Chapter 2

Drawing Primitives

Computer-generated animation in film and television, as well as state-of-the-art video games, features realistic water, fire, and other natural effects. Many people new to computer graphics are astounded to learn that these realistic and complex models are simple triangles and pixels as far as computer graphics hardware is concerned.

OpenGL is often referred to as a low-level API because of its minimal support for higher-order primitives, data structures such as scene graphs, or support for loading 2D image files or 3D model files. Instead, OpenGL focuses on rendering low-level primitives efficiently and with a variety of basic, yet flexible, rendering features. As a result of this "tools not rules" approach, OpenGL is the preferred low-level API for a variety of middleware and applications that feature higher-order primitives, scene graph data structures, and file loaders.

In this chapter, *OpenGL® Distilled* covers the OpenGL primitive types and how to control their appearance with several basic rendering features.

What You'll Learn

This chapter covers the following aspects of primitive rendering:

- Primitive types—The ten primitive types for rendering point, line, and polygonal primitives.
- Buffer objects and vertex arrays—Generally recognized as the most efficient method for specifying geometry.
- Rendering details—OpenGL commands for hidden surface removal, transparency, and displaying co-planar primitives.

- Performance issues—Some tips to help your application run as efficiently as possible on most OpenGL implementations.

What You Won't Learn

Because this book presents only OpenGL's most essential commands, several aspects of primitive rendering aren't covered in this chapter:

- The **`glBegin`**`()`/**`glEnd`**`()` paradigm—*OpenGL® Distilled* covers the **`glBegin`**`()`/**`glEnd`**`()` paradigm for illustrative purposes only. Most OpenGL implementations avoid using **`glBegin`**`()`/**`glEnd`**`()` to specify geometry due to its inherent performance issues.
- Vertex data—This chapter covers normal and texture-coordinate data and omits other vertex data, such as vertex attributes (used in vertex shaders), edge flags, and fog coordinates.
- Mapping and unmapping buffer objects—This chapter doesn't discuss the interface for dynamically altering portions of buffer object data.
- Evaluators—OpenGL allows programmers to render implicit curves and surfaces from control points.
- Rectangles—Because you can specify vertices to render any desired shape, this shorthand interface for drawing rectangles in the z=0 plane is rarely used.
- Full vertex array functionality—This book presents a subset of the vertex array interface and doesn't cover interleaved arrays; vertex array data types other than `GL_FLOAT` and `GL_DOUBLE`; and some vertex array rendering commands, such as **`glDrawArrays`**`()`.
- This book doesn't cover all features that affect the final color and appearance of rendered geometry, such as fog, stencil, vertex and fragment shaders, and other related features.

Though useful in many rendering circumstances, these features aren't essential for OpenGL programming. If your application requires this functionality, see *OpenGL® Programming Guide, OpenGL® Reference Manual,* and *OpenGL® Shading Language*.

2.1 OpenGL Primitives

In OpenGL, applications render primitives by specifying a primitive type and a sequence of vertices with associated data. The primitive type determines how OpenGL interprets and renders the sequence of vertices.

2.1.1 Primitive Types

OpenGL provides ten different primitive types for drawing points, lines, and polygons, as shown in Figure 2-1.

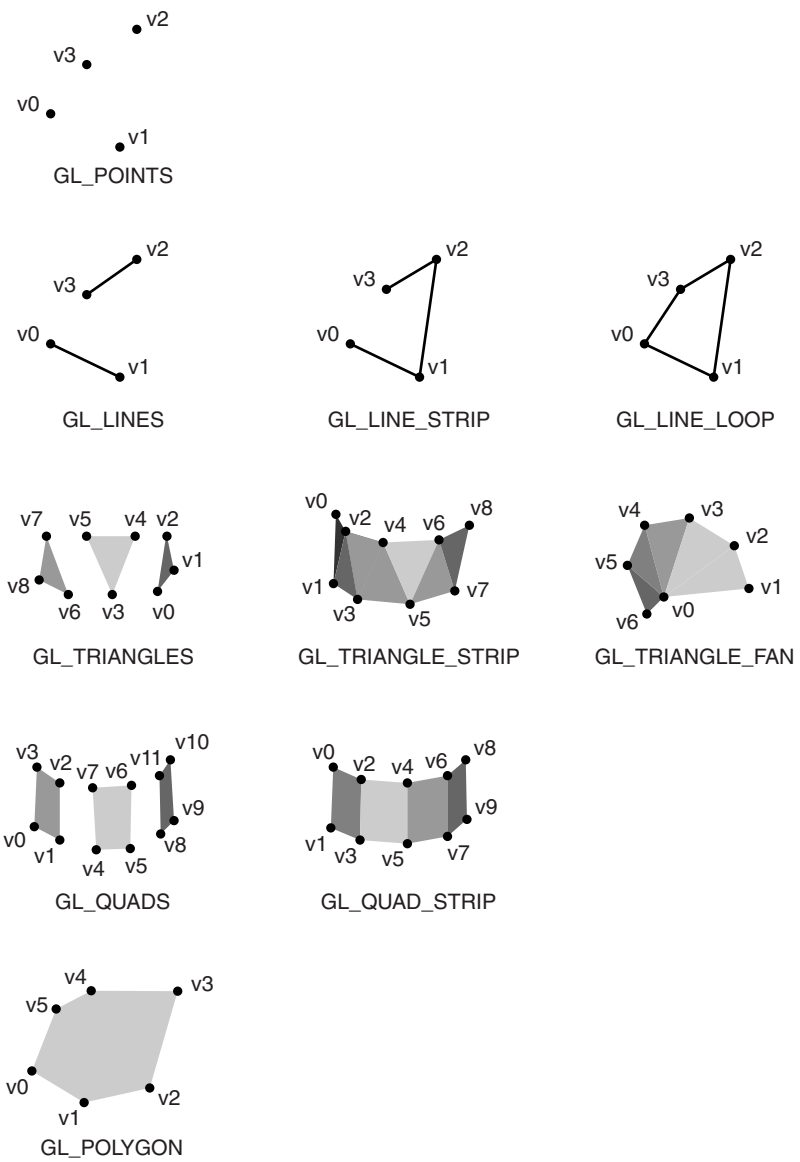

Figure 2-1 OpenGL primitive types.

OpenGL interprets the vertices and renders each primitive using the following rules:

- `GL_POINTS`—Use this primitive type to render mathematical points. OpenGL renders a point for each vertex specified.
- `GL_LINES`—Use this primitive to draw unconnected line segments. OpenGL draws a line segment for each group of two vertices. If the application specifies *n* vertices, OpenGL renders *n*/2 line segments. If *n* is odd, OpenGL ignores the final vertex.
- `GL_LINE_STRIP`—Use this primitive to draw a sequence of connected line segments. OpenGL renders a line segment between the first and second vertices, between the second and third, between the third and fourth, and so on. If the application specifies *n* vertices, OpenGL renders *n*–1 line segments.
- `GL_LINE_LOOP`—Use this primitive to close a line strip. OpenGL renders this primitive like a `GL_LINE_STRIP` with the addition of a closing line segment between the final and first vertices.
- `GL_TRIANGLES`—Use this primitive to draw individual triangles. OpenGL renders a triangle for each group of three vertices. If your application specifies *n* vertices, OpenGL renders *n*/3 triangles. If *n* isn't a multiple of 3, OpenGL ignores the excess vertices.
- `GL_TRIANGLE_STRIP`—Use this primitive to draw a sequence of triangles that share edges. OpenGL renders a triangle using the first, second, and third vertices, and then another using the second, third, and fourth vertices, and so on. If the application specifies *n* vertices, OpenGL renders *n*–2 connected triangles. If *n* is less than 3, OpenGL renders nothing.
- `GL_TRIANGLE_FAN`—Use this primitive to draw a fan of triangles that share edges and also share a vertex. Each triangle shares the first vertex specified. If the application specifies a sequence of vertices *v,* OpenGL renders a triangle using v_0, v_1, and v_2; another triangle using v_0, v_2, and v_3; another triangle using v_0, v_3, and v_4; and so on. If the application specifies *n* vertices, OpenGL renders *n*–2 connected triangles. If *n* is less than 3, OpenGL renders nothing.
- `GL_QUADS`—Use this primitive to draw individual convex quadrilaterals. OpenGL renders a quadrilateral for each group of four vertices. If the application specifies *n* vertices, OpenGL renders *n*/4 quadrilaterals. If *n* isn't a multiple of 4, OpenGL ignores the excess vertices.

