Appendix A
Order of Operations

This book describes all the operations performed between when vertices are initially specified and fragments are finally written into the framebuffer. The chapters of this book are arranged in an order that facilitates learning rather than in the exact order in which these operations are actually performed. Sometimes the exact order of operations doesn’t matter—for example, surfaces can be converted to polygons and then transformed, or transformed first and then converted to polygons, with identical results—and different implementations of OpenGL might do things differently.

This appendix describes a possible order; any implementation is required to give equivalent results. If you want more details than are presented here, see the OpenGL Reference Manual.

This appendix has the following major sections:

"Overview"

"Geometric Operations"

"Pixel Operations"

"Fragment Operations"

"Odds and Ends"

Overview

This section gives an overview of the order of operations, as shown in Figure A-1. Geometric data (vertices, lines, and polygons) follows the path through the row of boxes that include evaluators and per-vertex operations, while pixel data (pixels, images, and bitmaps) is treated differently for part of the process. Both types of data undergo the rasterization and per-fragment operations before the final pixel data is written into the framebuffer.

Figure A-1 Order of Operations

All data, whether it describes geometry or pixels, can be saved in a display list or processed immediately.
When a display list is executed, the data is sent from the display list just as if it were sent by the application.

All geometric primitives are eventually described by vertices. If evaluators are used, that data is converted to vertices and treated as vertices from then on. Vertex data may also be stored in and used from specialized vertex arrays. Per−vertex calculations are performed on each vertex, followed by rasterization to fragments. For pixel data, pixel operations are performed, and the results are either stored in the texture memory, used for polygon stippling, or rasterized to fragments.

Finally, the fragments are subjected to a series of per−fragment operations, after which the final pixel values are drawn into the framebuffer.

**Geometric Operations**

Geometric data, whether it comes from a display list, an evaluator, the vertices of a rectangle, or as raw data, consists of a set of vertices and the type of primitive it describes (a vertex, line, or polygon). Vertex data includes not only the \((x, y, z, w)\) coordinates, but also a normal vector, texture coordinates, a RGBA color, a color index, material properties, and edge−flag data. All these elements except the vertex’s coordinates can be specified in any order, and default values exist as well. As soon as the vertex command \texttt{glVertex*()} is issued, the components are padded, if necessary, to four dimensions (using \(z = 0\) and \(w = 1\)), and the current values of all the elements are associated with the vertex. The complete set of vertex data is then processed. (If vertex arrays are used, vertex data may be batch processed and processed vertices may be reused.)

**Per−Vertex Operations**

In the per−vertex operations stage of processing, each vertex’s spatial coordinates are transformed by the modelview matrix, while the normal vector is transformed by that matrix’s inverse transpose and renormalized if specified. If automatic texture generation is enabled, new texture coordinates are generated from the transformed vertex coordinates, and they replace the vertex’s old texture coordinates. The texture coordinates are then transformed by the current texture matrix and passed on to the primitive assembly step.

Meanwhile, the lighting calculations, if enabled, are performed using the transformed vertex and normal vector coordinates, and the current material, lights, and lighting model. These calculations generate new colors or indices that are clamped or masked to the appropriate range and passed on to the primitive assembly step.

**Primitive Assembly**

Primitive assembly differs, depending on whether the primitive is a point, a line, or a polygon. If flat shading is enabled, the colors or indices of all the vertices in a line or polygon are set to the same value. If special clipping planes are defined and enabled, they’re used to clip primitives of all three types. (The clipping–plane equations are transformed by the inverse transpose of the modelview matrix when they’re specified.) Point clipping simply passes or rejects vertices; line or polygon clipping can add additional vertices depending on how the line or polygon is clipped. After this clipping, the spatial coordinates of
each vertex are transformed by the projection matrix, and the results are clipped against the standard viewing planes \( x = \pm \alpha \), \( y = \pm \alpha \), and \( z = \pm \alpha \).

If selection is enabled, any primitive not eliminated by clipping generates a selection–hit report, and no further processing is performed. Without selection, perspective division by \( w \) occurs and the viewport and depth–range operations are applied. Also, if the primitive is a polygon, it’s then subjected to a culling test (if culling is enabled). A polygon might convert to vertices or lines, depending on the polygon mode.

Finally, points, lines, and polygons are rasterized to fragments, taking into account polygon or line stipples, line width, and point size. Rasterization involves determining which squares of an integer grid in window coordinates are occupied by the primitive. If antialiasing is enabled, coverage (the portion of the square that is occupied by the primitive) is also computed. Color and depth values are also assigned to each such square. If polygon offset is enabled, depth values are slightly modified by a calculated offset value.

**Pixel Operations**

Pixels from host memory are first unpacked into the proper number of components. The OpenGL unpacking facility handles a number of different formats. Next, the data is scaled, biased, and processed using a pixel map. The results are clamped to an appropriate range depending on the data type and then either written in the texture memory for use in texture mapping or rasterized to fragments.

If pixel data is read from the framebuffer, pixel–transfer operations (scale, bias, mapping, and clamping) are performed. The results are packed into an appropriate format and then returned to processor memory.

The pixel copy operation is similar to a combination of the unpacking and transfer operations, except that packing and unpacking is unnecessary, and only a single pass is made through the transfer operations before the data is written back into the framebuffer.

**Texture Memory**

OpenGL Version 1.1 provides additional control over texture memory. Texture image data can be specified from framebuffer memory, as well as processor memory. All or a portion of a texture image may be replaced. Texture data may be stored in texture objects, which can be loaded into texture memory. If there are too many texture objects to fit into texture memory at the same time, the textures that have the highest priorities remain in the texture memory.

**Fragment Operations**

If texturing is enabled, a texel is generated from texture memory for each fragment and applied to the fragment. Then fog calculations are performed, if they’re enabled, followed by the application of coverage (antialiasing) values, if antialiasing is enabled.

Next comes scissoring, followed by the alpha test (in RGBA mode only), the stencil test, and the depth–buffer test. If in RGBA mode, blending is performed. Blending is followed by dithering and logical operation. All these operations may be disabled.

The fragment is then masked by a color mask or an index mask, depending on the mode, and drawn into
the appropriate buffer. If fragments are being written into the stencil or depth buffer, masking occurs after the stencil and depth tests, and the results are drawn into the framebuffer without performing the blending, dithering, or logical operation.

Odds and Ends

Matrix operations deal with the current matrix stack, which can be the modelview, the projection, or the texture matrix stack. The commands glMultMatrix*(), glLoadMatrix*(), and glLoadIdentity() are applied to the top matrix on the stack, while glTranslate*(), glRotate*(), glScale*(), glOrtho(), and glFrustum() are used to create a matrix that’s multiplied by the top matrix. When the modelview matrix is modified, its inverse transpose is also generated for normal vector transformation.

The commands that set the current raster position are treated exactly like a vertex command up until when rasterization would occur. At this point, the value is saved and is used in the rasterization of pixel data.

The various glClear() commands bypass all operations except scissoring, dithering, and writemasking.
Appendix B
State Variables

This appendix lists the queryable OpenGL state variables, their default values, and the commands for obtaining the values of these variables. The OpenGL Reference Manual contains detailed information on all the commands and constants discussed in this appendix. This appendix has these major sections:

"The Query Commands"

"OpenGL State Variables"

The Query Commands

In addition to the basic commands to obtain the values of simple state variables (commands such as glGetIntegerv() and glEnable(), which are described in "Basic State Management" in Chapter 2), there are other specialized commands to return more complex state variables. The prototypes for these specialized commands are listed here. Some of these routines, such as glGetError() and glGetString(), have been discussed in more detail elsewhere in the book.

To find out when you need to use these commands and their corresponding symbolic constants, use the tables in the next section, "OpenGL State Variables." Also see the OpenGL Reference Manual.

void glGetClipPlane(GLenum plane, GLdouble *equation);

GLenum glGetError(void);

void glGetLightfv(GLenum light, GLenum pname, TYPE *params);

void glGetMapfv(GLenum target, GLenum query, TYPE *v);

void glGetMaterialfv(GLenum face, GLenum pname, TYPE *params);

void glGetPixelMapf(GLubyte *mask);

const GLubyte *gluGetString(GLenum name);

void glGetTexEnvfv(GLenum target, GLenum pname, TYPE *params);

void glGetTexGenfv(GLenum coord, GLenum pname, TYPE *params);

void glGetTexImage(GLenum target, GLint level, GLenum format, GLenum type, GLvoid *pixels);

void glGetTexParameterfv(GLenum target, GLenum pname, TYPE *params);

void glGetTexParameterfv(GLenum target, GLint level, GLenum pname, TYPE *params);

void gluGetNurbsProperty(GLUnurbsObj *nobj, GLenum property, GLsizei *value);

const GLubyte *gluGetString(GLenum name);
void gluGetTessProperty(GLUtesselator *tess, GLenum which, GLdouble *data);

**OpenGL State Variables**

The following pages contain tables that list the names of queryable state variables. For each variable, the tables list a description of it, its attribute group, its initial or minimum value, and the suggested `glGet*()` command to use for obtaining it. State variables that can be obtained using `glGetBooleanv()`, `glGetIntegerv()`, `glGetFloatv()`, or `glGetDoublev()` are listed with just one of these commands—the one that’s most appropriate given the type of data to be returned. (Some vertex array variables can be queried only with `glGetPointerv()`.) These state variables can’t be obtained using `glIsEnabled()`. However, state variables for which `glIsEnabled()` is listed as the query command can also be obtained using `glGetBooleanv()`, `glGetIntegerv()`, `glGetFloatv()`, and `glGetDoublev()`. State variables for which any other command is listed as the query command can be obtained only by using that command.

If one or more attribute groups are listed, the state variable belongs to the listed group or groups. If no attribute group is listed, the variable doesn’t belong to any group. `glPushAttrib()`, `glPushClientAttrib()`, `glPopAttrib()`, and `glPopClientAttrib()` may be used to save and restore all state values that belong to an attribute group. (See "Attribute Groups" in Chapter 2 for more information.)

All queryable state variables, except the implementation-dependent ones, have initial values. If no initial value is listed, you need to consult either the section where that variable is discussed or the OpenGL Reference Manual to determine its initial value.

**Current Values and Associated Data**

<table>
<thead>
<tr>
<th>State Variable</th>
<th>Description</th>
<th>Attribute Group</th>
<th>Initial Value</th>
<th>Get Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_CURRENT_COLOR</td>
<td>Current color</td>
<td>current</td>
<td>1, 1, 1, 1</td>
<td><code>glGetIntegerv()</code>, <code>glGetFloatv()</code></td>
</tr>
<tr>
<td>GL_CURRENT_INDEX</td>
<td>Current color index</td>
<td>current</td>
<td>1</td>
<td><code>glGetIntegerv()</code>, <code>glGetFloatv()</code></td>
</tr>
<tr>
<td>GL_CURRENT_TEXTURE_COORDS</td>
<td>Current texture coordinates</td>
<td>current</td>
<td>0, 0, 0, 1</td>
<td><code>glGetFloatv()</code></td>
</tr>
<tr>
<td>GL_CURRENT_NORMAL</td>
<td>Current normal</td>
<td>current</td>
<td>0, 0, 1</td>
<td><code>glGetFloatv()</code></td>
</tr>
<tr>
<td>GL_CURRENT_RASTER_POSITION</td>
<td>Current raster position</td>
<td>current</td>
<td>0, 0, 1</td>
<td><code>glGetFloatv()</code></td>
</tr>
<tr>
<td>GL_CURRENT_RASTER_DISTANCE</td>
<td>Current raster distance</td>
<td>current</td>
<td>0</td>
<td><code>glGetFloatv()</code></td>
</tr>
<tr>
<td>GL_CURRENT_RASTER_COLOR</td>
<td>Color associated with raster</td>
<td>current</td>
<td>1, 1, 1, 1</td>
<td><code>glGetIntegerv()</code>, <code>glGetFloatv()</code></td>
</tr>
<tr>
<td>GL_CURRENT_RASTER_INDEX</td>
<td>Color index associated with raster position</td>
<td>current</td>
<td>1</td>
<td><code>glGetIntegerv()</code>, <code>glGetFloatv()</code></td>
</tr>
<tr>
<td>GL_CURRENT_RASTER_TEXTURE_COORDS</td>
<td>Texture coordinates associated with raster position</td>
<td>current</td>
<td>0, 0, 1</td>
<td><code>glGetFloatv()</code></td>
</tr>
<tr>
<td>GL_CURRENT_RASTER_POSITION_VALID</td>
<td>Raster position valid bit</td>
<td>current</td>
<td>GL_TRUE</td>
<td><code>glGetBooleanv()</code></td>
</tr>
<tr>
<td>GL_EDGE_FLAG</td>
<td>Edge flag</td>
<td>current</td>
<td>GL_TRUE</td>
<td><code>glGetBooleanv()</code></td>
</tr>
</tbody>
</table>

*Table B–1* State Variables for Current Values and Associated Data

**Vertex Array**

OpenGL Programming Guide – Appendix B, State Variables – 2
<table>
<thead>
<tr>
<th>State Variable</th>
<th>Description</th>
<th>Attribute Group</th>
<th>Initial Value</th>
<th>Get Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_VERTEX_ARRAY</td>
<td>Vertex array enable</td>
<td>vertex−array</td>
<td>GL_FALSE</td>
<td>glIsEnabled()</td>
</tr>
<tr>
<td>GL_VERTEX_ARRAY_SIZE</td>
<td>Coordinates per vertex</td>
<td>vertex−array</td>
<td>4</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_VERTEX_ARRAY_TYPE</td>
<td>Type of vertex coordinates</td>
<td>vertex−array</td>
<td>GL_FLOAT</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_VERTEX_ARRAY_STRIDE</td>
<td>Stride between vertices</td>
<td>vertex−array</td>
<td>0</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_VERTEX_ARRAY_POINTER</td>
<td>Pointer to the vertex array</td>
<td>vertex−array</td>
<td>NULL</td>
<td>glGetPointerv()</td>
</tr>
<tr>
<td>GL_NORMAL_ARRAY</td>
<td>Normal array enable</td>
<td>vertex−array</td>
<td>GL_FALSE</td>
<td>glIsEnabled()</td>
</tr>
<tr>
<td>GL_NORMAL_ARRAY_TYPE</td>
<td>Type of normal coordinates</td>
<td>vertex−array</td>
<td>GL_FLOAT</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_NORMAL_ARRAY_STRIDE</td>
<td>Stride between normals</td>
<td>vertex−array</td>
<td>0</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_NORMAL_ARRAY_POINTER</td>
<td>Pointer to the normal array</td>
<td>vertex−array</td>
<td>NULL</td>
<td>glGetPointerv()</td>
</tr>
<tr>
<td>GL_COLOR_ARRAY</td>
<td>RGBA color array enable</td>
<td>vertex−array</td>
<td>GL_FALSE</td>
<td>glIsEnabled()</td>
</tr>
<tr>
<td>GL_COLOR_ARRAY_SIZE</td>
<td>Colors per vertex</td>
<td>vertex−array</td>
<td>4</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_COLOR_ARRAY_TYPE</td>
<td>Type of color components</td>
<td>vertex−array</td>
<td>GL_FLOAT</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_COLOR_ARRAY_STRIDE</td>
<td>Stride between colors</td>
<td>vertex−array</td>
<td>0</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_COLOR_ARRAY_POINTER</td>
<td>Pointer to the color array</td>
<td>vertex−array</td>
<td>NULL</td>
<td>glGetPointerv()</td>
</tr>
<tr>
<td>GL_INDEX_ARRAY</td>
<td>Color−index array enable</td>
<td>vertex−array</td>
<td>GL_FALSE</td>
<td>glIsEnabled()</td>
</tr>
<tr>
<td>GL_INDEX_ARRAY_TYPE</td>
<td>Type of color indices</td>
<td>vertex−array</td>
<td>GL_FLOAT</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_INDEX_ARRAY_STRIDE</td>
<td>Stride between color indices</td>
<td>vertex−array</td>
<td>0</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_INDEX_ARRAY_POINTER</td>
<td>Pointer to the index array</td>
<td>vertex−array</td>
<td>NULL</td>
<td>glGetPointerv()</td>
</tr>
<tr>
<td>GL_TEXTURE_COORD_ARRAY</td>
<td>Texture coordinate array enable</td>
<td>vertex−array</td>
<td>GL_FALSE</td>
<td>glIsEnabled()</td>
</tr>
<tr>
<td>GL_TEXTURE_COORD_ARRAY_SIZE</td>
<td>Texture coordinates per element</td>
<td>vertex−array</td>
<td>4</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_TEXTURE_COORD_ARRAY_TYPE</td>
<td>Type of texture coordinates</td>
<td>vertex−array</td>
<td>GL_FLOAT</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_TEXTURE_COORD_ARRAY_STRIDE</td>
<td>Stride between texture coordinates</td>
<td>vertex−array</td>
<td>0</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_TEXTURE_COORD_ARRAY_POINTER</td>
<td>Pointer to the texture coordinate array</td>
<td>vertex−array</td>
<td>NULL</td>
<td>glGetPointerv()</td>
</tr>
<tr>
<td>GL_EDGE_FLAG_ARRAY</td>
<td>Edge flag array enable</td>
<td>vertex−array</td>
<td>GL_FALSE</td>
<td>glIsEnabled()</td>
</tr>
<tr>
<td>GL_EDGE_FLAG_ARRAY_STRIDE</td>
<td>Stride between edge flags</td>
<td>vertex−array</td>
<td>0</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_EDGE_FLAG_ARRAY_POINTER</td>
<td>Pointer to the edge flag array</td>
<td>vertex−array</td>
<td>NULL</td>
<td>glGetPointerv()</td>
</tr>
</tbody>
</table>

