# The Design of the OpenGL Graphics Interface

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

## Abstract

OpenGL is an emerging graphics standard that provides advanced rendering features while maintaining a simple programming model. Because OpenGL is rendering-only, it can be incorporated into any window system (and has been, into the X Window System and a soon-to-be-released version of Windows) or can be used without a window system. An OpenGL implementation can efficiently accommodate almost any level of graphics hardware, from a basic framebuffer to the most sophisticated graphics subsystems. It is therefore a good choice for use in interactive 3D and 2D graphics applications.

We describe how these and other considerations have governed the selection and presentation of graphical operators in OpenGL. Complex operations have been eschewed in favor of simple, direct control over the fundamental operations of 3D and 2D graphics. Higher-level graphical functions may, however, be built from OpenGL's low-level operators, as the operators have been designed with such layering in mind.

**CR Categories and Subject Descriptors:** I.3.3 **[Computer Graphics]:** Picture/Image Generation; I.3.7 **[Computer Graphics]:** Three-Dimensional Graphics and Realism

## 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 applications that run on a variety of 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.

To be viable, a graphics standard intended for interactive 3D applications must satisfy several criteria. It must be implementable on platforms with varying graphics capabilities without compromising the graphics performance of the underlying hardware and without sacrificing control over the hardware's operation. It must provide a natural interface that allows a programmer to describe rendering operations tersely. Finally, the interface must be flexible enough to accommodate extensions so that as new graphics operations become significant or available in new graphics subsystems, these operations can be provided without disrupting the original interface.

OpenGL meets these criteria by providing a simple, direct interface to the fundamental operations of 3D graphics rendering. It supports basic graphics primitives such as points, line segments, polygons, and images, as well as basic rendering operations such as affine and projective transformations and lighting calculations. It also supports advanced rendering features such as texture mapping and antialiasing.

There are several other systems that provide an API (Application Programmer's Interface) for effecting graphical rendering. In the case of 2D graphics, the PostScript page description language[5] 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. Besides providing graphical rendering operators, PostScript is also a stack-based programming language.

The X window system[9] 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 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 systems are in use. One relatively well-known system is PHIGS (Programmer's Hierarchical Interactive Graphics System). Based on GKS[6] (Graphics Kernel System), PHIGS is an ANSI (American National Standards Institute) standard. PHIGS (and its descendant, PHIGS+[11]) 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 lack of support for advanced rendering features such as texture mapping.

PEX[10] extends X to include the ability to manipulate and draw

3D objects. (PEXlib[7] is an API employing the PEX protocol.) Originally based on PHIGS, PEX allows *immediate mode* rendering, meaning that objects can be displayed as they are described rather than having to first complete a display list. PEX currently lacks advanced rendering features (although a compatible version that provides such features is under design), and is available only to users of X. Broadly speaking, however, the methods by which graphical objects are described for rendering using PEX (or rather, PEXlib) are similar to those provided by OpenGL.

Like both OpenGL and PEXlib, Renderman[16] is an API that provides a means to render goemetric objects. Unlike these interfaces, however, Renderman provides a programming language (called a shading language) for describing how these objects are to appear when drawn. This programmability allows for generating very realistic-looking images, but it is impractical to implement on most graphics accelerators, making Renderman a poor choice for interactive 3D graphics.

Finally, there are APIs that provide access to 3D rendering as a result of methods for describing higher-level graphical objects. Chief among these are HOOPS[17] and IRIS Inventor[15]. The objects provided by these interfaces are typically more complex than the simple geometry describable with APIs like PEXlib or OpenGL; they may comprise not only geometry but also information about how they are drawn and how they react to user input. HOOPS and Inventor free the programmer from tedious descriptions of individual drawing operations, but simple access to complex objects generally means losing fine control over rendering (or at least making such control difficult). In any case, OpenGL can provide a good base on which to build such higher-level APIs.

# 2 OpenGL

In this section we present a brief overview of OpenGL. For a more comprehensive description, the reader is referred to [8] or [13].

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 issuing *commands* in the form of function or procedure calls.

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 Design Considerations

Designing any API requires tradeoffs between a number of general factors like simplicity in accomplishing common operations vs. generality, or many commands with few arguments vs. few commands with many arguments. In this section we describe considerations peculiar to 3D API design that have influenced the development of OpenGL.

