PUBLISHED IN THE
NOVEMBER/DECEMBER 1994 ISSUE OF *The X Journal*.

OpenGL™ and X, Column 1: An OpenGL Toolkit

Mark J. Kilgard *
Silicon Graphics Inc.
Revision : 1.8

December 17, 1996

A much more complete introduction to GLUT can be found in my book titled **Programming OpenGL for the X Window System** *(Addison-Wesley, ISBN 0-201-48359-9). Chapter 4 contains an expanded tutorial covering the entire GLUT 3 programming interface. Chapter 5 explains numerous interesting GLUT-based OpenGL programs. An appendix provides a complete reference for the GLUT programming interface.*

Welcome to the "OpenGL and X" column. This column is the outgrowth of my three-part series on programming OpenGL with the X Window System. The purpose of the column is to continue explaining how to put interactive 3D graphics into your X programs.

In the past year, there's been remarkable progress adopting OpenGL as the premier Application Programming Interface (API) for interactive 3D graphics. Major workstation vendors are supplying OpenGL for their X workstations: Digital, IBM, and Silicon Graphics. Independent software vendors (ISVs) have or are readying ports of OpenGL for Sun and Hewlett-Packard workstations. And OpenGL is not limited to the X world. OpenGL is also supported in a new version of IBM's OS/2 and in Microsoft's Windows NT 3.5 (code named Daytona). A very exciting development is the announcement of custom graphics chips designed specifically for OpenGL rendering. These chips, like the GLint chip from 3Dlabs, promise to make OpenGL graphics inexpensive and accessible.

These developments point the way for OpenGL to be the catalyst that brings interactive 3D graphics into the computing mainstream. The past decade has shown that it takes a technically mature software interface with broad-based support to launch new computer technologies. For example, PostScript has changed the way we all think about computer generated hardcopy. And the TCP/IP networking protocols have opened the Internet to the world. PostScript and TCP/IP didn't invent laser printing or networking, but these two standards made possible the widespread adoption of the technologies they enabled. OpenGL will do the same for 3D graphics.

This column will assume some familiarity with OpenGL. For readers who need this background, I encourage you to read *The X Journal* back issues containing the original "OpenGL and X" series. Also you should be able to find *The OpenGL Programming Guide* and *The OpenGL Reference Manual* (often called the "red" and "blue" books respectively because of the color of their covers) in most computer literate bookstores.

Because OpenGL is window-system independent (the OpenGL API can work just as well for X as it does for Windows NT or OS/2), an OpenGL program uses the window management functions of its host window system. In the "red" book, OpenGL examples were presented using the AUX library. AUX is a window-system independent toolkit for using OpenGL. The AUX interface lets you open a single window, render OpenGL, and handle basic input, but that is about the extent of its functionality. It's good for examples but not appropriate for anything much more sophisticated.

Two alternatives to AUX are using Motif or Xlib to write your OpenGL applications. Either alternative works. In practice, Motif is likely to be the window system API most X programmers will use for sophisticated OpenGL applications. Unfortunately, both Motif and Xlib are rather complex. Many programmers wishing to explore 3D graphics with OpenGL will want something in between the toyish feel of AUX and the complexity of Motif or Xlib. That's where this issue's column helps out: supplying a reasonable toolkit to explore OpenGL. Future columns will use the described toolkit to demonstrate specific OpenGL functionality like lighting and texturing.

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

The toolkit I describe is named GLUT, short for the open**GL** **U**tility **T**oolkit. The API for GLUT is designed to be simple and straightforward. Unlike more complex APIs like Xlib or Motif, very little setup is needed to begin displaying graphics. Also the GLUT API avoids obvious window system dependencies. The intent is that a GLUT program could be recompiled for OS/2 or NT if a GLUT implementation for those window systems was available. An implementation of GLUT for X is available through the Internet.[1]

So what does GLUT do for you? GLUT supports the following functionality:

- Multiple windows for OpenGL rendering.
- Callback driven event processing.
- An "idle" routine and timers (like X Toolkit work procedures and timeouts).
- Standard X command line argument processing.
- A simple pop-up menu facility.
- Miscellaneous window management functions.

All the routines in the API begin with `glut`; constants begin with `GLUT_`. This article won't describe the entire API, just the basics you'll need to begin using GLUT to explore OpenGL.

GLUT does have limitations and isn't suitable for every purpose, but I think you'll find it a reasonable vehicle to explore 3D graphics using OpenGL without getting bogged down in writing X or Motif code. 3D should be fun; not a chore.

A Short GLUT Example

Figure 1 is a simple GLUT program that draws a lighted sphere. The program merely renders the sphere when the window needs redrawing. But the example demonstrates the basics of opening a window with GLUT and rendering an image into it.