- GL_QUAD_STRIP—Use this primitive to draw a sequence of quadrilaterals that share edges. If the application specifies a sequence of vertices *v,* OpenGL renders a quadrilateral using v_0, v_1, v_3, and v_2; another quadrilateral using v_2, v_3, v_5, and v_4; and so on. If the application specifies *n* vertices, OpenGL renders (*n*-2)/2 quadrilaterals. If *n* is less than 4, OpenGL renders nothing.
- GL_POLYGON—Use GL_POLYGON to draw a single filled convex *n*-gon primitive. OpenGL renders an *n*-sided polygon, where *n* is the number of vertices specified by the application. If *n* is less than 3, OpenGL renders nothing.

For GL_QUADS, GL_QUAD_STRIP, and GL_POLYGON, all primitives must be both planar and convex. Otherwise, OpenGL behavior is undefined. The GLU library supports polygon tessellation, which allows applications to render filled primitives that are nonconvex or self-intersecting, or that contain holes. See the "gluTess" set of functions in *OpenGL® Reference Manual* for more information.

2.1.2 Vertex Sharing

Note that GL_LINE_STRIP, GL_LINE_LOOP, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, and GL_QUAD_STRIP all share vertices among their component line segments, triangles, and quadrilaterals. In general, you should use these primitives when possible and practical to reduce redundant per-vertex computation.

You could render a two-quadrilateral GL_QUAD_STRIP primitive by using GL_QUADS, for example. Rendered as a GL_QUAD_STRIP, your application would need to send only six unique vertices. The GL_QUADS version of this primitive, however, would require eight vertices, two of which are redundant. Passing identical vertices to OpenGL increases the number of per-vertex operations and could create a performance bottleneck in the rendering pipeline.

2.2 Specifying Vertex Data

OpenGL allows the application to specify primitives in several ways. *OpenGL® Distilled* briefly covers the **glBegin**()/**glEnd**() paradigm for illustrative purposes only. You should avoid using **glBegin**()/**glEnd**() because of its inherent call overhead, which inhibits application performance. Instead,

use buffer objects and vertex arrays, presented later in this chapter. Vertex arrays dramatically reduce function call overhead compared with **glBegin**()/**glEnd**() and also allow vertex sharing. Using buffer objects causes OpenGL to store vertex data in high-performance server memory, which allows your application to avoid expensive data copies at render time.

2.2.1 Drawing Primitives Using glBegin()/glEnd()

OpenGL version 1.0 features a flexible interface for primitive rendering called the **glBegin**()/**glEnd**() paradigm. Contemporary OpenGL features more efficient rendering mechanisms, which are the focus of this chapter. Because OpenGL is backward compatible with older versions, however, many applications still use the **glBegin**()/**glEnd**() paradigm in spite of its inherent performance issues. This chapter covers **glBegin**()/**glEnd**() briefly because it illustrates the OpenGL concept of per-vertex state.

Applications render primitives by surrounding vertices with a pair of functions, **glBegin**() and **glEnd**(). Applications specify the primitive type by passing it as a parameter to **glBegin**().

---

void **glBegin**(GLenum *mode*);
void **glEnd**(void);

---

glBegin() and **glEnd**() surround commands that specify vertices and vertex data. *mode* specifies the primitive type to draw and must be one of GL_POINTS, GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP, GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_QUADS, GL_QUAD_STRIP, or GL_POLYGON.

▶ OpenGL version: 1.0 and later.

Between **glBegin**() and **glEnd**(), applications specify vertices and vertex states such as the current primary color, current normal and material properties (for lighting), and current texture coordinates (for texture mapping).[1] Applications specify colors, normals, material properties, texture coordinates, and vertices each with individual function calls. *OpenGL® Distilled* doesn't cover these function calls in detail, but their names are self-explanatory, as you'll see in the following examples.

1. OpenGL allows the application to specify a large amount of per-vertex states not covered in this book, such as color indices, fog coordinates, edge flags, and vertex attributes. See *OpenGL® Programming Guide* for more information.

The **glBegin**()/**glEnd**() paradigm serves as an excellent illustration of OpenGL per-vertex state. The following code, for example, demonstrates how to draw a red triangle:

```
glBegin( GL_TRIANGLES );
  glColor3f( 1.f, 0.f, 0.f ); // Sets current primary color to red
  glVertex3f( 0.f, 0.f, 0.f ); // Specify three vertices
  glVertex3f( 1.f, 0.f, 0.f );
  glVertex3f( 0.f, 1.f, 0.f );
glEnd();
```

Recall that OpenGL is a state machine. The **glColor3f**() command sets the current primary color state using the RGBA value (1.0, 0.0, 0.0, 1.0) for red (the alpha value 1.0 is implicit in the **glColor3f**() command). As OpenGL receives **glVertex3f**() commands, it copies the current primary color state into the state associated with each vertex.

An application can set different states at each vertex. Consider the following example, which draws a triangle with red, green, and blue vertices:

```
glBegin( GL_TRIANGLES );
  glColor3f( 1.f, 0.f, 0.f );
  glVertex3f( 0.f, 0.f, 0.f );
  glColor3f( 0.f, 1.f, 0.f );
  glVertex3f( 1.f, 0.f, 0.f );
  glColor3f( 0.f, 0.f, 1.f );
  glVertex3f( 0.f, 1.f, 0.f );
glEnd();
```

In this case, the current primary color state is different at the time OpenGL receives each **glVertex3f**() command. As a result, the state associated with the first vertex has red stored in its primary color, with green and blue stored in the state for the subsequent vertices.

Note When specifying a primitive with different primary colors at each vertex, your application can direct OpenGL to interpolate those colors linearly or render the primitive using only the final vertex color. For more information, see "Smooth and Flat Shading" later in this chapter.

OpenGL doesn't limit the number of vertices an application can specify between **glBegin**() and **glEnd**(). If your application calls **glBegin**(GL_QUADS), followed by four million vertices and **glEnd**(), for example, OpenGL renders 1 million individual quadrilaterals from your vertex data.

Individual function calls to specify vertices and vertex states provide great flexibility for the application developer. The function call overhead inherent in this interface, however, dramatically limits application performance. Even though display lists (see "Performance Issues" later in this chapter) allow implementations to optimize data sent using the **glBegin**()/**glEnd**()

paradigm, many OpenGL vendors expect that developers will simply use mechanisms that are inherently more efficient and easier to optimize, such as buffer objects and vertex arrays. For this reason, avoid using the **glBegin**()/**glEnd**() paradigm.

Regardless, many applications use **glBegin**()/**glEnd**() extensively, possibly for historical reasons. Before OpenGL version 1.1 was released, the **glBegin**()/**glEnd**() paradigm was the only option available. Performance may be another reason; **glBegin**()/**glEnd**() is perfectly adequate for applications that have extremely light rendering requirements or are not performance critical.

2.2.2 Drawing Primitives Using Vertex Arrays

Vertex arrays allow applications to render primitives from vertex and vertex state data stored in blocks of memory. Under some circumstances, OpenGL implementations can process the vertices and cache the results for efficient reuse. Applications specify primitives by indexing into the vertex array data.

Vertex arrays, a common OpenGL extension in version 1.0, became part of the OpenGL core in version 1.1, with additional feature enhancements through version 1.4. Before version 1.5, applications using vertex arrays could store data only in client storage. Version 1.5 introduced the buffer object feature, allowing applications to store data in high-performance server memory.

This book focuses primarily on using buffer objects. The section "Vertex Array Data" later in this chapter, however, shows how to specify blocks of data with and without buffer objects. The example source code uses buffer objects when the OpenGL runtime version is 1.5 or later and uses the pre-1.5 interface otherwise.

2.2.2.1 Buffer Objects

Applications use buffer objects to store vertex data[2] in server memory.

Each buffer object requires a unique identifier. Obtain a buffer object identifier with **glGenBuffers**().

2. The GL_ARB_pixel_buffer_object extension allows buffer objects to contain pixel data so that applications can store blocks of pixel data in server memory. See Chapter 7, "Extensions and Versions," for an example. This functionality may become part of OpenGL version 2.1.

void **glGenBuffers**(GLsizei *n*, GLuint* *buffers*);
GLboolean **glIsBuffer**(GLuint *buffer*);

glGenBuffers() obtains *n* buffer identifiers from its pool of unused buffers. *n* is the number of buffer identifiers desired by the application. **glGenBuffers**() stores the identifiers in *buffers*.

glIsBuffer() returns GL_TRUE if *buffer* is an existing buffer object.

▶ OpenGL version: 1.5 and later.

You might need to create four buffer objects to store vertex data, normal data, texture coordinate data, and indices in vertex arrays. The following code obtains identifiers for four buffer objects:

```
GLuint bufObjects[4];
glGenBuffers( 4, bufObjects );
```

Before you can store data in the buffer object, your application must bind it using **glBindBuffer**().

void **glBindBuffer**(GLenum *target*, GLuint *buffer*);

Specifies the active buffer object and initializes it if it's new. Pass a *target* value of GL_ARRAY_BUFFER to use the buffer object as vertex array data, and pass a *target* value of GL_ELEMENT_ARRAY_BUFFER to use the buffer object as vertex array indices. *buffer* is the identifier of the buffer object to bind.

