INTRODUCTION TO OPENGL

As a software interface for graphics hardware, OpenGL renders multidimensional objects into a frame buffer. OpenGL is industry-standard graphics software with which programmers can create high-quality still and animated three-dimensional color images.

Where Applicable:
OpenGL is built for compatibility across hardware and operating systems. This architecture makes it easy to port OpenGL programs from one system to another. While each operating system has unique requirements, the OpenGL code in many programs can be used as is.

Developer Audience:
Designed for use by C/C++ programmers
Run-time Requirements:
OpenGL can run on Linux and all versions of 32 bit Microsoft Windows.

Most Widely Adopted Graphics Standard
OpenGL is the premier environment for developing portable, interactive 2D and 3D graphics applications. Since its introduction in 1992, OpenGL has become the industry's most widely used and supported 2D and 3D graphics application programming interface (API), bringing thousands of applications to a wide variety of computer platforms.
OpenGL fosters innovation and speeds application development by incorporating a broad set of rendering, texture mapping, special effects, and other powerful visualization functions. Developers can leverage the power of OpenGL across all popular desktop and workstation platforms, ensuring wide application deployment.

High Visual Quality and Performance
Any visual computing application requiring maximum performance-from 3D animation to CAD to visual simulation-can exploit high-quality, high-performance OpenGL capabilities. These capabilities allow developers in diverse markets such as broadcasting, CAD/CAM/CAE, entertainment, medical imaging, and virtual reality to produce and display incredibly compelling 2D and 3D graphics.

Developer-Driven Advantages

- **Industry standard**
  An independent consortium, the OpenGL Architecture Review Board, guides the OpenGL specification. With broad industry support, OpenGL is the only truly open, vendor-neutral, multiplatform graphics standard.
- **Stable**
  OpenGL implementations have been available for more than seven years on a wide variety of platforms. Additions to the specification are well controlled, and proposed updates are announced in time for developers to adopt changes. Backward compatibility requirements ensure that existing applications do not become obsolete.
- **Reliable and portable**
  All OpenGL applications produce consistent visual display results on any OpenGL API-compliant hardware, regardless of operating system or windowing system.

• **Evolving**

Because of its thorough and forward-looking design, OpenGL allows new hardware innovations to be accessible through the API via the OpenGL extension mechanism. In this way, innovations appear in the API in a timely fashion, letting application developers and hardware vendors incorporate new features into their normal product release cycles.

• **Scalable**

OpenGL API-based applications can run on systems ranging from consumer electronics to PCs, workstations, and supercomputers. As a result, applications can scale to any class of machine that the developer chooses to target.

• **Easy to use**

OpenGL is well structured with an intuitive design and logical commands. Efficient OpenGL routines typically result in applications with fewer lines of code than those that make up programs generated using other graphics libraries or packages. In addition, OpenGL drivers encapsulate information about the underlying hardware, freeing the application developer from having to design for specific hardware features.

• **Well-documented**

Numerous books have been published about OpenGL, and a great deal of sample code is readily available, making information about OpenGL inexpensive and easy to obtain.

Main purpose of OpenGL

As a software interface for graphics hardware, the main purpose of OpenGL is to render two- and three-dimensional objects into a frame buffer. These objects are described as sequences of vertices (that define geometric objects) or pixels (that define images).

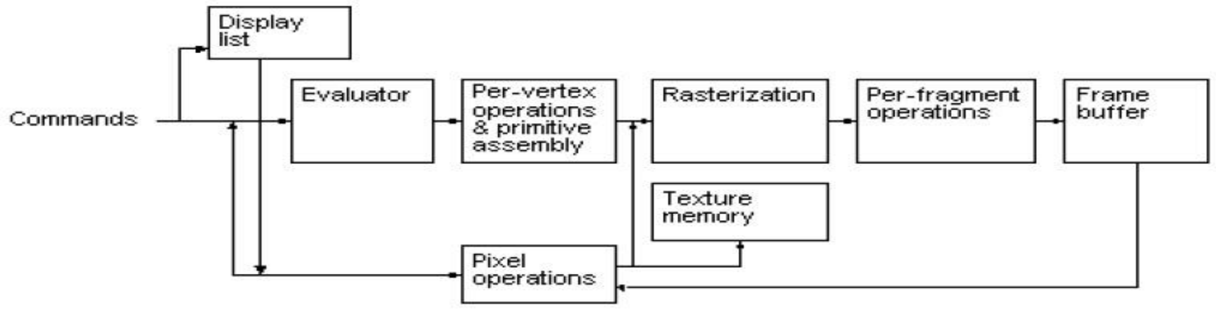
OpenGL performs several processes on this data to convert it to pixels to form the final desired image in the frame buffer.

