Table of Contents

Table of Contents

OpenGL FAQ and Troubleshooting Guide v1.2001.11.01

1 About the FAQ

2 Getting Started

1 About the FAQ

1.010 Introduction

The OpenGL Technical FAQ and Troubleshooting Guide will answer some basic technical questions and explain frequently misunderstood topics, features, and concepts.

All text, example code, and code snippets in this FAQ are in the public domain. The text, example code, and code snippets can be used and copied freely. Hyperlinks to text and example code not contained in this FAQ may or may not be public domain, and their usage may be restricted accordingly.

1.020 How to contribute, and the contributors

This FAQ is maintained by Paul Martz (email: martz@frii.com).

Contribute to the FAQ by contacting Paul Martz, the FAQ maintainer. Suggestions, topics, corrections, information, and pointers to information are welcome.

The following people have explicitly contributed written material to this FAQ: Brian Bailey, Brett Johnson, Paul Martz, Samuel Paik, Joel Parris, and Thant Tessman.

Several people have unwittingly contributed information through conversations with the FAQ maintainer and/or their several informative postings to the comp.graphics.api.opengl newsgroup. A partial list includes: Darren Adams, Stephane Albi, Mark B. Allan, Pierre Alliez, Steve Baker, Konstantin Baumann, Ron Bielaski, Kevin Bjorke, Lars Blaabjerg, Frans Bouma, Anders Brodersen, Michael Brooks, Jeff Burrell, Won Chun, Mike Coplien, Bart De Lathouwer, Angus Dorbie, Bob Ellison, Glenn Forney, Ron Fosner, Phil Frisbie Jr, Michael I. Gold, Paul Groves, Charles E. Hardwidge, Jason Harrison, Michael S. Harrison, Mike Heck, Chris Hecker, Scott Heiman, Helios, Blaine Hodge, Steve Humphreys, Michael Kennedy, Marco Klemm, Mark Kilgard, Sam Kirchmeier, Oliver Kurowski, Michael Kurth, Wolfram Kuss, Bruce Lamming, Robert Lansdale, Jon Leech, Stuart Levy, Barthold Lichtenbelt, Mike Lischke, Ben Loftin, Hrafn Loftsson, Konstantinos Manthos, Jean−Luc Martinez, Steve McAndrewSmith, Phil McRevis, David Melinosky, Reed Mideke, Mark Morley, Teri Morrison, Duncan Murdoch, Doug Newlin, Geert Poels, David Poon, Lev Povalahev, Dirk Reiners, Stephane Routelous, Schneide, Shaleh, Dave Shreiner, Hal Snyder, Andrew F. Vesper, Jon White, Lucian Wischik, Mitch Wolberg, Karsten Wutzke, and Zed.

Jeff Molofee's OpenGL code was the inspiration for Brian Bailey's MFC example (accessible from question 5.160). Jeff maintains the NeHe Web page.

Special thanks to Yukio Andoh for the Japanese translation, and Thomas Kern for the German translation.

1.030 Download the entire FAQ as a Zip file

Download the entire FAQ in a single zip file (~180KB).

1.031 Printing the PDF FAQ

The entire FAQ is available as a single PDF file for easy printing.

PDF FAQ (~610KB)
Zipped PDF FAQ (~324KB)

1.040 Change Log

Date	Notes
November 1, 2001	Table of Contents: Fixed typo. 1.020: Changed email address for FAQ maintainer. 2.005: Updated hyperlink. 2.005: Added hyperlink to C++ 3D math library web page. 2.010: Updated hyperlink. 2.020: Updated hyperlink. 2.130: Updated hyperlink. 3.010: Updated hyperlink. 3.015: New question, "Where can I get GLUT?" 3.025: New question, "The GLUT source code license is very restrictive. Is there an alternative?" 3.027 New question, "Why does glutTimerFunc() only execute my callback once?" 3.040: Updated hyperlink. 3.110: Updated hyperlink. 4.040: Updated hyperlink. 4.060: Updated hyperlink. 5.030: Added information on selecting a software−only pixel format. 5.121: Added information. 5.150: Added information. 5.180: Updated hyperlink 5.190, 5.200, 5.210: Added lots of additional information. 9.005: Fixed typo. 9.085: Clarified use of trig functions. 9.130: Added information. 10.010: Added link to new culling tutorial. 17.xxx: Updated and added links regarding texture mapped fonts. 21.090: Updated hyperlink. 22.010: Changed hyperlink. 23.030: Added information on OpenGL 1.3 and the proposed OpenGL 2.0. 23.050: Added information. 24.040: Updated hyperlink. 24.140: Added information. 24.150: Removed GPT information and added GLAnalyze Pro information. Appendix B: Added information on weight.cpp. viewcull.c: Fixed bounding box initialization typo. weight.cpp: New file.
March 8, 2001	history.htm: New file, links on OpenGL's history. Table of Contents: Added link to history.htm as Appendix C.

	2.020: Fixed broken hyperlink to online blue book.
January 17, 2001	2.080: Removed copywritten material and added a public domain replacement.
October 15, 2000	Table of Contents: Version is now present in masthead. Table of Contents: Corrected Kanji characters. 7.030: Added more information on multiple monitor support. 17.010, 17.030: Repaired or removed broken links.
October 8, 2000	source.htm: New file, a consolidated index to FAQ source code. Table of Contents: Added links to German and Japanese translations. Table of Contents: Added link to source.htm as Appendix B. 2.005: Fixed broken link. 2.010: Added information. 2.110: Added information. 5.030: Added information on disabling hardware rendering. 5.070: Added information. 5.080: Added information. 5.121: New question, "Why does my double buffered window appear incomplete or contain black stripes?" 5.160: Fixed HTML. 5.180: New question, "Where can I find MFC examples?" 5.190: New question, "What do I need to know about mixing WGL and GDI calls?" 5.200: New question, "Why does my code crash under Windows NT or 2000 but run fine under 9x?" 5.210: New question, "How do I properly use WGL functions?" 7.030: New question, "How can I use multiple monitors?" 8.110: New question, "How can I create a stereo view?" 9.005: Added information. 9.162: New question, "How can I transform an object with a given yaw, pitch, and roll?" 10.020: Added information. 18.140: Added information. 24.160: Added information. viewcull.c: Fixed bug with incorrect matrix mode.
August 24, 2000	Table of Contents: Added access to mslinks.htm as an appendix in the main table of contents 1.031: New question, "Printing the PDF FAQ" lookat.cpp: Fix comment typo.
August 1, 2000	mslinks.htm: New file, contains links to OpenGL information on Microsoft Web sites. 2.005: Added information. 2.010: Added link to mslinks.htm. 2.050: Added information. 2.080: Fixed incorrect parameters to XCreateWindow().

	2.160: New question, "What are the OpenGL Conformance Tests?" 3.020: Fixed typo. 5.010: Added link to mslinks.htm. 5.050: Added information. 5.150 New question, "What do I need to know to use OpenGL with MFC?" 5.160: New question, "How can I use OpenGL with MFC?" 5.170: New question, "Is OpenGL inherently slower when used with MFC?" 6.010: New question, "How do I use overlay planes?" 7.020: Added information. 9.001: Fixed hyperlink. 17.010: Removed broken hyperlink. 23.010: Added information. 23.090: Added information. 24.050: Fixed hyperlink.
July 6, 2000	2.005: Added information. 2.020: Added hyperlink to online OpenGL Reference Manual. 3.030: Corrected code snippet. 3.070: Added information. 3.090: Added information. 4.020: Added hyperlink to GLE web site. 5.040: Added information. 7.020 New question, "What user interface system should I use?" 9.011 New question, "How are coordinates transformed? What are the different coordinate spaces?" 16.150: New question, "How can I use vertex arrays to share vertices?" 17.010: Added information. 17.030: Added additional links to GLTT. 21.090: Added information. 21.110: Added link to source for using TGA files as texture maps. Corrected bogus hyperlink. 22.020: Added information. 22.100 New question, "How doI make triangle strips out of triangles?" 23.070: Added link to modified glext.h for hiding function pointers. 23.090: Added direct links to glext.h, wglext.h, and glxext.h. 24.040: Added information. 24.050: Added information. 24.150: Added link to Intel's GPT. viewcull.c: Comment tweak.
June 6, 2000	Added a link to GLTT in question 17.030.
June 3, 2000	Updated with miscellaneous corrections and additional information.
May 30, 2000	Fixed HTML problems in index.htm, and sections 1 and 2.
May 29, 2000	Updated with miscellaneous corrections..

May 28, 2000	Version 1.0. Significant changes include a full technical edit for hyperlinks, notational conventions, grammar, and spelling. Filled in many holes. Corrected lots of incorrect information.
April 16, 2000	Beta version.
March 19, 2000	Updated with miscellaneous corrections, additions, and changes.
March 12, 2000	Alpha version.

2 Getting Started

2.005 Where can I find 3D graphics info?

The comp.graphics.algorithms FAQ contains 3D graphics information that isn't specific to OpenGL.

For general OpenGL and 3D graphics information, Advanced Graphics Programming Techniques Using OpenGL is a good online source of information.

An excellent general computer graphics text is *Computer Graphics: Principles and Practice*, Second Edition, by James Foley, et al. ISBN 0−201−12110−7. This book may be out of print, however, some online book retailers still seem to have it for sale. Try amazon.com. There may be a third edition planned for release in January 2001

Delphi code for performing basic vector, matrix, and quaternion operations can be found here.

A C++ 3D math library.

Here's another source for linear algebra source code.

2.010 Where can I find examples, tutorials, documentation, and other OpenGL information?

OpenGL is the most extensively documented 3D graphics API to date. Information is all over the Web and in print. It would be impossible to exhaustively list all sources of OpenGL information. This FAQ therefore provides links to large storehouses of information and sites that maintain many links to other OpenGL sites.

OpenGL Organization Web Page

SGI's OpenGL FTP archive and SGI's OpenGL Web site.

HP's OpenGL subject index.

OpenGL Basics FAQ

OpenGL Game Developer's FAQ. In addition to information on OpenGL, the OpenGL Game Developer's FAQ has information on subscribing to the OpenGL Game Developer's mailing list.

The EFnet #OpenGL FAQ

Samuel Paik has created a large repository of links to OpenGL information on Microsoft Web sites.

The OpenGL org web site has the current OpenGL specification and manual pages. The v1.1 spec is also available here as a web page.

A repository of OpenGL implementations for several platforms

The GLUT source code distribution contains several informative OpenGL examples and demos.

Codeguru maintains a small, but growing list, of useful OpenGL sample code.

Lucian Wischik's Web page at http://www.wischik.com/lu/programmer/wingl.html contains excellent information on Microsoft Windows OpenGL, especially with 3dfx hardware.

The NeHe Web page has many links to other sites and plenty of useful tutorials. Many people have found this site useful.

See Blaine Hodge's Web page for info on Win32 OpenGL programming.

An interactive OpenGL tutorial can be found here.

Check gamedev.net for OpenGL tutorials and articles.

2.020 What OpenGL books are available?

There are several books on OpenGL, but the two most revered are the "red" and "blue" books:

OpenGL Programming Guide, Third Edition, Mason Woo et al.
ISBN 0−201−60458−2 (aka the red book)

OpenGL Reference Manual, Third Edition, Dave Shreiner (Editor), et al.
ISBN 0−201−65765−1 (aka the blue book)

The third edition of these books describes OpenGL 1.2. The original and second editions describe 1.0 and 1.1, respectively.

The OpenGL Org web site has a link to an online version of the 1.1 Programming Guide.

For the OpenGL Reference Manual, here are two sources:

HP's Web−browsable OpenGL Reference Manual, Second Edition (for OpenGL 1.1).
Manual pages similar to the OpenGL Reference Manual.

In addition to the red and blue books, see the green book for X Windows programming, and the white book for Microsoft Windows programming. You can obtain a more exhaustive list of OpenGL books by visiting the www.opengl.org Web site.

2.030 What OpenGL chat rooms and newsgroups are available?

The Usenet newsgroup, devoted to OpenGL programming, is comp.graphics.api.opengl.

The #OpenGL IRC channel is devoted to OpenGL discussion.

2.040 What OpenGL implementations come with source code?

The Mesa library is an OpenGL look−alike. It has an identical interface to OpenGL. The only

reason it can't be called "OpenGL" is because its creator hasn't purchased a license from the OpenGL ARB.

The OpenGL Sample Implementation is also available.

2.050 What compiler can I use?

OpenGL programs are typically written in C and C++. You can also program OpenGL from Delphi (a Pascal−like language), Basic, Fortran, Ada, and others.

Borland

Programming OpenGL with Borland compilers is the same as with any other compiler, with one exception: OpenGL apps can produce floating point exceptions at run time. To disable these harmless errors, add the following to your app before you call an OpenGL function:

```
_control87(MCW_EM, MCW_EM);
```

Borland users need to be aware that versions prior to 4.0 only support OpenGL 1.0 out of the box. Download the OpenGL SDK from Microsoft to use OpenGL v1.1, or v1.2 when it becomes available.

Use Borland's implib utility to generate Borland−compatible .LIB export libraries from Microsoft−compatible .DLL libraries. If you accidently link with Microsoft−format .LIB files, you will receive a linker error like the following:

```
C:\BORLAND\BCC55\LIB\GLUT32.LIB' contains invalis OMF record, type 0x21 (possibly COOF)
```

The bornews.borland.com Usenet news server has two newsgroups that pertain to graphics: borland.public.delphi.graphics and borland.public.cppbuilder.graphics.

The Borland Community is an online source of FAQs that address Borland compiler issues.

For information on how to use OpenGL through the commercial version of Borland C++ Builder, visit Scott Heiman's Web page. For information on the free version, go here.

The book *Delphi Developer's Guide to OpenGL* by Jon Jacobs is available. The author maintains a web page for this book.

Information on using OpenGL from Delphi can be found here and at the Delphi3D web page. Code and utilities for using OpenGL through Delphi are available.

Visual Basic

Here are three sites with info on how to use OpenGL through Visual Basic:

http://www.softoholic.bc.ca/opengl/down.htm
http://www.weihenstephan.de/~syring/ActiveX/
http://www.ieighty.net/~davepamn/colorcube.html.

2.060 What do I need to compile and run OpenGL programs?

The following applies specifically to C/C++ usage.

To compile and link OpenGL programs, you'll need OpenGL header files and libraries. To run OpenGL programs you may need shared or dynamically loaded OpenGL libraries, or a vendor−specific OpenGL Installable Client Driver (ICD) specific to your device. Also, you may need include files and libraries for the GLU and GLUT libraries. Where you get these files and libraries will depend on which OpenGL system platform you're using.

The OpenGL Organization maintains a list of links to OpenGL developer and end−user files. You can download most of what you need from there.

Under Microsoft Windows 9x, NT, and 2000:

If you're using Visual C++, your compiler comes with include files for OpenGL and GLU, as well as .lib files to link with.

For GLUT, download these files. Install glut.h in your compiler's include directory, glut32.lib in your compiler's lib directory, and glut32.dll in your Windows system directory (c:\windows\system for Windows 9x, or c:\winnt\system32 for Windows NT/2000).

In summary, a fully installed Windows OpenGL development environment will look like this:

File	Location
gl.h glut.h glu.h	[compiler]\include\gl
Opengl32.lib glut32.lib glu32.lib	[compiler]\lib
Opengl32.dll glut32.dll glu32.dll	[system]

where [compiler] is your compiler directory (such as c:\Program Files\Microsoft Visual Studio\VC98) and [system] is your Windows 9x/NT/2000 system directory (such as c:\winnt\system32 or c:\windows\system).

If you're on a hardware platform that accelerates OpenGL, you'll need to install the ICD for your device. This may have shipped with your hardware, or you can download it from your hardware vendor's Web page. Your vendor may also provide a replacement or addition for gl.h, which provides definitions and declarations for vendor−specific OpenGL extensions. See the extensions section in this FAQ for more information.

If you see files such as opengl.lib and glut.lib, these are SGI's unsupported libraries for Microsoft Windows. They should not be used. To use hardware acceleration, the Microsoft libraries are recommended. More info on the SGI libraries can be found here. Always link

with either all Microsoft libraries (e.g., glu32.lib, glut32.lib, and opengl32.lib) or all SGI libraries (e.g., glu.lib, glut.lib, and opengl.lib). You can't use a combination of both Microsoft libarires and SGI libraries. However, you can install both sets of libraries on the same system. If you use SGI's .lib files, you'll need the corresponding .dll files installed in your system folder. (i.e., linking against opengl.lib requires that opengl.dll is installed at run time).

You'll need to instruct your compiler to link with the OpenGL, GLU, and GLUT libraries. In Visual C++ 6.0, you can accomplish this with the Project menu's Settings dialog box. Scroll to the Link tab. In the Object/library modules edit box, add glut32.lib, glu32.lib, and opengl32.lib to the end of any text that is present.

For UNIX or UNIX−like operating systems:

If you don't find the header files and libraries that you need to use in standard locations, you need to point the compiler and linker to their location with the appropriate −I and −L options. The libraries you link with must be specified at link time with the −l option; −lglut −lGLU −lGL −lXmu −lX11 is typical.

If you want to use GLUT, you need to download it. If you can't find the precompiled binaries, you'll want to download the source and compile it. GLUT builds easily on many platforms, and comes with many README files explaining how to do a build. The GLUT compiler uses the imake utility, which makes it easy to build GLUT on new platforms.

For Linux, Macintosh, and other systems:

Mesa is a free OpenGL−like library that is available on a number of platforms. You might also check the Developer section at The OpenGL Organization's Web page for information about OpenGL for your specific platform.

2.070 Why am I getting compile, link, and runtime errors?

Most compile and link errors stem from either a system that doesn't have the OpenGL development environment installed correctly, or failure to instruct the compiler where to find the include and library files.

If you are encountering these problems in the Windows 9x/NT/2000 environment, read question 2.060 above to ensure that you've installed all files in their correct locations, and that you've correctly instructed the linker to find the .lib files.

Also, note that you'll need to put an #include <windows.h> statement before the #include<GL/gl.h>. Microsoft requires system DLLs to use a specific calling convention that isn't the default calling convention for most Win32 C compilers, so they've annotated the OpenGL calls in gl.h with some macros that expand to nonstandard C syntax. This causes Microsoft's C compilers to use the system calling convention. One of the include files included by windows.h defines the macros.

Another caveat for Win32 developers: With Microsoft Visual C++ (and probably most other Win32 C compilers), the standard Win32 application entry point is WinMain with four parameters, rather than main(int argc, char **argv). Visual C++ has an option to include code to parse the standard Win32 application entry, and call main with a parsed command line; this is called a console application instead of a Win32 application. If you download

code from the Net and try to build it, make sure you've configured your compiler to build the right kind of application, either console or Win32. This can be controlled with linker options or pragmas. Microsoft Visual C++ supports the following pragmas for controlling the entry point and application type:

```
// Use one of:
#pragma comment (linker, "/ENTRY:mainCRTStartup")
#pragma comment (linker, "/ENTRY:wmainCRTStartup")
#pragma comment (linker, "/ENTRY:WinMainCRTStartup")
#pragma comment (linker, "/ENTRY:wWinMainCRTStartup")
// Use one of:
#pragma comment (linker, "/SUBSYSTEM:WINDOWS")
#pragma comment (linker, "/SUBSYSTEM:CONSOLE")
```

The following is a table of errors and their possible causes and solutions. It is targeted toward Microsoft Visual C++ users, but the types of errors can apply, in general, to any platform.

Example error text	Possible cause and solution
d:\c++\file.c(20) : warning C4013: 'glutDestroyWindow' undefined; assuming extern returning int d:\c++\file.c(71) : warning C4013: 'glMatrixMode' undefined; assuming extern returning int d:\c++\file.c(71) : error C2065: 'GL_MODELVIEW' : undeclared identifier	Didn't #include gl.h, glu.h, or glut.h A GLUT source file should: #include <GL/glut.h> Non−GLUT source files should: #include <GL/glu.h> #include <GL/gl.h>
c:\program files\microsoft visual studio\vc98\include\gl\gl.h(1152) : error C2054: expected '(' to follow 'WINGDIAPI' c:\program files\microsoft visual studio\vc98\include\gl\gl.h(1152) : error C2085: 'APIENTRY' : not in formal parameter list	Didn't #include windows.h or included it after gl.h. Source files that use neither GLUT nor MFC, but which make calls to OpenGL, should: #include <windows.h> #include <GL/gl.h>
d:c++\file.c(231) : warning C4305: 'initializing' : truncation from 'const double ' to 'float '	Floating−point constants (e.g., 1.0) default to type double. This is a harmless warning that can be disabled in Visual C++ with: #ifdef WIN32 #pragma warning(disable : 4305) #endif at the top of the source file.
file.obj : error LNK2001: unresolved external symbol __imp__glMatrixMode@4 file.obj : error LNK2001: unresolved external symbol __imp__glViewport@16	Didn't link with opengl32.lib, glu32.lib, or glut32.lib. Section 2.060 above describes how to inform the Visual C++ 6 linker about the location of the .lib files.

file.obj : error LNK2001: unresolved external symbol __imp__glLoadIdentity@0	
The dynamic link library OPENGL.dll could not be found in the specified path..	Failure to correctly install .dll files. See section 2.060 above for information on where these files should be installed for your Windows system.
Nothing renders, just a blank window.	Mixed linkage against .lib files from both Microsoft and SGI can cause this. Make sure you specify either glut32.lib, glu32.lib opengl32.lib or glut.lib, glu.lib, and opengl.lib to the linker, but not a combination of the files from these two file sets.
LIBCD.lib(wincrt0.obj) : error LNK2001: unresolved external symbol _WinMain@16 Debug/test.exe : fatal error LNK1120: 1 unresolved externals Error executing link.exe.	Not an OpenGL question per se, but definitely a FAQ on comp.graphics.api.opengl due to the way GLUT works in Microsoft Windows. You should instruct your compiler to build a console application. It's trying to find the Win32 entry point, but your code wasn't written as a Win32 application.
Multiple access violations appear when running a Microsoft OpenGL MFC−based application.	Set the CS_OWNDC style in the PreCreate*() routines in the view class.
Floating−point exceptions occur at runtime. The application was built with Borland C.	Add the following to your app before you call any OpenGL functions: _control87(MCW_EM, MCW_EM); This is from Borland's own FAQ article #17197.

2.080 How do I initialize my windows, create contexts, etc.?

It depends on your windowing system. Here's some basic info, but for more details, refer to the documentation for your specific windowing system or a newsgroup devoted to programming in it.

GLUT

The basic code for creating an RGB window with a depth buffer, and an OpenGL rendering context, is as follows:

```
#include <GL/glut.h>

int main(int argc, char** argv)
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500,500);
    glutInitWindowPosition(0,0);
```

```
    glutCreateWindow("Simple");

    /* ... */
}
```

The calls to set the window size and position are optional, and GLUT uses a default size and location if they are left out.

X Windows

You can create an RGB window with a depth buffer in X Windows using the following code:

```
#include <GL/glx.h>
#include <GL/gl.h>

main (int argc, char **argv)
{
    Display display;
    XVisualInfo *vinfo;
    XSetWindowAttributes swattr;
    int attrList[20];
    int indx=0;
    Window wid;
    GLXContext util_glctx;

    if (!(display = XOpenDisplay ("")))
        exit ();

    attrList[indx] = GLX_USE_GL; indx++;
    attrList[indx] = GLX_DEPTH_SIZE; indx++;
    attrList[indx] = 1; indx++;
    attrList[indx] = GLX_RGBA; indx++;
    attrList[indx] = GLX_RED_SIZE; indx++;
    attrList[indx] = 1; indx++;
    attrList[indx] = GLX_GREEN_SIZE; indx++;
    attrList[indx] = 1; indx++;
    attrList[indx] = GLX_BLUE_SIZE; indx++;
    attrList[indx] = 1; indx++;
    attrList[indx] = None;

    vinfo = glXChooseVisual(display, DefaultScreen(dpy), attrList);
    if (vinfo == NULL) {
        printf ("ERROR: Can't open window\n");
        exit (1);
    }

    swattr.colormap=XCreateColormap (display ,RootWindow (display,vinfo->screen),
        vinfo->visual, AllocNone);
    swattr.background_pixel = BlackPixel (display, vinfo->screen);
    swattr.border_pixel = BlackPixel (display, vinfo->screen);

    wid = XCreateWindow(display,RootWindow(display, vinfo->screen),
          30, 30, width, height, 0, vinfo->depth, CopyFromParent,
          vinfo->visual,CWBackPixel | CWBorderPixel | CWColormap, &swattr);

    util_glctx = glXCreateContext(display, vinfo, NULL, True);
    if (util_glctx == NULL) {
        printf("glXCreateContext failed \n");
        return(-1);
    }
```

```
    if (!glXMakeCurrent(display, wid, util_glctx)) {
        printf("glXMakeCurrent failed \n");
        return(-1);
    }
}
```

Microsoft Windows 9x/NT/2000

The window must be created with the following bits OR'd into the window style: WS_CLIPCHILDREN | WS_CLIPSIBLINGS. Do this either when CreateWindow is called (in a typical Win32 app) or during the PreCreateWindow function (in an MFC app).

Once the window is created (when a WM_CREATE message arrives or in the OnInitialUpdate callback), use the following code to set the pixel format, create a rendering context, and make it current to the DC.

```
// Assume:
// HWND hWnd;

HDC hDC = GetDC (hWnd);
PIXELFORMATDESCRIPTOR pfd;

memset(&pfd, 0, sizeof(PIXELFORMATDESCRIPTOR));
pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR);
pfd.nVersion = 1;
pfd.dwFlags = PFD_SUPPORT_OPENGL | PFD_DRAW_TO_WINDOW;
pfd.iPixelType = PFD_TYPE_RGBA;
pfd.cColorBits = 24;
pfd.cDepthBits = 32;
pfd.iLayerType = PFD_MAIN_PLANE;

int pixelFormat = ChoosePixelFormat(hDC, &pfd);
if (pixelFormat == 0) {
    // Handle error here
}

BOOL err = SetPixelFormat (hDC, pixelFormat, &pfd);
if (!err) {
    // Handle error here
}

hRC = wglCreateContext(hDC);
if (!hRC) {
    // Handle error here
}

err = wglMakeCurrent (hDC, hRC);
if (!err) {
    // Handle error here
}
```

You can then make the rendering context noncurrent, and release the DC with the following calls:

```
WglMakeCurrent(NULL,NULL);
ReleaseDC (hWnd, hDC);
```

2.090 How do I create a full−screen window?

Prior to GLUT 3.7, you can generate a full−screen window using a call to glutFullScreen(void). With GLUT 3.7 and later, a more flexible interface was added.

With glutGameModeString(), an application can specify a desired full−screen width and height, as well as the pixel depth and refresh rate. You specify it with an ASCII character string of the form [width]x[height]:[depth]@[hertz]. An application can use this mode if it's available with a call to glutEnterGameMode(void). Here's an example:

```
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
glutGameModeString("640x480:16@60");
glutEnterGameMode();
```

Also, see the "Full Screen Rendering" section in the OpenGL game developer's FAQ.

2.100 What is the general form of an OpenGL program?

There are no hard and fast rules. The following pseudocode is generally recognized as good OpenGL form.

```
program_entrypoint
{
    // Determine which depth or pixel format should be used.
    // Create a window with the desired format.
    // Create a rendering context and make it current with the window.
    // Set up initial OpenGL state.
    // Set up callback routines for window resize and window refresh.
}

handle_resize
{
    glViewport(...);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    // Set projection transform with glOrtho, glFrustum, gluOrtho2D, gluPerspective, etc
}

handle_refresh
{
    glClear(...);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    // Set view transform with gluLookAt or equivalent

    // For each object (i) in the scene that needs to be rendered:
        // Push relevant stacks, e.g., glPushMatrix, glPushAttrib.
        // Set OpenGL state specific to object (i).
        // Set model transform for object (i) using glTranslatef, glScalef, glRotatef, a
        // Issue rendering commands for object (i).
        // Pop relevant stacks, (e.g., glPopMatrix, glPopAttrib.)
    // End for loop.

    // Swap buffers.
}
```

2.110 My window is blank. What should I do?

A number of factors can cause a blank window when you're expecting a rendering. A blank window is generally caused by insufficient knowledge of 3D graphics fundamentals, insufficient knowledge of basic OpenGL mechanisms, or simply a mistake in the code.

There are a number of OpenGL books and online resources as well.

What follows is a list some of the more common causes of the dreaded "Black Window Syndrome" and what to do to fix it.

- Your application may have made an erroneous call to OpenGL. Make liberal calls to glGetError(). You might create a macro or inline function, which does the following:

```
{
    GLint err = glGetError();
    if (err != GL_NO_ERROR) DisplayErrorMessage();
}
```

Place this code block after suspect groups of OpenGL function calls, and take advantage of the preprocessor, which will ensure that the calls can be eliminated easily in a production compile (i.e., #ifdef DEBUG...#endif).

glGetError() is the only way to tell whether you've issued an erroneous function call at runtime. If an OpenGL function generates an error, OpenGL won't process the offending function. This is often the cause of incorrect renderings or blank windows.