Table B–2 (continued)  Vertex Array State Variables

### Transformation

<table>
<thead>
<tr>
<th>State Variable</th>
<th>Description</th>
<th>Attribute Group</th>
<th>Initial Value</th>
<th>Get Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_MODELVIEW_MATRIX</td>
<td>Modelview matrix stack</td>
<td>—</td>
<td>Identity</td>
<td>glGetFloatv()</td>
</tr>
<tr>
<td>GL_PROJECTION_MATRIX</td>
<td>Projection matrix stack</td>
<td>—</td>
<td>Identity</td>
<td>glGetFloatv()</td>
</tr>
<tr>
<td>GL_TEXTURE_MATRIX</td>
<td>Texture matrix stack</td>
<td>—</td>
<td>Identity</td>
<td>glGetFloatv()</td>
</tr>
<tr>
<td>GL_VIEWPORT</td>
<td>Viewport origin and extent</td>
<td>viewport</td>
<td>—</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_DEPTH_RANGE</td>
<td>Depth range near and far</td>
<td>viewport</td>
<td>0, 1</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_MODELVIEW_STACK_DEPTH</td>
<td>Modelview matrix stack pointer</td>
<td>—</td>
<td>1</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_PROJECTION_STACK_DEPTH</td>
<td>Projection matrix stack pointer</td>
<td>—</td>
<td>1</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_TEXTURE_STACK_DEPTH</td>
<td>Texture matrix stack pointer</td>
<td>—</td>
<td>1</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_MATRIX_MODE</td>
<td>Current matrix mode</td>
<td>transform</td>
<td>GL_MODELVIEW</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_NORMALIZE</td>
<td>Current normal normalization on/off</td>
<td>transform/enable</td>
<td>GL_FALSE</td>
<td>glIsEnabled()</td>
</tr>
<tr>
<td>GL_CLIP_PLANEi</td>
<td>User clipping plane</td>
<td>transform</td>
<td>0, 0, 0</td>
<td>glGetClipPlane()</td>
</tr>
</tbody>
</table>

OpenGL Programming Guide – Appendix B, State Variables – 3
### Table B–3 Transformation State Variables

#### Coloring

<table>
<thead>
<tr>
<th>State Variable</th>
<th>Description</th>
<th>Attribute Group</th>
<th>Initial Value</th>
<th>Get Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_FOG_COLOR</td>
<td>Fog color</td>
<td>fog</td>
<td>0, 0, 0, 0</td>
<td>glGetFloatv()</td>
</tr>
<tr>
<td>GL_FOG_INDEX</td>
<td>Fog index</td>
<td>fog</td>
<td>0</td>
<td>glGetFloatv()</td>
</tr>
<tr>
<td>GL_FOG_DENSITY</td>
<td>Exponential fog density</td>
<td>fog</td>
<td>1.0</td>
<td>glGetFloatv()</td>
</tr>
<tr>
<td>GL_FOG_START</td>
<td>Linear fog start</td>
<td>fog</td>
<td>0.0</td>
<td>glGetFloatv()</td>
</tr>
<tr>
<td>GL_FOG_END</td>
<td>Linear fog end</td>
<td>fog</td>
<td>1.0</td>
<td>glGetFloatv()</td>
</tr>
<tr>
<td>GL_FOG_MODE</td>
<td>Fog mode</td>
<td>fog</td>
<td>GL_EXP</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_FOG</td>
<td>True if fog enabled</td>
<td>fog/enable</td>
<td>GL_FALSE</td>
<td>glIsEnabled()</td>
</tr>
<tr>
<td>GL_SHADE_MODE</td>
<td>glShadeModel() setting</td>
<td>lighting</td>
<td>GL_SMOOTH</td>
<td>glGetIntegerv()</td>
</tr>
</tbody>
</table>

#### Table B–4 Coloring State Variables

**Lighting**

See also Table 5–1 and Table 5–3 for initial values.

<table>
<thead>
<tr>
<th>State Variable</th>
<th>Description</th>
<th>Attribute Group</th>
<th>Initial Value</th>
<th>Get Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_LIGHTING</td>
<td>True if lighting is enabled</td>
<td>lighting/e enable</td>
<td>GL_FALSE</td>
<td>glIsEnabled()</td>
</tr>
<tr>
<td>GL_COLOR_MATERIAL</td>
<td>True if color tracking is enabled</td>
<td>lighting</td>
<td>GL_FALSE</td>
<td>glIsEnabled()</td>
</tr>
<tr>
<td>GL_COLOR_MATERIAL_PARAMETER</td>
<td>Material properties tracking current color</td>
<td>lighting</td>
<td>GL_AMBIENT_AND_DIFFUSE</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_COLOR_MATERIAL_FACE</td>
<td>Face(s) affected by color tracking</td>
<td>lighting</td>
<td>GL_FRONT_AND_BACK</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_AMBIENT</td>
<td>Ambient material color</td>
<td>lighting</td>
<td>(0.2, 0.2, 0.2, 1.0)</td>
<td>glGetMaterialfv()</td>
</tr>
<tr>
<td>GL_DIFFUSE</td>
<td>Diffuse material color</td>
<td>lighting</td>
<td>(0.8, 0.8, 0.8, 1.0)</td>
<td>glGetMaterialfv()</td>
</tr>
<tr>
<td>GL_SPECULAR</td>
<td>Specular material color</td>
<td>lighting</td>
<td>(0.0, 0.0, 0.0, 1.0)</td>
<td>glGetMaterialfv()</td>
</tr>
<tr>
<td>GL_EMISSION</td>
<td>Emissive material color</td>
<td>lighting</td>
<td>(0.0, 0.0, 0.0, 1.0)</td>
<td>glGetMaterialfv()</td>
</tr>
<tr>
<td>GL_SHININESS</td>
<td>Specular exponent of material</td>
<td>lighting</td>
<td>0.0</td>
<td>glGetMaterialfv()</td>
</tr>
<tr>
<td>GL_LIGHT_MODEL_AMBIENT</td>
<td>Ambient scene color</td>
<td>lighting</td>
<td>(0.2, 0.2, 0.2, 1.0)</td>
<td>glGetFloatv()</td>
</tr>
<tr>
<td>GL_LIGHT_MODEL_LOCAL_VIEWER</td>
<td>Viewer is local</td>
<td>lighting</td>
<td>GL_FALSE</td>
<td>glGetBooleanv()</td>
</tr>
<tr>
<td>GL_LIGHT_MODEL_TWO_SIDE</td>
<td>Use two–sided lighting</td>
<td>lighting</td>
<td>GL_FALSE</td>
<td>glGetBooleanv()</td>
</tr>
<tr>
<td>GL_AMBIENT</td>
<td>Ambient intensity of light i</td>
<td>lighting</td>
<td>(0.0, 0.0, 0.0, 1.0)</td>
<td>glGetLightfv()</td>
</tr>
<tr>
<td>GL_DIFFUSE</td>
<td>Diffuse intensity of light i</td>
<td>lighting</td>
<td>—</td>
<td>glGetLightfv()</td>
</tr>
<tr>
<td>GL_SPECULAR</td>
<td>Specular intensity of light i</td>
<td>lighting</td>
<td>—</td>
<td>glGetLightfv()</td>
</tr>
<tr>
<td>GL_POSITION</td>
<td>Position of light i</td>
<td>lighting</td>
<td>(0.0, 0.0, 1.0, 0.0)</td>
<td>glGetLightfv()</td>
</tr>
<tr>
<td>GL_CONSTANT_ATTENUATION</td>
<td>Constant attenuation factor</td>
<td>lighting</td>
<td>1.0</td>
<td>glGetLightfv()</td>
</tr>
<tr>
<td>GL_LINEAR_ATTENUATION</td>
<td>Linear attenuation factor</td>
<td>lighting</td>
<td>0.0</td>
<td>glGetLightfv()</td>
</tr>
<tr>
<td>GL_QUADRATIC_ATTENUATION</td>
<td>Quadratic attenuation factor</td>
<td>lighting</td>
<td>0.0</td>
<td>glGetLightfv()</td>
</tr>
<tr>
<td>GL_SPOT_DIRECTION</td>
<td>Spotlight direction of light i</td>
<td>lighting</td>
<td>(0.0, 0.0, −1.0)</td>
<td>glGetLightfv()</td>
</tr>
</tbody>
</table>
### Table B–5 (continued) Lighting State Variables

<table>
<thead>
<tr>
<th>State Variable</th>
<th>Description</th>
<th>Attribute Group</th>
<th>Initial Value</th>
<th>Get Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_SPOT_EXPONENT</td>
<td>Spotlight exponent of light (i)</td>
<td>lighting</td>
<td>0.0</td>
<td>glGetUniformLocation()</td>
</tr>
<tr>
<td>GL_SPOT_CUTOFF</td>
<td>Spotlight angle of light (i)</td>
<td>lighting</td>
<td>180.0</td>
<td>glGetLightfv()</td>
</tr>
<tr>
<td>GL_LIGHT(i)</td>
<td>True if light (i) enabled</td>
<td>lighting/enable</td>
<td>GL_FALSE</td>
<td>glEnable()</td>
</tr>
<tr>
<td>GL_COLOR_INDEXES</td>
<td>ca, cd, and cs for color–index lighting</td>
<td>lighting/enable</td>
<td>0, 1, 1</td>
<td>glGetMaterialfv()</td>
</tr>
</tbody>
</table>

### Table B–6 (continued) Rasterization State Variables

<table>
<thead>
<tr>
<th>State Variable</th>
<th>Description</th>
<th>Attribute Group</th>
<th>Initial Value</th>
<th>Get Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_POINT_SIZE</td>
<td>Point size</td>
<td>point</td>
<td>1.0</td>
<td>glGetFloatv()</td>
</tr>
<tr>
<td>GL_POINT_SMOOTH</td>
<td>Point antialiasing on</td>
<td>point/enable</td>
<td>GL_FALSE</td>
<td>glEnable()</td>
</tr>
<tr>
<td>GL_LINE_WIDTH</td>
<td>Line width</td>
<td>line</td>
<td>1.0</td>
<td>glGetFloatv()</td>
</tr>
<tr>
<td>GL_LINE_SMOOTH</td>
<td>Line antialiasing on</td>
<td>line/enable</td>
<td>GL_FALSE</td>
<td>glEnable()</td>
</tr>
<tr>
<td>GL_LINE_STIPPLE_PATTERN</td>
<td>Line stipple pattern</td>
<td>line/enable</td>
<td>GL_FALSE</td>
<td>glEnable()</td>
</tr>
<tr>
<td>GL_LINE_STIPPLE_REPEAT</td>
<td>Line stipple repeat</td>
<td>line</td>
<td>1</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_LINE_STIPPLE</td>
<td>Line stipple enable</td>
<td>line/enable</td>
<td>GL_FALSE</td>
<td>glEnable()</td>
</tr>
<tr>
<td>GL_CULL_FACE</td>
<td>Polygon culling enabled</td>
<td>polygon/enable</td>
<td>GL_FALSE</td>
<td>glEnable()</td>
</tr>
<tr>
<td>GL_CULL_FACE_MODE</td>
<td>Cull front-/back-facing polygons</td>
<td>polygon</td>
<td>GL_BACK</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_FRONT_FACE</td>
<td>Polygon front–face CW/CCW indicator</td>
<td>polygon</td>
<td>GL_CCW</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_POLYGON_SMOOTH</td>
<td>Polygon antialiasing on</td>
<td>polygon/enable</td>
<td>GL_FALSE</td>
<td>glEnable()</td>
</tr>
<tr>
<td>GL_POLYGON_MODE</td>
<td>Polygon rasterization mode (front and back)</td>
<td>polygon</td>
<td>GL_FILL</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_POLYGON_OFFSET_FACTOR</td>
<td>Polygon offset factor</td>
<td>polygon</td>
<td>0</td>
<td>glGetFloatv()</td>
</tr>
<tr>
<td>GL_POLYGON_OFFSET_BIAS</td>
<td>Polygon offset bias</td>
<td>polygon</td>
<td>0</td>
<td>glGetFloatv()</td>
</tr>
<tr>
<td>GL_POLYGON_OFFSET_POINT</td>
<td>Polygon offset enable for GL_POINT mode rasterization</td>
<td>polygon/enable</td>
<td>GL_FALSE</td>
<td>glEnable()</td>
</tr>
<tr>
<td>GL_POLYGON_OFFSET_LINE</td>
<td>Polygon offset enable for GL_LINE mode rasterization</td>
<td>polygon/enable</td>
<td>GL_FALSE</td>
<td>glEnable()</td>
</tr>
<tr>
<td>GL_POLYGON_OFFSET_FILL</td>
<td>Polygon offset enable for GL_FILL mode rasterization</td>
<td>polygon/enable</td>
<td>GL_FALSE</td>
<td>glEnable()</td>
</tr>
<tr>
<td>GL_POLYGON_STIPPLE</td>
<td>Polygon stipple enable</td>
<td>polygon</td>
<td>GL_FALSE</td>
<td>glEnable()</td>
</tr>
<tr>
<td></td>
<td>Polygon stipple pattern</td>
<td>polygon–stipple</td>
<td>1’s</td>
<td>glGetPolygonStipple()</td>
</tr>
</tbody>
</table>

### Table B–6 (continued) Texturing State Variables

<table>
<thead>
<tr>
<th>State Variable</th>
<th>Description</th>
<th>Attribute Group</th>
<th>Initial Value</th>
<th>Get Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_TEXTURE_x</td>
<td>True if (x)-D texturing enabled ((x) is 1D or 2D)</td>
<td>texture/enable</td>
<td>GL_FALSE</td>
<td>glEnable()</td>
</tr>
<tr>
<td>GL_TEXTURE_BINDING_x</td>
<td>Texture object bound to (x)-D texturing enabled ((x) is 1D or 2D)</td>
<td>texture</td>
<td>GL_FALSE</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_TEXTURE</td>
<td>(x)-D texture image at level of detail</td>
<td>GL_FALSE</td>
<td>glGetTexImage()</td>
<td></td>
</tr>
<tr>
<td>GL_TEXTURE_WIDTH</td>
<td>(x)-D texture image(i)'s width</td>
<td>0</td>
<td>glGetTexLevelParameter*()</td>
<td></td>
</tr>
<tr>
<td>GL_TEXTURE_HEIGHT</td>
<td>(x)-D texture image(i)'s height</td>
<td>0</td>
<td>glGetTexLevelParameter*()</td>
<td></td>
</tr>
<tr>
<td>State Variable</td>
<td>Description</td>
<td>Attribute Group</td>
<td>Initial Value</td>
<td>Get Command</td>
</tr>
<tr>
<td>------------------------------</td>
<td>------------------------</td>
<td>-----------------</td>
<td>---------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>GL_TEXTURE_BORDER</td>
<td>x-D texture image (i)’s border width</td>
<td>—</td>
<td>0</td>
<td>glGetTexLevelParameter*()</td>
</tr>
<tr>
<td>GL_TEXTURE_INTERNAL_FORMAT</td>
<td>x-D texture image (i)’s internal image format</td>
<td>—</td>
<td>1</td>
<td>glGetTexLevelParameter*()</td>
</tr>
<tr>
<td>GL_TEXTURE_RED_SIZE</td>
<td>x-D texture image (i)’s red resolution</td>
<td>—</td>
<td>0</td>
<td>glGetTexLevelParameter*()</td>
</tr>
<tr>
<td>GL_TEXTURE_GREEN_SIZE</td>
<td>x-D texture image (i)’s green resolution</td>
<td>—</td>
<td>0</td>
<td>glGetTexLevelParameter*()</td>
</tr>
<tr>
<td>GL_TEXTURE_BLUE_SIZE</td>
<td>x-D texture image (i)’s blue resolution</td>
<td>—</td>
<td>0</td>
<td>glGetTexLevelParameter*()</td>
</tr>
<tr>
<td>GL_TEXTURE_ALPHA_SIZE</td>
<td>x-D texture image (i)’s alpha resolution</td>
<td>—</td>
<td>0</td>
<td>glGetTexLevelParameter*()</td>
</tr>
<tr>
<td>GL_TEXTURE_LUMINANCE_SIZE</td>
<td>x-D texture image (i)’s luminance resolution</td>
<td>—</td>
<td>0</td>
<td>glGetTexLevelParameter*()</td>
</tr>
<tr>
<td>GL_TEXTURE_INTENSITY_SIZE</td>
<td>x-D texture image (i)’s intensity resolution</td>
<td>—</td>
<td>0</td>
<td>glGetTexLevelParameter*()</td>
</tr>
<tr>
<td>GL_TEXTURE_BORDER_COLOR</td>
<td>Texture border color</td>
<td>texture</td>
<td>0, 0, 0, 0</td>
<td>glGetTexParameter*()</td>
</tr>
<tr>
<td>GL_TEXTURE_MIN_FILTER</td>
<td>Texture minification function</td>
<td>texture</td>
<td>GL_LINEAR</td>
<td>glGetTexParameter*()</td>
</tr>
<tr>
<td>GL_TEXTURE_MAG_FILTER</td>
<td>Texture magnification function</td>
<td>texture</td>
<td>GL_LINEAR</td>
<td>glGetTexParameter*()</td>
</tr>
<tr>
<td>GL_TEXTURE_WRAP_x</td>
<td>Texture wrap mode ((x) is S or T)</td>
<td>texture</td>
<td>GL_REPEAT</td>
<td>glGetTexParameter*()</td>
</tr>
<tr>
<td>GL_TEXTURE_PRIORITY</td>
<td>Texture object priority</td>
<td>texture</td>
<td>1</td>
<td>glGetTexParameter*()</td>
</tr>
<tr>
<td>GL_TEXTURE_RESIDENCY</td>
<td>Texture residency</td>
<td>texture</td>
<td>GL_FALSE</td>
<td>glGetTexParameter*()</td>
</tr>
<tr>
<td>GL_TEXTURE_ENV_MODE</td>
<td>Texture application function</td>
<td>texture</td>
<td>GL_MODULATE</td>
<td>glGetTexEnviv()</td>
</tr>
<tr>
<td>GL_TEXTURE_ENV_COLOR</td>
<td>Texture environment color</td>
<td>texture</td>
<td>0, 0, 0, 0</td>
<td>glGetTexEnvfv()</td>
</tr>
<tr>
<td>GL_TEXTURE_GEN_x</td>
<td>Texgen enabled ((x) is S, T, R, or Q)</td>
<td>texture/eenable</td>
<td>GL_FALSE</td>
<td>glIsEnabled()</td>
</tr>
<tr>
<td>GL_EYE_PLANE</td>
<td>Texgen plane equation coefficients</td>
<td>texture</td>
<td>—</td>
<td>glGetTexGenfv()</td>
</tr>
<tr>
<td>GL_OBJECT_PLANE</td>
<td>Texgen object linear coefficients</td>
<td>texture</td>
<td>—</td>
<td>glGetTexGenfv()</td>
</tr>
<tr>
<td>GL_TEXTURE_GEN_MODE</td>
<td>Function used for texgen</td>
<td>texture</td>
<td>GL_EYE_LINEAR</td>
<td>glGetTexGenfv()</td>
</tr>
</tbody>
</table>