▶ OpenGL version: 1.5 and later.

The command **glBindBuffer**(GL_ARRAY_BUFFER, bufObjects[0]) binds the first buffer object ID obtained in the previous code listing, bufObjects[0], for use with vertex array data. If a buffer object is already bound, **glBindBuffer**() unbinds it and then binds *buffer*.

To unbind a buffer object, call **glBindBuffer**() with a buffer ID of zero. After **glBindBuffer**(GL_ARRAY_BUFFER, 0) is called, for example, no buffer is associated with vertex array data.

When an application binds a buffer object for the first time, OpenGL creates an empty buffer object. To load the buffer object with data, call **glBufferData**().

void **glBufferData**(GLenum *target*, GLsizeiptr *size*, const GLvoid* *data*, GLenum *usage*);

Copies data from host memory to the active buffer object. *target* must be `GL_ARRAY_BUFFER` or `GL_ELEMENT_ARRAY_BUFFER`; *size* indicates the size of the data in bytes; and *data* points to the data.

usage is a hint to OpenGL, stating the application's intended usage of the buffer data. A *usage* parameter of `GL_STATIC_DRAW` indicates that the application intends to specify the data once and use it to draw several times. Other values for *usage* are described in Section 2.9, "Buffer Objects," of *The OpenGL Graphics System*.

▶ OpenGL version: 1.5 and later.

OpenGL doesn't limit the amount of data you can store in a buffer object. Some implementations, however, can provide maximum performance only if buffer object data is below an implementation-specific size and must fall back to a slower rendering path when buffer objects are too large. Currently, OpenGL doesn't provide a mechanism to query this implementation-specific limit. OpenGL vendors often make this type of information available to developers in implementation-specific documentation, however.

Applications typically create buffer objects and load them with data at initialization time. You might create a single buffer object and load it with three vertices by using the following code:

```
// Obtain a buffer identifier from OpenGL
GLuint bufferID;
glGenBuffers( 1, &bufferID );

// Define three vertices to draw a right triangle.
const GLfloat vertices[] = {
    0.f, 0.f, 0.f,
    1.f, 0.f, 0.f,
    0.f, 1.f, 0.f };

// Bind the buffer object, OpenGL initially creates it empty.
glBindBuffer( GL_ARRAY_BUFFER, bufferID );
// Tell OpenGL to copy data from the 'vertices' pointer into
//   the buffer object.
glBufferData( GL_ARRAY_BUFFER, 3*3*sizeof(GLfloat), vertices,
    GL_STATIC_DRAW );
```

To render the three vertices stored in this buffer object as a triangle, your application must bind the buffer object before issuing the appropriate ver-

tex array pointer commands. See the next section, "Vertex Array Data," for additional information.

Some applications need to specify dynamic data. Because buffer objects exist in server memory, OpenGL provides an interface that allows applications to map the buffer object in client memory. This interface isn't covered in this book. For more information, see Section 2.9, "Buffer Objects," of *The OpenGL Graphics System*. Applications can replace all data in a buffer object with the **`glBufferData`**`()` command.

When your application no longer needs the buffer object, call **`glDeleteBuffers`**`()`. This command empties the specified buffers and places their IDs in OpenGL's pool of unused buffer object IDs.

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

Returns buffer object identifiers to the unused pool and deletes buffer object resources. The parameters are the same as for **`glGenBuffers`**`()`.

▶ OpenGL version: 1.5 and later.

Applications typically delete buffers when the application exits but should delete buffers to conserve server memory whenever the application no longer needs the buffer object.

2.2.2.2 Vertex Array Data

When rendering primitives with vertex arrays, your application must tell OpenGL where to obtain vertex array data. You can either submit the data directly or bind a buffer object, which tells OpenGL to obtain the data from that buffer. Both methods use the same interface.

void **glVertexPointer**(GLint *size*, GLenum *type*, GLsizei *stride*, const GLvoid* *pointer*);
void **glNormalPointer**(GLenum *type*, GLsizei *stride*, const GLvoid* *pointer*);
void **glTexCoordPointer**(GLint *size*, GLenum *type*, GLsizei *stride*, const GLvoid* *pointer*);

Submits arrays of vertices, normals, and texture coordinates to OpenGL for use with vertex arrays. *type* indicates the type of data being submitted. Applications typically use `GL_FLOAT` for single-precision vertices,

normals, and texture coordinates. If your application uses buffer objects, most OpenGL implementations optimize for `GL_FLOAT` data.[3]

stride lets you interleave data. If your data is tightly packed (noninterleaved), specify a *stride* of zero. Otherwise, specify a byte distance between the vertices, normals, or texture coordinates.

pointer points to your data or indicates an offset into a buffer object.

`glVertexPointer``()` and **`glTexCoordPointer`**`()` additionally take a *size* parameter. Use a *size* of 3 when sending 3D (*xyz*) vertices with **`glVertexPointer`**`()`. For 2D (*st*) texture coordinates, specify a *size* of 2 to **`glTexCoordPointer`**`()`. Because normals always consist of three elements, **`glNormalPointer`**`()` doesn't require a *size* parameter.

OpenGL supports sending other vertex data besides normals and texture coordinates. See *OpenGL® Programming Guide* for more information.

▶ OpenGL version: 1.1 and later.

To enable and disable vertex, normal, and texture coordinate arrays, call **`glEnableClientState`**`()` and **`glDisableClientState`**`()` with the parameters `GL_VERTEX_ARRAY`, `GL_NORMAL_ARRAY`, and `GL_TEXTURE_COORD_ARRAY`, respectively. If your application renders a vertex array primitive without enabling the normal or texture coordinate arrays, OpenGL renders the primitive without that data. Note that if your application fails to call **`glEnableClientState`**`( GL_VERTEX_ARRAY )`, OpenGL will render nothing; the vertex array must be enabled to render geometry with vertex arrays.

`glEnableClientState``()` and **`glDisableClientState`**`()` are similar in concept to **`glEnable`**`()` and **`glDisable`**`()`, except that the former enables and disables OpenGL client state features, whereas the latter enables and disables OpenGL server state features. See "glEnableClientState" in *OpenGL® Reference Manual* for more information.

Without Buffer Objects

Most programmers will code their applications to use buffer objects but also need to allow for the case in which buffer objects are unavailable, such

3. For color data, implementations typically optimize for `GL_UNSIGNED_BYTE`. See "glColorPointer" in *OpenGL® Reference Manual* for details on specifying color values per-vertex by using vertex arrays.

as when running on a pre-1.5 version of OpenGL. The example code that accompanies this book demonstrates both methods.

When not using buffer objects, the *pointer* parameter is a simple `GLvoid*` address of array data. The following code demonstrates how to enable and specify a vertex array by using **`glVertexPointer`**`()`:

```
const GLfloat data[] = {
    -1.f, -1.f, 0.f,
    1.f, -1.f, 0.f,
    0.f, 1.f, 0.f };

// Enable the vertex array and specify the data.
glEnableClientState( GL_VERTEX_ARRAY );
glVertexPointer( 3, GL_FLOAT, 0, data );
```

Although this may be simpler than using buffer objects, it requires OpenGL to copy the vertex array from client memory each time the application renders a primitive that uses it. This is illustrated in Figure 2-2.

The next section discusses buffer objects (the preferred method), which eliminate the copy from client memory to server memory. Buffer objects allow vertex array rendering commands to source vertex array data directly from high-performance server memory.

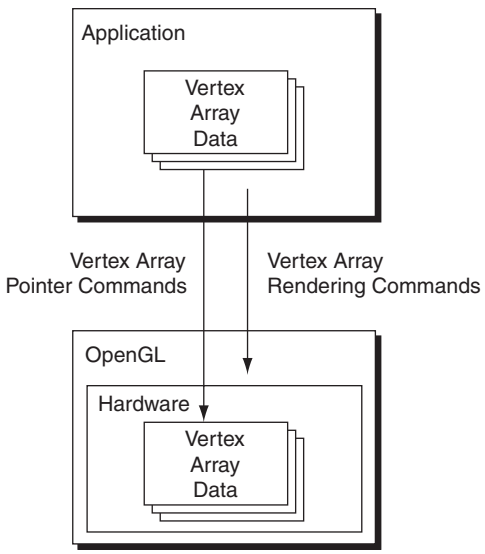

Figure 2-2 Rendering with vertex arrays before OpenGL version 1.5. OpenGL must copy the vertex array data each time the application renders primitives that use it.

