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:

\(\text{WinDir}\): This is your Windows system directory (e.g., C:\WINDOWS).
\(\text{VCpp}\): 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).

\[\text{glut32.dll} \rightarrow \text{WinDir}\backslash\text{system32} \text{(or wherever opengl32.dll is)}\]
\[\text{glut32.lib} \rightarrow \text{VCpp}\backslash\text{lib}\]
\[\text{glut.h} \rightarrow \text{VCpp}\backslash\text{include}\backslash\text{GL}\]

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.
Programming Assignment 1: Single-User Pong

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

Overview: The goal of this assignment is to learn the basics of OpenGL and GLUT and (hopefully) generate a simple and fun application. The game is a single-user variant of the famous, and now embarrassingly primitive, Atari arcade game “Pong”. There is a ball, which bounces around within the window, and a paddle, which the user can move. Your program must implement the following elements.

Rotating Ball: The ball should be rendered as a colored object of your choice (square, circle, triangle, etc.) which rotates as it moves. When the ball hits a wall or the paddle it should respond in a “believable” manner. This response will involve both a change of velocity and a change in rotation. (It goes without saying that the ball must be drawn or colored in a manner that makes clear that it is rotating.) Whenever it hits another object obliquely, it will spin in an appropriate manner—the faster and more oblique the collision, the greater the spin.

Paddle: The paddle should be movable, under the user’s control. For example, the paddle can move up or down when the user hits ‘↑’ or ‘↓’, respectively. You may render it however you like, e.g., as a rectangle. (Putting the paddle under mouse control is also allowable.) It should not be possible for the user to move the paddle outside the boundary of the playing area.

Collision response: The response of your ball to collisions should be reasonably realistic. (It is not necessary to simulate physics, but the behavior of the ball on collision should not be “unexpected” or “unbelievable”).

Smooth animation: Use glutIdleFunc() to continuously update the state of your game.

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 paddle, nor too slow to be boring. (An video showing the original Pong can be found by searching for “Atari pong” on Youtube.com.)

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

Move the paddle: For example, using the arrow keys.

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.NET”), how to compile your program (very important), how to run and execute your program, any special features you have implemented (very important), and any bugs or limitations that you are aware of.

**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).

**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.

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

**Paddle response:** In the original Pong game, the velocity of the paddle affects the vertical speed of the ball. Hitting the ball when the paddle is moving down, for example, added additional vertical velocity to the ball.

**Window resizing:** It should be possible for the TA to resize the 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.)

**Scoring:** As in the original Atari game, print a score on the screen. See the glut documentation on how to generate text.

**Animation adaptation:** Your program should adjust automatically when run on machines with different refresh rates, by keeping track of the actual elapsed time of execution. (See http://www.firstobject.com/getmillicount-milliseconds-portable-c++.htm for a code fragment that will provide you with wall-clock time. (Warning: Although this claims to return time in milliseconds, your system’s clock resolution may be lower. It is a good idea to average times over many refresh cycles to get a better estimate. If the window is resized, the estimate will likely need to be updated.)

**Additional objects:** Add another ball or additional obstacles. The additional obstacles might also be destroyed when the ball hits them (somewhat like the old arcade game “Breakout”).

**Start-up screen:** Show a start-up screen when the game starts, and an ending screen showing the final score when it ends.
Programming Assignment 2: Katamari Lite (Part I)

Handed out Thu, Oct 8. The program must be submitted to the grader by Tue, Oct 27 (any time up to midnight). Submission instructions will be forthcoming. See the syllabus for the late policy.

Overview. This project has been inspired by a well known video game, called Katamari Damacy. In this game the player pushes around a rolling ball on the ground, called the Katamari. The ball is sticky, and it can pick up objects that are sufficiently small. Objects that are too large behave as obstacles, from which the Katamari simply bounces off. As the ball picks up more objects, it gets larger, and hence it can pick up larger objects. Implementing a simple version of this game will allow us to explore many aspects of computer graphics (modeling, lighting, texturing), game-programming (user-interaction and camera control), basic 3-dimensional geometry (affine geometry, rotations, and quaterions), and basic physics (rigid-body physics). For this first part of the project, we will not worry about the stickiness, and instead will focus just on the rolling behavior of the ball and collision detection and response.

Your program will allow the user to push the Katamari around a simple 3-dimensional environment, which consists of a flat plane containing a number of simple geometric objects. Each object sits on the ground. It is important that the rotation of the Katamari is visually evident, and hence it should not be rendered as a monochrome sphere. In our program the Katamari is rendered as a rotating cube, but it could be rendered in many other ways (e.g., a texture-mapped sphere). The objects in our program are just colored cubes, but in the next phase we may add additional object shapes. (In our program we have rendered each object the union of a cube and sphere to make them look a little nicer. But they behave as a cubes for the sake of collision detection).

In this 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 have achieved the main learning objectives we have in mind. (If you are in doubt, please check with us.) The following are the main elements, which your program must implement.

User Input: The user controls the Katamari using keyboard input. You can do this using the arrow keys. (But, if you prefer, you may use keyboard key input instead.) Each press of a key induces an instantaneous impulse to the rolling Katamari. Thus, the more frequently a key is pressed, the faster the ball 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.

Your program should simulate a frictional force, so that once the keys are no longer pressed the Katamari will slow down and eventually stop. For extra credit, you may also add more realistic physics (e.g., taking into account that heavy and faster moving balls have greater linear and angular momentum and are harder to turn).

Note that hitting a key does not change the velocity to the given value, it merely induces an impulse in this direction. For example, if the user hits ‘↑’ quickly followed by ‘←’, the motion will be a diagonal combination of these two.

