To be published in the
November/December issue of *The X Journal.*

OpenGL™ and X, Part 1: An Introduction

Mark J. Kilgard *
Silicon Graphics Inc.
Revision : 1.16

October 4, 1993

Abstract

The OpenGL™ graphics system is a high-performance, window system independent 2D and 3D graphics interface. The technology was developed by Silicon Graphics and is now controlled by the OpenGL Architecture Review Board. OpenGL's GLX extension integrates OpenGL with the X Window System. This article describes OpenGL's functionality and how it is used with X. A simple OpenGL program using Xlib is presented. OpenGL is compared and contrasted with PEX, a 3D graphics interface designed specifically for X. The two subsequent articles in this series describe how to integrate OpenGL with Xlib and Motif programs.

1 Introduction

The OpenGL™ graphics system is a powerful software interface for graphics hardware that allows graphics programmers to produce high-quality color images of 2D and 3D objects. The technology was developed by Silicon Graphics Inc. (SGI) and is the result of ten years of experience designing production software interfaces for a full spectrum of graphics hardware.

OpenGL is now controlled by an industry consortium known as the OpenGL Architectural Review Board (ARB) currently composed of Digital Equipment, IBM, Intel, Microsoft, and SGI. The interface is licensed to a large number of computer software and hardware vendors and OpenGL implementations are now appearing on the market.

This article is the first of a series of three articles explaining OpenGL to the users of the X Window System. This article introduces the reader to OpenGL's features, particularly how they apply to X. This section will introduce the reader to OpenGL's philosophy and history. Section 2 will explore OpenGL's rich feature set. Section 3 discusses OpenGL's integration with the X Window System via the GLX extension. Section 4 presents a simple OpenGL program for X. Section 5 compares and contrasts OpenGL to PEX, a 3D graphics interface designed specifically for X. Section 6 tells where to find more information about OpenGL.

The second article in the series will explain in more detail how to use OpenGL in conjunction with Xlib. The third article will describe how to use OpenGL with Motif.

1.1 Design Philosophy

To appreciate OpenGL it is useful to understand its design philosophy. OpenGL provides a layer of abstraction between graphics hardware and an application program. It is visible to the programmer as a set of routines consisting of about 120 distinct commands. Together these routines make up the OpenGL application programming interface (API). The routines allow graphics primitives (points, lines, polygons, bitmaps, and images) to be rendered to a frame buffer. Using the available primitives and the operations that control their rendering, high-quality color graphics images of 3D objects can be rendered.

The designers of OpenGL present the graphics system as a state machine [7] that controls a well-defined set of drawing operations. The routines that OpenGL supplies provide a means for the programmer to manipulate OpenGL's state machine to generate the desired graphics output. Figure 1 shows a simplified view of OpenGL's abstract state machine. Specifying OpenGL as a state machine allows consistent, precise specification and eliminates ambiguity about what a given operation does and does not do.

The model used for interpretation of OpenGL com-

*Mark graduated with B.A. in Computer Science from Rice University and is a Member of the Technical Staff at Silicon Graphics. He can be reached by electronic mail addressed to `mjk@sgi.com`

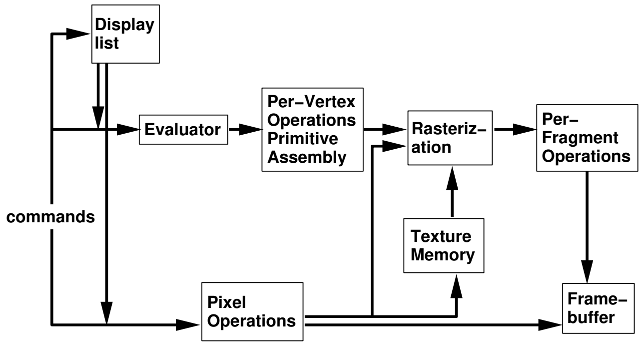

Figure 1: High-level, abstract OpenGL machine.

mands is *client-server*. This is an abstract model and does *not* demand OpenGL be implemented as distinct client and server processes. A client-server approach means the boundary between a program and the OpenGL implementation is well-defined to clearly specify how data is passed between the program and OpenGL. This allows OpenGL to operate over a *wire protocol* much as the X protocol operates but does not mandate such an approach.

The OpenGL specification is *window system independent* meaning it provides rendering functionality but does not specify how to manipulate windows or receive events from the window system. This allows the OpenGL interface to be implemented for distinct window systems. For example, OpenGL has been implemented for both the X Window System and Windows NT.

The specification which describes how OpenGL intergrates with the X Window System is known as GLX. It is an extension to the core X protocol for communicating OpenGL commands to the X server. It also supports window system specific operations such as creating rendering contexts, binding those contexts to windows, and other window system specific operations.

GLX does not demand OpenGL commands be executed by the X server. The GLX specification explicitly allows OpenGL to render directly to the hardware if supported by the implementation. This is possible when the program is running on the same machine as the graphics hardware. This potentially allows extremely high performance rendering because OpenGL commands do not need to be sent through the X server to get to the graphics hardware.

Graphics systems are often classified as one of two types: procedural or descriptive. Procedural means the programmer is determining what to draw by issuing a specific sequence of commands. Descriptive means the programmer sets up a model of the scene to be rendered and leaves how to draw the scene up to the graphics system. OpenGL is procedural. In a descriptive system, the programmer gives up control of exactly how the scene is to be rendered. Being procedural allows the programmer a high degree of control to achieve the best performance. It is expected that descriptive graphics systems will be implemented using OpenGL as a low level interface. SGI's Inventor toolkit [8] is one example of such a descriptive graphics system.

An overriding goal of OpenGL is to allow the construction of portable and interoperable 3D graphics programs. For this reason, OpenGL's rendering functionality must be implemented in its entirety. This means all the complex 3D rendering functionality described later in the article can be used with any OpenGL implementation. Previous graphics standards often allowed subsetting; too often the result was programs that could not be expected to work on distinct implementations.

