Installing/Using OpenGL

Introduction: This document describes a bit about compiling and running OpenGL programs for C/C++ on the various platforms around campus. In particular we will consider the following platforms.

CSIC Linux Lab: The Dell machines running Redhat Linux in the CSIC Linux Lab on the third floor of the CSIC building. (Note that the WAM-lab Linux machines do not have GLUT installed, and so cannot be used at this time. If you have your own Linux system, these instructions are applicable, but you will need to install the OpenGL, GLU, and GLUT libraries.)

PC Windows: Your own PC running Microsoft Windows.

To understand this more concretely, while you are reading this you should also download the sample program, which we have made available. From the class web page go to the “OpenGL” link in the index and then follow the link to the “Sample OpenGL Program,” or go directly to the following link. More detailed information can be found in the various “Readme” files contained within the bundle.

OpenGL is the most widely used graphics library standard, that is, it is just a specification for a graphics library, which has been implemented by a number of vendors. OpenGL consists of two principal components: GL (basic OpenGL) and GLU (OpenGL utilities). GL is responsible for the basic low-level rendering tasks, and GLU provides support for some higher-level operations, such as drawing curved surfaces. In addition, it is necessary to use a toolkit for creating windows and handling user interaction. For C/C++ programming, we will use GLUT (OpenGL utility toolkit). (Java programmers will need to use a different toolkit, for example, the Java AWT (Abstract Window Toolkit), but we will not discuss that here.

Installing OpenGL/Glut on your own PC: The following description assumes that you running on a PC running Microsoft Windows (98, 2000, NT or XP) and have Microsoft Visual Studio 6 or Visual Studio.NET. This does not apply to Linux or Mac’s, however. You first need to know the names of the following two directories on your system:

\(<WinDir>\) : This is your Windows system directory (e.g., C:\WINDOWS or C:\WINNT).
\(<VCpp>\) : Your visual C++ directory. For Visual Studio 6 this is something like:

```
C:\Program Files\Microsoft Visual Studio\VC98
```

For Visual Studio.NET this might be:

```
C:\Program Files\Microsoft Visual Studio.NET 2003\Vc7\PlatformSDK
```

If you are not sure, search for the file opengl32.lib.

OpenGL should already be installed on your machine. To verify that OpenGL is installed on your system, first do a search for the files opengl32.dll and glu32.dll. They should appear in your windows system directory (with lots of other dll files). You need to install Glut, however. The easiest way to do this is to visit the following web page:

http://www.xmission.com/~nate/glut.html
It contains precompiled GLUT libraries. (Download the “GLUT for Win32 dll, lib and header file”
not the “source code distribution”.) After unbundling the file, copy the following files to the following
directories:

\glut32.dll \rightarrow (WinDir)\SYSTEM32 (or wherever opengl32.dll is)
\glut32.lib \rightarrow (VCpp)\lib
\glut.h \rightarrow (VCpp)\include\GL.

By the way, the exact directory in which these files are installed is less important than the fact that
the system can locate them. As long as these files are stored in directories that lie on the appropriate
environment variables, e.g., PATH or INCLUDE, your system should be able to locate them.

Now, you should be ready to go. If you have Visual Studio 6, then the quickest way to proceed is to
go to the directory VisualCPP and double click the workspace file sample1.dsw. If you have Visual
Studio.NET, go to the directory VisualStudioNET and double click the solution file sample1.sln.

Please read the “Readme” files carefully for more detailed instructions on how to construct your own
programs.

**CSIC Linux Lab:** Compiling the programs involves a bewildering number of options, in order to specify
the location of the OpenGL and GLUT include files, libraries, and the runtime library directories. The
easiest way to get started is the use the “Makefile” given in the Sample OpenGL program, mentioned
above. Edit the file to see which options can be adjusted. Enter “make” to compile the sample
program, after which you should be able to run the resulting executable.

Unfortunately, there is no widespread agreement on how the various directories should be configured
on Unix/Linux platforms, and each system administrator makes his/her own choices when installing
things. Commands like “locate” can often be used to help you locate where these files are on any par-
ticular Unix/Linux system. In case you are interested, in the CSIC Linux Labs the library files
libGL
and libGLU are located in /usr/X11R6/lib and libglut is located in /user/local/freeglut. The include
files gl.h and glu.h are located in /usr/include/GL and glut.h is located /usr/local/freeglut/include/GL.

**Remote Execution:** If you have an X-server on your PC at home (e.g., XFree86 or Reflection) you can
remotely log into the WAM or Glue labs, compile your program, and run it. The graphics should
appear on your PC display. Hint: before trying this with an untested OpenGL program, try a known
X11 application (for example, enter “xv” or “gimp”). If that works, then try running your program.
If everything is configured properly, the graphics should appear on your screen. Beware, it may be
quite slow because the graphics is being shipped over the network, but it is an option for your initial
development and debugging.
Lighting in OpenGL

This handout briefly describes a number of OpenGL’s commands for controlling lighting and shading. See the reference documentation and tutorials on the web for more information.

**Options:** Many of the capabilities of OpenGL can either be turned on or turned off. This is handled through various options, which can be either enabled or disabled. Here are a number of the options related to lighting.

`glEnable(GLenum cap), glDisable(GLenum cap):`
Enable/disable some option. The following options are useful for 3-dimensional hidden surface removal and lighting. By default, all are initially disabled.

- **GL_DEPTH_TEST:** Enables hidden surface removal (depth-buffering). In addition to setting this option, you also need to enable the depth buffer in your initialization code, by adding `GLUT_DEPTH` to `glutInitDisplayMode`, for example:

  ```
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH)
  ```

  By disabling this option you can temporarily suspend hidden surface removal (e.g. for writing text onto the window).

- **GL_LIGHTING:** Enables lighting (but individual lights must be activated using the option below).

- **GL_LIGHT*:** Turn on/off a light source, for example `glEnable(GL_LIGHT3)` turns on light source 3.

- **GL_NORMALIZE:** Normal vectors must be of unit length for correct lighting and shading. This automatic normalizes the length of normal vectors to unit length prior to drawing.

**Lighting:** In OpenGL there are at least eight light sources (`GL_LIGHT0` through `GL_LIGHT7`). (The actual number on any implementation can be determined by a call to `glGetIntegerv()`. If lighting is enabled (see `glEnable()`) then the shading of each object depends on which light sources are turned on (enabled) and the materials and surface normals of each of the objects in the scene. Note that when lighting is enabled, it is important that each vertex be associated with a proper normal vector, by calling `glNormal*()`, prior to generating the vertex. Once set, the normal is associated with all vertices until changed again.

- **glShadeModel(GLenum mode):**
  The mode may be either `GL_FLAT` or `GL_SMOOTH`. In flat shading every point on a polygon is shaded according to its first vertex. In smooth shading the shading from each of the various vertices is interpolated.

- **glLightModelf(GLenum pname, GLfloat param):**
- **glLightModelfv(GLenum pname, const GLfloat *params):**
  Defines general lighting model parameters. The first version is for defining scalar parameters, and the second is for vector parameters. One important parameter is the global intensity of ambient light (independent of any light sources). Its `pname` is `GL_LIGHT_MODEL_AMBIENT` and `params` is a pointer to an RGBA vector.

- **glLightf(GLenum light, GLenum pname, GLfloat param):**
**glLightfv(GLenum light, GLenum pname, const GLfloat *params):**
Defines parameters for a single light source. The first version is for defining scalar parameters, and the second is for vector parameters. The first argument indicates which light source this applies to. The argument pname gives one of the properties to be assigned. These include the following:

- **GL_POSITION** (vector) \((x, y, z, w)\) of position of light
- **GL_AMBIENT** (vector) RGBA of intensity of ambient light
- **GL_DIFFUSE** (vector) RGBA of intensity of diffuse light
- **GL_SPECULAR** (vector) RGBA of intensity of specular light

By default, illumination intensity does not decrease, or attenuate, with distance. In general, if \(d\) is the distance from the light source to the object, and the light source is not a point at infinity, then the intensity attenuation is given by \(1/(a + bd + cd^2)\) where \(a\), \(b\), and \(c\) are specified by the following parameters:

- **GL_CONSTANT_ATTENUATION** (scalar) \(a\)-coefficient
- **GL_LINEAR_ATTENUATION** (scalar) \(b\)-coefficient
- **GL_QUADRATIC_ATTENUATION** (scalar) \(c\)-coefficient.

Normally light sources send light uniformly in all directions. To define a spotlight, set the following parameters.

- **GL_SPOT_CUTOFF** (scalar) maximum spread angle of spotlight
- **GL_SPOT_DIRECTION** (vector) \((x, y, z, w)\) direction of spotlight
- **GL_SPOT_EXPONENT** (scalar) exponent of spotlight distribution

**Note:** In addition to defining these properties, each light source must also be enabled. See `glEnable()`.

**Surface Properties:** When lighting is used, surface properties are given through the command `glMaterial*()`, rather than `glColor*()`.

**glMaterialf(GLenum face, GLenum pname, GLfloat param):**
**glMaterialfv(GLenum face, GLenum pname, const GLfloat *params):**

Defines surface material parameters for subsequently defined objects. The first version is for defining scalar parameters, and the second is for vector parameters. Polygonal objects in OpenGL have two sides. You can assign properties either to the front, back, or both sides. (The front side is the one from which the vertices appear in counterclockwise order.) The first argument indicates the side. The possible values are **GL_FRONT**, **GL_BACK**, and **GL_FRONT_AND_BACK**. The second argument is the specific property. Possibilities include:

- **GL_EMISSION** (vector) RGBA of the emitted coefficients
- **GL_AMBIENT** (vector) RGBA of the ambient coefficients
- **GL_DIFFUSE** (vector) RGBA of the diffuse coefficients
- **GL_SPECULAR** (vector) RGBA of the specular coefficients
- **GL_SHININESS** (scalar) single number in the range \([0, 128]\) that indicates degree of shininess.
Shade Model: Because OpenGL only deals with flat objects, programmers need to use many small flat polygonal faces to approximate smooth surfaces, such as spheres, say. But this raises the question of whether the user wants the object to appear smoothly shaded or to clearly see the boundaries between adjoining faces. This is done through the shading model, whose argument is either GL_SMOOTH (the default) or GL_FLAT.

```c
  glShadeModel(GL_SMOOTH);
```

The shading interpolation can be handled in one of two ways. In the classical Gouraud interpolation the illumination is computed exactly at the vertices (using the above formula) and the values are interpolated across the polygon. In Phong interpolation, the normal vectors are given at each vertex, and the system interpolates these vectors in the interior of the polygon. Then this interpolated normal vector is used in the above lighting equation. This produces more realistic images, but takes considerably more time. OpenGL uses Gouraud shading. Just before a vertex is given (with glVertex*()), you should specify its normal vertex (with glNormal*()), which is discussed below.