## 3.1 Performance

A fundamental consideration in interactive 3D graphics is performance. Numerous calculations are required to render a 3D scene of even modest complexity, and in an interactive application, a scene must generally be redrawn several times per second. An API for use in interactive 3D applications must therefore provide efficient access to the capabilities of the graphics hardware of which it makes use. But different graphics subsystems provide different capabilities, so a common interface must be found.

The interface must also provide a means to switch on and off various rendering features. This is required both because some hardware may not provide support for some features and so cannot provide those features with acceptable performance, and also because even with hardware support, enabling certain features or combinations of features may decrease performance significantly. Slow rendering may be acceptable, for instance, when producing a final image of a scene, but interactive rates are normally required when manipulating objects within the scene or adjusting the viewpoint. In such cases the performance-degrading features may be desirable for the final image, but undesirable during scene manipulation.

## 3.2 Orthogonality

Since it is desirable to be able to turn features on and off, it should be the case that doing so has few or no side effects on other features. If, for instance, it is desired that each polygon be drawn with a single color rather than interpolating colors across its face, doing so should not affect how lighting or texturing is applied. Similarly, enabling or disabling any single feature should not engender an inconsistent state in which rendering results would be undefined. These kinds of feature independence are necessary to allow a programmer to easily manipulate features without having to generate tests for particular illegal or undesirable feature combinations that may require changing the state of apparently unrelated features. Another benefit of feature independence is that features may be combined in useful ways that may have been unforseen when the interface was designed.

## 3.3 Completeness

A 3D graphics API running on a system with a graphics subsystem should provide some means to access all the significant functionality of the subsystem. If some functionality is available but not provided, then the programmer is forced to use a different API to get at the

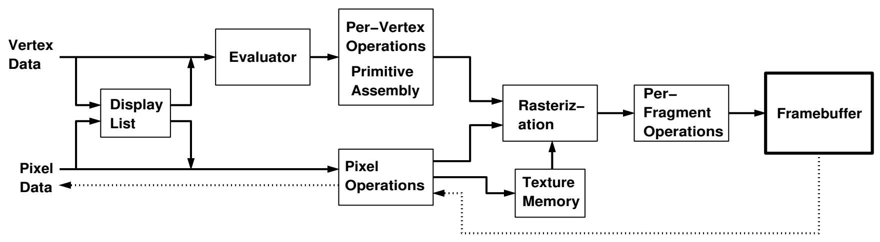



Figure 1. Block diagram of OpenGL.

missing features. This may complicate the application because of interaction between the two APIs.

On the other hand, if an implementation of the API provides certain features on one hardware platform, then, generally speaking, those features should be present on any platform on which the API is provided. If this rule is broken, it is difficult to use the API in a program that is certain to run on diverse hardware platforms without remembering exactly which features are supported on which machines. In platforms without appropriate acceleration, some features may be poor performers (because they may have to be implemented in software), but at least the intended image will eventually appear.

### 3.4 Interoperability

Many computing environments consist of a number of computers (often made by different companies) connected together by a network. In such an environment it is useful to be able to issue graphics commands on one machine and have them execute on another (this ability is one of the factors responsible for the success of X). Such an ability (called *interoperability*) requires that the model of execution of API commands be *client-server*: the client issues commands, and the server executes them. (Interoperability also requires that the client and the server share the same notion of how API commands are encoded for transmission across the network; the client-server model is just a prerequisite.) Of course the client and the server may be the same machine.

Since API commands may be issued across a network, it is impractical to require a tight coupling between client and server. A client may have to wait for some time for an answer to a request presented to the server (a *roundtrip*) because of network delays, whereas simple server requests not requiring acknowledgement can be buffered up into a large group for efficient transmission to and execution by the server.

### 3.5 Extensibility

As was discussed in the introduction, a 3D graphics API should, at least in principle, be extendable to incorporate new graphics hardware features or algorithms that may become popular in the future. Although attainment of this goal may be difficult to gauge until long after the API is first in use, steps can be taken to help to achieve it. Orthogonality of the API is one element that helps achieve this goal. Another is to consider how the API would have been affected if features that were consciously omitted were added to the API.

### 3.6 Acceptance

It might seem that design of a clean, consistent 3D graphics API would be a suficient goal in itself. But unless programmers decide to use the API in a variety of applications, designing the API will have served no purpose. It is therefore worthwhile to consider the effect of design decisions on programmer acceptance of the API.

## 4 Design Features

In this section we highlight the general features of OpenGL's design and provide illustratations and justifications of each using specific examples.