- Incorrect placement of *zFar* and *zNear* clipping planes with respect to the geometry can cause a blank window. The geometry is clipped and nothing is rendered. *zFar* and *zNear* clipping planes are parameters to the glOrtho(), gluOrtho2D(), glFrustum(), and gluPerspective() calls. For glFrustum() and gluPerspective(), it's important to remember that the *zNear* and *zFar* clipping planes are specified as distances in front of the eye. So, for example, if your eye is at (0,0,0), which it is in OpenGL eye coordinate space, and the *zNear* clipping plane is at 2.0 and all of your geometry is in a unit cube centered at the origin, the *zNear* plane will clip all of it and render nothing. You'll need to specify a ModelView transform to push your geometry back, such as a call to glTranslatef(0,0,−3).

Similarly, the *zFar* clipping plane might be a problem if it is placed at, for example, 10.0, and all of your geometry is further than 10.0 units from the eye.

- Incorrect transforms in general can cause a blank window. Your code is attempting to set the view and modeling transform correctly, but due to some problem, the net transformation is incorrect, and the geometry doesn't fall within the view volume. This is usually caused by a bug in the code or a lack of understanding of how OpenGL transforms work.

It's usually best to start simple and work your way to more complex transformations. Make code changes slowly, checking as you go, so you'll see where your mistakes came from.

- Another cause of the blank window is a failure to call glEnd() or failure to call glBegin(). Geometry that you specify with one of the glVertex*() routines must be wrapped with a glBegin()/glEnd() pair to be processed by OpenGL. If you leave out both glBegin() and glEnd(), you won't get an error, but nothing will render.

If you call glBegin(), but fail to call glEnd() after your geometry, you're not guaranteed that anything will render. However, you should start to see OpenGL errors once you call functions (e.g., glFlush()) that can't be called within a glBegin()/glEnd() pair. If you call glEnd() but fail to call glBegin(), the glEnd() call will generate an error. Checking for errors is always a good idea.

- Failure to swap buffers in a double−buffered window can cause blank windows. Your primitives are drawn into the back buffer, but the window on the screen is blank. You need to swap buffers at the end of each frame with a call to SwapBuffers, glXSwapBuffers, or glutSwapBuffers.

- Failure to glClear() the buffers, in particular the depth buffer, is yet another cause. Call glClear() at the start of every frame to remedy this failue.
- Some OpenGL implementations have bugs that can cause blank windows or other incorrect rendering. Try your application on another implementation. Correct behavior on one or more other implementations is strong evidence of a bug in the first implementation.

2.120 The first frame is rendered correctly, but subsequent frames are incorrect or further away or I just get a blank screen. What's going on?

This is often caused by a failure to realize that OpenGL matrix commands multiply, rather than load over the top of the current matrix.

Most OpenGL programs start rendering a frame by setting the ModelView matrix to the identity with a call to glLoadIdentity(). The view transform is then multiplied against the identity matrix with, for example, a call to gluLookAt(). Many new programmers assume the gluLookAt() call will load itself onto the current matrix and therefore fail to initialize the matrix with the glLoadIdentity() call. Rendering successive frames in this manner causes successive camera transforms to multiply onto each other, which normally results in an incorrect rendering.

2.130 What is the AUX library?

Very important: Don't use AUX. Use GLUT instead.

The AUX library was developed by SGI early in OpenGL's life to ease creation of small OpenGL demonstration programs. It's currently neither supported nor maintained. Developing OpenGL programs using AUX is strongly discouraged. Use the GLUT instead. It's more flexible and powerful and is available on a wide range of platforms.

For related information, see the GLUT Section and SGI's GLUT FAQ.

2.140 What support for OpenGL does {Open,Net,Free}BSD or Linux provide?

The X Windows implementation, XFree86 4.0, includes support for OpenGL using Mesa or the OpenGL Sample Implementation. XFree86 is released under the XFree86 license. http://www.xfree86.org/

SGI has released the OpenGL Sample Implementation as open source. It can be built as an X server GLX implementation. It has been released under SGI Free Software License B.

http://oss.sgi.com/projects/ogl−sample/

The Mesa 3D Graphics Library is an OpenGL clone that runs on many platforms, including MS−DOS, Win32, *BSD and Linux. On PC UNIX platforms Mesa can be built to use GGI, X Windows, and as an X server GLX implementation. Mesa is hardware accelerated for a number of 3D graphics accelerators. Mesa 3.1 and later was released under an XFree86−style license. Versions prior to 3.1 were released under GPL.
http://mesa3d.sourceforge.net/

Utah−GLX is a hardware accelerated GLX implementation for the Matrox MGA−G200 and G−400, ATI 3D RAGE PRO, Intel i810, NVIDIA RIVA, and S3 ViRGE. Utah−GLX is based on Mesa. It is not clear what license Utah−GLX is released under.
http://utah−glx.sourceforge.net/

Metro Link OpenGL and Extreme 3D are GLX extensions for Metro Link X servers. Metro Link OpenGL is a software implementation that can use accelerated X operations to gain a performance advantage over other software implementations. Metro Link Extreme 3D is a hardware−accelerated implementation for REALimage, GLINT GMX 1000, 2000, GLINT DMX, GLINT MX, GLINT TX, and Permedia 2 and 3. http://www.metrolink.com/

Xi Graphics 3D Accelerated−X is an X server with GLX support. Supported devices include: ATI Xpert 2000, ATI Rage Fury Pro, ATI Rage Fury, ATI Rage Magnum, ATI All−in−Wonder 128 (all ATI RAGE 128 I believe), 3Dlabs Oxygen VX1, 3Dlabs Permedia 3 Create! (Permedia 3), Diamond Stealth III S540, Diamond Stealth III S540 Extreme, Creative Labs 3D Blaster Savage4 (S3 Savage4), Number Nine SR9, 3Dfx Voodoo 3000, 3Dfx Voodoo 3500 software.

2.150 Where is OpenGL 1.2?

When this was written (early 2000), few OpenGL 1.2 implementations are available. Sun and IBM are shipping OpenGL 1.2. The OpenGL−like Mesa library also supports 1.2. The OpenGL Sample Implementation is also available.

Microsoft hasn't released OpenGL 1.2 yet. As of their most recent official announcement, it is to be included in a later Windows 2000 service pack. Once Microsoft releases OpenGL 1.2, you'll probably need a new driver to take advantage of its features.

Many OpenGL vendors running on Microsoft already support OpenGL 1.2 functionality through extensions to OpenGL 1.1.

OpenGL vendors that run on OS other than Microsoft will release OpenGL 1.2 on their own schedules.

The OpenGL 1.2 specification is available from http://www.opengl.org. The red and blue books have recently been revised to cover OpenGL 1.2 functionality.

2.160 What are the OpenGL Conformance Tests?

The OpenGL Conformance Tests are a suite of tests that the OpenGL ARB uses to certify an

OpenGL implementation conforms to the OpenGL spec, and, after paying the licensing fee, is therefore entitled to call itself "OpenGL". The source code for the conformance tests can be licensed from the OpenGL ARB.

The conformance tests were recently upgraded to test the full OpenGL 1.2 functionality. They do not exercise extension entry points. They will, however, report the full list of extensions that an implementation claims to support.

covogl is a special conformance test that simply calls every standard entry point. It is a "coverage" test, meant to ensure that all entry points exist and don't crash. All the other tests are intended to test spec conformance for a specific rendering task.

The test mustpass.c tests a defined core of functionality that all OpenGL implementations must support. (You must be able to render a line," etc.) Vendors that fail other tests are still allowed to use the name "OpenGL", but they must be able to show that they understand the bugs, and are working to resolve the issue in a future release.

The ability to push and pop state is thoroughly tested. Each test that runs is of the form:

push state
change state
run test
pop state
check all state values (via glGet*()) to make sure they have returned to the default values.

Some tests have some built−in error that allows for some variation from the OpenGL specification. For example, OpenGL spec states that when rasterizing a triangle, the center of each rendered pixel must be within the mathematical boundary of the triangle. However, the conformance test for rasterizing triangles allows pixels to be as much as 1/2 pixel outside this boundary without reporting an error.

Conversely, some tests appear to test for more than the spec calls for. For example, the test for alpha test requires 10 percent (between 4 and 5 bits) precision to pass, whereas the spec calls for only a single bit of precision.

Some tests don't make sense if you are not intimately familiar with the spec. For example, the spec says it's perfectly OK to not antialias polygons when the user has requested it, and the conformance tests allow this. Another example is dithering; the spec allows for a great deal of implementation variety, including no dithering at all, and as a consequence, the conformance tests won't display an error if your implementation doesn't dither.

All tests support path levels that execute the same tests with a variety of state settings that should still produce the same result. For example, rendering a triangle with polygon stipple disabled should produce the same result as rendering it with polygon stipple enabled and a stipple pattern of all 1 bits. Again, this should be identical to rendering with blending enabled and a blend function of (GL_ONE,GL_ZERO). A number of path levels are available, each testing more and more complex combinations of state settings.

All tests are run on all available pixel formats or visual types, including (if available) color index.

All tests verify correct rendering with glReadPixels(). Some tests read the entire test window, while other read only a few key pixels. In general, the tests use GL_RGBA and GL_FLOAT as the *type and format*. However, the readpix.c test thoroughly tests all *type and format* combinations. If glReadPixels() is broken, all

tests could fail.

If glReadPixels() is slow, the conformance tests can take a long time to run. Furthermore, since all tests run at all path levels on all available pixel formats and visuals, it could take several days of serial compute time to run the entire test suite.

The conformance tests find many bugs. However, they don't guarantee a bug−free implementation. An implementation that passes the full suite of conformance tests might still be so buggy that many applications won't be able to run.

3 GLUT

3.010 What is GLUT? How is it different from OpenGL?

Because OpenGL doesn't provide routines for interfacing with a windowing system or input devices, an application must use a variety of other platform−specific routines for this purpose. The result is nonportable code.

Furthermore, these platform−specific routines tend to be full−featured, which complicates construction of small programs and simple demos.

GLUT is a library that addresses these issues by providing a platform−independent interface to window management, menus, and input devices in a simple and elegant manner. Using GLUT comes at the price of some flexibility.

A large amount of information on GLUT is at the GLUT FAQ.

3.015 Where can I get GLUT?

The GLUT FAQ has the GLUT source code tree for download.

The GLUT FTP archive has several useful files, including specs, FAQs, and zips of current and older versions.

Nate Robins web page has the most recent GLUT, v3.7.5,

3.020 Should I use GLUT?

Your application might need to do things that GLUT doesn't allow, or it may need to use platform−specific libraries to accomplish nongraphical tasks. In this case, consider not using GLUT for your application's windowing and input needs, and instead use platform−specific libraries .

Ask yourself the following questions:

- Will my application run only on one platform?
- Do I need to use more than one rendering context?
- Do I need to share display lists or texture objects between rendering contexts?
- Do I need to use input devices that GLUT doesn't provide an interface for?
- Do I need to use platform−specific libraries for other tasks, such as sound or text?

If you answered yes to any of these questions, you need to evaluate whether GLUT is the right choice for your application.

3.025 The GLUT source code license is very restrictive. Is there an alternative?

Yes. See the freeglut initiative. It claims to be a 100% compatible replacement for GLUT 3.7.

3.027 Why does glutTimerFunc() only execute my callback once?

GLUT's function for starting a timer and specifying a timer callback, glutTimerFunc(), is specified to execute the callback after the elapsed time, then delete the timer. This is the way timers work in several APIs.

Often, repeated callbacks are useful. Implement this by resetting the timer within your callback routine. For example, the following timer callback routine resets the timer for repeated callbacks:

```
static void timerCallback (int value)
{
   /* Do timer processing */
   /* maybe glutPostRedisplay(), if necessary */

   /* call back again after elapsedUSecs have passed */
   glutTimerFunc (elapsedUSecs, timerCallback, value);
}
```

3.030 I need to set up different tasks for left and right mouse button motion. However, I can only set one glutMotionFunc() callback, which doesn't pass the button as a parameter.

You can easily set up different tasks depending on the state of the SHIFT, ALT, and CTRL keys by checking their state with glutGetModifiers().

To set up different tasks for the left and right mouse buttons, you need to swap the motion function depending on which mouse button is in use. You can do this with a mouse function callback that you set with glutMouseFunc(). The first parameter to this routine will indicate which button caused the event (GLUT_LEFT, GLUT_MIDDLE, or GLUT_RIGHT). The second parameter indicates the button state (GLUT_UP or GLUT_DOWN).

To illustrate, here's an example glutMouseFunc() callback routine:

```
/* Declarations for our motion functions */
static void leftMotion (int x, int y);
static void rightMotion (int x, int y);
static void mouseCallback (int mouse, int state, int x, int y)

{
   if (state==GLUT_DOWN) {
      /* A button is being pressed. Set the correct motion function */
      if (button==GLUT_LEFT)
         glutMotionFunc (leftMotion);
      else if (button==GLUT_RIGHT)
         glutMotionFunc (rightButton);
    }
}
```

3.040 How does GLUT do…?

It is often desirable to find out how glut creates windows, handles input devices, displays menus, or any of a number of other tasks. The best way to find out how GLUT does something is to download the GLUT source and see how it is written.

3.050 How can I perform animations with GLUT?

GLUT allows your application to specify a callback routine for rendering a frame. You can

force executing this routine by calling glutPostRedisplay() from another callback routine, and returning control to glutMainLoop().

To create an animation that runs as fast as possible, you need to set an idle callback with glutIdleFunc(). The callback you pass as a parameter will be executed by glutMainLoop() whenever nothing else is happening. From this callback, you call glutPostRedisplay().

To create a timed animation, use glutTimerFunc() instead of glutIdleFunc(). glutTimerFunc() will call your callback only after the specified time elapses. This callback disables itself, so for continuous updates, your callback must call both glutPostRedisplay(), then glutTimerFunc() again to reset the timer.

3.060 Is it possible to change a window's size *after* it's opened (i.e., after i called glutInitWindowSize(); and glutCreateWindow();)?

Once your code enters the glutMainLoop() and one of your callback routines is called, you can call glutReshapeWindow(int width, int height).

Note that glutReshapeWindow() doesn't instantly resize your window. It merely sends a message to GLUT to resize the window. This message is processed once you return to glutMainLoop().

3.070 I have a GLUT program that allocates memory at startup. How do I deallocate this memory when the program exits?

If the user exits your program through some input that you can catch, such as a key press or menu selection, the answer is trivial. Simply free the resources in the appropriate input event handler.

Usually, this question comes up because the user has killed the program through window frame controls, such as the Microsoft Windows Close Window icon in the upper right corner of the title bar. In this case, your program won't get a GLUT event indicating the program is exiting. In fact, when the window is destroyed, glutMainLoop() simply calls exit(0).

For simple resources such as memory deallocation, this should not be a problem. The OS will free any memory that the process was using.

Of greater concern is prompting the user to save work or flushing data held in software buffers to files.

When using C++, the simplest solution to this problem is to wrap your GLUT application inside of a C++ class and create it with global scope. The C++ language guarantees that the class' destructor is called when the object goes out of scope.

Another option is to use the ANSI C/C++ atexit() call to specify the address of a function to execute when the program exits. You need to declare your buffers and data pointers with global scope so they're acccessible to the atexit() callback routine. More information can be found in any ANSI C/C++ reference. atexit() is only available with C/C++.

One final option is to hack the GLUT source, and add an explicit callback to your code when glutMainLoop() catches the destroy window event/message. This is distasteful, for it means

you must now include the entire hacked glutMainLoop() function in your application.

3.080 How can I make my GLUT program detect that the user has closed the window?

The same way as the previous section 3.070 shows.

3.090 How can I make glutMainLoop() return to my calling program?

glutMainLoop() isn't designed to return to the calling routine. GLUT was designed around the idea of an event−driven application, with the exit method being captured through an input event callback routine, such as a GLUT menu or keyboard callback handler.

If you insist on returning to your program from glutMainLoop(), there is only one way to do so. You need to download the GLUT source and hack gluMainLoop() to do what you want it to. Then compile and link into your program this hacked version of glutMainLoop(). Steve Baker has a Web site with the details on how to hack glutMainLoop() to eliminate this problem.

3.100 How do I get rid of the console window in a Windows GLUT application?

With Visual C++ 6.0, go to the Project menu, Settings… dialog. Select the Link tab. In the Project options edit box, add /SUBSYSTEM:WINDOWS /ENTRY:mainCRTStartup to the end of the present text. Link options are similar for other Windows compilers.

3.110 My GLUT question isn't answered here. Where can I get more info?

The GLUT FAQ is an excellent source of information on GLUT.

4 GLU

4.010 What is GLU? How is it different from OpenGL?

If you think of OpenGL as a low−level 3D graphics library, think of GLU as adding some higher−level functionality not provided by OpenGL. Some of GLU's features include:

- Scaling of 2D images and creation of mipmap pyramids
- Transformation of object coordinates into device coordinates and vice versa
- Support for NURBS surfaces
- Support for tessellation of concave or bow tie polygonal primitives
- Specialty transformation matrices for creating perspective and orthographic projections, positioning a camera, and selection/picking
- Rendering of disk, cylinder, and sphere primitives
- Interpreting OpenGL error values as ASCII text

The best source of information on GLU is the OpenGL red and blue books and the GLU specification, which you can obtain from the OpenGL org Web page.

4.020 How does GLU render sphere, cylinder, and disk primitives?

There is nothing special about how GLU generates these primitives. You can easily write routines that do what GLU does. You can also download the Mesa source, which contains a GLU distribution, and see what these routines are doing.

The GLU routines approximate the specified primitive using normal OpenGL primitives, such as quad strips and triangle fans. The surface is approximated according to user parameters. The vertices are generated using calls to the sinf() and cosf() math library functions.

If you are interested in rendering cylinders and tubes, you'll want to examine the GLE library. GLE comes as part of the GLUT distribution.

4.030 How does gluPickMatrix() work?

It simply translates and scales so that the specified pick region fills the viewport. When specified on the projection matrix stack, prior to multiplying on a normal projection matrix (such as gluPerspective(), glFrustum(), glOrtho(), or gluOrtho2D()), the result is that the view volume is constrained to the pick region. This way only primitives that intersect the pick region will fall into the view volume. When glRenderMode() is set to GL_SELECT, these primitives will be returned.

4.040 How do I use GLU tessellation routines?

GLU provides tessellation routines to let you render concave polygons, self−intersecting polygons, and polygons with holes. The tessellation routines break these complex primitives up into (possibly groups of) simpler, convex primitives that can be rendered by the OpenGL API. This is done by providing the data of the simpler primitives to your application from callback routines that your application must provide. Your app can then send the data to OpenGL using normal API calls.

An example program is available in the GLUT distribution under progs/redbook/tess.c. (Download the GLUT distribution).

The usual steps for using tessellation routines are:

1. Allocate a new GLU tessellation object:

```
GLUtesselator *tess = gluNewTess();
```

2. Assign callbacks for use with this tessellation object:

```
gluTessCallback (tess, GLU_TESS_BEGIN, tcbBegin);
gluTessCallback (tess, GLU_TESS_VERTEX, tcbVertex);
gluTessCallback (tess, GLU_TESS_END, tcbEnd);
```

2a. If your primitive is self−intersecting, you must also specify a callback to create new vertices:

```
gluTessCallback (tess, GLU_TESS_COMBINE, tcbCombine);
```

3. Send the complex primitive data to GLU:

```
// Assumes:
//    GLdouble data[numVerts][3];
// ...and assumes the array has been filled with 3D vertex data.

gluTessBeginPolygon (tess, NULL);
gluTessBeginContour (tess);
for (i=0; i<sizeof(data)/(sizeof(GLdouble)*3);i++)
   gluTessVertex (tess, data[i], data[i]);
gluTessEndContour (tess);
gluEndPolygon (tess);
```

4. In your callback routines, make the appropriate OpenGL calls:

```
void tcbBegin (GLenum prim);
{
   glBegin (prim);
}

void tcbVertex (void *data)
{
   glVertex3dv ((GLdouble *)data);
}

void tcbEnd ();
{
   glEnd ();
}

void tcbCombine (GLdouble c[3], void *d[4], GLfloat w[4], void **out)
{
   GLdouble *nv = (GLdouble *) malloc(sizeof(GLdouble)*3);

   nv[0] = c[0];
   nv[1] = c[1];
   nv[2] = c[2];
   *out = nv;
}
```

The above list of steps and code segments is a bare−bones example and is not intended to demonstrate the full capabilities of the tessellation routines. By providing application−specific data as parameters to gluTessBeginPolygon() and gluTessVertex() and handling the data in the appropriate callback routines, your application can color and texture map these primitives as it would a normal OpenGL primitive.

4.050 Why aren't my tessellation callback routines called?

Normally your tessellation callback routines are executed when you call gluEndPolygon(). If they are not being called, an error has occurred. Typically this is caused when you haven't defined a GLU_TESS_COMBINE* callback for a self−intersecting primitive.

You might try defining a callback for GLU_TESS_ERROR to see if it's called.

4.060 How do I use GLU NURBS routines?

The GLU NURBS interface converts the B−Spline basis control points into Bezier basis equivalents and calls directly to the OpenGL Evaluator routines to render the surface.

An example program is available in the GLUT distribution under progs/redbook/surface.c. (Download the GLUT distribution).

4.070 How do I use gluProject() and gluUnProject()?

Both routines take a ModelView matrix, Projection matrix, and OpenGL Viewport as parameters.

gluProject() also takes an XYZ−object space coordinate. It returns the transformed XYZ window (or device) coordinate equivalent.

gluUnProject() does the opposite. It takes an XYZ window coordinate and returns the back−transformed XYZ object coordinate equivalent.

The concept of window space Z is often confusing. It's the depth buffer value expressed as a GLdouble in the range 0.0 to 1.0. Assuming a default glDepthRange(), a window coordinate with a Z value of 0.0 corresponds to an eye coordinate located on the *zNear* clipping plane. Similarly, a window space Z value of 1.0 corresponds to an eye space coordinate located on the *zFar* plane. You can obtain any window space Z value by reading the depth buffer with glReadPixels().

5 Microsoft Windows Specifics

5.010 What's a good source for Win32 OpenGL programming information?

Samuel Paik has created a large repository of links to OpenGL information on Microsoft Web sites.

See Blaine Hodge's web page. Be aware that some examples on this page use the AUX library, which is not recommended.

5.020 I'm looking for a Wintel OpenGL card in a specific price range, any suggestions?

The consumer−level 3D graphics marketplace moves fast. Any information placed in this FAQ would be soon outdated.

You might post a query on this topic to the comp.graphics.api.opengl newsgroup, or one of the many newsgroups devoted to Wintel−based 3D games. You might also do a Web search.

Tom's Hardware Guide and Fast Graphics have a lot of information on current graphics cards.

5.030 How do I enable and disable hardware rendering on a Wintel card?

Currently, OpenGL doesn't contain a switch to enable or disable hardware acceleration. Some vendors might provide this capability with an environment variable or software switch.

If you install your graphics card, but don't see hardware accelerated rendering check for the following:

- Did you install the device driver / OpenGL Installable Client Driver (ICD)? (How do I do that?)
- Is your desktop in a supported color depth? (Usually 16− and 32−bit color are accelerated. See your device vendor for details.)
- Did your application select an accelerated pixel format?

You might also have acceleration problems if you're trying to set up a multimonitor configuration. Hardware accelerated rendering might not be supported on all (or any) devices in this configuration.

To force software rendering from your application, choose a pixel format that is not hardware accelerated. To do this, you can not use ChoosePixelFormat(), which always selects a hardware accelerated pixel format when one is available. Instead, use DescribePixelFormat() to iterate through the list of available pixel formats. Any format with the PFD_GENERIC_FORMAT attribute bit set will not be hardware accelerated.

Ron Fosner has a source code snippet that shows how to select a software−only pixel format, and how to select a pixel format based on other weighting criteria.

An example of iterating over available pixel formats can be found here.

A less tasteful method to disable hardware acceleration is to move or rename your OpenGL

ICD.

Also, check your device's documentation to see if your device driver supports disabling hardware acceleration by a dialog box.

5.040 How do I know my program is using hardware acceleration on a Wintel card?

OpenGL doesn't provide a direct query to determine hardware acceleration usage. However, this can usually be inferred by using indirect methods.

If you are using the Win32 interface (as opposed to GLUT), call DescribePixelFormat() and check the returned dwFlags bitfield. If PFD_GENERIC_ACCELERATED is clear and PFD_GENERIC_FORMAT is set, then the pixel format is only supported by the generic implementation. Hardware acceleration is not possible for this format. For hardware acceleration, you need to choose a different format.

If glGetString(GL_VENDOR) returns something other than "Microsoft Corporation", it means you're using the board's ICD. If it returns "Microsoft Corporation", this implies you chose a pixel format that your device can't accelerate. However, glGetString(GL_VENDOR) also returns this if your device has an MCD instead of an ICD, which means you might still be hardware accelerated in this case.

Another way to check for hardware acceleration is to temporarily remove or rename the ICD, so it can't be loaded. If performance drops, it means you were hardware accelerated before. Don't forget to restore the ICD to its original location or name. (To find your ICD file name, run the regedit utility and search for a key named "OpenGLdrivers".)

You can also gather performance data by rendering into the back buffer and comparing the results against known performance statistics for your device. This method is particularly useful for devices that revert to software rendering for some state combinations or OpenGL features. See the section on performance for more information.

5.050 Where can I get the OpenGL ICD for a Wintel card?

If your device supports OpenGL, the manufacturer should provide an ICD (commonly referred to as the device driver) for it. After you install the ICD, your OpenGL application can use the device's hardware capabilities.

If your device didn't come with an ICD on disk, you'll need to check the manufacturer's Web page to see where you can download the latest drivers. The chip manufacturer will probably have a more current ICD than the board manufacturer. Find the device driver download page, get the latest package for your device, and install it per the instructions provided.

Check Reactor Critical for nVidia device drivers. They often have more current and better performing OpenGL device drivers than nVidia makes available from their web page.

GLsetup, a free utility, is available. According to the GLsetup Web page, it "detects a user's 3D graphics hardware and installs the correct device drivers." Windows 2000 device drivers might not be supported. You can get it from http://www.glsetup.com.

5.060 I'm using a Wintel card, and an OpenGL feature doesn't seem to work. What's going on?

It could simply be a bug in your code. However, if the same code works fine on another OpenGL implementation, this implies the problem is in your graphics device or its ICD. See the previous question for information on obtaining the latest ICD for your device.

5.070 Can I use OpenGL with DirectDraw?

Moxing OpenGL rendering calls with rendering calls from other APIs (such as DirectDraw) in the same window won't work on some drivers, and is therefore unportable. I don't recommended it.

5.080 Can I use use DirectDraw to change the screen resolution or desktop pixel depth?

You can create a window and use DirectDraw to change the display resolution and/or pixel depth. Then, get the window's DC and create an OpenGL context from it. This is known to work on some devices.

While we're on the subject, Microsoft doesn't require, and consequently does not test for, the ability to render OpenGL into a DirectDraw surface. Just because you can get a surface's DC does not mean that OpenGL rendering is supported. Always check for error returns when creating contexts or maiing them current.

5.090 My card supports OpenGL, but I don't get acceleration regardless of which pixel format I use.

Are you in 8bpp? There are few 3D accelerators for PCs that support acceleration in 8bpp.

5.100 How do I get hardware acceleration?

The pixel format selects hardware acceleration. Pay attention to the flags GENERIC_FORMAT and GENERIC_ACCELERATED. You want both of them on if you're using a 3D−DDI or an MCD and neither on if you are using an ICD. You may have to iterate using DescribePixelFormat() instead of only using ChoosePixelFormat().

5.110 Why doesn't OpenGL hardware acceleration work with multiple monitors?

In Windows 98, Microsoft decided to disable OpenGL hardware acceleration when multiple monitors are enabled. In Windows NT 4.0, some drivers support multiple monitors when using identical (or nearly identical) cards. I don't believe multiple monitors and hardware accelerated OpenGL work with different types of cards. I don't know the story with Windows 2000, but it's likely to be similar to Windows NT 4.0.

5.120 Why does my MFC window flash, even though I'm using double buffering?

Your view class should have an OnEraseBkgnd() event handler. You can eliminate flashing by overriding this function and returning TRUE. This tells Windows that you've cleared the window. Of course, you didn't really clear the window. However, overriding the function keeps Microsoft from trying to do it for you, and should prevent flashing.

5.121 Why does my double buffered window appear incomplete or contain black stripes?

This is a problem with MS OpenGL. The bug is in the generic code, or possibly in MS Windows itself, because it occurs even with pure software rendering.

Microsoft's product support page contains information on this issue: Clipping Problems with Generic Implementation of OpenGL for Windows 2000

To work around the bug, try one of these two methods:

- Create the OpenGL drawing window, but don't make it visible immediately. Get the screen size and set the window's size to be the same as the screen. Now set the pixel format and create the HGLRC. Set the window's size back to whatever it should be and make the window visible. This hack is invisible to the user, but doesn't always work.
- When the window is resized larger, destroy and re−create the window. This is really ugly and visible to the user, but it seems to always work.