1.2 History of OpenGL

A brief history of OpenGL explains how OpenGL came to be and what inspired its development. OpenGL is the successor to a graphics library known as IRIS GL (GL stands for graphics library) developed by SGI as a hardware independent graphics interface for use across a full line of graphics workstations. IRIS GL [4] is used by more than 1,500 3D graphics applications. IRIS GL was developed over the last decade and has been implemented on numerous graphics devices of varying sophistication.

OpenGL is not backward-compatible with IRIS GL. OpenGL has removed dated IRIS GL functionality or replaced it with more general functionality. The routines and symbols comprising the OpenGL API have been named to avoid name space conflicts (all names start with either `gl` or `GL_`). The window system dependent portions of IRIS GL are not part of OpenGL. What has been preserved is the spirit of the API. OpenGL retains IRIS GL's ability to render 3D objects quickly and efficiently.

OpenGL has been proposed as a graphics standard to bring 3D graphics programming into the mainstream of applications programming. For this reason, the OpenGL ARB was formed. The ARB licenses OpenGL and directs further development. Currently, over 20 companies have licensed OpenGL and intend to release or have already released commercial implementations. Numerous universities have also licensed OpenGL.

2 OpenGL's Functionality

OpenGL is not a high-level 3D graphics interface. When you build a graphics program using OpenGL, you start with a few simple primitives. The sophistication comes from combining the primitives and using them in various modes. Figure 2 shows the available geometric primitives. Notice the ordering of the vertices, in particular for primitives such as the `GL_TRIANGLE_STRIP` and the `GL_TRIANGLE_FAN`.

To begin a primitive, the `glBegin` routine passes in the primitive type as an argument. Then a list of vertex coordinates are given. OpenGL has a family of routines to specify vertex coordinates. All the routines begin with

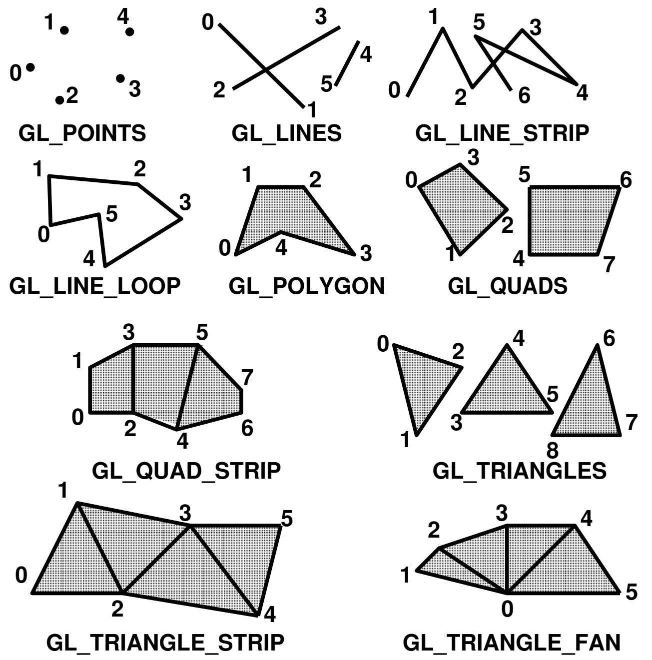

Figure 2: OpenGL Geometric Primitives.

the name `glVertex`. The suffix to a specific `glVertex` routine tells the type and number of coordinates for the vertex. For example, `glVertex3f` indicates a three coordinate vertex consisting of floating point values is to be generated.

An OpenGL primitive is completed by calling `glEnd`. Along with the coordinates of each vertex, per-vertex information about color, material, normals, edge drawing, and texturing can be specified between a `glBegin` and `glEnd`. Figure 3 shows an example of how a polygon might be generated. Notice how `glColor3f` is used to change the current color. Each vertex is drawn according to the current color.

```
glShadeModel(GL_SMOOTH);
glBegin(GL_POLYGON); /* pentagon */
  glColor3f(0.0, 1.0, 0.0); /* green */
  glVertex3f(0.0, 1.0, 0.0);
  glVertex3f(0.7, 1.0, 0.0);
  glColor3f(0.0, 0.0, 1.0); /* blue */
  glVertex3f(1.4, 0.6, 0.0);
  glVertex3f(1.4, 0.4, 0.0);
  glVertex3f(0.0, 0.0, 0.0);
glEnd();
```

Figure 3: Example of generating a 3D polygon with smooth shading between vertices.

2.1 Observations About Primitives

OpenGL tends to be function call intensive. There is not a complex `RenderPolygonWithGratuitousArguments` command. Instead primitives are constructed by calling multiple OpenGL routines. Calling multiple routines gives the program flexibility and control over the primitives generated.

OpenGL is flexible about what format information is passed to it. For example, the `glVertex3i` accepts integers while `glVertex3f` and `glVertex3d` take single and double precision floating point respectively. It is very advantageous for OpenGL to have several basically identical routines which accept different data types. It allows the programmer the flexibility to decide how to store the data. A programmer whose data is in integer format does not want to convert it to floating point to pass it to the graphics system. And another programmer does not want to convert floating point data into integers. Conversions between data types can be expensive. High performance graphics hardware can be designed to accept multiple data formats and totally off load the task of format conversion from the host processor.

You can start to see why it makes sense to consider OpenGL as a state machine. Commands such as `glColor3f` change the state of the current color. Subsequent vertices use the current color. `glBegin` puts OpenGL into a state to start drawing the specified primitive. The multiple `glVertex` routines load up one at a time the vertices for a given primitive. Nearly all of OpenGL's state that can be set by the programmer can also be queried by the programmer. The `glGetFloatv(GL_CURRENT_COLOR, &float_array)` call, for example, will retrieve the setting of the current color.

2.2 Two Color Models

OpenGL has two different color modes: *RGBA* and *color index*. The `glColor3f` call has already been demonstrated but not explained. This call assumes OpenGL's RGBA color mode. The routine takes three floating point parameters between 0.0 and 1.0 which specify the degree of red, green, and blue for the current color. For X users, RGBA roughly corresponds to the `TrueColor` visual type while color index corresponds to `PsuedoColor`. The color mode is fixed for a given window the same way X windows are created with a single, fixed visual.