Camera motion: Intuitively, the camera is located behind the Katamari. Your program should allow the camera to be rotated about the Katamari. For example, in our program dragging the mouse (with a left button pressed) to the left or right rotates the camera to the left or right, respectively, about the center of the Katamari. (If you prefer, you could use two keyboard keys for this purpose instead.) For extra credit, you can allow the camera to be moved in other ways, e.g., higher/lower or closer/farther from the Katamari. In the later phases of the project, as the Katamari grows, the camera position periodically adjusts itself automatically, so the Katamari takes up about the same amount of screen space.

Objects: As mentioned above, the world is occupied with a collection of geometric objects. These objects will be provided in a file, which specifies the locations and properties of the various objects. The format of this file will be given later, but each line of the file will contains the following information: the object type (currently cubes only, its position, its size, its color, and so on. For extra credit, you may augment the collection of allowable objects or provide object model files of greater complexity.
**Collision detection and response:** When the Katamari hits an object, it should respond in a physically believable matter. As in Project 1, we do not require that you simulate physics exactly, but the collision response should seem natural to the user. For example, if the Katamari hits a planar surface, the component of its velocity that is orthogonal to the point of contact should be reversed (and perhaps multiplied by a scale factor $\leq 1$ to model the fact that the collision is not perfectly elastic). The component of the velocity parallel to the surface should not be affected (or might decrease slightly, to simulate a frictional force at the instant of contact). Collision detection does not need to be exact. For example, the user will usually not notice a slight penetration with objects if the collision response quickly resolves the situation.

**Rotational Motion and Quaternions:** As the Katamari moves, it should rotate as if it is rolling along the ground. This rotation does not need to be physically accurate, but the impression should be there, nonetheless. (Those of you who have played the game will know that the rotational motion is not physically realistic, and the Katamari will actually penetrate the ground at times.) One way to achieve this motion is to determine the direction along which the Katamari is currently moving, and then make the rotation consistent with this direction. As the direction of motion changes, the axis of rotation will change as well. The rotation speed should be consistent with the Katamari’s linear motion speed. (For example, it should not look like a tire spinning on ice.)

An important element of this project is handling motion involving rotation. It will not be sufficient simply to invoke OpenGL’s Modelview transformations, since (for the next phase of the project) your program will need to allow the Katamari pick up objects, and this will require your program to perform collision detection involving a rotating non-spherical object. At a minimum, your program will need to perform basic matrix operations (creating, multiplying, and inverting matrices for rotation and translation). Unfortunately, rotation matrices alone provide a poor representation for the sort of unpredictable rotations that arise when simulating physics. For this reason, you are required to use quaternions to represent the Katamari’s rotation. We will discuss quaternions in class.

**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. Normally, a skybox consists of the six sides of a large cube. For this project, you may only need images for the four walls, since you will render a floor on which to place the Katamari and the sky may not even be visible (depending on how you design your camera control). 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.

**External Resources** An important learning objective with this project 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 either Jun-Cheng or 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 materials that you download from the course web page and images. 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, and (if you modified the code) the degree to which you modified it.
Programming Assignment 3: Katamari Lite (Part II)

Handed out Tue, Nov 17. The program must be submitted to the grader by Tue, Dec 1 (any time up to midnight). Submission instructions are the same as with Project 2. See the syllabus for the late policy.

Overview. In this project we continue the development of the Katamari Lite. The new elements are described below.

Attaching Objects: When the Katamari has a collision that satisfies some condition (e.g., the object is sufficiently small relative to the current Katamari) it is attached to the Katamari. This should be done in a manner that is “believable”. For example, the object should be attached to the location (roughly) at its contact point with the Katamari, and the object should not suddenly appear to change in its orientation. Note that not only is the Katamari “sticky”, but its attachments are sticky as well, in the sense that new collisions may result in new attachments being attached to existing attachments.

Hint: In order to do this, we created a list of attachments, which is associated with our Katamari object. Each entry of the list contains a description of the object that is attached (e.g., its type, size, color). Each entry of this list also indicates the position of the attachment relative to the base Katamari, that is, the Katamari without any translation or rotation. By relative position, I mean the translation vector and rotation quaternion. The advantage of this representation is that, in order to render each attachment, you can apply to each attachment the same transformation that you apply to render your Katamari.

How do you determine the relative position of the object relative to the base Katamari? Suppose that the Katamari’s current rotation quaternion is $q$ at the time the collision is detected. The orientation of the attachment relative to the base quaternion would be given by $q^{-1}$. Similarly, the translation vector from the center of the Katamari to the attachment’s center at the moment of contact will need to be transformed by $q^{-1}$.

Once you know the relative position of each attachment to the base Katamari, you can reuse the same OpenGL code for drawing your Katamari from Project 2, with the following addition. For each attachment, you will do a push-pop sequence to generate the appropriate local transformation to position and orient the attachment, and then render it. Now, all that is needed to render the whole assembly is to put the entire drawing procedure in a push-pop pair that translates and rotates the Katamari (and all its attachments) into the desired global position and orientation (just as you did in Project 2). This is a nice illustration of the use of local coordinate systems and local transformations in OpenGL.

Advanced Collision Detection: With objects attached to your Katamari, collision detection and response becomes more complicated than in Project 2. In particular, whenever an attachment $A$ collides with another object $O$, this must be treated in the same manner as a collision between the Katamari and $O$. If $O$ is sufficiently small, then $O$ will become attached at this point.

Hint: To keep collision detection simple, rather than computing full polyhedron-polyhedron collision detection, we approximated all objects with spheres, and performed simple sphere-sphere collision detection. You may implement more accurate collision detection for extra credit.

Attachments and Rolling: The attachments should affect the visual depiction of your Katamari’s rolling behavior with the ground. If the Katamari has an attachment $A$, and it rolls over the top of $A$, the center of the Katamari will be lifted above the ground level (as opposed to $A$ sinking below the ground).

