CMSC 427: Computer Graphics

Fall 2010

http://www.cs.umd.edu/class/fall2010/cmsc427/

Instructor: Dave Mount. Office: AVW 3373. Email: mount@cs.umd.edu. Office hours: (See class web page.) I am generally available immediately after class for questions. If the question is short (a minute or so) drop by my office any time. Please send me email if you cannot make these times. (Don't be shy about doing this. I always set aside at least one hour each week for "unscheduled" office hours.)

Class Time: Tue, Thur 2:00-3:15, CSIC 1122.

Teaching Assistant: Philip Yang. Office: AVW 1112. Email: phi@cs.umd.edu. Office hours: (See class web page.) If you cannot make these times, please feel free to contact Philip to set up another time.

Course Overview: This course provides an introduction to the principles of computer graphics. In particular, the course will consider methods for modeling 3-dimensional objects and efficiently generating photorealistic renderings on color raster graphics devices. The emphasis of the course will be placed on understanding how the various elements that underlie computer graphics (algebra, geometry, algorithms and data structures, optics, and photometry) interact in the design of graphics software systems.

Text:

Required: *Fundamentals of Computer Graphics*, (3rd Edition), by Peter Shirley and Steve Marschner, 2009. ISBN: 978–1–56881–469–8.

Recommended Reference: *OpenGL Programming Guide: The Official Guide to Learning OpenGL*, (6th Edition), by OpenGL Architecture Review Board, et al., Addison-Wesley, 2007. (Also known as "the Red Book.")

Prerequisites: MATH 240 (Linear Algebra) and CMSC 420 (Data Structures). Knowledge of C or C++.

The course involves a considerable amount of mathematical reasoning involving 3-dimensional objects (points, lines, spheres, and polygons). Knowledge of and the ability to solve problems in linear algebra and calculus will be assumed. Although the course will be "self-contained" and all this material will be reviewed before using, it would be a good idea to review your old class notes and textbooks on calculus and linear algebra.

The course involves some nontrivial programming projects. Although a specific knowledge of data structures is not essential, I will assume that you are capable of writing and debugging moderately complex programs in either C or C++.

Course Work: Course work will consist of a combination of written homework assignments (around 2) and programming projects (around 4). Homeworks are due at the start of class. Late homeworks are not allowed (so just turn in whatever you have done by the due date). Programming assignments will typically be due at midnight of the due date. They are subject to the following late penalties: up to six hours late: 5% of the total; up to 24 hours late: 10%, and then 20% for every additional day late.

There will be two exams: a midterm and a comprehensive final. Tentative weights: Homeworks and projects 35%, midterm 25%, final exam 40%. The *final exam* will be *Thu, Dec 16, 10:30am–12:30pm.*

As a courtesy to the grader, homework assignments are to be written up neatly and clearly, and programming assignments must be clear and well-documented. Although you may develop your program on whatever system you like, for final grading your program must execute either on a workstation (Microsoft Windows, Linux) in the Linux labs. (Which do not have special graphics cards extensions.) If you develop your program on some other platform, it is your responsibility to see that it can be compiled and executed on one of these machines. Excessive delays due to system incompatibilities will result in point penalties.

Some homeworks and projects will have a special challenge problem. Points from the challenge problems are *extra credit*. This means that I do not consider these points until *after* the final course cutoffs have been set. Each semester extra credit points usually account for at least few students getting one higher letter grade.

Academic Dishonesty: All class work is to be done independently. It is best to try to solve problems on your own, since problem solving is an important component of the course, and exam problems are often based on modifications of homework problems. You are allowed to discuss class material, homework problems, and general solution strategies with your classmates. But, when it comes to formulating or writing solutions you must work alone. You may use free and publicly available sources, such as books, journal and conference publications, and web pages, as research material for your answers. (You will not lose points for using external sources.)

You *may not* use any service that involves payment, and you *must* clearly and explicitly cite all outside sources and materials that you made use of. I consider the use of uncited external sources as portraying someone else's work as your own, and as such it is a violation of the University's policies on academic dishonesty. Instances will be dealt with harshly and typically result in a failing course grade.

Unless otherwise specified, you should assume that that the University Code of Academic Integrity (http://www.president.umd.edu/policies/docs/III-100A.pdf) applies.

Syllabus: The topics and order listed below are tentative and subject to change.

Introduction: Basic concepts, graphics devices, graphics programming.

Graphics programming: OpenGL, graphics primitives, color, viewing, event-driven I/O, GL toolkit, frame buffers.

Geometry in Computer Graphics: Review of linear algebra, affine geometry, (points, vectors, affine transformations), homogeneous coordinates, change of coordinate systems, quaternions.

3D Viewing: Scaling, rotation, translation, orthogonal and perspective transformations, 3-d clipping.

Illumination and Shading: Diffuse and specular reflection, Phong and Gouraud shading, transparency, and shadows.

Texturing and Surface Detail: Texture-, bump-, and reflection-mapping.

Real-Time Shadows and Reflection: Real-time shadows, light-maps, shadow Z-buffer, shadow volumes, buffering techniques.

Depth Cues and Hidden Surface Removal: Culling methods, z-buffer algorithm, depth-sort, BSP trees.

Rasterization and Filling: Bresenham's algorithm, filling, sampling and anti-aliasing.

Sampling and Aliasing: Spatial and temporal aliasing, sampling theory, prefiltering, super-sampling, stochastic sampling, signal processing.

Physically-Based Modeling: Basic physics, kinematics, kinetics, integration, mass-spring systems.

Ray Tracing: Ray-tracing model, reflective and transparent objects, shadows, light transport.

Animation: Key frames, interpolation, motion capture, physics-based animation, collision detection/response.

Curves and Surfaces: Representations of curves and surfaces, interpolation, Bezier, B-spline curves and surfaces, NURBS, subdivision surfaces.

Shading languages: GPU programming.

Global Illumination: Gamma-correction, halftoning, color models, radiosity, photon mapping.

A Short Introduction to C++ for Graphics Programmers (who know Java and C)

This is a brief introduction to C++, for people who know Java and C. We will focus particularly on aspect of C++ that will be the most useful for graphics programming, and will intentionally avoid some aspects of C++, which, while important, are not essential for writing programs needed in this course.
Disclaimer: This information has all be hastily thrown together. The code fragments have not been tested. I apologize in advance for any misinformation.

Primitive types and objects: In Java, everything is either a primitive type (int, char, float, etc.) or an object. Objects in Java are essentially references to objects. C++ inherits its basic types from C, and adds an two additional types, classes and references. The C++ types include primitive types (int, char, float etc.), enumerations, C-style structures and classes, pointers, and references.

As in C, a pointer is an address in memory, and an array is a pointer to the first element of an array. References in C++ are similar to references in Java, but they can be used for primitive types as well as for objects. Their most common use is in passing parameters to functions. Note that structures and classes in C++ behave differently than in Java. In Java, assigning one class object to another copies the reference. In C++, assigning one class object (by default) does a byte-by-byte copy of one object to the other.

Constants: One interesting feature of C++ is the ability to declare that an object is constant. This means that, once initialized, its value cannot be changed. (This avoids the `#define` constants that crop up in many C programs.) Here is an example:

```
const float FREEZING_POINT = 32.0;
const int WINDOW_WIDTH = 800;
const int WINDOW_HEIGHT = 300;
const char QUIT = 'q';
```

By convention, constants are expressed using all capital letters.

Classes: Class syntax in C++ is quite similar to Java. Note however, that a semicolon is placed at the end of a C++ class. Unlike Java, it is possible to define class methods outside the class body as well as inside. (In C++, class methods are usually called *member functions*.) In C++, it is common practice to define short 1-line member functions inside the class body and longer member functions outside the body. Here is a simple example, which defines a two dimensional vector object.

```
class Vector2d {
  private:
    double x;                           // member data
    double y;
  public:
    Vector2d() { x = y = 0; }           // default constructor
    Vector2d(double xx, double yy);     // constructor
    double getX() { return x; }         // getters
    double getY() { return y; }
    void setX (double xx) { x = xx; }   // setters
    void setY (double yy) { y = yy; }
    Vector2d addTo(Vector2d v);         // add to vector v
};
                        // member functions defined outside the class
Vector2d::Vector2d(double xx, double yy)
  { x = xx; y = yy; }

Vector2d Vector2d::addTo(Vector2d v)
  { x += v.x; y += v.y; }
```

We have defined the getters and setters inside the class. We have chosen to define the constructor and the `addTo` function outside the class. (In this case, the functions are all so short, they could have all been defined inside the class body.) Notice that, when outside the class, it is necessary to use the *scope resolution operator*, "`::`", to indicate that a name is associated with a particular class. Thus, when we define the function `addTo` outside the class, we need to specify `Vector2d::addTo`, so the compiler knows we are talking about a member of Vector2d.

What is the difference between defining a method inside or outside the class body? C++ takes a definition inside the class body to be a hint that this should be an *inline function*, which means that the function is expanded, rather than being called. This produces more efficient code, but excessive use of this feature results in unnecessarily long executable files (so called, "code bloat").

Stream I/O: Input and output in C++ is performed by the operators >> and <<, respectively. The standard output stream is called "cout" and the standard stream is called "cin." Here is a simple example.

```
int x, y;
cin >> x >> y;    // input x and y
cout << "The value of x is " << x << " and y is " << y << "\n";
```

The character "`\n`" generates an end-of-line. It is also possible to use standard C I/O (printf and scanf), but it is not a good idea to mix C++ stream I/O with C standard I/O in the same program.

Include files and namespaces: Following C's convention, declarations are stored in files ending in ".h" and most code is stored in files ending in ".cpp" or ".cc". Objects like cin and cout are defined by the system. At the start of each program, it is common to begin with a number of directives that include common system declarations. Here are some of the most useful ones.

```
#include <cstdlib>    // standard definitions from C (such as NULL)
#include <cstdio>     // standard I/O for C-style I/O (scanf, printf)
#include <cmath>      // standard C math definitions (sqrt, sin, cos, etc.)
#include <iostream>   // C++ stream I/O
#include <string>     // C++ string manipulation
#include <vector>     // C++ STL vector (an expandable vector object)
#include <list>       // C++ STL list (a linked list)
```

In order to keep your program names from clashing with C++ program objects, many named entities are organized into *namespaces*. Most system objects are stored in a namespace called "std". This includes, for example, cin and cout, mentioned above. To access objects from this namespace, you need to use a scope resolution operator, "::". For example, to refer to cin, you would use `std::cin` and the refer to cout you would use `std::cout`. To avoid this extra verbiage, you can invoke the "using" command to provide direct access to these names.

```
using std::cin;              // make std::cin accessible
using std::cout;             // make std::cout accessible
using namespace std;         // make all of std accessible
```

Memory Allocation and Deallocation: One of the principal differences with C++ and Java is the need to explicitly allocate and deallocate memory. Failure to deallocate memory that has bee allocated results in a *memory leak*, which if not handled, can cause your program to exhaust all its available memory prematurely and crash. As in Java, memory is allocated using `new`. This returns a pointer to the newly allocated object. Unlike the primitive C function `malloc`, the `new` operator returns an object of the specified type, and performs initialization by invoking the constructor. For example, to allocate an object of type Vector2d, we could do the following.

```
Vector2d* p;                 // p is a pointer to a Vector2d
p = new Vector2d(3, -4);     // allocate a vector, initialized to (3, -4)
p->setY(2.6);                // set its y-coordinate to 2.6
cout << p->getX();           // print its x-coordinate
delete p;                    // deallocate p's memory
```

Recall from C that, when dealing with pointers, the "`*`" operator is used to dereference its value and if p is a pointer to a structure or class, then "`p->xxx`" is used to access member xxx.