Briefly, I'll explain each GLUT call in the example.

```
glutInit(&argc, argv);
```

This routine initializes GLUT. Most importantly, it processes any command line arguments GLUT understands (for X, this would be options like `-display` and `-geometry`). Any command line arguments recognized by GLUT are stripped out, leaving the remaining options for your program to process. Every GLUT program should call `glutInit` before any other GLUT routine.

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

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

GLfloat light_diffuse[] = { 1.0, 0.0, 0.0, 1.0 };
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
GLUquadricObj *qobj;

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glCallList(1); /* render sphere display list */
    glutSwapBuffers();
}

void gfxinit(void)
{
    qobj = gluNewQuadric();
    gluQuadricDrawStyle(qobj, GLU_FILL);
    glNewList(1, GL_COMPILE); /* create sphere display list */
    gluSphere(qobj, /* radius */ 1.0,
        /* slices */ 20, /* stacks */ 20);
    glEndList();
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_DEPTH_TEST);
    glMatrixMode(GL_PROJECTION);
    gluPerspective( /* field of view in degrees */ 40.0,
        /* aspect ratio */ 1.0,
        /* Z near */ 1.0, /* Z far */ 10.0);
    glMatrixMode(GL_MODELVIEW);
    gluLookAt(0.0, 0.0, 5.0,   /* eye is at (0,0,5) */
              0.0, 0.0, 0.0,   /* center is at (0,0,0) */
              0.0, 1.0, 0.);   /* up is in +Y direction */
    glTranslatef(0.0, 0.0, -1.0);
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutCreateWindow("sphere");
    glutDisplayFunc(display);
    gfxinit();
    glutMainLoop();
}
```

Figure 1: Program using GLUT to render a lighted sphere using OpenGL. The GLUT specific lines are in bold; notice that very little GLUT-code is required.

When a window is created, its type is determined by GLUT's current *display mode*. The display mode is a set of indicators that determine the frame buffer capabilities of the window. The call to `glutInitDisplayMode` means a subsequently created window should be created with the following capabilities:

- Double buffering.
- An RGB color model (*TrueColor*).
- A depth buffer or Z buffer (for hidden surface elimination).

Other indicators like `GLUT_STENCIL` could be combined into the display mode value to request additional capabilities like a stencil buffer.

```
glutCreateWindow("sphere");
```

Create a window for OpenGL rendering named "sphere." If a connection
to the X server hasn't yet been made, `glutCreateWindow` will do so and make sure that OpenGL is supported by the X server. Details such as picking the correct visual and colormap and communicating information like the window name to the window manager are handled by GLUT. Along with the window, an associated OpenGL rendering context is created. This means each window has its own private set of OpenGL state.

[1] To obtain GLUT, use anonymous `ftp` to `sgigate.sgi.com` and retrieve the files in the `pub/opengl/xjournal/GLUT` directory. The library source code, documentation, and example source code are all provided. You'll need an OpenGL development environment to compile and use GLUT.

While this example doesn't use it, `glutCreateWindow` actually returns an integer identifier for the window. For an application with multiple windows, this lets you control multiple windows. Because of GLUT's design, you will rarely need a window's identifier. This is because GLUT keeps a *current window* as part of its state. Most window operations implicitly affect the current window. When a window is created, the current window is implicitly set to the new window.

For example:

```
glutDisplayFunc(display);
```

will register the `display` function as the routine to be called when the current (and just created) window needs to be drawn.

Also notice that OpenGL routines that initialize OpenGL's state in `gfxinit()` implicitly affect the OpenGL context for the current window.

If you do need to change the current window, you can call `glutSetWindow(winnum)`, where `winnum` is a window identifier. You can also call `glutGetWindow()` that returns the current window identifier.

Whenever a callback is made for a specific window, the current window is implicitly set to the window prompting the callback. So when the `display` callback registered with `glutDisplayFunc` is called, we know that GLUT has implicitly changed the current window to the window that needs to be redisplayed. This means `display` can call OpenGL rendering routines and the routines will affect the correct window.

```
glutMainLoop();
```

This routine serves the same purpose as the X Toolkit's `XtAppMainLoop` routine; it begins event processing and maps any windows that have been created. It never exits. Callback functions registered with GLUT are called as necessary. For example, the `display` routine in the example will be called whenever `Expose` events are received by the program from the X server.