Table B–7 (continued) Texturing State Variables

Pixel Operations

<table>
<thead>
<tr>
<th>State Variable</th>
<th>Description</th>
<th>Attribute Group</th>
<th>Initial Value</th>
<th>Get Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_SCISSOR_TEST</td>
<td>Scissoring enabled</td>
<td>scissor/enable</td>
<td>GL_FALSE</td>
<td>glIsEnabled()</td>
</tr>
<tr>
<td>GL_SCISSOR_BOX</td>
<td>Scissor box</td>
<td>scissor</td>
<td>—</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_ALPHA_TEST</td>
<td>Alpha test enabled</td>
<td>color−buffer/enable</td>
<td>GL_FALSE</td>
<td>glIsEnabled()</td>
</tr>
<tr>
<td>GL_ALPHA_TEST_FUNC</td>
<td>Alpha test function</td>
<td>color−buffer</td>
<td>GL_ALWAYS</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_ALPHA_TEST_REF</td>
<td>Alpha test value</td>
<td>color−buffer</td>
<td>0</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_STENCIL_TEST</td>
<td>Stenciling enabled</td>
<td>stencil−buffer/enable</td>
<td>GL_FALSE</td>
<td>glIsEnabled()</td>
</tr>
<tr>
<td>GL_STENCIL_FUNC</td>
<td>Stencil function</td>
<td>stencil−buffer</td>
<td>GL_ALWAYS</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_STENCIL_VALUE_MASK</td>
<td>Stencil mask</td>
<td>stencil−buffer</td>
<td>1’s</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_STENCIL_REF</td>
<td>Stencil reference value</td>
<td>stencil−buffer</td>
<td>0</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_STENCIL_FAIL</td>
<td>Stencil fail action</td>
<td>stencil−buffer</td>
<td>GL_KEEP</td>
<td>glGetIntegerv()</td>
</tr>
</tbody>
</table>
Table B–8 (continued)  Pixel Operations

<table>
<thead>
<tr>
<th>State Variable</th>
<th>Description</th>
<th>Attribute Group</th>
<th>Initial Value</th>
<th>Get Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_DRAW_BUFFER</td>
<td>Buffers selected for drawing</td>
<td>color-buffer</td>
<td></td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_INDEX_WRITEMASK</td>
<td>Color-index writemask</td>
<td>color-buffer</td>
<td>1’s</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_COLOR_WRITEMASK</td>
<td>Color write enables; R, G, B, or A</td>
<td>color-buffer</td>
<td>GL_TRUE</td>
<td>glGetBooleanv()</td>
</tr>
<tr>
<td>GL_DEPTH_WRITEMASK</td>
<td>Depth buffer enabled for writing</td>
<td>depth-buffer</td>
<td>GL_TRUE</td>
<td>glGetBooleanv()</td>
</tr>
<tr>
<td>GL_STENCIL_WRITEMASK</td>
<td>Stencil-buffer writemask</td>
<td>stencil-buffer</td>
<td>1’s</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_COLOR_CLEAR_VALUE</td>
<td>Color-buffer clear value (RGBA mode)</td>
<td>color-buffer</td>
<td>0, 0, 0, 0</td>
<td>glGetFloatv()</td>
</tr>
<tr>
<td>GL_INDEX_CLEAR_VALUE</td>
<td>Color-buffer clear value (color-index mode)</td>
<td>color-buffer</td>
<td>0</td>
<td>glGetFloatv()</td>
</tr>
<tr>
<td>GL_DEPTH_CLEAR_VALUE</td>
<td>Depth-buffer clear value</td>
<td>depth-buffer</td>
<td>1</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_STENCIL_CLEAR_VALUE</td>
<td>Stencil-buffer clear value</td>
<td>stencil-buffer</td>
<td>0</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_ACCUM_CLEAR_VALUE</td>
<td>Accumulation-buffer clear value</td>
<td>accum-buffer</td>
<td>0</td>
<td>glGetFloatv()</td>
</tr>
</tbody>
</table>

Table B–9  Framebuffer Control State Variables

<table>
<thead>
<tr>
<th>State Variable</th>
<th>Description</th>
<th>Attribute Group</th>
<th>Initial Value</th>
<th>Get Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_UNPACK_SWAP_BYTES</td>
<td>Value of GL_UNPACK_SWAP_BYTES</td>
<td>pixel-store</td>
<td>GL_FALSE</td>
<td>glGetBooleanv()</td>
</tr>
<tr>
<td>GL_UNPACK_LSB_FIRST</td>
<td>Value of GL_UNPACK_LSB_FIRST</td>
<td>pixel-store</td>
<td>GL_FALSE</td>
<td>glGetBooleanv()</td>
</tr>
<tr>
<td>GL_UNPACK_ROW_LENGTH</td>
<td>Value of GL_UNPACK_ROW_LENGTH</td>
<td>pixel-store</td>
<td>0</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_UNPACK_SKIP_ROWS</td>
<td>Value of GL_UNPACK_SKIP_ROWS</td>
<td>pixel-store</td>
<td>0</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_UNPACK_SKIP_PIXELS</td>
<td>Value of GL_UNPACK_SKIP_PIXELS</td>
<td>pixel-store</td>
<td>0</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_UNPACK_ALIGNMENT</td>
<td>Value of GL_UNPACK_ALIGNMENT</td>
<td>pixel-store</td>
<td>4</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_PACK_SWAP_BYTES</td>
<td>Value of GL_PACK_SWAP_BYTES</td>
<td>pixel-store</td>
<td>GL_FALSE</td>
<td>glGetBooleanv()</td>
</tr>
<tr>
<td>GL_PACK_LSB_FIRST</td>
<td>Value of GL_PACK_LSB_FIRST</td>
<td>pixel-store</td>
<td>GL_FALSE</td>
<td>glGetBooleanv()</td>
</tr>
<tr>
<td>GL_PACK_ROW_LENGTH</td>
<td>Value of GL_PACK_ROW_LENGTH</td>
<td>pixel-store</td>
<td>0</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_PACK_SKIP_ROWS</td>
<td>Value of GL_PACK_SKIP_ROWS</td>
<td>pixel-store</td>
<td>0</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_PACK_SKIP_PIXELS</td>
<td>Value of GL_PACK_SKIP_PIXELS</td>
<td>pixel-store</td>
<td>0</td>
<td>glGetIntegerv()</td>
</tr>
</tbody>
</table>
Table B–10 (continued) Pixel State Variables

<table>
<thead>
<tr>
<th>State Variable</th>
<th>Description</th>
<th>Attribute Group</th>
<th>Initial Value</th>
<th>Get Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_ORDER</td>
<td>1D map order</td>
<td>—</td>
<td>1</td>
<td>glGetMapiv()</td>
</tr>
<tr>
<td>GL_ORDER</td>
<td>2D map orders</td>
<td>—</td>
<td>1, 1</td>
<td>glGetMapiv()</td>
</tr>
<tr>
<td>GL_COEFF</td>
<td>1D control points</td>
<td>—</td>
<td>—</td>
<td>glGetMapfv()</td>
</tr>
<tr>
<td>GL_COEFF</td>
<td>2D control points</td>
<td>—</td>
<td>—</td>
<td>glGetMapfv()</td>
</tr>
<tr>
<td>GLDOMAIN</td>
<td>1D domain endpoints</td>
<td>—</td>
<td>—</td>
<td>glGetMapfv()</td>
</tr>
<tr>
<td>GLDOMAIN</td>
<td>2D domain endpoints</td>
<td>—</td>
<td>—</td>
<td>glGetMapfv()</td>
</tr>
<tr>
<td>GLMAP1_x</td>
<td>1D map enables: x is map type</td>
<td>eval/enable</td>
<td>GL_FALSE</td>
<td>glIsEnabled()</td>
</tr>
<tr>
<td>GLMAP2_x</td>
<td>2D map enables: x is map type</td>
<td>eval/enable</td>
<td>GL_FALSE</td>
<td>glIsEnabled()</td>
</tr>
<tr>
<td>GLMAP1_GRID_DOMAIN</td>
<td>1D grid endpoints</td>
<td>eval</td>
<td>0, 1</td>
<td>glGetFloatv()</td>
</tr>
<tr>
<td>GLMAP2_GRID_DOMAIN</td>
<td>2D grid endpoints</td>
<td>eval</td>
<td>0, 1; 0, 1</td>
<td>glGetFloatv()</td>
</tr>
<tr>
<td>GLMAP1_GRID_SEGMENTS</td>
<td>1D grid divisions</td>
<td>eval</td>
<td>1</td>
<td>glGetFloatv()</td>
</tr>
<tr>
<td>GLMAP2_GRID_SEGMENTS</td>
<td>2D grid divisions</td>
<td>eval</td>
<td>1, 1</td>
<td>glGetFloatv()</td>
</tr>
<tr>
<td>GLAUTO_NORMAL</td>
<td>True if automatic normal generation enabled</td>
<td>eval</td>
<td>GL_FALSE</td>
<td>glIsEnabled()</td>
</tr>
</tbody>
</table>

Table B–11 Evaluator State Variables

<table>
<thead>
<tr>
<th>State Variable</th>
<th>Description</th>
<th>Attribute Group</th>
<th>Initial Value</th>
<th>Get Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_PERSPECTIVE_CORRECTION_HINT</td>
<td>Perspective correction hint</td>
<td>hint</td>
<td>GL_DONT_CARE</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_POINT_SMOOTH_HINT</td>
<td>Point smooth hint</td>
<td>hint</td>
<td>GL_DONT_CARE</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_LINE_SMOOTH_HINT</td>
<td>Line smooth hint</td>
<td>hint</td>
<td>GL_DONT_CARE</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_POLYGON_SMOOTH_HINT</td>
<td>Polygon smooth hint</td>
<td>hint</td>
<td>GL_DONT_CARE</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_FOG_HINT</td>
<td>Fog hint</td>
<td>hint</td>
<td>GL_DONT_CARE</td>
<td>glGetIntegerv()</td>
</tr>
</tbody>
</table>
### Table B–12: Hint State Variables

<table>
<thead>
<tr>
<th>State Variable</th>
<th>Description</th>
<th>Attribute Group</th>
<th>Minimum Value</th>
<th>Get Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_MAX_LIGHTS</td>
<td>Maximum number of lights</td>
<td>—</td>
<td>8</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_MAX_CLIP_PLANES</td>
<td>Maximum number of user clipping planes</td>
<td>—</td>
<td>6</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_MAX_MODELVIEW_STACK_DEPTH</td>
<td>Maximum modelview–matrix stack depth</td>
<td>—</td>
<td>32</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_MAX_PROJECTION_STACK_DEPTH</td>
<td>Maximum projection–matrix stack depth</td>
<td>—</td>
<td>2</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_MAX_TEXTURE_STACK_DEPTH</td>
<td>Maximum depth of texture matrix stack</td>
<td>—</td>
<td>2</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_SUBPIXEL_BITS</td>
<td>Number of bits of subpixel precision in x and y</td>
<td>—</td>
<td>4</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_MAX_TEXTURE_SIZE</td>
<td>See discussion in &quot;Texture Proxy&quot; in Chapter 9</td>
<td>—</td>
<td>64</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_MAX_PIXEL_MAP_TABLE</td>
<td>Maximum size of a glPixelMap() translation table</td>
<td>—</td>
<td>32</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_MAX_NAME_STACK_DEPTH</td>
<td>Maximum selection–name stack depth</td>
<td>—</td>
<td>64</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_MAX_LIST_NESTING</td>
<td>Maximum display–list call nesting</td>
<td>—</td>
<td>64</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_MAX_EVAL_ORDER</td>
<td>Maximum evaluator polynomial order</td>
<td>—</td>
<td>8</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_MAX_VIEWPORT_DIMS</td>
<td>Maximum viewport dimensions</td>
<td>—</td>
<td>—</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_MAX_ATTRIB_STACK_DEPTH</td>
<td>Maximum depth of the attribute stack</td>
<td>—</td>
<td>16</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_MAX_CLIENT_ATTRIB_STACK_DEPTH</td>
<td>Maximum depth of the client attribute stack</td>
<td>—</td>
<td>16</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_AUX_BUFFERS</td>
<td>Number of auxiliary buffers</td>
<td>—</td>
<td>0</td>
<td>glGetBooleanv()</td>
</tr>
<tr>
<td>GL_RGBA_MODE</td>
<td>True if color buffers store RGBA</td>
<td>—</td>
<td>—</td>
<td>glGetBooleanv()</td>
</tr>
<tr>
<td>GL_INDEX_MODE</td>
<td>True if color buffers store indices</td>
<td>—</td>
<td>—</td>
<td>glGetBooleanv()</td>
</tr>
<tr>
<td>GL_DOUBLEBUFFER</td>
<td>True if front and back buffers exist</td>
<td>—</td>
<td>—</td>
<td>glGetBooleanv()</td>
</tr>
<tr>
<td>GL_STEREO</td>
<td>True if left and right buffers exist</td>
<td>—</td>
<td>—</td>
<td>glGetBooleanv()</td>
</tr>
<tr>
<td>GL_POINT_SIZE_RANGE</td>
<td>Range (low to high) of antialiased point sizes</td>
<td>—</td>
<td>1, 1</td>
<td>glGetFloatv()</td>
</tr>
<tr>
<td>GL_POINT_SIZE_GRANULARITY</td>
<td>Antialiased point–size granularity</td>
<td>—</td>
<td>—</td>
<td>glGetFloatv()</td>
</tr>
<tr>
<td>GL_LINE_WIDTH_RANGE</td>
<td>Range (low to high) of antialiased line widths</td>
<td>—</td>
<td>1, 1</td>
<td>glGetFloatv()</td>
</tr>
<tr>
<td>GL_LINE_WIDTH_GRANULARITY</td>
<td>Antialiased line–width granularity</td>
<td>—</td>
<td>—</td>
<td>glGetFloatv()</td>
</tr>
</tbody>
</table>

### Table B–13 (continued): Implementation–Dependent State Variables

<table>
<thead>
<tr>
<th>State Variable</th>
<th>Description</th>
<th>Attribute Group</th>
<th>Minimum Value</th>
<th>Get Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_RED_BITS</td>
<td>Number of bits per red component in color buffers</td>
<td>—</td>
<td>—</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_GREEN_BITS</td>
<td>Number of bits per green component in color buffers</td>
<td>—</td>
<td>—</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_BLUE_BITS</td>
<td>Number of bits per blue component in color buffers</td>
<td>—</td>
<td>—</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_ALPHA_BITS</td>
<td>Number of bits per alpha component in color buffers</td>
<td>—</td>
<td>—</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_INDEX_BITS</td>
<td>Number of bits per index in color buffers</td>
<td>—</td>
<td>—</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_DEPTH_BITS</td>
<td>Number of depth–buffer bitplanes</td>
<td>—</td>
<td>—</td>
<td>glGetIntegerv()</td>
</tr>
</tbody>
</table>
- GL_STENCIL_BITS: Number of stencil bitplanes
  - glGetIntegerv()
- GL_ACCUM_RED_BITS: Number of bits per red component in the accumulation buffer
  - glGetIntegerv()
- GL_ACCUM_GREEN_BITS: Number of bits per green component in the accumulation buffer
  - glGetIntegerv()
- GL_ACCUM_BLUE_BITS: Number of bits per blue component in the accumulation buffer
  - glGetIntegerv()
- GL_ACCUM_ALPHA_BITS: Number of bits per alpha component in the accumulation buffer
  - glGetIntegerv()