With Buffer Objects

As shown in the preceding section, "Without Buffer Objects," before OpenGL version 1.5, applications could submit data only directly with these functions; the *pointer* parameter could only be an address in application-accessible memory, and OpenGL accessed the data from client memory to render primitives. OpenGL version 1.5 overloads these functions, however. If your application has bound a buffer object to `GL_ARRAY_BUFFER`, OpenGL interprets *pointer* as an offset into the buffer object data.

Because C doesn't allow function overloading, *pointer* must still be a `GLvoid*` address, even though the buffer object feature requires an offset. When obtaining data from a buffer object, OpenGL computes this offset by subtracting a pointer to NULL from *pointer,* where both addresses are treated as `char*`.

Applications commonly direct OpenGL to use buffer object data from the start of the buffer, for example. In this case, the offset into the buffer is 0. If you store vertices in the buffer object as tightly packed single-precision floats, call **`glVertexPointer`**`()` as follows:

```
glVertexPointer( 3, GL_FLOAT, 0, (GLvoid*)((char*)NULL) );
```

Here, *pointer* is a `char*` pointer to NULL, typecast as a `GLvoid*`. To determine the offset into the buffer object, OpenGL subtracts a pointer to NULL from *pointer*, and in this case, the result is an offset of 0.

As a second example, consider how you would call **`glVertexPointer`**`()` if you wanted to skip the first vertex in the buffer object. Because a single vertex is 12 bytes long (`3 * sizeof( GLfloat)`) on most 32-bit systems, this requires an offset of 12 bytes into the buffer object. Your application would call **`glVertexPointer`**`()` as follows:

```
glVertexPointer( 3, GL_FLOAT, 0, (GLvoid*)(((char*)NULL) + 12) );
```

pointer is a `char*` pointer to NULL plus 12 bytes, so when OpenGL subtracts a pointer to NULL from *pointer,* it obtains a result of 12. When rendering, OpenGL uses this as an offset into the buffer object before obtaining vertex data, effectively skipping the first vertex.

Note When you use a vertex array pointer command with buffer objects, the data stored in the buffer object must meet alignment requirements as determined by the *type* parameter. `GLfloat` data is typically aligned on 4-byte boundaries, for example. If your application calls **`glVertexPointer`**`()` with the *type* parameter set to `GL_FLOAT`, each float element stored in the corresponding buffer object must lie on a 4-byte boundary.

The following code shows a convenience routine that takes the desired offset in bytes as a parameter and returns a `GLvoid*` value to use as the *pointer* parameter to the vertex array pointer functions:

```
inline GLvoid* bufferObjectPtr( unsigned int idx )
{
    return (GLvoid*)( ((char*)NULL) + idx );
}
```

The example code uses this function when using vertex arrays with buffer objects. Using this function, your application would call **`glVertexPointer`**`()` as follows to specify a 12-byte buffer object offset:

```
glVertexPointer( 3, GL_FLOAT, 0, bufferObjectPtr( 12 ) );
```

Without buffer objects, the vertex array pointer commands incur the expense of copying vertex array data each time the vertex array rendering commands are issued. With buffer objects, OpenGL copies this data at initialization time, when the application calls **`glBufferData`**`()`. At render time, the vertex array pointer commands simply pass in an offset to the bound buffer object, eliminating render-time copies from client memory to server memory. This is illustrated in Figure 2-3.

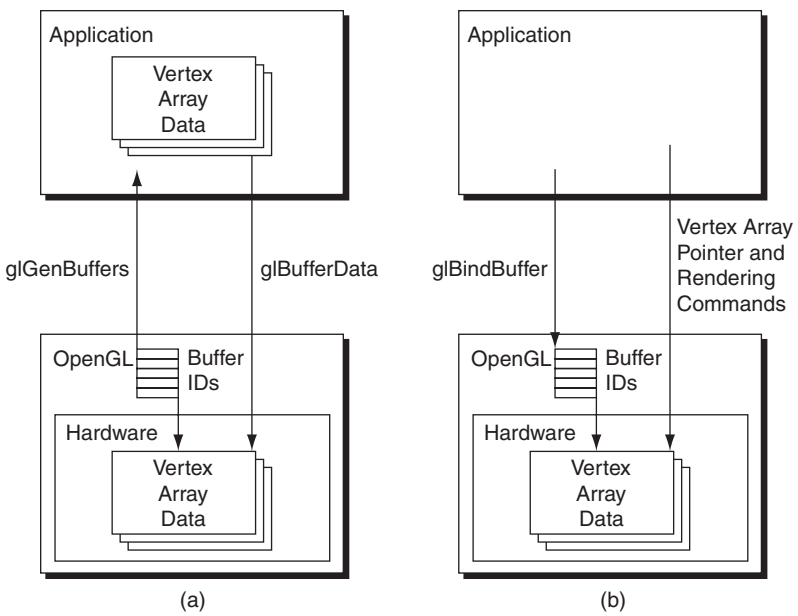

Figure 2-3 Vertex arrays with buffer objects. Initialization operations are shown in (a). The application allocates a buffer object ID with **`glGenBuffers`**`()` and stores vertex array data with **`glBufferData`**`()`. At render time (b), the application binds the buffer object and sends vertex array pointer and rendering commands.

2.2.2.3 Vertex Array Rendering Commands

So far, this chapter has covered creating and using buffer objects, and specifying vertex arrays with and without buffer objects. OpenGL doesn't draw anything, however, until your application issues vertex array rendering commands that tell OpenGL what primitives to draw from the vertex array data. This section covers those commands.

OpenGL provides several commands for drawing primitives with vertex arrays; *OpenGL® Distilled* covers three of them. (See *OpenGL® Programming Guide, OpenGL® Reference Manual,* and *The OpenGL Graphics System* for information on other vertex array commands.) These three commands are:

- **`glDrawElements`**`()`—This command renders a single OpenGL primitive with vertices and vertex data specified by vertex array indices.
- **`glDrawRangeElements`**`()`—This optimized form of **`glDrawElements`**`()` restricts selected vertex data to a specified range of index values.
- **`glMultiDrawElements`**`()`—This command renders multiple OpenGL primitives of the same type. It, too, uses indices to obtain vertex data from vertex arrays.

All three commands use indices to obtain vertices, normals, and texture coordinates from the data specified by your application in **`glVertexPointer`**`()`, **`glNormalPointer`**`()`, and **`glTexCoordPointer`**`()`. Before OpenGL version 1.5, copying these indices was a potential performance bottleneck for large blocks of static indices. Buffer objects, however, allow the application to store the indices in high-performance server memory. The following sections demonstrate this technique.

Using glDrawElements()

Applications should use **`glDrawElements`**`()` to draw a single OpenGL primitive. **`glDrawElements`**`()` was added to OpenGL in version 1.1. The example code that comes with this book falls back to using this command in versions of OpenGL that don't support more-preferred interfaces.

---

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

Draws a single primitive from vertex array data. *mode* is any valid OpenGL primitive type, as listed in the section "Primitive Types" earlier in this chapter and illustrated in Figure 2-1. *count* is the number of vertices to render.

indices is an array of indices into the enabled vertex arrays. *type* is the data type of the *indices* array and must be `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT`, or `GL_UNSIGNED_INT`.

▶ OpenGL version: 1.1 and later.

This command causes OpenGL to read *count* indices from the *indices* array and use them to index into the enabled vertex arrays to obtain vertices and vertex data. Then it uses that data to render a primitive of type *mode*. Using **`glDrawElements`**`()` vastly reduces the number of function calls made by applications using the **`glBegin`**`()`/**`glEnd`**`()` paradigm.

If a buffer object is bound to `GL_ELEMENT_ARRAY_BUFFER` when the **`glDrawElements`**`()` command is issued, OpenGL interprets *indices* as an offset into that buffer and obtains the indices from the bound buffer object. This is more common with **`glDrawRangeElements`**`()` and **`glMultiDrawElements`**`()`, because an OpenGL implementation that supports buffer objects also supports these more efficient interfaces.

Calling **`glDrawElements`**`()` with enabled normal and texture coordinate arrays leaves the current normal and current texture-coordinate states undefined.

Note **`glDrawElements`**`()` should be used only when the OpenGL implementation doesn't provide higher-performance rendering commands such as **`glDrawRangeElements`**`()` and **`glMultiDrawElements`**`()`, described next.