You should be able to guess that the RGB in RGBA stands for red, green, and blue. The A may be unfamiliar. It stands for *alpha*. The alpha value is used when two colors are to be averaged together for blending operations. Alpha represents the opacity of the color. 1.0 is totally opaque while 0.0 is totally transparent. For example, one could use alpha to render a scene with green glass. The frame buffer can support an alpha component which allows alpha values to be stored. Each pixel in the frame buffer would have an associated alpha value. The alpha value is not visible on the display. It is just used to determine how a pixel to be drawn is blended with the current pixel value in the frame buffer. The `glAlphaFunc` and `glBlendFunc` routines control precisely how alpha buffering operates. The `glColor4f` command is a variation on `glColor3f` which takes a fourth parameter specifying alpha (`glColor3f` implicitly sets alpha to 1.0).

RGBA supports a tri-linear palette for the full range of colors, making it very useful for rendering realistic scenes. OpenGL supports lighting, fog, and smooth shading most effectively in RGBA mode. Since a lot of hardware has limited color resolution, an application can request OpenGL use dithering for better color resolution (at the expense of spatial resolution).

Many modes in OpenGL, such as dithering, are enabled and disabled using the `glEnable` and `glDisable` commands. For example, dithering is enabled by calling `glEnable(GL_DITHER)`. Then drawing would be done with dithering enabled. You can think of `glEnable` and `glDisable` as ways to affect the operation of the OpenGL state machine.

The color index model assumes a readable and writable linear colormap. Usually window systems specify how colors are allocated and arranged so OpenGL does not have any specific routines to allocate colors. For example in X, an Xlib color allocation routine such as `XAllocColor` would be used. The `glIndex` family of routines is used to set the current color index. The advantage of color index is that the color of a given pixel value can be changed. There is a level of indirection between the pixel values in the frame buffer and the colors on the screen.

2.3 Ancillary Buffers

The drawing surface for OpenGL is generically referred to as the *frame buffer*. In actuality, the frame buffer might be a window created by your computer's window system or an in-memory data structure (like an X pixmap). OpenGL's frame buffer can logically be considered a set of buffers. A buffer is logically just a two-dimensional array of values. The most important buffer is the image buffer which contains the actual color information and possibly the alpha component but there are also other types of buffers. A window system might support multiple frame buffer configurations, each supporting different types of buffers. Multiple windows of different configurations can be displayed at one time though a single window has a fixed frame buffer configuration. In X, visuals are overloaded to also describe supported OpenGL frame buffer configurations.

The non-image buffers are often referred to as *ancillary* or helper buffers. While they do not contain the image itself, they can be essential in properly generating the image.

2.3.1 The Depth Buffer

For 3D graphics, the *depth buffer* (also commonly referred to as a Z buffer) is nearly essential. While the screen only has two dimensions, 3D graphics seeks to simulate a third. When 3D primitives are rendered, they are rasterized into a collection of *fragments*. Each fragment corresponds to a single pixel and includes color, depth, and sometimes texture-coordinate values. The X and Y values for a fragment determine where on the screen the pixel should appear. A fragment's Z value or depth is used to determine how "near" the fragment is. When the depth buffer is enabled, the fragment is drawn only if its Z value is "nearer" than the current Z value for the corresponding pixel in the depth buffer. When the fragment is drawn into the frame buffer, its Z value replaces the previous value in the depth buffer. Normally, when the scene starts to be rendered, the entire depth buffer is cleared to the "farthest" value. As a 3D scene is rendered, the depth buffer automatically sorts the fragments being drawn so only the nearest fragment at each pixel location gets drawn. Things logically behind other things are automatically eliminated from the scene. This is the normal use for a depth buffer, although other uses are possible.

2.3.2 The Stencil Buffer

Another buffer supported by OpenGL is the stencil buffer. Like the depth buffer, the stencil buffer can be used to eliminate certain pixels from being drawn. The stencil buffer acts in much the same way as a cardboard stencil used with a can of spray paint. You can "draw" values into the stencil buffer using the normal OpenGL rendering primitives. Then a stencil test can be defined and stenciling enabled.

One possible use of the stencil buffer is in a flight simulator. Imagine that the view outside the plane is to fit into an irregularly shaped windshield. The rendering of the view outside the plane should not interfere with the rendering of the instruments "inside" the cockpit. If the windshield area is drawn in the stencil buffer, then a stencil test can be set up to make sure the windshield view is only drawn where the windshield stencil has been drawn. There are many other uses for stencil buffers.

2.3.3 The Accumulation Buffer

Yet another buffer supported by OpenGL is the accumulation buffer [2] which can be used for antialiasing, motion blur, simulating photographic depth of field, and rendering soft shadows from multiple light sources. You do not render directly into the accumulation buffer. Instead, you render a series of images, accumulating each into the accumulation buffer, combining the images. Then the accumulated image can be dumped back into the image buffer for display. The effect is much the same as the one a photographer gets from multiply exposing a piece of film.

Motion blur is one use. Imagine drawing a scene several times with each frame corresponding to a slightly different point in time. By accumulating the frames (with decayed intensity for earlier frames), you can achieve an effect similar to motion blur since still objects are sharp but moving objects are blurred by their accumulation in slightly differing locations.

2.3.4 Double Buffering

Double buffering means having two sets of image buffers, one *front* visible buffer and another *back* non-visible buffer. Unlike simple 2D, 3D images may take substantially more time to generate. And depth, alpha, and accumulation buffers all mean that the image being drawn at any moment might be quite different from the final image. It would be quite distracting for the viewer to see each scene while it was "under construction" and would destroy the illusion of a smoothly animated scene. Double buffering allows for one image to be rendered while another is being displayed.