**Table B–14 Implementation–Dependent Pixel–Depth State Variables (continued)**

<table>
<thead>
<tr>
<th>State Variable</th>
<th>Description</th>
<th>Attribute Group</th>
<th>Initial Value</th>
<th>Get Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_LIST_BASE</td>
<td>Setting of glListBase()</td>
<td>list</td>
<td>0</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_LIST_INDEX</td>
<td>Number of display list under construction; 0 if none</td>
<td></td>
<td>0</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_LIST_MODE</td>
<td>Mode of display list under construction; undefined if none</td>
<td></td>
<td>0</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_ATTRIB_STACK_DEPTH</td>
<td>Attribute stack pointer</td>
<td></td>
<td>0</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_CLIENT_ATTRIB_STACK_DEPTH</td>
<td>Client attribute stack pointer</td>
<td></td>
<td>0</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_NAME_STACK_DEPTH</td>
<td>Name stack depth</td>
<td></td>
<td>0</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_RENDER_MODE</td>
<td>glRenderMode() setting</td>
<td></td>
<td></td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_SELECTION_BUFFER_POINTER</td>
<td>Pointer to selection buffer</td>
<td>select</td>
<td>0</td>
<td>glGetPointerv()</td>
</tr>
<tr>
<td>GL_SELECTION_BUFFER_SIZE</td>
<td>Size of selection buffer</td>
<td>select</td>
<td>0</td>
<td>glGetPointerv()</td>
</tr>
<tr>
<td>GL_FEEDBACK_BUFFER_POINTER</td>
<td>Pointer to feedback buffer</td>
<td>feedback</td>
<td>0</td>
<td>glGetPointerv()</td>
</tr>
<tr>
<td>GL_FEEDBACK_BUFFER_SIZE</td>
<td>Size of feedback buffer</td>
<td>feedback</td>
<td>0</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td>GL_FEEDBACK_BUFFER_TYPE</td>
<td>Type of feedback buffer</td>
<td>feedback</td>
<td>GL_2D</td>
<td>glGetIntegerv()</td>
</tr>
<tr>
<td></td>
<td>Current error code(s)</td>
<td></td>
<td>0</td>
<td>glGetError()</td>
</tr>
</tbody>
</table>

**Table B–15 Miscellaneous State Variables**
Appendix C
OpenGL and Window Systems

OpenGL is available on many different platforms and works with many different window systems. OpenGL is designed to complement window systems, not duplicate their functionality. Therefore, OpenGL performs geometric and image rendering in two and three dimensions, but it does not manage windows or handle input events.

However, the basic definitions of most window systems don’t support a library as sophisticated as OpenGL, with its complex and diverse pixel formats, including depth, stencil, and accumulation buffers, as well as double-buffering. For most window systems, some routines are added to extend the window system to support OpenGL.

This appendix introduces the extensions defined for several window and operating systems: the X Window System, the Apple Mac OS, OS/2 Warp from IBM, and Microsoft Windows NT and Windows 95. You need to have some knowledge of the window systems to fully understand this appendix.

This appendix has the following major sections:

"GLX: OpenGL Extension for the X Window System"
"AGL: OpenGL Extension to the Apple Macintosh"
"PGL: OpenGL Extension for IBM OS/2 Warp"
"WGL: OpenGL Extension for Microsoft Windows NT and Windows 95"

GLX: OpenGL Extension for the X Window System

In the X Window System, OpenGL rendering is made available as an extension to X in the formal X sense. GLX is an extension to the X protocol (and its associated API) for communicating OpenGL commands to an extended X server. Connection and authentication are accomplished with the normal X mechanisms.

As with other X extensions, there is a defined network protocol for OpenGL’s rendering commands encapsulated within the X byte stream, so client–server OpenGL rendering is supported. Since performance is critical in three–dimensional rendering, the OpenGL extension to X allows OpenGL to bypass the X server’s involvement in data encoding, copying, and interpretation and instead render directly to the graphics pipeline.

The X Visual is the key data structure to maintain pixel format information about the OpenGL window. A variable of data type XVisualInfo keeps track of pixel information, including pixel type (RGBA or color index), single or double–buffering, resolution of colors, and presence of depth, stencil, and accumulation buffers. The standard X Visuals (for example, PseudoColor, TrueColor) do not describe the pixel format details, so each implementation must extend the number of X Visuals supported.

The GLX routines are discussed in more detail in the OpenGL Reference Manual. Integrating OpenGL applications with the X Window System and the Motif widget set is discussed in great detail in OpenGL Programming for the X Window System by Mark Kilgard (Reading, MA: Addison–Wesley Developers)
Press, 1996), which includes full source code examples. If you absolutely want to learn about the internals of GLX, you may want to read the GLX specification, which can be found at
ftp://sgigate.sgi.com/pub/opengl/doc/

Initialization

Use glXQueryExtension() and glXQueryVersion() to determine whether the GLX extension is defined for an X server and, if so, which version is present. glXQueryExtensionsString() returns extension information about the client–server connection, glXGetClientString() returns information about the client library, including extensions and version number. glXQueryServerString() returns similar information about the server.

glXChooseVisual() returns a pointer to an XVisualInfo structure describing the visual that meets the client’s specified attributes. You can query a visual about its support of a particular OpenGL attribute with glXGetConfig().

Controlling Rendering

Several GLX routines are provided for creating and managing an OpenGL rendering context. You can use such a context to render off-screen if you want. Routines are also provided for such tasks as synchronizing execution between the X and OpenGL streams, swapping front and back buffers, and using an X font.

Managing an OpenGL Rendering Context

An OpenGL rendering context is created with glXCreateContext(). One of the arguments to this routine allows you to request a direct rendering context that bypasses the X server as described previously. (Note that to do direct rendering, the X server connection must be local, and the OpenGL implementation needs to support direct rendering.) glXCreateContext() also allows display–list and texture–object indices and definitions to be shared by multiple rendering contexts. You can determine whether a GLX context is direct with glXIsDirect().

To make a rendering context current, use glXMakeCurrent(); glXGetCurrentContext() returns the current context. You can also obtain the current drawable with glXGetCurrentDrawable() and the current X Display with glXGetCurrentDisplay(). Remember that only one context can be current for any thread at any one time. If you have multiple contexts, you can copy selected groups of OpenGL state variables from one context to another with glXCopyContext(). When you’re finished with a particular context, destroy it with glXDestroyContext().

Off-Screen Rendering

To render off-screen, first create an X Pixmap and then pass this as an argument to glXCreateGLXPixmap(). Once rendering is completed, you can destroy the association between the X and GLX Pixmaps with glXDestroyGLXPixmap(). (Off-screen rendering isn’t guaranteed to be supported for direct renderers.)

Synchronizing Execution
To prevent X requests from executing until any outstanding OpenGL rendering is completed, call glXWaitGL(). Then, any previously issued OpenGL commands are guaranteed to be executed before any X rendering calls made after glXWaitGL(). Although the same result can be achieved with glFinish(), glXWaitGL() doesn’t require a round trip to the server and thus is more efficient in cases where the client and server are on separate machines.

To prevent an OpenGL command sequence from executing until any outstanding X requests are completed, use glXWaitX(). This routine guarantees that previously issued X rendering calls are executed before any OpenGL calls made after glXWaitX().

**Swapping Buffers**

For drawables that are double-buffered, the front and back buffers can be exchanged by calling glXSwapBuffers(). An implicit glFlush() is done as part of this routine.

**Using an X Font**

A shortcut for using X fonts in OpenGL is provided with the command glXUseXFont(). This routine builds display lists, each of which calls glBitmap(), for each requested character from the specified font and font size.

**GLX Prototypes**

**Initialization**

Determine whether the GLX extension is defined on the X server:

```c
Bool glXQueryExtension ( Display *dpy, int *errorBase, int *eventBase );
```

Query version and extension information for client and server:

```c
Bool glXQueryVersion ( Display *dpy, int *major, int *minor );
```

```c
const char* glXGetClientString ( Display *dpy, int name );
```

```c
const char* glXQueryServerString ( Display *dpy, int screen, int name );
```

```c
const char* glXQueryExtensionsString ( Display *dpy, int screen );
```

Obtain the desired visual:

```c
XVisualInfo* glXChooseVisual ( Display *dpy, int screen, int *attribList );
```

```c
int glXGetConfig ( Display *dpy, XVisualInfo *visual, int attrib, int *value );
```

**Controlling Rendering**

Manage or query an OpenGL rendering context:

```c
GLXContext glXCreateContext ( Display *dpy, XVisualInfo *visual, ...
```
GLXContext shareList, Bool direct);
void glXDestroyContext ( Display *dpy, GLXContext context );
void glXCopyContext ( Display *dpy, GLXContext source, 
GLXContext dest, unsigned long mask );
Bool glXIsDirect ( Display *dpy, GLXContext context );
Bool glXMakeCurrent ( Display *dpy, GLXDrawable draw, 
GLXContext context );
GLXContext glXGetCurrentContext (void);
Display* glXGetCurrentDisplay (void);
GLXDrawable glXGetCurrentDrawable (void);

Perform off−screen rendering:
GLXPixmap glXCreateGLXPixmap ( Display *dpy, XVisualInfo *visual, 
Pixmap pixmap );
void glXDestroyGLXPixmap ( Display *dpy, GLXPixmap pix );

Synchronize execution:
void glXWaitGL (void);
void glXWaitX (void);

Exchange front and back buffers:
void glXSwapBuffers ( Display *dpy, GLXDrawable drawable );

Use an X font:
void glXUseXFont ( Font font, int first, int count, int listBase );

AGL: OpenGL Extension to the Apple Macintosh

This section covers the routines defined as the OpenGL extension to the Apple Macintosh (AGL), as defined by Template Graphics Software. An understanding of the way the Macintosh handles graphics rendering (QuickDraw) is required. The Macintosh Toolbox Essentials and Imaging With QuickDraw manuals from the Inside Macintosh series are also useful to have at hand.

For more information (including how to obtain the OpenGL software library for the Power Macintosh), you may want to check out the web site for OpenGL information at Template Graphics Software:
http://www.sd.tgs.com/Products/opengl.htm

For the Macintosh, OpenGL rendering is made available as a library that is either compiled in or resident as an extension for an application that wishes to make use of it. OpenGL is implemented in software for systems that do not possess hardware acceleration. Where acceleration is available (through the QuickDraw 3D Accelerator), those capabilities that match the OpenGL pipeline are used with the remaining functionality being provided through software rendering.
The data type AGLPixelFmtID (the AGL equivalent to XVisualInfo) maintains pixel information, including pixel type (RGBA or color index), single- or double-buffering, resolution of colors, and presence of depth, stencil, and accumulation buffers.

In contrast to other OpenGL implementations on other systems (such as the X Window System), the client/server model is not used. However, you may still need to call glFlush() since some hardware accelerators buffer the OpenGL pipeline and require a flush to empty it.

**Initialization**

Use aglQueryVersion() to determine what version of OpenGL for the Macintosh is available.

The capabilities of underlying graphics devices and your requirements for rendering buffers are resolved using aglChoosePixelFormat(). Use aglListPixelFmts() to find the particular formats supported by a graphics device. Given a pixel format, you can determine which attributes are available by using aglGetConfig().

**Rendering and Contexts**

Several AGL routines are provided for creating and managing an OpenGL rendering context. You can use such a context to render into either a window or an off-screen graphics world. Routines are also provided that allow you to swap front and back rendering buffers, adjust buffers in response to a move, resize or graphics device change event, and use Macintosh fonts. For software rendering (and in some cases, hardware-accelerated rendering) the rendering buffers are created in your application memory space. For the application to work properly you must provide sufficient memory for these buffers in your application’s SIZE resource.

**Managing an OpenGL Rendering Context**

An OpenGL rendering context is created (at least one context per window being rendered into) with aglCreateContext(). This takes the pixel format you selected as a parameter and uses it to initialize the context.

Use aglMakeCurrent() to make a rendering context current. Only one context can be current for a thread of control at any time. This indicates which drawable is to be rendered into and which context to use with it. It's possible for more than one context to be used (not simultaneously) with a particular drawable.

Two routines allow you to determine which is the current rendering context and drawable being rendered into: aglGetCurrentContext() and aglGetCurrentDrawable().

If you have multiple contexts, you can copy selected groups of OpenGL state variables from one context to another with aglCopyContext(). When a particular context is finished with, it should be destroyed by calling aglDestroyContext().

**On-screen Rendering**

With the OpenGL extensions for the Apple Macintosh you can choose whether window clipping is performed when writing to the screen and whether the cursor is hidden during screen writing operations. This is important since these two items may affect how fast rendering can be performed. Call
aglSetOptions() to select these options.

Off–screen Rendering
To render off–screen, first create an off–screen graphics world in the usual way, and pass the handle into aglCreateAGLPixmap(). This routine returns a drawable that can be used with aglMakeCurrent(). Once rendering is completed, you can destroy the association with aglDestroyAGLPixmap().

Swapping Buffers
For drawables that are double–buffered (as per the pixel format of the current rendering context), call aglSwapBuffers() to exchange the front and back buffers. An implicit glFlush() is performed as part of this routine.

Updating the Rendering Buffers
The Apple Macintosh toolbox requires you to perform your own event handling and does not provide a way for libraries to automatically hook in to the event stream. So that the drawables maintained by OpenGL can adjust to changes in drawable size, position and pixel depth, aglUpdateCurrent() is provided.

This routine must be called by your event processing code whenever one of these events occurs in the current drawable. Ideally the scene should be rerendered after a update call to take into account the changes made to the rendering buffers.

Using an Apple Macintosh Font
A shortcut for using Macintosh fonts is provided with aglUseFont(). This routine builds display lists, each of which calls glBitmap(), for each requested character from the specified font and font size.

Error Handling
An error–handling mechanism is provided for the Apple Macintosh OpenGL extension. When an error occurs you can call aglGetError() to get a more precise description of what caused the error.

AGL Prototypes

Initialization
Determine AGL version:

GLboolean aglQueryVersion ( int *major, int *minor );

Pixel format selection, availability, and capability:

AGLPixelFmtID aglChoosePixelFmt ( GDHandle *dev, int ndev, int *attribs );

int aglListPixelFmts ( GDHandle dev, AGLPixelFmtID **fmts );
GLboolean aglGetConfig ( AGLPixelFmtID pix, int attrib, int *value );

**Controlling Rendering**

Manage an OpenGL rendering context:

AGLContext aglCreateContext ( AGLPixelFmtID pix, AGLContext shareList );

GLboolean aglDestroyContext ( AGLContext context );

GLboolean aglCopyContext ( AGLContext source, AGLContext dest, GLuint mask );

GLboolean aglMakeCurrent ( AGLDrawable drawable, AGLContext context );

GLboolean aglSetOptions ( int opts );

AGLContext aglGetCurrentContext (void);

AGLDrawable aglGetCurrentDrawable (void);

Perform off-screen rendering:

AGLPixmap aglCreateAGLPixmap ( AGLPixelFmtID pix, GWorldPtr pixmap );

GLboolean aglDestroyAGLPixmap ( AGLPixmap pix );

Exchange front and back buffers:

GLboolean aglSwapBuffers ( AGLDrawable drawable );

Update the current rendering buffers:

GLboolean aglUpdateCurrent (void);

Use a Macintosh font:

GLboolean aglUseFont ( int familyID, int size, int first, int count, int listBase );

Find the cause of an error:

GLenum aglGetError (void);

**PGL: OpenGL Extension for IBM OS/2 Warp**

OpenGL rendering for IBM OS/2 Warp is accomplished by using PGL routines added to integrate OpenGL into the standard IBM Presentation Manager. OpenGL with PGL supports both a direct OpenGL context (which is often faster) and an indirect context (which allows some integration of Gpi and OpenGL rendering).

The data type VISUALCONFIG (the PGL equivalent to XVisualInfo) maintains the visual configuration, including pixel type (RGBA or color index), single- or double-buffering, resolution of colors, and presence of depth, stencil, and accumulation buffers.
To get more information (including how to obtain the OpenGL software library for IBM OS/2 Warp, Version 3.0), you may want to start at

http://www.austin.ibm.com/software/OpenGL/

Packaged along with the software is the document, OpenGL On OS/2 Warp, which provides more detailed information. OpenGL support is included with the base operating system with OS/2 Warp Version 4.

**Initialization**

Use `pglQueryCapability()` and `pglQueryVersion()` to determine whether the OpenGL is supported on this machine and, if so, how it is supported and which version is present. `pglChooseConfig()` returns a pointer to an VISUALCONFIG structure describing the visual configuration that best meets the client’s specified attributes. A list of the particular visual configurations supported by a graphics device can be found using `pglQueryConfigs()`.

