The OpenGL Graphics Interface

Mark Segal
Kurt Akeley
Silicon Graphics Computer Systems
2011 N. Shoreline Blvd., Mountain View, CA 94039
USA

Abstract

Graphics standards are receiving increased attention in the computer graphics community as more people write programs that use 3D graphics and as those already possessing 3D graphical programs want those programs to run on a variety of computers.

OpenGL is an emerging graphics standard that provides advanced rendering features while maintaining a simple programming model. Its procedural interface allows a graphics programmer to describe rendering tasks, whether simple or complex, easily and efficiently. Because OpenGL is rendering-only, it can be incorporated into any window system (and has been, into the X Window System and the soon-to-be-released Windows NT) or can be used without a window system. Finally, OpenGL is designed so that it can be implemented to take advantage of a wide range of graphics hardware capabilities, from a basic framebuffer to the most sophisticated graphics subsystems.

1 Introduction

Computer graphics (especially 3D graphics, and interactive 3D graphics in particular) is finding its way into an increasing number of applications, from simple graphing programs for personal computers to sophisticated modeling and visualization software on workstations and supercomputers. As the interest in computer graphics has grown, so has the desire to be able to write an application so that it runs on a variety platforms with a range of graphical capabilites. A graphics standard eases this task by eliminating the need to write a distinct graphics driver for each platform on which the application is to run.

Several standards have succeeded in integrating specific domains of 2D graphics. The PostScript page description language[4] has become widely accepted, making it relatively easy to electronically exchange, and, to a limited degree, manipulate static documents containing both text and 2D graphics. The X window system[7] has become standard for UNIX workstations. A programmer uses X to obtain a window on a graphics display into which either text or 2D graphics may be drawn; X also provides a standard means for obtaining user input from such devices as keyboards and mice. The adoption of X by most workstation manufacturers means that a single program can produce 2D graphics or obtain user input on a variety of workstations by simply recompiling the program. This integration even works across a network: the program may run on one workstation but display on and obtain user input from another, even if the workstations on either end of the network are made by different companies.

For 3D graphics, several standards have been proposed, but none has (yet) gained wide acceptance. One relatively well-known system is PHIGS (Programmer's Hierarchical Interactive Graphics System). Based on GKS[5] (Graphics Kernel System), PHIGS is an ANSI (American National Standards Institute) standard. PHIGS (and its descendant, PHIGS+[9]) provides a means to manipulate and draw 3D objects by encapsulating object descriptions and attributes into a *display list* that is then referenced when the object is displayed or manipulated. One advantage of the display list is that a complex object need be described only once even if it is to be displayed many times. This is especially important if the object to be displayed must be transmitted across a low-bandwidth channel (such as a network). One disadvantage of a display list is that it can require considerable effort to re-specify the object if it is being continually modified as a result of user interaction. Another difficulty with PHIGS and PHIGS+ (and with GKS) is that they lack support for advanced rendering features such as texture mapping.

PEX[8], which is often said to be an acronym for PHIGS Extension to X, extends X to include the ability to manipulate and draw 3D objects. (PEXlib[6] is the programmer's interface to the PEX protocol.), Among other extensions, PEX adds *immediate mode* rendering to PHIGS, meaning that objects can be displayed as they are described rather than having to first complete a display list. One difficulty with PEX has been that different suppliers of the PEX interface have chosen to support different features, making program portability problematic. PEX also lacks advanced rendering feaures, and is available only to users of X.

2 OpenGL

OpenGL ("GL" for "Graphics Library") provides advanced rendering features in either immediate mode or display list mode. While OpenGL is a relatively new standard, it is very similar in both its functionality and its interface to Silicon Graphics' IRIS GL, and there are many succesful 3D applications that currently use IRIS GL for their 3D rendering.

Like the graphics systems already discussed, OpenGL is a software interface to graphics hardware. The interface consists of a set of several hundred procedures and functions that allow a programmer to specify the objects and operations involved in producing high-quality graphical images, specifically color images of three-dimensional objects. Like PEX, OpenGL integrates 3D drawing into X, but can also be integrated into other window systems (e.g. Windows/NT) or can be used without a window system.

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

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

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

The model for interpretation of OpenGL commands is client-server.

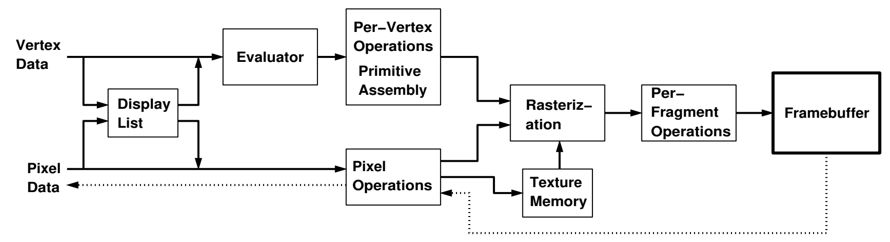

Figure 1. Block diagram of OpenGL.