5.130 What's the difference between opengl.dll and opengl32.dll?

According to *OpenGL Reference Manual* editor Dave Shreiner:

"Unless there's an absolutely compelling reason ... I really would suggest using opengl32.dll, and letting the old opengl.dll thing die.

"opengl.dll comes from the now totally unsupported OpenGL for Windows release of OpenGL for Microsoft Windows by SGI. We stopped supporting that release over two years −− like no one ever touches the code. ...

"Now, why use opengl32.dll? For the most part, SGI provides Microsoft with the ICD kit, sample drivers, and software OpenGL implementation that you find there. Its really the same code base (with fixes and new features) as the opengl.dll, its only that we got Microsoft to ship and support it (in a manner of speaking)."

More information on linking with opengl.dll can be found here.

5.140 Should I use Direct3D or OpenGL?

A good comparison of the two can be found here.

5.150 What do I need to know to use OpenGL with MFC?

You need to be familiar with both OpenGL and the Microsoft Foundation Class (MFC). An online MFC reference is available, the MFC FAQ. You don't need to be an MFC guru to add OpenGL to an MFC application. Familiarity with C++ can make mastering MFC easier, and the more you know about MFC, the more you can concentrate on your OpenGL code. If you have only a rudimentary knowledge of MFC, look at the downloadable source code example below, and look at the steps necessary to recreate it.

Joel Parris' OpenGL/MFC web site contains lots of useful information.

Samuel Paik's repository of links to OpenGL information on Microsoft Web sites also has information on using OpenGL and MFC.

Here's a list of books that might be helpful.

OpenGL Programming for Windows 95 and Windows NT, by Ron Fosner. This is also known as the white book. It contains good information on using OpenGL in Microsoft Windows. Much of the information in it can be found on the MSDN Web site, but the book presents the information in a more logical and easily digestable format, and comes with good demos.

Opengl Superbible: The Complete Guide to Opengl Programming for Windows NT and Windows 95, by Richard S. Wright and Michael Sweet. This book contains a chapter on OpenGL and MFC.

MFC Programming with Visual C++ 6 Unleashed, by David White, et al. The book contains a short chapter on OpenGL and focuses more on DirectX.

5.160 How can I use OpenGL with MFC?

To add OpenGL to an MFC application, you need to do at least the following:

- Add glu32.lib opengl32.lib to the list of object/library modules to link with.
- When your View class's OnInitialUpdate() function is called, set the pixel format and create a rendering context as you would for a Win32 application.
- Render your OpenGL scene when the View needs to be updated, or add a Run message handler to your Application class that updates when idle.

You can render OpenGL into any CWnd object, including frame windows, dialog boxes, and controls.

Download this example, which demonstrates OpenGL in a CStatic form control. This code uses a CGlView class that takes any CWnd as a parameter to its constructor. Rather than create a View derived from a CFormView, you could just as easily create an SDI application, and pass "this" (an instantiation of a CView) as a parameter to the constructor. Follow these steps to recreate this sample code using Microsoft Visual C++ v6.0:

1. If you haven't done so already, download the example. You'll need to borrow code from it in the steps that follow.
2. Create an MFC application using the AppWizard. Use defaults, except derive your View class from a CFormView. The project will open in the resource editor. Add a FORM control to the open CFormView. Call it IDC_OPENGLWIN.
3. Select Project−>Settings...−>Link, and add glu32.lib opengl32.lib to the list of objects/library modules.
4. Select Project−>Add To Peoject−>Files... and add the CGlView.cpp OpenGL view class source file from the above example code.
5. From the class view, right click your application's View class and select Add Member Variable... Set the variable type to CGlView *, the name to m_pclGLView, and the access to Private.
6. In your application's View class header file, add #include "CGlView.h" just before the class definition.
7. Find the global declaration of "theApp". Immediately after this declaration, add two new global variables:

```
CGlView *g_pclGLView = NULL;
MSG msg;
```

- In the wizard bar, set the application's View class, set the filter to All Class Members, and select the OnInitialUpdate member function.
- For the CGlView class to work, it needs a CWnd to initialize OpenGL for that window. For this example, our CWnd is the CStatic FORM control we added in step 1. After the existing code in this function, add the following:

```
CStatic pclStatic = (CStatic *)GetDlgItem(IDC_OPENGLWIN);
m_pclGLView = new CGlView(pclStatic);
```

- Open the class wizard with View−>ClassWizard. From the message map tab, select your project's Application class. Add a function handler for the Run message. Replace the generated code with the Run message handler from the downloaded example.

5.170 Is OpenGL inherently slower when used with MFC?

Nothing in MFC guarantees a slow−running OpenGL application. However, some poorly written MFC applications might run slowly. This is a possibility in any development environment and is not specific to OpenGL. Here are some things to look out for:

1. Build the application as Release instead of Debug. Disable the TRACE debugging feature.
2. Avoid MFC classes such as CArray, CMap, and CList that perform inefficient data copies.
3. You may be able to improve performance by avoiding the WM_PAINT message. See the question above for example source that does this.
4. MFC classes are general purpose. For maximum performance, write a tuned implementation of an MFC class.
5. Use standard efficient programming techniques such as avoiding redundant calls, etc.

5.180 Where can I find MFC examples?

This FAQ contains an example.

Alan Oursland, Using OpenGL in Visual C++ Version 4.x, DevCentral Learning Center, http://devcentral.iftech.com/learning/tutorials/mfc−win32/opengl/. This is good but dated. It will get you started with a SDI MFC OpenGL application.

Mahesh Venkitachalam, OpenGL Code, http://home.att.net/~bighesh/ogl.html. Mahesh presents OpenGL in a no application wizard, minimal MFC program along with some OpenGL techniques.

Roman Podobedov, Skeleton of OpenGL program for Windows (MFC). http://madli.ut.ee/~romka/opengl/demos/win32_eng.htm. This is a minimal MFC program with no controls or application wizard.

Paul Martz, Generating Random Fractal Terrain. http://www.gameprogrammer.com/fractal.html. This is a good example of the MFC SDI approach. However, the primary focus of the example is terrain, to which OpenGL and MFC take a back seat.

[5] Pierre Alliez, Starting OpenGL in a Dialog. http://codeguru.earthweb.com/opengl/texture_mapping.shtml.

Pierre Alliez, Starting Rendering Modes. http://www.codeguru.com/opengl/start.shtml. This is a splitter window example.

Pierre Alliez, How to snap an OpenGL client and send it to the clipboard, http://codeguru.earthweb.com/opengl/snap.shtml.

Pierre Alliez, A small VRML viewer using OpenGL and MFC. http://www.codeproject.com/opengl/wrl_viewer.asp.

Uwe Kotyczka, OpenGLSample.zip, http://www.virtue.nu/kotyczka/opengl_en.html. This rather large and impressive MFC contribution demonstrates, multiple OpenGL views, rubber banding, color ramp, mouse trackball type control, OpenGL printing, etc., in a MFC MDI and SDI framework. This was built with VC++ 6.0 (SP4) .

5.190 What do I need to know about mixing WGL and GDI calls?

On the Win32 platform a number of platform specific function calls are duplicated in the OpenGL ICD mechanism and the GDI. This may cause confusion as they appear to be functionally identical, the only difference being whether wgl precedes the rest of the function name. To ensure correct operation of OpenGL use ChoosePixelformat, DescribePixelformat, GetPixelformat, SetPixelformat, and SwapBuffers, instead of the wgl equivalents, wglChoosePixelformat, wglDescribePixelformat, wglGetPixelformat, wglSetPixelformat, and wglSwapBuffers. In all other cases use the wgl function where available. Using the five wgl functions is only of interest to developers run−time linking to an OpenGL driver. Not using the functions as described may result in a black OpenGL window, or a correctly functioning application in Windows 9x that produces a black OpenGL window on Windows NT/2000.

5.200 Why does my code produce a black screen under Windows NT or 2000 but run fine under 9x?

Incorrect mixing of GDI and wgl functions may result in OpenGL functioning correctly under Windows 95 and not functioning correctly under Windows NT/2000. Incorrect functioning will result in a black OpenGL window under Windows NT/2000.

5.210 How do I properly use WGL functions?

As described in section 5.190, ChoosePixelformat, DescribePixelformat, GetPixelformat, SetPixelformat, and SwapBuffers, are used when going through the OpenGL ICD mechanism. wglChoosePixelformat, wglDescribePixelformat, wglGetPixelformat, wglSetPixelformat, and wglSwapBuffers, are used when run−time linking to the OpenGL driver. The wgl functions specific to outline and bitmap fonts require special attention only by developers linking directly to the OpenGL driver. All other developers should use wglUseFontBitmaps and wglUseFontOutlines as per Microsoft platform documentation. wglUseFontBitmaps and wglUseFontOutlines come in two flavours, depending on whether a unicode or non−unicode platform is being targeted. When run−time linking use wglUseFontBitmapsW and wglUseFontOutlinesW for unicode platforms. wglUseFontBitmapsA and wglUseFontOutlinesA for non−unicode platforms. All other wgl functions may be used freely as per Microsoft platform documentation.

Charles E. Hardwidge has tutorial articles and examples for download that address these issues. The idea is to use WGL, GDI, and OpenGL functions such that the Microsoft

OpenGL ICD mechanism isn't assumed.

6 Windows, Buffers, and Rendering Contexts

6.010 How do I use overlay planes?

Overlays planes allow layered rendering. They support a transparent color to let rendering in underlying planes show through. Any combination of overlay and underlay planes are possible, but a typical implementation consists of a single overlay plane along with the main framebuffer plane. It's common for overlay planes to be available only in color index mode, though RGB overlays are available on some devices. The transparent color is normally (0,0,0) for RGB overlays and index 0 for color index overlays. Rendering to the overlay plane is non−destructive to the main plane and vice versa.

How you access overlay planes depends on the windowing system interface. While GLUT 3.7 has entry points defined for the use of overlays, currently the entry points are disabled. To use overlays, your application needs to use WGL, GLX, or another platform−specific interface.

For both WGL and GLX, the basic idea is the same: Create two rendering contexts, one for the main plane and another for the overlay planes. Once they're created, make either context current, depending on whether your application needs to render to the overlay or the main plane.

In WGL, use ChoosePixelFormat() to select a pixel format with an overlay or underlay plane. When your application calls this function, use the bReserved field of the PIXELFORMATDESCRIPTOR to indicate the desired overlay or underlay plane. A value of 1 indicates one overlay plane. The iLayerType field needs to be set to PFD_MAIN_PLANE.

After setting the pixel format, create the rendering context for the main plane as usual, then create a second rendering context for the overlay plane as follows:

```
HGLRC hORC;
hORC = wglCreateLayerContext (hDC, a);
```

Check the return value the same way you would for a call to wglCreateContext(). A value of NULL indicates failure.

In GLX, use glXChooseVisual() to obtain a list of visuals with overlays. Add GLX_LEVEL to the attribute list, followed by the fullword indication of the desired level. Positive values indicate overlay; negative values indicate underlay. A value of 0 indicates the main plane, which is the default. A typical application will call glXChooseVisual() twice, once with GLX_LEVEL set to 0, and again with GLX_LEVEL set to 1. After each call, choose one of the returned visuals to create a rendering context.

When your application can't find a pixel format or visual that supports overlay, there are two common causes. Overlay might not be available on your platform, or you could be asking for an RGB overlay when only color index is available.

7 Interacting with the Window System, Operating System, and Input Devices

7.010 How do I obtain the window width and height or screen max width and height?

To obtain the window size on Win32, use the following code:

```
RECT rect;
HWND hwnd;
GetClientRect(hwnd, &rect);
/* rect.top and rect.left will always be 0, 0, respectively.
   The width and height are in rect.right and rect.bottom. */
```

For the screen size in pixels on Win32:

```
int width = GetSystemMetrics(SM_CXSCREEN);
int height = GetSystemMetrics(SM_CYSCREEN);
```

To obtaining the screen and window width and height using GLUT:

```
int screenWidth, screenHeight, windowWidth, windowHeight;

screenWidth = glutGet(GLUT_SCREEN_WIDTH);
screenHeight = glutGet(GLUT_SCREEN_HEIGHT);
windowWidth = glutGet(GLUT_WINDOW_WIDTH);
windowHeight = glutGet(GLUT_WINDOW_HEIGHT);
```

7.020 What user interface system should I use?

Most user interface (UI) systems, such as Motif, are restricted to a subset of operating systems that support OpenGL. GLUT is available on a variety of windowing systems, and it supports hierarchical menus. However, this fills the UI requirements of only simple applications.

The GLUI toolkit implements buttons, checkboxes, radio buttons, and spinners, which are layered on top of GLUT. Therefore, this UI is window system independent. Go to http://www.cs.unc.edu/~rademach/glui/ for more details.

7.030 How can I use multiple monitors?

Many OpenGL implementations support multiple monitor configurations. These come in a variety of different flavors:

- One display is hardware accelerated, the rest are not.
- All heads are accelerated as long as OpenGL windows do not span display boundaries.
- As above, with support for OpenGL windows that span multiple displays.
- All of the above, with support for stereo.
- All of the above, with support for heterogeneous graphics cards vendors.

Some Macintosh configurations, such as the ATI Rage 128 Mobility, allow dual displays. However, hardware acceleration is disabled if the OpenGL window spans both displays. If

the window lies completely on one display or the other, full hardware acceleration is available.

Microsoft operating systems allow two OpenGL devices in a single system, but only the primary device is hardware accelerated. To work around this issue, many vendors have provided their own multiple monitor solutions, so that hardware acceleration is available on both displays.

3Dlabs supports multi−head on GVX1, GVX210 and GVX420. The latter two cards are a single AGP card with dual monitor output. The GVX1 supports one display per device, and comes in both AGP and PCI versions to support signle−AGP slot systems.

HP Visualize Center and Visualize Workgroup allow full hardware acceleration in windows spanning two or more displays under HP−UX. Visualize Center blends multiple projector displays seamlessly on a wall−sized screen. Stereo is supported to produce a completely immersive environment.

Matrox under Linux and Microsoft operating systems supports DualHead, digital flat panel, and TV out for the G400, DualHead for the G450, and multiple monitors for the G200 series. . In addition to supporting a single logical display, their Microsoft Windows device drivers also support a dual desktop mode.

The ***NVIDIA*** Quadro2 MXR and GeForce2 GTS devices both support two displays from a single card. Under Miscrosoft Windows operating systems, the Display Properties dialog features a TwinView tab that allows the user to configure the displays. In Vertical Span mode, a single logical desktop spans both monitors. In this configuration, an application that creates a window the size of the screen will automatically create a window that fills both monitors. No change to the application is required, and both displays are accelerated in hardware. NVIDIA Linux drivers do not currently support hardware accelerated rendering, according to the NVIDIA Linux FAQ.

All versions of ***Sun*** OpenGL for Solaris on SPARC support accelerated multihead configurations provided the display is OpenGL accelerated. With Solaris 8 and Sun OpenGL 1.2.1 an accelerated OpenGL window can span multiple heads provided the display devices are the same and the device is OpenGL accelerated. Sun OpenGL is available from: http://sun.com/software/graphics/opengl

8 Using Viewing and Camera Transforms, and gluLookAt()

8.010 How does the camera work in OpenGL?

As far as OpenGL is concerned, there is no camera. More specifically, the camera is always located at the eye space coordinate (0., 0., 0.). To give the appearance of moving the camera, your OpenGL application must move the scene with the inverse of the camera transformation.

8.020 How can I move my eye, or camera, in my scene?

OpenGL doesn't provide an interface to do this using a camera model. However, the GLU library provides the gluLookAt() function, which takes an eye position, a position to look at, and an up vector, all in object space coordinates. This function computes the inverse camera transform according to its parameters and multiplies it onto the current matrix stack.

8.030 Where should my camera go, the ModelView or Projection matrix?

The GL_PROJECTION matrix should contain only the projection transformation calls it needs to transform eye space coordinates into clip coordinates.

The GL_MODELVIEW matrix, as its name implies, should contain modeling and viewing transformations, which transform object space coordinates into eye space coordinates. Remember to place the camera transformations on the GL_MODELVIEW matrix and never on the GL_PROJECTION matrix.

Think of the projection matrix as describing the attributes of your camera, such as field of view, focal length, fish eye lens, etc. Think of the ModelView matrix as where you stand with the camera and the direction you point it.

The game dev FAQ has good information on these two matrices.

Read Steve Baker's article on projection abuse. This article is highly recommended and well−written. It's helped several new OpenGL programmers.

8.040 How do I implement a zoom operation?

A simple method for zooming is to use a uniform scale on the ModelView matrix. However, this often results in clipping by the *zNear* and *zFar* clipping planes if the model is scaled too large.

A better method is to restrict the width and height of the view volume in the Projection matrix.

For example, your program might maintain a zoom factor based on user input, which is a floating−point number. When set to a value of 1.0, no zooming takes place. Larger values result in greater zooming or a more restricted field of view, while smaller values cause the opposite to occur. Code to create this effect might look like:

```
static float zoomFactor; /* Global, if you want. Modified by user input. Initially 1.0 */

/* A routine for setting the projection matrix. May be called from a resize
   event handler in a typical application. Takes integer width and height
   dimensions of the drawing area. Creates a projection matrix with correct
   aspect ratio and zoom factor. */
void setProjectionMatrix (int width, int height)
{
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective (50.0*zoomFactor, (float)width/(float)height, zNear, zFar);
   /* ...Where 'zNear' and 'zFar' are up to you to fill in. */
}
```

Instead of gluPerspective(), your application might use glFrustum(). This gets tricky, because the *left, right, bottom,* and *top* parameters, along with the *zNear* plane distance, also affect the field of view. Assuming you desire to keep a constant *zNear* plane distance (a reasonable assumption), glFrustum() code might look like this:

```
glFrustum(left*zoomFactor, right*zoomFactor,
    bottom*zoomFactor, top*zoomFactor,
    zNear, zFar);
```

glOrtho() is similar.

8.050 Given the current ModelView matrix, how can I determine the object−space location of the camera?

The "camera" or viewpoint is at (0., 0., 0.) in eye space. When you turn this into a vector [0 0 0 1] and multiply it by the inverse of the ModelView matrix, the resulting vector is the object−space location of the camera.

OpenGL doesn't let you inquire (through a glGet* routine) the inverse of the ModelView matrix. You'll need to compute the inverse with your own code.

8.060 How do I make the camera "orbit" around a point in my scene?

You can simulate an orbit by translating/rotating the scene/object and leaving your camera in the same place. For example, to orbit an object placed somewhere on the Y axis, while continuously looking at the origin, you might do this:

```
gluLookAt(camera[0], camera[1], camera[2], /* look from camera XYZ */
               0, 0, 0, /* look at the origin */
               0, 1, 0); /* positive Y up vector */
glRotatef(orbitDegrees, 0.f, 1.f, 0.f);/* orbit the Y axis */
/* ...where orbitDegrees is derived from mouse motion */

glCallList(SCENE); /* draw the scene */
```

If you insist on physically orbiting the camera position, you'll need to transform the current camera position vector before using it in your viewing transformations.

In either event, I recommend you investigate gluLookAt() (if you aren't using this routine already).

8.070 How can I automatically calculate a view that displays my entire model? (I know the bounding sphere and up vector.)

The following is from a posting by Dave Shreiner on setting up a basic viewing system:

First, compute a bounding sphere for all objects in your scene. This should provide you with two bits of information: the center of the sphere (let (c.x, c.y, c.z) be that point) and its diameter (call it "diam").

Next, choose a value for the *zNear* clipping plane. General guidelines are to choose something larger than, but close to 1.0. So, let's say you set

```
zNear = 1.0;
zFar = zNear + diam;
```

Structure your matrix calls in this order (for an Orthographic projection):

```
GLdouble left = c.x − diam;
GLdouble right = c.x + diam;
GLdouble bottom c.y − diam;
GLdouble top = c.y + diam;

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(left, right, bottom, top, zNear, zFar);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

This approach should center your objects in the middle of the window and stretch them to fit (i.e., its assuming that you're using a window with aspect ratio = 1.0). If your window isn't square, compute *left*, *right*, *bottom*, and *top*, as above, and put in the following logic before the call to glOrtho():

```
GLdouble aspect = (GLdouble) windowWidth / windowHeight;

if ( aspect < 1.0 ) { // window taller than wide
   bottom /= aspect;
   top /= aspect;
} else {
   left *= aspect;
   right *= aspect;
}
```

The above code should position the objects in your scene appropriately. If you intend to manipulate (i.e. rotate, etc.), you need to add a viewing transform to it.

A typical viewing transform will go on the ModelView matrix and might look like this:

```
GluLookAt (0., 0., 2.*diam,
                c.x, c.y, c.z,
                0.0, 1.0, 0.0);
```

8.080 Why doesn't gluLookAt work?

This is usually caused by incorrect transformations.

Assuming you are using gluPerspective() on the Projection matrix stack with *zNear* and *zFar* as the third and fourth parameters, you need to set gluLookAt on the ModelView matrix stack, and pass parameters so your geometry falls between *zNear* and *zFar*.

It's usually best to experiment with a simple piece of code when you're trying to understand viewing transformations. Let's say you are trying to look at a unit sphere centered on the origin. You'll want to set up your transformations as follows:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(50.0, 1.0, 3.0, 7.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(0.0, 0.0, 5.0,
          0.0, 0.0, 0.0,
          0.0, 1.0, 0.0);
```

It's important to note how the Projection and ModelView transforms work together.

In this example, the Projection transform sets up a 50.0−degree field of view, with an aspect ratio of 1.0. The *zNear* clipping plane is 3.0 units in front of the eye, and the *zFar* clipping plane is 7.0 units in front of the eye. This leaves a Z volume distance of 4.0 units, ample room for a unit sphere.

The ModelView transform sets the eye position at (0.0, 0.0, 5.0), and the look−at point is the origin in the center of our unit sphere. Note that the eye position is 5.0 units away from the look at point. This is important, because a distance of 5.0 units in front of the eye is in the middle of the Z volume that the Projection transform defines. If the gluLookAt() call had placed the eye at (0.0, 0.0, 1.0), it would produce a distance of 1.0 to the origin. This isn't long enough to include the sphere in the view volume, and it would be clipped by the *zNear* clipping plane.

Similarly, if you place the eye at (0.0, 0.0, 10.0), the distance of 10.0 to the look at point will result in the unit sphere being 10.0 units away from the eye and far behind the *zFar* clipping plane placed at 7.0 units.

If this has confused you, read up on transformations in the OpenGL red book or OpenGL Specification. After you understand object coordinate space, eye coordinate space, and clip coordinate space, the above should become clear. Also, experiment with small test programs. If you're having trouble getting the correct transforms in your main application project, it can be educational to write a small piece of code that tries to reproduce the problem with simpler geometry.

8.090 How do I get a specified point (XYZ) to appear at the center of the scene?

gluLookAt() is the easiest way to do this. Simply set the X, Y, and Z values of your point as the fourth, fifth, and sixth parameters to gluLookAt().

8.100 I put my gluLookAt() call on my Projection matrix and now fog, lighting, and texture mapping don't work correctly. What happened?

Look at question 8.030 for an explanation of this problem.

8.110 How can I create a stereo view?

Paul Bourke has assembled information on stereo OpenGL viewing.

- 3D Stereo Rendering Using OpenGL
- Calculating Stereo Pairs
- Creating Anaglyphs using OpenGL

9 Transformations

9.001 I can't get transformations to work. Where can I learn more about matrices?

A thorough explanation of basic matrix math and linear algebra is beyond the scope of this FAQ. These concepts are taught in high school math classes in the United States.

If you understand the basics, but just get confused (a common problem even for the experienced!), read through Steve Baker's review of matrix concepts and his article on Euler angles.

Delphi code for performing basic vector, matrix, and quaternion operations can be found here.

9.005 Are OpenGL matrices column−major or row−major?

For programming purposes, OpenGL matrices are 16−value arrays with base vectors laid out contiguously in memory. The translation components occupy the 13th, 14th, and 15th elements of the 16−element matrix.

Column−major versus row−major is purely a notational convention. Note that post−multiplying with column−major matrices produces the same result as pre−multiplying with row−major matrices. The OpenGL Specification and the OpenGL Reference Manual both use column−major notation. You can use any notation, as long as it's clearly stated.

Sadly, the use of column−major format in the spec and blue book has resulted in endless confusion in the OpenGL programming community. Column−major notation suggests that matrices are not laid out in memory as a programmer would expect.

A summary of Usenet postings on the subject can be found here.

9.010 What are OpenGL coordinate units?

The short answer: Anything you want them to be.

Depending on the contents of your geometry database, it may be convenient for your application to treat one OpenGL coordinate unit as being equal to one millimeter or one parsec or anything in between (or larger or smaller).

OpenGL also lets you specify your geometry with coordinates of differing values. For example, you may find it convenient to model an airplane's controls in centimeters, its fuselage in meters, and a world to fly around in kilometers. OpenGL's ModelView matrix can then scale these different coordinate systems into the same eye coordinate space.

It's the application's responsibility to ensure that the Projection and ModelView matrices are constructed to provide an image that keeps the viewer at an appropriate distance, with an appropriate field of view, and keeps the *zNear* and *zFar* clipping planes at an appropriate range. An application that displays molecules in micron scale, for example, would probably not want to place the viewer at a distance of 10 feet with a 60 degree field of view.

9.011 How are coordinates transformed? What are the different coordinate spaces?

Object Coordinates are transformed by the ModelView matrix to produce Eye Coordinates.

Eye Coordinates are transformed by the Projection matrix to produce Clip Coordinates.

Clip Coordinate X, Y, and Z are divided by Clip Coordinate W to produce Normalized Device Coordinates.

Normalized Device Coordinates are scaled and translated by the viewport parameters to produce Window Coordinates.

Object coordinates are the raw coordinates you submit to OpenGL with a call to glVertex*() or glVertexPointer(). They represent the coordinates of your object or other geometry you want to render.

Many programmers use a World Coordinate system. Objects are often modeled in one coordinate system, then scaled, translated, and rotated into the world you're constructing. World Coordinates result from transforming Object Coordinates by the modelling transforms stored in the ModelView matrix. However, OpenGL has no concept of World Coordinates. World Coordinates are purely an application construct.

Eye Coordinates result from transforming Object Coordinates by the ModelView matrix. The ModelView matrix contains both modelling and viewing transformations that place the viewer at the origin with the view direction aligned with the negative Z axis.

Clip Coordinates result from transforming Eye Coordinates by the Projection matrix. Clip Coordinate space ranges from $-W_c$ to W_c in all three axes, where W_c is the Clip Coordinate W value. OpenGL clips all coordinates outside this range.

Perspective division performed on the Clip Coordinates produces Normalized Device Coordinates, ranging from −1 to 1 in all three axes.

Window Coordinates result from scaling and translating Normalized Device Coordinates by the viewport. The parameters to glViewport() and glDepthRange() control this transformation. With the viewport, you can map the Normalized Device Coordinate cube to any location in your window and depth buffer.

For more information, see the OpenGL Specification, Figure 2.6.

9.020 How do I transform only one object in my scene or give each object its own transform?

OpenGL provides matrix stacks specifically for this purpose. In this case, use the ModelView matrix stack.

A typical OpenGL application first sets the matrix mode with a call to glMatrixMode(GL_MODELVIEW) and loads a viewing transform, perhaps with a call to gluLookAt().More information is available on gluLookAt().

Then the code renders each object in the scene with its own transformation by wrapping the rendering with calls to glPushMatrix() and glPopMatrix(). For example:

```
glPushMatrix();
```

```
glRotatef(90., 1., 0., 0.);
gluCylinder(quad,1,1,2,36,12);
glPopMatrix();
```

The above code renders a cylinder rotated 90 degrees around the X−axis. The ModelView matrix is restored to its previous value after the glPopMatrix() call. Similar call sequences can render subsequent objects in the scene.

