEX and STENCIL_INDEX. (Other formats generate the error INVALID_ENUM.) Each byte is taken as a series of eight bits, each of which is a single element. The single-bit elements within each byte are ordered from most significant to least significant if the value of UNPACK_LSB_FIRST is FALSE; otherwise, the ordering is from least significant to most significant.

The groups in memory are treated as being arranged in a rectangle. This rectangle consists of a series of *rows*, with the first element of the first group of the first row pointed to by the pointer passed to **DrawPixels**. If the value of UNPACK_ROW_LENGTH is not positive, then the number of groups in a row is *width*; otherwise the number of groups is UNPACK_ROW_LENGTH. If the first element of a row is at location $p$ in memory, then the location of the first element of the next row is obtained by skipping

$$k = \begin{cases} nl & s \geq a, \\ a/s\lceil snl/a \rceil & s < a \end{cases} \tag{3.5}$$

elements, where $n$ is the number of elements in a group, $l$ is the number of groups in the row, $a$ is the value of UNPACK_ALIGNMENT, and $s$ is the size, in

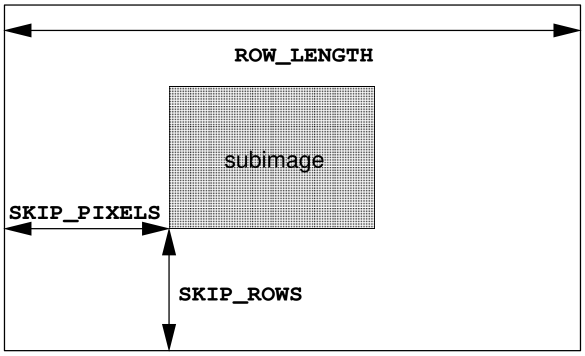


Figure 3.8. Selecting a subimage from an image. The indicated parameter names are prefixed by UNPACK_ for **DrawPixels** and by PACK_ for **ReadPixels**.

bytes, of an element. In the case of 1-bit elements, the location of the next row is obtained by skipping

$$k = 8a \left\lceil \frac{nl}{8a} \right\rceil \tag{3.6}$$

elements. The allowable values of UNPACK_ALIGNMENT are 1, 2, 4, or 8; other values result in the error INVALID_VALUE.

There is a mechanism for selecting a sub-rectangle of groups from a larger containing rectangle. This mechanism relies on three integer parameters: UNPACK_ROW_LENGTH, UNPACK_SKIP_ROWS, and UNPACK_SKIP_PIXELS. Before obtaining the first group from memory, the pointer supplied to **DrawPixels** is effectively advanced by (UNPACK_SKIP_PIXELS)$n$ + (UNPACK_SKIP_ROWS)$k$ elements. Then *width* groups are obtained from contiguous elements in memory (without advancing the pointer), after which the pointer is advanced by $k$ elements. *height* sets of *width* groups of values are obtained this way. See Figure 3.8.

### Conversion to floating-point

This step applies only to groups of components. It is not performed on indices. Each element in a group is converted to a floating-point value according to the appropriate formula in Table 2.4 (section 2.12) for color components.

### Conversion to RGB

This step is applied only if the *format* is `LUMINANCE` or `LUMINANCE_ALPHA`. If the *format* is `LUMINANCE`, then each group of one element is converted to a group of R, G, and B (three) elements by copying the original single element into each of the three new elements. If the *format* is `LUMINANCE_ALPHA`, then each group of two elements is converted to a group of R, G, B, and A (four) elements by copying the first original element into each of the first three new elements and copying the second original element to the A (fourth) new element.

### Final Expansion to RGBA

This step is performed only for non-depth component groups. Each group is converted to a group of 4 elements as follows: if a group does not contain an A element, then A is added and set to 1.0. If any of R, G, or B is missing from the group, each missing element is added and assigned a value of 0.0.

### Arithmetic on Components

This step applies only to component groups. Each component is multipled by an appropriate signed scale factor: `RED_SCALE` for an R component, `GREEN_SCALE` for a G component, `BLUE_SCALE` for a B component, and `ALPHA_SCALE` for an A component, or `DEPTH_SCALE` for a depth component. Then the result is added to the the appropriate signed bias: `RED_BIAS`, `GREEN_BIAS`, `BLUE_BIAS`, `ALPHA_BIAS`, or `DEPTH_BIAS`.

### Arithmetic on Indices

This step applies only to indices. If the index is a floating-point value, it is converted to fixed-point, with an unspecified number of bits to the right of the binary point. Indices that are already integers remain so; any fraction bits in the resulting fixed-point value are zero.

The fixed-point index is then shifted by |`INDEX_SHIFT`| bits, left if `INDEX_SHIFT` $> 0$ and right otherwise. In either case the shift is zero-filled. Then, the signed integer offset `INDEX_OFFSET` is added to the index.

### RGBA to RGBA Lookup

This step applies only to RGBA component groups, and is skipped if MAP_COLOR is FALSE. First, each component is clamped to the range $[0, 1]$. There is a table associated with each of the R, G, B, and A component elements: PIXEL_MAP_R_TO_R for R, PIXEL_MAP_G_TO_G for G, PIXEL_MAP_B_TO_B for B, and PIXEL_MAP_A_TO_A for A. Each element is multiplied by an integer one less than the size of the corresponding table, and, for each element, an address is found by rounding this value to the nearest integer. For each element, the addressed value in the corresponding table replaces the element.

### Index Lookup

This step applies only to indices. If the GL is in RGBA mode, then the integer part of the index is used to reference 4 tables of color components: PIXEL_MAP_I_TO_R, PIXEL_MAP_I_TO_G, PIXEL_MAP_I_TO_B, and PIXEL_MAP_I_TO_A. Each of these tables must have $2^n$ entries for some integer value of $n$ ($n$ may be different for each table). For each table, the index is first rounded to the nearest integer; the result is ANDed with $2^n - 1$, and the resulting value used as an address into the table. The indexed value becomes an R, G, B, or A value, as appropriate. The group of four elements so obtained replaces the index, changing the group's type to "component."

If the GL is in color index mode and if MAP_COLOR is TRUE, then the index is looked up in the PIXEL_MAP_I_TO_I table (otherwise, the index is not looked up). Again, the table must have $2^n$ entries for some integer $n$, and the integer part of the index is ANDed with $2^n - 1$, producing a value. This value addresses the table, and the value in the table replaces the index. The floating-point table value is first rounded to a fixed-point value with unspecified precision.

Finally, if *format* is STENCIL_INDEX and if MAP_STENCIL is TRUE, then the index is looked up as described in the preceding paragraph, but using the PIXEL_MAP_S_TO_S table.

### Final Conversion

For a color index, final conversion consists of masking the bits of the index to the left of the binary point by $2^n - 1$, where $n$ is the number of bits in an index buffer. For RGBA components, each element is clamped to $[0, 1]$. The resulting values are converted to fixed-point according to the rules given in section 2.12.9 (Final Color Processing).

For a depth component, an element is first clamped to $[0, 1]$ and then converted to fixed-point as if it were a window $z$ value (see section 2.9.1, Controlling the Viewport).

Stencil indices are masked by $2^n - 1$, where $n$ is the number of bits in the stencil buffer.

### Conversion to Fragments

The conversion of a group to fragments is controlled with

void **PixelZoom**( float $z_x$, float $z_y$ ) ;

Let $(x_{rp}, y_{rp})$ be the current raster position (section 2.11). (If the current raster position is invalid, then **DrawPixels** is ignored.) If a particular group (index or components) is the $n$th in a row and belongs to the $m$th row, consider the region in window coordinates bounded by the rectangle with corners

$$(x_{rp} + z_x n, y_{rp} + z_y m) \qquad \text{and} \qquad (x_{rp} + z_x(n + 1), y_{rp} + z_y(m + 1))$$

(either $z_x$ or $z_y$ may be negative). Any fragments whose centers lie inside of this rectangle (or on its bottom or left boundaries) are produced in correspondence with this particular group of elements.

A fragment arising from a group consisting of color data takes on the color index or color components of the group; the depth and texture coordinates are taken from the current raster position's associated data. A fragment arising from a depth component takes the component's depth value; the color and texture coordinates are given by those associated with the current raster position. Groups arising from **DrawPixels** with a *format* of STENCIL_INDEX are treated specially and are described in section 4.3.1.

## 3.7 Bitmaps

Bitmaps are rectangles of zeros and ones specifying a particular pattern of fragments to be produced. Each of these fragments has the same associated data. These data are those associated with the *current raster position.*

Bitmaps are sent using

void **Bitmap**( sizei $w$, sizei $h$, float $x_{bo}$, float $y_{bo}$, float $x_{bi}$, float $y_{bi}$, ubyte *data[]* ) ;

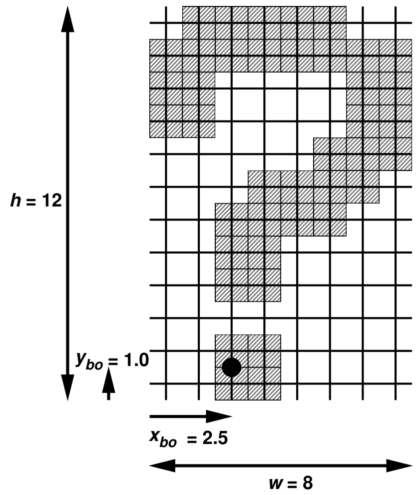


Figure 3.9. A bitmap and its associated parameters. $x_{bi}$ and $y_{bi}$ are not shown.

$w$ and $h$ comprise the integer width and height of the rectangular bitmap, respectively. $(x_{bo}, y_{bo})$ gives the floating-point $x$ and $y$ values of the bitmap's origin. $(x_{bi}, y_{bi})$ gives the floating-point $x$ and $y$ increments that are added to the raster position after the bitmap is rasterized. *data* is a pointer to a bitmap.

Like a polygon pattern, a bitmap is unpacked from memory according to the procedure given in section 3.6.3 for **DrawPixels**; it is as if the *width* and *height* passed to that command were equal to $w$ and $h$, respectively, the *type* were `BITMAP`, and the *format* were `COLOR_INDEX`. The unpacked values (before any conversion or arithmetic would have been performed) are bitwise ANDed with 1 to obtain a stipple pattern of zeros and ones. See Figure 3.9.

A bitmap sent using **Bitmap** is rasterized as follows. First, if the current raster position is invalid (the valid bit is reset), the bitmap is ignored. Otherwise, a rectangular array of fragments is constructed, with lower left corner at

$$(x_{ll}, y_{ll}) = (\lfloor x_{rp} - x_{bo} \rfloor, \lfloor y_{rp} - y_{bo} \rfloor)$$

and upper right corner at $(x_{ll} + w, y_{ll} + h)$ where $w$ and $h$ are the width and height of the bitmap, respectively. Fragments in the array are produced if the corresponding bit in the bitmap is 1 and not produced otherwise. The associated data for each fragment are those associated with the current raster position. Once the fragments have been produced, the current raster position is updated:

$$(x_{rp}, y_{rp}) \leftarrow (x_{rp} + x_{bi}, y_{rp} + y_{bi}).$$

The $z$ and $w$ values of the current raster position remain unchanged.

## 3.8 Texturing

Texturing maps a portion of a specified image onto each primitive for which texturing is enabled. This mapping is accomplished by using the color of an image at the location indicated by a fragment's $(s, t)$ coordinates to modify the fragment's RGBA color ($r$ is currently ignored). Texturing is specified only for RGBA mode; its use in color index mode is undefined.

The GL provides a means to specify the details of how texturing of a primitive is effected. These details include specification of the image to be texture mapped, the means by which the image is filtered when applied to the primitive, and the function that determines what RGBA value is produced given a fragment color and an image value.

The command

> `void` **TexImage2D**( `enum` *target*, `int` *level*, `int` *components*, `sizei` *width*, `sizei` *height*, `int` *border*, `enum` *format*, `enum` *type*, `void` **data* ) ;

is used to specify a texture image. Currently, *target* must be `TEXTURE_2D`. The arguments *width*, *height*, *format*, *type*, and *data* correspond precisely to the corresponding arguments to **DrawPixels** (refer to section 3.6.3); they specify the image's *width* and *height*, a *format* of the image data, the *type* of those data, and a pointer to the image data in memory. The image is taken from memory exactly as if these arguments were passed to **DrawPixels**, but the process stops just before final conversion. (The *format*s `STENCIL_INDEX` and `DEPTH_COMPONENT` are not allowed.) Components are selected from the R, G, B, and A values so extracted to obtain a texture with *components* components (the significance of the number of components is described below). Table 3.6 summarizes the mapping of R, G, B, and A values to texture

| Components | RGBA Values | Texture Components |
|---|---|---|
| 1 | R | $L$ |
| 2 | R, A | $L$, $A$ |
| 3 | R, G, B | $C$ |
| 4 | R, G, B, A | $C$, $A$ |

Table 3.6: Correspondence of texture components to extracted R, G, B, and A values. See section 3.8.3 for a description of the texture components $L$, $A$, and $C$.

components. Specifying a number of *components* other than 1, 2, 3, or 4 generates the error `INVALID_VALUE`.

The image itself (pointed to by *data*) is a sequence of groups of values. The first group is the lower left corner of the texture image. Subsequent groups fill out rows of width *width* from left to right; *height* rows are stacked from bottom to top.

The *level* argument to **TexImage2D** is an integer *level-of-detail* number. Levels of detail are discussed below, under **Mipmapping**. The main texture image has a level of detail number of 0. If a level-of-detail less than zero or greater than the base 2 logarithm of the maximum texture width or height (see below) is specified, the error `INVALID_VALUE` is generated.

The *border* argument to **TexImage2D** is a border width. The significance of borders is described below. The *border* width affects the required dimensions of the texture image: it must be the case that $width = 2^n + 2b$ and $height = 2^m + 2b$. where $b$ is the (non-negative) border width. If *width* and *height* do not satisfy these relationships, then the error `INVALID_VALUE` is generated. Currently, if $b$ is not either 0 or 1, then the error `INVALID_VALUE` is generated. The maximum allowable width or height of an image is implementation dependent, but must be at least 64 (or $64 + 2b$ with a border of width $b$). An excessive width or height, or a width or height less than zero, generates the `INVALID_VALUE` error.

Another command,

void **TexImage1D**( enum *target*, int *level*, int *components*, sizei *width*, int *border*, enum *format*, enum *type*, void **data* ) ;

is used to specify one-dimensional texture images. Currently, *target* must be the texture target TEXTURE_1D. For the purposes of decoding the texture image, **TexImage1D** is equivalent to calling **TexImage2D** with corresponding arguments and a *height* argument of 1, except that the *height* of the image is always 1, regardless of the value of *border*. It must be the case that $width = 2^n + 2b$ for some integer $n$ where $b$ is the value of *border*, or the error INVALID_VALUE is generated.

An image with zero height or width (or zero width, for **TexImage1D**) indicates the null texture. If the null texture is specified for level-of-detail zero, it is as if texturing were disabled.

The image indicated to the GL by the image pointer is decoded and copied into the GL's internal memory. This copying effectively places the decoded image inside a border of the maximum allowable width (currently 1) whether or not a border has been specified (see Figure 3.10). If no border or a border smaller than the maximum allowable width has been specified, then the image is still stored as if it were surrounded by a border of the maximum possible width. Any excess border (which surrounds the specified image, including any border) is assigned unspecified values. A one-dimensional texture has a border only at its left and right ends.

We shall refer to the (possibly border augmented) decoded image as the *texture array*. A two-dimensional texture array has width $2^n + 2b$ and height $2^m + 2b$, where $b$ is the maximum allowable border width; a one-dimensional texture array has width $2^n + 2b$ and height 1.

An element $(i, j)$ of the texture array is called a *texel* (for a 1-dimensional texture, $j$ is irrelevant). The *texture value* used in texturing a fragment is determined by that fragment's associated $(s, t)$ coordinates, but may not correspond to any actual texel. See Figure 3.10.

Various parameters control how the texture array is treated when applied to a fragment. Each parameter is set by calling

void **TexParameter{if}**( enum *target*, enum *pname*,
T *param* ) ;
void **TexParameter{if}v**( enum *target*, enum *pname*,
T *params* ) ;

*target* is the target, either TEXTURE_1D or TEXTURE_2D, *pname* is a symbolic constant indicating the parameter to be set; the possible constants and corresponding parameters are summarized in Table 3.7. In the first form of the command, *param* is a value to which to set a single-valued parameter; in the second form of the command, *params* is an array of parameters whose type depends on the parameter being set. If the values for TEXTURE_BORDER_COLOR

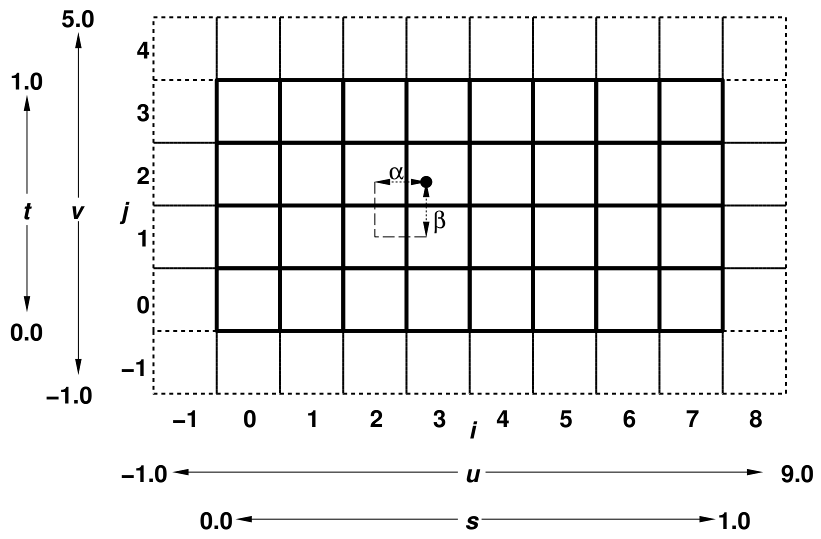


Figure 3.10. A texture image and the coordinates used to access it. This is a two-dimensional texture with $n = 3$ and $m = 2$. A one-dimensional texture would consist of a single horizontal strip. $\alpha$ and $\beta$, values used in blending adjacent texels to obtain a texture value, are also shown.

| Name | Type | Legal Values |
|---|---|---|
| TEXTURE_WRAP_S | integer | CLAMP, REPEAT |
| TEXTURE_WRAP_T | integer | CLAMP, REPEAT |
| TEXTURE_MIN_FILTER | integer | NEAREST, LINEAR, NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, LINEAR_MIPMAP_LINEAR |
| TEXTURE_MAG_FILTER | integer | NEAREST, LINEAR |
| TEXTURE_BORDER_COLOR | 4 floats | any 4 values in $[0, 1]$ |

Table 3.7: Texture parameters and their values.

are specified as integers, the conversion for signed integers from Table 2.4 is applied to convert the values to floating-point. Each of the four values set by TEXTURE_BORDER_COLOR is clamped to lie in $[0, 1]$.

### Texture Wrap Modes

If TEXTURE_WRAP_S or TEXTURE_WRAP_T is set to REPEAT, then the GL ignores the integer part of $s$ or $t$ coordinates, respectively, using only the fractional part. (For a number $r$, the fractional part is $r - \lfloor r \rfloor$, regardless of the sign of $r$; recall that the *floor* function truncates towards $-\infty$.) CLAMP causes $s$ or $t$ coordinates to be clamped to the range $[0, 1]$. The initial state is for both $s$ and $t$ behavior to be that given by REPEAT.