OpenGL supports this notion. The `glDrawBuffer` routine can be used to determine to what buffer primitives should be drawn. A window system specific routine is available to make the back buffer visible.

Double buffering is often achieved by rendering the non-visible image buffer into memory and then quickly copying the buffers contents to screen memory. A better alternative is to build hardware that actually supports two sets of image buffers. Then the cost of a buffer swap can be extremely low since no data has to be copied. Instead, the video controller can just change to scanning image pixels out of the other buffer.

2.3.5 Stereo

Stereo is similar to double buffering in that more than one image buffer is supported. Instead of front and back, left and right are provided (though generally stereo and double buffering are combined, requiring four image buffers). Special stereo video hardware alternates between scanning out the left and right buffers every screen refresh. Goggles synchronized with the vertical refresh of the screen alternately open and close LCD shutters so the left eye sees the left frame and the right eye sees the right frame. By carefully drawing the scene twice with slightly different perspective into the left and right buffers, the viewer experiences an optical illusion of 3D.

While double buffering is common on graphics workstations, stereo requires special hardware and tends to be rather expensive so many OpenGL implementations may not support stereo.

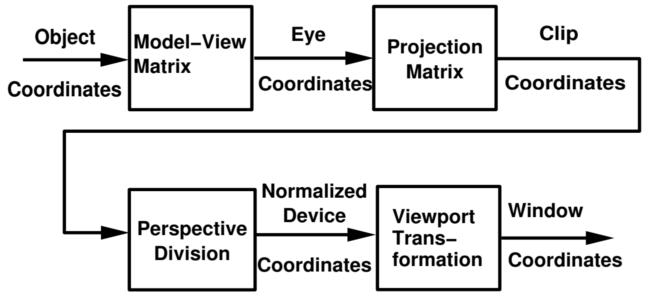

Figure 4: Stages of vertex transformation.

2.4 Viewing

One of the most difficult initial hurdles in learning 3D graphics programming is how to properly set up a view. It is very easy to get a black screen because the viewing for the scene is not properly initialized.

3D computer graphics uses matrix transformations to properly orient, view, clip, and map the model to the screen. OpenGL's various stages in mapping vertices in object coordinates into pixels in window coordinates are pictured in Figure 4.

An OpenGL programmer is responsible for loading the *modelview* and *projection* matrices. The modelview matrix determines how the vertices of OpenGL primitives are transformed to eye coordinates. The projection matrix transforms vertices in eye coordinates to clip coordinates.

A number of OpenGL routines deal with manipulating these matrices. The `glMatrixMode` routine is called with an argument of `GL_MODELVIEW` or `GL_PROJECTION` to determine what is the current modifiable matrix. Then `glLoadIdentity` may be called to set the currently modifiable matrix to the identity matrix. Then routines such as `glRotatef`, `glTranslatef`, and `glScalef` may be called to manipulate the currently modifiable matrix. `glLoadMatrixf` loads a specific matrix and `glMultMatrixf` multiplies the current matrix by some specified matrix and store the result as the current matrix. Understanding exactly how these different commands should be properly used is beyond the scope of this article.

The final step in establishing a view of your model is the *viewport* transformation. It determines how the scene gets mapped onto the computer screen. The `glViewport` routine specifies the rectangle in the window of into which the final image is to be mapped. By default, the entire window is used. `glViewport` is commonly invoked when an OpenGL window is resized.

2.5 Other Features

There are a large number of OpenGL features worth mentioning but their full introduction is beyond the scope of this article.

Just specifying 3D primitives and determining how to map them to the screen is not enough to achieve realistic images. OpenGL also supports a number of lighting models that simulate the effects of lighting on primitives. Light sources can be defined and material properties can be specified to achieve realistic lighting effects.

So far polygons have been described as basically shaded or flat surfaces. But OpenGL allows polygons to be rendered which have a 1D or 2D texture mapped onto the polygon. For example, the surface of a desk could be textured with a wood grain image for greater realism. Texture mapping can greatly enhance the visual impact of a scene without increasing the geometric complexity.

Polygons are the basic primitive for much 3D rendering but OpenGL also supports bitmaps and images. And OpenGL provides evaluator commands for the efficient rendering of curves and surfaces.

Because 3D rendering eventually appears on a screen with limited resolution, OpenGL provides various techniques to eliminate "jaggies" resulting from aliasing problems. OpenGL provides antialiasing support for points, lines, and polygons. Techniques using the alpha, stencil, or accumulation buffers can also be used to minimize aliasing problems.

Computer images often appear unrealistically sharp and well-defined. OpenGL supports "fog" to provide an effect that simulates atmospheric effects. Haze, mist, smoke, and pollution can all be simulated. When fog is enabled, objects farther away begin to fade into the specified fog color.

Users of 3D want to do more than just see 3D images; they want to interact with them. OpenGL supports a *selection* mechanism that allows the user to pick an object or objects drawn to a certain region of the screen. And *feedback* can be used to obtain the results of rendering calculations.

Often a sequence of OpenGL commands are rendered repeatedly. OpenGL supports display lists which allow commands to be compiled for later execution. Display lists can even call other display lists allowing hierarchies of display lists. For networked 3D applications, display lists can greatly minimize the protocol bandwidth needed and increase performance. The `glNewList` and `glEndList` are used to create a display list. A created display list can be executed using the `glCallList` routine.

One thing to keep in mind about OpenGL is that the features described above are not isolated functionality. Each feature can be combined with others for advanced effects. For example, lighting, fog, display lists, texture mapping, and double buffering can all be used simultaneously.

2.6 The GLU Library

The core OpenGL API focuses on rendering functionality but there are a number of tasks common to many 3D programs that are not strictly related to rendering. For this reason, the OpenGL standard also provides the OpenGL Utility Library (GLU). The GLU routines (all prefixed with `glu`) fall into one of the following areas:

- Manipulating images for use in texturing.
- Transforming Coordinates.
- Polygon tessellation.
- Rendering spheres, cylinders, and disks.
- Non-Uniform Rational B-Spline (NURBS) curves and surfaces.
- Describing errors.