The following topics present a global view of how OpenGL works:

- **Primitives and Commands** discusses points, line segments, and polygons as the basic units of drawing; and the processing of commands.
- **OpenGL Graphic Control** describes which graphic operations OpenGL controls and which it does not control.
- **Execution Model** discusses the client/server model for interpreting OpenGL commands.
- **Basic OpenGL Operation** gives a high-level description of how OpenGL processes data to produce a corresponding image in the frame buffer

Basic OpenGL Operation

The following diagram illustrates how OpenGL processes data. As shown, commands enter from the left and proceed through a processing pipeline. Some commands specify geometric objects to be drawn, and others control how the objects are handled during various processing stages.



The processing stages in basic OpenGL operation are as follows:

- Display list Rather than having all commands proceed immediately through the pipeline, you can choose to accumulate some of them in a display list for processing later.
- Evaluator The evaluator stage of processing provides an efficient way to approximate curve and surface geometry by evaluating polynomial commands of input values.
- Per-vertex operations and primitive assembly OpenGL processes geometric primitives points, line segments, and polygons—all of which are described by vertices. Vertices are transformed and lit, and primitives are clipped to the view port in preparation for rasterization.
- Rasterization The rasterization stage produces a series of frame-buffer addresses and associated values using a two-dimensional description of a point, line segment, or polygon. Each fragment so produced is fed into the last stage, per fragment operations.
- Per-fragment operations these are the final operations performed on the data before it's stored as pixels in the frame buffer.
- Per-fragment operations include conditional updates to the frame buffer based on incoming and previously stored z values (for z buffering) and blending of incoming pixel colors with stored colors, as well as masking and other logical operations on pixel values.

OPENGL PROGRAMMING

Installing OpenGL and Running OpenGL Programs on Microsoft Windows 7 and Higher

The development platform for our class is a Microsoft Visual Studio 2010 Integrated Development Environment (IDE) running on Windows 7. So, described below is how to install a free version of that IDE on Windows 7. However, your program should run, possibly with minor modification, even using later versions of Visual Studio and on Windows 8.

Go to http://www.visualstudio.com/en-us/downloads#d-2010-express and follow the links there to download and install Microsoft Visual C++ 2010 Express edition. After Visual C++ has been successfully installed, do the following.

Install Free GLUT:
1. Download and unzip the file freeglut-MSVC-2.8.1-1.mp.zip
From http://files.transmissionzero.co.uk/software/development/GLUT/freeglut-MSVC.zip

On 32-bit Windows:

(a) Copy all the files from freeglut\include\GL to
C:\Program Files\Microsoft SDKs\Windows\v7.0A\Include\GL.
Note that you may have to create the GL folder.

(b) Copy the file freeglut.lib from freeglut\lib to
C:\Program Files\Microsoft SDKs\Windows\v7.0A\Lib.

(c) Copy the file freeglut.dll from freeglut\bin to C:\Windows\System32.

On 64-bit Windows:

(a) Copy all the files from freeglut\include\GL to
C:\Program Files(x86)\Microsoft SDKs\Windows\v7.0A\Include\GL.
Note that you may have to create the GL folder.

(b) Copy the file freeglut.lib from freeglut\lib\x64 to
C:\Program Files(x86)\Microsoft SDKs\Windows\v7.0A\Lib.

(c) Copy the file freeglut.dll from freeglut\bin\x64 to C:\Windows\SysWOW64.

Install GLEW:

1. Download glext.h from http://www.opengl.org/registry/api/glext.h to C:\Program Files\Microsoft Visual Studio 10.0\VC\include\GL.