## 3.8.1 Texture Minification

Applying a texture to a primitive implies a mapping from texture image space to framebuffer image space. In general, this mapping involves a reconstruction of the sampled texture image, followed by a homogeneous warping implied by the mapping to framebuffer space, then a filtering, followed finally by a resampling of the filtered, warped, reconstructed image before applying it to a fragment. In the GL this mapping is approximated by one of two simple filtering schemes. One of these schemes is selected based on whether the mapping from texture space to framebuffer space is deemed to *magnify* or *minify* the texture image. The choice is governed by a scale factor $\rho(x,y)$; if $\rho(x,y)$ less than or equal to some constant (the selection of the constant is described below in section 3.8.2) the texture is said to be magnified; if it is greater, the texture is minified. $\lambda(x,y) \equiv \log_2[\rho(x,y)]$ is called the *level of detail.*

Let $s(x,y)$ be the function that associates an $s$ texture coordinate with each set of window coordinates $(x,y)$ that lie within a primitive; define $t(x,y)$ analogously. Let $u(x,y) = 2^n s(x,y)$ and $v(x,y) = 2^m t(x,y)$ (for a one-dimensional texture, define $v(x,y) \equiv 0$). For a polygon, $\rho$ is given at a fragment with window coordinates $(x,y)$ by

$$\rho = \max\left\{\sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2}\right\} \tag{3.7}$$

where $\partial u/\partial x$ indicates the derivative of $u$ with respect to window $x$, and similarly for the other derivatives. For a line, the formula is

$$\rho = \sqrt{\left(\frac{\partial u}{\partial x}\Delta x + \frac{\partial u}{\partial y}\Delta y\right)^2 + \left(\frac{\partial v}{\partial x}\Delta x + \frac{\partial v}{\partial y}\Delta y\right)^2} \Big/ l, \tag{3.8}$$

where $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$ with $(x_1, y_1)$ and $(x_2, y_2)$ being the segment's window coordinate endpoints and $l = \sqrt{\Delta x^2 + \Delta y^2}$. For a point, pixel rectangle, or bitmap, $\rho \equiv 1$.

While it is generally agreed that equations 3.7 and 3.8 give the best results when texturing, they are often impractical to implement. Therefore, an implementation may approximate the ideal $\rho$ with a function $f(x,y)$ subject to these conditions:

1. $f(x,y)$ is continuous and monotonically increasing in each of $|\partial u/\partial x|$, $|\partial u/\partial y|$, $|\partial v/\partial x|$, and $|\partial v/\partial y|$,

2. Let

$$m_u = \max\left\{\left|\frac{\partial u}{\partial x}\right|, \left|\frac{\partial u}{\partial y}\right|\right\} \quad \text{and} \quad m_v = \max\left\{\left|\frac{\partial v}{\partial x}\right|, \left|\frac{\partial v}{\partial y}\right|\right\}.$$

   Then $\max\{m_u, m_v\} \le f(x,y) \le m_u + m_v$.

When $\lambda$ indicates minification, the value assigned to `TEXTURE_MIN_FILTER` is used to determine how the texture value for a fragment is selected. When `TEXTURE_MIN_FILTER` is `NEAREST`, the texel nearest (in Manhattan distance) to that specified by $(s,t)$ is obtained. This means the texel at location $(i,j)$ becomes the texture value, with $i$ given by

$$i = \begin{cases} \lfloor u \rfloor, & s < 1, \\ 2^n - 1, & s = 1. \end{cases} \tag{3.9}$$

(Recall that if `TEXTURE_WRAP_S` is `REPEAT`, then $0 \leq s < 1$.) Similarly, $j$ is found as

$$j = \begin{cases} \lfloor v \rfloor, & t < 1, \\ 2^m - 1, & t = 1. \end{cases} \tag{3.10}$$

For a one-dimensional texture, $j$ is irrelevant; the texel at location $i$ becomes the texture value.

When `TEXTURE_MIN_FILTER` is `LINEAR`, a $2 \times 2$ square of texels is selected. This square is obtained by first computing

$$i_0 = \begin{cases} \lfloor u - 1/2 \rfloor \bmod 2^n, & \texttt{TEXTURE\_WRAP\_S} \text{ is } \texttt{REPEAT}, \\ \lfloor u - 1/2 \rfloor, & \texttt{TEXTURE\_WRAP\_S} \text{ is } \texttt{CLAMP} \end{cases}$$

and

$$j_0 = \begin{cases} \lfloor v - 1/2 \rfloor \bmod 2^m, & \texttt{TEXTURE\_WRAP\_T} \text{ is } \texttt{REPEAT} \\ \lfloor v - 1/2 \rfloor, & \texttt{TEXTURE\_WRAP\_T} \text{ is } \texttt{CLAMP}. \end{cases}$$

Then

$$i_1 = \begin{cases} (i_0 + 1) \bmod 2^n, & \texttt{TEXTURE\_WRAP\_S} \text{ is } \texttt{REPEAT}, \\ i_0 + 1, & \texttt{TEXTURE\_WRAP\_S} \text{ is } \texttt{CLAMP} \end{cases}$$

and

$$j_1 = \begin{cases} (j_0 + 1) \bmod 2^m, & \texttt{TEXTURE\_WRAP\_T} \text{ is } \texttt{REPEAT}, \\ j_0 + 1, & \texttt{TEXTURE\_WRAP\_T} \text{ is } \texttt{CLAMP}. \end{cases}$$

Let

$$\alpha = frac(u - 1/2) \quad \text{and} \quad \beta = frac(v - 1/2)$$

where $frac(x)$ denotes the fractional part of $x$. Let $\tau_{ij}$ be the texel at location $(i,j)$ in the texture image. Then the texture value, $\tau$ is found as

$$\tau = (1-\alpha)(1-\beta)\tau_{i_0 j_0} + \alpha(1-\beta)\tau_{i_1 j_0} + (1-\alpha)\beta\tau_{i_0 j_1} + \alpha\beta\tau_{i_1 j_1} \tag{3.11}$$

for a two-dimensional texture. For a one-dimensional texture,

$$\tau = (1-\alpha)\tau_{i_0} + \alpha\tau_{i_1}$$

where $\tau_i$ indicates the texel at location $i$ in the one-dimensional texture. If any of the selected $\tau_{ij}$ (or $\tau_i$) in the above equations refer to a border texel with unspecified value, then the border color given by the current setting of `TEXTURE_BORDER_COLOR` is used instead of the unspecified value or values.

### Mipmapping

TEXTURE_MIN_FILTER values NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, and LINEAR_MIPMAP_LINEAR each require the use of a *mipmap*. A mipmap is an ordered set of arrays representing the same image; each array has a resolution lower than the previous one. If the texture has dimensions $2^n \times 2^m$, then there are $\max\{n, m\} + 1$ mipmap arrays. The first array is the original texture with dimensions $2^n \times 2^m$. Each subsequent array has dimensions $2^{(k-1)} \times 2^{(l-1)}$ where $2^k \times 2^l$ are the dimensions of the previous array. This is the case as long as both $k > 0$ and $l > 0$. Once either $k = 0$ or $l = 0$, each subsequent array has dimension $1 \times 2^{(l-1)}$ or $2^{(k-1)} \times 1$, respectively, until the last array is reached with dimension $1 \times 1$.

Each array in a mipmap is transmitted to the GL using **TexImage2D** or **TexImage1D**; the array being set is indicated with the *level-of-detail* argument. Level-of-detail numbers proceed from 0 for the original texture array through $p = \max\{n, m\}$ with each unit increase indicating an array of half the dimensions of the previous one as already described. If texturing is enabled (and TEXTURE_MIN_FILTER is one that requires a mipmap) at the time a primitive is rasterized and if the set of arrays 0 through $p$ is incomplete, based on the dimensions of array 0, then it is as if texture mapping were disabled. The set of arrays 0 through $p$ is incomplete if the numbers of components in each mipmap array are not the same, or if the border widths of the mipmap arrays are not the same, or if the dimensions of the mipmap arrays do not follow the sequence described above. Arrays indexed greater than $p$ are insignificant.

The mipmap is used in conjunction with the level of detail to approximate the application of an appropriately filtered texture to a fragment. Let $p = \max\{n, m\}$ and let $c$ be the value of $\lambda$ at which the transition from minification to magnification occurs (since this discussion pertains to minification, we are concerned only with values of $\lambda$ where $\lambda > c$). For NEAREST_MIPMAP_NEAREST, if $c < \lambda \leq 0.5$ then the mipmap array with level-of-detail of 0 is selected. Otherwise, the $d$th mipmap array is selected when $d - \frac{1}{2} < \lambda \leq d + \frac{1}{2}$ as long as $1 < d < p$. If $\lambda > p + \frac{1}{2}$, then the $p$th mipmap array is selected. The rules for NEAREST are then applied to the selected array.

The same mipmap array selection rules apply for LINEAR_MIPMAP_NEAREST as for NEAREST_MIPMAP_NEAREST but the rules for LINEAR are applied to the selected array.

For NEAREST_MIPMAP_LINEAR, the level $d-1$ and the level $d$ mipmap arrays are selected, where $d-1 \leq \lambda < d$, unless $\lambda \geq p$, in which case the $p$th mipmap array is used for both arrays. The rules for `NEAREST` are then applied to each of these arrays, yielding two corresponding texture values $\tau_{d-1}$ and $\tau_d$. The final texture value is then found as

$$\tau = (1 - frac[\log_2(\rho)])\tau_{d-1} + frac[\log_2(\rho)]\tau_d.$$

`LINEAR_MIPMAP_LINEAR` has the same effect as `NEAREST_MIPMAP_LINEAR` except that the rules for `LINEAR` are applied for each of the two mipmap arrays to generate $\tau_{d-1}$ and $\tau_d$.

### 3.8.2 Texture Magnification

When $\rho$ indicates magnification, the value assigned to `TEXTURE_MAG_FILTER` determines how the texture value is obtained. There are two possible values for `TEXTURE_MAG_FILTER`: `NEAREST` and `LINEAR`. `NEAREST` behaves exactly as `NEAREST` for `TEXTURE_MIN_FILTER` (equation 3.9 and 3.10 are used); `LINEAR` behaves exactly as `LINEAR` for `TEXTURE_MIN_FILTER` (equation 3.11 is used). The level-of-detail 0 texture array is always used for magnification.

Finally, there is the choice of $c$, the minification vs. magnification switch-over point. If the magnification filter is given by `LINEAR` and the minification filter is given by `NEAREST_MIPMAP_NEAREST` or `LINEAR_MIPMAP_NEAREST`, then $c = 0.5$. This is done to ensure that a minified texture does not appear "sharper" than a magnified texture. Otherwise $c = 0$.

The state necessary for texture can be divided into two categories. First, there are the two sets of mipmap arrays (one-dimensional and two-dimensional) and their number. Each array has associated with it a width and height (two-dimensional only), a border width, and a four-valued integer describing the number of components in the image. Each initial texture array is null (zero width and height, zero border width, 1 component). Next, there are the two sets of texture properties; each consists of the selected minification and magnification filters, the wrap modes for $s$ and $t$, and the `TEXTURE_BORDER_COLOR`. In the initial state, the value assigned to `TEXTURE_MIN_FILTER` is `NEAREST_MIPMAP_LINEAR`, and the value for `TEXTURE_MAG_FILTER` is `LINEAR`. Both $s$ and $t$ wrap modes are set to `REPEAT`. `TEXTURE_BORDER_COLOR` is (0,0,0,0).

### 3.8.3 Texture Environments and Texture Functions

The command

void **TexEnv{if}**( enum *target*, enum *pname*, T *param* ) ;
void **TexEnv{if}v**( enum *target*, enum *pname*, T *params* ) ;

sets parameters of the *texture environment* that specifies how texture values are interpreted when texturing a fragment. *target* must currently be the symbolic constant `TEXTURE_ENV`. *pname* is a symbolic constant indicating the parameter to be set. In the first form of the command, *param* is a value to which to set a single-valued parameter; in the second form, *params* is a pointer to an array of parameters: either a single symbolic constant or a value or group of values to which the parameter should be set. The possible environment parameters are `TEXTURE_ENV_MODE` and `TEXTURE_ENV_COLOR`. `TEXTURE_ENV_MODE` may be set to one of `MODULATE`, `DECAL`, or `BLEND`; `TEXTURE_ENV_COLOR` is set to an RGBA color by providing four single-precision floating-point values in the range $[0,1]$ (values outside this range are clamped to it). If integers are provided for `TEXTURE_ENV_COLOR`, then they are converted to floating-point as specified in Table 2.4 for signed integers.

The value of `TEXTURE_ENV_MODE` specifies a *texture function*. The result of this function depends on the fragment and the texture array value. The precise form of the function depends on the number of components of the texture arrays that were last specified. In the following table, $C$ is a triple of color values comprising each of R, G, and B, while $A$ (A) is treated separately. R, G, B, and A values, after being obtained from a supplied texture image, are in the range $[0,1]$. The subscript $f$ indicates a value or values pertaining to the incoming fragment, $t$ indicates a texture value, and $v$ indicates the color computed by the texture function. For a one component image, $L_t$ indicates that single component. For a two component image, $L_t$ is the first component, and $A_t$ is the second. A three component image has only a color value $C_t$, while a four component one has a color value $C_t$ and and alpha value $A_t$. The functions for the various combinations are summarized in Table 3.8.

The state required for the current texture environment consists of the three-valued integer indicating the texture function and four floating-point `TEXTURE_ENV_COLOR` values. In the initial state, the texture function is given by `MODULATE` and `TEXTURE_ENV_COLOR` is $(0,0,0,0)$.

### 3.8.4 Texture Application

Texturing is enabled or disabled using the generic **Enable** and **Disable** commands, respectively, with the symbolic constant `TEXTURE_1D` or `TEXTURE_2D` to

| | Texture Functions | | |
|---|---|---|---|
| cpts | `MODULATE` | `DECAL` | `BLEND` |
| 1 | $C_v = L_t C_f$<br>$A_v = A_f$ | undefined | $C_v = (1 - L_t)C_f + L_t C_c$<br>$A_v = A_f$ |
| 2 | $C_v = L_t C_f$<br>$A_v = A_t A_f$ | undefined | $C_v = (1 - L_t)C_f + L_t C_c$<br>$A_v = A_t A_f$ |
| 3 | $C_v = C_t C_f$<br>$A_v = A_f$ | $C_v = C_t$<br>$A_v = A_f$ | undefined |
| 4 | $C_v = C_t C_f$<br>$A_v = A_t A_f$ | $C_v = (1 - A_t)C_f + A_t C_t$<br>$A_v = A_f$ | undefined |

Table 3.8: Texture functions. Multiplication of a color triple by a scalar means multiplying each of R, G, and B by the scalar; multiplying two color triples means multiplying each component of the second by the corresponding component of the first. $C_c$ represents the red, green, and blue values assigned to `TEXTURE_ENV_COLOR`. $C_f$ and $C_v$ represent the red, green, and blue components of the fragment color prior to and after texture application. $A_f$ and $A_v$ represent the alpha component of the fragment prior to and after texture application.

enable the one-dimensional or two-dimensional texture, respectively. If both one- and two-dimensional textures are enabled, the two-dimensional texture is used. If all texturing is disabled, a rasterized fragment is passed on unaltered to the next stage of the GL (although its texture coordinates may be discarded). Otherwise, a texture value is found according to the parameter values of the currently bound texture image of the appropriate dimensionality using the rules given in sections 3.8.1 and 3.8.2. This texture value is used along with the incoming fragment in computing the texture function indicated by the currently bound texture environment. The result of this function replaces the incoming fragment's R, G, B, and A values. These are the color values passed to subsequent operations. Other data associated with the incoming fragment remain unchanged, except that the texture coordinates may be discarded.

The required state is two bits indicating whether each of one- or two-dimensional texturing is enabled or disabled. In the initial state, all texturing is disabled.

## 3.9 Fog

If enabled, fog blends a fog color with a rasterized fragment's post-texturing color using a blending factor $f$. Fog is enabled and disabled with the **Enable** and **Disable** commands using the symbolic constant `FOG`.

This factor $f$ is computed according to one of three equations:

$$f = \exp(-d \cdot z), \tag{3.12}$$

$$f = \exp(-(d \cdot z)^2), \text{or} \tag{3.13}$$

$$f = \frac{e - z}{e - s} \tag{3.14}$$

($z$ is the eye-coordinate distance from the eye, $(0, 0, 0, 1)$ in eye coordinates, to the fragment center). The equation, along with either $d$ or $e$ and $s$, is specified with

> `void` **Fog{if}**( `enum` *pname*, `T` *param* ) ;
> `void` **Fog{if}v**( `enum` *pname*, `T` *params* ) ;

If *pname* is `FOG_MODE`, then *param* must be, or *params* must point to an integer that is one of the symbolic constants `EXP`, `EXP2`, or `LINEAR`, in which case equation 3.12, 3.13, or 3.14, respectively, is selected for the fog calculation (if, when 3.14 is selected, $e = s$, results are undefined). If *pname* is `FOG_DENSITY`, `FOG_START`, or `FOG_END`, then *param* is or *params* points to a value that is $d$, $s$, or $e$, respectively. If $d$, $s$, or $e$ is specified less than zero, the error `INVALID_VALUE` results.

An implementation may choose to approximate the eye-coordinate distance from the eye to each fragment center by $|z_e|$. Further, $f$ need not be computed at each fragment, but may be computed at each vertex and interpolated as other data are.

No matter which equation and approximation is used to compute $f$, the result is clamped to $[0, 1]$ to obtain the final $f$.

$f$ is used differently depending on whether the GL is in RGBA or color index mode. In RGBA mode, if $C_r$ represents a rasterized fragment's R, G, or B value, then the corresponding value produced by fog is

$$C = fC_r + (1 - f)C_f.$$

(The rasterized fragment's A value is not changed by fog blending.) The R, G, B, and A values of $C_f$ are specified by calling **Fog** with *pname* equal to `FOG_COLOR`; in this case *params* points to four values comprising $C_f$. If these are not floating-point values, then they are converted to floating-point using the conversion given in Table 2.4 for signed integers. Each component of $C_f$ is clamped to $[0, 1]$ when specified. If $i_f$ is a color index, then a single value specifies $i_f$. Its integer part is masked with $2^n - 1$, where $n$ is the number of bits in a color index framebuffer.

In color index mode, the formula for fog blending is

$$I = i_r + (1 - f)i_f$$

where $i_r$ is the rasterized fragment's color index and $i_f$ is a single-precision floating-point value. $(1 - f)i_f$ is rounded to the nearest fixed-point value with the same number of bits to the right of the binary point as $i_r$. In this case, $i_f$ is set by calling **Fog** with *pname* set to `FOG_INDEX` and *param* being or *params* pointing to the single floating-point value that is $i_f$.

The state required for fog consists of a three valued integer to select the fog equation, three floating-point values $d$, $e$, and $s$, an RGBA fog color and a fog color index, and a single bit to indicate whether or not fog is enabled. In the initial state, fog is disabled, `FOG_MODE` is `EXP`, $d = 1.0$, $e = 1.0$, and $s = 0.0$; $C_f = (0, 0, 0, 0)$ and $i_f = 0$.

## 3.10 Antialiasing Application

Finally, if antialiasing is enabled for the primitive from which a rasterized fragment was produced, then the computed coverage value is applied to the fragment. In RGBA mode, the value is multiplied by the fragment's alpha (A) value to yield a final alpha value. In color index mode, the value is used to set the low order bits of the color index value as described in section 3.2.

# Chapter 4

# Per-Fragment Operations and the Framebuffer

The framebuffer consists of a set of pixels arranged as a two-dimensional array. The height and width of this array may vary from one GL implementation to another. For purposes of this discussion, each pixel in the framebuffer is simply a set of some number of bits. The number of bits per pixel may also vary depending on the particular GL implementation or context.

Corresponding bits from each pixel in the framebuffer are grouped together into a *bitplane*; each bitplane contains a single bit from each pixel. These bitplanes are grouped into several *logical buffers*. These are the *color*, *depth*, *stencil*, and *accumulation* buffers. The color buffer actually consists of a number of buffers: the *front left* buffer, the *front right* buffer, the *back left* buffer, the *back right* buffer, and some number of *auxiliary* buffers. Typically the contents of the front buffers are displayed on a color monitor while the contents of the back buffers are invisible. (Monoscopic contexts display only the front left buffer; stereoscopic contexts display both the front left and the front right buffers.) The contents of the auxiliary buffers are never visible. All color buffers must have the same number of bitplanes, although an implementation or context may choose not to provide right buffers, back buffers, or auxiliary buffers at all. Further, an implementation or context may not provide depth, stencil, or accumulation buffers.