The GLU is a separate but standard library that any OpenGL application can use.

3 OpenGL's X Support

GLX is an official part of the OpenGL standard for supporting the X Window System. It provides additional routines (prefixed by `glX`) for interfacing OpenGL with X. It also defines a wire protocol for supporting OpenGL as an X server extension. The GLX wire protocol allows workstations from different vendors to interoperate using 3D graphics the same way the X protocol provides 2D graphics interoperability. Some of the issues about integrating X and OpenGL are discussed by Karlton [3].

GLX allows rendering into X windows and pixmaps. An X server can support different visuals to describe the different types of windows supported by the server. For the core X protocol, a visual specifies one (or more) depths for the frame buffer and how pixel values are mapped to colors on the screen. X treats a drawable as basically a 2D array of pixels, but OpenGL has a much more sophisticated view of a drawable's frame buffer capabilities. GLX overloads the core X notion of a visual by associating additional information about OpenGL's frame buffer capabilities. In addition to an image buffer, OpenGL supports various types of ancillary buffers. For example, a window might also have a stencil buffer and a depth buffer. Modes such as stereo and double buffering are also supported. Multiple different frame buffer configurations can be supported by a single X server by exporting multiple visuals.

All OpenGL implementations for the X Window System must support at least one RGBA visual and at least one color index visual. Both visuals must support a stencil buffer of at least 1 bit and a depth buffer of at least 12 bits The required RGBA visual must have an accumulation buffer. The alpha component of the image buffer is not required for the RGBA visual (but input alpha is still used in all rendering calculations). Many implementations will supply many more than two visuals.

The GLX API supplies two routines, `glXGetConfig` and `glXChooseVisual`, to help programmers select an appropriate visual. Once the appropriate visual is selected, call `XCreateWindow` with the selected visual to create the window.

GLX supports off-screen rendering to pixmaps. First create a standard X pixmap of the desired depth using `XCreatePixmap`. Then call `glXCreateGLXPixmap` with the desired OpenGL visual. A new drawable of type `GLXPixmap` is returned which can be used for drawing OpenGL into the pixmap.

To render using OpenGL, an OpenGL rendering context must be created. The `glXCreateContext` routine creates such a context. An option to `glXCreateContext` allows the programmer to specify that direct rendering to the hardware should be done if supported by the implementation.

Before rendering, a rendering context must be bound to the desired drawable using `glXMakeCurrent`. OpenGL rendering commands implicitly use the current bound rendering context and one drawable. Just as a program can create multiple windows, a program can create multiple OpenGL rendering contexts. But a thread can only be bound to one rendering context and drawable at a time. Once bound, OpenGL rendering can begin. `glXMakeCurrent` can be called again to bind to a different window and/or rendering context.

The GLX stream of commands is considered distinct from the stream of X requests. Sometimes you may want to mix OpenGL and X rendering into the same window. If so synchronization can be achieved using the `glXWaitGL` and `glXWaitX` routines.

To swap the buffers of a double buffered window, `glXSwapBuffers` can be called. X fonts can be converted into per-glyph OpenGL display lists using the `glXUseXFont` routine.

4 A Simple Example Using X

Appendix A contains the C source code for a simple OpenGL program. This example demonstrates what is involved when programming OpenGL with Xlib. The program creates a window and draws a 3D cube (missing two faces) and allows the user to rotate the cube around the X, Y, and Z axes using the mouse buttons.

Besides demonstrating how to properly establish an X window for OpenGL rendering, the example demonstrates the use of double buffering, display lists, and establishing the proper viewing parameters.

4.1 Initialization

The following describes the steps involved in setting up a window to render OpenGL into it. The numbers listed correspond to numbers in the comments of the OpenGL program in Appendix A.

1. As in all X programs, `XOpenDisplay` should be called to open a connection to the X server.
2. Make sure the OpenGL GLX extension is supported by the X server.
3. Before creating the window, the program needs to select an appropriate visual. The GLX routine `glXChooseVisual` makes it easy to find the right visual. In the example, an RGBA (and TrueColor) visual with a depth buffer is desired and if possible, it should support double buffering.
4. Create an OpenGL rendering context by calling `glXCreateContext`.
5. Create a window with the selected visual. Most X programs always use the default visual but OpenGL programmers will need to be comfortable with using visuals other than the default. `XCreateWindow` is called.
6. Bind the rendering context to the window using `glXMakeCurrent`. Subsequent OpenGL rendering commands will use the current window and rendering context.
7. To display the window, `XMapWindow` should be called.
8. Set the desired OpenGL state. In this example, depth buffering is enabled, the clear color is set to black, and the 3D viewing volume is specified.
9. Begin dispatching X events.

Button presses change the angle of rotation for the object to be viewed and cause a redraw. Expose events also cause a redraw (without changing the rotation). Window resizes call `glViewport` to ensure the OpenGL viewport corresponds to the maximum dimensions of the window.

4.2 Scene Update

The `redraw` routine does all the OpenGL rendering. The code is slightly complicated by constructing a display list to draw the cube. The first time `redraw` is called, `glNewList` and `glEndList` are used to construct a display list for the object to be rendered. Subsequent redraws call the display list instead of rendering the object each time.

Creating a display list potentially allows improved performance since the commands can be compiled for faster execution. In the case of OpenGL across a network, display lists save having to send all the commands to render the scene whenever the window is redrawn.