Normal Vectors: Normal vectors are needed for performing lighting computations. OpenGL does not compute them, you need to compute them yourself. Normal vectors are specified, just prior to drawing the vertex with the comment glNormal*(). Normal vectors are assumed to be of unit length. For example, suppose that we wanted to draw a red triangle on the x,y-plane. Here is a code fragment that would do this.

```c
  GLfloat red[4] = {1.0, 0.0, 0.0, 1.0}; // RGB for red
  // set material color
  glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, red);
  glNormal3f(0, 0, 1); // set normal vector (up)
  glBegin(GL_POLYGON); // draw triangle on x,y-plane
  glVertex3f(0, 0, 0);
  glVertex3f(1, 0, 0);
  glVertex3f(0, 1, 0);
  glEnd();
```

You should be sure your normal vectors are of unit length. If not, enable GL_NORMALIZE so that OpenGL does it for you.
Textures, Fog and Color Blending in OpenGL

This handout briefly describes a number of OpenGL’s commands for controlling special effects, such as texture, fog and color blending. See the reference documentation and tutorials on the web for more information.

Options: Many of the capabilities of OpenGL can either be turned on or turned off. This is handled through various options, which can be either enabled or disabled.

`glEnable(GLenum cap)`, `glDisable(GLenum cap)`: Enable/disable some option. The following options are useful for texture mapping and fog. More details on controlling these effects are given below. By default, all are initially disabled.

- **GL_FOG**: Enables fog.
- **GL_BLEND**: Enables color blending (using the ‘A’ in RGBA) to achieve transparency and related effects.
- **GL_TEXTURE_2D**: Enables texture mapping.

Note that options may be enabled and disabled throughout the execution of the program. For example, texture mapping may be turned on before drawing one polygon, and then turned off for others.

**Blending and Fog**: Blending and fog are two OpenGL capabilities that allow you to produce interesting lighting and coloring affects. When a pixel is to be drawn on the screen, it normally overwrites any existing pixel color. When blending is enabled (by calling `glEnable(GL_BLEND)`) then the new (source) pixel is blended with the existing (destination) pixel in the frame buffer, depending on the ‘A’ value of the RGBA color. Note that GLUT_RGBA should be specified in `glutInitDisplayMode()`.

`glBlendFunc(GLenum sfactor, GLenum dfactor)`: Determines how new pixel values are blended with existing values. Whenever you draw pixel with blending enabled, OpenGL first determines whether the pixel is visible (through hidden surface removal, assuming that `GL_DEPTH_TEST` is enabled), and it then sets the value of the pixel to be some function of the existing pixel color (destination), the new pixel color (source), and the alpha (‘A’) component of the new color. OpenGL provides many different functions. See the reference manuals for complete information. For example, to achieve simple transparency, the call would be `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);`

**Beware**: The depth buffer treats transparent objects as if they were opaque. Thus, a totally transparent object ($A = 0$) will effectively conceal an opaque object that lies farther away. As a result, it is best to draw transparent objects last, or just disable the depth test. In this way, the farther opaque object will already exist in the frame buffer, so that its color may be blended with the transparent object.

Fog produces an effect whereby more distant objects are blended increasingly with a fog color, typically some shade of gray. It is enabled by calling `glEnable(GL_FOG).

`glFogf(GLenum pname, GLfloat param)`: 
glFogfv(GLenum pname, const GLfloat *params):

Specifies the parameters that define how fog is computed. The first version is for defining scalar parameters, and the second is for vector parameters. Here are some parameter names and their meanings. See the reference manual for complete details.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_FOG_MODE</td>
<td>(scalar) How rapidly does the fog grow with distance.</td>
</tr>
<tr>
<td>GL_FOG_START</td>
<td>(scalar) Distance where fog begins</td>
</tr>
<tr>
<td>GL_FOG_END</td>
<td>(scalar) Distance at which fog is total</td>
</tr>
<tr>
<td>GL_FOG_COLOR</td>
<td>(vector) RGBA of color of the fog</td>
</tr>
</tbody>
</table>

Texture Mapping: Texture mapping is the process of taking an image, presented typically as a 2-dimensional array of RGB values and mapping it onto a polygon. Setting up texture mapping involves the following steps: define a texture by specifying the image and its format (through glTexImage2d()), specify how object vertices correspond to points in the texture, and finally enable texture mapping. First, the texture must be input or generated by the program. OpenGL provides a wide variety of other features, but we will only summarize a few here, which are sufficient for handling a single 2-dimensional texture.

glTexImage2d: (GLenum target, int level, int internalFormat, int width, int height, int border, GLenum format, GLenum type, void *pixels):

This converts a texture stored in the array pixels into an internal format for OpenGL’s use. The first argument is typically GL_TEXTURE_2D. (But 1-dimensional textures exist as well.) The next parameter is used to specify the level, assuming multiple level texture maps, or mipmaps are used. We will assume single-level textures, so level will be 0. The internalFormat parameter specifies how OpenGL will store the texture internally. It is typically either GL_RGBA or GL_RGB. The width and height parameters give the width and height of the image. These must be powers of 2. We will assume no texture borders, so the border parameter will be 0. The format parameter is the format of your pixels array. The type parameter is the type of each color component in your pixel array. (If you are using the readBMPFile() function, for reading .bmp files, the last three parameters will be GL_RGB, GL_UNSIGNED_BYTE, and the pixel member of your RGBPixmap object.) See the reference manual for complete information.

glTexEnvf(GLenum target, GLenum pname, GLfloat param):

Specifies texture mapping environment parameters. The target must be GL_TEXTURE_ENV. The pname parameter must be GL_TEXTURE_ENV_MODE. This determines how a color from the texture image is to be merged with an existing color on the surface of the polygon. The param may be any of the following:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_MODULATE</td>
<td>multiply color components together</td>
</tr>
<tr>
<td>GL_BLEND</td>
<td>linearly blend color components</td>
</tr>
<tr>
<td>GL_DECAL</td>
<td>use the texture color</td>
</tr>
<tr>
<td>GL_REPLACE</td>
<td>use the texture color</td>
</tr>
</tbody>
</table>
There are subtle differences between GL_DECAL and GL_REPLACE when different formats are used or when the ‘A’ component of the RGBA color is not 1. See the reference manual for details. The default is GL_MODULATE, which is a good choice for combining textures with light.

glTexParameteri(GLenum target, GLenum pname, GLfloat param):

Specify how texture interpolation is to be performed. The first version is for defining scalar parameters, and the second is for vector parameters. Assuming 2-dimensional textures, the target is GL_TEXTURE_2D, the pname is either:

- GL_TEXTURE_MAG_FILTER: magnification filter
- GL_TEXTURE_MIN_FILTER: minification filter

Magnification is used when a pixel of the texture is smaller than the corresponding pixel of the screen onto which it is mapped and minification applies in the opposite case. Typical values are either

- GL_NEAREST: take the nearest texture pixel
- GL_LINEAR: take the weighted average of the 4 surrounding texture pixels

This procedure may also be invoked to specify other properties of texture mapping. Another important parameter involves how textures are wrapped in order to use a small texture tiles to cover a large area. See the options GL_TEXTURE_WRAP_S and GL_TEXTURE_WRAP_T in the OpenGL documentation.

glTexCoord*(...):

This is used when drawing each vertex. It is somewhat analogous to glNormal() in shading, because it specifies a value for each vertex, and OpenGL interpolates values for pixels in between.

It specifies the texture coordinates of subsequently defined vertices for texture mapping. For a standard 2-dimensional textures, the texture coordinates are a pair \((s, t)\) in the interval \([0, 1] \times [0, 1]\). The texture coordinate specifies the point on the image that are to be mapped to this vertex. OpenGL interpolates the mapping of intermediate points of the polygon.

Multiple Textures: The above material assumes that there is only one texture. Handling multiple textures involves two steps. First, you have to generate new texture objects. This is done with the command glGenTextures(). It generates an array consisting of the “names” (actually just integer identifiers) of the newly constructed texture objects. Next, whenever working with a specific texture you need to specify which of the existing textures (from glGenTextures()) is the current texture object. This is done with glBindTexture(). Here is an example of how to use these.

```c
static GLuint texName[5];  // texture names for 5 textures
glGenTextures(5, texName);  // create 5 texture names

// make texture 0 the current texture
glBindTexture(GL_TEXTURE_2D, texName[0]);
// ... operations/drawings involving texture 0

// make texture 2 the current texture
glBindTexture(GL_TEXTURE_2D, texName[2]);
// ... operations/drawings involving texture 2
```
Texture Mapping Utility: In order to use texture mapping, you must present a texture to OpenGL as an array. Typically, textures are given as image files in some standard format (.jpg, .gif, .ppm, .bmp). There are many programs that can convert from one to another, such as gimp on Linux systems, Adobe photoshop, or IrfanView (Windows freeware).

To help you with the task of inputting images, I have adapted a utility program, which I found in Hill’s Graphics book. It consists of a class RGBPixmap that stores an image. Its main method reads in an image from a .bmp file:

```cpp
bool readBMPFile( // read a .bmp file
    const string& fname,  // name of the file
    bool glPad,            // pad size up to a power of 2
    bool verbose);         // output summary
```

If the second parameter is true, then the image array is padded up to the next higher power of 2 in size. This is done because OpenGL expects texture maps whose dimensions are exact powers of 2. These additional entries are not initialized. Otherwise, the image size is not altered. If verbose argument is true, summary information is written to cerr. See the associated readme.txt file for information on how to compile it.

A template of how to use this in an OpenGL program is shown in Figs. 1 and 2. This assumes that you are using a single texture. It consists of two parts. The first part is the initialization of the texture, which is done only once, and is shown in Fig. 1. The second part involves settings that are done with each redrawing, and is given in Fig. 2.


```cpp
#include "RGBpixmap.h"

// ...

RGBpixmap myPixmap; // declare RGBpixmap object
glPixelStorei(GL_UNPACK_ALIGNMENT, 1); // store pixels by byte
// modulated colors
glTexEnv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
// read the image file
if (!myPixmap.readBMPFile("text0.bmp", true, true)) {
    cerr << "File text0.bmp cannot be read or illegal format" << endl;
    exit(1);
}
glTexImage2D( // initialize texture
    GL_TEXTURE_2D, // texture is 2-d
    0, // resolution level 0
    GL_RGB, // internal format
    myPixmap.nCols, // image width
    myPixmap.nRows, // image height
    0, // no border
    GL_RGB, // my format
    (unsigned char*)myPixmap.pixel); // the pixels

// set texture parameters

glTexParameteri(GL_TEXTURE_2D, /* assign parameters for the texture */);

// ...
```

Figure 1: One-time initialization of texture settings, and using `readBMPFile()` to input the texture from a file named `teset0.bmp`.