Color buffers consist of either unsigned integer color indices or R, G, B, and, optionally, A unsigned integer values. The number of bitplanes in each of the color buffers, the depth buffer, the stencil buffer, and the

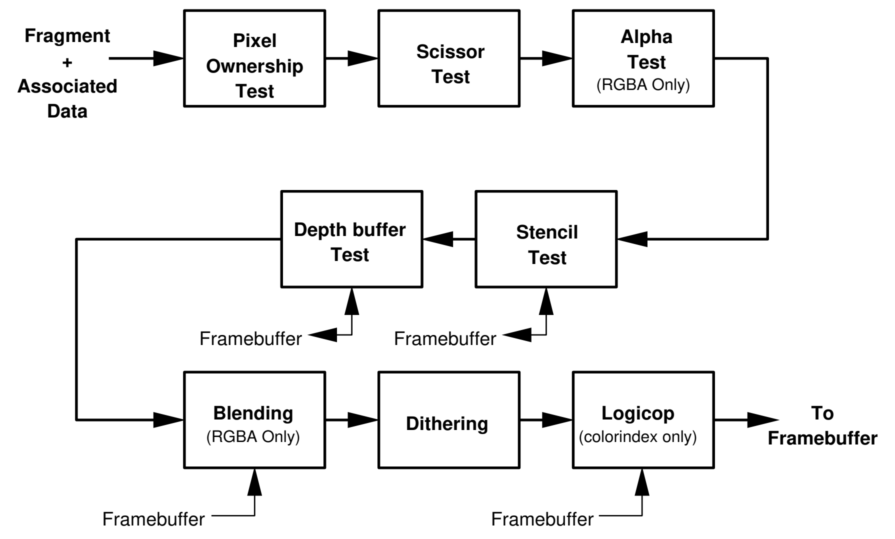


Figure 4.1. Per-fragment operations.

accumulation buffer is fixed and window dependent. If an accumulation buffer is provided, it must have at least as many bitplanes per R, G, and B color component as do the color buffers.

The initial state of all provided bitplanes is undefined.

## 4.1 Per-Fragment Operations

A fragment produced by rasterization with window coordinates of $(x_w, y_w)$ modifies the pixel in the framebuffer at that location based on a number of parameters and conditions. We describe these modifications and tests, diagrammed in Figure 4.1, in the order in which they are performed. Figure 4.1 diagrams these modifications and tests.

### 4.1.1 Pixel Ownership Test

The first test is to determine if the pixel at location $(x_w, y_w)$ in the framebuffer is currently owned by the GL (more precisely, by this GL context). If it is not, the window system decides the fate the incoming fragment. Possible results are that the fragment is discarded or that some subset of the subsequent per-fragment operations are applied to the fragment. This test allows the window system to control the GL's behavior, for instance, when a GL window is obscured.

### 4.1.2 Scissor test

The scissor test determines if $(x_w, y_w)$ lies within the scissor rectangle defined by four values. These values are set with

void **Scissor**( int *left*, int *bottom*, sizei *width*, sizei *height* ) ;

If $left \le x_w < left + width$ and $bottom \le y_w < bottom + height$, then the scissor test passes. Otherwise, the test fails and the fragment is discarded. The test is enabled or disabled using **Enable** or **Disable** using the constant SCISSOR_TEST. When disabled, it is as if the scissor test always passes. If either *width* or *height* is less than zero, then the error INVALID_VALUE is generated. The state required consists of four integer values and a bit indicating whether the test is enabled or disabled. In the initial state $left = bottom = 0$; *width* and *height* are determined by the size of the GL window. Initially, the scissor test is disabled.

### 4.1.3 Alpha test

This step applies only in RGBA mode. In color index mode, proceed to the next step. The alpha test discards a fragment conditional on the outcome of a comparison between the incoming fragment's alpha value and a constant value. The comparison is enabled or disabled with the generic **Enable** and **Disable** commands using the symbolic constant ALPHA_TEST. When disabled, it is as if the comparison always passes. The test is controlled with

void **AlphaFunc**( enum *func*, clampf *ref* ) ;

*func* is a symbolic constant indicating the alpha test function; *ref* is a reference value. *ref* is clamped to lie in $[0, 1]$, and then converted to a fixed-point value according to the rules given for an A component in section 2.12.9. For purposes of the alpha test, the fragment's alpha value is also rounded to the nearest integer. The possible constants specifying the test function are NEVER, ALWAYS, LESS, LEQUAL, EQUAL, GEQUAL, GREATER, or NOTEQUAL, meaning pass the fragment never, always, if the fragment's alpha value is less than, less than or equal to, equal to, greater than or equal to, greater than, or not equal to the reference value, respectively.

The required state consists of the floating-point reference value, an eight-valued integer indicating the comparison function, and a bit indicating if the comparison is enabled or disabled. The initial state is for the reference value to be 0 and the function to be `ALWAYS`. Initially, the alpha test is disabled.

### 4.1.4 Stencil test

The stencil test conditionally discards a fragment based on the outcome of a comparison between the value in the stencil buffer at location $(x_w, y_w)$ and a reference value. The test is controlled with

`void` **StencilFunc**( `enum` *func*, `int` *ref*, `uint` *mask* ) ;
`void` **StencilOp**( `enum` *sfail*, `enum` *dpfail*, `enum` *dppass* ) ;

The test is enabled or disabled with the **Enable** and **Disable** commands, using the symbolic constant `STENCIL_TEST`. When disabled, the stencil test and associated modifications are not made, and the fragment is always passed.

*ref* is an integer reference value that is used in the unsigned stencil comparison. It is clamped to the range $[0, 2^s - 1]$, where $s$ is the number of bits in the stencil buffer. *func* is a symbolic constant that determines the stencil comparison function; the eight symbolic constants are `NEVER`, `ALWAYS`, `LESS`, `LEQUAL`, `EQUAL`, `GEQUAL`, `GREATER`, or `NOTEQUAL`. Accordingly, the stencil test passes never, always, if the reference value is less than, less than or equal to, equal to, greater than or equal to, greater than, or not equal to the masked stored value in the stencil buffer. The $s$ least significant bits of *mask* are bitwise ANDed with both the reference and the stored stencil value. The ANDed values are those that participate in the comparison.

**StencilOp** takes three arguments that indicate what happens to the stored stencil value if this or certain subsequent tests fail or pass. *sfail* indicates what action is taken if the stencil test fails. The symbolic constants are `KEEP`, `ZERO`, `REPLACE`, `INCR`, `DECR`, and `INVERT`. These correspond to keeping the current value, setting it to zero, replacing it with the reference value, incrementing it, decrementing it, or bitwise inverting it. For purposes of increment and decrement, the stencil bits are considered as an unsigned integer; values clamp at 0 and the maximum representable value. The same symbolic values are given to indicate the stencil action if the depth buffer test (below) fails (*dpfail*), or if it passes (*dppass*).

If the stencil test fails, the incoming fragment is discarded. The state required consists of the most recent values passed to **StencilFunc** and **StencilOp**, and a bit indicating whether stencil testing is enabled or disabled. In the initial state, stenciling is disabled, the stencil reference value is zero, the stencil comparison function is `ALWAYS`, and the stencil *mask* is all ones. Initially, all three stencil operations are `KEEP`. If there is no stencil buffer, no stencil modification can occur, and it is as if the stencil tests always pass, regardless of any calls to **StencilOp**.

### 4.1.5 Depth buffer test

The depth buffer test discards the incoming fragment if a depth comparison fails. The comparison is enabled or disabled with the generic **Enable** and **Disable** commands using the symbolic constant `DEPTH_TEST`. When disabled, the depth comparison and subsequent possible updates to the depth buffer value are bypassed and the fragment is passed to the next operation. The stencil value, however, is modified as indicated below as if the depth buffer test passed. If enabled, the comparison takes place and the depth buffer and stencil value may subsequently be modified.

The comparison is specified with

`void` **DepthFunc**( `enum` *func* ) ;

This command takes a single symbolic constant: one of `NEVER`, `ALWAYS`, `LESS`, `LEQUAL`, `EQUAL`, `GREATER`, `GEQUAL`, `NOTEQUAL`. Accordingly, the depth buffer test passes never, always, if the incoming fragment's $z_w$ value is less than, less than or equal to, equal to, greater than, greater than or equal to, or not equal to the depth value stored at the location given by the incoming fragment's $(x_w, y_w)$ coordinates.

If the depth buffer test fails, the incoming fragment is discarded. The stencil value at the fragment's $(x_w, y_w)$ coordinates is updated according to the function currently in effect for depth buffer test failure. Otherwise, the fragment continues to the next operation and the value of the depth buffer at the fragment's $(x_w, y_w)$ location is set to the fragment's $z_w$ value. In this case the stencil value is updated according to the function currently in effect for depth buffer test success.

The necessary state is an eight-valued integer and a single bit indicating whether depth buffering is enabled or disabled. In the initial state the function is `LESS` and the test is disabled.

If there is no depth buffer, it is as if the depth buffer test always passes.

### 4.1.6 Blending

Blending combines the incoming fragment's R, G, B, and A values with the R, G, B, and A values stored in the framebuffer at the incoming fragment's $(x_w, y_w)$ location. This blending is dependent on the incoming fragment's alpha value and that of the corresponding currently stored pixel. Blending applies only in RGBA mode; in color index mode it is bypassed. Blending is enabled or disabled using **Enable** or **Disable** with the symbolic constant `BLEND`. If it is disabled, proceed to the next stage.

The command that controls blending is

`void` **BlendFunc**( `enum` *src*, `enum` *dst* ) ;

*src* indicates how to compute a source blending factor, while *dst* indicates how to compute a destination factor. The possible arguments and their corresponding computed source and destination factors are summarized in Tables 4.1 and 4.2. In these tables, $A$ is a single alpha value, and $C$ is a quadruplet of R, G, B, and A values. A subscript of $s$ indicates a value from an incoming fragment; one of $d$ indicates the corresponding current framebuffer value. Division of a quadruplet by a scalar means dividing each element by that value. Addition or subtraction of quadruplets or triplets means adding or subtracting them component-wise.

The computations in Tables 4.1 and 4.2 are effectively carried out in floating-point and yield floating-point blending factors. Destination (framebuffer) components referred to in the tables are taken to be fixed-point values represented according to the scheme given in section 2.12.9 (Final Color Processing), as are source (fragment) components. Any implied conversion to floating-point must leave 0 and 1 invariant.

The computed source and destination blending quadruplets are applied to the source and destination R, G, B, and A values to obtain a new set of values that are sent to the next operation. Let the source and destination blending quadruplets be $S$ and $D$, respectively. Then a quadruplet of values is computed as

$$C_s S + C_d D,$$

where multiplication of quadruplets means multiplying them component-wise. Then each value in this quadruplet is clamped to $2^n - 1$, where $n$ is the number of bits allocated to that color component in the framebuffer, and the four values are sent to the next operation.

The state required is two integers indicating the source and destination blending functions and a bit indicating whether blending is enabled

| Value | Blend Factors |
| --- | --- |
| ZERO | $(0,0,0,0)$ |
| ONE | $(1,1,1,1)$ |
| DST_COLOR | $R_d, G_d, B_d, A_d$ |
| ONE_MINUS_DST_COLOR | $(1,1,1,1) - (R_d, G_d, B_d, A_d)$ |
| SRC_ALPHA | $(A_s, A_s, A_s, A_s)$ |
| ONE_MINUS_SRC_ALPHA | $(1,1,1,1) - (A_s, A_s, A_s, A_s)$ |
| DST_ALPHA | $(A_d, A_d, A_d, A_d)$ |
| ONE_MINUS_DST_ALPHA | $(1,1,1,1) - (A_d, A_d, A_d, A_d)$ |
| SRC_ALPHA_SATURATE | $(f, f, f, 1)$ |

Table 4.1: Values controlling the source blending function and the source blending values they compute. $f = \min(A_s, 1 - A_d)$.

| Value | Blend factors |
| --- | --- |
| ZERO | $(0,0,0,0)$ |
| ONE | $(1,1,1,1)$ |
| SRC_COLOR | $R_s, G_s, B_s, A_s$ |
| ONE_MINUS_SRC_COLOR | $(1,1,1,1) - (R_s, G_s, B_s, A_s)$ |
| SRC_ALPHA | $(A_s, A_s, A_s, A_s)$ |
| ONE_MINUS_SRC_ALPHA | $(1,1,1,1) - (A_s, A_s, A_s, A_s)$ |
| DST_ALPHA | $(A_d, A_d, A_d, A_d)$ |
| ONE_MINUS_DST_ALPHA | $(1,1,1,1) - (A_d, A_d, A_d, A_d)$ |

Table 4.2: Values controlling the destination blending function and the destination blending values they compute.

or disabled. The initial state of the blending functions is `ONE` for the source function and `ZERO` for the destination function; initially, blending is disabled.

Blending occurs once for each color buffer currently enabled for writing (section 4.2.1) using each buffer's color for $C_d$. If a color buffer has no A value, then it is as if the destination A value is 1.

### 4.1.7 Dithering

Dithering selects between two color values or indices. In RGBA mode, consider the value of any of the color components as a fixed-point value with $m$ bits to the left of the binary point, where $m$ is the number of bits allocated to that component in the framebuffer; call each such value $c$. For each $c$, dithering selects a value $c_1$ such that $c_1 \in \{\max\{0, \lceil c \rceil - 1\}, \lceil c \rceil\}$ (after this selection, treat $c_1$ as a fixed point value in [0,1] with $m$ bits). This selection may depend on the $x_w$ and $y_w$ coordinates of the pixel. In color index mode, the same rule applies with $c$ being a single color index. $c$ must not be larger than the maximum value representable in the framebuffer for either the component or the index, as appropriate.

Many dithering algorithms are possible, but a dithered value produced by any algorithm must depend only the incoming value and the fragment's $x$ and $y$ window coordinates. If dithering is disabled, then each color component is truncated to a fixed-point value with as many bits as there are in the corresponding component in the framebuffer; a color index is rounded to the nearest integer representable in the color index portion of the framebuffer.

Dithering is enabled with **Enable** and disabled with **Disable** using the symbolic constant `DITHER`. The state required is thus a single bit. Initially, dithering is enabled. In RGBA mode, this is the last operation, and the result goes into the framebuffer. In color index mode, continue on to the last operation.

### 4.1.8 Logical Operation

Finally, a logical operation is applied between the incoming fragment and the value stored at the corresponding location in the framebuffer; the result replaces the current framebuffer value. The logical operation is enabled or disabled with **Enable** or **Disable** using the symbolic constant `LOGIC_OP`. The logical operation is selected by

void **LogicOp**( enum *op* ) ;

| Argument value | Operation |
|---|---|
| CLEAR | $0$ |
| AND | $s \wedge d$ |
| AND_REVERSE | $s \wedge \neg d$ |
| COPY | $s$ |
| AND_INVERTED | $\neg s \wedge d$ |
| NOOP | $d$ |
| XOR | $s\ xor\ d$ |
| OR | $s \vee d$ |
| NOR | $\neg(s \vee d)$ |
| EQUIV | $\neg(s\ xor\ d)$ |
| INVERT | $\neg d$ |
| OR_REVERSE | $s \vee \neg d$ |
| COPY_INVERTED | $\neg s$ |
| OR_INVERTED | $\neg s \vee d$ |
| NAND | $\neg(s \wedge d)$ |
| SET | $1$ |

Table 4.3: Arguments to **LogicOp** and their corresponding operations.

*op* is a symbolic constant. The possible constants and the corresponding logical operations are enumerated in Table 4.3; in this table, $s$ is the value of the incoming fragment and $d$ is the value stored in the framebuffer.

Note that the SET operation sets all bits of the result to 1. The result replaces the value in the framebuffer at the fragment's $(x, y)$ coordinates. The numeric values assigned to the symbolic constants are the same as the those assigned to the corresponding symbolic values in the X window system.

**LogicOp** applies only in color index mode; in RGBA mode it does not occur and the previous operation is the last one applied to incoming fragments. **LogicOp** occurs once for each color buffer selected for writing. The required state is an integer indicating the logical operation, and a bit to indicate whether the logical operation is enabled or disabled. The initial state is for the logic operation to be given by COPY, and it is disabled.

## 4.2 Whole Framebuffer Operations

The preceding sections described the operations that occur as individual fragments are sent to the framebuffer. This section describes operations that control or affect the whole framebuffer.

### 4.2.1 Selecting a Buffer for Writing

The first such operation is controlling the buffer into which color values are written. This is accomplished with

void **DrawBuffer**( enum *buf* ) ;

*buf* is a symbolic constant specifying zero, one, two, or four buffers for writing. The constants are `NONE`, `FRONT_LEFT`, `FRONT_RIGHT`, `BACK_LEFT`, `BACK_RIGHT`, `FRONT`, `BACK`, `LEFT`, `RIGHT`, `FRONT_AND_BACK`, and `AUX0` through `AUX`$n$, where $n+1$ is the number of available auxiliary buffers.

The constants refer to the four potentially visible buffers *front_left*, *front_right*, *back_left*, and *back_right*, and to the *auxiliary* buffers. Arguments other than `AUX`$i$ that omit reference to `LEFT` or `RIGHT` refer to both left and right buffers. Arguments other than `AUX`$i$ that omit reference to `FRONT` or `BACK` refer to both front and back buffers. `AUX`$i$ enables drawing only to *auxiliary* buffer $i$. Each `AUX`$i$ adheres to `AUX`$i$ = `AUX0` + $i$. The constants and the buffers they indicate are summarized in Table 4.4. If **DrawBuffer** is is supplied with a constant (other than `NONE`) that does not indicate any of the color buffers allocated to the GL context, the error `INVALID_OPERATION` results.

Indicating a buffer or buffers using **DrawBuffer** causes subsequent pixel color value writes to affect the indicated buffers. If more than one color buffer is selected for drawing, blending and logical operations are computed and applied independently for each buffer. Calling **DrawBuffer** with a value of `NONE` inhibits the writing of color values to any buffer.

Monoscopic contexts include only left buffers, while stereoscopic contexts include both left and right buffers. Likewise, single buffered contexts include only front buffers, while double buffered contexts include both front and back buffers. The type of context is selected at GL initialization.

The state required to handle buffer selection is a set of up to 4 bits, indicating whether the front left buffer, the front right buffer, the back left buffer, the back right buffer, and potentially integer to indicate which, if any, of the auxiliary buffers is enabled for color writing. In the initial state,

| symbolic constant | front left | front right | back left | back right | aux $i$ |
|---|---|---|---|---|---|
| NONE | | | | | |
| FRONT_LEFT | • | | | | |
| FRONT_RIGHT | | • | | | |
| BACK_LEFT | | | • | | |
| BACK_RIGHT | | | | • | |
| FRONT | • | • | | | |
| BACK | | | • | • | |
| LEFT | • | | • | | |
| RIGHT | | • | | • | |
| FRONT_AND_BACK | • | • | • | • | |
| AUX$i$ | | | | | • |

Table 4.4: Arguments to **DrawBuffer** and the buffers that they indicate.

the front buffer or buffers are enabled if there are no back buffers; otherwise, only the back buffer or buffers are enabled.

### 4.2.2 Fine Control of Buffer Updates

Four commands are used to mask the writing of bits to each of the logical framebuffers after all per-fragment operations have been performed. The commands

void **IndexMask**( uint *mask* ) ;
void **ColorMask**( boolean *r*, boolean *g*, boolean *b*, boolean *a* ) ;

control the color buffer or buffers (depending on which buffers are currently indicated for writing). The least significant $n$ bits of *mask*, where $n$ is the number of bits in a color index buffer, specify a mask. Where a 1 appears in this mask, the corresponding bit in the color index buffer (or buffers) is written; where a 0 appears, the bit is not written. This mask applies only in color index mode. In RGBA mode, **ColorMask** is used to mask the writing of R, G, B and A values to the color buffer or buffers. $r$, $g$, $b$, and $a$ indicate whether R, G, B, or A values, respectively, are written or not (a value of TRUE means that the corresponding value is written). In the initial state, all bits (in color index mode) and all color values (in RGBA mode) are enabled for writing.

The depth buffer can be enabled or disabled for writing $z_w$ values using

void **DepthMask**( boolean *mask* ) ;

If *mask* is non-zero, the depth buffer is enabled for writing; otherwise, it is disabled. In the initial state, the depth buffer is enabled for writing.

The command

void **StencilMask**( uint *mask* ) ;

controls the writing of particular bits into the stencil planes. The least significant $s$ bits of *mask* comprise an integer mask ($s$ is the number of bits in the stencil buffer), just as for **IndexMask**. The initial state is for the stencil plane mask to be all ones.

The state required for the various masking operations is two integers and a bit: an integer for color indices, an integer for stencil values, and a bit for depth values. A set of four bits is also required indicating which color components of an RGBA value should be written. In the initial state, the integer masks are all ones as are the bits controlling depth value and RGBA component writing.