The commands to render the object consist of four 3D rectangles of different colors. Notice the rectangles

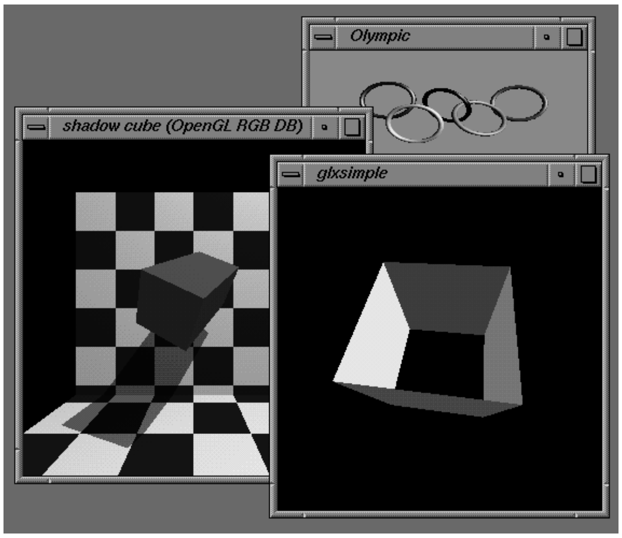

Figure 5: Screen snapshot of `glxsimple` with two other simple 3D OpenGL programs.

are generated by first calling `glBegin(GL_QUADS)` and ended with `glEnd`. Each rectangle is specified by four `glVertex3f` calls that specify the four vertexes of each rectangle. The `glColor3f` invocations tell what color each rectangle should be rendered. Figure 5 shows how the program looks.

If the window is double buffered, `glXSwapBuffers` is called on the window. By default rendering to double buffered windows takes place in the non-visible back buffer. Swapping buffers will quickly swap the front and back buffers avoiding any visual artifacts (the contents of the back buffer should considered undefined after a swap). In effect, the rendering of each frame can be done "behind the scenes."

`glFlush` is called to ensure that the OpenGL rendering commands are actually sent to the graphics hardware. A flush is implicitly done by `glXSwapBuffers` so the `glFlush` is only needed explicitly in the single buffered case.

5 Comparing OpenGL to PEX

OpenGL is not the only means for extending the X Window System to support 3D. PEX [9] is an extension developed by the X Consortium to add 3D capabilities to X. The currently available version is 5.1. A future release known as PEX 6.0 is intended to address many of PEX 5.1's problems but its specification is not yet finalized. An in-depth analysis of PEX 5.1 and OpenGL 1.0 is presented by Akin [1]. Here we discuss some of the most prominent distinctions.

5.1 Subsets and Baselines

One thing that makes PEX difficult to compare to OpenGL is that PEX allows much of its functionality to be optionally implemented. PEX classifies its functionality into one or more of three subsets: the immediate mode subset, the structure subset, or the PHIGS workstation subset. (PHIGS is a 3D graphics standard and stands for Programmer's Hierarchical Interactive Graphics System.) The PEX specification *explicitly* allows implementations to support one, two, or all three subsets. The result is that an application cannot depend on any given PEX server to supply the subset functionality the application might depend on. This problem is commonly referred to as "subsetting."

OpenGL mandates that that all its rendering functionality be supported. Even advanced features such as depth buffering, fog, lighting, anti-aliasing, and texturing must be supported in *all* implementations.

But still all OpenGL implementations are not totally identical. Rendering functionality is not a complete picture of OpenGL's capability. Rendering performance will depend on the implementation. And frame buffer capabilities will vary between implementations. Different depths of ancillary buffers will be supported; stereo and double buffering hardware may or may not actually be present; a frame buffer may or may not support the alpha component. But despite the possibility for variation, OpenGL for the X Window System does *mandate* that two visuals (one RGBA, the other color index) will be present with frame buffer capabilities sufficient for most common 3D applications. Stencil and depth buffers must be supported for the two required visuals. And an accumulation buffer must be supported for the RGBA visual. These required visuals guarantee all OpenGL implementations have a standard baseline of both rendering and frame buffer functionality which applications can rely on being present.

5.2 Programming Interfaces

There is an essential difference between PEX and OpenGL in how the two graphics systems are specified. OpenGL is fundamentally specified as an application programming interface. Like the X Window System, the fundamental specification for PEX is a wire protocol.

In PEX the choice of programming interface is left to the programmer. In X11R5 a PHIGS style API was supplied but this API for PEX has not gained much acceptance. Currently the PEX community is standardizing the PEXlib API which more readily exposes the wire protocol. But PEX implementation dependencies are also exposed, leaving the programmer to work around functionality missing due to subsetting in PEX implementations.

With OpenGL there is a single API which promises to be standard even across differing window systems (such as X and NT) and the full functionality of the API is available in all OpenGL implementations. The GLX specification does provide a wire protocol for network-transparent operation but the wire protocol is not the fundamental specification of OpenGL.

5.3 Rendering Functionality

PEX and OpenGL both support basic 3D rendering functionality. Both allow 3D and 2D lines and polygons to be rendered using standard modeling and viewing methods. PEX (depending on the implementation) and OpenGL also support picking, lighting, depth cueing, and hidden line and surface removal.

There are a number of sophisticated rendering features supported by OpenGL that PEX completely lacks. Alpha blending, texture and environment mapping, antialiasing (though some PEX implementations supply it as a nonstandard extension), accumulation buffer methods, and stencil buffering are all missing from PEX.

PEX does support features not available in OpenGL. PEX has extensive text support for stroke fonts which are fully transformable in 3D. B-Spline surfaces and curves are supported directly by PEX while OpenGL supports NURBS functionality via routines which are part of the GLU library. PEX can support cell arrays but the functionality is seldom implemented. Markers and quadrilateral meshes are supported by PEX as a rendering primitive; neither are supported as primitives by OpenGL. PEX supports self-intersecting contours and polygon lists with shared geometry, while OpenGL does not.

Double buffering and stereo support are built into OpenGL (though not all implementations will support double buffered or stereo visuals) while PEX relies on proprietary support or not yet nonstandardized X extensions for double buffering and stereo.

5.4 Display Lists

PEX and OpenGL both provide a means to store commands for later execution. In PEX (for implementations that support the structure or PHIGS workstation subsets), editable *structures* can be created and edited. A structure contains graphics primitives such as a polygon. Structures may also contain calls to execute other structures allowing them to be arranged in a hierarchical fashion. PHIGS supports structures so PEX does so too. Entire 3D models can be constructed out of a hierarchy of structures so that a redraw requires only retraversing the structure hierarchy.