... glEnable(GL_TEXTURE_2D); // enable texture mapping

```
glMaterialfv(GL_FRONT_AND_BACK, // white base color
    GL_AMBIENT_AND_DIFFUSE,
    glfv(white));

glBegin(GL_POLYGON); // draw the object
    glNormal3f (/*...specify normal coordinates for vertex 0...*/);
    glTexCoord2f(/*...specify texture coordinates for vertex 0...*/);
    glVertex3f (/*...specify vertex coordinates for vertex 0...*/);
    // ... (repeat for other vertices)

 glEnd();
```

Figure 2: Displaying a texture-mapped object.
OpenGL Sample Program

// File: sample.cpp
// Description: A sample OpenGL program

#include <cstdlib>  // standard definitions
#include <iostream>  // C++ I/O
#include <cstdio>  // C I/O (for sprintf)
#include <cmath>  // standard definitions
#include <GL/glut.h>  // GLUT
#include <GL/glu.h>  // GLU
#include <GL/gl.h>  // OpenGL
using namespace std;  // make std accessible

// Global data

GLint TIMER_DELAY = 10000;  // timer delay (10 seconds)
GLfloat RED_RGB[] = {1.0, 0.0, 0.0};  // drawing colors
GLfloat BLUE_RGB[] = {0.0, 0.0, 1.0};
static bool isReversed = false;  // draw reversed colors?

// Callbacks - Global variable "isReversed" describes the drawing state.

void myReshape(int w, int h) {
    cout << "MyReshape called width=" << w << " height=" << h << endl;
    glViewport (0, 0, w, h);  // update the viewport
    glMatrixMode(GL_PROJECTION);  // update projection
    glLoadIdentity();
    gluOrtho2D(0.0,1.0, 0.0, 1.0);  // map unit square to viewport
    glMatrixMode(GL_MODELVIEW);
    glutPostRedisplay();  // request redisplay
}

void drawObjects(GLfloat* diamColor, GLfloat* rectColor) {
    glColor3fv(diamColor);  // set diamond color
    glBegin(GL_POLYGON);  // draw the diamond
    glVertex2f(0.90, 0.50);
    glVertex2f(0.50, 0.90);
    glVertex2f(0.10, 0.50);
    glVertex2f(0.50, 0.10);
    glEnd();
    glColor3fv(rectColor);  // set rectangle color
    glRectf(0.25, 0.25, 0.75, 0.75);  // draw the rectangle
}

void myDisplay(void) {  // display callback
    cout << "MyDisplay called" << endl;
    glClearColor(0.5, 0.5, 0.5, 1.0);  // background is gray
    glClear(GL_COLOR_BUFFER_BIT);  // clear the window
    if (isReversed)  // draw the objects
        drawObjects(BLUE_RGB, RED_RGB);
    else
        drawObjects(RED_RGB, BLUE_RGB);
    glutSwapBuffers();  // swap buffers
}
void myTimer(int id) { // timer callback
    cout << "Timer just went off. Reversing colors." << endl;
    isReversed = !isReversed; // reverse drawing colors
    glutPostRedisplay(); // request redraw
    glutTimerFunc(TIMER_DELAY, myTimer, 0); // reset timer for 10 seconds
}

void myMouse(int b, int s, int x, int y) { // mouse click callback
    if (s == GLUT_DOWN) {
        cout << "Mouse click detected at coordinates x="
<< x << " and y=" << y << endl;
        if (b == GLUT_LEFT_BUTTON) {
            isReversed = !isReversed;
            cout << "Left mouse click. Reversing colors." << endl;
            glutPostRedisplay();
        }
    }
}

void myKeyboard(unsigned char c, int x, int y) { // keyboard callback
    switch (c) { // c is the key that is hit
    case 'q': // 'q' means quit
        exit(0);
        break;
    default:
        cout << "Hit q to quit. All other characters ignored" << endl;
        break;
    }
}

// Main program - It creates the game and then passes control to glut.
//-------------------------------------------------------------------------------
int main(int argc, char** argv) {
    cout << "Colors swap every 10 seconds.\n" << "Click left mouse button to swap colors.\n" << "Try resizing and covering/uncovering the window.\n" << "Hit q to quit." << endl;

    glutInit(&argc, argv); // OpenGL initializations
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB); // double buffering and RGB
    glutInitWindowSize(400, 400); // create a 400x400 window
    glutInitWindowPosition(0, 0); // ...in the upper left
    glutCreateWindow(argv[0]); // create the window

    glutDisplayFunc(myDisplay); // setup callbacks
    glutReshapeFunc(myReshape);
    glutMouseFunc(myMouse);
    glutKeyboardFunc(myKeyboard);
    glutTimerFunc(TIMER_DELAY, myTimer, 0);
    glutMainLoop(); // start it running
    return 0; // ANSI C expects this
Programming Assignment 1: A Simple 2D Game

Handed out Thu, Feb 9. The program must be submitted to the grader by Thu, Feb 16 (any time up to midnight). Submission instructions will be forthcoming. Here is the late policy: up to six hours late: 5% of the total; up to 24 hours late: 10%, and then 20% for every additional day late.

Overview. The goal of this assignment is to learn the basics of OpenGL and GLUT and to have fun. We have designed a very simple game where a character moves around on a region of the window called the board. Each component of the game has some basic requirements, on which your project will be graded. Feel free to add to or embellish upon any of the components as you see fit. (We give some suggestions below, but by all means, be creative.)

Design. (Roughly 10%) Before you begin, take a few moments to plan out your design. The goal here is to build a framework that will help you add features to the game. Although the minimal application is relatively small and it would be entirely possible to implement it with a single .cpp file and no objects, this simple structure will be inadequate for later assignments. A small amount of time planning a good design at this point will pay off later in the semester. Use what you know about object-oriented design. Some of the objects that you might want to encapsulate include a character class (which controls the motion and rendering of the moving character) and a board class (or more generally, something that stores the state of the environment). You may also consider creating classes for any relevant geometric objects, for example, a point class for storing a 2-dimensional point. (Note: GLUT callbacks cannot be class member functions.)

The Character. (Roughly 20%) The character will be your main source of interaction with the game. At the simplest level it is just a shape that moves around on the board. You will need to keep track of its position for rendering and collision detection and its speed and direction for maintaining its motion. The character must be drawn as an OpenGL object. Options include

- A colored square (using glRect*(())). (This is the minimum requirement.)
- A colored convex polygon (using GL_POLYGON). Experiment with glColor*(()) to change the color of the character.
- A circle. This can be approximated as an \( n \)-sided polygon for some large value of \( n \) (say 20 or more) or using gluDisk() and creating a gluQuadricon object.
- Any other 2D shape of your choosing formed by multiple convex polygons.
- A sprite, that is, a small bitmap (as in Super Mario Brothers or PacMan) that moves around the screen. You can use functions such as glReadPixels() and glDrawPixels() or use texture mapping (but this is getting way ahead of the class!). You will have to figure out how to input your image through file I/O. OpenGL does not offer much help here. For more information on this see the OpenGL FAQ.

http://www.opengl.org/resources/faq/technical/miscellaneous.htm#misc0050

The Board. (Roughly 20%) Create a board for the character to move around in. The character’s movement is confined within valid regions of the board. Options include

- A simple border drawn around the window, inset a little from the window’s border.
- (Optional) Add rooms connected by doorways or other obstacles to the board. The character cannot pass through walls or obstacles.
- (Optional) Add doors to the portions of the outer board. When the character passes through this door, it wraps around to the other side of the board (as in PacMan).
Character movement.  (Roughly 25%) The character should be in constant motion around the board, depending on a speed and direction that is controlled by the player. At a minimum the character should be able to move vertically and horizontally.

- Use glutIdleFunc() or glutTimerFunc() to create a smooth animation for the character.
- Use the arrow keys to control the movement direction. (Use the obvious mapping, e.g. ‘↑’ for up, ‘→’ for right, and so on.) See glutSpecialFunc().
- Use the ‘+’ and ‘-’ keys to increase and decrease the character’s speed.
- (Optional) Consider allowing the character to move in an arbitrary direction.
- (Optional) Change the character’s shape to fit its direction, so that it “faces” in the direction that it is moving.

Collision Detection.  (Roughly 20%) The character should not be able to pass through borders of the board (Hint: To determine whether a move is legal, check to see if the character’s bounding box is within a valid region of the board. If the move is legal, update the character’s position.) For example, if the character is moving left and hits a vertical wall, it stops there until the user changes its direction.

Other Elements.  (Roughly 5%)

- Use ‘q’ or ‘ESC’ to exit the game. (Note: the ‘ESC’ key has an ASCII numeric value of 27.)
- (Optional) If your system supports it, use ‘f’ to toggle into or out of full screen mode. (See glutFullScreen() and glutReshapeWindow() and glutPositionWindow().

Your Submission.  Your submission should contain everything that the TA will need to compile, execute, and test your program. This consists of:

- A file ReadMe.txt explaining what platform your program runs on (e.g., “Linux using g++” or “Windows using Visual Studio.NET”), how to compile your program (very important), how to run and execute your program, any special features you have implemented (very important), and any bugs or limitations that you are aware of.
- Files needed for compiling your program. (E.g. A Makefile if you are on a Unix system or the .sln and .vcproj files for Visual Studio.
- Your source files.
- Any additional files (e.g. image files).

These will all be combined in a bundle (using either zip or tar) and then submitted to the grader. Instructions will be forthcoming.
Program Assignment 2: 2-Dimensional Flocking

Handed out Thu, Feb 22. The program must be submitted to the grader by Tue, Mar 7 (any time up to midnight). Submission instructions will be forthcoming. Here is the late policy: up to six hours late: 5% of the total; up to 24 hours late: 10%, and then 20% for every additional day late.

Overview. Many video games and simulation visualization systems involve planning and coordinating the motion of a group of intelligent agents, such as birds in a flock or a school of fish. The purpose of this assignment is to implement a simple 2-dimensional program that models the behavior of a group of moving objects in the plane, called boids (a concept introduced by Craig Reynolds). Flocking motion satisfies some basic elements:

Separation: The boids attempt to maintain a certain minimum separation distance between each other and with any obstacles that may be present.

Cohesion: Ideally the boids should remain together as a group. (When an obstacle is present, the boids are allowed to split up to avoid the obstacle. Ideally they should regroup, once they have gone around the obstacle, assuming that the obstacle is not too large.)

Alignment: Boids tend to move at roughly the same speed and in roughly the direction as nearby boids.

The above properties define the nature of the motion at local level, but does not constrain the groups overall motion. The animator must be able to control the global motion of the boids as well. For example, the boids should move toward some goal point or follow a leader whose motion is prespecified. In this assignment, you are to write a program to simulate the motion of a group of two-dimensional boids, subject to the above general requirements. Basic credit will be based on the set of capabilities that you successfully implement. Extra credit will be based on the TA's subjective judgment of how interesting or complex your motion and rendering is.

Flocking. (Roughly 40%) Your program must implement the following basic elements of flocking.

Local flocking behavior: The motion exhibited by your flock should satisfy the three elements: separation, cohesion, and alignment, described above.

Global trends: In order to implement global control of motion the boids will attempt to follow the mouse cursor in the graphics window. The location of the cursor will be indicated (e.g. by a small ‘+’) drawn in the window.

Number of boids: The number of boids can be adjusted as the program is running. Hitting the ‘+’ key creates a new boid at a random location in the window, and ‘-’ removes a random boid from the scene.