That is, a program (the client) issues commands, and these commands are interpreted and processed by OpenGL (the server). The server may or may not operate on the same computer as the client.

The effects of OpenGL commands on the framebuffer are ultimately controlled by the window system that allocates framebuffer resources. It is the window system that determines which portions of the framebuffer that OpenGL may access at any given time and that communicates to OpenGL how those portions are structured. Similarly, display of framebuffer contents on a CRT monitor (including the transformation of individual framebuffer values by such techniques as gamma correction) is not addressed by OpenGL. Framebuffer configuration occurs outside of OpenGL in conjunction with the window system; the initialization of an OpenGL context occurs when the window system allocates a window for OpenGL rendering. Additionally, OpenGL has no facilities for obtaining user input, since it is expected that any window system under which OpenGL runs must already provide such facilities. These considerations make OpenGL independent of any particular window system.

3 Basic OpenGL Operation

Figure 1 shows a schematic diagram of OpenGL. Commands enter OpenGL on the left. Most commands may be accumulated in a *display list* for processing at a later time. Otherwise, commands are effectively sent through a processing pipeline.

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

Finally, pixel rectangles and bitmaps bypass the vertex processing portion of the pipeline to send a block of fragments directly through rasterization to the individual fragment operations, eventually causing a block of pixels to be written to the framebuffer. Values may also be read back from the framebuffer or copied from one portion of the framebuffer to another. These transfers may include some type of decoding or encoding.

3.1 The OpenGL Utility Library

A guiding principle in the design of OpenGL has been to provide program portability without mandating how higher-level graphical objects must be described. As a result, the basic OpenGL interface does not support some geometric objects that are traditionally associated with graphics standards. For instance, an OpenGL implementation need not render concave polygons. One reason for this omission is that concave polygon rendering algorithms are of necessity more complex than those for rendering convex polygons, and different concave polygon algorithms may be appropriate in different domains. In particular, if a concave polygon is to be drawn more than once, it is more efficient to first decompose it into convex polygons (or triangles) once and then draw the convex polygons.

A general concave polygon decomposer is provided as part of the OpenGL Utility Library, which is provided with every OpenGL implementation. The Utility Library also provides an interface, built on OpenGL's polynomial evaluators, to describe and display NURBS curves and surfaces (with domain space trimming), as well as a means for rendering spheres, cones, and cylinders. The Utility Library serves both as a means to render useful geometric objects and as a model for building other libraries that use OpenGL

Object	Interpretation of Vertices
point	each vertex describes the location of a point
line strip	series of connected line segments; each vertex after first describes the endpoint of next segment
line loop	same as line strip but final segment added from final vertex to first vertex
separate line	each pair of vertex describes a line segment
polygon	line loop formed by vertices describes the boundary of a convex polygon
triangle strip	each vertex after the first two describes a triangle given by that vertex and the previous two
triangle fan	each vertex after the first two describes a triangle given by that vertex, the previous vertex, and the first vertex
separate triangle	each consecutive triad of vertices describes a triangle
quadrilateral strip	each pair of vertices after the first two describes a quadrilateral given by that pair and the previous pair
independent quad	each consecutive group of four vertices describes a quadrilateral

Table 1: **glBegin/glEnd** objects.

for rendering.

4 The OpenGL Pipeline

4.1 Vertices and Primitives

In OpenGL, most geometric objects are drawn by enclosing a series of coordinate sets that specify vertices and optionally normals, texture coordinates, and colors between **glBegin/glEnd** command pairs. For example, to specify a triangle with vertices at $(0,0,0)$, $(0,1,0)$, and $(1,0,1)$, one could write:

```
glBegin(GL_POLYGON);
  glVertex3i(0,0,0);
  glVertex3i(0,1,0);
  glVertex3i(1,0,1);
glEnd();
```

The ten geometric objects that are drawn this way are summarized in Table 4.1. This particular group of objects was selected because each object's geometry is specified by a simple list of vertices, because each admits an efficient rendering algorithm, and because it was determined that taken together these objects satisfy the needs of nearly all graphics applications.