Hint: It is not required to simulate physics. In theory, attachments would alter the center of mass of the Katamari and this would further alter its rotational behavior. In our implementation, we kept things simple by simplying reusing the rotation code from Project 2. We simply computed the lowest $z$-coordinate of all the attachments, and lifted the Katamari by an appropriate amount before rendering it.

Camera motion: In the current implementation, the camera is linked directly to the Katamari center. Due to the presence of attachments, the Katamari center tends to bounce up and down. If the rotation speed is high, this
produces an undesirable degree of bounciness in the camera. To make the camera movement smoother, maintain the camera at a fixed distance above the ground. (This distance may generally be a function of the Katamari’s radius.)

**Mass-Spring System:** One new modeling element we will add is some sort of flexible cloth-like object using a mass-spring system. You may implement this in any way you like, provided that it is possible for your Katamari to interact with this object in order to elicit the desired behavior.

In our implementation, we created a curtain object, which hangs in the middle of the scene. When the Katamari rolls through the curtain it deforms as the Katamari rolls through it. Because the Katamari collision detection is rather crudely implemented (using an approximating sphere) it is understandable that the corners of the rendered Katamari may appear to penetrate the cloth.

We will discuss mass-spring systems in class, and there are a number of tutorials on the web, many of which contain sample source code. For example, there is a nice tutorial on cloth simulation, together with source code, which can be found at [http://cg.alexandra.dk](http://cg.alexandra.dk) (Search for “cloth simulation”). As always, if you use any external sources, be sure to document them in your ReadMe file and explain any enhancements or modifications you have made.

**Hint:** Accurate simulation of mass-spring systems can consume a lot of CPU time. You should expect to apply some additional optimizations (e.g., carefully balancing the size of your system, selecting the appropriate number of iterations, and applying tricks to speed-up collision detection with the cloth). When dealing with mass-spring systems, self-intersection is a major issue. This is when the cloth folds back to intersect itself. To keep things simple, we ignored self-intersection and we placed our curtain far enough away from the objects of the scene to avoid cloth-object intersection. Thus, we only needed to test for Katamari-cloth intersections.
Programming Project 4: Simple Ray Tracer

Handed out Tue, Dec 1. The program must be submitted to the grader by Mon, Dec 14 (any time up to midnight). See the syllabus for the late policy.

Overview: The goal of this project is to implement a very simple ray tracer. As usual, you are allowed some flexibility in designing your project input and output, but your project must support at least the following basic elements for partial credit: sphere objects, solid colors, and the basic elements of the Phong model. For full credit, you will also implement planes, checkerboard textures, and reflection and refraction. For extra credit you may add other features, including other object types (e.g., cones and cylinders), other types of textures, texture mapping, and antialiasing.

This program does not involve OpenGL or Glut. It inputs data from a text file, and generates an image file as output. (It is recommended, for debugging purposes, that you add some diagnostic output to the standard output.) It can be implemented in the programming language of your choice (e.g., C, C++, or Java), as long as the TA can compile and run your program. Note that the program’s running time can be high, so C or C++ are recommended.

Your program will read a viewing situation and 3-dimensional scene description from an input file and will output an image file as a .bmp file. We will provide software for writing an 2-dimensional RGB array to a .bmp file (see description below). You can view the result using any standard image viewing software, such as xv or gimp (on Unix) or Paint, Windows Picture Viewer (on Windows PCs). The input and output files and image width and height are specified on the command line. For example:

```
project4 infile.txt outfile.bmp 400 300
```

This will read input from file infile.txt and produce the output file outfile.bmp whose width and height are 400 and 300, respectively. Providing the ability to set the image size on the command line is useful, since when debugging you may want to start with small test images, and increase the image size (and running time) once you are more confident of your program’s correctness.

Input Format: The input to your program consists of five sections, which are described below. Each geometric object in the scene is described not only by its geometric properties, but its surface properties as well, consisting of its color, pattern, or texture (called is pigment) and surface finish, which describes its light reflection properties.

The input format is designed to be friendly to the program (not the user). All points and vectors are given in \((x; y; z)\) coordinates with respect to the world coordinate frame. All colors are given as a vector of RGB values in floating point (typically in the range 0 to 1).

The input format was designed for easy programming, not user friendliness. We will provide a test program and file ReadMe.txt, which has an annotated example of an input file.

Viewing situation: The input file begins with four lines that contain the camera and perspective information (similar to gluLookAt and gluPerspective). These are the coordinates of the eye, the center point at which the camera is pointing, the up vector, the \(y\) field-of-view (in degrees). There is no near or far clipping plane. You may assume that the viewing window is located one unit from the eye and centered along the viewing direction. The aspect ratio of the window is determined by the image width and height (from the command line arguments). An example is shown below. (Comments following the ‘#’ are not part of the file).

```
# THESE COMMENTS ARE NOT PART OF THE INPUT
1 -10 5  # eye at (1, -10, 5)
1 10 -3  # looking at (1, 10, -3)
0 0 1    # z-axis points up
20       # y-field of view is 20 degrees
```

From the viewing situation you will create a camera frame (an origin and three unit vectors), which will be used for generating rays. (The procedure was presented in Chapter 4 of the lecture notes.)
Light sources: The next line contains the number of light sources \( n_L \). Each of the successive \( n_L \) lines contains three triples of floating point numbers: the \((x, y, z)\) coordinates of the light source, the \((r, g, b)\) components of its intensity, and the \((a, b, c)\) values of the distance attenuation formula: \(1/(a + bd + cd^2)\), where \( d \) is the distance to the light source. Light sources are numbered from 0 to \( n_L - 1 \). The 0th light source in the list is always the ambient light. Its location and attenuation factors are given, but should be ignored by the program.

\[
\begin{array}{cccccccc}
2 & \text{# 2 lights} \\
0 & 0 & 0 & 0.5 & 0.5 & 0.5 & 0 & 0 & 0 & \text{# white ambient light} \\
0 & 10 & 50 & 1.5 & 0 & 0 & 0 & 0.1 & 0 & \text{# red light at (0,10,50)}
\end{array}
\]

(It may be helpful to review the Phong lighting equation presented in Chapter 5 of the lecture notes.)