**Controlling Rendering**

Several PGL routines are provided for creating and managing an OpenGL rendering context, capturing the contents of a bitmap, synchronizing execution between the Presentation Manager and OpenGL streams, swapping front and back buffers, using a color palette, and using an OS/2 logical font.

**Managing an OpenGL Rendering Context**

An OpenGL rendering context is created with `pglCreateContext()`. One of the arguments to this routine allows you to request a direct rendering context that bypasses the Gpi and render to a PM window, which is generally faster. You can determine whether a OpenGL context is direct with `pglIsIndirect()`.

To make a rendering context current, use `pglMakeCurrent();` `pglGetCurrentContext()` returns the current context. You can also obtain the current window with `pglGetCurrentWindow()`. You can copy some OpenGL state variables from one context to another with `pglCopyContext()`. When you’re finished with a particular context, destroy it with `pglDestroyContext()`.

**Access the Bitmap of the Front Buffer**

To lock access to the bitmap representation of the contents of the front buffer, use `pglGrabFrontBitmap()`. An implicit glFlush() is performed, and you can read the bitmap, but its contents are effectively read–only. Immediately after access is completed, you should call `pglReleaseFrontBitmap()` to restore write access to the front buffer.

**Synchronizing Execution**

To prevent Gpi rendering requests from executing until any outstanding OpenGL rendering is completed, call `pglWaitGL()`. Then, any previously issued OpenGL commands are guaranteed to be executed before any Gpi rendering calls made after `pglWaitGL()`.

To prevent an OpenGL command sequence from executing until any outstanding Gpi requests are
completed, use `pglWaitPM()`. This routine guarantees that previously issued Gpi rendering calls are executed before any OpenGL calls made after `pglWaitPM()`.

**Note:** OpenGL and Gpi rendering can be integrated in the same window only if the OpenGL context is an indirect context.

### Swapping Buffers

For windows that are double-buffered, the front and back buffers can be exchanged by calling `pglSwapBuffers()`. An implicit `glFlush()` is done as part of this routine.

### Using a Color Index Palette

When you are running in 8-bit (256 color) mode, you have to worry about color palette management. For windows with a color index Visual Configuration, call `pglSelectColorIndexPalette()` to tell OpenGL what color-index palette you want to use with your context. A color palette must be selected before the context is initially bound to a window. In RGBA mode, OpenGL sets up a palette automatically.

### Using an OS/2 Logical Font

A shortcut for using OS/2 logical fonts in OpenGL is provided with the command `pglUseFont()`. This routine builds display lists, each of which calls `glBitmap()`, for each requested character from the specified font and font size.

### PGL Prototypes

#### Initialization

Determine whether OpenGL is supported and, if so, its version number:

```c
long pglQueryCapability (HAB hab);
void pglQueryVersion (HAB hab, int *major, int *minor);
```

Visual configuration selection, availability and capability:

```c
PVISUALCONFIG pglChooseConfig (HAB hab, int *attribList);
PVISUALCONFIG * pglQueryConfigs (HAB hab);
```

#### Controlling Rendering

Manage or query an OpenGL rendering context:

```c
HGC pglCreateContext (HAB hab, PVISUALCONFIG pVisualConfig, HGC shareList, Bool IDirect);
Bool pglDestroyContext (HAB hab, HGC hgc);
Bool pglCopyContext (HAB hab, HGC source, HGC dest, GLuint mask);
```
Bool pglMakeCurrent (HAB hab, HGC hgc, HWND hwnd);
long pglIsIndirect (HAB hab, HGC hgc);
HGC pglGetCurrentContext (HAB hab);
HWND pglGetCurrentWindow (HAB hab);

Access and release the bitmap of the front buffer:

Bool pglGrabFrontBitmap (HAB hab, HPS *hps, HBITMAP *phbitmap);
Bool pglReleaseFrontBitmap (HAB hab);

Synchronize execution:

HPS pglWaitGL (HAB hab);
void pglWaitPM (HAB hab);

Exchange front and back buffers:

void pglSwapBuffers (HAB hab, HWND hwnd);

Finding a color−index palette:

void pglSelectColorIndexPalette (HAB hab, HPAL hpal, HGC hgc);

Use an OS/2 logical font:

Bool pglUseFont (HAB hab, HPS hps, FATTRS *fontAttribs,
long logicalId, int first, int count, int listBase);

WGL: OpenGL Extension for Microsoft Windows NT and Windows 95

OpenGL rendering is supported on systems that run Microsoft Windows NT and Windows 95. The functions and routines of the Win32 library are necessary to initialize the pixel format and control rendering for OpenGL. Some routines, which are prefixed by wgl, extend Win32 so that OpenGL can be fully supported.

For Win32/WGL, the PIXELFORMATDESCRIPTOR is the key data structure to maintain pixel format information about the OpenGL window. A variable of data type PIXELFORMATDESCRIPTOR keeps track of pixel information, including pixel type (RGBA or color index), single− or double− buffering, resolution of colors, and presence of depth, stencil, and accumulation buffers.

To get more information about WGL, you may want to start with technical articles available through the Microsoft Developer Network at

http://www.microsoft.com/msdn/

Initialization

Use GetVersion() or the newer GetVersionEx() to determine version information. ChoosePixelFormat() tries to find a PIXELFORMATDESCRIPTOR with specified attributes. If a good match for the
requested pixel format is found, then `SetPixelFormat()` should be called to actually use the pixel format. You should select a pixel format in the device context before calling `wglCreateContext()`.

If you want to find out details about a given pixel format, use `DescribePixelFormat()` or, for overlays or underlays, `wglDescribeLayerPlane()`.

### Controlling Rendering

Several WGL routines are provided for creating and managing an OpenGL rendering context, rendering to a bitmap, swapping front and back buffers, finding a color palette, and using either bitmap or outline fonts.

#### Managing an OpenGL Rendering Context

`wglCreateContext()` creates an OpenGL rendering context for drawing on the device in the selected pixel format of the device context. (To create an OpenGL rendering context for overlay or underlay windows, use `wglCreateLayerContext()` instead.) To make a rendering context current, use `wglMakeCurrent()`. `wglGetCurrentContext()` returns the current context. You can also obtain the current device context with `wglGetCurrentDC()`. You can copy some OpenGL state variables from one context to another with `wglCopyContext()` or make two contexts share the same display lists and texture objects with `wglShareLists()`. When you’re finished with a particular context, destroy it with `wglDestroyContext()`.

#### OpenGL Rendering to a Bitmap

Win32 has a few routines to allocate (and deallocate) bitmaps, to which you can render OpenGL directly. `CreateDIBitmap()` creates a device−dependent bitmap (DDB) from a device−independent bitmap (DIB). `CreateDIBSection()` creates a device−independent bitmap (DIB) that applications can write to directly. When finished with your bitmap, you can use `DeleteObject()` to free it up.

#### Synchronizing Execution

If you want to combine GDI and OpenGL rendering, be aware there are no equivalents to functions like `glXWaitGL()`, `glXWaitX()`, or `pglWaitGL()` in Win32. Although `glXWaitGL()` has no equivalent in Win32, you can achieve the same effect by calling `glFinish()`, which waits until all pending OpenGL commands are executed, or by calling `GdiFlush()`, which waits until all GDI drawing has completed.

#### Swapping Buffers

For windows that are double−buffered, the front and back buffers can be exchanged by calling `SwapBuffers()` or `wglSwapLayerBuffers()`, the latter for overlays and underlays.

#### Finding a Color Palette

To access the color palette for the standard (non−layer) bitplanes, use the standard GDI functions to set the palette entries. For overlay or underlay layers, use `wglRealizeLayerPalette()`, which maps palette entries from a given color−index layer plane into the physical palette or initializes the palette of an RGBA layer plane. `wglGetLayerPaletteEntries()` is used to query the entries in palettes of layer planes.
Using a Bitmap or Outline Font

WGL has two routines, `wglUseFontBitmaps()` and `wglUseFontOutlines()`, for converting system fonts to use with OpenGL. Both routines build a display list for each requested character from the specified font and font size.

**WGL Prototypes**

**Initialization**

Determine version information:

- `BOOL GetVersion ( LPOSPVERSIONINFO lpVersionInformation );`
- `BOOL GetVersionEx ( LPOSPVERSIONINFO lpVersionInformation );`

Pixel format availability, selection, and capability:

- `int ChoosePixelFormat ( HDC hdc, CONST PIXELFORMATDESCRIPTOR * ppfd );`
- `BOOL SetPixelFormat ( HDC hdc, int iPixelFormat, CONST PIXELFORMATDESCRIPTOR * ppfd );`
- `int DescribePixelFormat ( HDC hdc, int iPixelFormat, UINT nBytes, LPPIXELFORMATDESCRIPTOR ppfd );`
- `BOOL wglDescribeLayerPlane ( HDC hdc, int iPixelFormat, int iLayerPlane, UINT nBytes, LPLAYERPLANEDESCRIPTOR plpd );`

**Controlling Rendering**

Manage or query an OpenGL rendering context:

- `HGLRC wglCreateContext ( HDC hdc );`
- `HGLRC wglCreateLayerContext ( HDC hdc, int iLayerPlane );`
- `BOOL wglShareLists ( HGLRC hglrc1, HGLRC hglrc2 );`
- `BOOL wglDeleteContext ( HGLRC hglrc );`
- `BOOL wglCopyContext ( HGLRC hglrcSource, HGLRC hglrcDest, UINT mask );`
- `BOOL wglMakeCurrent ( HDC hdc, HGLRC hglrc );`
- `HGLRC wglGetCurrentContext (VOID);`
- `HDC wglGetCurrentDC (VOID);`

Access and release the bitmap of the front buffer:

- `HBITMAP CreateDIBitmap ( HDC hdc, CONST BITMAPINFOHEADER *lpbmih, DWORD fdwInit, CONST VOID *lpbInit, CONST BITMAPINFO *lpbmi, UINT fuUsage );`
HBITMAP CreateDIBSection ( HDC hdc, CONST BITMAPINFO *pbmi, 
UINT iUsage, VOID *ppvBits, HANDLE hSection, DWORD dwOffset );

BOOL DeleteObject ( HGDIOBJ hObject );

Exchange front and back buffers:

BOOL SwapBuffers ( HDC hdc );

BOOL wglSwapLayerBuffers ( HDC hdc, UINT fuPlanes );

Finding a color palette for overlay or underlay layers:

int wglGetLayerPaletteEntries ( HDC hdc, int iLayerPlane, int iStart, 
int cEntries, CONST COLORREF *pcr );

BOOL wglRealizeLayerPalette ( HDC hdc, int iLayerPlane, 
BOOL bRealize );

Use a bitmap or an outline font:

BOOL wglUseFontBitmaps ( HDC hdc, DWORD first, DWORD count, 
DWORD listBase );

BOOL wglUseFontOutlines ( HDC hdc, DWORD first, DWORD count, 
DWORD listBase, FLOAT deviation, FLOAT extrusion, int format, 
LPGLYPHMETRICSFLOAT lpgmf );
Appendix D
Basics of GLUT: The OpenGL Utility Toolkit

This appendix describes a subset of Mark Kilgard’s OpenGL Utility Toolkit (GLUT), which is fully documented in his book, OpenGL Programming for the X Window System (Reading, MA: Addison-Wesley Developers Press, 1996). GLUT has become a popular library for OpenGL programmers, because it standardizes and simplifies window and event management. GLUT has been ported atop a variety of OpenGL implementations, including both the X Window System and Microsoft Windows NT.

This appendix has the following major sections:

"Initializing and Creating a Window"

"Handling Window and Input Events"

"Loading the Color Map"

"Initializing and Drawing Three-Dimensional Objects"

"Managing a Background Process"

"Running the Program"

(See "How to Obtain the Sample Code" in the Preface for information about how to obtain the source code for GLUT.)

With GLUT, your application structures its event handling to use callback functions. (This method is similar to using the Xt Toolkit, also known as the X Intrinsics, with a widget set.) For example, first you open a window and register callback routines for specific events. Then, you create a main loop without an exit. In that loop, if an event occurs, its registered callback functions are executed. Upon completion of the callback functions, flow of control is returned to the main loop.

**Initializing and Creating a Window**

Before you can open a window, you must specify its characteristics: Should it be single-buffered or double-buffered? Should it store colors as RGBA values or as color indices? Where should it appear on your display? To specify the answers to these questions, call `glutInit()`, `glutInitDisplayMode()`, `glutInitWindowSize()`, and `glutInitWindowPosition()` before you call `glutCreateWindow()` to open the window.

```c
void glutInit(int argc, char **argv);
```

`glutInit()` should be called before any other GLUT routine, because it initializes the GLUT library. `glutInit()` will also process command line options, but the specific options are window system dependent. For the X Window System, `-iconic`, `-geometry`, and `-display` are examples of command line options, processed by `glutInit()`. (The parameters to the `glutInit()` should be the same as those to `main()`.)

```c
void glutInitDisplayMode(unsigned int mode);
```

OpenGL Programming Guide – Appendix D, Basics of GLUT: The OpenGL Utility Toolkit – 1
Specifies a display mode (such as RGBA or color−index, or single− or double−buffered) for windows created when glutCreateWindow() is called. You can also specify that the window have an associated depth, stencil, and/or accumulation buffer. The mask argument is a bitwise ORed combination of GLUT_RGBA or GLUT_INDEX, GLUT_SINGLE or GLUT_DOUBLE, and any of the buffer−enabling flags: GLUT_DEPTH, GLUT_STENCIL, or GLUT_ACCUM. For example, for a double−buffered, RGBA−mode window with a depth and stencil buffer, use GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH | GLUT_STENCIL. The default value is GLUT_RGBA | GLUT_SINGLE (an RGBA, single−buffered window).

```c
void glutInitWindowSize(int width, int height);
void glutInitWindowPosition(int x, int y);
```

Requests windows created by glutCreateWindow() to have an initial size and position. The arguments (x, y) indicate the location of a corner of the window, relative to the entire display. The width and height indicate the window’s size (in pixels). The initial window size and position are hints and may be overridden by other requests.

```c
int glutCreateWindow(char *name);
```

Opens a window with previously set characteristics (display mode, width, height, and so on). The string name may appear in the title bar if your window system does that sort of thing. The window is not initially displayed until glutMainLoop() is entered, so do not render into the window until then. The value returned is a unique integer identifier for the window. This identifier can be used for controlling and rendering to multiple windows (each with an OpenGL rendering context) from the same application.

### Handling Window and Input Events

After the window is created, but before you enter the main loop, you should register callback functions using the following routines.

```c
void glutDisplayFunc(void (*func)(void));
```

Specifies the function that’s called whenever the contents of the window need to be redrawn. The contents of the window may need to be redrawn when the window is initially opened, when the window is popped and window damage is exposed, and when glutPostRedisplay() is explicitly called.

```c
void glutReshapeFunc(void (*func)(int width, int height));
```

Specifies the function that’s called whenever the window is resized or moved. The argument `func` is a pointer to a function that expects two arguments, the new width and height of the window. Typically, `func` calls `glViewport()`, so that the display is clipped to the new size, and it redefines the projection matrix so that the aspect ratio of the projected image matches the viewport, avoiding aspect ratio distortion. If glutReshapeFunc() isn’t called or is deregistered by passing NULL, a default reshape function is called, which calls `glViewport(0, 0, width, height)`.

```c
void glutKeyboardFunc(void (*func)(unsigned int key, int x, int y));
```

Specifies the function, `func`, that’s called when a key that generates an ASCII character is pressed. The key callback parameter is the generated ASCII value. The x and y callback parameters indicate the location of the mouse (in window−relative coordinates) when the key was pressed.
void glutMouseFunc(void (*func)(int button, int state, int x, int y));

Specifies the function, func, that’s called when a mouse button is pressed or released. The button callback parameter is one of GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, or GLUT_RIGHT_BUTTON. The state callback parameter is either GLUT_UP or GLUT_DOWN, depending upon whether the mouse has been released or pressed. The x and y callback parameters indicate the location (in window-relative coordinates) of the mouse when the event occurred.

void glutMotionFunc(void (*func)(int x, int y));

Specifies the function, func, that’s called when the mouse pointer moves within the window while one or more mouse buttons is pressed. The x and y callback parameters indicate the location (in window-relative coordinates) of the mouse when the event occurred.

void glutPostRedisplay(void);

Marks the current window as needing to be redrawn. At the next opportunity, the callback function registered by glutDisplayFunc() will be called.

Loading the Color Map

If you’re using color-index mode, you might be surprised to discover there’s no OpenGL routine to load a color into a color lookup table. This is because the process of loading a color map depends entirely on the window system. GLUT provides a generalized routine to load a single color index with an RGB value, glutSetColor().

void glutSetColor(GLint index, GLfloat red, GLfloat green, GLfloat blue);

Loads the index in the color map, index, with the given red, green, and blue values. These values are normalized to lie in the range [0.0,1.0].