Randomness: While individuals in a flock appear to be doing the same thing, they are rarely doing the exact same thing. Add randomness to the motion to improve the realism of the flocking behavior.

Modeling the boids. (Roughly 25%) In this part of the assignment, you will model the boids using closed polygonal loops. Boids consist of a body, two appendages (wings) and a head. We have modeled all four parts out of the same polygonal loop, which is provided in boid2D.poly. You should scale the size of the boid to an appropriate size for your application. The general shape is sketched (not accurately) in the figure below left and on the right we show what it might look like when flapping.

Here is the breakdown for the boid model as seen in the demo. Assuming you have created an appropriate transformation, $T$ to translate, rotate and scale your boid, and letting $Scale_p$ be the amount that you uniformly scaled the model to fit into your scene.

The left wing has the following transformations (with respect to $T$):
- Translate $10/\text{Scale}_p$ in the y-direction.
- Scale 0.8 in the x-direction and 0.6 in the y-direction.
- Rotate 90 degrees clockwise (or an angle $\leq 90$ if flapping).

Similarly, the right wing has the following transformations (with respect to $T$):

- Translate $-10/\text{Scale}_p$ in the y-direction.
- Scale 0.8 in the x-direction and 0.6 in the y-direction.
- Rotate 90 degrees counter-clockwise (or an angle $\leq 90$ if flapping).

The head has the following transformations (with respect to $T$):

- Rotate 180 degrees
- Scale x- and y-direction by 0.8

Note that these are transformations that we happened to like. Please feel free to modify the parameters as you please, but it is a requirement of the project that your boid consist of independent pieces relative to a central body because we want you to practice using Modelview transformations for drawing them.

We have defined a .poly filetype containing a single convex polygonal loop. The format is as follows:

```
\langle \text{number of vertices } n \rangle
0 \hspace{1em} x_0 \hspace{1em} y_0
1 \hspace{1em} x_1 \hspace{1em} y_1
\ldots
n - 1 \hspace{1em} x_{n-1} \hspace{1em} y_{n-1}
```

Note that vertex indices start at 0. There is an implicit edge between every two consecutive vertices. There is also an edge from the last vertex to the first one. For example, a unit length square whose lower left corner is at the origin would have a .poly file that looks like this:

```
4
0 0 0
1 1 0
2 1 1
3 0 1
```

**Other considerations.** (Roughly 35%) In addition, your program should implement the following features:

**Wing Flapping:** Draw your boids so they look like birds flying. The motion should be realistic (e.g., all birds do not flap at exactly the same time), so add some random factor to the flapping motion.

**Obstacles:** Some number of disjoint circular obstacles are present in the scene. The locations of the obstacles are specified in a reasonable way at run time (e.g. by giving their radii and center coordinates in an input file). Boids should avoid obstacles. If the flock is split by flying around a small obstacle, it should regroup in a natural way.

**Remaining in the window:** Boids should not be allowed to fly outside the window.

**Pausing:** By hitting the ‘P’ key the simulation can be paused and resumed. The ‘+’ and ‘-’ keys should work when the program is paused or running.
Extra Credit Ideas. Here are some ideas for extra credit.

Other Models: Create a .poly file(s) with a new model for a boid. Your model can be any shape or set of shapes. Keep in mind that OpenGL’s GL_POLYGON mode will only render convex polygons. The shape that we will provide in boid2D.poly will be convex. But if you design your own shape and it is nonconvex, you will need to break it into convex pieces. Alternatively, you can have OpenGL do this for you. (Hint: look into glu*Tess* functions in Red Book Chapter 11.)

Multiple Flocks: Have multiple flocks, which should avoid each other. (You may assign boids to flocks randomly.)

Complex obstacles: Allow for more complex obstacles and/or overlapping obstacles.

Predator: Have a predator boid (a hawk) that flies around randomly (or under mouse control). The other boids must avoid the predator at all costs, even if it means violating the flocking rules. Once the predator is at some distance, the boids should resume their flocking behavior.

Other Targets: See what happens when you set the target of each boid to be the position of the previous boid in your ordering. You will probably still want the first boid to follow the cursor.

Programming Hints. Let us consider the simplest case in which there are no obstacles. Each boid is specified by its current location point \( P \), its directional angle \( \theta \), and its current speed \( s \). Instead of \( \theta \) and \( s \), it may be more convenient to use a velocity vector \( \mathbf{v} \). It is possible to convert one to the other using the following formulas:

\[
\mathbf{v} = \begin{pmatrix} s \cdot \cos \theta \\ s \cdot \sin \theta \\ 0 \end{pmatrix}
\]

and

\[ s = \| \mathbf{v} \| \quad \text{and} \quad \theta = \arctan(v_y/v_x). \]

(The best way to compute the arc tangent is using the built-in function \( \text{atan2}(v_y, v_x) \), which returns an angle from \(-\pi\) to \(+\pi\).) Whether you choose to represent velocity using \( s \) and \( \theta \) or using \( \mathbf{v} \) is up to you. Whichever you choose, you will probably need to be able to convert to the other.

In order to do the animation, you will need to set up a continuous event loop. This can either be done using \text{glutTimerFunc()} (with a small time delay, say 1/30 of a second) or \text{glutIdleFunc()}. Each time this callback is invoked, your program will update the locations of the boids, by adding the current velocity vector to their current location and calling \text{glutPostRedisplay()}. Rather than adjusting the location of the boid at each step, it is better to adjust the boid’s velocity vector incrementally, and then move the boid according to this velocity vector. This results in a smoother motion. Each step of the animation involves the following elements:

(a) For each boid, determine the other boids and obstacles that are close by (within some fixed distance, and perhaps giving higher priority to things that are in front).

(b) Update the velocity vector for each boid in order to satisfy the various flocking and obstacle/avoidance constraints.

(c) Move each boid by adding an appropriate scaled copy of its velocity vector.

(d) Redraw the scene.

The most challenging step is (b). Each of the various flocking properties has a certain “pull” on the velocity vector. For example, if you are too close to another boid, this tends to push the velocity vector away from this other boid. (See the figure below.) For cohesion, compute the locations of the boids within some limited radius and compute the centroid (center of mass) of these locations. The boid’s velocity vector will be set to move it towards this centroid. For alignment, compute the average of the velocity vectors of the boids in some limited distance, and change your velocity vector so that it is closer to the average. To avoid obstacles, check whether there is an nearby obstacle in front of the boid. If so, you want to turn the boid in the direction that most easily avoids the obstacle. This turn has the effect of influencing the current velocity vector.
In summary, there are a number of "corrections" to the current velocity. In order to determine the updated velocity, assign weights to these corrections and then add the weighted sum to the current velocity vector. The assignment of weights is a nontrivial task. Some forces (e.g., the desire to avoid obstacles or predators) are much stronger than others (e.g., the desire to maintain cohesion). It is probably a good idea to put some upper bound on how much the velocity can change, to avoid unreasonably fast velocity changes.

Note that because the various effects only serve to influence velocity, it may be the case that boids do occasionally crash into each other. This is not considered to be an error, provided that they quickly move apart.

If you are interested, further resources on Boids and flocking behavior can be found on the web at the URL “http://www.red3d.com/cwr/boids”.

Programming Assignment 3: 3-Dimensional Flocking

Handed out Tue, Mar 28. The program must be submitted to the grader by Tue, Apr 18 (any time up to midnight). Submission instructions will be forthcoming. Here is the late policy: up to six hours late: 5% of the total; up to 24 hours late: 10%, and then 20% for every additional day late.

Because this is a relatively long assignment we want to encourage people to start early. So we will give extra credit to anyone who submits an initial working program by Tue, Apr 11, which implements the project’s basic requirements.

For further information, see the sample executable of our program, and the ReadMe.txt file, which comes with it for various program parameters and settings.

Overview. In the last programming assignment we considered the simulation of a flock of synthetic birds, called boids. In this project we will consider a 3-dimensional flock of boids. These follow the same basic behaviors as 2-dimensional boids: separation, cohesion, alignment, and tendency to pursue a goal point. In this project, we will also add the following elements:

3D Modeling: Since the scene is now in 3-dimensional, we need create a 3D model for our boids.
Steerable Goal: Rather than having the goal simply follow the mouse, the goal point will be a flying point. Through keyboard inputs (or mouse, if you prefer) the user controls the flight path of the goal.
Multiple Views: The camera can be positioned in various locations, one at a fixed location, one behind the boids, one looking at the boids from the direction of the target, and one from a boid’s point of view (a sort of “boid’s-eye view”).
3-dimensional Obstacles: Obstacles will consist of colored vertical cylinders. Boids must avoid obstacles by flying around them.
Environment: The scene will take place within a texture mapped cube called a skybox. The skybox simulates a complex outdoor scene using only six rectangular textures.
Graphics effects: To add realism, the scene will be lighted. Fog and other effects can be added for extra credit.

Basic Requirements. As usual, there will be basic elements, which must be implemented for partial credit, and optional elements that can be added to this. Your program must implement the following basic elements for 60% credit.

General setup: The ground lies along the \( x, y \) coordinate plane and the \( z \) axis points up to the sky. The scene takes place within a large texture mapped cube, the skybox, which will provide your boids ample area to fly around. There is no requirement that the boids stay within this region, however. There will be large cylindrical obstacles within this region.

For example, our skybox was a large cube of side length \( 10,000 \) units. Our boids have an initial speed of around \( 10 \) units per second and are roughly \( 20 \) units in length.

Create an initial flock of boids. These can be placed randomly in a confined region of space. (We started ours with 10 boids placed randomly in a roughly \( 200 \times 200 \times 200 \) box near the point \((-250, -50, 100)\).)

The initial goal should be placed at a moderate distance away from the initial flock and should be given an initial velocity, so that the simulation starts in a reasonable state. (Ours was at \((500, 500, 500)\) with velocity \((0, -5, 0)\).)
**Steerable Goal:** The goal point is no longer tied directly to the mouse. Instead it flies through space under the user’s control. If you do nothing, the goal flies at a constant velocity. You can slow it down and speed it up and turn it left and right, up and down through some combination of keyboard and mouse input. (We used ‘<’ and ‘>’, for slowing down and speeding up and used the arrow keys for steering) For testing purposes, it should be possible to fly your goal anywhere: through obstacles, under the ground, outside the scene.

The goal point must be rendered on your image (although you may want an option that hides it) and it should be drawn in a manner that it is clearly distinguished from the boids. (We drew ours as a sort of 3-dimensional plus-sign, which is aligned with the coordinate axes."

Additionally, since we found steering the goal to be a bit unwieldy, we found it helpful to include a feature to reset the goal to a known position (in the demo, the ‘r’ key resets the goal).

**Implementation note:** There are a few ways to handle the goal control. We implemented our goal much like a boid, with a position $P_g$ and a velocity vector $\vec{v}_g$, but the goal does not observe any of the boid behavior rules. Incremental speeding up and slowing down were handled by scaling $\vec{v}_g$ vector times an appropriate factor.