2. Download and unzip the file glew-1.10.0-win32.zip from http://glew.sourceforge.net/.

On 32-bit Windows:
(a) Copy all the files from glew-1.10.0\include\GL to
C:\Program Files\Microsoft SDKs\Windows\v7.0A\Include\GL.
(b) Copy all the files from glew-1.10.0\lib\Release\Win32 to
C:\Program Files\Microsoft SDKs\Windows\v7.0A\Lib.

(c) Copy all the files from glew-1.10.0\bin\Release\Win32 to C:\Windows\System32.

On 64-bit Windows:
(a) Copy all the files from glew-1.10.0\include\GL to
C:\Program Files(x86)\Microsoft SDKs\Windows\v7.0A\Include\GL.

(b) Copy all the files from glew-1.10.0\lib\Release\Win32 to
C:\Program Files(x86)\Microsoft SDKs\Windows\v7.0A\Lib.

(c) Copy all the files from glew-1.10.0\bin\Release\Win32 to C:\Windows\SysWOW64.

Run a program:

1. Open Visual C++ 2010 from the Start Menu to bring up the welcome screen.

2. Create a new project by going to File -> New -> Project.

3. Select Win32 from the Installed Templates panel and then Win32 Console Application from the next panel. Name your project and select the folder where you want to save it. Uncheck the box which says “Create directory for solution". Click OK to bring up a wizard welcome window.

4. Click Application Settings for the settings dialog box.

5. Uncheck the Precompiled header box, check the Empty project box and choose Console application. Click Finish to see a new project window.

6. Right click on Source Files and choose Add -> New Item to bring up a dialog box.

7. Select Code from the Installed Templates panel and C++ File(.cpp) from the next panel. Name your file and click Add to see an empty code panel in the project window titled with your chosen name.

8. Copy any of our note/sample programs into or write your own in the code panel.

9. Save and build your project by going to Debug -> Build Solution. Then execute the program with Debug -> Start Debugging. If the program has been built successfully, then you should see no error in the output window.

For examples of OpenGl:

https://www.opengl.org/archives/resources/code/samples/glut_examples/examples/examples.html

https://www.opengl.org/archives/resources/code/samples/simple/

OpenGL Programming

For writing a program, using OpenGL, for any of the operating systems we can use OpenGL Utility Library named, “glut”. “glut” is used to initialize OpenGL on any platform e.g. Microsoft Windows or Linux etc because it is platform independent. “glut” can create window, get keyboard input and run an event handler or message loop in our graphics application.
All those functions that start with the prefix “gl” are the core OpenGL functions and those which start with “glu” or “glut” are the OpenGL utility library functions. Let’s write a program that uses “glut” and then uses OpenGL function to create graphics.

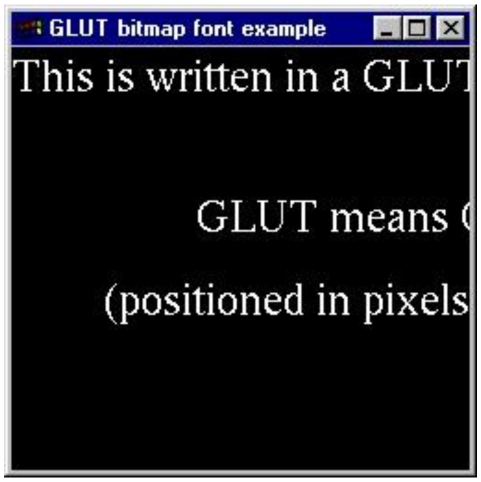
```
#include <GL/glut.h>
int main()
{
glutCreateWindow( "first graphics window" );
}
```

glutCreateWindow
glutCreateWindow creates a top-level window.

Usage
int glutCreateWindow(char *name);
name:
ASCII character string for use as window name.

Description: glutCreateWindow creates a top-level window. The name will be provided to the window system as the window's name. The intent is that the window system will label the window with the name.
Implicitly, the current window is set to the newly created window. Each created window has a unique associated OpenGL context. State changes to a window's associated OpenGL context can be done immediately after the window is created.
The display state of a window is initially for the window to be shown. But the window's display state is not actually acted upon until glutMainLoop is entered. This means until glutMainLoop is called, rendering to a created window is ineffective because the window cannot yet be displayed.
The value returned is a unique small integer identifier for the window. The range of allocated identifiers starts at one. This window identifier can be used when calling glutSetWindow.