### 4.1 Based on IRIS GL

OpenGL is based on Silicon Graphics' IRIS GL. While it would have been possible to have designed a completely new API, experience with IRIS GL provided insight into what programmers want and don't want in a 3D graphics API. Further, making OpenGL similar to IRIS GL where possible makes OpenGL much more likely to be accepted; there are many successful IRIS GL applications, and programmers of IRIS GL will have an easy time switching to OpenGL.

### 4.2 Low-Level

An essential goal of OpenGL is to provide device independence while still allowing complete access to hardware functionality. The API therefore provides access to graphics operations at the lowest possible level that still provides device independence. As a result, OpenGL 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 object themselves.

#### The OpenGL Utility Library

One benefit of a low-level API is that there are no requirements on how an application must represent or describe higher-level objects (since there is no notion of such objects in the API). Adherence to this principle means that the basic OpenGL API does not support some geometric objects that are traditionally associated with graphics APIs. 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. Another reason for the omission is that to render a general concave polygon, all of its vertices must first be known. Graphics subsystems do not generally provide the storage necessary for a concave polygon with a (nearly) arbitrary number of vertices. Convex polygons, on the other hand, can be reduced to triangles as they are specified, so no more than three vertices need be stored.

Another example of the distinction between low-level and high-level in OpenGL is the difference between OpenGL evaluators and NURBS. The evaluator interface provides a basis for building a general polynomial 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.

Concave polygons and NURBS are, however, common and useful operators, and they were familiar (at least in some form) to users of IRIS GL. Therefore, 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 of 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 for rendering.

In the client-server environment, a utility library raises an issue: utility library commands are converted into OpenGL commands on the client; if the server computer is more powerful than the client, the client-side conversion might have been more effectively carried out on the server. This dilemma arises not just with OpenGL but with any library in which the client and server may be distinct computers. In OpenGL, the base functionality reflects the functions efficiently performed by advanced graphics subsystems, because no matter what the power of the server computer relative to the client, the server's graphics subsystem is assumed to efficiently perform the functions it provides. If in the future, for instance, graphics subsystems commonly provide full trimmed NURBS support, then such functionality should likely migrate from the Utility Library to OpenGL itself. Such a change would not cause any disruption to the rest of the OpenGL API; another block would simply be added to the left side in Figure 1.

## 4.3 Fine-Grained Control

In order to minimize the requirements on how an application using the API must store and present its data, the API must provide a means to specify individual components of geometric objects and operations on them. This fine-grained control is required so that these components and operations may be specified in any order and so that control of rendering operations is flexible enough to accommodate the requirements of diverse applications.