Using glDrawRangeElements()

The **`glDrawRangeElements`**`()` command is identical to **`glDrawElements`**`()` but requires the application to specify the minimum and maximum values of the indices. Using this command often results in better performance than **`glDrawElements`**`()` because it allows OpenGL to make better use of internal vertex array data caches.

void **glDrawRangeElements**(GLenum *mode*, GLuint *start*, GLuint *end*, GLsizei *count*, GLenum *type*, const GLvoid* *indices*);

Draws a single primitive using a range of vertex array data. *mode*, *count*, *type*, and *indices* mean the same as in **`glDrawElements`**`()`. *start* and *end* denote lower and upper limits on the *count* index values that are used in the *indices* array.

▶ OpenGL version: 1.2 and later.

When **glDrawRangeElements**() was introduced in OpenGL version 1.2, it clearly outperformed **glDrawElements**() because it allowed OpenGL to process a single block of vertex data rather than element by element. Although buffer objects somewhat level the performance of these two commands, **glDrawRangeElements**() still largely replaces **glDrawElements**() in cases where the application knows the range of its indices.

If a buffer object is bound to GL_ELEMENT_ARRAY_BUFFER when the **glDrawRangeElements**() command is issued, OpenGL interprets *indices* as an offset into that buffer and obtains the indices from the bound buffer object. The example source code makes extensive use of this feature. See the section "Vertex Arrays Example Code" later in this chapter.

Like **glDrawElements**(), vertex states that are modified by enabled vertex arrays (such as the current normal and current texture coordinate) are left undefined after a call to **glDrawRangeElements**().

Using glMultiDrawElements()

The **glMultiDrawElements**() command draws multiple primitives of the same type. It's comparable to calling **glDrawElements**() repeatedly with the same *mode* parameter.

---

void **glMultiDrawElements**(GLenum *mode*, const GLsizei* *count*, GLenum *type*, const GLvoid** *indices*, GLsizei *primcount*);

Draws multiple primitives of the same type from vertex array data. *primcount* is the number of primitives to draw. All primitives drawn are of type *mode*.

indices is a two-dimensional array of type *type*. Each subarray in *indices* contains the indices for one of the primitives. The number of indices in the subarray is specified by the corresponding element of the *count* array.

▶ OpenGL version: 1.4 and later.

This command is simple in concept, yet difficult to explain in plain English text. As an illustration, consider that **glMultiDrawElements**() behaves as though it were executing the following code:

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

If a buffer object is bound to GL_ELEMENT_ARRAY_BUFFER when the **glMultiDrawElements**() command is issued, OpenGL interprets *indices* as an offset into that buffer and obtains the indices from the bound buffer object. The example source code makes extensive use of this feature. See the next section, "Vertex Arrays Example Code," for an example.

Like **glDrawElements**() and **glDrawRangeElements**(), vertex states that are modified by enabled vertex arrays (such as the current normal and current texture coordinate) are left undefined after a call to **glMultiDrawElements**().

2.2.2.4 Vertex Arrays Example Code

The example code, available from this book's Web site, contains C++ objects for rendering cylinders, spheres, planes, and tori. The code uses different rendering paradigms and issues different OpenGL commands depending on the OpenGL version; your application needs to do the same to run on a wide variety of OpenGL implementations:

- If the OpenGL version is 1.5 or later, the example code creates and uses buffer objects. Otherwise, it passes vertex data directly when issuing the vertex pointer commands.
- All objects require either **glDrawRangeElements**() or **glMultiDrawElements**() and use this interface if the OpenGL version is 1.2 or 1.4 or later, respectively. Otherwise, the code uses the **glDrawElements**() interface.
- The example code requires at least OpenGL version 1.1; it never falls back to using the **glBegin**()/**glEnd**() paradigm. This should be sufficient because OpenGL version 1.0 implementations are rarely encountered today.

To branch efficiently in the OpenGL version, the example code uses a singleton instance that queries the OpenGL version string by using **glGetString**(GL_VERSION) once and encodes it as an enum: Ver11 for version 1.1, Ver12 for version 1.2, and so on.

Listing 2-1 shows the draw() method of the Cylinder object. This code draws the cylinder body as a quad strip and optionally caps the ends of the cylinder with triangle fans. (Note that this example doesn't use display lists. Later in this chapter, "Performance Issues" covers this topic.)

Listing 2-1 Drawing with vertex arrays and buffer objects

```
void Cylinder::draw()
{
    if (!_valid)
    {
        if (!init())
            return;
    }

    glPushClientAttrib( GL_CLIENT_VERTEX_ARRAY_BIT );

    glEnableClientState( GL_VERTEX_ARRAY );
    glEnableClientState( GL_NORMAL_ARRAY );

    if (OGLDif::instance()->getVersion() >= Ver15)
    {
        glBindBuffer( GL_ARRAY_BUFFER, _vbo[1] );
        glNormalPointer( GL_FLOAT, 0, bufferObjectPtr( 0 ) );
        glBindBuffer( GL_ARRAY_BUFFER, _vbo[0] );
        glVertexPointer( 3, GL_FLOAT, 0, bufferObjectPtr( 0 ) );

        glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, _vbo[2] );
        glDrawRangeElements( GL_QUAD_STRIP, _idxStart,
                _cap1Idx, _numVerts, GL_UNSIGNED_SHORT,
                bufferObjectPtr( 0 ) );
        if (_drawCap1)
            glDrawRangeElements( GL_TRIANGLE_FAN, _cap1Idx,
                    _cap2Idx, _numCapVerts, GL_UNSIGNED_SHORT,
                    bufferObjectPtr((_cap1Start-_indices)
                    * sizeof(GLushort)) );
        if (_drawCap2)
            glDrawRangeElements( GL_TRIANGLE_FAN, _cap2Idx,
                    _idxEnd, _numCapVerts, GL_UNSIGNED_SHORT,
                    bufferObjectPtr((_cap2Start-_indices)
                    * sizeof(GLushort)) );
    }
    else
    {
        glVertexPointer( 3, GL_FLOAT, 0, _vertices );
        glNormalPointer( GL_FLOAT, 0, _normals );

        if (OGLDif::instance()->getVersion() >= Ver12)
        {
            glDrawRangeElements( GL_QUAD_STRIP, _idxStart,
                    _cap1Idx, _numVerts, GL_UNSIGNED_SHORT,
                    _indices );
            if (_drawCap1)
                glDrawRangeElements( GL_TRIANGLE_FAN, _cap1Idx,
                        _cap2Idx, _numCapVerts, GL_UNSIGNED_SHORT,
                        _cap1Start );
```

```
            if (_drawCap2)
                glDrawRangeElements( GL_TRIANGLE_FAN, _cap2Idx,
                        _idxEnd, _numCapVerts, GL_UNSIGNED_SHORT,
                        _cap2Start );
        }
        else
        {
            glDrawElements( GL_QUAD_STRIP, _numVerts,
                    GL_UNSIGNED_SHORT, _indices );
            if (_drawCap1)
                glDrawElements( GL_TRIANGLE_FAN, _numCapVerts,
                        GL_UNSIGNED_SHORT, _cap1Start );
            if (_drawCap2)
                glDrawElements( GL_TRIANGLE_FAN, _numCapVerts,
                        GL_UNSIGNED_SHORT, _cap2Start );
        }
    }

    glPopClientAttrib();

    OGLDIF_CHECK_ERROR;
}
```

The code first tests to see whether the `Cylinder` object has already been initialized, and if it hasn't, it calls the `init()` method, which generates the cylinder vertex and normal data. If the OpenGL version is 1.5 or later, it also allocates buffer objects and fills them with that data. Download the example source code to see the `Cylinder::init()` method.

The `draw()` method pushes the client attribute stack with a call to **`glPushClientAttrib`**`()`. When it calls **`glPopClientAttrib`**`()` at the end of the function, the client states changed by this function—such as the state of enabled arrays, bound buffer objects, and vertex array pointers—restore to their previous values.

The `Cylinder` object doesn't specify texture coordinates, so it needs only to enable the normal and vertex arrays. These arrays must be enabled regardless of the OpenGL version.

Next, the `draw()` method specifies the vertex data and issues the rendering commands. How `draw()` does this depends on the OpenGL version.

For OpenGL version 1.5, the `Cylinder` object uses three buffer objects: one for normal data, one for vertex data, and one for the indices into the arrays. Note that the normal and vertex buffer objects are bound to `GL_ARRAY_BUFFER` before calling **`glNormalPointer`**`()` and **`glVertexPointer`**`()`, respectively, and that the indices buffer object is bound to `GL_ELEMENT_ARRAY_BUFFER` before issuing the drawing commands.

The code draws three primitives—a quad strip and, optionally, two triangle fans—using three **`glDrawRangeElements`**`()` commands. The code doesn't use different buffer objects for each primitive. Instead, it passes in different offsets to each **`glDrawRangeElements`**`()` call (along with different minimum and maximum index range values) to access the specific data for each primitive.