Pigments: The next line contains the number of pigments \( n_P \). This is followed by a list of \( n_P \) pigment specifications, numbered from 0 to \( n_P - 1 \). Each pigment can be thought of as a function that maps the \((x, y, z)\) coordinates of a point to an RGB value. The following pigments are to be supported:

Solid: The word “solid” followed by the associated RGB value.

Checker: This defines a 3-dimensional checkerboard. It is specified by the word “checker” followed by two RGB triples \( C_0 \) and \( C_1 \), followed by a scalar \( s \), indicating the size of each square of the checkerboard. The result is a axis-aligned 3-dimensional checkerboard, whose side length is \( s \). The color of a point with coordinates \((x, y, z)\) is \( C_i \), where

\[
i = \left(\left\lfloor \frac{x}{s} \right\rfloor + \left\lfloor \frac{y}{s} \right\rfloor + \left\lfloor \frac{z}{s} \right\rfloor \right) \mod 2
\]

(Note that some care is needed when computing the floor function. You should first invoke the function \( \text{oof}(x) \) and then cast to an integer. If the result is negative, then negate it before applying the ‘\%’ operator.)

\[
\begin{array}{cccccccc}
2 & \text{# 2 pigments} \\
solid & 0.0 & 0.4 & 0.0 & \text{# Pigment 0: solid dark green} \\
checker & 1 & 0 & 0 & 0 & 0 & 1 & 2.0 & \text{# Pigment 1: red-blue checker, size 2}
\end{array}
\]

There is a background color, which we define to be gray (RGB = (0.5, 0.5, 0.5)). If no object is hit, then the background color is used and no shading is applied.

Surface finishes: The next line contains the number of surface finishes \( n_F \), numbered from 0 to \( n_F - 1 \). Each successive line contains seven surface finish parameters, \( \langle \rho_a, \rho_d, \rho_s, \alpha, \rho_r, \rho_t, \eta_i \rangle \). These are the ambient coefficient \( \rho_a \), the diffuse coefficient \( \rho_d \), the specular coefficient \( \rho_s \), the shininess \( \alpha \), the reflectivity coefficient \( \rho_r \), and transmission coefficient \( \rho_t \), and finally the index of refraction of the object’s interior \( \eta_i \). You may assume that the exterior of each object is air (that is, \( \eta_i = 1 \)). If your program does not support reflection or refraction, you may ignore the values of \( \rho_r, \rho_t \) and \( \eta_i \), but you still have to input them.

\[
\begin{array}{cccccc}
2 & \text{# 2 surface finishes} \\
0.3 & 0.1 & 1.0 & 500 & 0.9 & 0.0 & 0 & \text{# 0: highly specular and reflective} \\
0.0 & 0.7 & 0.0 & 50 & 0.0 & 0.5 & 1.5 & \text{# 1: diffuse, partially transparent}
\end{array}
\]

Objects: The next line contains the number of objects. Each line starts with two integers, which indicate the pigment used for this object (from 0 to \( n_P - 1 \)) and the surface finish for this object (from 0 to \( n_F - 1 \)). This is followed by a word giving the object type and description:

Sphere: The word “sphere” followed by the \((x, y, z)\) center coordinates radius \( r \).

Plane: The word “plane” followed by a tuple of floating-point values \( \langle a, b, c, d \rangle \), which represent the plane equation \( ax + by + cz + d = 0 \). (For refractive planes, the “interior” of the plane is defined to be the side where \( ax + by + cz + d < 0 \).) For example, the plane defined by the inequality \( z \leq -2 \) would be given by the tuple \( \langle 0, 0, 1, 2 \rangle \).
You may assume that there are at most 20 light sources, 50 pigments, 50 surface finishes, and 200 objects in a
scene.

**Program Requirements:** There will be basic elements, which must be implemented for partial credit, and optional
elements that can be added to this. Your program must implement the following basic elements for 75% credit.

**Shapes:** Implement sphere objects. (Input but ignore other object types.)

**Pigments:** Solid color.

**Surface finishes:** Support all basic elements of the Phong model: ambient, diffuse, specular, and attenuation. (Input
but ignore the parameters for reflection and refraction.)

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

**Shadows:** (5%) Implement shadow casting by shooting a ray to each of the light sources and evaluating their contribu-
tion only if the ray hits this source before any other object.

**Checkerboards:** (5%) Implement 3-dimensional checkerboard pigments.

**Shapes:** Include plane objects (5%).

**Reflection:** (5%) Implement reflection. (After 5 levels of recursion depth, return the default color.)

**Refraction:** (5%) Implement refraction. (You may assume that transparent object cast shadows on other objects.
After 5 levels of recursion depth, return the default color.)

**Ideas for Extra Credit:**

**More object types:** You can implement cylinders or cones. Triangles and circular disks are also pretty easy to imple-
ment. A nice shape to try is a torus, but you will probably need some help from the Web or other sources, since it
involves solving a polynomial equation of degree 4.

**More Textures:** The checkerboard texture is about the simplest one to implement. Another nice one is a gradient is
a texture that smoothly varies from one color value to another, and can be computed using a simple dot product.

**Fog:** Simple fog can be implemented as in OpenGL by mixing some fog color with the ray color, according to the
length of the ray.

Some processing involves the computation of matrix inverses. I can make source code available for a procedure
that will invert a matrix.