Array Allocation: It is also possible to use `new` and `delete` to allocate arrays. This is a common way to generate vectors whose size is known only at execution time. When deleting such an array, use "`delete [ ]`". Here is any example.

```
int n = 100;
Vector2d* p = new Vector2d[n];    // allocate an array of n vectors
p[5] = Vector2d(3, -4);           // set p[5] = (3,-4)
delete [] p;
```

Constructors and Destructors: The most common place where memory is allocated and deallocated is when classes are first constructed or destroyed, or in class member functions that insert new entries into a dynamic object. If a class allocates memory, it is important that when the class object ceases to exist, it must deallocate all the memory that it allocated. Whenever an object is about to cease to exist (e.g., the scope in which it was defined is exiting), the system automatically invokes a special class function called a *destructor*. Given a class `X`, the corresponding destructor is named `~X`. Here is a simple example, of a class, which allocates an array.

```
class VectorArray {
  public:
    VectorArray(int capac);         // constructor
    Vector2d at(int i) { return A[i]; }
    // some functions omitted...
    ~VectorArray();                 // destructor
  private:
    int n;                          // array capacity
    Vector2d* A;                    // array storage
};
                                    // constructor
VectorArray::VectorArray(int capac) {
  n = capac;
  A = new Vector2d[n];              // allocate array storage
}

VectorArray::~VectorArray() {
  delete [] A;                      // deallocate array storage
}
```

Note that you do not invoke the destructor (in fact, you can't). The system does it automatically.

Using STL Data Structures to Avoid Memory Allocation: A remarkably large amount of memory allocation and deallocation arises when dealing with two very common dynamic structures, vectors and lists. In C++, a *vector* is just an expandable 1-dimensional array (analogous to Java's ArrayList) and a *list* is just a standard doubly-linked list. Rather than going through the hassles of allocating your own vectors and lists, it is much simpler to use the built-in vector and list type provided by the C++ *Standard Template Library*, or STL. Here are a couple of examples of how to use an STL vector. To allocate a vector containing objects of type X, use the type declaration `vector<X>`.

```
int n = 100;
vector<int> myInts(n);            // allocate a vector with n ints
vector<Vector2d> myVects(n);      // allocate a vector with n Vector2d objects
myInts[5] = 14;                   // you can use "[]" to index entries
myVects[7] = Vector2d(3,-4);
myVects[7].setX(2);
myVects.push_back(Vector2d(1,5));  // you can add more entries to the vector
```

STL vectors and lists provide too many capabilities to be listed here. I will refer you to online documentation for more details. There are a number of other useful STL data structures, including dictionaries and priority queues.

An important aspect of STL vectors and lists are (1) they dynamically expand and contract, and (2) they provide their own memory allocation and deallocation. I have found that I virtually never need to do my own memory allocation if I rely on STL objects for all my simple data structures.

References: In C, all parameter passing to functions is performed *by value*. This has two important implications. First, altering the value of an formal parameter inside a function has no effect on the actual parameter in the calling function. If you want to modify the value of a parameter, you need to pass a pointer to the parameter. This is messy, since it implies that the function needs to dereference the resulting parameter whenever it uses it. Second, passing a large class or structure to a function means that its entire contents will be copied. This can be inefficient for very large structures. (Note that this does not apply to C++ arrays, however, since an array is just a pointer to its first element. However, this would apply if you were to pass an STL vector by value. Such an operation would involve making a duplicate copy of the entire vector.)

In Java, this was handled very elegantly by making all objects in references. Thus, small primitive types, such as int and float are passed by value, and all objects are passed by reference. If it is desired to change the value of a primitive type, standard wrappers, like `Integer` were defined.

In C++, this issue was addressed by defining a special type, called a reference. The following line defines the variable i to be an integer, and r to be a reference to this integer. All references to r are effectively "aliases" to references to i.

```
int i = 34;
int& r = i;                          // r is an alias for i
r = 27;                              // this is equivalent to i = 27
```

References solve the first issue arising in pass-by-value, since passing a reference parameter allows the function to alter that parameters actual value.

```
void f(int& r) {
  r = 27;                           // changes the actual parameter to 27
}
int main() {
  int i = 34;
  f(i);
  cout << i << "\n";                // this outputs 27
}
```

The other advantage of references, is that you can pass class objects to functions with the need to make copies of them. The following gives an example.

```
Vector2d add(Vector2d& u, Vector2d& v)
  { return Vector2d(u.getX() + v.getX(), u.getY() + v.getY()); }
```

Although passing class objects by reference is more efficient than by value, it has the downside that the function may inadvertently modify the value of the parameter, without the compiler being able to detect it. To handle this, C++ allows for something called a *constant reference*, which is a reference to an object that can be read, but not modified. Since the above function does not modify its arguments, it would more aptly be written in the following form:

```
Vector2d add(const Vector2d& u, const Vector2d& v)
  { return Vector2d(u.getX() + v.getX(), u.getY() + v.getY()); }
```

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: (Located on the third floor of the CSIC building.) If you have your own Linux system, these instructions are applicable, but adjustments may be needed depending on where you install the X11, OpenGL, GLU, and GLUT libraries.)

PC Windows: Your own PC running Microsoft Windows.

OpenGL is a 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).

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

$\langle WinDir \rangle$: This is your Windows system directory (e.g., C:\WINDOWS).

$\langle VCpp \rangle$: Your visual C++ root directory. For, example,
C:\Program Files\Microsoft Visual Studio 2005\VC\PlatformSDK

OpenGL is automatically installed on Windows machines. (To verify this, search for opengl32.dll and glu32.dll in your systems directory. You will probably need to install Glut, however. The easiest way to do this is to visit the following two web pages.

http://www.xmission.com/~nate/glut.html
http://pixel.cs.vt.edu/courses/4204/openglsetup.html

The first contains precompiled GLUT libraries. (Download the "GLUT for Win32 dll, lib and header file" not the "source code distribution".) The second explains where to put the essential files (glut32.dll, glut32.lib, and glut.h).

glut32.dll $\Longrightarrow$ $\langle WinDir \rangle$\system32 (or wherever opengl32.dll is)
glut32.lib $\Longrightarrow$ $\langle VCpp \rangle$\lib
glut.h $\Longrightarrow$ $\langle VCpp \rangle$\include\GL.

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. To check that you got it right, download the OpenGL sample program from the class web page, go to the directory VisualStudioNET, double click the solution file Sample1.sln, compile, and run it.

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 particular Unix/Linux system. In case you are interested, in the CSIC Linux Labs the library files libGL, libGLU, and libglut are located in /usr/lib. The include files gl.h, glu.h, and glut.h are located in /usr/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 CSIC 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, try "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, Effects, and Textures in OpenGL

This handout briefly describes a number of OpenGL's commands for controlling lighting, shading, fog, and texture mapping. 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 glGetIntergerv().) 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.

```
glShadeModel(GL_SMOOTH);
```

The shading interplation 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.

```
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.

Fog and Texture Options: 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.

GL_FOG_MODE	(scalar) How rapidly does the fog grow with distance. Either GL_LINEAR, GL_EXP or GL_EXP2
GL_FOG_START	(scalar) Distance where fog begins
GL_FOG_END	(scalar) Distance at which fog is total
GL_FOG_COLOR	(vector) RGBA of color of the fog

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.

glTexEnvi(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:

GL_MODULATE	multiply color components together
GL_BLEND	linearly blend color components
GL_DECAL	use the texture color
GL_REPLACE	use the texture color

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. When drawing things like skyboxes, that are not subject to lighting, GL_REPLACE is a good choice.

glTexParameterf(GLenum target, GLenum pname, GLfloat param)**:**
glTexParameterfv(GLenum target, GLenum pname, const GLfloat *params)**:**

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 screen is smaller than the corresponding texture pixel and "minification" applies in the when the pixel of the screen is larger than the corresponding texture pixel. Typical values are either

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

When minifying, there is another option, based on something called a *mipmap*. We won't discuss this, but if such a situation arises, mipmaps provide a good way to deal with this issue.

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.

```
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/drawing 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 provided a utility program (which I stole and adapted from a standard graphics text by F. S. Hill). It consists of a class RGBpixmap that stores an image. Its main method reads in an image from a .bmp file:

```
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.

```
#include "RGBpixmap.h"
// ...
RGBpixmap myPixmap;                             // declare RGBpixmap object
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);          // store pixels by byte
                                                // 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
    GL_UNSIGNED_BYTE,                           // my type
    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. This assumes a single texture object. Otherwise, a call to glBindTexture would be needed to set the current texture object.

```
// ...
glEnable(GL_TEXTURE_2D);                      // enable texture mapping
glMaterialfv(GL_FRONT_AND_BACK,               // white base color
           GL_AMBIENT_AND_DIFFUSE,
           glfv(white));
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
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();
glDisable(GL_TEXTURE_2D);                     // disable texture mapping
// ...
```

Figure 2: Displaying a texture-mapped object (with lighting). If no lighting is needed, your can disable lighting, omit the call to glMaterial, and use GL_REPLACE in place of GL_MODULATE.

Programming Assignment 1: Bubble Game

Handed out Thu, Sep 9. The program must be submitted by Thu, Sep 23 (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, two-dimensional geometry, and (hopefully) generate a simple and fun application. You are to implement a simple 2-dimensional computer game. You have some flexibility in how you implement the game (e.g., changing the user interface or modifying the game's behavior), subject to the requirement that your program contain all the same essential elements as ours does.