9.030 How do I draw 2D controls over my 3D rendering?

> The basic strategy is to set up a 2D projection for drawing controls. You can do this either on top of your 3D rendering or in overlay planes. If you do so on top of a 3D rendering, you'll need to redraw the controls at the end of every frame (immediately before swapping buffers). If you draw into the overlay planes, you only need to redraw the controls if you're updating them.
>
> To set up a 2D projection, you need to change the Projection matrix. Normally, it's convenient to set up the projection so one world coordinate unit is equal to one screen pixel, as follows:

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
gluOrtho2D (0, windowWidth, 0, windowHeight);
```

gluOrtho2D() sets up a Z range of −1 to 1, so you need to use one of the glVertex2*() functions to ensure your geometry isn't clipped by the *zNear* or *zFar* clipping planes.

Normally, the ModelView matrix is set to the identity when drawing 2D controls, though you may find it convenient to do otherwise (for example, you can draw repeated controls with interleaved translation matrices).

If exact pixelization is required, you might want to put a small translation in the ModelView matrix, as shown below:

```
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
glTranslatef (0.375, 0.375, 0.);
```

If you're drawing on top of a 3D−depth buffered image, you'll need to somehow disable depth testing while drawing your 2D geometry. You can do this by calling glDisable(GL_DEPTH_TEST) or glDepthFunc (GL_ALWAYS). Depending on your application, you might also simply clear the depth buffer before starting the 2D rendering. Finally, drawing all 2D geometry with a minimum Z coordinate is also a solution.

After the 2D projection is established as above, you can render normal OpenGL primitives to the screen, specifying their coordinates with XY pixel addresses (using OpenGL−centric screen coordinates, with (0,0) in the lower left).

9.040 How do I bypass OpenGL matrix transformations and send 2D coordinates directly for rasterization?

> There isn't a mode switch to disable OpenGL matrix transformations. However, if you set either or both matrices to the identity with a glLoadIdentity() call, typical OpenGL implementations are intelligent enough to know that an identity transformation is a no−op and will act accordingly.

More detailed information on using OpenGL as a rasterization−only API is in the OpenGL Game Developer’s FAQ.

9.050 What are the pros and cons of using absolute versus relative coordinates?

Some OpenGL applications may need to render the same object in multiple locations in a single scene. OpenGL lets you do this two ways:

1) Use “absolute coordinates". Maintain multiple copies of each object, each with its own unique set of vertices. You don't need to change the ModelView matrix to render the object at the desired location.

2) Use “relative coordinates". Keep only one copy of the object, and render it multiple times by pushing the ModelView matrix stack, setting the desired transform, sending the geometry, and popping the stack. Repeat these steps for each object.

In general, frequent changes to state, such as to the ModelView matrix, can negatively impact your application’s performance. OpenGL can process your geometry faster if you don't wrap each individual primitive in a lot of changes to the ModelView matrix.

However, sometimes you need to weigh this against the memory savings of replicating geometry. Let's say you define a doorknob with high approximation, such as 200 or 300 triangles, and you're modeling a house with 50 doors in it, all of which have the same doorknob. It's probably preferable to use a single doorknob display list, with multiple unique transform matrices, rather than use absolute coordinates with 10−15K triangles in memory.

As with many computing issues, it's a trade−off between processing time and memory that you'll need to make on a case−by−case basis.

9.060 How can I draw more than one view of the same scene?

You can draw two views into the same window by using the glViewport() call. Set glViewport() to the area that you want the first view, set your scene’s view, and render. Then set glViewport() to the area for the second view, again set your scene’s view, and render.

You need to be aware that some operations don't pay attention to the glViewport, such as SwapBuffers and glClear(). SwapBuffers always swaps the entire window. However, you can restrain glClear() to a rectangular window by using the scissor rectangle.

Your application might only allow different views in separate windows. If so, you need to perform a MakeCurrent operation between the two renderings. If the two windows share a context, you need to change the scene’s view as described above. This might not be necessary if your application uses separate contexts for each window.

9.070 How do I transform my objects around a fixed coordinate system rather than the object's local coordinate system?

If you rotate an object around its Y−axis, you'll find that the X− and Z−axes rotate with the object. A subsequent rotation around one of these axes rotates around the newly transformed axis and not the original axis. It's often desirable to perform transformations in a fixed coordinate system rather than the object’s local coordinate system.

The OpenGL Game Developer's FAQ contains information on using quaternions to store rotations, which may be useful in solving this problem.

The root cause of the problem is that OpenGL matrix operations postmultiply onto the matrix stack, thus causing transformations to occur in object space. To affect screen space transformations, you need to premultiply. OpenGL doesn't provide a mode switch for the order of matrix multiplication, so you need to premultiply by hand. An application might implement this by retrieving the current matrix after each frame. The application multiplies new transformations for the next frame on top of an identity matrix and multiplies the accumulated current transformations (from the last frame) onto those transformations using glMultMatrix().

You need to be aware that retrieving the ModelView matrix once per frame might have a detrimental impact on your application's performance. However, you need to benchmark this operation, because the performance will vary from one implementation to the next.

9.080 What are the pros and cons of using glFrustum() versus gluPerspective()? Why would I want to use one over the other?

glFrustum() and gluPerspective() both produce perspective projection matrices that you can use to transform from eye coordinate space to clip coordinate space. The primary difference between the two is that glFrustum() is more general and allows off−axis projections, while gluPerspective() only produces symmetrical (on−axis) projections. Indeed, you can use glFrustum() to implement gluPerspective(). However, aside from the layering of function calls that is a natural part of the GLU interface, there is no performance advantage to using matrices generated by glFrustum() over gluPerspective().

Since glFrustum() is more general than gluPerspective(), you can use it in cases when gluPerspective() can't be used. Some examples include projection shadows, tiled renderings, and stereo views.

Tiled rendering uses multiple off−axis projections to render different sections of a scene. The results are assembled into one large image array to produce the final image. This is often necessary when the desired dimensions of the final rendering exceed the OpenGL implementation's maximum viewport size.

In a stereo view, two renderings of the same scene are done with the view location slightly shifted. Since the view axis is right between the "eyes", each view must use a slightly off−axis projection to either side to achieve correct visual results.

9.085 How can I make a call to glFrustum() that matches my call to gluPerspective()?

The field of view (fov) of your glFrustum() call is:

fov*0.5 = arctan ((top−bottom)*0.5 / near)

Since *bottom* == −*top* for the symmetrical projection that gluPerspective() produces, then:

top = tan(fov*0.5) * near
bottom = −top

Note: fov must be in radians for the above formulae to work with the C math library. If you have comnputer your fov in degrees (as in the call to gluPerspective()), then calculate *top* as follows:

top = tan(fov*3.14159/360.0) * near

The *left* and *right* parameters are simply functions of the *top, bottom,* and *aspect:*

left = aspect * bottom
right = aspect * top

The *OpenGL Reference Manual* (where do I get this?) shows the matrices produced by both functions.

9.090 How do I draw a full−screen quad?

This question usually means, "How do I draw a quad that fills the entire OpenGL viewport?" There are many ways to do this.

The most straightforward method is to set the desired color, set both the Projection and ModelView matrices to the identity, and call glRectf() or draw an equivalent GL_QUADS primitive. Your rectangle or quad's Z value should be in the range of –1.0 to 1.0, with –1.0 mapping to the *zNear* clipping plane, and 1.0 to the *zFar* clipping plane.

As an example, here's how to draw a full−screen quad at the *zNear* clipping plane:

```
glMatrixMode (GL_MODELVIEW);
glPushMatrix ();
glLoadIdentity ();
glMatrixMode (GL_PROJECTION);
glPushMatrix ();
glLoadIdentity ();

glBegin (GL_QUADS);
glVertex3i (−1, −1, −1);
glVertex3i (1, −1, −1);
glVertex3i (1, 1, −1);
glVertex3i (−1, 1, −1);
glEnd ();

glPopMatrix ();
glMatrixMode (GL_MODELVIEW);
glPopMatrix ();
```

Your application might want the quad to have a maximum Z value, in which case 1 should be used for the Z value instead of −1.

When painting a full−screen quad, it might be useful to mask off some buffers so that only specified buffers are touched. For example, you might mask off the color buffer and set the depth function to GL_ALWAYS, so only the depth buffer is painted. Also, you can set masks to allow the stencil buffer to be set or any combination of buffers.

9.100 How can I find the screen coordinates for a given object−space coordinate?

You can use the GLU library gluProject() utility routine if you only need to find it for a few vertices. For a large number of coordinates, it can be more efficient to use the Feedback

mechanism.

To use gluProject(), you'll need to provide the ModelView matrix, projection matrix, viewport, and input object space coordinates. Screen space coordinates are returned for X, Y, and Z, with Z being normalized (0 <= Z <= 1).

9.110 How can I find the object−space coordinates for a pixel on the screen?

The GLU library provides the gluUnProject() function for this purpose.

You'll need to read the depth buffer to obtain the input screen coordinate Z value at the X,Y location of interest. This can be coded as follows:

```
GLdouble z;

glReadPixels (x, y, 1, 1, GL_DEPTH_COMPONENT, GL_DOUBLE, &z);
```

Note that *x* and *y* are OpenGL−centric with (0,0) in the lower−left corner.

You'll need to provide the screen space X, Y, and Z values as input to gluUnProject() with the ModelView matrix, Projection matrix, and viewport that were current at the time the specific pixel of interest was rendered.

9.120 How do I find the coordinates of a vertex transformed only by the ModelView matrix?

It's often useful to obtain the eye coordinate space value of a vertex (i.e., the object space vertex transformed by the ModelView matrix). You can obtain this by retrieving the current ModelView matrix and performing simple vector / matrix multiplication.

9.130 How do I calculate the object−space distance from the viewer to a given point?

Transform the point into eye−coordinate space by multiplying it by the ModelView matrix. Then simply calculate its distance from the origin. (If this doesn't work, you may have incorrectly placed the view transform on the Projection matrix stack.)

To retrieve the current Modelview matrix:

```
GLfloat m[16];
glGetFloatv (GL_MODELVIEW_MATRIX, m);
```

As with any OpenGL call, you must have a context current with a window or drawable in order for glGet*() function calls to work.

9.140 How do I keep my aspect ratio correct after a window resize?

It depends on how you are setting your projection matrix. In any case, you'll need to know the new dimensions (width and height) of your window. How to obtain these depends on which platform you're using. In GLUT, for example, the dimensions are passed as parameters to the reshape function callback.

The following assumes you're maintaining a viewport that's the same size as your window. If you are not, substitute viewportWidth and viewportHeight for windowWidth and

windowHeight.

If you're using gluPerspective() to set your Projection matrix, the second parameter controls the aspect ratio. When your program catches a window resize, you'll need to change your Projection matrix as follows:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(fov, (float)windowWidth/(float)windowHeight, zNear, zFar);
```

If you're using glFrustum(), the aspect ratio varies with the width of the view volume to the height of the view volume. You might maintain a 1:1 aspect ratio with the following window resize code:

```
float cx, halfWidth = windowWidth*0.5f;
float aspect = (float)windowWidth/(float)windowHeight;

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
/* cx is the eye space center of the zNear plane in X */
glFrustum(cx-halfWidth*aspect, cx+halfWidth*aspect, bottom, top, zNear, zFar);
```

glOrtho() and gluOrtho2D() are similar to glFrustum().

9.150 Can I make OpenGL use a left−handed coordinate space?

OpenGL doesn't have a mode switch to change from right− to left−handed coordinates. However, you can easily obtain a left−handed coordinate system by multiplying a negative Z scale onto the ModelView matrix. For example:

```
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
glScalef (1., 1., -1.);
/* multiply view transforms as usual... */
/* multiply model transforms as usual... */
```

9.160 How can I transform an object so that it points at or follows another object or point in my scene?

You need to construct a matrix that transforms from your object's local coordinate system into a coordinate system that faces in the desired direction. See this example code to see how this type of matrix is created.

If you merely want to render an object so that it always faces the viewer, you might consider simply rendering it in eye−coordinate space with the ModelView matrix set to the identity.

9.162 How can I transform an object with a given yaw, pitch, and roll?

The upper left 3x3 portion of a transformation matrix is composed of the new X, Y, and Z axes of the post−transformation coordinate space.

If the new transform is a roll, compute new local Y and X axes by rotating them "roll" degrees around the local Z axis. Do similar calculations if the transform is a pitch or yaw. Then simply construct your transformation matrix by inserting the new local X, Y, and Z axes into the upper left 3x3 portion of an identity matrix. This matrix can be passed as a parameter to glMultMatrix().

Further rotations should be computed around the new local axes. This will inevitably require rotation about an arbitrary axis, which can be confusing to inexperienced 3D programmers. This is a basic concept in linear algebra.

Many programmers apply all three transformations −− yaw, pitch, and roll −− at once as successive glRotate() calls about the X, Y, and Z axes. This has the disadvantage of creating gimbal lock, in which the result depends on the order of glRotate() calls.

9.170 How do I render a mirror?

Render your scene twice, once as it is reflected in the mirror, then once from the normal (non−reflected) view. Example code demonstrates this technique.

For axis−aligned mirrors, such as a mirror on the YZ plane, the reflected scene can be rendered with a simple scale and translate. Scale by −1.0 in the axis corresponding to the mirror's normal, and translate by twice the mirror's distance from the origin. Rendering the scene with these transforms in place will yield the scene reflected in the mirror. Use the matrix stack to restore the view transform to its previous value.

Next, clear the depth buffer with a call to glClear(GL_DEPTH_BUFFER_BIT). Then render the mirror. For a perfectly reflecting mirror, render into the depth buffer only. Real mirrors are not perfect reflectors, as they absorb some light. To create this effect, use blending to render a black mirror with an alpha of 0.05. glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA) is a good blending function for this purpose.

Finally, render the non−reflected scene. Since the entire reflected scene exists in the color buffer, and not just the portion of the reflected scene in the mirror, you will need to touch all pixels to overwrite areas of the reflected scene that should not be visible.

9.180 How can I do my own perspective scaling?

OpenGL multiplies your coordinates by the ModelView matrix, then by the Projection matrix to get clip coordinates. It then performs the perspective divide to obtain normalized device coordinates. It's the perspective division step that creates a perspective rendering, with geometry in the distance appearing smaller than the geometry in the foreground. The perspective division stage is accomplished by dividing your XYZ clipping coordinate values by the clipping coordinate W value, such as:

Xndc = Xcc/Wcc
Yndc = Ycc/Wcc
Zndc = Zcc/Wcc

To do your own perspective correction, you need to obtain the clipping coordinate W value. The feedback buffer provides homogenous coordinates with XYZ in device coordinates and W in clip coordinates. You might also glGetFloatv(GL_CURRENT_RASTER_POSITION,…) and the W value will again be in clipping coordinates, while XYZ are in device coordinates.

10 Clipping, Culling, and Visibility Testing

10.010 How do I tell if a vertex has been clipped or not?

You can use the OpenGL Feedback feature to determine if a vertex will be clipped or not. After you're in Feedback mode, simply send the vertex in question as a GL_POINTS primitive. Then switch back to GL_RENDER mode and check the size of the Feedback buffer. A size of zero indicates a clipped vertex.

Typically, OpenGL implementations don't provide a fast feedback mechanism. It might be faster to perform the clip test manually. To do so, construct six plane equations corresponding to the clip−coordinate view volume and transform them into object space by the current ModelView matrix. A point is clipped if it violates any of the six plane equations.

Here's a GLUT example that shows how to calculate the object−space view−volume planes and clip test bounding boxes against them.

Here is a tutorial titled Frustum Culling in OpenGL.

10.020 How do I perform occlusion or visibility testing?

OpenGL provides no direct support for determining whether a given primitive will be visible in a scene for a given viewpoint. At worst, an application will need to perform these tests manually. The previous question contains information on how to do this.

The code example from question 10.010 was combined with Nate Robins' excellent viewing tutorial to produce this view culling example code.

Higher−level APIs, such as Fahernheit Large Model, may provide this feature.

HP OpenGL platforms support an Occlusion Culling extension. To use this extension, enable the occlusion test, render some bounding geometry, and call glGetBooleanv() to obtain the visibility status of the geometry.

10.030 How do I render to a nonrectangular viewport?

OpenGL's stencil buffer can be used to mask the area outside of a non−rectangular viewport. With stencil enabled and stencil test appropriately set, rendering can then occur in the unmasked area. Typically an application will write the stencil mask once, and then render repeated frames into the unmasked area.

As with the depth buffer, an application must ask for a stencil buffer when the window and context are created.

An application will perform such a rendering as follows:

```
/* Enable stencil test and leave it enabled throughout */
glEnable (GL_STENCIL_TEST);

/* Prepare to write a single bit into the stencil buffer in the area outsi
glStencilFunc (GL_ALWAYS, 0x1, 0x1);
```

```
/* Render a set of geometry corresponding to the area outside the viewport

/* The stencil buffer now has a single bit painted in the area outside the

/* Prepare to render the scene in the viewport */
glStencilFunc (GL_EQUAL, 0x0, 0x1);

/* Render the scene inside the viewport here */

/* ...render the scene again as needed for animation purposes */
```

After a single bit is painted in the area outside the viewport, an application may render geometry to either the area inside or outside the viewport. To render to the inside area, use glStencilFunc(GL_EQUAL,0x0,0x1), as the code above shows. To render to the area outside the viewport, use glStencilFunc(GL_EQUAL,0x1,0x1).

You can obtain similar results using only the depth test. After rendering a 3D scene to a rectangular viewport, an app can clear the depth buffer and render the nonrectangular frame.

10.040 When an OpenGL primitive moves placing one vertex outside the window, suddenly the color or texture mapping is incorrect. What's going on?

There are two potential causes for this.

When a primitive lies partially outside the window, it often crosses the view volume boundary. OpenGL must clip any primitive that crosses the view volume boundary. To clip a primitive, OpenGL must interpolate the color values, so they're correct at the new clip vertex. This interpolation is perspective correct. However, when a primitive is rasterized, the color values are often generated using linear interpolation in window space, which isn't perspective correct. The difference in generated color values means that for any given barycentric coordinate location on a filled primitive, the color values may be different depending on whether the primitive is clipped. If the color values generated during rasterization were perspective correct, this problem wouldn't exist.

For some OpenGL implementations, texture coordinates generated during rasterization aren't perspective correct. However, you can usually make them perspective correct by calling glHint(GL_PERSPECTIVE_CORRECTION_HINT,GL_NICEST);. Colors generated at the rasterization stage aren't perspective correct in almost every OpenGL implementation, and can't be made so. For this reason, you're more likely to encounter this problem with colors than texture coordinates.

A second reason the color or texture mapping might be incorrect for a clipped primitive is because the color values or texture coordinates are nonplanar. Color values are nonplanar when the three color components at each vertex don't lie in a plane in 3D color space. 2D texture coordinates are always planar. However, in this context, the term nonplanar is used for texture coordinates that look up a texel area that isn't congruent in shape to the primitive being textured.

Nonplanar colors or texture coordinates aren't a problem for triangular primitives, but the problem may occur with GL_QUADS, GL_QUAD_STRIP and GL_POLYGON primitives. When using nonplanar color values or texture coordinates, there isn't a correct way to generate new values associated with clipped vertices. Even perspective−correct interpolation

can create differences between clipped and nonclipped primitives. The solution to this problem is to not use nonplanar color values and texture coordinates.

10.050 I know my geometry is inside the view volume. How can I turn off OpenGL's view−volume clipping to maximize performance?

Standard OpenGL doesn't provide a mechanism to disable the view−volume clipping test; thus, it will occur for every primitive you send.

Some implementations of OpenGL support the GL_EXT_clip_volume_hint extension. If the extension is available, a call to glHint(GL_CLIP_VOLUME_CLIPPING_HINT_EXT,GL_FASTEST) will inform OpenGL that the geometry is entirely within the view volume and that view−volume clipping is unnecessary. Normal clipping can be resumed by setting this hint to GL_DONT_CARE. When clipping is disabled with this hint, results are undefined if geometry actually falls outside the view volume.

10.060 When I move the viewpoint close to an object, it starts to disappear. How can I disable OpenGL's zNear clipping plane?

You can't. If you think about it, it makes sense: What if the viewpoint is in the middle of a scene? Certainly some geometry is behind the viewer and needs to be clipped. Rendering it will produce undesirable results.

For correct perspective and depth buffer calculations to occur, setting the *zNear* clipping plane to 0.0 is also not an option. The *zNear* clipping plane must be set at a positive (nonzero) distance in front of the eye.

To avoid the clipping artifacts that can otherwise occur, an application must track the viewpoint location within the scene, and ensure it doesn't get too close to any geometry. You can usually do this with a simple form of collision detection. This FAQ contains more information on collision detection with OpenGL.

If you're certain that your geometry doesn't intersect any of the view−volume planes, you might be able to use an extension to disable clipping. See the previous question for more information.

10.070 How do I draw glBitmap() or glDrawPixels() primitives that have an initial glRasterPos() outside the window's left or bottom edge?

When the raster position is set outside the window, it's often outside the view volume and subsequently marked as invalid. Rendering the glBitmap and glDrawPixels primitives won't occur with an invalid raster position. Because glBitmap/glDrawPixels produce pixels up and to the right of the raster position, it appears impossible to render this type of primitive clipped by the left and/or bottom edges of the window.

However, here's an often−used trick: Set the raster position to a valid value inside the view volume. Then make the following call:

```
glBitmap (0, 0, 0, 0, xMove, yMove, NULL);
```

This tells OpenGL to render a no−op bitmap, but move the current raster position by (x*Move*,y*Move*). Your application will supply (x*Move*,y*Move*) values that place the raster position outside the view volume. Follow this call with the glBitmap() or glDrawPixels() to do the rendering you desire.

10.080 Why doesn't glClear() work for areas outside the scissor rectangle?

The OpenGL Specification states that glClear() only clears the scissor rectangle when the scissor test is enabled. If you want to clear the entire window, use the code:

```
glDisable (GL_SCISSOR_TEST);
glClear (...);
glEnable (GL_SCISSOR_TEST);
```

10.090 How does face culling work? Why doesn't it use the surface normal?

OpenGL face culling calculates the signed area of the filled primitive in window coordinate space. The signed area is positive when the window coordinates are in a counter−clockwise order and negative when clockwise. An app can use glFrontFace() to specify the ordering, counter−clockwise or clockwise, to be interpreted as a front−facing or back−facing primitive. An application can specify culling either front or back faces by calling glCullFace(). Finally, face culling must be enabled with a call to glEnable(GL_CULL_FACE); .

OpenGL uses your primitive's window space projection to determine face culling for two reasons. To create interesting lighting effects, it's often desirable to specify normals that aren't orthogonal to the surface being approximated. If these normals were used for face culling, it might cause some primitives to be culled erroneously. Also, a dot−product culling scheme could require a matrix inversion, which isn't always possible (i.e., in the case where the matrix is singular), whereas the signed area in DC space is always defined.

However, some OpenGL implementations support the GL_EXT_ cull_vertex extension. If this extension is present, an application may specify a homogeneous eye position in object space. Vertices are flagged as culled, based on the dot product of the current normal with a vector from the vertex to the eye. If all vertices of a primitive are culled, the primitive isn't rendered. In many circumstances, using this extension results in faster rendering, because it culls faces at an earlier stage of the rendering pipeline.

11 Color

11.010 My texture map colors reverse blue and red, yellow and cyan, etc. What's happening?

Your texture image has the reverse byte ordering of what OpenGL is expecting. One way to handle this is to swap bytes within your code before passing the data to OpenGL.

Under OpenGL 1.2, you may specify GL_BGR or GL_BGRA as the "format" parameter to glDrawPixels(), glGetTexImage(), glReadPixels(), glTexImage1D(), glTexImage2D(), and glTexImage3D(). In previous versions of OpenGL, this functionality might be available in the form of the EXT_bgra extension (using GL_BGR_EXT and GL_BGRA_EXT as the "format" parameter).

11.020 How do I render a color index into an RGB window or vice versa?

There isn't a way to do this. However, you might consider opening an RGB window with a color index overlay plane, if it works in your application.

If you have an array of color indices that you want to use as a texture map, you might want to consider using GL_EXT_paletted_texture, which lets an application specify a color index texture map with a color palette.

11.030 The colors are almost entirely missing when I render in Microsoft Windows. What's happening?

The most probable cause is that the Windows display is set to 256 colors. To change it, you can increase the color depth by clicking the right mouse button on the desktop, then select Properties, the Settings tab, and change the number of colors in the Color Palette to a higher number.

11.040 How do I specify an exact color for a primitive?

First, you'll need to know the depth of the color buffer you are rendering to. For an RGB color buffer, you can obtain these values with the following code:

```
GLint redBits, greenBits, blueBits;

glGetIntegerv (GL_RED_BITS, &redBits);
glGetIntegerv (GL_GREEN_BITS, &greenBits);
glGetIntegerv (GL_BLUE_BITS, &blueBits);
```

If the depth value for each component is at least as large as your required color precision, you can specify an exact color for your primitives. Specify the color you want to use into the most significant bits of three unsigned integers and use glColor3ui() to specify the color.

If your color buffer isn't deep enough to accurately represent the color you desire, you'll need a fallback strategy. Trimming off the least significant bits of each color component is an acceptable alternative. Again, use glColor3ui() (or glColor3us(), etc.) to specify the color with your values stored in the most significant bits of each parameter.

In either event, you'll need to ensure that any state that could affect the final color has been disabled. The following code will accomplish this:

```
glDisable (GL_BLEND);
glDisable (GL_DITHER);
glDisable (GL_FOG);
glDisable (GL_LIGHTING);
glDisable (GL_TEXTURE_1D);
glDisable (GL_TEXTURE_2D);
glDisable (GL_TEXTURE_3D);
glShadeModel (GL_FLAT);
```

11.050 How do I render each primitive in a unique color?

You need to know the depth of each component in your color buffer. The previous question contains the code to obtain these values. The depth tells you the number of unique color values you can render. For example, if you use the code from the previous question, which retrieves the color depth in redBits, greenBits, and blueBits, the number of unique colors available is 2^(redBits+greenBits+blueBits).

If this number is greater than the number of primitives you want to render, there is no problem. You need to use glColor3ui() (or glColor3us(), etc) to specify each color, and store the desired color in the most significant bits of each parameter. You can code a loop to render each primitive in a unique color with the following:

```
/*
   Given: numPrims is the number of primitives to render.
   Given void renderPrimitive(unsigned long) is a routine to render the pr
   Given GLuint makeMask (GLint) returns a bit mask for the number of bits
 */

GLuint redMask = makeMask(redBits) << (greenBits + blueBits);
GLuint greenMask = makeMask(greenBits) << blueBits;
GLuint blueMask = makeMask(blueBits);
int redShift = 32 - (redBits+greenBits+blueBits);
int greenShift = 32 - (greenBits+blueBits);
int blueShift = 32 - blueBits;
unsigned long indx;

for (indx=0; indx<numPrims, indx++) {
   glColor3ui (indx & redMask << redShift,
               indx & greenMask << greenShift,
               indx & blueMask << blueShift);
   renderPrimitive (indx);
}
```

Also, make sure you disable any state that could alter the final color. See the question above for a code snippet to accomplish this.

If you're using this for picking instead of the ususal Selection feature, any color subsequently read back from the color buffer can easily be converted to the indx value of the primitive rendered in that color.

12 The Depth Buffer

12.010 How do I make depth buffering work?

Your application needs to do at least the following to get depth buffering to work:

1. Ask for a depth buffer when you create your window.
2. Place a call to glEnable (GL_DEPTH_TEST) in your program's initialization routine, after a context is created and made current.
3. Ensure that your *zNear* and *zFar* clipping planes are set correctly and in a way that provides adequate depth buffer precision.
4. Pass GL_DEPTH_BUFFER_BIT as a parameter to glClear, typically bitwise OR'd with other values such as GL_COLOR_BUFFER_BIT.

There are a number of OpenGL example programs available on the Web, which use depth buffering. If you're having trouble getting depth buffering to work correctly, you might benefit from looking at an example program to see what is done differently. This FAQ contains links to several web sites that have example OpenGL code.

12.020 Depth buffering doesn't work in my perspective rendering. What's going on?

Make sure the *zNear* and *zFar* clipping planes are specified correctly in your calls to glFrustum() or gluPerspective().

A mistake many programmers make is to specify a *zNear* clipping plane value of 0.0 or a negative value which isn't allowed. Both the *zNear* and *zFar* clipping planes are positive (not zero or negative) values that represent distances in front of the eye.

Specifying a *zNear* clipping plane value of 0.0 to gluPerspective() won't generate an OpenGL error, but it might cause depth buffering to act as if it's disabled. A negative *zNear* or *zFar* clipping plane value would produce undesirable results.

A *zNear* or *zFar* clipping plane value of zero or negative, when passed to glFrustum(), will produce an error that you can retrieve by calling glGetError(). The function will then act as a no−op.

12.030 How do I write a previously stored depth image to the depth buffer?

Use the glDrawPixels() command, with the format parameter set to GL_DEPTH_COMPONENT. You may want to mask off the color buffer when you do this, with a call to glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE); .

12.040 Depth buffering seems to work, but polygons seem to bleed through polygons that are in front of them. What's going on?

You may have configured your *zNear* and *zFar* clipping planes in a way that severely limits your depth buffer precision. Generally, this is caused by a *zNear* clipping plane value that's too close to 0.0. As the *zNear* clipping plane is set increasingly closer to 0.0, the effective precision of the depth buffer decreases dramatically. Moving the *zFar* clipping plane further away from the eye always has a negative impact on depth buffer precision, but it's not one as

dramatic as moving the *zNear* clipping plane.

The *OpenGL Reference Manual* description for glFrustum() relates depth precision to the *zNear* and *zFar* clipping planes by saying that roughly log2(*zFar/zNear*) bits of precision are lost. Clearly, as *zNear* approaches zero, this equation approaches infinity.

While the blue book description is good at pointing out the relationship, it's somewhat inaccurate. As the ratio (*zFar/zNear*) increases, less precision is available near the back of the depth buffer and more precision is available close to the front of the depth buffer. So primitives are more likely to interact in Z if they are further from the viewer.

It's possible that you simply don't have enough precision in your depth buffer to render your scene. See the last question in this section for more info.