**Debugging Tips:** While you are debugging your program, it is a good idea to start with very small images (e.g. 40
by 30). Since you have to do all the lighting computations, it is notoriously difficult to locate bugs in your program.
For this reason, it is a very good idea to design your program to run in a special test mode. In this mode the program
shoots pixels only by request. It reads the column and row index of the pixel from the input file, and then prints a
detailed trace of the pixel. This would include the coordinates of the ray, the object it hits, where it hits the object, the
base pigment, the normal vector, the light rays, the reflection and refraction rays, etc.

In our demo program, this is activated by adding the option -test on the command line. Many image viewing
programs allow you to query the value of a given pixel color (gimp and IrfanView for example). A good way to locate
errors is to zoom in on a problem area, query a pixel’s exact RGB color in your output image, and then run both your
program and our demo program in test mode.

When shooting rays off of an object’s surface (for shadow computation, reflection and refraction) beware of ray
intersections occurring very close to the surface, since this typically is the ray intersecting the same surface (due to
small floating point errors). This can be avoided by discarding any intersection with a very small $t$ value.
**BMP Output:** To simplify the process of producing an output file, the RGBpixmap program provides a procedure for storing pixels in an image object and outputting them to a file. See the files RGBpixmap.h, RGBpixmap.cpp, and the associated ReadMe file for more information.

The include file defines two objects, the first is RGBpixel, which stores an RGB color stored as three unsigned char's. The second is RGBpixmap, which stores a pixel array, pixel, where each pixel is of type RGBpixel. The constructor is given the number of rows and columns in the pixel map. There is a method setPixel(int col, int row, RGBpixel C), which sets the pixel color in a particular row and column in the pixel map to the color C. It also has a method writeBMPFile(const string& fileName), which outputs the pixel map as a .bmp file with the given name. Sample usage is shown below.

```cpp
#include "RGBpixmapV2.h"
...
string bmpFileName = "whatever.bmp"; // image file name
RGBpixmap* thePixmap = new RGBpixmap(nRows, nCols); // allocate pixel map
...
for (int row = 0; row < nRows; row++) { // generate all the rays
    for (int col = 0; col < nCols; col++) { // shoot ray for row and col and get pixel color...
        RGBpixel pixColor = cast final pixel color to unsigned char;
        thePixmap->setPixel(col, row, pixColor); // store in pixmap
    }
}
thePixmap->writeBMPFile(bmpFileName); // output .bmp file
```
Homework 1: OpenGL and Affine Geometry

Handed out Tue, Sep 15. Due at the start of class Thu, Sep 24. Late homeworks will not be accepted, so turn in whatever you have finished by then.

Problem 1. In addition to triangle strips, OpenGL supports a structure called a triangle fan, which is generated using the command `glBegin(GL_TRIANGLE_FAN)`. The first three vertices define the first triangle of the fan. Subsequently, each additional vertex is joined to the previous vertex and the first vertex. The resulting structure resembles a fan shape, where all the triangles share the first vertex. For example, to generate the mesh shown in Fig. 1(a), the following sequence could be used:

```c
    glBegin(GL_TRIANGLE_FAN);
    glVertex2fv(a); glVertex2fv(b); // followed by: c, d, e, f, ..., j
    glEnd();
```

As with triangle strips, the orientation of all the triangles is set to match the orientation of the first triangle in the list. Thus, in the above example, all the triangles are oriented counterclockwise.

(a) Show how to draw the shape shown in Fig. 1(b), using a single triangle fan. The orientation should be counterclockwise, and there should be no overlapping triangles or zero-area triangles.

(b) Given the triangulation shown in Fig 1(c), show how to generate this same set of triangles using a combination of triangle strips and triangle fans. There should be at most three OpenGL objects. (E.g., three triangle strips; two triangle strips and one triangle fan; one triangle strip and two triangle fans, etc.) The orientations should all be counterclockwise, and there should be no overlapping triangles or zero-area triangles.

![Figure 1: Triangle fans.](image)

Problem 2. An important operation in interactive graphics systems is called picking. This where the user clicks on a location in the graphics window, and the program is to determine which graphics object lies at this position. Suppose that the user has a graphics drawing area that extends horizontally (in world coordinates) from $x_{\text{min}}$ to $x_{\text{max}}$ and vertically from $y_{\text{min}}$ to $y_{\text{max}}$. (E.g., as might be given by `gluOrtho2d(xmin, xmax, ymin, ymax`). The viewport is of width $w$ and height $h$ and covers the entire graphics window. (See Fig 2.) Let $(x_m, y_m)$ be the mouse coordinates as generated by GLUT to the `glutMouseFunc()` callback function.

(a) As a function of these parameters, give the drawing coordinates $(x_w, y_w)$ of the point corresponding to the mouse location. Note: Recall that GLUT and OpenGL do not generally use the same convention as to the placement of the origin.

(b) Give a $3 \times 3$ transformation matrix that maps the homogeneous mouse column vector $[x_m, y_m, 1]^T$ onto the corresponding homogeneous world column vector $[x_w, y_w, 1]^T$. 


Problem 3. You are given two procedures, `drawEarth()`, which draws a picture of the earth centered at the origin and a procedure `drawSun()`, which does the same for the sun. Your task is to write a procedure, 

```plaintext
drawSunAndEarth(int day, int hour, int minute);
```

which draws a picture of both the sun and the earth, so that as the day, hour, and minute progress through an entire year, the earth both rotates on its axis and revolves around the sun. The parameter `day` indicates the day of the year (in the range 0 through 364), `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).

Assume the following (astronomically inaccurate) structure. The sun is placed at the origin, and does not rotate. The earth is at distance $R$ from the sun and it rotates about the sun in a circular orbit on the $(x, y)$-plane. For simplicity, assume that the earth’s axis is parallel to the $z$-axis (thus, we are ignoring the tilt of the axis). At the start of the year $(day, hour, minute) = (0, 0, 0)$, the earth should be placed at distance $R$ from the sun along the positive $x$-axis. During each 24-hour period the earth undergoes one full counterclockwise rotation about its axis, and during 365 days the earth makes exactly one full counterclockwise rotation about the sun.