Each vertex may be specified with two, three, or four coordinates (four coordinates indicate a homogeneous three-dimensional location). In addition, a *current normal*, *current texture coordinates*, and *current color* may be used in processing each vertex. OpenGL uses normals in lighting calculations; the current normal is a three-dimensional vector that may be set by

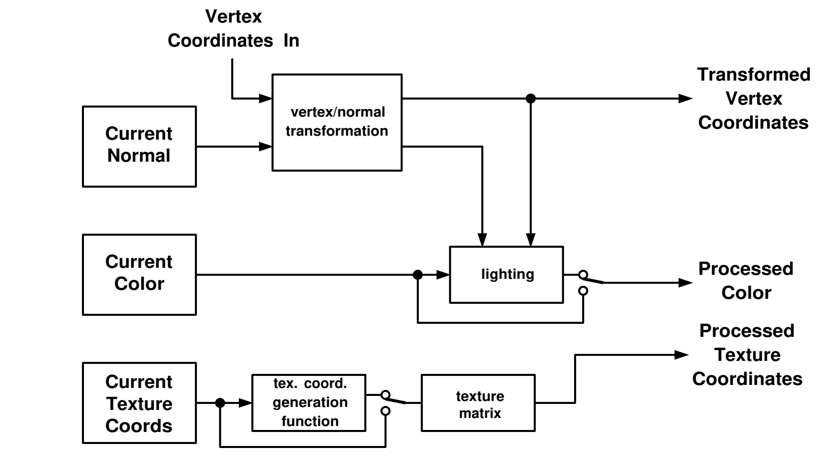

Figure 2. Association of current values with a vertex.

sending three coordinates that specify it. Color may consist of either red, green, blue, and alpha values (when OpenGL has been initialized to RGBA mode) or a single color index value (when initialization specified color index mode). One, two, three, or four texture coordinates determine how a texture image maps onto a primitive.

Each of the commands that specify vertex coordinates, normals, colors, or texture coordinates comes in several flavors to accomodate differing application's data formats and numbers of coordinates. Data may also be passed to these commands either as an argument list or as a pointer to a block of storage containing the data. The variants are distinguished (in the `C` language) by mnemonic suffixes.

Most OpenGL commands that do not specify vertices and associated information may not appear between **glBegin** and **glEnd**. This restriction allows implementations to run in an optimized mode while processing primitive specifications so that primitives may be processed as efficiently as possible.

When a vertex is specified, the current color, normal, and texture coordinates are used to obtain values that are then associated with the vertex (Figure 2). The vertex itself is transformed by the *model-view matrix*, a

4×4 matrix which can represent both linear and translational transformations. The color is obtained from either computing a color from lighting or, if lighting is disabled, from the current color. Texture coordinates are similarly passed through a *texture coordinate generation function* (which may be the identity). The resulting texture coordinates are transformed by the *texture matrix* (this matrix may be used to effectively scale or rotate a texture that is applied to a primitive). Figure 4 shows some results of using texture coordinate generation functions.

A number of commands control the values of parameters used in processing a vertex. One group of commands manipulates transformation matrices; these commands are designed to form an efficient means for generating and manipulating the transformations that occur in hierarchical 3D graphics scenes. A matrix may be loaded or multiplied by a scaling, rotation, translation, or general matrix. Another command controls which matrix is affected by a manipulation: the model-view matrix, the texture matrix, or the *projection* matrix (to be described presently). Each of these three matrix types also has an associated stack onto which matrices may be pushed or popped.

Lighting parameters are grouped into three categories: material parameters, that describe the reflectance characteristics of the surface being lit, light source parameters, that describe the emission properties of each light source, and lighting model parameters, that describe global properties of the lighting model. Lighting is performed on a per-vertex basis; lighting results are eventually interpolated across a line segment or polygon. The general form of the lighting equation includes terms for constant, diffuse, and specular illumination, each of which may be attenuated by the distance of the vertex from the light source. A programmer may sacrifice realism in favor of faster lighting calculations by indicating that the viewer, the light sources, or both should be assumed to be infinitely far from the scene. Figure 3 shows some results with lighting disabled and enabled.

4.2 Clipping and Projection

Once a primitive has been assembled from a group of vertices, it is subjected to clipping by *clip planes*. The positions of these planes (every OpenGL implementation must provide at least six) is specifiable using the **glClipPlane** command. Each plane may be enabled or disabled individually.

In the case of a point, the clip planes either have no effect on the point or annihilate it depending as the point lies inside or outside the intersection of

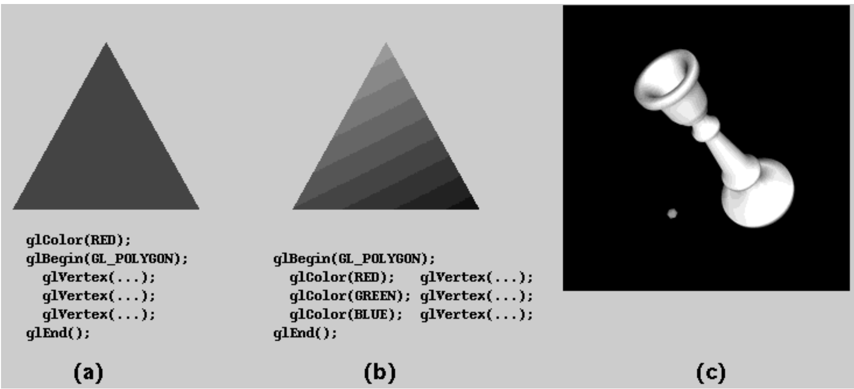

Figure 3. (a) A uniformly colored triangle. (b) A Gouraud shaded triangle. (c) A scene consisting of many lit, shaded polygons.

the half-spaces determined by the clip planes. In the case of a line segment or polygon, the clip planes may have no effect on, annihilate, or alter the original primitive. In the later case, new vertices may be created between edges described by original vertices; color and texture coordinate values for these new vertices are found by appropriately interpolating the values assigned to the original vertices.