The game consists of two principal components, a balloon, which bounces around within the graphics window, and a cannon, that shoots pellets at the balloon. The balloon floats, but is subject to gravity, and will slowly sink to the ground if no other action is taken. Whenever the balloon hits one of the sides of the window it will bounce off, although at a slightly lower speed than the speed it had when it hit the wall. Whenever a pellet hits the balloon, it pushes the balloon in the direction from which it hit. A direct hit to the center of the balloon causes a stronger impulse. The cannon can be moved to the left and right along the bottom of the window, and it can be aimed.

Balloon: The balloon should be rendered as a colored circular disk (which you will approximate by a many-sided polygon, that essentially looks circular) along with some sort of asymmetrical pattern within the disk, so we can tell whether you have implemented rotation. When the balloon hits a wall it should bounce off, subject to a damping factor. (We will discuss this below.)

Even if the balloon hits no object, it is subject to gravity. With each refresh cycle, the y-component of its velocity will be decreased by a quantity that is proportional to the amount of elapsed time. To give the sense that this is a balloon, it may be a good idea to impose some maximum and minimum constraints on how fast the balloon is allowed to move. This will create the impression that there is air resistance.

Cannon: The cannon sits at the bottom of the window. It is rendered as a rectangle (or any other shape that indicates the direction it is aimed towards). The user must be able to adjust both the cannon's direction and its position. (In our implementation, we used to two keys to adjust its left-right position and the mouse to adjust its direction.)

The cannon shoots pellets. (In our implementation, this is done when the user hits the left mouse button.) Each pellet travels along a linear path. (We did not impose gravity on them, but you may do so as an enhancement.) A pellet lives until it hits the balloon or hits a wall, after which it simply disappears.

Collision response: There are two types of collisions to be handled. The first is between the balloon and a side wall, which we have mentioned above. The other is between the balloon and a pellet. For partial credit, all hits behave the same. For full credit, the effect should depend on where the bullet hits the balloon. For example, a direct hit on the balloon's center achieves a stronger push than a glancing hit closer to the edge of the balloon. In contrast, a glancing hit along the side of the balloon does not push it very hard, but induces it to spin, as a normal balloon would be expected to do.

Smooth animation: Use glutIdleFunc() to continuously update the state of your game. You can track the elapsed time between updates using *ftime* (or whatever timing function your system provides). See the class web page for further information.

Playability: Assuming a reasonable window size, the game-play should be reasonable. That is, the ball should not move too fast to be hit by the cannon, nor too slow to be too sluggish and slow. (Note that different machines have different refresh rates, and your program which may look quite reasonable on your system may be too slow or too fast when the grader runs it. To achieve consistency, use a function like *ftime* (see the class web page) to determine the actual elapsed time between update cycles, rather than just adjusting a fudge factor that works for your system.

Achieving window-size independence can be done by providing a scale factor for velocities that is based on the screen size. As the screen becomes larger, objects move proportionately faster.

If you don't implement these enhancements, please provide a way for the TA to adjust the speed of your game. (For example, in our game, hitting '+' or '–' causes the animation speed of the game to increase or decrease, respectively.)

User Interaction: Your program should provide user-inputs for at least the following actions. You may add others.

Move the cannon left and right: For example, using two keyboard keys or the arrow keys.

Aim the cannon: For example, directing it towards the current mouse position.

Reset the game: For example, hitting 'r' resets the game to its starting configuration.

Speed-up or slow-down: Make it possible for the user to increase or decrease the speed of the ball, for example, by hitting the '+' and '–' keys, respectively.

Final Submission: Your submission will be in the form of a file archive. (You may use any standard archiving software, such as Winzip, WinRAR, or Unix tar and gzip. If you are unsure, check to see that the TA has your favorite archiver.) The submission should contain everything that the TA will need to compile, execute, and test your program. This will consist of:

Readme: A file (e.g., Readme.txt), which explains everything the grader will need to know about how to compile and run your program. For example, this will include the platform on which your program runs (e.g., "Linux using g++" or "Windows using Visual Studio 2010"), 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. If you borrowed code from elsewhere, even if you modified it, please mention the source here briefly.

Makefile or Solution files: 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.

Source files: Your program source files.

Resources: Any additional files needed for execution (e.g., images or model files used by your program).

Programming Style: We will be reading your code to see that you implemented everything in a reasonable manner. Although style does not constitute a major part of the final grade, we will deduct points for programs that are poorly documented or that have convoluted structure. Since most of you are not

familiar with C++ programming, we will not deduct points for poor C++ programming style, but we may offer some tips for future projects.

Optional Elements: For partial credit (and during your initial implementation), you can keep things simple by not implementing rotation of the balloon, only translation. This will constitute a 10% deduction.

For extra credit points, here are some ideas for extensions to your project. (See the course syllabus on how extra credit points are counted.)

Better Graphics: Replace our simple circular balloon and rectangular cannon with more interesting geometric shapes, with more interesting colors.

Multiple balloons: Have multiple balloons appear at random times. The user should keep them all aloft.

Resize and full screen: It should be possible for the player to resize your window and/or run the program in full-screen mode. Your program should behave in a reasonable manner. (For example, making the window vary narrow should not result in the ball turning into an elongated ellipse.)

Start-up screen: Show a start-up screen when the game starts, and an ending screen showing the final score when it ends.

Special Effects: When a pellet hits the ball, animate the effect, for example, by changing the ball color or having the pellet change its velocity.

Handling Collisions: It is not important to achieve realistic physics in this project, provided that the behavior of the balloon seems fairly realistic. Here are a few thoughts about how to simulate smooth motion and realistic collisions. Let's begin with the simplest case, in which we do not consider rotation.

Units: First, decide which units you want to use in representing your world. For simple 2-dimensional projects list this, it is natural to simply use pixels as the unit of distance. Note, however, that this means that your program will run more slowly if the window size increases. You may prefer to imagine that the game takes place a virtual window whose height is fixed at 1, and then scale all your graphics to fit the actual graphics window when drawing.

State: All moving objects are characterized by their current *physical* state. This consists of the position p of the center of the object and the velocity v of the object. Both of these are vector quantities (the position is a point in space and the velocity is a vector). In an object-oriented approach, you may want to associate each object with these two quantities, along with other information, such as the object's radius.

Updates: With each update cycle (e.g., glut idle event), update the state. Based on the amount of time Δ_t that has elapsed since the last update, the object position can be updated as $p \leftarrow p + \Delta_t \cdot v$. Next, update the velocity of the objects due to gravity. (In our implementation, only the balloon is influenced by gravity.) This is done by decreasing the y-component of the velocity by Δ_t times a coefficient that is based on the strength of gravity. (I would suggest adjusting this by trial and error.) Finally, check for collisions. Each collision may result in a change of velocity and possibly a change of position.

Collision Detection: Let's consider collisions with a pellet. Collisions with the wall are similar. Imagine for now that the balloon is centered at point b and has radius R. Suppose that a pellet is positioned at point p and has radius r (see Fig. 1(a)). To determine whether they intersect, compute the vector $u = p - b$, and test whether $\|v\| \leq R + r$. If so, the pellet intersects the ball.

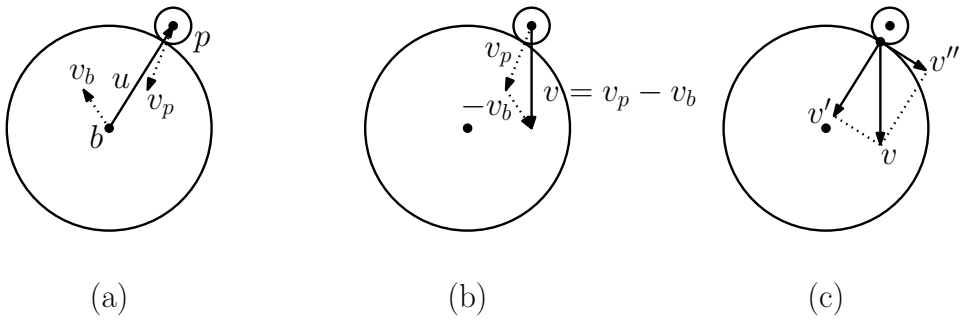


Figure 1: Geometry of collisions.

Collision Response: An important quantity of interest to the collision is the relative velocity at the moment of the collision. Let v_b denote the balloon's velocity and let v_p denote the pellet's velocity. Let $v = v_p - v_b$ be the *relative velocity* of the pellet relative to the balloon (see Fig. 1(b)). It may be convenient to think of the balloon as being at rest, and the pellet striking it at velocity v.

The observe the effect of the collision on the balloon, let's imagine a coordinate system placed at the point of contact between the two objects. Consider the vector $-u$, which is directed towards the center of the balloon and the relative velocity vector v. If the angle between these two vectors is small, the collision is "head on" and the effect will be much stronger than a glancing blow. To convert this into something we can compute with, let's decompose the vector v into two components, $v = v' + v''$, where v' is parallel to u and v'' is perpendicular to u (see Fig. 1(c)). This can be done with the following vector operations:

$$v' = \frac{(u \cdot v)}{(u \cdot u)} u \qquad \text{and} \qquad v'' = v - v'.$$

The more direct the collision, the larger that v' will be.

To compute the effect of the collision on the balloon, we set $v_b \leftarrow v_b + \alpha \cdot v'$, where α is a factor that takes into consideration various physical issues (such as the weight of the pellet, the weight of the ball, and the elasticity of the collision). I suggest adjusting α by trial and error.

Collisions with the wall are handled similarly. In fact, it is a bit simpler, because the vector u will be a horizontal or vertical vector (depending on which wall is hit) and this implies that v' and v'' will be horizontal and vertical as well. Because the walls are immovable, it is best to simply negate the appropriate component of the velocity. For example, if the right wall is hit, then the x-component of the velocity is changed from positive to negative. To simulate a collision that is not perfectly elastic, multiply the resulting velocity by a fudge factor between 0 and 1. (As the value gets closer to 1, the ball is very lively, and as it gets closer to 0, the balloon acts more like a lump of clay.)

Rotation: In order to process rotation, we add two more components to the state, the current angle ϕ of the ball (with respect to its starting position) its angular velocity ω (how fast it is rotating). These can be measured either in radians or degrees. (Math functions assume radians, but OpenGL functions assume degrees, so ultimately, you will need to deal with both.) By convention, positive rotations are counterclockwise and negative rotations are clockwise. Initially both

ϕ and ω can be set to 0. As with position, whenever a time period of Δ_t has elapsed, the angle can be updated to $\phi \leftarrow \phi + \Delta_t \cdot \omega$.

The value of ω is not affected by gravity, but it may change during collisions. Returning to our previous example, recall that v'' is the component of the relative velocity that is perpendicular to u. This vector is tangent to the balloon's boundary at the point of contact, and hence it determines the change of angular velocity. How do we compute the new angular velocity? We need to determine whether this tangent vector is directed clockwise or counterclockwise around the boundary. In the former case, the angular velocity is decreased, and in the latter, it is increased.