**Vertices and Associated Data**

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();
```

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 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 by mnemonic suffixes.

Using a procedure call to specify each individual group of data that together define a primitive means that an application may store data in any format and order that it chooses; data need not be stored in a form convenient for presentation to the graphics API because OpenGL accomodates almost any data type and format using the appropriate combination of data specification procedures. Another advantage of this scheme is that by simply combining calls in the appropriate order, different effects may be achieved. Figure 2 shows an example of a uniformly colored triangle obtained by specifying a single color that is inherited by all vertices of the triangle; a smooth shaded triangle is obtained by respecifying a color before each vertex. Not every possible data format is supported (byte values may not be given for vertex coordinates, for instance) only because it was found from experience with IRIS GL that not all formats are used. Adding the missing formats in the future, however, would be a trivial undertaking.

One disadvantage of using procedure calls on such a fine grain is that it may result in poor performance if procedure calls are costly. In such a situation an interface that specifies a format for a block of data that is sent all at once may have a performance advantage. The difficulty with specifying a block of data, however, is that it either constrains the application to store its data in one of the supported formats, or it requires the application to copy its data into a block structured in one of those formats, resulting in inefficiency. (Allowing any format arising from an arbitrary combination of individual data types is impractical because there are so many combinations.)

In OpenGL, the maximum flexibility provided by individual procedure calls was deemed more important than any inefficiency induced by using those calls. This decision is partly driven by the consideration that modern compilers and computer hardware have improved to the point where procedure calls are usually relatively

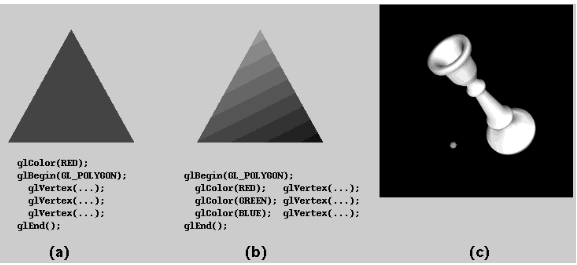



Figure 2. (a) A uniformly colored triangle. (b) A Gouraud shaded triangle. (c) A scene consisting of many lit, shaded polygons.

inexpensive, especially when compared with the work necessary to process the geometric data contained in the call. This is one area in which OpenGL differs significantly from PEX; with PEX, a primitive's vertices (and associated data) are generally presented all at once in a single array. If it turns out that fine-grained procedure calls are too expensive, then it may be necessary to add a few popular block formats to the OpenGL API or to provide a mechanism for defining such formats.

## 4.4 Modal

As a consequence of fine-grained control, OpenGL maintains considerable state, or modes, that determines how primitives are rendered. This state is present in lieu of having to present a large amount of information with each primitive that would describe the settings for all the operations to which the primitive would be subjected. Presenting so much information with each primitive is tedious and would result in excessive data being transmitted from client to server. Therefore, essentially no information is presented with a primitive except what is required to define it. Instead, a considerable proportion of OpenGL commands are devoted to controlling the settings of rendering operations.

One difficulty with a modal API arises in implementations in which separate processors (or processes) operate in parallel on distinct primitives. In such cases, a mode change must be broadcast to all processors so that each receives the new parameters before it processes its next primitive. A mode change is thus processed serially, halting primitive processing until all processors have received the change, and reducing performance accordingly. One way to lessen the impact of mode changes in such a system is to insert a processor that distributes work among the parallel processors. This processor can buffer up a string of mode changes, transmitting the changes all at once only when another primitive finally arrives[1].

Another way to handle state changes relies on defining groups of named state settings which can then be invoked simply by providing the appropriate name (this is the approach taken by X and PEX). With this approach, a single command naming the state setting changes the server's settings. This approach was rejected for OpenGL for several reasons. Keeping track of a number of state vectors (each of which may contain considerable information) may be impractical on a graphics subsystem with limited memory. Named state settings also conflict with the emphasis on fine-grained control; there are cases, as when changing the state of a single mode, when transmitting the change directly is more convenient and efficient than first setting up and then naming the desired state vector. Finally, the named state setting approach may still be used with OpenGL by encapsulating state changing commands in display lists (see below).

#### The Matrix Stack

Three kinds of transformation matrices are used in OpenGL: the *model-view* matrix, which is applied to vertex coordinates; the *texture* matrix, which is applied to texture coordinates; and the *projection* matrix, which describes the viewing frustum and is applied to vertex coordinates after they are transformed by the model-view matrix. Each of these matrices is $4 \times 4$.

Any of one these matrices may be loaded with or multiplied by a general transformation; commands are provided to specify the special cases of rotation, translation and scaling (since these cases take only a few parameters to specify rather than the 16 required for a general transformation). A separate command controls a mode indicating which matrix is currently affected by any of these manipulations. In addition, each matrix type actually consists of a stack of matrices that can be pushed or popped. The matrix on the top of the stack is the one that is applied to coordinates and that is affected by matrix manipulation commands.