After the clip planes (if any) have been applied, the vertex coordinates of the resulting primitive are transformed by the projection matrix. Then *view frustum* clipping occurs. View frustum clipping is like clip plane application, but with fixed planes: if coordinates after transformation are given by (x, y, z, w), then the six half spaces defined by these planes are $-w \le x$, $x \le w$, $-w \le y$, $y \le w$, $-w \le z$, $z \le w$.

With view frustum clipping completed, each group of vertex coordinates is projected by computing x/w, y/w, and z/w. The resulting values (which must each lie in [-1,1]) are multiplied and offset by parameters that control the size of the viewport into which primitives are to be drawn. The **glViewport** (for x/w and y/w) and **glDepthRange** (for z/w) commands control these parameters.

4.3 Rasterization

Rasterization converts a projected, viewport-scaled primitive into a series of *fragments*. Each fragment comprises a location of a pixel in the framebuffer along with color, texture coordinates, and depth (z). When a line segment or polygon is rasterized, these associated data are interpolated across the primitive to obtain a value for each fragment.

The rasterization of each kind of primitive is controlled by a corresponding group of parameters. One width affects point rasterization and another affects line segment rasterization. Additionally, a stipple sequence may be specified for line segments, and a stipple pattern may be speicifed for polygons.

Antialiasing may be enabled or disabled individually for each primitive type. When enabled, a coverage value is computed for each fragment describing the portion of that fragment that is covered by the projected primitive. This coverage value is used after texturing has been completed to modify the fragment's alpha value (in RGBA mode) or color index value (in color index mode).

4.3.1 Pixel Rectangles and Bitmaps

Pixel rectangles and *bitmaps* are the two primitives that are unaffected by the geometric operations that occur in the pipeline prior to rasterization. A pixel rectangle is a group of values destined for the framebuffer (typically the values represent colors, although provision is made for other types of data, such as depth values). The values, stored as a block of data in host memory, are sent using **glDrawPixels**. Arguments to **glDrawPixels** indicate the memory address of the data, the type of data, and the width and height of the rectangle that the data values form. In addition, two groups of parameters are maintained that control the decoding of the stored values. The first group describes how the values are packed in memory and provides a means for selecting a subrectangle from a larger containing rectangle. The second group controls conversions that may be applied to the values after they obtained: values may be scaled, offset and mapped by means of lookup tables. These various parameters form a flexible means for specifying rectangular images stored in a variety of formats.

Once obtained, the resulting values produce a rectangle of fragments. The location of this rectangle is controlled by the *current raster position*, which is treated very much like a point (including associating a color and texture coordinates with it), except that it is set with a separate command (**glRasterPos**) that does not occur between **glBegin** and **glEnd**. The rectangle's size is determined by its specified width and height as well as the setting of pixel rectangle zoom parameters (set with **glPixelZoom**).

A bitmap is similar to a pixel rectangle, except that it specifies a rectangle of zeros and ones, and is designed for describing characters that can be placed at a projected 3D location (through the current raster position). Each one in the bitmap produces a fragment whose associated values are those of the current raster position, while each zero produces no fragment. The **glBitmap** command also specifies offsets that control how the bitmap is placed with respect to the current raster position and how the current raster position is advanced after the bitmap is drawn (thus determining the relative positions of sequential bitmaps).

4.4 Texturing and Fog

OpenGL provides a general means for generating texture-mapped primitives (Figure 4). When texturing is enabled, each fragment's texture coordinates index a texture image, generating a *texel*. This texel may have between one and four components, so that a texture image may represent, for example, intensity only (one component), RGB color (three components), or RGBA color (four components). Once the texel is obtained, it modifies the fragment's color according to a specifiable texture *environment*.

A texture image is specified using **glTexImage**, which takes arguments similar to those of **glDrawpixels**, so that the same image format may be used whether that image is destined for the framebuffer or texture memory. In addition, **glTexImage** may be used to specify mipmaps[3] so that a texture may be filtered as it is applied to a primitive. The filter function (and whether or not it implies mipmaps) is controlled by a number of specifiable parameters using **glTexParameter**. The texture environment is selected with **glTexEnv**.