To move the goal velocity up and down, we added (to $\vec{v}_g$) a vector that is parallel to the z axis, and whose length depends on the length of $\vec{v}_g$. To turn left and right, we did something similar, but rather than using the $z$ unit vector, we used a vector $\vec{p}$ of unit length that is perpendicular to both $\vec{v}_g$ and the $z$ unit vector. (This perpendicular vector can be computed using the cross product.)

**Multiple Views:** Your program should support (at least) three different views. These views are based on the relative position of the flock and the goal. The user can switch between these, say through keyboard input.

In the descriptions below, let $C$ denote the centroid point of the flock. Let $G$ denote the location of the goal. (See the figure on the first page.) Let $\vec{u}$ denote the vector directed from $C$ to $G$ and let $d$ denote the distance from $C$ to $G$. Let $M = (C + G)/2$ be the midpoint between the flock centroid and goal. Let $r$ be the maximum distance of any boid to $C$, that is, it is the radius of a sphere centered at $C$ that contains all the boids.

**Default View:** This view is taken from an observer (a bird watcher?) located at a fixed position. (In our case this was at the point (0, 0, 500)) The view should be centered about $C$. You may adjust the field-of-view to simulate a telephoto lens, but this is not required.

**Boid’s Eye View:** This view is taken from the point of view of a random boid, and looks in the direction of the boid’s velocity. Since the center of the boid is inside the boid, you might find it necessary to translate the point of view forward a little in the direction of the boid’s velocity.

**Trailing View:** This view is taken from behind and above the flock, and should include the boids and the goal position.

For example, our approach was to move backwards from $C$ along the direction of $-\vec{u}$ to a distance of $d + 5r$ from $C$ and then upwards (parallel to the $z$ axis) by a distance of $d + r$. (See the above figure.) We then took a view centered at $M$ whose $y$ field of view is 30 degrees. Your distances may vary.

**Target View:** This view is taken from behind and above the target, and should include the boids and the goal position.

This view is analogous to the trailing view, however it will be looking from the front of the goal looking back towards the boids. In this case, move forwards from $C$ along the direction of $\vec{u}$ to a distance of $d + 5r$ and the upwards by a distance of $d + r$. As in the trailing view case, we took a view centered at $M$ whose $y$ field of view is 30 degrees.

For the trailing and target views, as the distance between the flock’s centroid and the goal grows, the camera will zoom out, and as it shrinks, the camera will zoom in. Since these views are not always perfect for viewing the scene, we found it useful to have controls that would zoom in/out and up/down through keyboard input by modifying the distances used above. In our code, the viewpoint can be switched using the numeric keys ‘1’–‘4’.

**Lighting:** Your program should use at least two light sources to illuminate your scene. (See our ReadMe.txt file for information on how we set up our lights. You may put yours elsewhere.)

**Additional Requirements.** The following requirements should be added for full credit. The point values are indicated with each one.
**Obstacles:** (10% total) The scene will have obstacles for the boids to avoid. Minimally, these obstacles will be cylinders that are aligned with the $z$-axis. For full credit, you will need to read in and render these obstacles. Implementing obstacle avoidance will be extra credit. The exact locations of the obstacles will be provided by a file, which we will provide. (See the file `ReadMe.txt` for explanation of the input file format.) You may create your own format for custom obstacles, but your program must still be able to read in our obstacle file. You can use OpenGL procedures for drawing these shapes. Please note, there is no `glutSolidCylinder()` function, and the `gluCylinder()` function does not render the caps on the two ends of the cylinder, so you will have to do this yourself.

**3-dimensional Boids:** (10%) Boids should be drawn as 3-dimensional surfaces. As before, they should face the direction in which they are flying. We created a surface of revolution by revolving the points in `boid3D.poly` about the $x$-axis (see the diagram below). For correct lighting, we also calculated the vertex normals for each generated vertex. Using this surface as a primitive, we created a 3-dimensional version of our four component boid from project 2. We have kept the four component model of our 2-dimensional models (as in project 2), and extended these into 3-dimensional boids with roughly the same look. Minimally, you will need to create a 3-dimensional shape that is able to orient itself in a certain direction and that can “flap” its wings.

**Skybox:** (10%) The skybox is a cube surrounding the scene. We texture-map the six sides of this cube to simulate a complicated scene with only 6 rectangles. We have provided a set of six images generated from the freeware program, `terragen`, in `.bmp` and `.ppm` format. We have also provided sample code for reading in a `.bmp` file and for using it to texture map (see image below for more details).

**Other features:** (10%) Below are some other features to implement for full credit.

- **Flapping:** Animate your boids so they flap their wings. As in project 2, to maintain realism, the boids should not all flap their wings in sync.
- **Pause/Single Step:** The user should be able to pause the program, advance it by a single step and then resume to continuous mode. This is very useful in debugging as well as our testing.
- **Reset Target:** Implement a simple function to reset the location of the target to a known location. (This is very useful!)
Ideas for Extra Credit. Here are some ideas for extra credit. Feel free to be creative and invent those of your own. Beware to avoid enhancements that involve OpenGL extensions, since we will likely not be able to test them.

Create your own skybox: Download Terragen (http://www.planetside.co.uk/terragen/), or some other tool and create your own textures for the skybox. There are numerous great examples and tutorials on the web for creating amazing outdoor scenes in terragen. You will probably find it helpful to use this script sky.tgs to generate the six views.

Fog: Add fog to your scene to make it look more realistic.

Banking: When a boid turns, it should bank in the direction of the turn. The angle at which the boid banks should depend on the sharpness of the turn. (Achieving good looking banking is rather tricky. You might consider the differences between the boid’s velocity vector and the goal’s velocity vector, where both are projected onto the x, y-coordinate plane. This method has a problem when the boid’s attempt to fly vertically, however.)

Obstacle avoidance: Reading in and rendering the obstacles is a requirement for this project. For extra credit implement obstacle avoidance as a new criterion for updating a boid’s velocity. Ideally, when the boid approaches an obstacle head on, it should attempt to go around the obstacle and similarly when it approaches an obstacle from the top or bottom.

Shadows: When the boids are close to the ground plane, have them cast shadows by projecting onto it. This can be as simple as a vertical projection of the 3-dimensional shape, or as complicated as a true light shadow by projecting from the light source onto the ground plane. From the light

Split Views: Rather than switching between views, have an option that creates multiple viewports showing the various views.
Programming Assignment 4: 3-d Flocking with Terrains

Handed out Tue, Apr 25. The program must be submitted to the grader by Tue, May 9 (any time up to midnight). Here is the late policy: up to six hours late: 5% of the total; up to 24 hours late: 10%, and then 20% for every additional day late.

For further information, see the sample executable of our program, and the ReadMe.txt file, which comes with it for various program parameters and settings.

Overview. In the last programming assignment we extended our simulation of a flock of boids to a 3-dimensional texture mapped environment. This project is a continuation of the last project, and as such has all the same requirements. We will make a copy of our Project-3 source code available, which you may borrow (or even steal) from. In this project we continue to work on the boid’s environment. We will be adding the following elements:

3D Procedural Terrain: The boids will fly over an actual mountain range rather than just a texture mapped floor.

Procedural Textures: To add realism without much extra geometry, you will generate a bumpy surface texture for the mountain range.

Reflective Lakes: We will use the stencil buffer to simulate the reflective surface of a lake.

Video: We will create a short recording of our projects. (A nice thing to add to your web page.)

Basic Requirements.

As usual, there will be basic elements, which must be implemented for partial credit, and optional elements that can be added to this. Your program must implement the following basic elements for 70% credit.

We will be using a Perlin noise function to generate our procedural terrains and textures.
An example is shown in the figure. Your code should depend on several parameters (such as the number of octaves and the persistence), to enable several interesting effects. You may use online resources such as http://freespace.virgin.net/hugo.elias/models/m_perlin.htm and Ken Perlin’s website http://www.noisemachine.com.

Procedural Terrain: To make our environment more interesting, we will be adding a simple 3D terrain. Using your Perlin noise function create a (monochromatic) image, which we will use as a height map for our terrain. We will use the locations in the image \((x, y)\) as locations and the pixel values as heights. Since this will be used as a mountain range, we would like the height map to have a nice coherence (which means that your Perlin noise should have a low persistence value). In place of the bottom image of the sky box, render the resulting terrain (e.g. as a series of triangle strips). Scale your terrain so it fills the entire bottom of the sky box. As you did in Project 3, compute normal vectors for shading.

Procedural Textures: We can also use the Perlin noise function to simulate a bumpy texture for our terrain. Render another Perlin noise function which will be used as our texture. Remember that OpenGL only supports textures whose dimensions are powers of 2. If you use GL_REPEAT as your GL_WRAP_PARAMETER, you can repeat the texture several times, which will enable you to maximize the variability in your texture compared to the storage space. Your texture should have dimensions of at least \(256 \times 256\), but can be larger depending on the capabilities of your graphics card (please limit it to \(1024 \times 1024\) for our testing purposes.

Outputting the Texture: For debugging purposes, your program should provide an option which allows us to view the procedural texture you created in the previous step. You may do this in one of two ways. Either output your texture to a (say, a .bmp) file or provide an input option to display the texture directly to the graphics window (e.g., using glDrawPixels). For grading purposes, you must indicate how we can obtain and view your image in your ReadMe.txt file.

Height-based Color: In addition to the Perlin noise texture (which is monochromatic), we can add color to our scene according to the elevation of each point. To do this, we will provide a 1-dimensional texture map image. (You
may create your own, but we have provided a sample 1-d texture in the file height4.bmp in the project resource directory.) You will input this image, but it will not be used as a texture map directly (since we are already using the Perlin noise for this). Instead, the elevation of each vertex will be mapped to pixel index within this texture, and this color will be applied to the vertex. OpenGL will blend the colors in between each vertex.

**Additional Requirements.** The following requirements should be added for full credit.

**Reflective Lakes:** (20% of total) We can simulate lakes by picking an arbitrary height along our terrain as the sea level and rendering a plane at that level. To create the impression of reflectivity, by using the stencil buffer to render the scene a second time, but this time transformed (through reflection and clipping) so that it lies beneath the water level.

We will discuss this more in class, and also see the page [http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=26](http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=26) for hints on how to do this.

**Video:** (10% of total) Record the generated frames of your animation and use them to create a video. Windows users can download a free trial of fraps ([http://www.fraps.com/](http://www.fraps.com/)), an OpenGL benchmarking and video capture tool. This program can automatically create videos using the ‘F9’ function key (the default storage location is in the install directory). The codec used by fraps to generate videos is specific to the fraps installation, so please use a program, such as virtual dub ([http://www.virtualdub.org/](http://www.virtualdub.org/)) to re-encode your video using a more popular codec such as DivX or Windows media.

Unfortunately, fraps only runs on Windows, but Linux users can use scrot or xvidcap (or a similar tool) to capture their videos. Refer to [http://freshmeat.net/search/?q=screen+capture&section=projects&Go.x=0&Go.y=0](http://freshmeat.net/search/?q=screen+capture&section=projects&Go.x=0&Go.y=0) for more options. If you find a really useful tool, please let us know so we can inform the rest of the class.