The retained state represented by these three matrix stacks simplifies specifying the transformations found in hierarchical graphical data structures. Other graphics APIs also employ matrix stacks, but often only as a part of more general attribute structures. But OpenGL is unique in providing three kinds of matrices which can be manipulated with the same commands. The texture matrix, for instance, can be used to effectively rotate or scale a texture image applied to primitive, and when combined with perspective viewing transformations, can even be used to obtain projective texturing effects such as spotlight simulation and shadow effects using shadow maps[14].

#### State Queries and Attribute Stacks

The value of nearly any OpenGL parameter may be obtained by an appropriate *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 are required so that various libraries may make use of OpenGL efficiently without interfering with one another.

## 4.5 Framebuffer

Most of OpenGL requires that the graphics hardware contain a framebuffer. This is a reasonable requirement since nearly all interactive graphics applications (as well as many non-interactive ones) run on systems with framebuffers. Some operations in OpenGL are achieved only through exposing their implementation using a framebuffer (transparency using alpha blending and hidden surface removal using depth buffering are two examples). Although OpenGL may be used to provide information for driving such devices as pen-plotters and vector displays, such use is secondary.

#### Multipass Algorithms

One useful effect of making the framebuffer explicit is that it enables the use of multipass algorithms, in which the same primitives are rendered several times. One example of a multipass algorithm employs an *accumulation buffer*[3]: a scene is rendered several times, each time with a slightly different view, and the results averaged in the framebuffer. Depending on how the view is altered on each pass, this algorithm can be used to achieve full-window anti-aliasing, depth-of-field effects, motion blur, or combinations of these. Multipass algorithms are simple to implement in OpenGL, because only a small number of parameters must be manipulated between passes, and changing the values of these parameters is both efficient and without side effects on other parameters that must remain constant.

#### Invariance

Consideration of multipass algorithms brings up the issue of how what is drawn in the framebuffer is or is not affected by changing parameter values. If, for instance, changing the viewpoint affected the way in which colors were assigned to primitives, the accumulation buffer algorithm would not work. For a more plausible example, if some OpenGL feature is not available in hardware, then an OpenGL implementation must switch from hardware to software when that feature is switched on. Such a switch may significantly affect what eventually reaches the framebuffer because of slight differences in the hardware and software implementations.

The OpenGL specification is not pixel exact; it does not indicate the exact values to which certain pixels must be set given a certain input. The reason is that such specification, besides being difficult, would be too restrictive. Different implementations of OpenGL run on different hardware with different floating-point formats, rasterization algorithms, and framebuffer configurations. It should be possible, nonetheless, to implement a variety of multipass algorithms and expect to get reasonable results.

For this reason, the OpenGL specification gives certain invariance rules that dictate under what circumstances one may expect identical results from one particular implementation given certain inputs (implementations on different systems are never required to produce identical results given identical inputs). These rules typically indicate that changing parameters that control an operation cannot affect the results due to any other operation, but that such invariance is not required when an operation is turned on or off. This makes it possible for an implementation to switch from hardware to software when a mode is invoked without breaking invariance. On the other hand, a programmer may still want invariance even when toggling some mode. To accommodate this case, any operation covered by the invariance rules admits a setting of its controlling parameters that cause the operation to act as if it were turned off even when it is on. A comparison, for instance, may be turned on or off, but when on, the comparison that is performed can be set to always (or never) pass.

### 4.6 Not Programmable

OpenGL does not provide a programming language. Its function may be controlled by turning operations on or off or specifying parameters to operations, but the rendering algorithms are essentially fixed. One reason for this decision is that, for performance reasons, graphics hardware is usually designed to apply certain operations in a specific order; replacing these operations with arbitrary algorithms is usually infeasible. Programmability would conflict with keeping the API close to the hardware and thus with the goal of maximum performance.

#### The Graphics Pipeline and Per-Fragment Operations

The model of command execution in OpenGL is that of a pipeline with a fixed topology (although stages may be switched in or out). The pipeline is meant to mimic the organization of graphics subsystems. The final stages of the pipeline, for example, consist of a series of tests on and modifications to fragments before they are eventually placed in the framebuffer. To draw a complex scene in a short amount of time, many fragments must pass through these final stages on their way to the framebuffer, leaving little time to process each fragment. Such high *fill rates* demand special purpose hardware that can only perform fixed operations with minimum access to external data.

Even though fragment operations are limited, many interesting and useful effects may be obtained by combining the operations appropriately. Per-fragment operations provided by OpenGL include

- alpha blending: blend a fragment's color with that of the corresponding pixel in the framebuffer based on an alpha value;
- depth test: compare a depth value associated with a fragment with the corresponding value already present in the framebuffer and discard or keep the fragment based on the outcome of the comparison;