### 4.2.3 Clearing the Buffers

The GL provides a means for setting portions of every pixel in a particular buffer to the same value. The argument to

void **Clear**( bitfield *buf* ) ;

is the bitwise OR of a number of values indicating which buffers are to be cleared. The values are COLOR_BUFFER_BIT, DEPTH_BUFFER_BIT, STENCIL_BUFFER_BIT, and ACCUM_BUFFER_BIT, indicating the buffers currently enabled for color writing, the depth buffer, the stencil buffer, and the accumulation buffer (see below), respectively. The value to which each buffer is cleared depends on the setting of the clear value for that buffer. If the mask is not a bitwise OR of the specified values, then the error INVALID_VALUE is generated.

void **ClearColor**( clampf *r*, clampf *g*, clampf *b*, clampf *a* ) ;

sets the clear value for the color buffers in RGBA mode. Each of the specified components is clamped to $[0,1]$ and converted to fixed-point according to the rules of section 2.12.9.

`void` **ClearIndex**( `float` *index* ) ;

sets the clear color index. *index* is converted to a fixed-point value with unspecified precision to the left of the binary point; the integer part of this value is then masked with $2^m - 1$, where $m$ is the number of bits in a color index value stored in the framebuffer.

`void` **ClearDepth**( `clampd` *d* ) ;

takes a floating-point value that is clamped to the range $[0,1]$ and converted to fixed-point according to the rules for a window $z$ value given in section 2.9.1. Similarly,

`void` **ClearStencil**( `int` *s* ) ;

takes a single integer argument that is the value to which to clear the stencil buffer. $s$ is masked to the number of bitplanes in the stencil buffer.

`void` **ClearAccum**( `float` *r*, `float` *g*, `float` *b*, `float` *a* ) ;

takes four floating-point arguments that are the values, in order, to which to set the R, G, B, and A values of the accumulation buffer (see the next section). These values are clamped to the range $[-1,1]$ when they are specified.

When **Clear** is called, the only per-fragment operations that are applied (if enabled) are the pixel ownership test, the scissor test, and dithering. The masking operations described in the last section (4.2.2) are also effective. If a buffer is not present, then a **Clear** directed at that buffer has no effect.

The state required for clearing is a clear value for each of the color buffer, the depth buffer, the stencil buffer, and the accumulation buffer. Initially, the RGBA color clear value is (0,0,0,0), the clear color index is 0, and the stencil buffer and accumulation buffer clear values are all 0. The depth buffer clear value is initially 1.0.

### 4.2.4 The Accumulation Buffer

Each portion of a pixel in the accumulation buffer consists of four values: one for each of R, G, B, and A. The accumulation buffer is controlled exclusively through the use of

```
void Accum( enum op, float value ) ;
```

(except for clearing it). *op* is a symbolic constant indicating an accumulation buffer operation, and *value* is a floating-point value to be used in that operation. The possible operations are ACCUM, LOAD, RETURN, MULT, and ADD.

The accumulation buffer operations apply identically to every pixel, so we describe the effect of each operation on an individual pixel. Accumulation buffer values are taken to be signed values in the range $[-1, 1]$. Using ACCUM obtains R, G, B, and A components from the buffer currently selected for reading (section 4.3.2). Each component, considered as a fixed-point value in [0,1] (see section 2.12.9), is converted to floating-point. Each result is then multiplied by *value*. The results of this multiplication are then added to the corresponding color component currently in the accumulation buffer, and the resulting color value replaces the current accumulation buffer color value. The LOAD operation has the same effect as ACCUM, but the computed values replace the corresponding accumulation buffer components rather than being added to them.

The RETURN operation takes each color value from the accumulation buffer, multiplies each of the R, G, B, and A components by *value*. The resulting color value is placed in the buffers currently enabled for color writing as if it were a fragment produced from rasterization, except that the only per-fragment operations applied are the pixel ownership test and, if enabled, dithering (section 4.1); color masking (section 4.2.2) is also applied.

The MULT operation multiplies each R, G, B, and A in the accumulation buffer by *value* and then returns the scaled color components to their corresponding accumulation buffer locations. ADD is the same as MULT except that *value* is added to each of the color components.

The color components operated on by **Accum** must be clamped only if the operation is RETURN. In this case, a value sent to the enabled color buffers is first clamped to $[0, 1]$. Otherwise, results are undefined if the result of an operation on a color component is too large (in magnitude) to be represented by the number of available bits. When the scissor test is enabled (section 4.1.2), then only those pixels within the current scissor box are updated by any **Accum** operation; otherwise, all pixels in the window are updated. If there is no accumulation buffer, or if the GL is in color index mode, **Accum** generates the error INVALID_OPERATION.

No state (beyond the accumulation buffer itself) is required for accumulation buffering.

## 4.3 Drawing, Reading, and Copying Pixels

Pixels may be written to and read from the framebuffer using the **DrawPixels** and **ReadPixels** commands. **CopyPixels** can be used to copy a block of pixels from one portion of the framebuffer to another.

### 4.3.1 Writing to the Stencil Buffer

The operation of **DrawPixels** was described in section 3.6.3, except if the *format* argument was `STENCIL_INDEX`. In this case, all operations described for **DrawPixels** take place, but window $(x, y)$ coordinates, each with the corresponding stencil index, are produced in lieu of fragments. Each coordinate-stencil index pair is sent directly to the per-fragment operations, bypassing the texture, fog, and antialiasing application stages of rasterization. Each pair is then treated as a fragment for purposes of the pixel ownership and scissor tests; all other per-fragment operations are bypassed. Finally, each stencil index is written to its indicated location in the framebuffer, subject to the current setting of **StencilMask**.

The error `INVALID_OPERATION` results if there is no stencil buffer.

### 4.3.2 Reading Pixels

The method for reading pixels from the framebuffer and placing them in client memory is diagrammed in Figure 4.2. We describe the stages of the pixel reading process in the order in which they occur.

Pixels are read using

void **ReadPixels**( int *x*, int *y*, sizei *width*, sizei *height*,
enum *format*, enum *type*, void **data* ) ;

The arguments after $x$ and $y$ to **ReadPixels** correspond to those of **DrawPixels**. The pixel storage modes that apply to **ReadPixels** are summarized in Table 4.5.

#### Obtaining Pixels from the Framebuffer

If the *format* is `DEPTH_COMPONENT`, then values are obtained from the depth buffer. If there is no depth buffer, the error `INVALID_OPERATION` occurs. If the *format* is `STENCIL_INDEX`, then values are taken from the stencil buffer; again, if there is no stencil buffer, the error `INVALID_OPERATION` occurs. For

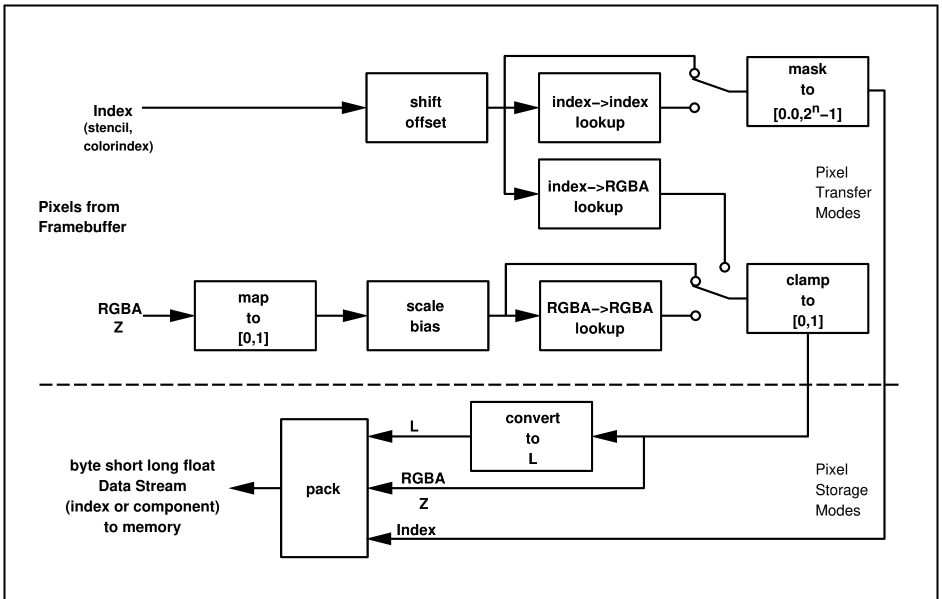


Figure 4.2. Operation of **ReadPixels**. The parameters controlling the stages above the dotted line are set with **PixelTransfer** or **PixelMap** while those controlling the stages below the line are set with **PixelStore**.

| Parameter Name | Type | Initial Value | Valid Range |
|---|---|---|---|
| `PACK_SWAP_BYTES` | boolean | `FALSE` | `TRUE`/`FALSE` |
| `PACK_LSB_FIRST` | boolean | `FALSE` | `TRUE`/`FALSE` |
| `PACK_ROW_LENGTH` | integer | 0 | $[0, \infty)$ |
| `PACK_SKIP_ROWS` | integer | 0 | $[0, \infty)$ |
| `PACK_SKIP_PIXELS` | integer | 0 | $[0, \infty)$ |
| `PACK_ALIGNMENT` | integer | 4 | 1,2,4,8 |

Table 4.5: **PixelStore** parameters pertaining to **ReadPixels**.

all other formats, the buffer from which values are obtained is one of the color buffers; the selection of color buffer is controlled with **ReadBuffer**.

The command

```
void ReadBuffer( enum src ) ;
```

takes a symbolic constant as argument. The possible values are FRONT_LEFT, FRONT_RIGHT, BACK_LEFT, BACK_RIGHT, FRONT, BACK, LEFT, RIGHT, and AUX0 through AUX$n$. FRONT and LEFT refer to the front left buffer, BACK refers to the back left buffer, and RIGHT refers to the front right buffer. The other constants correspond directly to the buffers that they name. If the requested buffer is missing, then the error INVALID_OPERATION is generated. The initial setting for **ReadBuffer** is FRONT if there is no back buffer and BACK otherwise.

**ReadPixels** obtains values from the selected buffer from each pixel with lower left hand corner at $(x + i, y + j)$ for $0 \le i < width$ and $0 \le j < height$; this pixel is said to be the $i$th pixel in the $j$th row. If any of these pixels lies outside of the window allocated to the current GL context, the values obtained for those pixels are undefined. Results are also undefined for individual pixels that are not owned by the current context. Otherwise, **ReadPixels** obtains values from the selected buffer, regardless of how those values were placed there.

The number of values obtained from the selected buffer depends on the *format*. If the *format* is LUMINANCE, R, G, and B values are obtained, while if it is LUMINANCE_ALPHA, then R, G, B, and A values are obtained. If the framebuffer does not support alpha values then the A that is obtained is 1.0. If the *format* is one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, LUMINANCE, or LUMINANCE_ALPHA, and the GL is in color index mode, then the color index is obtained. Otherwise, Table 3.5 gives the type and number of values that are obtained from the selected buffer for each pixel.

### Conversion of RGBA values

This step applies only if the GL is in RGBA mode, and then only if *format* is neither STENCIL_INDEX nor DEPTH_COMPONENT. The error INVALID_OPERATION results (in RGBA mode) if *format* is COLOR_INDEX.

The R, G, and B (and possibly A) values form a group of elements. Each element is taken to be a fixed-point value in [0,1] with $m$ bits, where $m$ is the number of bits in the corresponding color component of the selected buffer (see section 2.12.9).

### Conversion of Depth values

This step applies only if *format* is DEPTH_COMPONENT. An element is taken to be a fixed-point value in [0,1] with $m$ bits, where $m$ is the number of bits in the depth buffer (see section 2.9.1).

### Arithmetic on components

This step applies only to component groups. An R element is multiplied by RED_SCALE, a G element by GREEN_SCALE, a B element by BLUE_SCALE, and an A element by ALPHA_SCALE; a depth component is multiplied by DEPTH_SCALE. Next, RED_BIAS, GREEN_BIAS, BLUE_BIAS, ALPHA_BIAS, or DEPTH_BIAS is added to each resulting element, as appropriate.

### Arithmetic on Indices

This step applies only to indices. After the index is obtained from the selected buffer, the corresponding step for **DrawPixels** is applied to the integer index (there are no bits to the right of the binary point in this case).

### RGBA to RGBA Lookup

This step applies only to RGBA component groups. It is identical to the corresponding step for **DrawPixels**.

### Index Lookup

This step applies only to indices. If *format* is one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, LUMINANCE, or LUMINANCE_ALPHA, then the index is used to reference 4 tables of color components: PIXEL_MAP_I_TO_R, PIXEL_MAP_I_TO_G, PIXEL_MAP_I_TO_B, and PIXEL_MAP_I_TO_A. Each of these tables must have $2^n$ entries for some integer value of $n$ ($n$ may be different for each table). For each table, the index is first rounded to the nearest integer; this value is ANDed with $2^n - 1$, and the resulting value used as an address into the table. The indexed value becomes an R, G, B, or A value, as appropriate. The group of four elements so obtained replaces the index, changing the group's type to "component."

If the format is COLOR_INDEX and if MAP_COLOR is TRUE, then the index is looked up in the PIXEL_MAP_I_TO_I table (otherwise, the index is not looked up). Again, the table must have $2^n$ entries for some $n$, and the integer part of the index is ANDed with $2^n - 1$, producing a value. This value addresses the table, and the value in the table replaces the index. The floating-point table value is first rounded to a fixed-point value with unspecified precision.

Finally, if *format* is `STENCIL_INDEX` and if `MAP_STENCIL` is `TRUE`, then the index is looked up as described in the preceding paragraph, but using the `PIXEL_MAP_S_TO_S` table.

### Conversion to L

This step applies only to RGBA component groups, and only if the *format* is either `LUMINANCE` or `LUMINANCE_ALPHA`. A value L is computed as

$$L = R + G + B$$

where $R$, $G$, and $B$ are the values of the R, G, and B components. The single computed L component replaces the R, G, and B components in the group.

### Final Conversion

For an index, if the *type* is not `FLOAT`, final conversion consists of masking the index with the value given Table 4.6; if the *type* is `FLOAT`, then the integer index is converted to single-precision floating-point. For a component, each component is first clamped to $[0, 1]$. Then, the appropriate conversion formula from Table 4.6 is applied to the component.

### Placement in Client Memory

Groups of elements are placed in memory just as they are taken from memory for **DrawPixels**. That is, the $i$th group of the $j$th row (corresponding to the $i$th pixel in the $j$th row) is placed in memory just where the $i$th group of the $j$th row would be taken from for **DrawPixels**. See **Unpacking** under section 3.6.3. The only difference is that the storage mode parameters whose names begin with `PACK_` are used instead of those whose names begin with `UNPACK_`. If the *format* is `RED`, `GREEN`, `BLUE`, `ALPHA`, or `LUMINANCE`, only the corresponding single element is written. Otherwise the number of elements to be written is given by Table 3.5.

In correspondence with **DrawPixels**, if `PACK_SWAP_BYTES` is `TRUE`, there is no effect on a one-byte element, but bytes constituting a two-byte or four-byte element are reversed (so that they are in an order opposite to the

| *type* | Index Mask | Component Conversion |
|---|---|---|
| UNSIGNED_BYTE | $2^8 - 1$ | $(2^8 - 1)c$ |
| BYTE | $2^7 - 1$ | $[(2^8 - 1)c - 1]/2$ |
| BITMAP | $1$ | $1$ |
| UNSIGNED_SHORT | $2^{16} - 1$ | $2^{16} - 1$ |
| SHORT | $2^{15} - 1$ | $[(2^{16} - 1)c - 1]/2$ |
| UNSIGNED_INT | $2^{32} - 1$ | $2^{32} - 1$ |
| INT | $2^{31} - 1$ | $[(2^{32} - 1)c - 1]/2$ |
| FLOAT | *none* | $c$ |

Table 4.6: **ReadPixels** index masks and component conversion formulas. $c$ represents a component value to be converted.

client's native byte ordering for the indicated type) immediately prior to being placed in client memory. If `PACK_SWAP_BYTES` is `FALSE`, then no swapping occurs. If *type* is `BITMAP`, then each byte of client memory is eight bits, each of which is a single element. The single-bit elements within each byte are ordered from most significant to least significant if the value of `PACK_LSB_FIRST` is `FALSE`; otherwise, the ordering is from least significant to most significant. The `BITMAP` *type* is valid only if *format* is either `COLOR_INDEX` or `STENCIL_INDEX` (otherwise, results are undefined).

### 4.3.3 Copying Pixels

**CopyPixels** transfers a rectangle of pixel values from one region of the framebuffer to another. Pixel copying is diagrammed in Figure 4.3.

void **CopyPixels**( int $x$, int $y$, sizei *width*, sizei *height*,
  enum *type* ) ;

*type* is a symbolic constant that must be one of `COLOR`, `STENCIL`, or `DEPTH`, indicating that the values to be transferred are colors, stencil values, or depth values, respectively. The first four arguments have the same interpretation as the corresponding arguments to **ReadPixels**.

Values are obtained from the framebuffer, converted (if appropriate), subjected to arithmetic operations, and looked up in tables just as if **ReadPixels** were called with the corresponding arguments. If the *type* is `STENCIL` or `DEPTH`, then it is as if the *format* for **ReadPixels** were `STENCIL_INDEX`

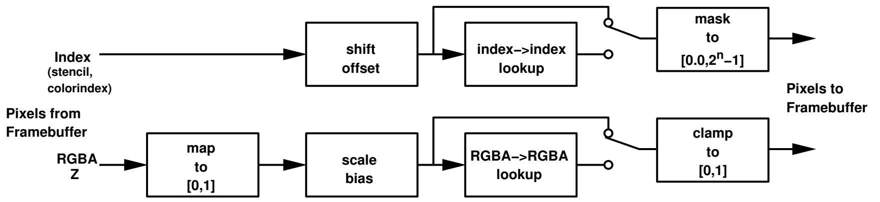


Figure 4.3. Operation of **CopyPixels**. All parameters affecting pixel copying are set with **PixelTransfer** or **PixelMap**.

or DEPTH_COMPONENT, respectively. If the *type* is COLOR, then if the GL is in RGBA mode, it is as if the *format* were RGBA, while if the GL is in color index mode, it is as if the *format* were COLOR_INDEX.

The groups of elements so obtained are then written to the framebuffer just as if **DrawPixels** had been given *width* and *height*, beginning with final conversion of elements. The effective *format* is the same as that already described.

### 4.3.4 Pixel draw/read state

The state required for pixel operations consists of the parameters that are set with **PixelStore**, **PixelTransfer**, and **PixelMap**. State set with **PixelStore** is GL client state. This state has been summarized in Tables 3.1, 3.2, and 3.3. The current setting of **ReadBuffer**, a twelve-valued integer, is also required, along with the current raster position (section 2.11). **PixelStore** state is client state.

# Chapter 5

# Special Functions

This chapter describes additional GL functionality that does not fit easily into any of the preceding chapters. This functionality consists of evaluators (used to model curves and surfaces), selection (used to locate rendered primitives on the screen), feedback (which returns GL results before rasterization), display lists (used to designate a group of GL commands for later execution by the GL), flushing and finishing (used to synchronize the GL command stream), and hints.

## 5.1 Evaluators

Evaluators provide a means to use a polynomial or rational polynomial mapping to produce vertex, normal, and texture coordinates, and colors. The values so produced are sent on to further stages of the GL as if they had been provided directly by the client. Transformations, lighting, primitive assembly, rasterization, and per-pixel operations are not affected by the use of evaluators.

Consider the $R^k$-valued polynomial $p(u)$ defined by

$$p(u) = \sum_{i=0}^{n} B_i^n(u) R_i \tag{5.1}$$

with $R_i \in R^k$ and

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i},$$

the $i$th Bernstein polynomial of degree $n$ (recall that $0^0 \equiv 1$ and $\binom{n}{0} \equiv 1$). Each $R_i$ is a *control point*. The relevant command is

| *target* | $k$ | Values |
|---|---|---|
| MAP1_VERTEX_3 | 3 | $x$, $y$, $z$ vertex coordinates |
| MAP1_VERTEX_4 | 4 | $x$, $y$, $z$, $w$ vertex coordinates |
| MAP1_INDEX | 1 | color index |
| MAP1_COLOR_4 | 4 | R, G, B, A |
| MAP1_NORMAL | 3 | $x$, $y$, $z$ normal coordinates |
| MAP1_TEXTURE_COORD_1 | 1 | $s$ texture coordinate |
| MAP1_TEXTURE_COORD_2 | 2 | $s$, $t$ texture coordinates |
| MAP1_TEXTURE_COORD_3 | 3 | $s$, $t$, $r$ texture coordinates |
| MAP1_TEXTURE_COORD_4 | 4 | $s$, $t$, $r$, $q$ texture coordinates |

Table 5.1: Values specified by the *target* to **Map1**. Values are given in the order in which they are taken.

void **Map1{fd}**( enum *type*, T $u_1$, T $u_2$, int *stride*, int *order*,
T *points* ) ;

*type* is a symbolic constant indicating the range of the defined polynomial. Its possible values, along with the evaluations that each indicates, are given in Table 5.1. *order* is equal to $n+1$; The error INVALID_VALUE results if *order* is less than one or greater than MAX_EVAL_ORDER. *points* is a pointer to a set of $n+1$ blocks of storage. Each block begins with $k$ single-precision floating-point or double-precision floating-point values, respectively. The rest of the block may be filled with arbitrary data. Table 5.1 indicates how $k$ depends on *type* and what the $k$ values represent in each case.