Example of using GLUT bitmap fonts:



This is written in a GLUT

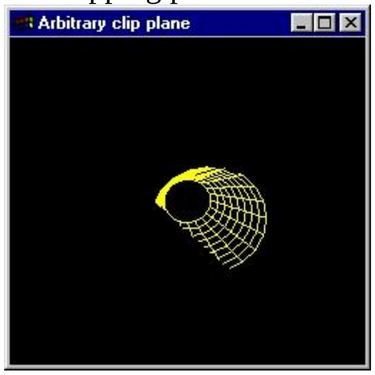
GLUT means O

(positioned in pixels

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

void *font = GLUT_BITMAP_TIMES_ROMAN_24;
void *fonts[] =
{
  GLUT_BITMAP_9_BY_15,
  GLUT_BITMAP_TIMES_ROMAN_10,
  GLUT_BITMAP_TIMES_ROMAN_24
};
char defaultMessage[] = "GLUT means OpenGL.";
char *message = defaultMessage;

void
selectFont(int newfont)
{
  font = fonts[newfont];
  glutPostRedisplay();
}

void
selectMessage(int msg)
{
  switch (msg) {
  case 1:
    message = "abcdefghijklmnop";
    break;
  case 2:
    message = "ABCDEFGHIJKLMNOP";
    break;
  }
}

void
selectColor(int color)
{
  switch (color) {
  case 1:
    glColor3f(0.0, 1.0, 0.0);
    break;
  case 2:
    glColor3f(1.0, 0.0, 0.0);
    break;
  case 3:
    glColor3f(1.0, 1.0, 1.0);
    break;
  }
  glutPostRedisplay();
}

void
tick(void)
{
  glutPostRedisplay();
}

void
output(int x, int y, char *string)
{
  int len, i;

  glRasterPos2f(x, y);
  len = (int) strlen(string);
  for (i = 0; i < len; i++) {
    glutBitmapCharacter(font, string[i]);
  }
}

void
display(void)
{
  glClear(GL_COLOR_BUFFER_BIT);
  output(0, 24, "This is written in a GLUT bitmap font.");
  output(100, 100, message);
  output(50, 145, "(positioned in pixels with upper-left origin)");
  glutSwapBuffers();
}

void
reshape(int w, int h)
{
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(0, w, h, 0);
  glMatrixMode(GL_MODELVIEW);
}

int
main(int argc, char **argv)
```

```
{
  int i, msg_submenu, color_submenu;

  glutInit(&argc, argv);
  for (i = 1; i < argc; i++) {
    if (!strcmp(argv[i], "-mono")) {
      font = GLUT_BITMAP_9_BY_15;
    }
  }
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
  glutInitWindowSize(500, 150);
  glutCreateWindow("GLUT bitmap font example");
  glClearColor(0.0, 0.0, 0.0, 1.0);
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutIdleFunc(tick);
  msg_submenu = glutCreateMenu(selectMessage);
  glutAddMenuEntry("abc", 1);
  glutAddMenuEntry("ABC", 2);
  color_submenu = glutCreateMenu(selectColor);
  glutAddMenuEntry("Green", 1);
  glutAddMenuEntry("Red", 2);
  glutAddMenuEntry("White", 3);
  glutCreateMenu(selectFont);
  glutAddMenuEntry("9 by 15", 0);
  glutAddMenuEntry("Times Roman 10", 1);
  glutAddMenuEntry("Times Roman 24", 2);
  glutAddSubMenu("Messages", msg_submenu);
  glutAddSubMenu("Color", color_submenu);
  glutAttachMenu(GLUT_RIGHT_BUTTON);
  glutMainLoop();
  return 0;             /* ANSI C requires main to return int. */
}
```

Clipping OpenGl program:
The main intent of this program is to demo the arbitrary clipping functionality, hence the rendering is kept simple (wireframe) and only one clipping plane is used.

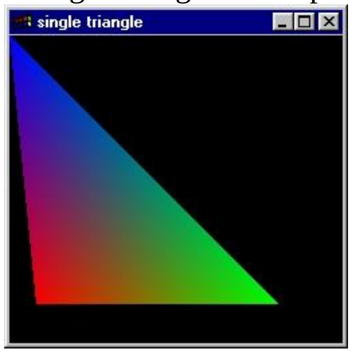