Initializing and Drawing Three-Dimensional Objects

Many sample programs in this guide use three-dimensional models to illustrate various rendering properties. The following drawing routines are included in GLUT to avoid having to reproduce the code to draw these models in each program. The routines render all their graphics in immediate mode. Each three-dimensional model comes in two flavors: wireframe without surface normals, and solid with shading and surface normals. Use the solid version when you’re applying lighting. Only the teapot generates texture coordinates.

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

void glutWireCube(GLdouble size);
void glutSolidCube(GLdouble size);

void glutWireTorus(GLdouble innerRadius, GLdouble outerRadius, GLint nsides, GLint rings);
void glutSolidTorus(GLdouble innerRadius, GLdouble outerRadius, GLint nsides, GLint rings);

void glutWireIcosahedron(void);
void glutSolidIcosahedron(void);
void glutWireOctahedron(void);
void glutSolidOctahedron(void);

void glutWireTetrahedron(void);
void glutSolidTetrahedron(void);

void glutWireDodecahedron(GLdouble radius);
void glutSolidDodecahedron(GLdouble radius);

void glutWireCone(GLdouble radius, GLdouble height, GLint slices,
GLint stacks);
void glutSolidCone(GLdouble radius, GLdouble height, GLint slices,
GLint stacks);

void glutWireTeapot(GLdouble size);
void glutSolidTeapot(GLdouble size);

**Managing a Background Process**

You can specify a function that’s to be executed if no other events are pending—for example, when the event loop would otherwise be idle—with `glutIdleFunc()`. This is particularly useful for continuous animation or other background processing.

```c
void glutIdleFunc(void (*func)(void));
```

Specifies the function, `func`, to be executed if no other events are pending. If NULL (zero) is passed in, execution of `func` is disabled.

**Running the Program**

After all the setup is completed, GLUT programs enter an event processing loop, `glutMainLoop()`.

```c
void glutMainLoop(void);
```

Enters the GLUT processing loop, never to return. Registered callback functions will be called when the corresponding events instigate them.
Appendix E
Calculating Normal Vectors

This appendix describes how to calculate normal vectors for surfaces. You need to define normals to use the OpenGL lighting facility, which is described in Chapter 5. "Normal Vectors" in Chapter 2 introduces normals and the OpenGL command for specifying them. This appendix goes through the details of calculating them. It has the following major sections:

"Finding Normals for Analytic Surfaces"

"Finding Normals from Polygonal Data"

Since normals are perpendicular to a surface, you can find the normal at a particular point on a surface by first finding the flat plane that just touches the surface at that point. The normal is the vector that’s perpendicular to that plane. On a perfect sphere, for example, the normal at a point on the surface is in the same direction as the vector from the center of the sphere to that point. For other types of surfaces, there are other, better means for determining the normals, depending on how the surface is specified.

Recall that smooth curved surfaces are approximated by a large number of small flat polygons. If the vectors perpendicular to these polygons are used as the surface normals in such an approximation, the surface appears faceted, since the normal direction is discontinuous across the polygonal boundaries. In many cases, however, an exact mathematical description exists for the surface, and true surface normals can be calculated at every point. Using the true normals improves the rendering considerably, as shown in Figure E-1. Even if you don’t have a mathematical description, you can do better than the faceted look shown in the figure. The two major sections in this appendix describe how to calculate normal vectors for these two cases:

"Finding Normals for Analytic Surfaces" explains what to do when you have a mathematical description of a surface.

"Finding Normals from Polygonal Data" covers the case when you have only the polygonal data to describe a surface.

![Figure E-1 Rendering with Polygonal Normals vs. True Normals](image)

Finding Normals for Analytic Surfaces

Analytic surfaces are smooth, differentiable surfaces that are described by a mathematical equation (or set of equations). In many cases, the easiest surfaces to find normals for are analytic surfaces for which
you have an explicit definition in the following form:

\[ V(s, t) = [X(s, t) \ Y(s, t) \ Z(s, t)] \]

where \( s \) and \( t \) are constrained to be in some domain, and \( X, Y, \) and \( Z \) are differentiable functions of two variables. To calculate the normal, find

\[ \frac{\partial V}{\partial s} \text{ and } \frac{\partial V}{\partial t} \]

which are vectors tangent to the surface in the \( s \) and \( t \) directions. The cross product

\[ \frac{\partial V}{\partial s} \times \frac{\partial V}{\partial t} \]

is perpendicular to both and, hence, to the surface. The following shows how to calculate the cross product of two vectors. (Watch out for the degenerate cases where the cross product has zero length!)

\[ \begin{bmatrix} v_x & v_y & v_z \\ w_x & w_y & w_z \end{bmatrix} \times \begin{bmatrix} v_x & v_y & v_z \\ w_x & w_y & w_z \end{bmatrix} = \begin{bmatrix} (v_y w_z - w_y v_z) \\ (w_x v_z - v_x w_z) \\ (v_x w_y - w_x v_y) \end{bmatrix} \]

You should probably normalize the resulting vector. To normalize a vector \([x \ y \ z]\), calculate its length

\[ \text{Length} = \sqrt{x^2 + y^2 + z^2} \]

and divide each component of the vector by the length.

As an example of these calculations, consider the analytic surface

\[ V(s, t) = [s^2 \ t^3 \ 3st] \]

From this we have

\[ \frac{\partial V}{\partial s} = [2s \ 0 \ -t], \ \frac{\partial V}{\partial t} = [0 \ 3t^2 \ -s], \ \text{and } \frac{\partial V}{\partial s} \times \frac{\partial V}{\partial t} = [-3t^3 \ 2st^2 \ 6st^2] \]

So, for example, when \( s=1 \) and \( t=2 \), the corresponding point on the surface is \((1, 8, 1)\), and the vector \((-24, 2, 24)\) is perpendicular to the surface at that point. The length of this vector is 34, so the unit normal vector is \((-24/34, 2/34, 24/34) = (-0.70588, 0.058823, 0.70588)\).

For analytic surfaces that are described implicitly, as \( F(x, y, z) = 0 \), the problem is harder. In some cases, you can solve for one of the variables, say \( z = G(x, y) \), and put it in the explicit form given previously:
\[ \mathbf{V}(s, t) = [s \ t \ \mathbf{G}(s, t)] \]

Then continue as described earlier.

If you can’t get the surface equation in an explicit form, you might be able to make use of the fact that the normal vector is given by the gradient

\[ \mathbf{VF} = \begin{bmatrix} \frac{\partial F}{\partial x} & \frac{\partial F}{\partial y} & \frac{\partial F}{\partial z} \end{bmatrix} \]

evaluated at a particular point \((x, y, z)\). Calculating the gradient might be easy, but finding a point that lies on the surface can be difficult. As an example of an implicitly defined analytic function, consider the equation of a sphere of radius 1 centered at the origin:

\[ x^2 + y^2 + z^2 - 1 = 0 \]

This means that

\[ \mathbf{F}(x, y, z) = x^2 + y^2 + z^2 - 1 \]

which can be solved for \(z\) to yield

\[ z = \pm \sqrt{1 - x^2 - y^2} \]

Thus, normals can be calculated from the explicit form

\[ \mathbf{V}(s, t) = [s \ t \ \pm \sqrt{1 - s^2 - t^2}] \]

as described previously.

If you could not solve for \(z\), you could have used the gradient

\[ \mathbf{VF} = \begin{bmatrix} 2x & 2y & 2z \end{bmatrix} \]

as long as you could find a point on the surface. In this case, it’s not so hard to find a point—for example, \((2/3, 1/3, 2/3)\) lies on the surface. Using the gradient, the normal at this point is \((4/3, 2/3, 4/3)\). The unit–length normal is \((2/3, 1/3, 2/3)\), which is the same as the point on the surface, as expected.

**Finding Normals from Polygonal Data**

As mentioned previously, you often want to find normals for surfaces that are described with polygonal
data such that the surfaces appear smooth rather than faceted. In most cases, the easiest way for you to do this (though it might not be the most efficient way) is to calculate the normal vectors for each of the polygonal facets and then to average the normals for neighboring facets. Use the averaged normal for the vertex that the neighboring facets have in common. Figure E-2 shows a surface and its polygonal approximation. (Of course, if the polygons represent the exact surface and aren’t merely an approximation—if you’re drawing a cube or a cut diamond, for example—don’t do the averaging. Calculate the normal for each facet as described in the following paragraphs, and use that same normal for each vertex of the facet.)
Figure E–2 Averaging Normal Vectors

To find the normal for a flat polygon, take any three vertices \(v_1, v_2,\) and \(v_3\) of the polygon that do not lie in a straight line. The cross product

\[
[v_1 - v_2] \times [v_2 - v_3]
\]

is perpendicular to the polygon. (Typically, you want to normalize the resulting vector.) Then you need to average the normals for adjoining facets to avoid giving too much weight to one of them. For instance, in the example shown in Figure E–2 if \(n_1, n_2, n_3,\) and \(n_4\) are the normals for the four polygons meeting at point P, calculate \(n_1 + n_2 + n_3 + n_4\) and then normalize it. (You can get a better average if you weight the normals by the size of the angles at the shared intersection.) The resulting vector can be used as the normal for point P.

Sometimes, you need to vary this method for particular situations. For instance, at the boundary of a surface (for example, point Q in Figure E–2), you might be able to choose a better normal based on your knowledge of what the surface should look like. Sometimes the best you can do is to average the polygon normals on the boundary as well. Similarly, some models have some smooth parts and some sharp corners (point R is on such an edge in Figure E–2). In this case, the normals on either side of the crease shouldn’t be averaged. Instead, polygons on one side of the crease should be drawn with one normal, and polygons on the other side with another.
Appendix F
Homogeneous Coordinates and Transformation Matrices

This appendix presents a brief discussion of homogeneous coordinates. It also lists the form of the transformation matrices used for rotation, scaling, translation, perspective projection, and orthographic projection. These topics are introduced and discussed in Chapter 3. For a more detailed discussion of these subjects, see almost any book on three–dimensional computer graphics—for example, Computer Graphics: Principles and Practice by Foley, van Dam, Feiner, and Hughes (Reading, MA: Addison–Wesley, 1990)—or a text on projective geometry—for example, The Real Projective Plane, by H. S. M. Coxeter, 2nd ed. (Cambridge: Cambridge University Press, 1961). In the discussion that follows, the term homogeneous coordinates always means three–dimensional homogeneous coordinates, although projective geometries exist for all dimensions.

This appendix has the following major sections:

"Homogeneous Coordinates"

"Transformation Matrices"

Homogeneous Coordinates

OpenGL commands usually deal with two– and three–dimensional vertices, but in fact all are treated internally as three–dimensional homogeneous vertices comprising four coordinates. Every column vector \((x, y, z, w)^T\) represents a homogeneous vertex if at least one of its elements is nonzero. If the real number \(a\) is nonzero, then \((x, y, z, w)^T\) and \((ax, ay, az, aw)^T\) represent the same homogeneous vertex. (This is just like fractions: \(x/y = (ax)/(ay)\).) A three–dimensional euclidean space point \((x, y, z)^T\) becomes the homogeneous vertex with coordinates \((x, y, z, 1.0)^T\), and the two–dimensional euclidean point \((x, y)^T\) becomes \((x, y, 0.0, 1.0)^T\).

As long as \(w\) is nonzero, the homogeneous vertex \((x, y, z, w)^T\) corresponds to the three–dimensional point \((x/w, y/w, z/w)^T\). If \(w = 0.0\), it corresponds to no euclidean point, but rather to some idealized "point at infinity." To understand this point at infinity, consider the point \((1, 2, 0, 0)\), and note that the sequence of points \((1, 2, 0, 1), (1, 2, 0, 0.01), \) and \((1, 2, 0, 0.0001)\), corresponds to the euclidean points \((1, 2), (100, 200), \) and \((10000, 20000)\). This sequence represents points rapidly moving toward infinity along the line \(2x = y\). Thus, you can think of \((1, 2, 0, 0)\) as the point at infinity in the direction of that line.

**Note:** OpenGL might not handle homogeneous clip coordinates with \(w < 0\) correctly. To be sure that your code is portable to all OpenGL systems, use only nonnegative \(w\) values.

Transforming Vertices

Vertex transformations (such as rotations, translations, scaling, and shearing) and projections (such as perspective and orthographic) can all be represented by applying an appropriate \(4\times4\) matrix to the coordinates representing the vertex. If \(v\) represents a homogeneous vertex and \(M\) is a \(4\times4\) transformation matrix, then \(Mv\) is the image of \(v\) under the transformation by \(M\). (In computer–graphics applications, the...
transformations used are usually nonsingular—in other words, the matrix $M$ can be inverted. This isn’t required, but some problems arise with nonsingular transformations.)

After transformation, all transformed vertices are clipped so that $x$, $y$, and $z$ are in the range $[-\omega, w]$ (assuming $w > 0$). Note that this range corresponds in euclidean space to $[-1.0, 1.0]$.

### Transforming Normals

Normal vectors aren’t transformed in the same way as vertices or position vectors. Mathematically, it’s better to think of normal vectors not as vectors, but as planes perpendicular to those vectors. Then, the transformation rules for normal vectors are described by the transformation rules for perpendicular planes.

A homogeneous plane is denoted by the row vector $(a, b, c, d)$, where at least one of $a$, $b$, $c$, or $d$ is nonzero. If $q$ is a nonzero real number, then $(a, b, c, d)$ and $(qa, qb, qc, qd)$ represent the same plane. A point $(x, y, z, w)^T$ is on the plane $(a, b, c, d)$ if $ax+by+cz+dw = 0$. (If $w = 1$, this is the standard description of a euclidean plane.) In order for $(a, b, c, d)$ to represent a euclidean plane, at least one of $a$, $b$, or $c$ must be nonzero. If they’re all zero, then $(0, 0, 0, d)$ represents the "plane at infinity," which contains all the "points at infinity."

If $p$ is a homogeneous plane and $v$ is a homogeneous vertex, then the statement "$v$ lies on plane $p$" is written mathematically as $pv = 0$, where $pv$ is normal matrix multiplication. If $M$ is a nonsingular vertex transformation (that is, a $4 \times 4$ matrix that has an inverse $M^{-1}$), then $pv = 0$ is equivalent to $pM^{-1}Mv = 0$, so $Mv$ lies on the plane $pM^{-1}$. Thus, $pM^{-1}$ is the image of the plane under the vertex transformation $M$.

If you like to think of normal vectors as vectors instead of as the planes perpendicular to them, let $v$ and $n$ be vectors such that $v$ is perpendicular to $n$. Then, $n^Tv = 0$. Thus, for an arbitrary nonsingular transformation $M$, $n^TM^{-1}Mv = 0$, which means that $nTM^{-1}$ is the transpose of the transformed normal vector. Thus, the transformed normal vector is $(M^{-1})^Tn$. In other words, normal vectors are transformed by the inverse transpose of the transformation that transforms points. Whew!

### Transformation Matrices

Although any nonsingular matrix $M$ represents a valid projective transformation, a few special matrices are particularly useful. These matrices are listed in the following subsections.

#### Translation

The call `glTranslate*(x, y, z)` generates $T$, where
Scaling

The call `glScale*(x, y, z)` generates $S$, where

\[
T = \begin{bmatrix}
1 & 0 & 0 & x \\
0 & 1 & 0 & y \\
0 & 0 & 1 & z \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\quad \text{and} \quad
T^{-1} = \begin{bmatrix}
1 & 0 & 0 & -x \\
0 & 1 & 0 & -y \\
0 & 0 & 1 & -z \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

Notice that $S^{-1}$ is defined only if $x$, $y$, and $z$ are all nonzero.

Rotation

The call `glRotate*(a, x, y, z)` generates $R$ as follows:

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

Also let

\[
S = \begin{bmatrix}
x & 0 & 0 & 0 \\
0 & y & 0 & 0 \\
0 & 0 & z & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\quad \text{and} \quad
S^{-1} = \begin{bmatrix}
\frac{1}{x} & 0 & 0 & 0 \\
0 & \frac{1}{y} & 0 & 0 \\
0 & 0 & \frac{1}{z} & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

The $R$ matrix is always defined. If $x=y=z=0$, then $R$ is the identity matrix. You can obtain the inverse of $R$, $R^{-1}$, by substituting $-\alpha$ for $a$, or by transposition.
The `glRotate*()` command generates a matrix for rotation about an arbitrary axis. Often, you’re rotating about one of the coordinate axes; the corresponding matrices are as follows:

$$\text{glRotate}^*(a, 1, 0, 0): \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos a & -\sin a & 0 \\ 0 & \sin a & \cos a & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{glRotate}^*(a, 0, 1, 0): \begin{bmatrix} \cos a & 0 & \sin a & 0 \\ 0 & 1 & 0 & 0 \\ -\sin a & 0 & \cos a & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{glRotate}^*(a, 0, 0, 1): \begin{bmatrix} \cos a & -\sin a & 0 & 0 \\ \sin a & \cos a & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

As before, the inverses are obtained by transposition.

**Perspective Projection**

The call `glFrustum(l, r, b, t, n, f)` generates $R$, where

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

and $R^{-1} = \begin{bmatrix} \frac{r-l}{2n} & 0 & 0 & \frac{r+l}{2n} \\ 0 & \frac{t-b}{2n} & 0 & \frac{t+b}{2n} \\ 0 & 0 & 0 & -1 \\ 0 & 0 & \frac{(f-r)}{2fn} & \frac{f+r}{2fn} \end{bmatrix}$

$R$ is defined as long as $l \neq r$, $t \neq b$, and $n \neq f$.