*stride* is the number of single- or double-precision values (as appropriate) in each block of storage. The error INVALID_VALUE results if *stride* is less than $k$. The order of the polynomial, *order*, is also the number of blocks of storage containing control points.

$u_1$ and $u_2$ give two floating-point values that define the endpoints of the pre-image of the map. When a value $u'$ is presented for evaluation, the formula used is

$$p'(u') = p(\frac{u' - u_1}{u_2 - u_1}).$$

The error INVALID_VALUE results if $u_1 = u_2$.

**Map2** is analogous to **Map1**, except that it describes bivariate polynomials of the form

$$p(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_i^n(u) B_j^m(v) R_{ij}.$$

The form of the **Map2** command is

void **Map2{fd}**( enum *target*, T $u_1$, T $u_2$, int *ustride*, int *uorder*, T $v_1$, T $v_2$, int *vstride*, int *vorder*, T *points* ) ;

*target* is a range type selected from the same group as is used for **Map1**, except that the string MAP1 is replaced with MAP2. *points* is a pointer to $(n+1)(m+1)$ blocks of storage ($uorder = n+1$ and $vorder = m+1$; the error INVALID_VALUE results if either *uorderorvorder* is less than one or greater than MAX_EVAL_ORDER). The values comprising $R_{ij}$ are located

$$(ustride)i + (vstride)j$$

values (either single- or double-precision floating-point, as appropriate) past the first value pointed to by *points*. $u_1$, $u_2$, $v_1$, and $v_2$ define the pre-image rectangle of the map; a domain point $(u', v')$ is evaluated as

$$p'(u', v') = p(\frac{u' - u_1}{u_2 - u_1}, \frac{v' - v_1}{v_2 - v_1}).$$

The evaluation of a defined map is enabled or disabled with **Enable** and **Disable** using the constant corresponding to the map as described above. The error INVALID_VALUE results if either *ustride* or *vstride* is less than k, or if $u_1$ is equal to $u_2$, or if $v_1$ is equal to $v_2$.

Figure 5.1 describes map evaluation schematically; an evaluation of enabled maps is effected in one of two ways. The first way is to use

void **EvalCoord{12}{fd}**( T *arg* ) ;
void **EvalCoord{12}{fd}v**( T *arg* ) ;

**EvalCoord1** causes evaluation of the enabled 1-dimensional maps. The argument is the value (or a pointer to the value) that is the domain coordinate, $u'$. **EvalCoord2** causes evaluation of the enabled 2-dimensional maps. The two values specify the two domain coordinates, $u'$ and $v'$, in that order.

When one of the **EvalCoord** commands is issued, all currently enabled maps of the indicated dimension are evaluated. Then, for each enabled map, it is as if a corresponding GL command were issued with the resulting coordinates, with one important difference. The difference is that when an

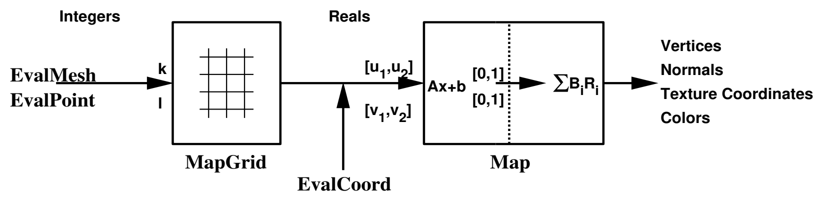


Figure 5.1. Map Evaluation.

evaluation is performed, the GL uses evaluated values instead of current values for those evaluations that are enabled (otherwise, the current values are used). The order of the effective commands is immaterial, except that **Vertex** (for vertex coordinate evaluation) must be issued last. Use of evaluators has no effect on the current color, normal, or texture coordinates.

No command is effectively issued if the corresponding map (of the indicated dimension) is not enabled. If more than one evaluation is enabled for a particular dimension (e.g. MAP1_TEXTURE_COORD_1 and MAP1_TEXTURE_COORD_2), then only the result of the evaluation of the map with the highest number of coordinates is used.

Finally, if either MAP2_VERTEX_3 or MAP2_VERTEX_4 is enabled, then the normal to the surface is computed analytically. If automatic normal generation is enabled, then this computed normal is used as the normal associated with a generated vertex. Automatic normal generation is controlled with **Enable** and **Disable** with symbolic the constant AUTO_NORMAL. If automatic normal generation is disabled, then a corresponding normal map, if enabled, is used to produce a normal. If neither automatic normal generation nor a normal map are enabled, then no normal is sent with a vertex resulting from an evaluation (the effect is that the current normal is used).

For MAP_VERTEX_3, let $q = p$. For MAP_VERTEX_4, let $q = (x/w, y/w, z/w)$, where $(x, y, z, w) = p$. Then let

$$m = \frac{\partial q}{\partial u} \times \frac{\partial q}{\partial v}.$$

Then the generated normal, $n$, is given by $n = m/\|m\|$.

The second way to carry out evaluations is to use a set of commands that provide for efficient specification of a series of evenly spaced values to be mapped. This method proceeds in two steps. The first step is to define a grid in the domain. This is done using

void **MapGrid1**{**fd**}( int $n$, T $u'_1$, T $u'_2$ ) ;

for a 1-dimensional map or

void **MapGrid2**{**fd**}( int $n_u$, T $u'_1$, T $u'_2$, int $n_v$, T $v'_1$, T $v'_2$ ) ;

for a 2-dimensional map. In the case of **MapGrid1** $u'_1$ and $u'_2$ describe an interval, while $n$ describes the number of partitions of the interval. The error `INVALID_VALUE` results if $n \leq 0$. For **MapGrid2**, $(u'_1, v'_1)$ specifies one two-dimensional point and $(u'_2, v'_2)$ specifies another. $n_u$ gives the number of partitions between $u'_1$ and $u'_2$, and $n_v$ gives the number of partitions between $v'_1$ and $v'_2$. If either $n_u \leq 0$ or $n_v \leq 0$, then the error `INVALID_VALUE` occurs.

Once a grid is defined, an evaluation on a rectangular subset of that grid may be carried out by calling

void **EvalMesh1**( enum *mode*, int $p_1$, int $p_2$ ) ;

*mode* is either `POINT` or `LINE`. The effect is the same as performing the following code fragment, with $\Delta u' = (u'_2 - u'_1)/n$:

```
Begin(type);
   for i = p1 to p2 step 1.0
      EvalCoord1(i * Δu' + u'1);
End();
```

where **EvalCoord1f** or **EvalCoord1d** is substituted for **EvalCoord1** as appropriate. If *mode* is `POINT`, then *type* is `POINTS`; if *mode* is `LINE`, then *type* is `LINE_STRIP`. The one requirement is that if either $i = 0$ or $i = n$, then the value computed from $i * \Delta u' + u'_1$ is precisely $u'_1$ or $u'_2$, respectively.

The corresponding commands for two-dimensional maps are

void **EvalMesh2**( enum *mode*, int $p_1$, int $p_2$, int $q_1$, int $q_2$ ) ;

*mode* must be `FILL`, `LINE`, or `POINT`. When *mode* is `FILL`, then these commands are equivalent to the following, with $\Delta u' = (u'_2 - u'_1)/n$ and $\Delta v' = (v'_2 - v'_1)/m$:

```
for i = q1 to q2 - 1 step 1.0
   Begin(QUAD_STRIP);
      for j = p1 to p2 step 1.0
         EvalCoord2(j * Δu' + u'1 , i * Δv' + v'1);
         EvalCoord2(j * Δu' + u'1 , (i + 1) * Δv' + v'1);
   End();
```

If *mode* is LINE, then a call to **EvalMesh2** is equivalent to

```
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);
   End();;
for i = p1 to p2 step 1.0
   Begin(LINE_STRIP);
   for j = q1 to q2 step 1.0
      EvalCoord2(i * Δu' + u'1 , j * Δv' + v'1);
   End();
```

If *mode* is POINT, then a call to **EvalMesh2** is equivalent to

```
Begin(POINTS);
   for i = q1 to q2 step 1.0
      for j = p1 to p2 step 1.0
         EvalCoord2(j * Δu' + u'1 , i * Δv' + v'1);
End();
```

Again, in all three cases, there is the requirement that $0 * \Delta u' + u'_1 = u'_1$, $n * \Delta u' + u'_1 = u'_2$, $0 * \Delta v' + v'_1 = v'_1$, and $m * \Delta v' + v'_1 = v'_2$.

An evaluation of a single point on the grid may also be carried out:

`void` **EvalPoint1**( `int` *p* ) ;

Calling it is equivalent to the command

```
EvalCoord1(p * Δu' + u'1);
```

with $\Delta u'$ and $u'_1$ defined as above.

`void` **EvalPoint2**( `int` *p*, `int` *q* ) ;

is equivalent to the command

$$\textbf{EvalCoord2}(p * \Delta u' + u'_1 \ , \ q * \Delta v' + v'_1);$$

The state required for evaluators potentially consists of 9 1-dimensional map specifications and 9 2-dimensional map specifications, as well as corresponding flags for each specification indicating which are enabled. Each map specification consists of one or two orders, an appropriately sized array of control points, and a set of two values (for a 1-dimensional map) or four values (for a 2-dimensional map) to describe the domain. The maximum possible order, for either $u$ or $v$, is implementation dependent (one maximum applies to both $u$ and $v$), but must be at least 8. Each control point consists of between one and four floating-point values (depending on the type of the map). Initially, all maps have order 1 (making them constant maps). All vertex coordinate maps produce the coordinates $(0,0,0,1)$ (or the appropriate subset); all normal coordinate maps produce $(0,0,1)$; RGBA maps produce (1,1,1,1); color index maps produce 1.0. In the initial state, all maps are disabled. A flag indicates whether or not automatic normal generation is enabled for 2-dimensional maps. In the initial state, automatic normal generation is disabled. Also required are two floating-point values and an integer number of grid divisions for the 1-dimensional grid specification and four floating-point values and two integer grid divisions for the 2-dimensional grid specification. In the initial state, the bounds of the domain interval for 1-D is 0 and 1.0, respectively; for 2-D, they are $(0,0)$ and $(1.0,1.0)$, respectively. The number of grid divisions is 1 for 1-D and 1 in both directions for 2-D. If any evaluation command is issued when no vertex map is enabled, nothing happens.

## 5.2 Selection

Selection is used by a programmer to determine which primitives are drawn into some region of a window. The region is defined by the current model-view and perspective matrices.

Selection works by returning an array of integer-valued *names*. This array represents the current contents of the *name stack*. This stack is controlled with the commands

void **InitNames**( void ) ;
void **PopName**( void ) ;
void **PushName**( uint *name* ) ;

void **LoadName**( uint *name* ) ;

**InitNames** empties (clears) the name stack. **PopName** pops one name off the top of the name stack. **PushName** causes *name* to be pushed onto the name stack. **LoadName** replaces the value on the top of the stack with *name*. Loading a name onto an empty stack generates the error INVALID_OPERATION. Popping a name off of an empty stack generates STACK_UNDERFLOW; pushing a name onto a full stack generates STACK_OVERFLOW. The maximum allowable depth of the name stack is implementation dependent but must be at least 64.

In selection mode, no pixels are produced from rasterization. The GL is placed in selection mode with

int **RenderMode**( enum *mode* ) ;

*mode* is a symbolic constant: one of RENDER, SELECT, or FEEDBACK. RENDER is the default, corresponding to rendering as described until now. SELECT specifies selection mode, and FEEDBACK specifies feedback mode (described below). Use of any of the name stack manipulation commands while the GL is not in selection mode has no effect.

Selection is controlled using

void **SelectBuffer**( sizei *n*, uint **buffer* ) ;

*buffer* is a pointer to an array of unsigned integers (called the selection array) to be potentially filled with names, and $n$ is an integer indicating the maximum number of values that can be stored in that array. Placing the GL in selection mode before **SelectBuffer** has been called results in an error of INVALID_OPERATION as does calling **SelectBuffer** while in selection mode.

In selection mode, if a point, line, polygon, or the valid coordinates produced by a **RasterPos** command intersects the clip volume (section 2.10) then this primitive (or **RasterPos** command) causes a selection *hit*. In the case of polygons, no hit occurs if the polygon would have been culled, but selection is based on the polygon itself, regardless of the setting of **PolygonMode**. When in selection mode, whenever a name stack manipulation command is executed or **RenderMode** is called and there has been a hit since the last time the stack was manipulated or **RenderMode** was called, then a *hit record* is written into the selection array.

A hit record consists of the following items in order: a non-negative integer giving the number of elements on the name stack at the time of the hit, a minimum depth value, a maximum depth value, and the name stack with the bottommost element first. The minimum and maximum depth values are the minimum and maximum taken over all the window coordinate $z$ values of each (post-clipping) vertex of each primitive that intersects the clipping volume since the last hit record was written. The minimum and maximum (each of which lies in the range $[0, 1]$) are each multiplied by $2^{32}-1$ and rounded to the nearest unsigned integer to obtain the values that are placed in the hit record.

Hit records are placed in the selection array by maintaining a pointer into that array. When selection mode is entered, the pointer is initialized to the beginning of the array. Each time a hit record is copied, the pointer is updated to point at the array element after the one into which the topmost element of the name stack was stored. If copying the hit record into the selection array would cause the total number of values to exceed $n$, then as much of the record as fits in the array is written and an overflow flag is set.

Selection mode is exited by calling **RenderMode** with an argument value other than `SELECT`. Whenever **RenderMode** is called in selection mode, it returns the number of hit records copied into the selection array and resets the **SelectBuffer** pointer to its last specified value. Values are not guaranteed to be written into the selection array until **RenderMode** is called. If the selection array overflow flag was set, then **RenderMode** returns $-1$ and clears the overflow flag. The name stack is cleared and the stack pointer reset whenever **RenderMode** is called.

The state required for selection consists of the address of the selection array and its maximum size, the name stack and its associated pointer, a minimum and maximum depth value, and several flags. One flag indicates the current **RenderMode** value. In the initial state, the GL is in the `RENDER` mode. Another flag is used to indicate whether or not a hit has occurred since the last name stack manipulation. This flag is reset upon entering selection mode and whenever a name stack manipulation takes place. One final flag is required to indicate whether the maximum number of copied names would have been exceeded. This flag is reset upon entering selection mode. This flag, the address of the selection array, and its maximum size are GL client state.

## 5.3 Feedback

Feedback, like selection, is a GL mode. The mode is selected by calling **RenderMode** with `FEEDBACK`. When the GL is in feedback mode, no pixels are produced by rasterization. Instead, information about primitives that would have been rasterized is fed back to the application using the GL.

Feedback is controlled using

void **FeedbackBuffer**( sizei *n*, enum *type*, float **buffer* ) ;

*buffer* is a pointer to an array of floating-point values into which feedback information will be placed, and *n* is a number indicating the maximum number of values that can be written to that array. *type* is a symbolic constant describing the information to be fed back for each vertex (see Figure 5.2). The error INVALID_OPERATION results if the GL is placed in feedback mode before a call to **FeedbackBuffer** has been made, or if a call to **FeedbackBuffer** is made while in feedback mode.

While in feedback mode, each primitive that would be rasterized (or bitmap or call to **DrawPixels** or **CopyPixels**, if the raster position is valid) generates a block of values that get copied into the feedback array. If doing so would cause the number of entries to exceed the maximum, the block is partially written so as to fill the array (if there is any room left at all). The first block of values generated after the GL enters feedback mode is placed at the beginning of the feedback array, with subsequent blocks following. Each block begins with a code indicating the primitive type, followed by values that describe the primitive's vertices and associated data. Entries are also written for bitmaps and pixel rectangles. Feedback occurs after polygon culling (section 3.5.1) and **PolygonMode** interpretation of polygons (section 3.5.4) has taken place. It may also occur after polygons with more than three edges are broken up into triangles (if the GL implementation renders polygons by performing this decomposition). $x$, $y$, and $z$ coordinates returned by feedback are window coordinates; if $w$ is returned, it is in clip coordinates. In the case of bitmaps and pixel rectangles, the coordinates returned are those of the current raster position.

The ordering rules for GL command interpretation also apply in feedback mode. Each command must be fully interpreted and its effects on both GL state and the values to be written to the feedback buffer completed before a subsequent command may be executed.

The GL is taken out of feedback mode by calling **RenderMode** with an argument value other than FEEDBACK. When called while in feedback mode, **RenderMode** returns the number of values placed in the feedback array and resets the feedback array pointer to be *buffer*. The return value never exceeds the maximum number of values passed to **FeedbackBuffer**.

| Type | coordinates | color | texture | total values |
|---|---|---|---|---|
| `2D` | $x$, $y$ | – | – | 2 |
| `3D` | $x$, $y$, $z$ | – | – | 3 |
| `3D_COLOR` | $x$, $y$, $z$ | $k$ | – | $3+k$ |
| `3D_COLOR_TEXTURE` | $x$, $y$, $z$ | $k$ | 4 | $7+k$ |
| `4D_COLOR_TEXTURE` | $x$, $y$, $z$, $w$ | $k$ | 4 | $8+k$ |

Table 5.2: Correspondence of feedback type to number of values per vertex. $k$ is 1 in color index mode and 4 in RGBA mode.

If writing a value to the feedback buffer would cause more values to be written than the specified maximum number of values, then the value is not written and an overflow flag is set. In this case, **RenderMode** returns $-1$ when it is called, after which the overflow flag is reset. While in feedback mode, values are not guaranteed to be written into the feedback buffer before **RenderMode** is called.

Figure 5.2 gives a grammar for the array produced by feedback. Each primitive is indicated with a unique identifying value followed by some number of vertices. A vertex is fed back as some number of floating-point values determined by the feedback *type*. Table 5.2 gives the correspondence between feedback *buffer* and the number of values returned for each vertex.

The command

`void` **PassThrough**( `float` *token* ) ;

may be used as a marker in feedback mode. *token* is returned as if it were a primitive; it is indicated with its own unique identifying value. The ordering of any **PassThrough** commands with respect to primitive specification is maintained by feedback. **PassThrough** may not occur between **Begin** and **End**. It has no effect when the GL is not in feedback mode.

The state required for feedback is the pointer to the feedback array, the maximum number of values that may be placed there, and the feedback *type*. An overflow flag is required to indicate whether the maximum allowable number of feedback values has been written; initially this flag is cleared. These state variables are GL client state. Feedback also relies on the same mode flag as selection to indicate whether the GL is in feedback, selection, or normal rendering mode.

feedback-list:
    feedback-item feedback-list
    feedback-item

feedback-item:
    point
    line-segment
    polygon
    bitmap
    pixel-rectangle
    passthrough

point:
    `POINT_TOKEN` vertex

line-segment:
    `LINE_TOKEN` vertex vertex
    `LINE_RESET_TOKEN` vertex vertex

polygon:
    `POLYGON_TOKEN` $n$ polygon-spec

polygon-spec:
    polygon-spec vertex
    vertex vertex vertex

bitmap:
    `BITMAP_TOKEN` vertex

pixel-rectangle:
    `DRAW_PIXEL_TOKEN` vertex
    `COPY_PIXEL_TOKEN` vertex

passthrough:
    `PASS_THROUGH_TOKEN` $f$

vertex:

`2D`:
    $f$ $f$

`3D`:
    $f$ $f$ $f$

`3D_COLOR`:
    $f$ $f$ $f$ color

`3D_COLOR_TEXTURE`:
    $f$ $f$ $f$ color tex

`4D_COLOR_TEXTURE`:
    $f$ $f$ $f$ $f$ color tex

color:
    $f$ $f$ $f$ $f$
    $f$

tex:
    $f$ $f$ $f$ $f$

Figure 5.2: Feedback syntax. $f$ is a floating-point number. $n$ is a floating-point integer giving the number of vertices in a polygon. The symbols ending with `_TOKEN` are symbolic floating-point constants. The labels under the "vertex" rule show the different data returned for vertices depending on the feedback *type*. `LINE_TOKEN` and `LINE_RESET_TOKEN` are identical except that the latter is returned only when the line stipple is reset for that line segment.