To perform the determination of clockwise or counterclockwise, let $v'' = (v''_x, v''_y)$ and let $u = (u_x, u_y)$. Let $z = u_x v''_y - u_y v''_x$. This is a scalar value. (Where did this formula come from? If you treat the the vectors v'' and u as vectors in 3-dimensional space, where the z-component is zero, the laws of physics says that their cross product $u \times v''$ gives the angular torque. If you were to compute this 3-dimensional cross product, you would find that the x- and y-components are zero, and the z component is the value of z as computed above. Don't worry about this for now, however.)

To update the angular velocity, set $\omega \leftarrow \omega + \beta \cdot z$, where β is a suitable factor that takes into consideration various physical issues (such as the second moment of inertia of the ball and the degree of friction). I suggest adjusting β by trial and error.

Homework 1: OpenGL and Affine Geometry

Handed out Thu, Sep 30. Due at the start of class Tue, Oct 12. Late homeworks will not be accepted, so turn in whatever you have finished by then.

Problem 1. Graphics modelers who are concerned with performance are interested in how to display graphics by transferring the smallest number of vertices to the GPU. Consider the shape shown in Fig. 1. Show how to draw this shape as a combination of OpenGL triangles, triangle strips, and triangle fans. For full credit your solution should (1) minimize the total number of calls to glVertex, (there are a number of correct solutions), and (2) the first triangle of each object (triangle, strip, or fan) should be oriented in counter clockwise order.

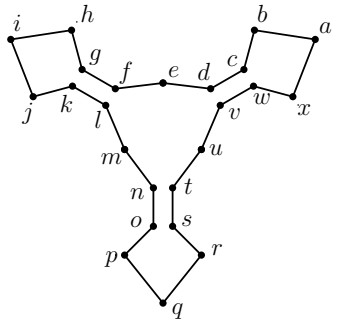


Figure 1: Draw using triangles, triangle strips and triangle fans.

Problem 2. You have landed a job with Air and Space Museum, and they want to produce a visualization of the moon rotating around the earth. You are given two functions, `drawEarth()`, which draws a picture of the earth centered at the origin and a function `drawMoon()`, which does the same for the moon. You are asked to write a function,

```
drawEarthAndMoon(int day, int hour, int minute);
```

which draws a picture of both the earth and moon for the given day, hour, and minute, over the course of a single lunar month. For simplicity, assume that a lunar month is exactly 28 days (it's actually a bit smaller) and an earth day is exactly 24 hours. The parameter *day* indicates the day of the month (in the range 0 through 27), *hour* indicates the hour within the day (in the range 0 through 23), and *minute* indicates the number of minutes (in the range 0 through 59).

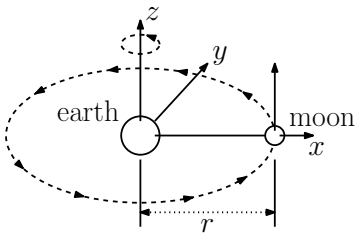


Figure 2: Earth and Moon visualization.

Assume the following (admittedly inaccurate) structure. The earth is placed at the origin and its axis of rotation is aligned with the z-axis. The moon is at distance r from the earth and it rotates about the earth in a circular orbit on the (x, y)-plane, from west to east (see Fig. 2). The moon's rotational axis is parallel to the z-axis. It

rotates about its axis so that, from an observer on the earth, the same side of the moon is always visible. At the start of the month, $(\text{day}, \text{hour}, \text{minute}) = (0, 0, 0)$, the moon should be placed at distance r from the earth along the positive x-axis. During each (24-hour) day, the earth undergoes one full counterclockwise rotation about its axis, and during each (28-day) month, the moon makes exactly one full counterclockwise rotation about the earth.

Give pseudocode for a procedure which given *day*, *hour*, and *minute*, generates the appropriate OpenGL transformation commands to render both the earth and the sun. Your procedure may assume that the matrix mode is `GL_MODELVIEW`, and you should save the current matrix state and restore this state when your procedure returns.

Problem 3. Consider the points and free vectors shown in Fig. 3, with the following homogeneous coordinates:

$$o = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \quad p = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \quad q = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \quad r = \begin{pmatrix} 2 \\ 2 \\ 1 \end{pmatrix} \quad s = \begin{pmatrix} 0 \\ -2 \\ 1 \end{pmatrix} \quad u = \begin{pmatrix} 2 \\ 1 \\ 0 \end{pmatrix} \quad v = \begin{pmatrix} -1 \\ 1 \\ 0 \end{pmatrix}.$$

Answer the following questions for these objects.

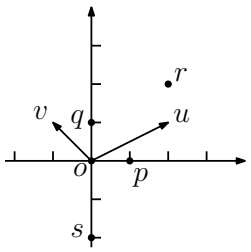


Figure 3: Affine geometry exercise.

(a) What is $\text{Or}_2(p, q, r)$? (Positive, zero, or negative?) Justify your answer by giving the determinant used to define orientation and the value of this determinant. (Recall the updated lecture notes for Lecture 4.)

(b) Repeat part (a), but for $\text{Or}_2(p, r, s)$.

(c) Express s as an affine combination of p, q, and r. (That is, $s = \alpha_1 p + \alpha_2 q + \alpha_3 r$ for what values of α_1, α_2, and α_3 that sum to 1.)

(d) Consider a coordinate frame F whose origin is p and whose basis vectors are u and v. What are the homogeneous coordinates of r relative to this frame? What are the homogeneous coordinates of the vector $w = r - s$ relative to this frame?

(e) Explain how to use the dot product to derive the angle between u and v. (You may express the angle as an appropriate trigonometric function, e.g., arccosine, arcsine, arctangent, of some scalar.)

Challenge Problem: Challenge problems count for extra credit points. These additional points are factored in only after the final cutoffs have been set, and can only increase your final grade.

Prove that your answer to Problem 1 is optimum in the sense that there is no way to draw the shape using triangles, triangle strips, and triangle fans that involves a smaller number of calls to glVertex than yours. You may assume that only vertices a through x are are used in calls to glVertex and that no two triangles are allowed to have overlapping interiors.

(Hint: Prove that no solution exists that uses a fewer number of triangles, triangle strips and triangle fans than yours. Then, argue that in order to minimize the number of calls to glVertex, it suffices to minimize the total number of triangles, triangle strips, and triangle fans.)

Programming Assignment 2: The 3-d Bubble Game (Phase I)

Handed out Thu, Oct 7. The first phase must be submitted to the grader by Thu, Oct 21 (any time up to midnight). (The due date may vary, depending on when the midterm date is set.) Submission instructions will be forthcoming. See the syllabus for the late policy.

Overview. The goal of this project is to implement a simple 3-dimensional animation involving a bouncing bubble amidst a collection of objects. The project will involve combining the following elements:

- Processing of both keyboard and mouse inputs
- Both automatic and user-controlled camera motion
- Lighting, shading, shadows, and texture mapping
- Simple physics:
  - Gravity, friction, air resistance
  - Collision detection and response (translational response and rotational response).

As always, you are welcome to modify our particular requirements, provided that your program implement at least the required elements. The assignment is broken into three phases.

Phase I: Implement keyboard and mouse inputs, user-controlled camera movement, rendering with light and texture mapping, motion with gravity, friction, and air resistance.

Phase II: Enhanced camera control, shadows, collisions with spherical objects, collision response with translation.

Phase III: Collisions with rectangular objects, collision response including both translation and rotation.

Phases II and III will be described later. In this first phase of the project, you are to implement the following elements. As always, we allow some flexibility in how you implement your program, provided that you achieve the main learning objectives. (If you are in doubt, please check with us.)

User Input: The user controls the bubble using both keyboard an mouse inputs. The bubble is moved horizontally using the arrow keys (forward, backward, left and right). Each press of a key induces an instantaneous *impulse* (e.g., kick) to the rolling bubble. Thus, the more frequently a key is pressed, the faster the bubble will move. Each impulse is generated relative to the camera position. For example, '↑' generates an impulse away from the viewer and '←' generates an impulse directed to the viewer's left. If the camera is repositioned (see below), the sense of each impulse is modified accordingly. If the user hits the right mouse button (or some other input of your choice), the bubble bounces upwards.

Your program should simulate a frictional with the ground, air resistance and gravity. Air resistance is applied with the object is not in contact with the ground, and friction applies when it is is.

Camera motion: Intuitively, the camera is located behind the bubble, but its elevation and distance can be adjusted. When the mouse is dragged up and down, the camera rotates upwards and downwards, relative to its current position. If the mouse is dragged left or right, the camera rotates horizontally

around the bubble. Through the use of two key inputs (we used 'o' and 'i') the camera can be zoomed out or zoomed in.

When the bubble moves upwards, the camera does not itself move upward. Instead, it continues to point to a position immediately underneath the bubble on the ground. It is possible for the bubble to leave the field of view. (In the next phase, we will add camera controls to handle this.)

Ground and Physics: In this first phase, there is only one object in the world with the bubble. This is broad rectangle, which we call the ground. The bubble is subject to friction, air resistance and gravity, and will bounce off the ground (with a bit of dampening). If it travels beyond the edge of the ground, it will start fall.

Rendering: Render the bubble as a solid sphere. Add a quick-and-dirty shadow on the ground, for example, by drawing a gray circular disk beneath it. Lighting should be turned off when drawing the shadow. (More sophisticated shadows will be added in later phases.)

Lighting: Your program should make use of at least one light source to illuminate elements of your scene. Extra credit points will be given for more elaborate lighting set-ups or special effects (such as spot lights or fog).

Texture mapping: Your program must have at least one element of texture mapping. One suggestion for achieving this is to add a skybox to your scene. A skybox is a large texture-mapped cube that surrounds the entire scene, which serves to provide a background image. We will provide some sample images and code for reading in image files and storing them in a manner that is compatible with OpenGL. You are free to generate your own skybox images, and you may apply texture mapping to other object in your scene.

Resources We will make a sample executable available on the class projects page (under "Projects") along with some other useful files (our object file and skybox images).

External Resources An important learning objective with this project (all phases) is your ability to process basic 3-dimensional geometry, rendering, animation, and simple physics (involving both linear and angular motion, collision detection, and collision response). In practice, much of this would be done with the aid of geometric modelers, a game engine, and a physics engine. However, for this project, we would like you to do as much of this as possible on your own, in order to acquire an understanding of how the internal elements of these systems work.

For this reason, other than OpenGL and GLUT, you are *not* allowed to make use of high-level software tools for tasks such as rendering, geometric modeling, or physics. However, you are allowed to download and use of small technical pieces of code (e.g., code for performing basic matrix or quaternion operations). If you are not sure whether some code satisfies our conditions of "fair use", please check with me.