```
glutSwapBuffers();
```

When we created the window, the display mode was set to request a double buffered window. After `display` renders the sphere using OpenGL routines, `glutSwapBuffers` is called to swap the current window's buffers to make the image rendered into the back buffer visible. So the sphere is displayed without the program's user seeing the rendering in progress.

Supporting more than one window is easy. To create a second window with the same red sphere add the following calls before `glutMainLoop`:

```
glutCreateWindow("a second window");
glutDisplayFunc(display);
gfxinit();
```

Because of the implicit update of the current window, `display` will also be registered for the second window and `gfxinit` will initialize the second window's OpenGL context. And remember that when a callback like `display` is called, the current window is implicitly set to the window needing the callback, so the rendering done by `display` will be directed into the correct window. The result of the output of the resulting program can be seen in Figure 2.

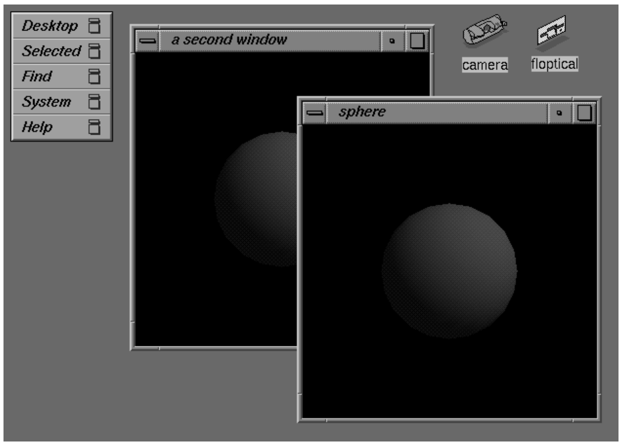

Figure 2: The two windows generated by the modified version of code in Figure 1.

User Input and Other Callbacks

An interactive 3D program needs a way to get input from the user. GLUT provides callback routines to be registered for each window for keyboard, mouse button, and mouse motion input.

```
glutKeyboardFunc(keyboard);
```

This routine registers a callback for keyboard input for the current window. The function `keyboard` might be written like this:

```
void keyboard(unsigned char key,
  int x, int y)
{
  printf("key `%c' pressed at (%d,%d)\n",
    key, x, y);
}
```

If the `keyboard` callback is registered for a window, the `keyboard` routine will be called when a key is pressed in the window. The ASCII character generated by the key press is passed in along with the location of the cursor within the window (relative to an origin at the upper left hand corner of the window).

```
glutMouseFunc(mouse);
glutMotionFunc(motion);
```

These routines register callbacks for mouse button changes and mouse motion (when buttons are down) for the current window. The following are example callbacks:

```
void mouse(int btn, int state,
  int x, int y)
{
  printf("button %d is %s at (%d,%d)\n",
    btn, state == GLUT_DOWN ? "down" : "up",
    x, y);
}

void motion(int x, int y)
{
  printf("button motion at (%d,%d)\n",
    x, y);
}
```

The callbacks that can be registered with GLUT are not limited to input devices. The already introduced glutDisplayFunc routine lets you know when to redraw the window. The callback registered by glutReshapeFunc is called when a window is resized. A default handler exists for handling window resizes that calls glViewPort(0,0,w,h), where w and h are the new width and height of the window. This makes the entire window available for OpenGL rendering. Usually this is appropriate, but you can call glutReshapeFunc to specify your own reshape callback if necessary.

Also callbacks can be registered for timeouts and when the program is "idling." A program doing continuous animation redraws each new scene as fast as the system will permit. This can be done by specifying an "idle" function using:

```
  glutIdleFunc(idle);
```

The function idle will be called whenever there is nothing else to do. If each time idle is called the program renders a new scene, the window is continuously animated. Event processing happens between calls to your idle function so be careful not to spend too much time in your idle function or you risk compromising your program's interactivity. There can be only one idle function registered at a time. If you call glutIdleFunc with NULL, the idle function is disabled. Idle callbacks are very much like X Toolkit work procedures.

```
  glutVisibilityFunc(visibility)
```

Because a program doing continuous animation is wasting its time if the window it is rendering into is completely obscured or unmapped, GLUT's glutVisibilityFunc routine registers a callback for the current window that is called when the window's visibility status changes. An example visibility callback:

```
void visibility(int status)
{
  if(status == GLUT_VISIBLE)
    glutIdleFunc(animate);
  else /* stop animating */
    glutIdleFunc(NULL);
}
```

Another type of callback is the timer callback that is registered by calling glutTimerFunc like this:

```
  glutTimerFunc(1000, timer, value);
```

The first parameter indicates the number of milliseconds to wait before calling the timer callback function. The third parameter is an integer that will be passed to the timer callback. You can register multiple timer functions. Timer callbacks are very much like X Toolkit timeout callbacks.

Menus

A common need for 3D programs is to turn on or off various modes based on user input. Pop-up menus provide a simple mechanism for this type of input.

GLUT provides an easy-to-use API for cascading pop-up menus. Menus can be created, changed, and "attached" to a mouse button within a window. If a menu is attached to a button in a window, pressing the button will trigger the pop-up menu. If a menu entry is selected, the callback function for the menu is called. The callback is passed the associated value for the selected menu entry. Here's an example:

```
  glutCreateMenu(choice_selected);
  glutAddMenuEntry("Enable lighting",1);
  glutAddMenuEntry("Disable lighting",2);
  glutAttachMenu(GLUT_RIGHT_BUTTON);
```

The result is a menu with two options to enable or disable lighting. The menu is associated with the current window and will be triggered when the right mouse button is pressed within the window. Selecting a menu item will call the choice_selected callback that might look like:

```
void choice_selected(int value)
{
  if(value == 1) glEnable(GL_LIGHTING);
  if(value == 2) glDisable(GL_LIGHTING);
  glutPostRedisplay();
}
```

Notice that instead of naively redrawing the window with lighting appropriately enabled or disabled, glutPostRedisplay is called. The advantage of "posting a redisplay" instead of performing the redraw explicitly is that removing the pop-up menu is likely to damage the current window.[2] A "posted redisplay" can be potentially *combined* with any pending Expose events caused by unmapping the pop-up menu. Multiple calls to glutPostRedisplay

[2] Actually GLUT will try to put the pop-up menu in the overlay planes (if overlays are supported) to avoid window damage normally generated by pop-up menus.

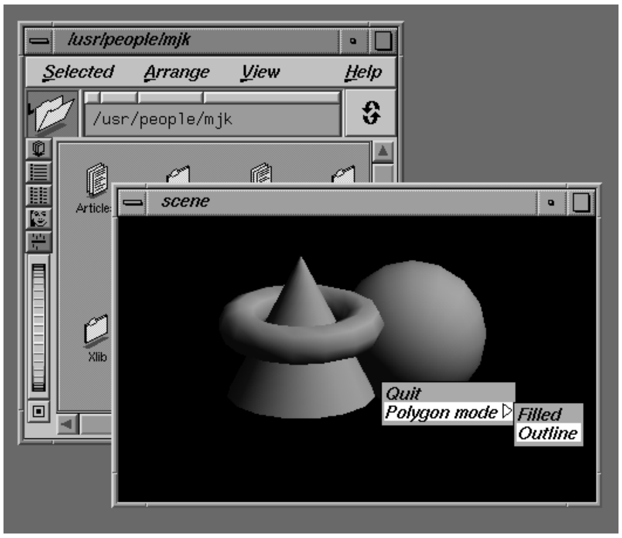

Figure 3: An example of GLUT cascaded pop-up menus.

and any pending `Expose` events will be combined if possible to minimize the redisplays necessary.

Like windows, GLUT maintains a *current menu* and the `glutCreateMenu` routines returns an integer identifier for the menu being created. The `glutSetMenu` and `glutGetMenu` routines set and query the current menu. And menu callbacks implicitly set the current menu to the menu generating the callback. The routines `glutAddMenuEntry` and `glutAttachMenu` operate on the current menu.

The menu identifier of a submenu is required for creating cascaded menus where one menu item can trigger the display of a submenu. Here's an example of creating a cascaded menu:

```
submenu = glutCreateMenu(polygon_mode);
glutAddMenuEntry("Filled", 1);
glutAddMenuEntry("Outline", 2);
glutCreateMenu(main_menu);
glutAddMenuEntry("Quit", 666);
glutAddSubMenu("Polygon mode",
  submenu);
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

Figure 3 shows what the above menu would look like. Menus can be cascaded arbitrarily deeply (but menu recursion is not permitted).

Routines exist to modify menu items in the current menu. The `glutChangeToMenuEntry` and `glutChangeToSubMenu` routines can be used to change an existing menu item in the current menu to an entry or submenu respectively. Menu items can also be deleted using `glutRemoveMenuItem`.

When menus are activated, the GLUT main loop continues to process events, handle timeouts, and call the idle function. Sometimes a program might want to suspend activity like the idle callback when a menu is in use. The `glutMenuStateFunc` can register a callback for this purpose, where the callback routine might look like:

```
void menu_status(int status)
{
  if(status == GLUT_MENU_IN_USE)
    glutIdleFunc(NULL);
  else
    glutIdleFunc(animate);
}
```

GLUT's menu support provides an easy way to let users see available options and select between various modes. In conjunction with the other input and window management support in the GLUT library, a programmer can quickly develop programs to explore OpenGL and 3D programming.

Next Time

In future columns, look forward to using the GLUT toolkit to demonstrate the wide variety of 3D graphics capabilities supported by OpenGL. The next column will explain OpenGL's lighting model. In the meantime, feel free to obtain the source code to GLUT and explore its documentation and examples.