- stencil test: compare a reference value with a corresponding value stored in the framebuffer and update the value or discard the fragment based on the outcome of the comparison.

Alpha blending is useful to achieve transparency or to blend a fragment's color with that of the background when antialiasing; the depth test can effect depth-buffering (and thus hidden surface removal); the stencil test can be used for a number of effects[12], including highlighting interference regions and simple CSG (Constructive Solid Geometry) operations. These (and other) operations may be combined to achieve, for instance, transparent interference regions with hidden surfaces removed, or any number of other effects.

The OpenGL graphics pipeline also induces a kind of orthogonality among primitives. Each vertex, whether it belongs to a point, line segment, or polygon primitive, is treated in the same way: its coordinates are transformed and lighting (if enabled) assigns it a color. The primitive defined by these vertices is then rasterized and converted to fragments, as is a bitmap or image rectangle primitive. All fragments, no matter what their origin, are treated identically. This homogeneity among operations removes unneeded special cases (for each primitive type) from the pipeline. It also makes natural the combination of diverse primitives in the same scene without having to set special modes for each primitive type.

### 4.7 Geometry and Images

OpenGL provides support for handling both 3D (and 2D) geometry and 2D images. An API for use with geometry should also provide support for writing, reading, and copying images, because geometry and images are often combined, as when a 3D scene is laid over a background image. Many of the per-fragment operations that are applied to fragments arising from geometric primitives apply equally well to fragments corresponding to pixels in an image, making it easy to mix images with geometry. For example, a triangle may be blended with an image using alpha blending. OpenGL supports a number of image formats and operations on image components (such as lookup tables) to provide flexibility in image handling.

#### Texture Mapping

Texture mapping provides an important link between geometry and images by effectively applying an image to geometry. OpenGL makes this coupling explicit by providing the same formats for specifying texture images as for images destined for the framebuffer.

Besides being useful for adding realism to a scene (Figure 3a), texture mapping can be used to achieve a number of other useful effects[4]. Figures 3b and 3c show two examples in which the texture coordinates that index a texture image are generated from vertex coordinates. OpenGL's orthogonality makes achieving such effects with texture mapping simply a matter of enabling the appropriate modes and loading the appropriate texture image, without affecting the underlying specification of the scene.

## 4.8 Immediate Mode and Display Lists

The basic model for OpenGL command interpretation is immediate mode, in which a command is executed as soon as the server receives it; vertex processing, for example, may begin even before specification of the primitive of which it is a part has been completed. Immediate mode execution is well-suited to interactive applications in which primitives and modes are constantly altered. In OpenGL, the fine-grained control provided by immediate mode is taken as far as possible: even individual lighting parameters (the diffuse reflectance color of a material, for instance) and texture images are set with individual commands that have immediate effect.

While immediate mode provides flexibility, its use can be inefficient if unchanging parameters or objects must be respecified. To accommodate such situations, OpenGL provides display lists. A display list encapsulates a sequence of OpenGL commands (all but a handful of OpenGL commands may be placed in a display list), and is stored on the server. The display list is given a numeric name by the application when it is specified; the application need only name the display list to cause the server to effectively execute all the commands contained within the list. This mechanism provides a straightforward, effective means for an application to transmit a group of commands to the server just once even when those same commands must be executed many times.

#### Display List Optimization

Accumulating commands into a group for repeated execution presents possibilities for optimization. Consider, for example, specifying a texture image. Texture images are often large, requiring a large, and therefore possibly slow, data transfer from client to server (or from the server to its graphics subsystem) whenever the image is respecified. For this reason, some graphics subsystems are equipped with sufficient storage to hold several texture images simultaneously. If the texture image definition is placed in a display list, then the server may be able to load that image just once when it is specified. When the display list is invoked (or re-invoked), the server simply indicates to the graphics subsystem that it should use the texture image already present in its memory, thus avoiding the overhead of respecifying the entire image.

Examples like this one indicate that display list optimization is required to achieve the best performance. In the case of texture image loading, the server is expected to recognize that a display list contains texture image information and to use that information appropriately. This expectation places a burden on the OpenGL implementor to make sure that special display list cases are treated as efficiently as possible. It also places a burden on the application writer to know to use display lists in cases where doing so could improve performance. Another possibility would have been to introduce special commands for functions that can be poor performers in immediate mode. But such specialization would clutter the API and blur the clear distinction between immediate mode and display lists.

#### Display List Hierarchies

Display lists may be redefined in OpenGL, but not edited. The lack of editing simplifies display list memory management on the server, eliminating the penalty that such management would incur. One display list may, however, invoke others. An effect similar to display list editing may thus be obtained by: (1) building a list that invokes a number of subordinate lists; (2) redefining the subordinate lists. This redefinition is possible on a fine grain: a subordinate display list may contain anything (even nothing), including just a single vertex or color command.