Also remember that as part of course academic-honesty policies, all externally derived resources (e.g., code snippets or geometric models that you download from the web) must be acknowledged. The only exceptions are image files and materials that we provide for you and which are made available on the course web page. As part of your final submission, your "Readme.txt" file must indicate what resources you downloaded, where they appear in your project, about how much code was downloaded. If you made modifications, let us know what you did.

Practice Problems for the Midterm Exam

The midterm exam will be on **Tue, Nov 2** 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. You are responsible generally for material covered in class or appearing on class assignments.

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

(a) In the call glutInitDisplayMode(GLUT_RGBA | 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) In gluLookAt(), in what direction is the up vector *not* allowed to point. Explain.

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

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

(e) Which of the following statements are true of perspective projections? (Select all that apply.)

(i) Lines are mapped to lines

(ii) Parallelism is preserved

(iii) Midpoints are preserved

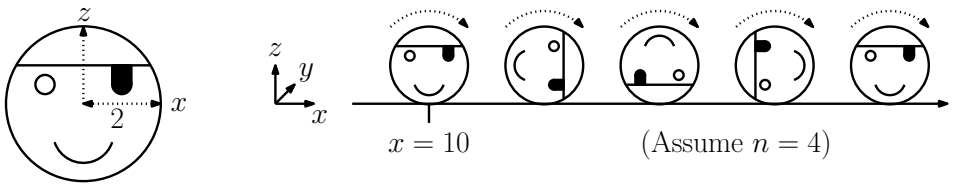
(iv) Angles are preserved (e.g., right triangles project to right triangles)

(f) Two spheres are being rendered using glutSolidSphere. One sphere is a pure *diffuse* reflector and the other is a pure *specular* reflector. Which of the two would require higher accuracy (that is, a greater number of slices and stacks) to produce a realistic rendering? Explain briefly.

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

(h) You have a triangle in 3-space, whose vertices are $p = (p_x, p_y, p_z)$, $q = (q_x, q_y, q_z)$, and $r = (r_x, r_y, r_z)$. Explain how to compute a vector v that is normal to this face, and directed to the front side (the side from which the vertices appear in counterclockwise order as $\langle p, q, r \rangle$). You are allowed to express your answer using vector operators, such as affine combinations, dot and cross products, etc.

Problem 2. You are given a procedure drawPirate(), which draws a 2-dimensional pirate face centered at the origin and lying on the x, z-plane. (See the figure below, part (a).) The radius of the circle forming the face is 1. Your goal is to produce a sequence of drawings of the face rolling along the x-axis, but scaled down to a radius of $1/2$. (See the figure below, part (b).)



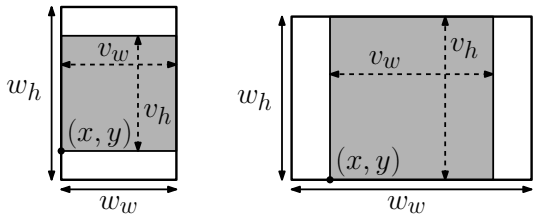
(a) (b)

To do this, you are to write a procedure rollingPirate(int n, int i). This procedure will be called $n + 1$ times, for $i = 0, 1, 2, \ldots, n$. Each call draws one image. When $i = 0$, the pirate will be displayed upright at $x = 10$. As i increases, the face rotates and translates to its next position. When $i = n$, it will undergo a full 360° rotation, as shown in the figure.

Give pseudocode for the procedure rollingPirate(n, i), which uses drawPirate( ) and the OpenGL matrix stack to draw the face at the desired location and rotation. On return, the Modelview matrix stack should be unchanged.

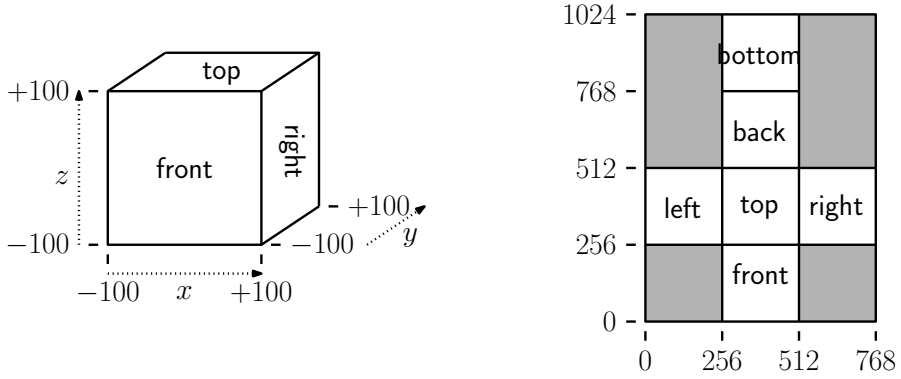
Problem 3. Suppose that you have a square graphics drawing area (height equals width). The user has just resized the graphics window so that it now has width w_w and height w_h, where w_w may not be equal to w_h. You want to define a viewport within this window that is (1) square and (2) centered as well as possible within the graphics window.

As a function of w_w and w_h, derive the arguments for glViewport() to achieve this effect. (See the figure below. The outer rectangle is the graphics window and the shaded rectangle is the drawing area defined by 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 v_w and v_h are the width and height, respectively, of the viewport.

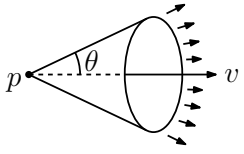


Problem 4. You are given a 6-sided cube, which is 200 units on each side and is centered about the origin. (See the figure below.) You are to wrap a texture around this box, which is presented to you as an image of height 1024 and width 768. (We will ignore the fact that OpenGL requires that texture dimensions are a power of 2.)

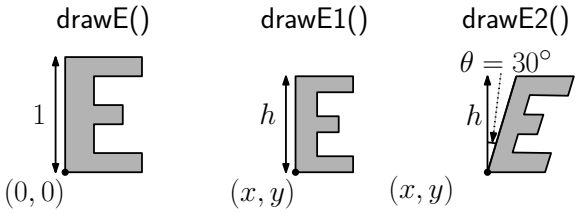
Give the OpenGL commands to draw *just the front face* of the cube (e.g., as a GL_POLYGON or GL_QUADS). The box should be drawn so that the vertices appear to be in counterclockwise order with respect to a viewer *outside* the box. You should specify the normal vector of the face (using glNormal3f), again directed outside the box. You should also specify the texture coordinates associated with each vertex (using glTexCoord2f).



Problem 5. Consider a 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 θ. 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 θ. Write a function which, given a point q in 3-space, and p, $\vec{v}$, and θ, determines whether q is illuminated by the spot-light.



Problem 6. In 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.

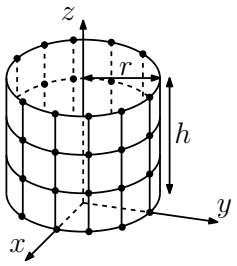


(a) Give code for a procedure drawE1(x, y, h), which draws the letter 'E' so that its lower left corner is at position (x, y) (and $z = 0$) and it has been uniformly scaled to be of height h. All three arguments are of type GLfloat and h is positive. Briefly explain.

(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 $\sin 30^\circ = 1/2$.)

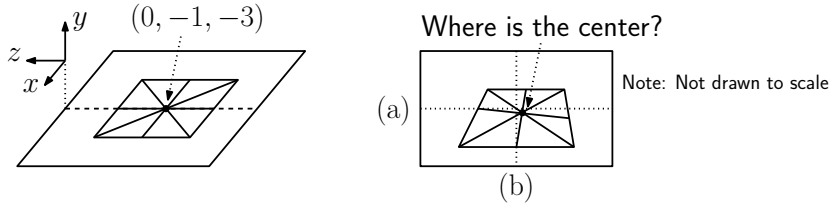
Problem 7.

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 we have $v_s = 4$ and $r_s = 8$.) Draw each face as a GL_POLYGON.



Give a procedure (in pseudocode) void cylinder(float h, float r, int vs, int rs), which draws 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 8. 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 square pizza of side length 2 centered at the point $(0, -1, -3)$. Assume that we compute a perspective projection of the pizza onto the view plane $z = -1$.

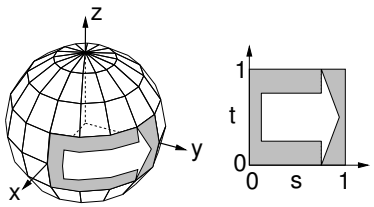


(a) Consider a horizontal line that bisects the projected pizza. Does the projected pizza center lie on, above, or below this line?

(b) Consider a vertical line that bisects the projected pizza. Does the projected pizza center lie on, left of, or right of this line?

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, imagine that the circular pizza is a square.)

Problem 9. 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.



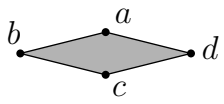
Give the *inverse wrapping function*, that maps a point (x, y, z) on the specified region of the sphere to the corresponding point (s, t) in texture space. (Note: If you cannot do this for the sphere you can get 50% partial credit by solving the same problem on a cylinder of height 4 ranging from $z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

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. (35 points; 3–5 points for each part or subpart) Short answer questions. Except where noted, explanations are not required, but may be given for partial credit.

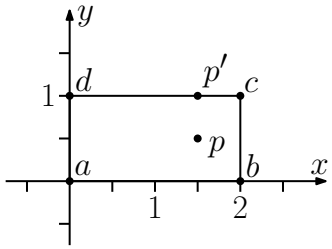
(a) Consider the four-sided shape shown below. Suppose that it is drawn using GL_TRIANGLE_STRIP, and back-face culling is *enabled*. Which of the following drawing orders produces a valid drawing of the shape?



$(i) : \langle a, b, c, d\rangle$ $(ii) : \langle a, b, d, c\rangle$ $(iii) : \langle c, b, d, a\rangle$ $(iv) : \langle d, a, c, b\rangle$

(b) In gluLookAt(), in what direction is the up vector *not* allowed to point. Explain.

(c) You are given a 2×1 rectangle with corner vertices a, b, c, and d, as shown below. Consider the points $p = (1.5, 0.5)$ and $p' = (1.5, 1)$.



(i) Express p' as an affine combination of d and c.

(ii) Is your answer to part (i) a convex combination?