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 1. The art technique, called trompe l’oeil (French for “trick the eye”) involves drawing a scene on a 2-dimensional surface so that (if viewed from the proper direction) creates the illusion that the viewer is seeing a 3-dimensional scene. For example Figure 1 shows a piece of pavement art (drawn on the sidewalk) which creates the illusion there is a giant 3-dimensional bottle of coke. The trick only works if the scene is viewed from the direction shown. Any other angle the coke bottle will appear to be terribly distorted.

Consider a point \((x, y)\) on the image plane. (The \(z\)-coordinate does not need to be given, since it must be \(-d\).)

(a) As a function of \((x, y)\), determine the coordinates of the point \((x’, y’, z’)\) on the pavement that projects to the point \((x, y)\) on the image plane. Show how you derived your answer.
(b) Give a $4 \times 4$ matrix that (after perspective normalization) transforms the point with homogeneous coordinates $[x, y, -d, 1]^T$ to $[x', y', z', 1]^T$.

(c) For what points $(x, y)$ on the image plane is this even possible, in the sense that the point is the projection of some pavement point $(x', y', z')$ that lies in front of the image plane?

(Once you know the inverse projection mapping, you can map a few key points from your desired 3-D scene onto the pavement, and then draw in the details.)

**Problem 2.** In a parallel universe, the OpenGL they have has an alternative way of specifying the camera’s position. The new command is `glutLookAt2(ex, ey, ez, a1, a2, a3)`. All arguments are of type GLdouble. As with the standard version of `gluLookAt`, the first three arguments, $(e_x, e_y, e_z)$, indicate the eye location relative to the world coordinates. The next three arguments indicate the angle at which the camera is pointed. By default (if all angles are zero) the camera is looking along the $x$-axis, with the $z$-axis pointing up and the $y$-axis to the camera left. The first angle, $a_1$, indicates the number of degrees of CCW rotation about the $z$-axis (which is a variant of the notion of azimuth, or compass direction), the second angle $a_2$, indicates the number of degrees of elevation above the $x, y$-plane, and the angle $a_3$ is the number of degrees of twist, which is a CCW rotation about the viewing direction.

Your problem is to determine the affine transformation that transforms world coordinates to camera (or view) coordinates. You may simply express your transformation directly as a $4 \times 4$ homogeneous transformation matrix, or as the combination (e.g., products or inverses) of other matrices.

![Figure 3: Illustrating Problem 2.](image)

**Problem 3.** Johnny was not paying attention in class, and in his program he issued the command that sets the light positions in OpenGL before invoking `gluLookAt`. Johnny noticed that something was wrong, because whenever the camera moved, the light sources appeared to move with the camera. Explain why this happens (based on your knowledge of `gluLookAt`, and how OpenGL transforms coordinates using the Modelview matrix).
Practice Problems for the Midterm Exam

The midterm exam will be on **Thu, Nov 5** in class. The exam will be closed-books, closed-notes, but you will be allowed one sheet of notes, front and back to use for the exam. The following problems have been assembled from old exams and homeworks. They do not necessarily reflect the actual difficulty of problems on the exam or the total length of the exam. Some topics that we have covered this year are new (e.g., quaternions and shadow volumes), and have not been covered in old homeworks on exams.

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

(a) In the call

```c
glutInitDisplayMode(GLUT_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) 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.

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

(d) 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)

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

(f) Two polygons \( A \) and \( B \) are drawn using the \( z \)-buffer algorithm, first \( A \) then \( B \). Polygon \( A \) is partially transparent blue (with \( \alpha = 0.5 \)) and is closer to the viewer. The polygon \( B \) is transparent red (with \( \alpha = 0.5 \)) and is farther from the viewer. You were expecting that in the region in which the two overlap, you would see a mixture of blue and red, but in fact, you only see one of the two colors (red or blue, possibly mixed with the background color). Which color is it and why do you only see this color? Would the result change if the drawing order changed (first \( B \) then \( A \)?)

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

**Problem 2.** Suppose that you have a square graphics window (height equals width). The user has just resized the window so that it now has width \( w_w \) and height \( w_h \). As a function of \( w_w \) and \( w_h \), derive the arguments for `glViewport()` so that the new viewport is the largest square that fits within the window and is centered within the window. (See the figure below. The outer rectangle is the graphics window and the shaded rectangle is the viewport.) Recall that the calling sequence is:

```c
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 3.

In this problem you may assume that \( z = 0 \), and we are using \texttt{glOrtho2d} for viewing. Suppose that you have an OpenGL procedure \texttt{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 \texttt{GL\_MODELVIEW}. You may call the procedure \texttt{drawE()}, but you may not modify its contents. On return, the OpenGL transformation stack should be unchanged.

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

(b) Give code for a procedure \texttt{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 4.

Your boss at \textit{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 \( v \) vertical stacks (along the \( z \)-axis) and \( r \) 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 \texttt{GL\_POLYGON}.

Give a procedure (in pseudocode):

\begin{verbatim}
void cylinder(float h, float r, int vs, int rs)
\end{verbatim}

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

Problem 5. Suppose that a viewer is located at the origin \((0, 0, 0)\), and is looking along the \((-z)\)-axis. On the plane \( y = -1 \), someone has put a circular pizza of radius 1 centered at the point \((0, -1, -3)\). Assume that we compute a perspective projection of the pizza onto the view plane \( z = -1 \).

The circular pizza appears to the viewer as an ellipse. (This is a fact.)
Consider a horizontal line that bisects the projected ellipse. Does the projected pizza center lie on, above, or below this line?