It's also possible that you are drawing coplanar primitives. Round−off errors or differences in rasterization typically create "Z fighting" for coplanar primitives. Here are some options to assist you when rendering coplanar primitives.

12.050 Why is my depth buffer precision so poor?

The depth buffer precision in eye coordinates is strongly affected by the ratio of *zFar* to *zNear*, the *zFar* clipping plane, and how far an object is from the *zNear* clipping plane.

You need to do whatever you can to push the *zNear* clipping plane out and pull the *zFar* plane in as much as possible.

To be more specific, consider the transformation of depth from eye coordinates

x_e, y_e, z_e, w_e

to window coordinates

x_w, y_w, z_w

with a perspective projection matrix specified by

glFrustum(l, r, b, t, n, f);

and assume the default viewport transform. The clip coordinates of z_c and w_c are

$$z_c = -z_e * (f+n)/(f-n) - w_e * 2*f*n/(f-n)$$

$$w_c = -z_e$$

Why the negations? OpenGL wants to present to the programmer a right−handed coordinate system before projection and left−handed coordinate system after projection.

and the ndc coordinate:

$$z_{ndc} = z_c / w_c = [-z_e * (f+n)/(f-n) - w_e * 2*f*n/(f-n)] / -z_e$$

$= (f+n)/(f-n) + (w_e / z_e) * 2*f*n/(f-n)$

The viewport transformation scales and offsets by the depth range (Assume it to be [0, 1]) and then scales by $s = (2^n-1)$ where n is the bit depth of the depth buffer:

$z_w = s * [(w_e / z_e) * f*n/(f-n) + 0.5 * (f+n)/(f-n) + 0.5]$

Let's rearrange this equation to express z_e / w_e as a function of z_w

$z_e / w_e = f*n/(f-n) / ((z_w / s) - 0.5 * (f+n)/(f-n) - 0.5)$

$= f * n / ((z_w / s) * (f-n) - 0.5 * (f+n) - 0.5 * (f-n))$

$= f * n / ((z_w / s) * (f-n) - f)$ [*]

Now let's look at two points, the *zNear* clipping plane and the *zFar* clipping plane:

$z_w = 0 => z_e / w_e = f * n / (-f) = -n$

$z_w = s => z_e / w_e = f * n / ((f-n) - f) = -f$

In a fixed-point depth buffer, z_w is quantized to integers. The next representable z buffer depth away from the clip planes are 1 and s-1:

$z_w = 1 => z_e / w_e = f * n / ((1/s) * (f-n) - f)$

$z_w = s-1 => z_e / w_e = f * n / (((s-1)/s) * (f-n) - f)$

Now let's plug in some numbers, for example, n = 0.01, f = 1000 and s = 65535 (i.e., a 16-bit depth buffer)

$z_w = 1 => z_e / w_e = -0.01000015$

$z_w = s-1 => z_e / w_e = -395.90054$

Think about this last line. Everything at eye coordinate depths from -395.9 to -1000 has to map into either 65534 or 65535 in the z buffer. Almost two thirds of the distance between the *zNear* and *zFar* clipping planes will have one of two z-buffer values!

To further analyze the z-buffer resolution, let's take the derivative of [*] with respect to z_w

$d (z_e / w_e) / d z_w = - f * n * (f-n) * (1/s) / ((z_w / s) * (f-n) - f)^2$

Now evaluate it at $z_w = s$

$d (z_e / w_e) / d z_w = - f * (f-n) * (1/s) / n$

$= - f * (f/n-1) / s$ [**]

If you want your depth buffer to be useful near the *zFar* clipping plane, you need to keep this value to less than the size of your objects in eye space (for most practical uses, world space).

12.060 How do I turn off the zNear clipping plane?

See this question in the Clipping section.

12.070 Why is there more precision at the front of the depth buffer?

After the projection matrix transforms the clip coordinates, the XYZ−vertex values are divided by their clip coordinate W value, which results in normalized device coordinates. This step is known as the perspective divide. The clip coordinate W value represents the distance from the eye. As the distance from the eye increases, 1/W approaches 0. Therefore, X/W and Y/W also approach zero, causing the rendered primitives to occupy less screen space and appear smaller. This is how computers simulate a perspective view.

As in reality, motion toward or away from the eye has a less profound effect for objects that are already in the distance. For example, if you move six inches closer to the computer screen in front of your face, it's apparent size should increase quite dramatically. On the other hand, if the computer screen were already 20 feet away from you, moving six inches closer would have little noticeable impact on its apparent size. The perspective divide takes this into account.

As part of the perspective divide, Z is also divided by W with the same results. For objects that are already close to the back of the view volume, a change in distance of one coordinate unit has less impact on Z/W than if the object is near the front of the view volume. To put it another way, an object coordinate Z unit occupies a larger slice of NDC−depth space close to the front of the view volume than it does near the back of the view volume.

In summary, the perspective divide, by its nature, causes more Z precision close to the front of the view volume than near the back.

A previous question in this section contains related information.

12.080 There is no way that a standard−sized depth buffer will have enough precision for my astronomically large scene. What are my options?

The typical approach is to use a multipass technique. The application might divide the geometry database into regions that don't interfere with each other in Z. The geometry in each region is then rendered, starting at the furthest region, with a clear of the depth buffer before each region is rendered. This way the precision of the entire depth buffer is made available to each region.

13 Drawing Lines over Polygons and Using Polygon Offset

13.010 What are the basics for using polygon offset?

It's difficult to render coplanar primitives in OpenGL for two reasons:

- Given two overlapping coplanar primitives with different vertices, floating point round−off errors from the two polygons can generate different depth values for overlapping pixels. With depth test enabled, some of the second polygon's pixels will pass the depth test, while some will fail.
- For coplanar lines and polygons, vastly different depth values for common pixels can result. This is because depth values from polygon rasterization derive from the polygon's plane equation, while depth values from line rasterization derive from linear interpolation.

Setting the depth function to GL_LEQUAL or GL_EQUAL won't resolve the problem. The visual result is referred to as *stitching*, *bleeding*, or *Z fighting*.

Polygon offset was an extension to OpenGL 1.0, and is now incorporated into OpenGL 1.1. It allows an application to define a depth offset, which can apply to filled primitives, and under OpenGL 1.1, it can be separately enabled or disabled depending on whether the primitives are rendered in fill, line, or point mode. Thus, an application can render coplanar primitives by first rendering one primitive, then by applying an offset and rendering the second primitive.

While polygon offset can alter the depth value of filled primitives in point and line mode, under no circumstances will polygon offset affect the depth values of GL_POINTS, GL_LINES, GL_LINE_STRIP, or GL_LINE_LOOP primitives. If you are trying to render point or line primitives over filled primitives, use polygon offset to push the filled primitives back. (It can't be used to pull the point and line primitives forward.)

Because polygon offset alters the correct Z value calculated during rasterization, the resulting Z value, which is stored in the depth buffer will contain this offset and can adversely affect the resulting image. In many circumstances, undesirable "bleed−through" effects can result. Indeed, polygon offset may cause some primitives to pass the depth test entirely when they normally would not, or vice versa. When models intersect, polygon offset can cause an inaccurate rendering of the intersection point.

13.020 What are the two parameters in a glPolygonOffset() call and what do they mean?

Polygon offset allows the application to specify a depth offset with two parameters, *factor* and *units*. *factor* scales the maximum Z slope, with respect to X or Y of the polygon, and *units* scales the minimum resolvable depth buffer value. The results are summed to produce the depth offset. This offset is applied in screen space, typically with positive Z pointing into the screen.

The *factor* parameter is required to ensure correct results for filled primitives that are nearly edge−on to the viewer. In this case, the difference between Z values for the same pixel

generated by two coplanar primitives can be as great as the maximum Z slope in X or Y. This Z slope will be large for nearly edge−on primitives, and almost non−existent for face−on primitives. The *factor* parameter lets you add this type of variable difference into the resulting depth offset.

A typical use might be to set *factor* and *units* to 1.0 to offset primitives into positive Z (into the screen) and enable polygon offset for fill mode. Two passes are then made, once with the model's solid geometry and once again with the line geometry. Nearly edge−on filled polygons are pushed substantially away from the eyepoint, to minimize interference with the line geometry, while nearly planar polygons are drawn at least one depth buffer unit behind the line geometry.

13.030 What's the difference between the OpenGL 1.0 polygon offset extension and OpenGL 1.1 (and later) polygon offset interfaces?

The 1.0 polygon offset extension didn't let you apply the offset to filled primitives in line or point mode. Only filled primitives in fill mode could be offset.

In the 1.0 extension, a *bias* parameter was added to the normalized (0.0 − 1.0) depth value, in place of the 1.1 *units* parameter. Typical applications might obtain a good offset by specifying a *bias* of 0.001.

See the GLUT example, which renders two cylinders, one using the 1.0 polygon offset extension and the other using the 1.1 polygon offset interface.

13.040 Why doesn't polygon offset work when I draw line primitives over filled primitives?

Polygon offset, as its name implies, only works with polygonal primitives. It affects only the filled primitives: GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_QUADS, GL_QUAD_STRIP, and GL_POLYGON. Polygon offset will work when you render them with glPolygonMode set to GL_FILL, GL_LINE, or GL_POINT.

Polygon offset doesn't affect non−polygonal primitives. The GL_POINTS, GL_LINES, GL_LINE_STRIP, and GL_LINE_LOOP primitives can't be offset with glPolygonOffset().

13.050 What other options do I have for drawing coplanar primitives when I don't want to use polygon offset?

You can simulate the effects of polygon offset by tinkering with glDepthRange(). For example, you might code the following:

```
glDepthRange (0.1, 1.0);
/* Draw underlying geometry */
glDepthRange (0.0, 0.9);
/* Draw overlying geometry */
```

This code provides a fixed offset in Z, but doesn't account for the polygon slope. It's roughly equivalent to using glPolygonOffset with a factor parameter of 0.0.

You can render coplanar primitives with the Stencil buffer in many creative ways. The *OpenGL Programming Guide* outlines one well−know method. The algorithm for drawing a polygon and its outline is as follows:

1. Draw the outline into the color, depth, and stencil buffers.
2. Draw the filled primitive into the color buffer and depth buffer, but only where the stencil buffer is clear.
3. Mask off the color and depth buffers, and render the outline to clear the stencil buffer.

On some SGI OpenGL platforms, an application can use the SGIX_reference_plane extension. With this extension, the user specifies a plane equation in object coordinates corresponding to a set of coplanar primitives. You can enable or disable the plane. When the plane is enabled, all fragment Z values will derive from the specified plane equation. Thus, for any given fragment XY location, the depth value is guaranteed to be identical regardless of which primitive rendered it.

14 Rasterization and Operations on the Framebuffer

14.010 How do I obtain the address of the OpenGL framebuffer, so I can write directly to it?

OpenGL doesn't provide a standard mechanism to let an application obtain the address of the framebuffer. If an implementation allows this, it's through an extension.

Typically, programmers who write graphics programs for a single standard graphics hardware format, such as the VGA standard under Microsoft Windows, will want the framebuffer's address. The programmers need to understand that OpenGL is designed to run on a wide variety of graphics hardware, many of which don't run on Microsoft Windows and therefore, don't support any kind of standard framebuffer format. Because a programmer will likely be unfamiliar with this proprietary framebuffer layout, writing directly to it would produce unpredictable results. Furthermore, some OpenGL devices might not have a framebuffer that the CPU can address.

You can read the contents of the color, depth, and stencil buffers with the glReadPixels() command. Likewise, glDrawPixels() and glCopyPixels() are available for sending images to and BLTing images around in the OpenGL buffers.

14.015 How do I use glDrawPixels() and glReadPixels()?

glDrawPixels() and glReadPixels() write and read rectangular areas to and from the framebuffer, respectively. Also, you can access stencil and depth buffer information with the *format* parameter. Single pixels can be written or read by specifying *width* and *height* parameters of 1.

glDrawPixels() draws pixel data with the current raster position at the lower left corner. Problems using glDrawPixels() typically occur because the raster position is set incorrectly. When the raster position is set with the glRasterPos*() function, it is transformed as if it were a 3D vertex. Then the glDrawPixels() data is written to the resulting device coordinate raster position. (This allows you to tie pixel arrays and bitmap data to positions in 3D space).

When the raster position is outside the view volume, it's clipped and the glDrawPixels() call isn't rendered. This occurs even when part of the glDrawPixels() data would be visible. Here's info on how to render when the raster position is clipped.

glReadPixels() doesn't use the raster position. Instead, it obtains its (X,Y) device coordinate address from its first two parameters. Like glDrawPixels(), the area read has *x* and *y* for the lower left corner. Problems can occur when reading pixels if:

- The area being read is from a window that is overlapped or partially offscreen. glReadPixels() will return undefined data for the obscured area. (More info.)
- Memory wasn't allocated for the return data (the 7th parameter is a NULL pointer) causing a segmentation fault, core dump, or program termination. If you think you've allocated enough memory, but you still run into this problem, try doubling the amount of memory you've allocated. If this causes your read to succeed, chances are you've miscalculated the amount of memory needed.

For both glDrawPixels() and glReadPixels(), keep in mind:

- The *width* and *height* parameters are in pixels.
- If the drawn or read pixel data seems correct, but is slightly off, make sure you've set alignment correctly. Argument values are controlled with the glPixelStore*() functions. The PACK and UNPACK values control sending and receiving pixel data, from and to OpenGL, respectively.

14.020 How do I change between double− and single−buffered mode, in an existing a window?

If you create a single−buffered window, you can't change it.

If you create a double−buffered window, you can treat it as a single−buffered window by setting glDrawBuffer() to GL_FRONT and replacing your swap buffers call with a glFlush() call. To switch back to double−buffered, you need to set glDrawBuffer() to GL_BACK, and call swap buffers at the end of the frame.

14.030 How do I read back a single pixel?

Use glReadPixels(), passing a value of one for the *width* and *height* parameters.

14.040 How do I obtain the Z value for a rendered primitive?

You can obtain a single pixel's depth value by reading it back from the depth buffer with a call to glReadPixels(). This returns the screen space depth value.

It could be useful to have this value in object coordinate space. If so, you'll need to pass the window X and Y values, along with the screen space depth value to gluUnProject(). See more information on gluUnProject() here.

14.050 How do I draw a pattern into the stencil buffer?

You can set up OpenGL state as follows:

```
glEnable(GL_STENCIL_TEST);
glStencilFunc(GL_ALWAYS, 0x1, 0x1);
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
```

Subsequent rendering will set a 1 bit in the stencil buffer for every pixel rendered.

14.060 How do I copy from the front buffer to the back buffer and vice versa?

You need to call glCopyPixels(). The source and destination of glCopyPixels() are set with calls to glReadBuffer() and glDrawBuffer(), respectively. Thus, to copy from the back buffer to the front buffer, you can code the following:

```
glReadBuffer (GL_BACK);
glDrawBuffer (GL_FRONT);
glCopyPixels (GL_COLOR);
```

14.070 Why don't I get valid pixel data for an overlapped area when I call glReadPixels() where part of the window is overlapped by another window?

This is due to a portion of the OpenGL specification called the Pixel Ownership test. If a

window is obscured by another window, it doesn't have to store pixel data for the obscured region. Therefore, a glReadPixels() call can return undefined data for the obscured region.

The Pixel Ownership test varies from one OpenGL implementation to the next. Some OpenGL implementations store obscured regions of a window, or the entire window, in an off−screen buffer. Such an implementation can return valid pixel data for an obscured window. However, many OpenGL implementations map pixels on the screen one−to−one to framebuffer storage locations and don't store (and can't return) pixel data for obscured regions of a window.

One strategy is to instruct the windowing system to bring the window forward to the top of the window stack, render, then perform the glReadPixels() call. However, such an approach still risks user intervention that might obscure the source window.

An approach that might work for some applications is to render into a nonvisible window, such as a Pixmap under X Windows. This type of drawing surface can't be obscured by the user, and its contents should always pass the pixel ownership test. Reading from such a drawing surface should always yield valid pixel data. Unfortunately, rendering to such drawing surfaces is often not accelerated by graphics hardware.

14.080 Why does the appearance of my smooth−shaded quad change when I view it with different transformations?

An OpenGL implementation may or may not break up your quad into two triangles for rendering. Whether it breaks it up or not (and if it does, the method used to split the quad) will determine how color is interpolated along the edges and ultimately across each scanline.

Many OpenGL applications avoid quads altogether because of their inherent rasterization problems. A quad can be rendered easily as a two−triangle GL_TRIANGLE_STRIP primitive with the same data transmission cost as the equivalent quad. Wise programmers use this primitive in place of quads.

14.090 How do I obtain exact pixelization of lines?

The OpenGL specification allows for a wide range of line rendering hardware, so exact pixelization may not be possible at all.

You might want to read the OpenGL specification and become familiar yourself with the diamond exit rule. Being familiar with this rule will give you the best chance to obtain exact pixelization. Briefly, the diamond exit rule specifies that a diamond−shaped area exists within each pixel. A pixel is rasterized by a line only if the mathematical definition of that line exits the diamond inscribed within that pixel.

14.100 How do I turn on wide−line endpoint capping or mitering?

OpenGL draws wide lines by rendering multiple width−1 component lines adjacent to each other. If the wide line is Y major, the component lines are offset in X; if the wide line is X major, the component lines are offset in Y. This can produce ugly gaps at the junction of line segments and differences in apparent width depending on the line segment's slope.

OpenGL doesn't provide a mechanism to cleanly join lines that share common vertices nor to

cleanly cap the endpoints.

One possible solution is to render smooth (antialiased) lines instead of normal aliased lines. To produce a clean junction, you need to draw lines with depth test disabled or the depth function set to GL_ALWAYS. See the question on rendering antialiased lines for more info.

Another solution is for the application to handle the capping and mitering. Instead of rendering lines, the application needs to render face−on polygons. The application will need to perform the necessary math to calculate the vertex locations to provide the desired capping and joining styles.

14.110 How do I render rubber−band lines?

The unspoken objective of this question is, "How can I render something, then erase it without disturbing what has already been rendered?"

Here are two common approaches.

One way is to use overlay planes. You draw the rubber−band lines into the overlay planes, then clear the overlay planes. The contents of the main framebuffer isn't disturbed. The disadvantage of this approach is that OpenGL devices don't widely support overlay planes.

The other approach is to render with logic op enabled and set to XOR mode. Assuming you're rendering into an RGBA window, your code needs to look like:

```
glEnable(GL_COLOR_LOGIC_OP);
glLogicOp(GL_XOR);
```

Set the color to white and render your lines. Where your lines are drawn, the contents of the framebuffer will be inverted. When you render the lines a second time, the contents of the framebuffer will be restored.

The logic op command for RGBA windows is only available with OpenGL 1.1. Under 1.0, you can only enable logic op in color index windows, and GL_LOGIC_OP is passed as the parameter to glEnable().

14.120 If I draw a quad in fill mode and again in line mode, why don't the lines hit the same pixels as the filled quad?

Filled primitives and line primitives follow different rules for rasterization.

When a filled primitive is rendered, a pixel is only touched if its exact center falls within the primitive's mathematical boundary.

When a line primitive is rasterized, ideally a pixel is only touched if the line exits a diamond inscribed in the pixel's boundary.

From these rules, it should be clear that a line loop specified with the same vertices as those used for a filled primitive, can rasterize pixels that the filled primitive doesn't.

(The OpenGL specification allows for some deviation from the diamond exit line rasterization rule, but it makes no difference in this scenario.)

14.130 How do I draw a full−screen quad?

See this question in the Transformation section.

14.140 How do I initialize or clear a buffer without calling glClear()?

Draw a full screen quad. See the Transformation section.

14.150 How can I make line or polygon antialiasing work?

To render smooth (antialiased) lines, an application needs to do the following:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glEnable(GL_LINE_SMOOTH);
```

If the scene consists entirely of smooth lines, you need to disable the depth test or set it to GL_ALWAYS.

If a scene contains both smooth lines and other primitives, turning depth test off isn't an option. You can achieve nearly correct rendering results if you treat the smooth lines as transparent primitives. The other (non−blended) primitives should be rendered first, then the smooth lines rendered last, in back to front order. See the transparency section for more information.

Even taking these precautions might not prevent some rasterization artifacts at the joints of smooth line segments that share common vertices. The fact that the depth test is enabled could conceivably cause some line endpoints to be rendered incorrectly. This is a rendering artifact that you may have to live with if the depth test must be enabled while smooth lines are rendered.

Not all OpenGL implementations support antialiased polygons. According to the OpenGL spec, an implementation can render an aliased polygon when GL_POLYGON_SMOOTH is enabled.

14.160 How do I achieve full−scene antialiasing?

See the *OpenGL Programming Guide, Third Edition*, p452, for a description of a multi−pass accumulation buffer technique. This method performs well on devices that support the accumulation buffer in hardware.

On OpenGL 1.2 implementations that support the optional imaging extension, a smoothing filter may be applied to the final framebuffer image.

Many devices support the multisampling extension.

15 Transparency, Translucency, and Blending

15.010 What is the difference between transparent, translucent, and blended primitives?

A transparent physical material shows objects behind it as unobscured and doesn't reflect light off its surface. Clear glass is a nearly transparent material. Although glass allows most light to pass through unobscured, in reality it also reflects some light. A perfectly transparent material is completely invisible.

A translucent physical material shows objects behind it, but those objects are obscured by the translucent material. In addition, a translucent material reflects some of the light that hits it, making the material visible. Physical examples of translucent materials include sheer cloth, thin plastic, and smoked glass.

Transparent and translucent are often used synonymously. Materials that are neither transparent nor translucent are opaque.

Blending is OpenGL's mechanism for combining color already in the framebuffer with the color of the incoming primitive. The result of this combination is then stored back in the framebuffer. Blending is frequently used to simulate translucent physical materials. One example is rendering the smoked glass windshield of a car. The driver and interior are still visible, but they are obscured by the dark color of the smoked glass.

15.020 How can I achieve a transparent effect?

OpenGL doesn't support a direct interface for rendering translucent (partially opaque) primitives. However, you can create a transparency effect with the blend feature and carefully ordering your primitive data. You might also consider using screen door transparency.

An OpenGL application typically enables blending as follows:

```
glEnable (GL_BLEND);
glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

After blending is enabled, as shown above, the incoming primitive color is blended with the color already stored in the framebuffer. glBlendFunc() controls how this blending occurs. The typical use described above modifies the incoming color by its associated alpha value and modifies the destination color by one minus the incoming alpha value. The sum of these two colors is then written back into the framebuffer.

The primitive’s opacity is specified using glColor4*(). RGB specifies the color, and the alpha parameter specifies the opacity.

When using depth buffering in an application, you need to be careful about the order in which you render primitives. Fully opaque primitives need to be rendered first, followed by partially opaque primitives in back−to−front order. If you don't render primitives in this order, the primitives, which would otherwise be visible through a partially opaque primitive, might lose the depth test entirely.

15.030 How can I create screen door transparency?

This is accomplished by specifying a polygon stipple pattern with glPolygonStipple() and by

rendering the transparent primitive with polygon stippling enabled (glEnable(GL_POLYGON_STIPPLE)). The number of bits set in the stipple pattern determine the amount of translucency and opacity; setting more bits result in a more opaque object, and setting fewer bits results in a more translucent object. Screendoor transparency is sometimes preferable to blending, becuase it's order independent (primitives don't need to be rendered in back−to−front order).

15.040 How can I render glass with OpenGL?

This question is difficult to answer, because what looks like glass to one person might not to another. What follows is a general algorithm to get you started.

First render all opaque objects in your scene. Disable lighting, enable blending, and render your glass geometry with a small alpha value. This should result in a faint rendering of your object in the framebuffer. (Note: You may need to sort your glass geometry, so it's rendered in back to front Z order.)

Now, you need to add the specular highlight. Set your ambient and diffuse material colors to black, and your specular material and light colors to white. Enable lighting. Set glDepthFunc(GL_EQUAL), then render your glass object a second time.

15.050 Do I need to render my primitives from back to front for correct rendering of translucent primitives to occur?

If your hardware supports destination alpha, you can experiment with different glBlendFunc() settings that use destination alpha. However, this won't solve all the problems with depth buffered translucent surfaces. The only sure way to achieve visually correct results is to sort and render your primitives from back to front.

15.060 I want to use blending but can’t get destination alpha to work. Can I blend or create a transparency effect without destination alpha?

Many OpenGL devices don't support destination alpha. In particular, the OpenGL 1.1 software rendering libraries from Microsoft don't support it. The OpenGL specification doesn't require it.

If you have a system that supports destination alpha, using it is a simple matter of asking for it when you create your window. For example, pass GLUT_ALPHA to glutInitDisplayMode(), then set up a blending function that uses destination alpha, such as:

```
glBlendFunc(GL_ONE_MINUS_DST_ALPHA,GL_DST_ALPHA);
```

Often this question is asked under the mistaken assumption that destination alpha is required to do blending. It's not. You can use blending in many ways to obtain a transparency effect that uses source alpha instead of destination alpha. The fact that you might be on a platform without destination alpha shouldn't prevent you from obtaining a transparency effect. See the *OpenGL Programming Guide* chapter 6 for ways to use blending to achieve transparency.

15.070 If I draw a translucent primitive and draw another primitive behind it, I expect the second primitive to show through the first, but it's not there?

Is depth buffering enabled?

If you're drawing a polygon that's behind another polygon, and depth test is enabled, then the new polygon will typically lose the depth test, and no blending will occur. On the other hand, if you've disabled depth test, the new polygon will be blended with the existing polygon, regardless of whether it's behind or in front of it.

15.080 How can I make part of my texture maps transparent or translucent?

It depends on the effect you're trying to achieve.

If you want blending to occur after the texture has been applied, then use the OpenGL blending feature. Try this:

```
glEnable (GL_BLEND);
glBlendFunc (GL_ONE, GL_ONE);
```

You might want to use the alpha values that result from texture mapping in the blend function. If so, (GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA) is always a good function to start with.

However, if you want blending to occur when the primitive is texture mapped (i.e., you want parts of the texture map to allow the underlying color of the primitive to show through), then don't use OpenGL blending. Instead, you'd use glTexEnv(), and set the texture environment mode to GL_BLEND. In this case, you'd want to leave the texture environment color to its default value of (0,0,0,0).

16 Display Lists and Vertex Arrays

16.010 Why does a display list take up so much memory?

An OpenGL display list must make a copy of all data it requires to recreate the call sequence that created it. This means that for every glVertex3f() call, for example, the display list must provide storage for 3 values (usually 32−bit float values in most implementations). This is where most of the memory used by a typical display list goes.

However, in most implementations, there's also some memory that's needed to manage the display lists of a given context and other overhead. In certain pathological cases, this overhead memory can be larger than the memory used to store the display list data!

16.020 How can I share display lists between different contexts?

If you're using Microsoft Windows, use the wglShareLists() function. If you are using GLX, see the *share* parameter to glXCreateContext().

GLUT does not allow display list sharing. You can obtain the GLUT source, and make your own glutCreateWindow() and glutSetWindow() function calls. You can then modify the source to expose display list sharing. When doing so, you need to make sure your modified routines still work with the rest of GLUT.

16.030 How does display list nesting work? Is the called list copied into the calling list?

No. Only the call to the enclosed display list is copied into the parent list. This way a program can delete or replace a child list, call the parent, and see changes that were made.

16.040 Can I do a particular function while a display list is called?

A display list call is an atomic operation and therefore, it can't be interrupted. You can't call part of it, for example, then do something, then call the rest of it. Nor can you have a display list somehow signal your program from some point within the list.

However, an application doesn't have to create one large monolithic display list. By creating several smaller lists to call sequentially, an application is free to perform tasks between calls to glCallList().

An application can also use multithreading, so one thread can perform one task while another thread is calling a display list.

16.050 How can I change an OpenGL function call in a display list that contains many other OpenGL function calls?

OpenGL display lists aren't editable, so you can't modify the call sequence in them or even see which calls are embedded in them.

One way of creating a pseudo−editable display list is to create a hierarchical display list. (i.e., create a display list parent that contains calls to glCallList()). Then you can edit the display list by replacing the child display lists that the parent list references.

16.060 How can I obtain a list of function calls and OpenGL call parameters from a display list?

Currently, there isn't a way to programatically obtain either the function calls contained within a list or the parameters to those calls. An application that requires this information must track the data stored in a display list.

One option is to use an OpenGL call logging utility. These utilities capture the OpenGL calls a program makes, enabling you to see the calls that an application stores in a display list.

16.070 I've converted my program to use display lists, and it doesn't run any faster. Why not?

Achieving the highest performance from display lists is highly dependent on the OpenGL implementation, but here are a few pointers:

First, make sure that your application's process size isn't becoming so large that it's causing memory thrashing. Using display lists generally takes more memory than immediate mode, so it's possible that your program is spending more time thrashing memory blocks than rendering OpenGL calls.

Display lists won't improve the performance of a fill−limited application. Try rendering to a smaller window, and if your application runs faster, it's likely that it's fill−limited.

Stay away from GL_COMPILE_AND_EXECUTE mode. Instead, create the list using GL_COMPILE mode, then execute it with glCallList().