OpenGL does not support structures in the same way PEX does. Instead *display lists* can be constructed which contain sequences of OpenGL commands. Like structures, a display list can contain a command to execute another display list, effectively allowing display lists to be combined into arbitrary networks. Unlike structures, OpenGL display lists are *not* editable. Once one is created, it is sealed and cannot be changed (except by destroy and recreating it). This write-only nature allows optimizations to be performed on display lists unavailable to structures. The commands in the display list can be optimized for faster execution.

Even though display lists cannot be edited, this should not be considered a disadvantage. The same effect as editing can be achieved by rewriting display lists called by other display lists.

Display lists and structures both minimize the amount of transfer overhead when running PEX or OpenGL over a network since the commands in a structure or display list can be executed repeatedly by only calling the display list by name. The commands themselves need to be transferred across the wire only once.

5.5 Portability

While PEX was designed to be vendor-independent and portable, the subsetting allowed by the PEX standard allows implementations of greatly varying functionality to claim to be "standard" PEX implementations. The fact that PEX explicitly allows multiple subsets perhaps indicates the PEX standard may be too large to implement fully and completely in a timely fashion. Anyone who has been disappointed by the functionality of the X11R5 sample implementation understands the problem.

OpenGL does not allow any subsetting of rendering functionality and therefore can expect much greater application portability. The need for interoperability testing for OpenGL is greatly reduced because OpenGL demands more consistent implementations.

Neither OpenGL nor PEX is *pixel exact*. This means neither specification is completely explicit about what pixels must be modified by each rendering operation (the core X protocol is largely pixel exact). Pixel exactness is not a totally desirable feature for 3D since much 3D graphics is done with floating point where numerical errors make exactness nearly impossible. But the OpenGL specification is much more rigorous than PEX about what is considered conformant behavior. Not only does this make conformance test design easier, but OpenGL programmers can have high confidence their scene will be rendered accurately on all compliant OpenGL implementations.

The OpenGL release kit includes a suite of conformance tests to verify rendering accuracy. No comprehensive test suites are yet available to validate PEX implementations.

5.6 Window System Dependency

PEX is very tightly coupled to the X Window System. Not only was it designed in the context of X but its semantics depend on X notions of drawables, events, and execution requirements.

But X is not the only significant window system on the market. For this reason, OpenGL was designed to be window system independent. This means its API can also be used with Windows NT and future window systems. Application developers wishing to develop 3D applications for both X and Windows machines will appreciate having a consistent model for 3D across the two window systems.

6 Finding Out More

The best place to find more information about graphics programming using OpenGL is the OpenGL Technical Library published by Addison-Wesley. Currently available is the OpenGL Reference Manual [6] and the OpenGL Programming Guide [5]. The first volume contains complete descriptions of all the OpenGL routines including the GLU and GLX routines. The second volume is an excellent introduction to OpenGL including all its advanced rendering features.

Those with Internet access can obtain OpenGL documentation and sample program source code by using anonymous `ftp` to `sgi.com`. PostScript documentation for all the routines that are part of the OpenGL, GLU, and GLX APIs may be obtained. Example code from the OpenGL Programming Guide (including the aux library) is also available.

Of course the best way to learn OpenGL is to program with it. Systems supporting OpenGL are currently shipping from a number of workstation hardware and software vendors. Check with your vendor for availability.