(b) Consider a vertical line that bisects the projected ellipse. 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 6.** Here we consider how to generate a procedural displacement map to simulate the circular ripples that might result when a rock is dropped into a pool of water. Let us assume that the water lies on the $z = 0$ plane. The ripples should have the following characteristics:

1. The ripples start from the central point $(c_x, c_y, 0)$.
2. The distance between two successive wave peaks should be $f$.
3. The distance between the highest and lowest level of the ripples is $h$.

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

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

**Problem 7.** 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 \(z = +2\).)

Problem 8. On the distant planet of Tatooine, they have a different way of specifying the perspective transformation. As with \texttt{gluPerspective}, they give the distances to the near and far clipping planes and the window’s aspect ratio, \(a = w/h\). However, rather than giving the \(y\)-field of view, they instead give the distance \(d\) to a sphere of a given radius \(r\) that is centered along the viewing direction. The \(y\)-field of view is set so that the sphere exactly fills the window’s height. (See the figure below.)

Write a procedure which, given these parameters, produces an equivalent call to \texttt{gluPerspective}. You may assume that the window is wider than tall, that is, \(a \geq 1\). Here are the function prototypes.

```c
void newPerspective(double d, double r, double aspect, double near, double far);
void gluPerspective(double fovy, double aspect, double near, double far);
```

(Hint: There are two cases, depending on whether \(a \geq 1\) (wide windows) or \(a < 1\) (tall windows). For partial credit, just do the case where \(a \geq 1\), since this is simpler.)

Problem 9. 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 \(\theta\). The spot light illuminates any point that lies within an infinite 3-dimensional cone whose apex is \(P\) and whose angular radius about \(\vec{v}\) is \(\theta\). Write a function which, given a point \(Q\) in 3-space, and \(P\), \(\vec{v}\), and \(\theta\), determines whether \(Q\) is illuminated by the spot-light.

Problem 10. Give pseudocode (or a mathematical formula) for the diagonal strip 2-d texture function shown in the figure below. The dark color is \(C_0\) and the white color is \(C_1\). The horizontal and vertical width of each strip is 1 unit (and hence the diagonal width is \(1/\sqrt{2}\)).
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. (25 points; 5–8 points each) Short answer questions.

(a) What is the principal advantage of drawing through the use of group-based methods like GL_TRIANGLE_STRIP and GL_TRIANGLE_FAN, as opposed to GL_TRIANGLES?

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

(c) 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.

(d) An object is both translating and tumbling in space. You are given its initial and nal positions, \( p_0 \) and \( p_1 \), and its initial and nal orientations, given as two quaternions, \( q_0 \) and \( q_1 \). You want to smoothly interpolate between its starting and ending states. To interpolate between \( p_0 \) and \( p_1 \) would it be better to use a Slerp or a Lerp? How about for \( q_0 \) and \( q_1 \)? Briefly explain your choices.

Problem 2. (20 points) 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 gure 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).)

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\(^\circ\) rotation clockwise.

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. (Hint 1: First, determine how far along the \( x \)-axis the face will roll in order to complete a 360\(^\circ\) rotation. Hint 2: Don’t use quaternions.)

Problem 3. (15 points) You are given a 6-sided sky box, which is 200 units on each side and is centered about the origin. (See the figure on the top of the next page.) You are to wrap a sky box 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 sky box (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 inside the box. You should specify the normal vector of the face (using glNormal3f), again directed into the box. You should also specify the texture coordinates associated with each vertex (using glTexCoord2f).
Problem 4. (20 points) You are given a light source at coordinates $\ell = (\ell_x, \ell_y, \ell_z)$ and a point $p = (p_x, p_y, p_z)$ (see the figure below, part (a)). The point $p$ casts a shadow on a point $s = (s_x, s_y, s_z)$ on the ground plane $z = 0$.

(a) Considering just the $x$- and $z$-coordinates, derive a function that determines the value of $s_x$ as a function of the coordinates of $\ell$ and $p$. (See the figure above, part (b).)

(b) Repeat (a) but for the $y$- and $z$-coordinates alone.

(c) Derive a $4 \times 4$ matrix, which (subject to perspective normalization) maps the homogeneous coordinates $[p_x, p_y, p_z, 1]^T$ to the corresponding shadow point $[s_x, s_y, s_z, 1]^T$. (To receive full credit, you must show how you derived your result.)

Problem 5. (20 points) Consider the set of line segments shown in the figure below (right). The front side of each segment is indicated by an arrow in the figure.

(a) Show the BSP tree that would result if the segments had been inserted in the order $1, 2, 3$. (Thus segment 1 will be stored in the root node of the BSP tree.) Each internal node of your BSP tree stores a line segments (or a fragment thereof), the left subtree contains the front side objects and the right subtree is for the back side.

(b) Show the spatial decomposition induced by your tree. Indicate which segments have been fragmented. (E.g., the fragments of segment 2 would be $2a, 2b, 2c, \ldots$)

(c) Suppose that the eye is located in the lower-left corner. Indicate the order in which the segments (including their fragments) would be rendered by the Painter’s algorithm.
Practice Problems for the Final Exam

The final will be on Fri, Dec 18, 1:30–3:30pm. 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. (Some material is new this semester—physical modeling, mass-spring systems, quaternions—and consequently there are not many practice problems on these topics.)

Problem 1. Short answer questions.

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

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

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

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

(e) Which of the two methods, Euler integration or Verlet integration, is more accurate (assuming reasonably small time steps)?

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

(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.

(i) The Nyquist-Shannon Sampling Theorem states that: (Select one.)

- A signal can be accurately reconstructed if the sampling rate is at least twice as high as highest frequency in the signal.
- Prefiltering with a sync signal produces the most accurate signal reconstruction from sampled data.
- Weighted Gaussian sampling is superior to uniform random sampling.
- Doubling the sampling rate decreases aliasing by $\sqrt{2}$.