Finally, after texturing, a fog function (if enabled) is applied to each fragment. The fog function blends the incoming color with a constant (specifiable) fog color according to a computed weighting factor. This factor is a function of the distance (or an approximation to the distance) from the viewer to the 3D point that corresponds to the fragment. Exponential fog simulates atmospheric fog and haze, while linear fog may be used to produce depth-cueing.

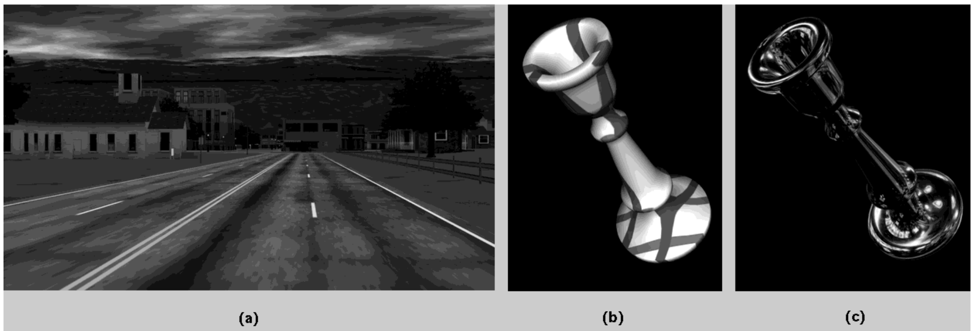

Figure 4. (a) A scene with a number of textures mapped onto primitives. (b) Contouring achieved with texture mapping and a texture coordinate generation function. (c) Reflectance mapping with a texture coordinate generation function.

4.5 The Framebuffer

The destination of rasterized fragments is the framebuffer, where the results of OpenGL rendering may be displayed. In OpenGL, the framebuffer consists of a rectangular array of pixels corresponding to the window allocated for OpenGL rendering. Each pixel is simply a set of some number of bits. Corresponding bits from each pixel in the framebuffer are grouped together into a *bitplane*; each bitplane contains a single bit from each pixel.

The bitplanes are grouped into several logical buffers: the *color*, *depth*, *stencil*, and *accumulation* buffers. The color buffer is where fragment color information is placed. The depth buffer is where fragment depth information is placed, and is typically used to effect hidden surface removal through z-buffering. The stencil buffer contains values each of which may be updated whenever a corresponding fragment reaches the framebuffer. Stencil values are useful in multi-pass algorithms, in which a scene is rendered several times, to achieve such effects as CSG (union, intersection, and difference) operations on a number of objects and capping of objects sliced by clip planes.

The accumulation buffer is also useful in multipass algorithms; it can be manipulated so that it averages values stored in the color buffer. This can effect such effects as full-screen anti-aliasing (by jittering the viewpoint for each pass), depth-of-field (by jittering the angle of view), and motion blur (by stepping the scene in time)[2]. Multi-pass algorithms are simple to implement in OpenGL, because only a small number of parameters must be manipulated before each pass, and changing the values of these parameters is both efficient and without side effects on the values of other parameters that must remain constant.

OpenGL supports both double-buffering and stereo, so the color buffer is further subdivided into four buffers: the front left & right buffers and the back left & right buffers. The front buffers are those that are typically displayed while the back buffers (in a double-buffered application) are being used to compose the next frame. A monoscopic application would use only the left buffers. In addition, there may be some number of auxiliary buffers (these are never displayed) into which fragments may be rendered. Any of the buffers may be individually enabled or disabled for fragment writing.

A particular copy of OpenGL may not provide depth, stencil, accumulation, or auxiliary buffers. Further, only some subset of the left & right front and left & right back buffers may be present. Different buffers may be available (each with varying numbers of bits) depending on the platform and window system on which OpenGL is running. Every window system must, however, provide at least one window type with a front (left) color buffer, and depth, stencil, and accumulation buffers. This guarantees a minimum configuration that a programmer may assume is present no matter where an OpenGL program is run.

4.6 Per-Fragment Operations

Before being placed into its corresponding frame buffer location, a fragment is subjected to a series of tests and modifications, each of which may be individually enabled, disabled, and controlled. The tests and modifications include the stencil test, the depth buffer test (typically used to achieve hidden surface removal), and blending. We briefly describe only a subset of the tests; for specifics, the reader should consult [10].

The stencil test, when enabled, compares the value in the stencil buffer corresponding to the fragment with a reference value. If the comparison succeeds, then the stored stencil value may be modified by a function such as increment, decrement, or clear, and the fragment proceeds to the next test. If the test fails, the stored value may be updated using a different function, and the fragment is discarded. Similarly, the depth buffer test compares the fragment's depth value with the corresponding value stored in the depth buffer. If the comparison succeeds, the fragment is passed to the next stage, and the fragment's depth value replaces the value stored in the depth buffer (if the depth buffer has been enabled for writing). If the comparison fails, the fragment is discarded, and no depth buffer modification occurs.