**Ideas for Extra Credit.** Here are some ideas for extra credit. Feel free to be creative. Beware to avoid enhancements that involve OpenGL extensions, since we might be able to test them.

**Height-based Texture:** The height-based color described above produces a rather crude looking shading, since it simply interpolates between the colors at the vertices. A more accurate method is to apply this as a texture map, in place of the Perlin noise texture. (This should be implemented as an option that can be enabled or disabled, so we can test both versions.)

**MultiTexture:** Combine the Perlin noise texture and the height-based texture using OpenGL’s capabilities for multi-texturing. Check the OpenGL documentation on how to do this. (Depending on your platform, you may need to install Glee. See [http://elf-stone.com/glee.php](http://elf-stone.com/glee.php).)

**Ground avoidance:** Your boids should treat the ground as an obstacle and avoid intersections, by flying above it.

**Perching:** Create the impression of boids perching on the ground. When a boid comes close to the ground, it lands and stays there for a fixed period of time. Then it flies off to rejoin the others.
Practice Problems for the Midterm Exam

The midterm exam will be on Thu, Mar 16 in class. The exam will be closed-books, closed-notes, but you will be allowed one sheet of notes, front and back to use for the exam. The following problems have been assembled from old exams and homeworks. They do not necessarily reflect the actual difficulty of problems on the exam or the total length of the exam. Some topics that we have covered this year are new, and have not been covered in old homeworks on exams.

**Problem 1.** Short answer questions. Explanations are not required, but may be given for partial credit

(a) In the call

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

explain in English (in a single sentence for each) the meaning of each of the capabilities that have been enabled.

(b) Name two different events or actions that could trigger a call to your display callback function. (This is the function passed to `glutDisplayFunc()`).

(c) What is back-face culling? For an average view, what fraction of the faces of a scene would be expected to be eliminated by this method? Explain briefly.

(d) Consider the hyperbola \( y^2 - x^2 = 1 \) in the projective plane. (Note this consists of two curves, one above the \( x \)-axis and one below the \( x \)-axis.) Consider the four extensions of the hyperbola out to infinity. What are the homogeneous coordinates of these points at infinity?

(e) In some graphics systems (not OpenGL) a left-handed coordinate frame is used. Give the \( 4 \times 4 \) matrix that performs a rotation counterclockwise about the \( x \)-axis by angle \( \theta \) in a left-handed frame. Contrast your result with the matrix for a right-handed frame.

(f) A user draws a triangle strip using `GL_TRIANGLES_STRIP` and gives \( n \) vertices. As a function of \( n \), how many triangles are produced? (Assuming no three collinear vertices and no duplicate vertices.)

(g) You are given a vertical line \( x = b \) and a pair of points \( P \) and \( Q \) in the plane. As a function of \( b \) and the coordinates of \( P \) and \( Q \), compute the affine combination of \( P \) and \( Q \) that lies on this vertical line. (See the figure below.)

(h) Which of the following statements is true of perspective projections? (Select all that apply)

(a) Lines are mapped to lines  
(b) Parallelism is preserved  
(c) Midpoints are preserved  
(d) Angles are preserved

(i) Given points \( P_0, P_1, P_2 \) in 3-space, and a viewer at point \( V \), give a geometric test to determine whether, from \( V \)'s location, the vertices of triangle \( \triangle P_0P_1P_2 \) appear in clockwise or counterclockwise order.
Problem 2. Consider the two frames $F$ and $G$ shown in the figure above.

(a) Express both $P$ and $w$ in homogeneous coordinates relative to frame $F$.
(b) Express both $P$ and $w$ in homogeneous coordinates relative to frame $G$.
(c) Give the $3 \times 3$ matrix which transforms a point represented in homogeneous coordinates relative to $G$ into its homogeneous coordinates relative to $F$. (If you wish, you may express your answer as the inverse of a matrix, without actually computing the inverse.)

Problem 3. Suppose that you want to linearly interpolate between the colors of the vertices of a triangle to the points in its interior. Consider the triangle in the figure above, with vertices at $(0, 0)$, $(1, 0)$ and $(1, 1)$. Let $C_0$, $C_1$ and $C_2$ denote the corresponding RGB color vectors assigned to these three vertices.

Derive a linear interpolation function $C(x, y)$ that maps a point $Q = (x, y)$ in the triangle to an RGB color vector by blending these three colors together. You may express your answer either as a formula (using affine geometry) or using pseudo-code. The final color should be a function of $x$ and $y$ and $C_0$, $C_1$, and $C_2$. Show your work.

Problem 4. Suppose that you have a square graphics (height equals width). The user has just resized the window so that it now has width $ww$ and height $wh$. As a function of $ww$ and $wh$, derive the arguments for glViewport() so that the new viewport is the largest square that fits within the window and is centered within the window. (See the figure below. The outer rectangle is the graphics window and the shaded rectangle is the viewport.) Recall that the calling sequence is:

```
glViewport(x, y, vw, vh);
```

where $(x, y)$ are the coordinates of the lower left corner of the viewport (where the origin is in the lower left corner of the window), and $vw$ and $vh$ are the width and height, respectively, of the viewport. (Hint: There are two cases, depending on whether the window is wider than tall, or taller than wide.)

Problem 5. OpenGL does not compute shadows. One way to produce the shadow of an object is to explicitly compute them yourself and just draw the shadows. Let us consider how to do derive a function to do this. Let $L = (\ell_x, \ell_y, \ell_z)^T$ be the coordinates of a light source and let $P = (p_x, p_y, p_z)^T$ be a point.

(a) Give a function that determines the projection of the point $P$ onto a point $Q = (q_x, q_y, q_z)^T$ on the $x, y$-coordinate plane, that is, the plane given by the equation $z = 0$. (Hint: Consider similar triangles as we did in deriving perspective transformations.)

(b) Express your answer to part (a) as a $4 \times 4$ projection matrix transformation $M$. This matrix should have the property that if $Q' = MP$, then after perspective normalization to $Q'$ (dividing by the last coordinate) we obtain the projected point $Q$.

Problem 6. Suppose that you are given a function drawWing(), which draws the wing shape shown in the figure below left. (This should be drawn on the $z = 0$ plane. In the figure the $z$-axis pointing up and out of the page.)

(a) Use the procedure drawWing() and other OpenGL functions (e.g., glPushMatrix(), glRotate*, glScale*, glTranslate*, etc.), to produce a procedure drawBird1() that draws the two wings shown in the center figure.
(b) Explain how to modify your solution to part (a) to produce a procedure `drawBird2()` that has exactly the same wing shape and size as in part (a), but the two triangles are now rotated up around the bird’s central axis, to simulate the flapping of a bird’s wings. The angle of rotation is 60 degrees. (Hint: Just show how to modify the solution to (a).)

Note: Your procedures `drawBird1()` and `drawBird2()` should be performed relative to the OpenGL matrix stack. In particular, their action is transformed by whatever matrix is currently at the top of the matrix stack, and on exit, the contents of the matrix stack should be restored to its original value.

Problem 7.
Throughout this problem you may assume that \( z = 0 \), and we are using `glOrtho2d` for viewing. Suppose that you have an OpenGL procedure `drawE()`, which draws an upper-case letter ‘E’ of height 1, so that its lower left corner coincides with the origin. Show how to achieve each of the following tasks using OpenGL. Assume that the current transformation mode is `GL_MODELVIEW`. You may call the procedure `drawE()`, but you may not modify its contents. On return, the OpenGL transformation stack should be unchanged.

(a) Give code for a procedure `drawE1(x, y, h)`, which draws the letter ‘E’ so that its lower left corner is at position \((x, y)\) (and \( z = 0 \)) and its height is \( h \). All three arguments are of type `GLfloat` and \( h \) is positive. Briefly explain.

(b) Give code for a procedure `drawE2(x, y, h)`, which draws an italic letter ‘E’ by slanting the letter by 30 degrees to the right. Again the lower left corner is at \((x, y)\) and the height is \( h \). (Hint: There is no OpenGL transformation which performs a shear, so you will need to derive the corresponding matrix. Recall that \( \cos 30^\circ = \sqrt{3}/2 \) and \( \sin 30^\circ = 1/2 \).)

Problem 8. Given a point \( P = (p_x, p_y, 1)^T \) in the plane and an angle \( \theta \), derive a transformation that rotates the plane by \( \theta \) degrees clockwise (not counterclockwise) about the point \( P \). As we did in class, express your answer as a \( 3 \times 3 \) matrix, so that it can be applied to a column vector in homogeneous coordinates. Show how you derived your answer.

Problem 9. You are given two circles in the plane of radii \( r_1 \) and \( r_2 \) centered at points \( C_1 = (c_{1x}, c_{1y}) \) and \( C_2 = (c_{2x}, c_{2y}) \), respectively. Derive an expression that tests whether these two circles overlap each other, but neither circle is contained within the other.

Problem 10. On the distant planet of Omicron Persei 8 (OP8), graphics viewports are designed so that the viewport origin is in the upper right corner. The \( x \)-axis points down and the \( y \)-axis points to the left. Let \( w \) and \( h \) denote the height and width of the OP8 viewport. (See the figure below.) Consider a rectangular drawing region (using the standard coordinate system) whose left and right sides are \( x_{\text{min}} \) and \( x_{\text{max}} \) and whose bottom and top sides are \( y_{\text{min}} \) and \( y_{\text{max}} \).

(a) Give the viewport transformation that maps a point \( P = (p_x, p_y) \) in the rectangular drawing region to the
corresponding point $V = (v_x, v_y)$ in an OP8 viewport. Express your transformation as two equations:

\[
v_x = \text{some function of } p_x \text{ and/or } p_y \\
v_y = \text{some function of } p_x \text{ and/or } p_y
\]

Show how you derived your answer.

(b) Suppose that the points $P$ and $V$ are expressed as homogeneous coordinates.

\[
P = \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix} \quad V = \begin{pmatrix} v_x \\ v_y \\ 1 \end{pmatrix}.
\]

Express the transformation of part (a) as a $3 \times 3$ transformation matrix $M$, so that

\[
V = MP.
\]

**Problem 11.** Your boss at Fred’s Pretty-Good Graphics Corp. wants you to write a procedure to generate a rendering of a cylinder in OpenGL. The cylinder is centered along the $z$-axis, has a height of $h$ units, and has a radius of $r$ units. Because OpenGL can only display polygons, you are to split the cylinder into $v_s$ vertical stacks (along the $z$-axis) and $r_s$ radial slices (around the $z$-axis). (For example, in the figure below left, $v_s = 4$ and $r_s = 8$.) Draw each face as a `GL_POLYGON`.

Give a procedure (in pseudocode):

```c
void cylinder(float h, float r, int vs, int rs);
```

to draw such a cylinder in OpenGL. (You may NOT use any GLUT procedures.) For full credit, you should specify both the vertices and associated normals, so that the shading of the cylinder will be smooth. You do not need to draw the top and bottom of the cylinder.