In some cases if you group your state changes together, the display list can optimize them as a group (i.e., it can remove redundant state changes, concatenate adjacent matrix changes, etc.).

Read the section on Performance for other tips.

16.080 To save space, should I convert all my coordinates to short before storing them in a display list?

No. Most implementations will convert your data to an internal format for storage in the display list anyway, and usually, that format will be single−precision float.

16.090 Will putting textures in a display list make them run faster?

In some implementations, a display list can optimize texture download and use of texture memory. In OpenGL 1.0, storing texture maps in display lists was the preferred method for optimizing texture performance. However, it resulted in increased memory usage in many implementations. Many vendors rallied around a better solution, texture objects, introduced in OpenGL 1.1. If your app is running on OpenGL 1.1 or later, texture objects are preferred.

16.100 Will putting vertex arrays in a display list make them run faster?

It depends on the implementation. In most implementations, it might decrease performance because of the increased memory use. However, some implementations may cache display lists on the graphics hardware, so the benefits of this caching could easily offset the extra memory usage.

16.110 When sharing display lists between contexts, what happens when I delete a display list in one context? Do I have to delete it in all the contexts to make it really go away?

When a display list is modified in one context (deleting is a form of modification), the results of that modification are immediately available in all shared contexts. So, deleting a display list in one context will cause it to cease to exist in all contexts in which it was previously visible.

16.120 How many display lists can I create?

There isn't a limit based on the OpenGL spec. Because a display list ID is a GLuint, 2^{32} display list identifiers are available. A more practical limit to go by is system memory resources.

16.130 How much memory does a display list use?

See the first question in this section. It depends on the implementation.

16.140 How will I know if the memory used by a display list has been freed?

This depends on the implementation. Some implementations free memory as soon as a display list is deleted. Others won't free the memory until it's needed by another display list or until the process dies.

16***.150 How can I use vertex arrays to share vertices?***

Because vertex arrays let you access a set of vertices and data by index, you might believe that they're designed to optimally share vertices. Indeed, a programmer new to vertex arrays might try to render a cube, in which each vertex is shared by three faces. The futility of this becomes obvious when you add normals for lighting and each instance of the shared vertex requires a unique normal. The only way to render a cube with normals is to include multiple copies of each vertex.

Vertex arrays weren't designed to improve vertex sharing. They were intended to let the programmer to specify blocks of dynamic geometry data with as few function calls as possible.

You can share vertices with vertex arrays the same way you do with OpenGL immediate mode, by the type of primitive used. GL_LINE_STRIP, GL_LINE_LOOP, GL_TRIANGLE_STRIP, and GL_QUAD_STRIP share vertices between their component line segments, triangles, and quads. Other primitives do not. The type of primitive you choose to use when using vertex arrays determines whether you share vertices.

Note, however, that sharing vertices is implementation dependent. The OpenGL Specification dictates vertex array behavior, and as long as an OpenGL implementation conforms to spec, it's free to optimize vertex sharing in vertex arrays.

Some implementations feature the EXT_compiled_vertex_array extension, which is explicitly designed to let implementations share transformed vertex array data.

17 Using Fonts

17.010 How can I add fonts to my OpenGL scene?

OpenGL doesn't provide direct font support, so the application must use any of OpenGL's other features for font rendering, such as drawing bitmaps or pixmaps, creating texture maps containing an entire character set, drawing character outlines, or creating 3D geometry for each character.

Use bitmaps or pixmaps

The most straightforward method for rendering simple fonts is to use a glBitmap() or glDrawPixels() call for each character. The result is simple 2D text, which is suitable for labeling GUI controls, annotating 3D parts, etc.

glBitmap() is the fastest and simplest of the two, and renders characters in the current color. You can also use glDrawPixels() if required. However, note that glDrawPixels() always draws a rectangle, so if you desire a transparent background, it must be removed with alpha test and/or blending.

Typically, each glBitmap() call, one for every glyph in the font, is stored in an individual display list, which is indexed by its ASCII character value. Thus, a single call to glCallLists() can render an entire string of characters.

In X Windows, the glXUseXFont() call is available to create these display lists painlessly from a given font.

If you're using Microsoft Windows, look at the MSDN documentation for wglUseFontBitmaps(). It's conceptually identical to glXUseXFonts().

For GLUT, you need to use the glutBitmapCharacter() routine, which generates a bitmap for the specified character from the specified GLUT bitmap font.

Use texture mapping

In many OpenGL implementations, rendering glBitmap() and glDrawPixels() primitives is inherently slower than rendering an equivalent texture mapped quad. Use texture mapped primitives to render fonts on such devices.

The basic idea is to create a single texture map that contains all characters in a font (or at least all the characters that need to be rendered). To render an individual character, draw a texture mapped quad with texture coordinates configured to select the desired individual character. If desired, you can use alpha test to discard background pixels.

Follow this link to info on texturemapped fonts, as well as other OpenGL tidbits.

A library for using texture mapped fonts can be found here. It comes with source code.

The GLUT source distribution comes with a texture mapped font demo.

The NeHe web page has a tutorial on using texture mapped fonts.

Stroked fonts

If you're using Microsoft Windows, look up the MSDN documentation on wglUseFontOutlines(). It contains example code for rendering stroked characters.

The glutStrokeCharacter() routine renders a single stroked character from a specified GLUT stroke font.

Geometric fonts

The NeHe web page has a tutorial for rendering geometric fonts. Look for the tutorial on outline fonts.

17.020 How can I use TrueType fonts in my OpenGL scene?

The NeHe web page has tutorials that show how to use TrueType fonts in a variety of ways.

See the Free Type library.

17.030 How can I make 3D letters, which I can light, shade, and rotate?

See the NeHe web page for a tutorial on using geometric fonts. Look for the tutorial on outline fonts.

See the Free Type library.

GLTT supports geometric TrueType fonts in OpenGL. It was formerly available from http://www.moonlight3d.org/gltt/, but fortunately is still available around the Web. Download GLTT v 2.4 (~125KB).

Glut 3.7 has an example called progs/contrib/text3d.c that may be informative.

18 Lights and Shadows

18.010 What should I know about lighting in general?

You must specify normals along with your geometry, or you must generate them automatically with evaluators, in order for lighting to work as expected. This is covered in question 18.020.

Lighting does not work with the current color as set by glColor*(). It works with material colors. Set the material colors with glMaterial*(). Material colors can be made to track the current color with the color material feature. To use color material, call glEnable(GL_COLOR_MATERIAL). By default, this causes ambient and diffuse material colors to track the current color. You can specify which material color tracks the current color with a call to glColorMaterial().

Changing the material colors with color material and glColor*() calls may be more efficient than using glMaterial*(). See question 18.080 for more information.

Lighting is computed at each vertex (and interpolated across the primitive, when glShadeModel() is set to GL_SMOOTH). This may cause primitives to appear too dark, even though a light is centered over the primitive. You can obtain more correct lighting with a higher surface approximation, or by using light maps.

A light's position is transformed by the current ModelView matrix at the time the position is specified with a call to glLight*(). This is analogous to how geometric vertices are transformed by the current ModelView matrix when they are specified with a call to glVertex*(). For more information on positioning your light source, see question 18.050.

18.020 Why are my objects all one flat color and not shaded and illuminated?

This effect occurs when you fail to supply a normal at each vertex.

OpenGL needs normals to calculate lighting equations, and it won't calculate normals for you (with the exception of evaluators). If your application doesn't call glNormal*(), then it uses the default normal of (0.0, 0.0, 1.0) at every vertex. OpenGL will then compute the same, or nearly the same, lighting result at each vertex. This will cause your model to look flat and lack shading.

The solution is to simply calculate the normals that need to be specified at any given vertex. Then send them to OpenGL with a call to glNormal3f() just prior to specifying the vertex, which the normal is associated with.

If you don't know how to calculate a normal, in most cases you can do it simply with a vector cross product. The *OpenGL Programming Guide* contains a small section explaining how to calculate normals. Also most basic 3D computer graphics books cover it, because it's not OpenGL−specific.

18.030 How can I make OpenGL automatically calculate surface normals?

OpenGL won't do this unless you're using evaluators.

18.040 Why do I get only flat shading when I light my model?

First, check the obvious. glShadeModel() should be set to GL_SMOOTH, which is the default value, so if you haven't called glShadeModel() at all, it's probably already set to GL_SMOOTH, and something else is wrong.

If glShadeModel() is set correctly, the problem is probably with your surface normals. To achieve a smooth shading effect, generally you need to specify a different normal at each vertex. If you have set the same normal at each vertex, the result, in most cases, will be a flatly shaded primitive.

Keep in mind that a typical surface normal is perpendicular to the surface that you're attempting to approximate.

This scenario can be tough to debug, especially for large models. The best debugging approach is to write a small test program that draws only one primitive, and try to reproduce the problem. It's usually easy to use a debugger to isolate and fix a small program, which reproduces the problem.

18.050 How can I make my light move or not move and control the light position?

First, you must understand how the light position is transformed by OpenGL.

The light position is transformed by the contents of the current top of the ModelView matrix stack when you specify the light position with a call to glLightfv(GL_LIGHT_POSITION,…). If you later change the ModelView matrix, such as when the view changes for the next frame, the light position isn't automatically retransformed by the new contents of the ModelView matrix. If you want to update the light’s position, you must again specify the light position with a call to glLightfv(GL_LIGHT_POSITION,…).

Asking the question “how do I make my light move” or “how do I make my light stay still” usually doesn't provide enough information to answer the question. For a better answer, you need to be more specific. Here are some more specific questions, and their answers:

- How can I make my light position stay fixed relative to my eye position? How do I make a headlight?

You need to specify your light in eye coordinate space. To do so, set the ModelView matrix to the identity, then specify your light position. To make a headlight (a light that appears to be positioned at or near the eye and shining along the line of sight), set the ModelView to the identity, set the light position at (or near) the origin, and set the direction to the negative Z axis.

When a light’s position is fixed relative to the eye, you don't need to respecify the light position for every frame. Typically, you specify it once when your program initializes.

- How can I make my light stay fixed relative to my scene? How can I put a light in the corner and make it stay there while I change my view?

As your view changes, your ModelView matrix also changes. This means you'll need to respecify the light position, usually at the start of every frame. A typical application will

display a frame with the following pseudocode:

```
Set the view transform.
Set the light position //glLightfv(GL_LIGHT_POSITION,133;)
Send down the scene or model geometry.
Swap buffers.
```

If your light source is part of a light fixture, you also may need to specify a modeling transform, so the light position is in the same location as the surrounding fixture geometry.

- How can I make a light that moves around in a scene?

  Again, you'll need to respecify this light position every time the view changes. Additionally, this light has a dynamic modeling transform that also needs to be in the ModelView matrix before you specify the light position. In pseudocode, you need to do something like:

```
Set the view transform
Push the matrix stack
Set the model transform to update the light146;s position
Set the light position //glLightfv(GL_LIGHT_POSITION,133;)
Pop the matrix stack
Send down the scene or model geometry
Swap buffers.
```

18.060 How can I make a spotlight work?

A spotlight is simply a point light source with a small light cone radius. Alternatively, a point light is just a spot light with a 180 degree radius light cone. Set the radius of the light cone by changing the cutoff parameter of the light:

```
glLightf (GL_LIGHT1, GL_SPOT_CUTOFF, 15.f);
```

The above call sets the light cone radius to 15 degrees for light 1. The light cone's total spread will be 30 degrees.

A spotlight's position and direction are set as for any normal light.

18.070 How can I create more lights than GL_MAX_LIGHTS?

First, make sure you really need more than OpenGL provides. For example, when rendering a street scene at night with many buildings and streetlights, you need to ask yourself: Does every building need to be illuminated by every single streetlight? When light attenuation and direction are accounted for, you may find that any given piece of geometry in your scene is only illuminated by a small handful of lights.

If this is the case, you need to reuse or cycle the available OpenGL lights as you render your scene.

The GLUT distribution comes with a small example that might be informative to you. It’s called multilight.c.

If you really need to have a single piece of geometry lit by more lights than OpenGL provides, you'll need to simulate the effect somehow. One way is to calculate the lighting for

some or all the lights. Another method is to use texture maps to simulate lighting effects.

18.080 Which is faster: making glMaterial() calls or using glColorMaterial()?*

Within a glBegin()/glEnd() pair, on most OpenGL implementations, a call to glColor3f() generally is faster than a call to glMaterialfv(). This is simply because most implementations tune glColor3f(), and processing a material change can be complex and difficult to optimize. For this reason, glColorMaterial() generally is recognized as the most efficient way to change an object's material color.

18.090 Why is the lighting incorrect after I scale my scene to change its size?

The OpenGL specification needs normals to be unit length to achieve typical lighting results. The current ModelView matrix transforms normals. If that matrix contains a scale transformation, transformed normals might not be unit length, resulting in undesirable lighting problems.

OpenGL 1.1 lets you call glEnable(GL_NORMALIZE), which will make all normals unit length after they're transformed. This is often implemented with a square root and can be expensive for geometry limited applications.

Another solution, available in OpenGL 1.2 (and as an extension to many 1.1 implementations), is glEnable(GL_RESCALE_NORMAL). Rather than making normals unit length by computing a square root, GL_RESCALE_NORMAL multiplies the transformed normal by a scale factor. If the original normals are unit length, and the ModelView matrix contains uniform scaling, this multiplication will restore the normals to unit length.

If the ModelView matrix contains nonuniform scaling, GL_NORMALIZE is the preferred solution.

18.100 After I turn on lighting, everything is lit. How can I light only some of the objects?

Remember that OpenGL is a state machine. You'll need to do something like this:

```
glEnable(GL_LIGHTING);
// Render lit geometry.
glDisable(GL_LIGHTING);
// Render non-lit geometry.
```

18.110 How can I use light maps (e.g., Quake-style) in OpenGL?

See this question in the Texture Mapping section.

18.120 How can I achieve a refraction lighting effect?

First, consider whether OpenGL is the right API for you. You might need to use a ray tracer to achieve complex light affects such as refraction.

If you're certain that you want to use OpenGL, you need to keep in mind that OpenGL doesn't provide functionality to produce a refraction effect. You'll need to fake it. The most likely solution is to calculate an image corresponding to the refracted rendering, and texture map it onto the surface of the primitive that's refracting the light.

18.130 How can I render caustics?

OpenGL can't help you render caustics, except for texture mapping. GLUT 3.7 comes with some demos that show you how to achieve caustic lighting effects.

18.140 How can I add shadows to my scene?

OpenGL does not support shadow rendering directly. However, any standard algorithm for rendering shadows can be used in OpenGL. Some algorithms are described at http://www.opengl.org. Follow the Coding Tutorials & Techniques link, then the Rendering Techniques link. Scroll down to the Lighting, Shadows, & Reflections section.

The GLUT 3.7 distribution comes with examples that demonstrate how to do this using projection shadows and the stencil buffer.

Projection shadows are ideal if your shadow is only to lie on a planar object. You can generate geometry of the shadow using glFrustum() to transform the object onto the projection plane.

Stencil buffer shadowing is more flexible, allowing shadows to lie on any object, planar or otherwise. The basic algorithm is to calculate a "shadow volume". Cull the back faces of the shadow volume and render the front faces into the stencil buffer, inverting the stencil values. Then render the shadow volume a second time, culling front faces and rendering the back faces into the stencil buffer, again inverting the stencil value. The result is that the stencil planes will now contain non−zero values where the shadow should be rendered. Render the scene a second time with only ambient light enabled and glDepthFunc() set to GL_EQUAL. The result is a rendered shadow.

Another mechanism for rendering shadows is outlined in the SIGGRAPH '92 paper *Fast Shadows and Lighting Effects Using Texture Mapping*, Mark Segal et al. This paper describes a relatively simple extension to OpenGL for using the depth buffer as a shadow texture map. Both the GL_EXT_depth_texture and the GL_EXT_texture3D (or OpenGL 1.2) extensions are required to use this method.

19 Curves, Surfaces, and Using Evaluators

19.010 How can I use OpenGL evaluators to create a B−spline surface?

OpenGL evaluators use a Bezier basis. To render a surface using any other basis, such as B−spline, you must convert your control points to a Bezier basis. The *OpenGL Programming Guide*, Chapter 12, lists a number of reference books that cover the math behind these conversions.

19.020 How can I retrieve the geometry values produced by evaluators?

OpenGL does not provide a straightforward mechanism for this.

You might download the Mesa source code distribution, and modify its evaluator code to return object coordinates rather than pass them into the OpenGL geometry pipeline.

Evaluators involve a lot of math, so their performance in immediate mode is sometimes unacceptable. Some programmers think they need to "capture" the generated geometry, and play it back to achieve maximum performance. Indeed, this would be a good solution if it were possible. Some implementations provide maximum evaluator performance through the use of display lists.

20 Picking and Using Selection

20.010 How can I know which primitive a user has selected with the mouse?

OpenGL provides the GL_SELECTION render mode for this purpose. However, you can use other methods.

You might render each primitive in a unique color, then use glReadPixels() to read the single pixel under the current mouse location. Examining the color determines the primitive that the user selected. Here's information on rendering each primitive in a unique color and information on using glDrawPixels().

Yet another method involves shooting a pick ray through the mouse location and testing for intersections with the currently displayed objects. OpenGL doesn't test for ray intersections (for how to do, see the BSP FAQ), but you'll need to interact with OpenGL to generate the pick ray.

One way to generate a pick ray is to call gluUnProject() twice for the mouse location, first with *winz* of 0.0 (at the near plane), then with *winz* of 1.0 (at the far plane). Subtract the near plane call's results from the far plane call's results to obtain the XYZ direction vector of your ray. The ray origin is the view location, of course.

Another method is to generate the ray in eye coordinates, and transform it by the inverse of the ModelView matrix. In eye coordinates, the pick ray origin is simply (0, 0, 0). You can build the pick ray vector from the perspective projection parameters, for example, by setting up your perspective projection this way

```
aspect = double(window_width)/double(window_height);
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
glFrustum(-near_height * aspect,
   near_height * aspect,
   -near_height,
   near_height, zNear, zFar );
```

you can build your pick ray vector like this:

```
int window_y = (window_height - mouse_y) - window_height/2;
double norm_y = double(window_y)/double(window_height/2);
int window_x = mouse_x - window_width/2;
double norm_x = double(window_x)/double(window_width/2);
```

(Note that most window systems place the mouse coordinate origin in the upper left of the window instead of the lower left. That's why *window_y* is calculated the way it is in the above code. When using a glViewport() that doesn't match the window height, the viewport height and viewport Y are used to determine the values for *window_y* and *norm_y*.)

The variables *norm_x* and *norm_y* are scaled between −1.0 and 1.0. Use them to find the mouse location on your *zNear* clipping plane like so:

```
float y = near_height * norm_y;
float x = near_height * aspect * norm_x;
```

Now your pick ray vector is (x, y, −zNear).

To transform this eye coordinate pick ray into object coordinates, multiply it by the inverse of the ModelView matrix in use when the scene was rendered. When performing this multiplication, remember that the pick ray is made up of a vector and a point, and that vectors and points transform differently. You can translate and rotate points, but vectors only rotate. The way to guarantee that this is working correctly is to define your point and vector as four−element arrays, as the following pseudo−code shows:

```
float ray_pnt[4] = {0.f, 0.f, 0.f, 1.f};
float ray_vec[4] = {x, y, −near_distance, 0.f};
```

The one and zero in the last element determines whether an array transforms as a point or a vector when multiplied by the inverse of the ModelView matrix.

20.020 What do I need to know to use selection?

Specify a selection buffer:

```
GLuint buffer[BUF_SIZE];
glSelectBuffer (BUF_SIZE, buffer);
```

Enter selection mode, render as usual, then exit selection mode:

```
GLint hits;

glRenderMode(GL_SELECT);
// ...render as usual...
hits = glRenderMode(GL_RENDER);
```

The call to glRenderMode(GL_RENDER) exits selection mode and returns the number of hit records stored in the selection buffer. Each hit record contains information on the primitives that were inside the view volume (controlled with the ModelView and Projection matrices).

That's the basic concept. In practice, you may want to restrict the view volume. The gluPickMatrix() function is a handy method for restricting the view volume size to within a set number of pixels away from a given (X,Y) position, such as the current mouse or cursor location.

You'll also want to use the name stack to specify unique names for primitives of interest. After the stack is pushed once, any number of different names may be loaded onto the stack. Typically, load a name, then render a primitive or group of primitives. The name stack allows for selection to occur on heirarchical databases.

After returning to GL_RENDER render mode, you'll need to parse the selection buffer. It will contain zero or more hit records. The number of hit records is returned by the call to glRenderMode(GL_RENDER). Each hit record contains the following information stored as unsigned ints:

- Number of names in the name stack for this hit record
- Minimum depth value of primitives (range 0 to $2^{32}−1$)
- Maximum depth value of primitives (range 0 to $2^{32}−1$)
- Name stack contents (one name for each unsigned int).

You can use the minimum and maximum Z values with the device coordinate X and Y if known (perhaps from a mouse click) to determine an object coordinate location of the picked primitive. You can scale the Z

values to the range 0.0 to 1.0, for example, and use them in a call to gluUnProject().

20.030 Why doesn't selection work?

This is usually caused by one of two things.

Did you account for the inverted Y coordinate? Most window systems (Microsoft Windows, X Windows, others?) usually return mouse coordinates to your program with Y=0 at the top of the window, while OpenGL assumes Y=0 is at the bottom of the window. Assuming you're using a default viewport, transform the Y value from window system coordinates to OpenGL coordinates as (windowHeight−y).

Did you set up the transformations correctly? Assuming you're using gluPickMatrix(), it should be loaded onto the Projection matrix immediately after a call to glLoadIdentity() and before you multiply your projection transform (using glFrustum(), gluPerspective(), glOrtho(), etc.). Your ModelView transformation should be the same as if you were rendering normally.

20.040 How can I debug my picking code?

A good technique for debugging picking or selection code is not to call glRenderMode(GL_SELECT). Simply comment out this function call in your code. The result is instead of performing a selection, your code will render the contents of the pick box to your window. This allows you to see visually what is inside your pick box.

Along with this method, it's generally a good idea to enlarge your pick box, so you can see more in your window.

20.050 How can I perform pick highlighting the way PHIGS and PEX provided?

There's no elegant way to do this, and that's why many former PHIGS and PEX implementers are now happy as OpenGL implementers. OpenGL leaves this up to the application.

After you've identified the primitive you need to highlight with selection, how you highlight it is up to your application. You might render the primitive into the displayed image in the front buffer with a different color set. You may need to use polygon offset to make this work, or at least set glDepthFunc(GL_EQUAL). You might only render the outline or render the primitive consecutive times in different colors to create a flashing effect.

21 Texture Mapping

21.010 What are the basic steps for performing texture mapping?

At the bare minimum, a texture map must be specified, texture mapping must be enabled, and appropriate texture coordinates must be set at each vertex. While these steps will produce a texture mapped primitive, typically they don't meet the requirements of most OpenGL 1.2 applications. Use the following steps instead.

- Create a texture object for each texture in use. The texture object stores the texture map and associated texture parameter state. See question 21.070 for more information on texture objects.
- Store each texture map or mipmap pyramid in its texture object, along with parameters to control its use.
- On systems with limited texure memory, set the priority of each texture object with glPrioritizeTextures() to minimize texture memory thrashing.
- Whem your application renders the scene, bind each texture object before rendering the geomtry to be texture mapped. Enable and disable texture mapping as needed.

21.020 I'm trying to use texture mapping, but it doesn't work. What's wrong?

Check for the following:

- Texture mapping should be enabled, and a texture map must be bound (when using texture objects) or otherwise submitted to OpenGL (for example, with a call to glTexImage2D()).
- Make sure you understand the different wrap, environment, and filter modes that are available. Make sure you have set appropriate values.
- Keep in mind that texture objects don't store some texture parameters. Texture objects bind to a target (either GL_TEXTURE_1D, GL_TEXTURE_2D, or GL_TEXTURE_3D), and the texture object stores changes to those targets. glTexGen(), for example, doesn't change the state of the texture target, and therefore isn't part of texture objects.
- If you're using a mipmapping filter (e.g., you've called glTexParameter*(), setting a min or mag filter that has MIPMAP in its name), make sure you've set all levels of the mipmap pyramid. All levels must be set, or texture mapping won't occur. You can set all levels at the same time with the gluBuild2DMipmaps() function. All levels of the mipmap pyramid must have the same number of components.
- Remember that OpenGL is a state machine. If you don't specify texture coordinates, either explicitly with glTexCoord*(), or generated automatically with glTexGen()), then OpenGL uses the current texture coordinate for all vertices. This may cause some primitives to be texture mapped with a single color or single texel value.
- If you're using multiple rendering contexts and need to share texture objects between contexts, you must explicitly enable texture object sharing. This is done with the wglShareLists() function in Microsoft Windows and glXCreateContext() under X Windows.
- Check for errors with glGetError().

21.030 Why doesn't lighting work when I turn on texture mapping?

There are many well−meaning texture map demos available on the Web that set the texture environment to GL_DECAL or GL_REPLACE. These environment modes effectively replace the primitive color with the texture color. Because lighting values are calculated before texture mapping (lighting is a per vertex operation, while texture mapping is a per fragment operation), the texture color replaces the colors calculated by lighting. The result is that lighting appears to stop working when texture mapping is enabled.

The default texture environment is GL_MODULATE, which multiplies the texture color by the primitive (or lighting) color. Most applications that use both OpenGL lighting and texture mapping use the GL_MODULATE texture environment.

Look for the following line in your code:

```
glTexEnv (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL); /* or GL_REPLACE
```

You should change GL_DECAL to GL_MODULATE, or simply delete the line entirely (since GL_MODULATE is the default).

21.040 Lighting and texture mapping work pretty well, but why don't I see specular highlighting?

Your geometry may have a nice white specular highlight when it's not texture mapped, but when you apply a non−white texture suddenly the highlight goes away even though the geometry is still lit. This is because GL_MODULATE multiplies the primitive's lighting color components with the texture color components. For example, assume a white specular highlight is being multiplied by a red texture map. The final color is then (1.0*1.0, 1.0*0.0, 1.0*0.0) or (1.0, 0.0, 0.0), which is red. The white specular highlight isn't visible.

OpenGL 1.2 solves this problem by applying specular highlights after texture mapping. This separate specular lighting mode is turned on by:

```
glLightModel (GL_LIGHT_MODEL_COLOR_CONTROL,GL_SEPARATE_SPECULAR_COLOR);
```

By default, it's set to GL_SINGLE_COLOR, which maintains backwards compatibility with OpenGL 1.1 and earlier.

If you're not using OpenGL 1.2, other solutions are available. Many vendors provide proprietary extensions for allowing you to apply the specular highlight after the texture map. See this example code for how to do this on HP systems. Many OpenGL vendors have settled on an the EXT_separate_specular_color extension.

Another method works on any OpenGL implementation, because it only uses regular OpenGL 1.0 functionality and doesn't depend on extensions. You need to render your geometry in two passes: first with normal lighting and texture mapping enabled, then the second pass will render the specular highlight. See this example code for a demonstration of how to do it.

21.050 How can I automatically generate texture coordinates?

Use the glTexGen() function.

21.060 Should I store texture maps in display lists?

See this question in the display list section.

21.070 How do texture objects work?

Texture objects store texture maps and their associated texture parameter state. They allow switching between textures with a single call to glBindTexture().

Texture objects were introduced in OpenGL 1.1. Prior to that, an application changed textures by calling glTexImage*(), a rather expensive operation. Some OpenGL 1.0 implementations simulated texture object functionality for texture maps that were stored in display lists.

Like display lists, a texture object has a GLuint identifier (the *textureName* parameter to glBindTexture()). OpenGL supplies your application with texture object names when your application calls glGenTextures(). Also like display lists, texture objects can be shared across rendering contexts.

Unlike display lists, texture objects are mutable. When a texture object is bound, changes to texture object state are stored in the texture object, including changes to the texture map itself.

The following functions affect and store state in texture objects: glTexImage*(), glTexSubImage*(), glCopyTexImage*(), glCopyTexSubImage*(), glTexParameter*(), and glPrioritizeTextures(). Since the GLU routines for building mipmap pyramids ultimately call glTexImage*(), they also affect texture object state.Noticeably absent from this list are glTexEnv*() and glTexGen*(); they do not store state in texture objects.

Here is a summary of typical texture object usage:

- Get a *textureName* from glGenTextures(). You'll want one name for each of the texture objects you plan to use.
- Initially bind a texture object with glBindTexture(). Specify the texture map, and any texture parameters. Repeat this for all texture objects your application uses.
- Before rendering texture mapped geometry, call glBindTexture() with the desired *textureName*. OpenGL will use the texture map and texture parameter state stored in that object for rendering.

21.080 Can I share textures between different rendering contexts?

Yes, if you use texture objects. Texture objects can be shared the same way display lists can. If you're using Microsoft Windows, see the wglShareLists() function. For a GLX platform, see the *share* parameter to glXCreateContext().

21.090 How can I apply multiple textures to a surface?

Note that EXT_multitexture and SGIS_multitexture are both obsolete. The preferred multitexturing extension is ARB_multitexture.