Blending mixes a fragment's color with the corresponding color already stored in the framebuffer (blending occurs once for each color buffer enabled for writing). The exact blending function may be specified with **glBlend-Function**.

Blending is the operation that actually achieves antialiasing for RGBA colors. Recall that the coverage computation only modifies a fragment's alpha value; this alpha value must be used to blend the fragment color with the already stored background color to obtain the antialiasing effect. Blending is also used to achieve transparency.

In addition to modifying individual framebuffer values with a series of fragments, a whole buffer or buffers may be cleared to some specifiable constant value. Clear values are maintained for the color buffers (all color buffers share a single value), the stencil buffer, the depth buffer, and the accumulation buffer.

4.7 Miscellaneous Functions

4.7.1 Evaluators

Evaluators allow the specification of polynomial functions of one or two variables whose values determine primitives' vertex coordinates, normal coordinates, color, or texture coordinates. A polynomial map, specified in terms of the Bezier basis[1] may be given for any of these groups of values. Once defined and enabled, the maps are invoked in one of two ways. The first way is to cause a single evaluation of each enabled map by specifying a point in the maps' domain using **glEvalCoord**. This command is meant to be placed between **glBegin** and **glEnd** so that individual primitives may be built each of which approximates a portion of a curve or surface. The second method is to specify a grid in domain space using **glEvalMesh**. Each vertex of the evaluated grid is a function of the defined polynomials. **glEvalMesh** generates its own primitives, and thus cannot be placed between **glBegin** and **glEnd**.

The evaluator interface provides a basis for building a more general curve and surface package on top of OpenGL. One advantage of providing the evaluators in OpenGL instead of a more complex NURBS interface is that applications that represent curves and surfaces as other than NURBS or that make use of special surface properties still have access to efficient polynomial evaluators (that may be implemented in graphics hardware) without incurring the costs of converting to a NURBS representation.

4.7.2 Display Lists

A display list encapsulates a group of OpenGL commands so that they may be later issued (in the order originally specified) by simply naming the display list. This is accomplished by surrounding the commands to be encapsulated with **glBeginList** and **glEndList**. **glBeginList** takes an integer argument that is the numeric name of the display list.

Display lists may be redefined, but not edited. The lack of editing simplifies display list memory management in the OpenGL server, eliminating the performance penalty such management would incur. Display lists may, however, be nested (one display list may invoke another). An effect similar to display list editing may thus be obtained by: (1) building a list containing a number of subordinate lists; (2) redefining the subordinate lists.

A single display list is invoked with **glCallList**. **glCallLists** calls a series of display lists in succession. Arguments to **glCallLists** specify an array of integers that are added to a *list base* to form the series of display list numbers. **glCallLists** is useful to display a string of characters when the commands that generate each character have been encapsulated in their own display list. Section 6 gives an example using **glCallLists**.

4.7.3 Feedback and Selection

As described so far, OpenGL renders primitives into the framebuffer. OpenGL has two additional modes. *Feedback* mode returns information about primitives (vertex coordinates, color, and texture coordinates) after they have been processed but before they are rasterized. This mode is useful, for instance, if OpenGL output is to be fed to a pen plotter instead of a framebuffer.

In *selection* mode, OpenGL returns a *hit* whenever a (clipped) primitive lies within the view frustum. This mode is used, for instance, to determine which portions of a scene lie within a region of the window centered around the mouse position (this is often termed *picking*). The Utility Library provides routines to manipulate the transformations so that when the scene is redrawn, only those portions that lie within a specified region about a specified position will return hits. Each hit returns the contents of the *selection stack*, which may be manipulated as the scene is drawn. By appropriately manipulating the stack, the application can identify the scene features that intersected the selection region.

4.7.4 OpenGL State

Finally, the value of nearly any OpenGL parameter may be obtained by an appropariate *get* command. There is also a stack of parameter values that may be pushed and popped. For stacking purposes, all parameters are divided into 21 functional groups; any combination of these groups may be pushed onto the attribute stack in one operation (a pop operation automatically restores only those values that were last pushed). The get commands and parameter stacks make it possible to implement various libraries, each without interfering with another's OpenGL usage.

5 Integration in a Window System

OpenGL draws 3D and 2D scenes into a framebuffer, but to be useful in a heterogeneous environment, OpenGL must be made subordinate to a window system that allocates and controls framebuffer resources. We describe how OpenGL is integrated into the X Window System, but integration into other window systems (Windows NT, for instance) is similar.

X provides both a procedural interface and a network protocol for creating and manipulating framebuffer windows and drawing certain 2D objects into those windows. OpenGL is integrated into X by making it a formal X extension called *GLX*. GLX consists of about a dozen calls (with corresponding network encodings) that provide a compact, general embedding of OpenGL in X. As with other X extensions (two examples are Display PostScript and PEX), there is a specific network protocol for OpenGL rendering commands encapsulated in the X byte stream.