**Orthographic Projection**

The call `glOrtho(l, r, b, t, n, f)` generates $R$, where

$R$ is defined as long as $l \neq r$, $t \neq b$, and $n \neq f$. 

OpenGL  Programming Guide − Appendix F,  Homogeneous Coordinates and Transformation Matrices − 4
\[ R = \begin{bmatrix}
\frac{2}{r - l} & 0 & 0 & \frac{r + l}{r - l} \\
0 & \frac{2}{t - b} & 0 & \frac{t + b}{t - b} \\
0 & 0 & \frac{-2}{f - n} & \frac{f + n}{f - n} \\
0 & 0 & 0 & 1
\end{bmatrix} \quad \text{and} \quad R^{-1} = \begin{bmatrix}
\frac{r - l}{2} & 0 & 0 & \frac{r + l}{2} \\
0 & \frac{t - b}{2} & 0 & \frac{t + b}{2} \\
0 & 0 & \frac{f - n}{2} & \frac{n + f}{2} \\
0 & 0 & 0 & 1
\end{bmatrix} \]

\( R \) is defined as long as \( l \neq r, t \neq b, \) and \( n \neq f. \)
Appendix G
Programming Tips

This appendix lists some tips and guidelines that you might find useful. Keep in mind that these tips are based on the intentions of the designers of the OpenGL, not on any experience with actual applications and implementations! This appendix has the following major sections:

"OpenGL Correctness Tips"

"OpenGL Performance Tips"

"GLX Tips"

OpenGL Correctness Tips

Perform error checking often. Call glGetError() at least once each time the scene is rendered to make certain error conditions are noticed.

Do not count on the error behavior of an OpenGL implementation—it might change in a future release of OpenGL. For example, OpenGL 1.1 ignores matrix operations invoked between glBegin() and glEnd() commands, but a future version might not. Put another way, OpenGL error semantics may change between upward-compatible revisions.

If you need to collapse all geometry to a single plane, use the projection matrix. If the modelview matrix is used, OpenGL features that operate in eye coordinates (such as lighting and application-defined clipping planes) might fail.

Do not make extensive changes to a single matrix. For example, do not animate a rotation by continually calling glRotate*() with an incremental angle. Rather, use glLoadIdentity() to initialize the given matrix for each frame, then call glRotate*() with the desired complete angle for that frame.

Count on multiple passes through a rendering database to generate the same pixel fragments only if this behavior is guaranteed by the invariance rules established for a compliant OpenGL implementation. (See Appendix H for details on the invariance rules.) Otherwise, a different set of fragments might be generated.

Do not expect errors to be reported while a display list is being defined. The commands within a display list generate errors only when the list is executed.

Place the near frustum plane as far from the viewpoint as possible to optimize the operation of the depth buffer.

Call glFlush() to force all previous OpenGL commands to be executed. Do not count on glGet*() or glIs*() to flush the rendering stream. Query commands flush as much of the stream as is required to return valid data but don’t guarantee completing all pending rendering commands.

Turn dithering off when rendering predithered images (for example, when glCopyPixels() is called).

Make use of the full range of the accumulation buffer. For example, if accumulating four images,
scale each by one-quarter as it’s accumulated.

If exact two-dimensional rasterization is desired, you must carefully specify both the orthographic projection and the vertices of primitives that are to be rasterized. The orthographic projection should be specified with integer coordinates, as shown in the following example:

```c
gluOrtho2D(0, width, 0, height);
```

where `width` and `height` are the dimensions of the viewport. Given this projection matrix, polygon vertices and pixel image positions should be placed at integer coordinates to rasterize predictably. For example, `glRecti(0, 0, 1, 1)` reliably fills the lower left pixel of the viewport, and `glRasterPos2i(0, 0)` reliably positions an unzoomed image at the lower left of the viewport. Point vertices, line vertices, and bitmap positions should be placed at half-integer locations, however. For example, a line drawn from `(x1, 0.5)` to `(x2, 0.5)` will be reliably rendered along the bottom row of pixels into the viewport, and a point drawn at `(0.5, 0.5)` will reliably fill the same pixel as `glRecti(0, 0, 1, 1)`.

An optimum compromise that allows all primitives to be specified at integer positions, while still ensuring predictable rasterization, is to translate x and y by 0.375, as shown in the following code fragment. Such a translation keeps polygon and pixel image edges safely away from the centers of pixels, while moving line vertices close enough to the pixel centers.

```c
glViewport(0, 0, width, height);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0, width, 0, height);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0.375, 0.375, 0.0);
/* render all primitives at integer positions */
```

Avoid using negative w vertex coordinates and negative q texture coordinates. OpenGL might not clip such coordinates correctly and might make interpolation errors when shading primitives defined by such coordinates.

Do not assume the precision of operations, based upon the data type of parameters to OpenGL commands. For example, if you are using `glRotated()`, you should not assume that geometric processing pipeline operates with double-precision floating point. It is possible that the parameters to `glRotated()` are converted to a different data type before processing.

### OpenGL Performance Tips

Use `glColorMaterial()` when only a single material property is being varied rapidly (at each vertex, for example). Use `glMaterial()` for infrequent changes, or when more than a single material property is being varied rapidly.

Use `glLoadIdentity()` to initialize a matrix, rather than loading your own copy of the identity matrix.
Use specific matrix calls such as glRotate*, glTranslate*, and glScale* rather than composing your own rotation, translation, or scale matrices and calling glMultMatrix().

Use query functions when your application requires just a few state values for its own computations. If your application requires several state values from the same attribute group, use glPushAttrib() and glPopAttrib() to save and restore them.

Use display lists to encapsulate potentially expensive state changes.

Use display lists to encapsulate the rendering calls of rigid objects that will be drawn repeatedly.

Use texture objects to encapsulate texture data. Place all the glTexImage* calls (including mipmaps) required to completely specify a texture and the associated glTexParameter* calls (which set texture properties) into a texture object. Bind this texture object to select the texture. If the situation allows it, use gl*TexSubImage() to replace all or part of an existing texture image rather than the more costly operations of deleting and creating an entire new image.

If your OpenGL implementation supports a high−performance working set of resident textures, try to make all your textures resident; that is, make them fit into the high−performance texture memory. If necessary, reduce the size or internal format resolution of your textures until they all fit into memory. If such a reduction creates intolerably fuzzy textured objects, you may give some textures lower priority, which will, when push comes to shove, leave them out of the working set.

Use evaluators even for simple surface tessellations to minimize network bandwidth in client−server environments.

Provide unit−length normals if it’s possible to do so, and avoid the overhead of GL_NORMALIZE. Avoid using glScale* when doing lighting because it almost always requires that GL_NORMALIZE be enabled.

Set glShadeModel() to GL_FLAT if smooth shading isn’t required.

Use a single glClear() call per frame if possible. Do not use glClear() to clear small subregions of the buffers; use it only for complete or near−complete clears.

Use a single call to glBegin(GL_TRIANGLES) to draw multiple independent triangles rather than calling glBegin(GL_TRIANGLES) multiple times, or calling glBegin(GL_POLYGON). Even if only a single triangle is to be drawn, use GL_TRIANGLES rather than GL_POLYGON. Use a single call to glBegin(GL_QUADS) in the same manner rather than calling glBegin(GL_POLYGON) repeatedly. Likewise, use a single call to glBegin(GL_LINES) to draw multiple independent line segments rather than calling glBegin(GL_LINES) multiple times.

Some OpenGL implementations benefit from storing vertex data in vertex arrays. Use of vertex arrays reduces function call overhead. Some implementations can improve performance by batch processing or reusing processed vertices.

In general, use the vector forms of commands to pass precomputed data, and use the scalar forms of commands to pass values that are computed near call time.
Avoid making redundant mode changes, such as setting the color to the same value between each vertex of a flat−shaded polygon.

Be sure to disable expensive rasterization and per−fragment operations when drawing or copying images. OpenGL will even apply textures to pixel images if asked to!

Unless absolutely needed, avoid having different front and back polygon modes.

GLX Tips

Use glXWaitGL() rather than glFinish() to force X rendering commands to follow GL rendering commands.

Likewise, use glXWaitX() rather than XSync() to force GL rendering commands to follow X rendering commands.

Be careful when using glXChooseVisual(), because boolean selections are matched exactly. Since some implementations won’t export visuals with all combinations of boolean capabilities, you should call glXChooseVisual() several times with different boolean values before you give up. For example, if no single−buffered visual with the required characteristics is available, check for a double−buffered visual with the same capabilities. It might be available, and it’s easy to use.
Appendix H
OpenGL Invariance

OpenGL is not a pixel-exact specification. It therefore doesn’t guarantee an exact match between images produced by different OpenGL implementations. However, OpenGL does specify exact matches, in some cases, for images produced by the same implementation. This appendix describes the invariance rules that define these cases.

The obvious and most fundamental case is repeatability. A conforming OpenGL implementation generates the same results each time a specific sequence of commands is issued from the same initial conditions. Although such repeatability is useful for testing and verification, it’s often not useful to application programmers, because it’s difficult to arrange for equivalent initial conditions. For example, rendering a scene twice, the second time after swapping the front and back buffers, doesn’t meet this requirement. So repeatability can’t be used to guarantee a stable, double-buffered image.

A simple and useful algorithm that counts on invariant execution is erasing a line by redrawing it in the background color. This algorithm works only if rasterizing the line results in the same fragment x,y pairs being generated in both the foreground and background color cases. OpenGL requires that the coordinates of the fragments generated by rasterization be invariant with respect to framebuffer contents, which color buffers are enabled for drawing, the values of matrices other than those on the top of the matrix stacks, the scissor parameters, all writemasks, all clear values, the current color, index, normal, texture coordinates, and edge-flag values, the current raster color, raster index, and raster texture coordinates, and the material properties. It is further required that exactly the same fragments be generated, including the fragment color values, when framebuffer contents, color buffer enables, matrices other than those on the top of the matrix stacks, the scissor parameters, writemasks, or clear values differ.

OpenGL further suggests, but doesn’t require, that fragment generation be invariant with respect to the matrix mode, the depths of the matrix stacks, the stencil parameters (other than stencil enable), the depth test parameters (other than depth test enable), the blending parameters (other than enable), the logical operation (but not logical operation enable), and the pixel-storage and pixel-transfer parameters. Because invariance with respect to several enables isn’t recommended, you should use other parameters to disable functions when invariant rendering is required. For example, to render invariantly with blending enabled and disabled, set the blending parameters to GL_ONE and GL_ZERO to disable blending rather than calling glEnable(GL_BLEND). Alpha testing, stencil testing, depth testing, and the logical operation all can be disabled in this manner.

Finally, OpenGL requires that per-fragment arithmetic, such as blending and the depth test, is invariant to all OpenGL state except the state that directly defines it. For example, the only OpenGL parameters that affect how the arithmetic of blending is performed are the source and destination blend parameters and the blend enable parameter. Blending is invariant to all other state changes. This invariance holds for the scissor test, the alpha test, the stencil test, the depth test, blending, dithering, logical operations, and buffer writemasking.

As a result of all these invariance requirements, OpenGL can guarantee that images rendered into different color buffers, either simultaneously or separately using the same command sequence, are pixel identical. This holds for all the color buffers in the framebuffer or all the color buffers in an off-screen
buffer, but it isn’t guaranteed between the framebuffer and off-screen buffers.
Glossary

accumulation buffer
Memory (bitplanes) that is used to accumulate a series of images generated in the color buffer. Using the accumulation buffer may significantly improve the quality of the image, but also take correspondingly longer to render. The accumulation buffer is used for effects such as depth of field, motion blur, and full-scene antialiasing.

aliasing
A rendering technique that assigns to pixels the color of the primitive being rendered, regardless of whether that primitive covers all or only a portion of the pixel’s area. This results in jagged edges, or jaggies.

alpha
A fourth color component. The alpha component is never displayed directly and is typically used to control color blending. By convention, OpenGL alpha corresponds to the notion of opacity rather than transparency, meaning that an alpha value of 1.0 implies complete opacity, and an alpha value of 0.0 complete transparency.

ambient
Ambient light is nondirectional and distributed uniformly throughout space. Ambient light falling upon a surface approaches from all directions. The light is reflected from the object independent of surface location and orientation with equal intensity in all directions.

animation
Generating repeated renderings of a scene, with smoothly changing viewpoint and/or object positions, quickly enough so that the illusion of motion is achieved. OpenGL animation is almost always done using double-buffering.

antialiasing
A rendering technique that assigns pixel colors based on the fraction of the pixel’s area that’s covered by the primitive being rendered. Antialiased rendering reduces or eliminates the jaggies that result from aliased rendering.

application-specific clipping
Clipping of primitives against planes in eye coordinates; the planes are specified by the application using glClipPlane().

attribute group
A set of related state variables, which OpenGL can save or restore together at one time.

back faces
See faces.

bit
Binary digit. A state variable having only two possible values: 0 or 1. Binary numbers are constructions of one or more bits.
A rectangular array of bits. Also, the primitive rendered by the `glBitmap()` command, which uses its `bitmap` parameter as a mask.

A rectangular array of bits mapped one-to-one with pixels. The framebuffer is a stack of bitplanes.

Reduction of two color components to one component, usually as a linear interpolation between the two components.

A group of bitplanes that store a single component (such as depth or green) or a single index (such as the color index or the stencil index). Sometimes the red, green, blue, and alpha buffers together are referred to as the color buffer, rather than the color buffers.

God’s programming language.

The object-oriented programming language of a pagan deity.

The computer from which OpenGL commands are issued. The computer that issues OpenGL commands can be connected via a network to a different computer that executes the commands, or commands can be issued and executed on the same computer. See also server.

The main memory (where program variables are stored) of the client computer.

The coordinate system that follows transformation by the projection matrix and precedes perspective division. View-volume clipping is done in clip coordinates, but application-specific clipping is not.

Elimination of the portion of a geometric primitive that’s outside the half-space defined by a clipping plane. Points are simply rejected if outside. The portion of a line or of a polygon that’s outside the half-space is eliminated, and additional vertices are generated as necessary to complete the primitive within the clipping half-space. Geometric primitives and the current raster position (when specified) are always clipped against the six half-spaces defined by the left, right, bottom, top, near, and far planes of the view volume. Applications can specify optional application-specific clipping planes to be applied in eye coordinates.

A single value that represents a color by name, rather than by value. OpenGL color indices are treated as continuous values (for example, floating-point numbers), while operations such as interpolation and dithering are performed on them. Color indices stored in the framebuffer are always integer values,
however. Floating-point indices are converted to integers by rounding to the nearest integer value.

**color−index mode**
An OpenGL context is in color−index mode if its color buffers store color indices rather than red, green, blue, and alpha color components.

**color map**
A table of index−to−RGB mappings that’s accessed by the display hardware. Each color index is read from the color buffer, converted to an RGB triple by lookup in the color map, and sent to the monitor.

**components**
Single, continuous (for example, floating−point) values that represent intensities or quantities. Usually, a component value of zero represents the minimum value or intensity, and a component value of one represents the maximum value or intensity, though other ranges are sometimes used. Because component values are interpreted in a normalized range, they are specified independent of actual resolution. For example, the RGB triple (1, 1, 1) is white, regardless of whether the color buffers store 4, 8, or 12 bits each.

Out−of−range components are typically clamped to the normalized range, not truncated or otherwise interpreted. For example, the RGB triple (1.4, 1.5, 0.9) is clamped to (1.0, 1.0, 0.9) before it’s used to update the color buffer. Red, green, blue, alpha, and depth are always treated as components, never as indices.

**concave**
Not convex.

**context**
A complete set of OpenGL state variables. Note that framebuffer contents are not part of OpenGL state, but that the configuration of the framebuffer is.

**convex**
A polygon is convex if no straight line in the plane of the polygon intersects the polygon more than twice.

**convex hull**
The smallest convex region enclosing a specified group of points. In two dimensions, the convex hull is found conceptually by stretching a rubber band around the points so that all of the points lie within the band.

**coordinate system**
In $n$−dimensional space, a set of $n$ linearly independent vectors anchored to a point (called the origin). A group of coordinates specifies a point in space (or a vector from the origin) by indicating how far to travel along each vector to reach the point (or tip of the vector).

**culling**
The process of eliminating a front face or back face of a polygon so that it isn’t drawn.

**current matrix**
A matrix that transforms coordinates in one coordinate system to coordinates of another system. There
are three current matrices in OpenGL: the modelview matrix transforms object coordinates (coordinates specified by the programmer) to eye coordinates; the perspective matrix transforms eye coordinates to clip coordinates; the texture matrix transforms specified or generated texture coordinates as described by the matrix. Each current matrix is the top element on a stack of matrices. Each of the three stacks can be manipulated with OpenGL matrix—manipulation commands.

**current raster position**
A window coordinate position that specifies the placement of an image primitive when it’s rasterized. The current raster position and other current raster parameters are updated when `glRasterPos()` is called.

**decal**
A method of calculating color values during texture application, where the texture colors replace the fragment colors or, if alpha blending is enabled, the texture colors are blended with the fragment colors, using only the alpha value.