## 5.4 Display Lists

A display list is simply a group of GL commands and arguments that has been stored for subsequent execution. The GL may be instructed to process a particular display list (possibly repeatedly) by providing a number that uniquely specifies it. Doing so causes the commands within the list to be executed just as if they were given normally. The only exception pertains to commands that rely upon client state. When such a command is accumulated into the display list (that is, when issued, not when executed), the client state in effect at that time applies to the command. Only server state is affected when the command is executed.

A display list is begun by calling

`void` **NewList**( `uint` *n*, `enum` *mode* ) ;

*n* is a positive integer to which the display list that follows is assigned, and *mode* is a symbolic constant that controls the behavior of the GL during display list creation. If *mode* is `COMPILE`, then commands are not executed as they are placed in the display list. If *mode* is `COMPILE_AND_EXECUTE`. then commands are executed as they are encountered, then placed in the display list. If $n = 0$, then the error `INVALID_VALUE` is generated.

After calling **NewList** all subsequent GL commands are placed in the display list (in the order the commands are issued) until a call to

`void` **EndList**( `void` ) ;

occurs, after which the GL returns to its normal command execution state. It is only when **EndList** occurs that the specified display list is actually associated with the index indicated with **NewList**. The error `INVALID_OPERATION` is generated if **EndList** is called without a previous matching **NewList**, or if **NewList** is called a second time before calling **EndList**.

Once defined, a display list is executed by calling

`void` **CallList**( `uint` *n* ) ;

*n* gives the index of the display list to be called. This causes the commands saved in the display list to be executed, in order, just as if they were issued without using a display list. If $n = 0$, then the error `INVALID_VALUE` is generated.

The command

`void` **CallLists**( `sizei` *n*, `enum` *type*, `void` **lists* ) ;

provides an efficient means for executing a number of display lists. $n$ is an integer indicating the number of display lists to be called, and *lists* is a pointer that points to an array of offsets. Each offset is constructed as determined by *lists* as follows. First, *type* may be one of the constants `BYTE`, `UNSIGNED_BYTE`, `SHORT`, `UNSIGNED_SHORT`, `INT`, `UNSIGNED_INT`, or `FLOAT` indicating that the array pointed to by *lists* is an array of bytes, unsigned bytes, shorts, unsigned shorts, integers, unsigned integers, or floats, respectively. In this case each offset is found by simply converting each array element to an integer (floating point values are truncated). Further, *type* may be one of `2_BYTES`, `3_BYTES`, or `4_BYTES`, indicating that the array contains sequences of 2, 3, or 4 unsigned bytes, in which case each integer offset is constructed according to the following algorithm:

$offset \leftarrow 0$
**for** $i = 1$ **to** $b$
    $offset \leftarrow offset$ shifted left 8 bits
    $offset \leftarrow offset + byte$
    advance to next *byte* in the array

$b$ is 2, 3, or 4, as indicated by *type*. If $n = 0$, **CallLists** does nothing.

Each of the $n$ constructed offsets is taken in order and added to a display list base to obtain a display list number. For each number, the indicated display list is executed. The base is set by calling

void **ListBase**( uint *base* ) ;

to specify the offset.

Indicating a display list index that does not correspond to any display list has no effect. **CallList** or **CallLists** may appear inside a display list. (If the *mode* supplied to **NewList** is `COMPILE_AND_EXECUTE`, then the appropriate lists are executed, but the **CallList** or **CallLists**, rather than those lists' constituent commands, is placed in the list under construction.) To avoid the possibility of infinite recursion resulting from display lists calling one another, an implementation dependent limit is placed on the nesting level of display lists during display list execution. This limit must be at least 64.

Two commands are provided to manage display list indices.

uint **GenLists**( sizei $s$ ) ;

returns an integer $n$ such that the indices $n, \ldots, n + s - 1$ are previously unused (i.e. there are $s$ previously unused display list indices starting at $n$).

**GenLists** also has the effect of creating an empty display list for each of the indices $n, \ldots, n+s-1$, so that these indices all become used. **GenLists** returns 0 if there is no group of $s$ contiguous previously unused display list indices, or if $s = 0$.

`boolean` **IsList**( `uint` *list* ) ;

returns `TRUE` if *list* is the index of some display list.

A contiguous group of display lists may be deleted by calling

`void` **DeleteLists**( `uint` *list*, `sizei` *range* ) ;

where *list* is the index of the first display list to be deleted and *range* is the number of display lists to be deleted. All information about the display lists is lost, and the indices become unused. Indices to which no display list corresponds are simply ignored. If $range = 0$, nothing happens.

Certain commands, when made within a display list, are not compiled into the display list but are executed immediately. These are: **IsList**, **GenLists**, **DeleteLists**, **FeedbackBuffer**, **SelectBuffer**, **RenderMode**, **ReadPixels**, **PixelStore**, **Flush**, **Finish**, as well as **IsEnabled** and all of the **Get** commands (see Chapter 6).

Display lists require one bit of state to indicate whether a GL command should be executed immediately or placed in a display list. In the initial state, commands are executed immediately. If the bit indicates display list creation, an index is required to indicate the current display list being defined. Another bit indicates, during display list creation, whether or not commands should be executed as they are compiled into the display list. One integer is required for the current **ListBase** setting; its initial value is zero. Finally, state must be maintained to indicate which integers are currently in use as display list indices. In the initial state, no indices are in use.

## 5.5 Flush and Finish

The command

`void` **Flush**( `void` ) ;

indicates that all commands that have previously been sent to the GL must complete in finite time.

The command

```
void Finish( void );
```

forces all previous GL commands to complete. **Finish** does not return until all effects from previously issued commands on GL client and server state and the framebuffer are fully realized.

## 5.6 Hints

Certain aspects of GL behavior, when there is room for variation, may be controlled with hints. A hint is specified using

```
void Hint( enum target, enum hint );
```

*target* is a symbolic constant indicating the behavior to be controlled, and *hint* is a symbolic constant indicating what type of behavior is desired. *target* may be one of PERSPECTIVE_CORRECTION_HINT, indicating the desired quality of parameter interpolation; POINT_SMOOTH_HINT, indicating the desired sampling quality of points; LINE_SMOOTH_HINT, indicating the desired sampling quality of lines; POLYGON_SMOOTH_HINT, indicating the desired sampling quality of polygons; and FOG_HINT, indicating whether fog calculations are done per pixel or per vertex. *hint* must be one of FASTEST, indicating that the most efficient option should be chosen; NICEST, indicating that the highest quality option should be chosen; and DONT_CARE, indicating no preference in the matter.

The interpretation of hints is implementation dependent. An implementation may ignore them entirely.

# Chapter 6

# State and State Requests

The values of most GL state variables can be obtained using a set of **Get** commands. There are four commands for obtaining simple state variables:

void **GetBooleanv**( enum *value*, boolean **data* ) ;
void **GetIntegerv**( enum *value*, int **data* ) ;
void **GetFloatv**( enum *value*, float **data* ) ;
void **GetDoublev**( enum *value*, double **data* ) ;

The commands obtain boolean, integer, floating-point, or double-precision state variables. *value* is a symbolic constant indicating the state variable to return, *data* is a pointer to an array of the indicated type in which to place the returned data. In addition

boolean **IsEnabled**( enum *value* ) ;

can be used to determine if *value* is currently enabled (as with **Enable**) or disabled.

If a **Get** command is issued that returns value types different from the type of the value being obtained, a type conversion is performed. If **GetBooleanv** is called, a floating-point or integer value converts to FALSE if and only if it is zero (otherwise it converts to TRUE). If **GetIntegerv** (or any of the **Get** commands below) is called, a boolean value is interpreted as either 1 or 0, and a floating-point value is rounded to the nearest integer, unless the value is a an RGBA color component or a normal coordinate. In these cases, the **Get** command converts the floating-point value to an integer according the INT entry of Table 4.6; a value not in $[-1, 1]$ converts to an undefined value. If **GetFloatv** is called, a boolean value is interpreted as either 1.0 or 0.0, an integer is coerced to floating-point, and a double-precision floating-point value is converted to single-precision. Analogous conversions are carried out in the case of **GetDoublev**. If a value is so large in magnitude that it cannot be represented with the requested type, then the nearest value representable using the requested type is returned.

Other commands exist to obtain state variables that are indexed by a target. These are

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{if}v**( enum *coord*, enum *value*, T *data* ) ;
void **GetTexParameter{if}v**( enum *target*, enum *value*, T *data* ) ;
void **GetTexLevelParameter{if}v**( 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* ) ;

**GetClipPlane** always returns four double-precision values in *eqn*; these are the coefficients of the plane equation of *plane* in eye coordinates (these coordinates are those that were computed when the plane was specified).

**GetLight** places information about *value* (a symbolic constant) for *light* (also a symbolic constant) in *data*. POSITION or SPOT_DIRECTION returns values in eye coordinates (again, these are the coordinates that were computed when the position or direction was specified).

**GetMaterial**, **GetTexGen**, **GetTexEnv**, and **GetTexParameter** are similar to **GetLight**, placing information about *value* for the target indicated by their first argument into *data*. The *face* argument to **GetMaterial** must be either FRONT or BACK, indicating the front or back material, respectively. The *env* argument to **GetTexEnv** must currently be TEXTURE_ENV. The *coord* argument to **GetTexGen** must be one of S, T, R, or Q. For **GetTexGen**, EYE_LINEAR coefficients are returned in the eye coordinates that were computed when the plane was specified; OBJECT_LINEAR coefficients are returned in object coordinates.

For **GetTexParameter** and **GetTexLevelParameter**, *target* must currently be either TEXTURE_1D or TEXTURE_2D, indicating the target from which information is to be obtained. *value* is a symbolic value indicating which texture parameter is to be obtained. The *lod* argument to **GetTexLevelParameter** determines which level-of-detail's state is returned.

If the *lod* argument is less than zero or if it is larger than the maximum allowable level-of-detail then the error INVALID_VALUE occurs.

For **GetPixelMap**, the *map* must be a map name from Table 3.3. For **GetMap**, *map* must be one of the map types described in section 5.1, and *value* must be one of ORDER, COEFF, or DOMAIN.

**GetTexImage** is used to obtain texture images.

```
int GetTexImage( enum tex, int lod, enum format,
    enum type, void *img ) ;
```

It is somewhat different from the other get commands; *tex* is a symbolic value indicating which texture is to be obtained. TEXTURE_1D indicates a one-dimensional texture, while TEXTURE_2D indicates a two-dimensional texture. *lod* is a level-of-detail number, *format* is a pixel format from Table 3.5, *type* is a pixel type from Table 3.4, and *img* is a pointer to a block of memory. **GetTexImage** obtains component groups from a texture image with the indicated level-of-detail (the number of components in a group is the number of components of the texture; the components are assigned among R, G, B, and A according to Table 3.6) starting with the first group in the first row, and continuing by obtaining groups in order from each row and proceeding from the first row to the last. These groups are then packed and placed in client memory as described in section 4.3.2 under **ReadPixels**. The row length and number of rows is determined by the size of the texture image (including any borders). Calling **GetTexImage** with *lod* less than zero or larger than the maximum allowable causes the error INVALID_VALUE. Calling **GetTexImage** with *format* of COLOR_INDEX, STENCIL_INDEX, or DEPTH_COMPONENT causes the error INVALID_ENUM.

The command

```
void GetPolygonStipple( void *pattern ) ;
```

obtains the polygon stipple. The pattern is packed into memory according to the procedure given in section 4.3.2 for **ReadPixels**; it is as if the *height* and *width* passed to that command were both equal to 32, the *type* were BITMAP, and the *format* were COLOR_INDEX.

Finally,

```
ubyte *GetString( enum name ) ;
```

returns a pointer to a static string describing some aspect of the current GL connection. The possible values for *name* are VENDOR, RENDERER, VERSION, and EXTENSIONS. The format of the string pointed to by the value returned by **GetString** is implementation dependent.

The tables on the following pages indicate which state variables are obtained with what commands. State variables that can be obtained using any of **GetBooleanv**, **GetIntegerv**, **GetFloatv**, or **GetDoublev** are listed with just one of these commands – the one that is most appropriate given the type of the data to be returned. These state variables cannot be obtained using **IsEnabled**. However, state variables for which **IsEnabled** is listed as the query command can also be obtained using **GetBooleanv**, **GetIntegerv**, **GetFloatv**, and **GetDoublev**. State variables for which any other command is listed as the query command can be obtained only by using that command.

Unless otherwise indicated, multi-valued state variables return their multiple values in the same order as they are given as arguments to the commands that set them. For instance, the two **DepthRange** parameters are returned in the order $n$ followed by $f$. Similarly, points for evaluator maps are returned in the order that they appeared when passed to **Map1**. **Map2** returns $R_{ij}$ in the $[(uorder)i + j]$th block of values (see page 115 for $i$, $j$, *uorder*, and $R_{ij}$ ).

Besides providing a means to obtain the values of state variables, the GL also provides a means to save and restore groups of state variables. The **PushAttrib** and **PopAttrib** commands are used for this purpose. The command

```
void PushAttrib( bitfield mask ) ;
```

takes a bitwise OR of symbolic constants indicating which groups of state variables to push onto an attribute stack. Each constant refers to a group of state variables. The classification of each variable into a group is indicated in the following tables of state variables. The command

```
void PopAttrib( void ) ;
```

resets the values of those state variables that were saved with the last **PushAttrib**. Those not saved remain unchanged. It is an error to pop an empty stack or push onto a full one. Table 6.1 shows the attribute groups with their corresponding symbolic constant names.

The depth of the attribute stack is implementation dependent but must be at least 16. The state required is potentially 16 copies of each state variable, 16 masks indicating which groups of variables are stored in each

| Attribute | Constant |
|---|---|
| accum-buffer | ACCUM_BUFFER_BIT |
| color-buffer | COLOR_BUFFER_BIT |
| current | CURRENT_BIT |
| depth-buffer | DEPTH_BUFFER_BIT |
| enable | ENABLE_BIT |
| eval | EVAL_BIT |
| fog | FOG_BIT |
| hint | HINT_BIT |
| lighting | LIGHTING_BIT |
| line | LINE_BIT |
| list | LIST_BIT |
| pixel | PIXEL_MODE_BIT |
| point | POINT_BIT |
| polygon | POLYGON_BIT |
| polygon-stipple | POLYGON_STIPPLE_BIT |
| scissor | SCISSOR_BIT |
| stencil-buffer | STENCIL_BUFFER_BIT |
| texture | TEXTURE_BIT |
| transform | TRANSFORM_BIT |
| viewport | VIEWPORT_BIT |
| - | ALL_ATTRIB_BITS |

Table 6.1: Attribute groups

stack entry, and an attribute stack pointer. In the initial state, the attribute stack is empty.

In the tables that follow, a type is indicated for each variable. Table 6.2 explains these types. The type actually identifies all state associated with the indicated description; in certain cases only a portion of this state is returned. This is the case with all matrices, where only the top entry on the stack is returned; with clip planes, where only the selected clip plane is returned, with parameters describing lights, where only the value pertaining to the selected light is returned; with textures, where only the selected texture or texture parameter is returned; and with evaluator maps, where only the selected map is returned. Finally, a "–" in the attribute column indicates that the indicated value is not included in any attribute group (and thus can not be pushed or popped with **PushAttrib** or **PopAttrib**).

| Type code | Explanation |
|---|---|
| $B$ | Boolean |
| $C$ | Color (floating-point R, G, B, and A values) |
| $CI$ | Color index (floating-point index value) |
| $T$ | Texture coordinates (floating-point $s$, $t$, $r$, $q$ values) |
| $N$ | Normal coordinates (floating-point $x$, $y$, $z$ values) |
| $V$ | Vertex, including associated data |
| $Z$ | Integer |
| $Z^+$ | Non-negative integer |
| $Z_k$, $Z_{k*}$ | $k$-valued integer ($k*$ indicates $k$ is minimum) |
| $R$ | Floating-point number |
| $R^+$ | Non-negative floating-point number |
| $R^k$ | $k$-tuple of floating-point numbers |
| $P$ | Position ($x$, $y$, $z$, $w$ floating-point coordinates) |
| $D$ | Direction ($x$, $y$, $z$ floating-point coordinates) |
| $M4$ | $4 \times 4$ floating-point matrix |
| $I$ | Image |
| $A$ | Attribute stack entry, including mask |
| $n \times type$ | $n$ copies of type $type$ ($n*$ indicates $n$ is minimum) |

Table 6.2: State variable types

GL Internal begin-end state variables (inaccessible)

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| – | $Z_{11}$ | – | 0 | When $\neq 0$, indicates **begin/end** object | 2.6.1 | – |
| – | $V$ | – | – | Previous vertex in **Begin/End line** | 2.6.1 | – |
| – | $B$ | – | – | Indicates if *line-vertex* is the first | 2.6.1 | – |
| – | $V$ | – | – | First vertex of a **Begin/End line loop** | 2.6.1 | – |
| – | $Z^{+}$ | – | – | Line stipple counter | 3.4 | – |
| – | $n \times V$ | – | – | Vertices inside of **Begin/End polygon** | 2.6.1 | – |
| – | $Z^{+}$ | – | – | Number of *polygon-vertices* | 2.6.1 | – |
| – | $2 \times V$ | – | – | Previous two vertices in a **Begin/End triangle strip** | 2.6.1 | – |
| – | $Z_3$ | – | – | Number of vertices so far in triangle strip: 0, 1, or more | 2.6.1 | – |
| – | $Z_2$ | – | – | Triangle strip A/B vertex pointer | 2.6.1 | – |
| – | $3 \times V$ | – | – | Vertices of the quad under construction | 2.6.1 | – |
| – | $Z_4$ | – | – | Number of vertices so far in quad strip: 0, 1, 2, or more | 2.6.1 | – |

Current Values and Associated Data

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| CURRENT_COLOR | $C$ | **GetIntegerv, GetFloatv** | 1,1,1,1 | Current color | 2.7 | current |
| CURRENT_INDEX | $CI$ | **GetIntegerv, GetFloatv** | 1 | Current color index | 2.7 | current |
| CURRENT_TEXTURE_COORDS | $T$ | **GetFloatv** | 0,0,0,1 | Current texture coordinates | 2.7 | current |
| CURRENT_NORMAL | $N$ | **GetFloatv** | 0,0,1 | Current normal | 2.7 | current |
| – | $C$ | – | - | Color associated with last vertex | 2.6 | – |
| – | $CI$ | – | - | Color index associated with last vertex | 2.6 | – |
| – | $T$ | – | - | Texture coordinates associated with last vertex | 2.6 | – |
| CURRENT_RASTER_POSITION | $R^4$ | **GetFloatv** | 0,0,0,1 | Current raster position | 2.11 | current |
| CURRENT_RASTER_DISTANCE | $R^+$ | **GetFloatv** | 0 | Current raster distance | 2.11 | current |
| CURRENT_RASTER_COLOR | $C$ | **GetIntegerv, GetFloatv** | 1,1,1,1 | Color associated with raster position | 2.11 | current |
| CURRENT_RASTER_INDEX | $CI$ | **GetIntegerv, GetFloatv** | 1 | Color index associated with raster position | 2.11 | current |
| CURRENT_RASTER_TEXTURE_COORDS | $T$ | **GetFloatv** | 0,0,0,1 | Texture coordinates associated with raster position | 2.11 | current |
| CURRENT_RASTER_POSITION_VALID | $B$ | **GetBooleanv** | *True* | Raster position valid bit | 2.11 | current |
| EDGE_FLAG | $B$ | **GetBooleanv** | *True* | Edge flag | 2.6.2 | current |