The ARB_multitexture spec is included in the OpenGL 1.2.1 spec: http://www.opengl.org/Documentation/Specs.html.

An example is on Michael Gold's Web page.

A useful snippet is available in the Advanced Graphics Programming Techniques Using

OpenGL. The Advanced99 FTP site has all source code available as a zip file.

21.100 How can I perform light mapping?

You can simulate lighting by creating a texture map that mimics the light pattern and by applying it as a texture to the lit surface. After you've created the light texture map, there's nothing special about how you apply it to the surface. It’s just like any other texture map. For this reason, this question really isn't specific to OpenGL.

The GLUT 3.7 distribution contains an example that uses texture mapping to simulate lighting called progs/advanced97/lightmap.c.

21.110 How can I turn my files, such as GIF, JPG, BMP, etc. into a texture map?

OpenGL doesn't provide support for this. With whatever libraries or home−brewed code you desire to read in the file, then by using the glTexImage2D call, transform the pixel data into something acceptable, and use it like any other texture map.

Source code for doing this with TGA files can be found here.

See the Miscellaneous section for info on reading and writing 2D image files.

21.120 How can I render into a texture map?

With OpenGL 1.1, you can use the glCopyTexImage2D() or glCopyTexSubImage2D() functions to assist with this task. glCopyTexImage2D() takes the contents of the framebuffer and sets it as the current texture map, while glCopyTexSubImage2D() only replaces part of the current texture with the contents of the framebuffer. There's a GLUT 3.7 example called multispheremap.c that does this.

21.130 What's the maximum size texture map my device will render hardware accelerated?

A good OpenGL implementation will render with hardware acceleration whenever possible. However, the implementation is free to not render hardware accelerated. OpenGL doesn't provide a mechanism to ensure that an application is using hardware acceleration, nor to query that it's using hardware acceleration. With this information in mind, the following may still be useful:

You can obtain an estimate of the maximum texture size your implementation supports with the following call:

```
GLint texSize;
glGetIntegerv(GL_MAX_TEXTURE_SIZE, &texSize);
```

If your texture isn't hardware accelerated, but still within the size restrictions returned by GL_MAX_TEXTURE_SIZE, it should still render correctly.

This is only an estimate, because the glGet*() function doesn't know what *format*, *internalformat*, *type*, and other parameters you'll be using for any given texture. OpenGL 1.1 and greater solves this problem by allowing texture proxy.

Here's an example of using texture proxy:

```
glTexImage2D(GL_PROXY_TEXTURE_2D, level, internalFormat,
        width, height, border, format, type, NULL);
```

Note the *pixels* parameter is NULL, because OpenGL doesn't load texel data when the *target* parameter is GL_PROXY_TEXTURE_2D. Instead, OpenGL merely considers whether it can accommodate a texture of the specified size and description. If the specified texture can't be accommodated, the width and height texture values will be set to zero. After making a texture proxy call, you'll want to query these values as follows:

```
GLint width;

glGetTexLevelParameteriv(GL_PROXY_TEXTURE_2D, 0,
        GL_TEXTURE_WIDTH, &width);

if (width==0) {
   /* Can't use that texture */
}
```

21.140 How can I texture map a sphere, cylinder, or any other object with multiple facets?

Texture map these objects using fractional texture coordinates. Each facet of an approximated surface or object will only show one small part of the texture map. Fractional texture coordinates determine what part of the texture map is applied to which facet.

22 Performance

22.010 What do I need to know about performance?

First, read chapters 11 through 14 of the book *OpenGL on Silicon Graphics Systems*. (You should be able to find this document by searching SGI's web site.) Although some of the information is SGI machine specific, most of the information applies to OpenGL programming on any platform. It's invaluable reading for the performance−minded OpenGL programmer.

Consider a performance tuning analogy: A database application spends 5 percent of its time looking up records and 95 percent of its time transmitting data over a network. The database developer decides to tune the performance. He sits down and looks at the code for looking up records and sees that with a few simple changes he can reduce the time it'll take to look up records by more than 50 percent. He makes the changes, compiles the database, and runs it. To his dismay, there's little or no noticeable performance increase!

What happened? The developer didn't identify the bottleneck before he began tuning. The most important thing you can do when attempting to boost your OpenGL program's performance is to identify where the bottleneck is.

Graphics applications can be bound in several places. Generally speaking, bottlenecks fall into three broad categories: CPU limited, geometry limited, and fill limited.

CPU limited is a general term. Specifically, it means performance is limited by the speed of the CPU. Your application may also be bus limited, in which the bus bandwidth prevents better performance. Cache size and amount of RAM can also play a role in performance. For a true CPU−limited application, performance will increase with a faster CPU. Another way to increase performance is to reduce your application's demand on CPU resources.

A geometry limited application is bound by how fast the computer or graphics hardware can perform vertex computations, such as transformation, clipping, lighting, culling, vertex fog, and other OpenGL operations performed on a per vertex basis. For many very low−end graphics devices, this processing is performed in the CPU. In this case, the line between CPU limited and geometry limited becomes fuzzy. In general, CPU limited implies that the bottleneck is CPU processing unrelated to graphics.

In a fill−limited application, the rate you can render is limited by how fast your graphics hardware can fill pixels. To go faster, you'll need to find a way to either fill fewer pixels, or simplify how pixels are filled, so they can be filled at a faster rate.

It's usually quite simple to discern whether your application is fill limited. Shrink the window size, and see if rendering speeds up. If it does, you're fill limited.

If you're not fill limited, then you're either CPU limited or geometry limited. One way to test for a CPU limitation is to change your code, so it repeatedly renders a static, precalculated scene. If the performance is significantly faster, you're dealing with a CPU limitation. The part of your code that calculates the scene or does other application−specific processing is causing your performance hit. You need to focus on tuning this part of your code.

If it's not fill limited and not CPU limited, congratulations! It's geometry limited. The per vertex features you've enabled or the shear volume of vertices you're rendering is causing your performance hit. You need to reduce the geometry processing either by reducing the number of vertices or reducing the calculations OpenGL must use to process each vertex.

22.020 How can I measure my application's performance?

To measure an application's performance, note the system time, do some rendering, then note the system time again. The difference between the two system times tells you how long the application took to render. Benchmarking graphics is no different from benchmarking any other operations in a computer system.

Many graphics programmers often want to measure frames per second (FPS). A simple method is to note the system time, render a frame, and note the system time again. FPS is then calculated as (1.0 / elapsed_time). You can obtain a more accurate measurement by timing multiple frames. For example if you render 10 frames, FPS would be (10.0 / elapsed_time).

To obtain primitives or triangles per second, add a counter to your code for incrementing as each primitive is submitted for rendering. This counter needs to be reset to zero when the system time is initially obtained. If you already have a complex application that is nearly complete, adding this benchmarking feature as an afterthought might be difficult. When you intend to measure primitives per second, it's best to design your application with benchmarking in mind.

Calculating pixels per second is a little tougher. The easiest way to calculate pixels per second is to write a small benchmark program that renders primitives of a known pixel size.

GLUT 3.7 comes with a benchmark called progs/bucciarelli/gltest that measures OpenGL rendering performance and is free to download. You can also visit the Standard Performance Evaluation Corporation, which has many benchmarks you can download free, as well as the latest performance results from several OpenGL hardware vendors.

22.030 Which primitive type is the fastest?

GL_TRIANGLE_STRIP is generally recognized as the most optimal OpenGL primitive type. Be aware that the primitive type might not make a difference unless you're geometry limited.

22.040 What's the cost of redundant calls?

While some OpenGL implementations make redundant calls as cheap as possible, making redundant calls generally is considered bad practice. Certainly you shouldn't count on redundant calls as being cheap. Good application developers avoid them when possible.

22.050 I have (n) lights on, and when I turned on (n+1), suddenly performance dramatically drops. What happened?

Your graphics device supports (n) lights in hardware, but because you turned on more lights than what's supported, you were kicked off the hardware and are now rendering in the software. The only solution to this problem, except to use less lights, is to buy better

hardware.

22.060 I'm using (n) different texture maps and when I started using (n+1) instead, performance drastically drops. What happened?

Your graphics device has a limited amount of dedicated texture map memory. Your (n) textures fit well in the texture memory, but there wasn't room left for any more texture maps. When you started using (n+1) textures, suddenly the device couldn't store all the textures it needed for a frame, and it had to swap them in from the computer's system memory. The additional bus bandwidth required to download these textures in each frame killed your performance.

You might consider using smaller texture maps at the expense of image quality.

22.070 Why are glDrawPixels() and glReadPixels() so slow?

While performance of the OpenGL 2D path (as its called) is acceptable on many higher−end UNIX workstation−class devices, some implementations (especially low−end inexpensive consumer−level graphics cards) never have had good 2D path performance. One can only expect that corners were cut on these devices or in the device driver to bring their cost down and decrease their time to market. When this was written (early 2000), if you purchase a graphics device for under $500, chances are the OpenGL 2D path performance will be unacceptably slow.

If your graphics system should have decent performance but doesn't, there are some steps you can take to boost the performance.

First, all glPixelTransfer() state should be set to their default values. Also, glPixelStore() should be set to its default value, with the exception of GL_PACK_ALIGNMENT and GL_UNPACK_ALIGNMENT (whichever is relevant), which should be set to 8. Your data pointer will need to be correspondingly double− word aligned.

Second, examine the parameters to glDrawPixels() or glReadPixels(). Do they correspond to the framebuffer layout? Think about how the framebuffer is configured for your application. For example, if you know you're rendering into a 24−bit framebuffer with eight bits of destination alpha, your type parameter should be GL_RGBA, and your format parameter should be GL_UNSIGNED_BYTE. If your type and format parameters don't correspond to the framebuffer configuration, it's likely you'll suffer a performance hit due to the per pixel processing that's required to translate your data between your parameter specification and the framebuffer format.

Finally, make sure you don't have unrealistic expectations. Know your system bus and memory bandwidth limitations.

22.080 Is it faster to use absolute coordinates or to use relative coordinates?

By using absolute (or "world") coordinates, your application doesn't have to change the ModelView matrix as often. By using relative (or "object") coordinates, you can cut down on data storage of redundant primitives or geometry.

A good analogy is an architectural software package that models a hotel. The hotel model has

hundreds of thousands of rooms, most of which are identical. Certain features are identical in each room, and maybe each room has the same lamp or the same light switch or doorknob. The application might choose to keep only one doorknob model and change the ModelView matrix as needed to render the doorknob for each hotel room door. The advantage of this method is that data storage is minimized. The disadvantage is that several calls are made to change the ModelView matrix, which can reduce performance. Alternatively, the application could instead choose to keep hundreds of copies of the doorknob in memory, each with its own set of absolute coordinates. These doorknobs all could be rendered with no change to the ModelView matrix. The advantage is the possibility of increased performance due to less matrix changes. The disadvantage is additional memory overhead. If memory overhead gets out of hand, paging can become an issue, which certainly will be a performance hit.

There is no clear answer to this question. It's model− and application−specific. You'll need to benchmark to determine which method is best for your model or application.

22.090 Are display lists or vertex arrays faster?

Which is faster varies from system to system.

If your application isn't geometry limited, you might not see a performance difference at all between display lists, vertex arrays, or even immediate mode.

22.100 How do I make triangle strips out of triangles?

As mentioned in 22.030, GL_TRIANGLE_STRIP is generally recognized as the most optimal primitive. If your geometry consists of several separate triangles that share vertices and edges, you might want to convert your data to triangle strips to improve performance.

To create triangle strips from separate triangles, you need to implement an algorithm to find and join adjacent triangles.

Code for doing this is available free on the Web. The Stripe package is one solution.

23 Extensions and Versions

23.010 Where can I find information on different OpenGL extensions?

The OpenGL extension registry is the central resource for OpenGL extensions. Also, the OpenGL org Web page maintains a lot of information on OpenGL extensions.

A list of extensions available on common consumer OpenGL devices is available.

Here's a similar list of extensions.

23.020 How will I know which OpenGL version my program is using?

It's commonplace for the OpenGL version to be named as a C preprocessor definition in gl.h. This enables your application to know the OpenGL version at compile time. To use this definition, your code might look like:

```
#ifdef GL_VERSION_1_2
   // Use OpenGL 1.2 functionality
#endif
```

OpenGL also provides a mechanism for detecting the OpenGL version at run time. An app may call glGetString(GL_VERSION), and parse the return string. The first part of the return string must be of the form [major−number].[minor−number], optionally followed by a release number or other vendor−specific information.

As with any OpenGL call, you need a current context to use glGetString().

23.030 What is the difference between OpenGL versions?

In OpenGL 1.1, the following features are available:

- Vertex Arrays, which are intended to decrease the number of subroutine calls required to transfer vertex data to OpenGL that is not in a display list
- Polygon Offset, which allows depth values of fragments resulting from the filled primitives' rasterization to be shifted forward or backwards prior to depth testing
- Logical Operations can be performed in RGBA mode
- Internal Texture Formats, which let an application suggest to OpenGL a preferred storage precision for texture images
- Texture Proxies, which allow an application to tailor its usage of texture resources at runtime
- Copy Texture and Subtexture, which allow an application to copy textures or subregions of a texture from the framebuffer or client memory
- Texture Objects, which let texture arrays and their associated texture parameter state be treated as a single texture object

In OpenGL 1.2, the following features are available:

- Three−dimensional texturing, which supports hardware accelerated volume rendering
- BGRA pixel formats and packed pixel formats to directly support more external file and hardware framebuffer types

- Automatically rescaling vertex normals changed by the ModelView matrix. In some cases, rescaling can replace a more expensive renormalization operation.
- Application of specular highlights after texturing for more realistic lighting effects
- Texture coordinate edge clamping to avoid blending border and image texels during texturing
- Level of detail control for mipmap textures to allow loading only a subset of levels. This can save texture memory when high−resolution texture images aren't required due to textured objects being far from the viewer.
- Vertex array enhancements to specify a subrange of the array and draw geometry from that subrange in one operation. This allows a variety of optimizations such as pretransforming, caching transformed geometry, etc.
- The concept of ARB−approved extensions. The first such extension is GL_ARB_imaging, a set of features collectively known as the Imaging Subset, intended for 2D image processing. Check for the extension string to see if this feature is available.

OpenGL 1.2.1 adds a second ARB−approved extension, GL_ARB_multitexture, which allows multiple texture maps to be applied to a single primitive. Again, check for the extension string to use this extension.

OpenGL 1.3 adds the following features:

- New texture mapping features: compression and cube mapping, new texture environments such as add, combine, and dot3, texture border clamp, and multitexture
- Multisampling
- Matrix transpose

3Dlabs has produced an OpenGL 2.0 discussion document (2.2MB) that proposes one possible direction for a major revision to OpenGL 1.x. It makes large portions of the OpenGL pipeline programmable, and also adds some memory management and timing features.

23.040 How can I code for different versions of OpenGL?

Because a feature or extension is available on the OpenGL development environment you use for building your app, it doesn't mean it will be available for use on your end user's system. Your code must avoid making feature or extension calls when those features and extensions aren't available.

When your program initializes, it must query the OpenGL library for information on the OpenGL version and available extensions, and surround version− and extension−specific code with the appropriate conditionals based on the results of that query. For example:

```
#include <stdlib.h>
   ...
int gl12Supported;

gl12Supported = atof(glGetString(GL_VERSION)) >= 1.2;
   ...
if (gl12Supported) {
   // Use OpenGL 1.2 functionality
}
```

23.050 How can I find which extensions are supported?

A call to glGetString(GL_EXTENSIONS) will return a space−separated string of extension names, which your application can parse at runtime.

Download and run the the GLView utility to view extensions supported by your system.

23.060 How can I code for extensions that may not exist on a target platform?

At runtime, your application can inquire for the existence of a specific extension using glGetString(GL_EXTENSIONS). Search the list of supported extensions for the specific extension you're interested in. For example, to see if the polygon offset extension interface is available, an application might say:

```
#include <string.h>
   ...
const GLubyte *str;
int glPolyOffExtAvailable;

str = glGetString (GL_EXTENSIONS);
glPolyOffExtAvailable = (strstr((const char *)str, "GL_EXT_polygon_offset"
      != NULL);
```

Your application can use the extension if it's available, but it needs a fallback plan if it's unavailable (i.e., some other way to obtain the same functionality).

If your application code needs to compile on multiple platforms, it must handle a development environment in which some extensions aren't defined. In C and C++, the preprocessor can protect extension−specific code from compiling when an extension isn't defined in the local development environment. For example:

```
#ifdef GL_EXT_polygon_offset
   glEnable (GL_POLYGON_OFFSET_EXT);
   glPolygonOffsetEXT (1., 1./(float)0x10000);
#endif /* GL_EXT_polygon_offset */
```

23.070 How can I call extension routines on Microsoft Windows?

Your application may find some extensions already available through Microsoft's opengl32.lib. However, depending on your OpenGL device and device driver, a particular vendor−specific extension may or may not be present at link time. If it's not present in opengl32.lib, you'll need to obtain the address of the extension's entry points at run time from the device's ICD.

Here's an example code segment that demonstrates obtaining function pointers for the ARB_multitexture extension:

```
/* Include the header that defines the extension. This may be a vendor−spe
   .h file, or GL/glExt.h as shown here, which contains definitions for al
   extensions. */
#include "GL/glExt.h"

/* Declare function pointers */
PFNGLACTIVETEXTUREARBPROC glActiveTextureARB;
PFNGLMULTITEXCOORD2FARBPROC glMultiTexCoord2fARB;
```

```
...
    /* Obtain the address of the extension entry points. */
    glActiveTextureARB = (PFNGLACTIVETEXTUREARBPROC)
        wglGetProcAddress("glActiveTextureARB");
    glMultiTexCoord2fARB = (PFNGLMULTITEXCOORD2FARBPROC)
        wglGetProcAddress("glMultiTexCoord2fARB");
```

After you obtain the entry point addresses of the extension functions you wish to use, simply call through them as normal function pointers:

```
    /* Set texture unit 0 min and mag filters */
    (*glActiveTextureARB) (GL_TEXTURE0_ARB);
    glTexParameterf (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameterf (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
...
    /* Draw multi textured quad */
    glBegin (GL_QUADS);
      (*glMultiTexCoord2fARB) (GL_TEXTURE0_ARB, 0.f, 0.f);
      (*glMultiTexCoord2fARB) (GL_TEXTURE1_ARB, 0.f, 0.f);
        glVertex3f (32.f,32.f, 0.f);
 ...
    glEnd();
```

More information on wglGetProcAddress() is available through the MSDN documentation.

You might find it annoying to explicitly call through a function pointer. A modified version of glext.h is available that doesn't eliminate the function pointer, but hides it with the C preprocessor, allowing for more aesthetically pleasing code.

23.080 How can I call extension routines on Linux?

Like Microsoft Windows (and unlike proprietary UNIX implementations), an extension entry point may or may not be defined in the static link library. At run time, a Linux application must load the function's address, and call through this function pointer.

Linix uses the OpenGL ABI.

23.090 Where can I find extension enumerants and function prototypes?

See the OpenGL extension registry.

For specific files:

glext.h
wglext.h
glxext.h

glext.h is not a replacement for gl.h, it's a supplement. It provides interfaces for all extensions not already defined by the platform−specific gl.h. This is necessary for platforms that support multiple graphics drivers where the gl.h from a central source (e.g. Microsoft or XFree86) can't track functionality provided by more frequently updated vendor drivers.

24 Miscellaneous

24.010 How can I render a wireframe scene with hidden lines removed?

The preferred method is to render your geometry in two passes: first render it in fill mode with color set to the background color, then render it again in line mode. Use polygon offset so the lines over the polygons render correctly. The polygon offset section might be helpful to you.

Often you need to preserve a nonuniform background, such as a gradient fill or an image. In this case, execute the fill pass with glColorMask() set to all GL_FALSE, then perform the line pass as usual. Again, use polygon offset to minimize Z fighting.

24.020 How can I render rubber−band lines?

See this question in the Rasterization section.

24.030 My init code calls glGetString() to find information about the OpenGL implementation, but why doesn't it return a string?

The most likely cause of this problem is that a context hasn't been made current. An OpenGL rendering context must exist and be made current to a window for any OpenGL calls to function and return meaningful values.

24.039 Where can I find 3D model files?

As this has little to do with OpenGL, what follows is by no means an exhaustive list:

http://www.3dfiles.com/
http://www.3dcafe.org/
http://www.saturn−online.de/~cosmo/
http://www.swma.net/

You can make your own 3D models using any package you desire, and then loading the geometry file. ModelMagic3D is shareware and comes with source code. GLScene is also available.

24.040 How can I load geometry files, such as 3DS, OBJ, DEM, etc. and render them with OpenGL?

OpenGL, being a 3D graphics API, has no built−in support for reading application−specific file formats. If you're writing an application that needs to read a specific file type, you'll need to add code to support a particular file type.

Many OpenGL users already have written code to do this, and in some cases, the code is available on the Web. A few are listed here. If you can't find what you are looking for, you might try doing a Web search.

This file format information covers a variety of different file formats.

Okino's PolyTrans can convert most major 3D file formats into OpenGL C code. Demos are available on their Web site.

Crossroads can import many file formats and output the data as C/C++ compilable data that is suitable for use with vertex arrays.

3DWinOGL is shareware that reads in any file format and returns OpenGL primitive data.

If you're using 3D Studio MAX, you should see an export format called ASE, which is ASCII (i.e., large file sizes), but is very easy to parse.

The XGL file format is intended to be capable of storing all OpenGL 3D information. An open source parser and a 3DS file converter are available.

Download the GLUT source distribution and look in progs/demos/smooth. The file glm.c contains routines for reading in Wavefront OBJ files.

glElite reads DXF, ASCII, and LightWave files. Information on glElite can be found at the following addresses: http://www.helsinki.fi/~tksuoran/lw.html and http://www.cs.helsinki.fi/~tksuoran/glelite/.

3D Exploration imports and exports several different file formats, including exporting to C/C++ source.

A 3DS import library in Delphi designed for use with OpenGL can be found here.

24.050 How can I save my OpenGL rendering as an image file, such as GIF, TIF, JPG, BMP, etc.? How can I read these image files and use them as texture maps?

To save a rendering, the easiest method is to use any of a number of image utilities that let you capture the screen or window, and save it is a file.

To accomplish this programmatically, you read your image with glReadPixels(), and use the image data as input to a routine that creates image files.

Similarly, to read an image file and use it as a texture map, you need a routine that will read the image file. Then send the texture data to OpenGL with glTexImage2D().

OpenGL will not read or write image files for you. To read or write image files, you can either write your own code, include code that someone else has written, or call into an image file library. The following links contain information on all three strategies.

This file format information covers a variety of different file formats.

The Independent JPEG Group has a free library for reading and writing JPEG files.

You can save your rendering as a JPEG image file, plus load JPEG and BMP files directly into OpenGL texture objects, using the C++ mkOpenGLJPEGImage class.

Source code for reading TGA files can be found here.

The gd library lets you create JPG and PNG files from within your program.

Imlib (search the "Download" section) is a wrapper library that allows a program to write out

JPEG, GIF, PNG, and TIFF files.

An image loader library in Delphi can be found here.

24.060 Can I use a BSP tree with OpenGL?

BSP trees can be useful in OpenGL applications.

OpenGL applications typically use the depth test to perform hidden surface removal. However, depending on your application and the nature of your geometry database, a BSP tree can enhance performance when used in conjunction with the depth test or when used in place of the depth test.

BSP trees also may be used to cull non−visible geometry from the database.

When rendering translucent primitives with blending enabled, BSP trees provide an excellent sorting method to ensure back−to−front rendering.

More information on BSP trees can be found at the BSP FAQ.

24.070 Can I use an octree with OpenGL?

Yes. Nothing in OpenGL prevents you from using an octree. An octree is especially helpful when used in conjunction with occlusion culling extensions (such as HP's GL_HP_occlusion_test).

24.080 Can I do radiosity with OpenGL?

OpenGL doesn't contain any direct support for radiosity, it doesn't prevent you from displaying a database containing precomputed radiosity values.

An application needs to perform its own radiosity iterations over the database to be displayed. After sufficient color values are computed at each vertex, the application renders the database as normal OpenGL primitives, specifying the computed color at each vertex. glShadeModel() should be set to GL_SMOOTH and lighting should be disabled.

24.090 Can I raytrace with OpenGL?

OpenGL contains no direct support for raytracing.

You might want to use raytracing to produce realistic shadows and reflections. However, you can simulate in many ways these effects in OpenGL without raytracing. See the section on shadows or the section on texture mapping for some algorithms.

You can use OpenGL as part of the ray intersection test. For example, a scene can be rendered with a unique color assigned to each primitive in the scene. This color can be read back to determine the primitive intersected by a ray at a given pixel. If the exact geometry is used in this algorithm, some aliasing may result. To reduce these aliasing artifacts, you can render bounding volumes instead.

Also, by changing the viewpoint and view direction, you can use this algorithm for

intersection testing of secondary rays.

A ray tracing application might also use OpenGL for displaying the final image. In this case, the application is responsible for computing the color value of each pixel. The pixels then can be rendered as individual GL_POINTS primitives or stored in an array and displayed via a call to glDrawPixels().

24.100 How can I perform CSG with OpenGL?

The *Opengl Programming Guide, Third Edition*, describes some techniques for displaying the results of CSG operations on geometric data.

The GLUT 3.7 distribution contains an example program called csg.c that may be informative.

24.110 How can I perform collision detection with OpenGL?

OpenGL contains no direct support for collision detection. Your application needs to perform this operation itself.

OpenGL can be used to evaluate potential collisions the same way it can evaluate ray intersections (i.e., the scene is rendered from the object's point of view, looking in the direction of motion, with an orthographic projection and a field−of−view restricted to the object's bounding rectangle.) Visible primitives are potential collision candidates. You can examine their Z values to determine range.

There's a free library for collision detection called I_COLLIDE available that you might find useful.

24.120 I understand OpenGL might cache commands in an internal buffer. Can I perform an abort operation, so these buffers are simply emptied instead of executed?

No. After you issue OpenGL commands, inevitably they'll be executed.

24.130 What's the difference between glFlush() and glFinish() and why would I want to use these routines?

The OpenGL spec allows an implementation to store commands and data in buffers, which are awaiting execution. glFlush() causes these buffers to be emptied and executed. Thus, any pending rendering commands will be executed, but glFlush() may return before their execution is complete. glFinish() instructs an implementation to not return until the effects of all commands are executed and updated.

A typical use of glFlush() might be to ensure rendering commands are exected when rendering to the front buffer.

glFinish() might be particularly useful if an app draws using both OpenGL and the window system's drawing commands. Such an application would first draw OpenGL, then call glFinish() before proceeding to issue the window system's drawing commands.

24.140 How can I print with OpenGL?

OpenGL currently provides no services for printing. The OpenGL ARB has discussed a GLS stream protocol, which would enable a more common interface for printing, but for now, printing is only accomplished by system−specific means.

On a Microsoft Windows platform, ALT−PrintScreen copies the active window to the clipboard. (To copy the entire screen, make the desktop active by clicking on it, then use ALT−PrintScreen.) Then you can paste the contents of the clipboard to any 2D image processing software, such as Microsoft Paint, and print from there.

You can capture an OpenGL rendering with any common 2D image processing packages that provide a screen or window capture utility, and print from there.

Also, can print programatically using any method available on your platform. For example in Microsoft Windows, you might use glReadPixels() to read your window, write the pixel data to a DIB, and submit the DIB for printing.

This tutorial contains sample source for a program that prints OpenGL images under Microsoft Windows.

This article contains a description of how to output to a vector−based image file such as Postscript or WMF.

24.150 Can I capture or log the OpenGL calls an application makes?

IBM has a product called ZAPdb that does this. It ships with many UNIX implementations, including IBM and HP. It was available on Windows NT in the past, but its current status is unknown. A non−IBM web page appears to have ZAPdb available for download.

3dpipeline.com offers a product called GLAlanyze Pro, which captures OpenGL call traces, as well as provides other analysis features.

There's a free utility called GLTrace2, which contains capture functionality similar to ZAPdb and GLAnalyze Pro. More info on GLTrace2 can be found here.

In theory, you could code a simple library that contains OpenGL function entry points, and logs function calls and parameters passed. Name this library opengl32.dll and store it in your Windows system folder (first, be careful to save the existing opengl32.dll). This shouldn't be a difficult programming task, but it might be tedious and time consuming. This solution is not limited to Microsoft Windows; using the appropriate library name, you can code this capture utility on any platform, provided your application is linked with a dynamically loadable library.

24.160 How can I render red−blue stereo pairs?

The Viewing section contains a question on creating a stereo view, and has a link to information on creating anaglyphs. The basic idea, In OpenGL, is as follows:

1. glColorMask (GL_TRUE, GL_FALSE, GL_FALSE, GL_FALSE)
2. Assuming the red image is the left image, set the projection and model−view matrices for the left image.
3. Clear color and depth buffers, and render the left image.

4. glColorMask (GL_FALSE, GL_FALSE, GL_TRUE, GL_FALSE)
5. Set the projection and model−view matrices for the right image.
6. Clear color and depth buffers and render the right image.
7. Swap buffers.