If the OpenGL version is less than 1.5, the code passes the vertex and normal data directly in its calls to **`glNormalPointer`**`()` and **`glVertexPointer`**`()`.

The code uses **`glDrawRangeElements`**`()` if the version is 1.2 or greater, but when less than version 1.5, the code must pass the indices directly to these commands rather than pass offsets.

If the version is less than 1.2, the code assumes version 1.1 and uses **`glDrawElements`**`()`. This is the least-desired code path, because OpenGL doesn't know what data to process until after it copies and starts to process each index.

Before returning to the calling application, `Cylinder::draw()` pops the client attribute stack and tests for OpenGL errors. `OGLDIF_CHECK_ERROR` is a CPP macro, described in Appendix D, "Troubleshooting and Debugging."

2.3 Drawing Details

When an application submits a primitive to OpenGL for rendering, OpenGL uses the current state to determine what operations to perform on the primitive data.

OpenGL supports a variety of flexible features that affect the appearance of rendered geometry. Some of the more complex operations, such as viewing, lighting, and texture mapping, are covered in their own chapters. This section explains a few of the simpler operations. *OpenGL® Programming Guide* and *OpenGL® Shading Language* present the complete OpenGL feature set.

2.3.1 Clearing the Framebuffer

Before issuing the first rendering command and periodically thereafter (typically, at the start of each frame), applications need to clear the contents of the framebuffer. OpenGL provides the **`glClear`**`()` command to perform this operation.

void **glClear**(GLbitfield *mask*);

Clears the framebuffer contents. *mask* is one or more bit values combined with the bitwise OR operator that specify the portion of the framebuffer to clear. If `GL_COLOR_BUFFER_BIT` is present in *mask*, **`glClear`**`()` clears the color buffer. If `GL_DEPTH_BUFFER_BIT` is present in *mask*, **`glClear`**`()` clears the depth buffer. If both bit values are present, **`glClear`**`()` clears both the color and depth buffers.

`glClear``()` can also clear other parts of the framebuffer, such as the stencil and accumulation buffers. For a complete list of bit values accepted by **`glClear`**`()`, see "glClear" in *OpenGL® Reference Manual*.

▶ OpenGL version: 1.0 and later.

Typically, applications clear both the color and depth buffers with a single **`glClear`**`()` call at the start of each frame.

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

Clearing multiple buffers with a single **`glClear`**`()` call is more efficient than clearing separate buffers with separate **`glClear`**`()` calls. There are rendering techniques that require clearing the depth and color buffer separately, however. Try to clear both buffers at the same time when possible.

Your application controls the color that **`glClear`**`()` writes to the color buffer by calling **`glClearColor`**`()`.

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

Specifies the clear color used by **`glClear`**`()`. *red*, *green*, *blue*, and *alpha* are four values specifying an RGBA color value and should be in the range 0.0 to 1.0. Subsequent calls to **`glClear`**`()` use this color value when clearing the color buffer.

▶ OpenGL version: 1.0 and later.

`glClearColor``()` sets the current clear color, which is black by default (0.0, 0.0, 0.0, 0.0). This is adequate for some applications. Applications that set the clear color usually do so at program init time.

Note that not all framebuffers contain an alpha channel. If the framebuffer doesn't have an alpha channel, OpenGL effectively ignores the *alpha* parameter when clearing. Specify whether your framebuffer contains an alpha channel with the GLUT command **`glutInitDisplayMode`**`()` or platform-specific framebuffer configuration calls.

You can also specify the depth value written into the depth buffer by **`glClear`**`()`. By default, **`glClear`**`( GL_DEPTH)BUFFER_BIT )` clears the depth buffer to the maximum depth value, which is adequate for many applications. To change this default, see "glDepthValue" in *OpenGL® Reference Manual*.

2.3.2 Modeling Transformations

Complex 3D scenes typically are composed of several objects or models displayed at specific locations within the scene. These objects are routinely modeled in their own modeling-coordinate system and transformed by the application to specific locations and orientations in the world-coordinate system. Consider a scene composed of an aircraft flying over a terrain model. Modelers create the aircraft by using a modeling software package and might use a coordinate system with the origin located in the center of the fuselage, with the aircraft nose oriented toward positive *y* and the top of the aircraft oriented toward positive *z*. To position and orient this aircraft model relative to the terrain model, the application must translate it laterally to the correct position and vertically to the correct altitude, and orient it to the desired pitch, roll, and aircraft heading.

OpenGL transforms all vertices through a geometry pipeline. The first stage of this pipeline is the modeling transformation stage. Your application specifies modeling transformations to position and orient models in the scene. To manage the transformation of your geometry effectively, you need to understand the full OpenGL transformation pipeline. See Chapter 3, "Transformation and Viewing," for details.

2.3.3 Smooth and Flat Shading

To simulate a smooth surface, OpenGL interpolates vertex colors during rasterization. To simulate a flat or faceted surface, change the default shade model from `GL_SMOOTH` to `GL_FLAT` by calling **`glShadeModel`**`( GL_FLAT )`.

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

Specifies smooth or flat shading. *mode* is either GL_SMOOTH or GL_FLAT. The default value of GL_SMOOTH causes OpenGL to use Gouraud shading to interpolate vertex color values during rasterization. GL_FLAT causes OpenGL to color subsequent primitives with the color of the vertex that completes the primitive.

▶ OpenGL version: 1.0 and later.

To determine the color of a primitive in flat shade mode, OpenGL uses the color of the vertex that completes the primitive. For GL_POINTS, this is simply the color of the vertex. For all line primitives (GL_LINES, GL_LINE_STRIP, and GL_LINE_LOOP), this is the color of the second vertex in a line segment.

For GL_TRIANGLES, OpenGL colors each triangle with the color of every third vertex. For both GL_TRIANGLE_STRIP and GL_TRIANGLE_FAN, OpenGL colors the first triangle with the color of the third vertex and colors each subsequent triangle with the color of each subsequent vertex.

For GL_QUADS, OpenGL colors each quadrilateral with the color of every fourth vertex. For GL_QUAD_STRIP, OpenGL uses the color of the fourth vertex to color the first quadrilateral in the strip and every other vertex thereafter to color subsequent quadrilaterals.

For GL_POLYGON, OpenGL colors the entire polygon with the color of the final vertex.

2.3.4 Polygon Mode

It shouldn't be a surprise that filled primitives are drawn filled by default. Applications can specify, however, that OpenGL render filled primitives as lines or points with the **glPolygonMode**() command.

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

Specifies the rendering style for filled primitives. *mode* is GL_POINT, GL_LINE, or GL_FILL; and *face* must be GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK to specify whether *mode* applies to front- or back-facing primitives, or both.

▶ OpenGL version: 1.0 and later.

Polygon mode is useful for design applications that allow the user to switch between solid and wireframe rendering. Many applications also use it to highlight selected primitives or groups of primitives.

2.3.5 The Depth Test

OpenGL supports the depth test (or *z*-buffer) hidden-surface-removal algorithm (Catmull 1974).

To use the depth test, your application must allocate a depth buffer when creating its display window. In GLUT, use a bitwise OR to include the `GLUT_DEPTH` bit in the parameter to **`glutInitDisplayMode`**`()` before calling **`glutCreateWindow`**`()`. Applications typically specify a double-buffered RGB window with a depth buffer by using the following call:

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

The depth test is disabled by default. Enable it with **`glEnable`**`( GL_DEPTH_TEST )`.

The depth-test feature discards a fragment if it fails to pass the depth comparison test. Typically, applications use the default comparison test, `GL_LESS`. In this case, a fragment passes the depth test if its window-space *z* value is less than the stored depth buffer value. Applications can change the default comparison test by calling **`glDepthFunc`**`()`. The `GL_LEQUAL` comparison test, which passes a fragment if its *z* value is less than or equal to the stored depth value, is useful in multipass algorithms. See "glDepthFunc" in *OpenGL® Reference Manual* for a complete description of this function.

OpenGL executes commands in the order in which they are sent by the application. This rule extends to rendering primitives; OpenGL processes vertex array rendering commands in the order in which they are sent by the application and renders primitives within a single vertex array in the order that the array indices specify. This feature allows applications to use the *painter's algorithm* to remove hidden surfaces, rendering a scene in order of decreasing distance. When applications disable the depth test, the first primitives rendered are overwritten by those that are rendered later.

2.3.6 Co-Planar Primitives

You might expect that fragments with the same window-space *x* and *y* values from co-planar primitives would also have identical window-space *z*

values. OpenGL guarantees this only under certain circumstances.[4] When co-planar filled primitives have different vertices, floating-point roundoff error usually results in different window-space *z* values for overlapping pixels. Furthermore, because line primitives don't have plane equations, it's impossible for unextended OpenGL to generate identical window-space *z* values for co-planar lines and filled primitives. For this reason, setting **glDepthFunc**(GL_LEQUAL) is insufficient to cause co-planar primitives to pass the depth test.