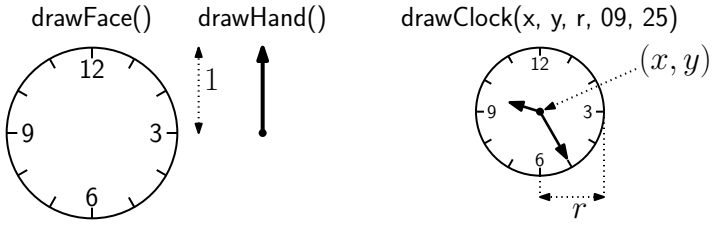
(iii) Express p as an affine combination of a, b, c, and d? (Hint: If you try to solve a linear system of equations, you are making this way too hard.)

(d) What is the principal difference between bump mapping and normal mapping? Of the two methods, which is more general (and why)?

Problem 2. (25 points) The following problem takes place in the x, y-plane. You are given two procedures (see the figure below):

drawFace() : Draws the face of a clock. The face is inside a circle of radius 1, centered at the origin.

drawHand() : Draws a hand of the clock. The length of the hand is 1, and it points straight up from the origin.

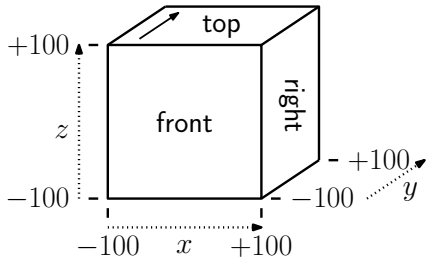


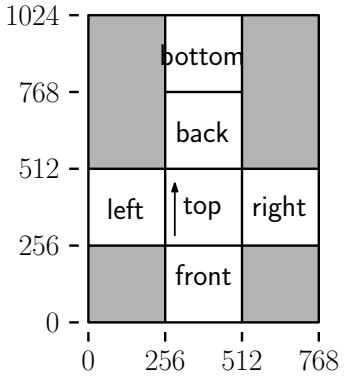
Using these two procedures, implement in OpenGL a drawing procedure drawClock(x, y, r, HH, MM). This draws a clock centered at the point (x, y), of radius r, with hands adjusted so that it shows the time $HH : MM$. The parameter HH is an integer from 0 to 11, indicating the hour, and the parameter MM is an integer from 0 to 59, indicating the minute.

The hands should be drawn at the appropriate locations. (You may either arrange the short hand to coincide exactly with the hour or to place it proportionately between two hours, depending on the minute value.) The long hand's length should be $3r/4$ and the short hand's length should be $r/2$. On return, the Modelview matrix stack should be unchanged.

Problem 3. (20 points) You are given a 6-sided cube, which is 200 units on each side and is centered about the origin. (See the figure below.) You are to wrap a texture around this box, which is presented to you as an image of height 1024 and width 768. (We will ignore the fact that OpenGL requires that texture dimensions are a power of 2.)

Give the OpenGL commands to draw *just the TOP face* of the cube (e.g., as a GL_POLYGON or GL_QUADS). The box should be drawn so that the vertices appear to be in counterclockwise order with respect to a viewer *outside* the box. You should specify the normal vector of the face (using glNormal3f), again directed outside the box. You should also specify the texture coordinates associated with each vertex (using glTexCoord2f).





Problem 4. (20 points) In the lecture on bump mapping, we discussed how to compute a normal vector from a parametric surface representation. Consider the parametric surface $p(u, v) = (u, 2v, 3uv)$, where u and v are real numbers. That is, $x(u, v) = u$, $y(u, v) = 2v$, and $z(u, v) = 3uv$.

(a) (8 points) Given a point (x, y, z) that lies on this surface, derive (as a function of x, y, and z) the corresponding parameter values u and v.

(b) (12 points) Derive a general formula, which given u and v, returns the (normalized) surface normal at $p(u, v)$. Show your work. (If you like, you may express your answer in terms of geometric operations, such as dot products and cross products.)

Programming Assignment 2: The 3-d Bubble Game (Phase II)

Handed out Tue, Nov 9. The second phase must be submitted to the grader by Tue, Nov 23 (any time up to 11:59:59pm). Submission instructions will be essentially the same as with the previous phase. See the syllabus for the late policy.

Overview. This is the second phase of the Bubble-Game project. Recall the overall goals of the project:

- Processing of both keyboard and mouse inputs
- Both automatic and user-controlled camera motion
- Lighting, shading, shadows, and texture mapping
- Simple physics:
  - Gravity, friction, air resistance
  - Collision detection and response (translational response and rotational response).

This is done in the context of a bubble that floats subject to user control in an environment with various objects. In Phase I, you implemented the basic manual camera control, the physical motion of the bubble (including gravity, friction, air resistance, and collisions with the ground), lighting, and texture mapping in the form of a sky box. In Phase II, you will implement automatic camera control, shadows, collisions with spherical objects, collision response (with translation). As always, we allow some flexibility in how you implement your program, provided that you achieve the main learning objectives. (If you are in doubt, please check with us.)

Automatic Camera Control: Recall that the camera is focused on a point that lies on the ground and is immediately below (or above) the center of the bubble. The spherical angles with respect to this point are controlled by dragging the mouse, and the distance is controlled by keyboard input. If the bubble ever moves outside the camera's field of view, the camera should automatically and smoothly zoom out (but keeping the same spherical angles) until the bubble becomes visible. If the bubble moves back into the field of view, the camera should move smoothly back to its default position (as specified by the user inputs).

(I found that getting this to look smooth and natural is challenging. One way to think about this is to imagine that there is a physical force, like a spring, that acts on the camera. When the bubble moves out of the field of view, this force tends to push the camera further out, and when the bubble moves back within the field of view, it tends to pull the camera back in.)

To create an initial sense of the overall environment, the camera should start at a great distance where the entire environment is visible, and then smoothly zoom into its initial position. (If you handle the camera using the above spring analogy, this can be implemented by placing the camera's initial position at an unnaturally large distance, and letting the spring pull it in to its proper starting location.)

Shadows: Your bubble and the other objects should cast shadows on the ground. Objects do not cast shadows onto each other, only on to the ground's upper surface. (When we get to Phase III, the ground will consist of a checkerboard pattern of blocks. It is not necessary to cast shadows onto the sides of these blocks.) Shadows should be cast accurately with respect to a nonvertical light source position (which may

be a point at infinity), rather than just being projected vertically straight down, as we did in Phase I. This must be done in such a way that, if the bubble moves off the edge of the ground block, the shadow must end at the edge of the block as well (see Fig. 1).

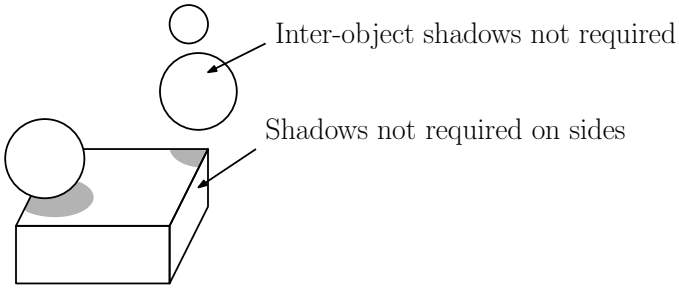


Figure 1: Shadows.

There are many ways to implement shadows. You may use whatever method you like. I would recommend the shadow-painting technique that we presented in class.

Transparency: For a more interesting look, draw your bubble as a transparent object. To do this, you will need to enable GL_BLEND, invoke glBlendFunc, and set the "A" component of the RGBA color. Information can be found on the web, for example:

http://www.opengl.org/resources/faq/technical/transparency.htm

Obstacles: You are to implement a collection of obstacles. You may design these to be any shapes that you like, but spheres are recommended because they are easy to draw and spherical physics is very simple. When the bubble hits any one of them, it bounces off in a physically realistic manner. At this point, you do not need to worry about rotation (but this will be added in Phase III). As with the ground, the collision should involve some dampening. The resources directory contains a file (objects.txt) which contains a description of the objects that we used, but you are not required to use this file. You may use whatever obstacles you like or specify them in whatever format you like.

Collision Detection and Response Although there are good data structures for performing collision detection (e.g., a quadtree), because the number of objects is relatively small, and assuming that you use a simple object type, like a sphere, it is efficient enough to determine collisions by brute force search, by comparing the bubble with each obstacle. (If you adopt a different set of objects or create objects of different types, this may change.)

Here is how to process collisions between spherical objects. We assume that the bubble is modeled as a ball centered at a point b with velocity v. The obstacles are rigid and do not move after a collision (but if you want to change this, you are welcome to). If the distance between the centers of the bubble and some obstacle is smaller than the sum of their radii, then they collide. Although, in theory, it is possible for the bubble to simultaneously collide with multiple obstacles, we will assume that the obstacles are sufficiently well separated that this cannot happen (or, if it does, process only one). If a collision is detected, let c denote the obstacle center, and let $u = c - b$ be a vector directed from the center of the bubble to the center of the obstacle that it collides with (see Fig. 2(a)).

To determine the collision response, we need to break the velocity v into two components, one aligned with the surface normal at the point. In particular, we decompose the vector v into two components, $v = v' + v''$, where v' is parallel to u and v'' is perpendicular to u (see Fig. 2(b)). As mentioned in class, this can be done with the following vector operations:

$$v' = \frac{(u \cdot v)}{(u \cdot u)} u \qquad \text{and} \qquad v'' = v - v'.$$

(In fact, the file Vector3d.h has two functions parProject and orthProject, which perform this useful decomposition.)

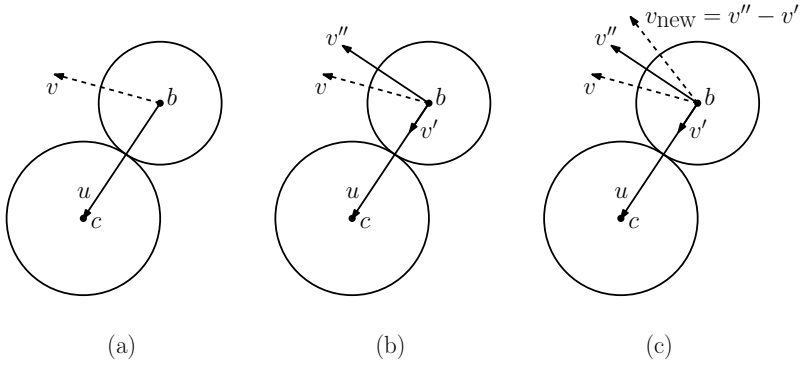


Figure 2: Geometry of collisions.

To compute the effect of the collision on the balloon, the obstacle exerts an impulse to the bubble in the direction $-v'$. Set $v_{\text{new}} \leftarrow v'' - \alpha \cdot v'$, where $0 < \alpha \leq 1$ is a factor that takes into consideration various physical issues (such as the elasticity of the collision).