Transformation state

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| MODELVIEW_MATRIX | $32 * \times M4$ | **GetFloatv** | Identity | Model-view matrix stack | 2.9.2 | – |
| PROJECTION_MATRIX | $2 * \times M4$ | **GetFloatv** | Identity | Projection matrix stack | 2.9.2 | – |
| TEXTURE_MATRIX | $2 * \times M4$ | **GetFloatv** | Identity | Texture matrix stack | 2.9.2 | – |
| VIEWPORT | $4 \times Z$ | **GetIntegerv** | see 2.9.1 | Viewport origin & extent | 2.9.1 | viewport |
| DEPTH_RANGE | $2 \times R^{+}$ | **GetFloatv** | 0,1 | Depth range near & far | 2.9.1 | viewport |
| MODELVIEW_STACK_DEPTH | $Z^{+}$ | **GetIntegerv** | 1 | Model-view matrix stack pointer | 2.9.2 | – |
| PROJECTION_STACK_DEPTH | $Z^{+}$ | **GetIntegerv** | 1 | Projection matrix stack pointer | 2.9.2 | – |
| TEXTURE_STACK_DEPTH | $Z^{+}$ | **GetIntegerv** | 1 | Texture matrix stack pointer | 2.9.2 | – |
| MATRIX_MODE | $Z_3$ | **GetIntegerv** | `MODELVIEW` | Current matrix mode | 2.9.2 | transform |
| NORMALIZE | $B$ | **IsEnabled** | *False* | Current normal normalization on/off | 2.9.3 | transform/enable |
| CLIP_PLANE$i$ | $6 * \times R^4$ | **GetClipPlane** | 0,0,0,0 | User clipping plane coefficients | 2.10 | transform |
| CLIP_PLANE$i$ | $6 * \times B$ | **IsEnabled** | *False* | $x$th user clipping plane enabled | 2.10 | transform/enable |

Coloring

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| FOG_COLOR | $C$ | **GetFloatv** | 0,0,0,0 | Fog color | 3.9 | fog |
| FOG_INDEX | $I$ | **GetFloatv** | 0 | Fog index | 3.9 | fog |
| FOG_DENSITY | $R$ | **GetFloatv** | 1.0 | Exponential fog density | 3.9 | fog |
| FOG_START | $R$ | **GetFloatv** | 0.0 | Linear fog start | 3.9 | fog |
| FOG_END | $R$ | **GetFloatv** | 1.0 | Linear fog end | 3.9 | fog |
| FOG_MODE | $Z_3$ | **GetIntegerv** | `EXP` | Fog mode | 3.9 | fog |
| FOG | $B$ | **IsEnabled** | *False* | True if fog enabled | 3.9 | fog/enable |
| SHADE_MODEL | $Z^+$ | **GetIntegerv** | `SMOOTH` | **ShadeModel** setting | 2.12.7 | lighting |

Lighting (see also Table 2.5 for defaults)

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| LIGHTING | $B$ | **IsEnabled** | *False* | True if lighting is enabled | 2.12.1 | lighting/enable |
| COLOR_MATERIAL | $B$ | **IsEnabled** | *False* | True if color tracking is enabled | 2.12.3 | lighting |
| COLOR_MATERIAL_PARAMETER | $Z_5$ | **GetIntegerv** | `AMBIENT_AND_DIFFUSE` | Material properties tracking current color | 2.12.3 | lighting |
| COLOR_MATERIAL_FACE | $Z_3$ | **GetIntegerv** | `FRONT_AND_BACK` | Face(s) affected by color tracking | 2.12.3 | lighting |
| AMBIENT | $2 \times C$ | **GetMaterialfv** | (0.2,0.2,0.2,1.0) | Ambient material color | 2.12.1 | lighting |
| DIFFUSE | $2 \times C$ | **GetMaterialfv** | (0.8,0.8,0.8,1.0) | Diffuse material color | 2.12.1 | lighting |
| SPECULAR | $2 \times C$ | **GetMaterialfv** | (0.0,0.0,0.0,1.0) | Specular material color | 2.12.1 | lighting |
| EMISSION | $2 \times C$ | **GetMaterialfv** | (0.0,0.0,0.0,1.0) | Emissive mat. color | 2.12.1 | lighting |
| SHININESS | $2 \times R$ | **GetMaterialfv** | 0.0 | Specular exponent of material | 2.12.1 | lighting |
| LIGHT_MODEL_AMBIENT | $C$ | **GetFloatv** | (0.2,0.2,0.2,1.0) | Ambient scene color | 2.12.1 | lighting |
| LIGHT_MODEL_LOCAL_VIEWER | $B$ | **GetBooleanv** | *False* | Viewer is local | 2.12.1 | lighting |
| LIGHT_MODEL_TWO_SIDE | $B$ | **GetBooleanv** | *False* | Use two-sided lighting | 2.12.1 | lighting |
| AMBIENT | $8 * \times C$ | **GetLightfv** | (0.0,0.0,0.0,1.0) | Ambient intensity of light $i$ | 2.12.1 | lighting |
| DIFFUSE | $8 * \times C$ | **GetLightfv** | see 2.5 | Diffuse intensity of light $i$ | 2.12.1 | lighting |
| SPECULAR | $8 * \times C$ | **GetLightfv** | see 2.5 | Specular intensity of light $i$ | 2.12.1 | lighting |
| POSITION | $8 * \times P$ | **GetLightfv** | (0.0,0.0,1.0,0.0) | Position of light $i$ | 2.12.1 | lighting |
| CONSTANT_ATTENUATION | $8 * \times R^+$ | **GetLightfv** | 1.0 | Constant atten. factor | 2.12.1 | lighting |
| LINEAR_ATTENUATION | $8 * \times R^+$ | **GetLightfv** | 0.0 | Linear atten. factor | 2.12.1 | lighting |
| QUADRATIC_ATTENUATION | $8 * \times R^+$ | **GetLightfv** | 0.0 | Quadratic atten. factor | 2.12.1 | lighting |
| SPOT_DIRECTION | $8 * \times D$ | **GetLightfv** | (0.0,0.0,-1.0) | Spotlight direction of light $i$ | 2.12.1 | lighting |
| SPOT_EXPONENT | $8 * \times R^+$ | **GetLightfv** | 0.0 | Spotlight exponent of light $i$ | 2.12.1 | lighting |
| SPOT_CUTOFF | $8 * \times R^+$ | **GetLightfv** | 180.0 | Spot. angle of light $i$ | 2.12.1 | lighting |
| LIGHT$i$ | $8 * \times B$ | **IsEnabled** | *False* | True if light $i$ enabled | 2.12.1 | lighting/enable |
| COLOR_INDEXES | $2 \times 3 \times R$ | **GetFloatv** | 0,1,1 | $c_a$, $c_d$, and $c_s$ for color index lighting | 2.12.1 | lighting/enable |

Rasterization

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| POINT_SIZE | $R^+$ | **GetFloatv** | 1.0 | Point size | 3.3 | point |
| POINT_SMOOTH | $B$ | **IsEnabled** | *False* | Point antialiasing on | 3.3 | point/enable |
| LINE_WIDTH | $R^+$ | **GetFloatv** | 1.0 | Line width | 3.4 | line |
| LINE_SMOOTH | $B$ | **IsEnabled** | *False* | Line antialiasing on | 3.4 | line/enable |
| LINE_STIPPLE_PATTERN | $Z^+$ | **GetIntegerv** | 1's | Line stipple | 3.4.2 | line |
| LINE_STIPPLE_REPEAT | $Z^+$ | **GetIntegerv** | 1 | Line stipple repeat | 3.4.2 | line |
| LINE_STIPPLE | $B$ | **IsEnabled** | *False* | Line stipple enable | 3.4.2 | line/enable |
| CULL_FACE | $B$ | **IsEnabled** | *False* | Polygon culling enabled | 3.5.1 | polygon/enable |
| CULL_FACE_MODE | $Z_3$ | **GetIntegerv** | `BACK` | Cull front/back facing polygons | 3.5.1 | polygon |
| FRONT_FACE | $Z_2$ | **GetIntegerv** | `CCW` | Polygon frontface CW/CCW indicator | 3.5.1 | polygon |
| POLYGON_SMOOTH | $B$ | **IsEnabled** | *False* | Polygon antialiasing on | 3.5 | polygon/enable |
| POLYGON_MODE | $2 \times Z_3$ | **GetIntegerv** | `FILL` | Polygon rasterization mode (front & back) | 3.5.4 | polygon |
| – | $I$ | **GetPolygonStipple** | 1's | Polygon stipple | 3.5 | polygon-stipple |
| POLYGON_STIPPLE | $B$ | **IsEnabled** | *False* | Polygon stipple enable | 3.5.2 | polygon/enable |

Texturing

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| TEXTURE_$x$ | $B$ | **IsEnabled** | *False* | True if $x$-D texturing enabled ($x$ is 1D or 2D) | 3.8.4 | texture/enable |
| TEXTURE | $n \times I$ | **GetTexImage** | see 3.8 | $x$-D texture image at l.o.d. $i$ | 3.8 | – |
| TEXTURE_WIDTH | $n \times Z^+$ | **GetTexLevelParameter** | 0 | $x$-D texture image $i$'s width | 3.8 | – |
| TEXTURE_HEIGHT | $n \times Z^+$ | **GetTexLevelParameter** | 0 | $x$-D texture image $i$'s height | 3.8 | – |
| TEXTURE_BORDER | $n \times Z^+$ | **GetTexLevelParameter** | 0 | $x$-D texture image $i$'s border width | 3.8 | – |
| TEXTURE_COMPONENTS | $n \times Z_4$ | **GetTexLevelParameter** | 1 | Texture image components | 3.8 | – |
| TEXTURE_BORDER_COLOR | $2 \times C$ | **GetTexParameter** | 0,0,0,0 | Texture border color | 3.8 | texture |
| TEXTURE_MIN_FILTER | $2 \times Z_6$ | **GetTexParameter** | see 3.8 | Texture minification function | 3.8.1 | texture |
| TEXTURE_MAG_FILTER | $2 \times Z_2$ | **GetTexParameter** | see 3.8 | Texture magnification function | 3.8.2 | texture |
| TEXTURE_WRAP_$x$ | $2 \times Z_2$ | **GetTexParameter** | `REPEAT` | Texture wrap mode ($x$ is S or T) | 3.8 | texture |
| TEXTURE_ENV_MODE | $Z_3$ | **GetTexEnviv** | `MODULATE` | Texture application function | 3.8.3 | texture |
| TEXTURE_ENV_COLOR | $C$ | **GetTexEnvfv** | 0,0,0,0 | Texture environment color | 3.8.3 | texture |
| TEXTURE_GEN_$x$ | $4 \times B$ | **IsEnabled** | *False* | Texgen enabled ($x$ is S, T, R, or Q) | 2.9.4 | texture/enable |
| EYE_LINEAR | $4 \times R^4$ | **GetTexGenfv** | see 2.9.4 | Texgen plane equation coefficients | 2.9.4 | texture |
| OBJECT_LINEAR | $4 \times R^4$ | **GetTexGenfv** | see 2.9.4 | Texgen object linear coefficients | 2.9.4 | texture |
| TEXTURE_GEN_MODE | $4 \times Z_3$ | **GetTexGeniv** | `EYE_LINEAR` | Function used for texgen | 2.9.4 | texture |

Pixel Operations

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| SCISSOR_TEST | $B$ | **IsEnabled** | *False* | Scissoring enabled | 4.1.2 | scissor/enable |
| SCISSOR_BOX | $4 \times Z$ | **GetIntegerv** | see 4.1.2 | Scissor box | 4.1.2 | scissor |
| STENCIL_TEST | $B$ | **IsEnabled** | *False* | Stenciling enabled | 4.1.4 | stencil-buffer/enable |
| STENCIL_FUNC | $Z_8$ | **GetIntegerv** | `ALWAYS` | Stencil function | 4.1.4 | stencil-buffer |
| STENCIL_VALUE_MASK | $Z^+$ | **GetIntegerv** | 1's | Stencil mask | 4.1.4 | stencil-buffer |
| STENCIL_REF | $Z^+$ | **GetIntegerv** | 0 | Stencil reference value | 4.1.4 | stencil-buffer |
| STENCIL_FAIL | $Z_6$ | **GetIntegerv** | `KEEP` | Stencil fail action | 4.1.4 | stencil-buffer |
| STENCIL_PASS_DEPTH_FAIL | $Z_6$ | **GetIntegerv** | `KEEP` | Stencil depth buffer fail action | 4.1.4 | stencil-buffer |
| STENCIL_PASS_DEPTH_PASS | $Z_6$ | **GetIntegerv** | `KEEP` | Stencil depth buffer pass action | 4.1.4 | stencil-buffer |
| ALPHA_TEST | $B$ | **IsEnabled** | *False* | Alpha test enabled | 4.1.3 | color-buffer/enable |
| ALPHA_TEST_FUNC | $Z_8$ | **GetIntegerv** | `ALWAYS` | Alpha test function | 4.1.3 | color-buffer |
| ALPHA_TEST_REF | $R^+$ | **GetIntegerv** | 0 | Alpha test reference value | 4.1.3 | color-buffer |
| DEPTH_TEST | $B$ | **IsEnabled** | *False* | Depth buffer enabled | 4.1.5 | depth-buffer/enable |
| DEPTH_FUNC | $Z_8$ | **GetIntegerv** | `LESS` | Depth buffer test function | 4.1.5 | depth-buffer |
| BLEND | $B$ | **IsEnabled** | *False* | Blending enabled | 4.1.6 | color-buffer/enable |
| BLEND_SRC | $Z_9$ | **GetIntegerv** | `ONE` | Blending source function | 4.1.6 | color-buffer |
| BLEND_DST | $Z_8$ | **GetIntegerv** | `ZERO` | Blending destination function | 4.1.6 | color-buffer |
| LOGIC_OP | $B$ | **IsEnabled** | *False* | Logic op enabled | 4.1.8 | color-buffer/enable |
| LOGIC_OP_MODE | $Z_{16}$ | **GetIntegerv** | `COPY` | Logic op function | 4.1.8 | color-buffer |
| DITHER | $B$ | **IsEnabled** | *True* | Dithering enabled | 4.1.7 | color-buffer/enable |

Framebuffer Control

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| DRAW_BUFFER | $Z_{14}$ | **GetIntegerv** | see 4.2.1 | Buffers selected for drawing | 4.2.1 | color-buffer |
| INDEX_WRITEMASK | $Z^+$ | **GetIntegerv** | 1's | Color index writemask | 4.2.2 | color-buffer |
| COLOR_WRITEMASK | $4 \times B$ | **GetBooleanv** | *True* | Color write enables; R, G, B, or A | 4.2.2 | color-buffer |
| DEPTH_WRITEMASK | $B$ | **GetBooleanv** | *True* | Depth buffer enabled for writing | 4.2.2 | depth-buffer |
| STENCIL_WRITEMASK | $Z^+$ | **GetIntegerv** | 1's | Stencil buffer writemask | 4.2.2 | stencil-buffer |
| COLOR_CLEAR_VALUE | $C$ | **GetFloatv** | 0,0,0,0 | Color buffer clear value (RGBA mode) | 4.2.3 | color-buffer |
| INDEX_CLEAR_VALUE | $CI$ | **GetFloatv** | 0 | Color buffer clear value (color index mode) | 4.2.3 | color-buffer |
| DEPTH_CLEAR_VALUE | $R^+$ | **GetIntegerv** | 1 | Depth buffer clear value | 4.2.3 | depth-buffer |
| STENCIL_CLEAR_VALUE | $Z^+$ | **GetIntegerv** | 0 | Stencil clear value | 4.2.3 | stencil-buffer |
| ACCUM_CLEAR_VALUE | $4 \times R^+$ | **GetFloatv** | 0 | Accumulation buffer clear value | 4.2.3 | accum-buffer |

Pixels

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| UNPACK_SWAP_BYTES | $B$ | **GetBooleanv** | *False* | Value of `UNPACK_SWAP_BYTES` | 4.3 | – |
| UNPACK_LSB_FIRST | $B$ | **GetBooleanv** | *False* | Value of `UNPACK_LSB_FIRST` | 4.3 | – |
| UNPACK_ROW_LENGTH | $Z^+$ | **GetIntegerv** | 0 | Value of `UNPACK_ROW_LENGTH` | 4.3 | – |
| UNPACK_SKIP_ROWS | $Z^+$ | **GetIntegerv** | 0 | Value of `UNPACK_SKIP_ROWS` | 4.3 | – |
| UNPACK_SKIP_PIXELS | $Z^+$ | **GetIntegerv** | 0 | Value of `UNPACK_SKIP_PIXELS` | 4.3 | – |
| UNPACK_ALIGNMENT | $Z^+$ | **GetIntegerv** | 4 | Value of `UNPACK_ALIGNMENT` | 4.3 | – |
| PACK_SWAP_BYTES | $B$ | **GetBooleanv** | *False* | Value of `PACK_SWAP_BYTES` | 4.3 | – |
| PACK_LSB_FIRST | $B$ | **GetBooleanv** | *False* | Value of `PACK_LSB_FIRST` | 4.3 | – |
| PACK_ROW_LENGTH | $Z^+$ | **GetIntegerv** | 0 | Value of `PACK_ROW_LENGTH` | 4.3 | – |
| PACK_SKIP_ROWS | $Z^+$ | **GetIntegerv** | 0 | Value of `PACK_SKIP_ROWS` | 4.3 | – |
| PACK_SKIP_PIXELS | $Z^+$ | **GetIntegerv** | 0 | Value of `PACK_SKIP_PIXELS` | 4.3 | – |
| PACK_ALIGNMENT | $Z^+$ | **GetIntegerv** | 4 | Value of `PACK_ALIGNMENT` | 4.3 | – |

Pixels (cont.)

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| MAP_COLOR | $B$ | **GetBooleanv** | *False* | True if colors are mapped | 4.3 | pixel |
| MAP_STENCIL | $B$ | **GetBooleanv** | *False* | True if stencil values are mapped | 4.3 | pixel |
| INDEX_SHIFT | $Z$ | **GetIntegerv** | 0 | Value of `INDEX_SHIFT` | 4.3 | pixel |
| INDEX_OFFSET | $Z$ | **GetIntegerv** | 0 | Value of `INDEX_OFFSET` | 4.3 | pixel |
| $x$_SCALE | $R$ | **GetFloatv** | 1 | Value of $x$`_SCALE`; $x$ is `RED`, `GREEN`, `BLUE`, `ALPHA`, or `DEPTH` | 4.3 | pixel |
| $x$_BIAS | $R$ | **GetFloatv** | 0 | Value of $x$`_BIAS`; $x$ is one of `RED`, `GREEN`, `BLUE`, `ALPHA`, or `DEPTH` | 4.3 | pixel |
| ZOOM_X | $R$ | **GetFloatv** | 1.0 | $x$ zoom factor | 4.3 | pixel |
| ZOOM_Y | $R$ | **GetFloatv** | 1.0 | $y$ zoom factor | 4.3 | pixel |
| $x$ | $8 \times 32 * \times R$ | **GetPixelMap** | 0's | RGBA **PixelMap** translation tables; $x$ is a map name from Table 3.3 | 4.3 | – |
| $x$ | $2 \times 32 * \times Z$ | **GetPixelMap** | 0's | Index **PixelMap** translation tables; $x$ is a map name from Table 3.3 | 4.3 | – |
| $x$_SIZE | $Z^+$ | **GetIntegerv** | 1 | Size of table $x$ | 4.3 | – |
| READ_BUFFER | $Z_3$ | **GetIntegerv** | see 4.3.2 | Read source buffer | 4.3 | pixel |

Evaluators (**GetMap** takes a map name)

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| ORDER | $9 \times Z_{8*}$ | **GetMapiv** | 1 | 1d map order | 5.1 | – |
| ORDER | $9 \times 2 \times Z_{8*}$ | **GetMapiv** | 1,1 | 2d map orders | 5.1 | – |
| COEFF | $9 \times 8 * \times R^n$ | **GetMapfv** | see 5.1 | 1d control points | 5.1 | – |
| COEFF | $9 \times 8 * \times 8 * \times R^n$ | **GetMapfv** | see 5.1 | 2d control points | 5.1 | – |
| DOMAIN | $9 \times 2 \times R$ | **GetMapfv** | see 5.1 | 1d domain endpoints | 5.1 | – |
| DOMAIN | $9 \times 4 \times R$ | **GetMapfv** | see 5.1 | 2d domain endpoints | 5.1 | – |
| MAP1_$x$ | $9 \times B$ | **IsEnabled** | *False* | 1d map enables: $x$ is map type | 5.1 | eval/enable |
| MAP2_$x$ | $9 \times B$ | **IsEnabled** | *False* | 2d map enables: $x$ is map type | 5.1 | eval/enable |
| MAP1_GRID_DOMAIN | $2 \times R$ | **GetFloatv** | 0,1 | 1d grid endpoints | 5.1 | eval |
| MAP2_GRID_DOMAIN | $4 \times R$ | **GetFloatv** | 0,1;0,1 | 2d grid endpoints | 5.1 | eval |
| MAP1_GRID_SEGMENTS | $Z^+$ | **GetFloatv** | 1 | 1d grid divisions | 5.1 | eval |
| MAP2_GRID_SEGMENTS | $2 \times Z^+$ | **GetFloatv** | 1,1 | 2d grid divisions | 5.1 | eval |
| AUTO_NORMAL | $B$ | **IsEnabled** | *False* | True if automatic normal generation enabled | 5.1 | eval |