Applications can apply a depth offset to polygonal primitives to resolve co-planarity issues.

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

Specifies parameters for altering fragment depth values. OpenGL scales the maximum window-space *z* slope of the current polygonal primitive by *factor,* scales the minimum resolvable depth buffer unit by *units,* and sums the results to obtain a depth offset value. When enabled, OpenGL adds this value to the window-space *z* value of each fragment before performing the depth test.

The depth offset feature is also referred to as *polygon offset* because it applies only to polygonal primitives.

▶ OpenGL version: 1.1 and later.

Depth offset applies to polygonal primitives but can be separately enabled and disabled for each of the three polygon modes. To enable depth offset in fill mode, call **glEnable**(GL_POLYGON_OFFSET_FILL). Use GL_POLYGON_OFFSET_POINT and GL_POLYGON_OFFSET_LINE to enable or disable depth offset for point and line mode, respectively.

Typically, applications call **glPolygonOffset**(1.f, 1.f) and **glEnable**(GL_POLYGON_OFFSET_FILL) to render filled primitives that are pushed back slightly into the depth buffer, then disable depth offset and draw co-planar geometry.

Figure 2-4 illustrates rendering co-planar primitives with and without depth offset.

4. See Appendix A, "Invariance," in *The OpenGL Graphics System*.

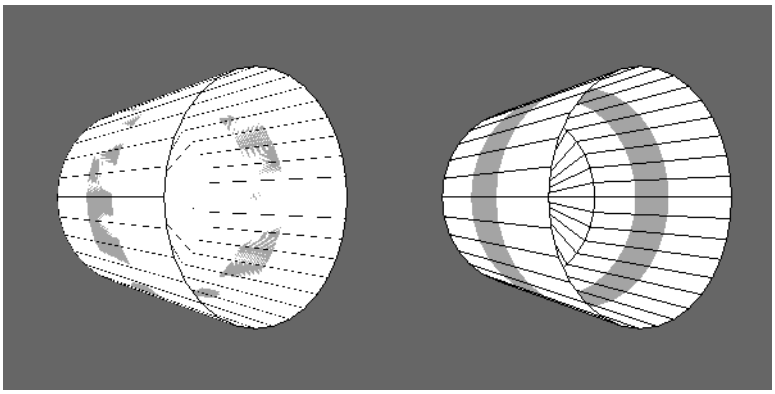

Figure 2-4 This figure illustrates the result of rendering co-planar primitives. When depth offset is disabled (left), differences in rasterization of co-planar primitives results in rendering artifacts known as "*z*-fighting." Depth offset (right) eliminates these artifacts.

The following code was used to render the cylinders in Figure 2-4:

```
if (enableOffset)
    glEnable( GL_POLYGON_OFFSET_FILL );
else
    glDisable( GL_POLYGON_OFFSET_FILL );

// Draw the large white cylinder
glPolygonOffset( 2.f, 2.f );
glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );
glColor3f( 1.f, 1.f, 1.f );
cylBody.draw();

// Draw the cylinder's center stripe
glPolygonOffset( 1.f, 1.f );
glColor3f( .6f, .6f, 1.f );
cylStripe.draw();

// Draw the cylinder in line mode
glPolygonMode( GL_FRONT_AND_BACK, GL_LINE );
glColor3f( 0.f, 0.f, 0.f );
cylBody.draw();
```

`cylBody` and `cylStripe` are instantiations of the `Cylinder` object defined in the example code. `Cylinder::draw()` appears in Listing 2-1. The full DepthOffset example source code is available from the book's Web site.

The code either enables or disables depth offset for filled primitives based on a Boolean variable. It draws the white cylinder with **`glPolygonOffset`** `( 2.f, 2.f )` to push it back in the depth buffer. The cylinder's bluish stripe is drawn with **`glPolygonOffset`**`( 1.f, 1.f )`, which also pushes

it back, but not as far as the first cylinder, to prevent *z*-fighting. Finally, the first cylinder draws a second time in line mode. Because depth offset isn't enabled for line mode, it's drawn with no offset.

When drawing 3D scenes with co-planar primitives, applications generally set *factor* and *units* to the same value. This ensures that depth offset resolves the co-planarity regardless of the primitive orientation.

2.3.7 Alpha and Transparency

Internally, OpenGL colors have four components: red, green, blue, and alpha. You can specify each of these components with **glColor4f**(*r*, *g*, *b*, *a*), but even if you specify a color with three components, such as **glColor3f**(*r*, *g*, *b*), OpenGL uses a full-intensity alpha internally.

Applications often store an opacity value in the alpha component, and use the OpenGL blending and alpha test features to render translucent and transparent primitives.

2.3.7.1 Blending

The blending feature combines the incoming (or source) fragment color with the stored (or destination) color in the framebuffer. Commonly used to simulate translucent and transparent surfaces in 3D rendering, blending also has applications in image processing.

This chapter covers simple OpenGL blending. Blending supports many variants that are not covered in this book. For full details on blending, see Section 4.1.8, "Blending," of *The OpenGL Graphics System* and Chapter 6, "Blending, Antialiasing, Fog, and Polygon Offset," of *OpenGL® Programming Guide*.

To use blending, your application must enable it and set an appropriate blending function. Enable and disable blending with GL_BLEND. Call **glEnable**(GL_BLEND) to enable blending, for example. Set the blend function with the **glBlendFunc**() command.

void **glBlendFunc**(GLenum *src*, GLenum *dst*);

Sets blending function parameters that control the combination of stored framebuffer colors with fragment colors. *src* and *dst* specify functions that operate on the source and destination colors, respectively. When blending is enabled, OpenGL replaces the fragment color with the sum of the output of these functions.

▶ OpenGL version: 1.0 and later.

When simulating translucent surfaces, applications typically call **glBlendFunc**(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA). This multiplies the source fragment color by the source alpha value (typically small to simulate a nearly transparent surface) and multiplies the destination color by 1 minus the source alpha value.

For a complete list of valid *src* and *dst* values, see Table 4.2 in *The OpenGL Graphics System*.

To render translucent surfaces correctly, applications should first render all opaque geometry into the framebuffer. Applications should further sort translucent geometry by distance from the viewer and render it in back-to-front order.

Note Sorting translucent geometry could negatively affect application performance. Order-independent transparency, currently an area of active research, is outside the scope of this book.

2.3.7.2 Alpha Test

The alpha-test feature discards fragments that have alpha values that fail to pass an application-specified comparison test. Applications use the alpha test to discard partially or completely transparent fragments that, if rendered, would have little or no effect on the color buffer.

To use the alpha test, your application must enable it and specify the comparison test. Enable and disable the alpha test with GL_ALPHA_TEST. Enable it by calling **glEnable**(GL_ALPHA_TEST), for example. To specify the comparison test, use **glAlphaFunc**().

void **glAlphaFunc**(GLenum *func*, GLclampf *ref*);

Specifies how the alpha test discards fragments. *func* specifies a comparison function and may be GL_ALWAYS, GL_EQUAL, GL_GEQUAL, GL_GREATER, GL_LEQUAL, GL_LESS, GL_NEVER, or GL_NOTEQUAL. *ref* specifies a constant floating-point reference value used in the comparison.

▶ OpenGL version: 1.0 and later.

Consider setting the following alpha-test state:

```
glEnable( GL_ALPHA_TEST );
glAlphaFunc( GL_NOTEQUAL, 0.f );
```

In this case, OpenGL renders fragments from subsequent geometry only if the fragment alpha value isn't equal to zero.

You might also want to discard fragments with an alpha that doesn't exceed a certain threshold. Setting the alpha function to **`glAlphaFunc`** `( GL_GREATER, 0.1f )` passes fragments only if their alpha value is greater than 0.1.

2.4 Performance Issues

Buffer objects and vertex arrays should result in acceptable performance in most rendering situations.

Visit OpenGL vendor Web sites to become informed about the limits of specific OpenGL implementations. If you suspect that your performance is significantly less than typical, add debugging code to obtain an accurate vertex count from your application. Compare your vertex count with vendor specifications, and calculate an expected performance rate. Ensure that you are not specifying more vertices than necessary.

The following sections provide some additional performance tips.

2.4.1 Display Lists

Display lists store a sequence of OpenGL commands in server memory for later execution. Applications execute the entire display list command sequence with a single command, **`glCallList`**`()`. Executing commands from display lists is often more efficient than issuing the same sequence of commands individually.