**Problem 12.** Suppose that a viewer is located at the origin $(0, 0, 0)$, and is looking along the $(-z)$-axis. On the plane $y = -1$, someone has put a circular pizza of radius 1 centered at the point $(0, -1, -3)$. Assume that we compute a perspective projection of the pizza onto the view plane $z = -1$.

The circular pizza appears to the viewer as an ellipse. (This is a fact.) For each of the following, indicate whether it is true or false. Give a formal justification for your answer based on your knowledge of the perspective
transformation. (Hint: You do not need to know the equation of an ellipse to solve this problem. If it makes your life easier, replace the circular pizza with a square.)

(a) True or false: The center point of the pizza appears to lie on a horizontal line that bisects the projected ellipse.

(b) True or false: The center point of the pizza appears to lie on a vertical line that bisects the projected ellipse.
Midterm Exam

This exam is closed-book and closed-notes. You may use 1 sheet of notes (front and back). Write answers in the exam booklet. If you have a question, either raise your hand or come to the front of class. Total point value is 100 points. Good luck!

Problem 1. (45 points; 3–6 points each) Short answer questions. (Keep your answers short.)

(a) When drawing triangle in a GL_TRIANGLES, OpenGL is very careful to draw all the triangles in a manner that makes their orientations consistent (all clockwise or all counterclockwise). Why is this important?

(b) Give one good reason for calling glutPostRedisplay() rather than calling your drawing callback function directly.

(c) Three points $p_0$, $p_1$, and $p_2$ define a triangle in 3-dimensional space. Explain how to generate a vector of unit length that is normal (orthogonal) to the plane on which triangle lies.

(d) You have an initially empty Modelview matrix stack and then you execute (in sequence) glLoadIdentity(), gluLookAt(), glPushMatrix(), and glTranslatef(). How many matrices are now on the Modelview stack? How many matrix multiplications have been performed?

(e) True or False: gluLookAt() should only be invoked when using perspective projection.

(f) Explain (in a couple of sentences) how the winding number method works to determine whether a point $q$ lies within a given polygon $P$.

(g) In the YIQ color model one of the three components stores luminance. What is the YIQ standard used for, and why is having a luminance channel useful?

(h) Two polygons are drawn using the z-buffer algorithm. Their projections overlap in the image plane. The closer polygon is partially transparent but the farther polygon does not appear at all. Based on your knowledge of the z-buffer algorithm, explain why this is so.

(i) What is the halfway vector and why is it relevant to computing specular reflection? (Answer in a couple of sentences.)

Problem 2. (15 points) The user has a drawing area that extends horizontally (in world coordinates) from $x_{\text{min}}$ to $x_{\text{max}}$ and vertically from $y_{\text{min}}$ to $y_{\text{max}}$. The viewport is of width $w$ and height $h$ and covers the entire graphics window. (See the figure below.) Let $(x_m, y_m)$ be the mouse coordinates as passed by GLUT to the glutPassiveMotionFunc callback function. Although the window need not be square, you may assume that the aspect ratios (width to height) of both the world and viewport windows are equal.

(a) As a function of these parameters, give the world coordinates $(x_w, y_w)$ of the point corresponding to the mouse location. Note: Assume that the mouse coordinates are as given by GLUT.

(b) Give a $3 \times 3$ transformation matrix that maps the homogeneous mouse coordinates $[x_m, y_m, 1]$ into its corresponding homogeneous world coordinates $[x_w, y_w, 1]$. (As usual, assume that these are column vectors.)

Problem 3. (20 points) An important utility is drawing text strings. This question will consider doing this, assuming you have access to a function that draws individual characters. Assume that each character is defined by an integer code (e.g., its ASCII or Unicode value). You are given the following:
The function `draw(i)` draws the character whose character code is \( i \), so that its lower left corner is at the origin. (See the figure below.)

Different characters have different widths. You are given an array of widths, where `width[i]` holds the width of the \( i \)-th character.

You are given a character string to draw, where `string[j]` contains the character code of the \( j \)-th character to be drawn, and `string.length` is the number of characters in the string.

```
width[\'C\']
```

```
\( \text{draw\('C\')} \)
```

```
\( \text{drawString1\('CMSC'\)} \)
```

```
\( \text{drawString2(x0, y0, s, theta, \"CMSC\")} \)
```

(a) Use the above and OpenGL transformations (`glTranslatef()`, `glRotatef()`, etc.) to implement a procedure `drawString1(string)` that draws the entire string on the \((x, y)\) plane so that its lower left corner coincides with the origin. (You must perform all the necessary translations so that the characters do not overlap. You may not use any of the built-in OpenGL or GLUT commands for drawing strings.)

(b) Implement a procedure

```
drawString2(x0, y0, s, theta, string)
```

that draws the given string so that its lower left corner is at the point \((x0, y0)\), it is scaled uniformly by the factor \( s \), and rotated by angle \( \theta \) (given in degrees). Except for the string, all the arguments are of type `GLfloat`. (Hint: You may call your procedure from part (a).)

Problem 4. (20 points) OpenGL clips objects that lie all or partially outside the view frustum. We will consider this in a 2-dimensional context, by ignoring the \( x \)-coordinates. Suppose that a view frustum is defined whose near and far clipping planes are at distances \( n \) and \( f \), respectively, and where \( t \) denotes half the height of the frustum on the near clipping plane. (See the figure below.) Consider a line segment \( \overline{pq} \), where \( p = (p_y, p_z) \) and \( q = (q_y, q_z) \).

```
\( \text{draw('C')} \)
```

```
\( \text{drawString1\('CMSC'\)} \)
```

```
\( \text{drawString2(x0, y0, s, theta, \"CMSC\")} \)
```

Part (a) Part (b)

(a) As a function of the parameters given, derive the implicit line equation \( g(y, z) = 0 \) for the line \( L \) that bounds the top of the view frustum.

(b) Let \( s \) denote the point (if any) where the line segment \( \overline{pq} \) intersects \( L \). Express \( s \) as an affine combination \( s = (1 - \alpha)p + \alpha q \) and solve for \( \alpha \).

(c) The line segment \( \overline{pq} \) may fail to intersect the upper edge of the view frustum between the near and far clipping planes. Explain in terms of the quantities you have how to determine this.
Practice Problems for the Final Exam

The final will be on Mon, May 15, 8:00–10:00am. The exam will be closed-books, closed-notes, but you will be allowed two sheets of notes, front and back to use for the exam. These problems have been assembled from old exams and homeworks. They do not necessarily reflect the actual difficulty of problems on the exam or the total length of the exam. (In particular, the topics of radiosity, shadows, Perlin noise, aliasing, and blobs were not covered in earlier semesters.) Also, do not forget to review material from before the midterm.

Problem 1. Short answer questions.

(a) Give a $4 \times 4$ matrix that performs the 3-dimensional affine transformation that translates a point by the vector $\mathbf{t} = (t_x, t_y, t_z)$, that is, it maps any point $P$ to $P + \mathbf{t}$.

(b) What is the reflection property that characterizes a pure diffuse reflector (also called a Lambertian reflector)? What is Lambert’s law of diffuse reflection?

(c) Explain the difference in how smooth shading is performed in Phong shading and Gouraud shading. Which method does OpenGL use?

(d) What is the inverse texture wrapping function, and why is it more relevant to the rendering process than the texture wrapping function?

(e) The Phong lighting model, as implemented in OpenGL, models light as a combination of four different effects. Name them. (No further explanation is needed.)

(f) What is back-face culling? For an average view, what fraction of the faces of a scene would be expected to be eliminated by this method? Explain briefly.

(g) You want to know whether a point $P$ lies on a given surface. From which representation of the surface is this question easier to answer: implicit or parametric?

(h) Define the angle of incidence between a ray and a surface to be the acute angle between the ray’s direction and the surface normal at the point of contact. As a ray goes from a medium of higher index of refraction to one of lower index of refraction does the angle of incidence tend to increase or decrease? Justify your answer.

(i) Let $P$ be a point in the interior of a Bézier curve of degree 3. True or false: The curve has $C^4$ parametric continuity at this point.

(j) State clearly which properties of the Bézier blending functions, $b_{i,d}(u)$, guarantee that the Bézier curve lies within the convex hull of the control points.

Problem 2. On the distant planet of Omicron Persei 8, they prefer a different way of specifying the perspective transformation. As with `gluPerspective`, they give the distances to the near and far clipping planes and the window’s aspect ratio, $a = w/h$. However, rather than giving the $y$-field of view, they instead give the distance $d$ to a sphere of a given radius $r$ that is centered along the viewing direction. The $y$-field of view is set so that the sphere exactly fills the window’s height. (See the figure below.) Write a procedure which, given these parameters, produces an equivalent call to `gluPerspective`. You may assume that the window is wider than tall, that is, $a \geq 1$. Here are the function prototypes.

```c
void op8Perspective(double d, double r, double aspect, double near, double far);
void gluPerspective(double fovy, double aspect, double near, double far);
```
(Hint: There are two cases, depending on whether \( a \geq 1 \) (wide windows) or \( a < 1 \) (tall windows). For partial credit, just do the case where \( a \geq 1 \), since this is simpler.)

**Problem 3.** Consider a new type of light called a spot-light. A spot-light is defined by giving a point \( P \), a vector \( \vec{v} \) (normalized to unit length), and an angle \( \theta \). The spot light illuminates any point that lies within an infinite 3-dimensional cone whose apex is \( P \) and whose angular radius about \( \vec{v} \) is \( \theta \). Write a function which, given a point \( Q \) in 3-space, and \( P \), \( \vec{v} \), and \( \theta \), determines whether \( Q \) is illuminated by the spot-light.

![Diagram of spot-light](image_url)

**Problem 4.** Consider the cone shown in the figure below. Its axis is along the \( z \)-axis, its apex is at height 3 on the \( z \)-axis and its base has radius \( r \) at the origin. We wish to wrap a rectangular texture shown in the figure below right around the central third of the cone. (Thus the bottom edge of the texture coincides with \( z = 1 \) and the top edge coincides to \( z = 2 \).) As \( s \) varies from 0 to 1, the texture should make one full revolution around the cone, starting from directly above the \( x \)-axis.

![Diagram of cone and texture](image_url)

Give the inverse wrapping function, which maps a point \((x, y, z)\) on the central third of the cone the corresponding point \((s, t)\) in texture space.

**Problem 5.** One way to speed up ray tracing algorithms is to enclose each object in a simpler enclosing shape (e.g. a sphere or a box) and first test intersection with the enclosing object. Its axis is aligned with the \( z \)-axis, its height is \( h \), and its base is located on the \( xy \)-plane and has radius \( r \). As a function of \( h \) and \( r \), compute the center and radius of the smallest (minimum radius) sphere enclosing this shape. (Hint: There are two cases to consider, one for fat cones and one for skinny cones.)