Resources The existing sample executable works for Phase II as well. Hitting the keys '1', '2', or '3' switches among the various phases. In addition the skybox image, we will also include a file with obstacles. You are not required to use our file nor our format. You may specify the objects in any way you like.

Phase-I Requirements Part of your grade (about 20%) will be based on the requirements from Phase I. Thus, if you still have bugs or unimplemented features from your Phase I implementation, please check with us. We will be happy to help you to get these elements working.

External Resources An important learning objective with this project (all phases) is your ability to process basic 3-dimensional geometry, rendering, animation, and simple physics (involving both linear and angular motion, collision detection, and collision response). In practice, much of this would be done with the aid of geometric modelers, a game engine, and a physics engine. However, for this project, we would like you to do as much of this as possible on your own, in order to acquire an understanding of how the internal elements of these systems work.

For this reason, other than OpenGL and GLUT, you are *not* allowed to make use of high-level software tools for tasks such as rendering, geometric modeling, or physics. However, you are allowed to download and use of small technical pieces of code (e.g., code for performing basic matrix or quaternion operations). If you are not sure whether some code satisfies our conditions of "fair use", please check with me.

Also remember that as part of course academic-honesty policies, all externally derived resources (e.g., code snippets or geometric models that you download from the web) must be acknowledged. The only exceptions are image files and materials that we provide for you and which are made available on the course web page. As part of your final submission, your "Readme.txt" file must indicate what resources you downloaded, where they appear in your project, about how much code was downloaded. If you made modifications, let us know what you did.

Programming Assignment 2: The 3-d Bubble Game (Phase III)

Handed out Wed, Nov 24. The second phase must be submitted to the grader by Thu, Dec 9 (any time up to 11:59:59pm). Submission instructions are the same as with previous phases. See the syllabus for the late policy.

Overview. This is the third and final phase of the Bubble-Game project. Recall the overall goals of the project:

- Processing both keyboard and mouse inputs
- Both automatic and user-controlled camera motion
- Lighting, shading, shadows, and texture mapping
- Simple physics:
  - Gravity, friction, air resistance
  - Collision detection and response (translational response and rotational response).

This is done in the context of a bubble that floats subject to user control in an environment with various objects. In Phase I, you implemented the basic manual camera control, the physical motion of the bubble (including gravity, friction, air resistance, and collisions with the ground), lighting, and texture mapping in the form of a sky box. In Phase II, you implemented automatic camera control, shadows, collisions with spherical objects, collision response (with translation). In this phase, you will implement rigid body physics (that is, both translation and rotation) and collisions with both spherical and rectangular objects.

As always, we allow some flexibility in how you implement your program, provided that you achieve the main learning objectives. (If you are in doubt, please check with us.)

Rigid-Body Physical State: Since rotations will be considered in this project, the physical state of your bubble (which, up to now, consisted only of the center position and linear velocity) must be enhanced to include the angular orientation and angular velocity. I would strongly recommend that you use the same method described in Lecture 14, where the angular orientation is specified as a unit quaternion, and the angular velocity is expressed as a 3-dimensional vector. (This way, your physics computations should match ours, which will make debugging much easier.)

Checkerboard Ground: The ground should be replaced with a more interesting surface formed from rectangular blocks. This should be done in a manner so it is easy for us to cause your bubble to collide with the corners and edges of these blocks (since one of the elements to be tested in this phase is collision with rectangular blocks). In our implementation, we created a 10×10 checkerboard and then shrunk each block of the checkerboard slightly. You may design some other kind of environment.

Fancier Bubble: You should render your bubble so that its rotation is evident. We did this by overlaying the drawing of a glutSolidSphere with a glutSolidIcosahedron, but you are welcome to investigate alternatives (e.g., different coloring or texture mapping).

Note that, even if the rendered shape is not a sphere, you may process collisions as if it were a sphere. If you change the bubble's shape, I would recommend not deviating too much from something that is spherical, since otherwise the simple spherical collision detection and response may look unrealistic.

Rolling on the Ground: When the bubble rolls along the ground, it should rotate at an angular velocity that is consistent with its linear velocity. For information on how to achieve this, see the description in Lecture 14. Note that, once the bubble leaves the ground, its angular velocity should remain constant (except for gradual slowing due to air resistance), until it lands on the ground again or collides with some other object.

Friction and Air Resistance: As with the earlier phases, the object should slow down due to friction and air resistance. When on the ground, this will also slow its angular velocity. When in the air, the angular velocity should also gradually slow down due to air resistance.

Collisions: You should process collisions between your bubble and both the spherical obstacles and rectangular blocks of the checkerboard ground. (You do not need to process collisions with your skybox.)

Obstacle Collisions: Assuming that obstacles are spheres, collision detection is the same as in Phase II. The collision response differs only in that rotation must be considered. See the remarks in Lecture 14 on how to handle this.

Rectangular Block Collisions: You must also handle collisions and collision response with the rectangular blocks. Unlike Phases I and II, where it sufficed to consider just the top and bottom surfaces of the ground, now you must detect and process collisions with all sides of the block including its edges and vertices.

Because a rectangular block consists of 6 faces, 12 edges, and 8 vertices, it is a good idea to give this some thought before designing a mess with 26 special cases. (Our rectangle collision detection code is only about 30-lines long, and makes heavy use of the symmetry of rectangles. For example, consider the vector from the center of the rectangle to the center of the bubble. This vector may lie in one of 8 different orthants, depending on the signs of its x-, y-, and z-coordinates. Up to a simple changes of signs, the processing within each of the 8 orthants is the same.)

The first operation is to compute the point on the boundary of the 3-D rectangle that is closest to the center of the bubble. Given this closest point, test whether it is closer to the bubble center than the bubble radius. If so, you have a collision. Once you know the point on the bubble where the collision takes place, then the collision response is essentially the same as described in Lecture 14.

Resources: The existing sample executable works for Phase III as well. Hitting the keys '1', '2', or '3' switches among the various phases. In addition the skybox image, the RGBpixmap code, and the file with obstacles, we also provide source code for a simple quaternion object, called, Quaternion. This contains all the quaternion functionality needed for the project.

Phase-I and -II Requirements: Part of your grade (about 20%) will be based on the successful implementation of the requirements from Phases I and II (at least those that are still relevant to Phase III). Thus, if you still have bugs or unimplemented features from your earlier implementations, please check with us. We will be happy to help you to get these elements working.

External Resources: An important learning objective with this project (all phases) is your ability to process basic 3-dimensional geometry, rendering, animation, and simple physics (involving both linear and angular motion, collision detection, and collision response). In practice, much of this would be done with the aid of geometric modelers, a game engine, and a physics engine. However, for this project, we would like you to do as much of this as possible on your own, in order to acquire an understanding of how the internal elements of these systems work.

For this reason, other than OpenGL and GLUT, you are *not* allowed to make use of high-level software tools for tasks such as rendering, geometric modeling, or physics. However, you are allowed to download and use of small technical pieces of code (e.g., code for performing basic matrix or quaternion operations). If you are not sure whether some code satisfies our conditions of "fair use", please check with me.

Also remember that as part of course academic-honesty policies, all externally derived resources (e.g., code snippets or geometric models that you download from the web) must be acknowledged. The only exceptions are image files and materials that we provide for you and which are made available on the course web page. As part of your final submission, your "Readme.txt" file must indicate what resources you downloaded, where they appear in your project, about how much code was downloaded. If you made modifications, let us know what you did.

Practice Problems for the Final Exam

The final will be on Thu, Dec 16, 10:30am–12:30pm. 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. 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. Also, do not forget to review material from before the midterm.

Problem 1. Short answer questions.

(a) For each of the following operations, indicate which OpenGL buffer is most relevant to the operation (just list one): Color buffer, depth buffer, accumulation buffer, or stencil buffer.

(i) Blending and motion blur.

(ii) Hidden surface removal.

(iii) Lighting and shading.

(iv) Masking.

(b) The *cross product* of two unit vectors $u \times v$ is of unit length. What can be said about their *dot product*, $(u \cdot v)$? (Pick one.)

(i) It must be 0.

(ii) It need not be 0, but it must be a number between -1 and $+1$.

(iii) It will be either -1 or $+1$, but we cannot determine which.

(iv) Nothing. The cross and dot product are unrelated to each other.

(c) Consider the following OpenGL sequence for drawing a shape on the x, y-coordinate plane.

```
glPushMatrix( );
  glTranslatef(3, 2, 0 );
  glRotatef( -30, 0, 0, 1 );
  drawShape( );
glPopMatrix( );
```

Which of the following best describes the effect of these transformations on the shape:

(i) Translate by $(3, 2)$, and then rotate clockwise by 30 degrees about the origin.

(ii) Translate by $(3, 2)$, and then rotate counterclockwise by 30 degrees about the origin.

(iii) Rotate clockwise by 30 degrees about the origin, and then translate by $(3, 2)$.

(iv) Rotate counterclockwise by 30 degrees about the origin, and then translate by $(3, 2)$.

(d) What is *Lambert's Cosine Law*? Explain briefly how this law is used to compute the diffuse illumination term in the Phong model:

$$\max(0, (n \cdot \ell))\, L\, C_d.$$

Recall that n is the surface normal, ℓ is the unit vector to the light source, L is the color of the light, and C_d is the diffuse color of the object.

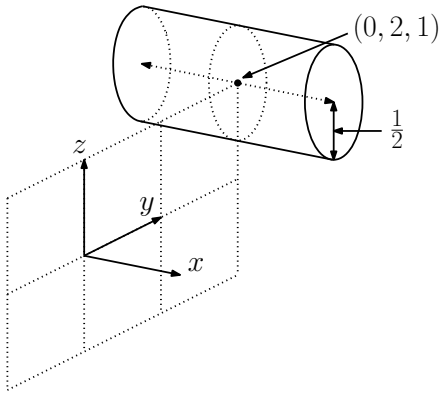
(e) Explain the difference in how smooth shading is performed in *Phong shading* and *Gouraud shading*. Which method does OpenGL use?

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

(g) 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?

(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.

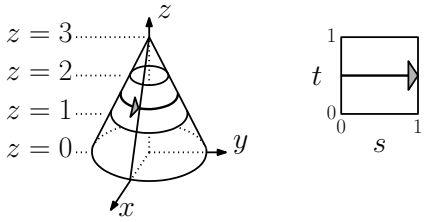
Problem 2. In this problem we derive the implicit and parametric representations of a cylinder. Consider an infinite cylinder of radius $1/2$ centered whose central axis is parallel to the x-axis, and which passes through the point $(0, 2, 1)$.



(a) Give an implicit function representation of this cylinder, by giving a function f such that $f(x, y, z) = 0$ for each point on the surface of the cylinder.

(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?

Problem 3. 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.