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, as \( f(x, y, z) = 0 \).

(b) Present a parametric representation for the same cylinder, e.g. as \( x(u, v), y(u, v), z(u, v) \). What are the range of values for \( u \) and \( v \)?

(c) Consider the coordinates of an arbitrary point \((x, y, z)\) on this cylinder. As a function of these three coordinates, what is the normal vector to the cylinder at this point? Explain how you derived your answer. (It is not necessary to normalize your normal vector.)

**Problem 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, \( \text{fogStart} \), \( \text{fogEnd} \), and the fog RGB color \( F \). Let \( C \) be the color returned by the ray tracing procedure (ignoring fog). Let \( d \) be the distance from the ray origin to the point of contact. If \( d \) is less than \( \text{fogStart} \) then \( C \) is used, if \( d \) is greater than \( \text{fogEnd} \) then \( F \) is returned. Otherwise, an appropriate mixture of the two colors is returned. Give pseudocode for a function, which returns the fog color, given the following parameters: the ray origin \( P \), the ray contact point \( Q \), the traced color \( C \), and the other fog parameters \( \text{fogStart} \), \( \text{fogEnd} \), and \( F \).

**Problem 6.** This problem involves computing the ray intersection for a 2-dimensional axis-parallel ellipse. (This is easy to extend to 3-space, but it is simpler in 2-space.) Let \( P + t\vec{u} \) be the ray, where \( P = (P_x, P_y) \) and \( \vec{u} = (u_x, u_y) \) and let \( C \) be the center of the ellipse and let \( r_x \) and \( r_y \) be the lengths of the two axes. For a point \( Q \) to lie on the ellipse it must satisfy the following implicit equation:

\[
\frac{(Q_x - C_x)^2}{r_x^2} + \frac{(Q_y - C_y)^2}{r_y^2} = 1
\]

(a) Reduce the ray intersection problem to a quadratic equation, and derive the values of the two roots.

(b) Explain how to determine which root leads to the first intersection point with the ray, and whether the ray hits from the inside or the outside, or misses.
(c) Derive a formula for the 2-dimensional normal vector.

(Note: In class we discussed an alternative method to solve this problem, by applying an appropriate transformation to the ray. In this problem, I would like you to solve the problem directly, without resorting to this method.)

**Problem 7.** 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.
Final Exam

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

Problem 1. (30 points; 2–6 points each) Short answer questions. (Keep your answers short.)

(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) In texture mapping, what is the inverse wrapping function? Why is the inverse wrapping function more important in texture mapping than the wrapping function?

(e) 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, $\ell$ 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.

(f) Recall that there are two common ways to define the interior of a self-intersecting shape, the even-odd rule and the winding number method. Briefly explain how these two methods work, and explain why the point $p$ in the figure below is classified differently by these two methods.

![Even-odd rule](image1)

![Winding number](image2)

Problem 1(g)
Consider a mass-spring system for a rope involving a sequence of particles \(a_1, a_2, \ldots, a_n\).

(i) A designer inserts springs between consecutive particles \((a_i, a_{i+1})\). What is the effect on the rope of increasing the spring constant on these springs?

(ii) A designer inserts springs between alternating particles \((a_{i-1}, a_{i+1})\). What is the effect on the rope of increasing the spring constant on these springs?

**Problem 2.** (10 points) Consider two infinite sequences \(a = [\ldots a_0, a_1, \ldots]\) and \(b = [\ldots b_0, b_1, b_2, \ldots]\).

Recall that their convolution is defined to be an infinite sequence \(c = (a * b)\) whose \(i\)th element is:

\[
c_i = \sum_{j=-\infty}^{\infty} a_j \cdot b_{i-j}.
\]

(a) Define an infinite sequence \(a\) so that if \(c = (a * b)\), then \(c_i = b_i - b_{i-1}\). That is, each element of \(c\) is the difference of two consecutive elements of \(b\). For example:

\[a * [\ldots 4, 7, 2, 8, 5, 12, \ldots] = [\ldots 3, -5, 6, -3, 7, \ldots].\]

Explain your answer briefly.

(b) Does \(a\) have finite support? (Explain briefly.)

**Problem 3.** (20 points) Consider the following ray generation problem. Consider a viewer located at the origin, looking along the negative \(z\)-axis. One unit away from the eye along the negative \(z\)-axis, there is a window of width 4 and height 3 centered about the \(z\)-axis. (See the figure below.) We want to map this window to an image of size 8 \(\times\) 6 image (8 pixels wide and 6 pixels tall), where the \((0, 0)\) pixel is in the upper-left corner.

(a) For a given row \(r\) and column \(c\), where \(0 \leq r < 6\) and \(0 \leq c < 8\), consider the ray emanating from the origin that passes through the upper-left corner of the pixel in row \(r\) and column \(c\). As a function of \(r\) and \(c\), what are the \((x, y, z)\) coordinates of the point on the image plane where this ray hits the image plane? (The figure above shows the situation for \(r = 2\) and \(c = 5\), but your formula must work for any \(r\) and \(c\).) Explain how you derived your answer.

(b) Suppose that you wanted to set up the same viewing situation in OpenGL. What would the arguments be to \texttt{gluLookAt} and \texttt{gluPerspective}? For \texttt{gluPerspective}, assume that the near clipping plane is located 1 unit from the viewer, and the far clipping plane is located 1000 units from the viewer. Recall the following OpenGL functions:

\[
gluLookAt(eyex, eyey, eyez, ctrx, ctry, ctrz, upx, upy, upz)
gluPerspective(fovy, aspect, near, far)
\]

Remember that \texttt{fovy} is the \(y\)-field of view given in degrees, and \texttt{aspect} is the ratio of the image width over the image height.
Problem 4. (20 points) 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 5. (20 points) 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?