**Problem 6.** Fog is a relatively easy enhancement to a ray tracer. Fog is defined by three parameters, \( \text{fogStart} \), \( \text{fogEnd} \), and the fog RGB color \( F \). Let \( C \) be the color returned by the ray tracing procedure (ignoring fog). Let \( d \) be the distance from the ray origin to the point of contact. If \( d \) is less than \( \text{fogStart} \) then \( C \) is used, if \( d \) is greater than \( \text{fogEnd} \) then \( F \) is returned. Otherwise, an appropriate mixture of the two colors is returned. Give pseudocode for a function, which returns the fog color, given the following parameters: the ray origin \( P \), the ray contact point \( Q \), the traced color \( C \), and the other fog parameters \( \text{fogStart} \), \( \text{fogEnd} \), and \( F \). You may use any of the utility functions provided in the Color.h file.

**Problem 7.** This problem involves computing the ray intersection for a 2-dimensional axis-parallel ellipse. (This is easy to extend to 3-space, but it is simpler in 2-space.) Let \( P + t \vec{u} \) be the ray, where \( P = (P_x, P_y) \) and \( \vec{u} = (u_x, u_y) \) and let \( C \) be the center of the ellipse and let \( r_x \) and \( r_y \) be the lengths of the two axes. For a point \( Q \) to lie on the ellipse it must satisfy the following implicit equation:

\[
\frac{(Q_x - C_x)^2}{r_x^2} + \frac{(Q_y - C_y)^2}{r_y^2} = 1
\]
(a) Reduce the ray intersection problem to a quadratic equation, and derive the values of the two roots.

(b) Explain how to determine which root leads to the first intersection point with the ray, and whether the ray hits from the inside or the outside, or misses.

(c) Derive a formula for the 2-dimensional normal vector.

Problem 8. Write a procedure to test whether a ray $P + tu$, for $t > 0$, intersects a rectangle lying on the $z = 0$ plane, whose corner coordinates are $(-1, -1, 0)$ and $(+1, +1, 0)$. If the ray does not intersect, then the procedure should return special value MISS to indicate this, and otherwise it should return the $t$-value of the intersection point.

Problem 9. Give pseudocode (or a mathematical formula) for the diagonal strip 2-d texture function shown in the figure below. The dark color is $C_0$ and the white color is $C_1$. The horizontal and vertical width of each strip is 1 unit (and hence the diagonal width is $1/\sqrt{2}$).

Problem 10. We discussed procedural textures in class. It is also possible to define a procedural bump map. (Recall that a bump map does not actually change the shape of a surface. Rather, it generates a perturbed normal vector for each surface point, so that the result of shading appears bumpy.)

Consider the zig-zag bump function shown in the figure below left. The bump ridges are parallel to the $y$-axis. The height of each bump is one unit, and the distance between the tops of two consecutive bumps is some given value $w$. (See the right part of the figure.) Derive a function, $f(x, y) = (n_x, n_y, n_z)$, which given the $(x, y)$-coordinates of a point, returns corresponding the 3-dimensional normal vector for this point.

Problem 11. Consider three control points $p_0, p_1, p_2$ in 3-dimensional space.

(a) Give the function $p(u)$ for the Bézier curve of degree 2 defined by these control points, where $0 \leq u \leq 1$.

(b) Compute the derivative of this curve, as a function of $u$ and evaluate this derivative at the point $u = 0$. Express your answer as a function of $p_0, p_1, p_2$. 

3
(c) What do the results of (b) imply about the tangency properties of the Bézier curve at $u = 0$?

(d) Repeat part (c) but for $u = 1$.

**Problem 12.** Recall that in Problem 11 of the Midterm Practice Problems, your boss at Fred’s Pretty Good Graphics Corp. wanted you to write a procedure to generate a rendering of a cylinder in OpenGL. The cylinder is centered along the $z$-axis, has a height of $h$ units, and has a radius of $r$ units. Because OpenGL can only display polygons, you are to split the cylinder into $v_s$ vertical stacks (along the $z$-axis) and $r_s$ radial slices (around the $z$-axis). (For example, in the figure below left, $v_s = 4$ and $r_s = 8$.) Draw each face as a GL_POLYGON.

We have already seen in the solutions to the Midterm Practice Problems how to draw the cylinder in OpenGL. Your boss also wants you to wrap a texture around your cylinder. The texture image is $256 \times 256$ pixels. The texture should be mapped using GL_REPEAT, so that exactly four copies of the image go all the way around the cylinder. Explain how to modify the above procedure to provide the proper texture coordinates for each vertex. (You do not need to give any of the other OpenGL texture commands.)
Final Exam

This exam is closed-book and closed-notes. You may use 2 sheets of notes (front and back). Write answers in the exam booklet. If you have a question, either raise your hand or come to the front of class. Total point value is 100 points. Good luck!

Problem 1. (25 points; 2–5 points each) Short answer questions. (Keep your answers short.)

(a) Because the human visual system is most sensitive to green light, one color standard uses 8 bits for green and 4 bits each for red and blue. If we stored an image as a pixel map in this system, what is the depth of a pixel in this pixel map? How many bytes would it take to store a $100 \times 100$ image?

(b) You are given three points $p_1$, $p_2$, $p_3$ in 3-dimensional space. They are not collinear. In English explain what the shape is determined by the set of all convex combinations of these three points. What shape is determined by the set of all affine combinations of these points.

(c) Two white line segments are scan converted onto a black background using Bresenham’s algorithm, one is horizontal and the other is a diagonal line with slope 1. Which of these two lines appears brighter? Explain why.

(d) What does it mean when we say that a hidden surface removal algorithm is an object-precision algorithm. How about an image-precision algorithm.

(e) Under normal circumstances, light source positions in OpenGL should be set after calling gluLookAt. Explain in a couple of sentences why this is so.

(f) A ray is shot at a transmissive and nonreflective surface, and total internal reflection occurs. From which side did the ray strike: the one of higher IOR (index of refraction) or the one of lower IOR?

(g) What is the maximum number of times that a ray can intersect a quadric surface? (1 time? 2 times? 4 times? It can be arbitrarily high?) Briefly explain.

Problem 2 (30 points; 2–8 points each) Multiple choice. Note that some problems ask for one response and others ask for multiple responses. Explanations are not required unless specified.

(a) Consider a world configuration where the $z$-axis points up and the $x, y$-plane sits atop a shaded carpet that extends to $y = +\infty$. (See the figure below.) A viewer is located at the point $(0, 0, 1)$ on the $z$-axis and is viewing horizontally in the direction $(0, 1, 0)$ using perspective projection. On the viewer’s window where does the horizon (at $y = +\infty$) appear? (Select one and explain briefly.)

- Above the center of the window.
- Below the center of the window.
- At the center of the window.
- It will not appear within the window.

(b) Recall the term for specular reflection in the Phong illumination model: $\max(0, \mathbf{n} \cdot \mathbf{h})^\alpha \cdot \mathbf{L}_s \cdot \mathbf{C}_s$, where $\mathbf{n}$ is the normal vector, $\mathbf{h}$ is the halfway vector, $\mathbf{L}_s$ is the light color and $\mathbf{C}_s$ is the object color. What is the effect of increasing the $\alpha$ term? (Select one.)

- Brighter specular point.
- Dimmer specular point.
- Larger specular point.
- Smaller specular point.

(c) The OpenGL texture parameter to GL_REPEAT is used for which of the following effects? (Select one.)

- The texture is repeatedly uploaded with each redisplay cycle.
- The texture is wrapped around the object.
- The texture is stretched to fit the object.
- The texture is distorted to fit the object.
(B) To decrease aliasing, repeated linear interpolation of texel colors is performed.
(C) The texture is repeatedly overlaid and randomly offset to simulate motion blur.
(D) The texture is repeatedly tiled across the surface if the texture coordinates lie outside \([0, 1]\).

(d) The Nyquist-Shannon Sampling Theorem states that: (Select one.)
(A) A signal can be accurately reconstructed if the sampling rate is at least twice as high as the highest frequency in the signal.
(B) Prefiltering with a sync signal produces the most accurate signal reconstruction from sampled data.
(C) Weighted Gaussian sampling is superior to uniform random sampling.
(D) Doubling the sampling rate decreases aliasing by \(\sqrt{2}\).

(e) Which of the following statements is true for radiosity. (Select all that apply.)
(A) It treats all object surfaces as if they are possible light emitters.
(B) It computes light energy in a manner that is independent of the viewer’s location.
(C) For efficiency it relies on the fact that the form factors for two surface patches can be computed without consideration of the other patches in the scene.
(D) It can only handle monochromatic (uncolored) light.

(f) Which of the following statements is true for a Bézier curve defined by control points \(p_0, p_1, \ldots, p_d\). (Select all that apply.)
(A) It lies within the convex hull of its control points.
(B) It cannot self intersect.
(C) It is a parametric polynomial curve of degree \(d\).
(D) Every control point has global support. (It affects every point on the curve except the endpoints.)

Problem 3. (15 points) In this problem we derive the implicit and parametric representations of a cylinder. Consider an infinite cylinder of radius \(1/2\) centered whose central axis is parallel to the \(x\)-axis, and which passes through the point \((0, 2, 1)\).

(a) Give an implicit function representation of this cylinder, as \(f(x, y, z) = 0\).
(b) Present a parametric representation for the same cylinder, e.g. as \(x(u, v), y(u, v), z(u, v)\). What are the range of values for \(u\) and \(v\)?
(c) Consider the coordinates of an arbitrary point \((x, y, z)\) on this cylinder. As a function of these three coordinates, what is the normal vector to the cylinder at this point? Explain how you derived your answer. (It is not necessary to normalize your normal vector.)

Problem 4. (15 points) Here we consider how to generate a procedural displacement map to simulate the circular ripples that might result when a rock is dropped into a pool of water. Let us assume that the water lies on the \(z = 0\) plane. The ripples should have the following characteristics:

- The ripples start from the central point \((c_x, c_y, 0)\).
The distance between two successive wave peaks should be $f$.

The distance between the highest and lowest level of the ripples is $h$.

(a) Let's first consider the easier problem of what the ripple looks like along the $x$-axis alone (ignoring the $y$-coordinate). Give the equation of a cosine function that has the desired properties and achieves its maximum value at $x = c_x$. (Hint: It has the form: $z = a \cos(b(x + d)) + e$, for some appropriate choices of $a$, $b$, $d$, and $e$.)

(b) Generalize your answer to (a) to produce the full 2-dimensional displacement map in the form $z = f(x, y)$.

(c) Modify your answer to (b) so that the ripple heights decrease gradually as the distance from $(c_x, c_y)$ increases. (There are many ways of doing this. Any reasonable answer will be accepted.)

**Problem 5.** (15 points) Consider a sphere centered at the origin with a radius of 2. We wish to wrap a rectangular texture shown in the figure below right around the central portion of the sphere (from $z = -1$ to $z = +1$). (See the figure below.) As $s$ varies from 0 to 1, the texture should make one quarter revolution around the sphere, so that it starts on the $x$-axis and ends at the $y$-axis.

Give the inverse wrapping function, that maps a point $(x, y, z)$ on the specified region of the sphere to the corresponding point $(s, t)$ in texture space. (Note: If you cannot do this for the sphere you can get 50% partial credit by solving the same problem on a cylinder of height 4 ranging from $z = -2$ to $z = +2$.)