**depth**
Generally refers to the \( z \) window coordinate.

**depth buffer**
Memory that stores the depth value at every pixel. To perform hidden—surface removal, the depth buffer records the depth value of the object that lies closest to the observer at every pixel. The depth value of every new fragment uses the recorded value for depth comparison and must pass the comparison test before being rendered.

**depth—cuing**
A rendering technique that assigns color based on distance from the viewpoint.

**diffuse**
Diffuse lighting and reflection accounts for the directionality of a light source. The intensity of light striking a surface varies with the angle between the orientation of the object and the direction of the light source. A diffuse material scatters that light evenly in all directions.

**directional light source**
See infinite light source.

**display list**
A named list of OpenGL commands. Display lists are always stored on the server, so display lists can be used to reduce network traffic in client–server environments. The contents of a display list may be preprocessed and might therefore execute more efficiently than the same set of OpenGL commands executed in immediate mode. Such preprocessing is especially important for computing intensive commands such as NURBS or polygon tessellation.

**dithering**
A technique for increasing the perceived range of colors in an image at the cost of spatial resolution. Adjacent pixels are assigned differing color values; when viewed from a distance, these colors seem to blend into a single intermediate color. The technique is similar to the halftoning used in black—and–white publications to achieve shades of gray.
double−buffering
OpenGL contexts with both front and back color buffers are double−buffered. Smooth animation is accomplished by rendering into only the back buffer (which isn’t displayed), then causing the front and back buffers to be swapped. See glutSwapBuffers() in Appendix D.

edge flag
A Boolean value at a vertex which marks whether that vertex precedes a boundary edge. glEndEdgeFlag*() may be used to mark an edge as not on the boundary. When a polygon is drawn in GL_LINE mode, only boundary edges are drawn.

element
A single component or index.

emission
The color of an object which is self−illuminating or self−radiating. The intensity of an emissive material is not attributed to any external light source.

evaluated
The OpenGL process of generating object−coordinate vertices and parameters from previously specified Bézier equations.

execute
An OpenGL command is executed when it’s called in immediate mode or when the display list that it’s a part of is called.

eye coordinates
The coordinate system that follows transformation by the modelview matrix and precedes transformation by the projection matrix. Lighting and application−specific clipping are done in eye coordinates.

faces
The sides of a polygon. Each polygon has two faces: a front face and a back face. Only one face or the other is ever visible in the window. Whether the back or front face is visible is effectively determined after the polygon is projected onto the window. After this projection, if the polygon’s edges are directed clockwise, one of the faces is visible; if directed counterclockwise, the other face is visible. Whether clockwise corresponds to front or back (and counterclockwise corresponds to back or front) is determined by the OpenGL programmer.

flat shading
Refers to a primitive colored with a single, constant color across its extent, rather than smoothly interpolated colors across the primitive. See Gouraud shading.

fog
A rendering technique that can be used to simulate atmospheric effects such as haze, fog, and smog by fading object colors to a background color based on distance from the viewer. Fog also aids in the perception of distance from the viewer, giving a depth cue.
Groups of graphical character representations generally used to display strings of text. The characters may be roman letters, mathematical symbols, Asian ideograms, Egyptian hieroglyphics, and so on.

**fragment**
Fragments are generated by the rasterization of primitives. Each fragment corresponds to a single pixel and includes color, depth, and sometimes texture-coordinate values.

**framebuffer**
All the buffers of a given window or context. Sometimes includes all the pixel memory of the graphics hardware accelerator.

**front faces**
See faces.

**frustum**
The view volume warped by perspective division.

**gamma correction**
A function applied to colors stored in the framebuffer to correct for the nonlinear response of the eye (and sometimes of the monitor) to linear changes in color-intensity values.

**geometric model**
The object-coordinate vertices and parameters that describe an object. Note that OpenGL doesn’t define a syntax for geometric models, but rather a syntax and semantics for the rendering of geometric models.

**geometric object**
See geometric model.

**geometric primitive**
A point, a line, or a polygon.

**Gouraud shading**
Smooth interpolation of colors across a polygon or line segment. Colors are assigned at vertices and linearly interpolated across the primitive to produce a relatively smooth variation in color. Also called smooth shading.

**group**
Each pixel of an image in client memory is represented by a group of one, two, three, or four elements. Thus, in the context of a client memory image, a group and a pixel are the same thing.

**half−spaces**
A plane divides space into two half−spaces.

**hidden−line removal**
A technique to determine which portions of a wireframe object should be visible. The lines that comprise the wireframe are considered to be edges of opaque surfaces, which may obscure other edges that are farther away from the viewer.

**hidden−surface removal**
A technique to determine which portions of an opaque, shaded object should be visible and which portions should be obscured. A test of the depth coordinate, using the depth buffer for storage, is a common method of hidden-surface removal.

**homogeneous coordinates**
A set of \( n+1 \) coordinates used to represent points in \( n \)-dimensional projective space. Points in projective space can be thought of as points in euclidean space together with some points at infinity. The coordinates are homogeneous because a scaling of each of the coordinates by the same nonzero constant doesn’t alter the point to which the coordinates refer. Homogeneous coordinates are useful in the calculations of projective geometry, and thus in computer graphics, where scenes must be projected onto a window.

**image**
A rectangular array of pixels, either in client memory or in the framebuffer.

**image primitive**
A bitmap or an image.

**immediate mode**
Execution of OpenGL commands when they’re called, rather than from a display list. No immediate-mode bit exists; the mode in immediate mode refers to use of OpenGL, rather than to a specific bit of OpenGL state.

**index**
A single value that’s interpreted as an absolute value, rather than as a normalized value in a specified range (as is a component). Color indices are the names of colors, which are dereferenced by the display hardware using the color map. Indices are typically masked rather than clamped when out of range. For example, the index 0xf7 is masked to 0x7 when written to a 4-bit buffer (color or stencil). Color indices and stencil indices are always treated as indices, never as components.

**indices**
Preferred plural of index. (The choice between the plural forms indices or indexes—as well as matrices or matrixes and vertices or vertexes—has engendered much debate between the authors and principal reviewers of this guide. The authors’ compromise solution is to use the –ices form but to state clearly for the record that the use of indice [sic], matrice [sic], and vertice [sic] for the singular forms is an abomination.)

**infinite light source**
A directional source of illumination. The radiating light from an infinite light source strikes all objects as parallel rays.

**interpolation**
Calculation of values (such as color or depth) for interior pixels, given the values at the boundaries (such as at the vertices of a polygon or a line).

**IRIS GL**
Silicon Graphics proprietary graphics library, developed from 1982 through 1992. OpenGL was designed...
with IRIS GL as a starting point.

**IRIS Inventor**
See Open Inventor.

**jaggies**
Artifacts of aliased rendering. The edges of primitives that are rendered with aliasing are jagged rather than smooth. A near-horizontal aliased line, for example, is rendered as a set of horizontal lines on adjacent pixel rows rather than as a smooth, continuous line.

**jittering**
A pseudo-random displacement (shaking) of the objects in a scene, used in conjunction with the accumulation buffer to achieve special effects.

**lighting**
The process of computing the color of a vertex based on current lights, material properties, and lighting-model modes.

**line**
A straight region of finite width between two vertices. (Unlike mathematical lines, OpenGL lines have finite width and length.) Each segment of a strip of lines is itself a line.

**local light source**
A source of illumination which has an exact position. The radiating light from a local light source emanates from that position. Other names for a local light source are point light source or positional light source. A spotlight is a special kind of local light source.

**logical operation**
Boolean mathematical operations between the incoming fragment’s RGBA color or color-index values and the RGBA color or color-index values already stored at the corresponding location in the framebuffer. Examples of logical operations include AND, OR, XOR, NAND, and INVERT.

**luminance**
The perceived brightness of a surface. Often refers to a weighted average of red, green, and blue color values that gives the perceived brightness of the combination.

**matrices**
Preferred plural of matrix. See indices.

**matrix**
A two-dimensional array of values. OpenGL matrices are all 4×4, though when stored in client memory they’re treated as 1×16 single-dimension arrays.

**modelview matrix**
The 4×4 matrix that transforms points, lines, polygons, and raster positions from object coordinates to eye coordinates.

**modulate**
A method of calculating color values during texture application, where the texture and the fragment colors are combined.

**monitor**
The device that displays the image in the framebuffer.

**motion blurring**
A technique that uses the accumulation buffer to simulate what appears on film when you take a picture of a moving object or when you move the camera while taking a picture of a stationary object. In animations without motion blur, moving objects can appear jerky.

**network**
A connection between two or more computers that allows each to transfer data to and from the others.

**nonconvex**
A polygon is nonconvex if there exists a line in the plane of the polygon that intersects the polygon more than twice.

**normal**
A three–component plane equation that defines the angular orientation, but not position, of a plane or surface.

**normalized**
To normalize a normal vector, divide each of the components by the square root of the sum of their squares. Then, if the normal is thought of as a vector from the origin to the point \((nx', ny', nz')\), this vector has unit length.

\[
\text{factor} = \sqrt{nx^2 + ny^2 + nz^2}
\]
\[
nx' = nx / \text{factor}
\]
\[
ny' = ny / \text{factor}
\]
\[
nz' = nz / \text{factor}
\]

**normal vectors**
See normal.

**NURBS**
Non–Uniform Rational B–Spline. A common way to specify parametric curves and surfaces. (See GLU NURBS routines in Chapter 12.)

**object**
An object–coordinate model that’s rendered as a collection of primitives.

**object coordinates**
Coordinate system prior to any OpenGL transformation.

**Open Inventor**
An object–oriented 3D toolkit, built on top of OpenGL, based on a 3D scene database and user interaction components. It includes objects such as cubes, polygons, text, materials, cameras, lights, trackballs and handle boxes.
orthographic
Nonperspective projection, as in some engineering drawings, with no foreshortening.

parameters
Values passed as arguments to OpenGL commands. Sometimes parameters are passed by reference to an OpenGL command.

perspective division
The division of x, y, and z by w, carried out in clip coordinates.

pixel
Picture element. The bits at location (x, y) of all the bitplanes in the framebuffer constitute the single pixel (x, y). In an image in client memory, a pixel is one group of elements. In OpenGL window coordinates, each pixel corresponds to a 1.0×1.0 screen area. The coordinates of the lower–left corner of the pixel are x,y are (x, y), and of the upper–right corner are (x+1, y+1).

point
An exact location in space, which is rendered as a finite–diameter dot.

point light source
See local light source.

polygon
A near–planar surface bounded by edges specified by vertices. Each triangle of a triangle mesh is a polygon, as is each quadrilateral of a quadrilateral mesh. The rectangle specified by glRect*() is also a polygon.

positional light source
See local light source.

primitive
A point, a line, a polygon, a bitmap, or an image. (Note: Not just a point, a line, or a polygon!)

projection matrix
The 4×4 matrix that transforms points, lines, polygons, and raster positions from eye coordinates to clip coordinates.

proxy texture
A placeholder for a texture image, which is used to determine if there are enough resources to support a texture image of a given size and internal format resolution.

quadrilateral
A polygon with four edges.

rasterized
Converted a projected point, line, or polygon, or the pixels of a bitmap or image, to fragments, each corresponding to a pixel in the framebuffer. Note that all primitives are rasterized, not just points, lines, and polygons.
**rectangle**
A quadrilateral whose alternate edges are parallel to each other in object coordinates. Polygons specified with `glRect*()` are always rectangles; other quadrilaterals might be rectangles.

**rendering**
Conversion of primitives specified in object coordinates to an image in the framebuffer. Rendering is the primary operation of OpenGL—it’s what OpenGL does.

**resident texture**
A texture image that is cached in special, high-performance texture memory. If an OpenGL implementation does not have special, high-performance texture memory, then all texture images are deemed resident textures.

**RGBA**
Red, Green, Blue, Alpha.

**RGBA mode**
An OpenGL context is in RGBA mode if its color buffers store red, green, blue, and alpha color components, rather than color indices.

**server**
The computer on which OpenGL commands are executed. This might differ from the computer from which commands are issued. See client.

**shading**
The process of interpolating color within the interior of a polygon, or between the vertices of a line, during rasterization.

**shininess**
The exponent associated with specular reflection and lighting. Shininess controls the degree with which the specular highlight decays.

**single-buffering**
OpenGL contexts that don’t have back color buffers are single-buffered. You can use these contexts for animation, but take care to avoid visually disturbing flashes when rendering.

**singular matrix**
A matrix that has no inverse. Geometrically, such a matrix represents a transformation that collapses points along at least one line to a single point.

**specular**
Specular lighting and reflection incorporates reflection off shiny objects and the position of the viewer. Maximum specular reflectance occurs when the angle between the viewer and the direction of the reflected light is zero. A specular material scatters light with greatest intensity in the direction of the reflection, and its brightness decays, based upon the exponential value shininess.

**spotlight**
A special type of local light source that has a direction (where it points to) as well as a position. A
spotlight simulates a cone of light, which may have a fall–off in intensity, based upon distance from the center of the cone.

**stencil buffer**
Memory (bitplanes) that is used for additional per–fragment testing, along with the depth buffer. The stencil test may be used for masking regions, capping solid geometry, and overlapping translucent polygons.

**stereo**
Enhanced three–dimensional perception of a rendered image by computing separate images for each eye. Stereo requires special hardware, such as two synchronized monitors or special glasses to alternate viewed frames for each eye. Some implementations of OpenGL support stereo by having both left and right buffers for color data.

**stipple**
A one– or two–dimensional binary pattern that defeats the generation of fragments where its value is zero. Line stipple are one–dimensional and are applied relative to the start of a line. Polygon stipple are two–dimensional and are applied with a fixed orientation to the window.

**tessellation**
Reduction of a portion of an analytic surface to a mesh of polygons, or of a portion of an analytic curve to a sequence of lines.

**texel**
A texture element. A texel is obtained from texture memory and represents the color of the texture to be applied to a corresponding fragment.

**textures**
One– or two–dimensional images that are used to modify the color of fragments produced by rasterization.

**texture mapping**
The process of applying an image (the texture) to a primitive. Texture mapping is often used to add realism to a scene. For example, you can apply a picture of a building facade to a polygon representing a wall.

**texture matrix**
The $4 \times 4$ matrix that transforms texture coordinates from the coordinates in which they’re specified to the coordinates that are used for interpolation and texture lookup.

**texture object**
A named cache that stores texture data, such as the image array, associated mipmaps, and associated texture parameter values: width, height, border width, internal format, resolution of components, minification and magnification filters, wrapping modes, border color, and texture priority.

**transformations**
The warping of spaces. In OpenGL, transformations are limited to projective transformations that include
anything that can be represented by a 4×4 matrix. Such transformations include rotations, translations, (nonuniform) scalings along the coordinate axes, perspective transformations, and combinations of these.

triangle
A polygon with three edges. Triangles are always convex.

vertex
A point in three−dimensional space.

vertex array
Where a block of vertex data (vertex coordinates, texture coordinates, surface normals, RGBA colors, color indices, and edge flags) may be stored in an array and then used to specify multiple geometric primitives through the execution of a single OpenGL command.

vertices
Preferred plural of vertex. See indices.

viewpoint
The origin of either the eye− or the clip−coordinate system, depending on context. (For example, when discussing lighting, the viewpoint is the origin of the eye−coordinate system. When discussing projection, the viewpoint is the origin of the clip−coordinate system.) With a typical projection matrix, the eye−coordinate and clip−coordinate origins are at the same location.

view volume
The volume in clip coordinates whose coordinates satisfy the three conditions
\[-w ≤ x ≤ w\]
\[-w ≤ y ≤ w\]
\[-w ≤ z ≤ w\]
Geometric primitives that extend outside this volume are clipped.

VRML
VRML stands for Virtual Reality Modeling Language, which is (according to the VRML Mission Statement) "a universal description language for multi−participant simulations." VRML is specifically designed to allow people to navigate through three−dimensional worlds that are placed on the World Wide Web. The first versions of VRML are subsets of the Open Inventor file format with additions to allow hyperlinking to the Web (to URLs—Universal Resource Locators).

window
A subregion of the framebuffer, usually rectangular, whose pixels all have the same buffer configuration. An OpenGL context renders to a single window at a time.

window−aligned
When referring to line segments or polygon edges, implies that these are parallel to the window boundaries. (In OpenGL, the window is rectangular, with horizontal and vertical edges). When referring to a polygon pattern, implies that the pattern is fixed relative to the window origin.

window coordinates
The coordinate system of a window. It’s important to distinguish between the names of pixels, which are discrete, and the window-coordinate system, which is continuous. For example, the pixel at the lower−left corner of a window is pixel (0, 0); the window coordinates of the center of this pixel are (0.5, 0.5, z). Note that window coordinates include a depth, or z, component, and that this component is continuous as well.

**wireframe**
A representation of an object that contains line segments only. Typically, the line segments indicate polygon edges.

**working set**
On machines with special hardware that increases texture performance, this is the group of texture objects that are currently resident. The performance of textures within the working set outperforms the textures outside the working set.

**X Window System**
A window system used by many of the machines on which OpenGL is implemented. GLX is the name of the OpenGL extension to the X Window System. (See Appendix C.)