OpenGL requires a region of a framebuffer into which primitives may be rendered. In X, such a region is called a *drawable*. A *window*, one type of drawable, has associated with it a *visual* that describes the window's framebuffer configuration. In GLX, the visual is extended to include information about OpenGL buffers that are not present in unadorned X (depth, stencil, accumulation, front, back, etc.).

X also provides a second type of drawable, the *pixmap*, which is an off-screen framebuffer. GLX provides a *GLX pixmap* that corresponds to an X pixmap, but with additional buffers as indicated by some visual. The GLX pixmap provides a means for OpenGL applications to render off-screen into a software buffer.

To make use of an OpenGL-capable drawable, the programmer creates an OpenGL context targeted to that drawable. When the context is created, a copy of an OpenGL renderer is initialized with the visual information about the drawable. This OpenGL renderer is conceptually (if not actually) part of the X server, so that, once created, an X client may *connect* to the OpenGL context and issue OpenGL commands (Figure 5). Multiple OpenGL contexts may be created that are targeted to distinct or shared drawables. Any OpenGL-capable drawable may also be used for standard X drawing (those buffers of the drawable that are unused by X are ignored by it). Calls are provided to synchronize drawing between OpenGL and X; it is the client's responsibility to carry out this synchronization if required.

A GLX client that is running on a computer of which the graphics subsystem is a part may avoid passing OpenGL tokens through the X server. Such direct rendering may result in increased graphics performance since the overhead of token encoding, decoding, and dispatching is eliminated. Direct rendering is supported but not required by GLX (a client may determine whether or not a server provides direct rendering). Direct rendering is feasible because sequentiality need not be maintained between X commands and OpenGL commands except where commands are explicitly synchronized.

6 Example: Three Kinds of Text

To illustrate the flexibility of OpenGL in performing different types of rendering tasks, we outline three methods for the particular task of displaying text. The three methods are: using bitmaps, using line segments to generate outlined text, and using a texture to generate antialiased text.

The first method defines a font as a series of display lists, each of which contains a single bitmap:

```
for i = start + 'a'  to  start + 'z' {
  glBeginList(i);
    glBitmap( ... );
  glEndList();
}
```

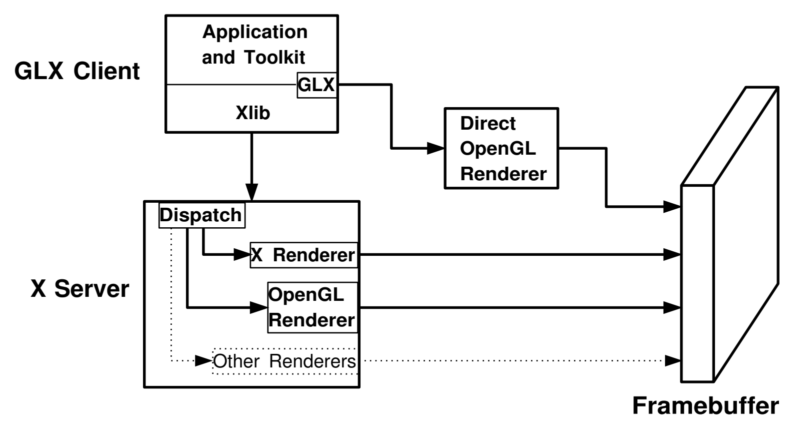

Figure 5. GLX client, X server, and OpenGL renderers.

Recall that `glBitmap` specifies both a pointer to an encoding of the bitmap and offsets that indicate how the bitmap is positioned relative to previous and subsequent bitmaps. In GLX, the effect of defining a number of display lists in this way may also be achieved by calling `glXUseXFont`. `glXUseXFont` generates a number of display lists, each of which contains the bitmap (and associated offsets) of a single character from the specified X font. In either case, the string "Bitmapped Text" whose origin is the projection of a location in 3D is produced by

```
glRasterPos3i(x, y, z);
glListBase(start);
glCallLists("Bitmapped Text", 14, GL_BYTE);
```

See Figure 6a. `glListBase` sets the display list base so that the subsequent `glCallLists` references the characters just defined. The second argument to `glCallLists` indicates the length of the string; the third argument indicates that the string is an array of 8-bit bytes (16- and 32-bit integers may be used to access fonts with more than 256 characters).

The second method is similar to the first, but uses line segments to outline each character. Each display list contains a series of line segments:

```
glTranslate(ox, oy, 0);
```

```
glBegin(GL_LINES);
  glVertex(...);
     ...
glEnd();
glTranslate(dx-ox, dy-oy, 0);
```

The initial `glTranslate` updates the transformation matrix to position the character with respect to a character origin. The final `glTranslate` updates that character origin in preparation for the following character. A string is displayed with this method just as in the previous example, but since line segments have 3D position, the text may be oriented as well as positioned in 3D (Figure 6b). More generally, the display lists could contain both polygons and line segments, and these could be antialiased.