A glxsimple.c

```
/* compile: cc -o glxsimple glxsimple.c -lGL -lX11 */
#include <stdio.h>
#include <stdlib.h>
#include <GL/glx.h>             /* this includes the necessary X headers */
#include <GL/gl.h>

static int snglBuf[] = {GLX_RGBA, GLX_DEPTH_SIZE, 16, None};
static int dblBuf[] = {GLX_RGBA, GLX_DEPTH_SIZE, 16, GLX_DOUBLEBUFFER, None};

Display       *dpy;
Window         win;
GLfloat        xAngle = 42.0, yAngle = 82.0, zAngle = 112.0;
GLboolean      doubleBuffer = GL_TRUE;

void
fatalError(char *message)
{
    fprintf(stderr, "glxsimple: %s\n", message);
    exit(1);
}

void
redraw(void)
{
    static GLboolean   displayListInited = GL_FALSE;

    if (displayListInited) {
        /* if display list already exists, just execute it */
        glCallList(1);
    } else {
        /* otherwise compile and execute to create the display list */
        glNewList(1, GL_COMPILE_AND_EXECUTE);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        /* front face */
        glBegin(GL_QUADS);
        glColor3f(0.0, 0.7, 0.1);       /* green */
        glVertex3f(-1.0, 1.0, 1.0);
        glVertex3f(1.0, 1.0, 1.0);
        glVertex3f(1.0, -1.0, 1.0);
        glVertex3f(-1.0, -1.0, 1.0);
        /* back face */
        glColor3f(0.9, 1.0, 0.0);       /* yellow */
        glVertex3f(-1.0, 1.0, -1.0);
        glVertex3f(1.0, 1.0, -1.0);
        glVertex3f(1.0, -1.0, -1.0);
        glVertex3f(-1.0, -1.0, -1.0);
        /* top side face */
        glColor3f(0.2, 0.2, 1.0);       /* blue */
        glVertex3f(-1.0, 1.0, 1.0);
        glVertex3f(1.0, 1.0, 1.0);
        glVertex3f(1.0, 1.0, -1.0);
        glVertex3f(-1.0, 1.0, -1.0);
        /* bottom side face */
        glColor3f(0.7, 0.0, 0.1);       /* red */
        glVertex3f(-1.0, -1.0, 1.0);
        glVertex3f(1.0, -1.0, 1.0);
        glVertex3f(1.0, -1.0, -1.0);
        glVertex3f(-1.0, -1.0, -1.0);
```

```
        glEnd();
        glEndList();
        displayListInited = GL_TRUE;
    }
    if(doubleBuffer) glXSwapBuffers(dpy, win); /* buffer swap does implicit glFlush */
       else glFlush(); /* explicit flush for single buffered case */
}

void
main(int argc, char **argv)
{
    XVisualInfo    *vi;
    Colormap        cmap;
    XSetWindowAttributes swa;
    GLXContext      cx;
    XEvent          event;
    GLboolean       needRedraw = GL_FALSE, recalcModelView = GL_TRUE;
    int             dummy;

    /*** (1) open a connection to the X server ***/

    dpy = XOpenDisplay(NULL);
    if (dpy == NULL) fatalError("could not open display");

    /*** (2) make sure OpenGL's GLX extension supported ***/

    if(!glXQueryExtension(dpy, &dummy, &dummy)) fatalError("X server has no OpenGL GLX extension");

    /*** (3) find an appropriate visual ***/

    /* find an OpenGL-capable RGB visual with depth buffer */
    vi = glXChooseVisual(dpy, DefaultScreen(dpy), dblBuf);
    if (vi == NULL) {
       vi = glXChooseVisual(dpy, DefaultScreen(dpy), snglBuf);
       if (vi == NULL) fatalError("no RGB visual with depth buffer");
       doubleBuffer = GL_FALSE;
    }
    if(vi->class != TrueColor) fatalError("TrueColor visual required for this program");

    /*** (4) create an OpenGL rendering context  ***/

    /* create an OpenGL rendering context */
    cx = glXCreateContext(dpy, vi, /* no sharing of display lists */ None,
                          /* direct rendering if possible */ GL_TRUE);
    if (cx == NULL) fatalError("could not create rendering context");

    /*** (5) create an X window with the selected visual ***/

    /* create an X colormap since probably not using default visual */
    cmap = XCreateColormap(dpy, RootWindow(dpy, vi->screen), vi->visual, AllocNone);
    swa.colormap = cmap;
    swa.border_pixel = 0;
    swa.event_mask = ExposureMask | ButtonPressMask | StructureNotifyMask;
    win = XCreateWindow(dpy, RootWindow(dpy, vi->screen), 0, 0, 300, 300, 0, vi->depth,
                        InputOutput, vi->visual, CWBorderPixel | CWColormap | CWEventMask, &swa);
    XSetStandardProperties(dpy, win, "glxsimple", "glxsimple", None, argv, argc, NULL);

    /*** (6) bind the rendering context to the window ***/

    glXMakeCurrent(dpy, win, cx);
```

```
    /*** (7) request the X window to be displayed on the screen ***/

    XMapWindow(dpy, win);

    /*** (8) configure the OpenGL context for rendering ***/

    glEnable(GL_DEPTH_TEST); /* enable depth buffering */
    glDepthFunc(GL_LESS);    /* pedantic, GL_LESS is the default */
    glClearDepth(1.0);       /* pedantic, 1.0 is the default */
    /* frame buffer clears should be to black */
    glClearColor(0.0, 0.0, 0.0, 0.0);
    /* set up projection transform */
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum(-1.0, 1.0, -1.0, 1.0, 1.0, 10.0);
    /* establish initial viewport */
    glViewport(0, 0, 300, 300); /* pedantic, full window size is default viewport */

    /*** (9) dispatch X events ***/

    while (1) {
        do {
            XNextEvent(dpy, &event);
            switch (event.type) {
            case ButtonPress:
                recalcModelView = GL_TRUE;
                switch (event.xbutton.button) {
                case 1: xAngle += 10.0; break;
                case 2: yAngle += 10.0; break;
                case 3: zAngle += 10.0; break;
                }
                break;
            case ConfigureNotify:
                glViewport(0, 0, event.xconfigure.width, event.xconfigure.height);
                /* fall through... */
            case Expose:
                needRedraw = GL_TRUE;
                break;
            }
        } while(XPending(dpy)); /* loop to compress events */
        if (recalcModelView) {
            glMatrixMode(GL_MODELVIEW);
            /* reset modelview matrix to the identity matrix */
            glLoadIdentity();
            /* move the camera back three units */
            glTranslatef(0.0, 0.0, -3.0);
            /* rotate by X, Y, and Z angles */
            glRotatef(xAngle, 0.1, 0.0, 0.0);
            glRotatef(yAngle, 0.0, 0.1, 0.0);
            glRotatef(zAngle, 0.0, 0.0, 1.0);
            recalcModelView = GL_FALSE;
            needRedraw = GL_TRUE;
        }
        if (needRedraw) {
            redraw();
            needRedraw = GL_FALSE;
        }
    }
}
```

References

[1] Allen Akin, "Analysis of PEX 5.1 and OpenGL 1.0," Silicon Graphics, August 3, 1992.

[2] Paul Haeberli, Kurt Akeley, "The Accumulation Buffer: Hardware Support for High-Quality Rendering," *Proceedings of SIGGRAPH '90*, August 1990, pp. 309-318.

[3] Phil Karlton, "Integrating the GL into the X Environment: A High Performance Rendering Extension Working with and Not Against X," *The X Resource: Proceeding of the 6th Annual X Technical Conference*, O'Reilly & Associates, Issue 1, Winter 1992.

[4] Patricia McLendon, *Graphics Library Programming Guide*, Silicon Graphics, 1991.

[5] Jackie Neider, Tom Davis, Mason Woo, *OpenGL Programming Guide: The official guide to learning OpenGL, Release 1*, Addison Wesley, 1993.

[6] OpenGL Architecture Review Board, *OpenGL Reference Manual: The official reference document for OpenGL, Release 1*, Addison Wesley, 1992.

[7] Mark Segal, Kurt Akeley, *The OpenGL™ Graphics System: A Specification*, Version 1.0, Silicon Graphics, June 30, 1992.

[8] Paul Strauss, Rikk Carey, "An Object-Oriented 3D Graphics Toolkit," *Proceedings of SIGGRAPH '92*, July 1992, pp. 341-347.

[9] Paula Womack, et.al., "PEX Protocol Specification, Version 5.1," The X Consortium, August 31, 1992.