There is a GLUT 3.7 demo that shows how to do this.

Appendix A Microsoft OpenGL Information

Submitted by Samuel Paik.

Windows Driver Development Kits

Preliminary Windows 2000 DDK

Mini Client Driver
S3Virge
[Sample Windows 2000 display driver supporting DirectDraw, Direct3D, OpenGL MCD, Video Port Extensions]

Windows Driver and Hardware Development

OpenGL for 3D Color Graphics Programming
[Summary of OpenGL support in Windows]
Driver Licensing Program for OpenGL and Direct3D
WHQL − Test Kits and Procedures
[OpenGL Conformance tests are included in the display driver tests]
GDI Display Drivers in Windows 2000
GDI Display Drivers in Windows 2000
Multimedia Components in Windows 95 and Windows 2000
Implementing Display Control Panel Extensions in Windows 95 and Windows 98
[Notes on acceptible "Wait for Vblank" usage]
Microsoft Releases New 3−D DDK
[New ICD kit announcement including SGI OpenGL improvements−−result of OpenGL truce with SGI]

Fluff articles

Industry Solutions: OpenGL Update
[Says OpenGL is important to Microsoft and that OpenGL 1.2 support will likely be available in a future Windows 2000 Service Pack]
Insider: Fixing Color Distortions in Windows 98 3D Screen Savers
Windows NT Workstation: Benchmark Results: Windows NT Workstation 4.0 Bests Unix Workstations in Two Industry−Standard Engineering Application Benchmarks
Windows NT Workstation: Windows NT Workstation and Windows 95: Technical Differences
[Windows 95 acquired OpenGL with Service Pack 1]
POCKETPC: Here Comes GAPI!
[OpenGL and DirectX are too heavyweight for CE, so yet another "Game API"]
PressPass: Microsoft Delivers Performance−Leading Version of OpenGL
[OpenGL 1.1 introduced for Windows 95 and Windows NT, 1.1 bundled with NT 4.0]
PressPass: Silicon Graphics and Microsoft Form Strategic Alliance To Define the Future of Graphics
[Fahrenheit project announcement−−goes with OpenGL truce]
PressPass: Microsoft and Silicon Graphics Define Distribution And Support of OpenGL on the Windows Platform
[Truce over OpenGL−−goes with Fahrenheit announcement. New DDK to incorporate old ICD DDK with code from SGI OpenGL]

OpenGL 3−D Graphics
[OpenGL technology brief]

MSDN Library

Platform SDK

- EMRGLSBOUNDEDRECORD − The EMRGLSBOUNDEDRECORD structure contains members for an enhanced metafile record generated by OpenGL functions. It contains data for OpenGL functions with information in pixel units that must be scaled when playing the metafile.
- EMRGLSRECORD − The EMRGLSRECORD structure contains members for an enhanced metafile record generated by OpenGL functions, It contains data for OpenGL functions that scale automatically to the OpenGL viewport.
- OpenGL
  - ♦ Legal Information
  - ♦ Overview
    - ◊ Introduction to OpenGL
      - · Primitives and Commands
      - · OpenGL Graphic Control
      - · Execution Model
      - · Basic OpenGL Operation
      - · OpenGL Processing Pipeline
        - • OpenGL Function Names
        - • Vertices
        - • Primitives
        - • Fragments
        - • Pixels
      - · Using Evaluators
      - · Performing Selection and Feedback
      - · Using Display Lists
      - · Managing Modes and Execution
      - · Obtaining State Information
      - · OpenGL Utility Library
  - ♦ Win32 Extensions to OpenGL
  - ♦ ◊ OpenGL on Windows NT, Windows 2000, and Windows 95/98
    - · Components
    - · Generic Implementation and Hardware Implementation
    - · Limitations
    - · Guide To Documentation
    - · Rendering Contexts
      - • Rendering Context Functions
    - · Pixel Formats
      - • Pixel Format Functions
    - · Front, Back, and Other Buffers
      - • Buffer Functions
    - · Fonts and Text
      - • Font and Text Functions
    - · OpenGL Color Modes and Windows Palette Management
      - • Palettes and the Palette Manager
      - • Palette Awareness

- Appendix
  - About OpenGL

OpenGL technical articles

OpenGL 1.1

[OpenGL 1.1 was first introduced into the Windows 9X line with Windows 95, OEM Service Release 2]

OpenGL I: Quick Start

This article describes GLEasy, a simple OpenGL program. OpenGL is a three−dimensional (3−D) graphics library included with the Microsoft® Windows NT® version 3.5 operating system. GLEasy is a Microsoft Foundation Class Library (MFC) application that provides a good starting point for investigations into the Windows NT implementation of OpenGL.

OpenGL II: Windows Palettes in RGBA Mode

If a program written for the Microsoft® Windows® operating system needs more than 16 colors and is running on an 8−bits−per−pixel (bpp) display adapter, the program must create and use a palette. OpenGL programs running on Windows NT® or (eventually) Windows 95 are no exception. OpenGL imposes additional requirements on the colors and their locations on the palette in RGBA mode. The articles "OpenGL I: Quick Start" and "Windows NT OpenGL: Getting Started" in the MSDN Library cover the basics of using OpenGL in a Windows−based program and are required reading for this article. Two sample applications, GLEasy and GLpal, accompany this article.

OpenGL III: Building an OpenGL C++ Class

This article discusses the development of a C++ class library for encapsulating OpenGLT code. The C++ class presented is for demonstration and educational purposes only. I will expand the class library for future OpenGL articles. The class library is not currently part of the Microsoft® Foundation Class Library (MFC), and there are no plans to add this class to MFC in the future. I assume that the reader has already read the first article in this series, "OpenGL I: Quick Start," in the MSDN Library. The class library is in the GLlib.DLL file included with this article. The EasyGL sample application, also included with this article, uses the classes in GLlib.DLL.

Color Index Mode

This article explores the Windows NTT implementation of OpenGLT color index mode. In color index mode, colors are specified as indexes into a palette instead of as levels of red, green, and blue. The EasyCI sample application (provided with this article) is a conversion of EasyGL that uses color index mode. EasyCI uses the GLlib.DLL, also included with this article.

OpenGL IV: Translating Windows DIBs

OpenGLT is a portable language for rendering three−dimensional (3−D) graphics. OpenGL does not understand Microsoft® Windows® device−independent bitmaps (DIBs); instead, it has its own format for representing images. This article explains how to translate a Windows DIB into a format usable with OpenGL. Some knowledge of the Windows DIB format and the Microsoft Foundation Class Library (MFC) is expected. The EasyDIB sample application and GLlib dynamic−link library (DLL) demonstrate the ideas presented in this article.

OpenGL VI: Rendering on DIBs with PFD_DRAW_TO_BITMAP

The PFD_DRAW_TO_BITMAP pixel format descriptor flag allows OpenGLT applications to render on a Microsoft® Windows® device−independent bitmap (DIB). The resulting DIB can be manipulated to the full extent using the commands in the Windows graphics device interface (GDI). This article explains how you can render OpenGL scenes on DIBs with PFD_DRAW_TO_BITMAP. The EasyBit sample application demonstrates the techniques presented in the article.

OpenGL VII: Scratching the Surface of Texture Mapping

This article explains how to apply bitmaps to OpenGLT surfaces to give them a realistic appearance. The bitmaps are known as *textures* and can resemble wood, marble, or any other interesting material or pattern. The process of applying or mapping a texture to a surface is known as *texture mapping*.

The EasyTex and PicCube sample applications demonstrate the concepts discussed in this article.

OpenGL VIII: wglUseFontOutlines

This article explains how to use the Win32® **wglUseFontOutlines** function. This function creates three−dimensional (3−D) characters based on a TrueType® font for use in OpenGLT−rendered scenes. The EasyFont sample application demonstrates using **wglUseFontOutlines**.

Windows NT OpenGL: Getting Started

OpenGL, an industry−standard three−dimensional software interface, is now a part of Microsoft® Windows NTT version 3.5. As a hardware−independent interface, the operating system needs to provide pixel format and rendering context management functions. Windows NT provides a generic graphics device interface (GDI) implementation for this as well as a device implementation. This article details these implementations, OpenGL/NT functions, and tasks that applications need to accomplish before OpenGL commands can be used to render images on the device surface.

CUBE: Demonstrates an OpenGL Application

CUBE is a simple OpenGLT application. It demonstrates how to integrate OpenGL with the MFC single document interface (SDI), and how OpenGL's resource contexts are used in conjunction with device contexts.

OPENGL: Demonstrates Using OpenGL

This sample creates a control that draws a spinning cube using the OpenGL graphics library. [Uses ATL: Active Template Library]

OpenGL Without the Pain: Creating a Reusable 3D View Class for MFC

DirectX 6.0 Goes Ballistic With Multiple New Features And Much Faster Code

Get Fast and Simple 3D Rendering with DrawPrimitive and DirectX 5.0

February 97 Microsoft Interactive Developer Column: Fun and Games

[claims OpenGL will be based on Direct3D Immediate Mode in the future−−I believe this work on this ended some time ago, may eventually be revived]

Poking Around Under the Hood: A Programmer's View of Windows NT 4.0

[What's new with Windows NT 4.0, including WGL (very misleading information)]

Windows NT Resource Kit: Registry Value Entries: Video Device Driver Entries

[OpenGL registry keys, among others]

Windows NT Resource Kit: Dynamic Link Library Files

[Annotated list of system DLLs]

DirectX Developer FAQ

[Notes that the DX7 Direct3D lighting model was changed to match OpenGL lighting]

Useful other articles

DIBs and Their Use

This article discusses the DIB (device−independent bitmap) concept from definition and structure to the API that uses it. Included is a small sample application that illustrates some of the most common methods of using DIBs to display and manipulate digital images. Functions discussed are **GetDIBits**, **SetDIBits**, **CreateDIBitmap**, **SetDIBitsToDevice**, **StretchDIBits**, and **CreateDIBPatternBrush**. This article does not discuss using palettes with DIBs.

Using DIBs with Palettes

This article discusses using palettes in conjunction with DIBs (device−independent bitmaps). It does not delve into involved uses of the Microsoft® WindowsT Palette Manager.

Creating Programs Without a Standard Windows User Interface Using Visual C++ and MFC

Microsoft® Visual C++T and the Microsoft Foundation Class Libraries (MFC) provided a very fast way to get a standard WindowsT−based application up and running. But what if you don't want the normal look and feel? Many games and educational applications have special user interface needs that can't be met with the standard Windows user interface. This article takes a look at creating a simple child's coloring game that uses only a single window and has no window border, caption,

buttons, cursor, or any other recognizable elements of a Windows user interface.

Knowledge Base

Current

Q254265 − 'Advanced' Button Under 'Display' Does Not Work After Installation of Windows NT 4.0 Drivers in Windows 2000

[Windows 2000] After you upgrade from Microsoft Windows NT 4.0 to Microsoft Windows 2000, or after you install Windows NT 4.0 drivers in Windows 2000, and you click the Advanced button on the Settings tab under Display in Control Panel, you may receive an error message.

Q253521 − INFO: OpenGL Drivers

OpenGL drivers have traditionally been provided by the hardware vendors who provide the 3D adapter in your computer.

Q247438 − OpenGL Support Not Available on nVidia TNT2 Card in Microsoft Windows 2000

[Windows 2000] When you attempt to play a game that requires support for the OpenGL standard (for three−dimensional graphics display) on a Microsoft Windows 2000−based computer, the game does not run. [ed note: Microsoft does not ship display drivers with OpenGL support with Windows 2000]

Q240896 − OpenGL Program May Cause an Invalid Page Fault Error Message if the Window is Moved or Resized

[Windows 95, 98, 98SE] When you move or resize a window, a program that uses OpenGL may perform an illegal operation, and then shutdown. For example, Microsoft Internet Explorer may generate an invalid page fault if a Java tool using OpenGL is running, and the window displaying the OpenGL graphic content is moved. Also, the following message may be generated in the Details section of the Application error dialog box:

Q233390 − BUG: First Chance Exceptions When Calling ChoosePixelFormat

[Windows 95, 98] The following error is displayed in the debug window of Visual C++: First−chance exception in myapp.exe (GDI32.DLL): 0xC0000005: Access Violation.

Q228099 − PRB: wglUseFontOutlines Does Not Handle DBCS

[Windows 98, NT 4.0] On Windows 98, the OpenGL function wglUseFontOutlines does not work with DBCS or UNICODE strings. On Windows NT, UNICODE strings work; however, DBCS strings do not.

Q227279 − OpenGL Screen Saver Prevents Power Management Standby Mode

[Windows 2000] When you configure your computer to use an OpenGL screen saver and the System Standby feature in Advanced Power Management (APM), your computer may not start the Standby mode.

Glide API Features Disabled on Video Adapter

[Windows NT 4.0; I don't see why this doesn't affect Windows 9X or Windows 2000. The description is confused] After you install Windows NT 4.0 Service Pack 4 on a computer with a proprietary 3Dfx function library file (such as the 3dfxgl.dll file installed during the installation of id Software's Quake II), you may not be able to access your video adapter's support for 3Dfx graphics.

Windows 98 Components for Typical, Portable and Compact Setup

[Lists components installed, OpenGL is not installed in "Compact" installation]

Q176752 − Glen.exe Shows How to Enumerate Pixel Formats in OpenGL

The GLEnum sample provides a demonstration of how to enumerate pixel formats and method for checking the available pixel formats provided on your machine. The GLEnum sample is included in Glen.exe.

GLEN.EXE: SAMPLE: Pixel Format Enumeration in OpenGL Demo

Q169954 − INFO: Layer Planes in OpenGL

Layer Planes are a new feature in the Microsoft implementation of OpenGL 1.1. Before using OpenGL layer planes, there are several new functions and some driver dependency issues that you should be aware of.

Q160817 − Demonstrates OpenGL Texture−Mapping Capabilities

GLTEXTUR.EXE provides a demonstration of how to use a Device−independent Bitmap (DIB) as a texture−map for OpenGL by pasting a DIB (chosen by the user) onto three different OpenGL objects.
GLTEXTUR.EXE: SAMPLE: Demonstrates OpenGL Texture−Mapping Capabilities

Q154877 − OpenGL 1.1 Release Notes & Components

Opengl95.exe contains the release notes for OpenGL version 1.1 for Windows 95 and all of the components associated with OpenGL such as the DLL, library, and include files.
Note that Windows 95 OSR2, Windows 98, and Windows NT already include OpenGL with the O.S., so this download is not necessary (or recommended) for those platforms
OPENGL95.EXE

Q152001 − GLLT.EXE Demonstrates Simple Lighting in OpenGL

The GLLight sample provides a demonstration of how the various light settings effect an OpenGL scene. The initial scene is simply a single white sphere with a single blue light (GL_LIGHT0) shining on it.

Q151489 − INFO: When to Select and Realize OpenGL Palettes

An OpenGL application must select and realize its palette before setting the current rendering context with wglMakeCurrent.

Q148301 − GLTex Demos How to Use DIBs for Texture Mapping

The GLTex sample provides a demonstration of how to use a DIB (device− independent bitmap) as a texture−map for OpenGL by pasting a DIB (chosen by the user) onto all sides of a three−dimensional cube. [Appears to have been superceded by Q160817, code no longer here.]

Q139967 − GLEXT: Demo of GL_WIN_swap_hint & GL_EXT_vertex_array

The GLEXT sample illustrates how to use the GL_WIN_swap_hint extension to speed up animation by reducing the amount of repainting between frames and how to use GL_EXT_vertex_array extension to provide fast rendering of multiple geometric primitives with one glDrawArraysEXT call. It also shows how to use glPixelZoom and glDrawPixels to display an OpenGL bitmap.

Q139653 − PRB: Antialiased Polygons Not Drawn in OpenGL Antipoly Sample

The antipoly sample in OpenGL SDK BOOK directory is unable to draw antialised polygons with the generic implementation of Windows NT and Windows 95 OpenGL.

Q136266 − Demonstration of OpenGL Material Property and Printing

The GLBMP sample illustrates how to define the material properties of the objects in the scene: the ambient, diffuse, and specular colors; the shininess; and the color of any emitted lights. This sample also demonstrates how to print an OpenGL image by writing the OpenGL image into a DIB section and printing the DIB section. The current version of Microsoft's implementation of OpenGL in Windows NT does not provide support for printing. To work around this current limitation, draw the OpenGL image into a memory bitmap, and then print the bitmap.
GLBMP.EXE: Sample: OpenGL Material Property & Printing

Q131130 − HOWTO: Set the Current Normal Vector in an OpenGL Application

[Information on using the cross product to obtain a normal vector for a polygon]

Q131024 − Drawing Three−Dimensional Text in OpenGL Appliations

GDI operations, such as TextOut, can be performed on an OpenGL window only if the window is single−buffered. The Windows NT implementation of OpenGL does not support GDI graphics in a double−buffered window. Therefore, you cannot use GDI functions to draw text in a double−buffered window, for example. To draw text in a double−buffered window, an application can use the wglUseFontBitmaps and wglUseFontOutlines functions to create display lists for characters in a font, and then draw the characters in the font with the glCallLists function.
The **wglUseFontOutlines** function is new to Windows NT 3.51 and can be used to draw 3−D characters of TrueType fonts. These characters can be rotated, scaled, transformed, and viewed like

any other OpenGL 3−D image. This function is designed to work with TrueType fonts.
The GLFONT sample shows how to use the **wglUseFontOutlines** function to create display lists for characters in a TrueType font and how to draw, scale, and rotate the glyphs in the font by using glCallLists to draw the characters and other OpenGL functions to rotate and scale them. You need the Win32 SDK for Windows NT 3.51 to compile this sample, and you need to incorporate **wglUseFontOutlines** in your own application. You also need Windows NT 3.51 to execute the application.
GLFONT.EXE: Sample: Drawing 3−D Text in an OpenGL App

Q127071 − MFCOGL a Generic MFC OpenGL Code Sample
Microsoft Windows NT's OpenGL can be used with the Microsoft Foundation Class (MFC) library. This article gives you the steps to follow to enable MFC applications to use OpenGL.
MFCOGL.EXE: Code Sample Demonstrates Using OpenGL with MFC

Q128122 − Implementing Multiple Threads in an OpenGL Application
It is possible to create multiple threads in an OpenGL application and have each thread call OpenGL functions to draw an image. You might want to do this when multiple objects need to be drawn at the same time or when you want to have certain threads perform the rendering of specific types of objects.
GLTHREAD.EXE: SAMPLE: Using Multiple Threads in OpenGL App

Q126019 − PRB: Most Common Cause of SetPixelFormat() Failure
SetPixelFormat() fails with incorrect class or window styles. [I'm not convinced this is the **most** common cause today.]

Q124870 − XFONT.C from SAMPLES\OPENGL\BOOK Subdirectory
XFONT.C from the SAMPLES\OPENGL\BOOK subdirectory is not in the MAKEFILE, and subsequently is never built.

OPENGL3.EXE: MSJ Source: Feb '95: OPENGL3.EXE
[The associated KB article Q124/2/06 has disappeared. This code apparently went with the Microsoft Systems Journal "Understanding Modelview Transformations in OpenGL for Windows NT"]

Q124034 − OpenGL Interface in Windows NT 3.5
This article defines and explains the OpenGL interface that is available and can be implemented in Windows NT version 3.5.

Q121381 − Microsoft Systems Journal: November 1994
This article lists the filenames and Snumbers for files available from online services that contain the source code described in articles published in the November 1994 issue of the "Microsoft Systems Journal."
CUBES.EXE: MSJ Source: Nov, 1994 cubes.exe
[This code apparently went with the Microsoft Systems Journal article introducing OpenGL with Windows NT 3.5: "3−D Graphics for Windows NT 3.5. Introducing the OpenGL Interface, Part II."]

Q121282 − OPENGL Screen Savers May Degrade Server Performance
If OPENGL screen savers are used on a Windows NT Server, network server performance (the Server's responsiveness to clients) may be degraded while the screen saver is running.

OPENGL.EXE: MSJ Source: Oct, 1994 opengl.exe
[Associated KB article Q119/8/62 appears to have disappeared. This code apparently went with the Microsoft Systems Journal article introducing OpenGL with Windows NT 3.5: "3−D Graphics for Windows NT 3.5. Introducing the OpenGL Interface, Part I."]

Archive

Q224792 − List of Bugs Fixed in Windows NT 4.0 Service Pack 1, 2, and 3
Err Msg: STOP 0x00000050 PAGE_FAULT_IN_NONPAGED_AREA
[Windows NT 4.0] When you run NetMeeting with sharing enabled, you may receive the following error message on a blue screen if you restart your computer and start NetMeeting again:

Q191359 − SMS: Windows 95 OpenGL Screen Saver May Cause Computer to Stop
[Windows 95 OSR2] Computers that are running Microsoft Windows 95 may lose their ability to safely shut down after the OpenGL or Mystify Your Mind screen saver is started and stopped several times. This may occur on computers that have the ATI 64 and ATI Rage Series video adapters installed.

Q189979 − OpenGL−Based Programs Do Not Work After Upgrade to Windows 98
[Windows 98] After you upgrade to Windows 98, your OpenGL−based programs may no longer work correctly, or may not work at all.

Q166334 − OpenGL Access Violation on Windows NT Version 4.0
[Windows NT 4.0] Under heavy stress, OpenGL applications may experience access violations. Also, OpenGl Line and Polygon texture clipping functions may fail when fogging is enabled.

Q166257 − Applications Using OpenGL Cause Access Violation in OPENGL.DLL
[Windows NT 4.0] A multi−threaded or multi−windowed application that uses OpenGL may cause an access violation in the Opengl.dll library.

Q166198 − Display Color Problem with OpenGL Applications in Windows NT 4.0 Service Pack 2
[Windows NT 4.0 SP2] After you apply Windows NT 4.0 Service Pack 2, coloring problems may occur with OpenGL applications where the wrong colors are drawn in a wide variety of situations. [See Q163677]

Q164158 − OpenGL Diffuse Settings Revert to Default
[Windows NT 4.0] When using OpenGL with Windows NT, the diffuse parameter changes back to the default when the color material changes from AMBIENT_AND_DIFFUSE to AMBIENT.

Q163677 − BUG: OpenGL Color Problems Using Service Pack 2 for Win NT 4.0
[Windows NT 4.0 SP2] When you use Service Pack 2 for Windows NT 4.0, various coloring problems may arise that are not present in previous versions. The coloring problems involve drawing the wrong colors in a variety of situations.

GLSP2FIX.EXE: BUG: OpenGL Color Problems Using Service Pack 2 for Win NT 4.0
Q160651
[pre−Windows NT 4.0 SP2] An application that uses OpenGL may crash with an exception 0xC0000090.

Q159129 − OpenGL Access Violation with Invalid OpenGL Context
[Pre−Windows NT 4.0 SP2] The API gluGetString causes an access violation and affects OpenGL operations.

Q156473 − BUG: Windows NT Version 4.0 Bug List − GDI
[Windows NT 4.0. Known bugs at time of release]

Q152841 − Windows NT 4.0 Service Pack 3 Readme.txt File (40−bit)
Q147798 − Windows NT 4.0 Service Pack 3 Readme.txt File (128−bit)
Access Violation in glsbCreateAndDuplicateSection API on PowerPC
[Windows NT 3.51 for PowerPC] When you install a OpenGL client video driver on your PowerPC computer running Windows NT and you run an OPENGL program, for example, the Windows NT Pipes screen saver, an access violation occurs in the glsbCreateAndDuplicateSection application programming interface (API).

Q134893 − 3D OpenGL Screen Saver Restores Windows NT 3.51 Help
[Windows NT 3.51] When you return to your desktop from any of Windows NT 3D OpenGL screen savers, any minimized Windows NT 3.51 Help files that use the Windows 95 Help engine are restored to full size.

Q134765 − Unknown Software Exception When Application Calls OpenGL
[Windows NT 3.51] An unknown software exception occurs when applications call OpenGL. When Windows NT attempts to shutdown the computer, a blue screen appears.

Q133322 − List of Confirmed Bugs in Windows NT Version 3.51
Q133220 − List of Confirmed Bugs in Windows NT Version 3.5
Q132866 − DOCERR: Printing an OpenGL Image

The documentation relating to printing an OpenGL image in the Win32 SDK versions 3.5, 3.51, and 4.0 is incorrect. The current version of Microsoft's implementation of OpenGL in Windows NT does not provide support for printing. More specifically, an application cannot call wglCreateContext or wglMakeCurrent on a printer device context.

Q132748 − Choosing a Workstation OS: Windows 95/Windows NT Workstation

Q128531 − README.TXT: Windows NT Version 3.51 U.S. Service Pack

Snow/White Noise with Mach 32 at 1024x768 − 65536 colors

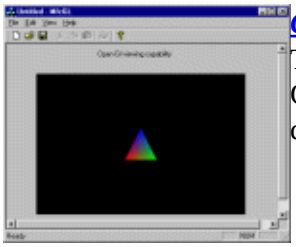
[Windows NT 3.5] When you use the ATI Mach 32 video adapter driver included with Windows NT version 3.5, white haze (also known as snow) may appear when you move windows on the desktop. This problem can also occur when you use the 3D Pipes (OpenGL) screen saver.

Q126128 − Message Popup Changes Color When Using OpenGL Screen Saver

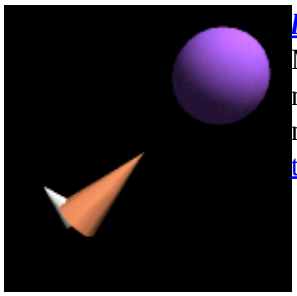
[Windows NT 3.5] When you run Windows NT with a 800 x 600 (256 color) or 1024 x 768 (256 color) video driver and test an OpenGL screen−saver, the Title Bar and OK button in the Messenger Service dialog box are red.

Appendix B Source Code Index

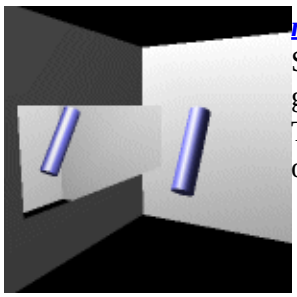
weight.cpp
This code snippet from Ron Fosner shows how to pick a pixel format based on a weighting scheme, and more importantly, how to force selection of a software pixel format. For discussion on selecting a software−only pixel format, see question 5.030.



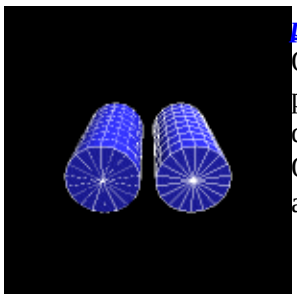
GlView.zip
This code demonstrates use of OpenGL and MFC. OpenGL is rendered into a CStatic form control. For more information on using OpenGL with MFC, see questions 5.150, 5.160, 5.170, and 5.180.



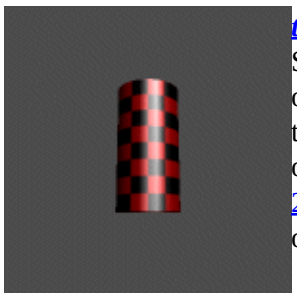
lookat.cpp
Many new OpenGL programmers are also new to linear algebra, and manipulating matrices can present a challenge. This code shows how to create a transformation matrix that will make an object point in a given direction. Section 9 on transformations may also be helpful.



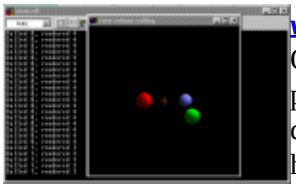
mirror.c
Stencil planes can be used to render mirrors in OpenGL, but because many low−end graphics devices do not support them efficiently, using stencil planes is not practical. This code demonstrates how to use the depth buffer to render mirrors. An overview of the technique can be found in question 9.170.



pgonoff.c
OpenGL provides the polygon offset feature to allow rendering of coplanar primitives, and especially coplanar lines or edges over polygons. This code demonstrates correct use of the OpenGL 1.1 polygon offset interface, as well as the OpenGL 1.0 polygon offset extension interface. See section 13 on polygon offset, and section 23 on extensions for more information.



twopass.cpp
Since GL_MODULATE texture environment mode multiplies color values, obtaining white specular highlights on texture mapped objects requires special techniques. This code demonstrates a portable two−pass method, and also shows use of HP's pre−specular extension on platforms that support it. Question 21.040 discusses the issues involved in specular highlights on texture mapped objects.



viewcull.c

OpenGL clips geometry to the view volume a single vertex at a time. For optimum performance, an application must "bulk cull" large amounts of geometry. This code demonstrates how to obtain object space plane equations for the view volume, and how to clip test bounding boxes against them. Section 10 on clipping contains more information.

Appendix C History

In March of 2001, Tom Impelluso started a thread on the history of OpenGL. In the near future, I will write this into a more formal FAQ article. In the meantime, you can review the thread through March 8th, 2001.

Allen Akin, who has been both a manager of a PEX development group at DEC and of an OpenGL development group at SGI, contributed his historic whitepaper to this FAQ, which compares the two APIs: Analysis of PEX 5.1 and OpenGL 1.0 from SIGGRAPH 1992.