```
#include <stdio.h>
#include <stdlib.h>
#include <GL/glut.h>
/* function declarations */
```

```
void
  drawScene(void), setMatrix(void), animateClipPlane(void), animation(void),
  resize(int w, int h), keyboard(unsigned char c, int x, int y);
/* global variables */
float ax, ay, az;       /* angles for animation */
GLUquadricObj *quadObj; /* used in drawscene */
GLdouble planeEqn[] = {0.707, 0.707, 0.0, 0.0};  /* initial clipping plane eqn */
int count = 0;
int clip_count = 0;
void menu(int choice)
{
  switch (choice) {
  case 1:
    count = 90;
    glutIdleFunc(animation);
    break;
  case 2:
    animateClipPlane();
    break;
  } }
int main(int argc, char **argv)
{
  glutInit(&argc, argv);
  quadObj = gluNewQuadric();  /* this will be used in drawScene */
  glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
  glutCreateWindow("Arbitrary clip plane");
  ax = 10.0;
  ay = -10.0;
  az = 0.0;
  glutDisplayFunc(drawScene);
  glutReshapeFunc(resize);
  glutKeyboardFunc(keyboard);
  glutCreateMenu(menu);
  glutAddMenuEntry("Rotate", 1);
  glutAddMenuEntry("Move clip plane", 2);
  glutAttachMenu(GLUT_RIGHT_BUTTON);
  glutMainLoop();
  return 0;             /* ANSI C requires main to return int. */
}
void
drawScene(void)
{
  glClearColor(0.0, 0.0, 0.0, 0.0);
  glClear(GL_COLOR_BUFFER_BIT);
  glPushMatrix();
  gluQuadricDrawStyle(quadObj, GLU_LINE);
  glColor3f(1.0, 1.0, 0.0);
  glRotatef(ax, 1.0, 0.0, 0.0);
  glRotatef(-ay, 0.0, 1.0, 0.0);
  glClipPlane(GL_CLIP_PLANE0, planeEqn);  /* define clipping plane */
  glEnable(GL_CLIP_PLANE0);  /* and enable it */
  gluCylinder(quadObj, 2.0, 5.0, 10.0, 20, 8);  /* draw a cone */
  glDisable(GL_CLIP_PLANE0);
  glPopMatrix();
  glutSwapBuffers();
}
void
setMatrix(void)
{
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(-15.0, 15.0, -15.0, 15.0, -10.0, 10.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}
void
animation(void)
{
  if (count) {
    ax += 5.0;
    ay -= 2.0;
    az += 5.0;
    if (ax >= 360)
      ax = 0.0;
    if (ay <= -360)
      ay = 0.0;
    if (az >= 360)
      az = 0.0;
    glutPostRedisplay();
    count--;
  }
  if (clip_count) {
    static int sign = 1;
    planeEqn[3] += sign * 0.5;
    if (planeEqn[3] > 4.0)
      sign = -1;
    else if (planeEqn[3] < -4.0)
      sign = 1;
    glutPostRedisplay();
    clip_count--;
  }
  if (count <= 0 && clip_count <= 0)
    glutIdleFunc(NULL);
}
void
animateClipPlane(void)
{
  clip_count = 5;
  glutIdleFunc(animation);
}
/* ARGSUSED1 */
void
keyboard(unsigned char c, int x, int y)
{
  switch (c) {
  case 27:
    exit(0);
    break;
  default:
    break;
  }
}
void
resize(int w, int h)
{
  glViewport(0, 0, w, h);
  setMatrix();
}
```

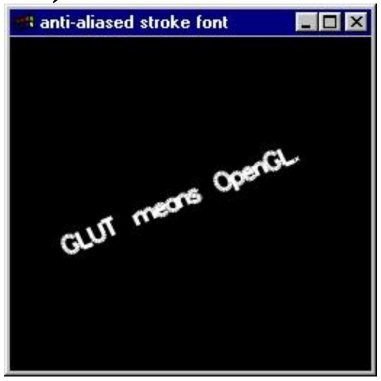
Draw a triangle using OpenGl:

Very simple example of how to draw a single triangle with OpenGL:



```
#include <GL/glut.h>
Void reshape(int w, int h)
{
  /* Because Gil specified "screen coordinates" (presumably with an      upper-left origin), this short bit of code sets up the coordinate system to correspond to actual window coodrinates.  This code      wouldn't be required if you chose a (more typical in 3D) abstract coordinate system. */
  glViewport(0, 0, w, h);       /* Establish viewing area to cover entire window. */
  glMatrixMode(GL_PROJECTION);  /* Start modifying the projection matrix. */
  glLoadIdentity();             /* Reset project matrix. */
  glOrtho(0, w, 0, h, -1, 1);   /* Map abstract coords directly to window coords. */
  glScalef(1, -1, 1);           /* Invert Y axis so increasing Y goes down. */
  glTranslatef(0, -h, 0);       /* Shift origin up to upper-left corner. */
}

Void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT);
  glBegin(GL_TRIANGLES);
    glColor3f(0.0, 0.0, 1.0);  /* blue */
    glVertex2i(0, 0);
    glColor3f(0.0, 1.0, 0.0);  /* green */
    glVertex2i(200, 200);
    glColor3f(1.0, 0.0, 0.0);  /* red */
    glVertex2i(20, 200);
  glEnd();
  glFlush();  /* Single buffered, so needs a flush. */
}
int
main(int argc, char **argv)
{
  glutInit(&argc, argv);
  glutCreateWindow("single triangle");
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutMainLoop();
  return 0;             /* ANSI C requires main to return int. */
}
```

Stroke fonts using GLUT (antialiased):



```
#include <string.h>
#include <GL/glut.h>
void *font = GLUT_STROKE_ROMAN;
void *fonts[] = {GLUT_STROKE_ROMAN, GLUT_STROKE_MONO_ROMAN};
char defaultMessage[] = "GLUT means OpenGL.";
char *message = defaultMessage;
int angle = 0;

void selectFont(int newfont)
{
  font = fonts[newfont];
  glutPostRedisplay();
}

Void selectMessage(int msg)
{
  switch (msg) {
  case 1:
    message = "abcdefghijklmnop";
    break;
  case 2:
    message = "ABCDEFGHIJKLMNOP";
    break;
  }
}
```

```
Void tick(void)
{
  angle -= 2;
  glutPostRedisplay();
}

Void display(void)
{
  int len, i;
  glClear(GL_COLOR_BUFFER_BIT);
  glPushMatrix();
  glRotatef(angle, 0.0, 0.0, 1.0);
  glTranslatef(-750, 0, 0);
  len = (int) strlen(message);
  for (i = 0; i < len; i++) {
    glutStrokeCharacter(font, message[i]);
  }
  glPopMatrix();
  glutSwapBuffers();
}
int
main(int argc, char **argv)
{
  int i, submenu;

  glutInit(&argc, argv);
  for (i = 1; i < argc; i++) {
    if (!strcmp(argv[i], "-mono")) {
      font = GLUT_STROKE_MONO_ROMAN;
    }
  }
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
  glutInitWindowSize(600, 600);
  glutCreateWindow("anti-aliased stroke font");
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(0, 2000, 0, 2000);
  glMatrixMode(GL_MODELVIEW);
  glEnable(GL_LINE_SMOOTH);
  glEnable(GL_BLEND);
  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
  glLineWidth(3.0);
  glTranslatef(1000, 1000, 0);
  glClearColor(0.0, 0.0, 0.0, 1.0);
  glColor3f(1.0, 1.0, 1.0);
  glutDisplayFunc(display);
  glutIdleFunc(tick);
  submenu = glutCreateMenu(selectMessage);
  glutAddMenuEntry("abc", 1);
  glutAddMenuEntry("ABC", 2);
  glutCreateMenu(selectFont);
  glutAddMenuEntry("Roman", 0);
  glutAddMenuEntry("Mono Roman", 1);
  glutAddSubMenu("Messages", submenu);
  glutAttachMenu(GLUT_RIGHT_BUTTON);
  glutMainLoop();
  return 0;             /* ANSI C requires main to return int. */
}
```