To use display lists, your application must obtain, store, and eventually dispose of display list identifiers. For each display list your application uses, you must store a sequence of OpenGL commands in it. Finally, when you want to execute the stored commands, you'll need to issue the **`glCallList`**`()` command.

To obtain unused display list identifiers, use **`glGenLists`**`()`. To dispose of display list identifiers when they are no longer needed, call **`glDeleteLists`**`()`.

GLuint **glGenLists**(GLsizei *s*);
void **glDeleteLists**(GLuint *list*, GLsizei *range*);
GLboolean **glIsList**(GLuint *list*);

Use these commands to manage display list identifiers. **`glGenLists`**`()` creates a sequence of *s* unused display list identifiers and marks them as used. It returns the first identifier in the sequence. **`glDeleteLists`**() deletes the sequence of *range* identifiers starting at *list*. **`glIsList`**`()` returns `GL_TRUE` if *list* is an existing display list.

▶ OpenGL version: 1.0 and later.

To obtain a single display list identifier, use the following code:

```
GLuint listID;
listID = glGenLists( 1 );
```

Your application can obtain multiple display list identifiers by passing in an *s* parameter greater than 1. In this case, the returned identifier is the base, and additional identifiers follow sequentially.

When you call **`glDeleteLists`**`()`, OpenGL marks the sequence of identifiers from *list* through *list*+*range*-1 as unused.

To store OpenGL commands in a display list, issue the **`glNewList`**`()` command followed by the sequence of commands you want to store. Stop storing commands in a display list with the **`glEndList`**`()` command.

void **glNewList**(GLuint *n*, GLenum *mode*);
void **glEndList**(void);

Use these commands to specify OpenGL commands to store in a display list. *n* is the display list identifier. When *mode* is `GL_COMPILE` , subsequent OpenGL commands are stored in the display list. Specifying a *mode* of `GL_COMPILE_AND_EXECUTE` also stores commands but additionally executes them. `GL_COMPILE_AND_EXECUTE` is less efficient than `GL_COMPILE`, so avoid `GL_COMPILE_AND_EXECUTE`.

▶ OpenGL version: 1.0 and later.

As an example, consider the **`glBegin`**`()`/**`glEnd`**`()` paradigm code for drawing a triangle. Because this method of specifying geometry uses so many

function calls, it's an excellent candidate for storing in a display list. The following code stores in a display list the commands for drawing a triangle:

```
glNewList( listID, GL_COMPILE );

glBegin( GL_TRIANGLES );
  glColor3f( 1.f, 0.f, 0.f );
  glVertex3f( 0.f, 0.f, 0.f );
  glVertex3f( 1.f, 0.f, 0.f );
  glVertex3f( 0.f, 1.f, 0.f );
glEnd();

glEndList();
```

OpenGL doesn't store all commands in a display list. In particular, commands that affect client state aren't stored in display lists, such as **glPushClientAttrib**(), **glPopClientAttrib**(), **glEnableClientState**(), **glBindBuffer**(), **glVertexPointer**(), **glNormalPointer**(), and **glTexCoordPointer**(). If these commands are executed between **glNewList**() and **glEndList**(), OpenGL executes them immediately. For a complete list of commands that are not stored in a display list, see Section 5.4, "Display Lists," of *The OpenGL Graphics System*.

Applications can store the vertex array rendering commands **glDrawElements**(), **glDrawRangeElements**(), and **glMultiDrawElements**() in display lists. When an application stores a vertex array rendering command in a display list, OpenGL copies the necessary vertex array data into the display list at the same time.

You shouldn't store vertex array rendering commands in display lists if you're using buffer objects, because buffer objects usually store vertex data in high-performance server memory. In this situation, storing the vertex array rendering commands in a display list would cause OpenGL to make a second copy of the data in high-performance server memory. This could adversely affect application performance by unnecessarily consuming additional memory. On the other hand, if your application is running on an OpenGL implementation that doesn't support buffer objects, storing vertex array rendering commands in display lists could boost application performance.

As stated previously, storing a vertex array rendering command in a display list causes OpenGL to copy all necessary vertex array data into the list at the same time. For this reason, your application must enable the appropriate arrays and issue vertex array pointer commands as though it is actually rendering the arrays.

To execute the commands stored in a display list, call **glCallList**().

void **glCallList**(GLuint *n*);

Executes stored OpenGL commands from a display list. *n* is the display list identifier to execute.

▶ OpenGL version: 1.0 and later.

An added benefit of display lists is the elimination of application branches and loops. A `for` loop, for example, at the machine level typically incurs the overhead of an increment, a compare, and a branch per loop iteration. Highly tuned applications often partially unwind loops to minimize this overhead. When creating a display list, the application executes the loop only while creating the list. OpenGL stores only the resulting OpenGL commands, which are executed with a **glCallList**() command as a fully unwound loop.

The drawback to display lists should be obvious. When creating a display list, OpenGL makes a copy of both the commands and their data, possibly creating a burden on memory. If memory is a scarce resource for your application, you should use display lists only for the most performance-critical portions of your code.

2.4.2 Face Culling

The OpenGL face-culling feature discards filled primitives based on the direction they face. Applications typically use face culling to discard faces from the back of models. This boosts performance by not bothering to rasterize primitives that won't be visible in the final image.

To use face culling, your application needs to enable it. You might also need to specify the faces that OpenGL should cull.

OpenGL effectively uses the right-hand rule to determine whether a primitive is front- or back-facing. If the vertices are ordered counterclockwise, the primitive is front facing. Note that a front face can become a back face due to a modeling transformation or change in viewing angle. For this reason, OpenGL performs face culling in window-coordinate space. (Chapter 3, "Transformation and Viewing," describes all OpenGL coordinate spaces. For now, think of window coordinates as pixels onscreen.)

To enable face culling, call **glEnable**(GL_CULL_FACE). By default, this culls back faces, or faces with clockwise-ordered window-coordinate vertices.

You can tell OpenGL to cull front faces instead by calling **glCullFace** (GL_FRONT).

Typically, applications enable face culling when the application renders solid geometry with a complete hull. Many applications create geometry using the right-hand rule so that counterclockwise vertices define front faces. Not all applications behave this way, however, and it's common for 3D model files to store faces with a vertex ordering that doesn't match the OpenGL default. You can change this default with the **glFrontFace**() command. See "glFrontFace" in *OpenGL® Reference Manual*.

For more information on face culling, see Chapter 2, "State Management and Drawing Geometric Objects," of *OpenGL® Programming Guide*.

2.4.3 Vertex Array Size

Most OpenGL implementations can provide maximum performance only if the number of vertices stored in a vertex array is below an implementation-specific threshold. An application can store as many vertices in a vertex array as necessary, but if the number exceeds this threshold, performance could suffer.

Applications can query this implementation-specific threshold with the following code:

```
GLint maxVerts;
glGetIntegerv( GL_MAX_ELEMENTS_VERTICES, &maxVerts );
```

When using **glDrawRangeElements**(), the minimum and maximum index parameters, *start* and *end,* should specify a range smaller than GL_MAX_ELEMENTS_VERTICES to obtain maximum performance.

OpenGL implementations have a similar limit on the number of vertex array indexes. Applications can query this value by calling **glGetIntegerv**() and passing in GL_MAX_ELEMENTS_INDICES. The *count* parameter to both **glDrawElements**() and **glDrawRangeElements**() should be less than GL_MAX_ELEMENTS_INDICES to obtain maximum performance.

Many applications disregard these upper limits and simply specify as many vertices and indices as necessary. If performance is a concern, however, modify your application to respect these limits where feasible.

2.5 More Information

Chapter 2, "State Management and Drawing Geometric Objects," of *OpenGL® Programming Guide* is a much more comprehensive discussion of drawing and state control. It includes discussion of topics outside the scope of *OpenGL® Distilled*, such as line and point state, edge flags, interleaved vertex arrays, and mapping and unmapping buffer objects.

Appendix B, "State Variables," of *OpenGL® Programming Guide* and Chapter 6, "State and State Requests," of *The OpenGL Graphics System* contain thorough reference material on OpenGL state and state queries.

For detailed information regarding any of the vertex array commands discussed in this chapter (**`glVertexPointer`**`()`, **`glDrawRangeElements`**`()`, etc), see *OpenGL® Reference Manual*. (At this time this book went to press, the fifth edition of that book, which discusses buffer objects, was not yet available.)

The book's Web site contains example source code that demonstrates correct usage of the vertex array and buffer object features, as well as the use of several rendering features, such as polygon mode, depth test, and alpha test. Additional example code is available from the OpenGL Web site at `http://www.opengl.org`.

2.6 References

(Catmull 1974) Catmull, Edwin. "A Subdivision Algorithm for Computer Display of Curved Surfaces." Ph.D. thesis, University of Utah, 1974.