Finally, a different approach may be taken by creating a texture image containing an array of characters. A certain range of texture coordinates thus corresponds to each character in the texture image. Each character may be drawn in any size and in any 3D orientation by drawing a rectangle with the appropriate texture coordinates at its vertices:

```
glTranslate(ox, oy, 0);
glBegin(GL_QUADS)
  glTexCoord( ... );
  glVertex( ... );
     ...
glEnd();
glTranslate(dx-ox, dy-oy, 0);
```

If each group of commands for each character is enclosed in a display list, and the commands for describing the texture image itself are enclosed in another display list called `TEX`, then the string "Texture mapped text!!" may be displayed by:

```
glCallList(TEX);
glCallLists("Texture mapped text!!", 22, GL_BYTE);
```

One advantage of this method is that, by simply using appropriate texture filtering, the resulting characters are antialiased (Figure 6c).

7 Conclusion

OpenGL is a flexible procedural interface that allows a programmer to describe a variety of 3D rendering tasks. It does not enforce a particular

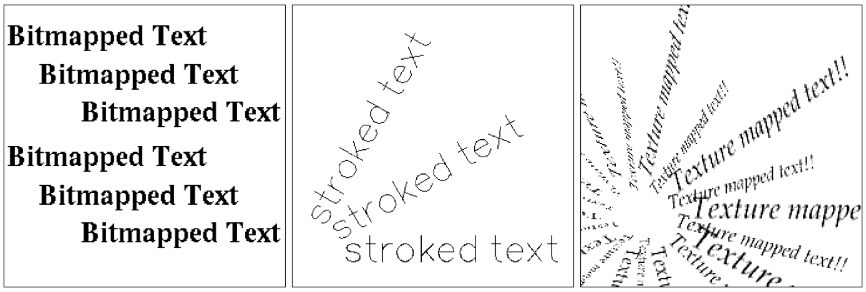

Figure 6. (a) Bitmap example. (b) Stroke font example. (c) Texture mapped font example.

method of describing 3D objects, but instead provides the basic means by which those objects, no matter how described, may be rendered. This mechanistic view of rendering provides for efficient use of graphics hardware, whether that hardware is a simple framebuffer or a graphics subsystem capable of directly manipulating 3D data. OpenGL is rendering-only, so it is independent of the methods by which user input and other window system functions are achieved, making the rendering portions of a graphical program that uses OpenGL platform-independent.

Because OpenGL imposes minimum structure on 3D rendering, it is an excellent base on which to build libraries for handling structured geometric objects, no matter what the particular structures may be. Examples of such libraries include object-oriented graphics toolkits that provide methods to display and manipulate complex objects endowed with a variety of attributes[11][12]. A library that uses OpenGL for its rendering inherits OpenGL's platform independence, making such a library available to a wide programming audience.

References

[1] Gerald Farin. *Curves and Surfaces for Computer Aided Geometric Design*. Academic Press, Boston, Ma., second edition, 1990.

[2] Paul Haeberli and Kurt Akeley. The accumulation buffer: Hardware support for high-quality rendering. In *Proceedings of SIGGRAPH '90*,

pages 309–318, 1990.

[3] Paul S. Heckbert. A survey of texture mapping. *IEEE CG & A*, pages 56–67, November 1986.

[4] Adobe Systems Incorporated. *PostScript Language Reference Manual.* Addison-Wesley, Reading, Mass., 1986.

[5] International Standards Organization. International standard information processing systems — computer graphics — graphical kernel system for three dimensions (GKS-3D) functional description. Technical Report ISO Document Number 9905:1988(E), American National Standards Institute, New York, 1988.

[6] Jeff Stevenson. PEXlib specification and C language binding, version 5.1P. *The X Resource*, Special Issue B, September 1992.

[7] Adrian Nye. *X Window System User's Guide*, volume 3 of *The Definitive Guides to the X Window System.* O'Reilly and Associates, Sebastapol, Ca., 1987.

[8] Paula Womack, ed. PEX protocol specification and encoding, version 5.1P. *The X Resource*, Special Issue A, May 1992.

[9] PHIGS+ Committee, Andries van Dam, chair. PHIGS+ functional description, revision 3.0. *Computer Graphics*, 22(3):125–218, July 1988.

[10] Mark Segal and Kurt Akeley. The OpenGL graphics system: A specification. Technical report, Silicon Graphics Computer Systems, Mountain View, Ca., 1992.

[11] Paul S. Strauss and Rikk Carey. An object-oriented 3D graphics toolkit. In *Proceedings of SIGGRAPH '92*, pages 341–349, 1992.

[12] Garry Wiegand and Bob Covey. *HOOPS Reference Manual, Version 3.0.* Ithaca Software, 1991.