Hints

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| PERSPECTIVE_CORRECTION_HINT | $Z_3$ | **GetIntegerv** | DONT_CARE | Perspective correction hint | 5.6 | hint |
| POINT_SMOOTH_HINT | $Z_3$ | **GetIntegerv** | DONT_CARE | Point smooth hint | 5.6 | hint |
| LINE_SMOOTH_HINT | $Z_3$ | **GetIntegerv** | DONT_CARE | Line smooth hint | 5.6 | hint |
| POLYGON_SMOOTH_HINT | $Z_3$ | **GetIntegerv** | DONT_CARE | Polygon smooth hint | 5.6 | hint |
| FOG_HINT | $Z_3$ | **GetIntegerv** | DONT_CARE | Fog hint | 5.6 | hint |

Implementation Dependent Values

| Get value | Type | Get Cmnd | Minimum Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| MAX_LIGHTS | $Z^+$ | **GetIntegerv** | 8 | Maximum number of lights | 2.12.1 | – |
| MAX_CLIP_PLANES | $Z^+$ | **GetIntegerv** | 6 | Maximum number of user clipping planes | 2.10 | – |
| MAX_MODELVIEW_STACK_DEPTH | $Z^+$ | **GetIntegerv** | 32 | Maximum model-view stack depth | 2.9.2 | – |
| MAX_PROJECTION_STACK_DEPTH | $Z^+$ | **GetIntegerv** | 2 | Maximum projection matrix stack depth | 2.9.2 | – |
| MAX_TEXTURE_STACK_DEPTH | $Z^+$ | **GetIntegerv** | 2 | Maximum number depth of texture matrix stack | 2.9.2 | – |
| SUBPIXEL_BITS | $Z^+$ | **GetIntegerv** | 4 | Number of bits of subpixel precision in x & y | 3 | – |
| MAX_TEXTURE_SIZE | $Z^+$ | **GetIntegerv** | 64 | Maximum height or width of a texture image (w/o borders) | 3.8 | – |
| MAX_PIXEL_MAP_TABLE | $Z^+$ | **GetIntegerv** | 32 | Maximum size of a **PixelMap** translation table | 3.6.2 | – |
| MAX_NAME_STACK_DEPTH | $Z^+$ | **GetIntegerv** | 64 | Maximum selection name stack depth | 5.2 | – |
| MAX_LIST_NESTING | $Z^+$ | **GetIntegerv** | 64 | Maximum display list call nesting | 5.4 | – |
| MAX_EVAL_ORDER | $Z^+$ | **GetIntegerv** | 8 | Maximum evaluator polynomial order | 5.1 | – |
| MAX_VIEWPORT_DIMS | $2 \times Z^+$ | **GetIntegerv** | see 2.9.1 | Maximum viewport dimensions | 2.9.1 | – |
| MAX_ATTRIB_STACK_DEPTH | $Z^+$ | **GetIntegerv** | 16 | Maximum depth of the attribute stack | 6 | – |

More Implementation Dependent Values

| Get value | Type | Get Cmnd | Minimum Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| AUX_BUFFERS | $Z^+$ | **GetIntegerv** | 0 | Number of auxiliary buffers | 4.2.1 | – |
| RGBA_MODE | $B$ | **GetBooleanv** | – | True if color buffers store rgba | 2.7 | – |
| INDEX_MODE | $B$ | **GetBooleanv** | – | True if color buffers store indexes | 2.7 | – |
| DOUBLEBUFFER | $B$ | **GetBooleanv** | – | True if front & back buffers exist | 4.2.1 | – |
| STEREO | $B$ | **GetBooleanv** | – | True if left & right buffers exist | 6 | – |
| POINT_SIZE_RANGE | $2 \times R^+$ | **GetFloatv** | 1,1 | Range (lo to hi) of antialiased point sizes | 3.3 | – |
| POINT_SIZE_GRANULARITY | $R^+$ | **GetFloatv** | – | Antialiased point size granularity | 3.3 | – |
| LINE_WIDTH_RANGE | $2 \times R^+$ | **GetFloatv** | 1,1 | Range (lo to hi) of antialiased line widths | 3.4 | – |
| LINE_WIDTH_GRANULARITY | $R^+$ | **GetFloatv** | – | Antialiased line width granularity | 3.4 | – |

Implementation Dependent Pixel Depths

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| RED_BITS | $Z^+$ | **GetIntegerv** | - | Number of bits per red component in color buffers | 4 | – |
| GREEN_BITS | $Z^+$ | **GetIntegerv** | - | Number of bits per green component in color buffers | 4 | – |
| BLUE_BITS | $Z^+$ | **GetIntegerv** | - | Number of bits per blue component in color buffers | 4 | – |
| ALPHA_BITS | $Z^+$ | **GetIntegerv** | - | Number of bits per alpha component in color buffers | 4 | – |
| INDEX_BITS | $Z^+$ | **GetIntegerv** | - | Number of bits per index in color buffers | 4 | – |
| DEPTH_BITS | $Z^+$ | **GetIntegerv** | - | Number of depth buffer planes | 4 | – |
| STENCIL_BITS | $Z^+$ | **GetIntegerv** | - | Number of stencil planes | 4 | – |
| ACCUM_RED_BITS | $Z^+$ | **GetIntegerv** | - | Number of bits per red component in the accumulation buffer | 4 | – |
| ACCUM_GREEN_BITS | $Z^+$ | **GetIntegerv** | - | Number of bits per green component in the accumulation buffer | 4 | – |
| ACCUM_BLUE_BITS | $Z^+$ | **GetIntegerv** | - | Number of bits per blue component in the accumulation buffer | 4 | – |
| ACCUM_ALPHA_BITS | $Z^+$ | **GetIntegerv** | - | Number of bits per alpha component in the accumulation buffer | 4 | – |

Miscellaneous

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| LIST_BASE | $Z^+$ | **GetIntegerv** | 0 | Setting of **ListBase** | 5.4 | list |
| LIST_INDEX | $Z^+$ | **GetIntegerv** | 0 | number of display list under construction; 0 if none | 5.4 | – |
| LIST_MODE | $Z^+$ | **GetIntegerv** | 0 | Mode of display list under construction; undefined if none | 5.4 | – |
| – | $16 * \times A$ | – | empty | Attribute stack | 6 | – |
| ATTRIB_STACK_DEPTH | $Z^+$ | **GetIntegerv** | 0 | Attribute stack pointer | 6 | – |
| NAME_STACK_DEPTH | $Z^+$ | **GetIntegerv** | 0 | Name stack depth | 5.2 | – |
| RENDER_MODE | $Z_3$ | **GetIntegerv** | `RENDER` | **RenderMode** setting | 5.2 | – |
| – | $Z^+$ | – | – | Selection buffer pointer | 5.2 | – |
| – | $Z^+$ | – | – | Selection buffer size | 5.2 | – |
| – | $Z^+$ | – | – | Feedback buffer pointer | 5.3 | – |
| – | $Z^+$ | – | – | Feedback buffer size | 5.3 | – |
| – | $Z_5$ | – | – | Feedback type | 5.3 | – |
| – | $n \times Z_8$ | **GetError** | 0 | Current error code(s) | 2.5 | – |
| – | $n \times B$ | – | *False* | True if there is a corresponding error | 2.5 | – |

# Appendix A

# Invariance

The OpenGL specification is not pixel exact. It therefore does not guarantee an exact match between images produced by different GL implementations. However, the specification does specify exact matches, in some cases, for images produced by the same implementation. The purpose of this appendix is to identify and provide justification for those cases that require exact matches.

## A.1 Repeatability

The obvious and most fundamental case is repeated issuance of a series of GL commands. For any given GL and framebuffer state *vector*, and for any GL command, the resulting GL and framebuffer state must be identical whenever the command is executed on that initial GL and framebuffer state.

One purpose of repeatability is avoidance of visual artifacts when a double-buffered scene is redrawn. If rendering is not repeatable, swapping between two buffers rendered with the same command sequence may result in visible changes in the image. Such false motion is distracting to the viewer. Another reason for repeatability is testability.

Repeatability, while important, is a weak requirement. Given only repeatability as a requirement, two scenes rendered with one (small) polygon changed in position might differ at every pixel. Such a difference, while within the law of repeatability, is certainly not within its spirit. Additional invariance rules are desirable to ensure useful operation.

## A.2 Multi-pass Algorithms

Invariance is necessary for a whole set of useful multi-pass algorithms. Such algorithms render multiple times, each time with a different GL mode vector, to eventually produce a result in the framebuffer. Examples of these algorithms include:

- "Erasing" a primitive from the framebuffer by redrawing it, either in a different color or using the XOR logical operation.
- Using stencil operations to compute capping planes.

On the other hand, invariance rules can greatly increase the complexity of high-performance implementations of the GL. Even the weak repeatability requirement significantly constrains a parallel implementation of the GL. Because GL implementations are required to implement ALL GL capabilities, not just a convenient subset, those that utilize hardware acceleration are expected to alternate between hardware and software modules based on the current GL mode vector. A strong invariance requirement forces the behavior of the hardware and software modules to be identical, something that may be very difficult to achieve (for example, if the hardware does floating-point operations with different precision than the software).

What is desired is a compromise that results in many compliant, high-performance implementations, and in many software vendors choosing to port to OpenGL.

## A.3 Invariance Rules

For a given instantiation of an OpenGL rendering context:

**Rule 1** *For any given GL and framebuffer state vector, and for any given GL command, the resulting GL and framebuffer state must be identical each time the command is executed on that initial GL and framebuffer state.*

**Rule 2** *Changes to the following state values have no side effects (the use of any other state value is not affected by the change):*

***Required:***

- *Framebuffer contents (all bitplanes)*
- *The color buffers enabled for writing*

- *The values of matrices other than the top-of-stack matrices*
- *Scissor parameters (other than enable)*
- *Writemasks (color, index, depth, stencil)*
- *Clear values (color, index, depth, stencil, accumulation)*

◦ *Current values (color, index, normal, texture coords, edgeflag)*

◦ *Current raster color, index and texture coordinates.*

◦ *Material properties (ambient, diffuse, specular, emission, shininess)*

***Strongly suggested:***

- *Matrix mode*
- *Matrix stack depths*
- *Alpha test parameters (other than enable)*
- *Stencil parameters (other than enable)*
- *Depth test parameters (other than enable)*
- *Blend parameters (other than enable)*
- *Logical operation parameters (other than enable)*
- *Pixel storage and transfer state*
- *Evaluator state (except as it affects the vertex data generated by the evaluators)*

**Corollary 1** *Fragment generation is invariant with respect to the state values marked with • in Rule 2.*

**Corollary 2** *The window coordinates (x, y, and z) of generated fragments are also invariant with respect to*

***Required:***

- *Current values (color, color index, normal, texture coords, edgeflag)*
- *Current raster color, color index, and texture coordinates*
- *Material properties (ambient, diffuse, specular, emission, shininess)*

**Rule 3** *The arithmetic of each per-fragment operation is invariant except with respect to parameters that directly control it (the parameters that control the alpha test, for instance, are the alpha test enable, the alpha test function, and the alpha test reference value).*

**Corollary 3** *Images rendered into different color buffers, either simultaneously or separately using the same command sequence, are pixel identical. (Note that this does not hold between X * pixmaps and color buffers, however.)*

## A.4 What All This Means

Hardware accelerated GL implementations are expected to default to software operation when some GL state vectors are encountered. Even the weak repeatability requirement means, for example, that OpenGL implementations cannot apply hysteresis to this swap, but must instead guarantee that a given mode vector implies that a subsequent command *always* is executed in either the hardware or the software machine.

The stronger invariance rules constrain when the switch from hardware to software rendering can occur, given that the software and hardware renderers are not pixel identical. For example, the switch can be made when blending is enabled or disabled, but it should not be made when a change is made to the blending parameters.

Because floating point values may be represented using different formats in different renderers (hardware and software), many OpenGL state values may change subtly when renderers are swapped. This is the type of state value change that Rule 1 seeks to avoid.

---

*X is a registered trademark of the MIT X Consortium.

# Appendix B

# Corollaries

The following observations are derived from the body and the other appendixes of the specification. Absence of an observation from this list in no way impugns its veracity.

1. The CURRENT_RASTER_TEXTURE_COORDINATES must be maintained correctly at all times, including periods while texture mapping is not enabled, and when the GL is in color index mode.

2. When requested, texture coordinates returned in feedback mode are always valid, including periods while texture mapping is not enabled, and when the GL is in color index mode.

3. The error semantics of upward compatible OpenGL revisions may change. Otherwise, only additions can be made to upward compatible revisions.

4. GL query commands are not required to satisfy the semantics of the **Flush** or the **Finish** commands. All that is required is that the queried state be consistent with complete execution of all previously executed GL commands.

5. Application specified point size and line width must be returned as specified when queried. Implementation dependent clamping affects the values only while they are in use.

6. Bitmaps and pixel transfers do not cause selection hits.

7. The mask specified as the third argument to **StencilFunc** affects the operands of the stencil comparison function, but has no direct effect on the update of the stencil buffer. The mask specified by **StencilMask** has no effect on the stencil comparison function; it limits the effect of the update of the stencil buffer.

8. Polygon shading is completed before the polygon mode is interpreted. If the shade model is `FLAT`, all of the points or lines generated by a single polygon will have the same color.

9. A display list is just a group of commands and arguments, so errors generated by commands in a display list must be generated when the list is executed. If the list is created in `COMPILE` mode, errors should not be generated while the list is being created.

10. **RasterPos** does not change the current raster index from its default value in an RGBA mode GL context. Likewise, **RasterPos** does not change the current raster color from its default value in a color index GL context. Both the current raster index and the current raster color can be queried, however, regardless of the color mode of the GL context.

11. A material property that is attached to the current color via **ColorMaterial** always takes the value of the current color. Attempts to change that material property via **Material** calls have no effect.

12. **Material** and **ColorMaterial** can be used to modify the RGBA material properties, even in a color index context. Likewise, **Material** can be used to modify the color index material properties, even in an RGBA context.

13. Their is no atomicity requirement for OpenGL rendering commands, even at the fragment level.

14. Because rasterization of non-antialiased polygons is point sampled, polygons that have no area generate no fragments when they are rasterized in `FILL` mode, and the fragments generated by the rasterization of "narrow" polygons may not form a continuous array.

15. OpenGL does not force left- or right-handedness on any of its coordinates systems. Consider, however, the following conditions: (1) the object coordinate system is right-handed; (2) the only commands used

to manipulate the model-view matrix are **Scale** (with positive scaling values only), **Rotate**, and **Translate**; (3) exactly one of either **Frustum** or **Ortho** is used to set the projection matrix; (4) the near value is less than the far value for **DepthRange**. If these conditions are all satisfied, then the eye coordinate system is right-handed and the clip, normalized device, and window coordinate systems are left-handed.

16. ColorMaterial has no effect on color index lighting.

17. (No pixel dropouts or duplicates.) Let two polygons share an identical edge (that is, there exist vertices A and B of an edge of one polygon, and vertices C and D of an edge of the other polygon, and the coordinates of vertex A (resp. B) are identical to those of vertex C (resp. D), and the state of the the coordinate transfomations is identical when A, B, C, and D are specified). Then, when the fragments produced by rasterization of both polygons are taken together, each fragment intersecting the interior of the shared edge is produced exactly once.

# Index of GL calls

*GetString, 131

Accum, 105
AlphaFunc, 94

Begin, 12, 14–19, 21, 47, 55, 58, 59, 62, 65, 117, 118, 123
Bitmap, 76, 77
BlendFunc, 97

CallList, 19, 125, 126
CallLists, 19, 125, 126
Clear, 103, 104
ClearAccum, 104
ClearColor, 103
ClearDepth, 104
ClearIndex, 104
ClearStencil, 104
ClipPlane, 31
Color, 19, 20, 36, 48
Color3, 20
Color4, 20
Colorb, 37
Colorf, 37
Colori, 37
ColorMask, 102
ColorMaterial, 42, 44, 45, 158
Colors, 37
Colorub, 37, 48
Colorui, 37, 48
Colorus, 37, 48
CopyPixels, 66–68, 106, 111, 112, 122
CullFace, 62, 63

DeleteLists, 127
DepthFunc, 96
DepthMask, 103
DepthRange, 23, 132, 159
Disable, 28, 30, 32, 37, 42, 52, 55, 58, 62, 65, 87, 89, 94–97, 99, 115, 116
DrawBuffer, 101, 102
DrawPixels, 64, 66–72, 76–78, 106, 109, 110, 112, 122

EdgeFlag, 18, 19
EdgeFlagv, 18
Enable, 28, 30, 32, 37, 42, 52, 55, 58, 62, 65, 87, 89, 94–97, 99, 115, 116, 129
End, 12, 14–17, 19, 21, 55, 62, 65, 117, 123
EndList, 125
EvalCoord, 19, 115
EvalCoord1, 115, 117, 118
EvalCoord1d, 117
EvalCoord1f, 117
EvalCoord2, 115, 118, 119
EvalMesh1, 117
EvalMesh2, 117, 118
EvalPoint, 19

EvalPoint1, 118
EvalPoint2, 118

FeedbackBuffer, 122, 127
Finish, 127, 128, 157
Flush, 127, 157
Fog, 89, 90
FrontFace, 41, 62
Frustum, 25, 26, 159

GenLists, 126, 127
Get, 24, 127, 129
GetBooleanv, 129, 132
GetClipPlane, 130
GetDoublev, 129, 130, 132
GetError, 11
GetFloatv, 129, 132
GetIntegerv, 129, 132
GetMap, 130, 131
GetMaterial, 130
GetPixelMap, 130, 131
GetPolygonStipple, 131
GetString, 132
GetTexEnv, 130
GetTexGen, 130
GetTexImage, 131
GetTexLevelParameter, 130
GetTexParameter, 130

Hint, 128

Index, 19–21
IndexMask, 102, 103
InitNames, 119, 120
IsEnabled, 127, 129, 132
IsList, 127

Light, 41–43
LightModel, 41, 43
LineStipple, 58
LineWidth, 55
ListBase, 126, 127
LoadIdentity, 25
LoadMatrix, 24, 25
LoadName, 120
LogicOp, 99, 100

Map1, 114, 115, 132
Map2, 114, 115, 132
MapGrid1, 117
MapGrid2, 117
Material, 19, 41–43, 46, 158
MatrixMode, 24
MultMatrix, 24, 25

NewList, 125, 126
Normal, 19, 20
Normal3, 8, 9, 20, 28
Normal3d, 9
Normal3dv, 9
Normal3f, 9
Normal3fv, 9

Ortho, 25, 27, 159

PassThrough, 123
PixelMap, 66, 68–70, 107, 112
PixelStore, 66–68, 70, 107, 112, 127
PixelTransfer, 66, 68, 70, 107, 112
PixelZoom, 76
PointSize, 52
PolygonMode, 61, 65, 66, 120, 122
PolygonStipple, 64
PopAttrib, 132, 134
PopMatrix, 27
PopName, 119, 120
PushAttrib, 132, 134
PushMatrix, 27
PushName, 119, 120

RasterPos, 33, 120, 158
RasterPos2, 34
RasterPos3, 34
RasterPos4, 34
ReadBuffer, 108, 112
ReadPixels, 66–68, 71–73, 106–108, 111, 127, 131
Rect, 21, 62
RenderMode, 121–123, 127
Rotate, 25, 159

Scale, 25, 26, 159
Scissor, 94
SelectBuffer, 120, 121, 127
ShadeModel, 47
StencilFunc, 95, 96, 158
StencilMask, 103, 106, 158
StencilOp, 95, 96

TexCoord, 19, 20
TexCoord1, 20
TexCoord2, 20
TexCoord3, 20
TexCoord4, 20
TexEnv, 87
TexGen, 29, 30
TexImage1D, 79, 80, 85
TexImage2D, 78–80, 85
TexParameter, 80
Translate, 25, 26, 159

Vertex, 7, 19, 34, 116
Vertex2, 19, 21
Vertex2sv, 8
Vertex3, 19
Vertex3f, 7
Vertex4, 19
Viewport, 24