There is no automatic saving or restoring of modes associated with display list execution. (If desired, such saving and restoring may be performed explicitly by encapsulating the appropriate commands in the display list.) This allows the highest possible performance in executing a display list, since there is almost no overhead associated with its execution. It also simplifies controlling the modal behavior of display list hierarchies: only modes explicitly set are affected.

Lack of automatic modal behavior in display lists also has a disadvantage: it is difficult to execute display lists in parallel, since the modes set in one display list must be in effect before a following display list is executed. In OpenGL, display lists are generally not used for defining whole scenes or complex portions of scenes but rather for encapsulating groups of frequently repeated mode setting commands (describing a texture image, for instance) or commands describing simple geometry (the polygons approximating a torus, for instance).

## 4.9 Depth buffer

The only hidden surface removal method directly provided by OpenGL is the depth (or $z$) buffer. This assumption is in line with that of the graphics hardware containing a framebuffer. Other hidden surface removal methods may be used with OpenGL (a BSP tree[2] coupled with the painter's algorithm, for instance), but it is assumed that such methods are never supported in hardware and thus need not be supported explicitly by OpenGL.

## 4.10 Local Shading

The only shading methods provided by OpenGL are local. That is, methods for determining surface color such as ray-tracing or radiosity that require obtaining information from other parts of the scene are not directly supported. The reason is that such methods require knowledge of the global scene database, but so far specialized graphics hardware is structured as a pipeline of localized operations and does not provide facilities to store and traverse the large amount of data necessary to represent a complex scene. Global shading methods may be used with OpenGL only if the shading can be precomputed and the results associated with graphical objects before they are transmitted to OpenGL.

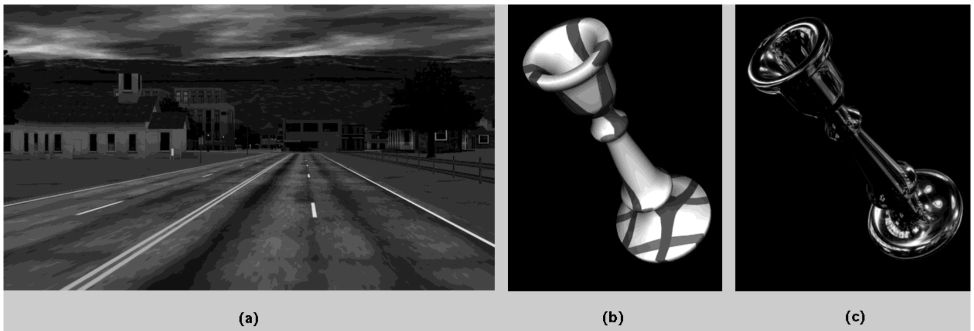



Figure 3. (a) A scene with a number of textures mapped onto primitives. (b) Contouring achieved with texture mapping and a texture coordinate generation function. (c) Reflectance mapping with a texture coordinate generation function.

## 4.11 Rendering Only

OpenGL provides access to rendering operations only. There are no facilities for obtaining user input from such devices as keyboards and mice, since it is expected that any system (in particular, a window system) under which OpenGL runs must already provide such facilities. Further, the effects of OpenGL commands on the framebuffer are ultimately controlled by the window system (if there is one) that allocates framebuffer resources. The window system determines which portions of the framebuffer OpenGL may access and communicates to OpenGL how those portions are structured. These considerations make OpenGL window system independent.

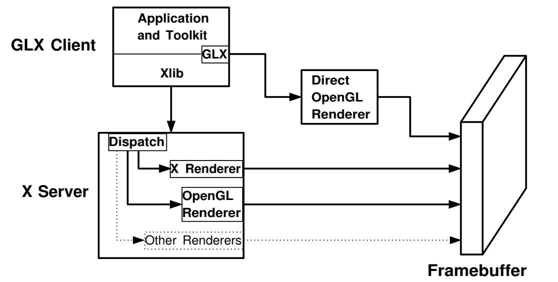



Figure 4. GLX client, X server, and OpenGL renderers.

**Integration in X**

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 4). 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 X).

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. Direct rendering is feasible because sequentiality need not be maintained between X commands and OpenGL commands except where commands are explicitly synchronized.

Because OpenGL comprises rendering operations only, it fits well into already existing window systems (integration into Windows is similar to that described for X) without duplicating operations already present in the window system (like window control or mouse event generation). It can also make use of window system features such as off-screen rendering, which, among other uses, can send the results of OpenGL commands to a printer. Rendering operations provided by the window system may even be interspersed with those of OpenGL.

## 4.12 API not Protocol

PEX is primarily specified as a network protocol; PEXlib is a presentation of that protocol through an API. OpenGL, on the other hand, is primarily specified as an API; the API is encoded in a specified network protocol when OpenGL is embedded in a system (like X) that requires a protocol. One reason for this preference is that an applications programmer works with the API and not with a protocol. Another is that different platforms may admit different protocols (X places certain constraints on the protocol employed by an X extension, while other window systems may impose different constraints). This means that the API is constant across platforms even when the protocol cannot be, thereby making it possible to use the same source code (at least for the OpenGL portion) without regard for any particular protocol. Further, when the client and server are the same computer, OpenGL commands may be transmitted directly to a graphics subsystem without conversion to a common encoding.

Interoperability between diverse systems is not compromised by preferring an API specification over one for a protocol. Tests in which an OpenGL client running under one manufacturer's implementation was connected to another manufacturer's OpenGL server have provided excellent results.

## 5 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:

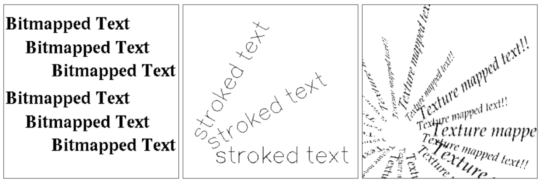
```
for i = start + 'a'  to  start + 'z' {
  glBeginList(i);
    glBitmap( ... );
  glEndList();
}
```

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 5a. glListBase sets the display list base so that the subsequent glCallLists references the characters just defined. glCallLists invokes a series of display lists specified in an array; each value in the array is added to the display list base to obtain the number of the display list to use. In this case the array is an array of bytes representing a string. 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:




(a) (b) (c)

Figure 5. (a) Bitmap example. (b) Stroke font example. (c) Texture mapped font example.