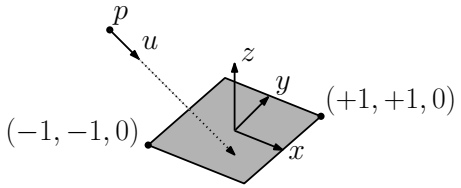


Give the *inverse wrapping function*, which maps a point (x, y, z) on the central third of the cone the corresponding point (s, t) in texture space.

Problem 4. 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.)

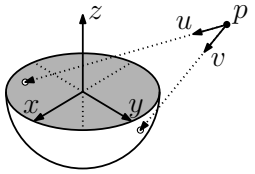
Problem 5. Fog is a relatively easy enhancement to a ray tracer. Fog is defined by three parameters, fogStart, fogEnd, and the fog RGB color F. Let C be the color returned by the ray tracing procedure (ignoring fog). Let d be the distance from the ray origin to the point of contact. If d is less than fogStart then C is used, if d is greater than 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 fogStart, fogEnd, and F.

Problem 6. Write a procedure to test whether a ray $p + t\vec{u}$, 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 7. You have been asked you to produce a ray-intersection procedure for cereal-bowl shape for the upcoming thriller "Cap'n Crunch vs. Lucky the Leprechaun: The Curse of the Magical Marshmallow".

The cereal bowl is the bottom-half of a unit sphere, which is centered at the origin. Assume that the z-axis points up.

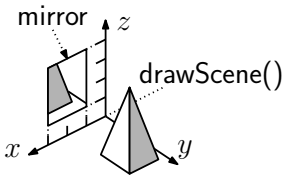


(a) Let p be a point and u be a unit vector. Given a ray $p + tu$, present a procedure (as either mathematical formulas or pseudo-code) that determines the value t of the first intersection of the ray with the bowl. If there is no intersection with the bowl, your program should detect this case. Explain how you derived your answer. (Hint: It may be useful to recall the quadratic formula, which states that the roots of $ax^2 + bx + c = 0$ are $(-b \pm \sqrt{b^2 - 4ac})/2a$.)

(b) Give a procedure which determines the normal vector at the point of intersection. The normal should be of unit length and directed to the same side of the bowl from which the ray hits.

Problem 8. In this problem, we will consider how to render a scene with a mirror in OpenGL. You have a procedure, drawScene(), which draws a given scene. You may assume that the scene resides entirely

in the positive (x, y, z)-orthant (that is, $x \geq 0$, $y \geq 0$, and $z \geq 0$ for all objects in the scene). Imagine that on the $y = 0$ plane, there is a rectangular mirror, as shown in the figure below, which ranges from $[1, 2]$ on the x-axis and $[1, 4]$ on the z-axis. You are to write an OpenGL procedure to render this scene and its reflection in the mirror through the use of the stencil buffer. The mirror is a perfect reflector, and hence no color blending is required.



(a) Give a step-by-step high-level description of how this will be done. You do *not* need to give specific OpenGL commands, but it should be clear how to translate your ideas into OpenGL operations (e.g., "save the matrix state", "disable the depth test", "draw a polygon with vertices ...", etc.). It should be clear from your answer what order the operations are to be performed, what specific transformations are to be applied, what shapes are to be drawn, etc. In this part you may *ignore lighting*.

(b) If lighting is to be applied to the objects rendered in the mirror, should the light positions be modified, and if so, how?

Final Exam

This exam is closed-book and closed-notes. You may use 2 sheets of notes (front and back). Write all 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. (40 points; 2–7 points each) Short answer questions. (Explanations are not required.)

(a) For each of the following operations, indicate which glut callback(s) would be most appropriate (there may be *more* than one). You may chose from the following: glutDisplayFunc, glutIdleFunc, glutKeyboardFunc, glutMotionFunc, glutMouseFunc, glutPassiveMotionFunc, glutReshapeFunc, glutSpecialFunc, glutTimerFunc.

(i) The graphics window has just been created.

(ii) The graphics window has just switched to "full screen mode."

(iii) Advance the physics/animation state over a small time period Δt.

(iv) Process a mouse dragging event (moving the mouse while a button depressed).

(v) Process the user hitting the up-arrow key ("↑").

(b) The vector cross product is said to be *skew symmetric*. What does this mean?

(c) Suppose you have a function drawShape(), which draws a picture of the figure below left. Give the OpenGL commands to produce a drawing of the figure on the right, using only this function and a *single* call to glRotatef and a *single* call to glScalef.

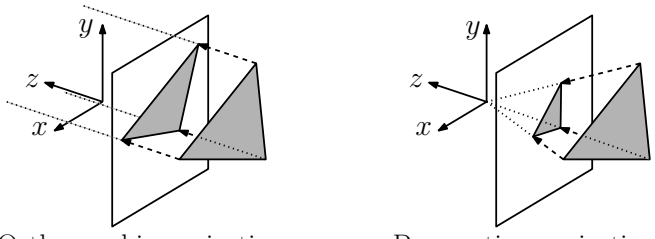
(d) Repeat (c), but this time *reverse* the order in which you call glRotatef and glScalef.

(e) A 3-dimensional scene is rendered using an *orthographic projection* (see the figure below). Answer true or false for each of the following statements.

(i) Parallel line segments in the scene are *always* parallel in the image.

(ii) Parallel line segments in the scene are *never* parallel in the image.

(iii) If two line segments form an acute angle in the scene, they form an acute angle in the image.



Orthographic projection Perspective projection

(f) Repeat (e), but now assume that you have a *perspective projection*.

(g) Of the shadowing methods we studied (shadow painting, light maps, shadow volumes):

(i) Which method(s) are capable of producing soft shadows?

(ii) Which method(s) are dynamic (shadows changing with each redraw cycle)?

(iii) Which method(s) make use of the OpenGL stencil buffer?

(h) Match each of the definitions below to one of the following common physics terms: mass, center of mass, moment of inertia, torque, contact force, field force, environmental force, kinematics, kinetics, entropy. (Some terms may be used more than once.)

(i) *Bouyancy* in water is an example of this.

(ii) *Magnetism* is an example of this.

(iii) The degree to which a body resists changes in *linear velocity*.

(iv) The degree to which a body resists changes in *angular velocity*.

(v) The theory that explains how *force* affects *motion*.

(vi) The amount of "stuff" that a body has.

(vii) The point about which (unconstrained) rotation occurs.

Problem 2. (10 points) The purpose of the question is to explain the meanings of the various elements of the Phong lighting model. The formula is given below. (I have intentionally eliminated some terms and simplified others.) Let L be the light color, C the object color, d the distance from the light source to the surface point, n the normal vector, ℓ the light vector, and h the halfway vector.

$$L \cdot C \;+\; \frac{1}{a + bd + cd^2}\Big(\max(0, n \cdot \ell) L \cdot C \;+\; \max(0, n \cdot h)^\alpha L\Big)$$

(a) Which term of the formula controls *ambient component* of the illumination?

(b–d) Repeat (a) for: (b) *diffuse* component, (c) *specular* component, and (d) *attenuation*.

(e) In the term "$\max(0, n \cdot h)^\alpha L$," the object color C does *not* appear. Why not?

(f) What visual effect is produced by *increasing* the value of α?

(g) How is the *halfway vector* defined?

(h) Among the three components (ambient, diffuse, and specular), which depend on the location of the *viewer*?

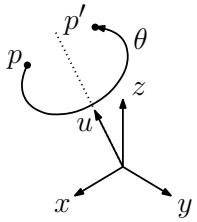
Problem 3. (10 points) We wish to perform a rotation of θ degrees about a unit vector $u = (u_x, u_y, u_z)$ using a quaternion representation. (See the figure below left.) We will apply this rotation to a point $p = (p_x, p_y, p_z)$.

(a) Express this rotation as a *unit quaternion* $\mathbf{q}$. (You may express $\mathbf{q}$ as a 4-element vector or in the form (s, v), where s is a scalar and v is a vector.)

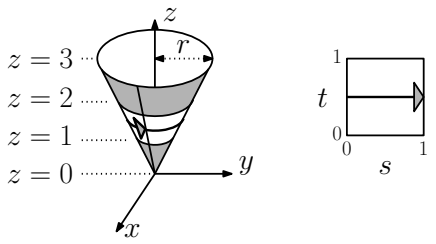
(b) Express p as a *pure quaternion*, denoted $\mathbf{p}$.

(c) What is $\mathbf{q}^{-1}$?

(d) Let p' be the image of p under this rotation. Express p' in terms of quaternion operations on $\mathbf{q}$ and $\mathbf{p}$.



Problem 3



Problem 4

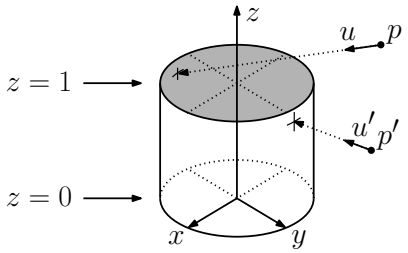
Problem 4. (20 points) Consider the cone shown in the figure above right. Its axis is the z-axis, its apex is at the origin, and its base has radius r and is located at $z = 3$. We wish to wrap a rectangular texture around the central third of the cone. (Thus the bottom edge of the texture coincides with $z = 1$ and the top edge coincides with $z = 2$.) As s varies from 0 to 1, the texture should make one full revolution around the cone, starting from directly above the x-axis.

(a) Give a parametric representation of the cone. That is, given two real parameters u and v, show that any point $p = (x, y, z)$ that lies on the cone can be expressed as $x(u, v)$, $y(u, v)$ and $z(u, v)$, for some choice of u and v.

(b) Given your parameterization from (a), what are the ranges of values for u and v in order to generate the above cone?

(c) 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. (20 points) Consider a hollow cylinder, whose axis is aligned with the z-axis, has a radius of 1, and extends from $z = 0$ to $z = 1$.



(a) Give an implicit representation for the *infinite cylinder* (no constraint on z). That is, give a function f such that $f(x, y, z) = 0$ for each point (x, y, z) on the cylinder.

(b) Given a point p and unit vector u, consider the ray $p + tu$. Present a pseudo-code procedure that determines the value t of the *first intersection* of the ray with the hollow cylinder. If there is no intersection, indicate this. Note that the ray may hit the cylinder either from the *inside* (as in ray (p, u) in the figure) or from the *outside* (as in ray (p', u')).

Explain how you derived your answer. (Hint: It may be useful to recall the quadratic formula, which states that the roots of $ax^2 + bx + c = 0$ are $(-b \pm \sqrt{b^2 - 4ac})/2a$.)

(c) Give a procedure which determines the normal vector at the point of intersection. The normal should be of unit length and directed to the same side of the cylinder from which the ray hits.