```
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 5b). 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 (along with the setting of the list base) are enclosed in another display list called TEX, then the string "Texture mapped text!!" may be displayed by:

```
glCallList(TEX);
glCallLists("Texture mapped text!!",21,
                                GL_BYTE);
```

One advantage of this method is that, by simply using appropriate texture filtering, the resulting characters are antialiased (Figure 5c).

## 6 Conclusion

OpenGL is a 3D graphics API intended for use in interactive applications. It has been designed to provide maximum access to hardware graphics capabilities, no matter at what level such capabilities are available. This efficiency stems from a flexible interface that provides direct control over fundamental operations. OpenGL does not enforce a particular method of describing 3D objects and how they should appear, but instead provides the basic means by which those objects, no matter how described, may be rendered. Because OpenGL imposes minimum structure on 3D rendering, it provides an excellent base on which to build libraries for handling structured geometric objects, no matter what the particular structures may be.

The goals of high performance, feature orthogonality, interoperability, implementability on a variety of systems, and extensibility have driven the design of OpenGL's API. We have shown the effects of these and other considerations on the presentation of rendering operations in OpenGL. The result has been a straightforward API with few special cases that should be easy to use in a variety of applications.

Future work on OpenGL is likely to center on improving implementations through optimization, and extending the API to handle new techniques and capabilities provided by graphics hardware. Likely candidates for inclusion are image processing operators, new texture mapping capabilities, and other basic geometric primitives such as spheres and cylinders. We believe that the care taken in the design of the OpenGL API will make these as well as other extensions simple, and will result in OpenGL's remaining a useful 3D graphics API for many years to come.

# References


[1] Kurt Akeley. RealityEngine graphics. In *SIGGRAPH 93 Conference Proceedings*, pages 109–116, August 1993.

[2] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. *Computer Graphics (SIGGRAPH '80 Proceedings)*, 14(3):124–133, July 1980.

[3] Paul Haeberli and Kurt Akeley. The accumulation buffer: Hardware support for high-quality rendering. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(2):309–318, July 1990.

[4] Paul Haeberli and Mark Segal. Texture mapping as a fundamental drawing primitive. In *Proceedings of the Fourth Eurographics Workshop on Rendering*, pages 259–266, June 1993.

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

[6] 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.

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

[8] Jackie Neider, Mason Woo, and Tom Davis. *OpenGL Programming Guide*. Addison-Wesley, Reading, Ma., 1993.

[9] 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.

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

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

[12] Jarek Rossignac, Abe Megahed, and Bengt-Olaf Schneider. Interactive inspection of solids: Cross-sections and interferences. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):353–360, July 1992.

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

[14] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadows and lighting effects using texture mapping. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):249–252, July 1992.

[15] Paul S. Strauss and Rikk Carey. An object-oriented 3D graphics toolkit. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):341–349, July 1992.

[16] Steve Upstill. *The RenderMan Companion*. Addison-Wesley, Reading, Mass., 1990.

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